Capabilities and Applications

Sean Couch
FLASH Capabilities Span a Broad Range...

- Cellular detonation
- Helium burning on neutron stars
- Gravitational collapse/Jeans instability
- Gravitationally confined detonation
- Laser-driven shock instabilities
- Rayleigh-Taylor instability
- Orzag/Tang MHD vortex
- Richtmyer-Meshkov instability
- Wave breaking on white dwarfs
- Nova outbursts on white dwarfs

Shortly: Relativistic accretion onto NS

Intracluster interaction

Turbulent Nuclear Burning

The ASC/Alliances Center for Astrophysical Thermonuclear Flashes
The University of Chicago

Tuesday, September 28, 2010
Capabilities

- Infrastructure
  - Configuration (setup)
  - Mesh Management
  - Parallel I/O
  - Monitoring
    - Performance and progress
  - Verification
    - FlashTest
      - Unit and regression testing

- Physics
  - Hydrodynamics, MHD, RHD
  - Equation of State
  - Nuclear Physics and other Source Terms
  - Gravity
  - Particles, active and passive
  - Material Properties
  - Cosmology
The Hydro unit directory tree.

The ASC/Alliances Center for Astrophysical Thermonuclear Flashes
The University of Chicago

Figure 13.1: The Hydro unit directory tree.
14.1 Introduction

The \texttt{Eos} unit implements the equation of state needed by the hydrodynamics and nuclear burning solvers. The function \texttt{Eos} provides the interface for operating on a one-dimensional vector. The same interface can be used for a single cell by reducing the vector size to 1. Additionally, this function can be used to find the thermodynamic quantities either from the density, temperature, and composition or from the density, internal energy, and composition. For user's convenience, a wrapper function (\texttt{Eos\_wrapped}) is provided, which takes a section of a block and translates it into the data format required by the \texttt{Eos} function, then calls the function. Upon return from the \texttt{Eos} function, the wrapper translates the returned data back to the same section of the block.

Figure 14.1: The \texttt{Eos} directory tree.
Figure 15.1: The organizational structure of physics source terms, which include units such as Burn and Stir. Shaded units include only stub implementations.
16.1 Introduction

The Gravity unit supplied with FLASH3 computes gravitational source terms for the code. These source terms can take the form of the gravitational potential $\phi_s x t$ or the gravitational acceleration $g_s x t = -\nabla \phi_s x t$.

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

$$\nabla^2 \phi_s x t = 4\pi G \rho_s x t.$$
Physics Capabilities

Figure 17.1: The Particles unit main subunit.

Figure 17.2: The Particles unit with ParticlesInitialization and ParticlesMapping subunits.
The Cosmology unit solves the Friedmann equation for the scale factor in an expanding universe and 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.

18.1 Algorithms and Equations

The Cosmology unit makes several assumptions about the interpretation of physical quantities that enable any hydrodynamics or materials units written for a non-expanding universe to work unmodified in a cosmological context. All calculations are assumed to take place in comoving coordinates $x = r/a(t)$ where $r$ is a proper position vector and $a(t)$ is the time-dependent cosmological scale factor. The present epoch is defined to correspond to $a(t) = 1$; in the following discussion we use $t = t_0$ to refer to the age of the Universe at the present epoch. The gas velocity $v$ is taken to be the comoving peculiar velocity $\dot{x}$.
FLASH3 Transition

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

Figure 19.1: The materialProperties directory tree.
Breakdown of FLASH code research areas for primary research tool users
The Simulation Unit

Figure 22.1: The Simulation unit directory tree. Only some of the provided simulation implementations are shown. Users are expected to add their own simulations to the tree.
The Simulation Unit

- Typical Unit, obeys architecture, naming conventions, inheritance, etc. rules.
- Special Unit in that it always “wins” inheritance and parameter wars.
- FLASH problems is defined by directories in FLASH3/source/Simulation/SimulationMain.
- The Simulation directory gives people working on a particular problem a place to put problem specific code that replaces the default functionality in the main body of the code.
- It’s also a place to tell the setup script which units this problem will need from the rest of the code.
What’s in the Simulation Directory?

- Normal UnitMain implementation requirements
  - Simulation_data, Simulation_init, (Simulation_finalize), Simulation_initBlock
  - Makefile (with usually Simulation_data only)
  - Config file
  - Possibly other API functions: e.g. Simulation_initSpecies

- Specific to simulations:
  - Parameter files flash.par, testUG.par, etc.
  - Replacements for routines located elsewhere in directory tree
  - Routines that implement local functions e.g. sim-derivedVariables.F90
Required Code for a New Simulation

There are certain pieces of code that all simulations must implement:

- Simulation_data.F90: Fortran module which stores data and parameters specific to the Simulation.
- Simulation_init.F90: Reads the runtime parameters, and performs other necessary unit initializations.
- Simulation_initBlock.F90: Sets initial conditions in a single block.

Optionally, a simulation could implement:

- Simulation_initSpecies.F90: To give the properties of the species involved in a multispecies simulation
Customized Code for a new Simulation

- In a FLASH simulation directory, you can place code that overrides the functionality you would pick up from other code units.

- In the custom code you can modify:
  - Boundary conditions (Grid_applyBCEdge.F90)
  - Refinement criterion (Grid_markRefineDerefine.F90)
  - Diagnostic integrated quantities for output (in the flash.dat file), e.g., total mass (a default) or vorticity (IO_writeIntegralQuantities.F90)
  - Diagnostics to compute new grid scope variables (Grid_computeUserVars.F90)

- In general, this is a place to hack the code in ways specific to your problem, and you can hack basically anything.
Creating New Problems

- A new FLASH problem is created by making a directory for it in FLASH3/source/Simulation/SimulationMain. This is where the setup script looks for the problem specific files.

- The source files in a simulation directory that a user will need to modify are:
  - Simulation_data.F90: Fortran module which stores data and parameters specific to the Simulation.
  - Simulation_init.F90: Fortran routine which reads the runtime parameters, and performs other necessary initializations.
  - Simulation_initBlock.F90: Fortran routine for setting initial conditions in a single block.
  - Simulation_initSpecies.F90: Optional Fortran routine for initializing species properties if multiple species are being used.

- Custom implementation of any kernel routine in FLASH can be placed here.
Simulation_data

- A Fortran module containing all data specific to the simulation unit.
- All names should be prefixed with sim_ to make it clear that data belongs to the simulation unit.
- Remember to use the save attribute to prevent data going out of scope.

```fortran
module Simulation_data
    implicit none
    real, save :: sim_pAmbient, sim_xAngle, sim_yAngle, sim_zAngle
end module Simulation_data
```
Simulation_init

- Initializes the simulation unit.
  - Called once at the beginning of the simulation in both new and restarted application runs.
  - Eliminates the need for FLASH2 "if (firstcall)" code fragments.

- Example usage:
  - Stores runtime parameter values in Simulation_data private variables.
  - Calculates any runtime parameter derived quantities.
  - Reads a lookup table from a file.
The Config file and Simulation_init

Config file declares the runtime parameters.

Simulation_init extracts the value of runtime parameters.

The runtime parameter's default value can be overridden in a flash.par.
Simulation_initBlock

- Applies initial conditions to the physical domain
  - Initializes Grid data one block at a time.
  - Only called in new application runs (not in restarts).

- Block abstraction allows it to be used with different Grid implementations
  - Called once in UG simulations.
  - Called many times in AMR simulations.

- Generating an initial grid in AMR simulations:
  - Simulation_initBlock is applied to all blocks at the base refinement level.
  - Grid unit refines blocks if refinement criteria met.
  - Simulation_initBlock is re-applied to all blocks.

Repeats
Simulation_initBlock: Finding cell types

- The Grid API contains a portable way to find the internal cells and guard cells in a particular block.
  - Essential for NFBS Uniform grid mode where block sizes are not always the same size.

\[
\text{Grid_getBlkIndexLimits(blockId, blkLimits, blkLimitsGC, optional: gridDataStruct)}
\]

- The arrays \textit{blkLimits} and \textit{blkLimitsGC} contain the lower and upper bounds of a block. For cell-centered PARAMESH data:
  \[
  \text{blkLimits(LOW, IAXIS)} = \text{NGUARD} + 1; \text{blkLimits(HIGH, IAXIS)} = \text{NXB} + \text{NGUARD} \\
  \text{blkLimitsGC(LOW, IAXIS)} = 1; \text{blkLimitsGC(HIGH, IAXIS)} = \text{NXB} + 2 \times \text{NGUARD}
  \]

- The input argument \textit{gridDataStruct} specifies the underlying grid datastructure, e.g. cell-centered, face-centered, scratch data structure.
Simulation_initBlock: Accessing each cell

- Many Grid API functions available to read / write Grid data:
  - Grid_getPointData, Grid_putPointData
  - Grid_getRowData, Grid_putRowData
  - Most general is Grid_getBlkPtr:

  Grid_getBlkPtr(blockID, dataPtr, optional: gridDataStruct)

- Sets the pointer dataPtr to the block indicated by blockID for the data structure gridDataStruct. Free the pointer using Grid_releaseBlkPtr (has same arguments as Grid_getBlkPtr).

- To obtain actual cells coordinates use Grid_getCellCoords:

  Grid_getCellCoords(axis, blockID, edge, guardcell, coordinates, size)

- This stores coordinates for the cells on axis axis (IAXIS, JAXIS, KAXIS) at cell location edge (LEFT_EDGE, RIGHT_EDGE, CENTER) in the array coordinates(size).
Excerpt from a Simulation_initBlock

```fortran
subroutine Simulation_initBlock(blockID, myPE)

  call Grid_getBlkIndexLimits(blockID,blkLimits,blkLimitsGC)
  sizeX = blkLimitsGC(HIGH,IAXIS) - blkLimitsGC(LOW,IAXIS) + 1 !Num cells inc. guard.
  allocate(xCoord(sizeX))
  call Grid_getCellCoords(IAXIS, blockID, CENTER, .true., xCoord, sizeX)

  call Grid_getBlkPtr(blockId,solnData)
  !Loop over each internal cell and initialize data

  do  i = blkLimits(LOW,IAXIS), blkLimits(HIGH,IAXIS)
      If (xCoord(i) > sim_xpos) solnData(DENS_VAR,i,j,k) = …
  end do
  call Grid_releaseBlkPtr(blockID,solnData)

end subroutine Simulation_initBlock
```

The ASC/Alliances Center for Astrophysical Thermonuclear Flashes
The University of Chicago
Simulation_initSpecies

- Implementation only required when working with multiple species.
  - Called from Multispecies_init to initialize fluid properties.
  - Called in new and restarted application runs.
  - Called before Simulation_init.

- General purpose Simulation_initSpecies implementations are available for nuclear networks and ionization (See Simulation/SimulationComposition directory).

- May want to create derived quantities in Simulation_init from the fluids initialized in Simulation_initSpecies.
The Config file and Simulation_initSpecies

Config file declares the species.

SPECIES FLD1
SPECIES FLD2

subroutine Simulation_initSpecies()
    use Multispecies_interface, ONLY : Multispecies_setProperty
    implicit none
    include "Flash.h"
    include "Multispecies.h"
    call Multispecies_setProperty(FLD1_SPEC, A, 1.)
    call Multispecies_setProperty(FLD1_SPEC, Z, 1.)
    call Multispecies_setProperty(FLD1_SPEC, GAMMA, 1.66666666667e0)
    call Multispecies_setProperty(FLD2_SPEC, A, 4.0)
    call Multispecies_setProperty(FLD2_SPEC, Z, 2.0)
    call Multispecies_setProperty(FLD2_SPEC, GAMMA, 2.0)
end subroutine Simulation_initSpecies
Working with block lists

- A single processor contains some portion of the total grid data in one or more blocks.
  - Possible to access data in a grid-package specific way.
  - However, we recommend using Grid API functions so that code is independent of a particular grid-package.

Grid_getListOfBlocks(blockType, listofBlocks, count, optional: refinementLevel)

- Returns the actual block IDs in listofBlocks and the number of block IDs in count. The returned block IDs must satisfy the criteria set by blockType and refinementLevel input arguments.

- NOTE: Any code using this function must “use” the function prototype because this function has an optional argument.