FLASH User's Guide

Version 2.3

June 2003

ASCI FLASH Center
University of Chicago

License

0.1 Acknowledgments in Publication

All publications resulting from the use of the FLASH Code must acknowledge the ASCI/Alliance Center for Astrophysical Thermonuclear Flashes. Addition of the following text to the paper acknowledgments will be sufficient.

"The software used in this work was in part developed by the DOE-supported ASCI/Alliance Center for Astrophysical Thermonuclear Flashes at the University of Chicago."

This is a summary of the rules governing the dissemination of the "Flash Code" by the ASCI/Alliances Center for Astrophysical Thermonuclear Flashes (Flash Center) to users outside the Center, and constitutes the License Agreement for users of the Flash Code. Users are responsible for following all of the applicable rules described below.

0.2 Full License Agreement

- Public release. We expect to be able to publically release versions of the Flash Code via the Center's web site, and expect to release any given version of the Flash Code within a year of its formal deposition in the Center's code archive.
- Decision process. At present, release of the Flash Code to users not located at the University of Chicago or at Argonne National Laboratory is governed soley by the Director and the Management Committee; decisions related to public release of the Flash Code will be made in the same manner.
- Distribution rights. The Flash Code, and any part of this code, can only be released and distributed by the Flash Center; individual users of the code are not free to re-distribute the Flash Code, or any of its components, outside the Center. While the Flash Code is currently not export-controlled, we are nevertheless required to insure that we identify all users of this code. For this reason, we require that all users sign a hardcopy version of this License Agreement, and send it to the Flash Center. Distribution of the Flash Code can only occur once a signed License Agreement is received by us.
- Modifications and Acknowledgments. You may make modifications to the Flash Code, and we encourage you to send such modifications to the Center; as noted in 3. above, you are not free to distribute this code to others. As resources permit, we plan to incorporate such modifications in subsequent releases of the Flash Code, and we will acknowledge your contributions. Note that modifications that do not make it into an officially-released version of Flash will not be supported by us.

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

  - a) The software must carry prominent notices stating that you changed specified portions of the Flash Code. This will assist us in clearly identifying the portions of the code that you have contributed.
  - b) The software must display the following acknowledgement: "This product includes software developed by and/or derived from the ASCI/Alliances Center for Astrophysical Thermonuclear Flashes (http://flash.uchicago.edu) to which the U.S. Government retains certain rights."
  - c) The present code header section, describing the origins of the Flash Code and of its components, must remain intact, and should be included in all modified versions of the code. Furthermore, all publications resulting from the use of the Flash Code, or any modified version or portion of

Acknowledgments

The FLASH Code Group is supported by the ASCI FLASH Center at the University of Chicago under U. S. Department of Energy contract B341495. Some of the test calculations described here were performed using the Origin 2000 computer at Argonne National Laboratory and the ASCI Nirvana computer at Los Alamos National Laboratory.

Considerable non-FLASH contributors include:

MZ is supported by the Scientific Discovery through Advanced Computing (SciDAC) program of the DOE, grant number DE-FC02-01ER41176 to the Supernova Science Center/UCSC.

PR is supported by the University of Illinois at Urbana-Champaign and the National Center for Supercomputing Applications (NCSA).

LJD is supported by the Krell Institute Computational Science Graduate Fellowship.

PARAMESH was developed under NASA Contracts/Grants NAG5-2652 with George Mason University; NAS5-32350 with Raytheon/STX; NAG5-6029 and NAG5-10026 with Drexel University; NAG5-9016 with the University of Chicago; and NCC5-494 with the GEST Institute. For information on PARAMESH please contact its main developers; Peter MacNeice (macneice@alfven.gsfc.nasa.gov) or Kevin Olson (olson@bohr.gsfc.nasa.gov).

Contents

Part I

Getting Started

Chapter 1

Introduction

FLASH is a modular, adaptive-mesh, parallel simulation code capable of handling general compressible flow problems found in many astrophysical environments. FLASH is designed to allow users to configure initial and boundary conditions, change algorithms, and add new physics modules with minimal effort. It uses the PARAMESH library to manage a block-structured adaptive grid, placing resolution elements only where they are needed most. FLASH uses the Message-Passing Interface (MPI) library to achieve portability and scalability on a variety of different parallel computers.

The Center for Astrophysical Thermonuclear Flashes, or 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). The goal of the Center is to address several problems related to thermonuclear flashes on the surfaces of compact stars (neutron stars and white dwarfs), in particular X-ray bursts, Type Ia supernovae, and novae. To solve these problems requires the participants in the Center to develop new simulation tools capable of handling the extreme resolution and physical requirements imposed by conditions in these explosions and to do so while making efficient use of the parallel supercomputers developed by the ASCI project, the most powerful constructed to date.

1.1 What's new in FLASH 2.3

We continue to make substantial improvements and expansions to the FLASH code originally developed by Fryxell *et al.* (2000) and to progress toward the ASCI FLASH Center's goal of increased problem-solving capability, modularization, and ease of development. Since the release of FLASH 2.3 in November 2002, many new improvements and additions have been made to the FLASH code, including:

- Increased portability;
- Improved modularity;
- New solvers;
- Additional test setups;
- Cleaner interfaces; and
- Numerous bug fixes.

Some specific new features include:

- Much faster Poisson solvers;
- Support for cosmological simulations in comoving coordinates;
- Improved support for runtime visualization; and

- Improved `fidlr` routines.

Finally, starting with FLASH 2.3, incremental updates of the code will be more readily available. The FLASH web page will contain intermediate releases which will have bug fixes, optimizations and new features.

1.2 About the user's guide

This user's guide is designed to enable individuals unfamiliar with the FLASH code to quickly get acquainted with its structure and to move beyond the simple test problems distributed with FLASH, customizing it to suit their own needs. Chapter 2 (Quick start) discusses how to get started quickly with FLASH, describing how to configure, build, and run the code with one of the included test problems and then to examine the resulting output. Users unfamiliar with the capabilities of FLASH, who wish to quickly 'get their feet wet' with the code, should begin with this section. Users who want to get started immediately using FLASH to create new problems of their own will want to refer to Chapter 3 (The FLASH configuration script) and Chapter 4 (Setting Up New Problem).

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

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

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

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

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

Toro, E. F. *Riemann Solvers and Numerical Methods for Fluid Dynamics, 2nd Edition* (Springer, 1999)

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

Part III describes the different test problems distributed with FLASH. Part IV describes in more detail the analysis tools distributed with FLASH, including `fidlr` and `sfocu`. Finally, Part V gives detailed instructions for extending FLASH's capabilities by integrating new solvers into the code.

Chapter 2

Quick start

This section describes how to get up and running quickly with FLASH by showing how to configure and build it to solve the Sedov explosion problem, how to run it, and how to examine the output using IDL.

2.1 System requirements

You should verify that you have the following:

- A copy of the FLASH source code distribution. This is most likely available either as a Unix tar file or as a local Concurrent Versions System (CVS) source tree. To request a copy of the distribution, click on the "Code Request" link at the FLASH Code Group web site, `http://flash.uchicago.edu/flashcode/`. You will be asked to fill out a short form before receiving download instructions. Please remember the user name and password you use to download the code; you will need these to get bug fixes and updates to FLASH.
- A Fortran 90 compiler and a C compiler. Most of FLASH is written in Fortran 90. Information available at the Fortran Market web site (`http://www.fortran.com/`) can help you select a Fortran 90 compiler for your system.
- An installed copy of the Message-Passing Interface (MPI) library. A freely available implementation of MPI called MPICH has been created at Argonne National Laboratory and can be accessed on the World Wide Web at `http://www-unix.mcs.anl.gov/mpi/mpich/`.
- To use the Hierarchical Data Format (HDF) for output files, you will need an installed copy of the freely available HDF library. Currently, FLASH supports HDF version 4.x and HDF5 version 1.4.x (the two formats are not compatible). The serial version of HDF5 is the current default FLASH format. HDF is available from the HDF Project of the National Center for Supercomputing Applications (NCSA) at `http://hdf.ncsa.uiuc.edu/`. The contents of HDF output files produced by the FLASH `io/amr/hdf*` modules are described in detail in Chapter 7.
- To use the output analysis tools described in this section, a copy of the IDL language from Research Systems, Inc. (`http://www.rsinc.com/`). IDL is a commercial product. It is not required for the analysis of FLASH output, but without it the user will need to write his or her own analysis tools. (FLASH output formats are described in Chapter 7.) The currently available IDL routines were written and tested with IDL 5.4. You are encouraged to upgrade if you are using an earlier version. New versions of IDL come out frequently, and sometimes break backwards compatibilty, but every effort will be made to support them.

  The FLASH code group is working with members of the Mathematics and Computer Science Division at Argonne National Laboratory to develop more sophisticated visualization tools to distribute either as part of or alongside FLASH. Currently, the code group provides a runtime visualization module that can help in analysis (see Chapter 16).

- 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 1.6 and higher uses macro concatenation when constructing Makefiles.

- A copy of the Python language, version 1.5.2 or later is required to run the `setup` script. Several different versions of Python are freely available at `http://www.python.org`.

FLASH has been tested on the following Unix-based platforms. In addition, it may work with others not listed (see Chapter 22).

- SGI single- and multi-processor systems running IRIX
- Intel- and Alpha-based single- and multi-processor systems running Linux, including clusters
- Cray/SGI T3E running UNICOS
- The ASCI Nirvana machine, built by SGI
- IBM SP2/3 systems, including ASCI Blue Pacific and Frost (LLNL), Quad (ANL), Blue Horizon (SDSC), and Seaborg (NERSC).
- IBM SP4 systems
- Sun E10K Starfire Clusters
- Compaq TRU64 Unix Clusters
- The ASCI Red machine, built by Intel

2.2 Unpacking and configuring FLASH for quick start

To begin, unpack the FLASH source code distribution. If you have a Unix tar file, type '`tar xvf FLASH`*X.Y*`.tar`' (without the quotes), where *X.Y* is the FLASH version number (for example, use `FLASH 2.3.tar` for FLASH version 2.3). If you are working with a CVS source tree, use '`cvs checkout FLASH`*X.Y*' to obtain a personal copy of the tree. You may need to obtain permission from the local CVS administrator to do this. In either case you will create a directory called `FLASH`*X.Y*`/`. Type '`cd FLASH`*X.Y*' to enter this directory.

Next, configure the FLASH source tree for the Sedov explosion problem using the `setup` script. Type

```
./setup sedov -auto
```

This configures FLASH for the `sedov` problem using the default hydrodynamic solver, equation of state, mesh package, and I/O format defined for this problem. For the purpose of this quick start example, we will use the default I/O format, HDF5. The source tree is configured to create a two-dimensional code by default. 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. You may need to edit the library locations in this file.

From the FLASH root directory (*i.e.* the directory from which you ran `setup`), execute `gmake`. This will compile the FLASH code. If you should have problems and need to recompile, '`gmake clean`' will remove all object files from the `object/` directory, leaving the source configuration intact; '`gmake realclean`' will remove all files and links from `object/`. After '`gmake realclean`,' a new invocation of `setup` is required before the code can be built. The building can take a long time on some machines; doing a parallel build (`gmake -j 4` for example) can significantly speed things up, 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.*, 2 for $X.Y = 2.3$). 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 have HDF4 but not HDF5, then you need to rerun the `setup` script. Type

  ```
  ./setup sedov -auto --with-module=io/amr/hdf4
  ```

  If you have neither HDF4 or HDF5, you can still setup and compile FLASH, but you will not be able to generate either a checkpoint or a plot file. You can setup FLASH without IO by typing

  ```
  ./setup sedov -auto --with-module=io/null
  ```

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

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

```
http://flash.uchicago.edu/flashcode/
```

FLASH expects to find a text file named `flash.par` in the directory from which it is run. This file sets the values of various runtime parameters that determine the behavior of FLASH. If it is not present, FLASH will abort; `flash.par` must be created in order for the program to run (note: most of the distributed setups already come with a `flash.par`, which will be *copied* into the `object/` directory). 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, create `flash.par` in the main FLASH directory with the contents of Fig. 2.1.

This example instructs FLASH to use up to four levels of adaptive mesh refinement (AMR) (through the `lrefine_max` parameter) and to name the output files appropriately (`basenm`). We will not be starting from a checkpoint file (`restart = .false.` — this is the default, but here it is explicitly set for clarity). Output files are to be written every 0.005 time units (`trstrt`) and will be created until $t = 0.02$ or 1000 timesteps have been taken (`tmax` and `nend` respectively), whichever comes first. The ratio of specific heats for the gas (γ) is taken to be 1.4, and all four boundaries of the two-dimensional grid have outflow (zero-gradient or Neumann) boundary conditions (see via the `[xy][lr]_boundary_type` parameters).

Note the format of the file – each line is a comment (denoted by a hash mark, `#`), blank, or of the form *variable* = *value*. 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 `paramFile.txt`, 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 `setups/sedov/` directory.

2.3 Running FLASH

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

mpirun -np N object/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 executable can take one command-line argument, the name of the runtime parameter file. The default parameter file name is `flash.par`. This is system-dependent and is not permitted by some machines (or MPI versions).

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

```
# runtime parameters
lrefine_max = 4

basenm = "sedov_4_"
restart = .false.
trstrt = 0.005

nend = 1000
tmax = 0.02

gamma = 1.4

xl_boundary_type = "outflow"
xr_boundary_type = "outflow"

yl_boundary_type = "outflow"
yr_boundary_type = "outflow"

plot_var_1 = "dens"
plot_var_2 = "temp"
plot_var_3 = "pres"
```

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

- `flash.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. Messages indicating when the code refined and what output resulted are also contained in `flash.log`.
- `flash.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_4_hdf5_chk_000*` are the different checkpoint files. These are complete dumps of the entire simulation at intervals of `trstrt` and are suitable for use in restarting the simulation. They are also the primary output products of FLASH.
- `sedov_4_hdf5_plt_cnt_000*` are plot files. These are files containing only density, temperature, and pressure (in single precision for some I/O modules). They are designed to be written more frequently than checkpoint files for the purpose of making simulation movies (or for analyses that do not require all of the checkpointed quantities).
- `amr_log` includes various messages from the PARAMESH adaptive mesh refinement package.

We will use the `xflash` routine under IDL to examine the output. Before doing so, we need to set the values of two environment variables, `IDL_PATH` and `XFLASH_DIR`. Under `csh` this can be done using the commands

```
setenv XFLASH_DIR "$PWD/tools/fidlr2"
setenv IDL_PATH "${XFLASH_DIR}:$IDL_PATH"
```

If you get a message indicating that `IDL_PATH` is not defined, enter

`setenv IDL_PATH "$XFLASH_DIR":`*idl-root-path*`:`*idl-root-path*`/lib` where *idl-root-path* points to the directory in which IDL is installed. Finally, since FLASH 2.3 uses HDF5 as the default I/O module, you need to setup the wrappers that communicate between IDL and the HDF5 library. The reader is referred to §20.1.2.

Now run IDL (`idl`) and enter `xflash` at the `IDL>` prompt. You should see the main widget as shown in Fig. 2.2. Select any of the output files through the *File/Open Prototype...* dialog box. This will define a

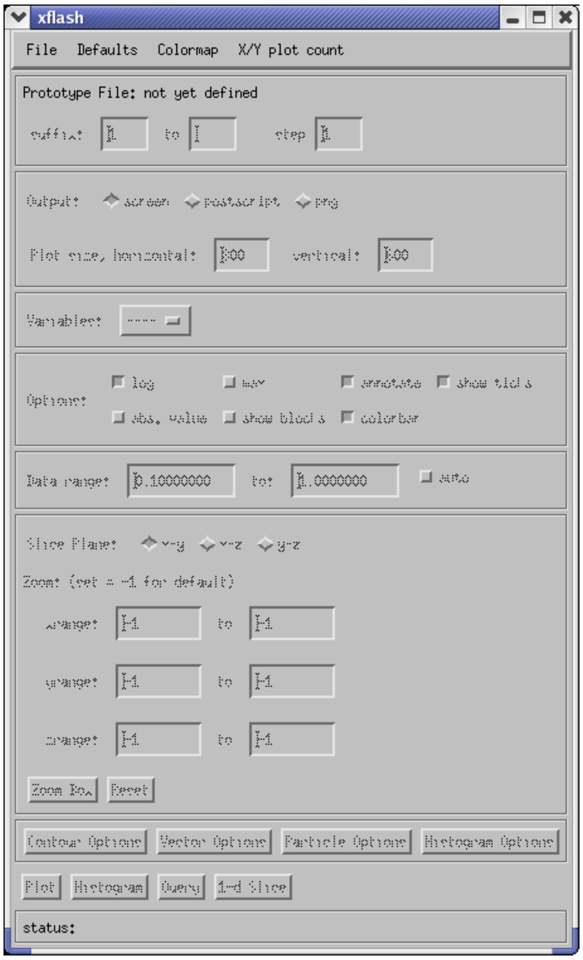


Figure 2.2: The main `xflash` widget.

prototype file for the dataset, which is used by `fidlr2` to describe the dataset. With the prototype defined, enter one of the suffixes in the first suffix box in the main widget. `xflash` can generate output for a number of consecutive files, but if you fill in only the beginning suffix, only one file is read. Click the *auto* box next to the data range to automatically scale our plot to the data. Select the desired plotting variable and colormap. Under 'Options,' select whether to plot the logarithm of the desired quantity and select whether to plot the outlines of the AMR blocks. For very highly refined grids, the block outlines can obscure the data, but they are useful for verifying that FLASH is putting resolution elements where they are needed. Finally, click 'Velocity Options' to overlay the velocity field. The 'xskip' and 'yskip' parameters enable you to plot only a fraction of the vectors, so that they do not obscure the background plot.

When the control panel settings are to your satisfaction, click the 'Plot' button to generate the plot. For Postscript, GIF, or PNG output, a file is created in the current directory. The result should look something like Fig. 2.3, although this figure was generated from a run with eight levels of refinement rather than the four used in the quick start example run. With fewer levels of refinement, the Cartesian grid causes the explosion to appear somewhat diamond-shaped.

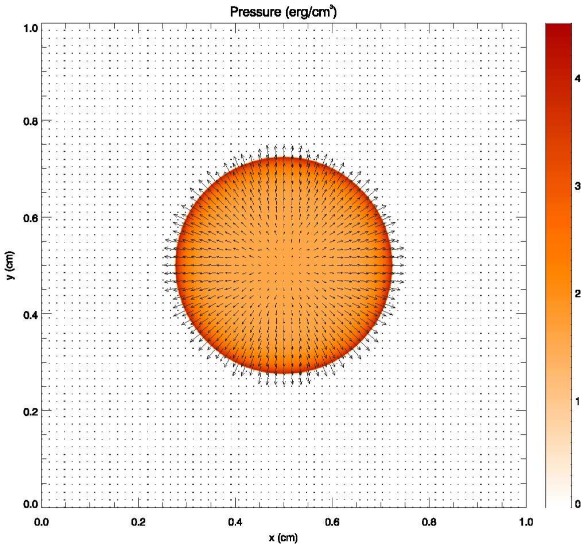


Figure 2.3: Example of `xflash` output for the Sedov problem with eight levels of refinement.

FLASH is intended to be customized by the user to work with interesting initial and boundary conditions. In the following sections, we will cover in more detail the algorithms and structure of FLASH and the sample problems and tools distributed with it.

Chapter 3

The FLASH configuration script (setup)

The setup script, found in the FLASH root directory, provides the primary command-line interface to the FLASH source code. It configures the source tree for a given problem and target machine and creates files needed to parse the runtime parameter file and make the FLASH executable. More description of what setup does may be found in Chapter 5. Here we describe its basic usage.

Running setup without any options prints a message describing the available options:

```
$ ./setup
usage:  setup <problem-name> [options]

        problems: see setups/ directory
        options: -auto -[123]d -maxblocks=<#> -nxb=<#> -nyb=<#> -nzb=<#>
                 -portable -verbose -force [-site=<site> | -ostype=<ostype>]
                 [-debug | -test] -preprocess -objdir=<relative obj directory>
                 -with-module=<module>
```

Available values for the mandatory option (the name of the problem to configure) are determined by scanning the setups/ directory.

A "problem" consists of a set of initial and boundary conditions, possibly additional physics (*e.g.*, a subgrid model for star formation), and a set of adjustable parameters. The directory associated with a problem contains source code files that implement the initial conditions and, in a few cases, the boundary conditions and extra physics. Also present is a configuration file, read by setup, which contains information on required physics modules and adjustable parameters.

setup determines site-dependent configuration information by looking in source/sites/ for a directory with the same name as the output of the hostname command; failing this, it looks in the directory source/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. Only one of these options can be used. The directory for each site or operating system type contains a makefile fragment (Makefile.h) that sets command names, compiler flags, library paths, and any replacement or additional source files needed to compile FLASH for that machine type.

setup uses the problem and site/OS type, together with a user-supplied file called Modules, which lists the code modules to include, to generate a directory called object/ that contains links to the appropriate source files and makefile fragments. It also creates the master makefile (object/Makefile) and several Fortran include files that are needed by the code in order to parse the runtime parameter file. After running setup, the user can make the FLASH executable by running gmake in the object/ directory (or from the FLASH root directory, if the -portable option is not used with setup). Parallel builds, using the -j argument to gmake should work and can significantly speed up the build process.

The optional command-line modifiers have the following interpretations:

`-verbose`
: Normally `setup` echoes to the standard output summary messages indicating what it is doing. Including the `-verbose` option causes it to also list the links it creates.

`-portable`
: This option creates a portable build directory by copying instead of linking to the source files in `source/` and `setups/`. The resulting build directory can be placed into a `tar` archive and sent to another machine for building (use the Makefile created by `setup` in the tar file).

`-auto`
: This modifier replaces `-defaults`, which is still present in the code but has been deprecated. Normally `setup` requires that the user supply a plain text file called `Modules` (in the FLASH root directory) that specifies which code modules to include. A sample `Modules` file appears in Fig. 3.1. Each line is either a comment (preceded by a hash mark (`#`)) or a module include statement of the form `INCLUDE` *module*. Sub-modules are indicated by specifying the path to the sub-module in question; in the example, the sub-module `gamma` of the `eos` module is included. If a module has a default sub-module but no sub-module is specified, `setup` automatically selects the default using the module's configuration file.

The `-auto` option enables `setup` to generate a "rough draft" of a `Modules` file for the user. The configuration file for each problem setup specifies a number of code module requirements; for example, a problem may require the perfect-gas equation of state (`materials/eos/gamma`) and an unspecified hydro solver (`hydro`). With `-auto`, `setup` creates a `Modules` file by converting these requirements into module include statements. In addition, it checks the configuration files for the required modules and includes any of *their* required modules, eliminating duplicates. Most users configuring a problem for the first time will want to run `setup` with `-auto` to generate a `Modules` file and then to edit `Modules` directly to specify different sub-modules. After editing `Modules` in this way, re-run `setup` without `-auto` to incorporate the changes into the code configuration.

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

`-maxblocks=#`
: This option is also used by `setup` in constructing the makefile compiler options. It determines the amount of memory allocated at runtime to the adaptive mesh refinement (AMR) block data structure. For example, to allocate enough memory on each processor for 500 blocks, use `-maxblocks=500`. If the default block buffer size is too large for your system, you may wish to try a smaller number here (the defaults are currently defined in `source/driver/physicaldata.fh`). Alternatively, you may wish to experiment with larger buffer sizes, if your system has enough memory.

`-nxb=# -nyb=# -nzb=#`
: These options are also used by `setup` in constructing the makefile compiler options. The mesh on which the problem is solved is composed of blocks. These options determine the number of interior cells (not counting guard cells) for each block. The default value for each of these options is 8.

```
# Modules file constructed for rt problem by setup -auto

INCLUDE driver/time_dep
INCLUDE hydro
INCLUDE materials/eos/gamma
INCLUDE gravity/constant
INCLUDE mesh
INCLUDE io
```

Figure 3.1: Example of the `Modules` file used by `setup` to determine which code modules to include.

`-debug`	The default `Makefile` built by setup will use the optimized setting for compilation and linking. Using `-debug` will force setup to use the flags relevant for debugging (*e.g.*, including `-g` in the compilation line).
`-test`	When FLASH is tested by the automated test suite, `test` will choose the proper compilation arguments for the test executable.
`-preprocess`	This option will preprocess all of the files before compilation. This is useful for machines whose compilers do not support preprocessing.
`-objdir`	Overrides the default `object` directory with one whose name is specified by this parameter.
`-with-module=<module>`	Add the module specified by `<module>` to the `setup`-generated `Modules` file.

When `setup` is run, it reads all of the `Config` files in the module directories to find the runtime parameters that the code understands. A file named `paramFile.txt` is generated by setup and contains a list of all of the runtime parameters that are understood by FLASH and some brief comments describing their purpose. In addition to the name, comments (if available), the default value, and the module that owns the parameter are listed. This file provides a useful way to determine which parameters can be used in a `flash.par` for a given problem.

To set runtime parameters to values other than the defaults, create a runtime parameter file named `flash.par` in the directory from which FLASH is to be run. The format of this file is described briefly in Chapter 2 and in more detail in Sec. 4.3.

`setup` also creates two functions that are used by FLASH. `buildstamp` takes a file logical unit number and outputs the date and time the current FLASH executable was setup, along with the platform information. `flash_release` returns a character string containing the full version number (including the minor version number) of the present build of FLASH.

Chapter 4

Setting up new problems

Every problem that is run with FLASH requires a directory in `FLASH2.3/setups`. This is where the FLASH `setup` script looks to find the problem-specific files. The FLASH distribution includes a number of pre-written setups. However, most FLASH users will want to define their own problems, so it is important to understand the techniques for adding a customized problem setup.

Each setups directory contains the routines that initialize the FLASH grid. The directory also includes parameter files that `setup` uses to select the proper physics modules from the FLASH source tree. When the user runs `setup`, the proper source files are selected and linked to the `object/` directory (Chapter 3).

There are two files that must be included in the setup directory for any problem. These are

`Config`	lists the modules required for the problem and defines additional runtime parameters.
`init_block.F90`	Fortran routine for setting initial conditions in a single block.

We will look in detail at these files for an `example` setup. This is a simple setup that creates a domain with hot ash inside a circle of radius `radius` centered at (`xctr`, `yctr`, `zctr`). The density is uniformly set at `rho_ambient` and the temperature is `t_perturb` inside the circle and `t_ambient` outside.

To create a new setup, we first create the new directory and then add the `Config` and `init_block.F90` files. The easiest way to construct these files is to use files from another setup as a template.

4.1 Creating a `Config` file

The simplest way to construct a `Config` file is to copy one from another setup that incorporates the same physics as the new problem. `Config` serves two principal purposes: (1) to specify the required modules and (2) to register runtime parameters. The `Config` file for the `example` problem contains the following:

```
# configuration file for our example problem

REQUIRES driver/time_dep
REQUIRES materials/eos/gamma
REQUIRES materials/composition/fuel+ash
REQUIRES io
REQUIRES mesh
REQUIRES hydro
```

These lines define the FLASH modules required by the setup. We are going to carry two fluids (fuel and ash), so we require the composition module `fuel+ash`. At runtime, this module will initialize the multifluid database to carry the two fluids and set up their properties. We also require I/O, meshing, and hydrodynamics, but we do not specify particular sub-modules of these modules; any sub-modules of io, mesh, and hydro will satisfy these requirements. However, we require the simple gamma-law equation of state (materials/eos/gamma) for this problem, so we specify it explicitly. In constructing the list of requirements for a problem, it is important to keep them as general as the problem allows. Specific modules satisfying the requirements are given in the Modules file when we actually run setup (the Modules file and its format are introduced in Chapter 3).

After defining the modules, the Config file lists any runtime parameters specific to this problem:

```
# runtime parameters
PARAMETER rho_ambient   REAL    1.
PARAMETER t_ambient     REAL    1.
PARAMETER t_perturb     REAL    5.
PARAMETER radius        REAL    0.2
PARAMETER xctr          REAL    0.5
PARAMETER yctr          REAL    0.5
PARAMETER zctr          REAL    0.5
```

Here we define the ambient density (rho_ambient), the ambient and perturbed temperatures (t_ambient, t_perturb), the radius of the perturbed region (radius), and the coordinates of the center of the perturbation (xctr, yctr, zctr). All of these parameters are floating point numbers. We also give the default values for each parameter (in case they are not assigned values in the runtime parameter file; see below).

The routine init_block (or any other FLASH function) can access any of these variables through a simple database subroutine call. The default value of any parameter (like rho_ambient) can be overridden at runtime by specifying a different value in a file flash.par (the runtime parameter file); for example, rho_ambient = 100. All parameters required for initialization of the new problem should be added to Config.

4.2 Creating an init_block.F90

The routine init_block is called by the framework to initialize data in each AMR block. The framework first forms the grid at the lowest level of refinement and calls init_block to initialize the data in each block. The code checks the refinement criteria in each block it has created and then refines the blocks according to these criteria. It then calls init_block to initialize the newly created blocks. This process repeats until the mesh reaches the maximum refinement level in the areas marked for refinement.

The basic structure of the routine init_block should consist of

1. Fortran module use statements to access the runtime databases.
2. Declaration of local variables.
3. Calls to the database to obtain the values of runtime parameters.
4. Initialization of the variables.
5. Calls to the database to store the values of solution variables.

Any of the setups may be used as a template. We continue to look at the example setup and describe it in detail below.

The first part of an init_block consists of use-associating the Fortran 90 modules that provide access to the variable database (dBase), the multifluid database (multifluid_database), and the runtime parameter database (runtime_parameters). We will also need the ModuleEos module to access the pointwise equation of state.

Each database module exposes a relatively small number of public procedures and constants (see Sec. 5.1.2 for details). To help make clear what public variables from these modules a routine uses, we use the ONLY clause in the use statement. In addition to listing the functions we intend to use, we also list any parameters that we need from these modules, such as the dimension (ndim), the number of zones in each direction (nxb, nyb, nzb), the number of guard cells (nguard), and the number of fluids (ionmax).

```
subroutine init_block(block_no)
!
! sample init_block -- initialize a circle with high temperature fuel
! surrounded by ash.
!
  use multifluid_database, ONLY: find_fluid_index

  use runtime_parameters, ONLY: get_parm_from_context, GLOBAL_PARM_CONTEXT

  use dBase, ONLY: nxb, nyb, nzb, nguard, ionmax, &
       k2d, k3d, ndim,  &
       CARTESIAN, &
       dBasePropertyInteger, &
       dBaseKeyNumber, dBaseSpecies, &
       dBaseGetCoords, dBasePutData

  use ModuleEos, ONLY: eos
```

Next come the local declarations. In this example, there are loop indices, one dimensional scratch arrays, integer keys that will be used in the database calls, and other scratch variables needed for the initialization.

```
  implicit none

  integer :: i, j, k, block_no, n

  logical, save :: firstCall = .TRUE.

  real, save :: smallx

! variables needed for the eos call
  real :: temp_zone
  real :: pel, eel, ptot, eint, abar, zbar, entropy
  real :: dpt, dpd, ded, det, dst, dsd, c_v, c_p, gamma
  real :: xalfa, xxni, xxne, xxnp

  integer, save :: iXvector, iYvector, iZvector
  integer, save :: iXcoord,  iYcoord,  iZcoord

  integer, save :: iPoint
  integer, save :: izn

  real :: dist

  integer, save :: idens, itemp, ipres, iener, igame, igamc
  integer, save :: ivelx, ively, ivelz, inuc_begin
  integer, save :: ifuel, iash

! save the parameters that describe this initialization
  real, save :: rho_ambient, t_ambient, t_perturb
  real, save :: radius
  real, save :: xctr, yctr, zctr
```

```
! compute the maximum length of a vector in each coordinate direction
! (including guard cells)
  integer, parameter :: q = max(nxb+2*nguard, &
                                nyb+2*nguard*k2d, &
                                nzb+2*nguard*k3d)

  real, dimension(q) :: x, y, z
  real :: xx, yy, zz

  real, dimension(q)  :: rho, p, t, game, gamc, vx, vy, vz, e
  real, dimension(ionmax) :: xn

  integer, save :: MyPE, MasterPE

  integer :: meshGeom
```

Please note that FLASH promotes all floating point variables to double precision at compile time for maximum portability. We therefore declare all floating point variables with `real` in the source code. Note also that a lot of these variables are explicitly saved. These variables will not change through the simulation. They include the runtime parameters that we defined above and the keys that will be used in database calls (*e.g.* `idens`).

The variable `firstCall` is `.true.` the first time through this `init_block`, when these saved variables will be filled, and then it is set to be `.false.` for subsequent entries into `init_block`. We start by getting the mesh geometry from the database and check to see if it is one that we intend to support—in this case Cartesian only.

The next part of the code calls the database to get the values we need to initialize the domain. In addition to the runtime parameters and any physical constants, we also create integer keys that will be used in the variable database calls. Most of the database calls are overloaded to accept either a string or an integer key to select a variable. String comparisons are expensive, so we make them once when getting the key and save the result for later use.

```
  if (firstCall) then

     MyPE = dBasePropertyInteger('MyProcessor')
     MasterPE = dBasePropertyInteger('MasterProcessor')


!------------------------------------------------------------------------------
! make sure that we are running in a geometry that we intend to support
!------------------------------------------------------------------------------
     meshGeom = dBasePropertyInteger("MeshGeometry")

     if (meshGeom /= CARTESIAN) then
        call abort_flash("ERROR: init_block only supports Cartesian geometry")
     endif


!------------------------------------------------------------------------------
! grab the parameters relevant for this problem
!------------------------------------------------------------------------------
     call get_parm_from_context(GLOBAL_PARM_CONTEXT, 'smallx', smallx)

     call get_parm_from_context(GLOBAL_PARM_CONTEXT, 'rho_ambient', rho_ambient)

     call get_parm_from_context(GLOBAL_PARM_CONTEXT, 't_ambient', t_ambient)
```

```
     call get_parm_from_context(GLOBAL_PARM_CONTEXT, 't_perturb', t_perturb)

     call get_parm_from_context(GLOBAL_PARM_CONTEXT, 'radius', radius)

     call get_parm_from_context(GLOBAL_PARM_CONTEXT, 'xctr', xctr)
     call get_parm_from_context(GLOBAL_PARM_CONTEXT, 'yctr', yctr)
     call get_parm_from_context(GLOBAL_PARM_CONTEXT, 'zctr', zctr)
```

It is sometimes useful to have the `init_block` routine print some output, such as echoing runtime parameters to the screen. This is best done in the `firstCall` block.

```
     if (MyPE == MasterPE) then
        print *, 'Initializing the example setup'
     endif
```

It is also useful to do some error checking to make sure the code was set up the way you intended when the `init_block` was written. The function `abort_flash` will print out an error message and abort the code.

```
     if (ionmax /= 2) then
        call abort_flash('Error: ionmax /= 2 in init_block')
     endif
```

Next we get integer keys for the different database calls we will be making. Most of the database calls are overloaded to accept a string or an integer to specify which variable is being stored, the coordinate direction, *etc.* We do the string to integer conversion here, so it is only executed once each time FLASH is run.

```
!------------------------------------------------------------------------------
! get the pointers into the solution vector
!------------------------------------------------------------------------------
     idens = dBaseKeyNumber('dens')

     ivelx = dBaseKeyNumber('velx')
     ively = dBaseKeyNumber('vely')
     ivelz = dBaseKeyNumber('velz')

     iener = dBaseKeyNumber('ener')
     ipres = dBaseKeyNumber('pres')
     itemp = dBaseKeyNumber('temp')

     igame = dBaseKeyNumber('game')
     igamc = dBaseKeyNumber('gamc')

     inuc_begin = dBaseSpecies(1)

     if (idens < 0 .OR. ivelx < 0 .OR. ively < 0 .OR. ivelz < 0 .OR. &
         iener < 0 .OR. ipres < 0 .OR. itemp < 0 .OR. &
         igame < 0 .OR. igamc < 0) then
        call abort_flash(``ERROR: variable dBaseKeys are invalid'')
     endif

     call find_fluid_index('fuel', ifuel)
     call find_fluid_index('ash', iash)

     if (ifuel < 0 .OR. iash < 0) then
```

```
      call abort_flash(''ERROR: fluids are no found'')
   endif

   iXvector = dBaseKeyNumber('xVector')
   iYvector = dBaseKeyNumber('yVector')
   iZvector = dBaseKeyNumber('zVector')

   iPoint   = dBaseKeyNumber('Point')

   iXcoord  = dBaseKeyNumber('xCoord')
   iYcoord  = dBaseKeyNumber('yCoord')
   iZcoord  = dBaseKeyNumber('zCoord')

   izn      = dBaseKeyNumber('zn')

   if (iXvector < 0 .OR. iYvector < 0 .OR. iZvector < 0 .OR. iPoint < 0 .OR. &
       iXcoord < 0 .OR. iYcoord < 0 .OR. iZcoord < 0 .OR. izn < 0) then
      call abort_flash(''ERROR: coordinate dBaseKeys are invalid'')
   endif

   firstCall = .FALSE.
endif
```

The next part of the routine involves setting up the initial conditions. This could be code for interpolating a given set of initial conditions, constructing some analytic model, or reading in a table of initial values.

In the present example, we begin by getting the coordinates for the zones in the current block. This is done by a set of calls to `dBaseGetCoords`. The key `izn` that we defined above in the lookup of "`zn`"tells the database that we want the coordinates of the zone centers. We define the direction with `iXcoord`, `iYcoord`, and `iZcoord`, which we also set in the lookups above. The results are stored in the vectors `x`, `y`, and `z`.

```
x(:) = 0.0
y(:) = 0.0
z(:) = 0.0

if (ndim == 3) call dBaseGetCoords(izn, iZcoord, block_no, z)
if (ndim >= 2) call dBaseGetCoords(izn, iYcoord, block_no, y)
call dBaseGetCoords(izn, iXcoord, block_no, x)
```

Next comes a set of loops (one for each dimension) over all of the interior zones in the block. We note that the loops make use of the `k2d` parameter, which is equal to 1 for 2 and 3-d simulations and 0 otherwise, and the `k3d` parameter, which is equal to 1 only for 3-d simulations. This provides a convenient way to construct a general set of loops that will work regardless of the dimensionality. Inside these loops, the values of the density, velocity, abundances, *etc.* are set. We also usually make a call to the equation of state to ensure that these quantities are thermodynamically consistent.

```
!-----------------------------------------------------------------------------
! loop over all of the zones in the current block and set the temperature,
! density, and thermodynamics variables.
!-----------------------------------------------------------------------------
  do k = nguard*k3d+1, nguard*k3d+nzb
     zz = z(k)

     do j = nguard*k2d+1, nguard*k2d+nyb
        yy = y(j)
```

```
      do i = nguard+1, nguard+nxb
         xx = x(i)
```

For the present problem, we are making a hot circular region of fuel. We want to compute the distance of the current zone from the center of the circular region, test whether we are inside the circle, and set the temperature and composition accordingly. Remember that we know the value of the runtime parameters we set up in the `Config` file from the calls to `get_parm_from_context` made above.

```
!------------------------------------------------------------------------------
! compute the distance from the center -- handle this specially for 1, 2, and
! 3 dimensions.
!------------------------------------------------------------------------------

         if (ndim == 1) then
            dist = xx - xctr
         elseif (ndim == 2) then
            dist = sqrt((xx-xctr)**2 + (yy-yctr)**2)
         else
            dist = sqrt((xx-xctr)**2 + (yy-yctr)**2 + (zz-zctr)**2)
         endif

         if (dist <= radius) then
            temp_zone = t_perturb

            xn(ifuel) = smallx
            xn(iash)  = 1.0 - smallx

         else
            temp_zone = t_ambient

            xn(ifuel) = 1.0 - smallx
            xn(iash)  = smallx

         endif
```

We now know the density, composition, and temperature for the current zone. We can find the pressure, internal energy, and gamma corresponding to these value from a call to the equation of state.

```
!------------------------------------------------------------------------------
! get the pressure and internal energy corresponding to the ambient density
! and perturbed temperature
!------------------------------------------------------------------------------
         call eos(rho_ambient, temp_zone, ptot, eint, xn, entropy, &
              abar, zbar, dpt, dpd, det, ded, dst, dsd, c_v, c_p, &
              gamma, pel, xxne, xalfa,1)

         rho(i) = rho_ambient
         t(i)   = temp_zone

         vx(i) = 0.0
         vy(i) = 0.0
         vz(i) = 0.0

         p(i) = ptot
         e(i) = eint + 0.5*(vx(i)**2 + vy(i)**2 + vz(i)**2)
```

```
          game(i) = p(i)/(eint*rho(i)) + 1.0
          gamc(i) = gamma
```

We note that the energy stored by FLASH is the specific total energy, so we add the specific kinetic energy to the specific internal energy returned from the EOS call. In the present case, the kinetic energy is zero since all of our velocities are zero. This step is shown for completeness.

Now that we have the correct state for the current zone, we want to put these values back into the database. We show two methods here. First, the composition is stored one point at a time, using a call to `dBasePutData`. We use the key `inuc_begin`, which we obtained above, as the starting key for the composition variables. We use the fact that the composition variables have contiguous keys to create a loop over all species.

We exit the inner loop (over the x-coordinate) and store the remaining variables one vector at a time. This is also done with the `dBasePutData` function, but this time using the `iXvector` key instead of `iPoint`.

```
!-------------------------------------------------------------------------------
! finally, fill the solution array
!-------------------------------------------------------------------------------
           do n=1,ionmax
              call dBasePutData(inuc_begin-1+n,ipoint, &
                      i, j, k, block_no, xn(n))
           enddo

        enddo

        call dBasePutData(idens, iXvector, j, k, block_no, rho)
        call dBasePutData(iener, iXvector, j, k, block_no, e)
        call dBasePutData(itemp, iXvector, j, k, block_no, t)
        call dBasePutData(ipres, iXvector, j, k, block_no, p  )

        call dBasePutData(ivelx, iXvector, j, k, block_no, vx )
        call dBasePutData(ively, iXvector, j, k, block_no, vy )
        call dBasePutData(ivelz, iXvector, j, k, block_no, vz )

        call dBasePutData(igame, iXvector, j, k, block_no, game)
        call dBasePutData(igamc, iXvector, j, k, block_no, gamc)

     enddo
  enddo

  return
end subroutine init_block
```

When `init_block` returns, the database will now have the values of the initial model for the current block. `init_block` will be called for every block that is created as the code refines the initial model.

We encourage you to run the `example` setup to see this code in action. This setup can be used as the basis for a much more complicated problem. For a demonstration of how to initialize the domain with a one-dimensional initial model, look at the `sample_map` setup.

More generally, a setup also may include customized versions of some of the FLASH routines or other routines. Examples of FLASH routines that may be customized for a particular problem are

`init_1d.F90`	A routine that reads in a 1-d initial model file.
`init_mat.F90`	Fortran routine for initializing the materials module.
`Makefile`	The `Make` include file for the setup. This file is the `Makefile` for any problem-specific routines that are not part of the standard FLASH distribution (like `init_1d` above).
`mark_grid_refinement.F90`	Fortran routine for marking blocks to be refined, modified for this specific problem.

Users are encouraged to put any modifications of core FLASH files in the `setups` directory in which they are working. This makes it easier to distribute patches to our user base.

An additional file required to run the code is `flash.par`. It contains flags and parameters for running the code. Copies of `flash.par` may be kept in the setup directory for easy distribution.

4.3 The runtime parameter file (`flash.par`)

The file `flash.par` is read at runtime and sets the values of runtime parameters. The `flash.par` file for the `example` setup is

```
# Parameters for the example setup
rho_ambient     = 1.0
t_ambient       = 1.0
t_perturb       = 10.
radius          = .2

# for starting a new run
restart         = .false.
cpnumber        = 0
ptnumber        = 0

# dump checkpoint files every trstrt seconds
trstrt          = 4.0e-4

# dump plot files every tplot seconds
tplot           = 5.0e-5

# go for nend steps or tmax seconds, whichever comes first
nend            = 1000
tmax            = 1.0e5

# initial, and minimum timesteps
dtini           = 1.0e-16
dtmin           = 1.0e-20
dtmax           = 1.0e2

# Grid geometry
geometry = "cartesian"

# Size of computational volume
xmin            = 0.0
```

```
xmax            = 1.0
ymin            = 0.0
ymax            = 1.0
zmin            = 0.0
zmax            = 1.0

# Boundary conditions
xl_boundary_type = "outflow"
xr_boundary_type = "outflow"
yl_boundary_type = "outflow"
yr_boundary_type = "outflow"
zl_boundary_type = "outflow"
zr_boundary_type = "outflow"

# Variables for refinement test
refine_var_1    = "dens"
refine_var_2    = "pres"
refine_var_3    = "none"
refine_var_4    = "none"

# Refinement levels
lrefine_max     = 3
lrefine_min     = 1

# Number of lowest-level blocks
nblockx         = 1
nblocky         = 1
nblockz         = 1

# Hydrodynamics parameters
cfl             = 0.8

# Simulation-specific parameters
basenm          = "example_3lev_"
run_number      = "001"
run_comment     = "A simple FLASH 2.3 example"
log_file        = "flash_example.log"
```

In this example, flags are set for a "cold start" of the simulation, grid geometry, boundary conditions, and refinement. Parameters are also set for the ambient temperature and density and for details of the run, such as the number of timesteps between checkpoint files, and the initial, minimum and final values of the timestep.

`setup` produces a file named `paramFile.txt` each time it is run. This file lists all possible runtime parameters and the values to which they were set initially, as well as a brief description of the parameters.

Running the example setup with the `-auto` option and the `flash.par` provided will produce five checkpoint files and 29 plot files. The initial temperature distribution, as visualized by the magic of the fidlr tools, appears in Fig. 4.1

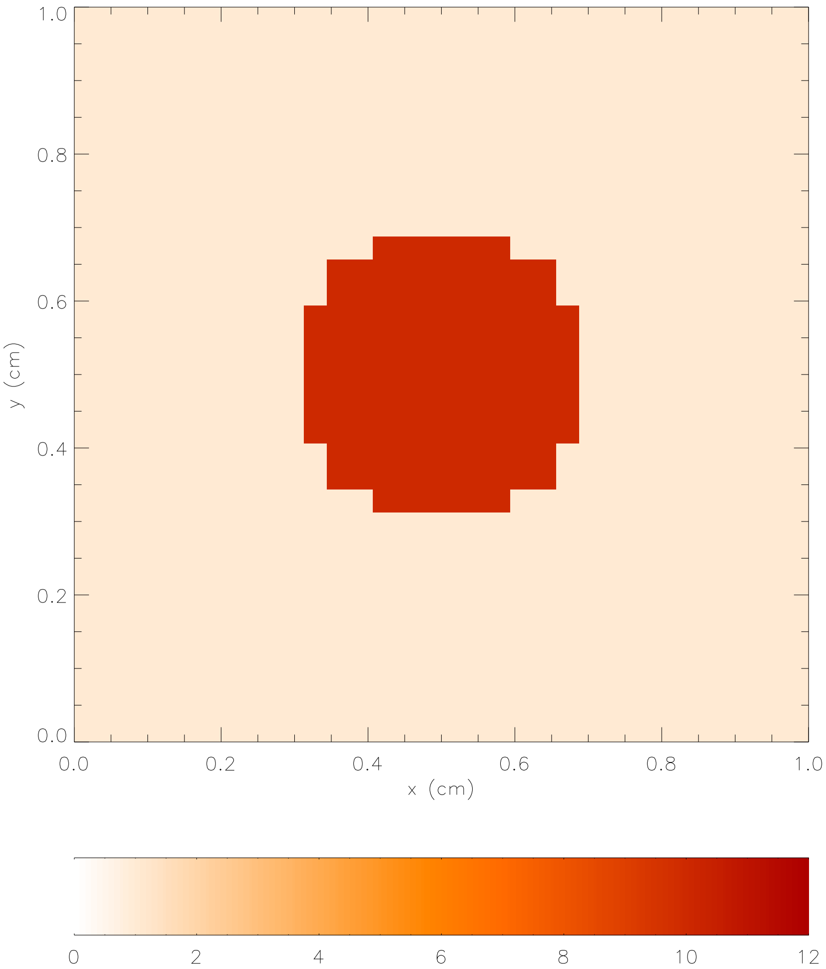


Figure 4.1: Image of the initial temperature distribution in the `example` setup.

Part II

Structure and Modules

Chapter 5

Overview of FLASH architecture

The FLASH source code is a collection of components called FLASH modules. FLASH modules can be combined in a variety of ways to form specific FLASH applications. Of course, not all available FLASH modules are necessarily used when solving any one particular problem. Thus, it is important to distinguish between the entire FLASH source code and a given FLASH application.

5.1 Structure of a FLASH module

Most generally, a FLASH module represents some well-defined, top-level unit of functionality useful for a given class of problems. Its structure conforms to a small set of rules that facilitate its interactions with other modules in the creation of an application. Primary among these are the rules governing the retrieval and modification of data on the solution grid. A module must also announce a general set of requirements to the framework as well as publish a public interface of its services. Here we focus on the internal structure of a FLASH module appropriate for users wishing to extend the current FLASH functionality.

First, it is important to recall how a selected group of FLASH modules is combined to form a particular application. This process is carried out entirely by the FLASH `setup` tool, which uses configuration information provided by the modules and problem setup to properly parse the source tree and isolate the source files needed to set-up a specific problem. For performance reasons, `setup` ties modules together statically before the application is compiled.

Each FLASH module is divided into three principal components:

a) Configuration layer
b) "Wrapper" or "interface" layer
c) Algorithm

Additionally, a module may contain sub-modules which inherit from and override the functionality in the parent module. Each of these components is discussed in detail in the following sections.

5.1.1 Configuration layer

Information about module dependencies, default sub-modules, runtime parameter definitions, library requirements, and so on is stored in plain text files named `Config` in the different module directories. These are parsed by `setup` when configuring the source tree and are used to create the code needed to register module variables, to implement the runtime parameters, to choose sub-modules when only a generic module has been specified, to prevent mutually exclusive modules from being included together, and to flag problems when dependencies are not resolved by some included module. In the future they may contain additional information about module interrelationships.

5.1.1.1 Configuration file syntax

The syntax of the configuration files is as follows. 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-module*
  Specifies which sub-module of the current module is to be used as a default if a specific sub-module has not been indicated in the modules file (Sec. 6.1.3). For example, the `Config` file for the materials/eos module specifies gamma as the default sub-module. If no sub-module is explicitly included (*i.e.* `INCLUDE materials/eos` is placed in modules), then this command instructs `setup` to assume that the gamma submodule was meant (as though `INCLUDE materials/eos/gamma` had been placed in modules).

- `EXCLUSIVE` *sub-module...*
  Specify a list of sub-modules that cannot be included together. If no `EXCLUSIVE` instruction is given, it is perfectly legal to simultaneously include more than one sub-module in the code.

- `REQUIRES` *module[/sub-module[/sub-module...]]* [`OR` *module[/sub-module...]*]...
  Specify a module requirement. Module requirements can be general, not specifying sub-modules, so that module dependencies can be independent of particular algorithms. For example, the statement `REQUIRES materials/eos` in a module's `Config` file indicates to `setup` that the materials/eos module is needed by this module. No particular equation of state is specified, but some EOS is needed, and as long as one is included by modules, the dependency will be satisfied. More specific dependencies can be indicated by specifying sub-modules. For example, `materials/eos` is a module with several possible sub-modules, each corresponding to a different equation of state. For example, to specify a requirement for the Helmholtz EOS, use `REQUIRES materials/eos/helmholtz`. Giving a complete set of module requirements is helpful to the end user, because `setup` uses them to generate the modules file when invoked with the -auto option.

- `CONFLICTS` *module1[/sub-module[/sub-module...]] ...*
  Specifies that the current module is not compatible with and may not be used with the specific module list that follows. Setup issues an error if the user attempts to set up a conflicting module configuration.

- `PARAMETER` *name type default*
  Specify 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.

- `VARIABLE name` *[attribute_list]*
  Register variable with the framework with name *name* and attributes defined by *attribute_list*. These variables can later be accessed by the program at runtime using the database accessor methods (see Sec. 5.1.2). Valid attributes are as follows:

  - `ADVECT/NOADVECT`
    A variable Q with the `ADVECT` property obeys an advection equation,

$$\frac{\partial Q}{\partial t} + \nabla \cdot (Q\mathbf{v}) = 0 \ . \tag{5.1}$$

- RENORM/NORENORM
  Variables $\{Q_i\}$ marked with the RENORM property obey the constraint

$$\sum_i Q_i = 1 \; . \tag{5.2}$$

- CONSERVE/NOCONSERVE
  Variables marked with the CONSERVE property obey conservation laws (*e.g.* momentum vs. velocity).

- LIBRARY *name*
  Specify a library requirement. Different FLASH modules require different external libraries, and they must inform `setup` so it can link the libraries into the executable. Valid library names are HDF4 and HDF5. Support for more libraries can be added by modifying the site-specific `Makefile.h` files to include appropriate Makefile macros.

- FLUX name
  Register flux variable *name* with the framework.

`Config` files also support the inclusion of special parameter comments. 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 the parameter's comment. 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 as

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

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

5.1.2 Interface layer and database module

After the module `Config` and `Makefile` are written, the source code files that carry out the specific work of the modules must be added. These source files can be separated into two broad categories, which we term "wrapper functions" and "algorithms." In this section we discuss how to construct wrapper functions.

When constructing a FLASH module, the designer must define a public interface of procedures that the module exposes to clients (*i.e.* other modules in an application). This is true regardless of the specific development language or syntactic features chosen to organize the procedures. These public functions are then defined in one or more source code files that form what we refer to as the interface layer.

Currently, there is no language-level formality in FLASH for enforcing the distinction between the public interface and private module functions that the interface harnesses. The developer is certainly encouraged to implement this within the chosen development language – static functions in C; private class functions in C++; private subroutines in a Fortran module, *etc.*. However, nothing in FLASH will force this distinction and carry out the associated name-hiding within an application.

The most important aspect of the interface/algorithm distinction is related to the rules for data access. Wrapper functions communicate directly with the FLASH `database` module to access grid data (see below). However, algorithms are not permitted to query the `database` module directly. Instead, they must receive all data via a formal function argument list. Thus, when a module A wishes to request the services of module B, A calls one of B's public wrapper functions. Rather than being required to pass all necessary data to B through a procedure argument list, B may "pull" data it needs access to from the database, marshal it as necessary, call the modules' algorithm(s), receive the updated data, and update the database. The following subsections describe the methods provided by the database module in more detail.

5.1.2.1 dBaseGetData/dBasePutData

Usage

```
call dBaseGetData([variable, [direction, [q1, [q2, [q3,]]]]] block, storage)
call dBasePutData([variable, [direction, [q1, [q2, [q3,]]]]] block, storage)
```

Description

Data exchange with grid variables. All variables registered with the framework via the `VARIABLE` keywords within a module Config file can be read/written with this pair of functions. The data is assumed to be composed of one or more structured blocks, each with an integer block id = [1,num_blocks], where num_blocks is the total number of blocks on a given processor.

Arguments

character(:)	`variable`	Variable names – these must match the names registered in the Config file; if no name is present, all variables will be exchanged
character(:)	`direction`	Used when exchanging a subsection of a particular block of data. Specifies the shape of the data and how `q1, q2, q3` should be interpreted; if not present, all elements of the specified variable will be exchanged
integer	`q1, q2, q3`	Coordinate of data exchanged in order i−j−k (see examples)
integer	`block`	Integer block ID on a given processor specifying the patch of data to access

real	`storage`
	`storage(:)`
	`storage(:,:)`
	`storage(:,:,:)`
		Allocated storage to receive result. Rank and shape of the storage array should match the rank and shape of data taken or put

List of valid variable directions

$$\texttt{direction} = \begin{cases} \text{“xyzCube”} \\ \text{“xyPlane”} \\ \text{“xzPlane”} \\ \text{“yzPlane”} \\ \text{“yxPlane”} \\ \text{“zxPlane”} \\ \text{“zyPlane”} \\ \text{“xVector”} \\ \text{“yVector”} \\ \text{“zVector”} \\ \text{“Point”} \end{cases}$$

Integers

Note that integer "key values" can be used in place of strings to specify variable names and directions. These are provided for performance reasons, when many accesses to the database are required (*e.g.* in a nested loop). These integers for variables and directions are not publicly available but can be accessed through the `dBaseKeyNumber()` routines.

Examples

Given a Config file with the variable registration specification

```
VARIABLE dens
```

the variable `dens` can be accessed from within FLASH as, for example,

```
use dBase, ONLY: nxb, nyb, nzb
real, dimension(nxb,nyb,nzb) :: density
do this_block = 1, total_blocks
  !get x-y-z cube of density at block this_block
  call dBaseGetData("dens", this_block, density)
  call foo(density)
end do

use dBase, ONLY: nxb, nzb
real, dimension(nxb,nzb) :: density
do this_block = 1, total_blocks
  do j = 1, nyb
    ! get an x-z slice of data at y-level j for block this_block
    call dBaseGetData("dens", "xzPlane", j, density)
    call foo(density)
  end do
end do

use dBase, ONLY: nxb
real, dimension(nxb) :: density
do this_block = 1, total_blocks
  do j = 1, nyb
    do k = 1, nzb
      ! get x-line of data at points j,k on block this_block
      call dBaseGetData("dens", "xVector", j, k, this_block, density)
      call foo(density)
    end do
   end do
end do
```

5.1.2.2 dBaseGetCoords/dBasePutCoords

Usage

```
call dBaseGetCoords(variable, direction, [q,] block, storage)
call dBasePutCoords(variable, direction, [q,] block, storage)
```

Description

Access global coordinate information for a given block, including guard cells.

Arguments

character(:)	`variable`	Specifies what coordinate information we want: left, right, or center coordinate of the zone or the zone width (see below)
character(:)	`direction`	Specifies x-, y-, or z-coordinate
integer	`q`	If given, allows user to get/put a single coordinate point for a specified direction. Default is to give the complete 1d array of all coords for the given block
integer	`block`	Integer block ID on a given processor
real	`storage` `storage(:)`	Allocated storage to receive result

Strings

$$\texttt{direction} = \begin{cases} \text{“xCoord”} \\ \text{“yCoord”} \\ \text{“zCoord”} \end{cases} \qquad \texttt{variable} = \begin{cases} \text{“znl”} : \text{left zone boundary} \\ \text{“zn”} : \text{zone center} \\ \text{“znr”} : \text{right zone boundary} \\ \text{“dz”} : \text{zone width} \end{cases}$$

Integers

Integers for variables and directions are not publicly available but can be accessed through `dBaseKeyNumber()`.

Example

```
real, DIMENSION(block_size) :: xCoords, data
do i = 1, lnblocks
  call dBaseGetCoords("zn", "xCoord", i, xCoords)
  call dBaseGetData("dens", "xVector", 0, 0, i, data)
  call foo(data,xCoords)
enddo
```

5.1.2.3 dBaseGetBoundaryFluxes/dBasePutBoundaryFluxes

Usage

```
call dBaseGetBoundaryFluxes(time_context, position, direction, block, storage)
call dBasePutBoundaryFluxes(time_context, position, direction, block, storage)
```

Description

Access boundary fluxes for all flux variables on a specified block associated with a given time level. Currently, only the current and previous timestep are supported.

Arguments

integer	`time_context`	Specifies fluxes stored at current or previous timestep
integer	`position`	Specifies left or right boundary in a given direction
character	`direction`	Specifies x-, y-, or z-coordinate
integer	`block`	Integer block ID on a given processor specifying the patch of data to access
real	`storage(:,:,:)`	Return buffer of size `nFluxes * dim1 * dim2`

Strings

$$\text{direction} = \begin{cases} \text{“xCoord”} \\ \text{“yCoord”} \\ \text{“zCoord”} \end{cases}$$

Integers

$$\texttt{time\_context} = \begin{cases} \text{-1: previous time-step} \\ \text{0: current time-step} \end{cases} \qquad \texttt{position} = \begin{cases} \text{0: left} \\ \text{1: right} \end{cases}$$

Example

```
real, dimension(nfluxes,nyb,nzb): xl_bound_fluxes, xr_bound_fluxes
do this_block = 1, num_blocks
  call dBaseGetBoundaryFluxes(0,0,"xCoord", this_block, xl_bound_fluxes)
  call dBaseGetBoundaryFluxes(0,1,"xCoord", this_block, xr_bound_fluxes)
  call foo(xl_bound_fluxes, xr_bound_fluxes)
  call dBasePutBoundaryFluxes(0,0,"xCoord",this_block,xl_bound_fluxes)
  call dBasePutBoundaryFluxes(0,1,"xCoord",this_block,xr_bound_fluxes)
end do
```

5.1.2.4 dBaseKeyNumber

Usage

```
result = dBaseKeyNumber(keyname)
```

Description

For faster performance, `dBase`{`Get`,`Put`}{`Data`,`Coords`} can be called with integer arguments instead of strings. Each of the string arguments accepted by the Get/Put methods can be replaced by a corresponding integer. However, these integers are not publicly available. To obtain them, one must call `dBaseKeyNumber()`.

Arguments and return type

character	`keyname`	String with "variable" or "direction" name, as in get/put data/coords; names of variables must match `Config` description
integer	`dBaseKeyNumber`	Integer assigned for the string key name

Strings

$$\texttt{keyname} = \begin{cases} \text{“xyzCube”} & & & \\ \text{“xyPlane”} & & & \\ \text{“xzPlane”} & \text{“RefineVariable1”} & \text{“OldTemp”} & \text{“rhoflx”} \\ \text{“yzPlane”} & \text{“RefineVariable2”} & \text{“Shock”} & \text{“uflx”} \\ \text{“yxPlane”} & \text{“RefineVariable3”} & \text{“znl”} & \text{“utflx”} \\ \text{“zxPlane”} & \text{“RefineVariable4”} & \text{“zn”} & \text{“uttflx”} \\ \text{“zyPlane”} & \text{“xCoord”} & \text{“znr”} & \text{“pflx”} \\ \text{“xVector”} & \text{“yCoord”} & \text{“znl0”} & \text{“eflx”} \\ \text{“yVector”} & \text{“zCoord”} & \text{“znr0”} & \text{“eintflx”} \\ \text{“zVector”} & & \text{“ugrid”} & \text{“nucflx\_begin”} \\ \text{“Point”} & & & \end{cases}$$

Typically, the call to `dBaseKeyNumber` is performed once, in a firstcall block at the top of a routine. The integer that stores the result will be declared with the Fortran `save` keyword, so the key value will be valid on subsequent entries into the routine.

Example

```
integer :: idens, ixCoord
idens   = dBaseKeyNumber("dens")
ixCoord = dBaseKeyNumber("xCoord")
call dBasePutData(idens, ixCoord, block_no, data)
```

5.1.2.5 dBaseSpecies

Usage

```
result = dBaseSpecies(index)
```

Description

Maps species number (from one to maximum number of species) to variable number (actual index in "unk" array).

Arguments and return type

integer	`index`	Species number from one to maximum number of species
integer	`dBaseSpecies`	Actual index in "unk" array

At present, all of the species are stored with adjacent indices in the solution array. Thus, one can find the index of the first isotope with a call to `dBaseSpecies(1)` and increment this value by 1 to get the next species.

5.1.2.6 dBaseVarName

Usage

```
result = dBaseVarName(keynumber)
```

Description

Given a key number, return the associated variable name.

Arguments and return type

integer	`keynumber`	Variable keynumber obtained with call to `dBaseKeyNumber`
character(len = 4)	`dBaseVarName`	String name of variable as defined in `Config`; if variable does not exist, returns "null"

Example

```
integer       :: idens
char(len = 4) :: name
idens = dBaseKeyNumber("dens")
name  = dBaseVarName(idens)     ! name now = "dens"
```

5.1.2.7 dBasePropertyInteger

Usage

```
result = dBasePropertyInteger(property)
```

Description

Accessor methods for integer-valued scalar variables.

Arguments and return type

character	`property`	String with variable name (see below)
integer	`dBasePropertyInteger`	Property value

Strings

`property` =

"Dimensionality"	Dimensionality of problem defined at setup
"NumberOfVariables"	Total number of solution variables defined
"NumberOfSpecies"	Number of nuclear species defined
"NumberOfGuardCells"	Width of guard cell region on each boundary of a block
"NumberOfFluxes"	Total number of fluxes defined
"NumberOfNamedVariables"	Total number of solution variables excluding nuclear abundances
"NumberOfAdvectVariables"	Total number of variables with the ADVECT attribute
"NumberOfRenormVariables"	Total number of variables with the RENORM attribute
"NumberOfConserveVariables"	Total number of variables with the CONSERVE attribute for a given problem
"MaxNumberOfBlocks"	Total number of allocated blocks on a given processor. May exceed the number of blocks currently defined in the AMR hierarchy, since block memory is allocated statically.
"LocalNumberOfBlocks"	Total number of blocks on a given processor
"MaxBlocks_tr"	Statically allocated buffer size for work arrays
"NumberOfGuards_work"	Guard cells for scratch array
"xDimensionExists"	Value of 1 if x-dimension is defined for given problem, 0 otherwise
"yDimensionExists"	Value of 1 if y-dimension is defined for given problem, 0 otherwise
"zDimensionExists"	Value of 1 if z-dimension is defined for given problem, 0 otherwise
"xBlockSize"	Number of zones in x-direction for AMR blocks, excluding guard cells
"yBlockSize"	Number of zones in y-direction for AMR blocks, excluding guard cells
"zBlocksize"	Number of zones in z-direction for AMR blocks, excluding guard cells
"xLowerBound"	Beginning x-index of a block (including guard cells)
"yLowerBound"	Beginning y-index of a block (including guard cells)
"zLowerBound"	Beginning z-index of a block (including guard cells)
"xUpperBound"	Ending x-index of a block (including guard cells)
"yUpperBound"	Ending y-index of a block (including guard cells)
"zUpperBound"	Ending z-index of a block (including guard cells)
"CurrentStepNumber"	Current time-step number
"BeginStepNumber"	Initial time-step number
"MyProcessor"	Local processor ID assigned by MPI
"MasterProcessor"	Master processor ID
"NumberOfProcessors"	Total number of processors

5.1.2.8 dBasePropertyReal

Usage

```
result = dBasePropertyReal(property)
```

Description

Accessor methods for real-valued scalar variables.

Arguments and return type

character	`property`	String with variable name, see below
integer	`dBasePropertyReal`	Property value

Strings

property =		
	"Time"	Current value of the simulation time
	"OldTimeStep"	Value of the timestep at the old time
	"Redshift"	Current value of the redshift
	"OldRedshift"	Value of the redshift at the old time
	"ScaleFactor"	Current value of the scale factor
	"OldScaleFactor"	Value of the scale factor at the old time
	"CPUSeconds"	Amount of computer time used
	"TimeStep"	Current value of the simulation timestep

5.1.2.9 dBaseSetProperty

Usage

```
call dBaseSetProperty(property, value)
```

Description

Mutator methods for writable scalar variables. See `dBasePropertyInteger/Real` for documentation on property names.

Arguments

character	`property`	String with variable name
integer real	`value`	New property value

Strings

$$\texttt{property} = \begin{cases} \text{“CurrentStepNumber”} \\ \text{“BeginStepNumber”} \\ \text{“MyProcessor”} \\ \text{“MasterProcessor”} \\ \text{“NumberOfProsessors”} \\ \text{“Time”} \\ \text{“TimeStep”} \\ \text{“OldTimeStep”} \\ \text{“Redshift”} \\ \text{“CPUSeconds”} \\ \text{“OldRedshift”} \\ \text{“OldSacaleFactor”} \\ \text{“ScaleFactor”} \end{cases}$$

5.1.2.10 Various pointer-returning functions

Each of these functions allows FLASH developers to hook directly into an internal data structure in the database. In general, these functions will offer better performance then their corresponding `dBaseGet/Put` counterparts and will require less memory overhead. However, the interfaces are more complicated and the functions are less flexible and less safe, so it is suggested that developers strongly consider using `dBaseGet/PutData` when performance differences are small.

Each function returns a Fortran 90 pointer to the solution vector on the specified block. If no block is specified, a pointer is returned to all blocks on the calling processor. Currently the array index layout is assumed to be `(var,nx,ny,nz,block)` in row-major ordering. The scratch (`unksm`) array stores variables with no guard cells; this name should probably be changed in the future.

Argument and return type

integer	`block`
real, DIMENSION(:,:,:,:,:), POINTER	`dBaseGetDataPtrAllBlocks`
real, DIMENSION(:,:,:,:), POINTER	`dBaseGetDataPtrSingleBlock`
real, DIMENSION(:,:,:), POINTER	`dBaseGetPtrToXCoords`
real, DIMENSION(:,:,:), POINTER	`dBaseGetPtrToYCoords`
real, DIMENSION(:,:,:), POINTER	`dBaseGetPtrToZCoords`
real, DIMENSION(:,:,:,:,:), POINTER	`dBaseGetScratchPtrAllBlocks`
real, DIMENSION(:,:,:,:), POINTER	`dBaseGetScratchPtrSingleBlock`

`dBaseGetDataPtrAllBlocks()`

Return an F90 pointer to the left-hand-side solution vector for all blocks on a given processor, arranged in row-major order as `(var,nx,ny,nz,block)`. `dBaseKeyNumber` must still be called to access the elements of the array.

`dBaseGetDataPtrSingleBlock(block_no)`

Return an F90 pointer to the left-hand-side solution vector on a specified block, arranged as `(var,nx,ny,nz)`.

`dBaseGetPtrToXCoords()`

Return an F90 pointer to an array containing information on the x-coordinates of the AMR blocks. The array returned is arranged as `(block_position, i, block_number)`, where `block_position` values denote center, left, or right coordinates, and are obtained by calling `dBaseKeyNumber` with “zn”, “znl”, “znr” and using the corresponding index to access the appropriate row in the array. For example,

```
real, pointer, dimension(:,:,:) :: xCoords
real                            :: x, xl, xr
integer                         :: izn, iznl, iznr
izn  = dBaseKeyNumber("izn")
iznl = dBaseKeyNumber("iznl")
iznr = dBaseKeyNumber("iznr")
xCoord => dBaseGetPtrToXCoords()
do this_block = 1, num_blocks
  do i = 1, blocksize
    x  = xCoord(izn, i, this_block)     ! get first center coord
    xl = xCoord(iznl, i, this_block)    ! get first left coord
    xr = xCoord(iznr, i, this_block)    ! get first right coord
    ...
  enddo
enddo
```

`dBaseGetPtrToYCoords()`
See `dBaseGetPtrToXCoords()`.

`dBaseGetPtrToZCoords()`
See `dBaseGetPtrToXCoords()`.

`dBaseGetScratchPtrAllBlocks()`
Return an F90 pointer to a scratch array of size `(2, nxb, nyb, nzb, maxblocks)`.

`dBaseGetScratchPtrSingleBlock(block_no)`
Return an F90 pointer to a scratch array of size `(2, nxb, nyb, nzb)`.

5.1.2.11 AMR tree interface functions

These functions enable FLASH developers to directly access the data structures used by PARAMESH to describe the adaptive mesh. In general they should not be needed by developers of physics modules. Also, they may not be available in future versions of FLASH.

Arguments and return types

integer	`block`
integer, DIMENSION (max_faces)	`dBaseNeighborBlockList`
integer, DIMENSION (max_faces)	`dbaseNeighborBlockProcList`
integer, DIMENSION (max_child)	`dBaseChildBlockList`
integer, DIMENSION (max_child)	`dBaseChildBlockProcList`
integer	`dBaseParentBlockList`
integer	`dBaseParentBlockProcList`
integer	`dBaseRefinementLevel`
integer, DIMENSION (max_faces)	`dbaseNeighborType`
real, DIMENSION (max_dim)	`dBaseBlockCoord`
real, DIMENSION (max_dim)	`dBaseBlockSize`
integer	`dBaseNodeType`
logical	`dBaseRefine`
logical	`dBaseDeRefine`

dBaseNeighborBlockList (block)

Given a block ID, return an array of block ID's that are the neighbors of the specified block. The returned array is of size `max_faces` = 6, but not all of the six elements will have meaningful values if the problem is run in fewer than three dimensions. Assuming the function is called as `NEIGH = dBaseNeighborBlockList()`, the ordering is as follows. The neighbor on the lower x-face of block L is at NEIGH(1,L), the neighbor on the upper x-face at NEIGH(2,L), the lower y-face at NEIGH(3,L), the upper y-face at NEIGH(4,L), the lower z-face at NEIGH(5,L), and the upper z-face at NEIGH(6,L). If any of these values are set to -1 or lower, there is no neighbor to this block at its refinement level. However there may be a neighbor to this block's parent. If the value is -20 or lower, then this face lies on an external boundary, and the user is required to apply some boundary condition there. In this case, the value is used to specify which type of boundary condition the user wishes to implement.

dbaseNeighborBlockProcList (block)

Given a block ID, return an array of size `max_faces` = 6 elements containing processor ID's identifying the processor on which a given neighbor resides. Ordering is identical to `dBaseNeighborBlockList()`.

dBaseChildBlockList (block)

Given a block ID, return an array of size `max_child` = 2 * `max_dim` elements containing the block ID's of the child blocks of the specified block. The children of a parent are numbered according to the Fortran array ordering convention, *i.e.* child 1 is at the lower x, y, and z corner of the parent, child 2 at the higher x coordinate but lower y and z, child 3 at lower x, higher y and lower z, child 4 at higher x and y and lower z, and so on.

dBaseChildBlockProcList (block)

Given a block ID, return an array of size `max_child` elements containing processor ID's of the children of the specified block. Ordering is identical to `dBaseChildBlockList()`.

dBaseParentBlockList (block)

Given a block ID, return the ID of the block's parent block.

dBaseParentBlockProcList (block)

Given a block ID, return the processor ID upon which the block's parent resides.

dBaseRefinementLevel (block)

Given a block ID, return that block's integer level of refinement.

dBaseNodeType (block)

Given a block ID, return the block's node type. If the node type is 1, then the node is a leaf node. If it is 2, then the node is a parent with at least 1 leaf child. Otherwise, it is set to 3 and does not have any up-to-date data.

dbaseNeighborType (block)

Given a block ID, return an array of size `(max_faces, maxblocks_tr)` containing the type ID's of the neighbors of the specified block. `max_faces` = `max_dim` * 2, where `max_dim` is the maximum possible dimensionality (3).

dBaseBlockCoord (block)

Given a block ID, return an array of size `max_dim` containing the x-,y-,and z-coordinates of the center of the block.

dBaseBlockSize (block)

Given a block ID, return an array of size `max_dim` containing the block size in the x-, y-, and z-directions.

dBaseRefine (block)
Given a block ID, return `.true.`, if that block is set for refinement when `amr_refine_derefine()` is called next and `.false.` otherwise.

dBaseDeRefine (block)
Given a block ID, return `.true.`, if that block is set to be derefined when `amr_refine_derefine()` is called next and `.false.` otherwise.

5.1.3 Algorithms

Within each module is one or more procedures that perform the bulk of the computational work for the module. A principal strategy behind the FLASH architecture is to decouple these procedures as much as possible from the details of the framework in which they are embedded. This is accomplished by requiring that all module algorithms communicate data only through function argument lists. That is, algorithms may not query the database directly nor may they depend on the existence of externally defined or global variables. This design ensures that algorithms can be tested, developed, and interchanged in complete isolation from the larger, more complicated framework.

Thus, each algorithm in a module should have a well-defined argument list. It is up to the algorithm developer to make this as general or restrictive as he/she sees fit. However, it is important to keep in mind that the more rigid the argument list, the less chance that another algorithm can share its interface. The consequence is that the developer would have to add an entirely new wrapper function for just slightly different functionality.

5.2 The FLASH source tree

An abstract representation of the FLASH architecture appears in Fig. 5.1. Each box in this figure represents a component (FLASH module), which publishes a small set of public methods to its clients. These public methods are expressed through virtual function definitions (stubs under Fortran 90), which are implemented by real functions supplied by sub-modules. Typically, each component represents a different class of solver. For instance, for time-dependent problems, the driver uses time-splitting techniques to compose the different solvers, which are divided into different classes on the basis of their ability to be composed in this fashion and upon natural differences in solution method (*e.g.*, hyperbolic solvers for hydrodynamics, elliptic solvers for radiation and gravity, ODE solvers for source terms, *etc.*).

The adaptive mesh refinement module is treated in the same way as the solvers. The means by which the driver shares data with the solver objects is the primary way in which the architecture affects the overall performance of the code. Choices here, in order of decreasing performance and increasing flexibility, include global memory, argument-passing, and messaging. FLASH 2.2 has eliminated global variable access in favor of a well-defined set of accessor and mutator methods managed by the centralized database module. When done with an eye toward optimization, the effects on performance are tolerable, and the benefits for maintainability and extensibility are significant. This is discussed in greater detail below.

5.2.1 Code infrastructure

The structure of the FLASH source tree reflects the module structure of the code, as shown in Fig. 5.2. The source code is organized into one set of directories, while the code is built in a separate directory using links to the appropriate source files. The links are created by a source configuration script called `setup`, which makes the links using options selected by the user and then creates an appropriate makefile in the build directory. The user then builds the executable with a single invocation of `gmake`.

Source code for each of the different code modules is stored in subdirectories under `source/`. The code modules implement different physics, such as hydrodynamics, nuclear burning, and gravity, or different major program components, such as the main driver code and the input/output code. Each module directory contains source code files, makefile fragments indicating how to build the module, and a configuration file (see Chapter 3).

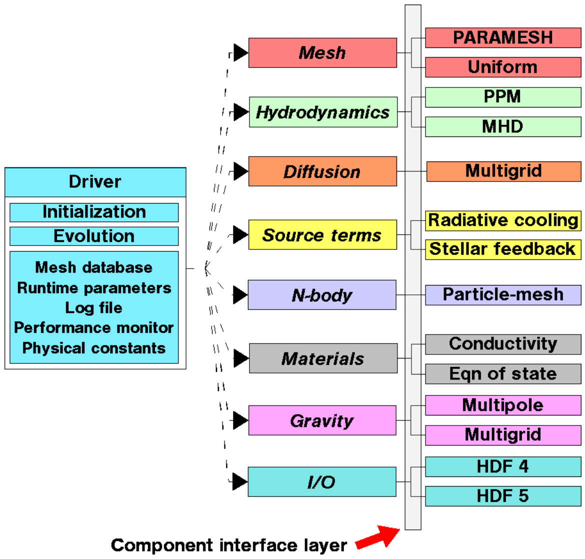


Figure 5.1: Abstract representation of the FLASH architecture.

Each module subdirectory may also have additional sub-module directories underneath it. These contain code, makefiles, and configuration files specific to different variants of the module. For example, the `hydro/` module directory (as shown in Fig. 9.1) can contain files which are generic to hydrodynamical solvers, while its `explicit/` subdirectory contains files specific to explicit `hydro` schemes, and its `implicit/` subdirectory contains files specific to implicit solvers. Configuration files for other modules which need hydrodynamics can specify hydro as a requirement without mentioning a specific solver; the user can then choose one solver or the other when building the code (via the modules file (Chapter 3)).

When `setup` configures the source tree, it treats each sub-module as inheriting all of the source code, configuration files, and makefiles in its parent module's directory, so generic code does not have to be duplicated. Sub-modules can themselves have sub-modules, so for example, one might have `hydro/explicit/split/ppm` and `hydro/implicit/ppm`. Source files at a given level of the directory hierarchy override files with the same name at higher levels, whereas makefiles and configuration files are cumulative. This permits modules to supply stub routines that are treated as 'virtual functions' to be overridden by specific sub-modules, and it permits sub-module directories to be self-contained.

When a module is not explicitly included by `Modules`, only one thing is done differently by `setup`: sub-modules are not included, except for a `null` sub-module, if it is present. Most top-level modules should contain only stub files to be overridden by sub-modules, so this behavior allows the module to be 'not included' without extra machinery (such as the special stub files and makefiles required by earlier versions of FLASH). In those cases in which the module's files are not appropriate for the 'not included' case, the `null` sub-module allows one to override them with appropriate versions.

New solvers and new physics can be added. At the current stage of development of FLASH it is probably best to consult the authors of FLASH (see Sec. 22.3) for assistance in this. Some general guidelines for adding solvers to FLASH 2.2 may be found in Chapter 21.

The `setups/` directory has a structure similar to that of `source/`. In this case, however, each of the "modules" represents a different initial model or problem, and the problems are mutually exclusive; only one is included in the code at a time. Also, the problem directories have no equivalent of sub-modules. A makefile fragment specific to a problem need not be present, but if it is, it is called `Makefile`. Chapter 4 describes how to add new problem directories.

The `setup` script creates a directory called `object/` in which the executable is built. In this directory, `setup` creates links to all of the source code files found in the specified module and sub-module directories as well as the specified problem directory. (A source code file has the extension `.c, .C, .f, .f90, .F90, .F, .fh`, or `.h`.) Because the problem `setup` directory and the machine-dependent directory are scanned

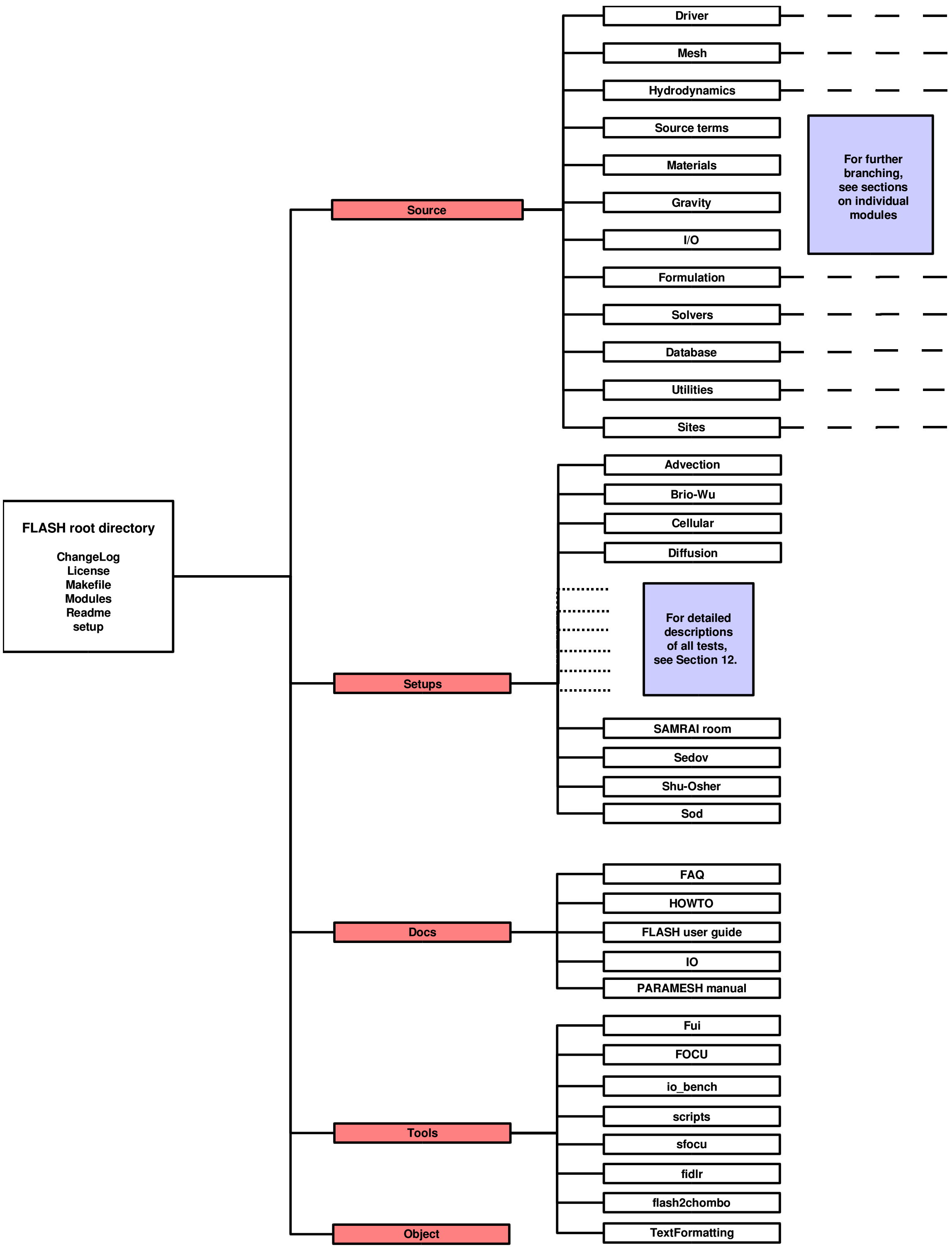


Figure 5.2: Directory structure of FLASH 2.2.

last, links to files in these directories override the "defaults" taken from the `source/` tree. Hence special variants of routines needed for a particular problem can be used in place of the standard versions by simply giving the files containing them the same names.

Using information from the configuration files in the specified module and problem directories, `setup` creates a file named `init_global_parms.F90` to parse the runtime parameter file and initialize the runtime parameter database. It also creates a file named `rt_parms.txt`, which concatenates all of the PARAMETER statements found in the appropriate configuration files and so can be used as a "master list" of all of the runtime parameters available to the executable.

`setup` also creates makefiles in `object/` for each of the included modules. Each copy is named *Makefile.module*, where *module* is `driver`, `hydro`, `gravity`, and so forth. Each of these files is constructed by concatenating the makefiles found in each included module path. Including, *e.g.*, `hydro/explicit/split/ppm` causes `Makefile.hydro` to be generated from files named `Makefile` located in `hydro/`, `hydro/explicit/`, `hydro/explicit/split/`, and `hydro/explicit/split/ppm/`. If the module is not explicitly included, then only `hydro/Makefile` is used, under the assumption that the subroutines at this level are to be used when the module is not included. The `setup` script creates a master makefile (`Makefile`) in `object/` which includes all of the different modules' makefile fragments together with the site- or operating system-dependent `Makefile.h`.

The master `Makefile` created by `setup` creates a temporary subroutine, `buildstamp.F90`, which echoes the date, time, and location of the build to the FLASH log file when FLASH is run. To ensure that this subroutine is regenerated each time the executable is linked, the `Makefile` deletes `buildstamp.F90` immediately after compiling it.

The `setup` script can be run with the `-portable` option to create a directory with real files which can be collected together with `tar` and moved elsewhere for building. In this case the build directory is assigned the name `object_problem/`. Further information on the options available with `setup` may be found in Chapter 3.

Additional directories included with FLASH are `tools/`, which contains tools for working with FLASH and its output (see Sec. 19), and `docs/`, which contains documentation for FLASH (including this user's guide) and the PARAMESH library.

5.3 Modules included with FLASH: a brief overview

The current FLASH distribution comes with a set of core components that form the backbone of many common problems, namely database, driver, hydro, io, mesh, particles, source_terms, gravity, and materials. A detailed discussion of the role of each of these modules is presented in Part IV. Here we give a brief overview of each.

Chapter 6 describes in detail the various driver modules that may be implemented with FLASH. In addition to the default driver, which controls the initialization, evolution, and output of a FLASH simulation, four new driver modules have been written to implement different explicit time advancement algorithms. Three are written in the delta formulation: `euler1`, `rk3`, and `strang_delta`. The fourth, `strang_state`, is written in the state-vector formulation. This chapter also includes a subsection concerning simulation services, runtime parameters, and logfiles.

Chapter 7 describe the FLASH I/O modules, which control how FLASH data structures are stored on different platforms and in different formats. Discussed in this section are the I/O module `hdf4`, which uses the Hierarchical Data Format (HDF) for storing simulation data, and the two major HDF5 modules (serial and parallel versions).

Chapter 8 describes the mesh module, together with the PARAMESH package of subroutines for the parallelization and adaptive mesh refinement (AMR) portion of FLASH.

Chapter 9 describes the two hydrodynamic modules included in FLASH 2.x. The first is based on the PROMETHEUS code (Fryxell, Müller, and Arnett 1989); the second is based on Kurganov numerical methods. Sec. 9.3 describes the magnetohydrodynamics module included with the FLASH code.

Chapter 10 discusses the material properties module, which handles the tracking of multiple fluids in FLASH simulations. It includes the equation of state module, which implements the EOS for the hydrodynamical and nuclear burning solvers; the composition submodule, which sets up the different compositions needed by FLASH; and the stellar conductivity module, which may be used for computing the opacity of stellar material.

Chapter 11 describes various source terms, including the nuclear burning module, which calculates the nuclear burning rate during a hydrodynamical simulation, and the stirring module, which adds a divergence-free, time-correlated 'stirring' velocity at selected modes in a given hydrodynamical simulation. Also included are modules for ionization, heating, and cooling.

Chapter 12 describes the gravity module, which computes the gravitational potential or gravitational acceleration source terms for the code. It includes several sub-modules: the `constant` submodule, the plane parallel sub-module, the `ptmass` submodule, and the Poisson submodules.

Chapter 13 describes the particle module, which follows the evolution of both physical particles and Lagrangian mass tracers.

Chapter 14 describes the cosmology module, which provides features required for solving cosmological problems in comoving coordinates. These include the evolution of the scale factor (via the Friedmann equation), the redshift terms in the comoving Euler equations, and a library of useful cosmological functions (*e.g.*, to convert redshifts to times).

Chapter 15 describes several solvers included with FLASH, including solvers for ordinary differential equations (ODE) and multipole and multigrid Poisson solvers.

Chapter 16 describes the 2d runtime visualization module used to produce simple pictures of a FLASH simulation.

Finally, Chapter 17 describes the utilities module, which is a collection of reusable high-level utility functions that simplify programming in FLASH.

Chapter 6

Driver modules

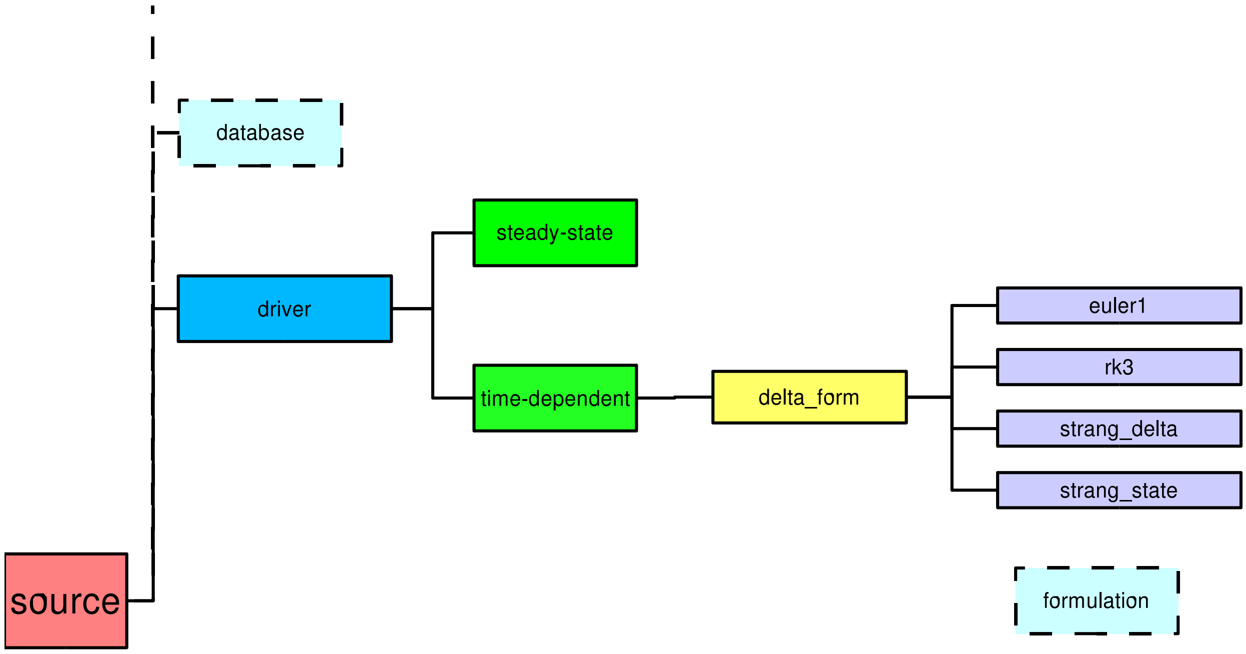


Figure 6.1: The driver module directory.

The `driver` module controls the initialization, evolution, and output of FLASH simulations. Initialization can be from scratch or from a stored checkpoint file. The drivers can use any of several different operator-splitting techniques to combine different physics operators and integrate them in time or can call a single operator if the problem of interest is not time-dependent. Output involves the production of checkpoint files, plot files, analysis data, and log file time stamps. In addition to these functions, the driver supplies important simulation services to the rest of the FLASH framework, including Fortran modules to handle runtime parameters, physical constants, memory usage reports, and log file management (these are discussed further in Chapter 7).

The initialization and termination routines and the simulation services modules are common to both time-dependent and time-independent drivers and thus are included at the highest level of `driver`. The file `flash.F90` contains the main FLASH program (equivalent to `main()` in C) and calls these routines as needed. The default `flash.F90` is empty and is intended to be overridden by submodules of `driver`. At this time only time-dependent drivers are supplied with FLASH; these are submodules of the `driver/time_dep` module. The `time_dep` version of `flash.F90` calls the FLASH initialization routine, loops over timesteps, and then calls the FLASH termination routine. During the time loop, it computes new timesteps, calls an evolution routine (`evolve()`), and calls output routines as necessary.

The details of each available time integration method are completely determined by the version of `evolve()` supplied by that method. The default time update method is to call each physics module's update routine for two equal timesteps – thus, hydro, source terms, gravity, hydro, source terms, gravity. The hydrodynamics update routines take a "sweep order" argument in case they are directionally split; in this case, the first call uses the ordering $x-y-z$, and the second call uses $z-y-x$. Each of the update routines is assumed to directly modify the solution variables. At the end of each pair of timesteps, the condition for updating the mesh refinement pattern is tested, and a refinement update is carried out if required.

The alternative "delta formulation" drivers (`driver/time_dep/delta_form`) modify a set of variables containing the *change* in the solution during the timestep. The change is only applied to the solution variables after all operators have been invoked. This technique permits more general time integration methods, such as Runge-Kutta methods, to be employed, and it provides a more flexible method for composing operators. However, only a few physics modules can make use of it as yet. More details on the delta formulation drivers appear in Section 6.1.

The driver module supplies certain runtime parameters regardless of which type of driver is chosen. These are described in Table 6.1.

Table 6.1: `driver` module parameters.

Parameter	Type	Default	Description
`nend`	integer	100	Maximum number of timesteps to take before halting the simulation
`restart`	boolean	.false.	Set to .true. to restart the simulation from a checkpoint file
`run_number`	string	""	Identification number for run
`run_comment`	string	""	Identifying comment for run
`log_file`	string	"flash.log"	Name of log file
`tinitial`	real	0.	Initial simulation time
`tmax`	real	1.	Maximum simulation time to advance before halting the simulation
`zinitial`	real	-1.	Initial simulation redshift (ignored if < 0; used to set `tinitial` if > 0)
`zfinal`	real	-2.	Final simulation redshift (ignored if < 0)
`dtini`	real	10^{-10}	Initial timestep
`dtmin`	real	10^{-10}	Minimum timestep
`dtmax`	real	10^{5}	Maximum timestep
`small`	real	10^{-10}	Generic small cutoff value for dimensionless positive definite quantities

Table 6.1: `driver` module parameters (continued).

Parameter	Type	Default	Description
`smlrho`	real	10^{-10}	Cutoff value for density
`smallp/e/t/u/x`	real	10^{-10}	Cutoff values for pressure, energy, temperature, velocity, and advected abundances
`x/y/zmin`	real	0	Minimum x, y, and z coordinates for grid
`x/y/zmax`	real	1	Maximum x, y, and z coordinates for grid
`geometry`	string	`"cartesian"`	Grid geometry — valid values are `"cartesian"`, `"cylindrical"`, and `"spherical"`
`igrav`	integer	0	If set to 1, use gravity
`iburn`	integer	0	If set to 1, use nuclear burning
`iheat`	integer	0	If set to 1, use heating processes
`icool`	integer	0	If set to 1, use cooling processes
`wall_clock_time_limit`	real	604800	Maximum simulation time in seconds
`print_tstep_loc`	boolean	.false.	when .true., it prints the x,y, and z coordinates of the zone that is determining the timestep (for all limiters).

6.1 Delta-formulation and Strang-state driver modules

These driver modules implement different explicit time advancement algorithms. This usage is slightly different than that of the default driver module, which does not directly implement a time advancement algorithm; the default driver and hydro modules each implement parts of the Strang splitting time advancement. In this section are listed the time advancement tasks common to all of the alternative drivers. In the following subsections, the details of each time advancement method will be described.

The three driver modules written in the delta formulation are `euler1`, `rk3`, and `strang_delta`. They make appropriate calls to the physics modules and update the solution by calling functions provided by the formulation module. The `strang_state` driver is written in the state-vector formulation; it also calls the physics modules, but does not update the solution. To use these modules, first choose the driver by including *one* of the following lines in the `Modules` file:

```
INCLUDE driver/time_dep/delta_form/euler1
INCLUDE driver/time_dep/delta_form/rk3
INCLUDE driver/time_dep/delta_form/strang_delta
INCLUDE driver/time_dep/delta_form/strang_state
```

The directory names for some of the alternative modules are misleading. All the alternative time advancements are in a directory named `delta_form`, regardless of their formulation.

The time advancement module determines which formulation module should be used; two instantiations are possible. For `euler1`, `rk3`, or `strang_delta`, specify

```
INCLUDE formulation/state_form/delta_form
```

but for `strang_state` specify

```
INCLUDE formulation/state_form
```

The services provided by the formulation module for the delta formulation are a superset of those provided for the state-vector formulation, which explains the directory structure used. For both instantiations, the formulation module contains (i) subroutines for updating the conserved and auxiliary variables locally (on a block or on a face of a block) given a local operator $L_{physics}(U)$ and (ii) a parameter which declares which formulation is being used. For the delta formulation, the module also (iii) declares the global ΔU array and contains subroutines for accessing it and (iv) provides a subroutine to update the variables globally.

For delta formulation time advancements, the delta formulation driver modules use the formulation module to hold and access the global ΔU array and to update the solution. In the state-vector formulation, the formulation module is not directly used by the driver; instead, the physics modules call the update subroutines that the formulation module provides.

The alternative driver modules discretize the left-hand side of

$$\frac{\partial V}{\partial t} = \text{(spatial difference terms)} + \text{(source terms)} \ . \tag{6.1}$$

The time advancement algorithm is contained in a subroutine named `evolve`. Each call to `evolve` advances the solution through one timestep for the `euler1` and `rk3` modules and through two timesteps for the `strang_state` and `strange_delta` modules. Each time advancement algorithm begins with a vector of primary variables V^n at time t^n and an associated set of auxiliary variables W^n. Primarily through calls to other physics modules, `evolve` applies a set of operations to the variables to produce an updated vector V^{n+1} at time $t^{n+1} = t^n + \Delta t$. Depending on the formulation, the time advancement may or may not update the auxiliary variables – in the state-vector formulation, the other physics modules update them.

The distinction between V and W is that time-dependent differential equations are solved to determine the primary variables. The auxiliary variables are obtained from the primary variables through algebraic relations. Often the primary variables are the conserved variables U, and in the rest of this section, U will replace V. However, the time advancement algorithms implemented do not require this correspondence.

The time advancement algorithms are written generally, in that each differential equation is treated in the same way. The distinction between the equations (for example, between the x-momentum equation and the total energy equation) is expressed in the other physics modules. The time advancement algorithm does not need to know the identity of the variables on which it operates, except possibly to update the auxiliary variables from the primary variables, but this update is handled by a call to a subroutine provided by the formulation module.

6.1.1 The `euler1` module

The `euler1` module implements the first-order, Euler explicit scheme

$$U^{n+1} = U^n + \Delta t L(U^n) \ , \tag{6.2}$$

where $L(U)$ represents all of the physics modules. The Euler explicit method is implemented in the delta formulation. No runtime parameters are defined for this module.

At the beginning of a timestep, ΔU is set to zero. Each of the physics modules is called with U^n as the initial state and adds its contribution to ΔU. After all the physics modules have been called, the global ΔU array holds $L(U^n)$. Eq. (6.2) yields U^{n+1}. Finally, the auxiliary variables are updated from the conserved variables with a call to the global update subroutine provided by the formulation module.

Note that because all the physics modules start with the same initial state, the order in which the physics modules are called does not affect the results (except possibly through floating point roundoff differences when contributing to ΔU).

The set of steps, consisting of calls to physics modules, updating the conserved variables, and updating the auxiliary variables, is often called a *stage*. The majority of the computational cost of a stage is in the calls to the other physics modules; this component corresponds to a "function evaluation" for ordinary differential equation solvers. In the Euler explicit algorithm, there is one stage per timestep.

6.1.2 The `rk3` module

Runge-Kutta schemes are a class of ordinary differential equation solvers which are appreciated for their higher order of accuracy, ease of implementation, and relatively low storage requirements. There are many third-order Runge-Kutta methods; all require at least three stages. Most require at least three storage locations per primary variable, but the one implemented in the delta formulation in FLASH, derived by Williamson (J. Comp. Phys. 35:48, 1980), requires only two.

The two storage registers will be referred to as U and ΔU. The global solution vector U holds U^n at the beginning of the timestep and then intermediate solutions $U^{(\cdot)}$ at the end of each stage. The manipulation of the global ΔU array is more complicated. ΔU accumulates contributions from the physics modules during a stage, but it also holds results from previous stages; it is important to distinguish between the results of the physics modules $L(U)$ and the quantity held in the global ΔU array. First the algorithm will be shown; then the usage of the global ΔU array will be discussed. For the equation

$$\frac{\partial U}{\partial t} = L(U) \ , \tag{6.3}$$

Williamson's algorithm is, starting with U^n,

$$\begin{aligned}
U^{(1)} &= U^n + \frac{1}{3}\Delta t\Big[L(U^n)\Big] && (6.4)\\
U^{(2)} &= U^{(1)} + \frac{15}{16}\Delta t\Big[L(U^{(1)}) - \frac{5}{9}L(U^n)\Big] && (6.5)\\
U^{n+1} = U^{(3)} &= U^{(2)} + \frac{8}{15}\Delta t\Big[L(U^{(2)}) - \frac{153}{128}\Big(L(U^{(1)}) - \frac{5}{9}L(U^n)\Big)\Big] \ . && (6.6)
\end{aligned}$$

$U^{(m)}$ is the result of the m-th stage, and the auxiliary variables are updated each time a new $U^{(m)}$ is computed.

The algorithm is implemented using the following steps to attain the low storage. At the beginning of the timestep, ΔU is set to zero. During the first stage each physics module contributes to ΔU, so after all have contributed, ΔU holds the bracketed term in eq. (6.4), $L(U^n)$. $U^{(1)}$ is then computed using eq. (6.4). Stage 1 is completed by multiplying ΔU by $-5/9$, which is required for the following stages. The process is repeated for stage 2: after the physics modules have contributed, ΔU holds the bracketed term in eq. (6.5); $U^{(2)}$ is computed by eq. (6.5) and stored in U; then ΔU is multiplied by $-153/128$. Stage 3 is similar, but ends after $U^{(3)} = U^{n+1}$ is computed and stored. It is critical that the only changes made to the ΔU array are those just listed; no physics module should change the value of ΔU, except to add its contribution, and since ΔU holds information from previous stages, it should not be reset to zero except at the beginning of the timestep. No runtime parameters are defined for this module.

6.1.3 `strang_state` and `strang_delta` modules

The second-order accurate splitting method (Strang 1968) is attractive because of its low memory requirements. The algorithm is based on the operator splitting approach, in which a set of simple subproblems is solved rather than a single, complicated problem. Each subproblem typically accounts for one term in a system of partial differential equations, representing a particular type of physics for which an appropriate (specialized) numerical method is available. If all the subproblems are computed to at least second-order accuracy, the basic operator splitting method is first-order accurate; however, the Strang splitting scheme recovers second-order accuracy over two timesteps. In the first timestep, the subproblems are solved in a given sequence. Second-order accuracy is obtained by reversing the sequence in the second timestep.

A key feature of the operator splitting approach is that the output of one subproblem is the input to the next subproblem. This allows for an implementation that globally, stores only the current solution, but it can also cause problems including accuracy losses due to decoupling various physical effects (splitting errors) and difficulties implementing boundary conditions. In practice it has been found that splitting errors are reduced when the subproblems are ordered in increasing stiffness, *i.e.* the stiffest subproblem is solved last in the sequence; this has recently been supported by numerical analysis (Sportisse 2000).

Two driver modules implement an algorithm similar to the Strang splitting time advancement. Since the sequence is not exactly reversed in the second step compared to the first, the algorithm is not the true

Strang splitting. However, the nuclear burning source terms are very stiff, and there are sound arguments for computing them last. The `strang_state` module implements the algorithm in the state-vector formulation and is recommended for "production" runs due to its low memory requirements. The `strang_delta` driver is implemented in the delta formulation and is provided for testing and comparison. For both versions, one call to `evolve` (which implements the time advancement algorithm) advances the solution from t^n to t^{n+2}, *i.e.* over *two* timesteps.

In the `strang_state` driver, the sequence of calls to physics modules in the first timestep is

hydro(x-sweep)
hydro(y-sweep)
hydro(z-sweep)
gravity
source terms

In the second timestep, only the order of the hydro calls is reversed

hydro(z-sweep)
hydro(y-sweep)
hydro(x-sweep)
gravity
source terms

Mesh refinement and derefinement are executed only after the second step, not between the two steps; also the timestep is held constant for the two steps. The y- and z-sweeps of hydro are not called unless that dimension is included in the simulation. The same algorithm is used in the `strang_delta` module, but after each call to a physics module, a call to a subroutine is necessary to update the solution. When the `strang_state` driver is used, these calls are made by each physics module. No runtime parameters are defined for either module.

6.1.4 The `formulation` modules

The purposes of this module class are

1. To provide functions usable by physics modules and driver modules to update the solution locally (on a block or on a face of a block) or globally (on all blocks).

2. If needed by the time advancement (driver) module, to provide storage space for the global ΔU array and functions to access it.

The alternative time advancement methods (drivers) are implemented in either the state-vector or delta formulations. There are two corresponding instantiations of the formulation module. In the state-vector instantiation, only the local update functions in item (1) are provided; drivers in the state-vector formulation do not require any other services. The delta instantiation provides both local and global update functions and global ΔU array storage as required by drivers in the delta formulation.

The services provided to delta formulation drivers are a superset of those provided to drivers in the state-vector formulation, and the directory structure is used to express that. The `/formulation/state_form` directory contains the local update subroutines and a version of `formulation_Module` suitable for the state-vector instantiation. `formulation_Module` defines a module in the Fortran90 sense as opposed to the FLASH hierarchy sense. The `formulation/state_form/delta_form` directory contains the global update subroutine and the version of `formulation_Module` required for the delta instantiation.

When `/formulation/state_form` is specified in the `Modules` file, the local update functions and the first `formulation_Module` are built into the executable, as appropriate for drivers in the state-vector formulation; when `/formulation/state_form/delta_form` is specified in the `Modules` file, the local update functions, the global update function, and the second version of `formulation_Module` are used in the executable, as required by drivers in the delta formulation. This use of the FLASH code framework and directory hierarchy allows static allocation of the global ΔU array when needed but saves that memory when not. At the same time, it allows local update functions to be used by both state-vector and delta formulations without duplicating code.

Currently the update functions apply only to the particular variable sets described. The local update functions must be given the (old) conserved variables in the order $\rho_1, \cdots, \rho_{ionmax}$, ρu, ρv, ρw, ρE, and they store in the database $X_1, \cdots, X_{ionmax}$, ρ, P, T, γ, u, v, w, and E. The mapping from the conserved variables to the database variables is not general; it is specific to the variables just listed. Variables other than those specifically listed will not be updated, and their influence on the variables just listed will be ignored. Development of more flexible update routines is underway. However, changes will most likely be internal to the local and global update functions, and the organization of these modules is not expected to change.

6.1.4.1 State-Vector Instantiation

In this subsection the local update functions, named `du_update_block`, `du_update_xface`, `du_update_yface`, and `du_update_zface`, are described. These subroutines accept local arrays of conserved variables and their changes as inputs, compute updated conserved variables, compute auxiliary variables from algebraic relations (with the aid of appropriate equation of state calls), and store the updated variables in the database.

These subroutines accept the block number, a local ΔU, local conserved variables U, the timestep Δt, and a scalar factor c, all as passed arguments. The face update routines also accept an index specifying which grid plane to update. The conserved variables are updated by

$$U^{new} = U^{old} + c\Delta t \Delta U \ . \tag{6.7}$$

The factor c allows an update to an intermediate time between t^n and t^{n+1}, often required by Runge-Kutta time advancement methods; it is intended for use by drivers in the delta formulation through the global update subroutine.

From the updated conserved variables, all variables stored in the database are computed. The density ρ and species mass fractions X_s are obtained from the species densities ρ_s. The velocity components u, v, and w and the total energy per unit mass E are computed from the momenta and total energy per unit volume, respectively, by dividing by ρ. The internal energy ϵ is calculated by subtracting the kinetic energy per unit mass $(u^2 + v^2 + w^2)/2$ from E. The temperature T, pressure P, and ratio of specific heats γ are obtained through a call to the equation of state, for which ρ, X_s, and ϵ are inputs.

Finally, the updated variables are stored in the variable database. The variables stored are X_s, ρ, P, T, γ, u, v, w, and E. Only the interior cells of a block or face are updated; for all guard cells, zeros are stored for all updated variables. None of the calculations described above are executed for the guard cells.

For the state-vector formulation, there are only a few tasks for the Fortran 90 module `formulation_Module`. First, it defines a Fortran logical parameter `delta_formulation` to be `.false.`. This parameter is designed to be accessed by physics modules. When `.false.`, it indicates that each physics module should update the solution; while the local update routines just described are recommended for this purpose, there is no requirement that they be used. Second, `formulation_Module` defines several parameters for sizing arrays and a set of integers (indices) used to access the variable database; these are used by the local update subroutines.

In the state-vector instantiation, `formulation_Module` does not declare the global ΔU array. It does define some functions which are used to access that array, but in this instantiation they do not perform any operations – they are 'stub' functions. The reason for defining them is as follows. If a physics module is written so that either the state-vector or delta formulation can be used, it must include calls to functions which give access to the global ΔU array. When the state-vector formulation is used these calls are not made, but many compilers raise errors when these functions are not defined. By defining them in this instantiation of `formulation_Module`, such errors are avoided. The stub functions are *contained*, in the Fortran 90 sense, in `formulation_Module`. The local update functions are not contained in `formulation_Module`, although they directly access the array sizing parameters and database indices therein.

6.1.4.2 Delta Instantiation

In this section the global update subroutine `du_update` and the version of `formulation_Module` used for the delta formulation are described. The global update routine is a wrapper to the local update subroutine du_update_block. Two arguments, c and Δt, are passed into du_update. For each block, it gets ρ, X_s, u, v, w, and E from the database; computes the (old) conserved variables from these; gets the ΔU for the block from the global ΔU array; and calls du_update_block. Recall that du_update_block computes the updated variables and stores them in the database.

For the delta instantiation, formulation_Module defines the same array-sizing parameters and database indices as in the state-vector instantiation. However, it defines the parameter delta_formulation to be .true., indicating to the physics modules that their contributions should be added to the global ΔU array. The delta instantiation of formulation_Module statically allocates the global ΔU array and defines several functions to manipulate it. Each element of the global ΔU array is set to zero by du_zero. A physics module can add its local ΔU for a block to the global array by calling du_block_to_global; the subroutines du_xface_to_global, du_yface_to_global and du_zface_to_global do the same for faces (slices) of a block. These subroutines are contained in formulation_Module and are the actual, working versions of the stub functions defined in the state-vector instantiation.

The global ΔU array is a public, module-scope variable in the Fortran 90 sense. The du_update subroutine is not contained in formulation_Module but can access the array-sizing parameters and database indices in the module. It can also access the global ΔU array directly and is the only subroutine not contained in formulation_Module allowed to do so.

6.2 Simulation services

6.2.1 Runtime parameters

The driver module provides a Fortran 90 module called runtime_parameters. The routines in this module maintain 'parameter contexts,' essentially small databases of runtime parameters. Contexts can be created and destroyed, and runtime parameters can be added to them, have their values modified, and be queried as to their value or data type. These features allow a program to maintain several contexts for different code modules without having to declare and share the parameters explicitly. User-written subroutines (*e.g.*, for initialization) should use the routines in this module to access the values of any runtime parameters they require.

An example of the application of this module is to use the read_parameters() routine (separately supplied) to parse a text-format input file containing parameter settings. The calling program declares a context, adds parameters to it, and then calls read_parameters() to parse the input file. Finally, the context is queried to obtain the input values. Such a program might look like the following code fragment.

```
program test
use runtime_parameters
type (parm_context_type) :: context
real  :: x_init
...
call create_parm_context (context)
call add_parm_to_context (context, "x_init", 4.)
...
call read_parameters ("input.par", context)
call get_parm_from_context (context, "x_init", x_init)
...
end
```

Parameter names supplied as arguments to the routines are stored or compared in a case-insensitive fashion. Thus N_x and n_x refer to the same parameter, and

```
integer n_x
call add_parm_to_context (context, "N_x", 32)
call get_parm_from_context (context, "n_X", n_x)
write (*,*) n_x
call set_parm_in_context (context, "n_x", 64)
call get_parm_from_context (context, "N_X", n_x)
write (*,*) n_x
```

prints

```
32
64
```

The following routines, data types, and public constants are provided by this module. Note that the main FLASH initialization routine (`init_flash()`) and the initialization code created by `setup` already handle the creation of the database and the parsing of the parameter file, so users will mainly be interested in querying the database for parameter values.

- `parm_context_type`

  Data type for contexts.

- `global_parm_context`

  A parameter context available to all program units which use the `runtime_parameters` module. This is made available only for programs that share the rest of their data among routines via included common blocks. Programs which use modules should declare their own contexts within their modules.

- `parm_{real,int,str,log,invalid}`

  Constants returned by get_parm_type_from_context().

- `create_parm_context` (c)

  Create context c.

- `destroy_parm_context` (c)

  Destroy context c, freeing up the memory occupied by its database.

- `add_parm_to_context` (c,p,v)

  Add a parameter named p to context c. p is a character string naming the parameter, and v is the default or initial value to assign to the parameter. v can be of type real, integer, string, or logical. The type of v sets the type of the parameter; subsequent sets or gets of the parameter must be of the same type, or an error message will be printed.

- `set_parm_in_context` (c,p,v)

  Set the value of parameter p in context c equal to v. p is a character string naming the parameter, which must already have been created by `add_parm_to_context` (else an error message is printed). The type of v must match the type of the initial value used to create the parameter, else an error message is printed.

- `get_parm_from_context` (c,p,v)

  Query context c for the value of parameter p and return this value in the variable v. Parameter p must already exist, and the type of v must match the type of the initial value used to create the parameter, else an error message is printed.

- `get_parm_type_from_context` (c, p, t)

  Query context c for the data type of parameter p. The result is returned in t, which must be of type integer. Possible return values are `parm_real`, `parm_int`, `parm_str`, `parm_log`, and `parm_invalid`. `parm_invalid` is returned if the named parameter does not exist.

- `list_parm_context` (c, l)

  Print (to I/O unit l) the names and values of all parameters associated with context c.

- `bcast_parm_context` (c, p, r)

  Broadcast the parameter names and values from a specified context c to all processors. p is the calling processor's rank, and r is the rank of the root processor (the one doing the broadcasting).

6.2.2 Physical constants

The driver supplies a Fortran 90 module called `physical_constants`, which maintains a centralized database of physical constants. The database can be queried by string name and optionally converted to any chosen system of units. The default system of units is CGS. 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 and to update the constants used by the code as improved measurements become available.

For example, a program using this module might obtain the value of Newton's gravitational constant G in units of $\mathrm{Mpc}^3\ \mathrm{Gyr}^{-2}\ \mathrm{M}_\odot^{-1}$ by calling

```
call get_constant_from_db ("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 the 1998 Review of Particle Properties, Eur. Phys. J. C 3, 1 (1998), which in turn takes most of its values from Cohen, E. R. and Taylor, B. N., Rev. Mod. Phys. 59, 1121 (1987). The following routines are supplied by this module.

- `get_constant_from_db` $(n, v$[, *units*])

  Return the value of the physical constant named n in the variable v. Optional unit specifications are used to convert the result. If the constant name or one or more unit names aren't recognized, a value of 0 is returned.

- `add_constant_to_db` $(n, v, len, time, mass, temp, chg)$

  Add a physical constant to the database. n is the name to assign, and v is the value in CGS units. *len*, *time*, *mass*, *temp*, and *chg* are the exponents of the various base units used in defining the unit scaling of the constant. For example, a constant with units (in CGS) of $\mathrm{cm}^3\ \mathrm{s}^{-2}\ \mathrm{g}^{-1}$ would have *len*=3, *time*=-2, *mass*=-1, *temp*=0, and *chg*=0.

- `add_unit_to_db` (t, n, v)

  Add a unit of measurement to the database. t is the type of unit ("length," "time," "mass," "charge," "temperature"), n is the name of the unit, and v is its value in terms of the corresponding CGS unit. Compound units are not supported, but they can be created as physical constants.

- `init_constants_db ()`

  Initialize the constants and units databases. Can be called by the user program, but doesn't have to be, since it is automatically called when needed (*i.e.* if a "get" or "add" is called before initialization).

- `list_constants_db` (*lun*)

  List the constants and units databases to the specified logical I/O unit.

- `destroy_constants_db ()`

  Deallocate the memory used by the constants and units databases. Another initialization call will then be required before they can be accessed again.

Chapter 7

FLASH I/O modules and output formats

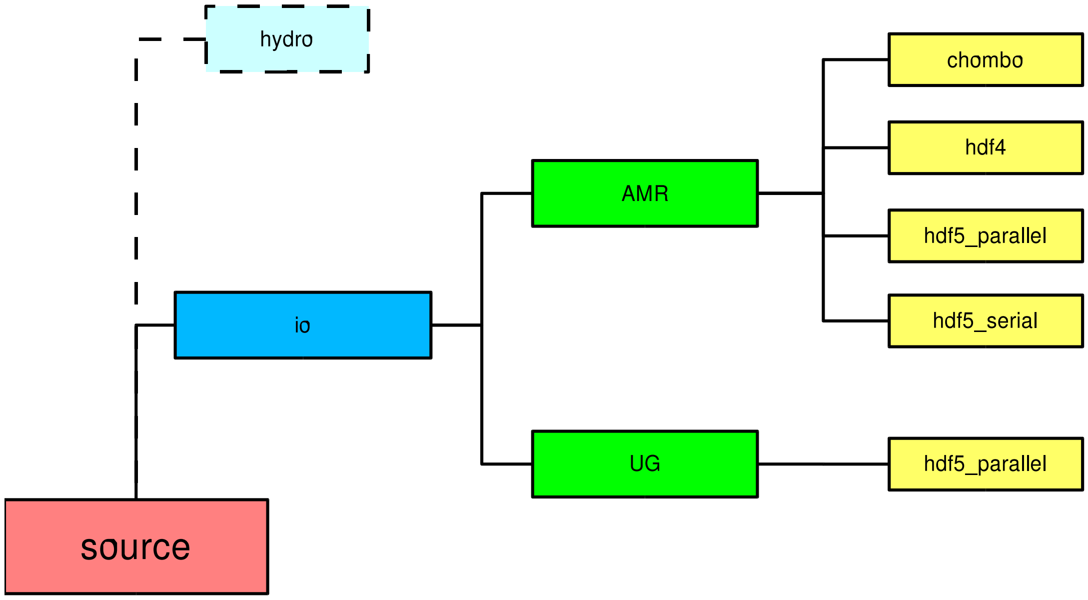


Figure 7.1: The io module directory.

Currently FLASH can store simulation data in two basic output formats: Hierarchical Data Format (HDF) (sometimes called HDF 4), and HDF5. In general, these formats are not compatible, but some tools for translating from one format to the other exist. These formats control how the binary data is stored on disk, how to address it, and what to do about different data storage on different platforms. The mapping of FLASH data-structures to records in these files is controlled by the FLASH I/O modules. These file formats have different strengths and weaknesses, and the data layout is different for each file type.

Different techniques can be used to write the data to disk: move all the data to a single processor for output; have each processor write to a separate file; or parallel access to a single file. In general, parallel access to a single file will provide the best performance. 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 best solution.

The `hdf4` I/O module uses the HDF version 4 format. This format provides an 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.

HDF is limited to files $<$ 2 GB in size. Furthermore, the official release of HDF does not support parallel I/O. To address these limitations, HDF5 was released. HDF5 is supported on a large variety of platforms and offers the same functionality as HDF, with the addition of large file support and parallel I/O via MPI-I/O. Information about the different versions of HDF can be found at `http://hdf.ncsa.uiuc.edu`. This section assumes that you already have the necessary HDF libraries installed on your machine.

As of FLASH 2.2, the default I/O format is HDF5; the default I/O module is `hdf5_serial` (see Table 7.1). The I/O modules in FLASH have two responsibilities—generating and restarting from checkpoint files, and generating plot files. A checkpoint file contains all the information needed to restart the simulation. The data is stored at the same precision (8-byte reals) as it is carried in the code and includes all of the variables. A plotfile contains all the information needed to interpret the tree structure maintained by FLASH and includes a user-defined subset of the variables. Furthermore, the data may be stored at reduced precision to conserve space.

The type of output you create will depend on the type of machine on which you are running, the size of the resulting dataset, and what you plan to do with the datafiles once created. Table 7.1 summarizes the different modules which come with FLASH 2.3.

Table 7.1: I/O modules available in FLASH.

Module name	Description
`amr/hdf4`	Hierarchical Data Format (HDF) 4 output. A single HDF file is created by the master processor and all data are moved to this processor via explicit MPI sends and receives before writing to the file.
`amr/hdf5_serial`	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.
`amr/hdf5_parallel`	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 MPI-IO layer in HDF5.
`null`	Don't write any checkpoint files or plotfiles.

It is strongly recommended that you use one of the HDF5 output formats. These are currently the best performing I/O modules in FLASH. Furthermore, support for HDF5 exists for just about every platform you are likely to encounter.

7.1 General parameters

There are several parameters that control the frequency of output, the type of output, and the name of the output files. These parameters are the same for each module, although not every module is required to implement every parameters. Some of these parameters are used in the top level I/O routines (`initout.F90`, `output.F90`, and `finalout.F90`) to determine when to output, while others are used to determine the resulting filename. Table 7.2 gives a description of the I/O parameters.

Table 7.2: General I/O parameters.

Parameter	Type	Default value	Description
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.
basenm	STRING	"chkpnt"	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 or checkpoint file, the file format, and a 4-digit file number. See Sec. 7.1.1 for a description of how FLASH output files are named.
cpnumber	INTEGER	10000	The number of the current checkpoint file. This number is appended to the end of the basename when creating the filename. When restarting a simulation, this indicates which checkpoint file to use.
ptnumber	INTEGER	1	The number of the current plotfile. This number is appended to the end of the basename when creating the filename.
restart	BOOLEAN	.false.	A logical variable indicating whether the simulation is restarting from a checkpoint file (.true.) or starting from scratch (.false.).
nrstrt	INTEGER	10000	The number of timesteps desired between subsequent checkpoint files.
trstrt	REAL	1.	The amount of simulation time desired between subsequent checkpoint files.
tplot	REAL	1.	The amount of simulation time desired between subsequent plotfiles.
zrstrt	REAL	1.E99	The desired redshift interval between checkpoint files.

Table 7.2: FLASH I/O parameters (continued).

Parameter	Type	Default value	Description
zplot	REAL	1.E99	The desired redshift interval between plotfiles.
corners	BOOLEAN	.false.	A logical variable indicating whether to interpolate the data to cell corners before outputting. This only applies to plotfiles.
plot_var_1, ... plot_var_12	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.
memory_stat_freq	INTEGER	100000	The number of timesteps to elapse between memory statistic dumps to the log file (flash.log)
wr_integrals_freq	INTEGER	1	The number of timesteps to elapse between outputs to the scalar/integral data file (flash.dat)

7.1.1 Output file names

FLASH constructs the output filenames based on the user-supplied basename 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:

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

where hdf or hdf5 is picked depending on the I/O module used and the number at the end of the filename is the current cpnumber. The general plotfile filename is:

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

where hdf or hdf5 is picked depending on the I/O module used, crn and cnt indicate data stored at the cell corners or centers respectively, and the number at the end of the filename is the current ptnumber.

7.2 Restarting a simulation

In a typical production run, your simulation can be interrupted for a number of reasons—*e.g.* machine crashes, the present queue window closes, the machine runs out of disk space, or (*gasp*) a bug in FLASH. Once the problem is fixed, you do not want to start over from the beginning of the simulation but would rather pick up where you left off.

There are many ways to get FLASH to produce a restart file:

- Amount of simulation time elapsed

  The trstrt runtime parameter specifies the number of seconds in simulation time between restart files. If the simulation is being controlled by redshift rather than time (zinitial > 0), the zrstrt parameter can be used to force checkpoints at a fixed interval in redshift.

- Number of timesteps elapsed

  The `nrstrt` runtime parameter specifies the number of timesteps between restart dumps.

- End of a simulation

  When the number of timesteps equals `nend` or the simulation time equals `tmax`, a restart file is produced, and the simulation ends.

- Wall clock time elapsed

  The `wall_clock_checkpoint` gives the number of seconds in wall clock time between checkpoint files. The counter is started when the simulation begins, and checkpoint files will be produced at multiples of this time.

- The `.dump_restart` mechanism

  Creating a file named `.dump_restart` in the output directory that the master processor is writing to will cause FLASH to output a checkpoint file and stop the simulation. This is useful if you know a machine is going down or a queue window is about to end and you want to produce one last checkpoint so you don't lose all of the evolution time since the last one was written.

These different methods can be combined without problems. Each counter (number of timesteps between last checkpoint, amount of simulation time single last checkpoint, and the amount of wall clock time elapsed since the last checkpoint) is independent of the others.

FLASH is capable of restarting from any of the checkpoint files it produces. You will want to make sure the file you wish to restart from is valid (*i.e.* the code did not stop while outputting). To tell FLASH to restart, set the `restart` runtime parameter to `.true.` in your `flash.par`. You will also want to set `cpnumber` to the number of the file from which you wish to restart. Finally, if you are producing plotfiles, you will want to set `ptnumber` to the number of the next plotfile you want FLASH to output. Sometimes several plotfiles may be produced after the last valid checkpoint file, so resetting `ptnumber` to the first plotfile produced after the checkpoint from which you are restarting will ensure that there are no gaps in your output. The `restart` script in `tools/scripts/jobs/` will automatically modify your `flash.par` to pick up where a simulation left off by examining the logfile. To use it, make sure that it is set as executable and that it is in your path and type

```
% restart -logfile my_simulation.log
```

where `my_simulation.log` is the name of the FLASH logfile (usually `flash.log`, unless renamed with the `logfile` runtime parameter).

7.3 Output formats

7.3.1 HDF

The HDF module writes the data to disk using the HDF 4.x library. This module should be supported with HDF 4.1r2 or later. A single file containing the data on all processors is created, if the total size of the dataset is < 2 GB. If there is more than 2 GB of data to be written, multiple files are created to store the data. The number of files used to store the dataset is contained in the *number of files* record. Each file contains a subset of the blocks (stored in the *local blocks* record) out of the total number of blocks in the simulation. The blocks are divided along processor boundaries.

The HDF module performs serial I/O — each processor's data are moved to the master processor to be written to disk. This has the advantage of producing a single file for the entire simulation but is less efficient than if each processor wrote to disk directly. The plotfiles produced with the `HDF` module contain single precision data (Note: versions of FLASH before 2.1 produced double-precision HDF plotfiles). Support for corner data is available with this module. Support for particle I/O is not yet available with this module.

Machine Compatibility

HDF has been tested successfully on most machines, with the exception of ASCI Red. The HDF library will properly handle different byte orderings across platforms. The IDL tools provided with FLASH will read the FLASH HDF data.

Data Format

Table 7.3 summarizes the records stored in a FLASH HDF file. The format of the plotfiles and checkpoint files are the same, with the only difference being the number of unknowns stored. The records contained in an HDF file can be found via the `hdp` command that is part of the HDF distribution. The syntax is `hdp dumpsds -h filename`. The contents of a record can be output by `hdp dumpsds -i N filename`, where `N` is the index of the record.

As described above, HDF cannot produce files > 2 GB in size. This is overcome in the FLASH HDF module by splitting the dataset into multiple files when it would otherwise produce a file larger than 2 GB in size.

When reading a single unknown from a FLASH HDF file, you will suffer performance penalties; there will be many non-unit-stride accesses on the first dimension, since all the variables for a single zone are stored contiguously in the file.

Table 7.3: FLASH HDF file format.

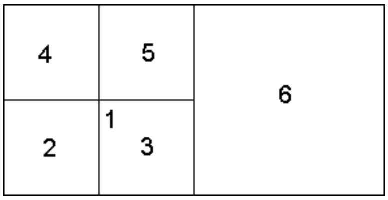
Record label	Description of the record
file creation time	`character*40 file_creation_time` The time and date that the file was created.
FLASH version	`character*20 flash_version` The complete version number of the FLASH distribution you are running. The format is FLASH 2.3.YYYYMMDD where YYYYMMDD is the date of the release. This data is contained in the file `RELEASE` and the version number is obtained from the `flash_release` function.
FLASH build date	`character*80 flash_build_date` The date and time that the FLASH executable was compiled. This is generated by a subroutine that is created at compile time by the Makefile.
FLASH build directory	`character*80 flash_build_directory` The complete path to the FLASH root directory of the source tree used when compiling the FLASH executable. This is generated by a subroutine that is created at compile time by the Makefile.
FLASH build machine	`character*80 flash_build_machine` The name of the machine (and anything else returned from `uname -a`) that FLASH was compiled on. This is generated by a subroutine that is created at compile time by the Makefile.
FLASH setup call	`character*80 flash_setup_call` The complete syntax of the `setup` command used when creating the current FLASH executable. This is generated by a subroutine that is created at compile time by the Makefile.

Table 7.3: HDF 4 format (continued).

Record label	Description of the record
run comment	`character*80 run_comment` The run_comment that was defined for the present simulation. This is a runtime parameter that is useful for annotating a simulation.
total blocks	`integer tot_blocks` The total number of blocks in the simulation. Note: the number of blocks contained in this file may be less than the total number of blocks if the output is spread across multiple files (see 'number of files' and 'local blocks')
time	`real time` The simulation time at file output.
timestep	`real dt` The current timestep.
number of steps	`integer nsteps` The number of timesteps from the start of the calculation.
number of blocks per zone	`real nblocks_per_zone(3)` This record is misnamed and actually contains the number of zones per block in each direction: `nblocks_per_zone(1)` — x-direction `nblocks_per_zone(2)` — y-direction `nblocks_per_zone(3)` — z-direction
number of files	`integer num_files` The number of files that the dataset comprises. Because the file size cannot be larger than 2 GB, the data is split into multiple files if necessary, with each containing roughly the same number of blocks.
local blocks	`integer local_blocks` The number of blocks in this file. If there are multiple files, this number will be less than 'total blocks'.
unknown names	`character*4 unk_names(nvar)` Four-character names corresponding to the first index of the `unk` array. They serve to identify the variables stored in the 'unknowns' record.
refine level	`integer lrefine(local_blocks)` The refinement level for each block.

Table 7.3: HDF 4 format (continued).

Record label	Description of the record
node type	`integer nodetype(local_blocks)` 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 nodes contain the data that is to be used for plotting purposes.
gid	`integer gid(nfaces+1+nchild,local_blocks)` 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 number of its parent and children. The first `nfaces` elements point to the neighbors (at the same level of refinement). The faces are numbered from minimum to maximum coordinate with x first, followed by y, then z. A -1 indicates that there is no neighbor at the same level of refinement. A number less than or equal to -20 indicates that you are on the physical boundary of the domain. If the neighbor points to the current block, it means that there are periodic boundary conditions. The next element points to the parent of the current block, and the last `nchild` elements point to the children of the current block. Below is an example of a simple domain with all boundaries set to -20.



Looking at block number 5 (2-d):

```
gid(1,block_no) = 4
gid(2,block_no) = -1
gid(3,block_no) = 3
gid(4,block_no) = -20

gid(5,block_no) = 1   (the parent)

gid(6,block_no) = -1  (the children)
gid(7,block_no) = -1
gid(8,block_no) = -1
gid(9,block_no) = -1
```

Looking at block number 1:

```
gid(1,block_no) = -20
gid(2,block_no) = 6
gid(3,block_no) = -20
gid(4,block_no) = -20

gid(5,block_no) = -1

gid(6,block_no) = 2
gid(7,block_no) = 3
gid(8,block_no) = 4
gid(9,block_no) = 5
```

Table 7.3: HDF 4 format (continued).

Record label	Description of the record
coordinates	`real coord(ndim,local_blocks)` This array stores the coordinates of the center of the block. `coord(1,block_no)` = x-coordinate `coord(2,block_no)` = y-coordinate `coord(3,block_no)` = z-coordinate
block size	`real size(ndim,local_blocks)` This array stores the dimensions of the current block. `size(1,block_no)` = x-size `size(2,block_no)` = y-size `size(3,block_no)` = z-size
bounding box minimum	`real bnd_box_min(ndim,local_blocks)` This array stores the coordinate of the minimum block edge in each direction. `bnd_box_min(1,block_no)` = minimum x-edge `bnd_box_min(2,block_no)` = minimum y-edge `bnd_box_min(3,block_no)` = minimum z-edge
bounding box maximum	`real bnd_box_max(ndim,local_blocks)` As above, but the maximum edge value in each direction.
processor number	`integer proc_num(local_blocks)` The processor number on which each block was stored. This is not used by FLASH but is useful for debugging purposes to look at the domain decomposition.
unknowns	`real unk(nvar,nx,ny,nz,local_blocks)` `nx` = no. of zones/block in x `ny` = no. of zones/block in y `nz` = no. of zones/block in z This array holds the unknowns. The variables corresponding to the first argument are listed in the 'unknown names' record. Note that for a plot file with `CORNERS=.true.` in the parameter file, the information is interpolated to the zone corners before being stored. This is useful for certain plotting packages.

7.3.2 HDF5

There are two major HDF5 modules – the serial and parallel versions. The format of the output files produced by these modules is identical; only the method by which they are written differs. It is possible to create a checkpoint file with the parallel routines and restart FLASH from that file using the serial routines. In each module, the plot data are written out in single precision to conserve space. These modules require HDF5 1.4.0 or later. At the time of this writing, the current version of HDF5 is 1.4.4.

7.3.2.1 Machine Compatibility

The HDF5 modules have been tested successfully on the ASCI platforms and on a Linux cluster. Performance varies widely across the platforms, but the parallel version is usually faster than the serial version. Experience on performing parallel I/O 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 I/O module with an NFS mounted filesystem. In this case, all of the information will still have to pass through the node off of which the disk is hanging, resulting in contention, and it is recommended that the serial version of the HDF5 module be used. A shared object library provided with FLASH provides IDL with the ability to read the FLASH files.

7.3.2.2 Data Format

The data format FLASH uses for HDF5 output files is similar to that of the HDF files, but there are a few differences that make a record-to-record translation impossible. These changes were made to maximize performance. Instead of putting all the unknowns in a single HDF record, `unk(nvar,nx,ny,nz,tot_blocks)`, as in the HDF file, each variable is stored in its own record, labeled by the four-character variable name. A number of smaller records (*time*, *timestep*, *number of blocks*, ...) are stored in a single structure in the HDF5 file to reduce the number of writes required. Finally, the two bounding box records in the HDF file are merged into a single record in the HDF5 file. This allows for easier access to a single variable when reading from the file. The HDF5 format is summarized in Table 7.4. Note that particle I/O is available only with the HDF5 modules at this time.

Table 7.4: FLASH HDF5 file format.

Record label	Description of the record
file creation time	`character*40 file_creation_time` The time and date that the file was created.
file format version	`integer file_format_version` An integer given the version number of the HDF5 file format. This is incremented anytime changes are made to the layout of the file.
FLASH version	`character*80 flash_version` The version of FLASH used for the current simulation. This is returned by `flash_release`, using the `RELEASE` function.
FLASH build date	`character*80 flash_build_date` The date and time that the FLASH executable was compiled. This is generated by a subroutine that is created at compile time by the Makefile.
FLASH build directory	`character*80 flash_build_directory` The complete path to the FLASH root directory of the source tree used when compiling the FLASH executable. This is generated by a subroutine that is created at compile time by the Makefile.
FLASH build machine	`character*80 flash_build_machine` The name of the machine (and anything else returned from `uname -a`) on which FLASH was compiled. This is generated by a subroutine that is created at compile time by the Makefile.

Table 7.4: HDF5 format (continued).

Record label	Description of the record
FLASH setup call	`character*80 flash_setup_call` The complete syntax of the `setup` command used when creating the current FLASH executable. This is generated by a subroutine that is created at compile time by the Makefile.
run comment	`character*80 run_comment` The run_comment that was defined for the present simulation. This is a runtime parameter that is useful for annotating a simulation.

simulation parameters	Several records are packed into a C structure
	
	`typedef struct sim_params_t {`
	`  int total_blocks;`
	`  int nsteps;`
	`  int nxb;`
	`  int nyb;`
	`  int nzb;`
	`  double time;`
	`  double timestep;`
	`  double redshift;`
	`} sim_params_t;`
	
	`sim_params_t sim_params;`
	
	`sim_params.total_blocks`: total number of blocks.
	`sim_params.nsteps`: the total number of steps to this point.
	`sim_params.nxb`: number of zones / block in the x-direction.
	`sim_params.nyb`: number of zones / block in the y-direction.
	`sim_params.nzb`: number of zones / block in the z-direction.
	`sim_params.time`: the current simulation time.
	`sim_params.timestep`: the current timestep.
	`sim_params.redshift`: the current redshift.

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(tot_blocks)` This array stores the refinement level for each block.
node type	`integer nodetype(tot_blocks)` 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.

Table 7.4: HDF5 format (continued).

Record label	Description of the record
gid	`integer gid(nfaces+1+nchild,tot_blocks)` 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. See the description in the HDF 4 table for full details.
coordinates	`real coord(ndim,tot_blocks)` This array stores the coordinates of the center of the block. `coord(1,block_no)` = x-coordinate `coord(2,block_no)` = y-coordinate `coord(3,block_no)` = z-coordinate
block size	`real size(ndim,tot_blocks)` This array stores the dimensions of the current block. `size(1,block_no)` = x size `size(2,block_no)` = y size `size(3,block_no)` = z size
bounding box	`real bnd_box(2,ndim,tot_blocks)` This array stores the minimum (`bnd_box(1,:,:)`) and maximum (`bnd_box(2,:,:)`) coordinate of a block in each spatial direction.
variable	`real unk(nx,ny,nz,tot_blocks)` `nx` = number of zones/block in x `ny` = number of zones/block in y `nz` = number of zones/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 zone corners and stored.

particle tracers	Particle data (Only in checkpoint files). These are stored as an array of structures defined via
	
	`typedef struct particle {`
	`  int intProperty[NUMINTPROPS];`
	`  double realProperty[NUMREALPROPS];`
	`} particle_type;`
	
	`particle_type particles[];`
	
	`NUMINTPROPS` and `NUMREALPROPS` are the number of integer and double-precision properties defined for the particles, respectively. The Fortran/C structure is mapped directly onto an HDF5 structure datatype with fields whose names are the string names of the different properties. Currently the number of properties and their order in the structure must be the same in the checkpoint file and the compiled code when restarting from a checkpoint file. This restriction may be relaxed in the next release.

7.4 Working with output files

The HDF 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 datatype, rank, and dimensions. The HDF formats also avoid difficulties associated with different platforms storing numbers differently (big endian vs. little endian). `IDL` routines for reading the FLASH HDF and HDF5 formats are provided in `tools/fidlr2/`. These can be used interactively though the `IDL` command line (see Sec. 20.5).

7.5 User-defined variables

`user_var` is called just before output files are written. This routine allows the calculation of user-defined variables, which can then be written out for plotting or analysis. These user-defined variables must be defined in the FLASH code (see Configuration layer, Chapter 5).

The supplied `user_var`, in `source/io/user_var.F90`, looks for two variables, `vrtz` and `cond` — the z-component of vorticity, and a (thermal) conductivity. If either of these are present, the values are calculated and stored in the FLASH data structures. If the output file to be written is a checkpoint file, then both of these files are written out to disk. If the output file is a plot file, these variables will be written if the `plot_var` variables are set accordingly (see the beginning of this chapter.) The supplied `user_var` can be used as a template for calculating other user-defined values. The `cond` variable is supplied as a convenience; it could fairly easily be calculated in a post-processing step. However, calculating `vrtz` would be more complicated, as its calculation requires a numerical derivative, meaning that guard cell information (not stored in the FLASH file structures) is required. Note that these variables are stored in the FLASH data structures and thus require memory, even though they are not used for computation.

Chapter 8

Mesh module

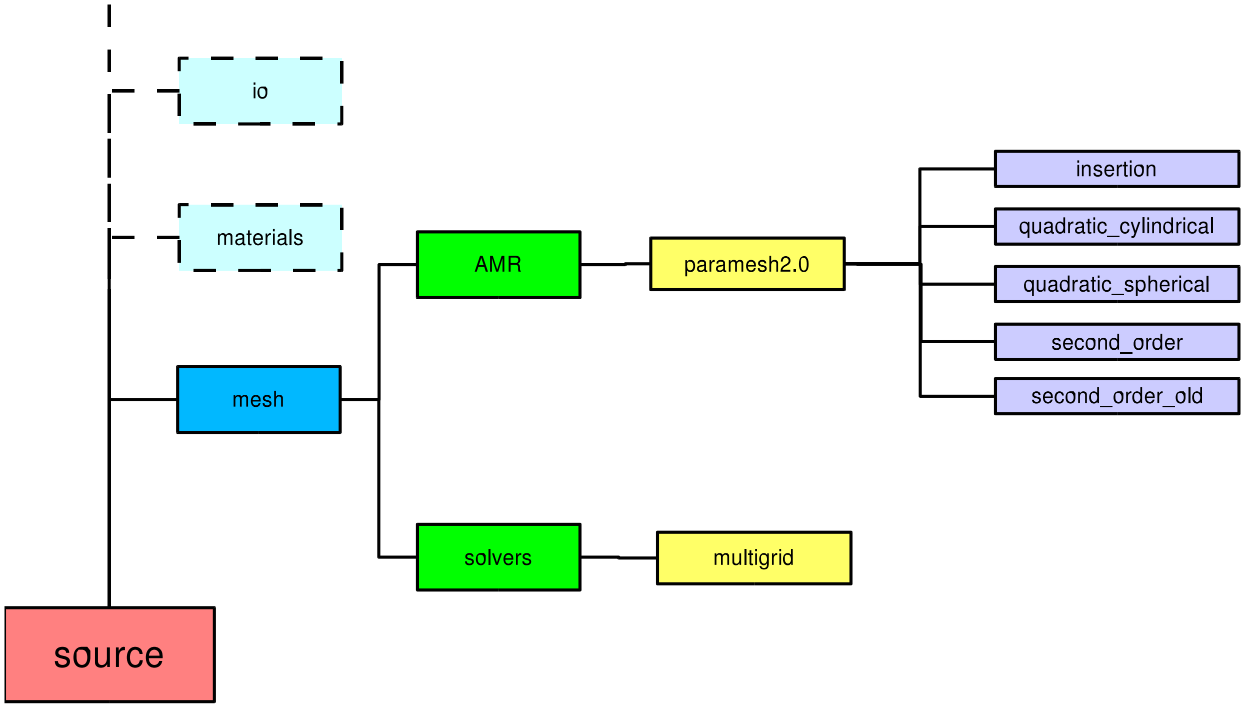


Figure 8.1: The mesh module directory.

8.1 Introduction

The mesh module is responsible for maintaining the grid used to discretize the simulation. Most of the interprocessor communication is handled by the mesh module, removing this hassle from the user. Each mesh module divides the computational domain into one or more sub-domains or blocks, which contains a number of computational zones (nxb in the x-direction, nyb in the y-direction, and nzb in the z-direction). The mesh module also manages the coordinate information for each block.

Flash provides the option of using either an adaptive or a single-level mesh. We describe both types of meshes here. For the most part, the application programmers do not have to know the details of the mesh (even whether it is single-level or adaptive), only that they are given a block of data from the database along with the coordinates of each zone, and they can do whatever operations they see fit on it before returning it to the database. A perimeter of width nguard guard cells surrounds each block of data, providing the data from the neighboring blocks or boundary conditions.

At present, the memory for the mesh is allocated statically. The total number of blocks a processor can manage is set to MAXBLOCKS, which can be overridden at setup time with the -maxblocks=# argument. The amount of memory consumed by the data portion of the code is $\mathtt{nvar} \times (2 * \mathtt{nguard} + \mathtt{nxb}) \times (2 * \mathtt{nguard} + \mathtt{nyb}) \times (2 * \mathtt{nguard} + \mathtt{nzb}) \times \mathtt{MAXBLOCKS}$. Note that this is not the total amount of memory used by the code, since fluxes, temporary variables, and coordinate information also consume a large amount of memory.

The geometry of both types of meshes are set in a consistent way (although not all geometries are supported by the two meshes yet). At present, only 1-d spherical, 2-d cylindrical, and Cartesian geometries are supported. Initial support for 2-d spherical coordinates is present in the code, but not all the modules are aware of this geometry. The boundary conditions are also set in a consistent manner.

8.2 Specifying the computational domain

The size of the computational domain in physical units is specified at runtime through the (xmin, xmax), (ymin, ymax), and (zmin, zmax) runtime parameters. When working with angular coordinates (see below), the extrema are specified in units of π.

Regardless of whether the mesh is single-level or adaptive, it is decomposed into Nblockx × Nblocky × Nblockz blocks initially. This is useful if your domain has a non-unit aspect ratio but you still want square (or cubical) zones. These blocks are then either further refined (if working with an adaptive mesh) or left as is (when working with a single-level mesh). For an adaptive mesh, parameters are interpreted by divide_domain, which, at the moment, does this decomposition on one processor, so the number of blocks created in this fashion cannot exceed MAXBLOCKS. For a single-level mesh, these parameters are used by a series of routines to create the grid. This is done in parallel, so that MAXBLOCKS does not limit the initial grid.

8.3 Mesh geometry

Work has begun on expanding the number of geometries that the different FLASH modules understand. For the most part, this requires that gradients and divergences have the appropriate area factors and that the volume of a cell is computed properly for a given geometry. The mesh module must initialize the grid properly for each geometry and any AMR operations (such as restriction, prolongation, and flux-averaging) must respect the geometry. The hydrodynamic methods in FLASH are finite-volume methods, so the interpolation must be conservative in the geometry the we are running with.

The geometry is set at runtime through the geometry runtime parameter. This can be set to one of "cartesian", "spherical", "cylindrical", or "polar". Together with the dimensionality of the problem, this serves to completely specify what the different coordinate axes are. This is summarized in Table 8.1. Again we note that not all modules or meshes support all the geometries.

8.3.1 dBase support

The FLASH dBase module provides a property, "MeshGeometry", which can be compared to a dBase parameter, {CARTESIAN, SPHERICAL, CYLINDRICAL, POLAR} to determine which of the supported geometries we are using. A module writer can use this to branch based on the geometry type—see Fig. 8.2. Furthermore, this provides a mechanism for a module to determine at runtime whether it supports the current geometry, and if not, to abort.

Table 8.1: Different geometry types

name	dimensionality	axisymmetric	x-coord	y-coord	x-coord
cartesian	1	n	x		
cartesian	2	n	x	y	
cartesian	3	n	x	y	z
cylindrical	2	y	r	z	
cylindrical	3	n	r	z	ϕ
spherical	1	y	r		
spherical	2	y	r	θ	
spherical	3	n	r	θ	ϕ
polar	1	y	r		
polar	2	n	r	ϕ	

Coordinate information for the mesh can be determined via the `dBaseGetCoords` function, as described in Sec. 5.1.2.2. This routine can provide the coordinates of the zone at the left edge, right edge, or center or the width of the zone.

The volume of a zone can be obtained via the `dBaseGetCellVolume` function. The volume of a zone is dependent on the geometry, and is computed as below.

Cartesian

1-d	Δx
2-d	$\Delta x \Delta y$
3-d	$\Delta x \Delta y \Delta z$

Since the 1- and 2-d Cartesian geometries are not axisymmetric, we cannot define a true volume for these zones, so the length and area are returned respectively.

cylindrical

1-d	N/A
2-d	$\pi(r_r^2 - r_l^2)\Delta z$
3-d	$(r_r^2 - r_l^2)\Delta z \Delta\phi$

1-d cylindrical is not supported. In 2-d cylindrical coordinates, the domain is axisymmetric, so we integrate over the ϕ coordinate.

spherical

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	$\frac{1}{3}(r_r^3 - r_l^3)(cos(\theta_l) - cos(\theta_r))\Delta\phi$

In spherical coordinates, we can compute a true volume for all dimensions.

polar

1-d	$\pi(r_r^2 - r_l^2)$
2-d	$\frac{1}{2}\pi(r_r^2 - r_l^2)\Delta\phi$
3-d	N/A

In polar coordinates, the volumes are actually areas, since the domain is always a disk.

```
use dBase, ONLY: CARTESIAN, CYLINDRICAL, SPHERICAL, POLAR

integer, save :: meshGeom

meshGeom = dBasePropertyInteger("MeshGeometry")

select case (meshGeom)

case (CARTESIAN)

! do Cartesian stuff here ...

case (SPHERICAL)

! do spherical stuff here ...

case (CYLINDRICAL)

! do cylindrical stuff here ...

case (POLAR)

! do polar stuff here ...

end select
```

Figure 8.2: Branching based on geometry type

Table 8.2: Hydrodynamical boundary conditions supported by FLASH.

*xx*_boundary_type	Description
`periodic`	Periodic (‘wrap-around’)
`reflect`	Non-penetrating boundaries; zero-gradient with transverse velocity reflected
`outflow`	Zero-gradient boundary conditions; allows shocks to leave the domain
`hydrostatic`	Supports the fluid ‘above’ against gravity
`user`	User-defined

8.4 Boundary conditions

Boundary conditions in FLASH are handled by filling of guard cells by calculation, rather than by getting values from neighboring blocks. FLASH currently has built-in support for the boundary conditions listed in Table 8.2. The boundaries are selected by setting variables such as *e.g.*, `xl_boundary_type` (for the ‘left’ $X-$boundary) in `flash.par`.

The boundary conditions are implemented in two files, `tot_bnd` and `user_bnd`. Both of these files are in the particular mesh directories, since they require access to mesh-specific data structures. `tot_bnd` implements the FLASH-defined boundary conditions. `user_bnd` is a template for the definition of user-defined boundary conditions. If a boundary is set to `user`, the routine in `user_bnd` will be called to fill that boundary’s guard cells.

To create custom boundary conditions for your problem, you need only modify `user_bnd.F90`. It is best to put a copy of it in your problem setups directory, where it will automatically be linked into the `object/` directory. This file is separated into 6 sections, for $\pm x$, $\pm y$, $\pm z$. Templates exist for each section giving the loop over the appropriate guard cells for that boundary. The data for the current block is passed through the argument list as `solnData` and can be filled according to whatever boundary condition the problem requires. An example of an inflow boundary is provided in the `windtunnel` setup.

8.5 Adaptive mesh

8.5.1 Introduction

We use a package known as PARAMESH (MacNeice *et al.* 1999) for the parallelization and adaptive mesh refinement (AMR) portion of FLASH. PARAMESH consists of a suite of subroutines which handle refinement/derefinement, distribution of work to processors, guard cell filling, and flux conservation. In this section we briefly describe this package and the ways in which it has been modified for use with FLASH.

8.5.2 Algorithm

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 an entirely 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 (by default, PARAMESH uses the density and pressure). 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

$$E_i = \frac{\mid u_{i+1} - 2u_i + u_{i-1} \mid}{\mid u_{i+1} - u_i \mid + \mid u_i - u_{i-1} \mid + \epsilon[\mid u_{i+1} \mid -2 \mid u_i \mid + \mid u_{i-1} \mid]} \;, \tag{8.1}$$

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. The constant ϵ is given a value of 10^{-2}, and can be overridden through the `refine_filter` runtime parameter. 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 featuresin the flow variable u. When extending this criterion to multidimensions, all cross derivatives are computed, and the following generalization of the above expression is used

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

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

8.5.3 Usage

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) as well as to 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. 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. 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. Two rules govern the establishment of refined child blocks in PARAMESH. First, the cells of a refined child block must be one-half as large as those of its parent block in each spatial dimension. Second, a block's children must be nested;

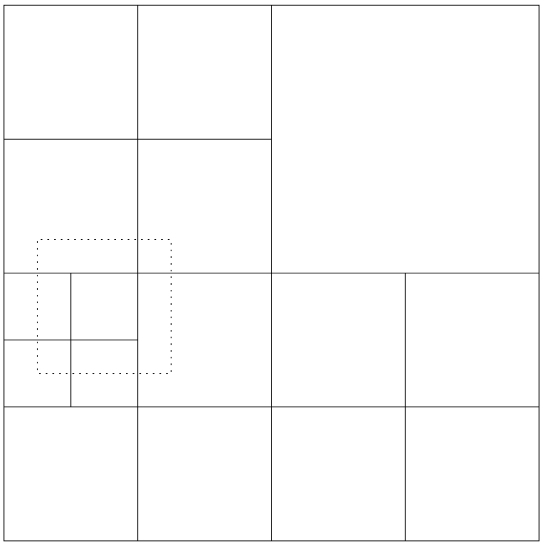


Figure 8.3: A simple computational domain showing varying levels of refinement. The dotted lines around one block outline the guard cells for that block.

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. A simple domain is shown in Fig. 8.3.

Each block contains `nxb` $\times$ `nyb` $\times$ `nzb` interior cells and a set of guard cells (Fig. 8.4). 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 modules (usually hydrodynamics); for the explicit PPM algorithm distributed with FLASH, four guard cells are needed in each direction, as illustrated in

PARAMESH handles the filling of guard cells with information from other blocks or a user-specified external boundary routine. If a block's neighbor has the same level of refinement, PARAMESH fills its guard cells using a direct copy from the neighbor's interior cells. If the neighbor has a different level of refinement, the neighbor's interior cells are used to interpolate guard cell values for the block. If the block and its neighbor are stored in the memory of different processors, PARAMESH handles the appropriate parallel communication (blocks are not split between processors). PARAMESH supports only linear interpolation for guard cell filling at jumps at refinement, but it is easily extended to allow other interpolation schemes.

In FLASH, several different interpolation methods can be chosen at setup time. Each interpolation scheme is stored in a subdirectory under `/source/mesh/amr/paramesh2.0`. You should choose a method that matches the geometry of the simulation. At present, not all geometries are supported (the code will abort if you try to run with an unsupported geometry). Once each block's guard cells are filled, it can be updated independently of the other blocks. PARAMESH also enforces flux conservation at jumps in refinement, as described by Berger and Colella (1989). At jumps in refinement, the fluxes of mass, momemtum, energy (total and internal), and species density in the fine cells across boundary cell faces are summed and passed to their parent. The parent and the neighboring cell are at the same level of refinement (because PARAMESH limits the jumps in refinement to be one level between blocks). The flux in the parent that was computed by the more accurate fine zones is taken as the correct flux through the interface and is passed to the corresponding coarse face on the neighboring block (see Fig. 8.5). The summing allows a geometrical weighting to be implemented for non-Cartesian geometries, which ensures that the proper corrected flux is computed.

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

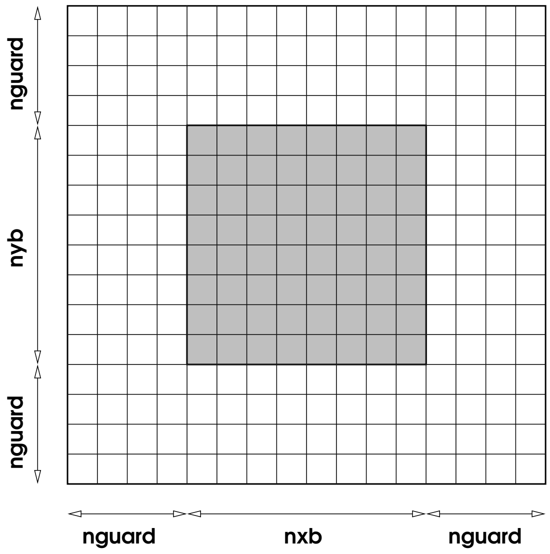


Figure 8.4: A single 2-D AMR block showing the interior zones (shaded) and the perimeter of guard cells.

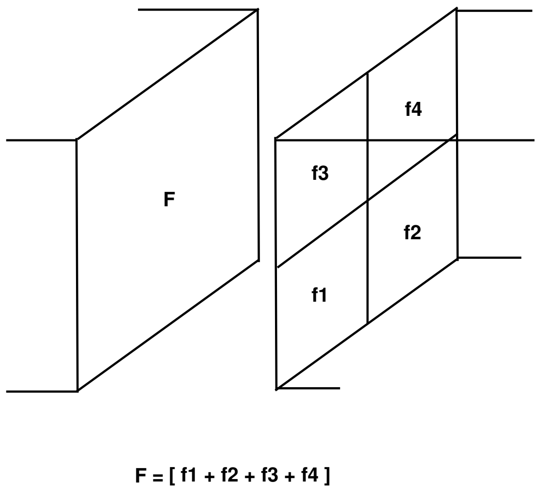


Figure 8.5: Flux conservation at a jump in refinement. The fluxes in the fine cells are added and replace the coarse cell flux (F)

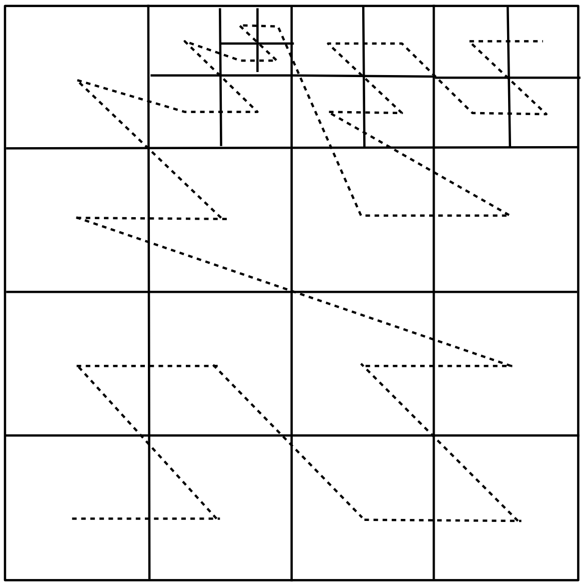


Figure 8.6: Morton space-filling curve

involves deletion of a block and all its siblings (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 Fig. 8.6.

During the distribution step, each block is assigned a weight (an estimate of 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 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.

8.5.3.1 Dividing the computational domain

Dividing the domain is the first step in the mesh-generation process. This routine is responsible for creating the initial top-level block(s) and setting the neighbors of these blocks correctly. These initial blocks then form the top of the tree, and new blocks may be created by refining these top blocks.

By default, FLASH generates an initial mesh with only one top-level block. There are times when this is inconvenient; for instance, when simulating a domain longer in one dimension than the other while wanting equal spatial resolution in each dimension.

`divide_domain()` creates an initial mesh of `nblockx * nblocky * nblockz` top level blocks, where `nblockx`, `nblocky`, and `nblockz` are runtime parameters which default to 1. These blocks are all created on one processor, and thus, the total number of these top-level blocks may not exceed the compiled-in parameter `MAXBLOCKS`.

Since `divide_domain()` is responsible for setting the neighbors of top-level blocks correctly (to either other top-level blocks or to external boundary conditions calculated by `tot_bnd`), this is also where periodic boundary conditions are initially set up. If periodic boundary conditions are set in the x-direction, for instance, the blocks that are first 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.

8.5.3.2 Message buffering

In the maintenance of the tree structure during refinement or derefinement, many small messages must be sent between processors. On any system with a non-negligible latency time for sending messages, communication costs can be significantly reduced by batching these many small messages into fewer large messages.

The routines in `batchsend.F90` and `batchsend_dbl.F90` do simple message buffering. In several AMR routines, all blocks need to send to their neighbors a small number of pieces of data along with a tag, as well as to receive some data. The processors they are to send to and receive from are known ahead of time. The routines `b_int_sendrecv()`, `b_logical_sendrecv()`, and `b_dbl_sendrecv()` take as input arrays containing the messages, tags, and processors to which to send or from which to receive and batch them so that as few messages as possible go between processors.

Because of the amount of copying and memory allocation involved in the process, this buffering does have a cost and thus, under some circumstances, may produce a loss rather than a gain in performance. Thus, the message buffering may be turned on or off with the logical runtime parameter `msgbuffer`, which is `.false.` by default.

8.5.4 Choice of grid geometry

Currently, FLASH supports three different types of grid geometry: one-, two-, and three-dimensional Cartesian grids; two-dimensional cylindrical (r, z) grids; and one-dimensional spherical (r) grids. We describe the usage of each type of grid geometry in turn.

8.5.4.1 Cartesian geometry

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

```
geometry = "cartesian"
```

in the runtime parameter file. When running problems that have spherical or cylindrical symmetry on a Cartesian mesh, it is recommended that the refinement marking routine be designed to always refine the origin in order to minimize grid geometry effects there (see Sec. 8.5.6).

The multigrid Poisson solver (`solvers/poisson/multigrid`) supplied with FLASH 2.3 works only in Cartesian geometries. The multipole solver (`solvers/poisson/multipole`) works in any supported "closed" geometry, including 1D spherical, 2D axisymmetric cylindrical, and 3D Cartesian geometries.

8.5.4.2 Cylindrical geometry

Axisymmetric cylindrical geometry (r, z) is supported by FLASH in two dimensions (polar (r,θ) geometry and 3D cylindrical (r,z,θ) geometries are not yet supported.) It is assumed that the cylindrical radial coordinate is in the 'x'-direction, and the cylindrical z-coordinate is in the 'y'-direction. To run FLASH with cylindrical coordinates, the `geometry` parameter must be set properly:

```
geometry = "cylindrical"
```

These parameters are interpreted by the hydrodynamics solvers and add the necessary geometrical factors to the divergence terms.

As discussed in Sec. 8.5.3, to ensure conservation at a jump in refinement, a flux correction step is taken. Here we use the fluxes leaving the fine zones adjacent to a coarse zone to make a more accurate flux entering the coarse zone.

Fig. 8.7 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

$$A = \pi(r_r^2 - r_l^2) \ , \tag{8.3}$$

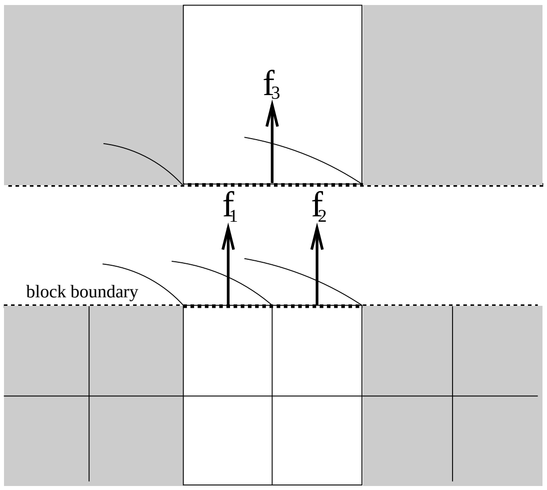


Figure 8.7: Cartoon showing two fine zones and a coarse zone at a jump in refinement in the cylindrical 'z' direction. The block boundary has been cut apart here for illustrative purposes. The fluxes out of the fine blocks are shown as f_1 and f_2. These will be used to compute a more accurate flux entering the coarse flux f_3. The area that the flux passes through is shown as the annuli at the top of each fine zone and the annulus below the coarse zone.

where r_r and r_l are maximum and minimum zone coordinates in the radial direction, respectively. The flux entering the coarse zone above the jump in refinement is corrected to agree with the fluxes leaving the fine zones that border it. This correction is weighted according to the areas

$$f_3 = \frac{A_1 f_1 + A_2 f_2}{A_3} \ . \tag{8.4}$$

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 zones.

When using the multipole Poisson solver in 2D axisymmetric geometry, the gravitational boundary type should be set to `"isolated"`. In this geometry multipole moments $\ell > 0$ (`mpole_lmax`) can now be accommodated, but only the $m = 0$ terms are used.

8.5.4.3 Spherical geometry

One-dimensional spherical problems (using the radial coordinate r) can be performed by specifying

```
geometry = "spherical"
```

in the runtime parameter file. Flux corrections use area weightings as for 2D cylindrical geometry. If the minimum radius is chosen to be zero (`xmin = 0.`), the left-hand boundary type should be reflecting. When using the multipole Poisson solver in 1D spherical coordinates, the gravitational boundary type should be `"isolated"`. Note that in this case it does not make sense to use a multipole moment ℓ (`mpole_lmax`) larger than 0.

8.5.4.4 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 zone-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.

When using a supported non-Cartesian geometry, you should use the geometrically correct prolongation routines in `source/mesh/amr/paramesh2.0/quadratic_cylindrical` (for cylindrical coordinates) or `source/mesh/amr/paramesh2.0/quadratic_spherical` (for spherical coordinates) by including this in your `Modules` file. Other geometry types can be added in a manner analogous to that implemented here.

The default restriction routines understand Cartesian, 1-d spherical, and 2-d cylindrical geometries by default. A zone-volume weighted average is used when restricting the child data up to the parent. For example, in 2-d, the restriction would look like

$$\langle f \rangle_{i,j} = \frac{V_{ic,jc}\,\langle f \rangle_{ic,jc} + V_{ic+1,jc}\,\langle f \rangle_{ic+1,jc} + V_{ic,jc+1}\,\langle f \rangle_{ic,jc+1} + V_{ic+1,jc+1}\,\langle f \rangle_{ic+1,jc+1}}{V_{i,j}} \ , \tag{8.5}$$

where $V_{i,j}$ is the volume of the zone with indices, i, j, and the ic, jc indices refer to the children. In practice, we compute the volume of the parent zone as the sum of the children to ensure conservation.

8.5.4.4.1 2-d cylindrical coordinates

In cylindrical coordinates, we use a bi-quadratic polynomial, neglecting the cross terms, to reconstruct the data in coarse zones before averaging over the fine zones to fill the child data. The general form of the polynomial is

$$f(r,z) = a_1 r^2 + a_2 r + a_3 + a_4 z^2 + a_5 z \ . \tag{8.6}$$

A quadratic polynomial is chosen, since it is symmetric about the parent zone whose children we wish to fill. A quadratic polynomial should lead to a third order accurate expression for the child data, but it is not guaranteed to be monotonic (neither is a bi-linear polynomial in a non-Cartesian geometry). Furthermore, there is no guarantee that the prolongation will preserve the property that the mass fractions sum to unity. We add monotonicity constraints that restore these properties.

The coefficients of the interpolating polynomial are found by forcing it to reproduce the zone-averages in the coarse cells we are using for the reconstruction

$$\frac{2\pi}{V_{i,j}} \int_{z_{j-1/2}}^{z_{j+1/2}} dz \int_{r_{i-1/2}}^{r_{i+1/2}} r dr f(r,z) = \langle f \rangle_{i,j} \ , \tag{8.7}$$

with

$$V_{i,j} = \pi (r_{i+1/2}^2 - r_{i-1/2}^2)(z_{j+1/2} - z_{j-1/2}) \ . \tag{8.8}$$

For convenience, we define

$$\Delta^n r_i = (r_{i+1/2}^n - r_{i-1/2}^n) \ . \tag{8.9}$$

Since there are five unknowns in the reconstruction polynomial $(a_1, a_2, a_3, a_4, a_5)$, we need five coarse zones on which to apply this constraint. Our arrangement is shown in Fig. 8.8. After applying the constraint and solving for the unknowns, we need to integrate the reconstruction polynomial over the children

$$\langle f \rangle_{ic,jc} = \frac{2\pi}{V_{ic,jc}} \int_{z_{jc-1/2}}^{z_{jc+1/2}} dz \int_{r_{ic-1/2}}^{r_{ic+1/2}} r dr f(r,z) \ . \tag{8.10}$$

To ensure that the newly initialized child data retains the property that the sum of the mass fractions is unity, we compute the sum of the abundances, truncated to fall between `smallx` and 1 and compare this to a tolerance (typically 10^{-8}). If we exceed the tolerance, we mark all of the abundances of all the children of the current parent to be renormalized (all the abundances need to be treated the same to retain the proper sum, and all the children of the same parent need to be renormalized to ensure conservation, if any of them fail this test).

For the other variables (*i.e.* those that are not abundances), we check to see if the newly created child data falls outside the extrema set by the parents in the stencil of the reconstruction polynomial. If they do, we monotonize all of the children of that parent using direct insertion.

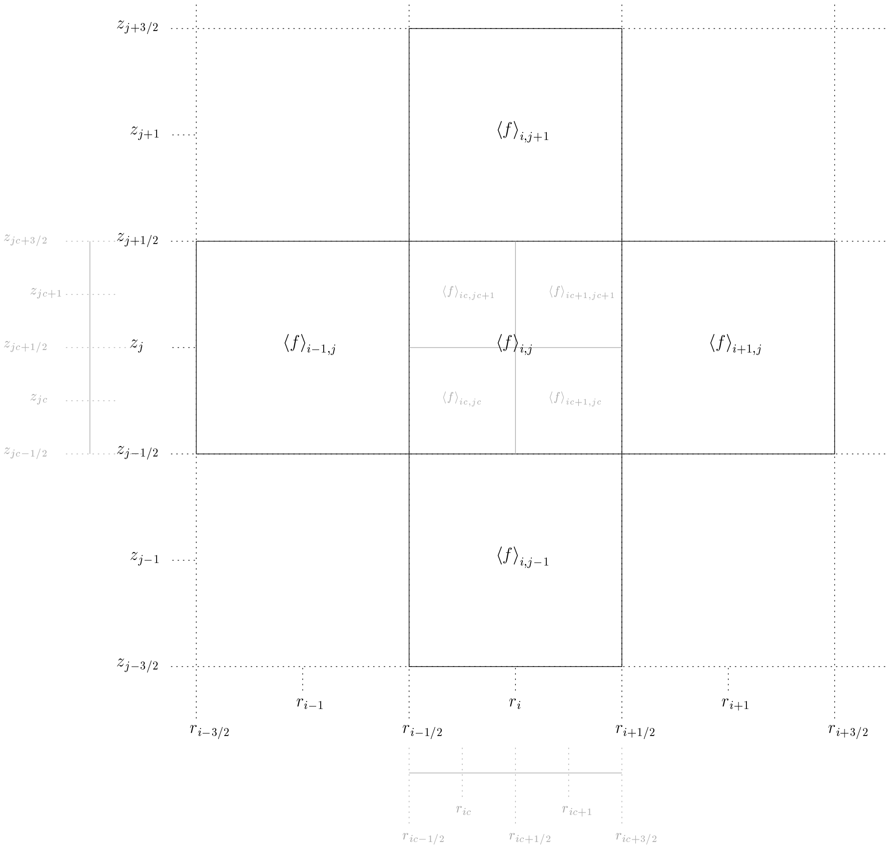


Figure 8.8: Coarse zone arrangement for conservative prolongation in 2-d cylindrical coordinates. We want to use the zone-average data in the five coarse zones ($\langle f \rangle_{i,j}$, $\langle f \rangle_{i+1,j}$, $\langle f \rangle_{i-1,j}$, $\langle f \rangle_{i,j-1}$, $\langle f \rangle_{i,j+1}$) to initialize the children (shown in gray) of the i, j zone. The gray axes indicate the index space for the child zones.

8.5.4.4.2 1-d spherical coordinates

The prolongation routine for 1-d spherical coordinates works analogously to the 2-d cylindrical one described above. The reconstruction polynomial is a simple quadratic

$$f(r) = a_1 r^2 + a_2 r + a_3 \ , \tag{8.11}$$

with the constraint that

$$\frac{4\pi}{V_i} \int_{r_{i-1/2}}^{r_{i+1/2}} r^2 dr f(r) = \langle f \rangle_i \ . \tag{8.12}$$

This constraint is applied to the parent of the children we are filling and to the parents on either side. These three constraints are solved to yield the coefficients of the reconstruction polynomial (a_1, a_2, a_3). The reconstruction polynomial is then integrated over the two newly created children

$$\langle f \rangle_{ic} = \frac{4\pi}{V_{ic}} \int_{r_{ic-1/2}}^{r_{ic+1/2}} r^2 dr f(r) \ , \tag{8.13}$$

where ic is the child zone index.

The same monotonicity constraints as described in the cylindrical prolongation section above are used in spherical coordinates as well.

8.5.5 Using a single-level grid in PARAMESH

By default, FLASH will run a problem on an adaptive mesh, keeping the level of refinement of a block between `lrefine_min` and `lrefine_max`. Sometimes it is useful to run a problem with a single-level mesh. While there is a single-level mesh module distributed with FLASH (see Sec. ??), the `paramesh` module also can be run in a single-level 'mode', which will have only slightly more overhead that the single-level mesh module. The basic steps to set this up are outlined below.

A typical problem in FLASH is set up with a single block at the top of the tree. As the refinement criteria is applied to the initial conditions, this block and any children are refined to create the initial mesh. If you are running on a single-level grid, there is no need to carry around the entire tree hierachry, only the leaf blocks are needed. To get around this, we can use the `divide_domain` functionality (see Sec. 8.5.3.1) to create as many top level blocks as are needed to satisfy our resolution needs. This is accomplished by using the `nblockx`, `nblocky`, and `nblockz` runtime parameters to specify how many blocks to create in each direction.

Since you are placing the same resolution everywhere in the domain, it is no longer advantageous to use small blocks. Instead, the number of zones in a block can be increased, which will reduce the memory overhead (ratio of guard cells to interior zones in a single block). When you run setup on a problem, you can set these values as arguments to setup. Currently, you cannot set `nxb`, `nyb`, and `nzb` as runtime parameters. Please note that some sections of the code assume that these quantities are even.

Be aware that while there is benefit to having a smaller number of larger blocks, you can swing too far in this direction and experience a performance hit. PARAMESH does all of its load balancing based on blocks. Additionally, a block must exist entirely on one processor. Therefore, make sure you have at least as many blocks as the number of processors on which you plan to run. Blocks which are too large can also result in poor cache performance.

The number of computational zones in each direction can be computed as

$$N_x^{\rm zones} = \texttt{nblockx} \times \texttt{nxb} \tag{8.14}$$

for the x-direction. Adjust `nxb` and `nblockx` to get the desired number of zones in the x-direction (and similarly for the other coordinate directions). You can then set `lrefine_max = lrefine_min = 1`.

Since there will be no refinement in the problem, the next step is to instruct the code to no longer check for refinement. This is accomplished by setting `nref` to a very large number. `nref` is the frequency (in time steps) at which to check the refinement criteria and defaults to 2.

Finally, since there will be no jumps in refinement, the flux conservation step is not necessary. This can be eliminated by commenting out the `FLUX` preprocessor definition in `hydro_sweep`. This will instruct the code to skip over the conservation step.

8.5.6 Modifying the refinement criteria with `MarkRefLib`

Sometimes, it may be desirable to refine a particular region of the grid, independent of the second derivative of the variables. This could 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. The `MarkRefLib` module contains methods that can refine a rectangular or circular region or using some variable threshold. It is intended to be used inside the `mark_grid_refinement` routine. A copy of `mark_grid_refinement.F90` should be placed in the `setups` directory for the problem you are working on. To use this library,

```
use markRefLib
```

should be placed in the header of the function. Then the call to the `markRefInRadius`, `markRefInRect`, or `markRefVarThreshold` routine should be made in the region marked "insert user specified refinement criteria here". We discuss the individual methods below.

- `markRefInRadius`
  The general call to `markRefInRadius` is

  ```
  call markRefInRadius(xc, yc, zc, radius, lref)
  ```

  `markRefInRadius` takes an x (`xc`), y (`yc`), and z (`zc`) coordinate of the center of a sphere (circle in 2-d) and a radius (`radius`) and ensures that the mesh is refined up to `lref` levels for any blocks in that sphere (if `lref`> 0) or is refined once (if `lref`< 0).

- `markRefWithRadius`
  The general call to `markRefWithRadius` is

  ```
  call markRefWithRadius(xc, yc, zc, radius, lref)
  ```

  `markRefWithRadius` takes an x (`xc`), y (`yc`), and z (`zc`) coordinate of the center of a sphere (circle in 2-d), and a radius (`radius`) and ensures that the mesh is refined up to `lref` levels for any blocks containing points on the surface of that sphere (if `lref`> 0) or is refined once (if `lref`< 0).

- `markRefInRect`
  The general call to `markRefInRect` is

  ```
  call markRefInRect(xlb, xrb, ylb, yrb, zlb, zrb, lref, contained)
  ```

  `markRefInRect` refines blocks containing any points within a given rectangular region having lower left coordinate (`xlb`,`ylb`,`zlb`) and upper right coordinate (`xrb`,`yrb`,`zrb`). "Rectangular" is interpreted on a dimension-by-dimension basis: the region is an interval, rectangle, or rectangular parallelepiped in 1/2/3D Cartesian geometry; the rectangular cross-section of a rectangular torus in 2D axisymmetric $(r - z)$ cylindrical geometry; an annular wedge in 2D polar $(r - \theta)$ cylindrical geometry; or an annulus in 1D spherical (r) geometry. Either blocks are brought up to a specific level of refinement `lref` (if `lref`> 0), or each block is refined once (if `lref`< 0). If the `contained` parameter is nonzero, only blocks completely contained within the rectangle are refined; otherwise blocks with any overlap at all are refined.

- `markRefVarThreshold`
  To refine on a critical value of a variable, `markRefVarThreshold` takes an array containing that variable in the form

  ```
  real, dimension(nxb,nyb,nzb,maxblocks), :: VarVect
  ```

  a threshold value for the variable (`var_th`), and a flag (`icmp`) indicating whether to refine if the variable is less than the threshold (`icmp` < 0) or greater than the threshold (`icmp` > 0). Any block meeting that criterion is refined up to `lref` levels of refinement. The general form of the call is

  ```
  call markRefVarThreshold(VarVect, var_th, icmp, lref)
  ```

- `markRefOnEllipsoidSurface`
  The general call to `markRefOnEllipsoidSurface` is

  ```
  call markRefOnEllipsoidSurface(x, y, z, a1, a2, a3, lref)
  ```

  `markRefOnEllipsoidSurface` refines all blocks containing points on an ellipsoidal surface centered on (`xc`,`yc`,`zc`) with semimajor axes (`a1`,`a2`,`a3`). Either blocks are brought up to a specific level of refinement `lref` (if `lref`> 0) or each block is refined once (if `lref`< 0).

Chapter 9

Hydrodynamics modules

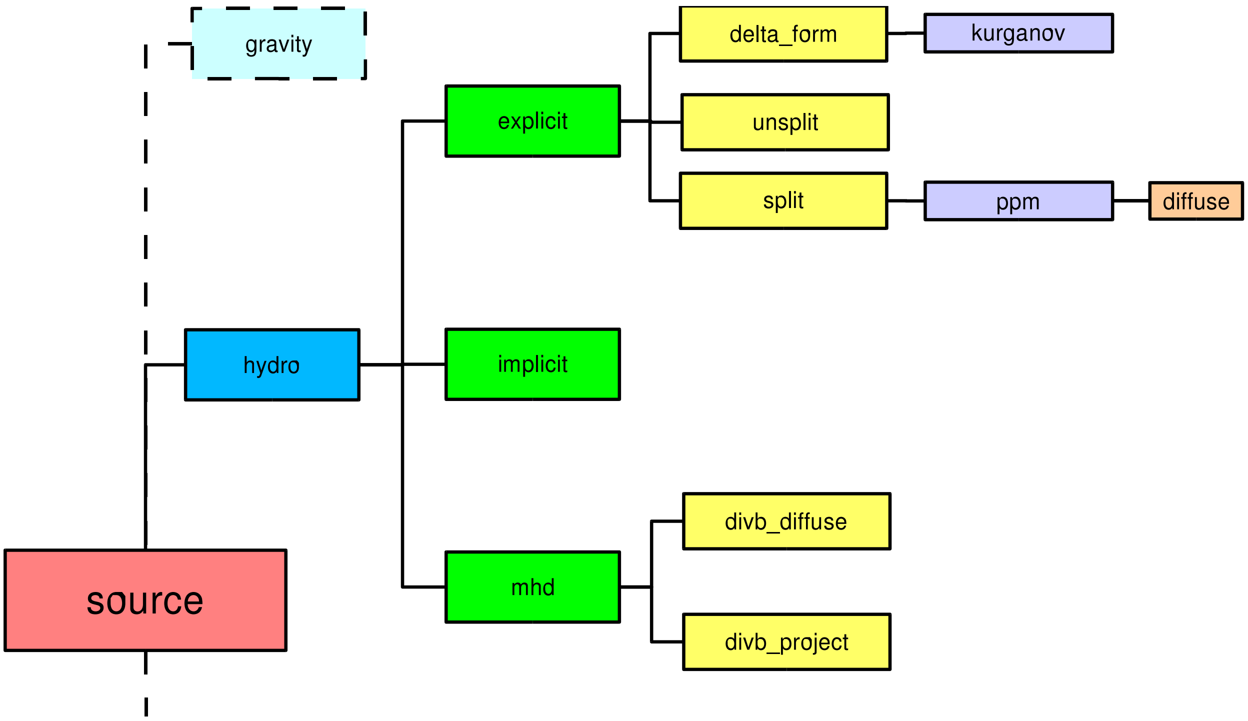


Figure 9.1: The hydrodynamics module directory.

The `hydro` module solves Euler's equations for compressible gas dynamics in one, two, or three spatial dimensions. These equations can be written in conservative form as

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0 \tag{9.1}$$

$$\frac{\partial \rho \mathbf{v}}{\partial t} + \nabla \cdot (\rho \mathbf{v}\mathbf{v}) + \nabla P = \rho \mathbf{g} \tag{9.2}$$

$$\frac{\partial \rho E}{\partial t} + \nabla \cdot [(\rho E + P)\, \mathbf{v}] = \rho \mathbf{v} \cdot \mathbf{g}\ , \tag{9.3}$$

Table 9.1: Runtime parameters used with the hydrodynamics (`hydro`) modules.

Variable	Type	Default	Description
`eint_switch`	real	0	If $\epsilon <$ `eint_switch` $\cdot \frac{1}{2}\vert\mathbf{v}\vert^2$, 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.

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,

$$E = \epsilon + \frac{1}{2}|\mathbf{v}|^2 \ , \tag{9.4}$$

$\mathbf{g}$ 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

$$P = (\gamma - 1)\rho\epsilon \ , \tag{9.5}$$

where γ is the ratio of specific heats. More general equations of state are discussed in Sec. 10.2.1.

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

$$\epsilon = E - \frac{1}{2}|\mathbf{v}|^2 \tag{9.6}$$

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

$$\frac{\partial\rho\epsilon}{\partial t} + \nabla \cdot [(\rho\epsilon + P)\,\mathbf{v}] - \mathbf{v} \cdot \nabla P = 0 \ . \tag{9.7}$$

If the internal energy is a small fraction of the kinetic energy (determined via the runtime parameter `eint_switch`), then the total energy is recomputed using the internal energy from eq. (9.7) and the velocities from the momentum equation. Numerical experiments using the PPM solver included with FLASH showed that using eq. (9.7) when the internal energy falls below 10^{-4} of the kinetic energy helps avoid the truncation errors while not affecting the dynamics of the simulation.

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

$$\frac{\partial\rho X_\ell}{\partial t} + \nabla \cdot (\rho X_\ell \mathbf{v}) = 0 \ , \tag{9.8}$$

where X_ℓ is the mass fraction of the ℓth species, with the constraint that $\sum_\ell X_\ell = 1$. FLASH will enforce this constraint if you set the runtime parameter `irenorm` equal to 1. Otherwise, FLASH will only restrict the abundances to fall between `smallx` and 1. The quantity 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.

All hydrodynamic modules, as well as the MHD module described in Sec. 9.3, supply the runtime parameters and solution variables described in Tables 9.1 and 9.2. Two hydrodynamic modules are included. The first, discussed in Sec. 9.1, is based on the directionally split piecewise-parabolic method (PPM) and makes use of second-order Strang time splitting. The second, discussed in Sec. 9.2, is based on Kurganov methods and can make use of Strang splitting or Runge-Kutta time advancement. Explicit solvers such as PPM make use of the additional runtime parameter described in Table 9.3.

Table 9.2: Solution variables used with the hydrodynamics (`hydro`) modules.

Variable	Attributes	Description
`dens`	ADVECT NORENORM CONSERVE	density
`velx`	ADVECT NORENORM NOCONSERVE	x-component of velocity
`vely`	ADVECT NORENORM NOCONSERVE	y-component of velocity
`velz`	ADVECT NORENORM NOCONSERVE	z-component of velocity
`pres`	ADVECT NORENORM NOCONSERVE	pressure
`ener`	ADVECT NORENORM NOCONSERVE	specific total energy $(T+U)$
`temp`	ADVECT NORENORM NOCONSERVE	temperature

Table 9.3: Runtime parameters used with the explicit hydrodynamics (`hydro/explicit`) modules.

Variable	Type	Default	Description
`cfl`	real	0.8	Courant-Friedrichs-Lewy (CFL) factor; must be less than 1 for stability in explicit schemes

9.1 The piecewise-parabolic method (PPM)

9.1.1 Algorithm

FLASH includes a directionally split piecewise-parabolic method (PPM) solver descended from the PROMETHEUS code (Fryxell, Müller, and Arnett 1989). The basic PPM algorithm is described in detail in Woodward and Colella (1984) and Colella and Woodward (1984). It is a higher-order version of the method developed by Godunov (1959). FLASH implements the direct-Eulerian version of PPM.

Godunov's method uses a finite-volume spatial discretization of the Euler equations together with an explicit forward time difference. Time-advanced fluxes at cell boundaries are computed using the numerical solution to Riemann's shock tube problem at each boundary. Initial conditions for each Riemann problem are determined by assuming the nonadvanced 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 this 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 user's guide. However, for comparison with other codes, we note that the implementation of PPM in FLASH 2.x uses the direct Eulerian formulation of PPM and the technique for allowing nonideal equations of state described by Colella and Glaz (1985). For multidimensional problems, FLASH 2.x uses second-order operator splitting (Strang 1968). We note below the extensions to PPM that we've implemented.

The PPM algorithm includes a steepening mechanism to keep contact discontinuities from spreading

Table 9.4: Runtime parameters used with the PPM (`hydro/explicit/split/ppm`) module.

Variable	Type	Default	Description
`epsiln`	real	0.33	PPM shock detection parameter ϵ
`omg1`	real	0.75	PPM dissipation parameter ω_1
`omg2`	real	10	PPM dissipation parameter ω_2
`igodu`	integer	0	If set to 1, use the Godunov method (completely flatten all interpolants)
`vgrid`	real	0	Scale factor for grid velocity (not implemented)
`nriem`	integer	10	Max number of iterations to use in Riemann solver
`rieman_tol`	real	10^{-5}	Convergence factor for Riemann solver
`ppm_modifystates`	boolean	`.false.`	Modify input states to Riemann solver to take into account gravity.
`cvisc`	real	0.1	Artificial viscosity constant
`hybrid_riemann`	boolean	`.false.`	use HLLE in shocks to remove odd-even decoupling instability
`use_cma_flattening`	boolean	`.false.`	use additional flattening on the mass fractions to prevent overshoots
`use_steepening`	boolean	`.true.`	use contact steepening to sharpen contact discontinuities

over too many zones. Its use requires some care, since under certain circumstances, it can produce incorrect results. For example, it is possible for the code to interpret a very steep (but smooth) density gradient as a contact discontinuity. When this happens, the gradient is usually turned into a series of contact discontinuities, producing a stair step appearance in one-dimensional flows or a series of parallel contact discontinuities in multi-dimensional flows. Under resolving the flow in the vicinity of a steep gradient is a common cause of this problem. The directional splitting used in our implementation of PPM can also aggravate the situation. The contact steepening can be disabled at runtime by setting `use_steepening = .false.`.

The version of PPM in the FLASH code has an option to more closely couple the hydrodynamic solver with a gravitational source term. This can noticeably reduce spurious velocities caused by the operator splitting of the gravitational acceleration from the hydrodynamics. In our 'modified states' version of PPM, when calculating the left and right states for input to the Riemann solver, we locally subtract off from the pressure field the pressure that is locally supporting the atmosphere against gravity; this pressure is unavailable for generating waves. This can be enabled by setting `ppm_modifiedstates = .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 fixes they describe, the renormalization of the average mass fraction state as returned from the Riemann solvers (CMA eq. 13), and the (optional) additional flattening of the mass fractions to reduce overshoots (CMA eq. 14-16). The latter procedure is off by default and can be enabled by setting `use_cma_flattening = .true.`.

Finally, there is an odd-even instability that can occur with shocks that are aligned with the grid. This was first pointed out by Quirk (1997), who tested several different Riemann solvers on a problem designed to demonstrate this instability. The solution he proposed is to use a hybrid Riemann solver, using the regular solver in most regions but switching to an HLLE solver inside shocks. We've implemented such a procedure, which can be enabled by setting `hybrid_riemann = .true.`. The `odd_even` test problem can be used to examine the effects of this hybrid Riemann solver at removing this instability.

9.1.2 Usage

The `hydro/explicit/split/ppm` module supplies the runtime parameters described in Table 9.4.

9.1.3 Diffusion

Any of several diffusive processes can be added to the Euler equations in the PPM module. All of these are treated explicitly in FLASH and follow the same approach. A diffusive flux, which is proportional to the gradient of the quantity, is calculated by finite difference. The fluxes are then calculated and added to the fluxes generated by the PPM module. This addition is done before any of the zones are updated in the hydro step. This ensures conservation, since the total flux (including the diffusive flux) will be corrected during the flux conservation step.

To include a diffusive process, you must use `hydro/explicit/split/ppm/diffuse` in your `Modules` file. Then the logical runtime parameters `diffuse_therm`, `diffuse_visc`, and `diffuse_species` should be set to `.true.` or `.false.` depending on whether you wish to include these diffusive terms or not in your simulation. Each diffusive process has an associated coefficient, which defaults to a constant. To override this default, pick the desired form of the coefficient from the materials module.

All of the diffusive processes take on a form like

$$\frac{\partial X}{\partial t} = \nabla \cdot D \nabla X \ , \tag{9.9}$$

which, when solved explicitly, has a timestep limiter of the form

$$t_{\rm diff} < \frac{1}{2}\frac{(\delta x)^2}{D} \ . \tag{9.10}$$

This timestep limiter is used for all of the diffusive processes by computing the maximum diffusion coefficient of all the processes and finding the minimum timestep. This timestep will be used if it is smaller than the hydro timestep.

9.1.3.1 Thermal Diffusion

The energy equation in the PPM module can be modified to include thermal diffusion

$$\frac{\partial \rho E}{\partial t} + \nabla \cdot (\rho E + P)\,\mathbf{v} = \rho \mathbf{v} \cdot \mathbf{g} + \nabla \cdot (\sigma(X_i, \rho, T)\nabla T) e_{nuc}(X_i, \rho, T) \ , \tag{9.11}$$

where $\sigma(X_i, \rho, T)$ is the conductivity and

$$F_{heat} = -\sigma(X_i, \rho, T)\nabla T \tag{9.12}$$

is the explicit heat flux.

There are several conductivity modules in `source/materials/conductivity` that are available for use with this routine. To use conductivity, include one of them in your `Modules` file. `conductivity/stellar` uses a conductivity appropriate for the degenerate matter of stellar interiors. `conductivity/spitzer` implements a conductivity according to the formulation of Spitzer (1962). In `conductivity/constant`, the heat conductivity is assumed constant; σ is set equal to the runtime parameter `conductivity_constant`. In `conductivity/constant-diff`, the thermal diffusivity ($\lambda = \frac{\sigma}{\rho c_p}$) is kept equal to the runtime parameter `diff_constant`. This is equivalent to diffusing temperature directly, *e.g.*

$$\frac{\partial T}{\partial t} + \nabla \cdot T\mathbf{v} = \nabla \cdot (\lambda \nabla T) \ . \tag{9.13}$$

9.1.3.2 Viscosity

With viscosity, the momentum is the quantity that is diffused

$$\frac{\rho \mathbf{v}}{\rho t} + \nabla \cdot \rho \mathbf{v}\mathbf{v} + \nabla P = \rho \mathbf{g} + \nabla \cdot (\nu \nabla \mathbf{v}) \ . \tag{9.14}$$

The fluxes are calculated as in the thermal diffusion, although there is one flux for each velocity component. There are two viscosity modules in `source/materials/viscosity`. In `viscosity/constant`, the viscosity ν is assumed constant and set by the runtime parameter `diff_visc_nu`. `viscosity/spitzer` uses a viscosity computed according to the classical Spitzer (1962) prescription. Total energy fluxes are not updated by viscosity, since it is assumed the effect is small.

9.1.3.3 Species Diffusion

This module diffuses the partial density of species. Thus, there are `ionmax` fluxes updated, and they are controlled by an assumed constant diffusivity `diff_spec_D`

$$\frac{\partial \rho X_i}{\partial t} + \nabla \cdot (\rho X_i \mathbf{v}) = \nabla \cdot (D \nabla \rho X_i) \ . \tag{9.15}$$

9.2 The Kurganov hydrodynamics module

The two Kurganov schemes are implemented in the `kurganov` hydro module, which is compatible with all the alternative driver modules. The module is organized as follows. The subroutine `hydro_3d` is essentially a wrapper to the subroutines `kurganov_block_x`, `kurganov_block_y`, and `kurganov_block_z`. These three subroutines implement the reconstruction step for each of the spatial dimensions and call `kurganov_line`, which calculates either the KT or KNP numerical fluxes.

9.2.1 Algorithm

Kurganov and his collaborators have developed a class of numerical methods for hyperbolic conservation laws. Compared to other methods for such systems, the Kurganov methods are simple and inexpensive, because they do not rely on characteristic decompositions or Riemann solvers; the only information they require from the equations is the maximum and minimum signal propagation speeds. However, they may have a lower critical timestep for stability and higher dissipation as a result. The Kurganov methods evolved from methods developed by Tadmor and his colleagues, but differ in that staggered grids are used only as a device in the derivation of the schemes and are not used in the implementation. The `kurganov` hydro module provides two of Kurganov's high-resolution central schemes, each second-order accurate.

There are two parts to each Kurganov method, (i) reconstruction of the conserved variables, which provides cell interface values, and (ii) computation of the interface flux from those interface values. Various second- and third-order reconstruction algorithms have been developed; one-dimensional reconstructions can be extended dimension-by-dimension, but some multidimensional third-order reconstructions have been proposed. Two formulas are available for computing the fluxes from the interface values. For the first numerical flux, several wave speeds were represented by a single estimate, the maximum magnitude of the eigenvalues of the flux Jacobian (Kurganov and Tadmor 2000). Later, two estimates (maximum and minimum eigenvalues) were used, resulting in an improved numerical flux with reduced numerical dissipation (Kurganov, Noelle, and Petrova 2001).

We now describe the second-order reconstruction used in the new hydro module. The reconstruction is one-dimensional and is presented for an equispaced mesh. An arbitrary mesh cell is referred to by subscript i. The reconstruction uses the cell-averaged conserved variables U_i at nearby cells to produce values of the conserved variables at the left and right sides of each cell interface, $U^l_{i+\frac{1}{2}}$ and $U^r_{i+\frac{1}{2}}$, respectively. To update a given cell in each spatial direction, fluxes at two interfaces must be computed; for the reconstruction used in FLASH, this update requires a five-point stencil.

The first step is to compute *limited slopes*, $(U_x)_i$

$$(U_x)_i = \text{minmod}\left[\theta\left(U_{i+1} - U_i\right), \theta\left(U_i - U_{i-1}\right), \frac{1}{2}\left(U_{i+1} - U_{i-1}\right)\right] \ , \tag{9.16}$$

where the minmod function returns the smallest argument in magnitude if all arguments are the same sign and zero if they are not. The parameter $1 \leq \theta \leq 2$ gives some control over the limiter. The minmod limiter, which is one of the most diffusive, is recovered for $\theta = 1$. The monotonized central limiter (Colella and Woodward 1984) is specified for $\theta = 2$; it is significantly less dissipative and is recommended for most cases. Other limiters may also be used to compute the slopes; each has its pros and cons.

Once the slopes have been determined, the interface values are calculated by

$$\begin{aligned} U^l_{i+\frac{1}{2}} &= U_i + \frac{1}{2}(U_x)_i \\ U^r_{i+\frac{1}{2}} &= U_{i+1} - \frac{1}{2}(U_x)_{i+1} \ . \end{aligned} \tag{9.17}$$

Then, on each side of the interface, the speed of sound is computed. This requires the density, mass fractions, and internal energy to be computed; then the equation of state module is called, which returns the pressure and the ratio of specific heats. Finally, the speeds of sound $c^l_{i+\frac{1}{2}}$ and $c^r_{i+\frac{1}{2}}$ are computed by $c = (\gamma P/\rho)^{\frac{1}{2}}$.

The Kurganov-Noelle-Petrova (KNP) numerical flux is defined by

$$F_{i+\frac{1}{2}} = \frac{a^-_{i+\frac{1}{2}} F(U^r_{i+\frac{1}{2}}) + a^+_{i+\frac{1}{2}} F(U^l_{i+\frac{1}{2}})}{a^+_{i+\frac{1}{2}} + a^-_{i+\frac{1}{2}}} + a_{i+\frac{1}{2}} \left(U^r_{i+\frac{1}{2}} - U^l_{i+\frac{1}{2}} \right) \ , \tag{9.18}$$

where for the x-direction,

$$a^+_{i+\frac{1}{2}} = \max\left(0, u^l_{i+\frac{1}{2}} + c^l_{i+\frac{1}{2}}, u^r_{i+\frac{1}{2}} + c^r_{i+\frac{1}{2}}\right) \tag{9.19}$$

$$a^-_{i+\frac{1}{2}} = -\min\left(0, u^l_{i+\frac{1}{2}} - c^l_{i+\frac{1}{2}}, u^r_{i+\frac{1}{2}} - c^r_{i+\frac{1}{2}}\right) \tag{9.20}$$

$$a_{i+\frac{1}{2}} = -\left(a^+_{i+\frac{1}{2}} a^-_{i+\frac{1}{2}}\right) \Big/ \left(a^+_{i+\frac{1}{2}} + a^-_{i+\frac{1}{2}}\right) . \tag{9.21}$$

Eqs. (9.19) and (9.20) are specific to the x-direction because u, the x-component of the velocity, appears; for the y- and z-directions, the appropriate velocity components, v and w, respectively should replace u.

The Kurganov-Tadmor (KT) numerical flux is

$$F_{i+\frac{1}{2}} = \frac{1}{2}\left[F(U^r_{i+\frac{1}{2}}) + F(U^l_{i+\frac{1}{2}}) + a_{i+\frac{1}{2}}\left(U^r_{i+\frac{1}{2}} - U^l_{i+\frac{1}{2}}\right)\right] \ , \tag{9.22}$$

where for the x-direction,

$$a_{i+\frac{1}{2}} = \max\left(|u^l_{i+\frac{1}{2}} + c^l_{i+\frac{1}{2}}|, |u^r_{i+\frac{1}{2}} + c^r_{i+\frac{1}{2}}|\right) . \tag{9.23}$$

As in eqns. (9.19) and (9.20), the appropriate velocity components should be used in eq. (9.23) for the y- and z-directions.

9.2.2 Usage

9.2.2.1 Interaction with alternative driver modules

The `kurganov` hydro module requires the use of the alternative driver and formulation modules. The user should therefore be familiar with section 6.1. The `kurganov` module can be used with drivers written in either delta or state-vector formulations. (Almost all explicit physics modules can be written to do so, and the `kurganov` module can be used as a model.) The line for the `Modules` file to specify the `kurganov` module is

```
INCLUDE hydro/explicit/delta_form/kurganov
```

This directory name is somewhat misleading; even though `delta_form` is in the pathname, the above line is appropriate in the `Modules` file for both delta and state-vector formulations.

The alternative time advancement methods call `hydro_3d`, described in the next section, to invoke the `kurganov` module. In the context of the alternative driver modules, the hydro module does the following on each block. The contribution of the module $L_{hydro}(U)$ is computed. If a delta formulation time advancement has been specified, then $L_{hydro}(U)$ is added to the global ΔU. If a state-vector formulation time advancement is being used, calls to update the solution on the block from $L_{hydro}(U)$ are made. In both cases, the `formulation` module provides the subroutines for these actions.

9.2.2.2 The `hydro_3d` wrapper

The `hydro_3d` subroutine was written as generally as possible; with minimal modification, it can be used as a wrapper for most shock-capturing schemes which compute numerical fluxes. It accepts as an argument the spatial direction – x, y, z, or all – for which the fluxes should be computed. It can therefore be used with time advancement methods based on directional splitting.

Table 9.5: Runtime parameters used with the `kurganov` hydro module.

Para. name	Type	Default	Description
`knp`	integer	1	Specifies KNP numerical flux is to be used; for KT, set knp=0
`lim_theta`	real	1.0	Adjusts slope limiter; 1.0 $\leq$ `lim_theta` $\leq$ 2.0; choose 1.0 for minmod limiter, 2.0 for monotonized centered limiter

In order, the tasks handled by `hydro_3d` include

1. fill guard cells
2. loop over blocks
   (a) eos call for guard cells
   (b) get field data
   (c) get mesh data
   (d) for each applicable direction
      i. get fluxes on equispaced grid: call `kurganov_block` for the appropriate direction
      ii. apply geometry factors
      iii. update global ΔU or locally update solution, depending on formulation
      iv. save block boundary fluxes for AMR correction
3. AMR flux correction
4. loop over blocks
   (a) updates to coarse-block boundaries from AMR flux correction

By managing all these tasks, all the interaction between the Kurganov shock-capturing schemes and the rest of the FLASH code framework is encompassed in `hydro_3d`. The part of the hydro module that is spcific to the Kurganov schemes is restricted to the `kurganov_block_[xyz]` and `kurganov_line` subroutines.

The `kurganov` module accepts two runtime parameters, listed in Table 9.5. The integer `knp` selects which numerical flux formula is to be used, KNP or KT. The real-valued `lim_theta` is θ in eq. (9.16); lower values result in more damping, higher values in less, within the range listed in the table. The runtime parameter `cfl` is common to all explicit hydro modules; Table 9.5 lists only those specific to the Kurganov schemes. Because of the reconstruction, a five-point stencil is required at each mesh cell; consequently, two guard cells are required on each side of a block. Corner guard cells are not required, since the reconstruction is one-dimensional.

9.2.2.3 Caveats

At present, the compatibility of the alternative modules with the rest of the FLASH code is limited. The alternative modules are incompatible with the default `hydro` module, which actually implements parts of the Strang splitting time advancement in addition to the PPM spatial discretization. The time advancement methods implemented in the delta formulation are not compatible with physics modules that update the field variables, and those implemented in the state-vector formulation have not been tested with the other physics modules; these deficiencies are being addressed. Currently, only Cartesian coordinates are supported by the `kurganov` hydro module. The alternative modules have been extensively tested only for a nonreacting, single component gas (`ionmax`=1) using the gamma-law equation of state.

Table 9.6: Additional solution variables used with the MHD (`hydro/mhd`) module.

Variable	Attributes	Description
`magx`	ADVECT NORENORM CONSERVE	x-component of magnetic field
`magy`	ADVECT NORENORM CONSERVE	y-component of magnetic field
`magz`	ADVECT NORENORM CONSERVE	z-component of magnetic field
`divb`	ADVECT NORENORM NOCONSERVE	divergence of magnetic field

Table 9.7: Additional runtime parameters used with the MHD (`hydro/mhd`) module.

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".
`resistive_mhd`	logical	.false.	Include/exclude non-ideal MHD terms.
`killdivb`	logical	.true.	Enable/disable divergence cleaning.

9.3 The magnetohydrodynamics module

9.3.1 Description

The magnetohydrodynamics module included with the FLASH code solves the equations of both ideal and non-ideal MHD. As discussed in Sec. 9.3.2, the MHD module replaces the hydrodynamics module in simulations of magnetized fluids. The two modules are conceptually very similar, and they share the same algorithmic structure. Therefore, in the current version of the FLASH code, the MHD module is a submodule of the hydrodynamics module.

The currently released version of the MHD module uses directional splitting to evolve the equations of ideal and resistive magnetohydrodynamics. Like the hydro modules, the MHD module makes one sweep in each spatial direction to advance physical variables from one time level to the next. In each sweep, the module uses AMR functionality to fill in guard cells and impose boundary conditions. Then it reconstructs characteristic variables and uses these variables to compute time-averaged interface fluxes of conserved quantities. In order to enforce conservation at jumps in refinement, the module makes flux conservation calls to AMR, which redistributes affected fluxes using the appropriate geometric area factors. Finally, the module updates the solution and calls the EOS module to ensure thermodynamical consistency.

After all sweeps are completed, the MHD module enforces magnetic field divergence cleaning. Two options are available: diffusive and elliptic projection cleaning. In order to select a particular method, the user must, respectively, specify either `mhd/divb_diffuse` or `mhd/divb_project` in the problem `Config file`. The default method is diffusive.

The interface of the MHD module is minimal. The module honors all of hydrodynamics module variables, interface functions and runtime parameters described earlier in this chapter. In addition, it declares four global variables and three runtime parameters, which are listed in Tables 9.6 and 9.7.

9.3.2 Algorithm

The magnetohydrodynamic (MHD) module in the FLASH code is based on a finite-volume, cell-centered method that was recently proposed by Powell *et al.* (1999). This particular choice of solver is made to comply with the data structure layout required by the existing hydrodynamics module. As a result of this choice, the MHD module in the FLASH code is fully compatible and is, in fact, swappable with the hydro module.

The MHD module in the FLASH code solves the equations of compressible magnetohydrodynamics in one, two and three dimensions. Written in non-dimensional (hence without 4π or μ_0 coefficients), conservation form, these equations are

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0 \tag{9.24}$$

$$\frac{\partial \rho \mathbf{v}}{\partial t} + \nabla \cdot (\rho \mathbf{v}\mathbf{v} - \mathbf{B}\mathbf{B}) + \nabla p_* = \rho \mathbf{g} + \nabla \cdot \tau \tag{9.25}$$

$$\frac{\partial \rho E}{\partial t} + \nabla \cdot (\mathbf{v}(\rho E + p_*) - \mathbf{B}(\mathbf{v} \cdot \mathbf{B})) = \rho \mathbf{g} \cdot \mathbf{v} + \nabla \cdot (\mathbf{v} \cdot \tau + \sigma \nabla T) + \nabla \cdot (\mathbf{B} \times (\eta \nabla \times \mathbf{B})) \tag{9.26}$$

$$\frac{\partial \mathbf{B}}{\partial t} + \nabla \cdot (\mathbf{v}\mathbf{B} - \mathbf{B}\mathbf{v}) = -\nabla \times (\eta \nabla \times \mathbf{B}) \tag{9.27}$$

where

$$p_* = p + \frac{B^2}{2}, \tag{9.28}$$

$$E = \frac{1}{2}v^2 + \epsilon + \frac{1}{2}\frac{B^2}{\rho}, \tag{9.29}$$

$$\tau = \mu \left((\nabla \mathbf{v}) + (\nabla \mathbf{v})^{\mathrm{T}} - \frac{2}{3}(\nabla \cdot \mathbf{v}) \right) \tag{9.30}$$

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, μ is the viscosity, 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. Starting with version 2.0, the MHD module supports 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 this chapter.

The ideal part of the above equations of magnetohydrodynamics are solved using a high-resolution, finite-volume numerical scheme with MUSCL-type (van Leer 1979) limited gradient reconstruction. In order to maximize the accuracy of the solver, the reconstruction procedure is applied to characteristic variables. Since this may cause certain variables, such as density and pressure, to fall outside of physically meaningful bounds, extra care is taken in the limiting step to prevent this from happening. All other variables are calculated in the module from the interpolated characteristic variables. The non-ideal (viscous, heat conduction and resistive) terms are added in conservative fashion as corrections to ideal fluxes that are generated by the ideal MHD solver (note that in the MHD module non-ideal terms add contributions to the energy equation). All diffusive terms are evaluated using straightforward second-order, central difference schemes.

In order to resolve discontinuous Riemann problems that occur at computational cell interfaces, the code employs a Roe-type solver derived in Powell *et al.* (1999). This solver provides full characteristic decomposition of the ideal MHD equations and is, therefore, particularly useful for plasma flow simulations that feature complex wave interaction patterns. The time integration in the MHD module is done using a second-order, one-step method due to Hancock (Toro 1997). For linear systems with unlimited gradient reconstruction, this method can be shown to coincide with the classic Lax-Wendroff scheme.

A difficulty particularly associated with solving the MHD equations numerically lies in the solenoidality of the magnetic field. The notorious $\nabla \cdot \mathbf{B} = 0$ condition, a strict physical law, is very hard to satisfy in discrete computations. Being only an initial condition of the MHD equations, it enters the equations indirectly and is not, therefore, guaranteed to be generally satisfied unless special algorithmic provisions are made. Without discussing this issue in much detail, which goes well beyond the scope of this user's guide (for example, see Tóth (2000) and references therein), we will remind that there are three commonly accepted methods to enforce the $\nabla \cdot \mathbf{B}$ condition: the elliptic projection method (Brackbill and Barnes 1980), the constrained transport method (Evans and Hawley 1988), and the truncation-level error method (Powell *et al.* 1999). In the FLASH code, the truncation-error and elliptic cleaning methods are implemented.

In the truncation-error method, the solenoidality of the magnetic field is enforced by including several terms proportional to $\nabla \cdot \mathbf{B}$. This removes the effects of unphysical magnetic tension forces parallel to the field and stimulates passive advection of magnetic monopoles if they are spuriously created. In many applications, this method has been shown to be an efficient and sufficient way to generate solutions of high physical quality. However, it has also been shown (Tóth, 2000) that this method can sometimes, for example in strongly discontinuous and stagnated flows, lead to accumulation of magnetic monopoles, whose strength is sufficient to corrupt the physical quality of computed solutions. In order to eliminate this deficiency, the FLASH code also uses a simple yet very effective method originally due to Marder (1987) to destroy the magnetic monopoles on the scale on which they are generated. In this method, a diffusive operator proportional to $\nabla\nabla\cdot\mathbf{B}$ is added to the induction equation, so that the equation becomes

$$\frac{\partial\mathbf{B}}{\partial t} + \nabla\cdot(\mathbf{vB} - \mathbf{Bv}) = -\nabla\times(\eta\nabla\times\mathbf{B}) - \mathbf{v}\nabla\cdot\mathbf{B} + \eta_a\nabla\nabla\cdot\mathbf{B} \ , \tag{9.31}$$

with the artificial diffusion coefficient η_a chosen to match that of grid numerical diffusion. In the FLASH code, $\eta_a = \frac{\lambda}{2}(\frac{1}{\Delta x} + \frac{1}{\Delta y} + \frac{1}{\Delta z})^{-1}$, where λ is the largest characteristic speed in the flow. Since the grid magnetic diffusion Reynolds number is always on the order of unity, this operator locally destroys magnetic monopoles at the rate at which they are created. Recent numerical experiments (Linde and Malagoli, submitted; Powell *et al.* (2001)) indicate that this approach can very effectively bring the strength of spurious magnetic monopoles to levels that are sufficiently low that generated solutions remain physically consistent. The entire $\nabla\cdot\mathbf{B}$ control process is local and very inexpensive compared to other methods. Moreover, one can show that this process is asymptotically convergent (Munz *et al.*, 2000), and each of its applications is equivalent to one Jacobi iteration in solving the Poisson equation in the elliptic projection method. The caveat is that this method only suppresses but does not completely eliminate magnetic monopoles. Whether this is acceptable depends on a particular physical problem.

In order to eliminate magnetic monopoles completely, the FLASH code includes an elliptic projection method. In this method, the unphysical divergence of magnetic field can be removed to any desired level down to machine precision. This is achieved by solving a Poisson equation for a correcting scalar field, whose gradient removes contaminated field components when subtracted from the magnetic field. The Poisson solver needed for this operation is the multigrid solver that is also used by the gravity module.

9.3.3 New features

The following features are new in the FLASH 2.3 release:

- Viscous terms are added to the MHD module.
- Heat conduction is added to the MHD module.

In order to avoid adding separate logical tests and to preserve backward compatibility with previously released versions of the code, all non-ideal terms in this release are grouped together in the "resistive MHD" part of the module. Among other things, this means that now `resistive_mhd` flag variable turns on/off resistive, viscous and heat conduction simultaneously. This is important to remember, because, unlike that for resistivity, the default values of kinematic viscosity and thermal conductivity in FLASH are not equal to zero. Therefore, if resistive MHD part of the module is turned on, one must provide appropriate values for all transport coefficients in the input parameter file. Section 10 discusses in detail how this should be done.

Chapter 10

Material properties modules

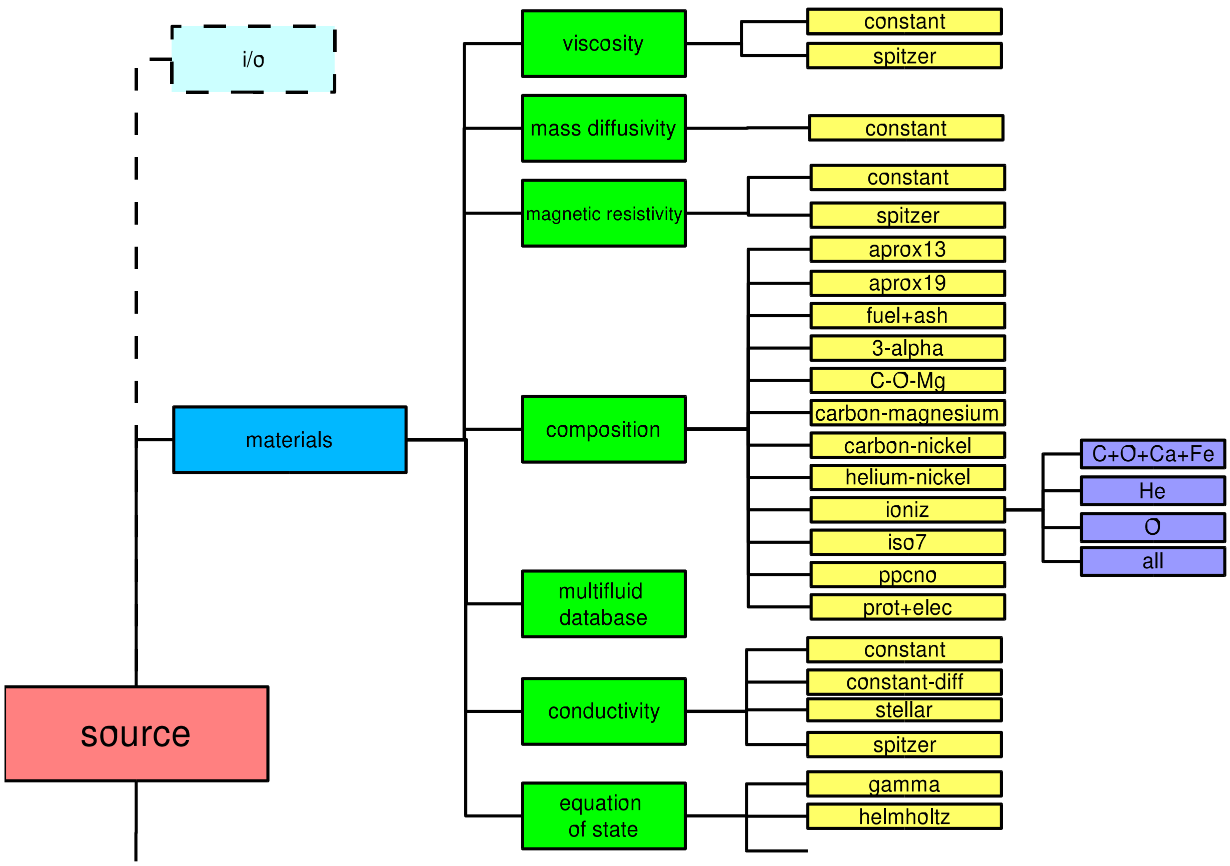


Figure 10.1: The materials module directory.

FLASH has the ability to track multiple fluids, each of which can have its own properties. The materials module handles these, as well as other things like EOS, composition, and conductivities.

10.1 The multifluid database

To access any of the fluid properties, you must use the multifluid database. This can be accomplished in any FLASH routine by including the line

```
use multifluid_database
```

along with any of the other modules you have included. This module provides interface functions that can be used to set or query a fluid's properties. As with the other databases in FLASH, most of the properties have both a string name and an integer key that can be used in the database call. Calling the function with the integer key will be faster, since it avoids expensive string comparisons. The available properties are listed in Table 10.1.

Table 10.1: Properties available through the multifluid database.

Name	**Integer key**	**Property**	**Data type**
"name"	N/A	fluid name	string
"short name"	N/A	chemical symbol	string
"num total"	mf_prop_A	A	real
"num positive"	mf_prop_Z	Z	real
"num neutral"	mf_prop_N	N	real
"num negative"	mf_prop_E	E	real
"binding energy"	mf_prop_Eb	binding energy	real
"adiabatic index"	mf_prop_gamma	gamma	real
"EOS weighting"	mf_prop_EOS	1.0 if it contributes to the EOS, 0.0 otherwise	real

Once the multifluid database is initialized (usually by the `materials/composition` module function `init_mat()`, the integer `nfluids` is publicly available, giving the number of fluids carried by FLASH.

An example of using the multifluid database to define two fluids to be tracked is provided by the `example` setup discussed in Sec. 2.2. Note that, by default, all fluids have an equation-of-state (EOS) weighting of unity, meaning that they contribute to the pressure and temperature and that their abundances are restricted or renormalized at the end of each timestep. Setting this weighting to zero for a fluid enables that fluid to be passively advected without contributing to the equation of state or abundance renormalization.

We now briefly discuss the various interfaces to the multifluid database. Many of these functions are overloaded to accept either string or integer properties (as listed in the table above) or to include optional arguments. We discuss only the generic interface here.

- `add_fluid_to_db(name, short_name, properties, status)`

  A quick way to set a number of properties for an individual fluid in a single subroutine call. Looks for the next uninitialized fluid (`init_mfluid_db()` sets the names of all fluids to `UNINITIALIZED`) and sets its properties according to the values specified in the subroutine call. Properties can be specified in the order A, Z, N, E, ... or by keyword. For example,

  ```
  call add_fluid_to_db ("helium", "He", A=4., Z=2., N=2.)
  ```

  Properties not specifically initialized are set to 0. The status parameter is an optional status variable.

  This function call is usually used when initializing the fluids in FLASH. Each composition sets the properties for all the fluids in the routine `init_materials()`.

- `set_fluid_property(f, p, v, status)`

  Set the property `p` of fluid `f` to value `v`. The fluid, `f` can be specified either using its string name or index. `p` is either a string identifier or integer key specifying the property.. The value `v` can be real-valued or string-valued, depending on the property being modified. `status` is an optional variable that will be set to 0 if the operation was successful, -1 if not. Reasons why the operation can fail include: `f` out of bounds if `f` is an index; `f` not found if `f` is a string name; `p` not a valid property identifier; `v` not a valid value for the given property.

- `get_fluid_property(f, p, v, status)`

  Like the set version, but `v` now receives the value instead of setting it.

- `get_mfluid_property(p, v, status)`

  Return the value of the property *p* for all defined fluids. `v` must be an array of the correct type. `status` is an optional exit status variable. `v` is filled with values up to the minimum of (its size, number of fluids).

- `find_fluid_index(f, i)`

  Find the database index `i` of a fluid named `f`. If `init_mfluid_db()` has not been called, or if the fluid name is not found, this function terminates with an error. Errors are signaled by setting i to `MFLUID_STATUS_FAIL`.

- `query_mfluid_sum(p, w, v, status)`

  Given a property name `p` and an array of weights `w`, return the weighted sum of the chosen property in `v`. `w` should be an array of length equal to the number of fluids in the database or else a one-element array. If it is neither, or if the named property is invalid, the routine terminates. Typically, the weights used are the mass fractions of each of the fluids in the database. The optional status variable is set to `MFLUID_STATUS_OK` or `MFLUID_STATUS_FAIL` depending on whether the summing operation was successful.

- `query_mfluid_suminv(p, w, v, status)`

  Same as `query_mfluid_sum()`, but compute the weighted sum of the inverse of the chosen property.

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

  $$\frac{1}{\bar{A}} = \sum \frac{X_i}{A_i} a \ , \tag{10.1}$$

  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 query_mfluid_suminv(mf_prop_A, xn(:), abarinv, error)
  abar = 1.e0 / abarinv
  ```

  where `xn(:)` is an array of the mass fractions of each species in FLASH.

- `query_mfluid_sumfrc(p, w, v, status)`

  Same as `query_mfluid_sum()`, but compute the weighted sum of the chosen property divided by the total number of particles (A).

- `query_mfluid_sumsqr(p, w, v, status)`

  Same as `query_mfluid_sum()`, but compute the weighted sum of the square of the chosen property.

- `init_mfluid_db()`

  Initialize the multifluid database. If this has not been called and one of the other routines is called, that routine terminates with an error. The typical FLASH user will never need to call this, since this call is part of the `init_materials()` call in the `composition` submodule.

- `list_mfluid_db(lun)`

  List the contents of the multifluid database in a snappy table format. Output goes to the logical I/O unit indicated by the lun parameter.

10.2 Equations of state

The `eos` module implements the equation of state needed by the hydrodynamical and nuclear burning solvers. Interfaces are provided to operate on an entire block (`eos3d`), on a one-dimensional vector (`eos1d`), or for a single zone (`eos`). Additionally, these functions can be used to find the thermodynamic quantities either from the density, temperature, and composition or from the density, internal energy, and composition.

Three sub-modules are available in FLASH 2.2 and later: `gamma`, which implements a perfect-gas equation of state; `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). Full details of this equation of state are provided in Timmes & Swesty (1999).

10.2.1 Algorithm

As described above, FLASH evolves the Euler equations for compressible, inviscid flow. This system of equations must be closed by an additional equation that provides a relation between the thermodynamic quantities of the gas. This 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; considerable work can go into development of a robust and efficient EOS. While FLASH is capable of using any general equation of state, we discuss here only the routines for the three equations of state 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 gamma-law EOS consists 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. In this section we discuss the physics of these equations of state; the interfaces between the EOS routines and the codes are discussed in Sec. 10.2.2.

FLASH uses the method of Colella & Glaz (1995) to handle general equations of state. General equations of state contain 4 adiabatic indices (Chandrasekhar 1939), but the method of Colella & Glaz parameterizes the EOS and requires only two of the adiabatic indices. The first is necessary to calculate the adiabatic sound speed and is given by

$$\gamma_1 = \frac{P}{\rho}\frac{\partial P}{\partial \rho} \; . \tag{10.2}$$

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

$$\gamma_4 = 1 + \frac{P}{\rho\epsilon} \; . \tag{10.3}$$

These two adiabatic indices are stored as the variables `gamc` and `game`. All EOS routines must return γ_1, and γ_4 is calculated from eq. (10.3).

The gamma-law EOS 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

$$P = (\gamma - 1)\,\rho\epsilon \; . \tag{10.4}$$

We also have an expression relating pressure to the temperature T

$$P = \frac{N_a k}{\bar{A}}\rho T \; , \tag{10.5}$$

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

$$\frac{1}{\bar{A}} = \sum_i \frac{X_i}{A_i} \; , \tag{10.6}$$

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

$$\epsilon = \frac{1}{\gamma - 1}\frac{N_a k}{\bar{A}} T \ . \tag{10.7}$$

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

$$\frac{1}{(\gamma - 1)} = \sum_i \frac{1}{(\gamma_i - 1)}\frac{X_i}{A_i} \ . \tag{10.8}$$

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 performance, use of the gamma-law EOS is limited, due to its restricted range of applicability for astrophysical flash problems.

The Helmholtz EOS provided with the FLASH distribution contains more physics and is appropriate for addressing astrophysical phenomena in which electrons and positrons may be relativistic and/or degenerate and in which radiation may significantly contribute to the thermodynamic state. 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

$$P_{\rm tot} = P_{\rm rad} + P_{\rm ion} + P_{\rm ele} + P_{\rm pos} + P_{\rm coul} \tag{10.9}$$

$$\epsilon_{\rm tot} = \epsilon_{\rm rad} + \epsilon_{\rm ion} + \epsilon_{\rm ele} + \epsilon_{\rm pos} + \epsilon_{\rm coul} \ . \tag{10.10}$$

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

$$P_{\rm rad} = \frac{aT^4}{3} \tag{10.11}$$

$$\epsilon_{\rm rad} = \frac{3P_{\rm rad}}{\rho} \tag{10.12}$$

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 eqs. (10.4)-(10.5) 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

$$N_{\rm ele} = \frac{8\pi\sqrt{2}}{h^3}\ m_{\rm e}^3\ c^3\ \beta^{3/2}\ \left[F_{1/2}(\eta,\beta)\ +\ F_{3/2}(\eta,\beta)\right] \tag{10.13}$$

$$N_{\rm pos} = \frac{8\pi\sqrt{2}}{h^3}\ m_{\rm e}^3\ c^3\ \beta^{3/2}\left[F_{1/2}\left(-\eta - 2/\beta, \beta\right)\ +\ \beta\ F_{3/2}\left(-\eta - 2/\beta, \beta\right)\right] \ , \tag{10.14}$$

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

$$F_k(\eta,\beta) = \int_0^\infty \frac{x^k\ (1 + 0.5\ \beta\ x)^{1/2}\ dx}{\exp(x-\eta)+1} \ . \tag{10.15}$$

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

$$N_{\rm ele,matter} = \frac{\bar{Z}}{\bar{A}}\ N_a\ \rho = \bar{Z}\ N_{\rm ion} \ , \tag{10.16}$$

and charge neutrality requires

$$N_{\rm ele,matter} = N_{\rm ele} - N_{\rm pos} \ . \tag{10.17}$$

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 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,

$$F = \epsilon - T\ S \tag{10.18}$$

$$dF = -S\ dT + \frac{P}{\rho^2}\ d\rho \ , \tag{10.19}$$

is the appropriate thermodynamic potential for use when the temperature and density are the natural thermodynamic variables. The free energy table distributed with FLASH was produced from the Timmes EOS (Timmes & Arnett 1999). The Timmes EOS evaluates the Fermi-Dirac integrals (Equation 10.15) 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 eq. (10.17). 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 $Y_{\rm e} = 1$. One reason for not including contributions from photons and ions in the table is that these components of the Helmholtz EOS are very simple (eqs. (10.11)–(10.12)), 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 $Y_{\rm e} = 1$ is that the 2-dimensional table is valid for *any* composition. Separate planes for each $Y_{\rm e}$ are not necessary (or desirable), since simple multiplication by $Y_{\rm e}$ 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.

10.2.2 Usage

There are three interfaces to the EOS, reflecting the different modes in which it is used. A block interface runs the EOS on all the zones in a block, the vector interface runs the EOS on a one-dimensional vector of zones, and a pointwise interface updates the thermodynamics for a single zone. All interfaces are contained in Fortran 90 modules to provide compile time argument checking.

10.2.2.1 The block interface, `eos3d`

After each update from the hydrodynamics or burning, it is necessary to update the pressure and temperature for the entire block. The `eos3d` function is optimized for updating all of the zones in a single block. This function will always take the internal energy, density, and temperature as input and use them to find the temperature, pressure, and adiabatic indices for this thermodynamical state. This information is obtained through database calls for all of the zones in the block. `eos3d` takes three arguments

```
use ModuleEos3d

call eos3d(solnData, iblock, iflag)
```

where `solnData` is one block of data, `iblock` is the block number to operate on, and `iflag` specifies which region of the block to update. Setting `iflag` to `0` will update all of the interior zones (*i.e.* exclude guard cells). A value of `7` will update all the zones in a block (interior zones + guard cells). Values between 1 and 6 update the individual regions of guard cells (upper and lower regions in the three coordinate directions).

For some equations of state, it is necessary to perform a Newton-Raphson iteration to find the temperature and pressure corresponding to the internal energy, density, and composition, because the equation of state is more naturally state in terms of temperature and density. In these cases, `eos3d` will do the necessary root finding up to a tolerance defined in the function (typically 1×10^{-8}).

10.2.2.2 The vector interface, `eos1d`

An alternate interface to the equation of state is provided by `eos1d`. This function operates on a vector, taking density, composition, and either internal energy or temperature as input, and returning pressure, γ_1, and either the temperature or internal energy (which ever was not used as input).

In `eos1d`, all the input is taken from the argument list

```
use ModuleEos1d

call eos1d (input, kbegin, kend, rho, tmp, p, ei, gamc, xn, q, qn)
```

Here, `input` is an integer flag that specifies whether the temperature (`input = 1`) or internal energy (`input = 2`) compliment the density and composition as input. Two other integers, `kbegin` and `kend` specify the beginning and ending indices in the input vectors on which to operate. The arrays `rho`, `tmp`, `p`, `ei`, and `gamc` are of length `q`, and contain the density, temperature, pressure, internal energy, and γ_1, respectively. The array `xn(q,qn)` contains the composition (for `qn` fluids) for all of the input zones.

This equation of state interface is useful for initializing a problem. The user is given direct control over from where the input comes and where it ultimately is stored, since everything is passed through the argument list. This 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.

10.2.2.3 The point interface, `eos`

The `eos` interface provides the most information and flexibility. No assumptions about the layout of the data are made. This function simply takes density, composition, and either temperature or internal energy as input, and returns a host of thermodynamic quantities. Most of the information provided here is not provided anywhere else, such as the electron pressure, degeneracy parameter, and thermodynamic derivatives. The interface is

```
use ModuleEos

call eos(dens, temp, pres, ener, xn, abar, zbar, dpt, dpd, det, ded, &
         c_v, c_p, gammac, pel, ne, eta, input)
```

The arguments `dens`, `temp`, `pres`, and `ener` are the density, temperature, pressure, and internal energy respectively. `xn` is a vector containing the composition (the length of this vector is `ionmax`, supplied by the `common` module. `abar` and `zbar` are the average atomic mass and proton number, which are returned at the end of the call. Four thermodynamic derivatives are provided, pressure with respect to temperature (`dpt`) and density (`dpd`), and energy with respect to temperature (`det`) and density (`ded`). The specific heats at constant volume (`c_v`) and constant pressure (`c_p`) are also provided. Finally, γ_1 (`gammac`), the electron pressure (`pel`), the electron number density (`ne`), and the electron degeneracy pressure (`eta`) are also returned. The integer `input` specifies whether temperature (`input = 1`) or internal energy (`input = 2`) are used together with the density and composition as input.

10.2.2.4 Runtime parameters

There are very few runtime parameters used with these equations of state. The gamma-law EOS takes only one parameter, the value of `gamma` used to relate the internal energy and pressure (see Table 10.2).

Table 10.2: Runtime parameters used with the `eos/gamma` module.

Variable	Type	Default	Description
`gamma`	real	1.6667	Ratio of specific heats for the gas (γ)

The `helmholtz` module also takes a single runtime parameter, 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 `coulomb_mult` is set to zero, the Coulomb corrections are not applied (see Table 10.3).

Table 10.3: Runtime parameters used with the `eos/helmholtz` module.

Variable	Type	Default	Description
`coulomb_mult`	real	1.0	Multiplication factor for Coulomb corrections.

The `helmholtz` EOS requires an input file `helm_table.dat` that contains the table for the electron contributions. This table is currently ASCII for portability purposes. When the table is first read in, a binary version called `helm_table.bdat` is created. This can be used for subsequent restarts on the same machine but may not be portable across platforms.

10.3 Compositions

The `composition` module sets up the different compositions needed by FLASH. In general, there is one composition for each of the burners located in `source/source_terms/burn/` as well as a proton and electron composition used by the radiative losses module, `source/source_terms/cool/radloss`, compositions of ions of different elements used by the ionization modules in `source/source_terms/ioniz/`, and a generic fuel and ash composition. You will only need to write your own module if you wish to carry around different numbers or types of fluid than any of the predefined modules. These modules set up the names of the fluid (both a long name, recognized by the main FLASH database and a short name that can be queried through the `multifluid` database) as well as their general properties. If you use a burn module, you are required to use the corresponding composition module.

The `Config` file in each composition directory specifies the number of fluids. The general syntax is

```
NUMSPECIES 2
```

This example sets up 2 fluids. This file is read by `setup` and used to initialize the `ionmax` parameter in FLASH. This parameter is publically available in the `dBase` database and can be used to initialize arrays to the number of fluids tracked by FLASH.

Each composition directory also contains a file named `init_mat.F90` that sets the properties of each fluid. This routine is called at the start of program execution by `init_flash`. The general syntax of this file is

```
 subroutine init_materials

 use multifluid_database, ONLY: init_mfluid_db,
&                               add_fluid_to_db, n_fluids

 use common, ONLY: ionmax

 implicit none

 call init_mfluid_db (ionmax)

 if (n_fluids == 2) then

    call add_fluid_to_db ("fuel", "f", A=1., Z=1., Eb=1.)
    call add_fluid_to_db ("ash",  "a", A=1., Z=1., Eb=1.)

 else

    call abort_flash("init_mat: fuel+ash requires two fluids!")

 endif

 return
 end
```

Here we initialize the multifluid database through the call to `init_mfluid_db`. The value of `ionmax` is supplied through the `dBase` database. Next, each fluid is added to the database through the calls to `add_fluid_to_db`, specifying the full name and short name, the atomic mass, proton number, and binding energy. The atomic mass and proton numbers are used in the equations of state and are accessed via `multifluid` database calls.

The `example` setup discussed in Sec. 4.1 demonstrates how to setup a problem with two fluids (using the `fuel+ash` module). The same accessor methods that are used to store the solution data are used to store the fluid abundances in each zone.

Below we summarize the different compositions.

10.3.1 Fuel plus ash mixture (`fuel+ash`)

The `fuel+ash` composition is not directly associated with any burner but is intended for problems that wish to track two fluids, for example to study mixing.

Table 10.4: The `fuel+ash` composition.

Long name	**Short name**	**Mass**	**Charge**	**Binding energy**
fuel	f	1.	1.	1.0
ash	a	1.	1.	1.0

10.3.2 Minimal seven-isotope alpha-chain model (`iso7`)

`iso7` provides a very minimal alpha-chain, useful for problems that do not have enough memory to carry a larger set of isotopes. This is the complement to the `iso7` reaction network.

Table 10.5: The `iso7` composition.

Long name	Short name	Mass	Charge	Binding energy
helium-4	He4	4.	2.	28.29603
carbon-12	C12	12.	6.	92.16294
oxygen-16	O16	16.	8.	127.62093
neon-20	Ne20	20.	10.	160.64788
magnesium-24	Mg24	24.	12.	198.25790
silicon-28	Si28	28.	14.	236.53790
nickel-56	Ni56	56.	28.	484.00300

10.3.3 Thirteen-isotope alpha-chain model (`aprox13`)

`aprox13` is an alpha-chain composition suitable for helium or carbon burning. It includes all of the alpha elements up to ^{56}Ni and is the required composition for the `aprox13` network.

Table 10.6: The `aprox13` composition.

Long name	Short name	Mass	Charge	Binding energy
helium-4	He4	4.	2.	28.29603
carbon-12	C12	12.	6.	92.16294
oxygen-16	O16	16.	8.	127.62093
neon-20	Ne20	20.	10.	160.64788
magnesium-24	Mg24	24.	12.	198.25790
silicon-28	Si28	28.	14.	236.53790
sulfur-32	S32	32.	16.	271.78250
argon-36	Ar36	36.	18.	306.72020
calcium-40	Ca40	40.	20.	342.05680
titanium-44	Ti44	44.	22.	375.47720
chromium-48	Cr48	48.	24.	411.46900
iron-52	Fe52	52.	26.	447.70800
nickel-56	Ni56	56.	28.	484.00300

10.3.4 Nineteen-isotope alpha-chain model (`aprox19`)

`aprox19` builds on the `aprox13` alpha-chain and adds isotopes need for pp burning, CNO and hot CNO cycles, and photodisintegration. This composition module is required by the `aprox19` reaction network.

Table 10.7: The `aprox19` composition.

Long name	Short name	Mass	Charge	Binding energy
hydrogen-1	H1	1.	1.	0.00000
helium-3	He3	3.	2.	7.71819
helium-4	He4	4.	2.	28.29603
carbon-12	C12	12.	6.	92.16294
nitrogen-14	N14	14.	7.	104.65998
oxygen-16	O16	16.	8.	127.62093
neon-20	Ne20	20.	10.	160.64788
magnesium-24	Mg24	24.	12.	198.25790
silicon-28	Si28	28.	14.	236.53790
sulfur-32	S32	32.	16.	271.78250
argon-36	Ar36	36.	18.	306.72020
calcium-40	Ca40	40.	20.	342.05680
titanium-44	Ti44	44.	22.	375.47720
chromium-48	Cr48	48.	24.	411.46900
iron-52	Fe52	52.	26.	447.70800
iron-54	Fe54	54.	26.	471.76960
nickel-56	Ni56	56.	28.	484.00300
neutrons	n	1.	0.	0.00000
protons	p	1.	1.	0.00000

10.3.5 Proton-proton/CNO network model (`ppcno`)

`ppcno` is a composition group suitable for pp/CNO reactions. It is required by the `ppcno` reaction network.

Table 10.8: The `ppcno` composition.

Long name	Short name	Mass	Charge	Binding energy
hydrogen-1	H1	1.	1.	0.00000
hydrogen-2	H2	2.	1.	2.22500
helium-3	He3	3.	2.	7.71819
helium-4	He4	4.	2.	28.29603
lithium-7	Li7	7.	3.	39.24400
beryllium-7	Be7	7.	4.	37.60000
boron-8	B8	8.	5.	37.73800
carbon-12	C12	12.	6.	92.16294
carbon-13	C13	13.	6.	97.10880
nitrogen-13	N13	13.	7.	94.10640
nitrogen-14	N14	14.	7.	104.65998
nitrogen-15	N15	15.	7.	115.49320
oxygen-15	O15	15.	8.	111.95580
oxygen-16	O16	16.	8.	127.62093
oxygen-17	O17	17.	8.	131.76360
oxygen-18	O18	18.	8.	139.80800
fluorine-17	F17	17.	9.	128.22120
fluorine-18	F18	18.	9.	137.37060
fluorine-19	F19	19.	9.	147.80200

10.3.6 Proton-electron plasma composition (prot+elec)

The `prot+elec` composition is not associated with any burner but is intended for modules that need a plasma, such as the radiative losses module.

Table 10.9: The `prot+elec` composition.

Long name	**Short name**	**Mass**	**Charge**	**Binding energy**
proton	p	1.	1.	1.
electron	e	0.000544617	-1.	1.

10.3.7 Multi-ion plasma composition (ioniz)

The `ioniz` composition module provides compositions for use with the `source/source_terms/ioniz` module. The default is `ioniz/all`, which contains the ions of all of the twelve elements that the module can track. Several others include just a subset of the elements. For instance, the `ioniz/C+O+Ca+Fe` module includes just carbon, oxygen, calcium, and iron.

To use a different subset of elements than the ones included, a new module must be added. The new module must contain the implementation of a subroutine, `sct_element(idx())`, which takes an integer array the members of which control whether a given element is included in the composition. Additionally, the submodule must contain a `Config` file that includes the `NUMSPECIES` parameter set equal to the sum of all the numbers of ions of all of the elements included. Table 10.10 shows how many ions are tracked for each element. In order to decide the value of NUMSPECIES, add the numbers of ions for each element included, plus one each for hydrogen and electrons. For example, if a simulation were to include just nitrogen and carbon, then NUMSPECIES should be set to 17 = 7 (for C) + 8 (for N) + 1 (for hydrogen) + 1 (for electrons). If a simulation were to include all the elements, then NUMSPECIES would have the value 181.

Table 10.10: Number of ions for each element.

Element	Number of Ions
He	3
C	7
N	8
O	9
Ne	11
Mg	13
Si	15
S	17
Ar	19
Ca	21
Fe	27
Ni	29

10.4 Thermal conductivity

The `conductivity` sub-module implements a prescription for computing thermal conductivity coefficients used by the hydro solver. To use thermal conductivity in a FLASH simulation, the runtime parameter `diffuse_therm` must be set to `.true.` See Sec. 9.1.3 in the hydro module documentation for details on the modules and how the solver uses them.

10.4.1 Stellar thermal conductivity

Internal energy may be transported from warm regions into colder material by collisional and radiative processes. At large densities and cold temperatures, thermal transport by conduction dominates over the radiative processes. At small densities and hot temperatures, radiative processes dominate the transport of thermal energy. At intermediate densities and temperatures, both conductive and radiative processes contribute. As such, both radiative and conductive transport processes need to be considered.

FLASH provides one module for computing the opacity of stellar material (Timmes 2000; Timmes & Brown 2002). This module uses analytic fits from Iben (1975) and Christy (1966) for the radiative opacity, when all processes other than electron scattering are considered. An approximation formula from Weaver *et al.* (1978) for the Compton opacity, which includes a cutoff for frequencies less than the plasma frequency, is then added to form the total radiative opacity. Analytic fits from Iben (1975) are used for the thermal conductivity in the non-degenerate regime. In the degenerate regime, the thermal conductivity formalism of Yakovlev & Urpin (1980) is used. A smooth and continuous interpolation function joins the thermal conductivity expressions in the degenerate and non-degenerate regimes in the transition regions. Contributions from ion-electron, electron-electron, and phonon-electron scattering are summed to form the total thermal conductivity. An approximation formula for the electron-electron interaction integral $J(y)$ given by Potekhin, Chabrier & Yakovlev (1997), which is more complete than the approximation formula given by Timmes (1992), has been adopted. The radiative opacity is converted to an equivalent conductivity by $\sigma_{\rm rad} = 4ac\mathrm{T}^3/(3\rho\kappa_{\rm rad})$ before forming the total thermal conductivity.

10.4.2 Spitzer thermal conductivity

This module implements the thermal conductivity following the formulation of Spitzer (1962)

$$\sigma(X_i, \rho, T) = \kappa T^n \ , \qquad (10.20)$$

where $n = 5/2$ and $\kappa = 9.2 \times 10^{-7}$ is the plasma thermal conductivity, here assumed isotropic for simplicity.

10.5 Viscosity

The `viscosity` sub-module implements a prescription for computing viscosity coefficients used by the hydro solver. To use viscosity in a FLASH simulation, the runtime parameter `diffus_visc` must be set to `.true.`. See Sec. 9.1.3 in the hydro module documentation for details on how the solver uses these modules.

10.5.1 Spitzer viscosity

This module implements the coefficient of plasma compressional viscosity according to the classical Spitzer (1962) prescription

$$\nu = \kappa T^n \ , \qquad (10.21)$$

where $n = 5/2$ and $\kappa = 1.25 \times 10^{-16}$ (which corresponds to a Coulomb logarithm $\ln\Lambda = 20$ at typical coronal conditions).

10.6 Magnetic resistivity and viscosity

The magnetic resistivity sub-module (`source/materials/magnetic_resistivity`) provides routines that compute magnetic resistivity η and viscosity ν_m for a mixture of fully ionized gases. The default top level routines return zero values for both resistivity and viscosity. Specific routines for constant and variable resistivity are provided in `const` and `spitzer` subdirectories. By default, all routines return results in CGS units. However they provide an option to return results in SI units. The relationship between magnetic resistivity and viscosity is $\nu_m = \frac{c^2}{4\pi}\eta$ in CGS and $\nu_m = \frac{1}{\mu_0}\eta$ in SI.

10.6.1 Constant resistivity

This submodule returns constant magnetic resistivity and viscosity. The module declares two runtime variables, `resistivity` and `mvisc`, that are respectively the constant resistivity and viscosity. The default value for both variables is zero. The magnetic resistivity function reads in `resistivity` and returns it to the calling routine. The magnetic viscosity routine first checks whether `mvisc` is given a non-zero value in the parameter file. If it is, this non-zero value is returned; otherwise, the routine further reads in `resistivity` and converts it into magnetic viscosity.

10.6.2 Spitzer resistivity

This submodule implements the resistivity coefficient derived in Spitzer (1962)

$$\eta = \frac{e^2 \overline{Z} \ln \Lambda(\overline{Z})}{2 m_e \gamma_{\mathrm{E}}(\overline{Z})} \left(\frac{\pi m_e}{2kT} \right)^{\frac{3}{2}} , \tag{10.22}$$

where $\overline{Z} = \sum_i n_i Z_i^2 / \sum_i n_i Z_i$ is the average ionic charge of the plasma, $\ln \Lambda$ is the Coulomb logarithm, and γ_{E} is the correction factor that corrects the Lorentz gas resistivity to account for electron-electron collisions. In general, this factor must be computed using detailed kinetic models. We fit tabulated values given in Spitzer (1962) by

$$\gamma_{\mathrm{E}}(Z) = 0.582 + 0.418 \tanh \left(\frac{\ln Z}{2.614} \right) , \quad 1 \le Z < \infty. \tag{10.23}$$

The magnetic viscosity is computed directly from the resistivity.

Chapter 11

Local source terms

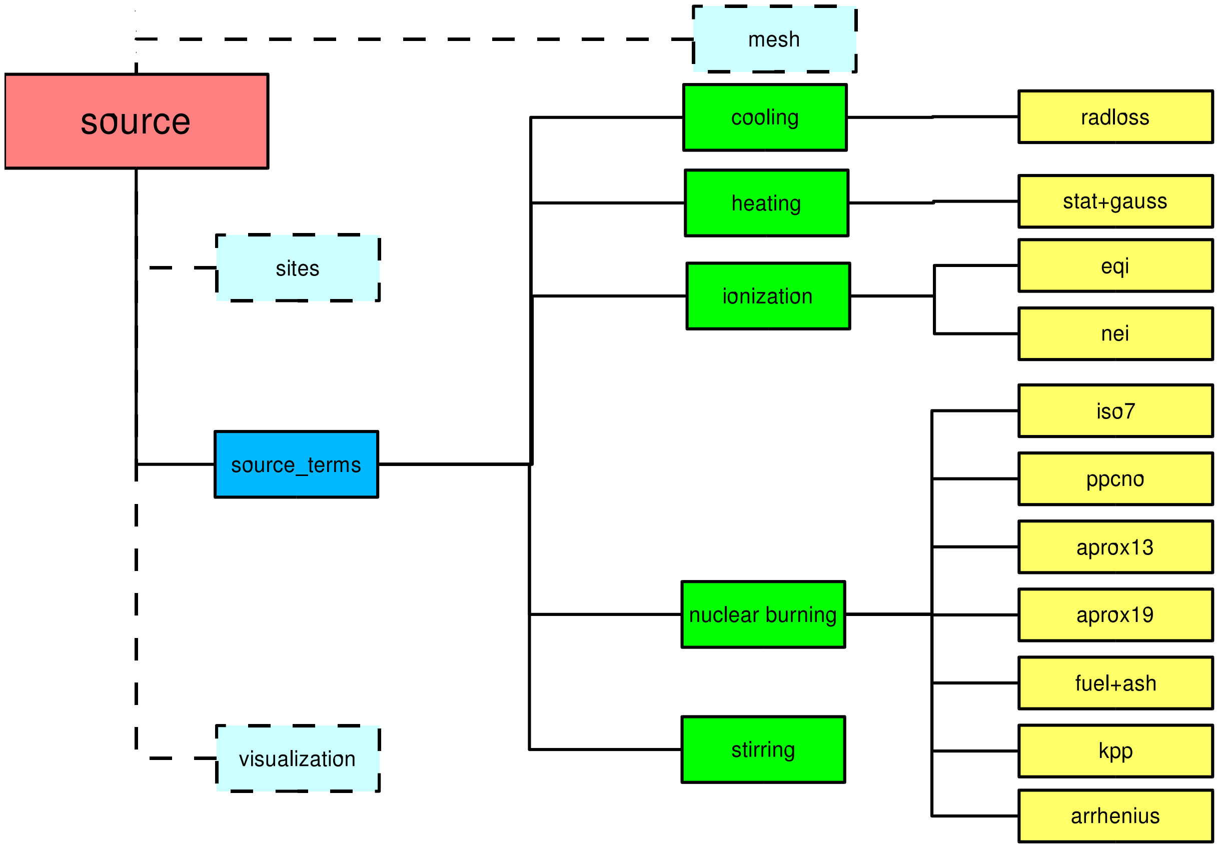


Figure 11.1: The source terms module directory.

11.1 The nuclear burning module

The nuclear burning module 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 module are `init_burn()`, which calls routines to set up the nuclear isotope tables needed by the module, and `burn()`, which calls the ODE solver and updates the hydrody-

Table 11.1: Runtime parameters used with the `burn` module.

Variable	Type	Default	Description
`tnucmin`	real	1.1×10^8	Minimum temperature in K for burning to be allowed
`tnucmax`	real	1.0×10^{12}	Maximum temperature in K for burning to be allowed
`dnucmin`	real	1.0×10^{-10}	Minimum density (g cm^{-3}) for burning to be allowed
`dnucmax`	real	1.0×10^{14}	Maximum density (g cm^{-3}) for burning to be allowed
`ni56max`	real	0.4	Maximum Ni^{56} mass fraction for burning to be allowed

namical variables in a single row of a single AMR block.

11.1.1 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 there 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 module includes a multidimensional shock detection algorithm that can be used to prevent burning in shocks. If the `shock_burning` parameter is set to `.false.`, this algorithm is used to detect shocks in the `burn_block` function and to switch off the burning in shocked zones.

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 zone 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.* `shock_burning = .false.`).

11.1.2 Algorithms

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

A user can choose between two implicit integration methods and two linear algebra packages in FLASH. The runtime parameter `ode_steper` controls which integration method is used in the simulation. The choice `ode_steper = 1` is the default and invokes a Bader-Deuflhard scheme. The choice `ode_steper = 2` invokes a Kaps-Rentrop scheme. The runtime parameter `algebra` controls which linear algebra package is used in the simulation. The choice `algebra = 1` is the default and invokes the sparse matrix MA28 package. The choice `algebra = 2` invokes the GIFT linear algebra routines. While any combination of the integration methods and linear algebra packages will produce correct answers, some combinations may execute more efficiently than other combinations for certain types of simulations. No general rules have been found for which combination is the best for a given simulation. Which combination is the most efficient depends on the timestep being taken, the spatial resolution of the model, the values of the local thermodynamic variables, and the composition. Experiment with the various combinations!

Timmes (1999) reviewed several methods for solving stiff nuclear reaction networks, providing the basis for the reaction network solvers included with FLASH. The scaling properties and behavior of three semi-implicit time integration algorithms (a traditional first-order accurate Euler method, a fourth-order accurate Kaps-Rentrop method, and a variable order Bader-Deuflhard method) and eight linear algebra packages (LAPACK, LUDCMP, LEQS, GIFT, MA28, UMFPACK, and Y12M) were investigated by running each of these 24 combinations on seven different nuclear reaction networks (hard-wired 13- and 19-isotope networks and soft-wired networks of 47, 76, 127, 200, and 489 isotopes). Timmes' analysis suggested that the best balance of accuracy, overall efficiency, memory footprint, and ease-of-use was provided by the two integration methods (Bader-Deuflhard and Kaps-Rentrop) and the two linear algebra packages (MA28 and GIFT) that are provided with the FLASH code.

11.1.2.1 Reaction networks

We begin by describing the equations solved by the nuclear burning module. 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

$$X_i = \rho_i/\rho = n_i A_i/(\rho N_A) \ , \tag{11.1}$$

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

$$Y_i = X_i/A_i = n_i/(\rho N_A) \ . \tag{11.2}$$

Mass conservation is then expressed by

$$\sum_{i=1}^{N} X_i = 1 \ . \tag{11.3}$$

At the end of each timestep, FLASH checks that the stored abundances satisfy eq. (11.3) to machine precision in order to avoid the unphysical buildup (or decay) of the abundances or energy generation rate. Roundoff errors in this equation can lead to significant problems in some contexts (*e.g.*, classical nova envelopes), where trace abundances are important.

The general continuity equation for the ith isotope is given in Lagrangian formulation by

$$\frac{dY_i}{dt} + \nabla \cdot (Y_i \mathbf{V}_i) = \dot{R}_i \ . \tag{11.4}$$

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

$$\dot{R}_i = \sum_{j,k} Y_l Y_k \lambda_{kj}(l) - Y_i Y_j \lambda_{jk}(i) \ , \tag{11.5}$$

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 eq. (11.5). The mass diffusion velocities $\mathbf{V}_i$ in eq. (11.4) are obtained from the solution of a multicomponent diffusion equation (Chapman & Cowling 1970; Burgers 1969; Williams 1988) and reflect the fact that mass diffusion processes arise from pressure, temperature, and/or abundance gradients as well as from external gravitational or electrical forces.

The case $\mathbf{V}_i \equiv 0$ is important for two reasons. First, mass diffusion is often unimportant when compared to other transport processes, such as thermal or viscous diffusion (*i.e.*, large Lewis numbers and/or small Prandtl numbers). Such a situation obtains, for example, in the study of laminar flame fronts propagating through the quiescent interior of a white dwarf. Second, this case permits the decoupling of the reaction network solver from the hydrodynamical solver through the use of operator splitting, greatly simplifying the algorithm. This is the method used by the default FLASH distribution. Setting $\mathbf{V}_i \equiv 0$ transforms eq. (11.4) into

$$\frac{dY_i}{dt} = \dot{R}_i \ , \tag{11.6}$$

which may be written in the more compact, standard form

$$\dot{\mathbf{y}} = \mathbf{f}\ (\mathbf{y})\ . \tag{11.7}$$

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 eqs. (11.6) and (11.7) can vary quite significantly. As a result, the nuclear reaction network equations are "stiff." A system of equations is stiff when the ratio of the maximum to the minimum eigenvalue of the Jacobian matrix $\tilde{\mathbf{J}} \equiv \partial\mathbf{f}/\partial\mathbf{y}$ is large and imaginary. This means that at least one of the isotopic abundances changes on a much shorter timescale than another. Implicit or semi-implicit time integration methods are generally necessary to avoid following this short-timescale behavior, requiring the calculation of the Jacobian matrix.

It is instructive at this point to look at an example of how Eq. (11.6) 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 eq. (11.7) through the terms

$$\begin{array}{ll} \dot{Y}(^4He) & = -Y(^4He)\ Y(^{12}C)\ R + \ldots \\ \dot{Y}(^{12}C) & = -Y(^4He)\ Y(^{12}C)\ R\ + \ldots \\ \dot{Y}(^{16}O) & = +Y(^4He)\ Y(^{12}C)\ R\ + \ldots \end{array} \quad , \tag{11.8}$$

where the ellipsis 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 $\tilde{\mathbf{J}}$=$\partial\mathbf{f}/\partial\mathbf{y}$

$$\begin{array}{ll} J(^4He,^4He) = -Y(^{12}C)\ R\ + \ldots & J(^4He,^{12}C) = -Y(^4He)\ R\ + \ldots \\ J(^{12}C,^4He) = -Y(^{12}C)\ R\ + \ldots & J(^{12}C,^{12}C) = -Y(^4He)\ R\ + \ldots \\ J(^{16}O,^4He) = +Y(^{12}C)\ R\ + \ldots & J(^{16}O,^{12}C) = +Y(^4He)\ R\ + \ldots \end{array} \quad . \tag{11.9}$$

Entries in the Jacobian matrix represent the flow, in number of nuclei s^{-1}, 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.

The FLASH code distribution includes several 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 `ppcno` network includes reactions for the pp and CNO cycles. A number of simple single-reaction networks are also provided. All the networks supplied with FLASH are examples of "hard-wired" reaction networks, 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.

11.1.2.2 Two linear algebra packages

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

The MA28 sparse matrix package used by FLASH is described by Duff, Erisman, & Reid (1986). This package, which has been described as the "Coke classic" of sparse linear algebra packages, uses a direct – as opposed to an iterative – method for solving linear systems. Direct methods typically divide the solution of $\tilde{\mathbf{A}} \cdot \mathbf{x} = \mathbf{b}$ 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 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 $\sim 5.0\times10^7$ lines of code! Fortunately, for small reaction networks (less than about 30 isotopes), GIFT generated routines are much smaller and generally faster than other linear algebra packages.

As discussed above, but which bears repeating, the FLASH runtime parameter `algebra` controls which linear algebra package is used in the simulation. `algebra = 1` is the default choice and invokes the sparse matrix MA28 package. `algebra = 2` invokes the GIFT linear algebra routines.

11.1.2.3 Two time integration methods

One of the time integration methods used by FLASH for evolving the reaction networks is a 4th-order accurate Kaps-Rentrop method. In essence, this method is an implicit Runge-Kutta algorithm. The reaction network is advanced over a timestep h according to

$$\mathbf{y}^{n+1} = \mathbf{y}^n + \sum_{i=1}^{4} b_i \Delta_i \ , \tag{11.10}$$

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

$$\begin{aligned}
(\tilde{\mathbf{1}}/\gamma h - \tilde{\mathbf{J}}) \cdot \Delta_1 &= \mathbf{f}(\mathbf{y}^n) & (11.11)\\
(\tilde{\mathbf{1}}/\gamma h - \tilde{\mathbf{J}}) \cdot \Delta_2 &= \mathbf{f}(\mathbf{y}^n + a_{21}\Delta_1) + c_{21}\Delta_1/h & (11.12)\\
(\tilde{\mathbf{1}}/\gamma h - \tilde{\mathbf{J}}) \cdot \Delta_3 &= \mathbf{f}(\mathbf{y}^n + a_{31}\Delta_1 + a_{32}\Delta_2) + (c_{31}\Delta_1 + c_{32}\Delta_2)/h & (11.13)\\
(\tilde{\mathbf{1}}^{\gamma} h - \tilde{\mathbf{J}}) \cdot \Delta_4 &= \mathbf{f}(\mathbf{y}^n + a_{31}\Delta_1 + a_{32}\Delta_2) + (c_{41}\Delta_1 + c_{42}\Delta_2 + c_{43}\Delta_3)/h \ . & (11.14)
\end{aligned}$$

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 makes use of the routine GRK4T given by Kaps & Rentrop (1979).

Another time integration method used by FLASH for evolving the reaction networks is the variable order Bader-Deuflhard method (*e.g.*, Bader & Deuflhard 1983). The reaction network is advanced over a large timestep H from $\mathbf{y}^n$ to $\mathbf{y}^{n+1}$ by the following sequence of matrix equations. First,

$$\begin{aligned}
h &= H/m \\
(\tilde{\mathbf{1}} - \tilde{\mathbf{J}}) \cdot \Delta_0 &= h\mathbf{f}(\mathbf{y}^n) \\
\mathbf{y}_1 &= \mathbf{y}^n + \Delta_0 \ .
\end{aligned} \tag{11.15}$$

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

$$\begin{aligned}
(\tilde{\mathbf{1}} - \tilde{\mathbf{J}}) \cdot \mathbf{x} &= h\mathbf{f}(\mathbf{y}_k) - \Delta_{k-1} \\
\Delta_k &= \Delta_{k-1} + 2\mathbf{x} \\
\mathbf{y}_{k+1} &= \mathbf{y}_k + \Delta_k \ ,
\end{aligned} \tag{11.16}$$

and closure is obtained by the last stage

$$\begin{aligned}
(\tilde{\mathbf{1}} - \tilde{\mathbf{J}}) \cdot \Delta_m &= h[\mathbf{f}(\mathbf{y}_m) - \Delta_{m-1}] \\
\mathbf{y}^{n+1} &= \mathbf{y}_m + \Delta_m \ .
\end{aligned} \tag{11.17}$$

This staged sequence of matrix equations is executed at least twice with $m = 2$ and $m = 6$, yielding a fifth-order method. The sequence may be executed a maximum of seven times, which yields a fifteenth-order method. The exact number of times the staged sequence is executed depends on the accuracy requirements (set to one part in 10^6 in FLASH) and the smoothness of the solution. Estimates of the accuracy of an integration step are made by comparing the solutions derived from different orders. The minimum cost of this method — which applies for a single timestep that met or exceeded the specified integration accuracy — is one Jacobian evaluation, eight evaluations of the right-hand side, two matrix decompositions, and ten backsubstitutions. This minimum cost can be increased at a rate of one decomposition (the expensive part) and m backsubstitutions (the inexpensive part) for every increase in the order $2k+1$. The cost of increasing the order is compensated for, hopefully, by being able to take correspondingly larger (but accurate) timestep. The controls for order versus step size are a built-in part of the Bader-Deuflhard method. The cost per step of this integration method is at least twice as large as the cost per step of either a traditional first-order accurate Euler method or the fourth-order accurate Kaps-Rentrop discussed above. However, if the Bader-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 eqs. (11.15) – (11.17), not all of the right-hand sides are known in advance, since the sequence of linear equations is staged. This staging feature of the integration method may make some matrix packages, such as MA28, a more efficient choice.

As discussed above, but which bears repeating, the FLASH runtime parameter `ode_steper` controls which integration method is used in the simulation. The choice `ode_steper = 1` is the default and invokes the variable order Bader-Deuflhard scheme. The choice `ode_steper = 2` invokes the fourth order Kaps-Rentrop scheme.

11.1.3 Energy generation rates and reaction rates

The instantaneous energy generation rate is given by the sum

$$\dot{\epsilon}_{\rm nuc} = N_A \sum_i \frac{dY_i}{dt} \ . \tag{11.18}$$

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

$$\dot{\epsilon}_{\rm nuc} = N_A \sum_i \frac{\Delta Y_i}{\Delta t} \ . \tag{11.19}$$

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

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

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

11.1.3.1 Temperature-based timestep limiting

The hydrodynamics methods implemented in FLASH are explicit, so 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 module is included in FLASH. This constraint does not allow any information to travel more than one computational zone per timestep. The timestep is

$$\Delta t = C \cdot min\left\{\frac{dx}{|v_x|+c_s}, \frac{dy}{|v_y|+c_s}, \frac{dz}{|v_z|+c_s}\right\} , \tag{11.20}$$

computed over all zones. The Courant number C is a prefactor that is set at runtime through the `cfl` parameter and is required to be less than 1.

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 zone may exceed the existing internal energy in that zone. When this happens, the two operations (hydrodynamics and nuclear burning) become decoupled.

To help fix this problem, it is sometimes useful to step along at a timestep determined by the change in temperature in a zone. FLASH includes a temperature based timestep limiter that tries to constrain the change in temperature in a zone to be less than a user defined parameter. To use this limiter, set `itemp_limit = 1` and specify the fractional temperature change `temp_factor` you are willing to tolerate. While there is no guarantee that the temperature change will be smaller than this, since the timestep was already taken by the time this was computed, this method is successful in restoring coupling between the hydrodynamics and burning operators. This timestep will be computed as

$$\Delta t = \texttt{temp\_factor} \cdot \frac{T}{\Delta T} \cdot \Delta t_{old} \; , \tag{11.21}$$

where ΔT is the difference in the temperature of a zone from one timestep to the next, and Δt_{old} is the last timestep. To prevent the timestep from varying wildly from one step to the next, it is useful to force the maximum change in timestep to be a small factor over the previous one through the `tstep_change_factor` parameter.

11.2 Ionization

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

In light of the above observations, FLASH contains the module `source_terms/ioniz/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 module.

The Euler equations plus the set of advection equations for all the ion species take the following form

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0 \tag{11.22}$$

$$\frac{\partial \rho \mathbf{v}}{\partial t} + \nabla \cdot (\rho \mathbf{v}\mathbf{v}) + \nabla P = \rho \mathbf{g} \tag{11.23}$$

$$\frac{\partial \rho E}{\partial t} + \nabla \cdot [(\rho E + P)\,\mathbf{v}] = \rho \mathbf{v} \cdot \mathbf{g} \; [\, + \, S \,] \tag{11.24}$$

$$\frac{\partial n_i^Z}{\partial t} + \nabla \cdot n_i^Z \mathbf{v} = R_i^Z \quad (i = 1, \ldots, N_{spec}) \; , \tag{11.25}$$

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 the ion i of the element Z, N_{spec} is the total number of species, and

$$R_i^Z = N_e[n_{i+1}^Z \alpha_{i-1}^Z + n_{i-1}^Z S_{i-1}^Z - n_i^Z(\alpha_i^Z + S_i^Z)] \; , \tag{11.26}$$

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

11.2.1 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 eqs. (11.22) - (11.25) are solved using the FLASH hydro solver with $R = 0$. After each transport step, the "stiff" system of ordinary differential equations for the NEI problem

$$\frac{\partial n_i^Z}{\partial t} = R_i^Z \;\; (i = 1, \ldots, N_{spec}) \tag{11.27}$$

Table 11.2: Runtime parameters used with the `source_terms/ioniz` module.

Variable	Type	Default	Description
`iioniz`	integer	0	Do we turn on ionization for this run? 1=yes, 0=no
`tneimin`	real	1.0×10^4	Min nei temperature
`tneimax`	real	1.0×10^7	Max nei temperature
`dneimin`	real	1.0	Min nei electron number density
`dneimax`	real	1.0×10^{12}	Max nei electron number density

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

Note that in the present version, the contribution of the ionization and recombination to the energy equation (the bracketed term in eq. (11.22)) is not accounted for. Also, it should be noted that the source term in the NEI module 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.

11.2.2 Usage

In order to run a FLASH executable that uses the ionization module, 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 can be found in `source/source_terms/ioniz/` in the FLASH distribution. Typically, after `setup` is run and the executable is built, this file will have to be copied to the location of the executable before it is run.

The `source_terms/ioniz` module supplies the runtime parameters described in Table 11.2. The parameter `iioniz` must be set to 1 in order for the ionization computations to take place.

There are two submodules of `source_terms/ioniz`: the default module, `source_terms/ioniz/nei`, and `source_terms/ioniz/eqi`. The former computes ion species for non-equilibrium ionization, and the latter computes ion species in the approximation of ionization equilibrium.

The `source_terms/ioniz` module requires that the module `materials/composition/ioniz` be used. This module sets up the ion species of the fluid. There are several submodules that include all or a subset of the possible elements and the ions of those elements. `materials/composition/ioniz/all` includes all of the elements and is the default submodule. `materials/composition/C+O+Ca+Fe` includes carbon, oxygen, calcium and iron. To use the `ioniz` module with a different subset of elements, a new submodule should be added to the compositions directory. Sec. 10.3.7 describes how to create a new composition.

11.3 Stirring

The addition of driving terms in a hydrodynamical simulation can be a useful feature, for example, for generating turbulent flows or for simulating the addition of power on larger scales (*e.g.* supernova feedback into the interstellar medium). The `stirring` module directly adds a divergence-free, time-correlated 'stirring' velocity at selected modes in the simulation.

The time-correlation is important for modeling realistic driving forces. Most large-scale driving forces are time-correlated, rather than white-noise; for instance, turbulent stirring from larger scales will be correlated on timescales related to the lifetime of an eddy on the scale of the simulation domain. This time correlation will lead to coherent structures in the simulation that will be absent with white-noise driving.

For each mode at each timestep, six separate phases (real and imaginary in each of the three spatial dimensions) are evolved by an Ornstein-Uhlenbeck (OU) random process. The OU process is a zero-mean

Table 11.3: Runtime parameters used with the `source_terms/stirring` module.

Variable	Type	Default	Description
`istir`	integer	1	Do we 'turn on' stirring for this run? 1=yes, 0=no.
`st_seed`	integer	2	Seed for the random number generator.
`st_energy`	real	.01	(RMS) specific energy/time/mode stirred in.
`st_decay`	real	.1	Decay time for OU random numbers; correlation time of the stirring.
`st_stirmax`	real	62.8	wavenumber corresponding to the smallest physical scale that stirring will occur on.
`st_stirmin`	real	31.4	wavenumber corresponding to the largest scale that stirring will occur on.

process, which at each step 'decays' the previous value by an exponential $e^{(\frac{\Delta t}{\tau})}$ and adds a Gaussian random variable with a given variance. Since the OU process represents a velocity, the variance is chosen to be the square root of the specific energy input rate (set by the runtime parameter `st_energy`) divided by the decay time τ (`st_decay`).

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.*, actually doing the sum of sin and cos terms. Since most drivings will involve a fairly small number of modes, this is more efficient than an FFT, since the FFT would involve large numbers of modes (equal to six times the number of cells in the domain), the vast majority of which would have zero amplitude.

11.4 Heating

11.4.1 Static + Gaussian heating

The `source_terms/heat/stat+gauss` module implements a phenomenological heating term of the plasma parameterized as a function of position and time. The specific implementation assumes that the heating function consists of the sum of two terms – a steady, uniform term Q_0 and a transient heating $Q_i(\mathbf{s},t)$, prescribed as a separable function of spatial coordinates and time

$$Q_i(\mathbf{s},t) = H_0 \times g(\mathbf{s}) \times f(t) \ . \tag{11.28}$$

where H_0 is the peak value of the heating rate, $g(\mathbf{s})$ is the distribution along the spatial coordinate $\mathbf{s} \equiv [x,y,z]$, in our case a 3-D Gaussian function,

$$g(\mathbf{s}) = \exp[-(\mathbf{s}-\mathbf{s}_0)^2/2\sigma^2] \ , \tag{11.29}$$

and f(t) is prescribed as a step function of time followed by an exponential decrease, *i.e.*,

$$f(t) = \begin{cases} 0, & t \le t^* \\ 1, & t^* < t \le t_0 \\ \exp[(t_0-t)/\tau], & t > t_0 \end{cases} \ , \tag{11.30}$$

where t^* is the beginning of the impulsive heating phase.

11.4.2 Usage

The runtime parameters used with the `stat+gauss` module are summarized in Table 11.4.

Table 11.4: Runtime parameters for the stat+gauss module.

Variable	Type	Default	Description
statheat	real	1.0×10^{-5}	Stationary heating (erg cm$^{-3}s^{-1}$)
qheat	real	0.0	Peak value of the transient heating rate (erg cm^{-3} s^{-1})
x0heat	real	1.0	X location (cm) of the transient heating
y0heat	real	1.0	Y location (cm) of the transient heating
z0heat	real	1.0	Z location (cm) of the transient heating
sigheat	real	1.0	Sigma (cm) of the transient heating
tstar	real	-1.0	Time (s) of beginning of the impulsive heating phase
t0heat	real	-1.0	Switch off time (s) of the transient heating
tau	real	1.0	Decay time (s) of the transient heating
theatmin	real	1.0×10^3	Minimum temperature (K) allowed in the stat+gauss module

11.5 Cooling

11.5.1 Radiative losses from an optically thin plasma

The source_terms/cool/radloss module implements radiative losses from an optically thin plasma. The radiative losses per unit emission $\Lambda(T)$ measured from an optically thin plasma (Raymond and Smith, 1977, Raymond 1978) have been implemented adopting a piecewise-power law approximation that provides a reasonable fit to $\Lambda(T)$. The expression adopted is given by Rosner, Tucker and Vaiana (1978) to model the energy losses from the transition region and corona in the temperature range

$$2 \times 10^4 < T < 10^8 K \tag{11.31}$$

and by Peres *et al.* (1982) to model the energy losses from the chromosphere in the range

$$4.44 \times 10^3 < T < 2 \times 10^4 K\ . \tag{11.32}$$

The resulting formulation is as follows:

$$\Lambda(T) = \begin{cases} (10^{-5.97}T)^{11.7} & 10^{3.65}\ \mathrm{K} < T < 10^{3.9}\ \mathrm{K} \\ (10^{-7.85}T)^{6.15} & 10^{3.9}\ \mathrm{K} < T < 10^{4.3}\ \mathrm{K} \\ 10^{-21.85} & 10^{4.3}\ \mathrm{K} < T < 10^{4.6}\ \mathrm{K} \\ 10^{-31}T^2 & 10^{4.6}\ \mathrm{K} < T < 10^{4.9}\ \mathrm{K} \\ 10^{-21.2} & 10^{4.9}\ \mathrm{K} < T < 10^{5.4}\ \mathrm{K} \\ 10^{-10.4}T^{-2} & 10^{5.4}\ \mathrm{K} < T < 10^{5.75}\ \mathrm{K} \\ 10^{-21.94} & 10^{5.75}\ \mathrm{K} < T < 10^{6.3}\ \mathrm{K} \\ 10^{-17.73}T^{-2/3} & 10^{6.3}\ \mathrm{K} < T < 10^{7}\ \mathrm{K} \\ 10^{-18.21}T^{-0.6} & 10^{7}\ \mathrm{K} < T < 10^{7.6}\ \mathrm{K} \\ 10^{-26.57}T^{1/2} & 10^{7.6}\ \mathrm{K} < T < 10^{8}\ \mathrm{K} \end{cases} \tag{11.33}$$

11.5.2 Usage

The runtime parameters used with the radloss module are summarized in Table 11.5. The module requires the use of a fluid composition containing at least protons and electrons. The composition module materials/composition/prot+elec and the modules in materials/composition/ioniz satisfy this condition.

Table 11.5: Runtime parameters used with the `radloss` module.

Variable	Type	Default	Description
`tradmin`	REAL	4.44×10^3	Minimum temperature (K) allowed in the `radloss` module
`tradmax`	REAL	1.1×10^8	Maximum temperature (K) allowed in the module
`dradmin`	REAL	1.0	Minimum electron number density (cm^{-3}) allowed in the module
`dradmax`	REAL	1.0×10^{14}	Maximum electron number density (cm^{-3}) allowed in the module

Chapter 12

Gravity module

12.1 Algorithms

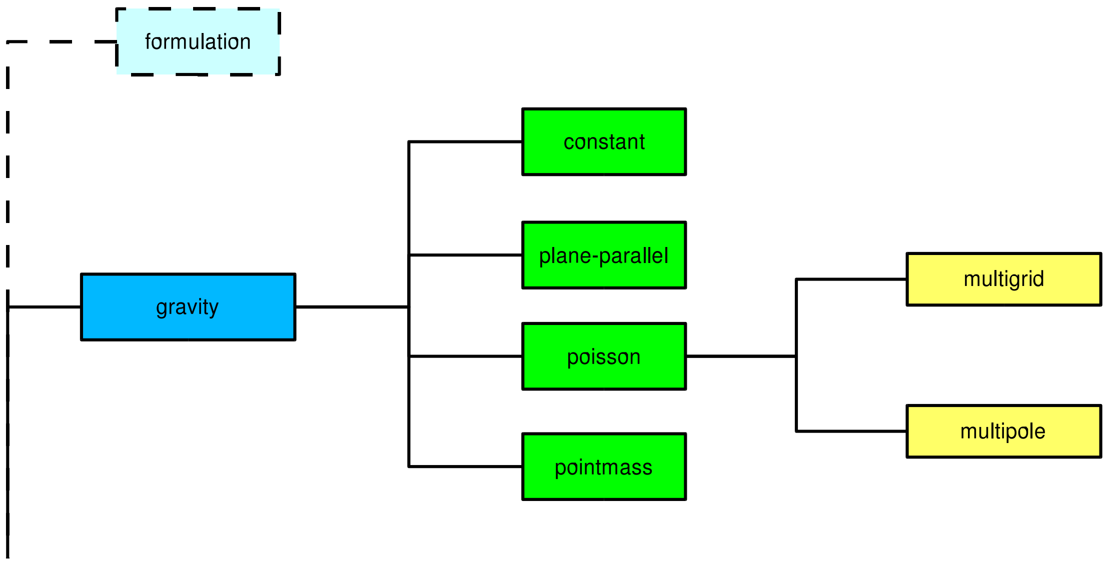


Figure 12.1: The gravity module directory.

The gravity module supplied with FLASH 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,

$$\mathbf{g}(\mathbf{x}) = -\nabla\phi(\mathbf{x}) \ . \tag{12.1}$$

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

$$\nabla^2\phi(\mathbf{x}) = 4\pi G\rho(\mathbf{x}) \ , \tag{12.2}$$

where G is Newton's gravitational constant. In the latter case, either periodic or isolated boundary conditions can be applied. In this section we describe the different external field modules distributed with FLASH, followed by two algorithms for solving the Poisson equation. Coupling of gravity to other modules (*e.g.*, hydrodynamics) is the responsibility of those other modules, but we also discuss here the gravitational coupling method used by the PPM hydrodynamics module distributed with FLASH.

12.1.1 Externally applied fields

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

1. *Constant gravitational field.* The gravitational acceleration is spatially constant and oriented along one of the coordinate axes.

2. *Plane-parallel gravitational field.* 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.

3. *Gravitational field of a point mass.* The acceleration falls off with the square of the distance from a given point. The acceleration vector is everywhere directed toward this point.

12.1.2 Self-gravity algorithms

The self-gravity algorithms supplied with FLASH solve the Poisson equation (12.2) for the gravitational potential $\phi(\mathbf{x})$. The modules implementing these algorithms can also return the acceleration field $\mathbf{g}(\mathbf{x})$; this is computed by finite-differencing the potential using the expressions

$$
\begin{aligned}
g_{x;ijk} &= \frac{1}{2\Delta x}\left(\phi_{i-1,j,k} - \phi_{i+1,j,k}\right) + \mathcal{O}(\Delta x^2) \\
g_{y;ijk} &= \frac{1}{2\Delta y}\left(\phi_{i,j-1,k} - \phi_{i,j+1,k}\right) + \mathcal{O}(\Delta y^2) \\
g_{z;ijk} &= \frac{1}{2\Delta z}\left(\phi_{i,j,k-1} - \phi_{i,j,k+1}\right) + \mathcal{O}(\Delta z^2)\ .
\end{aligned}
\tag{12.3}
$$

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.

See Chapter 15 for information regarding the Poisson solvers that can be used for gravity. Currently multigrid- and multipole-based solvers are available. Each Poisson solver has a gravity sub-module interface; for example, `gravity/poisson/multipole` requires `solvers/poisson/multipole`.

12.1.2.1 Multipole Poisson solver

The multipole Poisson solver is appropriate for spherical or nearly-spherical mass distributions with isolated boundary conditions.

12.1.2.2 Multigrid Poisson solver

The multigrid Poisson solver is appropriate for general mass distributions and can solve problems with more general boundary conditions. The algorithm distributed with FLASH is based on a multilevel refinement scheme described by Martin and Cartwright (1996). Isolated boundary conditions are implemented via a method based on James' (1978) algorithm.

12.1.3 Coupling of 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

$$\rho E \equiv \frac{1}{2}\rho v^2 + \rho\epsilon\ , \tag{12.4}$$

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, in FLASH the gravity module supplies ϕ and $\mathbf{g}$ and leaves the implementation of the fluid coupling to the hydrodynamics module. Finite-difference and finite-volume hydro schemes apply the source terms in their advection steps, sometimes at multiple intermediate timesteps and sometimes using staggered meshes for vector quantities like $\mathbf{v}$ and $\mathbf{g}$. For example, the PPM algorithm supplied with FLASH uses the following update steps to obtain the momentum and energy in zone i at timestep $n+1$

$$\begin{aligned}(\rho v)_i^{n+1} &= (\rho v)_i^n + \frac{\Delta t}{2} g_i^{n+1}\left(\rho_i^n + \rho_i^{n+1}\right) \\ (\rho E)_i^{n+1} &= (\rho E)_i^n + \frac{\Delta t}{4} g_i^{n+1}\left(\rho_i^n + \rho_i^{n+1}\right)\left(v_i^n + v_i^{n+1}\right) .\end{aligned} \tag{12.5}$$

Here g_i^{n+1} is obtained by extrapolation from ϕ_i^{n-1} and ϕ_i^n. The `poisson` gravity sub-module supplies a variable to contain the potential from the previous timestep; future releases of FLASH will likely permit the storage of several time levels of this quantity for hydrodynamics algorithms that require more steps. Currently, $\mathbf{g}$ is computed at zone centers, but this too is likely to be generalized as FLASH begins to support alternative discretization strategies. 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 (g_i^n) is fitted by interpolants along with the other state variables, and these interpolants are used to construct characteristic-averaged values of g in each zone. 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

$$\begin{aligned}v_{L,i+1/2} &\rightarrow v_{L,i+1/2} + \frac{\Delta t}{4}\left(g_{L,i+1/2}^{+} + g_{L,i+1/2}^{-}\right) \\ v_{R,i+1/2} &\rightarrow v_{R,i+1/2} + \frac{\Delta t}{4}\left(g_{R,i+1/2}^{+} + g_{R,i+1/2}^{-}\right) .\end{aligned} \tag{12.6}$$

Here $g_{X,i+1/2}^{\pm}$ is the acceleration averaged using the interpolant on the X side of the interface ($X = L, R$) for $v \pm c$ characteristics, which bring material to the interface between zones i and $i+1$ during the timestep.

12.2 Using the gravity modules

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

`INCLUDE gravity/`*sub-module[/algorithm]*

in your `Modules` file when you configure the code with `setup`. The available *sub-modules* include `constant`, `planepar`, `poisson`, and `ptmass`. If you are using the Poisson solver to compute the gravitational field, you may also specify an *algorithm*, currently `multipole` or `multigrid`. In this case you should also include the line

`INCLUDE solvers/poisson`*[/algorithm]*

in your `Modules` file. The function and usage of each of the gravity sub-modules are described in the following sections.

Note that to use any of the gravitational field routines in your code, you must use-associate the module `Gravity`. Most users will be concerned only with the following routines supplied by `Gravity`:

- `GravPotentialAllBlocks()`
  Computes the gravitational potential on the entire mesh. For the externally imposed field sub-modules, this currently does nothing. For `poisson`, it calls the Poisson solver using the solution variable `dens` as the source of the field. The potential is left in the solution variable `gpot`, and the previous contents of `gpot` are copied to `gpol`.

- `GravAccelAllBlocks(igpot, igrav, component)`
  Computes the gravitational acceleration on the entire mesh. The arguments accepted are

`igpot`	(integer)	Key number for the solution variable to use as the potential
`igrav`	(integer)	Key number for the solution variable to hold the acceleration
`component`	(integer)	Which component of the acceleration to compute ($1 = x$, $2 = y$, $3 = z$)

- `GravAccelOneBlock(igpot, igrav, component, block)`
  Computes the gravitational acceleration on a single block. Arguments are exactly the same as for `GravAccelAllBlocks()`, with the addition of the integer argument `block` specifying which block to update.

- `GravAccelOneLevel(igpot, igrav, component, level)`
  Computes the gravitational acceleration on a single refinement level. Arguments are the same as for `GravAccelAllBlocks()`, with the addition of the integer argument `level` specifying which level to update.

- `GravAccelOneRow(j, k, xyzswp, block_no, ivar, grav, nzn8)`
  Computes the gravitational acceleration for a row of zones in a specified direction in a given block. The arguments accepted by `GravAccelOneRow()` are:

`j, k`	(integer)	Row indices transverse to the sweep direction
`xyzswp`	(integer)	The sweep direction (`sweep_x`, `sweep_y`, `sweep_z`)
`block_no`	(integer)	The local block identifier
`ivar`	(integer)	The solution variable database key to use as the potential, if applicable.
`grav(:)`	(real)	Array to receive the component of the acceleration parallel to the sweep direction
`nzn8`	(integer)	The number of zones to update in `grav()`

12.2.1 Constant

The `constant` sub-module implements a spatially and temporally constant gravitational field parallel to one of the coordinate axes. The magnitude and direction of this field are set at runtime using the parameters listed in Table 12.1.

Table 12.1: Parameters for the `constant` gravity sub-module.

Variable	Type	Default	Description
`gconst`	real	-981	Gravitational acceleration
`gdirec`	string	“x”	Direction of acceleration vector (“x”, “y”, “z”)

12.2.2 Plane parallel

The `planepar` sub-module 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 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, so that the angular variation of the field direction may be ignored. The `planepar` field is cheaper to compute than the `ptmass` field, since no fractional powers of the distance are required. The runtime parameters describing this field are listed in Table 12.2.

Table 12.2: Runtime parameters used with the `planepar` gravity sub-module.

Variable	Type	Default	Description
`ptmass`	real	$1. \times 10^4$	Mass of field source
`ptxpos`	real	1	Position of field source in direction `ptdirn`
`gravsoft`	real	0.0001	Gravitational softening length
`ptdirn`	integer	1	Direction of acceleration vector ($1 = x$, $2 = y$, $3 = z$)

12.2.3 Point mass

The `ptmass` sub-module implements the gravitational field due to a point mass at a fixed location. A finite softening length may optionally be applied. The runtime parameters describing the field are listed in Table 12.3.

Table 12.3: Parameters for the `ptmass` gravity sub-module.

Variable	Type	Default	Description
`ptmass`	real	$1. \times 10^4$	Mass of field source
`ptxpos`	real	1	x-position of field source
`ptypos`	real	-10	y-position of field source
`ptzpos`	real	0	z-position of field source
`gravsoft`	real	0.0001	Gravitational softening length

12.2.4 Poisson

The `poisson` sub-module computes the gravitational field produced by the matter in a simulation. Currently, only Newtonian gravity is supported; the potential function produced by this sub-module satisfies Poisson's equation (12.2). Two different elliptic solvers are supplied with FLASH: a multipole solver, suitable for approximately spherical matter distributions, and a multigrid solver, which can be used with general matter distributions. The multipole solver accepts only isolated boundary conditions, whereas the multigrid solver supports both periodic and isolated boundary conditions (for gravity). Boundary conditions for the Poisson solver are specified using the `grav_boundary_type` parameter described in Table 12.4.

When using potential-based gravity modules it is strongly recommended that you use the `quadratic_cartesian` (quadratic) interpolants supplied by PARAMESH. This is because the gravitational acceleration is computed using finite differences. If the interpolants supplied by the mesh are not of at least the same order as the differencing scheme used, unphysical forces will be produced at refinement boundaries. Also, using constant or linear interpolants may cause the multigrid solver to fail to converge.

Table 12.4: Parameters for the `poisson` gravity sub-module.

Variable	Type	Default	Description
`grav_boundary_type`	string	"isolated"	Type of boundary conditions for potential ("isolated", "periodic")

The `poisson` sub-module supplies three solution variables, listed in Table 12.5 (the multigrid solver adds several to this total). See page 127 for an explanation of their meaning. Please see Chapter 15 for descriptions of the available Poisson solvers and their usage.

Table 12.5: Variables provided by the `poisson` sub-module.

Variable	Attributes	Description
`gpot`	NOADVECT NORENORM NOCONSERVE	Gravitational potential at the current timestep
`gpol`	NOADVECT NORENORM NOCONSERVE	Gravitational potential at the previous timestep
`dens`	ADVECT NORENORM CONSERVE	Matter density used as the source of the field

Chapter 13

Particle module

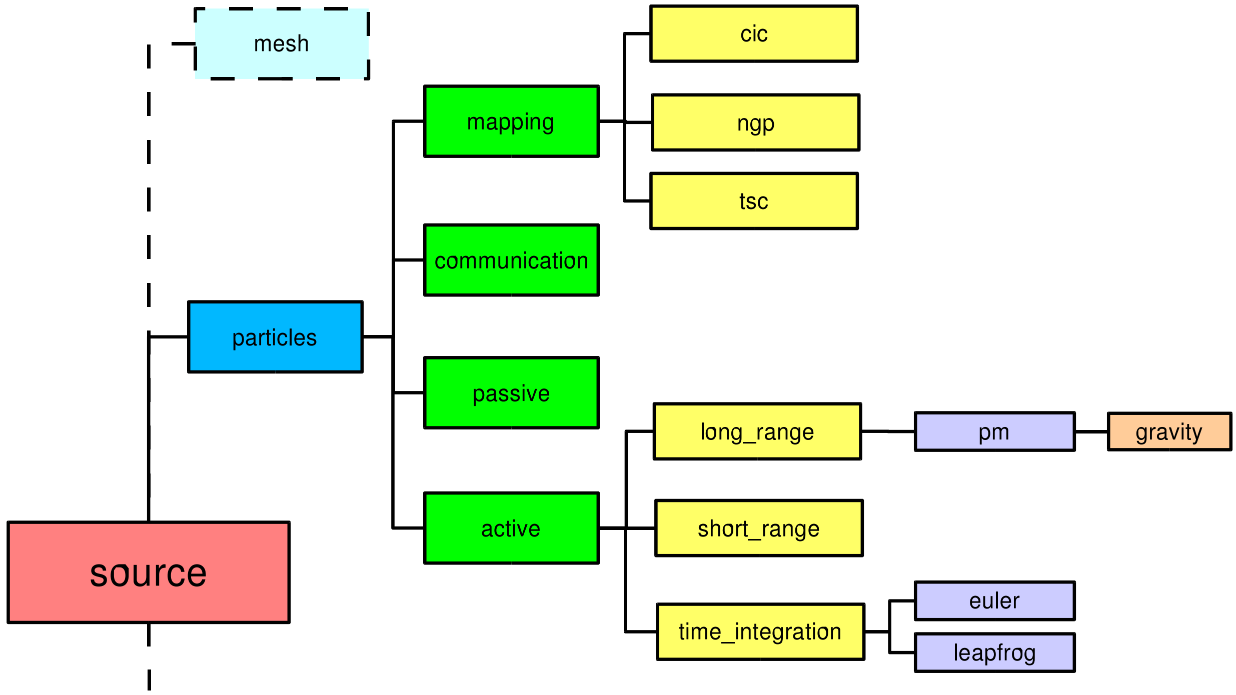


Figure 13.1: The `particles` module directory.

13.1 Algorithms

Most physics modules in FLASH work with grid-based quantities, However, the `particles` module follows the motion of Lagrangian mass tracers, which may or may not contribute to the dynamics. Particles are characterized by positions $\mathbf{x}_i$, velocities $\mathbf{v}_i$, and sometimes other quantities such as masses m_i or charges q_i. Their characteristic quantities are considered to be defined at their positions and can be set by interpolation from the mesh or be used to set mesh quantities by interpolation. 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.

We divide particles into two types, *active* and *passive.* Active particles experience forces and may themselves contribute to the dynamics (*e.g.*, through long-range forces or through collisions). 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 are

$$\begin{aligned} \frac{d\mathbf{x}_i}{dt} &= \mathbf{v}_i \\ m_i \frac{d\mathbf{v}_i}{dt} &= \mathbf{F}_{\mathrm{lr},i} + \mathbf{F}_{\mathrm{sr},i} \ , \end{aligned} \tag{13.1}$$

where $\mathbf{F}_{\mathrm{lr},i}$ represents the sum of all long-range forces (coupling all particles) 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. 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. Since $\mathbf{v}_i$ is externally imposed, only the first of the above equations is relevant.

An excellent introduction to the particle techniques used in FLASH is given by R. W. Hockney and J. W. Eastwood in *Computer Simulation using Particles* (IOP Publishing, 1988).

13.1.1 Active particles

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:

$$\begin{aligned} \mathbf{x}_i^{n+1} &= \mathbf{x}_i^n + \mathbf{v}_i^n \Delta t^n \\ \mathbf{v}_i^{n+1} &= \mathbf{v}_i^n + \mathbf{a}_i^n \Delta t^n \ . \end{aligned} \tag{13.2}$$

  Here $\mathbf{a}_i$ is the particle acceleration.

- **Variable-timestep leapfrog.** Particles are advanced using the following difference equations

$$\begin{aligned} \mathbf{x}_i^1 &= \mathbf{x}_i^0 + \mathbf{v}_i^0 \Delta t^0 \\ \mathbf{v}_i^{1/2} &= \mathbf{v}_i^0 + \frac{1}{2}\mathbf{a}_i^0 \Delta t^0 \\ \mathbf{v}_i^{n+1/2} &= \mathbf{v}_i^{n-1/2} + C_n \mathbf{a}_i^n + D_n \mathbf{a}_i^{n-1} \\ \mathbf{x}_i^{n+1} &= \mathbf{x}_i^n + \mathbf{v}_i^{n+1/2} \Delta t^n \ . \end{aligned} \tag{13.3}$$

  The coefficients C_n and D_n are given by

$$\begin{aligned} C_n &= \frac{1}{2}\Delta t^n + \frac{1}{3}\Delta t^{n-1} + \frac{1}{6}\left(\frac{\Delta t^{n2}}{\Delta t^{n-1}}\right) \\ D_n &= \frac{1}{6}\left(\Delta t^{n-1} - \frac{\Delta t^{n2}}{\Delta t^{n-1}}\right) \ . \end{aligned} \tag{13.4}$$

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

- **Cosmological variable-timestep leapfrog.** The coefficients in the leapfrog update are modified to take into account the effect of cosmological redshift on the particles. The particle positions $\mathbf{x}$ are interpreted as comoving positions, and the particle velocities $\mathbf{v}$ are interpreted as comoving peculiar velocities ($\mathbf{v} = \dot{\mathbf{x}}$). The resulting update steps are

$$
\begin{aligned}
\mathbf{x}_i^1 &= \mathbf{x}_i^0 + \mathbf{v}_i^0 \Delta t^0 \\
\mathbf{v}_i^{1/2} &= \mathbf{v}_i^0 + \frac{1}{2}\mathbf{a}_i^0 \Delta t^0 \\
\mathbf{v}_i^{n+1/2} &= \mathbf{v}_i^{n-1/2} \left[1 - \frac{A^n}{2}\Delta t^n + \frac{1}{3!}\Delta t^{n2}\left(A^{n2} - \dot{A}^n\right)\right]\left[1 - \Delta t^{n-1}\frac{A^n}{2} + \Delta t^{n-1^2}\frac{A^{n2} + 2\dot{A}^n}{12}\right] \\
&\quad + \mathbf{a}_i^n \left[\frac{\Delta t^{n-1}}{2} + \frac{\Delta t^{n2}}{6\Delta t^{n-1}} + \frac{\Delta t^{n-1}}{3} - \frac{A^n \Delta t^n}{6}\left(\Delta t^n + \Delta t^{n-1}\right)\right] \\
&\quad + \mathbf{a}_i^{n-1} \left[\frac{\Delta t^{n-1^2} - \Delta t^{n2}}{6\Delta t^{n-1}} - \frac{A^n \Delta t^{n-1}}{12}(\Delta t^n + \Delta t^{n-1})\right] \\
\mathbf{x}_i^{n+1} &= \mathbf{x}_i^n + \mathbf{v}_i^{n+1/2}\Delta t^n \ .
\end{aligned}
$$

Here we define $A \equiv -2\dot{a}/a$, where a is the scale factor. Note that the acceleration $\mathbf{a}_i^{n-1}$ from the previous timestep must be retained in order to obtain second-order time accuracy. Note also that using the `leapfrog_cosmo` time integration scheme only makes sense if the `cosmology` module is also used, since otherwise $a \equiv 1$ and $\dot{a} \equiv 0$.

13.1.2 Passive particles

Currently, passive particles are advanced using the forward Euler scheme described above for active particles, with velocities $\mathbf{v}_i^n$ obtained using particle-mesh interpolation from the gas grid.

13.2 Using the particle modules

The `particles` module contains several sub-modules:

- The `active` sub-module handles active particles, which contribute to or experience dynamics. This sub-module in turn contains three sub-modules:
  - `time_integration`, which collects different time integration schemes;
  - `long_range`, which collects different long-range force laws (requiring elliptic solvers or the like and dependent upon all other particles);
  - `short_range`, which collects different short-range force laws (directly summed or dependent upon nearest neighbors only).
- The `passive` sub-module handles passive tracer particles, which do not contribute or experience dynamics but instead obtain their velocities by mapping from the gas grid.
- The `communication` sub-module handles the redistribution of particles among processors and the application of boundary conditions to particles. This sub-module is always required when using particles.
- The `mapping` sub-module provides different methods for mapping particle quantities onto the mesh and *vice versa*. FLASH supplies five different mapping schemes: nearest grid point (`mapping/ngp`), cloud-in-cell (`mapping/cic`), triangular-shaped cloud (`mapping/tsc`), and cloud-in-cell for 1D spherical and 2D axisymmetric cylindrical coordinates (`mapping/cic_1dsph` and `mapping/cic_2dcylaxi`, respectively). Some type of mapping is always required when using particles (for now).

To include particles in your FLASH setup, it is necessary to include in your `Modules` file the lines

```
INCLUDE particles/[active || passive ]
INCLUDE particles/communication
INCLUDE particles/mapping/mapping-scheme
```

where *mapping-scheme* is one of the available mesh mapping methods (`ngp`, `cic`, `tsc`, `cic_1dsph`, or `cic_2dcylaxi`). You may also wish to specify the time integration scheme and/or force laws when using active particles. Note that at this time, it is necessary to explicitly specify the `maxblocks` setting when configuring the code with `setup`, *e.g.*,

```
setup orbit -3d -maxblocks=150 -auto
```

The initial particle positions and velocities are set by the `InitParticlePositions()` routine. This routine accepts a single integer argument, the local ID number of the mesh block on which particles are to be initialized. For passive tracer particles, the mesh refinement pattern is not dependent upon particles, so generally one can assume that the most recent call to `mark_grid_refinement()` will have left the block refinement flags in an appropriate state. We initialize particles only on blocks that will not be refined again, are leaf nodes, and have not already been initialized. An example appears in the `InitParticlePositions()` routine supplied with the `particles/passive` module. This version of the routine seeds the grid with a uniform array of particles and sets their initial velocities by interpolation from the gas grid.

Initial conditions for active particles are more complex. Here the refinement pattern may depend upon the particles, but we may wish to initialize more particles than can fit in the memory allocated to a single block (or a single processor). We need some means of predicting where particles will be, refining those regions, and then initializing particles only after the refined blocks have been set up. We do this by constructing a mesh-based "guide function" using a version of the `init_block()` routine. Generally, this is some simple approximation of the mesh-mapped particle density field written to the particle density solution variable (`pden`). The AMR package is directed to refine on this variable. After the block refinement pattern has been set up, the particles are then initialized in the same way as passive particles, and a call to the mesh-mapping routine is performed to initialize the `pden` variable with the correct density field (so that it can be included in the initial checkpoint file). An example of this type of initialization is provided by the `orbit` test problem supplied with FLASH. The `orbit` test problem also supplies an example `wr_integrals()` routine that is useful for writing individual particle trajectories to disk at every timestep. The runtime parameters supplied by the `particles` module are listed in Table 13.1.

Table 13.1: Runtime parameters used with the `particles` module.

Variable	Type	Default	Description
`ipart`	integer	0	If nonzero, evolve particles
`part_dt_factor`	real	0.5	Maximum distance (in zones) a particle is allowed to travel in one timestep. Should not be set larger than the number of guard cells.
`MaxParticlesPerProc`	integer	1	Maximum number of particles per processor (sets size of particle buffers)
`MaxParticlesPerBlock`	integer	10	Maximum number of particles to allow per block
`NumXparticles`	integer	1	For the default initialization routine, sets the number of particles along the x-dimension of the particle array
`NumYparticles`	integer	1	For the default initialization routine, sets the number of particles along the y-dimension of the particle array
`NumZparticles`	integer	1	For the default initialization routine, sets the number of particles along the z-dimension of the particle array

Particle attributes are defined for `setup` 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

`PROPERTY` *property-name* [`REAL`||`INTEGER`]

The particle attributes defined at the top level of the particle module are listed in Table 13.2.

Table 13.2: Particle attributes provided by the `particles` module.

Attribute	Type	Description
`particle_x`	REAL	x-coordinate of particle
`particle_y`	REAL	y-coordinate of particle
`particle_z`	REAL	z-coordinate of particle
`particle_x_vel`	REAL	x-component of particle velocity
`particle_y_vel`	REAL	y-component of particle velocity
`particle_z_vel`	REAL	z-component of particle velocity
`particle_tag`	INTEGER	Unique particle tag
`particle_block`	INTEGER	Current local ID of the mesh block containing the particle

13.2.1 Active particles

The `active` sub-module includes sub-modules for different time integration schemes, long-range force laws (coupling all particles), and short-range force laws (coupling nearby particles). The attributes listed in Table 13.3 are provided by this sub-module.

Available time integration schemes (under `active/time_integration`) include

- `euler` - Simple forward Euler integration.
- `leapfrog` - Variable-timestep leapfrog integration.
- `leapfrog_cosmo` - Cosmological variable-timestep leapfrog integration. (Assumes particle positions are in comoving coordinates and particle velocities are comoving peculiar velocities. See Section 13.1.1 for details.)

The leapfrog-based integrators supply the additional particle attributes listed in Table 13.4. To create new time integration sub-modules, one need specify only an alternative to the existing `AdvanceParticles()` subroutines.

Table 13.3: Particle attributes provided by `particles/active`

Attribute	Type	Description
`particle_mass`	REAL	Particle mass
`particle_x_acc`	REAL	x-component of particle acceleration
`particle_y_acc`	REAL	y-component of particle acceleration
`particle_z_acc`	REAL	z-component of particle acceleration

Table 13.4: Particle attributes provided by `particles/active/time_integration/leapfrog*`

Attribute	Type	Description
`particle_x_acc_old`	REAL	x-component of particle acceleration at previous timestep
`particle_y_acc_old`	REAL	y-component of particle acceleration at previous timestep
`particle_z_acc_old`	REAL	z-component of particle acceleration at previous timestep

Currently, only one long-range force law (gravitation) with one force method (particle-mesh) is included with FLASH. Long-range force laws are contained in the `particles/active/long_range` sub-module and are accessed through the `LongRangeForce()` interface routine. The `pm/gravity` long-range force sub-module requires that the `gravity` module be included in the code and defines the solution variables listed in Table 13.5. Although the particle density (`pden`) is defined, note that it is not necessary to use the `gravity/poisson` module; externally imposed gravitational fields can also be used. Future releases of FLASH will include tree-based long-range force solvers as well as other long-range force laws (*e.g.*, Coulomb forces for plasmas).

Table 13.5: Solution variables provided by the `particles/active/long_range/pm/gravity` sub-module.

Name	Attributes	Description
`pden`	NOADVECT NORENORM NOCONSERVE	Mesh-mapped particle density
`grav`	NOADVECT NORENORM NOCONSERVE	Gravitational acceleration component

In the current release, no short-range force laws (`particles/active/short_range`) are supplied with FLASH. The interface to this sub-module is expressed via the `ShortRangeForce()` subroutine. In future releases of FLASH, we expect short-range forces, such as the particle-particle component of P^3M and the local interactions in smoothed-particle hydrodynamics (SPH), to be implemented as sub-modules of `short_range`. In this connection note that the adaptive mesh supplied by PARAMESH is useful as a chaining mesh for efficiently locating nearest neighbors.

13.2.2 Passive particles

Passively advected tracer particles are advanced using forward Euler integration and do not supply or require any special particle attributes or solution variables (except gas velocity). The `passive` sub-module does require that a `hydro` module be included in the code. Using the default `InitParticlePositions()`, it is extremely straightforward to add passive tracer particles to any existing hydrodynamics problem; simply add to your `Modules` file the lines

```
INCLUDE particles/passive
INCLUDE particles/communication
INCLUDE particles/mapping/mapping-scheme
```

where *mapping-scheme* is one of the available mesh mapping methods (`ngp`, `cic`, `tsc`, `cic_1dsph`, or `cic_2dcylaxi`). Then run `setup` with an explicitly specified value for `maxblocks`, *e.g.*,

```
setup sedov -2d -maxblocks=500
```

and set the paramters `MaxParticlesPerProc`, `MaxParticlesPerBlock`, `NumXparticles`, `NumYparticles`, and `NumZparticles` as desired (see Table 13.1). The tracer particles will be initialized as a uniformly spaced array that will deform as the hydrodynamical flow evolves.

13.2.3 A note about particle I/O

In FLASH 2.3 particle data are written to and read from checkpoint files by the `hdf5_serial` and `hdf5_parallel` I/O modules (Chapter 7). For more information on the format of particle data written to HDF5 files, see Sec. 7.3.2. Particle data are not presently written to plotfiles, nor are they written by the `hdf4` I/O module.

Chapter 14

Cosmology module

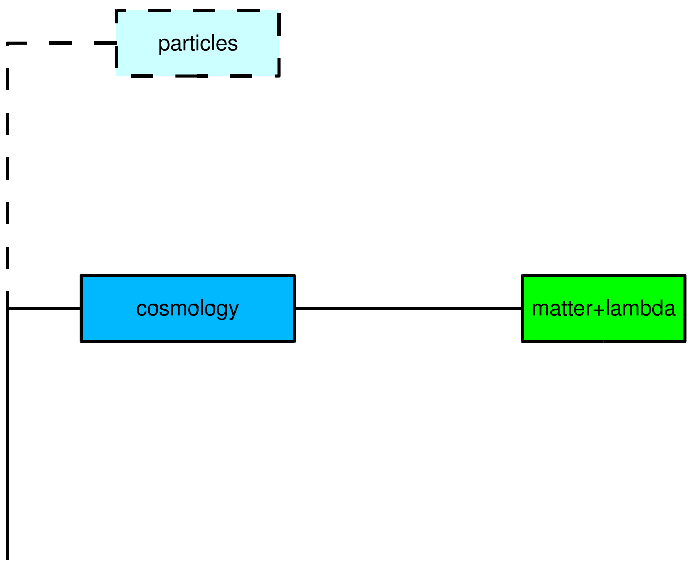


Figure 14.1: The cosmology module directory.

The `cosmology` module solves the Friedmann equation for the scale factor in an expanding universe, applies a cosmological redshift to the hydrodynamical quantities, and supplies library functions for various routine cosmological calculations needed by the rest of the code for initializing, performing, and analyzing cosmological simulations.

14.1 Algorithms and equations

The cosmology module makes several assumptions about the interpretation of physical quantities that enable any hydrodynamics or materials modules written for a non-expanding universe to work unmodified in a cosmological context. All calculations are assumed to take place in comoving coordinates $\mathbf{x} = \mathbf{r}/a$, where $\mathbf{r}$ is a proper position vector and $a(t)$ is the time-dependent cosmological scale factor. The present epoch is defined to correspond to $a = 1$; in the following discussion we use $t = t_0$ to refer to the age of the Universe at the present epoch. The gas velocity $\mathbf{v}$ is taken to be the comoving peculiar velocity $\dot{\mathbf{x}}$. The comoving gas density, pressure, temperature, and internal energy are defined to be

$$\begin{aligned} \rho &\equiv a^3\tilde{\rho} \\ p &\equiv a\tilde{p} \\ T &\equiv \frac{\tilde{T}}{a^2} \\ \rho\epsilon &\equiv a\tilde{\rho}\tilde{\epsilon} \ . \end{aligned} \tag{14.1}$$

The quantities marked with a tilde ($\tilde{\rho}$ *etc.*) are the corresponding "proper" or physical quantities. Note that, in terms of comoving quantities, the equation of state has the same form as for the proper quantities in noncomoving coordinates. For example, the perfect-gas equation of state is

$$\rho\epsilon = \frac{p}{\gamma - 1} = \frac{\rho k T}{(\gamma-1)\mu} \ . \tag{14.2}$$

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

$$\frac{\partial\rho}{\partial t} + \nabla\cdot(\rho\mathbf{v}) = 0 \tag{14.3}$$

$$\frac{\partial\rho\mathbf{v}}{\partial t} + \nabla\cdot(\rho\mathbf{v}\mathbf{v}) + \nabla p + 2\frac{\dot{a}}{a}\rho\mathbf{v} + \rho\nabla\phi = 0 \tag{14.4}$$

$$\frac{\partial\rho E}{\partial t} + \nabla\cdot[(\rho E + p)\mathbf{v}] + \frac{\dot{a}}{a}[(3\gamma-1)\rho\epsilon + 2\rho v^2] + \rho\mathbf{v}\cdot\nabla\phi = 0 \tag{14.5}$$

$$\frac{\partial\rho\epsilon}{\partial t} + \nabla\cdot[(\rho\epsilon + p)\mathbf{v}] - \mathbf{v}\cdot\nabla p + \frac{\dot{a}}{a}(3\gamma-1)\rho\epsilon = 0 \ . \tag{14.6}$$

Here E is the specific total energy, $\epsilon + \frac{1}{2}v^2$, and γ is the effective ratio of specific heats. The `cosmology` module applies the terms involving $\dot{a}$ via the `RedshiftHydro` routine.

The comoving potential ϕ in the above equations is the solution to the Poisson equation in the form

$$\nabla^2\phi = \frac{4\pi G}{a^3}(\rho - \bar{\rho}) \ , \tag{14.7}$$

where $\bar{\rho}$ is the comoving mean matter density. Note that, because of the presence of a in eq. (14.7), the gravity modules must explicitly divide their source terms by a^3. Modules like `gravity`, which require the scale factor or the redshift z ($a = (1+z)^{-1}$), can obtain them from the database via the `dBasePropertyReal` function (Sec. 5.1.2). Note also that if a collisionless matter component (particles) is also present, its density must be added to the gas density on the right-hand side of eq. (14.7). This is handled by the `gravity` module.

The comoving mean matter density is defined in terms of the critical density $\rho_{\rm crit}$ by

$$\begin{aligned} \bar{\rho} &\equiv \Omega_m\rho_{\rm crit} \\ \rho_{\rm crit} &\equiv \frac{3H^2}{8\pi G} \ . \end{aligned} \tag{14.8}$$

The Hubble parameter $H(t)$ (to be distinguished from the Hubble "constant" $H_0 \equiv H(t_0)$) is given by the Friedman equation

$$H^2(t) \equiv \left(\frac{\dot{a}}{a}\right)^2 = H_0^2\left(\frac{\Omega_m}{a^3} + \frac{\Omega_r}{a^4} + \Omega_\Lambda - \frac{\Omega_c}{a^2}\right) \ . \tag{14.9}$$

Here Ω_m, Ω_r, and Ω_Λ are the present-day densities, respectively, of matter, radiation, and cosmological constant, divided by $\rho_{\rm crit}$. The contribution of the overall spatial curvature of the Universe is given by

$$\Omega_c \equiv \Omega_m + \Omega_r + \Omega_\Lambda - 1 \ . \tag{14.10}$$

The `SolveFriedmannEquation` routine numerically solves the Friedmann equation to obtain the scale factor and its rate of change as functions of time. In principle, any good ODE integrator can be used; the `matter+lambda` sub-module uses the variable-order Adams algorithm to integrate the Friedmann equation under the assumption that $\Omega_r = 0$. Sub-modules can also use analytic solutions where appropriate.

Redshift terms for particles are handled separately by the appropriate time integration sub-modules of the `particles` module. For an example, see the `particles/active/time_integration/leapfrog_cosmo` sub-module (Chapter 13).

14.2 Using the cosmology module

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

`INCLUDE cosmology/`*sub-module*

in your `Modules` file when you configure the code with `setup`. At present only one sub-module is distributed with FLASH: `matter+lambda`. This sub-module assumes the contribution of radiation to be negligible in comparison with those of matter and the cosmological constant.

The runtime parameters available with the `cosmology` module are described in Table 14.1. Note that the total effective mass density is not explicitly specified but is inferred from the sum of the `OmegaMatter`, `OmegaRadiation`, and `CosmologicalConstant` parameters. The `MaxScaleChange` parameter sets the maximum allowed fractional change in the scale factor a during a single timestep. This is enforced by the `ExpansionTimestep` routine. The default value is set to 10^{99} to avoid interfering with non-cosmological simulations.

Table 14.1: Runtime parameters used with the `cosmology` module.

Parameter	Type	Default	Description
`OmegaMatter`	real	0.3	Ratio of total mass density to critical density at the present epoch (Ω_m)
`OmegaBaryon`	real	0.05	Ratio of baryonic (gas) mass density to critical density at the present epoch; must be $\leq$ `OmegaMatter` (Ω_b)
`CosmologicalConstant`	real	0.7	Ratio of the mass density equivalent in the cosmological constant to the critical density at the present epoch (Ω_Λ)
`OmegaRadiation`	real	5×10^{-5}	Ratio of the mass density equivalent in radiation to the critical density at the present epoch (Ω_r)
`HubbleConstant`	real	2.1065×10^{-18}	Value of the Hubble constant H_0 in sec^{-1}
`MaxScaleChange`	real	1×10^{99}	Maximum permitted fractional change in the scale factor during each timestep

The Fortran 90 module `CosmologicalFunctions` supplies a number of functions and routines that are helpful in initializing, performing, and analyzing cosmological simulations. They can be used in your own `init_block` routine to help in writing code that is not restricted to a single cosmological model. (The current list of routines is expected to grow with time.) The cosmological functions and routines are:

- `MassToLength(M, lambda)`
  (subroutine) Given a mass scale `M`, return the corresponding comoving diameter `lambda` of a sphere containing the given amount of mass. Obtain the values of cosmological parameters from the runtime parameter database.

- `MassToLengthConversion(M, lambda, N, Omega0, H0, G)`
  (subroutine) Given an array of mass scales `M(N)`, compute the comoving diameters `lambda(N)` of spheres

containing these masses. Compute the comoving critical density using the supplied values of `H0` and `G` and assume `Omega0` to be the present-day mass density parameter value.

- `CDMPowerSpectrum(k, Anorm, znorm, npspc, Omega0, h, Lambda0)`
  (real function) Return the present-day cold dark matter power spectrum at a given wavenumber `k`, given the normalization `Anorm`, the normalization redshift `znorm`, the primordial spectral index `npspc`, the present mass density `Omega0`, the Hubble constant in units of 100 km s^{-1} Mpc^{-1}, and the present density parameter due to the cosmological constant `Lambda0`. The wavenumber and normalization must use length units of Mpc, and the result is expressed in Mpc^3. The `matter+lambda` sub-module provides a fit to this power spectrum from Bardeen *et al.* (1986), which assumes baryons do not make a significant contribution to the mass density. Better fits are available; see *e.g.*, Hu and Sugiyama (1996) or Bunn and White (1997).

- `TopHatFilter(k, r)`
  (real function) Given a wavenumber `k` and the characteristic length scale `r` (cutoff radius for the top hat filter in real space), return the Fourier transform of the top hat filter.

- `ComputeVariance(lambda, Mass, Delta0, dDelta0dM, N, f, PwrSpc, Filter, Anorm, znorm, npspc, Omega0, h, Lambda0)`
  (subroutine) Given an array of comoving length scales `lambda(N)` and a processed power spectrum function `PwrSpc`, compute the linear variance $(\delta M/M)^2$ at the present epoch. The factor `f` is multiplied by the mass scale in applying the smoothing kernel. `Filter` is a Fourier-space filter function with the same interface as `TopHatFilter`. `Anorm`, `znorm`, `npspc`, `Omega0`, `h`, and `Lambda0` are passed to `PwrSpc` and have the same interpretations as in `CDMPowerSpectrum`. The linear variance is returned in the array `Delta0(N)`, and its derivative with respect to mass is returned in `dDelta0dM(N)`. The masses corresponding to the lengths `lambda` are returned in the array `Mass(N)`.

- `RedshiftToTime(z, t)`
  (subroutine) Compute the age of the Universe `t` corresponding to a redshift `z`. Obtain the values of cosmological parameters from the runtime parameter database.

- `RedshiftToTimeConversion(z, t, dtdz, N, Omega0, H0, Lambda0, c, Omegatot)`
  (subroutine) Given an array of ages `t(N)`, compute the corresponding redshifts `z(N)` and first derivatives `dtdz(N)` given a present-day mass density parameter `Omega0`, a Hubble constant `H0`, a cosmological constant density parameter `Lambda0`, the speed of light `c`, and the total present-day density parameter `Omegatot`.

- `ComputeDeltaCrit(z, dcrit, dcritdz, D, N, Omega0, H0, Lambda0, Omegatot)`
  (subroutine) Compute the linear overdensity at turnaround in the spherical collapse model at the given redshifts `z(N)`. `dcrit(N)` and `dcritdz(N)` return the critical overdensity and its redshift derivative, respectively. `D(N)` returns the growth factor for linear perturbations at the given redshifts. The remaining arguments are interpreted as for `RedshiftToTimeConversion`. For more details, see the appendix of Lacey and Cole (1993).

Chapter 15

Solvers module

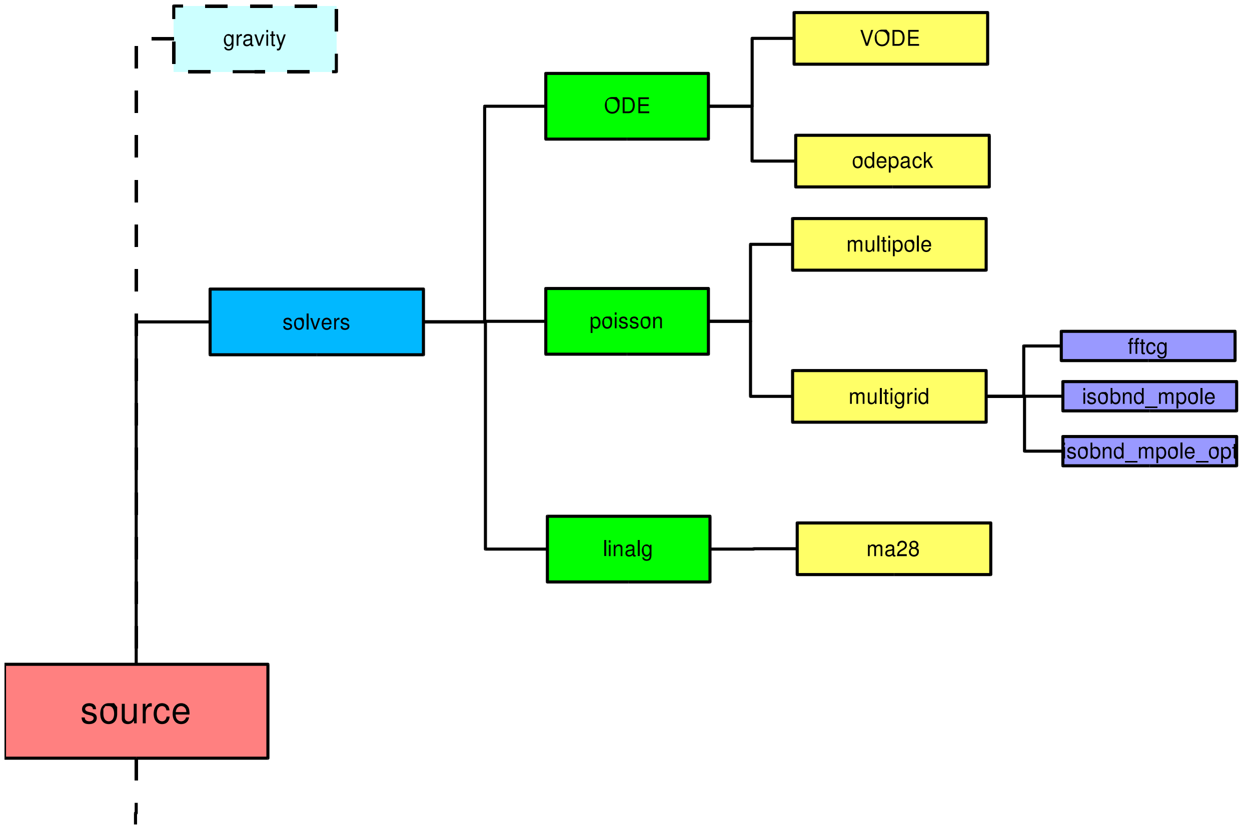


Figure 15.1: The solvers module directory.

15.1 Ordinary differential equations (ODE)

Some source terms, *e.g.* burn, need to solve systems of ordinary differential equations. There are many standard packages available for integrating systems of first order differential equations that can handle both stiff and non-stiff systems. We include the VODE package as a module, so that it can readily be used in FLASH. This is identical to the version available at netlib (http://www.netlib.org), but it has been converted to free-form Fortran 90 by the `ftof90` program in `tools/scripts/text_formatting`. In all cases, the single precision routines are provided, since FLASH relies on the compiler to promote the precision of all floating point variables and constants.

15.1.1 The VODE package

The VODE package (Brown *et al.* 1989) solves an initial value problem of the form

$$\frac{\mathrm{d}y_i}{\mathrm{d}t} = f_i(t, y_1, ..., y_N) \ . \tag{15.1}$$

The main interface to VODE is `svode`, which takes a pointer to a function that evaluates the right hand side of the ODEs, the initial value of all of the y_i's, and various parameters specifying the tolerance to reach when solving the ODEs, whether or not the system is stiff, and whether or not there is a user-supplied Jacobian to aid in the solution. The full details of how to use the VODE integrator are contained in the extensive comments at the top of `svode.F90` and are not repeated here.

15.2 Poisson equation

The `solvers/poisson` module supplies different algorithms for solving the general Poisson equation for a potential $\phi(\mathbf{x})$ given a source $\rho(\mathbf{x})$

$$\nabla^2\phi(\mathbf{x}) = \alpha\rho(\mathbf{x}) \ . \tag{15.2}$$

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.

15.2.1 Multipole Poisson solver

The multipole Poisson solver is appropriate for spherical or nearly-spherical source distributions with isolated boundary conditions. It currently works in 1D spherical, 2D axisymmetric cylindrical ($r - z$), and 3D Cartesian geometries. Because of the imposed symmetries, in the first case, only the monopole term ($\ell = 0$) makes sense, while in the second case, 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 $\mathbf{x}_{\mathrm{cm}}$

$$\mathbf{x}_{\mathrm{cm}} = \frac{\int d^3\mathbf{x}\,\mathbf{x}\rho(\mathbf{x})}{\int d^3\mathbf{x}\,\rho(\mathbf{x})} \ . \tag{15.3}$$

We will take $\mathbf{x}_{\mathrm{cm}}$ as our origin. In integral form, Poisson's equation (15.2) is

$$\phi(\mathbf{x}) = -\frac{\alpha}{4\pi}\int d^3\mathbf{x}'\,\frac{\rho(\mathbf{x}')}{|\mathbf{x}-\mathbf{x}'|} \ . \tag{15.4}$$

The Green's function for this equation satisfies the relationship

$$\frac{1}{|\mathbf{x}-\mathbf{x}'|} = 4\pi\sum_{\ell=0}^{\infty}\sum_{m=-\ell}^{\ell}\frac{1}{2\ell+1}\frac{r_<^\ell}{r_>^{\ell+1}}Y^*_{\ell m}(\theta',\varphi')Y_{\ell m}(\theta,\varphi) \ , \tag{15.5}$$

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

$$\begin{aligned} r_< &\equiv \min\{|\mathbf{x}|, |\mathbf{x}'|\} \\ r_> &\equiv \max\{|\mathbf{x}|, |\mathbf{x}'|\} \ . \end{aligned} \tag{15.6}$$

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

$$Y_{\ell m}(\theta,\varphi) \equiv (-1)^m\sqrt{\frac{2\ell+1}{4\pi}\frac{(\ell-m)!}{(\ell+m)!}}P_{\ell m}(\cos\theta)e^{im\varphi} \ . \tag{15.7}$$

$P_{\ell m}(x)$ are Legendre polynomials. Substituting eq. (15.5) into eq. (15.4), we obtain

$$\begin{aligned}\phi(\mathbf{x}) &= -\alpha \sum_{\ell=0}^{\infty} \sum_{m=-\ell}^{\ell} \frac{1}{2\ell+1} \Bigg\{ Y_{\ell m}(\theta,\varphi) \times \\ &\left[r^{\ell} \int_{r<r'} d^3\mathbf{x}' \frac{\rho(\mathbf{x}')Y^*_{\ell m}(\theta',\varphi')}{r'^{\ell+1}} + \frac{1}{r^{\ell+1}} \int_{r>r'} d^3\mathbf{x}' \rho(\mathbf{x}')Y^*_{\ell m}(\theta',\varphi') r'^{\ell} \right] \Bigg\} .\end{aligned} \tag{15.8}$$

In practice, we carry out the first summation up to some limiting multipole $\ell_{\max}$. By taking spherical harmonic expansions about the center of mass, we ensure that the expansions are dominated by low-multipole terms, so that for a given value of $\ell_{\max}$, the error created by neglecting high-multipole terms is minimized. Note that the product of spherical harmonics in eq. (15.8) is real-valued

$$\begin{aligned}\sum_{m=-\ell}^{\ell} Y^*_{\ell m}(\theta',\varphi') Y_{\ell m}(\theta,\varphi) &= \frac{2\ell+1}{4\pi} \Bigg[P_{\ell 0}(\cos\theta) P_{\ell 0}(\cos\theta') + \\ &2 \sum_{m=1}^{\ell} \frac{(\ell-m)!}{(\ell+m)!} P_{\ell m}(\cos\theta) P_{\ell m}(\cos\theta') \cos\left(m(\varphi-\varphi')\right) \Bigg] .\end{aligned} \tag{15.9}$$

Using a trigonometric identity to split up the last cosine in this expression and substituting for the inner sums in eq. (15.8), we obtain

$$\begin{aligned}\phi(\mathbf{x}) &= -\frac{\alpha}{4\pi} \sum_{\ell=0}^{\infty} P_{\ell 0}(\cos\theta) \left[r^{\ell} \mu^{\rm eo}_{\ell 0}(r) + \frac{1}{r^{\ell+1}} \mu^{\rm ei}_{\ell 0}(r) \right] - \\ &\frac{\alpha}{2\pi} \sum_{\ell=1}^{\infty} \sum_{m=1}^{\ell} P_{\ell m}(\cos\theta) \Bigg[(r^{\ell} \cos m\varphi) \mu^{\rm eo}_{\ell m}(r) + (r^{\ell} \sin m\varphi) \mu^{\rm oo}_{\ell m}(r) + \\ &\frac{\cos m\varphi}{r^{\ell+1}} \mu^{\rm ei}_{\ell m}(r) + \frac{\sin m\varphi}{r^{\ell+1}} \mu^{\rm oi}_{\ell m}(r) \Bigg] .\end{aligned} \tag{15.10}$$

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

$$\mu^{\rm ei}_{\ell m}(r) \equiv \frac{(\ell-m)!}{(\ell+m)!} \int_{r>r'} d^3\mathbf{x}'\, r'^{\ell} \rho(\mathbf{x}') P_{\ell m}(\cos\theta') \cos m\varphi' \tag{15.11}$$

$$\mu^{\rm oi}_{\ell m}(r) \equiv \frac{(\ell-m)!}{(\ell+m)!} \int_{r>r'} d^3\mathbf{x}'\, r'^{\ell} \rho(\mathbf{x}') P_{\ell m}(\cos\theta') \sin m\varphi' \tag{15.12}$$

$$\mu^{\rm eo}_{\ell m}(r) \equiv \frac{(\ell-m)!}{(\ell+m)!} \int_{r<r'} d^3\mathbf{x}'\, \frac{\rho(\mathbf{x}')}{r'^{\ell+1}} P_{\ell m}(\cos\theta') \cos m\varphi' \tag{15.13}$$

$$\mu^{\rm oo}_{\ell m}(r) \equiv \frac{(\ell-m)!}{(\ell+m)!} \int_{r<r'} d^3\mathbf{x}'\, \frac{\rho(\mathbf{x}')}{r'^{\ell+1}} P_{\ell m}(\cos\theta') \sin m\varphi' . \tag{15.14}$$

The procedure is thus to compute the moment integrals (eqs. (15.11) − (15.14)) for a given source field $\rho(\mathbf{x})$, and then to use these moments in eq. (15.10) 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 zone-averaged quantities discretized on a block-structured mesh with varying zone 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

$$\mu^{\rm ei}_{\ell m,q} \equiv \mu^{\rm ei}_{\ell m}(q\Delta) \qquad \mu^{\rm eo}_{\ell m,q} \equiv \mu^{\rm eo}_{\ell m}((q-1)\Delta) \tag{15.15}$$

$$\mu^{\rm oi}_{\ell m,q} \equiv \mu^{\rm oi}_{\ell m}(q\Delta) \qquad \mu^{\rm oo}_{\ell m,q} \equiv \mu^{\rm oo}_{\ell m}((q-1)\Delta) . \tag{15.16}$$

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 zone 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 zones to the tabulated moments requires some care. To reduce the error caused by the grid geometry, in each zone ijk we establish a subgrid consisting of N' points at the locations $\mathbf{x}'_{i'j'k'}$, where

$$x'_{i'} = x_i + (i' - 0.5(N'-1))\frac{\Delta x_i}{N'} \ , \qquad i' = 0\ldots N'-1 \tag{15.17}$$

$$y'_{j'} = y_j + (j' - 0.5(N'-1))\frac{\Delta y_j}{N'} \ , \qquad j' = 0\ldots N'-1 \tag{15.18}$$

$$z'_{k'} = z_k + (k' - 0.5(N'-1))\frac{\Delta z_k}{N'} \ , \qquad k' = 0\ldots N'-1 \ , \tag{15.19}$$

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

$$\mu^{\mathrm{ei}}_{\ell m,q\geq q'} \leftarrow \mu^{\mathrm{ei}}_{\ell m,q\geq q'} + \frac{(\ell-m)!}{(\ell+m)!}\frac{\Delta x_i \Delta y_j \Delta z_k}{N'^3} {r'}^{\ell}_{i'j'k'}\rho(\mathbf{x}'_{i'j'k'})P_{\ell m}(\cos\theta'_{i'j'k'})\cos m\varphi'_{i'j'k'} \tag{15.20}$$

$$\mu^{\mathrm{oi}}_{\ell m,q\geq q'} \leftarrow \mu^{\mathrm{oi}}_{\ell m,q\geq q'} + \frac{(\ell-m)!}{(\ell+m)!}\frac{\Delta x_i \Delta y_j \Delta z_k}{N'^3} {r'}^{\ell}_{i'j'k'}\rho(\mathbf{x}'_{i'j'k'})P_{\ell m}(\cos\theta'_{i'j'k'})\sin m\varphi'_{i'j'k'} \tag{15.21}$$

$$\mu^{\mathrm{eo}}_{\ell m,q\leq q'} \leftarrow \mu^{\mathrm{eo}}_{\ell m,q\leq q'} + \frac{(\ell-m)!}{(\ell+m)!}\frac{\Delta x_i \Delta y_j \Delta z_k}{N'^3} \frac{\rho(\mathbf{x}'_{i'j'k'})}{{r'}^{\ell+1}_{i'j'k'}}P_{\ell m}(\cos\theta'_{i'j'k'})\cos m\varphi'_{i'j'k'} \tag{15.22}$$

$$\mu^{\mathrm{oo}}_{\ell m,q\leq q'} \leftarrow \mu^{\mathrm{oo}}_{\ell m,q\leq q'} + \frac{(\ell-m)!}{(\ell+m)!}\frac{\Delta x_i \Delta y_j \Delta z_k}{N'^3} \frac{\rho(\mathbf{x}'_{i'j'k'})}{{r'}^{\ell+1}_{i'j'k'}}P_{\ell m}(\cos\theta'_{i'j'k'})\sin m\varphi'_{i'j'k'} \ , \tag{15.23}$$

where

$$q' = \left\lfloor \frac{|\mathbf{x}'_{i'j'k'}|}{\Delta} \right\rfloor + 1 \tag{15.24}$$

is the index of the radial sample within which the subzone center lies. These expressions introduce (hopefully) small errors when compared to eq. (15.11) − (15.14), 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 zone to be flat. Note that the total source computed by this method ($\mu^{\mathrm{ei}}_{\ell m,N_q}$) 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 2; 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 eqs. (15.20) − (15.23) are evaluated using trigonometric recurrence relations, and the factorials are pre-computed and stored at the beginning of the run.

When computing the zone-averaged potential, we again employ a subgrid, but here the subgrid points fall on zone boundaries to improve the continuity of the result. Using $N''+1$ subgrid points per dimension, we have

$$x''_{i''} = x_i + (i'' - 0.5N''))\frac{\Delta x_i}{N''} \ , \qquad i'' = 0\ldots N'' \tag{15.25}$$

$$y''_{j''} = y_j + (j'' - 0.5N''))\frac{\Delta y_j}{N''} \ , \qquad j'' = 0\ldots N'' \tag{15.26}$$

$$z''_{k''} = z_k + (k'' - 0.5N''))\frac{\Delta z_k}{N''} \ , \qquad k'' = 0\ldots N'' \ . \tag{15.27}$$

The default value of N'' is 6. The zone-averaged potential in zone ijk is then

$$\phi_{ijk} = \frac{1}{N''^3}\sum_{i''j''k''}\phi(\mathbf{x}''_{i''j''k''}) \ , \tag{15.28}$$

where the terms in the sum are evaluated via eq. (15.10) up to the limiting multipole order $\ell_{\max}$.

15.2.2 Multigrid Poisson solver

The multigrid Poisson solver is appropriate for general source distributions and can solve problems with periodic, isolated, Dirichlet, Neumann, or given-value boundary conditions. The algorithm distributed with FLASH is based on a multilevel refinement scheme described by Martin and Cartwright (1996). Isolated boundary conditions are implemented via a method based on James' (1978) algorithm.

Multilevel refinement algorithms (Brandt 1977; Trottenberg, Oosterlee, & Schüller 2001) solve elliptic equations such as the Poisson equation by accelerating the convergence of relaxation methods. The latter (*e.g.*, Jacobi, Gauss-Seidel, SOR) are straightforward but converge very slowly, because they accomplish the global coupling implied by an elliptic equation by a series of iterations that communicate information from one side of the grid to the other one zone at a time. Hence their convergence rate (fractional reduction in error per iteration) decreases with increasing grid size. Modal analysis shows that the longest-wavelength components of the error require the most iterations to decrease to a given level. By performing iterations on a sequence of increasingly coarser grids, multigrid algorithms bring all wavelengths into convergence at the same rate. This works, because long wavelengths on a fine mesh appear to be short wavelengths on a coarse mesh.

Adaptive mesh refinement (AMR) provides many benefits in conjunction with a multigrid solver. Where errors are unlikely to have short-wavelength components, it makes sense to avoid using fine grids, thus reducing storage requirements and the cost of relaxations on fine levels. The AMR package manages the multilevel mesh data structures and can handle all parallel communication, freeing the multigrid solver from such details. The AMR package also supplies many of the basic functions required by multigrid algorithms, including prolongation, restriction, and boundary condition updates. Therefore, we use a mesh hierarchy defined by the AMR package. With the Poisson equation, it is desirable to refine narrow peaks in the source field, since it requires the curvature of the solution (the potential) to undergo the largest small-scale fluctuations at such peaks. Therefore, when solving elliptic problems using the multigrid module, it may be a good idea to add to the second derivative criterion supplied with FLASH one which refines blocks based on their mean source contrast with respect to a fixed reference density. This is illustrated by the `jeans` problem setup.

AMR does introduce complications, however. Because the mesh hierarchy contains jumps in refinement, it is necessary to interpolate when setting guard cell values for fine blocks adjoining coarser blocks. As Martin and Cartwright point out, this requires an interpolation scheme with at least the same order of accuracy as the finite differencing scheme used. Thus quadratic interpolants must be used with the Poisson equation. However, unless the first derivative of the solution is also matched across jumps in refinement, unphysical forces will be produced at such boundaries, and the multigrid solver will fail to converge. Since we regard the solution on the finer level as being of higher quality than the solution on the coarser level, in such situations we allow the fine grid to determine the value of the first derivative on the boundary.

Before describing the algorithm, let us first define some terms. We work with approximations $\tilde{\phi}(\mathbf{x})$ to the solution $\phi(\mathbf{x})$. The *residual* is a measure of the error in $\tilde{\phi}(\mathbf{x})$; it is given by

$$\begin{aligned} R(\mathbf{x}) &\equiv \nabla^2\phi(\mathbf{x}) - \nabla^2\tilde{\phi}(\mathbf{x}) \\ &= \alpha\rho(\mathbf{x}) - \nabla^2\tilde{\phi}(\mathbf{x}) \ . \end{aligned} \tag{15.29}$$

The first term on the right-hand side is the *source* $S(\mathbf{x})$; it is computed outside of the multigrid solver and then is passed in. Since the Poisson equation is linear, the residual satisfies the equation

$$\nabla^2 C(\mathbf{x}) = R(\mathbf{x}) \ , \tag{15.30}$$

whose solution $C(\mathbf{x})$ is the *correction*

$$C(\mathbf{x}) \equiv \phi(\mathbf{x}) - \tilde{\phi}(\mathbf{x}) \ . \tag{15.31}$$

The source, solution, residual, and correction are all approximated by zone-averaged values on a hierarchy of meshes, each level of which consists of a number of blocks or patches of zones as prescribed by the adaptive mesh package. Where a given mesh block is not a "leaf node" – *i.e.*, it is overlain by other blocks at a higher level of refinement – only the residual and correction are defined (though storage may be allocated for the other variables as well). When discussing discretized quantities such as the solution ϕ, we will refer to them in the form $\phi^{b\ell}_{ijk}$, where b is the block number, ℓ is its level of refinement ($\ell = 1$ being the coarsest level), and ijk are zone indices within the block b. The notation $\mathcal{P}(b)$ will refer to the parent (coarser) block containing block b, while $\mathcal{C}(b)$ will refer collectively to the child (finer) blocks associated with b. $\mathcal{N}(b, \pm x/y/z)$ will refer to the block(s) neighboring block b in the $\pm x$-, y-, or z-directions. For conciseness, where a given neighbor is at a higher level of refinement, $\mathcal{N}$ will be understood to refer collectively to all of the neighboring blocks in that direction, with zone indices running from one to the product of the refinement factor with the size of block b in each dimension. Zone indices are assumed to run between $1\ldots n_x$, $1\ldots n_y$, and $1\ldots n_z$ in each block, with a factor of 2 refinement between levels. The generalization to different block/patch sizes and different refinement factors should be fairly straightforward.

Difference operators approximating ∇^2 on each grid level are defined for relaxation and for computing the residual. On level ℓ, which has zone spacings Δx_ℓ, Δy_ℓ, and Δz_ℓ in the x-, y-, and z-directions, we use

$$\begin{aligned}
\mathcal{D}^2_\ell \phi^{b\ell}_{ijk} \equiv & \frac{1}{\Delta x_\ell}\left(\mathcal{D}^{1x}_\ell \phi^{b\ell}_{i+1/2,j,k} - \mathcal{D}^{1x}_\ell \phi^{b\ell}_{i-1/2,j,k}\right) + && (15.32)\\
& \frac{1}{\Delta y_\ell}\left(\mathcal{D}^{1y}_\ell \phi^{b\ell}_{i,j+1/2,k} - \mathcal{D}^{1y}_\ell \phi^{b\ell}_{i,j-1/2,k}\right) + && (15.33)\\
& \frac{1}{\Delta z_\ell}\left(\mathcal{D}^{1z}_\ell \phi^{b\ell}_{i,j,k+1/2} - \mathcal{D}^{1z}_\ell \phi^{b\ell}_{i,j,k-1/2}\right) , && (15.34)
\end{aligned}$$

where

$$\begin{aligned}
\mathcal{D}^{1x}_\ell \phi^{b\ell}_{i+1/2,jk} &\equiv \frac{1}{\Delta x_\ell}\left(\phi^{b\ell}_{i+1,j,k} - \phi^{b\ell}_{i-1,j,k}\right) && (15.35)\\
\mathcal{D}^{1y}_\ell \phi^{b\ell}_{i,j+1/2,k} &\equiv \frac{1}{\Delta y_\ell}\left(\phi^{b\ell}_{i,j+1,k} - \phi^{b\ell}_{i,j-1,k}\right) && (15.36)\\
\mathcal{D}^{1z}_\ell \phi^{b\ell}_{ij,k+1/2} &\equiv \frac{1}{\Delta z_\ell}\left(\phi^{b\ell}_{i,j,k+1} - \phi^{b\ell}_{i,j,k-1}\right) . && (15.37)
\end{aligned}$$

In cases in which the required values of ϕ lie outside of a block, they are obtained from guard cells that are filled by the AMR package, possibly through restriction or prolongation from neighboring blocks. The derivative-matching procedure outlined above is applied only when computing the residual. In this case we replace the $\mathcal{D}^1$ operators at jumps in refinement with the following operators

$$\begin{aligned}
\mathcal{D'}^{1x}_\ell \phi^{b\ell}_{n_x+1/2,jk} &\equiv \mathcal{R}_\ell\left[\mathcal{D}^{1x}_{\ell+1} \phi^{\mathcal{N}(b,+x),\ell+1}_{1/2,2j-1,2k-1}\right] && (15.38)\\
\mathcal{D'}^{1x}_\ell \phi^{b\ell}_{1/2,jk} &\equiv \mathcal{R}_\ell\left[\mathcal{D}^{1x}_{\ell+1} \phi^{\mathcal{N}(b,-x),\ell+1}_{n_x+1/2,2j-1,2k-1}\right] && (15.39)\\
\mathcal{D'}^{1y}_\ell \phi^{b\ell}_{i,n_y+1/2,k} &\equiv \mathcal{R}_\ell\left[\mathcal{D}^{1y}_{\ell+1} \phi^{\mathcal{N}(b,+y),\ell+1}_{2i-1,1/2,2k-1}\right] && (15.40)\\
\mathcal{D'}^{1y}_\ell \phi^{b\ell}_{i,1/2,k} &\equiv \mathcal{R}_\ell\left[\mathcal{D}^{1y}_{\ell+1} \phi^{\mathcal{N}(b,-y),\ell+1}_{2i-1,n_y+1/2,2k-1}\right] && (15.41)\\
\mathcal{D'}^{1z}_\ell \phi^{b\ell}_{ij,n_z+1/2} &\equiv \mathcal{R}_\ell\left[\mathcal{D}^{1z}_{\ell+1} \phi^{\mathcal{N}(b,+z),\ell+1}_{2i-1,2j-1,1/2}\right] && (15.42)\\
\mathcal{D'}^{1z}_\ell \phi^{b\ell}_{ij,1/2} &\equiv \mathcal{R}_\ell\left[\mathcal{D}^{1z}_{\ell+1} \phi^{\mathcal{N}(b,-z),\ell+1}_{2i-1,2j-1,n_z+1/2}\right] . && (15.43)
\end{aligned}$$

The $\mathcal{D'}^1$ operators are used only where the neighboring block is at the next higher level of refinement. In these expressions $\mathcal{R}_\ell$ denotes the restriction operator that operates between levels ℓ and $\ell+1$. This is supplied along with the prolongation operator $\mathcal{I}_\ell$ by the AMR package. The relaxation operator is a Gauss-Seidel with Red-Black ordering iteration, while the coarse-grid solver applies the relaxation operator until convergence to within some threshold is attained.

Here are the steps in the multigrid algorithm:

1. Begin by initializing the solution, correction, and residual arrays to zero, if this is the first time the solver has been called. Otherwise, use the previous solution as our initial guess.

2. Compute the residual $R^{b\ell}_{ijk} = S^{b\ell}_{ijk} - \mathcal{D}^2_\ell \phi^{b\ell}_{ijk}$ on all leaf blocks. Compute the discrete L2 norm of the residual and the source.

3. Repeat the following steps until the ratio of the residual and source norms drops below some threshold, or until we have repeated some number of times.

4. Zero the correction C on the highest level of refinement, $\ell_{\max}$.

5. For each level ℓ from $\ell_{\max}$ down to 2:

   (a) Copy the solution $\phi^{b\ell}_{ijk}$ to a temporary variable $\tau^{b\ell}_{ijk}$.
   (b) Zero the correction variable $C^{b,\ell-1}_{ijk}$.
   (c) Apply the relaxation operator several times to the correction equation on level ℓ: $\mathcal{D}^2_\ell C^{b\ell}_{ijk} = R^{b\ell}_{ijk}$.
   (d) Add the correction $C^{b\ell}_{ijk}$ to the solution $\phi^{b\ell}_{ijk}$.
   (e) Compute the residual of the correction equation on all blocks (leaf or not) on level ℓ. Restrict this residual to $R^{b,\ell-1}_{ijk}$.
   (f) Compute the residual of the source equation on all leaf blocks of level $\ell - 1$ and leave the result in $R^{b,\ell-1}_{ijk}$.

6. Solve the correction equation on the coarsest level, applying the external boundary conditions. Correct the solution on the coarsest level.

7. For each level ℓ from 2 up to $\ell_{\max}$:

   (a) Prolongate the correction from level $\ell - 1$ and add the result to $C^{b\ell}_{ijk}$.
   (b) Replace $R^{b\ell}_{ijk}$ with the residual of the correction equation.
   (c) Zero a second temporary variable $\upsilon^{b\ell}_{ijk}$ on levels $\ell - 1$ and ℓ.
   (d) Apply the relaxation operator several times to $\mathcal{D}^2_\ell \upsilon^{b\ell}_{ijk} = R^{b\ell}_{ijk}$.
   (e) Add $\upsilon^{b\ell}_{ijk}$ to the correction $C^{b\ell}_{ijk}$.
   (f) Copy $\tau^{b\ell}_{ijk}$ back into $\phi^{b\ell}_{ijk}$ on all leaf blocks.
   (g) Add the correction $C^{b\ell}_{ijk}$ to the solution $\phi^{b\ell}_{ijk}$ on all leaf blocks.

8. Compute the residual $R^{b\ell}_{ijk} = S^{b\ell}_{ijk} - \mathcal{D}^2_\ell \phi^{b\ell}_{ijk}$ on all leaf blocks. Compute the discrete L2 norm of the residual.

The external boundary conditions accepted by the multigrid algorithm are Dirichlet, given-value, Neumann, and periodic boundaries. However, often isolated boundary conditions are desired. This means that the source ρ is assumed to be zero outside of the computational volume, and that the potential ϕ tends smoothly to zero at arbitrarily large distances. In order to accommodate this type of boundary condition, we use a variant of James' (1978) method. The steps are as follows:

1. Using the multigrid solver, compute a solution to the Poisson equation with Dirichlet boundaries. Call the solution $\phi_{\rm zb}$.

2. Assume that $\phi_{\rm zb} = 0$ everywhere outside the computational domain. Compute the image source distribution implied by $\phi_{\rm zb}$ under the assumption that no image source exists outside the surface of the domain. The image source lies on the surface of the domain and has a surface density $\sigma(\mathbf{x}_{\rm s}) = \mathbf{n}(\mathbf{x}_{\rm s}) \cdot \nabla\phi_{\rm zb}(\mathbf{x}_{\rm s})$, where $\mathbf{x}_{\rm s}$ is a point on the surface and $\mathbf{n}(\mathbf{x}_{\rm s})$ is a unit vector normal to the surface. For example, on the $+x$ boundary, the image surface density is (accounting for the fact that $\phi_{\rm zb}$ is a zone-averaged quantity) $\sigma^{+x}_{jk} = [7(\phi_{\rm zb})^{b1}_{n_x,jk} - (\phi_{\rm zb})^{b1}_{n_x-1,jk}]/2\Delta x_1$.

3. Using a variant of the multipole Poisson solver, compute the boundary face averages (not zone averages) of the image potential. The image source distribution is treated as a source field $S(\mathbf{x}) = \sigma(\mathbf{x})\delta(\mathbf{x}-\mathbf{x}_\mathrm{s})$.
4. Using the multigrid solver, compute a solution to the Laplace equation with the boundary values computed in the previous step. Call this solution ϕ_im.
5. The solution $\phi = \phi_\mathrm{zb} - \phi_\mathrm{im}$.

15.2.3 Using the Poisson solvers

The `poisson` sub-module solves the Poisson equation (15.2). Two different elliptic solvers are supplied with FLASH: a multipole solver, suitable for approximately spherical source distributions, and a multigrid solver, which can be used with general source distributions. The multipole solver accepts only isolated boundary conditions, whereas the multigrid solver supports Dirichlet, given-value, Neumann, periodic, and isolated boundary conditions. Boundary conditions for the Poisson solver are specified using an argument to the `poisson()` routine which can be set from different runtime parameters depending on the physical context in which the Poisson equation is being solved. The `poisson()` routine is the primary entry point to the Poisson solver module and has the following interface

`call poisson (`*pot_var*`,` *src_var*`,` *bnd_cond*`,` *alpha*`)` ,

where *pot_var* and *src_var* are the integer-valued database key numbers of the solution and source (density) variables, respectively. *bnd_cond* is an integer specifying the type of boundary conditions to employ. The following values are accepted

bnd_cond	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. Given-value boundaries are treated as Dirichlet boundaries with the boundary values subtracted from the outermost interior zones of the source; for this case the solution variable should contain the boundary values in its first layer of boundary zones on input to `poisson()`. Finally, *alpha* is real-valued and indicates the value of α multiplying the source function in eq. (15.2).

When solutions found using the Poisson solvers are to be differenced (*e.g.*, in computing the gravitational acceleration), it is strongly recommended that you use the `quadratic_cartesian` (quadratic) interpolants supplied by PARAMESH. If the interpolants supplied by the mesh are not of at least the same order as the differencing scheme used, unphysical forces will be produced at refinement boundaries. Also, using constant or linear interpolants may cause the multigrid solver to fail to converge.

15.2.3.1 Multipole

The `poisson/multipole` sub-module takes two runtime parameters, listed in Table 15.1. 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.

15.2.3.2 Multigrid

The `poisson/multigrid` sub-module is appropriate for general source distributions (*i.e.*, not necessarily spherical). In FLASH 2.2 and later the multigrid solver supports Dirichlet, Neumann, given-value, periodic, and isolated boundary conditions. All boundary conditions must be the same in each direction, though this is more a limitation of the interface than of the solver itself. The solver works in one, two, or three dimensions with Cartesian geometry.

Table 15.1: 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 poisson/multigrid sub-module has two sub-modules of its own, fftcg and isobnd_mpole. The fftcg sub-module implements a Fast Fourier Transform (FFT)-based coarse-grid solver that can be faster than the relax-to-convergence coarse-grid solver used by default. However, it requires that there be only one top-level block, as it relies on a serial transform library. The library used is Takuya Ooura's serial FFT package; it may be necessary to obtain the source for this package from his web site at

```
http://momonga.t.u-tokyo.ac.jp/~ooura/fft.html
```

If more than one top-level block is used, the Gauss-Seidel with Red-Black ordering relaxation solver is used on the coarse grid. Note that alternative coarse-grid solvers can be implemented by changing the mg_solve() routine and including the modified version in a new sub-module of poisson/multigrid.

The isobnd_mpole sub-module is actually a modified version of the multipole Poisson solver. It computes the boundary values of the image potential used when solving problems with isolated boundary conditions via the poisson_image_boundary() routine. As with the regular multipole solver, it accepts the mpole_lmax runtime parameter (Table 15.1).

In FLASH 2.2 and later, the parts of the multigrid solver that are not specific to the Poisson equation have become part of the mesh module, specifically mesh/solvers/multigrid. This module must be included when you use the solvers/poisson/multigrid module. The Config file for the multigrid Poisson solver is set up to enforce this requirement, so if you execute setup with the -auto flag, the multigrid mesh submodule should be included automatically. For example, to use the multigrid Poisson solver together with the FFT-based coarse-grid solver and isolated boundaries using multipole moments to solve the gravitational Poisson equation, your Modules file should include the following lines

```
INCLUDE gravity/poisson/multigrid
INCLUDE solvers/poisson/multigrid
INCLUDE solvers/poisson/multigrid/fftcg
INCLUDE solvers/poisson/multigrid/isobnd_mpole
INCLUDE mesh/solvers/multigrid
```

The runtime parameters which control the multigrid solver are summarized in Table 15.2. If you wish to increase the accuracy (and hence the execution time) of the solver, the first parameters to change are mgrid_max_residual_norm, which sets the termination condition for V-cycles, and mgrid_smooth_tol, which sets the termination condition for the coarse-grid iteration (if the FFT coarse-grid solver is not being used). Changing the other parameters from their default values is unlikely to help and may increase execution time. Also, if changing only mgrid_max_iter_change changes the answers you obtain, then either you have set the maximum residual norm too low (comparable to roundoff error on your computer) or there is a problem with the multigrid solver. This is because each successive V-cycle (if implemented correctly) reduces the norm of the residual by roughly the same factor until roundoff is reached. The default settings should be suitable for most applications.

Table 15.2: Runtime parameters used with `poisson/multigrid`.

Variable	Type	Default	Description
`mgrid_max_residual_norm`	real	1×10^{-6}	Maximum ratio of the norm of the residual to that of the right-hand side
`mgrid_max_iter_change`	real	1×10^{-3}	Maximum change in the norm of the residual from one iteration to the next
`mgrid_max_vcycles`	integer	100	Maximum number of V-cycles to take
`mgrid_nsmooth`	integer	4	Number of smoothing iterations to perform on each level
`mgrid_smooth_tol`	real	5×10^{-3}	Convergence criterion for the smoother
`mgrid_solve_max_iter`	integer	5000	Maximum number of iterations for solution on coarse grid

Note that the multigrid solver uses the contents of the solution variable on entry as the initial guess for the iteration. For problems in which the solver is called for one purpose (*e.g.*, gravity) during each timestep, the potential may change very little from timestep to timestep, particularly on large scales. Thus, the potential calculation for subsequent steps should take much less time than for the first step.

The multigrid solver requires several additional temporary solution variables, which are listed in Table 15.3. Most of these are of no interest to the end user, but it may occasionally be helpful to inspect `mgw2`, since this contains the residual on exit from the solver.

Table 15.3: Variables provided by `poisson/multigrid`.

Variable	Attributes	Description
`mgw1`	NOADVECT NORENORM NOCONSERVE	Work array (adjusted source)
`mgw2`	NOADVECT NORENORM NOCONSERVE	Work array (residual)
`mgw3`	NOADVECT NORENORM NOCONSERVE	Work array (correction)
`mgw4`	NOADVECT NORENORM NOCONSERVE	Work array (temporary)
`mgw5`	NOADVECT NORENORM NOCONSERVE	Work array (temporary)
`mgw6`	NOADVECT NORENORM NOCONSERVE	Work array (image potential)
`mgw7`	NOADVECT NORENORM NOCONSERVE	Work array (image mass)

Chapter 16

Runtime visualization module

FLASH contains a small module for the purposes of visualizing data in 2-d simulations. If used, the master processor will write PNG files to disk as the the simulation advances. The flexibility and image quality is much less than that of fidlr (see Chapter 20), but since the module's only requirement is the widely installed libpng, it can be easily used on most platforms. The module includes a copy of the gd graphics library (`http://www.boutell.com/gd/`).

16.1 Using the visualization module

The module (`source/visualization/native`) is not built into FLASH by default; to include it simply add it to your Modules file. It uses the gd library, distributed with FLASH in `lib/gd`. Since this is built by setup independently of FLASH, you may have to edit `lib/gd/source/Makefile`.

The output is determined entirely by the runtime parameters listed in Table 16.1. To draw more than one variable or domain at a time, every parameter name except `vis_freq` has a subscript, corresponding to a "visualization context". There are 10 such contexts available, numbered 0 through 9.

If the user does not specify certain parameters for a visualization context, reasonable defaults are used. For example, if no domain is specified, then the entire computational domain is plotted, unless the user has already defined a domain for a previous context. Likewise, if the user does not set a range for the variable to plot, we use the min/max of the variable at the beginning of the run. Currently, variables can be mapped only linearly onto the color table.

Table 16.1: Runtime parameters used with the `visualization` module.

Variable	Type	Default	Description
`vis_freq`	integer	1	Do visualization every `vis_freq` timesteps
`vis_var_#`	string	""	If not null, specify which variable in the database to draw. If null, the visualization context is not used. # is the visualization context number, in the range [0-9]. `vis_var_0` defaults to "dens".
`vis_colortable_#`	integer	0	Index in the range [0-6] specifying a colormap defined in `colortables.h`. These colormaps are the same as those available in fidlr.
`vis_drawblocks_#`	integer	0	If nonzero, draw block boundaries.
`vis_xmin_#`	real	0.	Sets lower x range of domain.
`vis_xmax_#`	real	0.	Sets upper x range of domain.
`vis_ymin_#`	real	0.	Sets lower y range of domain.
`vis_ymax_#`	real	0.	Sets upper y range of domain.
`vis_min_#`	real	0.	Lower range of linear interpolation.
`vis_max_#`	real	0.	Upper range of linear interpolation.

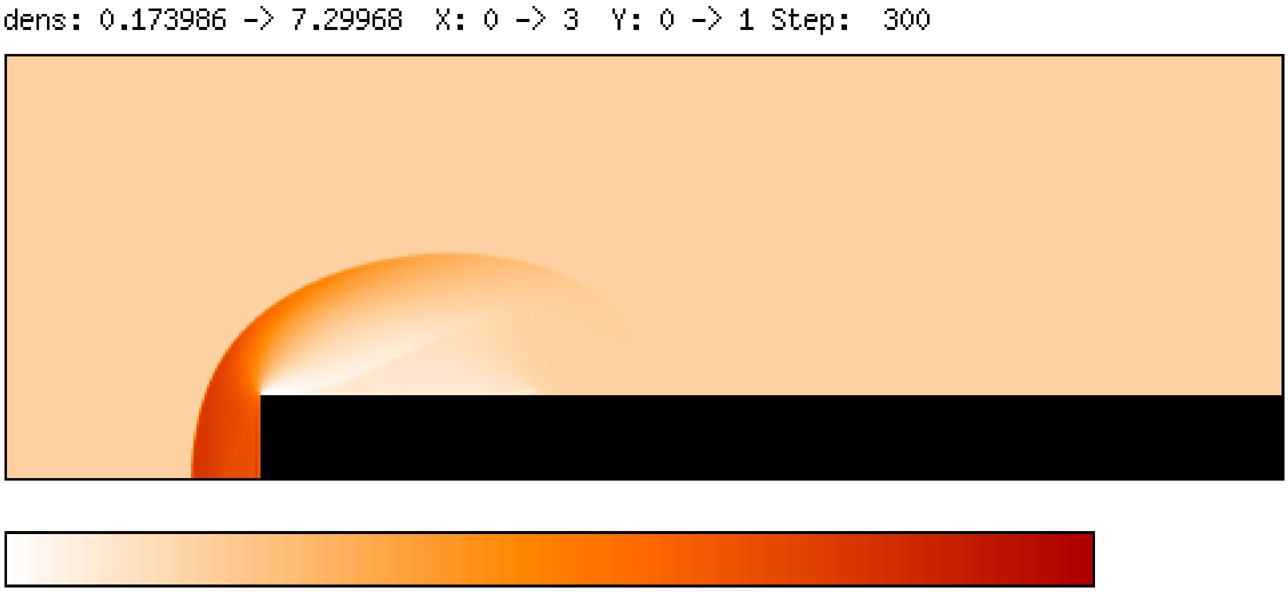


Figure 16.1: Density in the wind tunnel setup, as rendered by the runtime visualization module

Chapter 17

Utilities module

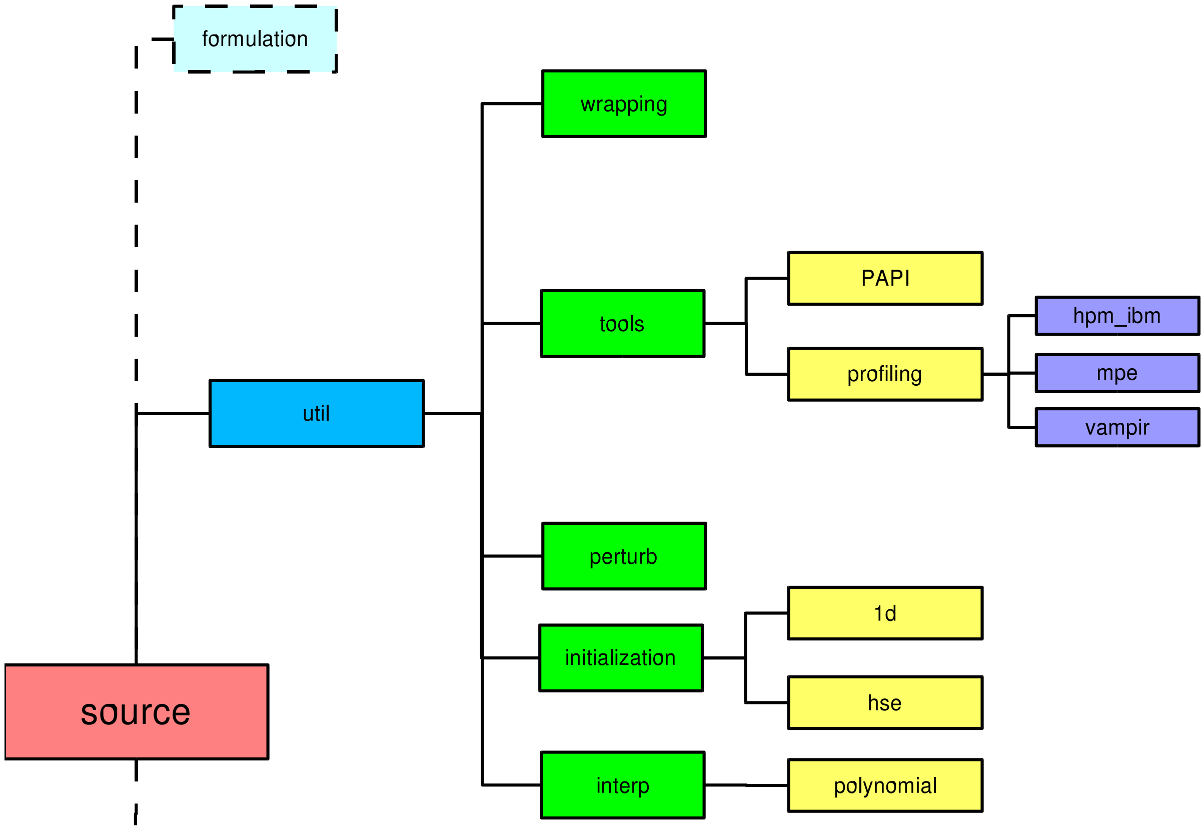


Figure 17.1: The util module directory.

The util module is a collection of reusable high-level utility functions that simplify programming in FLASH. Currently, these include problem initialization, temperature perturbation, wrapping, interpolation, performance timing, profiling, and log file maintanance.

17.1 Initialization

There are several routines available to assist you in initializing a problem. These are intended to be used in an `init_block.F90` to read in a 1-d initial model to be mapped onto the FLASH grid. The `sample_map` problem (see Sec. 18.4.1) illustrates how to use the `1d` initialization routine.

17.1.1 Reading one-dimensional initial models (`1d`)

The `1d` module is a tool that reads a 1-d initial model from a file and passes it back to the caller. This model file is required to specify the number of variables and their names in addition to the data for each point in the following format

```
# comment first -- this is my input data
number of variables =  24
dens
temp
  .
  .
Ni56
1.e-4 1.e8 1.e9  . . .   .83
2.e-4 1.e8 1.1e9  . . .   .82
  .
  .
  .
```

The first line of this file is a comment and is ignored by the reader. The second line specifies the number of variables contained in the input file—this does not have to equal the number of variables in FLASH. Next, we list the variable names, one per line, using the same names that FLASH recognizes (*i.e.* any of the names that appear in `VARIABLE` lines in the `Config` files). When the model is read in, these names are compared to the names of the variables that FLASH understands. Any variables contained in FLASH that are absent from the input file will be initialized to 0 (a warning will be printed to stdout). Any variables contained in the input file that are not defined in FLASH will be ignored.

The variables can appear in any order in the input file. `init_1d` will use the order of the variable name lines to determine which variable is which. After the variable name lines comes the data. Each line represents one point in the initial model. The spatial coordinate is given in the first column, followed by the variables in the same order as their declaration above. The initial model will be read until an `EOF` is encountered.

The data is passed through the argument list to the caller

```
integer, parameter :: n1d_max = 16384
integer :: n1d_model
real :: xzn(n1d_max), model_1d(n1d_max,nvar)

call init_1d(n1d_max, n1d_model, xzn, model_1d)
```

`n1d_max` is the maximum number of points in the initial model that you've allocated space for. If the initial model contains more points than this, an error will result, and FLASH will abort. `n1d_model` is the actual number of points read in and is returned to the caller.

The data is returned through `xzn` and `model_1d`. `xzn(i)` contains the coordinate of point `i` in the initial model. `model_1d(i,j)` contains the value of variable `j` for point `i` in the initial model. The variables are mapped from the order that they are stored in the initial model file to the order that they are stored in the data structures in FLASH. Therefore, we can use `dBaseKeyNumber` to access a particular variable from the model file. For example, provided density is defined in FLASH,

```
idens = dBaseKeyNumber('dens')
zone_dens = model_1d(i, idens)
```

stores the density in point `i` of the initial model in `zone_dens`. This allows you to initialize the solution variable in your `init_block` from your initial model. See the `sample_map` problem for examples.

17.1.2 Reading hydrostatic 1D initial models (hse)

The hse initialization routine is a variant of 1d. It has basically the same job—read in an initial model and return the data to the caller in a manner consistent with the data layout of FLASH. The difference is that hse will restore hydrostatic equilibrium to the initial model before returning the data to the caller. This will only work with the constant gravity modules (basically anything that provides a GravAccelOneZone function).

The data is read in as above and mapped to the same ordering as FLASH. Once the file is read in, we interpolate it onto a new grid whose zone spacing is equivalent to the finest FLASH zone. This is important, since it eliminates errors from mapping the new, HSE 1-d model onto a grid. If subsample_factor is set, then we map onto a grid that is subsample_factor times larger than the finest uniform grid. This is useful for averaging the result onto the FLASH grid to better approximate cell averaged quantities.

Next we tweak the structure of the density to yield a model that is in HSE on this grid with the EOS used by FLASH. There are several ways we can force the model into HSE. Remember that, since PPM is a finite volume scheme and thus carries around cell averaged quantities, the unknowns are not associated with a particular point in the zone. One point in the initial model needs to be defined as the reference point—we take its density, temperature, and composition as correct and use it as the basis for putting the other points into HSE. If reference_zone is set to 'base', then the bottommost zone is the reference. If it is set to 'fuel', then we use the bottom-most point of the fuel layer as the reference zone. The fuel layer is defined by the point where the composition specified by fuel_marker first goes above the value fuel_threshold.

The density structure is adjusted by integrating the equation of hydrostatic equilibrium outward from this reference point. A second-order method

$$\langle P\rangle_{+1} - \langle P\rangle_0 = \frac{g\delta}{2}\left(\langle\rho\rangle_{+1} + \langle\rho\rangle_0\right) \quad , \tag{17.1}$$

is used if hse_method = 1. A third-order differencing

$$\langle P\rangle_{+1} - \langle P\rangle_0 = \frac{g\delta}{12}\left(5\,\langle\rho\rangle_{+1} + 8\,\langle\rho\rangle_0 - \langle\rho\rangle_{-1}\right) \quad . \tag{17.2}$$

is used if hse_method = 4. Here, δ is the mesh spacing. Derivations and further explanations of this procedure is contained in Zingale *et al.* (2002).

17.2 Introducing temperature perturbations (perturb)

The perturb module allows the user to put a temperature perturbation on the grid to initiate nuclear burning. The perturbation can be done at either constant density or isobarically and in several different shapes (currently Gaussian, tophat, and truncated Gaussian perturbations are supported).

The general usage of the perturb module is

```
use perturbLib

call perturbTemp(xctr, yctr, zctr, radius_x, radius_y, radius_z,  &
                 temp_perturb, pert_shape,                        &
                 pert_type, block_no)
```

where xctr, yctr, and zctr are the coordinates of the center of the perturbation. radius_x, radius_y, and radius_z are the radius of the perturbation in each coordinate direction—allowing you to create a sphere (all lengths equal) or a generalized ellipsoid. The perturbation temperature is specified by temp_perturb and is shaped according to pert_shape, which can be set equal to any one of the following

- `gaussianPert`

$$T = T_{\rm perturb} \exp\{-[(x/R_x)^2 + (y/R_y)^2 + (z/R_z)^2]\} \tag{17.3}$$

- `tophatPert`

$$T = \begin{cases} T_{\rm perturb}; & \text{if} \quad (x/R_x)^2 + (y/R_y)^2 + (z/R_z)^2 \le 1 \\ T_{\rm ambient}; & \text{if} \quad (x/R_x)^2 + (y/R_y)^2 + (z/R_z)^2 > 1 \end{cases} \tag{17.4}$$

- `truncGaussPert`

$$T = \max\{\min\{1.1 \cdot \exp\{-[(x/R_x)^2 + (y/R_y)^2 + (z/R_z)^2]\},\ T_{\rm perturb}\},\ T_{\rm ambient}\} \tag{17.5}$$

Integer keys for these are publicly available through the module.

The perturbation is done without modifying the density if `perturb_type` is set to `tempOnly`. To modify the temperature field at a constant pressure (therefore, adjusting the density), `perturb_type` should be set to `isobaric`.

17.3 Wrapping Fortran functions to be called from C (`wrapping`)

The function of the wrapping module is to create the "glue" necessary to call Fortran functions and subroutines from C. The bulk of the module is a python script named `int2API.py` which, given a file describing the Fortran functions to be wrapped, generates the necessary Fortran and C code. This description file (with extension `.int`) has to be written by hand; it essentially looks like a list of C function prototypes. For a more detailed description of the syntax, see the usage message for `int2API`.

`int2API.py` is normally called by gmake at compile-time, since we've used pattern rules to make it look like any other compiler. Given a `foo.int` file, it will create `fooAPI-bridges.F90`, `fooAPI.c` and `fooAPI.h`. This last header file is all that needs to be included by C programs in order to call the wrapped Fortran functions.

Currently, there are prototypes (and therefore a C interface) for most of the default database functions (see `source/database/amr/paramesh2.0/dBase.int`) and a few of the runtime parameter accessors. For examples of the wrapped functions in action, see the `init_block.c` files in the sod, shu_osher and wind tunnel setups; they provide exactly the same functionality as their `init_block.F90` counterparts and are used as a test of `int2API.py`. As these wrapped functions have not been heavily tested, they may well behave strangely on some platforms, although they should be sufficient for simple tasks.

17.4 Monitoring performance

FLASH includes a set of routines for monitoring performance. They come in two flavors. The default set of routines collects the timing information using the functionality provided by MPI. The second set of routines is based upon PAPI, a public domain performance tool available from the University of Tennessee. These routines measure the time and flops in the segments being monitored. To use them, PAPI must be installed on the system, whereas the default set needs no extra software. The PAPI-based routines work well on some machines (and not so well on others) and therefore, must be used with caution. To invoke normal performance routines, the `Modules` file must include `util/tools` and for PAPI-based performance routines, `util/tools/PAPI`. The discussion in the rest of the section holds true for both sets, since the interfaces and functionality are identical.

The performance routines start or stop a timer at the beginning or end of the routine(s) to be monitored and accumulate performance information in dynamically assigned accounting segments. At the completion of the program, the routines write out a performance summary. We note that these routines are not recommended for use in timing very short segments of code due to the overhead in accounting.

All of the source code for the performance monitoring can be found in the fortran module file `perfmon.F90`. The list below contains the performance routines along with a short description of each. Many of the subroutines are overloaded to take either a module name or an integer index.

- `timer_init ()`

  Initializes the performance accounting database. Calls system time routines to subtract out their initialization overhead.

- `timer_create (module,id)`

  Creates a timer and returns a unique integer index for the timer.

- `timer_start(module)`

  Subroutine that begins monitoring code module `module` or module associated with index `id`. If `module` is not associated with a previously assigned accounting segment, the routine creates one, whereas if `id` is not associated with one, then nothing is done. The parameter `module` is specified with a string (max 30 characters). Calling `timer_start` on the same module more than once without first calling `timer_stop` causes the current timer for that module to be reset (the accumulated time in the corresponding accounting segment is not reset). Timing modules may be nested as many times as there are slots for accounting segments (see MaxModules setting). The routine may be called with an integer index instead of with the name of the module.

- `timer_stop(module)`

  Stops accumulating time in the accounting segment associated with code module `module`. If timer_stop is called for a module that does not exist or for a module that is not currently being timed, nothing happens. The routine may be called with an integer index instead of with the name of the module.

- `timer_value(module)`

  Returns the current value of the timer for the accounting segment associated with the code module `module` or referenced by index `id`. If `timer_value` is called for a module that does not exist, 0. is returned.

- `timer_reset(module)`

  Resets the accumulated time in the accounting segment corresponding to the specified code module. The routine may be called with an integer index instead of with the name of the module.

- `timer_lookup_index(module)`

  Function that returns an integer index given a string module name. The integer index can be used in any of the overloaded timer routines. If a timer name is not found, the function returns `timer_invalid`. Use of this function to obtain an integer index and subsequently calling the routines by that index rather than the string name is encouraged for performance reasons.

- `perf_summary (lun, n_period)`

  Subroutine that writes a performance summary of all current accounting segments to the file associated with logical unit number `lun`. Included is the average over `n_period` intervals (*e.g.* timesteps). If using PAPI, `mflops/sec` are also included for each segment. The accounting database is not reinitialized. `lun` and `n_period` are of default integer type. Calling `perf_summary` stops all currently running timers.

Below is a very simple example of calling the performance routines.

```
program example
use perfmon
integer i
call timer_init
do i = 1, 1000
  call timer_start ('blorg')
```

```
  call blorg
  call timer_stop ('blorg')
  call timer_start ('gloob')
  call gloob
  call timer_stop ('gloob')
enddo
call perf_summary (6, 1000)
end
```

17.5 Profiling with Jumpshot, Vampir, or IBM HPM

FLASH can be used with several external profiling toolkits: Jumpshot and Vampir to look at the MPI message traffic across different processors and IBM's Hardware Performance Monitor (HPM) to look at raw computational data. Jumpshot is freely available from Argonne National Laboratory as part of the MPE package in MPICH. Vampir is a commercial product installed at many supercomputing centers. Both Jumpshot and Vampir work in similar ways—all MPI calls are logged as the code is run, and a graphical view of time spent in each MPI routine as well as the flow of the different messages can be examined. By default, only the different MPI routines are labeled in these graphical views. HPM gathers computational data, and will report specifically on user defined sections of code.

The FLASH performance fortran module, `perfmon`, which includes the routines `timer_start` and `timer_stop`, invokes profiling. Any time a `timer_start` call is made, the appropriate Vampir, Jumpshot, or HPM 'start logging' routine is called and assigned the label that is used for the timer. The log is stopped with the `timer_stop` call. Thus, all the major blocks of code that are bracketed by `timer_start` and `timer_stop` become known to the profiling packages.

Because the machine that a user runs on may not have the necessary libraries needed to resolve these calls, this profiling is disabled by default. To enable it, the appropriate module needs to be included: `util/tools/profiling/mpe` for Jumpshot, `util/tools/profiling/vampir` for Vampir, and `util/tools/profiling/hpm_ibm` for HPM. Also, the appropriate macro needs to be defined in the `Makefile.h` (see section 22): `LIB_MPE` for Jumpshot, `LIB_VAMPIR` for Vampir, `LIB_HPM_IBM` for HPM. For example, `LIB_MPE = -L$(MPE_PATH)/lib -lmpe_f2cmpi -llmpe -lmpe`.

17.6 Log file maintenance

FLASH supplies a Fortran 90 module called `logfile` to manage the FLASH log file, which contains various types of useful information, warnings, and error messages produced by a FLASH run. User-written routines may also make use of this module as needed. 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, avoiding problems with unflushed buffers. For this reason, `logfile` routines should not be called within time-sensitive loops, as the routines will generate system calls.

An example program using the `logfile` module might appear as follows:

```
program test
use logfile
integer :: i
call create_logfile ("test.log",  "test.par", .false.)
call stamp_logfile ("beginning log file test...")
do i = 1, 10
  call open_logfile
  write (log_lun,*) 'i = ', i
  call close_logfile
enddo
call stamp_logfile ("done with log file test.")
end
```

The following routines, data types, and public constants are provided by this module.

- `create_logfile` (*name*, *parmfile*, *restart*)

  Creates the named log file and writes some header information to it, including the build stamp and the values of all runtime parameters in the global parameter context. The name of the parameter file is taken as an input; it is echoed to the log file. If *restart* is `.true.`, the file is opened in append mode.

- `stamp_logfile` (*string*)

  Write a date stamp and a specified string to the log file.

- `tstamp_logfile` (n, t, dt)

  Write a dated timestep stamp for step n, time t, timestep dt to the log file. n must be an integer, while t and dt must be reals.

- `write_logfile` (*string*)

  Write a string to the log file without a date stamp.

- `break_logfile()`

  Write a 'break' (a row of `=`) to the log file.

- `open_logfile()`

  Open the log file for writing, creating it first with a default name (`logfile`) if necessary. `open_logfile()` and `close_logfile()` should only be used if it is necessary to write something directly to the log file unit with some external routine.

- `close_logfile()`

  Close the log file.

- `log_lun`

  The logical unit number being used by the `logfile` module (to permit direct writes to the log file by external routines).

Part III

Test Cases

Chapter 18

The supplied test problems

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

18.1 Hydrodynamics test problems

18.1.1 The Sod shock-tube problem

The Sod problem (Sod 1978) is a one-dimensional flow discontinuity problem that provides a good test of a compressible code's ability to capture shocks and contact discontinuities with a small number of zones and to produce the correct profile in a rarefaction. It also tests a code's ability to correctly satisfy the Rankine-Hugoniot shock jump conditions. When implemented at an angle to a multidimensional grid, it can be used to detect irregularities in planar discontinuities produced by grid geometry or operator splitting effects.

We construct the initial conditions for the Sod problem by establishing a planar interface at some angle to the x- and y-axes. The fluid is initially at rest on either side of the interface, and the density and pressure jumps are chosen so that all three types of nonlinear, hydrodynamic waves (shock, contact, and rarefaction) develop. To the "left" and "right" of the interface we have

$$\begin{array}{rclcrcl} \rho_{\rm L} &=& 1. & \qquad & \rho_{\rm R} &=& 0.125 \\ p_{\rm L} &=& 1. & \qquad & p_{\rm R} &=& 0.1 \end{array} \tag{18.1}$$

The ratio of specific heats γ is chosen to be 1.4 on both sides of the interface.

In FLASH, the Sod problem (`sod`) uses the runtime parameters listed in Table 18.1 in addition to those supplied with the code. For this problem we use the `gamma` equation of state module and set `gamma` to 1.4. The default values listed in Table 18.1 are appropriate to a shock with normal parallel to the x-axis that initially intersects that axis at $x = 0.5$ (halfway across a box with unit dimensions).

Fig. 18.1 shows the result of running the Sod problem with FLASH on a two-dimensional grid with the analytical solution shown for comparison. The hydrodynamical algorithm used here is the directionally split piecewise-parabolic method (PPM) included with FLASH. In this run the shock normal is chosen to be parallel to the x-axis. With six levels of refinement, the effective grid size at the finest level is 256^2, so the finest zones have width 0.00390625. At $t = 0.2$, three different nonlinear waves are present: a rarefaction between $x = 0.263$ and $x = 0.486$, a contact discontinuity at $x = 0.685$, and a shock at $x = 0.850$. The two discontinuities are resolved with approximately two to three zones each at the highest level of refinement, demonstrating the ability of PPM to handle sharp flow features well. Near the contact discontinuity and

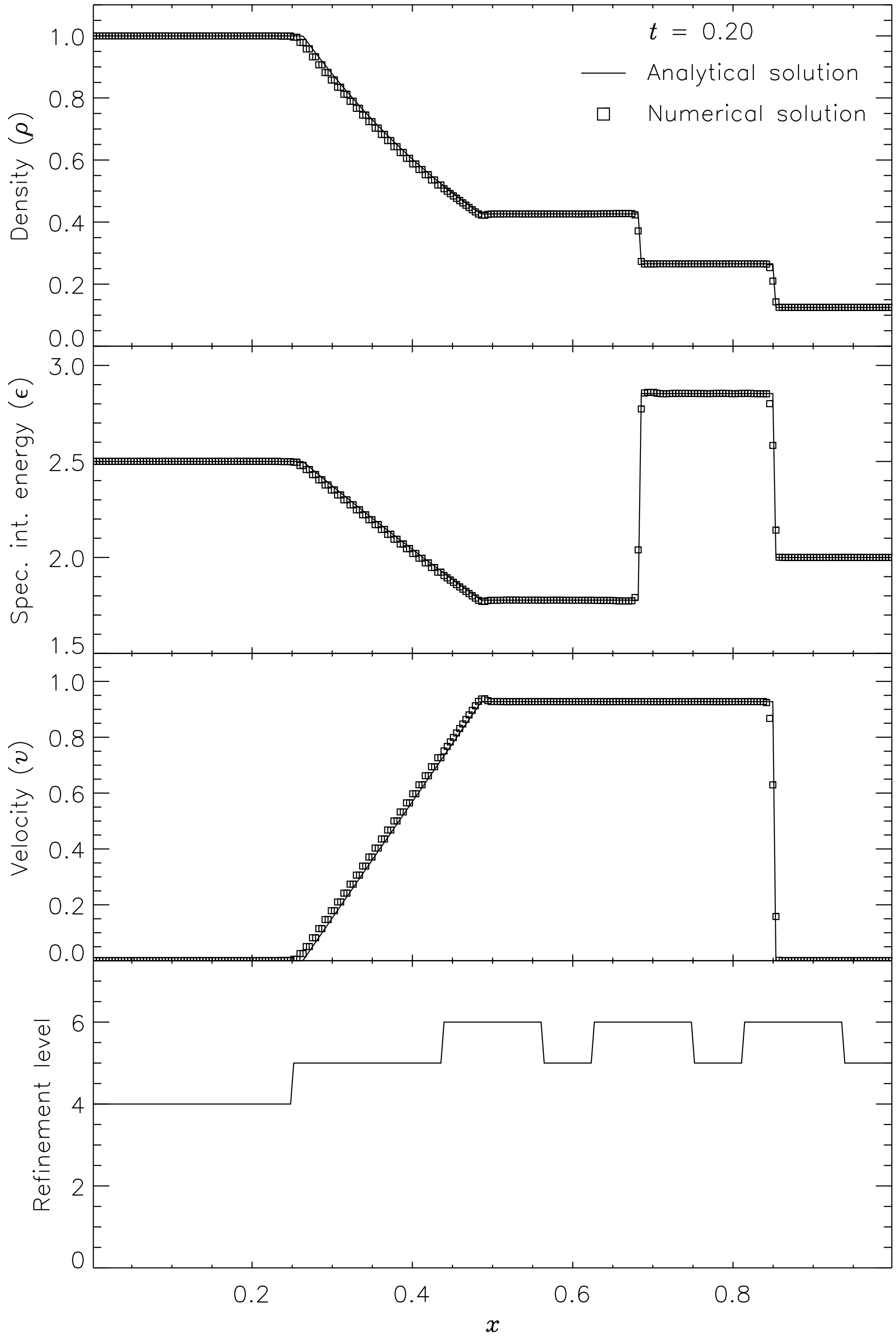


Figure 18.1: Comparison of numerical and analytical solutions to the Sod problem. A 2D grid with six levels of refinement is used. The shock normal is parallel to the x-axis.

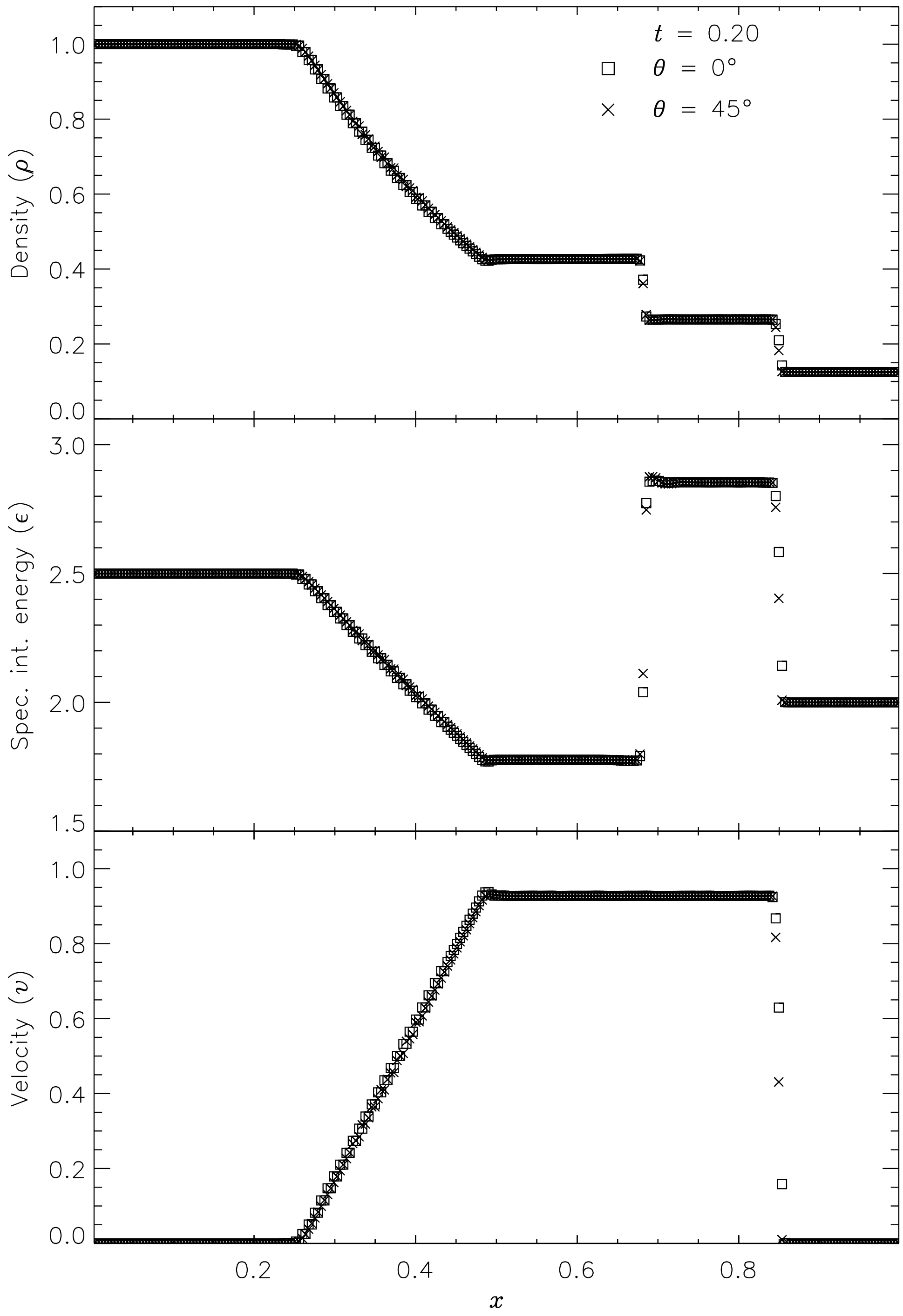


Figure 18.2: Comparison of numerical solutions to the Sod problem for two different angles (θ) of the shock normal relative to the x-axis. A 2D grid with six levels of refinement is used.

Table 18.1: Runtime parameters used with the `sod` test problem.

Variable	Type	Default	Description
`rho_left`	real	1	Initial density to the left of the interface (ρ_L)
`rho_right`	real	0.125	Initial density to the right (ρ_R)
`p_left`	real	1	Initial pressure to the left (p_L)
`p_right`	real	0.1	Initial pressure to the right (p_R)
`u_left`	real	0	Initial velocity (perpendicular to interface) to the left (u_L)
`u_right`	real	0	Initial velocity (perpendicular to interface) to the right (u_R)
`xangle`	real	0	Angle made by interface normal with the x-axis (degrees)
`yangle`	real	90	Angle made by interface normal with the y-axis (degrees)
`posn`	real	0.5	Point of intersection between the interface plane and the x-axis

in the rarefaction, we find small errors of about $1 - 2\%$ in the density and specific internal energy, with similar errors in the velocity inside the rarefaction. Elsewhere, the numerical solution is close to exact; no oscillations are present.

Fig. 18.2 shows the result of running the Sod problem on the same two-dimensional grid with different shock normals: parallel to the x-axis ($\theta = 0°$) and along the box diagonal ($\theta = 45°$). For the diagonal solution, we have interpolated values of density, specific internal energy, and velocity to a set of 256 points spaced exactly as in the x-axis solution. This comparison shows the effects of the second-order directional splitting used with FLASH on the resolution of shocks. At the right side of the rarefaction and at the contact discontinuity, the diagonal solution undergoes slightly larger oscillations (on the order of a few percent) than the x-axis solution. Also, the value of each variable inside the discontinuity regions differs between the two solutions by up to 10%. However, the location and thickness of the discontinuities is the same for the two solutions. In general, shocks at an angle to the grid are resolved with approximately the same number of zones as shocks parallel to a coordinate axis.

Fig. 18.3 presents a colormap plot of the density at $t = 0.2$ in the diagonal solution together with the block structure of the AMR grid. Note that regions surrounding the discontinuities are maximally refined, while behind the shock and contact discontinuity, the grid has de-refined, because the second derivative of the density has decreased in magnitude. Because zero-gradient outflow boundaries were used for this test, some reflections are present at the upper left and lower right corners, but at $t = 0.2$ these have not yet propagated to the center of the grid.

18.1.2 The Woodward-Colella interacting blast-wave problem

This problem was originally used by Woodward and Colella (1984) to compare the performance of several different hydrodynamical methods on problems involving strong shocks and narrow features. It has no analytical solution (except at very early times), but since it is one-dimensional, it is easy to produce a converged solution by running the code with a very large number of zones, permitting an estimate of the self-convergence rate. For FLASH, it also provides a good test of the adaptive mesh refinement scheme.

The initial conditions consist of two parallel, planar flow discontinuities. Reflecting boundary conditions are used. The density is unity and the velocity is zero everywhere. The pressure is large at the left and right and small in the center

$$p_L = 1000, \qquad p_M = 0.01, \qquad p_R = 100\,. \tag{18.2}$$

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

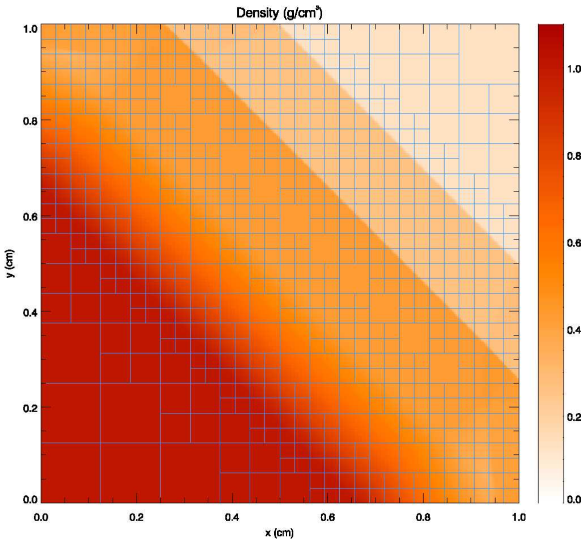


Figure 18.3: Density in the diagonal 2D Sod problem with six levels of refinement at $t = 0.2$. The outlines of AMR blocks are shown (each block contains 8×8 zones).

Fig. 18.4 shows the density and velocity profiles at several different times in the converged solution, demonstrating the complexity inherent in this problem. The initial pressure discontinuities drive shocks into the middle part of the grid; behind them, rarefactions form and propagate toward the outer boundaries, where they are reflected back into the grid. By the time the shocks collide at $t = 0.028$, the reflected rarefactions have caught up to them, weakening them and making their post-shock structure more complex. Because the right-hand shock is initially weaker, the rarefaction on that side reflects from the wall later, so the resulting shock structures going into the collision from the left and right are quite different. Behind each shock is a contact discontinuity left over from the initial conditions (at $x \approx 0.50$ and 0.73). The shock collision produces an extremely high and narrow density peak. The peak density should be slightly less than 30. However, the peak density shown in Fig. 18.4 is only about 18, since the maximum value of the density does not occur at exactly $t = 0.028$. Reflected shocks travel back into the colliding material, leaving a complex series of contact discontinuities and rarefactions between them. A new contact discontinuity has formed at the point of the collision ($x \approx 0.69$). By $t = 0.032$, the right-hand reflected shock has met the original right-hand contact discontinuity, producing a strong rarefaction, which meets the central contact discontinuity at $t = 0.034$. Between $t = 0.034$ and $t = 0.038$, the slope of the density behind the left-hand shock changes as the shock moves into a region of constant entropy near the left-hand contact discontinuity.

Fig. 18.5 shows the self-convergence of density and pressure when FLASH is run on this problem. We compare the density, pressure, and total specific energy at $t = 0.038$ obtained using FLASH with ten levels of refinement to solutions using several different maximum refinement levels. This figure plots the L1 error norm for each variable u, defined using

$$\mathcal{E}(N_{\rm ref}; u) \equiv \frac{1}{N(N_{\rm ref})} \sum_{i=1}^{N(N_{\rm ref})} \left| \frac{u_i(N_{\rm ref}) - A u_i(10)}{u_i(10)} \right| , \tag{18.3}$$

against the effective number of zones ($N(N_{\rm ref})$). In computing this norm, both the 'converged' solution $u(10)$ and the test solution $u(N_{\rm ref})$ are interpolated onto a uniform mesh having $N(N_{\rm ref})$ zones. Values of $N_{\rm ref}$ between 2 (corresponding to cell size $\Delta x = 1/16$) and 9 ($\Delta x = 1/2048$) are shown.

Although PPM is formally a second-order method, the convergence rate is only linear. This is not surprising, since the order of accuracy of a method applies only to smooth flow and not to flows containing discontinuities. In fact, all shock capturing schemes are only first-order accurate in the vicinity of discontinuities. Indeed, in their comparison of the performance of seven nominally second-order hydrodynamic

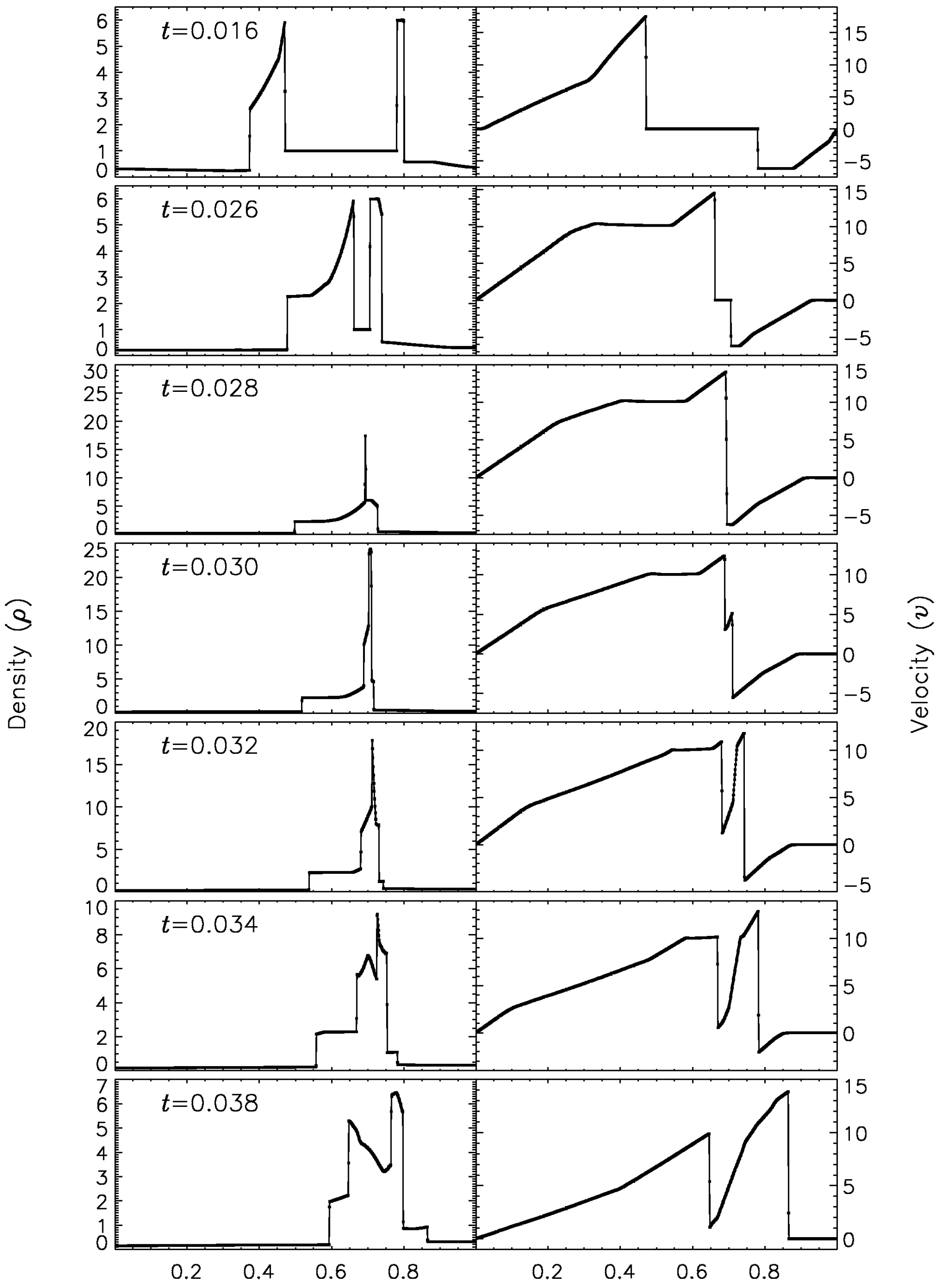


Figure 18.4: Density and velocity profiles in the Woodward-Colella interacting blast-wave problem as computed by FLASH using ten levels of refinement.

methods on this problem, Woodward and Colella found that only PPM achieved even linear convergence; the other methods were worse. The error norm is very sensitive to the correct position and shape of the strong, narrow shocks generated in this problem.

The additional runtime parameters supplied with the `2blast` problem are listed in Table 18.2. This problem is configured to use the perfect-gas equation of state (`gamma`) with `gamma` set to 1.4 and is run in a two-dimensional unit box. Boundary conditions in the y-direction (transverse to the shock normals) are taken to be periodic.

Table 18.2: Runtime parameters used with the `2blast` test problem.

Variable	Type	Default	Description
`rho_left`	real	1	Initial density to the left of the left interface (ρ_{L})
`rho_mid`	real	1	Initial density between the two interfaces (ρ_{M})
`rho_right`	real	1	Initial density to the right of the right interface (ρ_{R})
`p_left`	real	1000	Initial pressure to the left (p_{L})
`p_mid`	real	0.01	Initial pressure in the middle (p_{M})
`p_right`	real	100	Initial pressure to the right (p_{R})
`u_left`	real	0	Initial velocity (perpendicular to interface) to the left (u_{L})
`u_mid`	real	0	Initial velocity (perpendicular to interface) in the middle (u_{M})
`u_right`	real	0	Initial velocity (perpendicular to interface) to the right (u_{R})
`xangle`	real	0	Angle made by interface normal with the x-axis (degrees)
`yangle`	real	90	Angle made by interface normal with the y-axis (degrees)
`posnL`	real	0.1	Point of intersection between the left interface plane and the x-axis
`posnR`	real	0.9	Point of intersection between the right interface plane and the x-axis

18.1.3 The Sedov explosion problem

The Sedov explosion problem (Sedov 1959) is another purely hydrodynamical test in which we check the code's ability to deal with strong shocks and non-planar symmetry. The problem involves the self-similar evolution of a cylindrical or spherical blast wave from a delta-function initial pressure perturbation in an otherwise homogeneous medium. To initialize the code, we deposit a quantity of energy $E = 1$ into a small region of radius δr at the center of the grid. The pressure inside this volume p_0' is given by

$$p_0' = \frac{3(\gamma - 1)E}{(\nu + 1)\pi\, \delta r^{\nu}} \ , \tag{18.4}$$

where $\nu = 2$ for cylindrical geometry and $\nu = 3$ for spherical geometry. We set $\gamma = 1.4$. In running this problem we choose δr to be 3.5 times as large as the finest adaptive mesh resolution in order to minimize effects due to the Cartesian geometry of our grid. The density is set equal to $\rho_0 = 1$ everywhere, and the pressure is set to a small value $p_0 = 10^{-5}$ everywhere but in the center of the grid. The fluid is initially at rest. In the self-similar blast wave that develops for $t > 0$, the density, pressure, and radial velocity are all

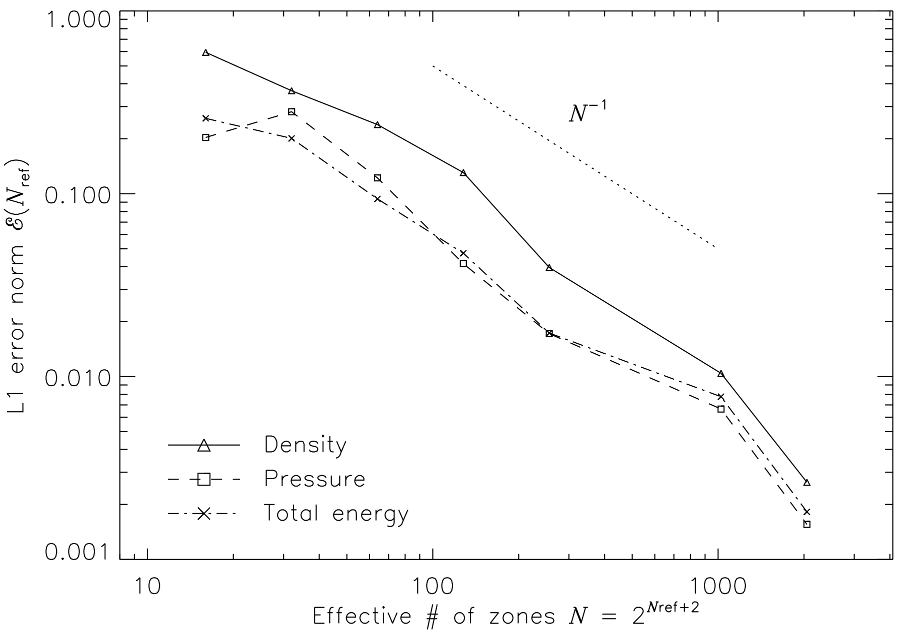


Figure 18.5: Self-convergence of the density, pressure, and total specific energy in the `2blast` test problem.

functions of $\xi \equiv r/R(t)$, where

$$R(t) = C_\nu(\gamma)\left(\frac{Et^2}{\rho_0}\right)^{1/(\nu+2)} \quad . \tag{18.5}$$

Here C_ν is a dimensionless constant depending only on ν and γ; for $\gamma = 1.4$, $C_2 \approx C_3 \approx 1$ to within a few percent. Just behind the shock front at $\xi = 1$ we have

$$\begin{aligned} \rho = \quad \rho_1 \quad &\equiv \quad \frac{\gamma+1}{\gamma-1}\rho_0 \\ p = \quad p_1 \quad &\equiv \quad \frac{2}{\gamma+1}\rho_0 u^2 \\ v = \quad v_1 \quad &\equiv \quad \frac{2}{\gamma+1}u \ , \end{aligned} \tag{18.6}$$

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

$$\begin{aligned} \rho(\xi)/\rho_1 \quad &\propto \quad \xi^{\nu/(\gamma-1)} \\ p(\xi)/p_1 \quad &= \quad \text{constant} \\ v(\xi)/v_1 \quad &\propto \quad \xi \ . \end{aligned} \tag{18.7}$$

Fig. 18.6 shows density, pressure, and velocity profiles in the two-dimensional, cylindrical Sedov problem at $t = 0.05$. Solutions obtained with FLASH on grids with 2, 4, 6, and 8 levels of refinement are shown in comparison with the analytical solution. In this figure we have computed average radial profiles in the following way. We interpolated solution values from the adaptively gridded mesh used by FLASH onto a uniform mesh having the same resolution as the finest AMR blocks in each run. Then, using radial bins with the same width as the zones in the uniform mesh, we binned the interpolated solution values, computing the average value in each bin. At low resolutions, errors show up as density and velocity overestimates behind the shock, underestimates of each variable within the shock, and a very broad shock structure. However, the central pressure is accurately determined, even for two levels of refinement. Because the density goes to a

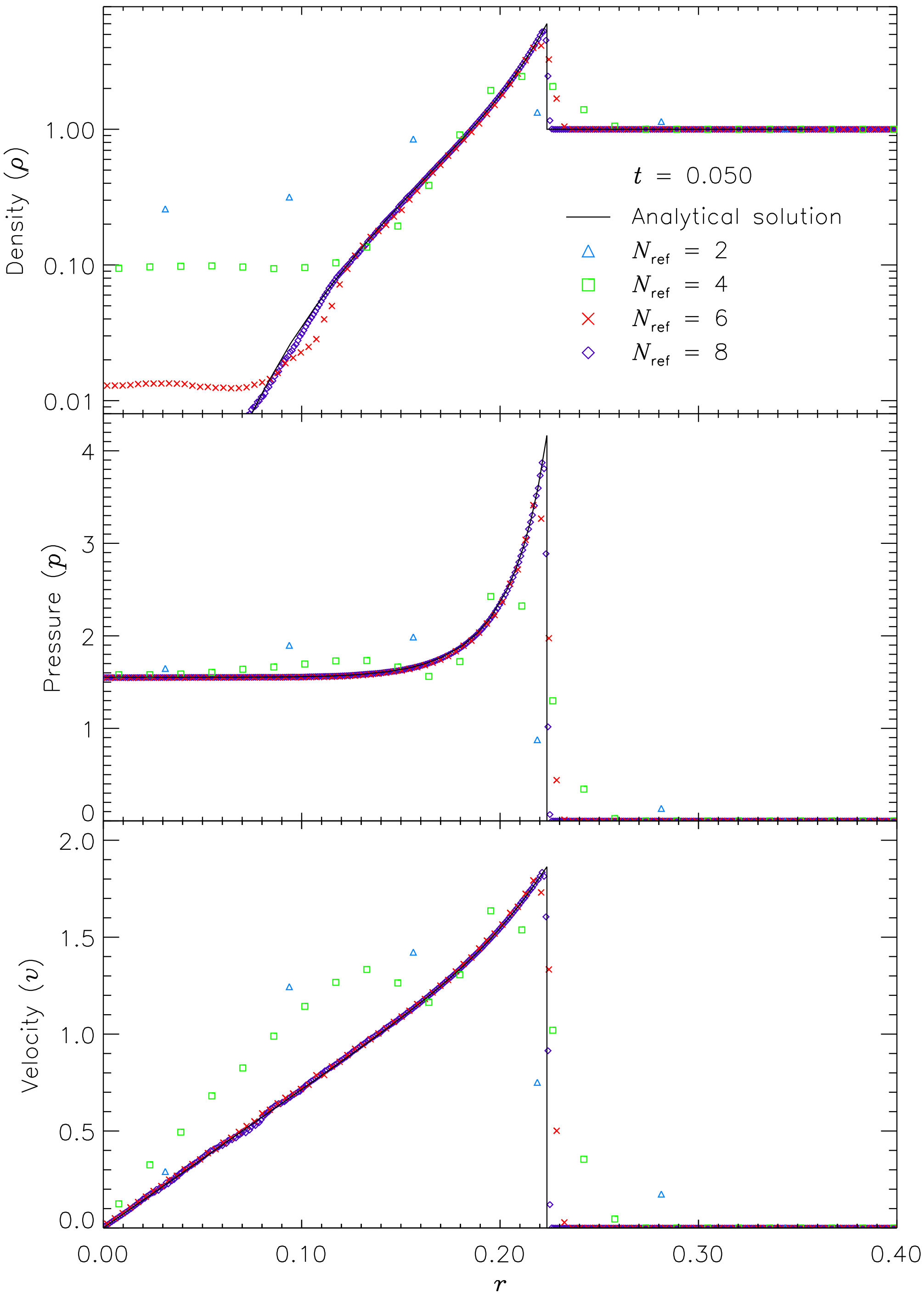


Figure 18.6: Comparison of numerical and analytical solutions to the Sedov problem in two dimensions. Numerical solution values are averages in radial bins at the finest AMR grid resolution in each run.

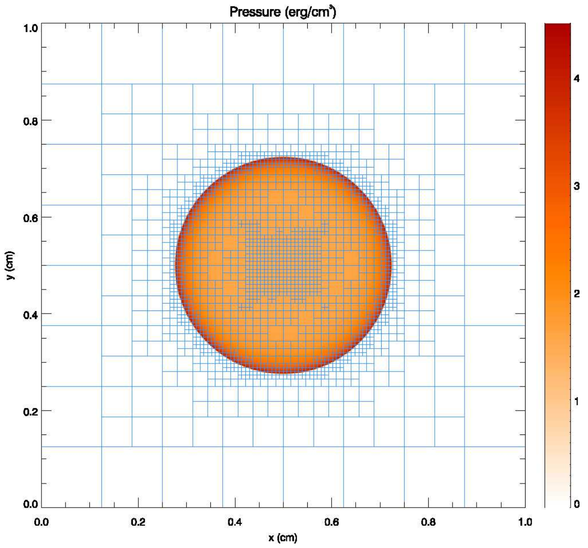


Figure 18.7: Pressure field in the 2D Sedov explosion problem with 8 levels of refinement at $t = 0.05$. The outlines of the AMR blocks are overlaid on the pressure colormap.

finite value rather than to its correct limit of zero, this corresponds to a finite truncation of the temperature (which should go to infinity as $r \to 0$). This error results from depositing the initial energy into a finite-width region rather than starting from a delta function. As the resolution improves and the value of δr decreases, the artificial finite density limit also decreases; by $N_{\rm ref} = 6$ it is less than 0.2% of the peak density. Except for the $N_{\rm ref} = 2$ case, which does not show a well-defined peak in any variable, the shock itself is always captured with about two zones. The region behind the shock containing 90% of the swept-up material is represented by four zones in the $N_{\rm ref} = 4$ case, 17 zones in the $N_{\rm ref} = 6$ case, and 69 zones for $N_{\rm ref} = 8$. However, because the solution is self-similar, for any given maximum refinement level, the structure will be four zones wide at a sufficiently early time. The behavior when the shock is underresolved is to underestimate the peak value of each variable, particularly the density and pressure.

Fig. 18.7 shows the pressure field in the 8-level calculation at $t = 0.05$ together with the block refinement pattern. Note that a relatively small fraction of the grid is maximally refined in this problem. Although the pressure gradient at the center of the grid is small, this region is refined because of the large temperature gradient there. This illustrates the ability of PARAMESH to refine grids using several different variables at once.

We have also run FLASH on the spherically symmetric Sedov problem in order to verify the code's performance in three dimensions. The results at $t = 0.05$ using five levels of grid refinement are shown in Fig. 18.8. In this figure we have plotted the average values as well as the root-mean-square (RMS) deviations from the averages. As in the two-dimensional runs, the shock is spread over about two zones at the finest AMR resolution in this run. The width of the pressure peak in the analytical solution is about 1 1/2 zones at this time, so the maximum pressure is not captured in the numerical solution. Behind the shock the numerical solution average tracks the analytical solution quite well, although the Cartesian grid geometry produces RMS deviations of up to 40% in the density and velocity in the de-refined region well behind the shock. This behavior is similar to that exhibited in the two-dimensional problem at comparable resolution.

The additional runtime parameters supplied with the `sedov` problem are listed in Table 18.3. This problem is configured to use the perfect-gas equation of state (`gamma`) with `gamma` set to 1.4 and is run in a unit box.

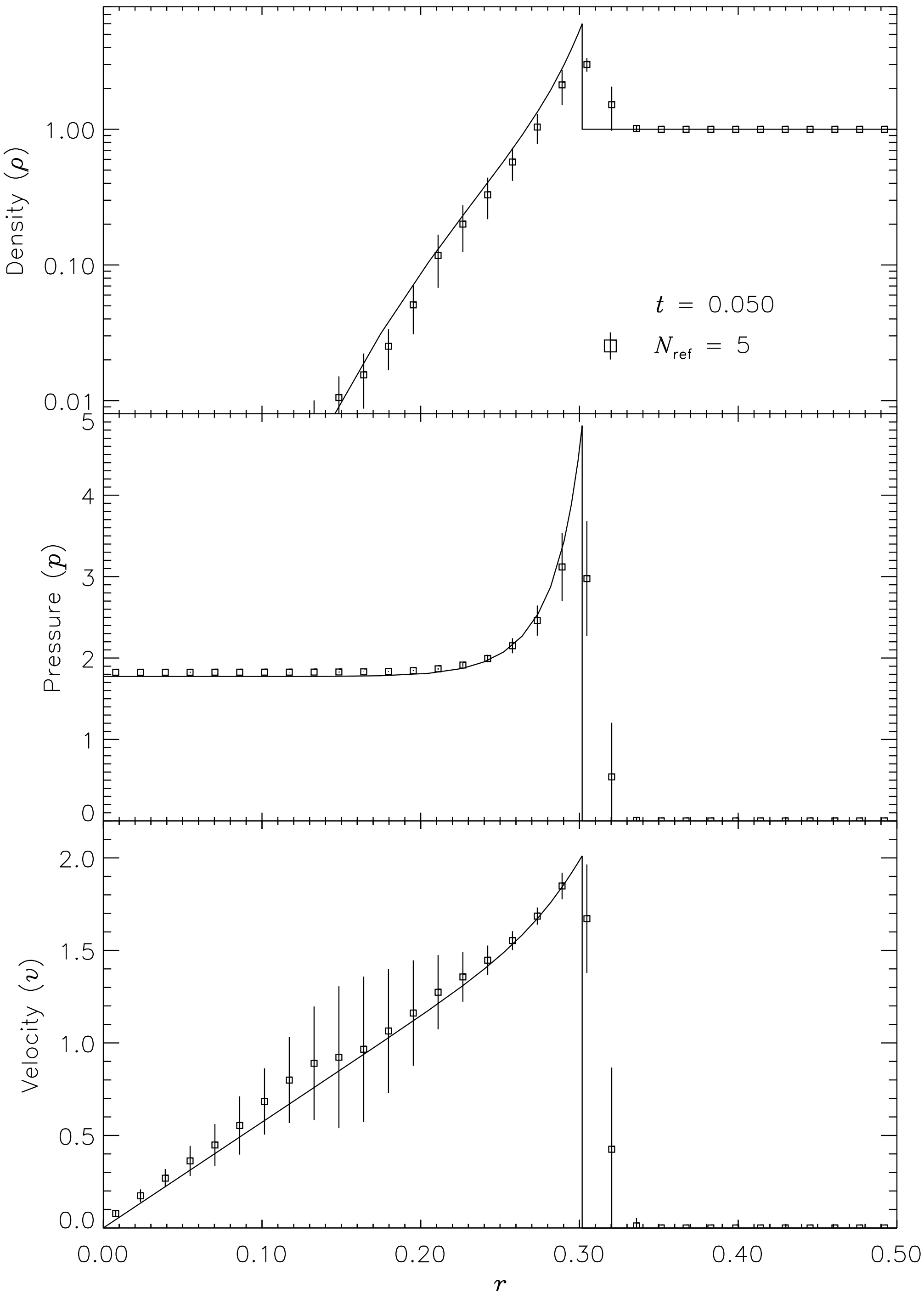


Figure 18.8: Comparison of numerical and analytical solutions to the spherically symmetric Sedov problem. A 3D grid with five levels of refinement is used.

Table 18.3: Runtime parameters used with the `sedov` test problem.

Variable	Type	Default	Description
`p_ambient`	real	10^{-5}	Initial ambient pressure (p_0)
`rho_ambient`	real	1	Initial ambient density (ρ_0)
`exp_energy`	real	1	Explosion energy (E)
`r_init`	real	0.05	Radius of initial pressure perturbation (δr)
`xctr`	real	0.5	x-coordinate of explosion center
`yctr`	real	0.5	y-coordinate of explosion center
`zctr`	real	0.5	z-coordinate of explosion center

18.1.4 The advection problem

In this problem we create a planar density pulse in a region of uniform pressure p_0 and velocity $\mathbf{u}_0$, with the velocity normal to the pulse plane. The density pulse is defined via

$$\rho(s) = \rho_1 \phi(s/w) + \rho_0 \left[1 - \phi(s/w)\right] \ , \tag{18.8}$$

where s is the distance of a point from the pulse midplane, w is the characteristic width of the pulse, and the pulse shape function ϕ is

$$\phi_{\rm SP}(\xi) = \left\{ \begin{array}{ll} 1 & |\xi| < 1 \\ 0 & |\xi| > 1 \end{array} \right. \ , \tag{18.9}$$

for a square pulse and

$$\phi_{\rm GP}(\xi) = e^{-\xi^2} \ , \tag{18.10}$$

for a Gaussian pulse. For these initial conditions, the Euler equations reduce to a single advection equation with propagation speed u_0; hence the density pulse should move across the computational volume at this speed without changing shape. Advection problems similar to this were first proposed by Boris and Book (1973) and Forester (1977).

Like the Sod problem, the advection problem tests the ability of the code to handle planar geometry and the code's treatment of contact discontinuities. In some ways, contact discontinuities are the most difficult type of hydrodynamic wave for a shock capturing code to get right. Shocks contain a self-steepening mechanism, so diffusive errors that tend to spread the shock structure do so only up to a certain limit. However, contact discontinuities are not self-steepening, so any artificial diffusion in the numerical method will continue to spread the discontinuity throughout the calculation. In addition, any numerical noise generated at a contact discontinuity tends to move with the interface, accumulating there as the calculation advances. The advection problems presented here compare the code's treatment of both leading and trailing contact discontinuities (for the square pulse) and the treatment of narrow flow features (for both the square and for the Gaussian pulse shapes). Many hydrodynamical methods have a tendency to clip narrow features or to distort pulse shapes by introducing artificial dispersion and dissipation (Zalesak 1987).

The additional runtime parameters supplied with the `advect` problem are listed in Table 18.4. This problem is configured to use the perfect-gas equation of state (`gamma`) with `gamma` set to 1.4 and is run in a unit box. The value of γ does not affect the analytical solution, but it does affect the timestep.

To demonstrate the performance of FLASH on the advection problem, we have performed tests of both the square and Gaussian pulse profiles with the pulse normal parallel to the x-axis ($\theta = 0°$) and at an angle to the x-axis ($\theta = 45°$) in two dimensions. The square pulse used $\rho_1 = 1$, $\rho_0 = 10^{-3}$, $p_0 = 10^{-6}$, $u_0 = 1$, and $w = 0.1$. With six levels of refinement in the domain $[0,1] \times [0,1]$, this value of w corresponds to having about 52 zones across the pulse width. The Gaussian pulse tests used the same values of ρ_1, ρ_0, p_0, and u_0, but with $w = 0.015625$. This value of w corresponds to about 8 zones across the pulse width at six levels of refinement. For each test, we performed runs at two, four, and six levels of refinement to examine the quality of the numerical solution as the resolution of the advected pulse improves. The runs with $\theta = 0°$ used zero-gradient (outflow) boundary conditions, while the runs performed at an angle to the x-axis used periodic boundaries.

Table 18.4: Runtime parameters used with the `advect` test problem.

Variable	Type	Default	Description
`rhoin`	real	1	Characteristic density inside the advected pulse (ρ_1)
`rhoout`	real	10^{-5}	Ambient density (ρ_0)
`pressure`	real	1	Ambient pressure (p_0)
`velocity`	real	10	Ambient velocity (u_0)
`width`	real	0.1	Characteristic width of advected pulse (w)
`pulse_fctn`	integer	1	Pulse shape function to use: 1=square wave, 2=Gaussian
`xangle`	real	0	Angle made by pulse plane with x-axis (degrees)
`yangle`	real	90	Angle made by pulse plane with y-axis (degrees)
`posn`	real	0.25	Point of intersection between pulse midplane and x-axis

Fig. 18.9 shows the advected density profiles at $t = 0.4$ compared to the analytical solution. The upper two frames of this figure depict the square pulse with $\theta = 0°$ and $\theta = 45°$, while the lower two frames depict the Gaussian pulse results. In each case the analytical density pulse has been advected a distance $u_0 t = 0.4$. In the figure the axis parallel to the pulse normal has been translated by this amount.

The advection results show the expected improvement with increasing AMR refinement level $N_{\rm ref}$. Inaccuracies appear as diffusive spreading, rounding of sharp corners, and clipping. Both in the square pulse test and in the Gaussian pulse test, diffusive spreading is limited to about one zone on either side of the pulse. For $N_{\rm ref} = 2$, the rounding of the square pulse and the clipping of the Gaussian pulse are quite severe; in the latter case, the pulse itself spans about two zones, which is the approximate smoothing length in PPM for a single discontinuity. For $N_{\rm ref} = 4$, the treatment of the square pulse is significantly better, but the amplitude of the Gaussian is still reduced by about 50%. In this case the square pulse discontinuities are still being resolved with 2–3 zones, but the zones are now a factor of 25 smaller than the pulse width. With six levels of refinement, the same behavior is observed for the square pulse, while the amplitude of the Gaussian pulse is now 93% of its initial value. The absence of dispersive effects (*i.e.* oscillations) despite the high order of the PPM interpolants is due to the enforcement of monotonicity in the PPM algorithm.

The diagonal runs are consistent with the runs which were parallel to the x-axis, with the possibility of a slight amount of extra spreading behind the pulse. However, note that we have determined density values for the diagonal runs by interpolation along the grid diagonal. The interpolation points are not centered on the pulses, so the density does not always take on its maximum value (particularly in the lowest-resolution case).

These results are consistent with earlier studies of linear advection with PPM (e.g., Zalesak 1987). They suggest that, in order to preserve narrow flow features in FLASH, the maximum AMR refinement level should be chosen so that zones are at least a factor 5–10 smaller than the narrowest features of interest. In cases in which the features are generated by shocks (rather than moving with the fluid), the resolution requirement is not as severe, since errors generated in the preshock region are driven into the shock rather than accumulating as it propagates.

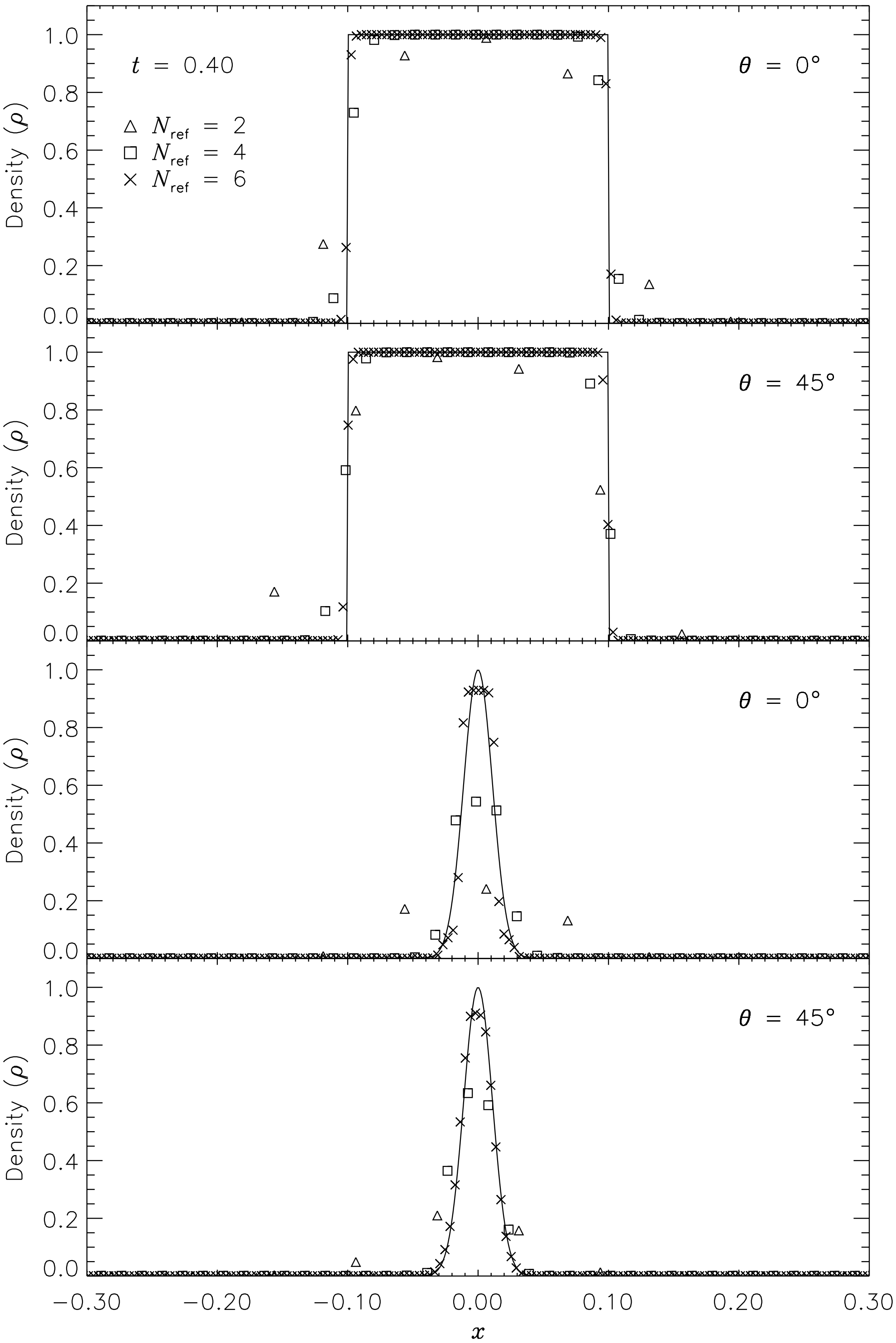


Figure 18.9: Density pulse in the advection tests for 2D grids at $t = 0.4$. Symbols represent numerical results using grids with different levels of refinement N_{ref} (2, 4, and 6).

18.1.5 The isentropic vortex problem

The two-dimensional isentropic vortex problem is often used as a benchmark for comparing numerical methods for fluid dynamics.The flowfield is smooth (there are no shocks or contact discontinuities) and contains no steep gradients, and the exact solution is known. It was studied by Yee, Vinokur, and Djomehri (2000) and by Shu (1998). In this subsection the problem is described, the FLASH control parameters are explained, and some results demonstrating how the problem can be used are presented.

The simulation domain is a square, and the center of the vortex is located at (x_{ctr}, y_{ctr}). The flowfield is defined in coordinates centered on the vortex center $(x' = x - x_{ctr}, y' = y - y_{ctr})$ with $r^2 = x'^2 + y'^2$. The domain is periodic, but it is assumed that off-domain vortices do not interact with the primary; practically, this assumption can be satisfied by ensuring that the simulation domain is large enough for a particular vortex strength. We find that a domain size of 10×10 (specified through the `driver` runtime parameters `xmin`, `xmax`, `ymin`, and `ymax`) is sufficiently large for a vortex strength (defined below) of 5.0. In the initialization below, x' and y' are the coordinates with respect to the nearest vortex in the periodic sense.

The ambient conditions are given by ρ_∞, u_∞, v_∞, and p_∞, and the nondimensional ambient temperature is $T^*_\infty = 1.0$. Using the equation of state, the (dimensional) T_∞ is computed from p_∞ and ρ_∞. Perturbations are added to the velocity and nondimensionalized temperature, $u = u_\infty + \delta u$, $v = v_\infty + \delta v$, and $T^* = T^*_\infty + \delta T^*$ according to

$$\delta u = -y' \frac{\beta}{2\pi} \exp\left(\frac{1-r^2}{2}\right) \tag{18.11}$$

$$\delta v = x' \frac{\beta}{2\pi} \exp\left(\frac{1-r^2}{2}\right) \tag{18.12}$$

$$\delta T^* = -\frac{(\gamma-1)\beta}{8\gamma\pi^2} \exp\left(1-r^2\right) \ , \tag{18.13}$$

where $\gamma = 1.4$ is the ratio of specific heats and $\beta = 5.0$ is a measure of the vortex strength. The temperature and density are then given by

$$T = \frac{T_\infty}{T^*_\infty} T^* \tag{18.14}$$

$$\rho = \rho_\infty \left(\frac{T}{T_\infty}\right)^{\frac{1}{\gamma-1}} \ . \tag{18.15}$$

At any location in space, the conserved variables (density, x- and y-momentum, and total energy) can be computed from the above quantities. The flowfield is initialized by computing cell averages of the conserved variables; in each cell, the average is approximated by averaging over `nx_subint` $\times$ `ny_subint` subintervals. The runtime parameters for the isentropic vortex problem are listed in Table 18.5.

Table 18.5: Parameters for the `isentropic_vortex` problem.

Variable	Type	Default	Description
`p_ambient`	real	1.0	Initial ambient pressure (p_∞)
`rho_ambient`	real	1.0	Initial ambient density (ρ_∞)
`u_ambient`	real	1.0	Initial ambient x-velocity (u_∞)
`v_ambient`	real	1.0	Initial ambient y-velocity (v_∞)
`vortex_strength`	real	5.0	Non-dimensional vortex strength
`xctr`	real	0.0	x-coordinate of vortex center
`yctr`	real	0.0	y-coordinate of vortex center
`nx_subint`	integer	10	number of subintervals in x-direction
`ny_subint`	integer	10	number of subintervals in y-direction

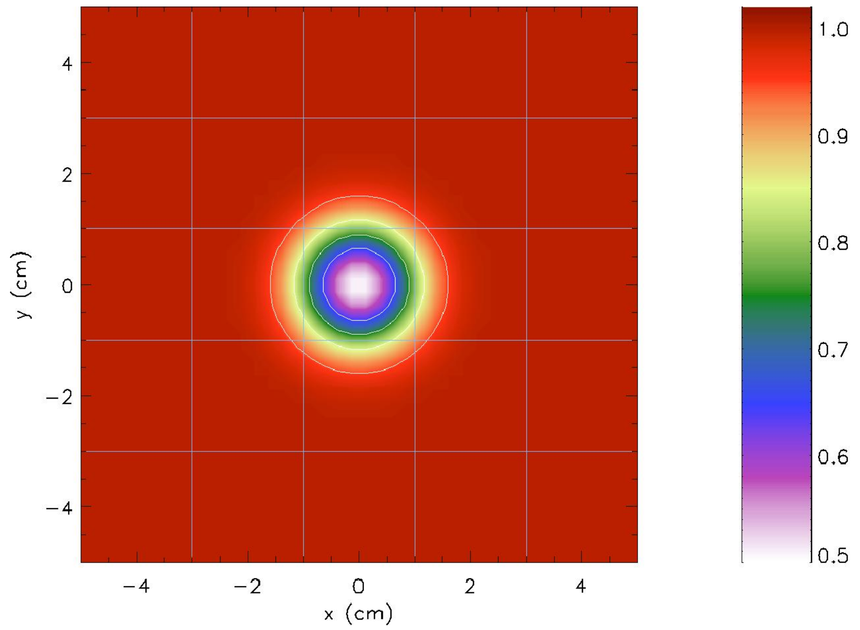


Figure 18.10: Density at $t = 0.0$ for the isentropic vortex problem. This is the initial condition and the exact solution at $t = 10.0, 20.0, \ldots$.

Fig. 18.10 shows the exact density distribution represented on a 40×40 uniform grid with $-5.0 \leq x, y \leq 5.0$. The borders of each grid block (8×8 cells) are superimposed. In addition to the shaded representation, contour lines are shown for $\rho = 0.95$, 0.85, 0.75, and 0.65. The density distribution is radially symmetric, and the minimum density is $\rho_{min} = 0.510287$. Because the exact solution of the isentropic vortex problem is the initial solution shifted by $(u_\infty t, v_\infty t)$, numerical phase (dispersion) and amplitude (dissipation) errors are easy to identify. Dispersive errors distort the shape of the vortex, breaking its symmetry. Dissipative errors smooth the solution and flatten extrema; for the vortex, the minimum in density at the vortex core will increase.

A numerical simulation using the PPM scheme was run to illustrate such errors. The simulation used the same grid shown in Fig. 18.10 with the same contour levels and color values. The grid is intentionally coarse and the evolution time long to make numerical errors visible. The vortex is represented by approximately 8 grid points in each coordinate direction and is advected diagonally with respect to the grid. At solution times of $t = 10, 20, \ldots$, *etc.*, the vortex should be back at its initial location.

Fig. 18.11 shows the solution at $t = 50.0$; only slight differences are observed. The density distribution is almost radially symmetric, although the minimum density has risen to 0.0537360. Accumulating dispersion error is clearly visible at $t = 100.0$ (Fig. 18.12), and the minimum density is now 0.601786.

Fig. 18.13 shows the density near $y = 0.0$ at three simulation times. The black line shows the initial condition. The red line corresponds to $t = 50.0$ and the blue line to $t = 100.0$. For the later two times, the density is not radially symmetric. The lines plotted are just representative profiles for those times, which give an idea of the magnitude and character of the errors.

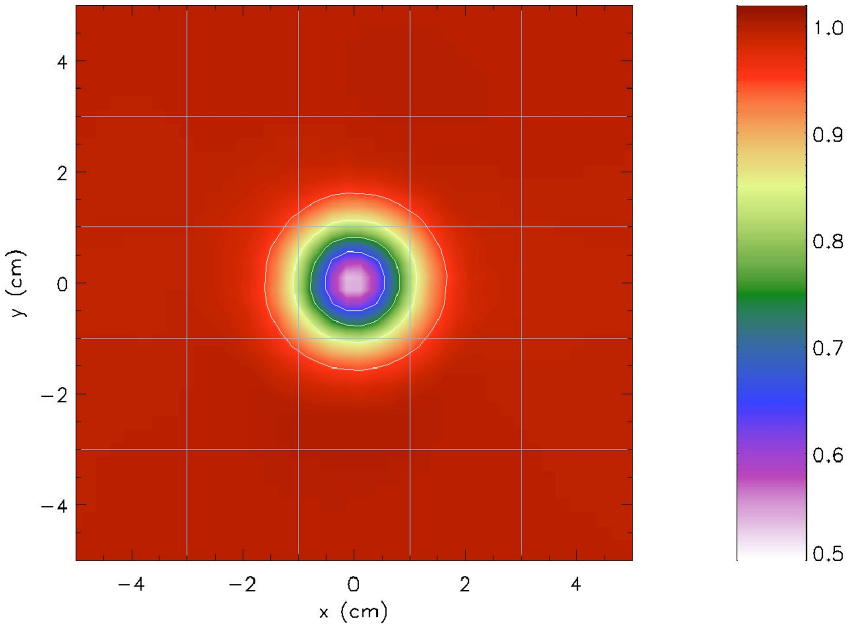


Figure 18.11: Density at $t = 50.0$ for the isentropic vortex problem.

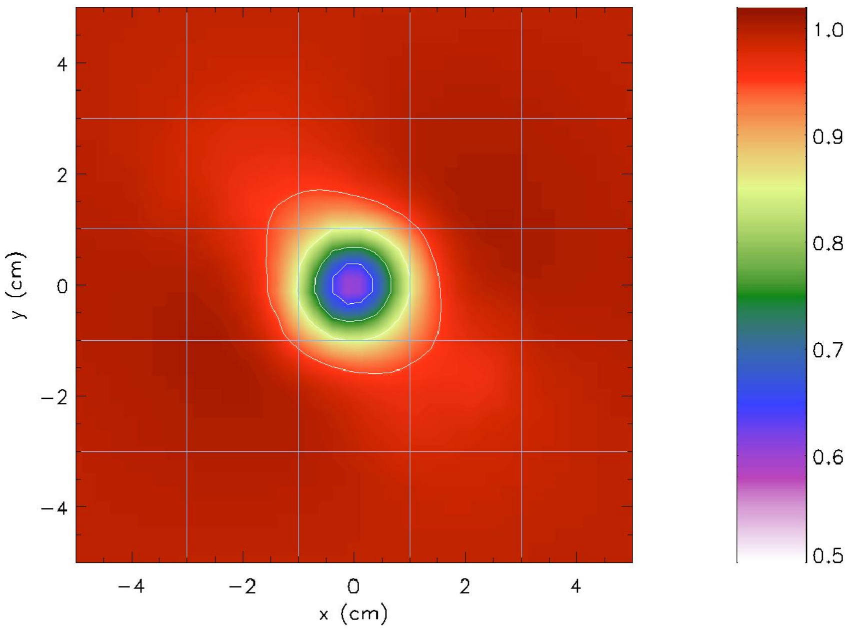


Figure 18.12: Density at $t = 100.0$ for the isentropic vortex problem.

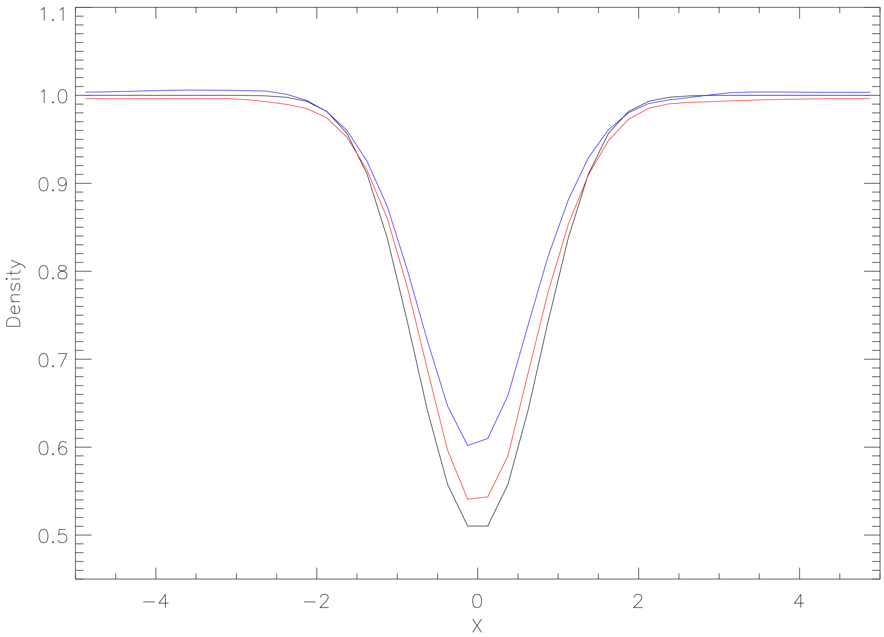


Figure 18.13: Representative density profiles for the isentropic vortex near $y = 0.0$ at $t = 0.0$ (black), $t = 50.0$ (red), and $t = 100.0$ (blue).

18.1.6 The problem of a wind tunnel with a step

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

The problem uses a two-dimensional rectangular domain three units wide and one unit high. Between $x = 0.6$ and $x = 3$ along the x-axis is a step 0.2 units high. The step is treated as a reflecting boundary, as are the lower and upper boundaries in the y-direction. For the right-hand x-boundary, we use an outflow (zero gradient) boundary condition, while on the left-hand side we use an inflow boundary. In the inflow boundary zones, we set the density to ρ_0, the pressure to p_0, and the velocity to u_0, with the latter directed parallel to the x-axis. The domain itself is also initialized with these values. We use

$$\rho_0 = 1.4, \qquad p_0 = 1, \qquad u_0 = 3\ , \qquad \gamma = 1.4, \tag{18.16}$$

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

The additional runtime parameters supplied with the `windtunnel` problem are listed in Table 18.6. This problem is configured to use the perfect-gas equation of state (`gamma`) with `gamma` set to 1.4. We also set `xmax` = 3, `ymax` = 1, `Nblockx` = 15, and `Nblocky` = 4 in order to create a grid with the correct dimensions. The version of `divide_domain` supplied with this problem adds three top-level blocks along the lower left-hand corner of the grid to cover the region in front of the step. Finally, we use `xlboundary` = −23 (user boundary condition) and `xrboundary` = −21 (outflow boundary) to instruct FLASH to use the correct boundary conditions in the x-direction. Boundaries in the y-direction are reflecting (−20).

Until $t = 12$, the flow is unsteady, exhibiting multiple shock reflections and interactions between different types of discontinuities. Fig. 18.14 shows the evolution of density and velocity between $t = 0$ and $t = 4$ (the period considered by Woodward and Colella). Immediately, a shock forms directly in front of the step

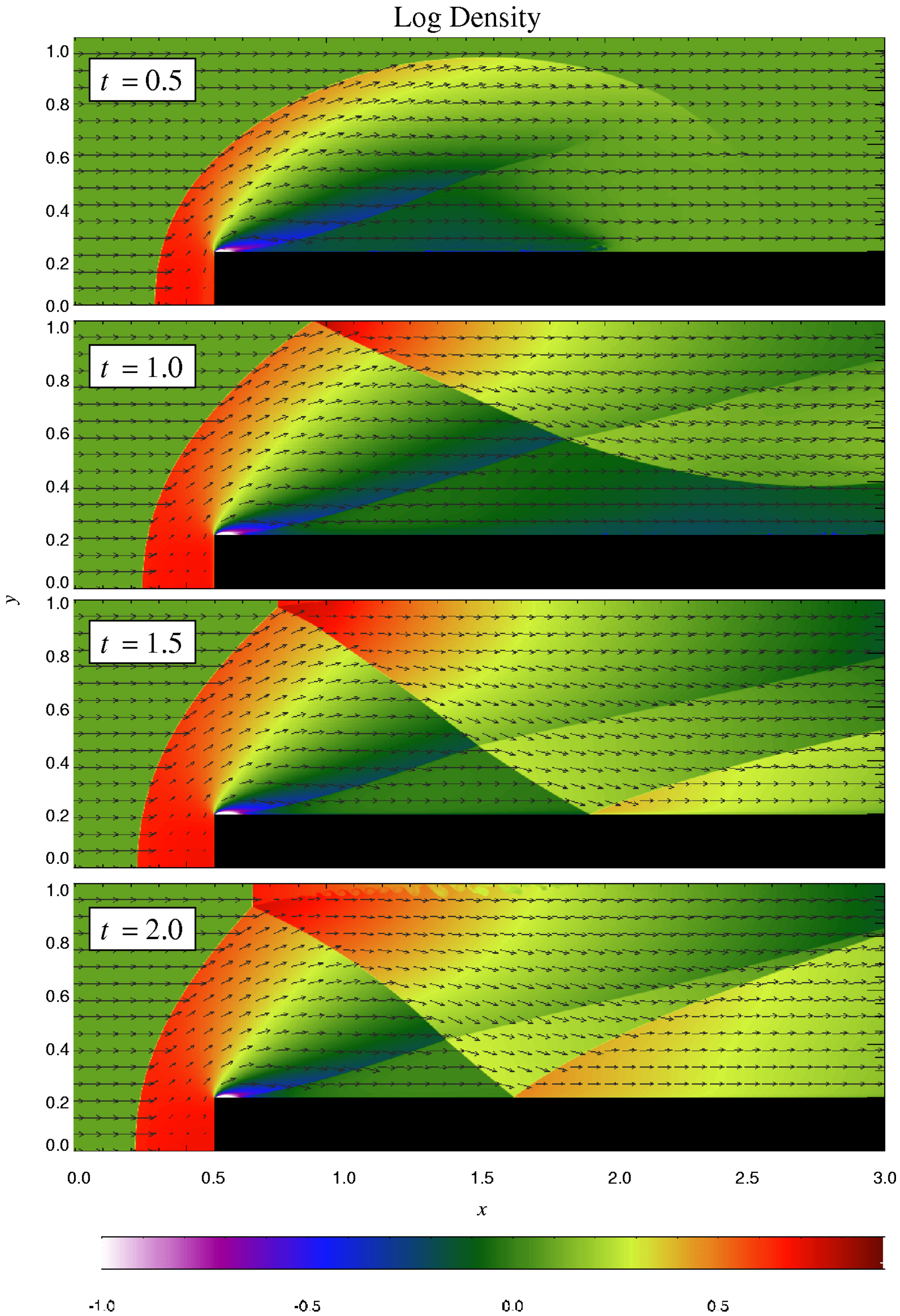


Figure 18.14: Density and velocity in the Emery wind tunnel test problem, as computed with FLASH. A 2D grid with five levels of refinement is used.

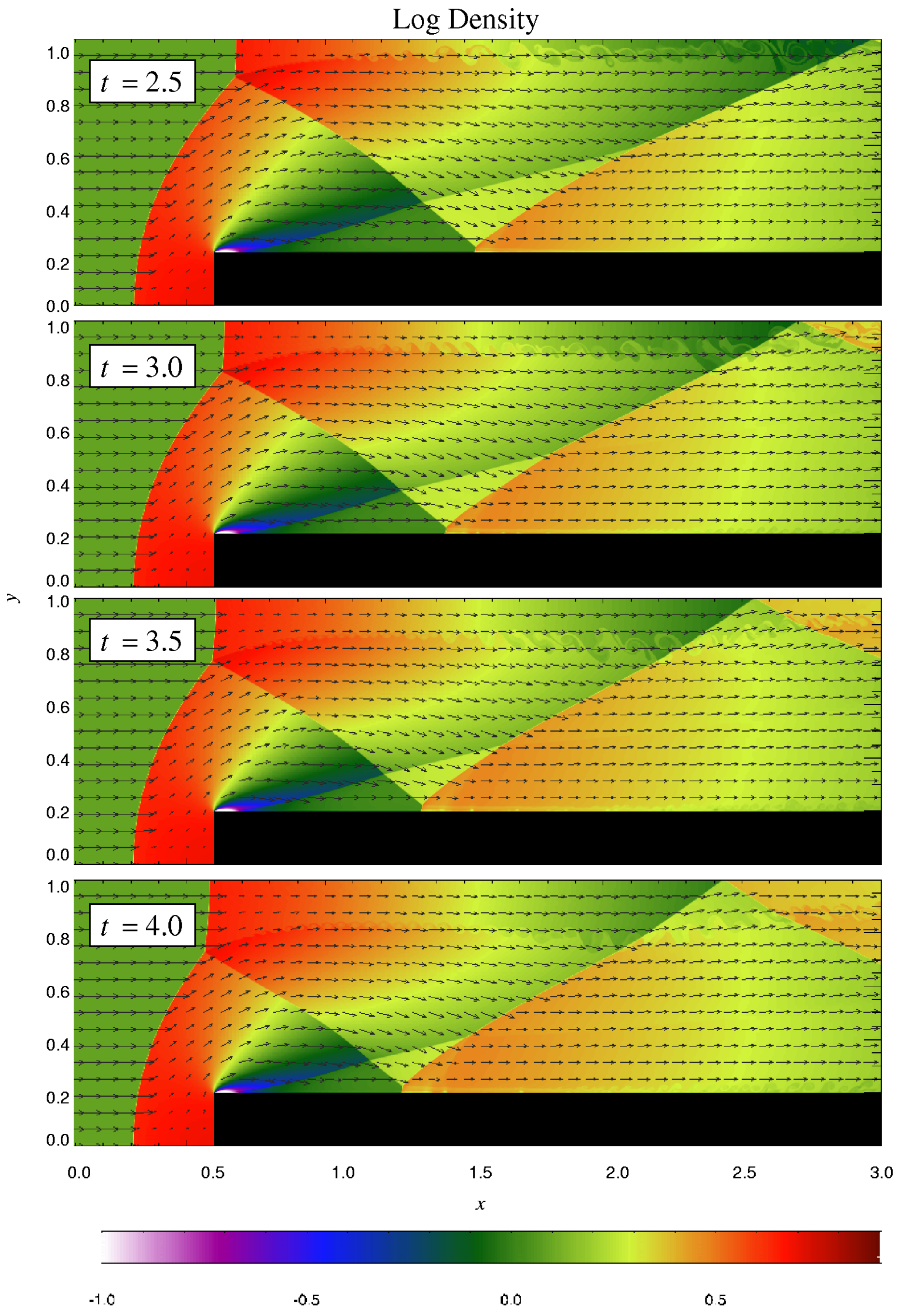


Figure 18.14: Density and velocity in the Emery wind tunnel test problem (continued).

Table 18.6: Runtime parameters used with the `windtunnel` test problem.

Variable	Type	Default	Description
`p_ambient`	real	1	Ambient pressure (p_0)
`rho_ambient`	real	1.4	Ambient density (ρ_0)
`wind_vel`	real	3	Inflow velocity (u_0)

and begins to move slowly away from it. Simultaneously, the shock curves around the corner of the step, extending farther downstream and growing in size until it strikes the upper boundary just after $t = 0.5$. The corner of the step becomes a singular point, with a rarefaction fan connecting the still gas just above the step to the shocked gas in front of it. Entropy errors generated in the vicinity of this singular point produce a numerical boundary layer about one zone thick along the surface of the step. Woodward and Colella reduce this effect by resetting the zones immediately behind the corner to conserve entropy and the sum of enthalpy and specific kinetic energy through the rarefaction. However, we are less interested here in reproducing the exact solution than in verifying the code and examining the behavior of such numerical effects as resolution is increased, so we do not apply this additional boundary condition. The errors near the corner result in a slight overexpansion of the gas there and a weak oblique shock where this gas flows back toward the step. At all resolutions we also see interactions between the numerical boundary layer and the reflected shocks that appear later in the calculation.

The shock reaches the top wall at $t \approx 0.65$. The point of reflection begins at $x \approx 1.45$ and then moves to the left, reaching $x \approx 0.95$ at $t = 1$. As it moves, the angle between the incident shock and the wall increases until $t = 1.5$, at which point it exceeds the maximum angle for regular reflection ($40°$ for $\gamma = 1.4$) and begins to form a Mach stem. Meanwhile the reflected shock has itself reflected from the top of the step, and here too the point of intersection moves leftward, reaching $x \approx 1.65$ by $t = 2$. The second reflection propagates back toward the top of the grid, reaching it at $t = 2.5$ and forming a third reflection. By this time in low-resolution runs, we see a second Mach stem forming at the shock reflection from the top of the step; this results from the interaction of the shock with the numerical boundary layer, which causes the angle of incidence to increase faster than in the converged solution. Fig. 18.15 compares the density field at $t = 4$ as computed by FLASH using several different maximum levels of refinement. Note that the size of the artificial Mach reflection diminishes as resolution improves.

The shear zone behind the first ("real") Mach stem produces another interesting numerical effect, visible at $t \geq 3$ — Kelvin-Helmholtz amplification of numerical errors generated at the shock intersection. The waves thus generated propagate downstream and are refracted by the second and third reflected shocks. This effect is also seen in the calculations of Woodward and Colella, although their resolution was too low to capture the detailed eddy structure we see. Fig. 18.16 shows the detail of this structure at $t = 3$ on grids with several different levels of refinement. The effect does not disappear with increasing resolution, for three reasons. First, the instability amplifies numerical errors generated at the shock intersection, no matter how small. Second, PPM captures the slowly moving, nearly vertical Mach stem with only 1–2 zones on any grid, so as it moves from one column of zones to the next, artificial kinks form near the intersection, providing the seed perturbation for the instability. Third, the effect of numerical viscosity, which can diffuse away instabilities on course grids, is greatly reduced at high resolution. This effect can be reduced by using a small amount of extra dissipation to smear out the shock, as discussed by Colella and Woodward (1984). This tendency of physical instabilities to amplify numerical noise vividly demonstrates the need to exercise caution when interpreting features in supposedly converged calculations.

Finally, we note that in high-resolution runs with FLASH, we also see some Kelvin-Helmholtz rollup at the numerical boundary layer along the top of the step. This is not present in Woodward and Colella's calculation, presumably because their grid resolution was lower (corresponding to two levels of refinement for us) and because of their special treatment of the singular point.

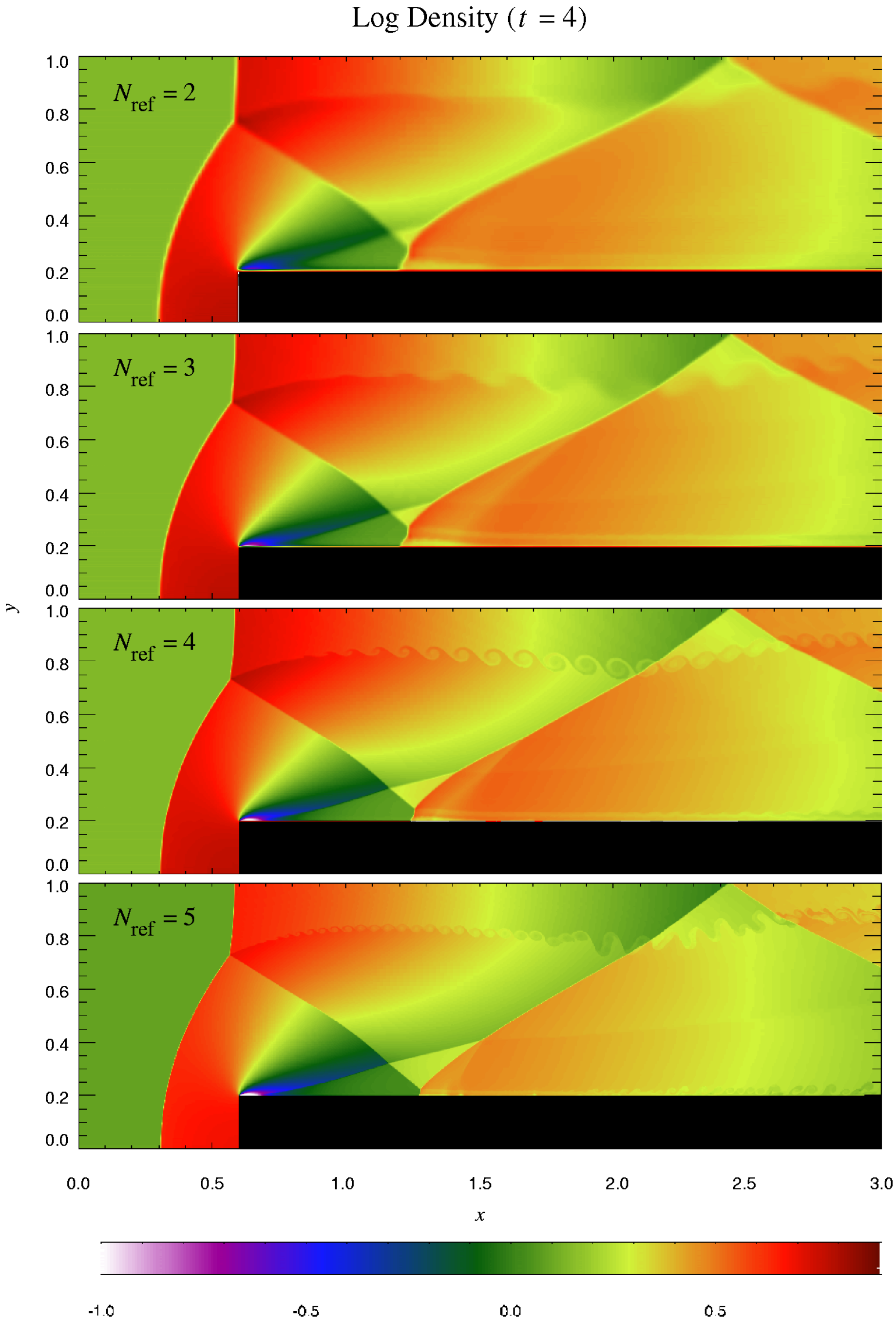


Figure 18.15: Density at t = 4 in the Emery wind tunnel test problem, as computed with FLASH using several different levels of refinement.

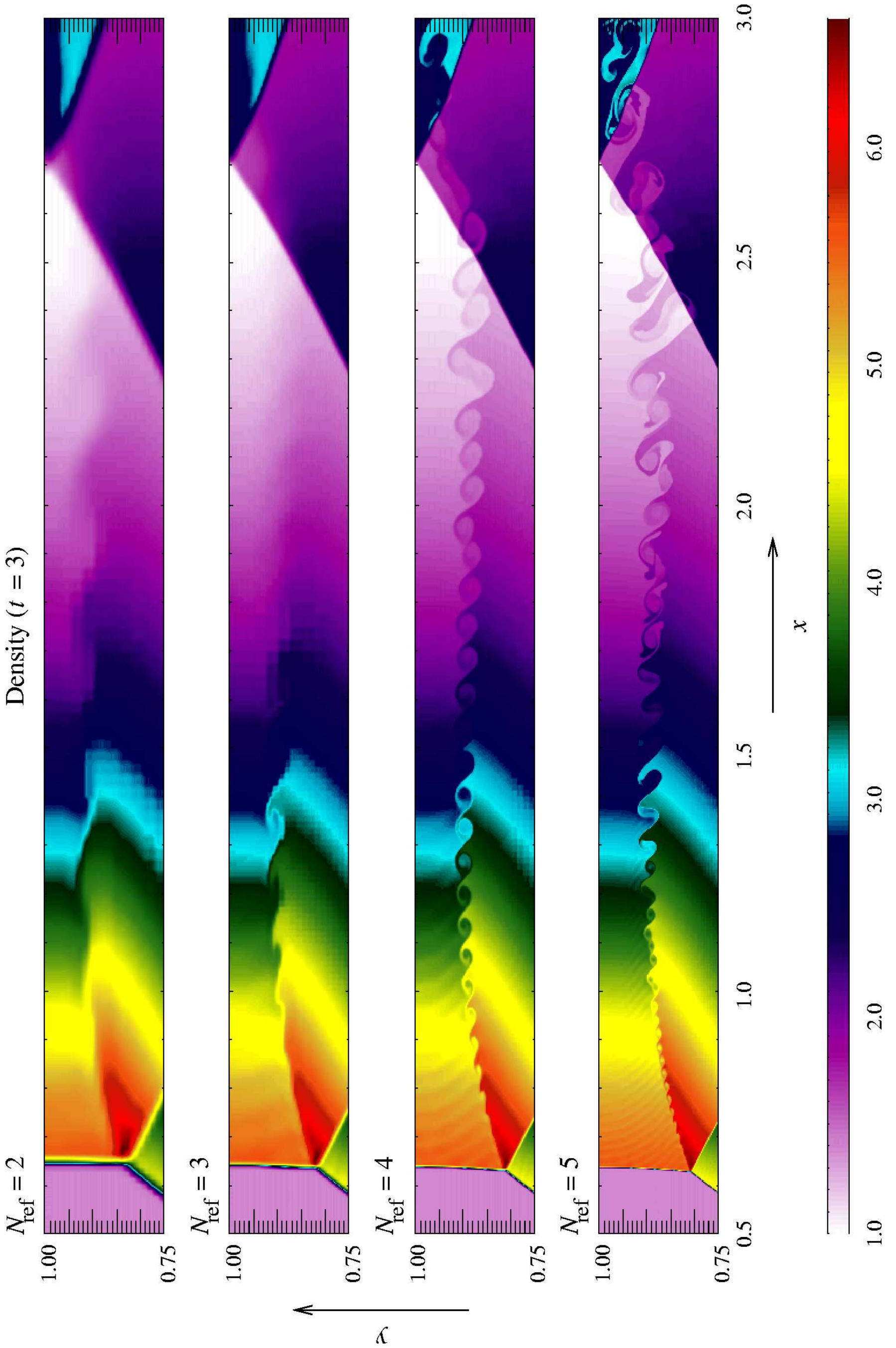


Figure 18.16: Detail of the Kelvin-Helmholtz instability seen at $t = 3$ in the Emery wind tunnel test problem for several different levels of refinement.

18.1.7 The Shu-Osher problem

The Shu-Osher problem (Shu and Osher, 1989) tests a shock-capturing scheme's ability to resolve small-scale flow features. It gives a good indication of the numerical (artificial) viscosity of a method. Since it is designed to test shock-capturing schemes, the equations of interest are the one-dimensional Euler equations for a single-species perfect gas.

In this problem, a (nominally) Mach 3 shock wave propagates into a sinusoidal density field. As the shock advances, two sets of density features appear behind the shock. One set has the same spatial frequency as the unshocked perturbations, but for the second set, the frequency is doubled. Furthermore, the second set follows more closely behind the shock. None of these features is spurious. The test of the numerical method is to accurately resolve the dynamics and strengths of the oscillations behind the shock.

The `shu_osher` problem is initialized as follows. On the domain $-4.5 \le x \le 4.5$, the shock is at $x = x_\mathtt{s}$ at $t = 0.0$. On either side of the shock,

$$\begin{array}{lcc} & x \le x_\mathtt{s} & x > x_\mathtt{s} \\ \rho(x) & \rho_\mathrm{L} & \rho_\mathtt{R}\,(1.0 + a_\rho \sin(f_\rho x)) \\ p(x) & p_\mathrm{L} & p_\mathtt{R} \\ u(x) & u_\mathrm{L} & u_\mathtt{R} \end{array} \tag{18.17}$$

where a_ρ is the amplitude and f_ρ is the frequency of the density perturbations. The `gamma` equation of state module is used with `gamma` set to 1.4. The runtime parameters and their default values are listed in Table 18.7. The initial density, x-velocity, and pressure distributions are shown in Fig. 18.17.

Table 18.7: Runtime parameters used with the `shu_osher` test problem.

Variable	Type	Default	Description
`posn`	real	-4.0	Initial shock location ($x_\mathtt{s}$)
`rho_left`	real	3.857143	Initial density to the left of the shock (ρ_L)
`rho_right`	real	1.0	Nominal initial density to the right (ρ_R)
`p_left`	real	10.33333	Initial pressure to the left (p_L)
`p_right`	real	1.0	Initial pressure to the right (p_R)
`u_left`	real	2.629369	Initial velocity to the left (u_L)
`u_right`	real	0.0	Initial velocity to the right (u_R)
`a_rho`	real	0.2	Amplitude of the density perturbations
`f_rho`	real	5.0	Frequency of the density perturbations

The problem is strictly one-dimensional; building 2-d or 3-d executables should give the same results along each x-direction grid line. For this problem, special boundary conditions are applied. The initial conditions should not change at the boundaries; if they do, errors at the boundaries can contaminate the results. To avoid this possibility, a boundary condition subroutine was written to set the boundary values to their initial values.

The purpose of the tests is to determine how much resolution, in terms of mesh cells per feature, a particular method requires to accurately represent small scale flow features. Therefore, all computations are carried out on equispaced meshes *without* adaptive refinement. Solutions are obtained at $t = 1.8$. The reference solution, using 3200 mesh cells, is shown in Fig. 18.18. This solution was computed using PPM at a CFL number of 0.8. Note the shock located at $x \simeq 2.4$, and the high frequency density oscillations just to the left of the shock. When the grid resolution is insufficient, shock-capturing schemes underpredict the amplitude of these oscillations and may distort their shape.

Fig. 18.19 shows the density field for the same scheme at 400 mesh cells and at 200 mesh cells. With 400 cells, the amplitudes are only slightly reduced compared to the reference solution; however, the shapes of the oscillations have been distorted. The slopes are steeper and the peaks and troughs are broader, which is

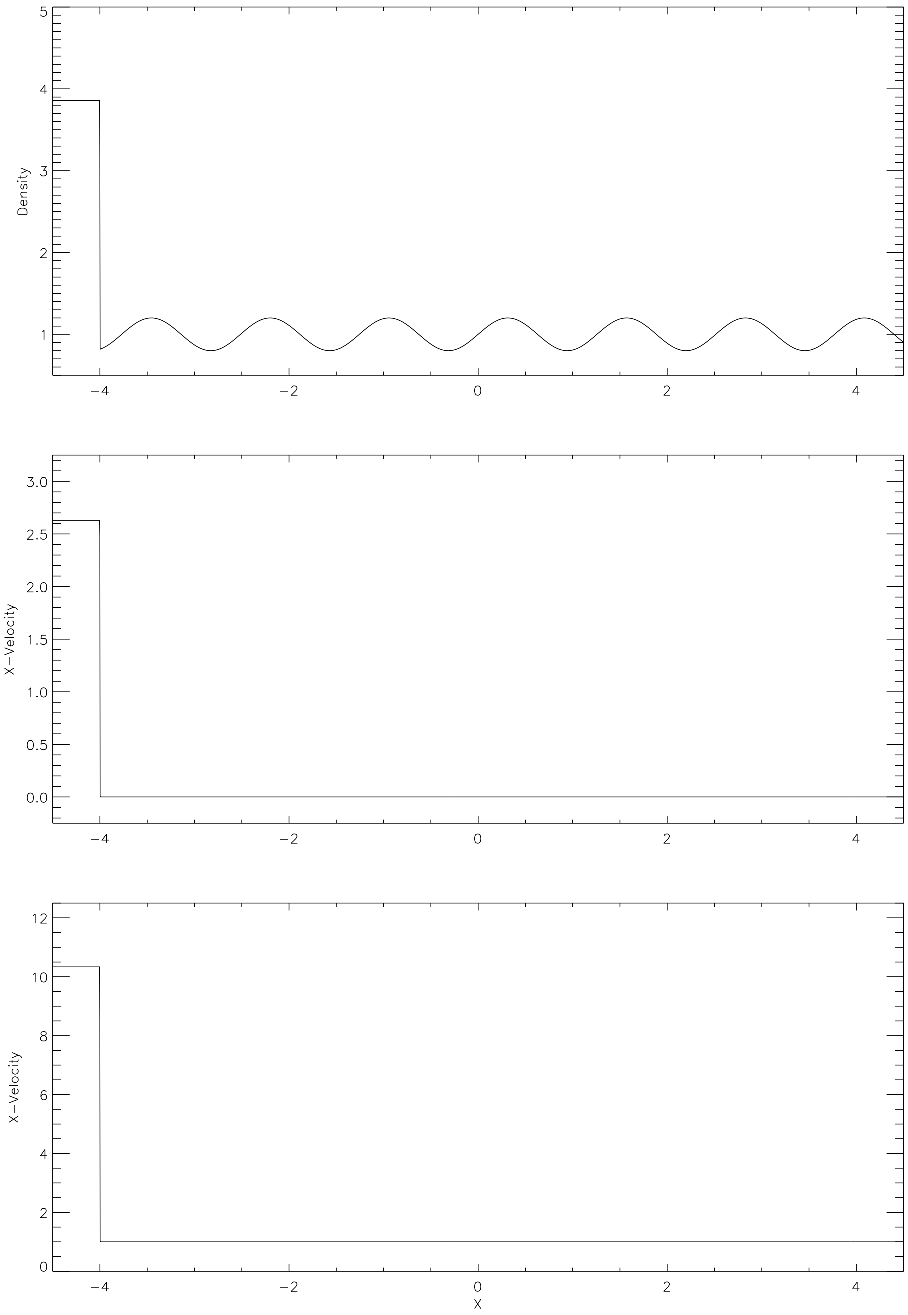


Figure 18.17: Initial density, x-velocity, and pressure for the Shu-Osher problem.

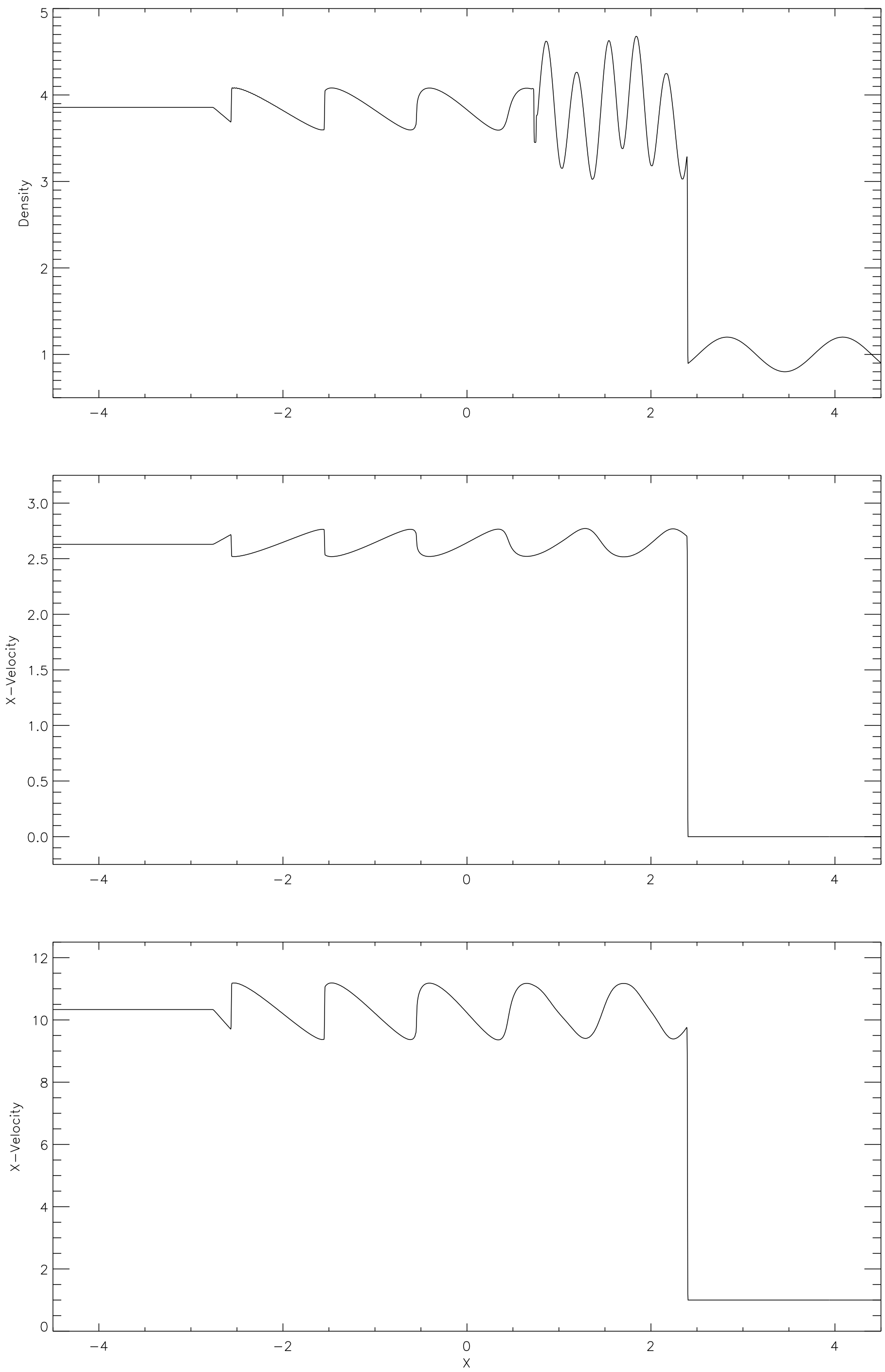


Figure 18.18: Density, x-velocity, and pressure for the reference solution at $t = 1.8$.

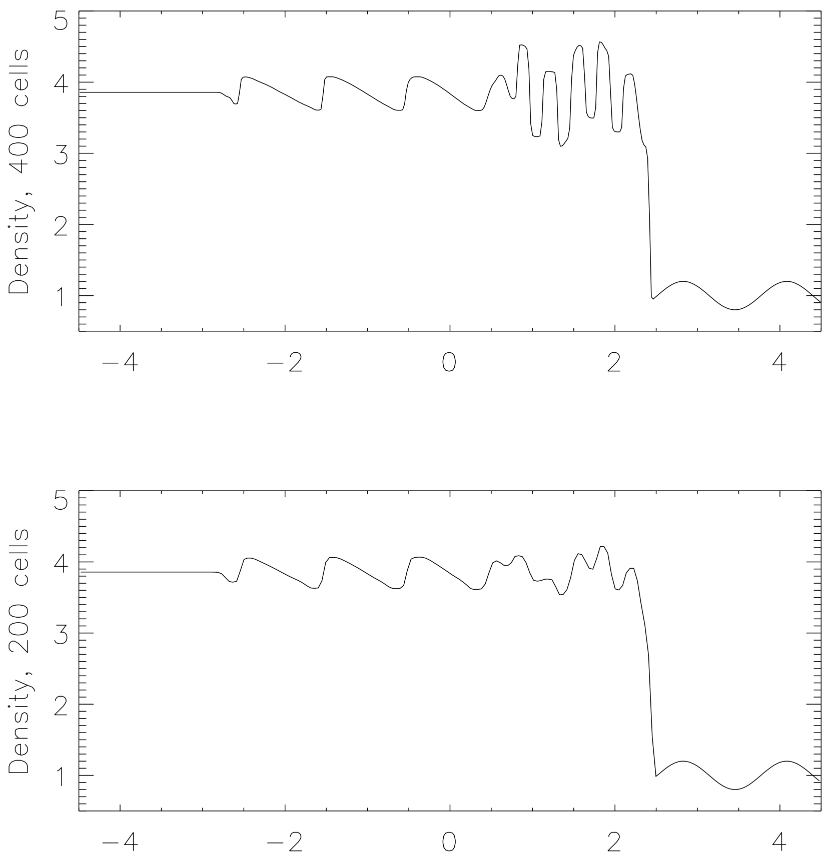


Figure 18.19: Density fields on 400 and 200 mesh cells from the PPM scheme.

the result of overcompression from the contact-steepening part of the PPM algorithm. For the solution on 200 mesh cells, the amplitudes of the high-frequency oscillations are significantly underpredicted.

18.1.8 The odd-even decoupling problem

The `odd_even` setup is designed to illustrate an odd-even decoupling phenomenon in grid-aligned shocks. The problem (and solution) was first pointed out by Quirk (1997), and our test case is taken from LeVeque (1998). The problem setup is simple. The domain has a uniform density ($\rho = 1$) and pressure ($P = 1$), with the flow in the left half of the domain ($x < 0.5$) given a velocity of 20, and the flow in the right half of the domain given a velocity of -20. A single zone in the center of the domain is given a 1% density perturbation. The PPM hydrodynamics solver is used for the evolution. The converging flow creates two planar shocks that move outward from the center of the domain in the x-direction. The density perturbation seeds the odd-even instability, and using the normal Riemann solver, large y-velocities result. The solution is to use the hybrid Riemann solver, described in Sec. 9.1.1. Quirk (1997) showed that an HLLE solver is effective in eliminating this instability. Fig. 18.20 shows the y-velocity at the end of the calculation, with and without the hybrid solver. In the run without the hybrid Riemann solver, the velocity is large and dominates the flow. In the case with the hybrid Riemann solver, the y-velocity is much smaller, and the reflections off the top and bottom boundaries can clearly be seen.

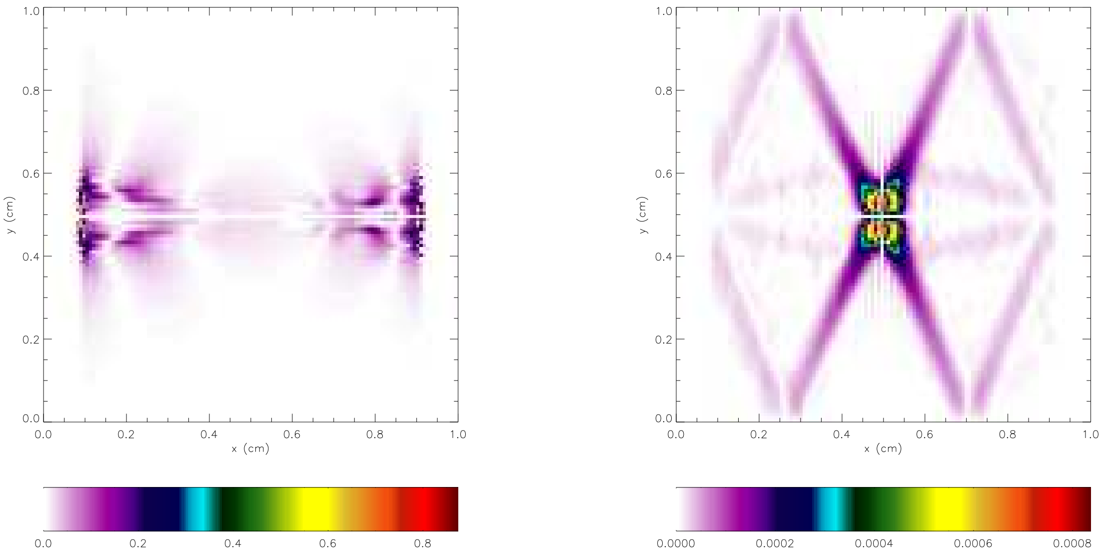


Figure 18.20: odd-even decoupling instability, without (left) and with (right) the hybrid Riemann solver enabled.

18.1.9 The Brio-Wu MHD shock tube problem

The Brio-Wu MHD shock tube problem (Brio and Wu, 1988) is a coplanar magnetohydrodynamic counterpart of the hydrodynamic Sod problem (Sec. 18.1.1). The initial left and right states are given by $\rho_l = 1$, $u_l = v_l = 0$, $p_l = 1$, $(B_y)_l = 1$; and $\rho_r = 0.125$, $u_r = v_r = 0$, $p_r = 0.1$, $(B_y)_r = -1$. In addition, $B_x = 0.75$ and $\gamma = 2$. This is a good problem to test wave properties of a particular MHD solver, because it involves two fast rarefaction waves, a slow compound wave, a contact discontinuity and a slow shock wave.

The conventional 800 point solution to this problem computed with FLASH 2.0 is presented in Figs. 18.21, 18.22, 18.23, 18.24, 18.25 . The figures show the distribution of density, normal and tangential velocity components, tangential magnetic field component and pressure at $t = 0.1$ (in non-dimensional units). As can be seen, the code accurately and sharply resolves all waves present in the solution. There is a small undershoot in the solution at $x \approx 0.44$, which results from a discontinuity-enhancing monotonized centered gradient limiting function (LeVeque 1997). This undershoot can be easily removed if a less aggressive limiter, *e.g.* a minmod or a van Leer limiter, is used instead. This, however, will degrade the sharp resolution of other discontinuities.

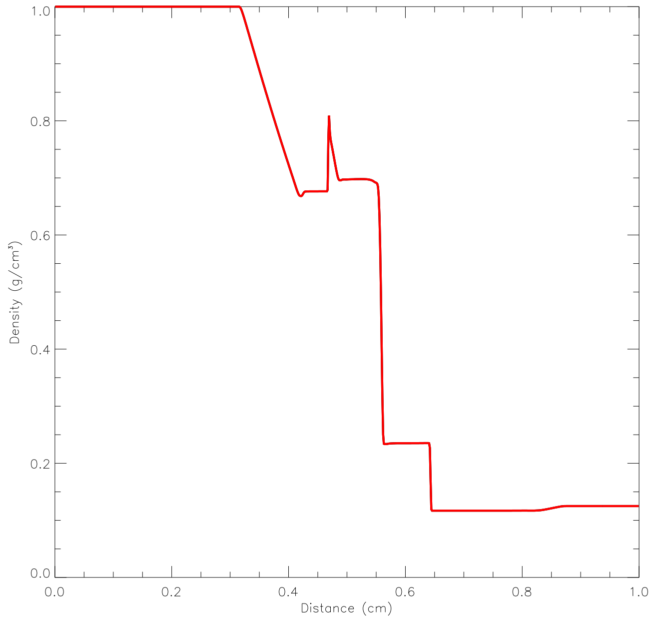


Figure 18.21: Density profile for the Brio-Wu shock tube problem.

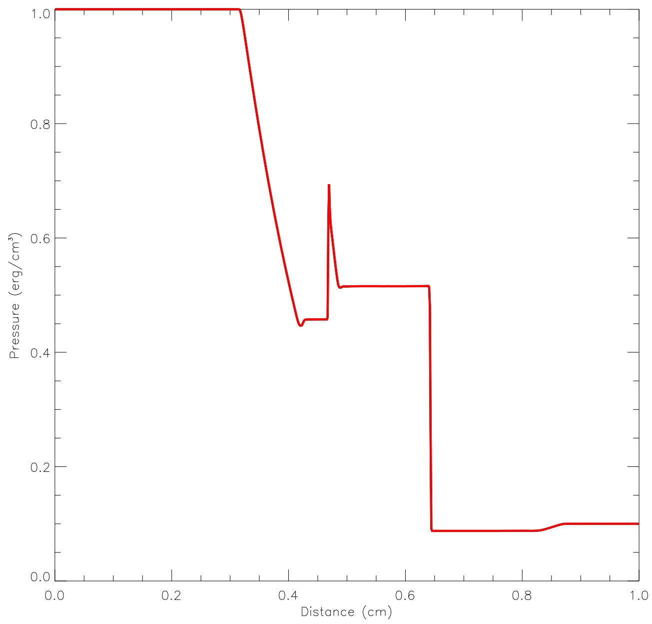


Figure 18.22: Pressure profile for the Brio-Wu shock tube problem.

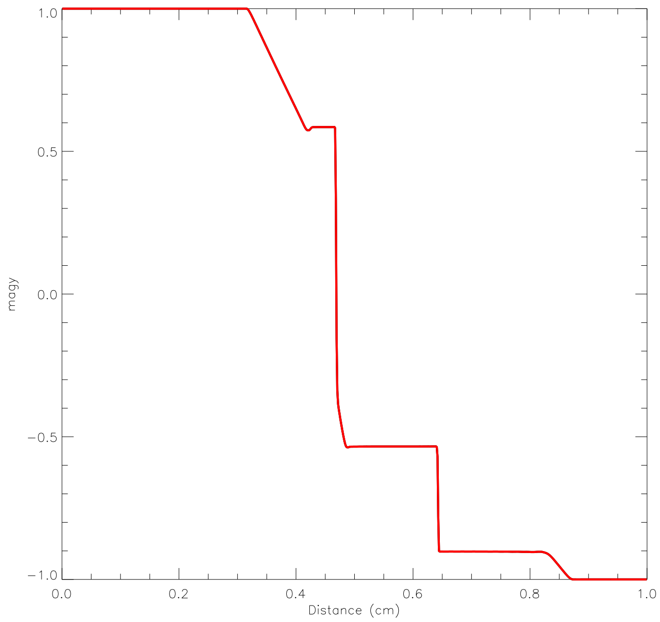


Figure 18.23: Tangential magnetic field profile for the Brio-Wu shock tube problem.

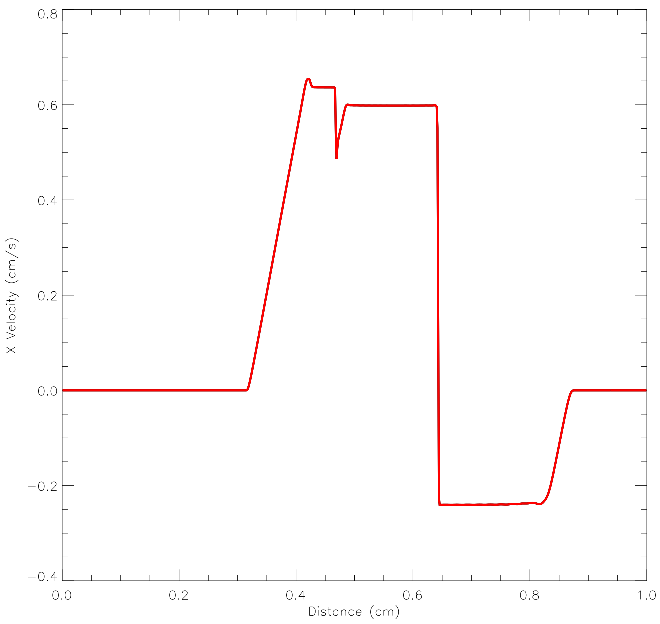


Figure 18.24: Normal velocity profile for the Brio-Wu shock tube problem.

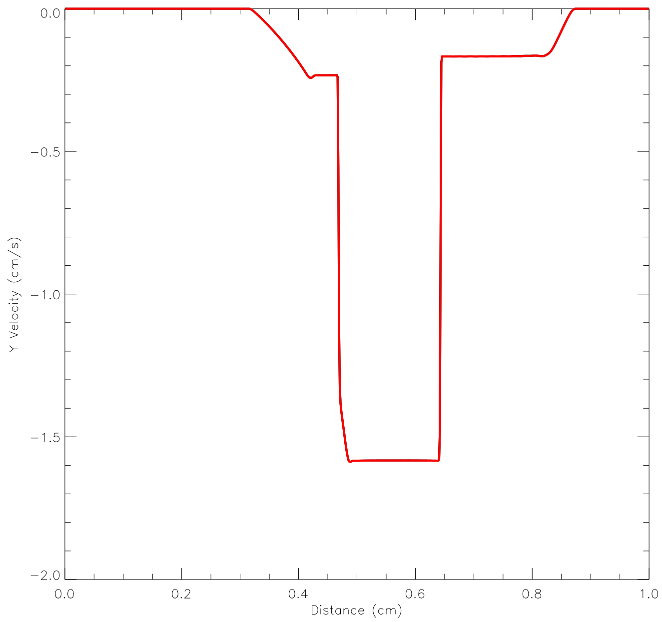


Figure 18.25: Tangential velocity profile for the Brio-Wu shock tube problem.

18.1.10 The Orszag-Tang MHD vortex problem

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

$$\mathbf{V} = V_0\left(-\sin(2\pi y), \sin(2\pi x), 0\right), \quad \mathbf{B} = B_0\left(-\sin(2\pi y), \sin(4\pi x), 0\right), \quad (x, y) =\in [0, 1]^2, \tag{18.18}$$

where B_0 is chosen so that the ratio of the gas pressure to the rms magnetic pressure is equal to 2γ. In this setup the initial density, the speed of sound and V_0 are set to unity; therefore, the initial pressure $p_0 = 1/\gamma$ and $B_0 = 1/\gamma$.

As the evolution time increases, the vortex flow pattern becomes increasingly complicated due to the nonlinear interactions of waves. A highly resolved simulation of this problem should produce two-dimensional MHD turbulence. Figs. 18.26 and 18.27 shows density and magnetic field contours at $t = 0.5$. As one can observe, the flow pattern at this time is already quite complicated. A number of strong waves have formed and passed through each other, creating turbulent flow features at all spatial scales.

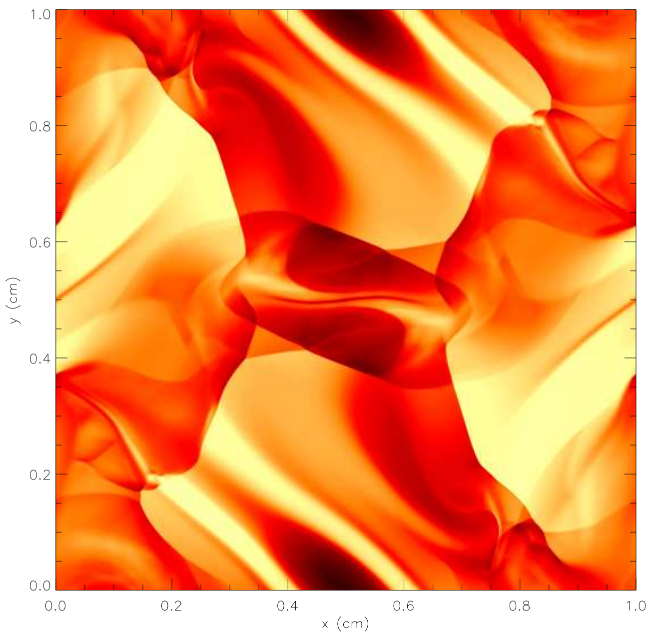


Figure 18.26: Density contours in the Oszag-Tang MHD vortex problem at $t = 0.5$.

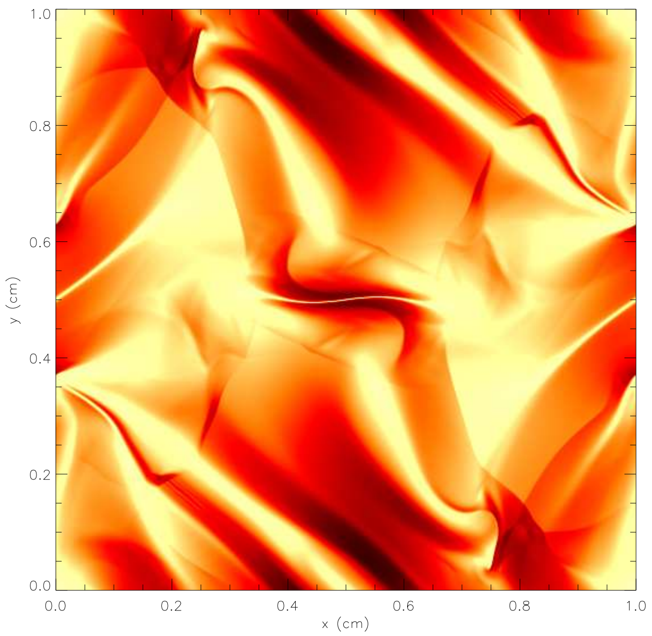


Figure 18.27: Magnetic field contours in the Oszag-Tang MHD vortex problem at $t = 0.5$.

18.2 Gravity test problems

18.2.1 The Jeans instability problem

The linear instability of self-gravitating fluids was first explored by Jeans (1902) in connection with the problem of star formation. The nonlinear phase of the instability is currently of great astrophysical interest, but the linear instability still provides a very useful test of the coupling of gravity to hydrodynamics in FLASH.

The `jeans` problem allows one to examine the behavior of sinusoidal, adiabatic density perturbations in both the pressure-dominated and gravity-dominated limits. This problem uses periodic boundary conditions. The equation of state is that of a perfect gas. The initial conditions at $t = 0$ are

$$\begin{aligned} \rho(\mathbf{x}) &= \rho_0 \left[1 + \delta \cos(\mathbf{k} \cdot \mathbf{x})\right] \\ p(\mathbf{x}) &= p_0 \left[1 + \gamma\delta \cos(\mathbf{k} \cdot \mathbf{x})\right] \\ \mathbf{v}(\mathbf{x}) &= 0\ , \end{aligned} \tag{18.19}$$

where the perturbation amplitude $\delta \ll 1$. The stability of the perturbation is determined by the relationship between the wavenumber $k \equiv |\mathbf{k}|$ and the Jeans wavenumber k_J, where k_J is given by

$$k_J \equiv \frac{\sqrt{4\pi G\rho_0}}{c_0}\ , \tag{18.20}$$

and where c_0 is the unperturbed adiabatic sound speed

$$c_0 = \sqrt{\frac{\gamma p_0}{\rho_0}} \tag{18.21}$$

(Chandrasekhar 1961). If $k > k_J$, the perturbation is stable and oscillates with frequency

$$\omega = \sqrt{c_0^2 k^2 - 4\pi G\rho_0}\ ; \tag{18.22}$$

otherwise, it grows exponentially, with a characteristic timescale given by $\tau = (i\omega)^{-1}$.

We checked the dispersion relation (18.22) for stable perturbations with $\gamma = 5/3$ by fixing ρ_0 and p_0 and performing several runs with different k. We followed each case for roughly five oscillation periods using a uniform grid in the box $[0, L]^2$. We used $\rho_0 = 1.5 \times 10^7$ g cm^{-3} and $p_0 = 1.5 \times 10^7$ dyn cm^{-2}, yielding $k_J = 2.747$ cm^{-1}. The perturbation amplitude δ was fixed at 10^{-3}. The box size L is chosen so that k_J is smaller than the smallest nonzero wavenumber that can be resolved on the grid

$$L = \frac{1}{2}\sqrt{\frac{\pi\gamma p_0}{G\rho_0^2}}\ . \tag{18.23}$$

This prevents roundoff errors at wavenumbers less than k_J from being amplified by the physical Jeans instability. We used wavevectors $\mathbf{k}$ parallel to and at 45 degrees to the x-axis. Each test calculation used the multigrid Poisson solver together with its default settings.

The resulting kinetic, thermal, and potential energies as functions of time for one choice of $\mathbf{k}$ are shown in Fig. 18.28 together with the analytic solution, which is given in two dimensions by

$$\begin{aligned} T(t) &= \frac{\rho_0\delta^2|\omega|^2L^2}{8k^2}\left[1 - \cos(2\omega t)\right] \\ U(t) - U(0) &= -\frac{1}{8}\rho_0 c_0^2\delta^2 L^2 \left[1 - \cos(2\omega t)\right] \\ W(t) &= -\frac{\pi G\rho_0^2\delta^2L^2}{2k^2}\left[1 + \cos(2\omega t)\right]\ . \end{aligned} \tag{18.24}$$

The figure shows that FLASH obtains the correct amplitude and frequency of oscillation. We computed the average oscillation frequency for each run by measuring the time interval required for the kinetic energy

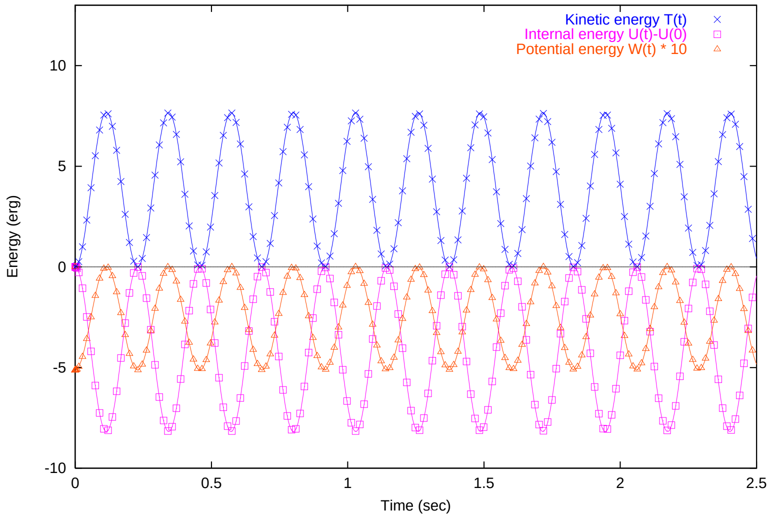


Figure 18.28: Kinetic, internal, and potential energy versus time for a stable Jeans mode with $k = 10.984$. Points indicate numerical values found using FLASH 2.0 with a four-level uniform grid. The analytic solution for each form of energy is shown using a solid line.

to undergo exactly ten oscillations. Fig. 18.29 compares the resulting dispersion relation to eq. (18.22). It can be seen from this plot that FLASH correctly reproduces equation (18.22). At the highest wavenumber ($k = 100$), each wavelength is resolved using only about 14 zones on a six-level uniform grid, and the average timestep (which depends on c_0, Δx, and Δy, and has nothing to do with k) turns out to be comparable to the oscillation period. Hence the frequency determined from the numerical solution for this value of k is somewhat more poorly determined than for the other runs. At lower wavenumbers, however, the frequencies are correct to less than 1%.

The additional runtime parameters supplied with the `jeans` problem are listed in Table 18.8. This problem is configured to use the perfect-gas equation of state (`gamma`) with `gamma` set to 1.67 and is run in a two-dimensional unit box. The refinement marking routine (`ref_marking.F90`) supplied with this problem refines blocks whose mean density exceeds a given threshold. Since the problem is not spherically symmetric, the multigrid Poisson solver should be used.

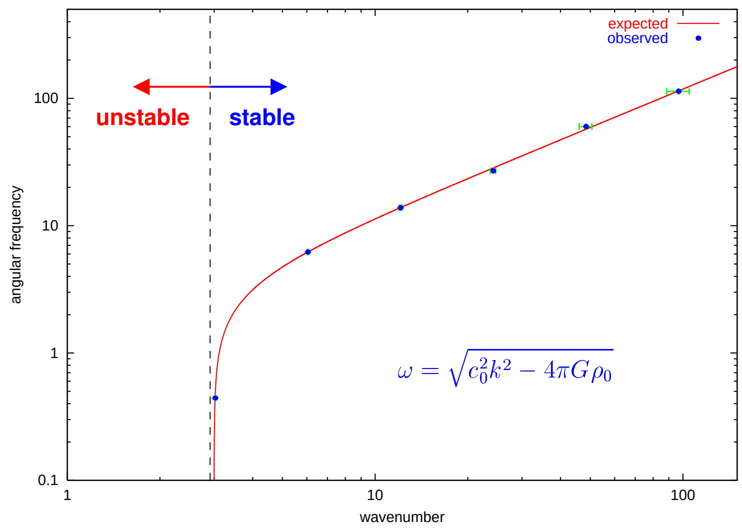


Figure 18.29: Computed versus expected Jeans dispersion relation (for stable modes) found using FLASH 1.62 with a six-level uniform grid.

Table 18.8: Runtime parameters used with the `jeans` test problem.

Variable	Type	Default	Description
`rho0`	real	1	Initial unperturbed density (ρ_0)
`p0`	real	1	Initial unperturbed pressure (p_0)
`amplitude`	real	0.01	Perturbation amplitude (δ)
`lambdax`	real	1	Perturbation wavelength in x direction ($\lambda_x = 2\pi/k_x$)
`lambday`	real	1	Perturbation wavelength in y direction ($\lambda_y = 2\pi/k_y$)
`lambdaz`	real	1	Perturbation wavelength in z direction ($\lambda_z = 2\pi/k_z$)
`delta_ref`	real	0.1	Refine a block if the maximum density contrast relative to $\rho_{\rm ref}$ is greater than this
`delta_deref`	real	-0.1	Derefine a block if the maximum density contrast relative to $\rho_{\rm ref}$ is less than this
`reference_density`	real	1	Reference density for grid refinement ($\rho_{\rm ref}$). Density contrast is used to determine which blocks to refine; it is defined as $$\max_{\rm block}\left\{\left\|\frac{\rho_{ijk}}{\rho_{\rm ref}}-1\right\|\right\}$$

18.2.2 The homologous dust collapse problem

The homologous dust collapse problem is used to test the ability of the code to solve self-gravitating problems in which the flow geometry is spherical and gas pressure is negligible. The problem was first described by Colgate and White (1966) and has been used by Mönchmeyer and Müller (1989) to test hydrodynamical schemes in curvilinear coordinates. As the Poisson solvers currently included with FLASH do not yet work in curvilinear coordinates, we solve this problem using a 3D Cartesian grid.

The initial conditions consist of a uniform sphere of radius r_0 and density ρ_0 at rest. The pressure p_0 is taken to be constant and very small

$$p_0 \ll \frac{4\pi G}{\gamma}\rho_0^2 r_0^2 \; . \tag{18.25}$$

We refer to such a nearly pressureless fluid as 'dust'. A perfect-gas equation of state is used, but the value of γ is not significant. Outflow boundary conditions are used for the gas, while isolated boundary conditions are used for the gravitational field.

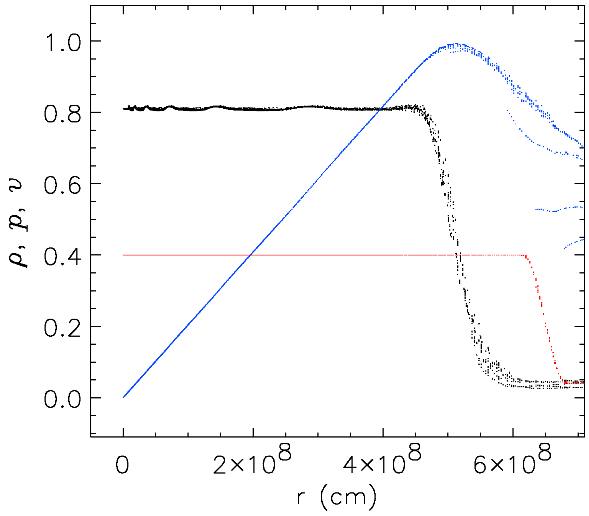
The collapse of the dust sphere is self-similar; the cloud should remain spherical with uniform density as it collapses. The radius of the cloud, $r(t)$, should satisfy

$$\left(\frac{8\pi G}{3}\rho_0\right)^{1/2} t = \left(1-\frac{r(t)}{r_0}\right)^{1/2}\left(\frac{r(t)}{r_0}\right)^{1/2} + \sin^{-1}\left(1-\frac{r(t)}{r_0}\right)^{1/2} \tag{18.26}$$

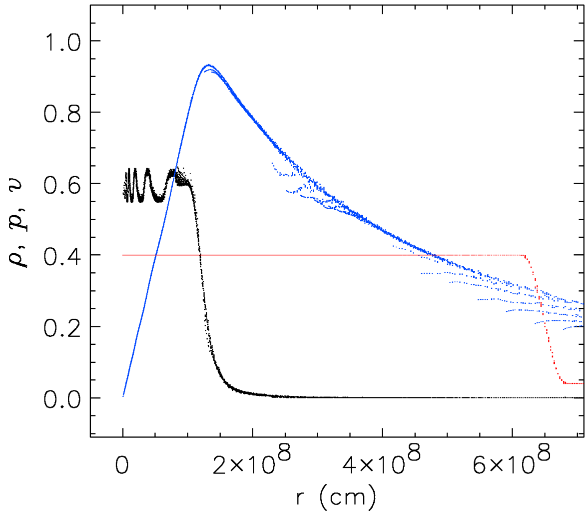
(Colgate & White 1966). Thus. we expect to test three things with this problem: the ability of the code to maintain spherical symmetry during an implosion (in particular, no block boundary effects should be evident); the ability of the code to keep the density profile constant within the cloud; and the ability of the code to obtain the correct collapse factor. The second of these is particularly difficult, because the edge of the cloud is very sharp and because the Cartesian grid breaks spherical symmetry most dramatically at the center of the cloud, which is where all of the matter ultimately ends up.

Results of a `dust_coll` run using FLASH 1.62 appear in Fig. 18.30. This run used 4^3 top-level blocks and seven levels of refinement, for an effective resolution of 2048^3. The multipole Poisson solver was used with a maximum multipole moment $\ell = 0$. The initial conditions used $\rho_0 = 10^9$ g cm^{-3} and $r_0 = 6.5 \times 10^8$ cm. In Fig. 18.30a, the density, pressure, and velocity are scaled by 2.43×10^9 g cm^{-3}, 2.08×10^{17} dyn cm^{-2}, and 7.30×10^9 cm s^{-1}, respectively. In Fig. 18.30b they are scaled by 1.96×10^{11} g cm^{-3}, 2.08×10^{17} dyn cm^{-2}, and 2.90×10^{10} cm s^{-1}. Note that within the cloud, the profiles are very isotropic, as indicated by the small dispersion in each profile. Significant anisotropy is only present for low-density material flowing in through the Cartesian boundaries. In particular, it is encouraging that the velocity field remains isotropic all the way into the center of the grid; this shows the usefulness of refining spherically symmetric problems near $r = 0$. However, as material flows inward past refinement boundaries, small ripples develop in the density profile due to interpolation errors. These remain spherically symmetric but increase in amplitude as they are compressed. Nevertheless, they are still only a few percent in relative magnitude by the second frame. The other numerical effect of note is a slight spreading at the edge of the cloud. This does not appear to worsen significantly with time. If one takes the radius at which the density drops to one-half its central value as the radius of the cloud, then the observed collapse factor agrees with our expectation from eq. (18.26). Overall our results, including the numerical effects, agree well with those of Mönchmeyer and Müller (1989).

The additional runtime parameters supplied with the `dust_coll` problem are listed in Table 18.9. This problem is configured to use the perfect-gas equation of state (`gamma`) with `gamma` set to 1.67 and is run in a three-dimensional box. The refinement marking routine (`ref_marking.F90`) supplied with this problem refines blocks containing the center of the cloud. Since the problem is spherically symmetric, either the multigrid or multipole solvers can be used.



(a)



(b)

Figure 18.30: Density (black), pressure (red), and velocity (blue) profiles in the homologous dust collapse problem at (a) $t = 0.0368$ sec and (b) $t = 0.0637$ sec. The density, pressure, and velocity are scaled as discussed in the text.

Table 18.9: Runtime parameters used with the `dust_coll` test problem.

Variable	Type	Default	Description
`rho0`	real	1	Initial cloud density (ρ_0)
`R_init`	real	0.05	Initial cloud radius (r_0)
`T_ambient`	real	1	Initial ambient temperature
`xctr`	real	0.5	x-coordinate of cloud center
`yctr`	real	0.5	y-coordinate of cloud center
`zctr`	real	0.5	z-coordinate of cloud center

18.2.3 The Huang-Greengard Poisson test problem

The `poistest` problem tests the convergence properties of the multigrid Poisson solver on a multidimensional, highly (locally) refined grid. This problem is described by Huang and Greengard (2000). The source function consists of a sum of thirteen two-dimensional Gaussians

$$\rho(x,y) = \sum_{i=1}^{13} e^{-\sigma_i[(x-x_i)^2+(y-y_i)^2]} , \tag{18.27}$$

where the constants σ_i, x_i, and y_i are given in Table 18.10. The very large range of widths and ellipticities of these peaks forces the mesh structure to be highly refined in some places. The density field and block structure are shown for a 14-level mesh in Fig. 18.31.

The `poistest` problem uses no additional runtime parameters beyond those required by the rest of the code.

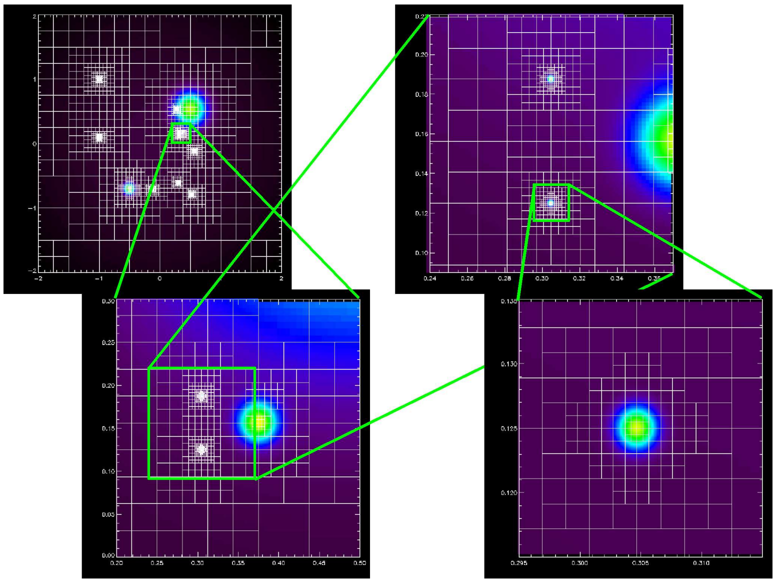


Figure 18.31: Density field and block structure for a 14-level mesh applied to the Huang-Greengard test problem. The effective resolution of the mesh is $65,536^2$.

Table 18.10: Constants used in the `poistest` problem.

i	1	2	3	4	5	6	7
x_i	0	-1	-1	0.28125	0.5	0.3046875	0.3046875
y_i	0	0.09375	1	0.53125	0.53125	0.1875	0.125
σ_i	0.01	4000	20000	80000	16	360000	400000

i	8	9	10	11	12	13	
x_i	0.375	0.5625	-0.5	-0.125	0.296875	0.5234375	
y_i	0.15625	-0.125	-0.703125	-0.703125	-0.609375	-0.78125	
σ_i	2000	18200	128	49000	37000	18900	

18.3 Particle test problems

18.3.1 The two-particle orbit problem

The `orbit` problem tests the mapping of particle positions to gridded density fields, the mapping of gridded potentials onto particle positions to obtain particle forces, and the time integration of particle motion. The initial conditions consist of two particles of unit mass and separation r_0 located at positions $(x, y, z) = (0.5(L_x \pm r_0), 0.5L_y, 0.5L_z)$, where (L_x, L_y, L_z) are the dimensions of the computational volume. The initial particle velocity vectors are parallel to the y-axis and have magnitude

$$|v| = \sqrt{\frac{2GM}{r_0}}\ , \tag{18.28}$$

if a constant gravitational field due to a point mass M at $(0.5L_x, 0.5L_y, 0.5L_z)$ is employed, or

$$|v| = \frac{1}{2}\sqrt{\frac{2G}{r_0}}\ , \tag{18.29}$$

if the particles are self-gravitating. The correct behavior is for the particles to orbit the center of the grid in a circle with constant velocity.

Fig. 18.32 shows a typical pair of particle trajectories for this problem, together with the AMR block structure at the ending time. The refinement marking routine supplied with this problem performs the standard second-derivative refinement supplied with PARAMESH plus particle-based refinement, in which a block is derefined if it contains fewer than `particle_deref_thresh` particles or refined if it contains more than `particle_ref_thresh`. It is important to apply the second-derivative criterion to the gridded particle density variable (`pden`) to ensure that particle clouds do not lie on fine-coarse block boundaries. When particle clouds do intersect refinement boundaries, the particles experience self-forces, and momentum is not conserved.

The two-particle orbit problem uses the runtime parameters listed in Table 18.11 in addition to the regular ones supplied with the code. Although it is not explicitly required by the configuration file for this problem, `orbit` should be run using conservative, quadratic interpolants (for example, one could use `mesh/amr/paramesh2.0/quadratic_cartesian`) with monotonicity enforcement off (`monotone = .false.`). It is necessary to turn off monotonicity enforcement because the small number of particles makes the gridded particle density field fairly discontinuous. No specific gravity module is required by the problem configuration file, because the problem is intended to be run with either a fixed external field or the particles' own field. If the particles are to orbit in an external field (`ext_field = .true.`), the field is assumed to be a central point-mass field (`gravity/ptmass`), and the parameters for that module should be assigned appropriate values. If the particles are self-gravitating (`ext_field = .false.`), the `gravity/poisson` module should be included in the code, and a Poisson solver that supports isolated boundary conditions should be used (`grav_boundary_type = "isolated"`). In either case, long-range forces for the particles must be turned on, or else they will not experience any accelerations at all. This can be done using the particle-mesh method by including the module `particles/active/long_range/pm/gravity`. (This is not required by the problem configuration file because the forces could also be obtained using a future direct N-body solver or a tree solver.) As of FLASH 2.1 both the multigrid and multipole solvers support isolated boundary conditions. This problem should be run in three dimensions.

Table 18.11: Runtime parameters used with the `orbit` test problem.

Variable	Type	Default	Description
`separation`	real	0.5	Initial particle separation (r_0)
`ext_field`	logical	`.false.`	Whether to make the particles self-gravitating or to have them orbit in an external potential. In the former case `gravity/poisson` should be used; in the latter, `gravity/ptmass`.

`particle_ref_thresh`	integer	1	Refine blocks containing more than this many particles
`particle_deref_thresh`	integer	0	Derefine blocks containing fewer than this many particles

18.3.2 The Zel'dovich pancake problem

The cosmological pancake problem (Zel'dovich 1970) provides a good simultaneous test of the hydrodynamics, particle dynamics, Poisson solver, and cosmological expansion modules. Analytic solutions well into the nonlinear regime are available for both N-body and hydrodynamical codes (Anninos & Norman 1994), permitting an assessment of the code's accuracy. After caustic formation the problem provides a stringent test of the code's ability to track thin, poorly resolved features and strong shocks using most of the basic physics needed for cosmological problems. Also, as pancakes represent single-mode density perturbations, coding this test problem is useful as a basis for creating more complex cosmological initial conditions.

We set the initial conditions for the pancake problem in the linear regime using the analytic solution given by Anninos and Norman (1994). In a universe with $\Omega_0 = 1$ at redshift z, a perturbation of wavenumber k which collapses to a caustic at redshift $z_c < z$ has comoving density and velocity given by

$$\begin{aligned}
\rho(x_e;z) &= \bar{\rho}\left[1+\frac{1+z_c}{1+z}\cos\left(kx_\ell\right)\right]^{-1} &\quad (18.30)\\
v(x_e;z) &= -H_0(1+z)^{1/2}(1+z_c)\frac{\sin kx_\ell}{kx_\ell}\ , &\quad (18.31)
\end{aligned}$$

where $\bar{\rho}$ is the comoving mean density. Here x_e is the distance of a point from the pancake midplane, and x_ℓ is the corresponding Lagrangian coordinate, found by iteratively solving

$$x_e = x_\ell - \frac{1+z_c}{1+z}\frac{\sin kx_\ell}{k}\ . \qquad (18.32)$$

The temperature solution is determined from the density under the assumption that the gas is adiabatic with ratio of specific heats γ:

$$T(x_e;z) = (1+z)^2\bar{T}_{\rm fid}\left[\left(\frac{1+z_{\rm fid}}{1+z}\right)^3\frac{\rho(x_e;z_{\rm fid})}{\rho(x_e;z)}\right]^{\gamma-1}\ . \qquad (18.33)$$

The mean temperature $\bar{T}_{\rm fid}$ is specified at a redshift $z_{\rm fid}$.

Dark matter particles are initialized using the same solution as the gas. The Lagrangian coordinates x_ℓ are assigned to lie on a uniform grid. The corresponding perturbed coordinates x_e are computed using equation (18.32). Particle velocities are assigned using equation (18.31).

At caustic formation ($z = z_c$), planar shock waves form in the gas on either side of the pancake midplane and begin to propagate outward. A small region at the midplane is left unshocked. Immediately behind the shocks, the comoving density and temperature vary approximately as

$$\begin{aligned}
\rho(x_e;z) &\approx \bar{\rho}\frac{18}{\left(kx_\ell\right)^2}\frac{\left(1+z_c\right)^3}{\left(1+z\right)^3} &\quad (18.34)\\
T(x_e;z) &\approx \frac{\mu H_0^2}{6k_Bk^2}\left(1+z_c\right)\left(1+z\right)^2\left(kx_\ell\right)^2\ .
\end{aligned}$$

At the midplane, which undergoes adiabatic compression, the comoving density and temperature are approximately

$$\begin{aligned}
\rho_{\rm center} &\approx \bar{\rho}\left[\frac{1+z_{\rm fid}}{1+z}\right]^3\left[\frac{3H_0^2\mu}{k_B\bar{T}_{\rm fid}k^2}\frac{\left(1+z_c\right)^4\left(1+z\right)^3}{1+z_{\rm fid}}\right]^{1/\gamma} &\quad (18.35)\\
T_{\rm center} &\approx \frac{3H_0^2\mu}{k_Bk^2}\left(1+z\right)^2\left(1+z_c\right)^4\frac{\bar{\rho}}{\rho_{\rm center}}\ .
\end{aligned}$$

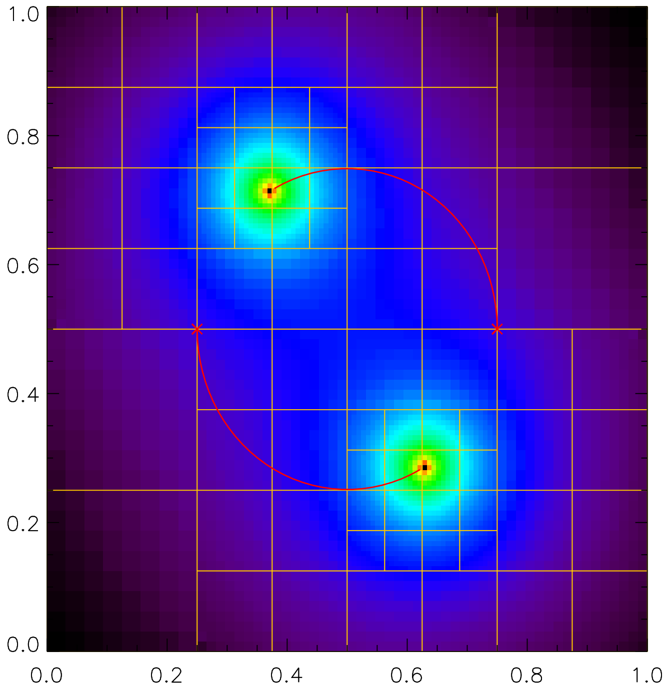


Figure 18.32: Typical particle trajectories in the `orbit` test problem, superimposed upon the log of the particles' mutual potential (colormap). The AMR block structure is also shown. A 3D grid with five levels of refinement was used.

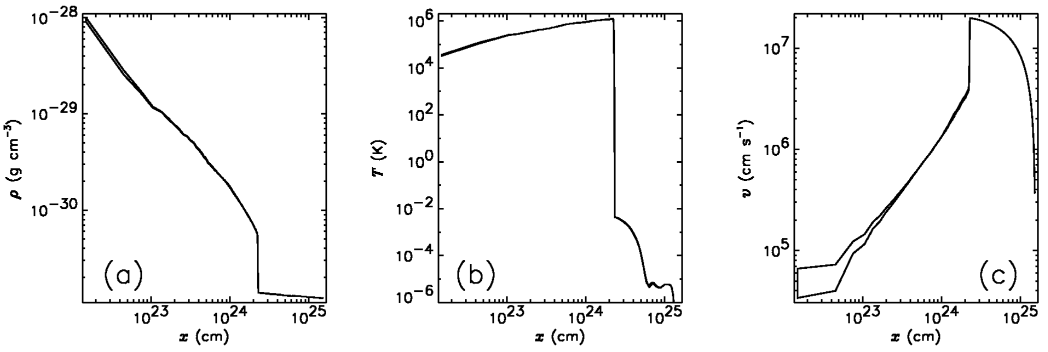


Figure 18.33: Example FLASH solution for the gas in a mixed particle/gas Zel'dovich pancake. A comoving wavelength $\lambda = 10$ Mpc, caustic redshift $z_c = 5$, fiducial redshift $z_{\rm fid} = 200$, and fiducial temperature $T_{\rm fid} = 550$ K were used together with a Hubble constant of 50 km s^{-1} Mpc^{-1}. The cosmological model was flat with a baryonic fraction of 0.15. Results are shown for redshift $z = 0$. An adaptive mesh with an effective resolution of 1024 zones was used. Other parameters for this run were as described in the text. The distance x is measured from the pancake midplane. (a) Gas density. (b) Gas temperature. (c) Gas velocity.

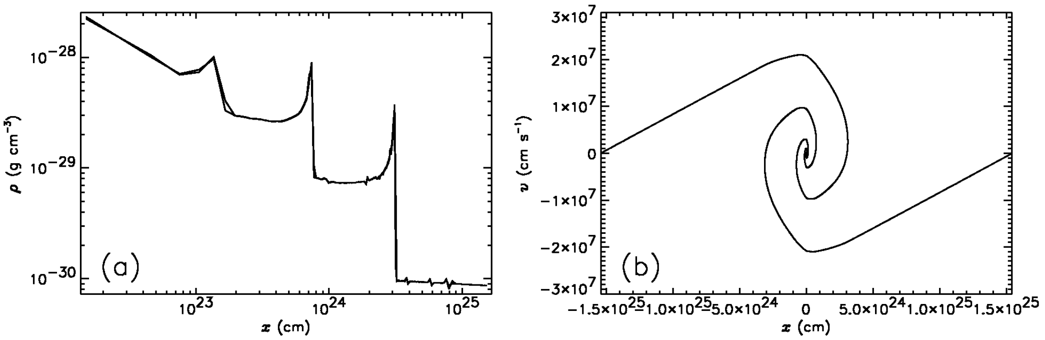


Figure 18.34: Example FLASH solution for the dark matter in a mixed particle/gas Zel'dovich pancake. Perturbation and cosmological parameters were the same as in Figure 18.33. Results are shown for redshift $z = 0$. An adaptive mesh with an effective resolution of 1024 zones was used. The number of particles used was 8192. Other parameters for this run were as described in the text. Distance x is measured from the pancake midplane. (a) Dark matter density. (b) Dark matter phase diagram showing particle positions x and velocities v.

An example FLASH calculation of the post-caustic gas solution appears in Figure 18.33.

Because they are collisionless, the particles behave very differently than the gas. As particles accelerate toward the midplane, their phase profile develops a backwards "S" shape. At caustic formation the velocity becomes multivalued at the midplane. The region containing multiple streams grows in size as particles pass through the midplane. At the edges of this region (the caustics, or the inflection points of the "S"), the particle density is formally infinite, although the finite force resolution of the particles keeps the height of these peaks finite. Some of the particles that have passed through the midplane fall back and form another pair of caustics, twisting the phase profile again. Because each of these secondary caustics contains five streams of particles rather than three, the second pair of density peaks are higher than the first pair. This caustic formation process repeats arbitrarily many times in the analytic solution. In practice, the finite number of particles and the finite force resolution limit the number of caustics that are observed. An example FLASH calculation of the post-caustic particle solution appears in Figure 18.34.

The `pancake` problem uses the runtime parameters listed in Table 18.12 in addition to the regular ones supplied with the code. Although it is not explicitly required by the configuration file for this problem, `pancake` should be run using conservative, quadratic interpolants. This problem uses periodic boundary conditions and is intrinsically one-dimensional, but it can be run using Cartesian coordinates in 1D, 2D, or 3D, with the pancake midplane tilted with respect to the coordinate axes if desired.

The refinement criteria used for the adaptive mesh in this problem are the second derivative of the gas density and a logarithmically spaced set of density thresholds for the gas and particles: overdensities between 1 and 3, 3 and 10, 10 and 30, etc. are refined by 1, 2, 3, etc. levels. Once refined, overdensities do not derefine.

Table 18.12: Runtime parameters used with the `pancake` test problem.

Variable	Type	Default	Description
`lambda`	real	3.0857×10^{24}	Wavelength of the initial perturbation ($2\pi/k$)
`zcaustic`	real	1.	Redshift at which pancake forms a caustic (z_c)
`Tfiducial`	real	100.	Fiducial gas temperature ($T_{\rm fid}$)
`zfiducial`	real	100.	Redshift at which gas temperature is $T_{\rm fid}$ ($z_{\rm fid}$)
`xangle`	real	0.	Angle made by pancake normal with the x-axis (degrees)
`yangle`	real	90.	Angle made by pancake normal with the y-axis (degrees)
`NumXparticles`	integer	1	Number of particles along x-side of initial particle "grid"
`NumYparticles`	integer	1	Number of particles along y-side of initial particle "grid"
`NumZparticles`	integer	1	Number of particles along z-side of initial particle "grid"

18.4 Other test problems

18.4.1 The sample_map problem

Frequently when doing simulations, one needs to initialize the computational domain with a one-dimensional model from a stellar evolution (or other) code. A simple framework for accomplishing this task is provided by the sample_map problem. This is intended to be a template for users to modify to suit their needs.

This problem is composed of two main routines, `init_1d` and the familiar `init_block`. The `init_1d` we use is provided by the `util/initialization/1d` module. It reads the initial model from disk, determines which variables are present and how they map into the variables defined in FLASH, and stores the initial model in arrays that are then used by `init_block`. The general format of an initial model file is a single comment line, a line giving the number of variables contained in the initial model, the 4-character names of each variable (one per line), followed by the data (spatial coordinate first), with the variables in the same order as the list of names. An example of this format follows.

```
# sample 1-d model
number of variables = 7
dens
pres
ener
gamc
game
fuel
```

```
ash
0.01 10.  100.  25.  1.4 1.4 1.0 0.0
0.02 9.5.  95.  25.  1.4 1.4 0.99 0.01
⋮
```

In the above sample file, we define seven variables. The first zone starts with the coordinate of the zone center (0.01) and then lists the density (10.), pressure (100.), and so forth, with one entry for each variable per line. The next zone of the initial model is listed immediately below this line. `init_1d` will continue to read in zones for the initial model until it encounters the end of the file.

FLASH contains more variables than the seven defined in this input file, and it will initialize any variables not specified in the input file to zero. Additionally, sometimes a variable is specified in the input file, but there is no corresponding variable defined in FLASH. In this case, `init_1d` will produce a warning, listing the variables it does not know about. Finally, there is no need for the variables to be listed in the same order as they are stored in the FLASH data structures—they will be sorted as each zone is read from the initial model.

The initial model is stored in two data structures: `xzn(N1D_MAX)` contains the coordinates of the initial model zone centers, and `model_1d(N1D_MAX, nvar)` contains the values of the variables defined in the initial model. These are stored in the same order as the variables in the solution array `unk` maintained by FLASH. `N1D_MAX` is a parameter specifying the maximum number of zones in the initial model (currently set to 2048).

These data structures are passed to the `init_block` function which loops over all of the zones in the current block, determines the x-, y-, and z-coordinates of the zone, and performs an interpolation to find the values of the initial variables in the current zone. This interpolation attempts to construct zone averages from the values of the initial model at the zone edges and center.

There are two parameters for this problem. `model_file` is a string that gives the name of the input file from which to read the initial model. `imap_dir` is an integer that specifies along which direction to map the initial model. `imap_dir` = 1 maps along the x-direction, 2 maps along the y-direction, and 0 maps in a circle in the x-y plane.

18.4.2 The non-equilibrium ionization test problem

The `neitest` problem tests the ability of FLASH to calculate non-equilibrium ionization (NEI) ion abundances. It simulates a stationary plasma flow through a temperature step profile. The solutions were checked using an independent stationary code based on a fifth order Runge–Kutta method with adaptive stepsize control by step-doubling (see Orlando *et al.* (1999)).

The test assumes a plasma with a mass density of 2×10^{-16} gm cm^{-3} flowing with a constant uniform velocity of 3×10^{5} cm s^{-1} through a temperature step between 10^4 K and 10^6 K (*cf.* Fig. 18.35). The plasma is in ionization equilibrium before going through the jump in the region at $T = 10^4$ K. The population fractions in equilibrium are obtained from the equations

$$[n_i^Z]_{eq} S_i^Z = [n_{i+1}^Z]_{eq} \alpha_{i+1}^Z \qquad (i = 1, ..., l_Z - 1) \tag{18.36}$$

$$\sum_{i=1}^{l_Z} [n_i^Z]_{eq} = A_Z n_p \tag{18.37}$$

The presence of a temperature jump causes a strong pressure difference, which in turn should cause significant plasma motions. Since the purpose is to test the NEI module, it is imposed that the pressure difference does not induce any plasma motion and, to this end, the hydro variables (namely, T, ρ, $\mathbf{v}$) are not updated. In practice, the continuity equations are solved with uniform density and velocity, while the momentum and energy equations are ignored.

Fig. 18.36 shows the population fractions for the 12 most abundant elements in astrophysical plasmas derived with the stationary code (Orlando *et al.* (1999)). The out of equilibrium ionization conditions are evident for all the elements just after the flow goes through the temperature jump.

The same problem was solved with the NEI module of the FLASH code, assuming that the plasma is initially in ionization equilibrium at $t = t_0$ over all the spatial domain. After a transient lasting approximately

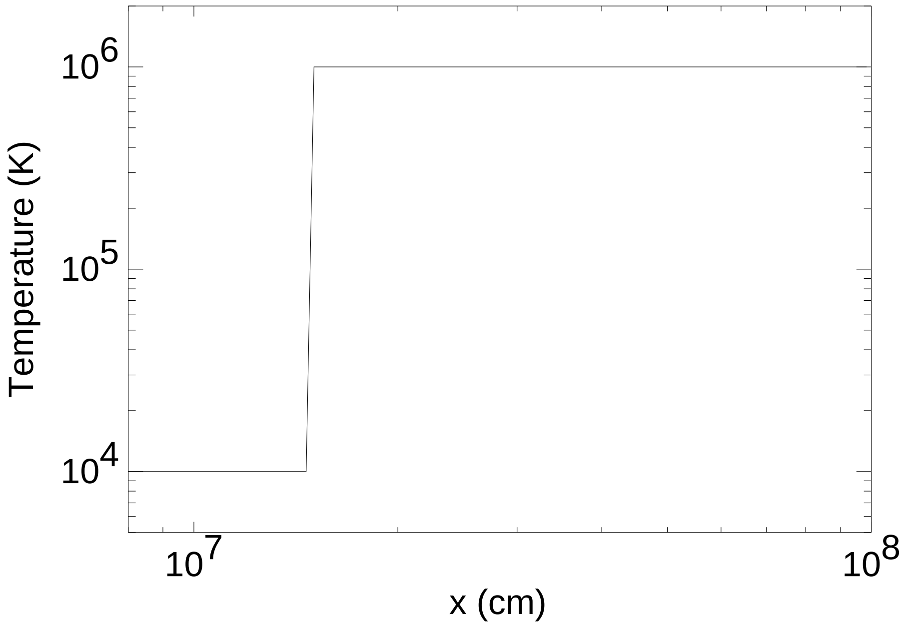


Figure 18.35: Temperature profile assumed for the test.

700 s, in which the population fractions evolve due to the plasma flow through the temperature jump, the system reaches the stationary configuration. Outflow boundary conditions (zero-gradient) are assumed at both the left and right boundaries. Fig. 18.37 shows the population fraction vs. space after 700 s.

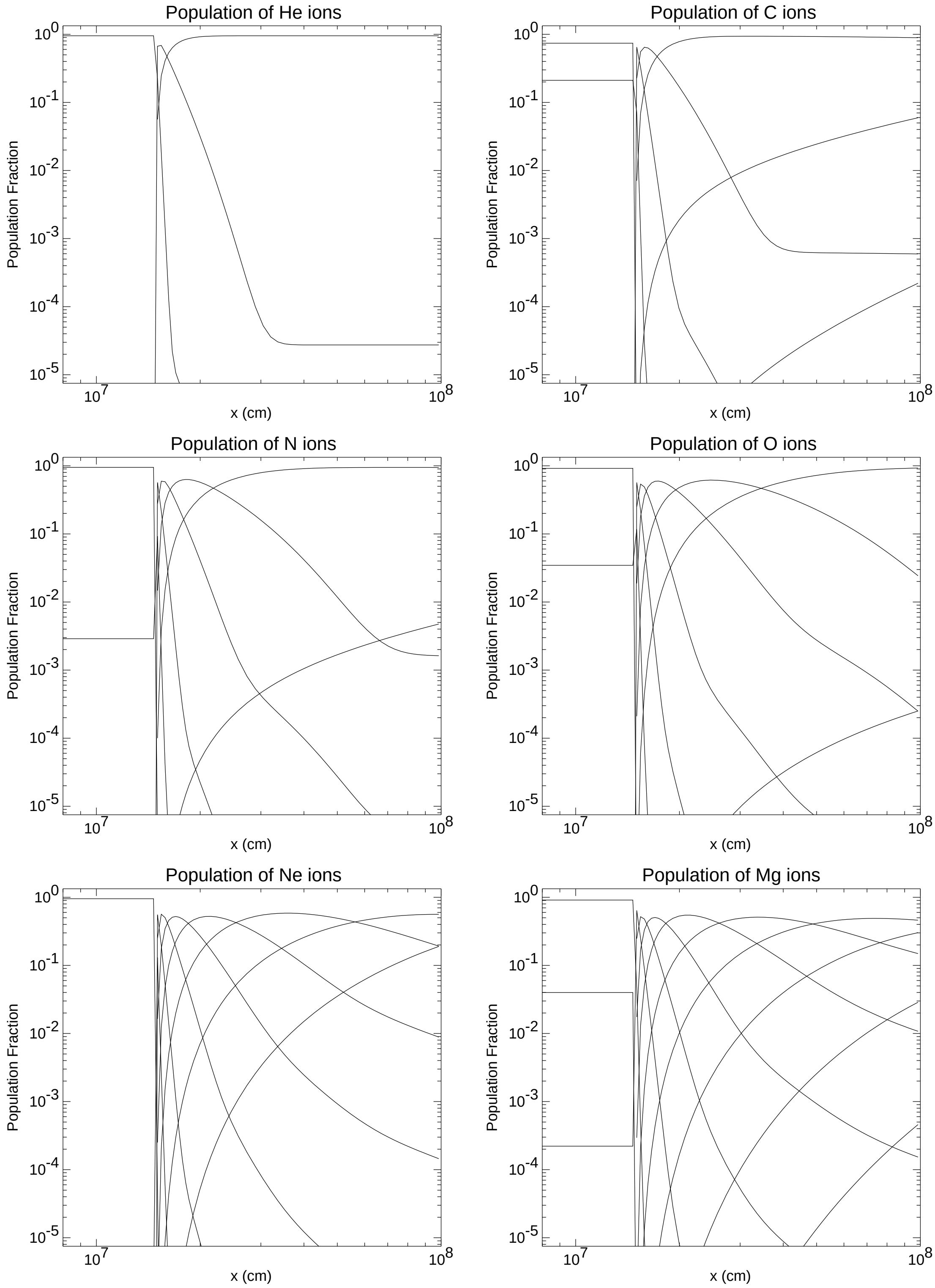


Figure 18.36: Numerical solutions of the stationary code. The figure shows the population fractions vs. space for the 12 elements most abundant in astrophysical plasmas assuming a stationary flow through a temperature jump.

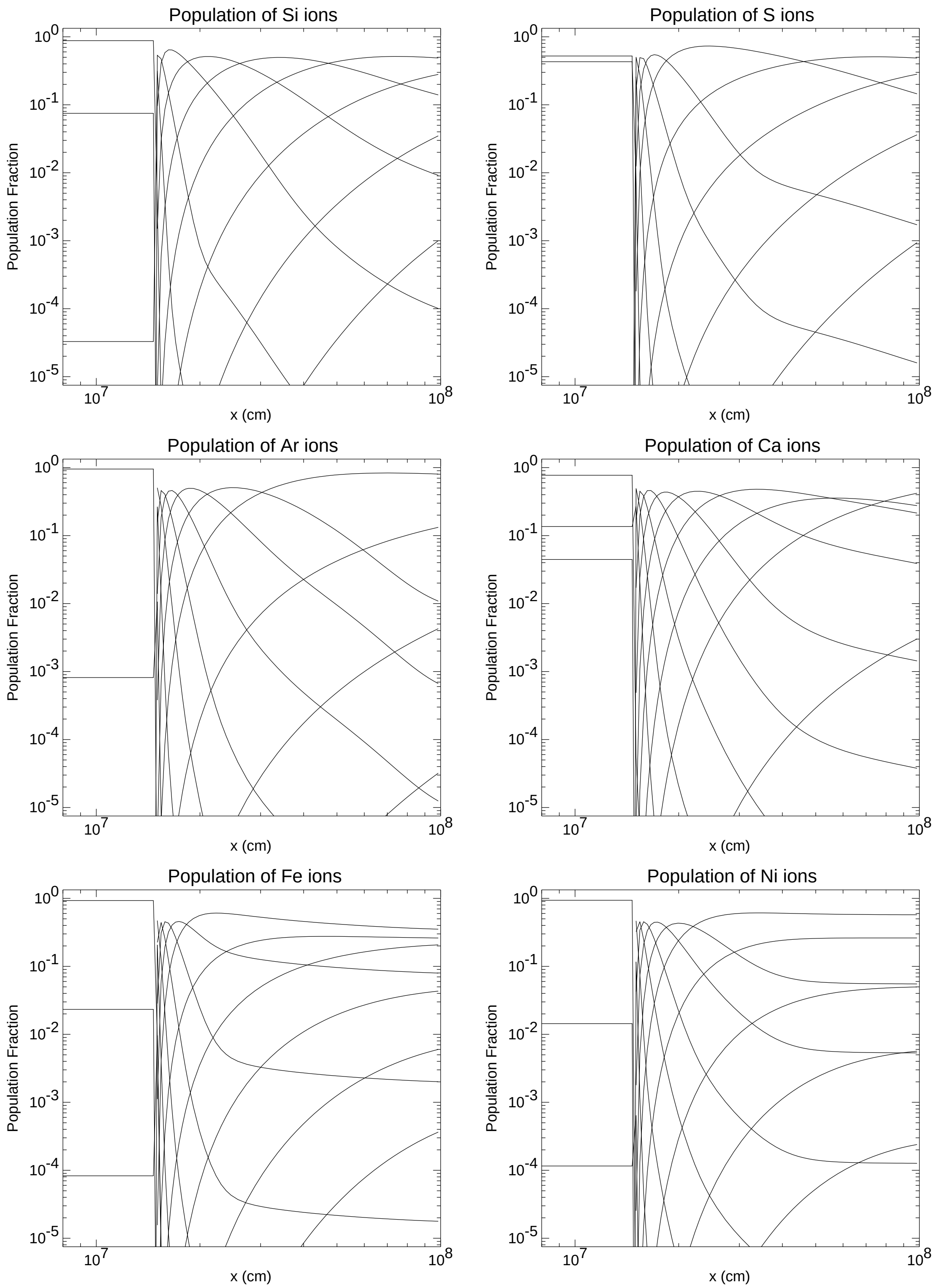


Figure 18.36: ... continued ...

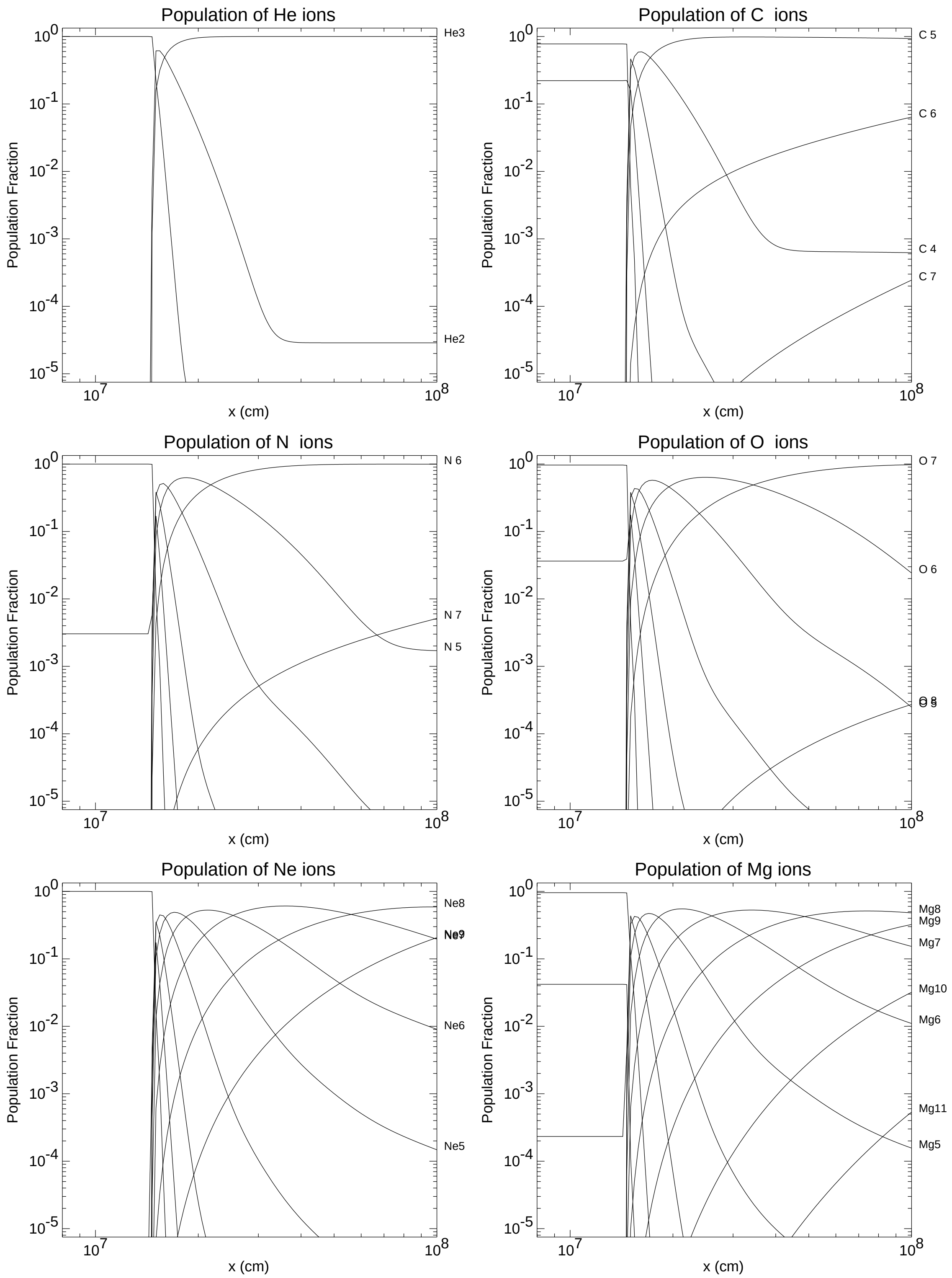


Figure 18.37: As in Fig. 18.36 for the solutions of the FLASH code.

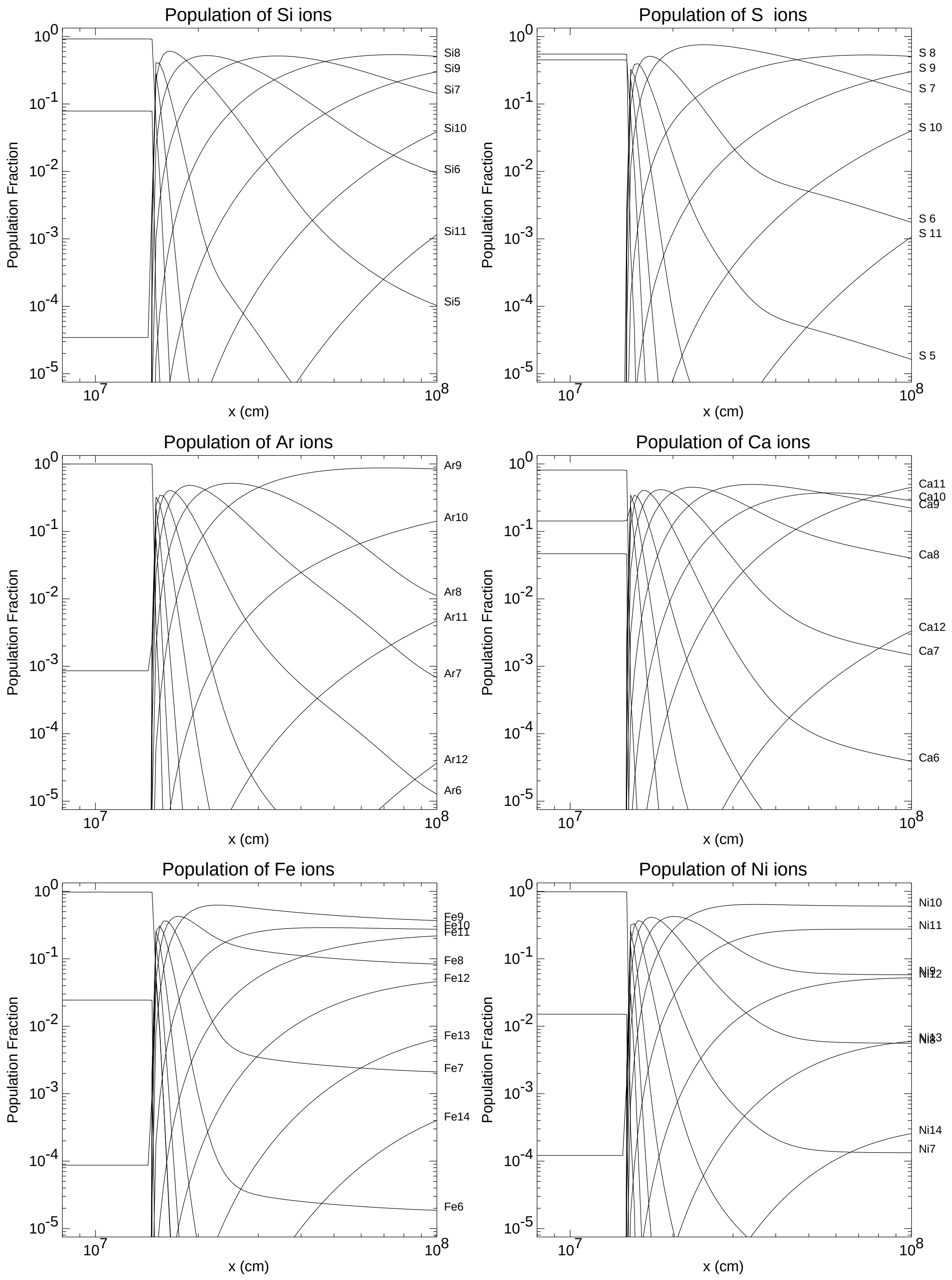


Figure 18.37: ... continued ...

Part IV

Tools

Chapter 19

Serial FLASH Output Comparison Utility (sfocu)

Sfocu (Serial Flash Output Comparison Utility) is intended as a replacement for focu, which was available in previous versions of FLASH and is mainly used as part of an automated testing suite called flash_test.

Sfocu is a serial utility which examines two FLASH checkpoint files and decides whether or not they are "equal" to ensure that any changes made to FLASH do not adversely affect subsequent simulation output. By "equal", we mean that

- The leaf-block structure matches – each leaf block must have the same position and size in both datasets.
- The data arrays in the leaf blocks (dens, pres...) are identical.

Thus, sfocu ignores information such as the particular numbering of the blocks, the timestamp, the build information, and so on.

Sfocu can read both HDF4 and HDF5 FLASH checkpoint files. It has not been tested with FLASH checkpoints that span multiple 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.

19.1 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_HDF4 build without HDF4 support

NO_HDF5 build without HDF5 support

NEED_MPI certain parallel versions of HDF5 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.

19.2 Using sfocu

There are no command line options. Simply run the command sfocu <file1> <file2>. You will most likely need to widen your terminal to view the output, since it is well over 80 columns. Sample output follows:

```
sfocu: comparing advect_2d_45deg_sqr_4lev_hdf_chk_0002 and advect_2d_45deg_gau_4lev_hdf_chk_0002

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)


advect_2d_45deg_sqr_4lev_hdf_chk_0002 has 2 leaf blocks that don't exist in advect_2d_45deg_gau_4lev_hdf_chk_0002
advect_2d_45deg_gau_4lev_hdf_chk_0002 has 8 leaf blocks that don't exist in advect_2d_45deg_sqr_4lev_hdf_chk_0002
Total leaf blocks compared: 53 (all other blocks are ignored)
-----+------------+-----------+-----------+-----------+-----------++-----------+-----------+-----------++-----------+-----------+-----------+
Var  | Bad Blocks | Min Error | Max Error | Abs Error | Mag Error ||    Sum    |    Max    |    Min    ||    Sum    |    Max    |    Min    |
-----+------------+-----------+-----------+-----------+-----------++-----------+-----------+-----------++-----------+-----------+-----------+
pres | 53         | 1.11e-16  | 3.234e-13 | 3.234e-13 | 3.234e-13 ||  3.39e+03 | 1         | 1         ||  3.39e+03 | 1         | 1         |
gamc | 0          | 0         | 0         | 0         | 0         ||  4.75e+03 | 1.4       | 1.4       ||  4.75e+03 | 1.4       | 1.4       |
game | 0          | 0         | 0         | 0         | 0         ||  4.75e+03 | 1.4       | 1.4       ||  4.75e+03 | 1.4       | 1.4       |
dens | 53         | 1.694e-16 | 1.998     | 0.6311    | 0.6311    ||  714      | 1         | 1e-05     ||  619      | 0.958     | 1e-05     |
velx | 53         | 1.256e-16 | 2.373e-13 | 1.678e-12 | 2.373e-13 ||  2.4e+04  | 7.07      | 7.07      ||  2.4e+04  | 7.07      | 7.07      |
temp | 53         | 1.803e-16 | 1.998     | 0.001194  | 0.9932    ||  2.43     | 0.0012    | 1.2e-08   ||  1.5      | 0.0012    | 1.26e-08  |
velz | 0          | 0         | 0         | 0         | 0         ||  0        | 0         | 0         ||  0        | 0         | 0         |
ener | 53         | 1.164e-16 | 1.997     | 2.483e+05 | 0.993     ||  5.05e+08 | 2.5e+05   | 52.5      ||  3.12e+08 | 2.5e+05   | 52.6      |
vely | 53         | 1.256e-16 | 2.007e-13 | 1.419e-12 | 2.007e-13 ||  2.4e+04  | 7.07      | 7.07      ||  2.4e+04  | 7.07      | 7.07      |
 1   | 53         | 1.11e-16  | 2.499e-09 | 2.499e-09 | 2.499e-09 ||  3.39e+03 | 1         | 1         ||  3.39e+03 | 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's possible for `sfocu` to miss machine-precision variations in the data on certain machines because of compiler or library issues (or possibly even because of bugs!). This has only been observed on one platform, where the compiler produced code that ignored IEEE rules until the right flag was found.

Chapter 20

FLASH IDL routines (fidlr2)

fidlr2 is a set of routines written in IDL that can read and plot data files produced by FLASH. These routines have been rewritten from previous versions of FLASH to provide more features and expandability. The routines include programs which can be run from the IDL command line to read 1D, 2D, or 3D FLASH datasets, interactively analyze datasets, and interpolate them onto uniform grids. A graphical interface to these routines (xflash) is provided, which enables users to read FLASH AMR datasets or global integral files (flash.dat) and make plots and histograms of the data. It is assumed that the user has some familiarity with IDL. The examples discussed below will use the Sedov data generated in the quickstart at the beginning of this manual.

The fidlr2 routines support 1-, 2-, and 3-dimensional datasets in the FLASH HDF or HDF5 formats. Both plotfiles and checkpoint files, which differ only in the number and numerical precision of the variables stored, are supported. Additionally, 1-d plots from the flash.dat files can be made as well. Since present versions of IDL do not directly support HDF5, the call_external function is used to interface with a set of C routines that read in the HDF5 data (see Chapter 7). Using these routines requires that the HDF5 library be installed on your system and that the shared-object wrapper library be compiled before reading in data. Since the call_external function is used, the demo version of IDL will not run the routines.

20.1 Installing and running fidlr2

fidlr2 is distributed with FLASH and is contained in the tools/fidlr2/ directory. In addition to the IDL procedures, a README file is present which will contain up-to-date information on changes and installation requirements.

These routines were written and tested using IDL v5.4 for Linux. They should work without difficulty on any UNIX machine with IDL installed—any functionality of fidlr2 under Windows is purely coincidental. Later versions of IDL no longer support GIF files due to copyright difficulties, so PNG images will be produced in their place. Most graphics packages, like xv or the GIMP, should be able to convert between these formats. It is possible to run IDL in a 'demo' mode if there are not enough licenses available. Unfortunately, some parts of fidlr2 will not function in this mode, since certain features of IDL are disabled. In particular, the call_external function is disabled, making the HDF5 wrappers inoperable. This guide assumes that you are running the full version of IDL.

Installation of fidlr2 requires defining some environment variables, making sure your IDL path is properly set, and compiling the support for HDF5 files. These procedures are described below.

20.1.1 Setting up fidlr2 environment variables

The FLASH IDL routines are located in the tools/fidlr2/ subdirectory of the FLASH root directory. To use them you must set two environment variables. First set the value of XFLASH_DIR to the location of the FLASH IDL routines; for example, under csh, use

setenv XFLASH_DIR *flash-root-path*/tools/fidlr2 , where *flash-root-path* is the absolute path of the FLASH root directory. This variable is used in the plotting routines to find the customized color table for `xflash`, as well as to identify the location of the shared-object libraries compiled for HDF5 support.

Next, make sure that you have an `IDL_DIR` environment variable set. This should point to the directory in which the IDL distribution is installed. For example, if IDL is installed in *idl-root-path*, then you would define

`setenv IDL_DIR` *idl-root-path* .

Finally, you need to tell IDL where to find the `fidlr2` routines. This is accomplished through the `IDL_PATH` environment variable

`setenv IDL_PATH ${XFLASH_DIR}:${IDL_DIR}:${IDL_DIR}/lib` .

If you already have an `IDL_PATH` environment variable defined, just add `XFLASH_DIR` to the beginning of it. You may wish to include these commands in your `.cshrc` (or the analogous versions in your `.profile` file, depending on your shell) to avoid having to reissue them every time you log in. It is important that the `${XFLASH_DIR}` come before the IDL directories in the path and that the `${IDL_DIR}/lib` directory be included as well (otherwise, the compound widget procedures, `cw_*.pro`, will not be found).

20.1.2 Setting up the HDF5 routines

For `fidlr2` to read HDF5 files, you need to install the HDF5 library on your machine and compile the wrapper routines. The HDF5 libraries can be obtained in either source or binary form for most Unix platforms from `http://hdf.ncsa.uiuc.edu`. Two sample Makefiles are provided. `Makefile.sgi` will build the shared-object library on an SGI. `Makefile.linux` will build the library on a Linux machine. In both cases, the Makefile will need to be edited to point to the directories where IDL and HDF5 are installed on the local machine. The compiler flags for a shared-object library for different versions of Unix can be found in

`${IDL_DIR}/external/call_external/C/callext_unix.txt` .

for IDL versions ≤ 5.5, and by typing

```
print, !MAKE_DLL
```

at the IDL prompt for IDL 5.6.

The compilation flags in the Makefile should be modified according to the instructions in that file or the output of the `!MAKE_DLL` IDL system variable. Again, the path to the HDF5 library needs to be supplied so the header files can be used in the compilation process, and for the linking stage. The path to the IDL header `export.h` is needed as well, to deal with IDL strings. IDL defines strings as structures, with a field of the `IDL_STRING` structure giving the length of the string. The datatype of this field has changed from version to version, so we need to get the actual definition from the IDL header files.

It is important that you compile the shared-object to conform to the same application binary interface (ABI) that IDL was compiled with. This is mainly an issue on an SGI, where IDL version 5.2.1 and later use the n32 ABI, while versions before this are o32. The HDF library will also need to be compiled in the same format. You can check the format of the HDF5 library and your version of IDL with the UNIX `file` command.

IDL interacts with external programs through the `call_external` function. Any arguments are passed by reference through the standard C command line argument interface, `argc` and `argv`. These are recast into pointers of the proper type in the C routines. The C wrappers call the appropriate HDF functions to read the data and return it through the `argc` pointers.

Finally, if the HDF5 library was installed as a shared-object, then the library must be in your shell's library search path. This can be set by defining the `LD_LIBRARY_PATH` environment variable to point to the HDF5 `lib/` subdirectory.

20.1.3 Running IDL

`fidlr2` uses 8-bit color tables for all of its plotting. On displays with higher color depths, it may be necessary to use color overlays to get the proper colors on your display. For SGI machines, launching IDL with the `start.pro` script will enable 8-bit pseudocolor overlays. For Linux boxes, setting the X color depth to 24-bits per pixel and launching IDL with the `start_linux.pro` script usually produces proper colors.

20.2 `fidlr2` data structures

For basic plotting operations, the easiest way to generate plots of FLASH data is to use the widget interface, `xflash`. For more advanced analysis, the read routines can be used to read the data into IDL, where it can be manipulated on the command line. In contrast to previous versions of `fidlr`, `fidlr2` does not use common blocks to share the data but rather passes everything though the argument lists. The tree and dataset parameters are returned as structures; the data itself is returned as a multidimensional array. The general way to read a FLASH HDF5 dataset is:

```
IDL> read_amr_new, 'sedov_2d_6lev_hdf_chk_0000', $
     TREE=tree, PARAMETERS=params, DATA=data, STORED_VARS=vars
```

The layout of the data reflects how it is stored in FLASH. It is assumed that the user is familiar with the block structured AMR format employed by FLASH (refer to Chapter 7 for full details of the output format). The optional arguments `TREE`, `PARAMETERS`, `DATA`, and `STORED_VARS` provide all of the information needed to interpret a dataset.

The `tree` structure array provides fields used to interpret the grid struture. `tree` takes an argument indicating the block number (zero based in IDL), and provides multiple fields giving the various parts of the tree. For example, `tree[0].lrefine` gives the refinement level of block 0. A full list of the tree fields is provided in Table 20.1.

Table 20.1: Fields in the `tree` structure.

Field	Description
`lrefine`	Refinement level of the block.
`nodeType`	Node type of the block. `tree[blk].nodeType` = 1 if block `blk` is a leaf block.
`gid[ngid]`	The global ID information for the current block, giving the (1-based) block numbers for the neighbors, parent, and children of the current block.
`coord[ndim]`	The coordinates of the block center in each direction.
`size[ndim]`	The size of the block in each direction.
`bndBox[2,ndim]`	The lower (`tree[blk].bndBox[0,*]`) and upper (`tree[blk].bndBox[1,*]`) coordinate of the block in each direction.

The `params` structure provides some basic information describing the dataset, including the total number of blocks (`params.totBlocks`) and the number of zones in each coordinate direction (`params.nxb`, `params.nyb`, and `params.nzb`). Table 20.2 lists the fields in the `params` structure.

All of the unknowns are stored together in the `data` array, which mirrors the `unk` array in FLASH. The list of variable names is contained in the string array `vars` through the `STORED_VAR` optional argument. The loaddata function provides a wrapper to read a single variable from a datafile and to merge it onto a uniform grid (2- and 3-d datasets) or to return a 1-d vector of data (1-d dataset). The coordinates are optionally returned.

Examples of using the fidlr2 routines from the command line are provided in Sec. 20.5. Additionally, some scripts demonstrating how to analyze FLASH data using the fidlr2 routines are described in Sec. 20.4 (see for example radial.pro).

Table 20.2: Fields in the params structure.

Field	Description
totBlocks	The total number of blocks in the dataset.
corners	A logical variable specifying whether the data was interpolated to corners before storage.
ndim	The number of dimensions of the simulation.
nxb	The number of zones per block in the x-direction.
nyb	The number of zones per block in the y-direction.
nzb	The number of zones per block in the z-direction.
ntopx	The number of top level blocks in the x-direction. This is equal to the Nblocksx FLASH runtime parameter.
ntopy	The number of top level blocks in the y-direction. This is equal to the Nblocksy FLASH runtime parameter.
ntopz	The number of top level blocks in the z-direction. This is equal to the Nblocksz FLASH runtime parameter.
time	The simulation time of the current dataset.
dt	The timestep used for the last step.

The driver routines provided with fidlr2 visualize the AMR data by first converting it to a uniform mesh. This allows for ease of plotting and manipulation, including contour plotting, but it is less efficient than plotting the native AMR structure. Efficiency is gained by recognizing that the data is restricted up the tree before output, so it is valid at all levels. If the dataset is much larger than the plot device, we ignore those levels that are finer than the device resolution and produce the plot using blocks closer to the root of the tree. In 3-d, slices through the dataset are made by putting only the 2-d slice itself onto a uniform grid. Analysis can still be performed directly on the native data through the command line.

20.3 xflash: widget interface to plotting FLASH datasets

The main interface to the fidlr2 routines for plotting FLASH datasets is xflash. Typing xflash at the IDL command prompt will launch the main xflash widget, shown in Fig. 20.1. xflash produces colormap plots of FLASH data with optional overlays of velocity vectors, contours, and the AMR block structure and histogram plots showing the distribution of data. The basic operation of xflash is to specify a single file in the dataset as a prototype for the FLASH simulation. The prototype is probed for the list of variables it contains, and then the remaining plot options become active.

`xflash` can output to the screen, postscript, or an image file (GIF/PNG). If the data is significantly higher resolution than the output device, `xflash` (through `xplot_amr.pro`) will sub-sample the image by one or more levels of refinement before plotting.

Once the image is plotted, the *query* (2-d data only) and *1-d slice* (1 and 2-d data only) buttons will become active. Pressing *query* and then clicking anywhere in the domain will pop up a window containing the values of all the FLASH variables in the zone nearest the cursor. The query function uses the actual FLASH data—not the interpolated/uniformly gridded data generated for the plots. Pressing *1-d slice* and then left-clicking on the plot will produce a 1-d slice vertically through the point. Right-clicking on the domain produces a horizontal slice through the data.

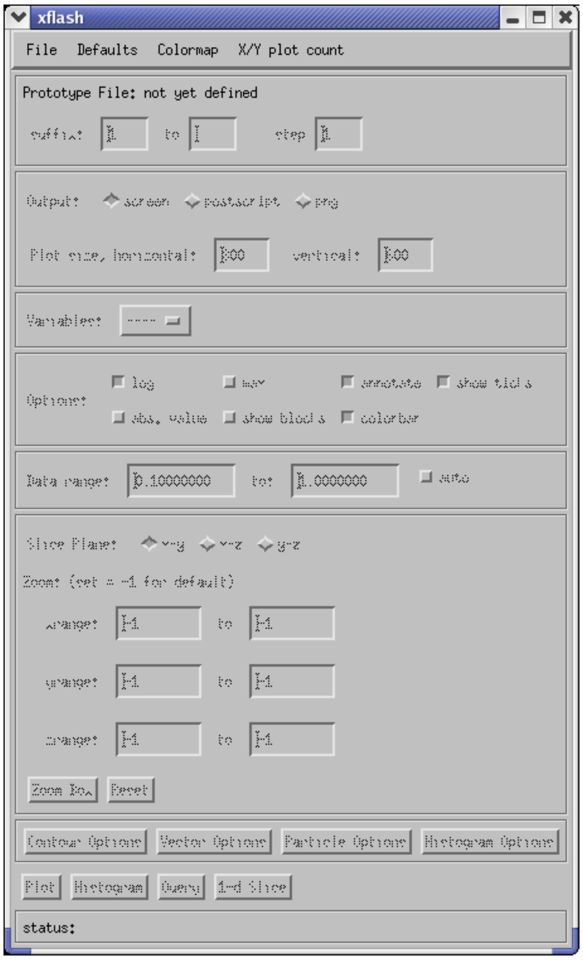


Figure 20.1: The main `xflash` widget.

The widget is broken into several sections, with some features initially disabled. Not all options are available in all dimensions. These sections are explained below.

File Menu

The file to be visualized is composed of the path, the basename (the same base name used in the flash.par file) with any file type information appended to it (*e.g.* `'hdf_chk_'`) and the range of suffixes through which to loop. By default, `xflash` sets the path to the working directory from which IDL was started. `xflash` requires a prototype file to work on a dataset. The prototype can be any of the files in the dataset that has the same name structure (*i.e.* everything is the same but the suffix) and contains the same variables.

File/Open prototype...

The *Open prototype...* menu option will bring up the file selection dialog box (see Fig. 20.2). Once a prototype is selected, the remaining options on the xflash widget become active, and the variable list is populated with the list of variables in the file (see Fig. 20.3).

`xflash` will automatically determine if the file is an HDF or HDF5 file and read the 'unknown names' record to get the variable list. This will work for both plotfiles and checkpoint files generated by FLASH. Some derived variables will also appear on the list (for example, sound speed), if the dependent variables are contained in the datafile. These derived variables are currently inoperable in 3-d.

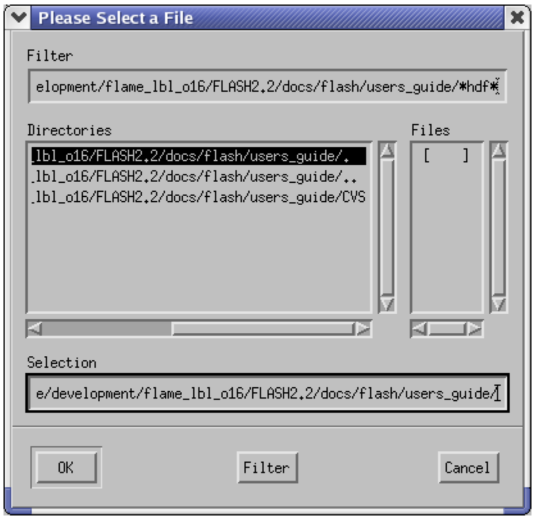


Figure 20.2: The `xflash` file selection dialog.

File/Information

The *Information* menu option becomes active once a prototype is defined. This will display a list of file information for the prototype file, showing the build information, runtime comment, number of variables, precision of the data, and FLASH version. Fig. 20.4 shows the file information widget. This information is also available from the IDL command line through the `file_information` procedure.

Defaults Menu

The *defaults* menu allows you to select one of the predefined problem defaults. This is provided for the convienence of users who want to plot the same problem repeatedly using the same data ranges. This will load the options (data ranges, velocity parameters, and contour options) for the problem as specified in the `xflash_defaults` procedure. When `xflash` is started, `xflash_defaults` is executed to read in the known problem names. The data ranges and velocity defaults are then updated. To add a problem to `xflash`, only the `xflash_defaults` procedure needs to be modified. The details of this procedure are provided in the comment header in `xflash_defaults`. It is not necessary to add a problem in order to plot a dataset, since all default values can be overridden through the widget.

Colormap Menu

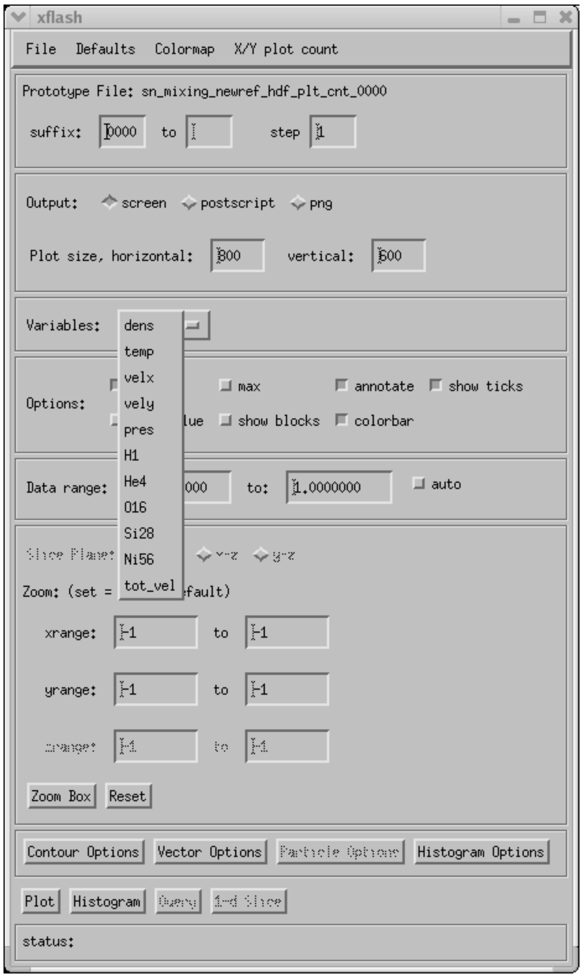


Figure 20.3: The `xflash` main window after a prototype has been selected, showing the variable list.

The *colormap menu* lists the colormaps available to `xflash`. These colormaps are stored in `flash_colors.tbl` in the `fidlr2` directory and differ from the standard IDL colormaps. The first 12 colors in the colormaps are reserved by `xflash` to hold the primary colors used for different aspects of the plotting. Additional colormaps can be created by using the `xpalette` function in IDL. It is suggested that new colormaps use one of the existing colormaps as a template in order to preserve the primary FLASH colors. These colormaps are used for 2-d and 3-d data only. At present, there is no control over the line color in 1-d.

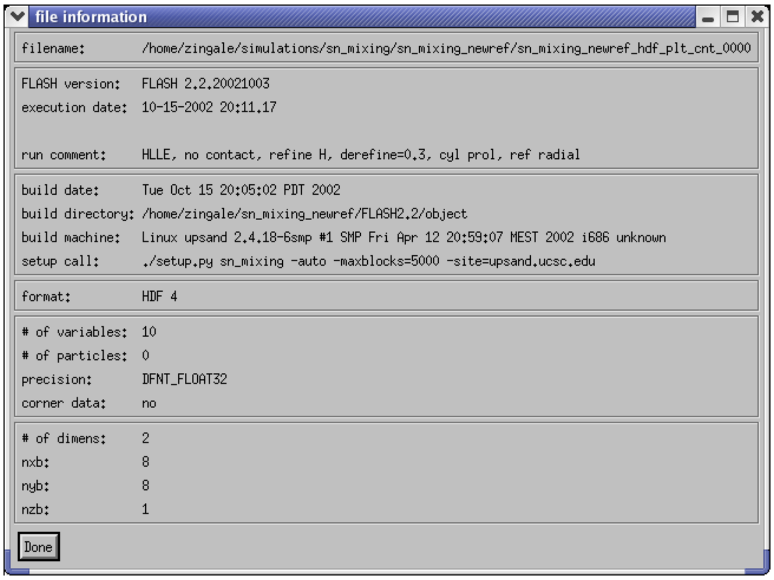


Figure 20.4: The `xflash` file information widget for the current prototype.

X/Y plot count

The *X/Y plot count* menu specifies how many plots to put on a single page when looping over suffixes in a dataset. At present, this only works for 2-d data. Note, the query and 1-d slice operations will not work if there are multiple plots per page.

File Options

The first section below the menu bar specifies the file options. This allows you to specify the range of files in the dataset (*i.e.* the suffixes) to loop over. The optional *step* parameter can be used to skip over files when looping through the dataset.

Output Options

A plot can be output to the screen (default), a Postscript file, or a GIF/PNG file. The output filenames are composed from the basename + variable name + suffix. For outputs to the screen or GIF/PNG, the *plot size* options allow you to specify the image size in pixels. For Postscript output, `xflash` chooses portrait or landscape orientation depending on the aspect ratio of the domain.

Variables

The variables dropbox lists the variables stored in the 'unknown names' record in the data file and any derived variables that `xflash` knows how to construct from these variables (*e.g.* sound speed). This allows you to choose the variable to be plotted. By default, `xflash` reads all the variables in a file in 1- and 2-d datasets, so switching the variable to plot can be done without re-reading. At present, there is no easy way to add a derived variable. Both the widget routine (`xflash.pro`) and the plotting backend (`xplot_amr.pro`) will need to be told about any new derived variables. Users wishing to add derived variables should look at how the total velocity (`tot_vel`) is computed.

Options

Table 20.3: `xflash` options

log	Plot the log of the variable.
max	When looping over a sequence of files, plot the max of the variable in each zone over all the files.
annotate	Toggle the title and time information.
abs value	Plot the absolute value of the dataset. This operation is performed before taking the log.
show blocks	Draw the block boundaries on the plot.
colorbar	Plot the colorbar legend for the data range.
show ticks	Show the axis tick marks on the plot.

The options block allows you to toggle various options on/off. Table 20.3 lists the various options available.

Data Range

These fields allow you to specify the range of the variable to plot. Data outside of the range will be set to the minimum or maximum values of the colormap. If the *auto* box is checked, the limits will be ignored, and the data will be scaled to the minimum and maximum values of the variable in the dataset.

Slice plane

The slice plane group is only active for 3-d datasets. This allows you to pick the plane that the plot is created in (x-y, x-z, y-z).

Zoom

The zoom options allow you to set the domain limits for the plot. A value of -1 uses the actual limit of the domain. For 3-d plots, only one field will be available in the direction perpendicular to the slice plane. The *zoom box* button puts a box cursor on the plot and allows you to select a region to zoom in on by positioning and resizing the box with the mouse. The *reset* button will reset the zoom limits.

Contour Options

This launches a dialog box that allows you to select up to 4 contour lines to plot on top of the colormap plot (see Fig. 20.5). The variable, value, and color are specified for each reference contour. To plot a contour, select the check box next to the contour number. This will allow you to set the variable from which to make the contour, the value of the contour, and the color. This is available in 2-d only at present.

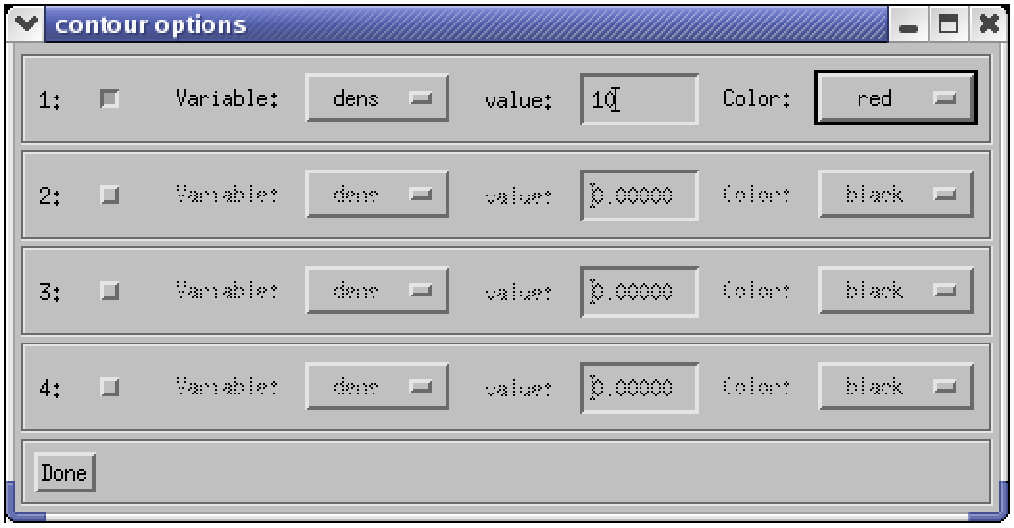


Figure 20.5: The `xflash` contour option subwidget.

Velocity Options

This launches a dialog box that allows you to set the velocity options used to plot velocity vectors on the plot (see Fig. 20.6). The plotting of velocity vectors is controlled by the `partvelvec.pro` procedure. *xskip* and *yskip* allow you to thin out the arrows. *typical velocity* sets the velocity to which to scale the vectors, and *minimum velocity* and *maximum velocity* specify the range of velocities for which to plot vectors. This is available in 2-d only.

Histogram Options

Pop up a dialog box (Fig. 20.7) that allows you to set the histogram options. Currently, only the number of bins and the scale of the y-axis can be set.

Plot

Create the colormap plot. The status of the plot will appear on the status bar at the bottom.

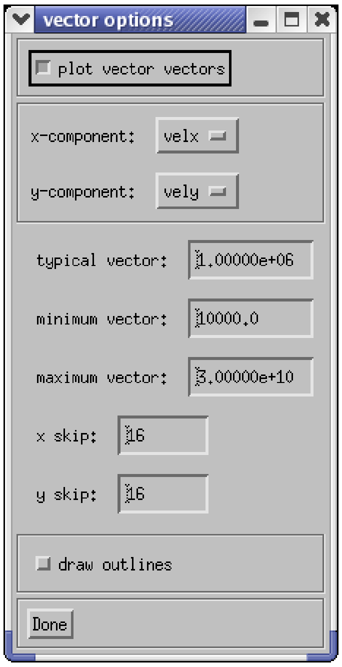


Figure 20.6: The `xflash` velocity option subwidget.

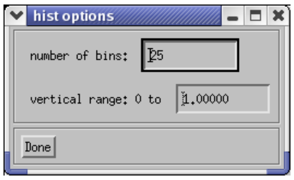


Figure 20.7: The xflash histogram options widget.

Histogram

Create a histogram of the data.

Query

The query button becomes active once the plot is created. Clicking on query and then clicking somewhere in the domain lists the data values at the cursor location (see Fig. 20.8).

1-d Slice

This is available for 2-d datasets only. Clicking on *1-d Slice* and then left-clicking in the domain will plot a 1-d slice of the current variable vertically through the point selected. A right-click will produce a horizontal slice. This function inherits the options chosen in the *Options* block.

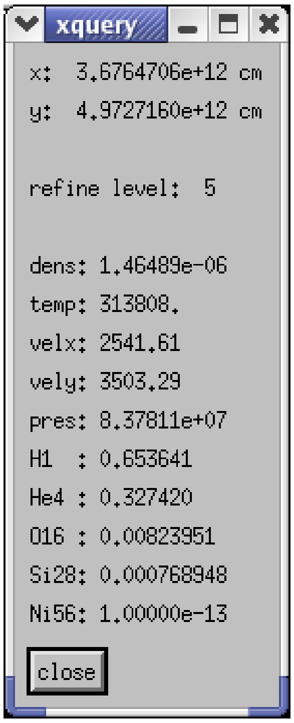


Figure 20.8: The `xflash` query widget, displaying information for a zone.

20.4 The `fidlr2` routines

Table 20.4 lists all of the `fidlr2` routines, grouped by function. Most of these routines rely on the common blocks to get the tree structure necessary to interpret a FLASH dataset. Command line analysis of FLASH data requires that the common blocks be defined, usually by executing `def_common.pro`.

Table 20.4: Description of the `fidlr2` routines

FLASH data readers	
`file_information.pro`	Dump out some basic information about the file, such as the number of variables stored, the runtime comment, the precision of the data, *etc.*
`get_dimensionality.pro`	Given a filename, return the dimensionality of the dataset.
`get_particle_number.pro`	Look at the particle records in the datafile and return the total number of particles in the dataset.
`get_var_list.pro`	Read the variable names record from a datafile and return the list of variables stored.
`openflashfile.pro`	Simple wrapper routine that reads a dataset block by block.
`read_amr.pro`	Read in FLASH data in HDF v 4.x format. This routine takes the filename and an optional variable argument and returns the tree, parameters, and data information through optional arguments. If the `DOUBLE` parameter is set, the data is returned as double precision. Otherwise, it is returned as single precision, regardless of how it is stored on disk.
`read_amr_hdf5.pro`	The HDF5 version of `read_amr.pro`. This routine uses the call_external function to access C wrappers of the HDF functions stored in `h5_wrappers.so`. The shared library must be compiled before using this routine. The options and arguments are the same as for the `read_amr` routine.
Driver routines	
`flame_profile_1d.pro`	A script that reads in a 1-d FLASH dataset and writes out a 1-d slice of data to an ASCII file. The format of the output file is identical to that required by the `sample_map` setup.
`flame_speed.pro`	Read in two FLASH files and compute the speed of a planar front by differencing.
`hist_driver.pro`	Driver routine for `hist.pro`. Loop over a range of files and produce histograms of the data.
`xcontour.pro`	The contour options widget. This allows you to select up to 4 reference contours to be overplotted on a 2-d plot. This widget is launched by `xflash.pro`.
`xfile_info.pro`	Widget interface to `file_information.pro` that reports the basic file information for `xflash.pro`.

Table 20.4: `fidlr2` routines—continued

`xflash.pro`	The main driver for 2d datasets. `xflash` provides a widget interface to select the variable, data range, contour options, output type, *etc.* This routine uses `xflash_defaults.pro` to define some default problem types and their options. Once the options are selected, `xplot_amr.pro` is used to create the plot.
`xflash_defaults.pro`	The problem default initialization file. Standard problems are given an entry in this file, defining the default values for the plot options. This file is read in by `xflash`.
`xhist.pro`	Widget interface to set the histogram options for `xflash`.
`xvelocity.pro`	The velocity options widget. This allows you to select the minimum, maximum, and typical velocities and the number of zones to skip when thinning out the vector field. This widget is launched by `xflash.pro.`
`xparticle.pro`	The particle options widget. This widget is launched by `xflash.pro.`
HDF routines	
`determine_file_type.pro`	IDL procedure to determine if a file is in HDF 4 format (return 1), HDF5 format (return 2), or neither (return -1). This routine uses the built in IDL HDF 4 implementation and some of the HDF5 wrappers in h5_wrappers.so.
`fhdf_read.pro`	Wrapper around IDL HDF 4 routines to read in a dataset given the file handle and dataset name.
`h5_file_interface.c`	HDF5 file open and close routines from the FLASH serial HDF5 implementation.
`h5_read.c`	Part of the serial HDF5 FLASH routines used to read in the header information from the data file.
`h5_wrappers.c`	Wrappers around the HDF5 library to read in the different records from the FLASH HDF5 file.
`h5_wrappers.so`	Shared-object library produced from the above routines. The IDL routines interface with this object through the `call`$_e$`xternal` function.
`hdf5_idl_interface.h`	Header file for the C wrappers.
`Makefile.sgi`, `Makefile.linux`	Makefile for compiling the HDF5 support on an SGI IRIX or Linux box, respectively. Other machines should behave similarly, but some of the compilation flags may differ.

Table 20.4: fidlr2 routines—continued

Merging routines	
extract_line.pro	Extract a 1-d line through a 2-d dataset given the line direction (x or y) and a point through which it passes.
loaddata.pro	Wrapper around the read and merge routine for 1-, 2-, and 3-d data. Given a filename and variable name, return a uniformly gridded array of data and (optionally) the coordinate information. 1-d data are returned as a vector and are not uniformly gridded.
merge_amr.pro	Take a 2-d or 3-d FLASH variable and associated parameter and tree structures and merge it onto a uniform grid (or 2-d slice of a 3-d volume if desired). A double precision slice can be made if the DOUBLE keyword is set and the data was read in with double precision.
Plotting routines	
colorbar2.pro	Create a horizontal colorbar.
color_index.pro	Return the names of the colortables available.
draw_blocks.pro	Draw the AMR block boundaries on the plot. This routine is called from xflash.
hist.pro	Make a histogram plot of a single variable from a single file.
readtab.pro	Given a columnar ASCII file containing floating point data, return an array containing that data. This routine automatically determines the number of rows and columns.
partvelvec.pro	Overplot the velocity vectors for velocities that fall within a specified minimum and maximum velocity. The vectors are scaled to a typical velocity. This routine also handles the plotting of the particle data.
vcolorbar.pro	Create a vertical colorbar given the data range and color bounds.
xplot1d_amr.pro	Back end to xflash—create a plot of a 1-d FLASH dataset using the options selected in the widget.
xplot3d_amr.pro	Back end to xflash—plot a slice through a 3-d FLASH dataset.
xplot_amr.pro	Back end to xflash—plot a 2-d dataset.

Table 20.4: `fidlr2` routines—continued

Utility routines	
`add_var.pro`	add_var is used to add a derived variable to the list of variables recognized by the `xflash` routines.
`color.pro`	color returns the index into the color table of a color specified by a string name.
`color_gif.pro`	Create a GIF or PNG image of the current plot window.
`courant.pro`	Loop over the blocks and return the block number in which the Courant condition is set.
`def_common.pro`	Define the common blocks that hold the variables read in from the read routines. This routine can be used on the IDL command line so that the FLASH data can be analyzed interactively.
`flash_colors.tbl`	Replace color table with the standard FLASH colormaps.
`nolabel.pro`	A hack used to plot an axis w/o numbers.
`query.pro`	A widget routine called by `xflash` that displays the data in a cell of the current plot.
`query1d.pro`	Query routine for 1-d data, called from `xflash`.
`scale3d_amr.pro`	Scale a uniformly gridded 3-d dataset into a single byte.
`scale_color.pro`	Scale a dataset into a single byte.
`sci_notat.pro`	Print a number in scientific notation.
`start.pro`, `start_linux.pro`	A script used to initialize IDL on the SGIs and Linux boxes.
`tvimage.pro`	Replacement for tv that will write to postscript or the screen in a device independent manner.
`undefine.pro`	Free up the memory used by a variable.
`var_index.pro`	Return the index into the data array of the variable label passed as an argument.
`write_brick_f77.pro`	Write a block of data to a file in f77 binary format.

20.5 fidlr2 command line examples

Most of the fidlr2 routines can be used directly from the command line to perform analysis not offered by the different widget interfaces. This section provides an example of using the fidlr2 routines.

Example. Report on the basic information about a FLASH data file.

```
IDL> file_information, 'sedov_2d_6lev_hdf_chk_0000'

--------------------------------------------------------------------------------
 file  = sedov_2d_6lev_hdf_chk_0000

 FLASH version:       FLASH 2.0.20010802
 file format:         HDF 4

 execution date:      08-03-2001  12:59.20
 run comment:         2D Sedov explosion, from t=0 with r_init = 3.5dx_min

 dimension:                  2
 geometry detected:   Cartesian (assumed)

 type of file:        checkpoint

 number of variables:           12
 variable precision:  DFNT_FLOAT64

 number of particles:        0

 nxb, nyb, nzb:                  8           8           1
 corners stored:      no

IDL>
```

Example. Read in the pressure field from a file, put it on a uniform grid, and make a contour plot.

```
IDL> spres = loaddata('sedov_2d_6lev_hdf_chk_0000', 'pres', XCOORDS=x, YCOORDS=y)
IDL> help, spres
SPRES           FLOAT     = Array[256, 256]
IDL> contour, spres, x, y
```

Part V

Going Further with FLASH

Chapter 21

Adding new solvers

Adding new solvers (either for new or existing physics) to FLASH is similar in some ways to adding a problem configuration. In general, one creates a subdirectory for the solver, placing it under the source subdirectory for the parent module, if the solver implements currently supported physics, or creating a new module subdirectory, if it does not. Put the source files required by the solver into this directory and then create the following files:

`Makefile`: The make include file for the module should set a macro with the name of the module equal to a list of the object files in the module. Optionally (recommended), add a list of dependencies for each of the source files in the module. For example, the `source_terms` module's make include file is

```
#       Makefile for source term solvers

source_terms = source_termsModule.o burn.o heat.o cool.o init_burn.o init_heat.o \
               init_cool.o tstep_burn.o tstep_heat.o tstep_cool.o init_src.o

source_termsModule.o    : source_termsModule.F90 dBase.o
burn.o          : burn.F90
heat.o          : heat.F90
cool.o          : cool.F90
init_src.o      : init_src.F90 dBase.o
init_burn.o     : init_burn.F90
init_heat.o     : init_heat.F90
init_cool.o     : init_cool.F90
tstep_burn.o    : tstep_burn.F90
tstep_heat.o    : tstep_heat.F90
tstep_cool.o    : tstep_cool.F90
```

Sub-module make include files use macro concatenation to add to their parent modules' make include files. For example, the `source_terms/burn` sub-module has the following make include file

```
#       Makefile for the nuclear burning sub-module

source_terms += burn_block.o net_auxillary.o net_integrate.o sparse_ma28.o \
                gift.o net.o shock_detect.o

burn_block.o     : burn_block.F90 dBase.o network_common.fh eos_common.fh net.o
net_auxillary.o  : net_auxillary.F90 network_common.fh eos_common.fh
net_integrate.o  : net_integrate.F90 network_common.fh
sparse_ma28.o    : sparse_ma28.F90
net.o            : net.F90 network_common.fh eos_common.fh
gift.o           : gift.F90
shock_detect.o   : shock_detect.F90 dBase.o

network_common.fh : network_size.fh
        touch network_common.fh

#       Additional dependencies

burn.o           : dBase.o network_common.fh net.o
init_burn.o      : dBase.o network_common.fh net.o
```

Note that the sub-module's make include file only makes reference to files provided at its own level of the directory hierarchy. If the sub-module provides special versions of routines to override those supplied by the parent module, they do not need to be mentioned again in the object list, because the sub-module's Makefile is concatenated with its parent's. However, if these special versions have additional dependencies, they can be specified as shown. Of course, any files supplied by the sub-module that are not part of the parent module should be mentioned in the sub-module's object list, and their dependencies should be included.

If you are creating a new top-level module, your source files at this level will be included in the code, even if you do not request the module in `Modules`. However, no sub-modules will be included. If you intend to have special versions of these files (stubs) that are used when the module is not included, create a sub-module named `null` and place them in it. `null` is automatically included if it is found and the module is not referenced in `Modules`.

If you create a top-level module with no `Makefile`, `setup` will automatically generate an empty one. For example, creating a directory named `my_module/` in `source/` causes `setup` to generate a `Makefile.my_module` in the build directory with contents

```
my_module =
```

If sub-modules of `my_module` exist and are requested, their Makefiles will be appended to this base.

`Config`: Create a configuration file for the module or sub-module you are creating. All configuration files in a sub-module path are used by `setup`, so a sub-module inherits its parent module's configuration. `Config` should declare any runtime parameters you wish to make available to the code when this module is included. It should indicate which (if any) other modules your module requires in order to function, and it should indicate which (if any) of its sub-modules should be used as a default if none is specified when `setup` is run. The configuration file format is described in Sec. 4.1.

This is all that is necessary to add a module or sub-module to the code. However, it is not sufficient to have the module routines called by the code! If you are creating a new solver for an existing physics module, the module itself should provide the interface layer to the rest of the code. As long as your sub-module provides the routines expected by the interface layer, the sub-module should be ready to work. However, if you are adding a new module (or if your sub-module has externally visible routines – a no-no for the future), you will need to add calls to your externally visible routines.

It is difficult to give complete general guidance; here we simply note a few things to keep in mind. If you wish to be able to turn your module on or off without recompiling the code, create a new runtime parameter (*e.g.*, `use_`*module*) in the `driver` module. You can then test the value of this parameter before calling your externally visible routines from the main code. For example, the `burn` module routines are only called if (`iburn .eq. 1`). (Of course, if the `burn` module is not included in the code, setting `iburn` to 1 will result in empty subroutine calls.)

You will need to add `use dBase` if you wish to have access to the global AMR data structure. Since this is the only mechanism for operating on the grid data in FLASH, you will probably want to do this. An alternative, if your module uses a pre-existing data structure, is to create an interface layer which converts the PARAMESH-inspired tree data structure used by FLASH into your data structure and then calls your routines. This will probably have some performance impact, but it will enable you to get things working quickly.

You may wish to create an initialization routine for your module that is called before anything (*e.g.*, setting initial conditions) is done. In this case you should call the routine init_*module*() and place a call to it (without any arguments) in the main initialization routine, `init_flash.F90`, which is part of the `driver` module. Be sure this routine has a stub available.

If your solver introduces a constraint on the timestep, you should create a routine named `tstep_`*module*() that computes this constraint. Add a call to this routine in `timestep.F90` (part of the `driver/time_dep` module) using your global switch parameter, if you have created one. See this file for examples. Your routine should operate on a single block and take three parameters: the timestep variable (a real variable which you should set to the smaller of itself and your constraint before returning), the minimum timestep location (an integer array with five elements), and the block identifier (an integer). Returning anything for the minimum location is optional, but the other timestep routines interpret it in the following way. The first three elements are set to the coordinates within the block of the zone contributing the minimum timestep. The fourth element is set to the block identifier, and the fifth is set to the current processor identifier (`MyPE`). This information tags, along with the timestep constraint, when blocks and solvers are compared across processors, and it is printed on stdout by the master processor along with the timestep information as FLASH advances.

If your solver is time-dependent, you will need to add a call to your solver in the `evolve()` routine (`driver/time_dep/evolve.F90`). If it is time-independent, add the call to `driver/steady/flash.F90`. The default version of `evolve()` implements second-order Strang time splitting within the time loop maintained by `driver/time_dep/flash.F90`. The `steady` version of `flash.F90` simply calls each operator once and then exits.

Try to limit the number of entry points to your module. This will make it easier to update it when the next version of FLASH is released. It will also help to keep you sane.

Chapter 22

Porting FLASH to other machines

Porting FLASH to new architectures should be fairly straightforward for most Unix or Unix-like systems. For systems which look nothing like Unix or which have no ability to interpret the setup script or makefiles, extensive reworking of the meta-code which configures and builds FLASH would be necessary. We do not treat such systems here; rather than do so, it would be simpler for us to do the port ourselves. The good news in such cases is that, assuming that you can get the source tree configured on your system and that you have a Fortran 90 compiler and the other requirements discussed in Chapter 2, you should be able to compile the code without making too many changes to the source. We have generally tried to stick to standard Fortran 90, avoiding machine-specific features.

For Unix-like systems, you should make sure that your system has `csh`, a `gmake` that permits included makefiles, `awk`, `sed`, and `python`. Next, create a directory in `source/sites/` with the name of your site (or a directory in `source/sites/Prototypes/` with the name of your operating system). This directory should contain at least a makefile fragment named `Makefile.h`. The best way to start is to copy one of the existing makefile fragments to your new directory and modify that. `Makefile.h` sets macros that define the names of your compilers, the compiler and linker flags, the names of additional object files needed for your machine but not included in the standard source distribution, and additional shell commands (such as file renaming and deletion commands) needed for processing the master makefile.

For most Unix systems, this will be all you need to do. However, in addition to `Makefile.h` you may need to create machine-specific subroutines that override the defaults included with the main source code. As long as the files containing these routines duplicate the existing routines' filenames, they do not need to be added to the machine-dependent object list in `Makefile.h`; `setup` will automatically find the special routine in the system-specific directory and link to it rather than to the general routine in the main source directories. An example of such a routine is `getarg()`, which returns command-line arguments and is used by FLASH to read the name of the runtime parameter file from the command line. This routine is not part of the Fortran 90 standard, but it is available on many Unix systems without the need to link to a special library. However, it is not available on the Cray T3E; instead, a routine named `pxfgetarg()` provides the same functionality. Therefore, we have encapsulated the `getarg()` functionality in a routine named `get_arguments()`, which is part of the `driver` module, in a file named `getarg.F90`. The default version simply calls `getarg()`. For the T3E, a replacement `getarg.F90` that calls `pxfgetarg()` is supplied. Since this file overrides a default file with the same name, `getarg.o` does not need to be added to the machine-dependent object list in `source/sites/Prototypes/UNICOS/Makefile.h`.

22.1 Writing a `Makefile.h`

To create a custom `Makefile.h` for your site, create a directory under source/sites that is the name returned by hostname (*e.g.* sphere.uchicago.edu). In this directory, copy the `Makefile.h` from the Prototypes directory that matches your system most closely. Currently, the Makefile in sphere.uchicago.edu is usually the most up-to-date and should be used as a starting point, if no other sites seem appropriate. To reflect your system, you must modify the different macros defined in `Makefile.h`.

22.1.1 Makefile macros

Listed and described below are macros defined in the Makefile.h:

`FCOMP` : the name of the Fortran 90 compiler

`CCOMP` : the name of the C compiler

`CPPCOMP` : the name of the C++ compiler

`LINK` : the name of the linker (usually the Fortran compiler should serve as the linker)

`PP` : a flag (if any) that should precede a preprocessor directive (typically -D)

`FFLAGS_OPT` : the Fortran compilation flags to produce an optimized executable. These are the flags used when -auto is given to setup.

`FFLAGS_DEBUG` : the Fortran compilation flags to produce an executable that can be used with a debugger (*e.g.* totalview). These flags are used when -debug is passed to setup.

`FFLAGS_TEST` : Fortran compilation flags to produce an executable suitable for testing. These usually involve less optimization. These flags are used when -test is passed to setup.

`CFLAGS_OPT` : the optimized set of compilation flags for C/C++ code.

`CFLAGS_DEBUG` : the debug compilation flags for C/C++ code.

`CFLAGS_TEST` : the test compilation flags for C/C++ code.

`LFLAGS_OPT` : linker flags used with the `_OPT` compilation flags. This usually ends in '-o' to rename the executable.

`LFLAGS_DEBUG` : linker flags used with the `_DEBUG` compilation.

`LFLAGS_TEST` : linker flags used with the `_TEST` compilation.

`LIB_OPT` : libraries to link in with the `_OPT` compilation. This should include the MPI library if an MPI wrapper for the linker was not used (*e.g.* `mpif90`).

`LIB_DEBUG` : libraries to link in the with the `_DEBUG` compilation.

`LIB_TEST` : libraries to link in with the `_TEST` compilation.

`LIB_HDF4` : the necessary link line required to link the `HDF4` library. This will be something of the form

```
L /path/to/library -lmfhdf -ldf -ljpeg -lz
```

`LIB_HDF5` the necessary link line required to link in the `HDF5` library. This will look something like

```
-L /path/to/library -lhdf5
```

For example, here's how you might modify the macros defined in the `Makefile.h` in sphere.uchicago.edu/. The first part of `Makefile.h` defines the paths for the libraries that FLASH requires (HDF/HDF5).

```
HDF4_PATH = /opt/pkgs/HDF/4.1r2_irix64v6.4-n32
HDF5_PATH = /opt/pkgs/HDF5-1.4.0-irix64n32
ZLIB_PATH = /opt/pkgs/zlib-1.1.3
```

These should be modified to reflect the locations on your system. For some machines, zlib is needed in addition to the HDF5 library in order to resolve all of the dependencies.

Next we setup the compilers and linker. We almost always use the Fortran 90 compiler as the linker, so the Fortran libraries are automatically linked in.

```
FCOMP   = f90
CCOMP   = cc
CPPCOMP = CC
LINK    = f90

# pre-processor flag
PP      = -D
```

These commands will need to be changed if your compiler names are different. Note that on some systems (*i.e.* those with MPICH as the MPI implementation), there are wrappers for the compilers (`mpif90`, `mpicc`, `mpiCC`) that automatically add the proper flags for the include files and the libraries to the link line when building. You are encouraged to use these wrappers. Some older versions of MPICH do not recognize `.F90` as a valid extension. For these, you can either update MPICH to a later version or edit `mpif90` and add `.F90` to the line that checks for a valid extension. The `PP` macro refers to the pre-processor and should be set to the flag that the compiler uses to pass information to the C preprocessor (usually `-D`).

We define three different setups of compiler flags as described earlier: the "_OPT" set for normal, fully optimized code, the "_DEBUG" set for debugging FLASH, and the "_TEST" set for regression testing. This latter set usually has less optimization. These three sets are picked with the `-auto`, `-debug`, and `-test` flags to `setup` respectively.

```
FFLAGS_OPT = -c -Ofast=ip27 -OPT:Olimit=0:IEEE_arithmetic=3:roundoff=3 -IPA \
             -r8 -d8 -i4 -cpp -r10000 -LNO
FFLAGS_DEBUG = -c -DEBUG:subscript_check=ON:verbose_runtime=ON -r8 -d8 -i4 \
               -cpp -g
FFLAGS_TEST = -c -r8 -d8 -i4 -cpp -O2

F90FLAGS =

CFLAGS_OPT = -IPA -Ofast=ip27 -c
CFLAGS_DEBUG = -g -c
CFLAGS_TEST = -c -O2
```

Next come the linker flags. Typically, these have only `-o` to rename the executable, and some debug flags (*e.g.* `-g`) for the "_DEBUG" set.

```
LFLAGS_OPT = -r8 -d8 -i4 -IPA -o
LFLAGS_DEBUG = -r8 -d8 -i4 -g -o
LFLAGS_TEST = -r8 -d8 -i4 -o
```

There are two different groups of library macros defined in the `Makefile.h`: one group corresponding to the "_OPT", "_DEBUG", "_TEST" flags defined above, and the other group for any libraries that are required by a specific module (*e.g.* HDF5). Any module can require a certain library by putting a line like

```
LIBRARY xxx
```

in its `Config` file. This library requirement will be satisfied in the `Makefile.h` by creating a macro called `LIB_xxx`, and setting it equal to the libraries (including any path) that are required to resolve the dependencies in that module.

For the sphere.uchicago.edu `Makefile.h` we have

```
LIB_HDF4 = -L$(HDF4_PATH)/lib -lmfhdf -ldf -lz

CFLAGS_HDF5 = -I $(HDF5_PATH)/include
LIB_HDF5 = -B static -L $(HDF5_PATH)/lib -lhdf5 \
           -B dynamic -L $(ZLIB_PATH)/lib -lz

CFLAGS_VISTOOLS = -I /scratch2/caceres/python/include/python2.1
LIB_VISTOOLS = -L /scratch2/caceres/python/src -lpython2.1 \
               -lpthread

LIB_OPT = -L/usr/lib32 -lmpi -lfastm
LIB_DEBUG = -L/usr/lib32 -lmpi -lfastm
LIB_TEST = -L/usr/lib32 -lmpi
```

Note that there are also two `CFLAGS_xxx` macros here. They specify the include path for the library if any header files are needed. When `setup` generates the master `Makefile`, it will append the individual `CFLAGS_xxx` lines to the master `CFLAGS` macro and the `LIB_xxx` lines to the master `LIB` macro.

Finally, we have a macro to specify any platform dependent code and some macros for basic file manipulation and library tools.

```
MACHOBJ =

MV = mv -f
AR = ar -r
RM = rm -f
CD = cd
RL = ranlib
ECHO = echo
```

On most platforms, these will not need to be modified.

It is strongly suggested that you use the compiler flags from the prototype `Makefile.h` that matches your system, if one exists. These are the flags that we use when testing FLASH.

22.2 Troubleshooting

This section addresses some known problems that users and developers alike have encountered in compiling FLASH on a number of different machines.

22.2.1 General questions about compiling FLASH

22.2.1.1 When I try to make FLASH, I get the following error:

```
make:
file 'Makefile.driver' line 34: Must be a separator (: or ::) for rules (bu39)
```

FLASH requires the use of GNU make (usually called `gmake`) to build the code.

22.2.1.2 I noticed that FLASH uses the REAL declaration for single precision. Is there a simple way to make sure the computer calculates to DOUBLE PRECISION, even though the variables are defined with REAL?

Although the variables are all declared as real instead of double precision (or real*8) in the code and constants are written as `1.e30` instead of `1.d30`, the Makefiles for the different platforms use compiler switches to promote the reals to double precision. This is necessary, since on some platforms (*e.g.* Cray T3E), double precision is 16-byte precision.

22.2.1.3 When I make FLASH, lots of compilation lines are output, but no object (.o) files are produced in object/—what's up?

Older versions of MPICH do not recognize .F90 as a valid Fortran file extension in the mpif90 compiler wrapper. All FLASH Fortran files end in .F90 to signify that they are free-form and require preprocessing. To fix this, edit the mpif90 wrapper and add ".F90" to the if test on the file extension. It should look something like this

```
    if [ "$ext" = ".f" -o "$ext" = ".F" -o "$ext" = ".f90" -o \
 "$ext" = ".for" -o "$ext" = ".FOR" -o "$ext" = ".F90" ] ; then
```

This should make mpif90 recognize .F90 files.

22.2.2 Runtime errors

22.2.2.1 The detonation problem dies after the first timestep on an SGI with the error message:

```
matrix is structurally singular, rank = 1~.
```

Originally, the MIPS Pro 7.3.X series of compilers did not work properly with FLASH, and we recommended that users try dropping down to the 7.2 series, or else compile without the -IPA switch. However, skipping the interprocedural inlining will significantly affect performance. Recently, however, this problem has been fixed by the 7.3.2.1m series compilers. A test suite comparison of the 7.2 series and 7.3.2.1 compilers found no differences in the FLASH results.

22.2.2.2 I get the an mmap error when running a large job (>∼ 32 processors) on an SGI.

Try recompiling and linking with the -64 switch (see the ABI man page). This will create a 64-bit executable. You will need to link in the 64-bit versions of the HDF libraries.

22.2.2.3 FLASH runs for a while, but all of a sudden it stops, without printing any errors to `stdout` – what's going on?

Most likely you've exceeded the maximum number of blocks that you have allocated on a processor – this is controlled by the `MAXBLOCKS` parameter. There may be an error message in the `amr_log` file generated by the PARAMESH library. Possible errors include:

```
ERROR memory overflow in restrict
        No. of blocks per processor exceeds maxblocks
```

There are two ways to fix this problem. Either increase the number of processors you are using or increase the number of blocks per processor. To increase the number of blocks per processor, you need to rerun setup with the -maxblocks flag. For example, to setup the Sedov problem with 1000 blocks per processor, use

```
./setup sedov -auto -maxblocks=1000
```

You want to ensure that the resulting executable fits into the available memory on a processor (leaving room for the operating system). You can check the size of the memory taken by the executable with the size command. Since flash uses very little dynamically allocated memory, this is a good measure of the memory requirements of FLASH. The default values of `MAXBLOCKS` are 500 for 2-d and 200 for 3-d.

22.2.2.4 I can compile FLASH with HDF5 fine, but when I run, I get an error—"error while loading shared libraries: libhdf5.so.0". How do I get HDF5 output to work?

You've installed the shared-object library for HDF5 and linked to it fine, because you specified the location of the library using -L on the link line. However, this library is not in your library path, so when the executable is run and tries to load it, it cannot find it. Try setting the LD_LIBRARY_PATH environment variable to the location of the library. For example, under tcsh

```
setenv LD_LIBRARY_PATH /opt/HDF5-1.4.2-patch1/lib/
```

22.2.2.5 FLASH segmentation faults on an IBM when running on multiple processors, what's up?

There is a bug in either FLASH (which we have not yet found) or in the AIX compilers that prevents runtime_parameters.F90 from working on multiple processors when optimized with -O3 or higher. If compiled with -O3, a segmentation fault will arise, because the pointer to the head of the integer linked list magically becomes null. The current work around is to compile runtime_parameters.F90 with -O2—everything else can handle -O3. Add a rule to your Makefile.h for runtime_parameters.o that uses less optimization (*i.e.* FFLAGS_TEST instead of FFLAGS_OPT).

22.2.2.6 When I run FLASH on an IBM machine using the AIX compilers, I get far more blocks than I do on other platforms – how do I fix this?

Don't use -qipa or -qhot with FLASH. It just doesn't work. The issue is somewhere in the PARAMESH library, but it has not been worked though yet. If you are insistent, you can try compiling all of amr_*.F90 without -qipa/-qhot and the rest with these flags, but we've never tested FLASH in this manner and do not support it.

22.3 Contacting the authors of FLASH

FLASH is still under active development, so many desirable features have not yet been included. Also, you will most likely encounter bugs in the distributed code. A mailing list has been established for reporting problems with the distributed FLASH code. To report a bug, send email to

`flash-bugs@flash.uchicago.edu`

giving your name and contact information and a description of the procedure you were following when you encountered the error. Information about the machine, operating system (`uname -a`), and compiler you are using is also extremely helpful. *Please do not send checkpoint files, as these can be very large.* If the problem can be reproduced with one of the standard test problems, send a copy of your runtime parameter file and your setup options. This situation is very desirable, as it limits the amount of back-and-forth communication needed for us to reproduce the problem. We cannot be responsible for problems that arise with a physics module or initial model you have developed yourself, but we will generally try to help with these if you can narrow down the problem to an easily reproducible effect that occurs in a short program run *and* if you are willing to supply your added source code.

We have also established a mailing list for FLASH users. This is a moderated mailing list and is intended both for FLASH users to communicate with each other about general usage issues (other than bugs) and for FLASH developers to announce the availability of new releases. The address for this mailing list is

`flash-users@flash.uchicago.edu`

Documentation (including this user's guide and a FAQ) and support information for FLASH can be obtained on the World Wide Web at

`http://flash.uchicago.edu/flashcode/` .

Chapter 23

References

Anninos, W. Y., & Norman, M. L. 1994, ApJ, 429, 434

Aparicio, J. M. 1998, ApJS, 117, 627

Bader, G. & Deuflhard, P. 1983, NuMat, 41, 373

Bardeen, J. M., Bond, J. R., Kaiser, N., & Szalay, A. S. 1986, ApJ, 304, 15

Berger, M. J. & Collela, P. 1989, JCP, 82, 64

Berger, M. J. & Oliger, J. 1984, JCP, 53, 484

Blinnikov, S. I., Dunina-Barkovskaya, N. V., & Nadyozhin, D. K. 1996, ApJS, 106, 171

Boris, J. P. & Book, D. L. 1973, JCP, 11, 38

Brackbill, J. & Barnes, D. C. 1980 JCP, 35, 426

Brandt, A. 1977, Math. Comp., 31, 333

Brio, M. & Wu, C. C. 1988 JCP, 75, 400

Brown, P. N., Byrne, G. D., & Hindmarsh, A. C. 1989, SIAM J. Sci. Stat. Comput., 10, 1038

Bunn, E. F. & White, M. 1997, ApJ, 480, 6

Burgers, J. M. 1969, Flow Equations for Composite Gases (New York: Academic)

Caughlan, G. R. & Fowler, W. A. 1988, Atomic Data and Nuclear Data Tables, 40, 283

Chandrasekhar, S. 1961, Hydrodynamic and Hydromagnetic Stability (Oxford: Clarendon)

Chandrasekhar, S. 1939, An Introduction to the Study of Stellar Structure (Toronto: Dover)

Chandrasekhar, S. 1987, Ellipsoidal Figures of Equilibrium (New York: Dover)

Chapman, S. & Cowling, T. G. 1970, The Mathematical Theory of Non-uniform Gases (Cambridge: CUP)

Christy, R. F. 1966, ApJ, 144, 108

Colella, P. & Glaz, H. M. 1985, JCP, 59, 264

Colella, P. & Woodward, P. 1984, JCP, 54, 174

Colgate, S. A., & White, R. H. 1966, ApJ, 143, 626

Decyk, V. K., Norton, C. D., & Szymanski, B. K. 1997, ACM Fortran Forum, 16 (also http://www.cs.rpi.edu/~szymansk/oof90.html)

DeZeeuw, D. & Powell, K. G. 1993, JCP, 104, 56

Duff, I. S., Erisman, A. M., & Reid, J. K. 1986, Direct Methods for Sparse Matrices (Oxford: Clarendon Press)

Emery, A. F. 1968, JCP, 2, 306

Eswaran, V. and Pope, S. B. 1988, Computers & Fluids, 16, 257–278

Evans, C. R. & Hawley, J. F. 1988, ApJ, 332, 659

Fletcher, C. A. J. 1991, Computational Techniques for Fluid Dynamics, 2d ed. (Berlin: Springer-Verlag)

Forester, C. K. 1977, JCP, 23, 1

Foster, P. N., & Chevalier, R. A. 1993, ApJ, 416, 303

Fryxell, B. A., Müller, E., & Arnett, D. 1989, in Numerical Methods in Astrophysics, ed. P. R. Woodward (New York: Academic)

Fryxell, B., Olson, K., Ricker, P., Timmes, F. X., Zingale, M., Lamb, D. Q., MacNeice, P., Rosner, R., Truran, J. W., & Tufo, H. 2000, ApJS, 131, 273

Godunov, S. K. 1959, Mat. Sbornik, 47, 271

Hu, W. & Sugiyama, N. 1996, ApJ, 471, 542

Huang, J., & Greengard, L. 2000, SIAM. J. Sci. Comput., 21, 1551

Iben, I. Jr. 1975, ApJ, 196, 525

Itoh, N., Hayashi, H., Nishikawa, A., & Kohyama, Y. 1996, ApJS, 102, 411

James, R. A. 1977, JCP, 25, 71

Jeans, J. H. 1902, Phil. Trans. Roy. Soc. (London), 199, 1

Khokhlov, A. M. 1997, Naval Research Lab memo 6406-97-7950

Kurganov, A., Noelle, S., & Petrova, G. 2001, SIAM. J. 23, 707

Kurganov, A., & Tadmor, E. 2000 JCP, 160, 241

Lacey, C. G. & Cole, S. 1993, MNRAS, 262, 627

Löhner, R. 1987, Comp. Meth. App. Mech. Eng., 61, 323

LeVeque, R. J. 1997, JCP, 131, 327

LeVeque, R. J. 1998, "Nonlinear Convservation Laws and Finite Volume Methods", R. J. LeVeque, D. Mihalas, E. A. Dorfi, E. Mueller (ed.), Computational Methods for Astrophysical Fluid Flow, Springer.

Linde, T., & Malagoli, A. 2001, JCP, submitted

MacNeice, P., Olson, K. M., Mobarry, C., de Fainchtein, R., & Packer, C. 1999, CPC, accepted

Marder, B. 1987, JCP, 68, 48

Martin, D., & Cartwright, K. 1996, "Solving Poisson's Equation using Adaptive Mesh Refinement." (`http://seesar.lbl.gov/anag/staff/martin/AMRPoisson.html`)

Mönchmeyer, R., & Müller, E. 1989, A&A, 217, 351

Munz, C. D., Omnes, P., Schneider, R., Sonnendrücker, & Voß, U. 2000, JCP, 161, 484

Nadyozhin, D. K. 1974, Nauchnye informatsii Astron, Sov. USSR, 32, 33

Orszag, S. A., Tang, C.-M., 1979, JFM, 90, 129

Parashar M., 1999, private communication (`http://www.caip.rutgers.edu/~parashar/DAGH`)

Plewa, T. & Müller, E. 1999, A&A, 342, 179.

Potekhin, A. Y., Chabrier, G., & Yakovlev, D. G. 1997, A&A323, 415

Powell, K. G., Roe, P. L., Linde, T. J. , Gombosi, T. I., & DeZeeuw, D. L. 1999, JCP, 154, 284

Press, W. H., Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P. 1992, Numerical Recipes in Fortran, 2d ed. (Cambridge: CUP)

Quirk, J. J., in Upwind and High-Resolution Schemes, ed. M. Yousuff Hussaini, Bram van Leer, and John Van Rosendale, 1997, (Berlin: Springer).

Ross, R., Nurmi, D., Cheng, A. & Zingale, M. 2001, Proceedings of SC2001.

Sedov, L. I. 1959, Similarity and Dimensional Methods in Mechanics (New York: Academic)

Shu, C.-W. 1998, in Advanced Numerical Approximation of Nonlinear Hyperbolic Equations, ed. A. Quarteroni, Lecture Notes in Mathematics, 1697, (Springer: Berlin)

Shu, C.-W. & Osher, S. 1989, JCP, 83, 32

Spitzer, L. 1962, Physics of Fully Ionized Gases. (New York: Wiley)

Sportisse, B. 2000, JCP, 161, 140

Sod, G. 1978, JCP, 27, 1

Strang, G. 1968, SIAM J. Numer. Anal., 5, 506

Timmes, F. X. 1992, ApJ, 390, L107

Timmes, F. X. 1999, ApJS, 124, 241

Timmes, F. X. 2000, ApJ, 528, 913

Timmes, F. X. & Arnett, D. 2000, ApJS, 125, 294

Timmes, F. X. & Brown, E. 2002 (in preparation)

Timmes, F. X. & Swesty, F. D. 2000, ApJS, 126, 501

Toro, E. F. 1997, Riemann Solvers and Numerical Methods for Fluid Dynamics. (New York: Springer-Verlag)

Tóth, G. 2000, JCP, 161, 605

Trottenberg, U., Oosterlee, C., & Schüller, A. 2001, Multigrid (San Diego: Academic Press)

van Leer, B. 1979, JCP, 32, 101

Wallace, R. K., Woosley, S. E., & Weaver, T. A. 1982, ApJ, 258, 696

Warren, M. S. & Salmon, J. K. 1993, in Proc. Supercomputing 1993 (Washington, DC: IEEE Computer Soc.)

Weaver, T. A., Zimmerman, G. B., & Woosley, S. E. 1978, ApJ, 225, 1021

Williams, F. A. 1988, Combustion Theory (Menlo Park: Benjamin-Cummings)

Williamson, J. H. 1980, JCP, 35, 48

Woodward, P. & Colella, P. 1984, JCP, 54, 115

Yakovlev, D. G., & Urpin, V. A. (YU) 1980 24, 303

Yee, H. C., Vinokur, M., and Djomehri, M. J. 2000, JCP, 162, 33

Zalesak, S. T. 1987, in Advances in Computer Methods for Partial Differential Equations VI, eds. Vichnevetsky, R. and Stepleman, R. S. (IMACS), 15

Zel'dovich, Ya. B. 1970, A&A, 5, 84

Zingale, M., Dursi, L. J., ZuHone, J., Calder, A. C., Fryxell, B., Plewa, T., Truran, J. W., Caceres, A., Olson, K., Ricker, P. M., Riley, K., Rosner, R., Siegel, A., Timmes, F. X., and Vladimirova, N. 2002, ApJS, 143, 539

Index