Enzo Documentation

Release 2.1

Enzo Developers

November 23, 2012

CONTENTS

This is the development site for Enzo, an adaptive mesh refinement (AMR), grid-based hybrid code (hydro + N-Body) which is designed to do simulations of cosmological structure formation. Links to documentation and downloads for all versions of Enzo from 1.0 on are available.

Enzo development is supported by grants AST-0808184 and OCI-0832662 from the National Science Foundation.

CHAPTER

ONE

ENZO PUBLIC LICENSE

University of Illinois/NCSA Open Source License

Developed by:

- Laboratory for Computational Astrophysics
- National Center for Supercomputing Applications
- University of Illinois in Urbana-Champaign

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal with the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimers.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimers in the documentation and/or other materials provided with the distribution.
3. Neither the names of The Laboratory for Computational Astrophysics, The National Center for Supercomputing Applications, The University of Illinois in Urbana-Champaign, nor the names of its contributors may be used to endorse or promote products derived from this Software without specific prior written permission.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE CONTRIBUTORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS WITH THE SOFTWARE.

University of California/BSD License

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the Laboratory for Computational Astrophysics, the University of California, nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

CHAPTER

TWO

GETTING STARTED WITH ENZO

2.1 Obtaining and Building Enzo

2.1.1 Enzo Compilation Requirements

Enzo can be compiled on any POSIX-compatible operating system, such as Linux, BSD (including Mac OS X), and AIX. In addition to a C/C++ and Fortran-90 compiler, the following libraries are necessary:

- HDF5, the hierarchical data format. Note that HDF5 also may require the szip and zlib libraries, which can be found at the HDF5 website. Note that compiling with HDF5 1.8 or greater requires that the compiler directive `H5_USE_16_API` be specified; typically this is done with `-DH5_USE_16_API` and it's set in most of the provided makefiles.
- MPI, for multi-processor parallel jobs. Note that Enzo will compile without MPI, but it's fine to compile with MPI and only run oon a single processor.

2.1.2 Mercurial Check Out Instructions

Enzo is provided in both a stable and an unstable form. **It is highly recommended that for any production run the stable version is used.** Additionally, we encourage anyone who uses Enzo to sign up for the Enzo Users' List. A source browser is also available.

Please visit the Google Code project website to access the Enzo source tree and read the latest source checkout instructions.

http://enzo.googlecode.com/

Updating a source tree with Mercurial is beyond the scope of this document; for more information, please peruse *Developer's Guide* and the Mercurial documentation. The mercurial commands of most use are `pull`, `update` and `incoming`.

2.1.3 Building Enzo

This is a quick, line by line example of checking out and building Enzo using current build system. A comprehensive list of the make system arguments can be found in *The Enzo Makefile System*.

This assumes that we're working from a checkout from the Enzo project page, located at http://enzo.googlecode.com/ . Checkout instructions can be found there, and for more detailed information about the structure of the Enzo source control repository, see *Introduction to Enzo Modification*.

Initializing the Build System

This just clears any existing configurations left over from a previous machine, and creates a couple of files for building.

```
~ $ cd enzo/
~/enzo $ ./configure
```

This should output a brief message saying that the build system has been initialized. To confirm that it ran, there should be a file called Make.config.machine in the src/enzo subdirectory.

Go to the Source Directory

The source code for the various Enzo components are laid out in the src/ directory.

```
~/enzo/src $ cd src/
~/enzo/src $ ls
Makefile      P-GroupFinder anyl          enzo          enzohop       inits
lcaperf       mpgrafic      ring
~/enzo/src $
```

Right now, we're just building the main executable (the one that does the simulations), so we need the `enzo/` directory.

```
~/enzo/src $ cd enzo/
```

Find the Right Machine File

We've chosen to go with configurations files based on specific machines. This means we can provide configurations files for most of the major NSF resources, and examples for many of the one-off (clusters, laptops, etc.).

These machine-specific configuration files are named `Make.mach.machinename`.

```
~/enzo/src/enzo $ ls Make.mach.*
Make.mach.darwin          Make.mach.nasa-discover   Make.mach.ncsa-cobalt
Make.mach.ornl-jaguar-pgi Make.mach.tacc-ranger     Make.mach.unknown
Make.mach.kolob           Make.mach.nasa-pleiades   Make.mach.nics-kraken
Make.mach.scinet          Make.mach.triton
Make.mach.linux-gnu       Make.mach.ncsa-abe        Make.mach.orange
Make.mach.sunnyvale       Make.mach.triton-intel
~/enzo/src/enzo $
```

In this example, we choose `Make.mach.darwin`, which is appropriate for Mac OS X machines.

Porting

If there's no machine file for the machine you're on, you will have to do a small amount of porting. However, we have attempted to provide a wide base of Makefiles, so you should be able to find one that is close, if not identical, to the machine you are attempting to run Enzo on. The basic steps are as follows:

1. Find a Make.mach file from a similar platform.
2. Copy it to Make.mach.site-machinename (site = sdsc or owner, machinename = hostname).
3. Edit the machine-specific settings (compilers, libraries, etc.).
4. Build and test.

If you expect that you will have multiple checkouts of the Enzo source code, you should feel free to create the directory $HOME/.enzo/ and place your custom makefiles there, and Enzo's build system will use any machine name-matching Makefile in that directory to provide or override Make settings.

Make sure you save your configuration file! If you're on a big system (multiple Enzo users), please post your file to the Enzo mailing list, and it will be considered for inclusion with the base Enzo distribution.

HDF5 Versions

If your system uses a version of HDF5 greater than or equal to 1.8, you probably need to add a flag to your compile settings, unless your HDF5 library was compiled using –with-default-api-version=v16. The simplest thing to do is to find the line in your Make.mach file that sets up MACH_DEFINES, which may look like this

```
MACH_DEFINES   = -DLINUX # Defines for the architecture; e.g. -DSUN, -DLINUX, etc.
```

and change it to

```
MACH_DEFINES   = -DLINUX -DH5_USE_16_API # Defines for the architecture; e.g. -DSUN, -DLINUX, etc.
```

This will ensure that the HDF5 header files expose the correct API for Enzo.

Build the Makefile

Now that you have your configuration file, tell the build system to use it:

```
~/enzo/src/enzo $ make machine-darwin

 *** Execute 'gmake clean' before rebuilding executables ***

   MACHINE: Darwin (OSX Leopard)

~/enzo/src/enzo $
```

You may also to know the settings (precision, etc.) that's being use. You can find this out using `make show-config`. For a detailed explanation of what these mean, see *The Enzo Makefile System*.

```
~/enzo/src/enzo $ make show-config

MACHINE: Darwin (OSX Leopard)
MACHINE-NAME: darwin

PARAMETER_MAX_SUBGRIDS:       100000
PARAMETER_MAX_BARYONS:        20
PARAMETER_MAX_TASKS_PER_NODE: 8
PARAMETER_MEMORY_POOL_SIZE:   100000

CONFIG_PRECISION:             64
CONFIG_PARTICLES:             64
CONFIG_INTEGERS:              64
CONFIG_PARTICLE_IDS:          64
CONFIG_INITS:                 64
CONFIG_IO:                    32
CONFIG_USE_MPI:               yes
CONFIG_OBJECT_MODE:           64
CONFIG_TASKMAP:               no
CONFIG_PACKED_AMR:            yes
CONFIG_PACKED_MEM:            no
```

```
CONFIG_LCAPERF:               no
CONFIG_PAPI:                  no
CONFIG_PYTHON:                no
CONFIG_ECUDA:                 no
CONFIG_OOC_BOUNDARY:          no
CONFIG_OPT:                   debug
CONFIG_TESTING:               no
CONFIG_TPVEL:                 no
CONFIG_PHOTON:                yes
CONFIG_HYPRE:                 no
CONFIG_EMISSIVITY:            no
CONFIG_USE_HDF4:              no
CONFIG_NEW_GRID_IO:           yes
CONFIG_BITWISE_IDENTICALITY:  yes
CONFIG_FAST_SIB:              yes
CONFIG_FLUX_FIX:              yes

~/enzo/src/enzo $
```

Build Enzo

The default build target is the main executable, Enzo.

```
~/enzo/src/enzo $ make
Updating DEPEND
pdating DEPEND
Compiling enzo.C
Compiling acml_st1.src
... (skipping) ...
Compiling Zeus_zTransport.C
Linking
Success!
~/enzo/src/enzo $
```

After compiling, you will have `enzo.exe` in the current directory.

Building other Tools

Building other tools is typically very straightforward; they rely on the same Makefiles, and so should require no porting or modifications to configuration.

Inits

```
~/enzo/src/ring $ cd ../inits/
~/enzo/src/inits $ make
Compiling enzo_module.src90
Updating DEPEND
Compiling acml_st1.src
...
Compiling XChunk_WriteIntField.C
Linking
Success!
```

This will produce `inits.exe`.

Ring

```
~/enzo/src/enzo $ cd ../ring/
~/enzo/src/ring $ make
Updating DEPEND
Compiling Ring_Decomp.C
Compiling Enzo_Dims_create.C
Compiling Mpich_V1_Dims_create.c
Linking
Success!
```

This will produce `ring.exe`.

YT

To install yt, you can use the installation script provided with the yt source distribution. See the yt homepage for more information.

2.2 How to run an Enzo test problem

Enzo comes with a set of pre-written parameter files which are used to test Enzo. This is useful when migrating to a new machine with different compilers, or when new versions of compilers and libraries are introduced. Also, all the test problems should run to completion, which is generally not a guarantee!

At the top of each Enzo parameter file is a line like `ProblemType = 23`, which tells Enzo the type of problem. You can see how this affects Enzo by inspecting `InitializeNew.C`. In this example, this gets called:

```
if (ProblemType == 23)
  ret = TestGravityInitialize(fptr, Outfptr, TopGrid, MetaData);
```

which then calls the routine in `TestGravityInitialize.C`, and so on. By inspecting the initializing routine for each kind of problem, you can see what and how things are being included in the simulation.

The test problem parameter files are inside doc/examples. Please see *Enzo Test Suite* for a full list of test problems. The files that end in .enzo are the Enzo parameter files, and .inits are inits parameter files. inits files are only used for cosmology simulations, and you can see an example of how to run that in *How to run a cosmology simulation*. Let's try a couple of the non-cosmology test problems.

2.2.1 ShockPool3D test

The ShockPool3D is a purely hydrodynamical simulation testing a shock with non-periodic boundary conditions. Once you've built enzo (*Obtaining and Building Enzo*), make a directory to run the test problem in. Copy enzo.exe and ShockPool3D.enzo into that directory. This example test will be run using an interactive session. On Kraken, to run in an interactive queue, type:

```
qsub -I -V -q debug -lwalltime=2:00:00,size=12
```

12 cores (one node) is requested for two hours. Of course, this procedure may differ on your machine. Once you're in the interactive session, inside your test run directory, enter:

```
aprun -n 12 ./enzo.exe -d ShockPool3D.enzo > 01.out
```

The test problem is run on 12 processors, the debug flag (-d) is on, and the standard output is piped to a file (01.out). This took about an hour and twenty minutes to run on Kraken. When it's finished, you should see `Successful run, exiting.` printed to stderr. Note that if you use other supercomputers, `aprun` may be replaced by 'mpirun', or possibly another command. Consult your computer's documentation for the exact command needed.

If you want to keep track of the progress of the run, in another terminal type:

```
tail -f 01.out
tail -f 01.out | grep dt
```

The first command above gives too verbose output to keep track of the progress. The second one will show what's more interesting, like the current cycle number and how deep in the AMR hierarchy the run is going (look for Level[n] where n is the zero-based AMR level number). This command is especially useful for batch queue jobs where the standard out always goes to a file.

2.2.2 GravityTest test

The GravityTest.enzo problem only tests setting up the gravity field of 5000 particles. A successful run looks like this and should take less than a second, even on one processor:

```
test2> aprun -n 1 ./enzo.exe GravityTest.enzo > 01.out
****** GetUnits:  1.000000e+00 1.000000e+00 1.000000e+00 1.000000e+00 *******
CWD test2
Global Dir set to test2
Successfully read in parameter file GravityTest.enzo.
INITIALIZATION TIME =   6.04104996e-03
Successful run, exiting.
```

2.2.3 Other Tests & Notes

All the outputs of the tests have been linked to on this page, below. Some of the tests were run using only one processor, and others that take more time were run using 16. All tests were run with the debug flag turned on (which makes the output log, 01.out more detailed). Enzo was compiled in debug mode without any optimization turned on (gmake opt-debug). The tests that produce large data files have only the final data output saved. If you wish to do analysis on these datasets, you will have to change the values of GlobalDir, BoundaryConditionName, BaryonFileName and ParticleFileName in the restart, boundary and hierarchy files to match where you've saved the data.

PressurelessCollapse

The PressurelessCollapse test required isolated boundary conditions, so you need to compile Enzo with that turned on (gmake isolated-bcs-yes). You will also need to turn off the top grid bookkeeping (gmake unigrid-transpose-no).

Input Files

A few of the test require some input files to be in the run directory. They are kept in input:

```
> ls input/
ATOMIC.DAT  cool_rates.in  lookup_metal0.3.data
```

You can either copy the files into your run directory as a matter of habit, or copy them only if they're needed.

2.2.4 Outputs

- AMRCollapseTest.tar.gz - 24 MB
- AMRShockPool2D.tar.gz - 35 KB
- AMRShockTube.tar.gz - 23 KB
- AMRZeldovichPancake.tar.gz - 72 KB
- AdiabaticExpansion.tar.gz - 31 KB
- CollapseTest.tar.gz - 5.4 MB
- CollideTest.tar.gz - 7.6 MB
- DoubleMachReflection.tar.gz - 2.1 MB
- ExtremeAdvectionTest.tar.gz - 430 KB
- GravityStripTest.tar.gz - 12 MB
- GravityTest.tar.gz - 99 KB
- GravityTestSphere.tar.gz - 4.6 MB
- Implosion.tar.gz - 5.6 MB
- ImplosionAMR.tar.gz - 3.5 MB

2.3 How to run a cosmology simulation

In order to run a cosmology simulation, you'll need to build enzo.exe, inits.exe and ring.exe (see *Obtaining and Building Enzo*) inits creates the initial conditions for your simulation, and ring splits up the root grid which is necessary if you're using parallel IO. Once you have built the three executables, put them in a common directory where you will run your test simulation. You will also save the inits and param files (shown and discussed below) in this directory.

2.3.1 Creating initial conditions

The first step in preparing the simulation is to create the initial conditions. The file inits uses is a text file which contains a list of parameters with their associated values. These values tell the initial conditions generator necessary information like the simulation box size, the cosmological parameters and the size of the root grid. The code then takes that information and creates a set of initial conditions. Here is an example inits file:

```
#
#  Generates initial grid and particle fields for a
#    CDM simulation
#
#  Cosmology Parameters
#
CosmologyOmegaBaryonNow      = 0.044
CosmologyOmegaMatterNow      = 0.27
CosmologyOmegaLambdaNow      = 0.73
CosmologyComovingBoxSize     = 10.0    // in Mpc/h
CosmologyHubbleConstantNow   = 0.71      // in units of 100 km/s/Mpc
CosmologyInitialRedshift     = 60
#
#  Power spectrum Parameters
#
```

```
PowerSpectrumType            = 11
PowerSpectrumSigma8          = 0.9
PowerSpectrumPrimordialIndex = 1.0
PowerSpectrumRandomSeed      = -584783758
#
#  Grid info
#
Rank                = 3
GridDims            = 32 32 32
InitializeGrids     = 1
GridRefinement      = 1
#
#  Particle info
#
ParticleDims        = 32 32 32
InitializeParticles = 1
ParticleRefinement  = 1
#
#  Overall field parameters
#
#
#  Names
#
ParticlePositionName = ParticlePositions
ParticleVelocityName = ParticleVelocities
GridDensityName      = GridDensity
GridVelocityName     = GridVelocities
```

inits is run by typing this command:

```
./inits.exe -d Example_Cosmology_Sim.inits
```

inits will produce some output to the screen to tell you what it is doing, and will write five files: `GridDensity`, `GridVelocities`, `ParticlePositions`, `ParticleVelocities` and `PowerSpectrum.out`. The first four files contain information on initial conditions for the baryon and dark matter componenets of the simulation, and are HDF5 files. The last file is an ascii file which contains information on the power spectrum used to generate the initial conditions.

It is also possible to run cosmology simulations using initial nested subgrids.

2.3.2 Parallel IO - the ring tool

This simulation is quite small. The root grid is only 32 cells on a side and we allow a maximum of three levels of mesh refinement. Still, we will use the ring tool, since it is important for larger simulations of sizes typically used for doing science. Additionally, if you wish to run with 64 or more processors, you should use `ParallelRootGridIO`, described in *Parallel Root Grid IO*.

The ring tool is part of the Enzo parallel IO (input-output) scheme. Examine the last section of the parameter file (see below) for this example simulation and you will see:

```
#
# IO parameters
#
ParallelRootGridIO = 1
ParallelParticleIO = 1
```

These two parameters turn on parallel IO for both grids and particles. In a serial IO simulation where multiple

processors are being used, the master processor reads in all of the grid and particle initial condition information and parcels out portions of the data to the other processors. Similarly, all simulation output goes through the master processor as well. This is fine for relatively small simulations using only a few processors, but slows down the code considerably when a huge simulation is being run on hundreds of processors. Turning on the parallel IO options allows each processor to perform its own IO, which greatly decreases the amount of time the code spends performing IO.

The process for parallelizing grid and particle information is quite different. Since it is known exactly where every grid cell in a structured Eulerian grid is in space, and these cells are stored in a regular and predictable order in the initial conditions files, turning on `ParallelRootGridIO` simply tells each processor to figure out which portions of the arrays in the GridDensity and `GridVelocities` belong to it, and then read in only that part of the file. The particle files (`ParticlePositions` and `ParticleVelocities`) store the particle information in no particular order. In order to efficiently parallelize the particle IO the ring tool is used. ring is run on the same number of processors as the simulation that you intend to run, and is typically run just before Enzo is called for this reason. In ring, each processor reads in an equal fraction of the particle position and velocity information into a list, flags the particles that belong in its simulation spatial domain, and then passes its portion of the total list on to another processor. After each portion of the list has made its way to every processor, each processor then collects all of the particle and velocity information that belongs to it and writes them out into files called `PPos.nnnn` and `PVel.nnnn`, where nnnn is the processor number. Turning on the `ParallelParticleIO` flag in the Enzo parameter file instructs Enzo to look for these files.

For the purpose of this example, you're going to run ring and Enzo on 4 processors (this is a fixed requirement). The number of processors used in an MPI job is set differently on each machine, so you'll have to figure out how that works for you. On some machines, you can request an 'interactive queue' to run small MPI jobs. On others, you may have to submit a job to the batch queue, and wait for it to run.

To start an interactive run, it might look something like this:

```
qsub -I -V -l walltime=00:30:00,size=4
```

This tells the queuing system that you want four processors total for a half hour of wall clock time. You may have to wait a bit until nodes become available, and then you will probably start out back in your home directory. You then run ring on the particle files by typing something like this:

```
mpirun -n 4 ./ring.exe pv ParticlePositions ParticleVelocities
```

This will then produce some output to your screen, and will generate 8 files: `PPos.0000` through `PPos.0003` and `PVel.0000` through `PVel.0003`. Note that the 'mpirun' command may actually be 'aprun' or something similar. Consult your supercomputer's documentation to figure out what this command should really be.

Congratulations, you're now ready to run your cosmology simulation!

2.3.3 Running an Enzo cosmology simulation

After all of this preparation, running the simulation itself should be straightforward. First, you need to have an Enzo parameter file. Here is an example compatible with the inits file above:

```
#
# AMR PROBLEM DEFINITION FILE: Cosmology Simulation (AMR version)
#
#  define problem
#
ProblemType                = 30      // cosmology simulation
TopGridRank                = 3
TopGridDimensions          = 32 32 32
SelfGravity                = 1       // gravity on
TopGridGravityBoundary     = 0       // Periodic BC for gravity
LeftFaceBoundaryCondition  = 3 3 3   // same for fluid
```

```
RightFaceBoundaryCondition = 3 3 3
#
#  problem parameters
#
CosmologySimulationOmegaBaryonNow       = 0.044
CosmologySimulationOmegaCDMNow      = 0.226
CosmologyOmegaMatterNow         = 0.27
CosmologyOmegaLambdaNow         = 0.73
CosmologySimulationDensityName          = GridDensity
CosmologySimulationVelocity1Name        = GridVelocities
CosmologySimulationVelocity2Name        = GridVelocities
CosmologySimulationVelocity3Name        = GridVelocities
CosmologySimulationParticlePositionName = ParticlePositions
CosmologySimulationParticleVelocityName = ParticleVelocities
CosmologySimulationNumberOfInitialGrids = 1
#
#  define cosmology parameters
#
ComovingCoordinates        = 1       // Expansion ON
CosmologyHubbleConstantNow = 0.71    // in km/s/Mpc
CosmologyComovingBoxSize   = 10.0  // in Mpc/h
CosmologyMaxExpansionRate  = 0.015   // maximum allowed delta(a)/a
CosmologyInitialRedshift   = 60.0      //
CosmologyFinalRedshift     = 3.0     //
GravitationalConstant      = 1       // this must be true for cosmology
#
#  set I/O and stop/start parameters
#
CosmologyOutputRedshift[0] = 25.0
CosmologyOutputRedshift[1] = 10.0
CosmologyOutputRedshift[2] = 5.0
CosmologyOutputRedshift[3] = 3.0
#
#  set hydro parameters
#
Gamma                  = 1.6667
PPMDiffusionParameter  = 0       // diffusion off
DualEnergyFormalism    = 1       // use total & internal energy
InterpolationMethod    = 1     // SecondOrderA
CourantSafetyNumber    = 0.5
ParticleCourantSafetyNumber = 0.8
FluxCorrection         = 1
ConservativeInterpolation = 0
HydroMethod            = 0
#
#  set cooling parameters
#
RadiativeCooling       = 0
MultiSpecies           = 0
RadiationFieldType     = 0
StarParticleCreation   = 0
StarParticleFeedback   = 0
#
#  set grid refinement parameters
#
StaticHierarchy           = 0    // AMR turned on!
MaximumRefinementLevel    = 3
MaximumGravityRefinementLevel = 3
```

```
RefineBy                   = 2
CellFlaggingMethod         = 2 4
MinimumEfficiency          = 0.35
MinimumOverDensityForRefinement = 4.0 4.0
MinimumMassForRefinementLevelExponent = -0.1
MinimumEnergyRatioForRefinement = 0.4

#
#  set some global parameters
#
GreensFunctionMaxNumber   = 100   // # of greens function at any one time


#
# IO parameters
#

ParallelRootGridIO = 1
ParallelParticleIO = 1
```

Once you've saved this, you start Enzo by typing:

```
mpirun -n 4 ./enzo.exe -d Example_Cosmology_Sim.param >& output.log
```

The simulation will now run. The -d flag ensures a great deal of output, so you may redirect it into a log file called `output.log` for later examination. This particular simulation shouldn't take too long, so you can run this in the same 30 minute interactive job you started when you ran inits. When the simulation is done, Enzo will display the message "Successful run, exiting."

Congratulations! If you've made it this far, you have now successfully run a cosmology simulation using Enzo!

2.4 Sample inits and Enzo parameter files

This page contains a large number of example inits and Enzo parameter files that should cover any possible kind of Enzo cosmology simulation that you are interested in doing. All should run with minimal tinkering. They can be downloaded separately below, or as a single tarball.

Note: unless otherwise specified, inits is run by calling

```
inits -d <name of inits parameter file>
```

and Enzo is run by calling

```
[mpirun ...] enzo -d <name of enzo parameter file>
```

In both cases, the -d flag displays debugging information, and can be omitted. Leaving out the -d flag can significantly speed up Enzo calculations. Also note that Enzo is an MPI-parallel program, whereas inits is not.

Unigrid dark matter-only cosmology simulation. This is the simplest possible Enzo cosmology simulation - a dark matter-only calculation (so no baryons at all) and no adaptive mesh. See the inits parameter file and Enzo parameter file.

AMR dark matter-only cosmology simulation. This is a dark matter-only cosmology calculation (using the same initial conditions as the previous dm-only run) but with adaptive mesh refinement turned on. See the inits parameter file and Enzo parameter file.

Unigrid hydro+dark matter cosmology simulation (adiabatic). This is a dark matter plus hydro cosmology calculation **without** adaptive mesh refinement and no additional physics. See the inits parameter file and Enzo parameter

file.

AMR hydro+dark matter cosmology simulation (adiabatic). This is a dark matter plus hydro cosmology calculation (using the same initial conditions as the previous dm+hydro run)**with** adaptive mesh refinement (refining everywhere in the simulation volume) and no additional physics. See the inits parameter file and Enzo parameter file.

AMR hydro+dark matter cosmology simulation (lots of physics). This is a dark matter plus hydro cosmology calculation (using the same initial conditions as the previous two dm+hydro runs) **with** adaptive mesh refinement (refining everywhere in the simulation volume) and including radiative cooling, six species primordial chemistry, a uniform metagalactic radiation background, and prescriptions for star formation and feedback. See the inits parameter file and Enzo parameter file.

AMR hydro+dark matter nested-grid cosmology simulation (lots of physics). This is a dark matter plus hydro cosmology calculation with two static nested grids providing excellent spatial and dark matter mass resolution for a single Local Group-sized halo and its progenitors. This simulation only refines in a small subvolume of the calculation, and includes radiative cooling, six species primordial chemistry, a uniform metagalactic radiation background, and prescriptions for star formation and feedback. All parameter files can be downloaded in one single tarball. Note that inits works differently for multi-grid setups. Instead of calling inits one time, it is called N times, where N is the number of grids. For this example, where there are three grids total (one root grid and two nested subgrids), the procedure would be:

```
NohProblem2DAMR.tar.gz - 650 KB
NohProblem3D.tar.gz - 34 MB
NohProblem3DAMR.tar.gz - 126 MB
ProtostellarCollapse_Std.tar.gz - 826 KB
SedovBlast.tar.gz - 4.1 MB
SedovBlastAMR.tar.gz - 1.6 MB
ShockPool2D.tar.gz - 250 KB
ShockPool3D.tar.gz - 91 KB
ShockTube.tar.gz - 16 KB
StripTest.tar.gz - 4.1 MB
WavePool.tar.gz - 20 KB
ZeldovichPancake.tar.gz - 36 KB
```

2.5 Writing Enzo Parameter Files

Putting together a parameter file for Enzo is possibly the most critical step when setting up a simulation, and is certainly the step which is most fraught with peril. There are over 200 parameters that one can set - see *Enzo Parameter List* for a complete listing. For the most part, defaults are set to be sane values for cosmological simulations, and most physics packages are turned off by default, so that you have to explicitly turn on modules. All physics packages are compiled into Enzo (unlike codes such as ZEUS-MP 1.0, where you have to recompile the code in order to enable new physics).

It is inadvisable for a novice to put together a parameter file from scratch. Several parameter files are available for download at *Sample inits and Enzo parameter files*. The simulations include:

- dark matter-only unigrid and AMR simulations,
- dark matter + hydro unigrid and AMR simulations,
- an AMR dm + hydro simulation with multiple nested grids and a limited refinement region.

In order to make the most of this tutorial it is advisable to have one or more of these parameter files open while reading this page. For the purposes of this tutorial we assume that the user is putting together a cosmology simulation and has already generated the initial conditions files using inits.

All parameters are put into a plain text file (one parameter per line), the name of which is fed into Enzo at execution time at the command line. Typically, a parameter is set by writing the parameter name, an equals sign, and then the parameter value or values, like this:

```
NumberOfBufferZones = 3
```

You must leave at least one space between the parameter, the equals sign, and the parameter value. It's fine if you use more than one space - after the first space, whitespace is unimportant. All lines which start with a # (pound sign) are treated as comments and ignored. In addition, you can have inline comments by using the same pound sign, or two forward slashes // after the parameter line.

```
NumberOfBufferZones = 3 // More may be needed depending on physics used.
```

2.5.1 Initialization parameters

Complete descriptions of all initialization parameters are given here. The most fundamental initialization parameter you have to set is `ProblemType`, which specifies the type of problem to be run, and therefore the way that Enzo initiates the data. A cosmology simulation is problem type 30. As started before, for the purposes of this introduction I'm assuming that you are generating a cosmology simulation, so you would put this line in the parameter file:

```
ProblemType = 30
```

`TopGridRank` specifies the spatial dimensionality of your problem (1, 2 or 3 dimensions), and must be set. `TopGridDimensions` specifies the number of root grid cells along each axis. For a 3D simulation with 128 grid cells along each axis on the root grid, put this in the parameter file:

```
TopGridRank = 3
TopGridDimensions = 128 128 128
```

Additionally, you must specify the names of the initial conditions files with contain the baryon density and velocity information and the dark matter particle positions and velocities. These are controlled via the parameters `CosmologySimulationDensityName`, `CosmologySimulationVelocity[123]Name` (where 1, 2 and 3 correspond to the x, y and z directions, respectively), `CosmologySimulationParticlePositionName` and `CosmologySimulationParticleVelocityName`. Assuming that the baryon velocity information is all in a single file, and that the baryon density and velocity file names are `GridDensity` and `GridVelocities`, and that the particle position and velocity files are named `ParticlePositions` and `ParticleVelocities`, these parameters would be set as follows:

```
CosmologySimulationDensityName = GridDensity
CosmologySimulationVelocity1Name = GridVelocities
CosmologySimulationVelocity2Name = GridVelocities
CosmologySimulationVelocity3Name = GridVelocities
CosmologySimulationParticlePositionName = ParticlePositions
CosmologySimulationParticleVelocityName = ParticleVelocities
```

Some more advanced are parameters in the Initialization Parameters section control domain and boundary value specifications. These should NOT be altered unless you really, really know what you're doing!

2.5.2 Cosmology

Complete descriptions of all cosmology parameters are given here and here. `ComovingCoordinates` determines whether comoving coordinates are used or not. In practice, turning this off turns off all of the cosmology machinery, so you want to leave it set to 1 for a cosmology simulation. `CosmologyInitialRedshift` and `CosmologyFinalRedshift` control the start and end times of the simulation, respectively. `CosmologyHubbleConstantNow` sets the Hubble parameter, and is specified at z=0 in units of 100 km/s/Mpc. `CosmologyComovingBoxSize` sets the size of the box to be simulated (in units of Mpc/h) at z=0. `CosmologyOmegaBaryonNow`, `CosmologyOmegaMatterNow`, `CosmologyOmegaCDMNow` and `CosmologyOmegaLambdaNow` set the amounts of baryons, total matter, dark matter and vacuum energy (in units of the critical density at z=0). An addition to the standard baryon fields that can be initialized, one can create

a metal tracer field by turning on `CosmologySimulationUseMetallicityField`. This is handy for simulations with star formation and feedback (described below). For example, in a cosmology simulation with box size 100 Mpc/h with approximately the cosmological parameters determined by WMAP, which starts at z=50 and ends at z=2, and has a metal tracer field, we put the following into the parameter file:

```
ComovingCoordinates = 1
CosmologyInitialRedshift = 50.0
CosmologyFinalRedshift = 2.0
CosmologyHubbleConstantNow = 0.7
CosmologyComovingBoxSize = 100.0
CosmologyOmegaBaryonNow = 0.04
CosmologyOmegaMatterNow = 0.3
CosmologyOmegaCDMNow = 0.26
CosmologyOmegaLambdaNow = 0.7
CosmologySimulationUseMetallicityField = 1
```

2.5.3 Gravity and Particle Parameters

The parameter list sections on gravity particle positions are here and here, respectively. The significant gravity-related parameters are `SelfGravity`, which turns gravity on (1) or off (0) and `GravitationalConstant`, which must be 1 in cosmological simulations. `BaryonSelfGravityApproximation` controls whether gravity for baryons is determined by a quick and reasonable approximation. It should be left on (1) in most cases. For a cosmological simulation with self gravity, we would put the following parameters into the startup file:

```
SelfGravity = 1
GravitationalConstant = 1
BaryonSelfGravityApproximation = 1
```

We discuss some AMR and parallelization-related particle parameters in later sections.

2.5.4 Adiabatic hydrodynamics parameters

The parameter listing section on hydro parameters can be found here. The most fundamental hydro parameter that you can set is `HydroMethod`, which lets you decide between the Piecewise Parabolic Method (aka PPM; option 0), or the finite-difference method used in the Zeus astrophysics code (option 2). PPM is the more advanced and optimized method. The Zeus method uses an artificial viscosity-based scheme and may not be suited for some types of work. When using PPM in a cosmological simulation, it is important to turn `DualEnergyFormalism` on (1), which makes total-energy schemes such as PPM stable in a regime where there are hypersonic fluid flows, which is quite common in cosmology. The final parameter that one must set is `Gamma`, the ratio of specific heats for an ideal gas. If `MultiSpecies` (discussed later in *Radiative Cooling and UV Physics Parameters*) is on, this is ignored. For a cosmological simulation where we wish to use PPM and have `Gamma` = 5/3, we use the following parameters:

```
HydroMethod = 0
DualEnergyFormalism = 1
Gamma = 1.66667
```

In addition to these three parameters, there are several others which control more subtle aspects of the two hydro methods. See the parameter file listing of hydro parameters for more information on these.

One final note: If you are interested in performing simulations where the gas has an isothermal equation of state (gamma = 1), this can be approximated without crashing the code by setting the parameter Gamma equal to a number which is reasonably close to one, such as 1.001.

2.5.5 AMR Hierarchy Control Parameters

These parameters can be found in the parameter list page here. They control whether or not the simulation uses adaptive mesh refinement, and if so, the characteristics of the adaptive meshing grid creation and refinement criteria. We'll concentrate on a simulation with only a single initial grid first, and then discuss multiple levels of initial grids in a subsection.

The most fundamental AMR parameter is `StaticHierarchy`. When this is on (1), the code is a unigrid code. When it is off (0), adaptive mesh is turned on. `RefineBy` controls the refinement factor - for example, a value of 2 means that a child grid is twice as highly refined as its parent grid. It is important to set `RefineBy` to 2 when using cosmology simulations - this is because if you set it to a larger number (say 4), the ratio of particle mass to gas mass in a cell grows by a factor of eight during each refinement, causing extremely unphysical effects. `MaximumRefinementLevel` determines how many possible levels of refinement a given simulation can attain, and `MaximumGravityRefinementLevel` defines the maximum level at which gravitational accelerations are computed. More highly refined levels have their gravitational accelerations interpolated from this level, which effectively provides smoothing of the gravitational force on the spatial resolution of the grids at `MaximumGravityRefinementLevel`. A simulation with AMR turned on, where there are 6 levels of refinement (with gravity being smoothed on level 4) and where each child grid is twice as highly resolved as its parent grid would have these parameters set as follows:

```
StaticHierarchy = 0
RefineBy = 2
MaximumRefinementLevel = 6
MaximumGravityRefinementLevel = 4
```

Once the AMR is turned on, you must specify how and where the hierarchy refines. The parameter `CellFlaggingMethod` controls the method in which cells are flagged, and can be set with multiple values. We find that refining by baryon and dark matter mass (options 2 and 4) are typically useful in cosmological simulations. The parameter `MinimumOverDensityForRefinement` allows you to control the overdensity at which a given grid is refined, and can is set with multiple values as well. Another very useful parameter is `MinimumMassForRefinementLevelExponent`, which modifies the cell masses/overdensities used for refining grid cells. See the parameter page for a more detailed explanation. Leaving this with a value of 0.0 ensures that gas mass resolution in dense regions remains more-or-less Lagrangian in nature. Negative values make the refinement super-Lagrangian (ie, each level has less gas mass per cell on average than the coarser level above it) and positive values make the refinement sub-lagrangian. In an AMR simulation where the AMR triggers on baryon and dark matter overdensities in a given cell of 4.0 and 8.0, respectively, where the refinement is slightly super-Lagrangian, these paramaters would be set as follows:

```
CellFlaggingMethod = 2 4
MinimumOverDensityForRefinement = 4.0 8.0
MinimumMassForRefinementLevelExponent = -0.1
```

At times it is very useful to constrain your simulation such that only a small region is adaptively refined (the default is to refine over an entire simulation volume). For example, if you wish to study the formation of a particular galaxy in a very large volume, you may wish to only refine in the small region around where that galaxy forms in your simulation in order to save on computational expense and dataset size. Two parameters, `RefineRegionLeftEdge` and `RefineRegionRightEdge` allow control of this. For example, if we only want to refine in the inner half of the volume (0.25 - 0.75 along each axis), we would set these parameters as follows:

```
RefineRegionLeftEdge = 0.25 0.25 0.25
RefineRegionRightEdge = 0.75 0.75 0.75
```

This pair of parameters can be combined with the use of nested initial grids (discussed in the next subsection) to get simulations with extremely high dark matter mass and spatial resolution in a small volume at reasonable computational cost.

Multiple nested grids

At times it is highly advantageous to use multiple nested grids. This is extremely useful in a situation where you are interested in a relatively small region of space where you need very good dark matter mass resolution and spatial resolution while at the same time still resolving large scale structure in order to preserve gravitational tidal forces. An excellent example of this is formation of the first generation of objects in the universe, where we are interested in a relatively small (10^6 solar mass) halo which is strongly tidally influenced by the large-scale structure around it. It is important to resolve this halo with a large number of dark matter particles in order to reduce frictional heating, but the substructure of the distant large-scale structure is not necessarily interesting, so it can be resolved by very massive particles. One could avoid the complication of multiple grids by using a single very large grid - however, this would be far more computationally expensive.

Let us assume for the purpose of this example that in addition to the initial root grid grids (having 128 grid cells along each axis) there are two subgrids, each of which is half the size of the one above it in each spatial direction (so subgrid 1 spans from 0.25-0.75 in units of the box size and subgrid 2 goes from 0.375-0.625 in each direction). If each grid is twice as highly refined spatially as the one above it, the dark matter particles on that level are 8 times smaller, so the dark matter mass resolution on grid #2 is 64 times better than on the root grid, while the total number of initial grid cells only increases by a factor of three (since each grid is half the size, but twice as highly refined as the one above it, the total number of grid cells remains the same). Note: See the page on generating initial conditions for more information on creating this sort of set of nested grids.

When a simulation with more than one initial grid is run, the total number of initial grids is specified by setting `CosmologySimulationNumberOfInitialGrids`. The parameter `CosmologySimulationGridDimension[#]` is an array of three integers setting the grid dimensions of each nested grid, and `CosmologySimulationGridLeftEdge[#]` and `CosmologySimulationGridRightEdge[#]` specify the left and right edges of the grid spatially, in units of the box size. In the last three parameters, "#" is replaced with the grid number. The root grid is grid 0. None of the previous three parameters need to be set for the root grid. For the setup described above, the parameter file would be set as follows:

```
CosmologySimulationNumberOfInitialGrids = 3
CosmologySimulationGridDimension[1] = 128 128 128
CosmologySimulationGridLeftEdge[1] = 0.25 0.25 0.25
CosmologySimulationGridRightEdge[1] = 0.75 0.75 0.75
CosmologySimulationGridLevel[1] = 1
CosmologySimulationGridDimension[2] = 128 128 128
CosmologySimulationGridLeftEdge[2] = 0.375 0.375 0.375
CosmologySimulationGridRightEdge[2] = 0.625 0.625 0.625
CosmologySimulationGridLevel[2] = 2
```

Multiple initial grids can be used with or without AMR being turned on. If AMR is used, the parameter `MinimumOverDensityForRefinement` must be modified as well. It is advisable to carefully read the entry for this parameter in the parameter list (in this section). The minimum overdensity needs to be divided by $r^{(d*l)}$, where r is the refinement factor, d is the dimensionality, and l is the zero-based highest level of the initial grids. So if we wish for the same values for `MinimumOverDensityForRefinement` used previous to apply on the most highly refined grid, we must divide the set values by $2^{(3*2)} = 64$. In addition, one should only refine on the highest level, so we must reset `RefineRegionLeftEdge` and `RefineRegionRightEdge`. The parameters would be reset as follows:

```
RefineRegionLeftEdge = 0.375 0.375 0.375
RefineRegionRightEdge = 0.625 0.625 0.625
MinimumOverDensityForRefinement = 0.0625 0.125
```

A note: When creating multi-level intial conditions, make sure that the initial conditions files for all levels have the same file name (ie, `GridDensity`), but that each file has an extension which is an integer corresponding to its level. For example, the root grid `GridDensity` file would be `GridDensity.0`, the level 1 file would be `GridDensity.1`, and so forth. The parameters which describe file names (discussed above in the section on initial-

ization parameters) should only have the file name to the left of the period the period (as in a simulation with a single initial grid), ie,

```
CosmologySimulationDensityName = GridDensity
```

Nested Grids and Particles

When initializing a nested grid problem, there can arise an issue of lost particles as a result of running ring. Please see *Particles in Nested Grid Cosmology Simulations* for more information.

2.5.6 I/O Parameters

These parameters, defined in more detail in *Controlling Enzo data output*, control all aspects of Enzo's data output. One can output data in a cosmological simulation in both a time-based and redshift-based manner. To output data regularly in time, one sets `dtDataDump` to a value greater than zero. The size of this number, which is in units of Enzo's internal time variable, controls the output frequency. See the Enzo user's manual section on output format for more information on physical units. Data can be output at specific redshifts as controlled by `CosmologyOutputRedshift[#]`, where # is the number of the output dump (with a maximum of 10,000 zero-based numbers). The name of the time-based output files are controlled by the parameter `DataDumpName` and the redshift-based output files have filenames controlled by `RedshiftDumpName`. For example, if we want to output data every time the code advances by dt=2.0 (in code units) with file hierarchiess named `time_0000`, `time_0001`, etc., and ALSO output explicitly at redshifts 10, 5, 3 and 1 with file hierarchy names `RedshiftOutput0000`, `RedshiftOutput0001`, etc., we would set these parameters as follows:

```
dtDataDump = 2.0
DataDumpName = time_
RedshiftDumpName = RedshiftOutput
CosmologyOutputRedshift[0] = 10.0
CosmologyOutputRedshift[1] = 5.0
CosmologyOutputRedshift[2] = 3.0
CosmologyOutputRedshift[3] = 1.0
```

Note that Enzo always outputs outputs data at the end of the simulation, regardless of the settings of `dtDataDump` and `CosmologyOutputRedshift`.

2.5.7 Radiative Cooling and UV Physics Parameters

Enzo comes with multiple ways to calculate baryon cooling and a metagalactic UV background, as described in detail here. The parameter `RadiativeCooling` controls whether or not a radiative cooling module is called for each grid. The cooling is calculated either by assuming equilibrium cooling and reading in a cooling curve, or by computing the cooling directly from the species abundances. The parameter `MultiSpecies` controls which cooling module is called - if `MultiSpecies` is off (0) the equilibrium model is assumed, and if it is on (1 or 2) then nonequilibrium cooling is calculated using either 6 or 9 ionization states of hydrogen and helium (corresponding to `MultiSpecies` = 1 or 2, respectively). The UV background is controlled using the parameter `RadiationFieldType`. Currently there are roughly a dozen backgrounds to choose from. `RadiationFieldType` is turned off by default, and can only be used when `Multispecies` = 1. For example, if we wish to use a nonequilibrium cooling model with a Haardt and Madau background with q_{alpha}= -1.8, we would set these parameters as follows:

```
RadiativeCooling = 1
MultiSpecies = 1
RadiationFieldType = 2
```

2.5.8 Star Formation and Feedback Physics Parameters

Enzo has multiple routines for star formation and feedback. Star particle formation and feedback are controlled separately, by the parameters `StarParticleCreation` and `StarParticleFeedback`. Multiple types of star formation and feedback can be used, e.g. models for Pop III stars for metal-free gas and models for Pop II stars for metal-enriched gas. These routines are disabled when these parameters are set equal to 0. These parameters are bitwise to allow multiple types of star formation routines can be used in a single simulation. For example if methods 1 and 3 are desired, the user would specify 10 (2^1+ 2^3), or if methods 0, 1 and 4 are wanted, this would be 19 (2^0+ 2^1+ 2^4). See *Star Formation and Feedback Parameters* for more details.

They are turned on when the i-th bit is flagged. The value of 2 is the recommended value. The most commonly used routines (2) are based upon an algorithm by Cen & Ostriker, and there are a number of free parameters. Note that it is possible to turn star particle formation on while leaving feedback off, but not the other way around.

For the star particle creation algorithm, stars are allowed to form only in cells where a minimum overdensity is reached, as defined by `StarMakerOverDensityThreshold`. Additionally, gas can only turn into stars with an efficiency controlled by `StarMakerMassEfficiency` and at a rate limited by `StarMakerMinimumDynamicalTime`, and the minimum mass of any given particle is controlled by the parameter `StarMakerMinimumStarMass`, which serves to limit the number of star particles. For example, if we wish to use the "standard" star formation scenario where stars can only form in cells which are at least 100 times the mean density, with a minimum dynamical time of 10^6 years and a minimum mass of 10^7 solar masses, and where only 10% of the baryon gas in a cell can be converted into stars in any given timestep, we would set these parameters as follows:

```
StarParticleCreation = 2
StarMakerOverDensityThreshold = 100.0
StarMakerMassEfficiency = 0.1
StarMakerMinimumDynamicalTime = 1.0e6
StarMakerMinimumStarMass = 1.0e7
```

Star particles can provide feedback into the Inter-Galactic Medium via stellar winds, thermal energy and metal pollution. The parameter `StarMassEjectionFraction` controls the fraction of the total initial mass of the star particle which is eventually returned to the gas phase. `StarMetalYield` controls the mass fraction of metals produced by each star particle that forms, and `StarEnergyToThermalFeedback` controls the fraction of the rest-mass energy of the stars created which is returned to the gas phase as thermal energy. Note that the latter two parameters are somewhat constrained by theory and observation to be somewhere around 0.02 and 1.0e-5, respectively. The ejection fraction is poorly constrained as of right now. Also, metal feedback only takes place if the metallicity field is turned on (`CosmologySimulationUseMetallicityField` = 1). As an example, if we wish to use the 'standard' star feedback where 25% of the total stellar mass is returned to the gas phase, the yield is 0.02 and 10^{-5} of the rest mass is returned as thermal energy, we set our parameters as follows:

```
StarParticleFeedback = 2
StarMassEjectionFraction = 0.25
StarMetalYield = 0.02
StarEnergyToThermalFeedback = 1.0e-5
CosmologySimulationUseMetallicityField = 1
```

When using the star formation and feedback algorithms it is important to consider the regime of validity of our assumptions. Each "star particle" is supposed to represent an ensemble of stars, which we can characterize with the free parameters described above. This purely phenomenological model is only reasonable as long as the typical mass of the star particles is much greater than the mass of the heaviest stars so that the assumption of averaging over a large population is valid. When the typical star particle mass drops to the point where it is comparable to the mass of a large star, these assumptions must be reexamined and our algorithms reformulated.

2.5.9 IO Parallelization Options

One of Enzo's great strengths is that it is possible to do extremely large simulations on distributed memory machines. For example, it is possible to intialize a 1024^3 root grid simulation on a linux cluster where any individual node has 1 or 2 GB of memory, which is on the order of 200 times less than the total dataset size! This is possible because the reading of initial conditions and writing out of data dumps is fully parallelized - at startup, when the parameter `ParallelRootGridIO` is turned on each processor only reads the portion of the root grid which is within its computational domain, and when `ParallelParticleIO` is turned on each processor only reads in the particles within its domain (though preprocessing is needed - see below). Additionally, the parameter `Unigrid` should be turned on for simulations without AMR, as it saves roughly a factor of two in memory on startup, allowing the code to perform even larger simulations for a given computer size. If we wish to perform an extremely large unigrid simulation with parallel root grid and particle IO, we would set the following parameters:

```
ParallelParticleIO = 1
ParallelRootGridIO = 1
Unigrid = 1
```

AMR simulations can be run with `ParallelRootGridIO` and `ParallelParticleIO` on, though you must be careful to turn off the `Unigrid` parameter. In addition, it is important to note that in the current version of Enzo you must run the program called "ring" on the particle position and velocity files before Enzo is started in order to take advantage of the parallel particle IO. Assuming the particle position and velocity files are named `ParticlePositions` and `ParticleVelocities`, respectively, this is done by running:

```
mpirun -np [N] ring ParticlePositions ParticleVelocities
```

Where mpirun is the executable responsible for running MPI programs and "-np [N]" tells the machine that there are [N] processors. This number of processors must be the same as the number which Enzo will be run with!

2.5.10 Notes

This page is intended to help novice Enzo users put together parameter files for their first simulation and therefore is not intended to be an exhaustive list of parameters nor a complete description of each parameter mentioned. It would be wise to refer to the Enzo user guide's *Enzo Parameter List* for a more-or-less complete list of AMR parameters, some of which may be extremely useful for your specific application.

2.6 Data Analysis Basics

Data analysis in Enzo can be complicated. There are excellent premade packages available for doing Enzo data analysis (see *SupportingCodes*.). However, it is likely that your data analysis needs will grow beyond these tools.

2.6.1 HDF5 Tools

Enzo reads in initial conditions files and outputs simulation data using the HDF5 structured data format (created and maintained by the NCSA HDF group). Though this format takes a bit more effort to code than pure C/C++ binary output, we find that the advantages are worth it. Unlike raw binary, HDF5 is completely machine-portable and the HDF5 library takes care of error checking. There are many useful standalone utilities included in the HDF5 package that allow a user to examine the contents and structure of a dataset. In addition, there are several visualization and data analysis packages that are HDF5-compatible. See the page on Data Vizualization for more information about this. The NCSA HDF group has an excellent tutorial on working with HDF5.

Note that as of the Enzo 2.0 code release, Enzo still supports reading the HDF4 data format, but not writing to it. We strongly suggest that new users completely avoid this and use the HDF5 version instead. Enzo's parallel IO only works with HDF5, and we are encouraging users migrate as soon as is feasible.

2.6.2 Using YT to Analyze Data

If you have installed YT along with Enzo (as suggested in the build instructions *Obtaining and Building Enzo*), you should be able to use it to find halos, examine profiles, prepare plots and handle data directly via physically meaningful objects. Documentation, a wiki and a mailing list are available for support and assistance with installation and usage as well as a brief introduction in these documents *Analyzing With YT*

2.6.3 Analysis with VisIt

Another tool that has a native reader for Enzo data is VisIt, a parallel VTK-based visualization and analysis tool.

From the VisIt Users website:

> VisIt is a free interactive parallel visualization and graphical analysis tool for viewing scientific data on Unix and PC platforms. Users can quickly generate visualizations from their data, animate them through time, manipulate them, and save the resulting images for presentations. VisIt contains a rich set of visualization features so that you can view your data in a variety of ways. It can be used to visualize scalar and vector fields defined on two- and three-dimensional (2D and 3D) structured and unstructured meshes. VisIt was designed to handle very large data set sizes in the tera- to peta-scale range and yet can also handle small data sets in the kilobyte range.

The caveat is that as of version 1.11.2, VisIt only understands the original unpacked AMR format. However, the packed-AMR is in the VisIt development version, and will be included in the next release (1.12). If would like this functionality sooner, it's not too much work. Here's how to begin:

1. Download the following:
   - The 1.11.2 source distribution
   - The 1.11.2 build_visit script
   - An updated avtEnzoFileFormat.C
   - An updated avtEnzoFileFormat.h
2. Untar the source tar file,
3. replace the two files named avtEnzo* in visit1.11.2/src/databases/Enzo/ with the ones you've just downloaded, and
4. retar the file, keeping the same directory structure.

(You can do this without untarring and retarring, but this is a bit clearer for those not familiar with tar.) From this point, you can build and install VisIt using the build_visit script. When you do this, remember to do two things:

- Use the TARBALL option to specify the tar file for the script to unpack. Failing to do this will cause the script to download a new tar file, without the changes that you need.
- Select **both** HDF5 and HDF4 as optional third-party libraries. This may not strictly be necessary, if you already have HDF5 and HDF4 installed on your system, but the script isn't clear on how to specify which HDF5 installation to use. (HDF4 needs to be available to satisfy a dependency check for building the Enzo reader. We'll ask to have this updated in future versions of VisIt.)

2.6.4 Writing your own tools, I - the Enzo Grid Hierarchy

Enzo outputs each individual adaptive mesh block as its own grid file. Each of these files is completely self-contained, and has information about all of the grid cells that are within that volume of space. Information on the size and spatial location of a given grid file can be obtained from the hierarchy file, which has the file extension ".hierarchy". This ascii file has a listing for each grid that looks something like this:

```
Grid = 26
GridRank          = 3
GridDimension     = 34 22 28
GridStartIndex    = 3 3 3
GridEndIndex      = 30 18 24
GridLeftEdge      = 0.5 0.28125 0.078125
GridRightEdge     = 0.71875 0.40625 0.25
Time              = 101.45392321467
SubgridsAreStatic = 0
NumberOfBaryonFields = 5
FieldType = 0 1 4 5 6
BaryonFileName = RedshiftOutput0011.grid0026
CourantSafetyNumber    = 0.600000
PPMFlatteningParameter = 0
PPMDiffusionParameter  = 0
PPMSteepeningParameter = 0
NumberOfParticles   = 804
ParticleFileName = RedshiftOutput0011.grid0026
GravityBoundaryType = 2
Pointer: Grid[26]->NextGridThisLevel = 27
```

`GridRank` gives the dimensionality of the grid (this one is 3D), `GridDimension` gives the grid size in grid cells, including ghost zones. `GridStartIndex` and `GridEndIndex` give the starting and ending indices of the non-ghost zone cells, respectively. The total size of the baryon datasets in each grid along dimension i is (1+ `GridEndIndex[i]-GridStartIndex[i]`). `GridLeftEdge` and `GridRightEdge` give the physical edges of the grids (without ghost zones) in each dimension. `NumberOfParticles` gives the number of dark matter particles (and/or star particles, for simulations containing star particles) in a given grid. Note that when there are multiple grids covering a given region of space at various levels of resolution, particles are stored in the most highly refined grid. `BaryonFileName` is the name of the actual grid file, and should be the same as `ParticleFileName`. `Time` is the simulation time, and should be the same as `InitialTime` in the parameter file for the same data dump. The other parameters for each entry are more advanced and probably not relevant for simple data analysis.

Possibly the greatest source of potential confusion in Enzo's datasets is the overlap of grid cells. In a simulation, when a given grid is further refined, the coarse cells which have not been refined are still kept. The solution to the hydro and gravity equations are still calculated on that level, but are updated with information from more highly refined levels. What this is means is that a volume of space which has been refined beyond the root grid is covered by multiple grid patches at different levels of resolution. Typically, when doing analysis you only want the most highly refined information for a given region of space (or the most highly refined up to a certain level) so that you don't double-count (or worse) the gas in a given cell. Look at this example analysis code.

2.6.5 Writing your own tools, II - Enzo Physical Units

Yet another significant source of confusion is the units that Enzo uses. When doing a cosmology simulation, the code uses a set of units that make most quantities on the order of unity (in principle). The Enzo manual section on the code output format *Enzo Output Formats* explains how to convert code units to cgs units. However, there are some subtleties:

Density fields All density fields are in the units described in the AMR guide **except** electron density. Electron density is only output when `MultiSpecies` is turned on, and in order to convert the electron density to cgs it must be multiplied by the code density conversion factor and then (m:sub:*e*/m:sub:*p*), where m:sub:*e*and m:sub:*p*are the electron and proton rest masses (making electron density units different from the other fields by a factor of m:sub:*e*/m:sub:*p*). The reason this is done is so that in the code the electron density can be computed directly from the abundances of the ionized species.

Energy fields There are two possible energy fields that appear in the code - Gas energy and total energy. Both are in units of **specific energy**, ie, energy per unit mass. When Zeus hydro is being used (`HydroMethod` = 2, there

should be only one energy field - "total energy". This is a misnomer - the Zeus hydro method only follows the specific internal (ie, thermal) energy of the gas explicitly. When the total energy is needed, it is calculated from the velocities. When PPM is used (`HydroMethod` = 0) the number of energy fields depends on whether or not `DualEnergyFormalism` is turned on or off. If it is ON (1), there is a "gas energy" field and a "total energy" field, where "gas energy" is the specific internal energy and "total energy" is "gas energy" plus the specific kinetic energy of the gas in that cell. If `DualEnergyFormalism` is OFF (0), there should only be "total energy", which is kinetic+internal specific energies. Confused yet?

Particle mass field Particle "masses" are actually stored as densities. This is to facilitate calculation of the gravitational potential. The net result of this is that, in order to calculate the stored particle "mass" to a physical mass, you must first multiply this field by the volume of a cell in which the particle resides. Remember that particle data is only stored in the most refined grid that covers that portion of the simulational volume.

When the simulation is done, Enzo will display the message "Successful run, exiting." Enzo is a complicated code, with a similarly complicated output format. See the Enzo User Guide page on the Enzo output format *Enzo Output Formats* for more information on the data outputs.

Congratulations! If you've made it this far, you have now successfully run a simulation using Enzo!

2.6.6 Example Data and Analysis

The sample data generated by this simulation is available online. You can use it as sample data for the the YT tutorial.

2.7 Controlling Enzo data output

How and when Enzo outputs data is described below. There are five ways to control when data is output, two output formats, and two pitfalls when determining how to output data from your Enzo simulation.

2.7.1 Data Formats and Files

There are two output formats for Enzo data. In both cases, each data dump gets its own directory.

Each data dump writes several key files. NNNN denotes the dump number (i.e. 0001) and basename is something like RedshiftOutput or data or DD}.

All output files are also restart files. It's not necessarily wise to write in 32 bit format if you're computing in 64, though, as you'll lose all the extra precision when you restart. (These are makefile flags.)

basenameNNNN:

```
The parameter file. This contains general simulation parameters,
dump time, cycle, and all the parameters defined here. It's worth
your time to be familiar with what's in this file.
```

basenameNNNN.hierarchy:

```
The hierarchy file in text format. Contains a description of the hierarchy. One
entry for each grid, including information like the Grid Size, the
position in the volume, it's position in the hierarchy.
```

basenameNNNN.boundary:

```
A description of the boundary (plain text.) Basically a meta
description and filename for the next file
```

basenameNNNN.boundary.hdf5:

```
Actually contains the boundary information.
```

basenameNNNN.harrays:

```
The hierarchy of grids stored in HDF5 binary format.
```

Packed AMR

This is the default output format. Each processor outputs all the grids it owns. In addition to the parameter, hierarchy, and boundary files which may or may not be described elsewhere, data is output in one basenameNNNN.taskmapCCCC} file for each processor, which contains a map between grid number and HDF5 file, and one basenameNNNN.cpuCCCC for each processor NNNN and CCCC are the dump number and cpu number, respectively.

basenameNNNN.cpuCCCC is an HDF5 file which contains an HDF5 group for each grid. Each grid in turn contains a dataset for each of the fields in the simulation.

```
~/DD0100>h5ls data0100.cpu0003
Grid00000002             Group
Grid00000026             Group
~/DD0100>h5ls data0100.cpu0003/Grid00000002
Density                  Dataset {16, 16, 32}
z-velocity               Dataset {16, 16, 32}
```

2.7.2 Pathnames

In previous versions of Enzo, the fully-qualified path to each file was output in the `.hierarchy` file, which requires modifying the `.hierarchy` file every time the data was moved. This has changed to be only the *relative* path to each data file, which largely eliminates the problem. To restore the old behavior, examine the parameters `GlobalDir` and `LocalDir`.

2.7.3 Timing Methods

There are 6 ways to trigger output from Enzo.

Cycle Based Output

```
CycleSkipDataDump = N
CycleLastDataDump = W
DataDumpName = data
```

One can trigger output every N cycles starting with cycle W using CycleSkipDataDump and CycleLastDataDump. Outputs are put in the directory DD0000 (or DD0001, etc.) and the basename is determined by DataDumpName.

CycleSkipDataDump <= 0 means cycle based output is skipped. The default is 0.

Pitfall 2: CycleLastDataDump defaults to zero and is incremented by CycleSkipDataDump every time output is done. If you change the value of CycleSkipDataDump and neglect to change CycleLastDataDump, Enzo will dump as long as CycleNumber >= CycleSkipDataDump + CycleLastDataDump. (So if you change CycleSkipDataDump from 0 to 10 from a Redshift dump at n=70, you'll get an output every timestep for 7 timesteps.)

Time Based Output

```
TimeLastDataDump = V
dtDataDump = W
```

Exactly like Cycle based output, but triggered whenever time >= TimeLastDataDump + dtDataDump. The same pitfall applies.

Redshift Based Output

```
CosmologyOutputRedshift[ 0 ] = 12
CosmologyOutputRedshiftName[ 0 ] = Redshift12
RedshiftDumpName             = RedshiftOutput
```

Outputs at the specified redshift. Any number of these can be specified.

CosmologyOutputRedshift[i] is the only necessary parameter, and is the ith redshift to output.

Any outputs with CosmologyOutputRedshiftName[i] specified has that name used for the output, and no number is appended. (so if CosmologyOutputRedshiftName[6] = BaconHat, the outputs will be BaconHat, BaconHat.hierarchy, etc.)

If CosmologyOutputRedshiftName[i] is omitted, RedshiftDumpName is used for the basename, and the output number is taken from the array index. (So CosmologyOutputRedshift[19] = 2.34 and RedshiftDumpName = MonkeyOnFire, at dump will be made at z=2.34 with files called MonkeyOnFire0019.hierarchy, etc.)

Force Output Now

The following two options are run time driven. These are especially useful for very deep simulations that spend the majority of their time on lower levels. Note that unless you have the parameter `FileDirectedOutput` turned on, these will not be available.

To force an output as soon as the simulation finished the next step on the finest resolution, make a file called outputNow:

```
touch outputNow
```

This will remove the file as soon as the output has finished.

Sub Cycle Based Output

To get the simulation to output every 10 subsycles (again at the finest level of resolution) put the number of subcycles to skip in a file called subcycleCount:

```
echo 10 > subcycleCount
```

Time Based Interpolated Output

Even when you are running simulations with a long dtDataDump, sometimes you may want to see or analyze the interim datadumps. Using dtInterpolatedDataDump, you can control Enzo to check if it should start outputting interpolated data based on the time passed (dtInterpolatedDataDump < dtDataDump).

```
dtDataDump = 1e-4
dtInterpolatedDataDump = 1e-5
```

This is mostly for making movies or looking at the interim data where the TopGrid dt is too long, and in principle, this output shouldn't be used for restart.

2.7.4 Friendly Note on Data Output

Enzo is content to output enough data to fill up a hard drive – for instance, your home directory. This should be noted before output parameters are set, particularly the Sub Cycle outputs, as Enzo has no prohibition against causing problems with quotas and file system size.

CHAPTER

THREE

USER GUIDE

This document provides a brief description of the compilation and operation of Enzo, a structured Adaptive Mesh Refinement (SAMR, or more loosely AMR) code which is primarily intended for use in astrophysics and cosmology. The User's Guide is intended to explain how to compile and run Enzo, the initial conditions generation code and the various analysis tools bundled with Enzo. The instructions on actually running the code are not comprehensive in that they are not machine or platform-specific. Arguably the most useful and important piece of this guide is *Enzo Parameter List*, which contains descriptions of all of the roughly 300 possible input parameters (as of September 2008). For more detailed information on the Enzo algorithms and on running Enzo on different platforms, you should refer to the *Getting Started with Enzo*. Detailed information on the algorithms used in Enzo will be available in the method paper (unreleased as of September 2008). In the meantime, see the *Enzo Primary References* for more concrete Enzo information.

This guide (and Enzo itself) was originally written by Greg Bryan. Since the original writing of both the simulation code and the User's Guide, the maintenance of Enzo and its associated tools and documentation was for some time largely driven by the Laboratory for Computational Astrophysics at The University of California, San Diego, but it is now a fully open source community with developers from Stanford, Columbia, Princeton, UCSD, University of Colorado, Michigan State, UC Berkeley, and many other universities. Your input in improving both the code and the User's Guide is appreciated – developement of the code is driven by working researchers, and we encourage everyone who has made useful changes to contribute those changes back to the community and to participate in the collaborative development of the code. Email inquiries and comments should be directed to the Enzo Users' List. Thank you!

3.1 Executables, Arguments, and Outputs

This page is a summary of all of the binaries that are created after `make; make install` is run in the Enzo code bundle. They should be located in the `bin` directory. Links to the various pages of the manual that describe a particular binary are also included.

3.1.1 enzo

This is the main simulation code executable. See *Running Enzo* for more detailed information.

When an Enzo simulation is run, at every datastep several files are output, inserted into subdirectories. The most important of these are the files with no extension and those ending in `.hierarchy`, of which there will be one of each for each datadump. For more information on the format of Enzo output, see *Enzo Output Formats*.

```
usage: ./enzo.exe [options] param_file
   options are:
      -d(ebug)
      -r(estart)
      -x(extract)
```

```
      -l(evel_of_extract) level
   -p(roject_to_plane) dimension
   -P(roject_to_plane version 2) dimension
      -m(smooth projection)
   -o(utput as particle data)
   -h(elp)
   -i(nformation output)
   -s(tart  index region) dim0 [dim1] [dim2]
   -e(nd    index region) dim0 [dim1] [dim2]
   -b(egin  coordinate region) dim0 [dim1] [dim2]
   -f(inish coordinate region) dim0 [dim1] [dim2]
```

3.1.2 inits

This is the initial conditions generator. See *Using inits* for more detailed information. Initial conditions with a single initial grid or multiple nested grids can be created with this executable. Output file names are user-specified, but in a standard cosmology simulation with a single initial grid there should be a file containing baryon density information, another containing baryon velocity information, and two more files containing particle position and velocity information. Simulations with multiple grids will have a set of these files for each level, appended with numbers to make them unique.

```
usage: inits [options] param_file
   options are:
      -d(ebug)
      -s(ubgrid) param_file
```

3.1.3 ring

ring must be run on the simulation particle position and velocity information before a simulation is executed when the Enzo runtime parameter `ParallelParticleIO` is set to 1. Running ring generates files called PPos.nnnn PVel.nnnn where nnnn goes from 0001 to the total number of processors that are used for the simulation. These files contain the particle position and velocity information for particles that belong to each processor individually, and will be read into the code instead of the monolithic particle position and velocity files. Note that if `ParallelParticleIO` is on and ring is NOT run, the simulation will crash.

```
usage:  ring [string] <particle position file> <particle velocity file>
```

[string] can be one of the following: pv, pvm, pvt, or pvmt. p, v, m and t correspond to position, velocity, mass, and type, respectively. The most common [string] choice is 'pv'. In that case, and if you use the default names for the particle position and velocity files, your usage will look like:

```
ring pv ParticlePositions ParticleVelocities
```

3.1.4 enzohop

The second (and generally favored) method used for finding density peaks in an Enzo simulation. More information can be found here. A file called `HopAnalysis.out` is output which contains halo position and mass information.

```
enzohop [-b #] [-f #] [-t #] [-g] [-d] amr_file
  -b)egin region
  -f)inish region
  -t)hreshold for hop (default 160)
```

```
  -g)as particles also used (normally just dm)
  -d)ebug
```

3.1.5 anyl

anyl is the analysis package written in C, previously known as enzo_anyl. Although the analysis toolkit for enzo that's being constantly updated is YT, anyl has its own value for some users. It creates radial, disk, vertical profiles for baryon (each species), dark matter, and star particles. Works with all AMR formats including HDF4 and packed HDF5.

```
usage: anyl.exe <amr file> <anyl parameter file>
```

3.2 Running Enzo

Once the code is compiled and a parameter file is prepared, starting the simulation is easy:

```
mpirun -np 1 enzo [-d] parameter_file
```

The syntax of the mpirun varies between mpi implementations. The example given here comes from a machine using a standard MPI implementation that is initiated by the 'mpirun' command, and implies the use of a single processors (the argument after the -np flag indicates the number of processors).

The -d flag triggers a debug option that produces a substantial amount of output. See *Getting Started with Enzo* for more detailed information on running simulations. You may also need to use *ring* if you are using parallel I/O.

3.2.1 Restarting

During a run, there are a number of forms of output. The largest will probably be the output of the full dataset as specified by parameters such as `dtDataDump` and the `CosmologyOutputRedshift`. Such outputs contain a number of different files (sometimes many files if there are a large number of grids) and are explained elsewhere. It is useful to have a fairly large number of such outputs if the run is a long one, both to provide more information to analyze, but also in case of an unintended interruption (crash). Fortunately, any full output can be used to restart the simulation:

```
mpirun -np 1 enzo [-d] -r output_name
```

3.2.2 Monitoring information

As the simulation runs, at every top grid timestep, it outputs a line of information to the ascii file OutputLevelInformation (which is overwritten on restart). The amount of information on this line can be quite extensive, but here the format is briefly summarized. The first number is the problem time, while the next 6 relate to general information about the entire run. Within these six numbers, the first is the maximum level currently in use, the second is the number of grids, the third is a number proportional to the memory used, the fourth is the mean axis ratio of all grids, and the last two are reserved for future use. Then, there are three spaces, and another group of numbers, all providing information about the first (top grid) level. This pattern of three spaces and six numbers is repeated for every level. An example of this file is provided below.

```
Cycle 151  Time 20.241365  MaxDepth 4  Grids 412  Memory(MB) 53.3117  Ratio 2.22582
   Level 0  Grids 2  Memory(MB) 13.8452  Coverage 1  Ratio 2  Flagged 0  Active 262144
   Level 1  Grids 304  Memory(MB) 31.4977  Coverage 0.166855  Ratio 2.43768  Flagged 0  Active 34992(
   Level 2  Grids 76  Memory(MB) 5.81878  Coverage 0.00329208  Ratio 1.66118  Flagged 0  Active 5523(
```

```
  Level 3  Grids 22  Memory(MB) 1.74578  Coverage 0.000125825  Ratio 1.63561  Flagged 0  Active 168
  Level 4  Grids 8  Memory(MB) 0.404286  Coverage 2.5034e-06  Ratio 1.21875  Flagged 0  Active 2688
```

The information for each level is:

1. number of grids on the level
2. memory usage (minus overhead). Actual memory usage is usually a factor of 10 higher.
3. the volume fraction of the entire region covered by grids on this level,
4. the mean axis ratio of grids on this level
5. the fraction of cells on this level which need refinement (unused)
6. the number of active cells on this level.

3.2.3 Debugging information

It is often useful to run with the debug flag turned on, particularly if the code is crashing for unknown reasons. However, the amount of output is quite large so it is useful to redirect this to a log file, such as:

```
mpirun -np 1 enzo -d -r output_name >& log_file
```

Some modules (the cooling unit is particularly bad for this), produce their own debugging logs in the form of fort.?? files. These can be ignored unless problems occur.

3.2.4 Test Problems

There are a number of built-in tests, which can be used to debug the system or characterize how well it solves a particular problem. (see *Enzo Test Suite* for a complete list.) Note that Enzo can run any problem after compilation, since no compilation flags affect simulation parameters. To run a particular test, cd to the [browser:public/trunk/doc/examples doc/examples] subdirectory of the Enzo source distribution (after compiling enzo) and use the following command-line:

```
mpirun -np 1 enzo [-d] test_name
```

The syntax of the mpirun various from mpi implementation. The example given here comes from the Origin2000 and implies a single processor (the argument after the -np flag indicates the number of processors).

The parameter test_name corresponds to the parameter file that specifies the type of test and the test particulars. This file is ascii, and can be edited. It consists of a series of lines (and optional comments) each of which specifies the value of one parameter. The parameters are discussed in more detail in *Enzo Parameter List*.

If you just type `enzo` without any arguments, or if the number of arguments is incorrect, the program should respond with a summary of the command-line usage.

The -d flag turns on a rather verbose debug option.

For example, to run the shock tube test, use:

```
mpirun -np 1 enzo ShockTube
```

or

```
enzo ShockTube
```

The response should be:

```
Successfully read in parameter file ShockTube.
Successful completion...
```

How do you know if the results are correct? New for v2.0, we have added more regression tests and answer tests, using LCAtest. We hope to add more answer tests, especially for large production-type simulations, e.g. a 512^3 cosmology simulation.

3.3 Enzo Test Suite

The Enzo test suite is a set of tools whose purpose is to perform regression tests on the Enzo codebase, in order to help developers discover bugs that they have introducted, to verify that the code is producing correct results on new computer systems and/or compilers, and, more generally, to demonstrate that Enzo is behaving as expected under a wide variety of conditions.

3.3.1 What's in the test suite?

The suite is composed of a large number of individual test problems that are designed to span the range of physics and dimensionalities that are accessible using the Enzo test code, both separately and in various permutations. Tests can be selected based on a variety of criteria, including (but not limited to) the physics included, the estimated runtime of the test, and the dimensionality. For convenience, three pre-created, overlapping sets of tests are provided:

1. The "quick suite" (`--quicksuite=True`). This set of tests should run in a few minutes on a single core of a laptop. It is composed of one-dimensional calculations that test critical physics packages both alone and in combination. The intent of this package is to be run relatively frequently (multiple times a day) to ensure that bugs have not been introduced during the code development process.

2. The "push suite" (`--pushsuite=True`). This set of tests should run in roughly an hour or two on a single core of a laptop or desktop machine. It is composed of one-, two- and three-dimensional calculations that test a wider variety of physics modules, both alone and in combination, than the "quick suite" described above. The intent of this package is to provide a thorough validation of the code prior to changes being pushed to the Google Code mercurial repository.

3. The "full suite" (`--fullsuite=True`). This set of tests should run in no more than 24 hours on 8-16 cores of a Linux cluster, and includes a variety of 3D, multiphysics tests that are not in the "quick" and "push" suites. This suite provides the most rigorous possible validation of the code in many different situations, and is intended to be run prior to major changes being pushed to the Google Code repository and prior to public releases of the code.

3.3.2 How to run the test suite

The Enzo test suite is run within the `run/` subdirectory of the Enzo source distribution, using the `test_runner.py` file. To run the test suite, follow these instructions:

1. Before running the test suite, you should download the "gold standard" datasets from http://enzo-project.org/tests/gold_standard.tar.gz, and untar that file into a convenient directory.

2. Compile Enzo. The gold standard calculations use opt-debug and 64-bit precision everywhere (`make opt-debug`, `make precision-64`, `make particles-64`, and `make integers-64`). If you use significantly different compilation options (higher-level optimization in particular) you may see somewhat different outputs that will result in failed tests.

3. Go into the `run/` subdirectory in the Enzo repository and type the following command:

```
./test_runner.py --quicksuite=True  --compare-dir=/path/to/gold_standard \\
        --output-dir=/enzo/test/directory
```

In this comand, `--quicksuite=True` instructs the test runner to use the quick suite (other possible keyboards here are '–pushsuite=True' and '–fullsuite=True'). `--output-dir=/enzo/test/directory` instructs the test runner to write output to the user-specified directory, and `--compare-dir=/path/to/gold_standard` instructs the test runner to use the set of data files in the listed directory as a gold standard for comparison. It is also possible to choose sets of tests that are sorted by dimensionality, physics modules, runtime, number of processors required, and other criteria. Type `./test_runner.py --help` for a more complete listing.

3.3.3 How to add a new test to the library

It is hoped that any newly-created or revised physics module will be accompanied by one or more test problems, which will ensure the continued correctness of the code. This sub-section explains the structure of the test problem system as well as how to add a new test problem to the library.

Test problems are contained within the `run/` directory in the Enzo repository. This subdirectory contains a tree of directories where test problems are arranged by the primary physics used in that problem (e.g., Cooling, Hydro, MHD). These directories may be further broken down into sub-directories (Hydro is broken into Hydro-1D, Hydro-2D, and Hydro-3D), and finally into individual directories containing single problems. A given directory contains, at minimum, the Enzo parameter file (having extension `.enzo`, described in detail elsewhere in the manual) and the Enzo test suite parameter file (with extension `.enzotest`). The latter contains a set of parameters that specify the properties of the test. Consider the test suite parameter file for InteractingBlastWaves, which can be found in the `run/Hydro/Hydro-1D/InteractingBlastWavest` directory:

```
name = 'InteractingBlastWaves'
answer_testing_script = None
nprocs = 1
runtime = 'short'
critical = True
cadence = 'nightly'
hydro = True
gravity = False
dimensionality = 1
max_time_minutes = 1
```

This allows the user to specify the dimensionality, physics used, the runtime (both in terms of 'short', 'medium', and 'long' calculations, and also in terms of an actual wall clock time), and whether the test problem is critical (i.e., tests a fundamental piece of the code) or not. A full listing of options can be found in the `run/README` file.

Once you have created a new problem type in Enzo and thoroughly documented the parameters in the Enzo parameter list, you should follow these steps to add it as a test problem:

1. Create a new subdirectory in the appropriate place in the `run/` directory. If your test problem uses multiple pieces of physics, put it under the most relevant one.

2. Add an Enzo parameter file, ending in the extension `.enzo`, for your test problem to that subdirectory.

3. Add an Enzo test suite parameter file, ending in the extension `.enzotest`. In that file, add any relevant parameters (as described in the `run/README` file).

4. Create a "gold standard" set of data for your test problem, by running with opt-debug and 64-bit precision for floats and integers. Contact Britton Smith (brittonsmith@gmail.com) and arrange to send him this data. Please try to minimize the quantity of data generated by your calculation by only writing out data at the end of the calculation, not during the interim (unless evolution of a quantity or quantities is important).

If you want to examine the output of your test problem for something specific, you can optionally add a script that is indicated by the `answer_testing_script` parameter. Look in the directory `run/Hydro/Hydro-3D/RotatingCylinder` for an example of how this is done.

Congratulations, you've created a new test problem!

3.3.4 What to do if you fix a bug in Enzo

It's inevitable that bugs will be found in Enzo, and that some of those bugs will affect the actual simulation results (and thus the test problems used in the problem suite). If you fix a bug that results in a change to some or all of the test problems, the gold standard solutions will need to be updated. Here is the procedure for doing so:

1. Run the "push suite" of test problems (`--pushsuite=True`) for your newly-revised version of Enzo, and determine which test problems now fail.

2. Visually inspect the failed solutions, to ensure that your new version is actually producing the correct results!

3. Email the enzo-developers mailing list at enzo-dev@googlegroups.com to explain your bug fix, and to show the results of the now-failing test problems.

4. Once the denizens of the mailing list concur that you have correctly solved the bug, create a new set of gold standard test problem datasets, following the instructions in the next section.

5. After these datasets are created, send the new gold standard datasets to Britton Smith (brittonsmith@gmail.com), who will update the gold standard dataset tarball (http://enzo-project.org/tests/gold_standard.tar.gz).

6. Push your Enzo changes to the repository.

3.3.5 How to create a new set of reference calculations

It may be necessary for you to generate a set of reference calculations for some reason. If so, here is how you do this.

1. First, build Enzo using the recommended set of compile options, which includes the debug optimization level (`make opt-debug`), and 64-bit precision everywhere (`make precision-64`, `make particles-64`, and `make integers-64`). You will now have an enzo binary in the `src/enzo` directory.

2. Go into the `run/` directory and call test_runner.py without the `--compare-dir` directory. If you are have multiple Enzo repositories, you can specify the one you want:

```
./test_runner.py --repo=/path/to/desired/enzo/repo \\
     --output-dir=/path/to/new/reference/directory
```

Note that you should only use the top-level directory in the repository, not src/enzo, and if you simply want to use the current repository (that is, the one your run directory is located in) you can leave out the `--repo` option. Once this step is completed, you should have a full set of test problems.

3. If you then want to compare against this set of test problems, use the following command:

```
./test_runner.py --repo=/path/to/desired/enzo/repo  \\
     --compare-dir=/path/to/new/reference/directory  \\
     --output-dir=/path/to/output/directory
```

3.4 Creating Cosmological Initial Conditions

There are two mechanisms for creating cosmological initial conditions with Enzo. The original mechanism, `inits`, has long been distributed with Enzo. It is exclusively serial. We also now distribute `mpgrafic` with modifications to support Enzo data formats.

3.4.1 Using inits

The inits program uses one or more ASCII input files to set parameters, including the details of the power spectrum, the grid size, and output file names. Each line of the parameter file is interpreted independently and can contain only

a single parameter. Parameters are specified in the form:

```
ParameterName = VALUE
```

Spaces are ignored, and a parameter statement must be contained on a single line. Lines which begin with the pound symbol (#) are assumed to be comments and ignored.

First, set the parameters in the file. There are a large number of parameters, but many don't need to be set since reasonable default values are provided. Modifying a provided example (see *Sample inits and Enzo parameter files*) is probably the easiest route, but for reference there is a list of the parameters, their meanings, and their default values.

Generating a single grid initialization (for simple Enzo runs) is relatively straightforward. Generating a multi-grid initialization for Enzo is somewhat more complicated, and we only sketch the full procedure here.

Single Grid Initialization

To run a single grid initialization, you must set at least the following parameters: Rank, GridDims, ParticleDims, as well as the appropriate Cosmology and Power Spectrum parameters. A sample parameter file is available, which sets up a small, single grid cosmology simulation (that is, single grid for the initial conditions, once Enzo is used, additional grids will be created).

After creating or modifying a parameter file, and compiling inits, run the code with:

```
inits [-d] parameter_file
```

Where parameter_file is the name of your modified parameter file (the -d turns on a debug option). This will produce a number of HDF files containing the initial grids and particles, which are in the correct units for use in Enzo.

Multiple-grid Initialization

New in version 2.1. The multi-grid (or nested) initialization can be used to refine in a specific region, such as the Lagrangian sphere of a halo. We assume that you have first run a single-grid simulation and identified a region out of which a halo will form and can put this in the form of the left and right corners of a box which describes the region. Then you add the following parameters to the single-grid initialization code:

```
MaximumInitialRefinementLevel = 2
RefineRegionLeftEdge          = 0.15523 0.14551 0.30074
RefineRegionRightEdge         = 0.38523 0.37551 0.53074
NewCenterFloat                = 0.270230055 0.260508984 0.415739357
```

MaximumInitialRefinementLevel indicates how many extra levels you want to generate (in this case two additional levels, or 3 in total, including the root grid). The next two parameters (RefineRegionLeftEdge and RefineRegionRightEdge) describe the region to be refined. The fourth (optional) parameter re-centers the grid on the halo to be resimulated.

Once you have added these parameters, run inits once on the new parameter file. It will give you a progress report as it runs (note that if MaximumInitialRefinementLevel is large, this can take a long time), and generate all of the necessary files (e.g. GridDensity.0, GridDensity.1, etc.).

It will also generate a file called EnzoMultigridParameters which you can then copy directly into the enzo parameter file, and it specifies the positions of the new grids. You will still need to set a few other parameters in the enzo parameter file, including RefineRegionLeftEdge and RefineRegionRightEdge so that it only refines in the specified region (typically this should match the most refined initial grid). Also set the MaximumRefinementLevel parameter and the parameter controlling the density to be refined (MinimumOverDensityForRefinement – this also applies to the root grid, so it needs to be divided by 8^l where l is the value of MaximumInitialRefinementLevel).

Note that it is also possible to generate each level of initial conditions manually. This should not really be necessary, but a rough guideline is given here. To do this, prepare multiple parameter file describing the individual parameter regions, and then top grid can be generated with:

```
inits [-d] -s SubGridParameterFile TopGridParameterFile
```

The -s flag provides the name of the sub-grid parameter file, which is required by inits so that the particles are not replicated in the sub-grid region. The sub-grids are made with the usual command line:

```
inits [-d] SubGridParameterFile
```

!Subgrids with MaxDims of 512 or larger will take some time and require a fair amount of memory since the entire region is generated and then the desired section extracted.

Inits Parameter List

Cosmology Parameters

CosmologyOmegaMatterNow This is the contribution of all non-relativistic matter (including HDM) to the energy density at the current epoch (z=0), relative to the value required to marginally close the universe. It includes dark and baryonic matter. Default: 1.0

CosmologyOmegaLambdaNow This is the contribution of the cosmological constant to the energy density at the current epoch, in the same units as above. Default: 0.0

CosmologyOmegaWDMNow This is the contribution due to warm dark matter alone. Ignored unless PowerSpectrumType = 13 or 14. Default: 0.0

CosmologyOmegaHDMNow This is the contribution due to hot dark matter alone. Default: 0.0

CosmologyOmegaBaryonNow The baryonic contribution alone. Default: 0.06

CosmologyComovingBoxSize The size of the volume to be simulated in Mpc/h (at z=0). Default: 64.0

CosmologyHubbleConstantNow The Hubble constant at z=0, in units of 100 km/s/Mpc. Default: 0.5

CosmologyInitialRedshift The redshift for which the initial conditions are to be generated. Default: 20.0

Power Spectrum Parameters

PowerSpectrumType This integer parameter indicates the routine to be used for generating the power spectrum. Default: 1 The following are currently available:

- 1 - CDM approximation from BBKS (Bardeen et al 1986) as modified by Peacock and Dodds (1994), to include, very roughly, the effect of baryons. This should not be used for high baryon universes or for simulations in which precision in the PS is important.
- 2 - CHDM approximate PS from Ma (1996). Roughly good for hot fractions from 0.05 to 0.3.
- 3 - Power-law (scale-free) spectra.
- 4 - Reads in a power-spectrum from a file (not working).
- 5 - CHDM approximate PS from Ma (1996), modified for 2 equal mass neutrinos.
- 6 - A CDM-like Power spectrum with a shape parameter (Gamma), that is specified by the parameter PowerSpectrumGamma.
- 11 - The Eisenstein and Hu fitting functions for low and moderate baryon fraction, including the case of one massive neutrino.

- 12 - The Eisenstein and Hu fitting functions for low and moderate baryon fraction, for the case of two massive neutrinos.
- 13 - A Warm Dark Matter (WDM) power spectrum based on the formulae of Bode et al. (2001 ApJ 556, 93). The WDM equivalent of the Eisenstein & Hu fitting function with one massive neutrino (so a WDM version of #11).
- 14 - A Warm Dark Matter (WDM) power spectrum based on the formulae of Bode et al. (2001 ApJ 556, 93). The WDM equivalent of the CDM BBKS approximation of Bardeen et al 1986 (the WDM version of #1).
- 20 - A transfer function from CMBFast is input for this option, based on the filenames described below.

PowerSpectrumSigma8 The amplitude of the linear power spectrum at z=0 as specified by the rms amplitude of mass-fluctuations in a top-hat sphere of radius 8 Mpc/h. Default: 0.6

PowerSpectrumPrimordialIndex This is the index of the mass power spectrum before modification by the transfer function. A value of 1 corresponds to the scale-free primordial spectrum. Default: 1.0.

PowerSpectrumRandomSeed This is the initial seed for all random number generation, which should be negative. The random number generator (Numerical Recipes RAN3) is machine-independent, so the same seed will produce the same results (with other parameters unchanged). Note also that because the spectrum is sampled strictly in order of increasing k-amplitude, the large-scale power will be the same even if you increase or decrease the grid size. Default: -123456789

PowerSpectrumkcutoff The spectrum is set to zero above this wavenumber (i.e. smaller scales are set to zero), which is in units of 1/Mpc. It only works for power spectrum types 1-6. A value of 0 means no cutoff. Default: 0.0

PowerSpectrumkmin/kmax These two parameters control the range of the internal lookup table in wavenumber (units 1/Mpc). Reasonably sized grids will not require changes in these parameters. Defaults: kmin = 1e-3, kmax = 1e+4.

PowerSpectrumNumberOfkPoints This sets the number of points in the PS look-up table that is generated for efficiency purposes. It should not require changing. Default: 10000.

PowerSpectrumFileNameRedshiftZero For input power spectra, such as those from CMBFAST, two transfer functions are required: one at z=0 to fix the amplitude (via Sigma8) and the other at the initial redshift to give the shape and amplitude relative to z=0. No default.

PowerSpectrumFileNameInitialRedshift see above.

PowerSpectrumGamma The shape parameter (Omega*h); ignored unless PowerSpectrumType = 6.

PowerSpectrumWDMParticleMass The mass of the dark matter particle in KeV for the Bode et al. warm dark matter (WDM) case. Ignored unless PowerSpectrumType = 13 or 14. Default: 1.0.

PowerSpectrumWDMDegreesOfFreedom The number of degrees of freedom of the warm dark matter particles for the Bode et al. warm dark matter model. Ignored unless PowerSpectrumType = 13 or 14. Default: 1.5.

PowerSpectrumGamma The shape parameter (Omega*h); ignored unless PowerSpectrumType = 6.

Grid Parameters: Basic

Rank Dimensionality of the problem, 1 to 3 (warning: not recently tested for Rank !=2). Default: 3

GridDims This sets the actual dimensions of the baryon grid that is to be created (and so it may be smaller than MaxDims in some cases). Example: 64 64 64 No default.

ParticleDims Dimensions of the particle grid that is to be created. No default.

InitializeGrids Flag indicating if the baryon grids should be produced (set to 0 if inits is being run to generate particles only). Default: 1

InitializeParticles Flag indicating if the particles should be produced (set to 0 if inits is being run to generate baryons only). Default: 1

ParticlePositionName This is the name of the particle position output file. This HDF file contains one to three Scientific Data Sets (SDS), one for dimensional component. Default: ParticlePositions

ParticleVelocityName The particle velocity file name, which must(!) be different from the one above, otherwise the order of the SDS's will be incorrect. Default: ParticleVelocities

ParticleMassName This is the name of the particle mass file, which is generally not needed (enzo generates its own masses if not provided). Default: None

GridDensityName The name of the HDF file which contains the grid density SDS. Default: GridDensity

GridVelocityName The name of the HDF file which contains the SDS's for the baryonic velocity (may be the same as GridDensityName). Default: GridVelocity

Grid Parameters: Advanced

MaximumInitialRefinementLevel Used for multi-grid (nested) initial code generation. This parameter speciesi the level (0-based) that the initial conditions should be generated to. So, for example, setting it to 1 generates the top grid and one additional level of refinement. Note that the additional levels are nested, keeping at least one coarse cell between the edge of a coarse grid and its refined grid. Default: 0

RefineRegionLeftEdge, RefineRegionRightEdge Species the left and right corners of the region that should be refined using the AutomaticSubgridGeneration method (see above parameter). Default: 0 0 0 - 1 1 1

NewCenterFloat Indicates that the final grid should be recenter so that this point is the new center (0.5 0.5 0.5) of the grid.

MaxDims All dimensions are specified as one to three numbers deliminated by spaces (and for those familiar with the KRONOS or ZEUS method of specifying dimensions, the ones here do not include ghost zones). An example is: 64 64 64. MaxDims are the dimensions of the conceptual high-resolution grid that covers the entire computational domain. For a single-grid initialization this is just the dimension of the grid (or of the particle grid if there are more particles than grid points). For multi-grid initializations, this is the dimensions of the grid that would cover the region at the highest resolution that will be used. It must be identical across all parameter files (for multi-grid initializations). The default is the maximum of GridDims or ParticleDims, whichever is larger (in other words unless you are using a multi-grid initialization, this parameter does not need to be set). Confused yet?

GridRefinement This integer is the sampling, for the baryon grid, in each dimension, relative to MaxDims. For single-grid initializations, this is generally 1. For multi-grids, it is the refinement factor relative to the finest level. In other words, if the grid covered the entire computational region, then each value in MaxDims would equal GridDims times the GridRefinement factor. Default: 1

ParticleRefinement Similar function as above, but for the particles. Note that it can also be used to generate fewer particles than grids (i.e. the GridRefinement and ParticleRefinement factors do not have to be the same). Default: 1

StartIndex For single-grid initializations, this should be the zero vector. For multi-grid initializations it specifies the index (a triplet of integers in 3D) of the left-hand corner of the grid to be generated. It is specified in terms of the finest conceptual grid and so ranges from 0 to MaxDims-1. Note also that for AMR, the start and end of a sub-grid must lie on the cell-boundary of it's parent. That means that this number must be divisible by the Refinement factor. The end of the sub-grid will be at index: StartIndex + GridRefinement*GridDims. The co-ordinate system used by this parameter is always the unshifted one (i.e. it does not change if NewCenter is set).

3.4.2 Using mpgrafic

New in version 2.0. This version of mpgrafic is a modified version of the public version of mpgrafic, found at

http://www2.iap.fr/users/pichon/mpgrafic.html

to produce files readable by Enzo. It has been modified to write HDF5 files in parallel.

Dependencies

- HDF5 with parallel and FORTRAN support (flags –enable-parallel –enable-fortran)
- FFTW v2 with MPI support and different single and double precision versions. It must be compiled once for single precision and another time for double precision. For the former, use the flags –enable-mpi –enable-type-prefix –enable-float. For double precision, use –enable-mpi –enable-type-prefix.

Approach

Non-nested initial conditions are created only using mpgrafic. However if the user wants nested initial conditions, a full-resolution grid (e.g. 256^3 grid for a 64^3 top grid with 2 nested grids) must be created first and then post-processed with degraf to create a degraded top-level grid and cropped (and degraded if not the finest level) grids for the nested grids.

As with the original inits Enzo package, the baryon density and velocities are written in a 3 dimensional array. The original inits writes the particle data in 1-d arrays. In mpgrafic, only the particle velocities are written in a 3-d array. Enzo has been modified to create the particle positions from the Zel'dovich approximation from these velocities, so it is not needed to write the positions anymore. Also it does not create particles that are represented by a finer grid at the same position.

One big benefit of writing the particle velocities in a 3-d array is avoiding the use of the RingIO tool because each processor knows which subvolume to read within the velocity data.

As of HDF5 version 1.8.2, there exists a bug that creates corrupted datasets when writing very large (e.g. $>2048^3$) datasets with multiple components (4-d arrays). The HDF5 I/O in mpgrafic works around this bug by creating one file per velocity component for both the baryons and particles.

How to run

First the user needs to compile both mpgrafic and degraf. The configure / make systems are set up similarly.

Configure flags:

--enable-enzo	turns on I/O for Enzo
--enable-double	creates files in double precision
--enable-onedim	creates one file per velocity component
--with-hdf=HDF5_DIR	sets directory for parallel HDF5

If FFTW is not present in the user's library path, the following variables must be also set

```
CFLAGS="-I ${FFTW_DIR}/include"
FCFLAGS="-I ${FFTW_DIR}/include"
LDFLAGS="-L ${FFTW_DIR}/lib"
```

To run in parallel, you can use FC=mpif90 and LD=h5pfc, which the compiler wrapper for parallel HDF5.

Example configure (for Mac OSX):

```
./configure LD="-bind_at_load" FC=mpif90 CC=mpicc --enable-enzo \
--enable-double --enable-onedim --with-hdf=/usr/local/hdf5/1.8.2p
```

Example configure scripts can be found in mpgrafic/mpgrafic-0.2/conf.*. After a successful configure, you can make mpgrafic or degraf by typing 'make'.

After the programs are compiled, you make the initial conditions by using a python script, make_ic.py, in the top directory that simplifies the user input into mpgrafic and degraf and the moving of files.

make_ic.py parameters

nprocs number of processors

boxsize box size in comoving Mpc (not Mpc/h)

resolution top-level grid resolution

n_levels level of the finest nested grid

inner_width width of the finest nested grid

buffer_cells number of cells separating nested grids

seed random seed (must be 9 digits)

name name of the data directory (saved in mpgrafic/data/name/)

center how much to shift the data in order to center on a particular region.

LargeScaleCorrection whether to use a noise file from a lower-resolution run

LargeScaleFile noise file from that lower-resolution run

OneDimPerFile whether we're using one file per velocity component

omega_m Omega matter

omega_v Omega lambda

omega_b Omega baryon

h0 Hubble constant in units of [km/s/Mpc]

sigma8 sigma_8

n_plawslope slope of power spectrum

After you set your parameters, run this script with

```
python make_ic.py
```

and it will re-compile mpgrafic and (for nested grids) degraf. Then it will run mpgrafic for the full-resolution box. If the user wants nested grids, it will copy the data files to mpgrafic/degraf and create the set of nested grid files.

The user cannot specify the initial redshift because mpgrafic determines it from the parameter sigstart that is the maximum initial density fluctuation. From this, mpgrafic calculates the initial redshift. This file is overwritten by the python script, so if you want to change this parameter, change it in the python script (routine write_grafic1inc).

The noise file is always kept in mpgrafic/mpgrafic-0.2/src and is named $seed_$resolution.dat, where $resolution is the top-level grid resolution. It can be re-used with LargeScaleFile if the user wants to re-simulate the volume at a higher resolution.

The data files are moved to mpgrafic/data/$name. If nested grids were created, degraf writes a set of parameters in enzo.params for copy-pasting into an Enzo parameter file. Now you can move the files to the simulation directory and start your Enzo cosmology simulation!

3.5 Running Large Simulations

Here we describe how to efficiently run a large simulation on a high number of processors, such as particular parameters to set and suggested number of MPI tasks for a given problem size. For a problem to be scalable, most of the code must be parallel to achieve high performance numbers on large MPI process counts (see Amdahl's Law). In general, the user wants to pick the number of processors so that computation is still dominant over communication time. If the processor count is too high, communication time will become too large and might even *slow* down the simulation!

For picking the number of processors for an Enzo run, a good starting point is putting a 64^3 box on each processor for both AMR and unigrid setups. For example, a 256^3 simulation would run well on $(256/64)^3 = 64$ processors. For nested grid simulations, the outer boxes usually require little computation compared to the "zoom-in" region, so the processor count should be based on the inner-most nested grid size. The user can experiment with increasing the processor count from this suggestion, but strong scaling (i.e. linear speedup with processor count) is not to be expected. Little performance gains (as of v2.0) can be expected beyond assigning a 32^3 cube per processor.

Note: The level-0 grid is only partitioned during the problem initialization. It will *never* be re-partitioned if the user restarts with a different number of processors. However, some performance gains can be expected even if a processor does not contain a level-0 grid because of the work on finer levels.

3.5.1 Important Parameters

- `LoadBalancing`: Default is 1, which moves work from overloaded to underutilized processes, regardless of the grid position. **New for v2.1**: In some cases but not always, speedups can be found in load balancing on a space filling curve (`LoadBalancing = 4`). Here the grids on each processor will be continuous on the space filling curve. This results in a grouped set of grids, requiring less communication from other processors (and even other compute nodes).
- `SubgridSizeAutoAdjust` and `OptimalSubgridsPerProcessor`: **New for v2.1** Default is ON and 16, respectively. The maximum subgrid size and edge length will be dynamically adjusted on each AMR level according to the number of cells on the level and number of processors. The basic idea behind increasing the subgrid sizes (i.e. coalescing grids) is to reduce communication between grids.
- `MinimumSubgridEdge` and `MaximumSubgridSize`: *Unused if SubgridAutoAdjust is ON*. Increase both of these parameters to increase the average subgrid size, which might reduce communication and speedup the simulation.
- `UnigridTranspose`: Default is 0, which is employs blocking MPI communication to transpose the root grid before and after the FFT. In level-0 grids 1024^3, this becomes the most expense part of the calculation. In these types of large runs, Option 2 is recommended, which uses non-blocking MPI calls; however it has some additional memory overhead, which is the reason it is not used by default.

3.5.2 Compile-time options

- `max-subgrids`: If the number of subgrids in a single AMR level exceeds this value, then the simulation will crash. Increase as necessary. Default: 100,000
- `ooc-boundary-yes`: Stores the boundary conditions out of core, i.e. on disk. Otherwise, each processor contains a complete copy of the external boundary conditions. This becomes useful in runs with large level-0 grids. For instance in a 1024^3 simulation with 16 baryon fields, each processor will contain a set of boundary conditions on 6 faces of 1024^2 with 16 baryon fields. In single precision, this requires 402MB! Default: OFF
- `fastsib-yes`: Uses a chaining mesh to help locate sibling grids when constructing the boundary conditions. Default: ON

3.6 Enzo Output Formats

Although there are a number of ways of specifying when (and how often) Enzo outputs information, there is only one type of output 'dump' (well, not quite – there are now movie dumps, see below), which can also be used to restart the simulation. The output format uses the following files, each of which begins with the output name, here we use the example base_name, and are then followed by the output number, ranging from 0000 to 9999 (if more than 10000 grids are generated then the number goes to 10000, etc.). When restarting, or other times when an output filename needs to be specified, use the name without any extension (e.g. `enzo -r base_name0000`).

3.6.1 Summary of Files

base_name0000 This ascii file contains a complete listing of all the parameter settings, both those specified in the initial parameter file, as well as all those for which default values were assumed. The parameters (see *Enzo Parameter List*) are in the same format as that used in the input file: parameter_name = value. This file is modifiable if you would like to restart from a certain point with different parameter values.

base_name0000.hierarchy This ascii file specifies the hierarchy structure as well as the names of the grid files, their sizes, and what they contain. It should not be modified.

base_name0000.cpu00001 The field information for each cpu (padded with zeros) is contained in separate files with a root 'Node' for each grid, padded with zeros to be eight digits. The format is the Hierarchy Data Format (HDF) version 5, a self-describing machine-independent data format developed and supported by the National Center for Supercomputing Applications (NCSA). More information can be found on their home page. Most scientific visualization packages support this format. Each field is stored as it's own one-, two- or three-dimensional Scientific Data Set (SDS), and is named for identification. Particles (if any) are included with a set of one-dimensional datasets under the top 'grid' node.

base_name0000.boundary An ascii file which specifies boundary information. It is not generally useful to modify.

base_name0000.boundary.hdf Contains field-specific boundary information, in HDF format.

base_name0000.radiation This ascii file is only generated if using the self-consistent radiation field.

3.6.2 Output Units

The units of the physical quantities in the grid SDS's are depend on the problem being run. For most test problems there is no physical length or time specified, so they can be be simply scaled. For cosmology there are a set of units designed to make most quantities of order unity (so single precision variables can be used). These units are defined below (rho0 = 3*OmegaMatterNow*(100*HubbleConstantNow km/s/Mpc)^2/(8*Pi*G)).

- length: ComovingBoxSize/HubbleConstantNow * Mpc / (1+z)
- density: rho0 * (1+z)^3
- time: 1/sqrt(4*Pi*G*rho0*(1+InitialRedshift)^3)
- temperature: K
- velocity: (length/time)*(1+z)/(1+InitialRedshift) (this is z independent)

The conversion factor is also given in the ascii output file (base_name0000): search for `DataCGSConversionFactor`. Each field has its own conversation factor, which converts that field to cgs units. Users can also set completely arbitrary internal units, as long as they are self-consistent: to see how to do this, go to *Enzo Internal Unit System*.

3.6.3 Streaming Data Format

Purpose: To provide data on every N-th timestep of each AMR level.

Method

We keep track of the elapsed timesteps on every AMR level. Every N-th timestep on a particular level L, all grids on levels >= L are written for the baryon fields (specified by the user in `MovieDataField`) and particles. The integers in `MovieDataField` correspond to the field element in `BaryonField`, i.e. 0 = Density, 7 = HII density. Temperature has a special value of 1000.

See *Streaming Data Format* for a full description of the streaming data format parameters.

File format

All files are written in HDF5 with one file per processor per top-level timestep. The filename is named AmiraDataXXXX_PYYY.hdf5 where XXXX is the file counter, which should equal the cycle number, and YYY is the processor number. Each file has a header indicating

- whether the data are cell-centered (1) or vertex-centered (0) [int]
- number of baryon fields written [int]
- number of particle fields written [int]
- field names with the baryon fields first, followed by the particle fields [array of variable-length strings]

The group names (grid-%d) are unique only in the file. Unique grids are identified by their timestep number attribute and position. Each grid has the following attributes:

- AMR level [int]
- Timestep [int]
- Code time [double]
- Redshift [double]
- Ghost zones flag for each grid face [6 x int]
- Number of ghost zones in each dimension [3 x int]
- Cell width [3 x double]
- Grid origin in code units [3 x double]
- Grid origin in units of cell widths [3 x long long]

In addition to the HDF5 files, a binary index file is created for fast I/O in post-processing. The filenames of the these files are the same as the main data files but with the extension .idx. The header consists of

- pi (to indicate endianness) [float]
- cell width on the top level [float]
- number of fields [char]
- cell-centered (1) or vertex-centered (0) [char]
- field names [number of fields x (64 char)]

For every grid written, an index entry is created with

- grid ID [int]

- code time [double]
- timestep [int]
- redshift [double]
- level [char]
- grid origin in units of cell widths [long long]
- grid dimensions [short]
- number of particles [int]

Lastly, we output an ASCII file with the code times and redshifts of every top level timestep for convenience when choosing files to read afterwards.

3.7 Analyzing With YT

3.7.1 What is YT?

YT is a python-based tool designed for analyzing and visualizing Adaptive Mesh Refinement data, specifically as output from Enzo. YT is completely free and open source, with an active and expanding development community, and it presents to the user both high-level and low-level APIs. The documentation contains a tutorial as well as an API reference, but here we will step through some simple steps toward creating script to make simple plots of a cosmological simulation.

This brief tutorial presupposes that you have run the installation script and are comfortable launching python. (The install script will tell you how!) It's also encouraged to launch the special YT-enhanced IPython shell via the command `iyt`, which (thanks to IPython!) features filesystem navigation and tab completion, along with interactive plotting capabilities.

3.7.2 Making Slices

Here is a sample script that will make a set of slices centered on the maximum density location, with a width of 100 kpc.

```
from yt.mods import *
pf = EnzoStaticOutput("RedshiftOutput0035.dir/RedshiftOutput0035")

pc = raven.PlotCollection(pf)
pc.add_slice("Density",0)
pc.add_slice("Density",1)
pc.add_slice("Density",2)
pc.set_width(100.0,'kpc')
pc.save("z35_100kpc")
```

If you put this into a file called `my_script.py`, you can execute it with `python2.5 my_script.py` and it will save out a set of images prefixed with `z35_100kpc` in PNG format.

3.7.3 Making Simple Radial Profiles

If you want to make radial profiles, you can generate and plot them very easily with YT. Here is a sample script to do so.

```
from yt.mods import *
pf = EnzoStaticOutput("RedshiftOutput0035.dir/RedshiftOutput0035")

pc = PlotCollection(pf)

pc.add_profile_sphere(100.0, 'kpc', ["Density", "Temperature"])
pc.save("z35_100kpc")

pc.switch_z("VelocityMagnitude")
pc.save("z35_100kpc")
```

To show the mass distribution in the Density-Temperature plane, we would make a phase diagram.

```
from yt.mods import *
pf = EnzoStaticOutput("RedshiftOutput0035.dir/RedshiftOutput0035")

pc = PlotCollection(pf)

pc.add_phase_sphere(100.0, 'kpc', ["Density", "Temperature", "CellMassMsun"], weight=None)
pc.save("z35_100kpc")
```

3.7.4 More Information

For more information on yt, see the yt website, where you will find mailing lists, documentation, API documentation, a cookbook and even a gallery of images.

3.8 Simulation Names and Identifiers

To help track and identify simulations and datasets, a few new lines have been added to the parameter file:

MetaDataIdentifier short string persisted across datasets

MetaDataSimulationUUID uuid persisted across datasets

MetaDataDatasetUUID unique dataset uuid

MetaDataRestartDatasetUUID input dataset uuid

MetaDataInitialConditionsUUID initial conditions uuid

The parameters stored during a run are members of the TopGridData struct.

3.8.1 MetaDataIdentifier

This is a character string without spaces (specifically, something that can be picked by "%s"), that can be defined in a parameter file, and will be written out in every following output. It's intended to be a human-friendly way of tracking datasets. For example

Example:

```
MetaDataIdentifier = Cosmology512_Mpc_run4
```

3.8.2 MetaDataSimulationUUID

The MetaDataSimulationUUID is a globally unique identifier for a collection of datasets. Universally Unique Identifiers (UUIDs) are opaque identifiers using random 128-bit numbers, with an extremely low chance of collision. Therefore, they are very useful when trying to label data coming from multiple remote resources (say, computers distributed around the world).

Example:

```
MetaDataSimulationUUID = e5f72b77-5258-45ba-a376-ffe11907fae1
```

Like the `MetaDataIdentifier`, the `MetaDataSimulationUUID` is read in at the beginning of a run, and then re-written with each output. However, if one is not found initially, a new one will be generated, using code from the ooid library included in Enzo.

UUIDs can be generated with a variety of tools, including the python standard library.

3.8.3 MetaDataDatasetUUID

A MetaDataDatasetUUID is created at each output.

Example:

```
MetaDataDatasetUUID = b9d78cc7-2ecf-4d66-a23c-a1dcd40e7955
```

MetaDataRestartDatasetUUID

While reading the parameter file, if a MetaDataDatasetUUID line is found, it is stored, and re-written as MetaDataRestartDatasetUUID. The intention of this is help track datasets across restarts and parameter tweaks.

Example:

```
MetaDataRestartDatasetUUID = b9d78cc7-2ecf-4d66-a23c-a1dcd40e7955
```

3.8.4 MetaDataInitialConditionsUUID

This is similar to `MetaDataRestartDatasetUUID`, except it's intended for tracking which initial conditions were used for a simulation.

Example:

```
MetaDataInitialConditionsUUID    = 99f71bdf-e56d-4daf-88f6-1ecd988cbc9f
```

3.8.5 Still to be done

- Add UUID generation to `inits` store it in the HDF5 output.
- Preserve the UUID when using `ring`.
- Have Enzo check for the UUID in both cases.

3.9 Embedded Python

Python can now be embedded inside Enzo, for inline analysis as well as interaction. This comes with several shortcomings, but some compelling strong points.

3.9.1 How To Compile

The configure option that controls compilation of the Python code can be toggled with

```
make python-yes
```

or to turn it off,

```
make python-no
```

This will look for the following variables in the machine-specific Makefile:

```
MACH_INCLUDES_PYTHON
MACH_LIBS_PYTHON
```

for an example of how to define these variables, see Make.mach.orange in the source repository.

3.9.2 How it Works

On Enzo startup, the Python interface will be initialized. This constitutes the creation of an interpreter within the memory-space of each Enzo process, as well as import and construct the NumPy function table. Several Enzo-global data objects for storing grid parameters and simulation parameters will be initialized and the Enzo module will be created and filled with those data objects.

Once the Python interface and interpreter have finished initializing, the module user_script will be imported – typically this means that a script named `user_script.py` in the current directory will be imported, but it will search the entire import path as well. Every `PythonSubcycleSkip` subcycles, at the bottom of the hierarchy in `EvolveLevel.C` the entire grid hierarchy and the current set of parameters will be exported to the Enzo module and then user_script.main() will be called.

3.9.3 How to Run

By constructing a script inside `user_script.py`, the Enzo hierarchy can be accessed and modified. The analysis toolkit yt has functionality that can abstract much of the data-access and handling. Currently several different plotting methods – profiles, phase plots, slices and cutting planes – along with all derived quantities can be accessed and calculated. Projections cannot yet be made, but halo finding can be performed with Parallel HOP only. The following script is an example of a script that will save a slice as well as print some information about the simulation. Note that, other than the instantiation of `lagos.EnzoStaticOutputInMemory`, this script is identical to one that would be run on an output located on disk.

Recipes and convenience functions are being created to make every aspect of this simpler.

```
from yt.mods import *

def main():
     pf = lagos.EnzoStaticOutputInMemory()
     pc = PlotCollection(pf)
     pc.add_slice("Density", 0)
     pc.save("%s" % pf)
     v, c = pf.h.find_max("Density")
     sp = pf.h.sphere(c, 1.0/pf['mpc'])
     totals = sp.quantities["TotalQuantity"](["CellMassMsun","Ones"], lazy_reader=True)
     print "Total mass within 1 mpc: %0.3e total cells: %0.3e" % (totals[0], totals[1])
```

3.9.4 Which Operations Work

The following operations in yt work:

- Derived quantities
- Slices
- Cutting planes
- Fixed Resolution Projections (i.e., non-adaptive)
- 1-, 2-, 3-D Profiles

This should enable substantial analysis to be conducted in-line. Unfortunate adaptive projections require a domain decomposition as they currently stand (as of yt-1.7) but this will be eliminated with a quad-tree projection method slated to come online in yt-2.0. In future versions of yt the volume rendering approach will be parallelized using kD-tree decomposition and it will also become available for inline processing.

Please drop a line to the yt or Enzo mailing lists for help with any of this!

3.9.5 Things Not Yet Done

- Adaptive Projections do not work.
- Particles are not yet exported correctly
- Speed could be improved, but should be extremely efficient for a small number of grids. Future versions will utilize intercommunicators in MPI to allow for asynchronous analysis.

3.10 The Enzo Hierarchy File - Explanation and Usage

The Enzo Hierarchy file is a representation of the internal memory state of the entire hierarchy of grids. As such, its format – while somewhat obtuse at first – reflects that context. Each grid entry has a set number of fields that describe its position in space, as well as the fields that are affiliated with that grid:

Note: We are in the process of transitioning to an HDF5-formatted Hierarchy File.

```
Grid = 1
Task              = 4
GridRank          = 3
GridDimension     = 38 22 22
GridStartIndex    = 3 3 3
GridEndIndex      = 34 18 18
GridLeftEdge      = 0 0 0
GridRightEdge     = 1 0.5 0.5
Time              = 646.75066015177
SubgridsAreStatic = 0
NumberOfBaryonFields = 8
FieldType = 0 1 4 5 6 19 20 21
BaryonFileName = ./RD0005/RedshiftOutput0005.cpu0000
CourantSafetyNumber    = 0.300000
PPMFlatteningParameter = 0
PPMDiffusionParameter  = 0
PPMSteepeningParameter = 0
NumberOfParticles   = 20
ParticleFileName = ./RD0005/RedshiftOutput0005.cpu0000
```

```
GravityBoundaryType = 0
Pointer: Grid[1]->NextGridThisLevel = 2
```

The final field, starting with "Pointer", is slightly more complicated and will be discussed below.

`Grid = 1`

This is the ID of the grid. Enzo grids are indexed internally starting at 1.

`Task = 3`

This grid was written by processor 3 and will be read in by it if restarting more than 4 processors.

`GridRank = 3`

This is the dimensionality of the grid.

`GridDimension = 38 22 22`

Dimensions, *including* ghost zones.

`GridStartIndex = 3 3 3`

The first index of data values *owned* by this grid.

`GridEndIndex = 34 18 18`

The final index *owned* by this grid. The active zones have dimensionality of GridEndIndex - GridStartIndex + 1.

`GridLeftEdge = 0 0 0`

In code units, between `DomainLeftEdge` and `DomainRightEdge`, the origin of this grid.

`GridRightEdge = 1 0.5 0.5`

In code units, between `DomainLeftEdge` and `DomainRightEdge`, the right-edge of this grid. `dx = (GridRightEdge - GridLeftEdge)/(GridEndIndex - GridStartIndex + 1).`

`Time = 646.75066015177`

The current time to which the baryon values in this grid have been evolved.

`SubgridsAreStatic = 0`

Whether refinement can occur in the subgrids.

`NumberOfBaryonFields = 8`

The number of data fields associated with this grid.

`FieldType = 0 1 4 5 6 19 20 21`

The integer identifiers of each field, in order, inside this grid.

`BaryonFileName = ./RD0005/RedshiftOutput0005.cpu0000`

The HDF5 file in which the baryons fields are stored.

`CourantSafetyNumber = 0.300000`

Courant safety number for this grid (governs timestepping.)

`PPMFlatteningParameter = 0`

Flattening parameter for this grid (governs PPM hydro.)

`PPMDiffusionParameter = 0`

Diffusion parameter for this grid (governs PPM hydro.)

`PPMSteepeningParameter = 0`

Steepening parameter for this grid (governs PPM hydro.)

`NumberOfParticles = 20`

How many particles are located in this grid at this timestep.

`ParticleFileName = ./RD0005/RedshiftOutput0005.cpu0000`

The HDF5 file in which the baryon fields and particle data are stored. This field will not exist if there aren't any particles in the grid.

`GravityBoundaryType = 0`

Boundary type inside gravity solver.

3.10.1 HDF5-formatted Hierarchy File

We are transitioning to an HDF5-formatted hierarchy file. This is an improvement because reading a large (many thousand grid) ASCII hierarchy file take a long time. [Other improvements?]

The structure of the file:

Although HDF5 tools like 'h5ls' and 'h5dump' can be used to explore the structure of the file, it's probably easiest to use python and h5py. This is how to open an example hierarchy file (from run/Cosmology/Hydro/AMRCosmologySimulation) in python.

```
>>> import h5py
>>> f = h5py.File('RD0007/RedshiftOutput0007.hierarchy.hdf5','r')
```

The root group ('/') contains a number of attributes.

```
>>> f.attrs.keys()
['Redshift', 'NumberOfProcessors', 'TotalNumberOfGrids']
>>> f.attrs['Redshift']
0.0
>>> f.attrs['NumberOfProcessors']
1
>>> f.attrs['TotalNumberOfGrids']
44
```

So we see that this is a z=0 output from a simulation run on a single core and it contains a total of 44 grids.

Now let's look at the groups contained in this file.

```
>>> f.keys()
['Level0', 'Level1', 'Level2', 'LevelLookupTable']
```

The simulation has two levels of refinement, so there are a total of three HDF5 groups that contain information about the grids at each level. Additionally, there is one more dataset ('LevelLookupTable') that is useful for finding which level a given grid belongs to. Let's have a closer look.

```
>>> level_lookup = f['LevelLookupTable']
>>> level_lookup.shape
(44,)
>>> level_lookup[:]
array([0, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

This shows you that the first grid is on level 0, the second on level 1, and all the remaining grids on level 2. Let's have a look at the 'Level2' group.

```
 >>> g = f['Level2']
 >>> g.keys()
['Grid00000003', 'Grid00000004', 'Grid00000005', ..., 'Grid00000043', 'Grid00000044']
```

Each level group also has one attribute, 'NumberOfGrids'.

```
>>> g.attrs['NumberOfGrids']
42
```

The hierarchy information about each of the grids is stored as both attributes and datasets.

```
>>> grid = g['Grid00000003']
>>> grid.attrs.keys()
['Task', 'GridRank', 'Time', 'OldTime', 'SubgridsAreStatic', 'NumberOfBaryonFields', 'FieldType',
 'BaryonFileName', 'CourantSafetyNumber', 'PPMFlatteningParameter', 'PPMDiffusionParameter',
 'PPMSteepeningParameter', 'ParticleFileName', 'GravityBoundaryType', 'NumberOfDaughterGrids',
 'NextGridThisLevelID', 'NextGridNextLevelID']
>>> grid.keys()
['GridDimension', 'GridEndIndex', 'GridGlobalPosition',
 'GridLeftEdge', 'GridRightEdge', 'GridStartIndex', 'NumberOfParticles']
```

Besides the parameters that have been described above, there are few new elements:

`GridGlobalPosition` is LeftGridEdge[] expressed in integer indices of this level, i.e. running from 0 to Root-GridDimension[] * RefinementFactors[]**level - 1. This may be useful for re-calculating positions in long double precision (which is not universally supported by HDF5) at runtime.

`NumberOfDaughterGrids` gives you the number of daughter grids.

`DaughterGrids` is a group that contains HDF5-internal soft links to the daugher datasets. Example:

```
>>> daughters = grid['DaughterGrids']
>>> daughters.keys()
['DaughterGrid0000', 'DaughterGrid0001', 'DaughterGrid0002', ..., 'DaughterGrid0041']
>>> daughters.get('DaughterGrid0000', getlink=True)
<SoftLink to "/Level2/Grid00000003">
```

In this case there are 42 daughter grids.

`ParentGrids` is a group that contains HDF5-internal soft links to parent grids on all levels above the present grid's level. Example for a level 2 grid:

```
>>> grid = f['Level2']['Grid00000044']
>>> parents = grid['ParentGrids']
>>> parents.keys()
['ParentGrid_Level0', 'ParentGrid_Level1']
>>> parents.get('ParentGrid_Level0', getlink=True)
<SoftLink to "/Level0/Grid00000001">
```

Lastly, there's one additional (experimental) feature that is available only if you've compiled with verson 1.8+ of HDF5. In that case you can set '#define HAVE_HDF5_18' in Grid_WriteHierarchyInformationHDF5.C [perhaps this should become a Makefile configuration option?], and then there will be an external HDF5 link to the HDF5 file containing the actual data for that grid. Example:

```
>>> grid.get('GridData', getlink=True)
>>> <ExternalLink to "Grid00000002" in file "./RD0007/RedshiftOutput0007.cpu0000"
```

3.10.2 Controlling the Hierarchy File Output Format

There are two new parameters governing the format of the hierarchy format:

`[OutputControl.]HierarchyFileInputFormat = 0, 1`

This specifies the format of the hierarchy file to be read in: 0 = ASCII, 1 = HDF5. Default set to 0 for now, but will change to 1 in the future.

`[OutputControl.]HierarchyFileOutputFormat = 0, 1, 2` [OutputControl.HierarchyFileOutputFormat in new-config]

This specifies the format of the hierarchy file to be written out: 0 = ASCII, 1 = HDF5, 2 = both. Default set to 2 for now, but will change to 1 in the future.

3.11 Enzo Flow Chart, Source Browser

Here's a cartoon of Enzo. This was written as a first look as the details of how enzo works. Black arrows indicate further flow charts. Grey boxes (usually) indicate direct links to the source code.

No guarantees are made regarding the correctness of this flowchart – it's meant to help get a basic understanding of the flow of Enzo before extensive code modifications. Also see the Enzo Source Browser. This is a second attempt at the same thing in a more dynamic way. It allows one to (in principle) see all the routines called from a function, in order, and jump to the source showing the call. It also allows you to see a reverse call stack of every routine that calls a particular function. It runs relatively well on the newer versions of Firefox. Older browsers have a harder time with the Javascript. This is also somewhat buggy, for a host of reasons.

3.11.1 Flow Chart Errors

As mentioned above, this was written with an old version of the code and has not been made perfect. Issues may remain.

Return CPU Time

This used to be an Enzo routine. Now, it's an MPI call, MPI_Wtime

CHAPTER

FOUR

ENZO PARAMETER LIST

The following is a largely complete list of the parameters that Enzo understands, and a brief description of what they mean. They are grouped roughly by meaning; an alphabetical list is also available. Parameters for individual test problems are also listed here.

This parameter list has two purposes. The first is to describe and explain the parameters that can be put into the initial parameter file that begins a run. The second is to provide a comprehensive list of all parameters that the code uses, including those that go into an output file (which contains a complete list of all parameters), so that users can better understand these output files.

The parameters fall into a number of categories:

external These are user parameters in the sense that they can be set in the parameter file, and provide the primary means of communication between Enzo and the user.

internal These are mostly not set in the parameter file (although strictly speaking they can be) and are generally used for program to communicate with itself (via the restart of output files).

obsolete No longer used.

reserved To be used later.

Generally the external parameters are the only ones that are modified or set, but the internal parameters can provide useful information and can sometimes be modified so I list them here as well. Some parameters are true/false or on/off boolean flags. Eventually, these may be parsed, but in the meantime, we use the common convention of 0 meaning false or off and 1 for true or on.

This list includes parameters for the Enzo 2.0 release.

4.1 Stopping Parameters

`StopTime` (external) This parameter specifies the time (in code units) when the calculation will halt. For cosmology simulations, this variable is automatically set by `CosmologyFinalRedshift`. *No default.*

`StopCycle` (external) The cycle (top grid timestep) at which the calculation stops. A value of zero indicates that this criterion is not be used. *Default: 100,000*

`StopFirstTimeAtLevel` (external) Causes the simulation to immediately stop when a specified level is reached. Default value 0 (off), possible values are levels 1 through maximum number of levels in a given simulation.

`NumberOfOutputsBeforeExit` (external) After this many datadumps have been written, the code will exit. If set to 0 (default), this option will not be used. Default: 0.

`StopCPUTime` (external) Causes the simulation to stop if the wall time exceeds `StopCPUTime`. The simulation will output if the wall time after the next top-level timestep will exceed `StopCPUTime`, assuming that the wall

time elapsed during a top-level timestep the same as the previous timestep. In units of seconds. Default: 2.592e6 (30 days)

`ResubmitOn` (external) If set to 1, the simulation will stop if the wall time will exceed `StopCPUTime` within the next top-level timestep and run a shell script defined in `ResubmitCommand` that should resubmit the job for the user. Default: 0.

`ResubmitCommand` (external) Filename of a shell script that creates a queuing (e.g. PBS) script from two arguments, the number of processors and parameter file. This script is run by the root processor when stopping with `ResubmitOn`. An example script can be found in input/resubmit.sh. Default: (null)

4.2 Initialization Parameters

`ProblemType` (external) This integer specifies the type of problem to be run. Its value causes the correct problem initializer to be called to set up the grid, and also may trigger certain boundary conditions or other problem-dependent routines to be called. The possible values are listed below. Default: none.

For other problem-specific parameters follow the links below. The problems marked with "hydro_rk" originate from the MUSCL solver package in the enzo installation directory `src/enzo/hydro_rk`. For the 4xx radiation hydrodynamics problem types, see the user guides in the installation directory `doc/implicit_fld` and `doc/split_fld`.

Problem Type	Description and Parameter List
1	*Shock Tube (1: unigrid and AMR)*
2	*Wave Pool (2)*
3	*Shock Pool (3: unigrid 2D, AMR 2D and unigrid 3D)*
4	*Double Mach Reflection (4)*
5	*Shock in a Box (5)*
6	Implosion
7	SedovBlast
8	KH Instability
9	2D/3D Noh Problem
10	*Rotating Cylinder (10)*
11	*Radiating Shock (11)*
12	*Free Expansion (12)*
20	*Zeldovich Pancake (20)*
21	*Pressureless Collapse (21)*
22	*Adiabatic Expansion (22)*
23	*Test Gravity (23)*
24	*Spherical Infall (24)*
25	*Test Gravity: Sphere (25)*
26	*Gravity Equilibrium Test (26)*
27	*Collapse Test (27)*
28	TestGravityMotion
29	TestOrbit
30	*Cosmology Simulation (30)*
31	*Isolated Galaxy Evolution (31)*
35	*Shearing Box Simulation (35)*
40	*Supernova Restart Simulation (40)*
50	*Photon Test (50)*
60	Turbulence Simulation
61	Protostellar Collapse
62	*Cooling Test (62)*

Continued on next page

Table 4.1 – continued from previous page

Problem Type	Description and Parameter List
101	3D Collapse Test (hydro_rk)
102	1D Spherical Collapse Test (hydro_rk)
106	Hydro and MHD Turbulence Simulation (hydro_rk)
107	Put Sink from restart
200	1D MHD Test
201	2D MHD Test
202	3D MHD Collapse Test
203	MHD Turbulent Collapse Test
207	Galaxy disk
208	AGN disk
300	Poisson solver test
400	Radiation-Hydrodynamics test 1 – constant fields
401	Radiation-Hydrodynamics test 2 – stream test
402	Radiation-Hydrodynamics test 3 – pulse test
403	Radiation-Hydrodynamics test 4 – grey Marshak test
404/405	Radiation-Hydrodynamics test 5 – radiating shock test
410/411	Radiation-Hydrodynamics test 10/11 – Static HI ionization
412	Radiation-Hydrodynamics test 12 – HI ionization of a clump
413	Radiation-Hydrodynamics test 13 – HI ionization of a steep region
414/415	Radiation-Hydrodynamics test 14/15 – Cosmological HI ionization
450-452	Free-streaming radiation tests

`TopGridRank` (external) This specifies the dimensionality of the root grid and by extension the entire hierarchy. It should be 1,2 or 3. Default: none

`TopGridDimensions` (external) This is the dimension of the top or root grid. It should consist of 1, 2 or 3 integers separated by spaces. For those familiar with the KRONOS or ZEUS method of specifying dimensions, these values do not include ghost or boundary zones. A dimension cannot be less than 3 zones wide and more than `MAX_ANY_SINGLE_DIRECTION - NumberOfGhostZones*2`. `MAX_ANY_SINGLE_DIRECTION` is defined in `fortran.def`. Default: none

`DomainLeftEdge`, `DomainRightEdge` (external) These float values specify the two corners of the problem domain (in code units). The defaults are: 0 0 0 for the left edge and 1 1 1 for the right edge.

`LeftFaceBoundaryCondition`, `RightFaceBoundaryCondition` (external) These two parameters each consist of vectors of integers (of length `TopGridRank`). They specify the boundary conditions for the top grid (and hence the entire hierarchy). The first integer corresponds to the x-direction, the second to the y-direction and the third, the z-direction. The possible values are: 0 - reflecting, 1 - outflow, 2 - inflow, 3 - periodic, 4 - shearing. For inflow, the inflow values can be set through the next parameter, or more commonly are controlled by problem-specific code triggered by the `ProblemType`. For shearing boundaries, the boundary pair in another direction must be periodic. Note that self gravity will not be consistent with shearing boundary conditions. Default: 0 0 0

`ShearingVelocityDirection` (external) Select direction of shearing boundary. Default is x direction. Changing this is probably not a good idea.

`AngularVelocity` (external) The value of the angular velocity in the shearing boundary. Default: 0.001

`VelocityGradient` (external) The value of the per code length gradient in the angular velocity in the shearing boundary. Default: 1.0

`BoundaryConditionName` (external) While the above parameters provide an easy way to set an entire side of grid to a given boundary value, the possibility exists to set the boundary conditions on an individual cell basis. This is most often done with problem specific code, but it can also be set by specifying a file which contains the information in the appropriate format. This is too involved to go into here. Default: none

InitialTime (internal) The time, in code units, of the current step. For cosmology the units are in free-fall times at the initial epoch (see *Enzo Output Formats*). Default: generally 0, depending on problem

Initialdt (internal) The timestep, in code units, for the current step. For cosmology the units are in free-fall times at the initial epoch (see *Enzo Output Formats*). Default: generally 0, depending on problem

4.3 Simulation Identifiers and UUIDs

These parameters help to track, identify and group datasets. For reference, Universally Unique Identifiers (UUIDs) are opaque identifiers using random 128-bit numbers, with an extremely low chance of collision. (See *Simulation Names and Identifiers* for a longer description of these parameters.)

MetaDataIdentifier (external) This is a character string without spaces (specifically, something that can be picked by "%s"), that can be defined in a parameter file, and will be written out in every following output, if it is found.

MetaDataSimulationUUID (internal) A UUID that will be written out in all of the following outputs. Like `MetaDataIdentifier`, an existing UUID will be kept, but if one is not found, and new one will be generated.

MetaDataDatasetUUID (internal) A UUID created for each specific output.

MetaDataRestartDatasetUUID (internal) If a `MetaDataDatasetUUID` UUID is found when the parameter file is read in, it will written to the following datasets. This is used to track simulations across restarts and parameter adjustments.

MetaDataInitialConditionsUUID (internal) This is similar to `MetaDataRestartDatasetUUID`, except it's used to track which initial conditions were used.

4.4 I/O Parameters

There are three ways to specify the frequency of outputs: time-based, cycle-based (a cycle is a top-grid timestep), and, for cosmology simulations, redshift-based. There is also a shortened output format intended for visualization (movie format). Please have a look at *Controlling Enzo data output* for more information.

dtDataDump (external) The time interval, in code units, between time-based outputs. A value of 0 turns off the time-based outputs. Default: 0

CycleSkipDataDump (external) The number of cycles (top grid timesteps) between cycle-based outputs. Zero turns off the cycle-based outputs. Default: 0

DataDumpName (external) The base file name used for both time and cycle based outputs. Default: data

RedshiftDumpName (external) The base file name used for redshift-based outputs (this can be overridden by the `CosmologyOutputRedshiftName` parameter). Normally a four digit identification number is appended to the end of this name, starting from 0000 and incrementing by one for every output. This can be over-ridden by including four consecutive R's in the name (e.g. RedshiftRRRR) in which case the an identification number will not be appended but the four R's will be converted to a redshift with an implied decimal point in the middle (i.e. z=1.24 becomes 0124). Default: RedshiftOutput

CosmologyOutputRedshift[NNNN] (external) The time and cycle-based outputs occur regularly at constant intervals, but the redshift outputs are specified individually. This is done by the use of this statement, which sets the output redshift for a specific identification number (this integer is between 0000 and 9999 and is used in forming the name). So the statement `CosmologyOutputRedshift[1] = 4.0` will cause an output to be written out at z=4 with the name RedshiftOutput0001 (unless the base name is changed either with the previous

parameter or the next one). This parameter can be repeated with different values for the number (NNNN) Default: none

CosmologyOutputRedshiftName[NNNN] (external) This parameter overrides the parameter `RedshiftOutputName` for this (only only this) redshift output. Can be used repeatedly in the same manner as the previous parameter. Default: none

OutputFirstTimeAtLevel (external) This forces Enzo to output when a given level is reached, and at every level thereafter. Default is 0 (off). User can usefully specify anything up to the maximum number of levels in a given simulation.

FileDirectedOutput If this parameter is set to 1, whenever the finest level has finished evolving Enzo will check for new signal files to output. (See *Force Output Now*.) Default 1.

XrayLowerCutoffkeV, XrayUpperCutoffkeV, XrayTableFileName (external) These parameters are used in 2D projections (`enzo -p ...`). The first two specify the X-ray band (observed at z=0) to be used, and the last gives the name of an ascii file that contains the X-ray spectral information. A gzipped version of this file good for bands within the 0.1 - 20 keV range is provided in the distribution in `input/lookup_metal0.3.data`. If these parameters are specified, then the second field is replaced with integrated emissivity along the line of sight in units of 10^{-23} erg/cm^2/s. Default: `XrayLowerCutoffkeV = 0.5`, `XrayUpperCutoffkeV = 2.5`.

ExtractFieldsOnly (external) Used for extractions (enzo -x ...) when only field data are needed instead of field + particle data. Default is 1 (TRUE).

dtRestartDump Reserved for future use.

dtHistoryDump Reserved for future use.

CycleSkipRestartDump Reserved for future use.

CycleSkipHistoryDump Reserved for future use.

RestartDumpName Reserved for future use.

HistoryDumpName Reserved for future use.

ParallelRootGridIO (external) Normally for the mpi version, the root grid is read into the root processor and then partitioned to separate processors using communication. However, for very large root grids (e.g. 512^3), the root processor may not have enough memory. If this toggle switch is set on (i.e. to the value 1), then each processor reads its own section of the root grid. More I/O is required (to split up the grids and particles), but it is more balanced in terms of memory. `ParallelRootGridIO` and `ParallelParticleIO` MUST be set to 1 (TRUE) for runs involving > 64 cpus! Default: 0 (FALSE). See also `Unigrid` below.

Unigrid (external) This parameter should be set to 1 (TRUE) for large cases–AMR as well as non-AMR–where the root grid is 512^3 or larger. This prevents initialization under subgrids at start up, which is unnecessary in cases with simple non-nested initial conditions. Unigrid must be set to 0 (FALSE) for cases with nested initial conditions. Default: 0 (FALSE). See also `ParallelRootGridIO` above.

UnigridTranspose (external) This parameter governs the fast FFT bookkeeping for Unigrid runs. Does not work with isolated gravity. Default: 0 (FALSE). See also `Unigrid` above.

OutputTemperature (external) Set to 1 if you want to output a temperature field in the datasets. Always 1 for cosmology simulations. Default: 0.

OutputCoolingTime (external) Set to 1 if you want to output the cooling time in the datasets. Default: 0.

OutputSmoothedDarkMatter (external) Set to 1 if you want to output a dark matter density field, smoothed by an SPH kernel. Set to 2 to also output smoothed dark matter velocities and velocity dispersion. Set to 0 to turn off. Default: 0.

OutputGriddedStarParticle (external) Set to 1 or 2 to write out star particle data gridded onto mesh. This will be useful e.g. if you have lots of star particles in a galactic scale simulation. 1 will output

just `star_particle_density`; and 2 will dump `actively_forming_stellar_mass_density`, `SFR_density`, etc. Default: 0.

`VelAnyl` (external) Set to 1 if you want to output the divergence and vorticity of velocity. Works in 2D and 3D.

`BAnyl` (external) Set to 1 if you want to output the divergence and vorticity of `Bfield`. Works in 2D and 3D.

`SmoothedDarkMatterNeighbors` (external) Number of nearest neighbors to smooth dark matter quantities over. Default: 32.

4.4.1 Streaming Data Format

`NewMovieLeftEdge`, `NewMovieRightEdge` (external) These two parameters control the region for which the streaming data are written. Default: `DomainLeftEdge` and `DomainRightEdge`.

`MovieSkipTimestep` (external) Controls how many timesteps on a level are skipped between outputs in the streaming data. Streaming format is off if this equals `INT_UNDEFINED`. Default: `INT_UNDEFINED`

`Movie3DVolume` (external) Set to 1 to write streaming data as 3-D arrays. This should always be set to 1 if using the streaming format. A previous version had 2D maximum intensity projections, which now defunct. Default: 0.

`MovieVertexCentered` (external) Set to 1 to write the streaming data interpolated to vertices. Set to 0 for cell-centered data. Default: 0.

`NewMovieDumpNumber` (internal) Counter for streaming data files. This should equal the cycle number.

`MovieTimestepCounter` (internal) Timestep counter for the streaming data files.

`MovieDataField` (external) A maximum of 6 data fields can be written in the streaming format. The data fields are specified by the array element of BaryonField, i.e. 0 = Density, 7 = HII Density. For writing temperature, a special value of 1000 is used. This should be improved to be more transparent in which fields will be written. Any element that equals `INT_UNDEFINED` indicates no field will be written. Default: `INT_UNDEFINED` x 6

`NewMovieParticleOn` (external) Set to 1 to write all particles in the grids. Set to 2 to write ONLY particles that aren't dark matter, e.g. stars. Set to 3/4 to write ONLY particles that aren't dark matter into a file separate from the grid info. (For example, `MoviePackParticle_P000.hdf5`, etc. will be the file name; this will be very helpful in speeding up the access to the star particle data, especially for the visualization or for the star particle. See `AMRH5writer.C`) Set to 0 for no particle output. Default: 0.

4.5 Hierarchy Control Parameters

`StaticHierarchy` (external) A flag which indicates if the hierarchy is static (1) or dynamic (0). In other words, a value of 1 takes the A out of AMR. Default: 1

`RefineBy` (external) This is the refinement factor between a grid and its subgrid. For cosmology simulations, we have found a ratio of 2 to be most useful. Default: 4

`MaximumRefinementLevel` (external) This is the lowest (most refined) depth that the code will produce. It is zero based, so the total number of levels (including the root grid) is one more than this value. Default: 2

`CellFlaggingMethod` (external) The method(s) used to specify when a cell should be refined. This is a list of integers, up to 9, as described by the following table. The methods combine in an "OR" fashion: if any of them indicate that a cell should be refined, then it is flagged. For cosmology simulations, methods 2 and 4 are probably most useful. Note that some methods have additional parameters which are described below. Default: 1

```
1 - refine by slope                    6  - refine by Jeans length
2 - refine by baryon mass              7  - refine if (cooling time < cell width/sound speed)
3 - refine by shocks                   11 - refine by resistive length
4 - refine by particle mass            12 - refine by defined region "MustRefineRegion"
5 - refine by baryon overdensity       13 - refine by metallicity
   (currently disabled)
101 - avoid refinement in regions
      defined in "AvoidRefineRegion"
```

RefineRegionLeftEdge, RefineRegionRightEdge (external) These two parameters control the region in which refinement is permitted. Each is a vector of floats (of length given by the problem rank) and they specify the two corners of a volume. Default: set equal to `DomainLeftEdge` and `DomainRightEdge`.

RefineRegionAutoAdjust (external) This is useful for multiresolution simulations with particles in which the particles have varying mass. Set to 1 to automatically adjust the refine region at root grid timesteps to only contain high-resolution particles. This makes sure that the fine regions do not contain more massive particles which may lead to small particles orbiting them or other undesired outcomes. Setting to any integer (for example, 3) will make AdjustRefineRegion to work at (RefineRegionAutoAdjust-1)th level timesteps because sometimes the heavy particles are coming into the fine regions too fast that you need more frequent protection. Default: 0.

RefineRegionTimeType (external) If set, this controls how the first column of a refinement region evolution file (see below) is interpreted, 0 for code time, 1 for redshift. Default: -1, which is equivalent to 'off'.

RefineRegionFile (external) The name of a text file containing the corners of the time-evolving refinement region. The lines in the file change the values of `RefineRegionLeft/RightEdge` during the course of the simulation, and the lines are ordered in the file from early times to late times. The first column of data is the time index (in code units or redshift, see the parameter above) for the next six columns, which are the values of `RefineRegionLeft/RightEdge`. For example, this might be two lines from the text file when time is indexed by redshift:

```
0.60 0.530 0.612 0.185 0.591 0.667 0.208
0.55 0.520 0.607 0.181 0.584 0.653 0.201
```

In this case, the refinement region stays at the z=0.60 value until z=0.55, when the box moves slightly closer to the (0,0,0) corner. There is a maximum of 300 lines in the file and there is no comment header line. Default: None.

MinimumOverDensityForRefinement (external) These float values (up to 9) are used if the `CellFlaggingMethod` is 2, 4 or 5. For method 2 and 4, the value is the density (baryon or particle), in code units, above which refinement occurs. When using method 5, it becomes rho [code] - 1. The elements in this array must match those in `CellFlaggingMethod`. Therefore, if `CellFlaggingMethod` = 1 4 9 10, `MinimumOverDensityForRefinement` = 0 8.0 0 0.

In practice, this value is converted into a mass by multiplying it by the volume of the top grid cell. The result is then stored in the next parameter (unless that is set directly in which case this parameter is ignored), and this defines the mass resolution of the simulation. Note that the volume is of a top grid cell, so if you are doing a multi-grid initialization, you must divide this number by $r^{(d*l)}$ where r is the refinement factor, d is the dimensionality and l is the (zero-based) lowest level. For example, for a two grid cosmology setup where a cell should be refined whenever the mass exceeds 4 times the mean density of the subgrid, this value should be 4 / $(2^{(3*1)})$ = 4 / 8 = 0.5. Keep in mind that this parameter has no effect if it is changed in a restart output; if you want to change the refinement mid-run you will have to modify the next parameter. Up to 9 numbers may be specified here, each corresponding to the respective `CellFlaggingMethod`. Default: 1.5

MinimumMassForRefinement (internal) This float is usually set by the parameter above and so is labeled internal, but it can be set by hand. For non-cosmological simulations, it can be the easier refinement criteria to specify. It is the mass above which a refinement occurs if the `CellFlaggingMethod` is appropriately set. For cosmological simulations, it is specified in units such that the entire mass in the computational volume is 1.0, otherwise it is in code units. There are 9 numbers here again, as per the above parameter. Default: none

MinimumMassForRefinementLevelExponent (external). This parameter modifies the behaviour of the above parameter. As it stands, the refinement based on the `MinimumMassForRefinement` (hereafter Mmin) parameter is complete Lagrangian. However, this can be modified. The actual mass used is Mmin*$r^{(l*alpha)}$ where r is the refinement factor, l is the level and alpha is the value of this parameter (`MinimumMassForRefinementLevelExponent`). Therefore a negative value makes the refinement super-Lagrangian, while positive values are sub-Lagrangian. There are up to 9 values specified here, as per the above two parameters. Default: 0.0

SlopeFlaggingFields[#] (external) If `CellFlaggingMethod` is 1, and you only want to refine on the slopes of certain fields then you can enter the number IDs of the fields. Default: Refine on slopes of all fields.

MinimumSlopeForRefinement (external) If `CellFlaggingMethod` is 1, then local gradients are used as the refinement criteria. All variables are examined and the relative slope is computed: abs(q(i+1)-q(i-1))/q(i). Where this value exceeds this parameter, the cell is marked for refinement. This causes problems if q(i) is near zero. This is a single integer (as opposed to the list of five for the above parameters). Entering multiple numbers here correspond to the fields listed in `SlopeFlaggingFields`. Default: 0.3

MinimumPressureJumpForRefinement (external) If refinement is done by shocks, then this is the minimum (relative) pressure jump in one-dimension to qualify for a shock. The definition is rather standard (see Colella and Woodward's PPM paper for example) Default: 0.33

MinimumEnergyRatioForRefinement (external) For the dual energy formalism, and cell flagging by shock-detection, this is an extra filter which removes weak shocks (or noise in the dual energy fields) from triggering the shock detection. Default: 0.1

MetallicityRefinementMinLevel (external) Sets the minimum level (maximum cell size) to which a cell enriched with metal above a level set by `MetallicityRefinementMinMetallicity` will be refined. This can be set to any level up to and including `MaximumRefinementLevel`. (No default setting)

MetallicityRefinementMinMetallicity (external) This is the threshold metallicity (in units of solar metallicity) above which cells must be refined to a minimum level of `MetallicityRefinementMinLevel`. Default: 1.0e-5

MustRefineRegionMinRefinementLevel (external) Minimum level to which the rectangular solid volume defined by `MustRefineRegionLeftEdge` and `MustRefineRegionRightEdge` will be refined to at all times. (No default setting)

MustRefineRegionLeftEdge (external) Bottom-left corner of refinement region. Must be within the overall refinement region. Default: 0.0 0.0 0.0

MustRefineRegionRightEdge (external) Top-right corner of refinement region. Must be within the overall refinement region. Default: 1.0 1.0 1.0

MustRefineParticlesRefineToLevel (external) The maximum level on which `MustRefineParticles` are required to refine to. Currently sink particles and MBH particles are required to be sitting at this level at all times. Default: 0

MustRefineParticlesRefineToLevelAutoAdjust (external) The parameter above might not be handy in cosmological simulations if you want your `MustRefineParticles` to be refined to a certain physical length, not to a level whose cell size keeps changing. This parameter (positive integer in pc) allows you to do just that. For example, if you set `MustRefineParticlesRefineToLevelAutoAdjust` = 128 (pc), then the code will automatically calculate `MustRefineParticlesRefineToLevel` using the boxsize and redshift information. Default: 0 (FALSE)

FluxCorrection (external) This flag indicates if the flux fix-up step should be carried out around the boundaries of the sub-grid to preserve conservation (1 - on, 0 - off). Strictly speaking this should always be used, but we have found it to lead to a less accurate solution for cosmological simulations because of the relatively sharp density gradients involved. However, it does appear to be important when radiative cooling is turned on and very dense structures are created. It does work with the ZEUS hydro method, but since velocity is face-centered,

momentum flux is not corrected. Species quantities are not flux corrected directly but are modified to keep the fraction constant based on the density change. Default: 1

InterpolationMethod (external) There should be a whole section devoted to the interpolation method, which is used to generate new sub-grids and to fill in the boundary zones of old sub-grids, but a brief summary must suffice. The possible values of this integer flag are shown in the table below. The names specify (in at least a rough sense) the order of the leading error term for a spatial Taylor expansion, as well as a letter for possible variants within that order. The basic problem is that you would like your interpolation method to be: multi-dimensional, accurate, monotonic and conservative. There doesn't appear to be much literature on this, so I've had to experiment. The first one (ThirdOrderA) is time-consuming and probably not all that accurate. The second one (SecondOrderA) is the workhorse: it's only problem is that it is not always symmetric. The next one (SecondOrderB) is a failed experiment, and SecondOrderC is not conservative. FirstOrderA is everything except for accurate. If HydroMethod = 2 (ZEUS), this flag is ignored, and the code automatically uses SecondOrderC for velocities and FirstOrderA for cell-centered quantities. Default: 1

```
0 - ThirdOrderA      3 - SecondOrderC
1 - SecondOrderA     4 - FirstOrderA
2 - SecondOrderB
```

ConservativeInterpolation (external) This flag (1 - on, 0 - off) indicates if the interpolation should be done in the conserved quantities (e.g. momentum rather than velocity). Ideally, this should be done, but it can cause problems when strong density gradients occur. This must(!) be set off for ZEUS hydro (the code does it automatically). Default: 1

MinimumEfficiency (external) When new grids are created during the rebuilding process, each grid is split up by a recursive bisection process that continues until a subgrid is either of a minimum size or has an efficiency higher than this value. The efficiency is the ratio of flagged zones (those requiring refinement) to the total number of zones in the grid. This is a number between 0 and 1 and should probably by around 0.4 for standard three-dimensional runs. Default: 0.2

NumberOfBufferZones (external) Each flagged cell, during the regridding process, is surrounded by a number of zones to prevent the phenomenon of interest from leaving the refined region before the next regrid. This integer parameter controls the number required, which should almost always be one. Default: 1

RefineByJeansLengthSafetyFactor (external) If the Jeans length refinement criterion (see `CellFlaggingMethod`) is being used, then this parameter specifies the number of cells which must cover one Jeans length. Default: 4

JeansRefinementColdTemperature (external) If the Jeans length refinement criterion (see `CellFlaggingMethod`) is being used, and this parameter is greater than zero, it will be used in place of the temperature in all cells. Default: -1.0

StaticRefineRegionLevel[#] (external) This parameter is used to specify regions of the problem that are to be statically refined, regardless of other parameters. This is mostly used as an internal mechanism to keep the initial grid hierarchy in place, but can be specified by the user. Up to 20 static regions may be defined (this number set in `macros_and_parameters.h`), and each static region is labeled starting from zero. For each static refined region, two pieces of information are required: (1) the region (see the next two parameters), and (2) the level at which the refinement is to occurs (0 implies a level 1 region will always exist). Default: none

StaticRefineRegionLeftEdge[#], StaticRefineRegionRightEdge[#] (external) These two parameters specify the two corners of a statically refined region (see the previous parameter). Default: none

AvoidRefineRegionLevel[#] (external) This parameter is used to limit the refinement to this level in a rectangular region. Up to MAX_STATIC_REGIONS regions can be used.

AvoidRefineRegionLeftEdge[#], AvoidRefineRegionRightEdge[#] (external) These two parameters specify the two corners of a region that limits refinement to a certain level (see the previous parameter). Default: none

RefineByResistiveLength (external) Resistive length is defined as the curl of the magnetic field over the magnitude of the magnetic field. We make sure this length is covered by this number of cells. Default: 2

LoadBalancing (external) Set to 0 to keep child grids on the same processor as their parents. Set to 1 to balance the work on one level over all processors. Set to 2 or 3 to load balance the grids but keep them on the same node. Option 2 assumes grouped scheduling, i.e. proc # = (01234567) reside on node (00112233) if there are 4 nodes. Option 3 assumes round-robin scheduling (proc = (01234567) -> node = (01230123)). Set to 4 for load balancing along a Hilbert space-filling curve on each level. Default: 1

LoadBalancingCycleSkip (external) This sets how many cycles pass before we load balance the root grids. Only works with LoadBalancing set to 2 or 3. NOT RECOMMENDED for nested grid calculations. Default: 10

4.6 Hydrodynamic Parameters

UseHydro (external) This flag (1 - on, 0 - off) controls whether a hydro solver is used. Default: 1

HydroMethod (external) This integer specifies the hydrodynamics method that will be used. Currently implemented are

Hydro method	Description
0	PPM DE (a direct-Eulerian version of PPM)
1	[reserved]
2	ZEUS (a Cartesian, 3D version of Stone & Norman). Note that if ZEUS is selected, it automatically turns off `ConservativeInterpolation` and the `DualEnergyFormalism` flags.
3	Runge Kutta second-order based MUSCL solvers.
4	Same as 3 but including Dedner MHD (Wang & Abel 2008). For 3 and 4 there are the additional parameters `RiemannSolver` and `ReconstructionMethod` you want to set.

Default: 0

More details on each of the above methods can be found at *Hydro and MHD Methods*.

RiemannSolver (external; only if HydroMethod is 3 or 4) This integer specifies the Riemann solver used by the MUSCL solver. Choice of

Riemann solver	Description
0	[reserved]
1	HLL (Harten-Lax-van Leer) a two-wave, three-state solver with no resolution of contact waves
2	[reserved]
3	LLF (Local Lax-Friedrichs)
4	HLLC (Harten-Lax-van Leer with Contact) a three-wave, four-state solver with better resolution of contacts
5	TwoShock

Default: 1 (HLL) for `HydroMethod` = 3; 5 (TwoShock) for `HydroMethod` = 0

RiemannSolverFallback (external) If the euler update results in a negative density or energy, the solver will fallback to the HLL Riemann solver that is more diffusive only for the failing cell. Only active when using the HLLC or TwoShock Riemann solver. Default: OFF.

ReconstructionMethod (external; only if HydroMethod is 3 or 4) This integer specifies the reconstruction method for the MUSCL solver. Choice of

Reconstruction Method	Description
0	PLM (piecewise linear)
1	PPM (piecwise parabolic)
2	[reserved]
3	[reserved]
4	[reserved]

Default: 0 (PLM) for `HydroMethod` = 3; 1 (PPM) for `HydroMethod` = 0

`Gamma` (external) The ratio of specific heats for an ideal gas (used by all hydro methods). If using multiple species (i.e. `MultiSpecies` > 0), then this value is ignored in favor of a direct calculation (except for PPM LR) Default: 5/3.

`Mu` (external) The molecular weight. Default: 0.6.

`ConservativeReconstruction` (external) Experimental. This option turns on the reconstruction of the left/right interfaces in the Riemann problem in the conserved variables (density, momentum, and energy) instead of the primitive variables (density, velocity, and pressure). This generally gives better results in constant-mesh problems has been problematic in AMR simulations. Default: OFF

`PositiveReconstruction` (external) Experimental and not working. This forces the Riemann solver to restrict the fluxes to always give positive pressure. Attempts to use the Waagan (2009), JCP, 228, 8609 method. Default: OFF

`CourantSafetyNumber` (external) This is the maximum fraction of the CFL-implied timestep that will be used to advance any grid. A value greater than 1 is unstable (for all explicit methods). The recommended value is 0.4. Default: 0.6.

`RootGridCourantSafetyNumber` (external) This is the maximum fraction of the CFL-implied timestep that will be used to advance ONLY the root grid. When using simulations with star particle creation turned on, this should be set to a value of approximately 0.01-0.02 to keep star particles from flying all over the place. Otherwise, this does not need to be set, and in any case should never be set to a value greater than 1.0. Default: 1.0.

`DualEnergyFormalism` (external) The dual energy formalism is needed to make total energy schemes such as PPM DE and PPM LR stable and accurate in the "hyper-Machian" regime (i.e. where the ratio of thermal energy to total energy < ~0.001). Turn on for cosmology runs with PPM DE and PPM LR. Automatically turned off when used with the hydro method ZEUS. Integer flag (0 - off, 1 - on). When turned on, there are two energy fields: total energy and thermal energy. Default: 0

`DualEnergyFormalismEta1`, `DualEnergyFormalismEta2` (external) These two parameters are part of the dual energy formalism and should probably not be changed. Defaults: 0.001 and 0.1 respectively.

`PressureFree` (external) A flag that is interpreted by the PPM DE hydro method as an indicator that it should try and mimic a pressure-free fluid. A flag: 1 is on, 0 is off. Default: 0

`PPMFlatteningParameter` (external) This is a PPM parameter to control noise for slowly-moving shocks. It is either on (1) or off (0). Default: 0

`PPMDiffusionParameter` (external) This is the PPM diffusion parameter (see the Colella and Woodward method paper for more details). It is either on (1) or off (0). Default: 1 [Currently disabled (set to 0)]

`PPMSteepeningParameter` (external) A PPM modification designed to sharpen contact discontinuities. It is either on (1) or off (0). Default: 0

`ZEUSQuadraticArtificialViscosity` (external) This is the quadratic artificial viscosity parameter C2 of Stone & Norman, and corresponds (roughly) to the number of zones over which a shock is spread. Default: 2.0

`ZEUSLinearArtificialViscosity` (external) This is the linear artificial viscosity parameter C1 of Stone & Norman. Default: 0.0

4.7 Magnetohydrodynamic Parameters

UseDivergenceCleaning (external) Method 1 and 2 are a failed experiment to do divergence cleaning using successive over relaxation. Method 3 uses conjugate gradient with a 2 cell stencil and Method 4 uses a 4 cell stencil. 4 is more accurate but can lead to aliasing effects. Default: 0

DivergenceCleaningBoundaryBuffer (external) Choose to *not* correct in the active zone of a grid by a boundary of cells this thick. Default: 0

DivergenceCleaningThreshold (external) Calls divergence cleaning on a grid when magnetic field divergence is above this threshold. Default: 0.001

PoissonApproximateThreshold (external) Controls the accuracy of the resulting solution for divergence cleaning Poisson solver. Default: 0.001

ResetMagneticField (external) Set to 1 to reset the magnetic field in the regions that are denser than the critical matter density. Very handy when you want to re-simulate or restart the dumps with MHD. Default: 0

ResetMagneticFieldAmplitude (external) The magnetic field values (in Gauss) that will be used for the above parameter. Default: 0.0 0.0 0.0

4.8 Cosmology Parameters

ComovingCoordinates (external) Flag (1 - on, 0 - off) that determines if comoving coordinates are used or not. In practice this turns on or off the entire cosmology machinery. Default: 0

CosmologyFinalRedshift (external) This parameter specifies the redshift when the calculation will halt. Default: 0.0

CosmologyOmegaMatterNow (external) This is the contribution of all non-relativistic matter (including HDM) to the energy density at the current epoch (z=0), relative to the value required to marginally close the universe. It includes dark and baryonic matter. Default: 0.279

CosmologyOmegaLambdaNow (external) This is the contribution of the cosmological constant to the energy density at the current epoch, in the same units as above. Default: 0.721

CosmologyComovingBoxSize (external) The size of the volume to be simulated in Mpc/h (at z=0). Default: 64.0

CosmologyHubbleConstantNow (external) The Hubble constant at z=0, in units of 100 km/s/Mpc. Default: 0.701

CosmologyInitialRedshift (external) The redshift for which the initial conditions are to be generated. Default: 20.0

CosmologyMaxExpansionRate (external) This float controls the timestep so that cosmological terms are accurate followed. The timestep is constrained so that the relative change in the expansion factor in a step is less than this value. Default: 0.01

CosmologyCurrentRedshift (information only) This is not strictly speaking a parameter since it is never interpreted and is only meant to provide information to the user. Default: n/a

4.9 Gravity Parameters

TopGridGravityBoundary (external) A single integer which specified the type of gravitational boundary conditions for the top grid. Possible values are 0 for periodic and 1 for isolated (for all dimensions). The isolated

boundary conditions have not been tested recently, so caveat emptor. Default: 0

SelfGravity (external) This flag (1 - on, 0 - off) indicates if the baryons and particles undergo self-gravity.

GravitationalConstant (external) This is the gravitational constant to be used in code units. For cgs units it should be 4*pi*G. For cosmology, this value must be 1 for the standard units to hold. A more detailed decription can be found at *Enzo Internal Unit System*. Default: 4*pi.

GreensFunctionMaxNumber (external) The Green's functions for the gravitational potential depend on the grid size, so they are calculated on a as-needed basis. Since they are often re-used, they can be cached. This integer indicates the number that can be stored. They don't take much memory (only the real part is stored), so a reasonable number is 100. [Ignored in current version]. Default: 1

GreensFunctionMaxSize Reserved for future use.

S2ParticleSize (external) This is the gravitational softening radius, in cell widths, in terms of the S2 particle described by Hockney and Eastwood in their book Computer Simulation Using Particles. A reasonable value is 3.0. [Ignored in current version]. Default: 3.0

GravityResolution (external) This was a mis-guided attempt to provide the capability to increase the resolution of the gravitational mesh. In theory it still works, but has not been recently tested. Besides, it's just not a good idea. The value (a float) indicates the ratio of the gravitational cell width to the baryon cell width. [Ignored in current version]. Default: 1

PotentialIterations (external) Number of iterations to solve the potential on the subgrids. Values less than 4 sometimes will result in slight overdensities on grid boundaries. Default: 4.

BaryonSelfGravityApproximation (external) This flag indicates if baryon density is derived in a strange, expensive but self-consistent way (0 - off), or by a completely reasonable and much faster approximation (1 - on). This is an experiment gone wrong; leave on. Well, actually, it's important for very dense structures as when radiative cooling is turned on, so set to 0 if using many levels and radiative cooling is on [ignored in current version]. Default: 1

MaximumGravityRefinementLevel (external) This is the lowest (most refined) depth that a gravitational acceleration field is computed. More refined levels interpolate from this level, provided a mechanism for instituting a minimum gravitational smoothing length. Default: `MaximumRefinementLevel` (unless `HydroMethod` is ZEUS and radiative cooling is on, in which case it is `MaximumRefinementLevel` - 3).

MaximumParticleRefinementLevel (external) This is the level at which the dark matter particle contribution to the gravity is smoothed. This works in an inefficient way (it actually smoothes the particle density onto the grid), and so is only intended for highly refined regions which are nearly completely baryon dominated. It is used to remove the discreteness effects of the few remaining dark matter particles. Not used if set to a value less than 0. Default: -1

4.9.1 External Gravity Source

These parameters set-up an external static background gravity source that is added to the acceleration field for the baryons and particles.

PointSourceGravity (external) This parameter indicates that there is to be a (constant) gravitational field with a point source profile (`PointSourceGravity` = 1) or NFW profile (`PointSourceGravity` = 2). Default: 0

PointSourceGravityConstant (external) If `PointSourceGravity` = 1, this is the magnitude of the point source acceleration at a distance of 1 length unit (i.e. GM in code units). If `PointSourceGravity` = 2, then it takes the mass of the dark matter halo in CGS units. `ProblemType` = 31 (galaxy disk simulation) automatically calculates values for `PointSourceGravityConstant` and `PointSourceGravityCoreRadius`. Default: 1

PointSourceGravityCoreRadius (external) For `PointSourceGravity = 1`, this is the radius inside which the acceleration field is smoothed in code units. With `PointSourceGravity = 2`, it is the scale radius, rs, in CGS units (see Navarro, Frank & White, 1997). Default: 0

PointSourceGravityPosition (external) If the `PointSourceGravity` flag is turned on, this parameter specifies the center of the point-source gravitational field in code units. Default: 0 0 0

ExternalGravity (external) This fulfills the same purpose as `PointSourceGravity` but is more aptly named. `ExternalGravity = 1` turns on an alternative implementation of the NFW profile with properties defined via the parameters `HaloCentralDensity`, `HaloConcentration` and `HaloVirialRadius`. Boxsize is assumed to be 1.0 in this case. `ExternalGravity = 10` gives a gravitational field defined by the logarithmic potential in Binney & Tremaine, corresponding to a disk with constant circular velocity. Default: 0

ExternalGravityConstant (external) If `ExternalGravity = 10`, this is the circular velocity of the disk in code units. Default: 0.0

ExternalGravityDensity Reserved for future use.

ExternalGravityPosition (external) If `ExternalGravity = 10`, this parameter specifies the center of the gravitational field in code units. Default: 0 0 0

ExternalGravityOrientation (external) For `ExternalGravity = 10`, this is the unit vector of the disk's angular momentum (e.g. a disk whose face-on view is oriented in the x-y plane would have `ExternalGravityOrientation = 0 0 1`). Default: 0 0 0

ExternalGravityRadius (external) If `ExternalGravity = 10`, this marks the inner radius of the disk in code units within which the velocity drops to zero. Default: 0.0

UniformGravity (external) This flag (1 - on, 0 - off) indicates if there is to be a uniform gravitational field. Default: 0

UniformGravityDirection (external) This integer is the direction of the uniform gravitational field: 0 - along the x axis, 1 - y axis, 2 - z axis. Default: 0

UniformGravityConstant (external) Magnitude (and sign) of the uniform gravitational acceleration. Default: 1

4.10 Particle Parameters

ParticleBoundaryType (external) The boundary condition imposed on particles. At the moment, this parameter is largely ceremonial as there is only one type implemented: periodic, indicated by a 0 value. Default: 0

ParticleCourantSafetyNumber (external) This somewhat strangely named parameter is the maximum fraction of a cell width that a particle is allowed to travel per timestep (i.e. it is a constant on the timestep somewhat along the lines of it's hydrodynamic brother). Default: 0.5

NumberOfParticles (obsolete) Currently ignored by all initializers, except for TestGravity and TestGravitySphere where it is the number of test points. Default: 0

NumberOfParticleAttributes (internal) It is set to 3 if either `StarParticleCreation` or `StarParticleFeedback` is set to 1 (TRUE). Default: 0

AddParticleAttributes (internal) If set to 1, additional particle attributes will be added and zeroed. This is handy when restarting a run, and the user wants to use star formation afterwards. Default: 0.

ParallelParticleIO (external) Normally, for the mpi version, the particle data are read into the root processor and then distributed to separate processors. However, for very large number of particles, the root processor may not have enough memory. If this toggle switch is set on (i.e. to the value 1), then Ring i/o is turned on and each processor reads its own part of the particle data. More I/O is required, but it is more balanced in terms of

memory. ParallelRootGridIO and ParallelParticleIO MUST be set for runs involving > 64 cpus! Default: 0 (FALSE).

ParticleSplitterIterations (external) Set to 1 to split particles into 13 particles (= 12 children+1 parent, Kitsionas & Whitworth (2002)). This should be ideal for setting up an low-resolution initial condition for a relatively low computational cost, running it for a while, and then restarting it for an extremely high-resolution simulation in a focused region. Currently it implicitly assumes that only DM (type=1) and conventional star particles (type=2) inside the RefineRegion get split. Other particles, which usually become Star class objects, seem to have no reason to be split. Default: 0

ParticleSplitterChildrenParticleSeparation (external) This is the spacing between the child particles placed on a hexagonal close-packed (HCP) array. In the unit of a cell size which the parent particle resides in. Default: 1.0

4.11 Parameters for Additional Physics

RadiativeCooling (external) This flag (1 - on, 0 - off) controls whether or not a radiative cooling module is called for each grid. There are currently several possibilities, controlled by the value of another flag. See *Radiative Cooling and UV Physics Parameters* for more information on the various cooling methods. Default: 0

- If the MultiSpecies flag is off, then equilibrium cooling is assumed and one of the following two will happen. If the parameter GadgetCooling is set to 1, the primordial equilibrium code is called (see below). If GadgetCooling is set to 0, a file called cool_rates.in is read to set a cooling curve. This file consists of a set of temperature and the associated cgs cooling rate; a sample compute with a metallicity Z=0.3 Raymond-Smith code is provided in input/cool_rates.in. This has a cutoff at 10000 K (Sarazin & White 1987). Another choice will be input/cool_rates.in_300K which goes further down to 300 K (Rosen & Bregman 1995).
- If the MultiSpecies flag is on, then the cooling rate is computed directly by the species abundances. This routine (which uses a backward differenced multi-step algorithm) is borrowed from the Hercules code written by Peter Anninos and Yu Zhang, featuring rates from Tom Abel. Other varieties of cooling are controlled by the MetalCooling parameter, as discused below.

GadgetCooling (external) This flag (1 - on, 0 - off) turns on (when set to 1) a set of routines that calculate cooling rates based on the assumption of a six-species primordial gas (H, He, no H2 or D) in equilibrium, and is valid for temperatures greater than 10,000 K. This requires the file TREECOOL to execute. Default: 0

MetalCooling (external) This flag (0 - off, 1 - metal cooling from Glover & Jappsen 2007, 2 - Cen et al (1995), 3 - Cloudy cooling from Smith, Sigurdsson, & Abel 2008) turns on metal cooling for runs that track metallicity. Option 1 is valid for temperatures between 100 K and 10^8K because it considers fine-structure line emission from carbon, oxygen, and silicon and includes the additional metal cooling rates from Sutherland & Dopita (1993). Option 2 is only valid for temperatures above 10^4K. Option 3 uses multi-dimensional tables of heating/cooling values created with Cloudy and optionally coupled to the MultiSpecies chemistry/cooling solver. This method is valid from 10 K to 10^8K. See the Cloudy Cooling parameters below. Default: 0.

MetalCoolingTable (internal) This field contains the metal cooling table required for MetalCooling option 1. In the top level directory input/, there are two files metal_cool.dat and metal_cool_pop3.dat that consider metal cooling for solar abundance and abundances from pair-instability supernovae, respectively. In the same directory, one can find an IDL routine (make_Zcool_table.pro) that generates these tables. Default: metal_cool.dat

MultiSpecies (external) If this flag (1, 2, 3- on, 0 - off) is on, then the code follows not just the total density, but also the ionization states of Hydrogen and Helium. If set to 2, then a nine-species model (including H2, H2+ and H-) will be computed, otherwise only six species are followed (H, H+, He, He+, He++, e-). If set to 3, then

a 12 species model is followed, including D, D+ and HD. This routine, like the last one, is based on work done by Abel, Zhang and Anninos. Default: 0

GadgetEquilibriumCooling (external) An implementation of the ionization equilibrium cooling code used in the GADGET code which includes both radiative cooling and a uniform metagalactic UV background specified by the `TREECOOL` file (in the `amr_mpi/exe` directory). When this parameter is turned on, `MultiSpecies` and `RadiationFieldType` are forced to 0 and `RadiativeCooling` is forced to 1. [Not in public release version]

PhotoelectricHeating (external) If set to be 1, Gamma_pe = 5.1e-26 erg/s will be added uniformly to the gas without any shielding (Tasker & Bryan 2008). At the moment this is still experimental. Default: 0

MultiMetals (external) This was added so that the user could turn on or off additional metal fields - currently there is the standard metallicity field (Metal_Density) and two additional metal fields (Z_Field1 and Z_Field2). Acceptable values are 1 or 0, Default: 0 (off).

4.11.1 Cloudy Cooling

Cloudy cooling from Smith, Sigurdsson, & Abel (2008) interpolates over tables of precomputed cooling data. Cloudy cooling is turned on by setting `MetalCooling` to 3. `RadiativeCooling` must also be set to 1. Depending on the cooling data used, it can be coupled with `MultiSpecies` = 1, 2, or 3 so that the metal-free cooling comes from the `MultiSpecies` machinery and the Cloudy tables provide only the metal cooling. Datasets range in dimension from 1 to 5. Dim 1: interpolate over temperature. Dim 2: density and temperature. Dim 3: density, metallicity, and temperature. Dim 4: density, metallicity, electron fraction, and temperature. Dim 5: density, metallicity, electron fraction, spectral strength, and temperature. See Smith, Sigurdsson, & Abel (2008) for more information on creating Cloudy datasets.

CloudyCoolingGridFile (external) A string specifying the path to the Cloudy cooling dataset.

IncludeCloudyHeating (external) An integer (0 or 1) specifying whether the heating rates are to be included in the calculation of the cooling. Some Cloudy datasets are made with the intention that only the cooling rates are to be used. Default: 0 (off).

CMBTemperatureFloor (external) An integer (0 or 1) specifying whether a temperature floor is created at the temperature of the cosmic microwave background (T_{CMB} = 2.72 (1 + z) K). This is accomplished in the code by subtracting the cooling rate at T_{CMB} such that Cooling = Cooling(T) - Cooling(T_{CMB}). Default: 1 (on).

CloudyElectronFractionFactor (external) A float value to account for additional electrons contributed by metals. This is only used with Cloudy datasets with dimension greater than or equal to 4. The value of this factor is calculated as the sum of (A_i * i) over all elements i heavier than He, where A_i is the solar number abundance relative to H. For the solar abundance pattern from the latest version of Cloudy, using all metals through Zn, this value is 9.153959e-3. Default: 9.153959e-3.

4.11.2 Inline Halo Finding

Enzo can find dark matter (sub)halos on the fly with a friends-of-friends (FOF) halo finder and a subfind method, originally written by Volker Springel. All output files will be written in the directory FOF/.

InlineHaloFinder (external) Set to 1 to turn on the inline halo finder. Default: 0.

HaloFinderSubfind (external) Set to 1 to find subhalos inside each dark matter halo found in the friends-of-friends method. Default: 0.

HaloFinderOutputParticleList (external) Set to 1 to output a list of particle positions and IDs for each (sub)halo. Written in HDF5. Default: 0.

HaloFinderMinimumSize (external) Minimum number of particles to be considered a halo. Default: 50.

HaloFinderLinkingLength (external) Linking length of particles when finding FOF groups. In units of cell width of the finest static grid, e.g. unigrid -> root cell width. Default: 0.1.

HaloFinderCycleSkip (external) Find halos every N^{th} top-level timestep, where N is this parameter. Not used if set to 0. Default: 3.

HaloFinderTimestep (external) Find halos every dt = (this parameter). Only evaluated at each top-level timestep. Not used if negative. Default: -99999.0

HaloFinderLastTime (internal) Last time of a halo find. Default: 0.

4.11.3 Inline Python

PythonSubcycleSkip (external) The number of times Enzo should reach the bottom of the hierarchy before exposing its data and calling Python. Only works with python-yes in compile settings.

4.11.4 Star Formation and Feedback Parameters

For details on each of the different star formation methods available in Enzo see *Active Particles: Stars, BH, and Sinks*.

StarParticleCreation (external) This parameter is bitwise so that multiple types of star formation routines can be used in a single simulation. For example if methods 1 and 3 are desired, the user would specify 10 (2^1 + 2^3), or if methods 1, 4 and 7 are wanted, this would be 146 (2^1 + 2^4 + 2^7). Default: 0

```
0 - Cen & Ostriker (1992)
1 - Cen & Ostriker (1992) with stocastic star formation
2 - Global Schmidt Law / Kravstov et al. (2003)
3 - Population III stars / Abel, Wise & Bryan (2007)
4 - Sink particles: Pure sink particle or star particle with wind feedback depending on
    choice for HydroMethod / Wang et al. (2009)
5 - Radiative star clusters  / Wise & Cen (2009)
6 - [reserved]
7 - Cen & Ostriker (1992) with no delay in formation
8 - Springel & Hernquist (2003)
9 - Massive Black Hole (MBH) particles insertion by hand / Kim et al. (2010)
10 - Population III stellar tracers
```

StarParticleFeedback (external) This parameter works the same way as `StarParticleCreation` but only is valid for `StarParticleCreation` = 0, 1, 2, 7 and 8 because methods 3, 5 and 9 use the radiation transport module and `Star_*.C` routines to calculate the feedback, 4 has explicit feedback and 10 does not use feedback. Default: 0.

StarFeedbackDistRadius (external) If this parameter is greater than zero, stellar feedback will be deposited into the host cell and neighboring cells within this radius. This results in feedback being distributed to a cube with a side of `StarFeedbackDistRadius+1`. It is in units of cell widths of the finest grid which hosts the star particle. Only implemented for `StarFeedbackCreation` = 0 or 1 with `StarParticleFeedback` = 1. (If `StarParticleFeedback` = 0, stellar feedback is only deposited into the cell in which the star particle lives). Default: 0.

StarFeedbackDistCellStep (external) In essence, this parameter controls the shape of the volume where the feedback is applied, cropping the original cube. This volume that are within `StarFeedbackDistCellSteps` cells from the host cell, counted in steps in Cartesian directions, are injected with stellar feedback. Its maximum value is `StarFeedbackDistRadius` * `TopGridRank`. Only implemented for `StarFeedbackCreation` = 0 or 1. See *Distributed Stellar Feedback* for an illustration. Default: 0.

StarMakerTypeIaSNe (external) This parameter turns on thermal and chemical feedback from Type Ia supernovae. The mass loss and luminosity of the supernovae are determined from fits of K. Nagamine. The ejecta are traced in a separate species field, `MetalSNIa_Density`. The metallicity of star particles that comes from this ejecta is stored in the particle attribute `typeia_fraction`. Can be used with `StarParticleCreation` = 0, 1, 2, 5, 7, and 8. Default: 0.

StarMakerPlanetaryNebulae (external) This parameter turns on thermal and chemical feedback from planetary nebulae. The mass loss and luminosity are taken from the same fits from K. Nagamine. The chemical feedback injects gas with the same metallicity as the star particle, and the thermal feedback equates to a 10 km/s wind. The ejecta are not stored in its own species field. Can be used with `StarParticleCreation` = 0, 1, 2, 5, 7, and 8. Default: 0.

Normal Star Formation

The parameters below are considered in `StarParticleCreation` method 0, 1, 2, 7 and 8.

StarMakerOverDensityThreshold (external) The overdensity threshold in code units (for cosmological simulations, note that code units are relative to the total mean density, not just the dark matter mean density) before star formation will be considered. For `StarParticleCreation` = 7 in cosmological simulations, however, `StarMakerOverDensityThreshold` should be in particles/cc, so it is not the ratio with respect to the `DensityUnits` (unlike most other star_makers). This way one correctly represents the Jeans collapse and molecular cloud scale physics even in cosmological simulations. Default: 100

StarMakerSHDensityThreshold (external) The critical density of gas used in Springel & Hernquist star formation (\rho_{th} in the paper) used to determine the star formation timescale in units of g cm^{-3}. Only valid for `StarParticleCreation` = 8. Default: 7e-26.

StarMakerMassEfficiency (external) The fraction of identified baryonic mass in a cell (Mass*dt/t_dyn) that is converted into a star particle. Default: 1

StarMakerMinimumMass (external) The minimum mass of star particle, in solar masses. Note however, the star maker algorithm 2 has a (default off) "stochastic" star formation algorithm that will, in a pseudo-random fashion, allow star formation even for very low star formation rates. It attempts to do so (relatively successfully according to tests) in a fashion that conserves the global average star formation rate. Default: 1e9

StarMakerMinimumDynamicalTime (external) When the star formation rate is computed, the rate is proportional to M_baryon * dt/max(t_dyn, t_max) where t_max is this parameter. This effectively sets a limit on the rate of star formation based on the idea that stars have a non-negligible formation and life-time. The unit is years. Default: 1e6

StarMassEjectionFraction (external) The mass fraction of created stars which is returned to the gas phase. Default: 0.25

StarMetalYield (external) The mass fraction of metals produced by each unit mass of stars created (i.e. it is multiplied by mstar, not ejected). Default: 0.02

StarEnergyToThermalFeedback (external) The fraction of the rest-mass energy of the stars created which is returned to the gas phase as thermal energy. Default: 1e-5

StarEnergyToStellarUV (external) The fraction of the rest-mass energy of the stars created which is returned as UV radiation with a young star spectrum. This is used when calculating the radiation background. Default: 3e-6

StarEnergyToQuasarUV (external) The fraction of the rest-mass energy of the stars created which is returned as UV radiation with a quasar spectrum. This is used when calculating the radiation background. Default: 5e-6

Population III Star Formation

The parameters below are considered in `StarParticleCreation` method 3.

`PopIIIStarMass` (external) Stellar mass of Population III stars created in `StarParticleCreation` method 3. Units of solar masses. The luminosities and supernova energies are calculated from Schaerer (2002) and Heger & Woosley (2002), respectively.

`PopIIIBlackHoles` (external) Set to 1 to create black hole particles that radiate in X-rays for stars that do not go supernova (< 140 solar masses and > 260 solar masses). Default: 0.

`PopIIIBHLuminosityEfficiency` (internal) The radiative efficiency in which the black holes convert accretion to luminosity. Default: 0.1.

`PopIIIOverDensityThreshold` (internal) The overdensity threshold (relative to the total mean density) before Pop III star formation will be considered. Default: 1e6.

`PopIIIH2CriticalFraction` (internal) The H_2 fraction threshold before Pop III star formation will be considered. Default: 5e-4.

`PopIIIMetalCriticalFraction` (internal) The metallicity threshold (relative to gas density, not solar) before Pop III star formation will be considered. Note: this should be changed to be relative to solar! Default: 1e-4.

`PopIIISupernovaRadius` (internal) If the Population III star will go supernova (140<M<260 solar masses), this is the radius of the sphere to inject the supernova thermal energy at the end of the star's life. Units are in parsecs. Default: 1.

`PopIIISupernovaUseColour` (internal) Set to 1 to trace the metals expelled from supernovae. Default: 0.

Radiative Star Cluster Star Formation

The parameters below are considered in `StarParticleCreation` method 5.

`StarClusterUseMetalField` (internal) Set to 1 to trace ejecta from supernovae. Default: 0.

`StarClusterMinDynamicalTime` (internal) When determining the size of a star forming region, one method is to look for the sphere with an enclosed average density that corresponds to some minimum dynamical time. Observations hint that this value should be a few million years. Units are in years. Default: 1e7.

`StarClusterIonizingLuminosity` (internal) The specific luminosity of the stellar clusters. In units of ionizing photons per solar mass. Default: 1e47.

`StarClusterSNEnergy` (internal) The specific energy injected into the gas from supernovae in the stellar clusters. In units of ergs per solar mass. Default: 6.8e48 (Woosley & Weaver 1986).

`StarClusterSNRadius` (internal) This is the radius of the sphere to inject the supernova thermal energy in stellar clusters. Units are in parsecs. Default: 10.

`StarClusterFormEfficiency` (internal) Fraction of gas in the sphere to transfer from the grid to the star particle. Recall that this sphere has a minimum dynamical time set by `StarClusterMinDynamicalTime`. Default: 0.1.

`StarClusterMinimumMass` (internal) The minimum mass of a star cluster particle before the formation is considered. Units in solar masses. Default: 1000.

`StarClusterCombineRadius` (internal) It is possible to merge star cluster particles together within this specified radius. Units in parsecs. This is probably not necessary if ray merging is used. Originally this was developed to reduce the amount of ray tracing involved from galaxies with hundreds of these radiating particles. Default: 10.

Massive Black Hole Particle Formation

The parameters below are considered in StarParticleCreation method 9.

MBHInsertLocationFilename (external) The mass and location of the MBH particle that has to be inserted. For example, the content of the file should be in the following form. For details, see `mbh_maker.src`. Default: `mbh_insert_location.in`

```
#order: MBH mass (in Ms), MBH location[3], MBH creation time
100000.0      0.48530579      0.51455688      0.51467896      0.0
```

4.11.5 Background Radiation Parameters

RadiationFieldType (external)

This integer parameter specifies the type of radiation field that is to be used. Except for `RadiationFieldType` = 9, which should be used with `MultiSpecies` = 2, UV backgrounds can currently only be used with `MultiSpecies` = 1 (i.e. no molecular H support). The following values are used. Default: 0

```
1. Haardt & Madau spectrum with q_alpha=1.5
2. Haardt & Madau spectrum with q_alpha = 1.8
3. Modified Haardt & Madau spectrum to match observations
   (Kirkman & Tytler 2005).
4. H&M spectrum (q_alpha=1.5. supplemented with an X-ray Compton heating
   background from Madau & Efstathiou (see astro-ph/9902080)
9. a constant molecular H2 photo-dissociation rate
10. internally computed radiation field using the algorithm of Cen & Ostriker
11. same as previous, but with very, very simple optical shielding fudge
12. Haardt & Madau spectrum with q_alpha=1.57
```

RadiationFieldLevelRecompute (external) This integer parameter is used only if the previous parameter is set to 10 or 11. It controls how often (i.e. the level at which) the internal radiation field is recomputed. Default: 0

RadiationSpectrumNormalization (external) This parameter was initially used to normalize the photo-ionization and photo-heating rates computed in the function `RadiationFieldCalculateRates()` and then passed on to the `calc_photo_rates()`, `calc_rad()` and `calc_rates()` routines. Later, the normalization as a separate input parameter was dropped for all cases by using the rates computed in `RadiationFieldCalculateRates()` with one exception: The molecular hydrogen (H2) dissociation rate. There a normalization is performed on the rate by multiplying it with `RadiationSpectrumNormalization`. Default: 1e-21

RadiationFieldRedshift (external) This parameter specifies the redshift at which the radiation field is calculated. Default: 0

RadiationShield (external) This parameter specifies whether the user wants to employ approximate radiative-shielding. This parameter will be automatically turned on when RadiationFieldType is set to 11. See `calc_photo_rates.src`. Default: 0

RadiationRedshiftOn (external) The redshift at which the UV background turns on. Default: 7.0.

RadiationRedshiftFullOn (external) The redshift at which the UV background is at full strength. Between z = `RadiationRedshiftOn` and z = `RadiationRedshiftFullOn`, the background is gradually ramped up to full strength. Default: 6.0.

RadiationRedshiftDropOff (external) The redshift at which the strength of the UV background is begins to gradually reduce, reaching zero by `RadiationRedshiftOff`. Default: 0.0.

RadiationRedshiftOff (external) The redshift at which the UV background is fully off. Default: 0.0.

AdjustUVBackground (external) Add description. Default: 1.

SetUVAmplitude (external) Add description. Default: 1.0.

SetHeIIHeatingScale (external) Add description. Default: 1.8.

RadiationSpectrumSlope (external) Add description. Default: 1.5.

4.11.6 Minimum Pressure Support Parameters

UseMinimumPressureSupport (external) When radiative cooling is turned on, and objects are allowed to collapse to very small sizes so that their Jeans length is no longer resolved, then they may undergo artificial fragmentation and angular momentum non-conservation. To alleviate this problem, as discussed in more detail in Machacek, Bryan & Abel (2001), a very simple fudge was introduced: if this flag is turned on, then a minimum temperature is applied to grids with level == `MaximumRefinementLevel`. This minimum temperature is that required to make each cell Jeans stable multiplied by the parameter below. More precisely, the temperature of a cell is set such that the resulting Jeans length is the square-root of the parameter `MinimumPressureSupportParameter`. So, for the default value of 100 (see below), this insures that the ratio of the Jeans length/cell size is at least 10. Default: 0

MinimumPressureSupportParameter (external) This is the numerical parameter discussed above. Default: 100

4.11.7 Radiative Transfer (Ray Tracing) Parameters

RadiativeTransfer (external) Set to 1 to turn on the adaptive ray tracing following Abel, Wise & Bryan 2007. Note that Enzo must be first recompiled after setting `make photon-yes`. Default: 0.

RadiativeTransferRadiationPressure (external) Set to 1 to turn on radiation pressure created from absorbed photon packages. Default: 0

RadiativeTransferInitialHEALPixLevel (internal) Chooses how many rays are emitted from radiation sources. The number of rays in Healpix are given through # = $12x4^{level}$. Default: 3.

RadiativeTransferRaysPerCell (external) Determines the accuracy of the scheme by giving the minimum number of rays to cross cells. The more the better (slower). Default: 5.1.

RadiativeTransferSourceRadius (external) The radius at which the photons originate from the radiation source. A positive value results in a radiating sphere. Default: 0.

RadiativeTransferPropagationRadius (internal) The maximum distance a photon package can travel in one timestep. Currently unused. Default: 0.

RadiativeTransferPropagationSpeed (internal) The fraction of the speed of light at which the photons travel. Default: 1.

RadiativeTransferCoupledRateSolver (internal) Set to 1 to calculate the new ionization fractions and gas energies after every radiative transfer timestep. This option is highly recommended to be kept on. If not, ionization fronts will propagate too slowly. Default: 1.

RadiativeTransferOpticallyThinH2 (external) Set to 1 to include an optically-thin H_2 dissociating (Lyman-Werner) radiation field. Only used if `MultiSpecies` > 1. If `MultiSpecies` > 1 and this option is off, the Lyman-Werner radiation field will be calculated with ray tracing. Default: 1.

RadiativeTransferSplitPhotonPackage (internal) Once photons are past this radius, they can no longer split. In units of kpc. If this value is negative (by default), photons can always split. Default: `FLOAT_UNDEFINED`.

RadiativeTransferPhotonEscapeRadius (internal) The number of photons that pass this distance from its source are summed into the global variable `EscapedPhotonCount[]`. This variable also keeps track of the number of photons passing this radius multiplied by 0.5, 1, and 2. Units are in kpc. Not used if set to 0. Default: 0.

RadiativeTransferInterpolateField (obsolete) A failed experiment in which we evaluate the density at the midpoint of the ray segment in each cell to calculate the optical depth. To interpolate, we need to calculate the vertex interpolated density fields. Default: 0.

RadiativeTransferSourceClustering (internal) Set to 1 to turn on ray merging from combined virtual sources on a binary tree. Default: 0.

RadiativeTransferPhotonMergeRadius (internal) The radius at which the rays will merge from their SuperSource, which is the luminosity weighted center of two sources. This radius is in units of the separation of two sources associated with one SuperSource. If set too small, there will be angular artifacts in the radiation field. Default: 2.5

RadiativeTransferTimestepVelocityLimit (external) Limits the radiative transfer timestep to a minimum value that is determined by the cell width at the finest level divided by this velocity. Units are in km/s. Default: 100.

RadiativeTransferPeriodicBoundary (external) Set to 1 to turn on periodic boundary conditions for photon packages. Default: 0.

RadiativeTransferTraceSpectrum (external) reserved for experimentation. Default: 0.

RadiativeTransferTraceSpectrumTable (external) reserved for experimentation. Default: `spectrum_table.dat`

RadiationXRaySecondaryIon (external) Set to 1 to turn on secondary ionizations and reduce heating from X-ray radiation (Shull & van Steenberg 1985). Currently only BH and MBH particles emit X-rays. Default: 0.

RadiationXRayComptonHeating (external) Set to 1 to turn on Compton heating on electrons from X-ray radiation (Ciotti & Ostriker 2001). Currently only BH and MBH particles emit X-rays. Default: 0.

4.11.8 Radiative Transfer (FLD) Parameters

RadiativeTransferFLD (external) Set to 2 to turn on the fld-based radiation solvers following Reynolds, Hayes, Paschos & Norman, 2009. Note that you also have to compile the source using `make photon-yes` and a `make hypre-yes`. Note that if FLD is turned on, it will force `RadiativeCooling = 0`, `GadgetEquilibriumCooling = 0`, and `RadiationFieldType = 0` to prevent conflicts. Default: 0.

ImplicitProblem (external) Set to 1 to turn on the implicit FLD solver, or 3 to turn on the split FLD solver. Default: 0.

RadHydroParamfile (external) Names the (possibly-different) input parameter file containing solver options for the FLD-based solvers. These are described in the relevant User Guides, located in `doc/implicit_fld` and `doc/split_fld`. Default: NULL.

RadiativeTransfer (external) Set to 0 to avoid conflicts with the ray tracing solver above. Default: 0.

RadiativeTransferFLDCallOnLevel (reserved) The level in the static AMR hierarchy where the unigrid FLD solver should be called. Currently only works for 0 (the root grid). Default: 0.

RadiativeTransferOpticallyThinH2 (external) Set to 0 to avoid conflicts with the built-in optically-thin H_2 dissociating field from the ray-tracing solver. Default: 1.

4.11.9 Radiative Transfer (FLD) Implicit Solver Parameters

These parameters should be placed within the file named in `RadHydroParamfile` in the main parameter file. All are described in detail in the User Guide in `doc/implicit_fld`.

`RadHydroESpectrum` (external)

Type of assumed radiation spectrum for radiation field, Default: 1.

```
-1 - monochromatic spectrum at frequency h nu_{HI} = 13.6 eV
0  - power law spectrum, (nu / nu_{HI} )^(-1.5)
1  - T = 1e5 blackbody spectrum
```

`RadHydroChemistry` (external) Use of hydrogen chemistry in ionization model, set to 1 to turn on the hydrogen chemistry, 0 otherwise. Default: 1.

`RadHydroHFraction` (external) Fraction of baryonic matter comprised of hydrogen. Default: 1.0.

`RadHydroModel` (external)

Determines which set of equations to use within the solver. Default: 1.

```
1  - chemistry-dependent model, with case-B hydrogen II recombination coefficient.
2  - chemistry-dependent model, with case-A hydrogen II recombination coefficient.
4  - chemistry-dependent model, with case-A hydrogen II
   recombination coefficient, but assumes an isothermal gas energy.
10 - no chemistry, instead uses a model of local thermodynamic
   equilibrium to couple radiation to gas energy.
```

`RadHydroMaxDt` (external) maximum time step to use in the FLD solver. Default: 1e20 (no limit).

`RadHydroMinDt` (external) minimum time step to use in the FLD solver. Default: 0.0 (no limit).

`RadHydroInitDt` (external) initial time step to use in the FLD solver. Default: 1e20 (uses hydro time step).

`RadHydroDtNorm` (external)

type of p-norm to use in estimating time-accuracy for predicting next time step. Default: 2.0.

```
 0 - use the max-norm.
>0 - use the specified p-norm.
<0 - illegal.
```

`RadHydroDtRadFac` (external) Desired time accuracy tolerance for the radiation field. Default: 1e20 (unused).

`RadHydroDtGasFac` (external) Desired time accuracy tolerance for the gas energy field. Default: 1e20 (unused).

`RadHydroDtChemFac` (external) Desired time accuracy tolerance for the hydrogen I number density. Default: 1e20 (unused).

`RadiationScaling` (external) Scaling factor for the radiation field, in case standard non-dimensionalization fails. Default: 1.0.

`EnergyCorrectionScaling` (external) Scaling factor for the gas energy correction, in case standard non-dimensionalization fails. Default: 1.0.

`ChemistryScaling` (external) Scaling factor for the hydrogen I number density, in case standard non-dimensionalization fails. Default: 1.0.

`RadiationBoundaryX0Faces` (external)

Boundary condition types to use on the x0 faces of the radiation field. Default: [0 0].

```
0 - Periodic.
1 - Dirichlet.
2 - Neumann.
```

RadiationBoundaryX1Faces (external) Boundary condition types to use on the x1 faces of the radiation field. Default: [0 0].

RadiationBoundaryX2Faces (external) Boundary condition types to use on the x2 faces of the radiation field. Default: [0 0].

RadHydroLimiterType (external)

Type of flux limiter to use in the FLD approximation. Default: 4.

```
0 - original Levermore-Pomraning limiter, à la Levermore & Pomraning, 1981 and Levermore, 1984.
1 - rational approximation to LP limiter.
2 - new approximation to LP limiter (to reduce floating-point cancellation error).
3 - no limiter.
4 - ZEUS limiter (limiter 2, but with no "effective albedo").
```

RadHydroTheta (external) Time-discretization parameter to use, 0 gives explicit Euler, 1 gives implicit Euler, 0.5 gives trapezoidal. Default: 1.0.

RadHydroAnalyticChem (external)

Type of time approximation to use on gas energy and chemistry equations. Default: 1 (if possible for model).

```
0 - use a standard theta-method.
1 - use an implicit quasi-steady state (IQSS) approximation.
```

RadHydroInitialGuess (external)

Type of algorithm to use in computing the initial guess for the time-evolved solution. Default: 0.

```
0 - use the solution from the previous time step (safest).
1 - use explicit Euler with only spatially-local physics (heating & cooling).
2 - use explicit Euler with all physics.
5 - use an analytic predictor based on IQSS approximation of
   spatially-local physics.
```

RadHydroNewtTolerance (external) Desired accuracy for solution to satisfy nonlinear residual (measured in the RMS norm). Default: 1e-6.

RadHydroNewtIters (external) Allowed number of Inexact Newton iterations to achieve tolerance before returning with FAIL. Default: 20.

RadHydroINConst (external) Inexact Newton constant used in specifying tolerances for inner linear solver. Default: 1e-8.

RadHydroMaxMGIters (external) Allowed number of iterations for the inner linear solver (geometric multigrid). Default: 50.

RadHydroMGRelaxType (external) Relaxation method used by the multigrid solver. Default: 1.

:: 1 - Jacobi. 2 - Weighted Jacobi. 3 - Red/Black Gauss-Seidel (symmetric). 4 - Red/Black Gauss-Seidel (non-symmetric).

RadHydroMGPreRelax (external) Number of pre-relaxation sweeps used by the multigrid solver. Default: 1.

RadHydroMGPostRelax (external) Number of post-relaxation sweeps used by the multigrid solver. Default: 1.

EnergyOpacityC0, EnergyOpacityC1, EnergyOpacityC2, EnergyOpacityC3, EnergyOpacityC4 (external) Parameters used in defining the energy-mean opacity used with `RadHydroModel` 10. Default: [1 1 0 1 0].

PlanckOpacityC0, PlanckOpacityC1, PlanckOpacityC2, PlanckOpacityC3, PlanckOpacityC4 (external) Parameters used in defining the Planck-mean opacity used with `RadHydroModel` 10. Default: [1 1 0 1 0].

4.11.10 Radiative Transfer (FLD) Split Solver Parameters

These parameters should be placed within the file named in `RadHydroParamfile` in the main parameter file. All are described in detail in the User Guide in `doc/split_fld`.

RadHydroESpectrum (external)

Type of assumed radiation spectrum for radiation field, Default: 1.

```
-1 - monochromatic spectrum at frequency h nu_{HI}= 13.6 eV
0  - power law spectrum, (nu / nu_{HI})^(-1.5)
1  - T=1e5 blackbody spectrum
```

RadHydroChemistry (external) Use of hydrogen chemistry in ionization model, set to 1 to turn on the hydrogen chemistry, 0 otherwise. Default: 1.

RadHydroHFraction (external) Fraction of baryonic matter comprised of hydrogen. Default: 1.0.

RadHydroModel (external)

Determines which set of equations to use within the solver. Default: 1.

::

1 - chemistry-dependent model, with case-B hydrogen II recombination coefficient.

4 - chemistry-dependent model, with case-A hydrogen II recombination coefficient, but assumes an isothermal gas energy.

10 - no chemistry, instead uses a model of local thermodynamic equilibrium to couple radiation to gas energy.

RadHydroMaxDt (external) maximum time step to use in the FLD solver. Default: 1e20 (no limit).

RadHydroMinDt (external) minimum time step to use in the FLD solver. Default: 0.0 (no limit).

RadHydroInitDt (external) initial time step to use in the FLD solver. Default: 1e20 (uses hydro time step).

RadHydroDtNorm (external)

type of p-norm to use in estimating time-accuracy for predicting next time step. Default: 2.0.

:: 0 - use the max-norm. >0 - use the specified p-norm. <0 - illegal.

RadHydroDtRadFac (external) Desired time accuracy tolerance for the radiation field. Default: 1e20 (unused).

RadHydroDtGasFac (external) Desired time accuracy tolerance for the gas energy field. Default: 1e20 (unused).

RadHydroDtChemFac (external) Desired time accuracy tolerance for the hydrogen I number density. Default: 1e20 (unused).

RadiationScaling (external) Scaling factor for the radiation field, in case standard non-dimensionalization fails. Default: 1.0.

EnergyCorrectionScaling (external) Scaling factor for the gas energy correction, in case standard non-dimensionalization fails. Default: 1.0.

ChemistryScaling (external) Scaling factor for the hydrogen I number density, in case standard non-dimensionalization fails. Default: 1.0.

RadiationBoundaryX0Faces (external) Boundary condition types to use on the x0 faces of the radiation field. Default: [0 0].

```
0 - Periodic.
1 - Dirichlet.
2 - Neumann.
```

RadiationBoundaryX1Faces (external) Boundary condition types to use on the x1 faces of the radiation field. Default: [0 0].

RadiationBoundaryX2Faces (external) Boundary condition types to use on the x2 faces of the radiation field. Default: [0 0].

RadHydroTheta (external) Time-discretization parameter to use, 0 gives explicit Euler, 1 gives implicit Euler, 0.5 gives trapezoidal. Default: 1.0.

RadHydroSolTolerance (external) Desired accuracy for solution to satisfy linear residual (measured in the 2-norm). Default: 1e-8.

RadHydroMaxMGIters (external) Allowed number of iterations for the inner linear solver (geometric multigrid). Default: 50.

RadHydroMGRelaxType (external) Relaxation method used by the multigrid solver. Default: 1.

```
Jacobi.
Weighted Jacobi.
Red/Black Gauss-Seidel (symmetric).
Red/Black Gauss-Seidel (non-symmetric).
```

RadHydroMGPreRelax (external) Number of pre-relaxation sweeps used by the multigrid solver. Default: 1.

RadHydroMGPostRelax (external) Number of post-relaxation sweeps used by the multigrid solver. Default: 1.

EnergyOpacityC0, EnergyOpacityC1, EnergyOpacityC2 (external) Parameters used in defining the energy-mean opacity used with RadHydroModel 10. Default: [1 1 0].

4.11.11 Massive Black Hole Physics Parameters

Following parameters are for the accretion and feedback from the massive black hole particle (`PARTICLE_TYPE_MBH`). More details will soon be described in Kim et al. (2010).

Accretion Physics

MBHAccretion (external) Set to 1 to turn on accretion based on the Eddington-limited spherical Bondi-Hoyle formula (Bondi 1952). Set to 2 to turn on accretion based on the Bondi-Hoyle formula but with fixed temperature defined below. Set to 3 to turn on accretion with a fixed rate defined below. Set to 4 to to turn on accretion based on the Eddington-limited spherical Bondi-Hoyle formula, but without v_rel in the denominator. Set to 5 to turn on accretion based on Krumholz et al.(2006) which takes vorticity into account. Set to 6 to turn on alpha disk formalism based on DeBuhr et al.(2010). 7 and 8 are still failed experiment. Add 10 to each of these options (i.e. 11, 12, 13, 14) to ignore the Eddington limit. See `Star_CalculateMassAccretion.C`. Default: 0 (FALSE)

MBHAccretionRadius (external) This is the radius (in pc) of a gas sphere from which the accreting mass is subtracted out at every timestep. Instead, you may want to try set this parameter to -1, in which case an approximate Bondi radius is calculated and used (from `DEFAULT_MU` and `MBHAccretionFixedTemperature`). If set to -N, it will use N*(Bondi radius). See `CalculateSubtractionParameters.C`. Default: 50.0

MBHAccretingMassRatio (external) There are three different scenarios you can utilize this parameter. (1) In principle this parameter is a nondimensional factor multiplied to the Bondi-Hoyle accretion rate; so 1.0 should give the plain Bondi rate. (2) However, if the Bondi radius is resolved around the MBH, the local density used to calculate Mdot can be higher than what was supposed to be used (density at the Bondi radius!), resulting

in the overestimation of Mdot. 0.0 < `MBHAccretingMassRatio` < 1.0 can be used to fix this. (3) Or, one might try using the density profile of $R^{-1.5}$ to estimate the density at the Bondi radius, which is utilized when `MBHAccretingMassRatio` is set to -1. See `Star_CalculateMassAccretion.C`. Default: 1.0

`MBHAccretionFixedTemperature` (external) This parameter (in K) is used when `MBHAccretion = 2`. A fixed gas temperature that goes into the Bondi-Hoyle accretion rate estimation formula. Default: 3e5

`MBHAccretionFixedRate` (external) This parameter (in Msun/yr) is used when `MBHAccretion = 3`. Default: 1e-3

`MBHTurnOffStarFormation` (external) Set to 1 to turn off star formation (only for `StarParicleCreation` method 7) in the cells where MBH particles reside. Default: 0 (FALSE)

`MBHCombineRadius` (external) The distance (in pc) between two MBH particles in which two energetically-bound MBH particles merge to form one particle. Default: 50.0

`MBHMinDynamicalTime` (external) Minimum dynamical time (in yr) for a MBH particle. Default: 1e7

`MBHMinimumMass` (external) Minimum mass (in Msun) for a MBH particle. Default: 1e3

Feedback Physics

`MBHFeedback` (external)

Set to 1 to turn on thermal feedback of MBH particles (`MBH_THERMAL` - not fully tested). Set to 2 to turn on mechanical feedback of MBH particles (`MBH_JETS`, bipolar jets along the total angular momentum of gas accreted onto the MBH particle so far). Set to 3 to turn on another version of mechanical feedback of MBH particles (`MBH_JETS`, always directed along z-axis). Set to 4 to turn on experimental version of mechanical feedback (*MBH_JETS*, bipolar jets along the total angular momentum of gas accreted onto the MBH particle so far + 10 degree random noise). Set to 5 to turn on experimental version of mechanical feedback (`MBH_JETS`, launched at random direction). Note that, even when this parameter is set to 0, MBH particles still can be radiation sources if `RadiativeTransfer` is on. See `Grid_AddFeedbackSphere.C`. Default: 0 (FALSE)

```
``RadiativeTransfer = 0`` & ``MBHFeedback = 0`` : no feedback at all
``RadiativeTransfer = 0`` & ``MBHFeedback = 1`` : purely thermal feedback
``RadiativeTransfer = 0`` & ``MBHFeedback = 2`` : purely mechanical feedback
``RadiativeTransfer = 1`` & ``MBHFeedback = 0`` : purely radiative feedback
``RadiativeTransfer = 1`` & ``MBHFeedback = 2`` : radiative and
  mechanical feedback combined (one has to change the following
  ``MBHFeedbackRadiativeEfficiency`` parameter accordingly, say from 0.1
  to 0.05, to keep the same total energy across different modes of
  feedback)
```

`MBHFeedbackRadiativeEfficiency` (external) The radiative efficiency of a black hole. 10% is the widely accepted value for the conversion rate from the rest-mass energy of the accreting material to the feedback energy, at the innermost stable orbit of a non-spinning Schwarzschild black hole (Shakura & Sunyaev 1973, Booth & Schaye 2009). Default: 0.1

`MBHFeedbackEnergyCoupling` (external) The fraction of feedback energy that is thermodynamically (for `MBH_THERMAL`) or mechanically (for `MBH_JETS`) coupled to the gas. 0.05 is widely used for thermal feedback (Springel et al. 2005, Di Matteo et al. 2005), whereas 0.0001 or less is recommended for mechanical feedback depending on the resolution of the simulation (Ciotti et al. 2009). Default: 0.05

`MBHFeedbackMassEjectionFraction` (external) The fraction of accreting mass that is returning to the gas phase. For either `MBH_THERMAL` or `MBH_JETS`. Default: 0.1

`MBHFeedbackMetalYield` (external) The mass fraction of metal in the ejected mass. Default: 0.02

MBHFeedbackThermalRadius (external) The radius (in pc) of a sphere in which the energy from MBH_THERMAL feedback is deposited. If set to a negative value, the radius of a sphere gets bigger in a way that the sphere encloses the constant mass (= 4/3*pi*(-MBHFeedbackThermalRadius)^3 Msun). The latter is at the moment very experimental; see Star_FindFeedbackSphere.C. Default: 50.0

MBHFeedbackJetsThresholdMass (external) The bipolar jets by MBH_JETS feedback are injected every time the accumulated ejecta mass surpasses MBHFeedbackJetsThresholdMass (in Msun). Although continuously injecting jets into the gas cells might sound great, unless the gas cells around the MBH are resolved down to Mdot, the jets make little or no dynamical impact on the surrounding gas. By imposing MBHFeedbackJetsThresholdMass, the jets from MBH particles are rendered intermittent, yet dynamically important. Default: 10.0

MBHParticleIO (external) Set to 1 to print out basic information about MBH particles. Will be automatically turned on if MBHFeedback is set to 2 or 3. Default: 0 (FALSE)

MBHParticleIOFilename (external) The name of the file used for the parameter above. Default: mbh_particle_io.dat

4.11.12 Conduction

Isotropic and anisotropic thermal conduction are implemented using the method of Parrish and Stone: namely, using an explicit, forward time-centered algorithm. In the anisotropic conduction, heat can only conduct along magnetic field lines. One can turn on the two types of conduction independently, since there are situations where one might want to use both. The Spitzer fraction can be also set independently for the isotropic and anisotropic conduction.

IsotropicConduction (external) Turns on isotropic thermal conduction using Spitzer conduction. Default: 0 (FALSE)

AnisotropicConduction (external) Turns on anisotropic thermal conduction using Spitzer conduction. Can only be used if MHD is turned on (HydroMethod = 4). Default: 0 (FALSE)

IsotropicConductionSpitzerFraction (external) Prefactor that goes in front of the isotropic Spitzer conduction coefficient. Should be a value between 0 and 1. Default: 1.0

AnisotropicConductionSpitzerFraction (external) Prefactor that goes in front of the anisotropic Spitzer conduction coefficient. Should be a value between 0 and 1. Default: 1.0

ConductionCourantSafetyNumber (external) This is a prefactor that controls the stability of the conduction algorithm. In its current explicit formulation, it must be set to a value of 0.5 or less. Default: 0.5

4.11.13 Shock Finding Parameters

For details on shock finding in Enzo see *Shock Finding*.

ShockMethod (external) This parameter controls the use and type of shock finding. Default: 0

```
0 - Off
1 - Temperature Dimensionally Unsplit Jumps
2 - Temperature Dimensionally Split Jumps
1 - Velocity Dimensionally Unsplit Jumps
2 - Velocity Dimensionally Split Jumps
```

ShockTemperatureFloor (external) When calculating the mach number using temperature jumps, set the temperature floor in the calculation to this value.

StorePreShockFields (external) Optionally store the Pre-shock Density and Temperature during data output.

4.12 Test Problem Parameters

4.12.1 Shock Tube (1: unigrid and AMR)

Riemann problem or arbitrary discontinuity breakup problem. The discontinuity initially separates two arbitrary constant states: Left and Right. Default values correspond to the so called Sod Shock Tube setup (test 1.1). A table below contains a series of recommended 1D tests for hydrodynamic method, specifically designed to test the performance of the Riemann solver, the treatment of shock waves, contact discontinuities, and rarefaction waves in a variety of situations (Toro 1999, p. 129).

Test	LeftDensity	LeftVelocity	LeftPressure	RightDensity	RightVelocity	RightPressure
1.1	1.0	0.0	1.0	0.125	0.0	0.1
1.2	1.0	-2.0	0.4	1.0	2.0	0.4
1.3	1.0	0.0	1000.0	1.0	0.0	0.01
1.4	1.0	0.0	0.01	1.0	0.0	100.0
1.5	5.99924	19.5975	460.894	5.99242	-6.19633	46.0950

ShockTubeBoundary (external) Discontinuity position. Default: 0.5

ShockTubeDirection (external) Discontinuity orientation. Type: integer. Default: 0 (shock(s) will propagate in x-direction)

ShockTubeLeftDensity, ShockTubeRightDensity (external) The initial gas density to the left and to the right of the discontinuity. Default: 1.0 and 0.125, respectively

ShockTubeLeftVelocity, ShockTubeRightVelocity (external) The same as above but for the velocity component in `ShockTubeDirection`. Default: 0.0, 0.0

ShockTubeLeftPressure, ShockTubeRightPressure (external) The same as above but for pressure. Default: 1.0, 0.1

4.12.2 Wave Pool (2)

Wave Pool sets up a simulation with a 1D sinusoidal wave entering from the left boundary. The initial active region is uniform and the wave is entered via inflow boundary conditions.

WavePoolAmplitude (external) The amplitude of the wave. Default: 0.01 - a linear wave.

WavePoolAngle (external) Direction of wave propagation with respect to x-axis. Default: 0.0

WavePoolDensity (external) Uniform gas density in the pool. Default: 1.0

WavePoolNumberOfWaves (external) The test initialization will work for one wave only. Default: 1

WavePoolPressure (external) Uniform gas pressure in the pool. Default: 1.0

WavePoolSubgridLeft, WavePoolSubgridRight (external) Start and end positions of the subgrid. Default: 0.0 and 0.0 (no subgrids)

WavePoolVelocity1(2,3) (external) x-,y-, and z-velocities. Default: 0.0 (for all)

WavePoolWavelength (external) The wavelength. Default: 0.1 (one-tenth of the box)

4.12.3 Shock Pool (3: unigrid 2D, AMR 2D and unigrid 3D)

The Shock Pool test sets up a system which introduces a shock from the left boundary. The initial active region is uniform, and the shock wave enters via inflow boundary conditions. 2D and 3D versions

available. (D. Mihalas & B.W. Mihalas, Foundations of Radiation Hydrodynamics, 1984, p. 236, eq. 56-40.)

ShockPoolAngle (external) Direction of the shock wave propagation with respect to x-axis. Default: 0.0

ShockPoolDensity (external) Uniform gas density in the preshock region. Default: 1.0

ShockPoolPressure (external) Uniform gas pressure in the preshock region. Default: 1.0

ShockPoolMachNumber (external) The ratio of the shock velocity and the preshock sound speed. Default: 2.0

ShockPoolSubgridLeft, ShockPoolSubgridRight (external) Start and end positions of the subgrid. Default: 0.0 and 0.0 (no subgrids)

ShockPoolVelocity1(2,3) (external) Preshock gas velocity (the Mach number definition above assumes a zero velocity in the laboratory reference frame. Default: 0.0 (for all components)

4.12.4 Double Mach Reflection (4)

A test for double Mach reflection of a strong shock (Woodward & Colella 1984). Most of the parameters are "hardwired": d0 = 8.0, e0 = 291.25, u0 = 8.25*sqrt(3.0)/2.0, v0 = -8.25*0.5, w0 = 0.0

DoubleMachSubgridLeft (external) Start position of the subgrid. Default: 0.0

DoubleMachSubgridRight (external) End positions of the subgrid. Default: 0.0

4.12.5 Shock in a Box (5)

A stationary shock front in a static 3D subgrid (Anninos et al. 1994). Initialization is done as in the Shock Tube test.

ShockInABoxBoundary (external) Position of the shock. Default: 0.5

ShockInABoxLeftDensity, ShockInABoxRightDensity (external) Densities to the right and to the left of the shock front. Default: `dL=1.0` and `dR = dL*((Gamma+1)*m^2)/((Gamma-1)*m^2 + 2)`, where `m=2.0` and `speed=0.9*sqrt(Gamma*pL/dL)*m`.

ShockInABoxLeftVelocity, ShockInABoxRightVelocity (external) Velocities to the right and to the left of the shock front. Default: `vL=shockspeed` and `vR=shockspeed-m*sqrt(Gamma*pL/dL)*(1-dL/dR)`, where `m=2.0`, `shockspeed=0.9*sqrt(Gamma*pL/dL)*m`.

ShockInABoxLeftPressure, ShockInABoxRightPressure (external) Pressures to the Right and to the Left of the shock front. Default: pL=1.0 and pR=pL*(2.0*Gamma*m^2 - (Gamma-1))/(Gamma+1), where m=2.0.

ShockInABoxSubgridLeft, ShockInABoxSubgridRight (external) Start and end positions of the subgrid. Default: 0.0 (for both)

4.12.6 Rotating Cylinder (10)

A test for the angular momentum conservation of a collapsing cylinder of gas in an AMR simulation. Written by Brian O'Shea (oshea@msu.edu).

RotatingCylinderOverdensity (external) Density of the rotating cylinder with respect to the background. Default: 20.0

RotatingCylinderSubgridLeft, RotatingCylinderSubgridRight (external) This pair of floating point numbers creates a subgrid region at the beginning of the simulation that will be refined to `MaximumRefinementLevel`. It should probably encompass the whole cylinder. Positions are in units of the box, and it always creates a cube. No default value (meaning off).

RotatingCylinderLambda (external) Angular momentum of the cylinder as a dimensionless quantity. This is identical to the angular momentum parameter lambda that is commonly used to describe cosmological halos. A value of 0.0 is non-rotating, and 1.0 means that the gas is already approximately rotating at the Keplerian value. Default: 0.05

RotatingCylinderTotalEnergy (external) Sets the default gas energy of the ambient medium, in Enzo internal units. Default: 1.0

RotatingCylinderRadius (external) Radius of the rotating cylinder in units of the box size. Note that the height of the cylinder is equal to the diameter. Default: 0.3

RotatingCylinderCenterPosition (external) Position of the center of the cylinder as a vector of floats. Default: (0.5, 0.5, 0.5)

4.12.7 Radiating Shock (11)

This is a test problem similar to the Sedov test problem documented elsewhere, but with radiative cooling turned on (and the ability to use `MultiSpecies` and all other forms of cooling). The main difference is that there are quite a few extras thrown in, including the ability to initialize with random density fluctuations outside of the explosion region, use a Sedov blast wave instead of just thermal energy, and some other goodies (as documented below).

RadiatingShockInnerDensity (external) Density inside the energy deposition area (Enzo internal units). Default: 1.0

RadiatingShockOuterDensity (external) Density outside the energy deposition area (Enzo internal units). Default: 1.0

RadiatingShockPressure (external) Pressure outside the energy deposition area (Enzo internal units). Default: 1.0e-5

RadiatingShockEnergy (external) Total energy deposited (in units of 1e51 ergs). Default: 1.0

RadiatingShockSubgridLeft, RadiatingShockSubgridRight (external) Pair of floats that defines the edges of the region where the initial conditions are refined to MaximumRefinementLevel. No default value.

RadiatingShockUseDensityFluctuation (external) Initialize external medium with random density fluctuations. Default: 0

RadiatingShockRandomSeed (external) Seed for random number geneator (currently using Mersenne Twister). Default: 123456789

RadiatingShockDensityFluctuationLevel (external) Maximum fractional fluctuation in the density level. Default: 0.1

RadiatingShockInitializeWithKE (external) Initializes the simulation with some initial kinetic energy if turned on (0 - off, 1 - on). Whether this is a simple sawtooth or a Sedov profile is controlled by the parameter `RadiatingShockUseSedovProfile`. Default: 0

RadiatingShockUseSedovProfile (external) If set to 1, initializes simulation with a Sedov blast wave profile (thermal and kinetic energy components). If this is set to 1, it overrides all other kinetic energy-related parameters. Default: 0

RadiatingShockSedovBlastRadius (external) Maximum radius of the Sedov blast, in units of the box size. Default: 0.05

RadiatingShockKineticEnergyFraction (external) Fraction of the total supernova energy that is deposited as kinetic energy. This only is used if `RadiatingShockInitializeWithKE` is set to 1. Default: 0.0

RadiatingShockCenterPosition (external) Vector of floats that defines the center of the explosion. Default: (0.5, 0.5, 0.5)

RadiatingShockSpreadOverNumZones (external) Number of cells that the shock is spread over. This corresponds to a radius of approximately N * dx, where N is the number of cells and dx is the resolution of the highest level of refinement. This does not have to be an integer value. Default: 3.5

4.12.8 Free Expansion (12)

This test sets up a blast wave in the free expansion stage. There is only kinetic energy in the sphere with the radial velocity proportional to radius. If let evolve for long enough, the problem should turn into a Sedov-Taylor blast wave.

FreeExpansionFullBox (external) Set to 0 to have the blast wave start at the origin with reflecting boundaries. Set to 1 to center the problem at the domain center with periodic boundaries. Default: 0

FreeExpansionMass (external) Mass of the ejecta in the blast wave in solar masses. Default: 1

FreeExpansionRadius (external) Initial radius of the blast wave. Default: 0.1

FreeExpansionDensity (external) Ambient density of the problem. Default: 1

FreeExpansionEnergy (external) Total energy of the blast wave in ergs. Default: 1e51

FreeExpansionMaxVelocity (external) Maximum initial velocity of the blast wave (at the outer radius). If not set, a proper value is calculated using the formula in Draine & Woods (1991). Default: `FLOAT_UNDEFINED`

FreeExpansionTemperature (external) Ambient temperature of the problem in K. Default: 100

FreeExapnsionBField (external) Initial uniform magnetic field. Default: 0 0 0

FreeExpansionVelocity (external) Initial velocity of the ambient medium. Default: 0 0 0

FreeExpansionSubgridLeft (external) Leftmost edge of the region to set the initial refinement. Default: 0

FreeExpansionSubgridRight (external) Rightmost edge of the region to set the initial refinement. Default: 0

4.12.9 Zeldovich Pancake (20)

A test for gas dynamics, expansion terms and self-gravity in both linear and non-linear regimes [Bryan thesis (1996), Sect. 3.3.4-3.3.5; Norman & Bryan (1998), Sect. 4]

ZeldovichPancakeCentralOffset (external) Offset of the pancake plane. Default: 0.0 (no offset)

ZeldovichPancakeCollapseRedshift (external) A free parameter which determines the epoch of caustic formation. Default: 1.0

ZeldovichPancakeDirection (external) Orientation of the pancake. Type: integer. Default: 0 (along the x-axis)

ZeldovichPancakeInitialTemperature (external) Initial gas temperature. Units: degrees Kelvin. Default: 100

ZeldovichPancakeOmegaBaryonNow (external) Omega Baryon at redshift z=0; standard setting. Default: 1.0

ZeldovichPancakeOmegaCDMNow (external) Omega CDM at redshift z=0. Default: 0 (assumes no dark matter)

4.12.10 Pressureless Collapse (21)

An 1D AMR test for the gravity solver and advection routines: the two-sided one-dimensional collapse of a homogeneous plane parallel cloud in Cartesian coordinates. Isolated boundary conditions. Gravitational constant G=1; free fall time 0.399. The expansion terms are not used in this test. (Bryan thesis 1996, Sect. 3.3.1).

PressurelessCollapseDirection (external) Coordinate direction. Default: 0 (along the x-axis).

PressurelessCollapseInitialDensity (external) Initial density (the fluid starts at rest). Default: 1.0

4.12.11 Adiabatic Expansion (22)

A test for time-integration accuracy of the expansion terms (Bryan thesis 1996, Sect. 3.3.3).

AdiabaticExpansionInitialTemperature (external) Initial temperature for Adiabatic Expansion test; test example assumes 1000 K. Default: 200. Units: degrees Kelvin

AdiabaticExpansionInitialVelocity (external) Initial expansion velocity. Default: 100. Units: km/s

AdiabaticExpansionOmegaBaryonNow (external) Omega Baryon at redshift z=0; standard value 1.0. Default: 1.0

AdiabaticExpansionOmegaCDMNow (external) Omega CDM at redshift z=0; default setting assumes no dark matter. Default: 0.0

4.12.12 Test Gravity (23)

We set up a system in which there is one grid point with mass in order to see the resulting acceleration field. If finer grids are specified, the mass is one grid point on the subgrid as well. Periodic boundary conditions are imposed (gravity).

TestGravityDensity (external) Density of the central peak. Default: 1.0

TestGravityMotionParticleVelocity (external) Initial velocity of test particle(s) in x-direction. Default: 1.0

TestGravityNumberOfParticles (external) The number of test particles of a unit mass. Default: 0

TestGravitySubgridLeft, TestGravitySubgridRight (external) Start and end positions of the subgrid. Default: 0.0 and 0.0 (no subgrids)

TestGravityUseBaryons (external) Boolean switch. Type: integer. Default: 0 (FALSE)

4.12.13 Spherical Infall (24)

A test based on Bertschinger's (1985) 3D self-similar spherical infall solution onto an initially overdense perturbation in an Einstein-de Sitter universe.

SphericalInfallCenter (external) Coordinate(s) for the accretion center. Default: top grid center

SphericalInfallFixedAcceleration (external) Boolean flag. Type: integer. Default: 0 (FALSE)

SphericalInfallFixedMass (external) Mass used to calculate the acceleration from spherical infall (GM/(4*pi*r^3*a)). Default: If SphericalInfallFixedMass is undefined and `SphericalInfallFixedAcceleration == TRUE`, then `SphericalInfallFixedMass = SphericalInfallInitialPerturbation * TopGridVolume`

SphericalInfallInitialPerturbation (external) The perturbation of initial mass density. Default: 0.1

SphericalInfallOmegaBaryonNow (external) Omega Baryon at redshift z=0; standard setting. Default: 1.0

SphericalInfallOmegaCDMNow (external) Omega CDM at redshift z=0. Default: 0.0 (assumes no dark matter) Default: 0.0

SphericalInfallSubgridIsStatic (external) Boolean flag. Type: integer. Default: 0 (FALSE)

SphericalInfallSubgridLeft, SphericalInfallSubgridRight (external) Start and end positions of the subgrid. Default: 0.0 and 0.0 (no subgrids)

SphericalInfallUseBaryons (external) Boolean flag. Type: integer. Default: 1 (TRUE)

4.12.14 Test Gravity: Sphere (25)

Sets up a 3D spherical mass distribution and follows its evolution to test the gravity solver.

TestGravitySphereCenter (external) The position of the sphere center. Default: at the center of the domain

TestGravitySphereExteriorDensity (external) The mass density outside the sphere. Default: `tiny_number`

TestGravitySphereInteriorDensity (external) The mass density at the sphere center. Default: 1.0

TestGravitySphereRadius (external) Radius of self-gravitating sphere. Default: 0.1

TestGravitySphereRefineAtStart (external) Boolean flag. Type: integer. Default: 0 (FALSE)

TestGravitySphereSubgridLeft, TestGravitySphereSubgridRight (external) Start and end positions of the subgrid. Default: 0.0 and 0.0 (no subgrids)

TestGravitySphereType (external) Type of mass density distribution within the sphere. Options include: (0) uniform density distrubution within the sphere radius; (1) a power law with an index -2.0; (2) a power law with an index -2.25 (the exact power law form is, e.g., $r^{-2.25}$, where r is measured in units of `TestGravitySphereRadius`). Default: 0 (uniform density)

TestGravitySphereUseBaryons (external) Boolean flag. Type: integer . Default: 1 (TRUE)

4.12.15 Gravity Equilibrium Test (26)

Sets up a hydrostatic exponential atmosphere with the pressure=1.0 and density=1.0 at the bottom. Assumes constant gravitational acceleration (uniform gravity field).

GravityEquilibriumTestScaleHeight (external) The scale height for the exponential atmosphere . Default: 0.1

4.12.16 Collapse Test (27)

A self-gravity test.

CollapseTestInitialTemperature (external) Initial gas temperature. Default: 1000 K. Units: degrees Kelvin

CollapseTestNumberOfSpheres (external) Number of spheres to collapse; must be <= `MAX_SPHERES=10` (see `Grid.h` for definition). Default: 1

CollapseTestRefineAtStart (external) Boolean flag. Type: integer. If TRUE, then initializing routine refines the grid to the desired level. Default: 1 (TRUE)

CollapseTestUseColour (external) Boolean flag. Type: integer. Default: 0 (FALSE)

CollapseTestUseParticles (external) Boolean flag. Type: integer. Default: 0 (FALSE)

CollapseTestSphereCoreRadius (external) An array of core radii for collapsing spheres. Default: 0.1 (for all spheres)

CollapseTestSphereDensity (external) An array of density values for collapsing spheres. Default: 1.0 (for all spheres)

CollapseTestSpherePosition (external) A two-dimensional array of coordinates for sphere centers. Type: float[MAX_SPHERES][MAX_DIMENSION]. Default for all spheres: 0.5*(DomainLeftEdge[dim] + DomainRightEdge[dim])

CollapseTestSphereRadius (external) An array of radii for collapsing spheres. Default: 1.0 (for all spheres)

CollapseTestSphereTemperature (external) An array of temperatures for collapsing spheres. Default: 1.0. Units: degrees Kelvin

CollapseTestSphereType (external) An integer array of sphere types. Default: 0

CollapseTestSphereVelocity (external) A two-dimensional array of sphere velocities. Type: float[MAX_SPHERES][MAX_DIMENSION]. Default: 0.0

CollapseTestUniformVelocity (external) Uniform velocity. Type: float[MAX_DIMENSION]. Default: 0 (for all dimensions)

CollapseTestSphereMetallicity (external) Metallicity of the sphere in solar metallicity. Default: 0.

CollapseTestFracKeplerianRot (external) Rotational velocity of the sphere in units of Keplerian velocity, i.e. 1 is rotationally supported. Default: 0.

CollapseTestSphereTurbulence (external) Turbulent velocity field sampled from a Maxwellian distribution with the temperature specified in CollapseTestSphereTemperature This parameter multiplies the turbulent velocities by its value. Default: 0.

CollapseTestSphereDispersion (external) If using particles, this parameter multiplies the velocity dispersion of the particles by its value. Only valid in sphere type 8 (cosmological collapsing sphere from a uniform density). Default: 0.

CollapseTestSphereCutOff (external) At what radius to terminate a Bonner-Ebert sphere. Units? Default: 6.5

CollapseTestSphereAng1 (external) Controls the initial offset (at r=0) of the rotational axis. Units in radians. Default: 0.

CollapseTestSphereAng2 (external) Controls the outer offset (at r=SphereRadius of the rotational axis. In both CollapseTestSphereAng1 and CollapseTestSphereAng2 are set, the rotational axis linearly changes with radius between CollapseTestSphereAng1 and CollapseTestSphereAng2. Units in radians. Default: 0.

CollapseTestSphereInitialLevel (external) Failed experiment to try to force refinement to a specified level. Not working. Default: 0.

4.12.17 Cosmology Simulation (30)

A sample cosmology simulation.

CosmologySimulationDensityName (external) This is the name of the file which contains initial data for baryon density. Type: string. Example: GridDensity. Default: none

CosmologySimulationTotalEnergyName (external) This is the name of the file which contains initial data for total energy. Default: none

CosmologySimulationGasEnergyName (external) This is the name of the file which contains initial data for gas energy. Default: none

CosmologySimulationVelocity[123]Name (external) These are the names of the files which contain initial data for gas velocities. `Velocity1` - x-component; `Velocity2` - y-component; `Velocity3` - z-component. Default: none

CosmologySimulationParticleMassName (external) This is the name of the file which contains initial data for particle masses. Default: none

CosmologySimulationParticlePositionName (external) This is the name of the file which contains initial data for particle positions. Default: none

CosmologySimulationParticleVelocityName (external) This is the name of the file which contains initial data for particle velocities. Default: none

CosmologySimulationParticleVelocity[123]Name (external) This is the name of the file which contains initial data for particle velocities but only has one component per file. This is more useful with very large (>=2048^3) datasets. Currently one can only use this in conjunction with `CosmologySimulationCalculatePositions`. because it expects a 3D grid structure instead of a 1D list of particles. Default: None.

CosmologySimulationCalculatePositions (external) If set to 1, Enzo will calculate the particle positions in one of two ways: 1) By using a linear Zeldo'vich approximation based on the particle velocities and a displacement factor [dln(growth factor) / dtau, where tau is the conformal time], which is stored as an attribute in the initial condition files, or 2) if the user has also defined either CosmologySimulationParticleDisplacement-Name or CosmologySimulationParticleDisplacement[123]Name, by reading in particle displacements from an external code and applying those directly. The latter allows the use of non-linear displacements. Default: 0.

CosmologySimulationParticleDisplacementName (external) This is the name of the file which contains initial data for particle displacements. Default: none

CosmologySimulationParticleDisplacement[123]Name (external) This is the name of the file which contains initial data for particle displacements but only has one component per file. This is more useful with very large (>=2048^3) datasets. Currently one can only use this in conjunction with `CosmologySimulationCalculatePositions`. because it expects a 3D grid structure instead of a 1D list of particles. Default: None.

CosmologySimulationNumberOfInitialGrids (external) The number of grids at startup. 1 means top grid only. If >1, then nested grids are to be defined by the following parameters. Default: 1

CosmologySimulationSubgridsAreStatic (external) Boolean flag, defines whether the subgrids introduced at the startup are static or not. Type: integer. Default: 1 (TRUE)

CosmologySimulationGridLevel (external) An array of integers setting the level(s) of nested subgrids. Max dimension `MAX_INITIAL_GRIDS` is defined in `CosmologySimulationInitialize.C` as 10. Default for all subgrids: 1, 0 - for the top grid (grid #0)

CosmologySimulationGridDimension[#] (external) An array (arrays) of 3 integers setting the dimensions of nested grids. Index starts from 1. Max number of subgrids `MAX_INITIAL_GRIDS` is defined in `CosmologySimulationInitialize.C` as 10. Default: none

CosmologySimulationGridLeftEdge[#] (external) An array (arrays) of 3 floats setting the left edge(s) of nested subgrids. Index starts from 1. Max number of subgrids `MAX_INITIAL_GRIDS` is defined in `CosmologySimulationInitialize.C` as 10. Default: none

CosmologySimulationGridRightEdge[#] (external) An array (arrays) of 3 floats setting the right edge(s) of nested subgrids. Index starts from 1. Max number of subgrids `MAX_INITIAL_GRIDS` is defined in `CosmologySimulationInitialize.C` as 10. Default: none

CosmologySimulationUseMetallicityField (external) Boolean flag. Type: integer. Default: 0 (FALSE)

CosmologySimulationInitialFractionH2I (external) The fraction of molecular hydrogen (H_2) at InitialRedshift. This and the following chemistry parameters are used if MultiSpecies is defined as 1 (TRUE). Default: 2.0e-20

CosmologySimulationInitialFractionH2II (external) The fraction of singly ionized molecular hydrogen (H2+) at InitialRedshift. Default: 3.0e-14

CosmologySimulationInitialFractionHeII (external) The fraction of singly ionized helium at InitialRedshift. Default: 1.0e-14

CosmologySimulationInitialFractionHeIII (external) The fraction of doubly ionized helium at InitialRedshift. Default: 1.0e-17

CosmologySimulationInitialFractionHII (external) The fraction of ionized hydrogen at InitialRedshift. Default: 1.2e-5

CosmologySimulationInitialFractionHM (external) The fraction of negatively charged hydrogen (H-) at InitialRedshift. Default: 2.0e-9

CosmologySimulationInitialFractionMetal (external) The fraction of metals at InitialRedshift. Default: 1.0e-10

CosmologySimulationInitialTemperature (external) A uniform temperature value at InitialRedshift (needed if the initial gas energy field is not supplied). Default: 550*((1.0 + InitialRedshift)/201)^2

CosmologySimulationOmegaBaryonNow (external) This is the contribution of baryonic matter to the energy density at the current epoch (z=0), relative to the value required to marginally close the universe. Typical value 0.06. Default: 1.0

CosmologySimulationOmegaCDMNow (external) This is the contribution of CDM to the energy density at the current epoch (z=0), relative to the value required to marginally close the universe. Typical value 0.94. Default: 0.0 (no dark matter)

CosmologySimulationManuallySetParticleMassRatio (external) This binary flag (0 - off, 1 - on) allows the user to manually set the particle mass ratio in a cosmology simulation. Default: 0 (Enzo automatically sets its own particle mass)

CosmologySimulationManualParticleMassRatio (external) This manually controls the particle mass in a cosmology simulation, when CosmologySimulationManuallySetParticleMassRatio is set to 1. In a standard Enzo simulation with equal numbers of particles and cells, the mass of a particle is set to CosmologySimulationOmegaCDMNow/CosmologySimulationOmegaMatterNow, or somewhere around 0.85 in a WMAP-type cosmology. When a different number of particles and cells are used (128 particles along an edge and 256 cells along an edge, for example) Enzo attempts to calculate the appropriate particle mass. This breaks down when ParallelRootGridIO and/or ParallelParticleIO are turned on, however, so the user must set this by hand. If you have the ratio described above (2 cells per particle along each edge of a 3D simulation) the appropriate value would be 8.0 (in other words, this should be set to (number of cells along an edge) / (number of particles along an edge) cubed. Default: 1.0.

4.12.18 Isolated Galaxy Evolution (31)

Initializes an isolated galaxy, as per the Tasker & Bryan series of papers.

GalaxySimulationRefineAtStart (external) Controls whether or not the simulation is refined beyond the root grid at initialization. (0 - off, 1 - on). Default: 1

GalaxySimulationInitialRefinementLevel (external) Level to which the simulation is refined at initialization, assuming GalaxySimulationRefineAtStart is set to 1. Default: 0

GalaxySimulationSubgridLeft, GalaxySimulationSubgridRight (external) Vectors of floats defining the edges of the volume which is refined at start. No default value.

GalaxySimulationUseMetallicityField (external) Turns on (1) or off (0) the metallicity field. Default: 0

GalaxySimulationInitialTemperature (external) Initial temperature that the gas in the simulation is set to. Default: 1000.0

GalaxySimulationUniformVelocity (external) Vector that gives the galaxy a uniform velocity in the ambient medium. Default: (0.0, 0.0, 0.0)

GalaxySimulationDiskRadius (external) Radius (in Mpc) of the galax disk. Default: 0.2

GalaxySimulationGalaxyMass (external) Dark matter mass of the galaxy, in Msun. Needed to initialize the NFW gravitational potential. Default: 1.0e+12

GalaxySimulationGasMass (external) Amount of gas in the galaxy, in Msun. Used to initialize the density field in the galactic disk. Default: 4.0e+10

GalaxySimulationDiskPosition (external) Vector of floats defining the center of the galaxy, in units of the box size. Default: (0.5, 0.5, 0.5)

GalaxySimulationDiskScaleHeightz (external) Disk scale height, in Mpc. Default: 325e-6

GalaxySimulationDiskScaleHeightR (external) Disk scale radius, in Mpc. Default: 3500e-6

GalaxySimulationDarkMatterConcentrationParameter (external) NFW dark matter concentration parameter. Default: 12.0

GalaxySimulationDiskTemperature (external) Temperature of the gas in the galactic disk. Default: 1.0e+4

GalaxySimulationInflowTime (external) Controls inflow of gas into the box. It is strongly suggested that you leave this off. Default: -1 (off)

GalaxySimulationInflowDensity (external) Controls inflow of gas into the box. It is strongly suggested that you leave this off. Default: 0.0

GalaxySimulationAngularMomentum (external) Unit vector that defines the angular momentum vector of the galaxy (in other words, this and the center position define the plane of the galaxy). This _MUST_ be set! Default: (0.0, 0.0, 0.0)

4.12.19 Shearing Box Simulation (35)

ShearingBoxProblemType (external) Value of 0 starts a sphere advection through the shearing box test. Value of 1 starts a standard Balbus & Hawley shearing box simulation. Default: 0

ShearingBoxRefineAtStart (external) Refine the simulation at start. Default: 1.0

ThermalMagneticRatio (external) Plasma beta (Pressure/Magnetic Field Energy) Default: 400.0

FluctuationAmplitudeFraction (external) The magnitude of the sinusoidal velocity perturbations as a fraction of the angular velocity. Default: 0.1

ShearingBoxGeometry (external) Defines the radius of the sphere for `ShearingBoxProblemType` = 0, and the frequency of the velocity fluctuations (in units of 2pi) for `ShearingBoxProblemType` = 1. Default: 2.0

4.12.20 Supernova Restart Simulation (40)

All of the supernova parameters are to be put into a restart dump parameter file. Note that ProblemType must be reset to 40, otherwise these are ignored.

SupernovaRestartEjectaCenter[#] (external) Input is a trio of coordinates in code units where the supernova's energy and mass ejecta will be centered. Default: FLOAT_UNDEFINED

SupernovaRestartEjectaEnergy (external) The amount of energy instantaneously output in the simulated supernova, in units of 1e51 ergs. Default: 1.0

SupernovaRestartEjectaMass (external) The mass of ejecta in the supernova, in units of solar masses. Default: 1.0

SupernovaRestartEjectaRadius (external) The radius over which the above two parameters are spread. This is important because if it's too small the timesteps basically go to zero and the simulation takes forever, but if it's too big then you loose information. Units are parsecs. Default: 1.0 pc

SupernovaRestartName (external) This is the name of the restart data dump that the supernova problem is initializing from.

SupernovaRestartColourField Reserved for future use.

4.12.21 Photon Test (50)

This test problem is modeled after Collapse Test (27), and thus borrows all of its parameters that control the setup of spheres. Replace CollapseTest with PhotonTest in the sphere parameters, and it will be recognized. However there are parameters that control radiation sources, which makes this problem unique from collapse test. The radiation sources are fixed in space.

PhotonTestNumberOfSources (external) Sets the number of radiation sources. Default: 1.

PhotonTestSourceType (external) Sets the source type. No different types at the moment. Default: 0.

PhotonTestSourcePosition (external) Sets the source position. Default: 0.5*(DomainLeftEdge + DomainRightEdge)

PhotonTestSourceLuminosity (external) Sets the source luminosity in units of photons per seconds. Default: 0.

PhotonTestSourceLifeTime (external) Sets the lifetime of the source in units of code time. Default: 0.

PhotonTestSourceRampTime (external) If non-zero, the source will exponentially increase its luminosity until it reaches the full luminosity when the age of the source equals this parameter. Default: 0.

PhotonTestSourceEnergyBins (external) Sets the number of energy bins in which the photons are emitted from the source. Default: 4.

PhotonTestSourceSED (external) An array with the fractional luminosity in each energy bin. The sum of this array must equal to one. Default: 1 0 0 0

PhotonTestSourceEnergy (external) An array with the mean energy in each energy bin. Units are in eV. Default: 14.6 25.6 56.4 12.0 (i.e. HI ionizing, HeI ionizing, HeII ionizing, Lyman-Werner)

PhotonTestInitialFractionHII (external) Sets the initial ionized fraction of hydrogen. Default: 1.2e-5

PhotonTestInitialFractionHeII (external) Sets the initial singly-ionized fraction of helium. Default: 1e-14

PhotonTestInitialFractionHeIII (external) Sets the initial doubly-ionized fraction of helium. Default: 1e-17

PhotonTestInitialFractionHM (external) Sets the initial fraction of H^-. Default: 2e-9

PhotonTestInitialFractionH2I (external) Sets the initial neutral fraction of H2. Default: 2e-20

PhotonTestInitialFractionH2II (external) Sets the initial ionized fraction of H2. Default: 3e-14

PhotonTestOmegaBaryonNow (obsolete) Default: 0.05.

4.12.22 Cooling Test (62)

This test problem sets up a 3D grid varying smoothly in log-space in H number density (x dimension), metallicity (y-dimension), and temperature (z-dimension). The hydro solver is turned off. By varying the `RadiativeCooling` and `CoolingTestResetEnergies` parameters, two different cooling tests can be run. 1) Keep temperature constant, but iterate chemistry to allow species to converge. This will allow you to make plots of Cooling rate vs. T. For this, set `RadiativeCooling` to 0 and `CoolingTestResetEnergies` to 1. 2) Allow gas to cool, allowing one to plot Temperature vs. time. For this, set `RadiativeCooling` to 1 and `CoolingTestResetEnergies` to 0.

CoolingTestMinimumHNumberDensity (external) The minimum density in code units at x=0. Default: 1 [cm^{-3}].

CoolingTestMaximumHNumberDensity (external) The maximum density in code units at x=‘‘DomainRightEdge[0]‘‘. Default: 1e6 [cm^{-3}].

CoolingTestMinimumMetallicity (external) The minimum metallicity at y=0. Default: 1e-6 [Z_{sun}].

CoolingTestMaximumMetallicity (external) The maximum metallicity at y=‘‘DomainRightEdge[1]‘‘. Default: 1 [Z_{sun}].

CoolingTestMinimumTemperature (external) The minimum temperature in Kelvin at z=0. Default: 10.0 [K].

CoolingTestMaximumTemperature (external) The maximum temperature in Kelvin at z=‘‘DomainRightEdge[2]‘‘. Default: 1e7 [K].

CoolingTestResetEnergies (external) An integer flag (0 or 1) to determine whether the grid energies should be continually reset after every iteration of the chemistry solver such that the temperature remains constant as the mean molecular weight varies slightly. Default: 1.

4.13 Other External Parameters

huge_number (external) The largest reasonable number. Rarely used. Default: 1e+20

tiny_number (external) A number which is smaller than all physically reasonable numbers. Used to prevent divergences and divide-by-zero in C++ functions. Modify with caution! Default: 1e-20.

An independent analog, `tiny`, defined in `fortran.def`, does the same job for a large family of FORTRAN routines. Modification of `tiny` must be done with caution and currently requires recompiling the code, since `tiny` is not a runtime parameter.

TimeActionParameter[#] Reserved for future use.

TimeActionRedshift[#] Reserved for future use.

TimeActionTime[#] Reserved for future use.

TimeActionType[#] Reserved for future use.

4.14 Other Internal Parameters

TimeLastRestartDump Reserved for future use.

TimeLastDataDump (internal) The code time at which the last time-based output occurred.

TimeLastHistoryDump Reserved for future use.

TimeLastMovieDump (internal) The code time at which the last movie dump occurred.

CycleLastRestartDump Reserved for future use.

CycleLastDataDump (internal) The last cycle on which a cycle dump was made

CycleLastHistoryDump Reserved for future use.

InitialCPUTime Reserved for future use.

InitialCycleNumber (internal) The current cycle

RestartDumpNumber Reserved for future use.

DataLabel[#] (internal) These are printed out into the restart dump parameter file. One Label is produced per baryon field with the name of that baryon field. The same labels are used to name data sets in HDF files.

DataUnits[#] Reserved for future use.

DataDumpNumber (internal) The identification number of the next output file (the 0000 part of the output name). This is used and incremented by both the cycle based and time based outputs. Default: 0

HistoryDumpNumber Reserved for future use.

MovieDumpNumber (internal) The identification number of the next movie output file. Default: 0

VersionNumber (internal) Sets the version number of the code which is written out to restart dumps.

CHAPTER

FIVE

PHYSICS MODULES IN ENZO

Here we will present an overview of the numerical techniques in Enzo's physics modules.

5.1 Active Particles: Stars, BH, and Sinks

There are many different subgrid models of star formation and feedback in the astrophysical literature, and we have included several of them in Enzo. There are also methods that include routines for black hole, sink, and Pop III stellar tracer formation. Here we give the details of each implementation and the parameters that control them.

5.1.1 Method 0: Cen & Ostriker

Source: star_maker2.F

This routine uses the algorithm from Cen & Ostriker (1992, ApJL 399, 113) that creates star particles when the following six criteria are met

1. The gas density is greater than the threshold set in the parameter `StarMakerOverDensityThreshold`. This parameter is in code units (i.e. overdensity with respect to the mean matter density)
2. The divergence is negative
3. The dynamical time is less than the cooling time or the temperature is less than 11,000 K. The minimum dynamical time considered is given by the parameter `StarMakerMinimumDynamicalTime` in *units of years*.
4. The cell is Jeans unstable.
5. The star particle mass is greater than `StarMakerMinimumMass`, which is in units of solar masses.
6. The cell does not have finer refinement underneath it.

These particles add thermal and momentum feedback to the grid cell that contains it until 12 dynamical times after its creation. In each timestep,

$$\begin{aligned} M_{\rm form} &= M_0[(1+x_1)\exp(-x_1) - (1+x_2)\exp(-x_2)] \\ x_1 &= (t-t_0)/t_{\rm dyn} \\ x_2 &= (t+dt-t_0)/t_{\rm dyn} \end{aligned}$$

of stars are formed, where M_0 and t_0 are the initial star particle mass and creation time, respectively.

- M_{ej} = M_{form} * `StarMakerEjectionFraction` of gas are returned to the grid and removed from the particle.
- M_{ej} * $v_{particle}$ of momentum are added to the cell.

- M_{form} * c^2 * `StarMakerEnergyToThermalFeedback` of energy is deposited into the cell.
- M_{form} * ((1 - Z_{star}) * `StarMetalYield` + M_{ej} * Z_{star}) of metals are added to the cell, where Z_{star} is the star particle metallicity. This formulation accounts for gas recycling back into the stars.

5.1.2 Method 1: Cen & Ostriker with Stochastic Star Formation

Source: star_maker3.F

This method is suitable for unigrid calculations. It behaves in the same manner as Method 1 except

- No Jeans unstable check
- **Stochastic star formation**: Keeps a global sum of "unfulfilled" star formation that were not previously formed because the star particle masses were under `StarMakerMinimumMass`. When this running sum exceeds the minimum mass, it forms a star particle.
- Initial star particle velocities are zero instead of the gas velocity as in Method 1.
- Support for multiple metal fields.

5.1.3 Method 2: Global Schmidt Law

Source: star_maker4.F

This method is based on the Kratsov (2003, ApJL 590, 1) paper that forms star particles that result in a global Schmidt law. This generally occurs when the gas consumption time depends on the local dynamical time.

A star particle is created if a cell has an overdensity greater than `StarMakerOverDensityThreshold`. The fraction of gas that is deposited into the star particle is dt/`StarMakerMinimumDynamicalTime` up to a maximum of 90% of the gas mass. Here the dynamical time is in *units of years*.

Stellar feedback is accomplished in the same way as Method 1 (Cen & Ostriker) but M_{form} = `StarMakerEjectionFraction` * (star particle mass).

5.1.4 Method 3: Population III Stars

Source: pop3_maker.F

This method is based on the Abel et al. (2007, ApJL 659, 87) paper that forms star particles that represents single metal-free stars. The criteria for star formation are the same as Method 1 (Cen & Ostriker) with the expection of the Jeans unstable check. It makes two additional checks,

1. The H_2 fraction exceeds the parameter `PopIIIH2CriticalFraction`. This is necessary because the cooling and collapse is dependent on molecular hydrogen and local radiative feedback in the Lyman-Werner bands may prevent this collapse.
2. If the simulation tracks metal species, the gas metallicity *in an absolute fraction* must be below `PopIIIMetalCriticalFraction`.

Stellar radiative feedback is handled by the *Radiative Transfer* module. By default, only hydrogen ionizing radiation is considered. To include helium ionizing radiation, set `PopIIIHeliumIonization` to 1. Supernova feedback through thermal energy injection is done by the *Star Particle Class*. The explosion energy is computed from the stellar mass and is deposited in a sphere with radius `PopIIISupernovaRadius` in *units of pc*. To track metal enrichment, turn on the parameter `PopIIISupernovaUseColour`.

5.1.5 Method 4: Sink particles

Source: sink_maker.C

5.1.6 Method 5: Radiative Stellar Clusters

Source: cluster_maker.F

This method is based on method 1 (Cen & Ostriker) with the Jeans unstable requirement relaxed. It is described in Wise & Cen (2009, ApJ 693, 984). The star particles created with this method use the adaptive ray tracing to model stellar radiative feedback. It considers both cases of Jeans-resolved and Jeans unresolved simulations. The additional criteria are

- The cell must have a minimum temperature of 10,000 K if the 6-species chemistry model (`MultiSpecies == 1`) is used and 1,000 K if the 9-species chemistry model is used.
- The metallicity must be above a critical metallicity (`PopIIIMetalCriticalFraction`) in absolute fraction.

When the simulation is Jeans resolved, the stellar mass is instantaneously created and returns its luminosity for 20 Myr. In the case when it's Jeans unresolved, the stellar mass follows the Cen & Ostriker prescription.

5.1.7 Method 6: Cen & Ostriker with no delay in formation

Source: star_maker7.F

This method relaxes the following criteria from the original Cen & Ostriker prescription. See Kim et al. (2011, ApJ 738, 54) for more details. It can be used to represent single molecular clouds.

- No Jeans unstable check
- No Stochastic star formation prescription that is implemented in Method 1.
- If there is a massive black hole particle in the same cell, the star particle will not be created.

The `StarMakerOverDensity` is in units of particles/cm^3 and not in overdensity like the other methods.

5.1.8 Method 7: Springel & Hernquist

Source: star_maker5.F

This method is based on the Springel & Hernquist method of star formation described in MNRAS, 339, 289, 2003. A star may be formed from a cell of gas if all of the following conditions are met:

1. The cell is the most-refined cell at that point in space.
2. The density of the cell is above a threshold.
3. The cell of gas is in the region of refinement. For unigrid, or AMR-everywhere simulations, this corresponds to the whole volume. But for zoom-in simulations, this prevents star particles from forming in areas that are not being simulated at high resolution.

If a cell has met these conditions, then these quantities are calculated for the cell:

- **Cell star formation timescale (Eqn 21 from Springel & Hernquist).** t_0^* and $\rho_{\rm th}$ are inputs to the model, and are the star formation time scale and density scaling value, respectively. Note that $\rho_{\rm th}$ is not the same as the critical density for star formation listed above. ρ is the gas density of the cell.

t_{ast}(rho)=t_0^{ast}left(frac{rho}{rho_{mathrm{th}}}right)^{-1/2}

- **Mass fraction in cold clouds, x (see Eqns. 16 and 18).** y is a dimensionless quantity calculated as part of the formulation; $u_{\rm SN} \equiv (1-\beta)\beta^{-1}\epsilon_{\rm SN}$ is the energy released from supernovae back into the gas (note that whether or not the energy is *actually* returned to the gas depends on if `StarFormationFeedback` is turned on or not); β is the fraction of stars that go supernova soon after formation; $\epsilon_{\rm SN}$ is the energy released from a nominal supernova and is set to 4e48 ergs; and finally $\Lambda(\rho, T, z)$ is the cooling rate of the cell of gas.

$$y \equiv \frac{t_* \Lambda(\rho, T, z)}{\rho[\beta u_{\rm SN} - (1-\beta)u_{\rm SN}]}$$
$$x = 1 + \frac{1}{2y} - \sqrt{\frac{1}{y} + \frac{1}{4y^2}}$$

Finally, a star particle of mass m_* is created with probability p_* (see Eqn. 39). For a cell, the quantity p_* is calculated (below) and compared to a random number p drawn evenly from [0, 1). If $p_* > p$, a star is created. m_* is a parameter of the model and is the minimum and only star mass allowed; m is the mass of gas in the cell; Δt is the size of the simulation time step that is operative for the cell (which changes over AMR levels, of course).

$$p_* = \frac{m}{m_*}\left\{1 - \exp\left[-\frac{(1-\beta)x\Delta t}{t_*}\right]\right\}$$

If this star formula is used with AMR, some caution is required. Primarily, the AMR refinement can not be too aggressive. Values of `OverDensityThreshold` below 8 are not recommended. This is because if refinement is more aggressive than 8 (i.e. smaller), the most-refined cells, where star formation should happen, can have less mass than a root-grid cell, and for a deep AMR hierarchy the most refined cells can have mass below m_*. Put another way, with aggressive refinement the densest cells where stars *should* form may be prevented from forming stars simply because their total mass is too low. Keeping `OverDensityThreshold` at 8 or above ensures that refined cells have at least a mass similar to a root-grid cell.

Another reason for concern is in AMR, Δt changes with AMR level. Adding a level of AMR generally halves the value of Δt, which affects the probability of making a star. In a similar way, a small value of `CourantSafetyFactor` can also negatively affect the function of this star formula.

5.1.9 Method 8: Massive Black Holes

Source: mbh_maker.C

5.1.10 Method 9: Population III stellar tracers

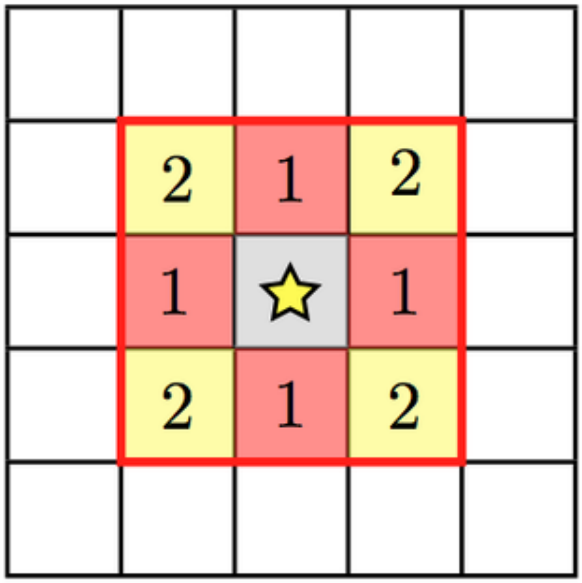
Source: pop3_color_maker.F

5.1.11 Distributed Stellar Feedback

The following applies to Methods 0 (Cen & Ostriker) and 1 (+ stochastic star formation).

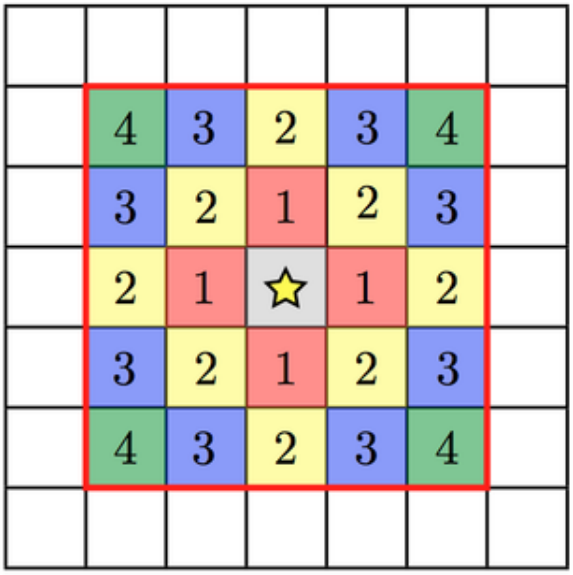
The stellar feedback can be evenly distributed over the neighboring cells if `StarFeedbackDistRadius` > 0. The cells are within a cube with a side `StarFeedbackDistRadius+1`. This cube can be cropped to the cells that are `StarFeedbackDistCellStep` cells away from the center cell, counted only in steps in Cartesian directions. Below we show a couple of *two-dimensional* examples. The number on the cells indicates the number cell steps each is from the central cell.

- `StarFeedbackDistRadius = 1`



Only cells with a step number <= `StarFeedbackDistCellStep` have feedback applied to them. So, `StarFeedbackDistCellStep` = 1 would result in only the cells marked with a "1" receiving energy. In three-dimensions, the eight corner cells in a 3x3x3 cube would be removed by setting `StarFeebackDistCellStep` = 2.

- `StarFeedbackDistRadius = 2`



Same as the figure above but with a radius of 2.

Feedback regions cannot extend past the host grid boundaries. If the region specified will extend beyond the edge of the grid, it is recentered to lie within the grid's active dimensions. This conserves the energy injected during feedback but results in the feedback sphere no longer being centered on the star particle it originates from. Due to the finite size of each grid, we do not recommend using a `StarFeedbackDistRadius` of more than a few cells.

Also see *Star Formation and Feedback Parameters*.

5.1.12 Notes

The routines included in `star_maker1.F` are obsolete and not compiled into the executable. For a more stable version of the algorithm, use Method 1.

5.2 Hydro and MHD Methods

There are four available methods in Enzo for calculating the evolution of the gas with and without magnetic fields. Below is a brief description of each method, including the parameters associated with each one and a link to further reading.

5.2.1 Method 0: Piecewise Parabolic Method (PPM)

Source: Grid_SolvePPM_DE.C

The PPM scheme uses a parabolic function to estimate the left and states of the Godunov problem. This more accurately represents both smooth gradients and discontinuities over linear interpolation, i.e. PLM.

Parameters

Main call: `HydroMethod = 0`

`RiemannSolver`: specifies the type of solver, where the following only works with the PPM solver.

1. HLL (Harten-Lax-van Leer) a two-wave, three-state solver with no resolution of contact waves. This is the most diffusive of the available three solvers in PPM. *New for version 2.1*
4. HLLC (Harten-Lax-van Leer with Contact) a three-wave, four-state solver with better resolution of contacts. The most resilient to rarefaction waves (e.g. blastwave interiors). *New for version 2.1*
5. **Default** Two-shock approximation. Iterative solver.

`RiemannSolverFallback`: allows for the Riemann solver to "fallback" to the more diffusive HLL solver when negative energies or densities are computed. Only applicable when using the HLLC and Two-shock solvers. The fluxes in the failing cell are recomputed and used in the Euler update of the gas quantities. *New for version 2.1*

`ConservativeReconstruction`: When interpolating (PPM) to the left and right states, interpolation occurs in the conserved variables (density, momentum, and energy) instead of the primitive variables (density, velocity, and pressure). This results in more accurate results in unigrid simulations but can cause errors with AMR. See Section 4.2.2 (steps 1-5) and Appendices A1 and B1 in Stone et al. (2008, ApJS 178, 137). *New for version 2.1*

`DualEnergyFormalism`: allows the total and thermal energy to be followed seperately during the simulation. Helpful when the velocities are high such that $E_{total} >> E_{thermal}$.

`PPMFlatteningParameter`

`PPMSteepeningParameter`

Links

P. R. Woodward and P. Colella. "A piecewise parabolic method for gas dynamical simulations," *J. Comp. Phys*, 54:174, 1984 link

5.2.2 Method 2: ZEUS

Source: ZeusSource.C, Zeus_xTransport.C, Zeus_yTransport.C, Zeus_zTransport.C, Grid_ZeusSolver.C, ZeusUtilities.C

ZEUS is a finite-difference method of solving hyperbolic PDEs instead of solving the Godunov problem. It is a very robust but relatively diffusive scheme.

Parameters

Main call: `HydroMethod = 2`

`ZEUSQuadraticArtificialViscosity`

`ZEUSLinearArtificialViscosity`

Links

J. M. Stone and M. L. Norman. "Zeus-2D: A radiation magnetohydrodynamics code for astrophysical flows in two space dimensions. I. The hydrodynamics algorithms and tests." *The Astrophysical Journal Supplement*, 80:753, 1992 link

J. M. Stone and M. L. Norman. "Zeus-2D: A radiation magnetohydrodynamics code for astrophysical flows in two space dimensions. II. The magnetohydrodynamic algorithms and tests." *The Astrophysical Journal Supplement*, 80:791, 1992 link

5.2.3 Method 3: MUSCL

New in version 2.0. The MUSCL [1] scheme is a second-order accurate extensive of Godunov's method for solving the hydrodynamics in one dimension. The implementation in Enzo uses second-order Runge-Kutta time integration. In principle, it can use any number of Riemann solvers and interpolation schemes. Here we list the compatible ones that are currently implemented.

Parameters

Parameter file call: `HydroMethod = 3`

`RiemannSolver`: specifies the type of solver, where the following only works with the MUSCL solver.

1. HLL (Harten-Lax-van Leer): a two-wave, three-state solver with no resolution of contact waves.
3. LLF (Local Lax-Friedrichs) is based on central differences instead of a Riemann problem. It requires no characteristic information. This is the most diffusive of the available three solvers in MUSCL.
4. HLLC (Harten-Lax-van Leer with Contact): a three-wave, four-state solver with better resolution of contacts. The most resilient to rarefaction waves (e.g. blastwave interiors).

If negative energies or densities are computed, the solution is corrected using a more diffusive solver, where the order in decreasing accuracy is HLLC -> HLL -> LLF.

`ReconstructionMethod`: specifies the type of interpolation scheme used for the left and right states in the Riemann problem.

0. PLM: **default**

[1] Monotone Upstream-centered Schemes for Conservation Laws

1. PPM: Currently being developed.

5.2.4 Method 4: MHD

New in version 2.0. The MHD scheme uses the same MUSCL framework as Method 3. To enforce $\div \cdot B = 0$, it uses the hyperbolic cleaning method of Dedner et al. (2002, JCP 175, 645).

Parameters

Parameter file call: `HydroMethod = 4`

5.2.5 Notes

`HydroMethod = 1` was an experimental implementation that is now obsolete, which is why it is skipped in the above notes.

5.3 Cooling and Heating of Gas

Enzo features a number of different methods for including radiative cooling. These range from simple tabulated, analytical approximations to very sophisticated non-equilibrium primoridal chemistry. All of these methods require the parameter `RadiativeCooling` be set to 1. Other parameters are required for using the various methods, which are described below.

5.3.1 MultiSpecies = 0: Sarazin & White

Source: solve_cool.F, cool1d.F

`RadiativeCooling` = 1

`MultiSpecies` = 0

This method uses an analytical approximation from Sarazin & White (1987, ApJ, 320, 32) for a fully ionized gas with metallicity of 0.5 solar. This cooling curve is valid over the temperature range from 10,000 K to 10^9 K. Since this assumes a fully ionized gas, the cooling rate is effectively zero below 10,000 K.

Note: In order use this cooling method, you must copy the file, cool_rates.in, from the input directory into your simulation directory.

5.3.2 MultiSpecies = 1, 2, or 3: Primordial Chemistry and Cooling

Source: multi_cool.F, cool1d_multi.F

This method follows the nonequilibrium evolution of primordial (metal-free) gas. The chemical rate equations are solved using a semi-implicit backward differencing scheme described by Abel et al. (1997, New Astronomy, 181) and Anninos et al. (1997, New Astronomy, 209). Heating and cooling processes include atomic line excitation, recombination, collisional excitation, free-free transitions, Compton scattering of the cosmic microwave background and photoionization from a variety of metagalactic UV backgrounds. For `MultiSpecies` > 1, molecular cooling is also included and UV backgrounds that include photodissociation may also be used. Numerous chemistry and cooling rates have been added or updated. For the exact reference for any given rate, users are encouraged to consult *calc_rates.F*.

1. Atomic

   `RadiativeCooling` = 1

   `MultiSpecies` = 1

   Only atomic species, H, H^+, He, He^+, He^{++}, and e^- are followed. Since molecular species are not treated, the cooling is effectively zero for temperatures below roughly 10,000 K.

2. Molecular Hydrogen

   `RadiativeCooling` = 1

   `MultiSpecies` = 2

   Along with the six species above, H_2, H_2^+, and H^- are also followed. In addition to the rates described in Abel et al. (1997) and Anninos et al. (1997), H2 formation via three-body reactions as described by Abel, Bryan, and Norman (2002, Science, 295, 93) is also included. This method is valid in the temperature range of 1 K to 10^8 K and up to number densities of roughly 10^9 cm^{-3}. Additionally, three-body heating (4.48eV per molecule formed or dissociated) is added as appropriate.

3. Deuterium

   `RadiativeCooling` = 1

   `MultiSpecies` = 3

   In addition to the nine species solved with `MultiSpecies` = 2, D, D^+, and HD are also followed. The range of validity is the same as for `MultiSpecies` = 2.

5.3.3 Metal Cooling

Three distinct methods to calculate the cooling from elements heavier than He exist. These are selected by setting the `MetalCooling` parameter to 1, 2, or 3.

1. John Wise's metal cooling.

   `RadiativeCooling` = 1

   `MetalCooling` = 1

2. Cen et al (1995) cooling. This uses output from a Raymond-Smith code to determine cooling rates from T > 10^4K. No ionizing background is used in computing cooling rates. This method has not been extensively tested in the context of Enzo.

   `RadiativeCooling` = 1

   `MetalCooling` = 2

3. Cloudy cooling.

   Source: cool1d_cloudy.F

   `RadiativeCooling` = 1

   `MetalCooling` = 3

   `MultiSpeces` = 1, 2, or 3

   Cloudy cooling operates in conjunction with the primordial chemistry and cooling from `MultiSpecies` set to 1, 2, or 3. As described in Smith, Sigurdsson, & Abel (2008), Cloudy cooling interpolates over tables of precomputed cooling data using the Cloudy photoionization software (Ferland et al. 1998, PASP, 110, 761, http://nublado.org). The cooling datasets can be from one to five dimensional. The range of validity will depends on the dataset used.

(a) Temperature

(b) Density and temperature.

(c) Density, metallicity, and temperature.

(d) Density, metallicity, electron fraction, and temperature.

(e) Density, metallicity, electron fraction, redshift of UV background, and temperature.

See *Cloudy Cooling* for additional parameters that control the behavior of the Cloudy cooling. For more information on obtaining or creating Cloudy cooling datasets, contact Britton Smith (brittonsmith@gmail.com).

5.3.4 UV Meta-galactic Backgrounds

Source: RadiationFieldCalculateRates.C

A variety of spatially uniform photoionizing and photodissociating backgrounds are available, mainly by setting the parameter `RadiationFieldType`. These radiation backgrounds are redshift dependent and work by setting the photoionization and photoheating coeffiecients for H, He, and He^+. See *Background Radiation Parameters* for the additional parameters that control the UV backgrounds.

5.4 Radiative Transfer

New in version 2.0.

5.4.1 Adaptive Ray Tracing

Solving the radiative transfer equation can be computed with adaptive ray tracing that is fully coupled with the hydrodynamics and energy / rate solvers. The adaptive ray tracing uses the algorithm of Abel & Wandelt (2002) that is based on the HEALPix framework.

For the time being, a detailed description and test suite can be found in the paper Wise & Abel (2011, MNRAS 414, 3458).

5.4.2 Flux Limited Diffusion

More details can be found in the paper Reynolds et al. (2009, Journal of Computational Physics 228, 6833).

5.5 Shock Finding

New in version 2.1. *Source: Grid_FindShocks.C*

Shock finding is accomplished using one of four methods. The primary method uses a coordinate-unsplit temperature jump (method 1), as described in Skillman et. al. 2008 with the exception that instead of searching across multiple grids for the pre- and post-shock cells, we terminate the search at the edge of the ghost zones within each grid.

Shock finding is controlled by the `ShockMethod` parameter, which can take the following values:

0 - Off

1 - Unsplit Temperature Jumps

2 - Dimensionally Split Temperature Jumps

3 - Unsplit Velocity Jumps

4 - Dimensionally Split Velocity Jumps

When `ShockMethod` nonzero, this will create a "Mach" field in the output files.

Note: Method 1 has been used the most by the developer, and therefore is the primary method. Method 2 has been tested quite a bit, but the downsides of using a dimensionally split method are outlined in the above paper. Methods 3 and 4 are more experimental and will run, but results may vary.

Additional Shock Finding Parameters:

`ShockTemperatureFloor` - When calculating the mach number using temperature jumps, set the temperature floor in the calculation to this value.

`StorePreShockFields` - Optionally store the Pre-shock Density and Temperature during data output.

CHAPTER

SIX

DEVELOPER'S GUIDE

Here we will document some of the actual code details in Enzo, and how to perform basic (and not so basic) tasks in Enzo. Before tackling any modification, please read the *Enzo Primary References* a basic knowledge of AMR and numerical methods is assumed throughout this documentation.

6.1 Introduction to Enzo Modification

Note: This is not a comprehensive document, but it does cover some of the grounds of modifying Enzo. Please don't hesitate to email the users' mailing list with any further questions about Enzo, Mercurial, or how to write and execute new test problems.

If this is the first time you've opened the hood to Enzo, welcome. If you're an old hand and have already added new physics to it, welcome back.

Enzo is an extremely powerful piece of software, but by no means a complete representation of the observable universe. It's quite likely that there will be some piece of physics that you'll want to model, and these span a broad range of software complexities. In all cases, whether it's a mildly invasive change such as a new background heating model or extremely invasive like adding relativistic non-neutral multi-fluid plasma physics, we strongly recommend taking advantage of some basic tools. These are outlined in the sections that follow. These tools prevent the developer from breaking existing features (which is far easier than one would expect), keeping track of your changes, and sharing those changes with others. We strongly recommend you start with getting LCATest running before you start programming, so mistakes can be caught early.

Additionally in the Tutorials section you'll see a pair of flow chart tools that are intended as educational tools, and several descriptions on how to actually add various components to the code. It is intended that these will be at least read in order, as doing complex things with the code require the ability to do the simpler things.

We are very happy to accept patches, features, and bugfixes from any member of the community! Enzo is developed using mercurial, primarily because it enables very easy and straightforward submission of changesets. We're eager to hear from you, and if you are developing Enzo, please subscribe to the users' mailing list:

http://groups.google.com/group/enzo-users

This document describes how to use Mercurial to make changes to Enzo, how to send those changes upstream, and how to navigate the Enzo source tree.

6.1.1 Mercurial Introduction

If you're new to Mercurial, these three resources are pretty great for learning the ins and outs:

- http://hginit.com
- http://hgbook.red-bean.com/read/
- http://mercurial.selenic.com/

The major difference between Mercurial (and other distributed version control systems) and centralized version control systems (like CVS, RCS, SVN) is that of the directed acyclic graph (DAG). Rather than having a single timeline of modifications, Mercurial (or "hg") can have multiple, independent streams of development.

There are a few concepts in Mercurial to take note of:

Changesets Every point in the history of the code is referred to as a changeset. These are specific states of the code, which can be recovered at any time in *any* checkout of the repository. These are analagous to revisions in Subversion.

Children If a changeset has changesets that were created from its state, those are called children. A changeset can have many children; this is how the graph of development branches.

Heads Every changeset that has no children is called a head.

Branches Every time the DAG branches, these are branches. Enzo also uses "named branches," where the branches have specific identifiers that refer to the feature under development or some other characteristic of a line of development.

On the Google Code wiki there is a list of active branches.

When you check out the Enzo repository, you receive a full and complete copy of the entire history of that repository; you can update between revisions at will without ever touching the network again. This allows not only for network-disconnected development, but it also means that if you are creating some new feature on top of Enzo you can (and should!) conduct local version control on your development. Until you choose explicitly to share changes, they will remain private to your checkout of the repository.

6.1.2 Enzo Source Trees

Enzo has two primary repositories, the "stable" repository which is curated and carefully modified, and the "development" repository which is where active development occurs. Please note that while we test and verify the results of the "stable" repository, the "unstable" repository is not guaranteed to be tested, verified, or even to provide correct answers.

Note: The "stable" Enzo source tree is *not* for general development. If you want to contribute to Enzo, make your changes to a fork of the development repository!

To conceptually – and technically! – separate these two repositories, they also live in different places. We keep the stable repository at Google Code, and the development repository at BitBucket. Enzo is (as of 2.1) developed in a relatively simple fashion:

1. On BitBucket, developers "fork" the primary development repository.
2. When a piece of work is ready to be shared, a "pull request" is issued. This notifies the current set of Enzo curators that a new feature has been suggested for inclusion.
3. After these features have been accepted, they are pulled into the development branch. New features will be aggregated into patch releases on the "stable" branch.
4. When a new patch release is issued, the current development branch is pushed to the "stable" branch on Google Code.

The idea here is that there is a double firewall: the development repository is very high-cadence and with high-turnover, but the stable repository is much slower, more carefully curated, and inclusions in it are well-tested.

- Stable code lives at: http://enzo.googlecode.com/
- Development code lives at: http://bitbucket.org/enzo/

6.1.3 How To Share Changes

Sharing your changes to Enzo is easy with Mercurial and the BitBucket repository.

Go here:

http://bitbucket.org/enzo/enzo-dev/fork

Now, clone your new repository. Make your changes there. Now go back and issue a pull request. For instance, you might do something like this:

1. Clone Enzo, make a few changes, commit them, and decide you want to share.
2. Fork the main enzo repository at that link.
3. Now, edit `.hg/hgrc` to add a new path, and push to that path.
4. Go to the BitBucket URL for your new repository and click "Pull Request". Fill it out, including a summary of your changes, and then submit. It will get evaluted – and it might not get accepted right away, but the response will definitely include comments and suggestions.

That's it! If you run into any problems, drop us a line on the Enzo Users' Mailing List.

6.1.4 How To Use Branching

Warning: In most cases, you do *not* need to make a new named branch! Do so with care, as it lives forever.

If you are planning on making a large change to the code base that may not be ready for many, many commits, or if you are planning on breaking some functionality and rewriting it, you can create a new named branch. You can mark the current repository as a new named branch by executing:

```
$ hg branch new_feature_name
```

The next commit and all subsequent commits will be contained within that named branch. At this point, add your branch here:

http://code.google.com/p/enzo/wiki/ActiveBranches

To merge changes in from another branch, you would execute:

```
$ hg merge some_other_branch
```

Note also that you can use revision specifiers instead of "some_other_branch". When you are ready to merge back into the main branch, execute this process:

```
$ hg merge name_of_main_branch
$ hg commit --close-branch
$ hg up -C name_of_main_branch
$ hg merge name_of_feature_branch
$ hg commit
```

When you execute the merge you may have to resolve conflicts. Once you resolve conflicts in a file, you can mark it as resolved by doing:

```
$ hg resolve -m path/to/conflicting/file.py
```

Please be careful when resolving conflicts in files.

Once your branch has been merged in, mark it as closed on the wiki page.

6.1.5 The Patch Directory

If you are experimenting with a code change or just debugging, then the patch directory, found in the top level of your Enzo directory, may be of use. Files put in here are compiled in preference to those in `/src/enzo`, so you can implement changes without overwriting the original code. To use this feature, run `make` from inside `/patch`. You may need to add `-I../src/enzo` to the `MACH_INCLUDES` line of your machine makefile (e.g. `Make.mach.triton`) to ensure the .h files are found when compiling.

As an example, suppose you wish to check the first few values of the acceleration field as Enzo runs through `EvolveLevel.C`. Copy `EvolveLevel.C` from `/src/enzo` into `/patch` and put the appropriate print statements throughout that copy of the routine. Then recompile Enzo from inside the patch directory. When you no longer want those changes, simply delete EvolveLevel.C from `/patch` and the next compile of the code will revert to using the original `/src/enzo/EvolveLevel.C`. If you make adjustments you wish to keep, just copy the patch version of the code into `/src/enzo` to replace the original.

6.1.6 How To Include Tests

If you have added any new functionality, you should add it as a test in the directory tree `run/` under the (possibly new!) appropriate directory. Your test file should consist of:

- A parameter file, ending in the extension `.enzo`
- A file of `notes.txt`, describing the problem file, the expected results, and how to verify correctness
- A test file, using the yt extension `enzo_test`, which verifies correctness. (For more information on this, see some of the example test files.)
- (optional) Scripts to plot the output of the new parameter file.

Please drop a line to the mailing list if you run into any problems!

6.2 Programming Guide

There are several coding practices that we should adhere to when programing for Enzo. Some are style, some are more important for the health of the code (and other Enzo users' projects/sanity).

6.2.1 Remember that other programmers will read your code

> "Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?" –Brian Kernighan "The Elements of Programming Style", 2nd edition, chapter 2

6.2.2 File Naming Convention

With very few exceptions, Enzo has a one function per file layout, with the file name being the function name. Object methods have the object name prepended to the beginning, such as the member of the grid class `SolveHydroEquations` lives in the file `Grid_SolveHydroEquations.C`.

This does create a large number of files. Familiarity with `grep` or `ack` and pipes like `ls -1 |grep` are essential.

Internal capitalization is used for C files, all lowercase with underscores for fortran files and header files. All Fortran files end with .F.

6.2.3 Comments

At the very least, put the following in things at the top of each of your functions:

- Your name
- The date you wrote it. If you modified it in a significant way, note the date and modification.
- Effects on global or class member variables.
- Variable names that are not obvious. As a rule of thumb, the name is not obvious.
- Primary references where applicable.

Two more rules:

- Write your comments now. You will not have time to come back and clean it up later.
- If you change something, change the comments. Now. Wrong comments are worse than no comments.

6.2.4 float is double

One must constantly be wary of the possibility of built in C types to be re-defined to higher precision types. This is outlined in *Variable precision in Enzo*.

6.2.5 Header Files

Header files must be included in the correct order. This is due, among other things, to the redefinition of float which is done in `macros_and_parameters.h`. This must be done before Enzo headers, but after external libraries. The order should be as follows:

```
#include "ErrorExceptions.h"
#include "svn_version.def"
#include "performance.h"
#include "macros_and_parameters.h"
#include "typedefs.h"
#include "global_data.h"
#include "units.h"
#include "flowdefs.h"
#include "Fluxes.h"
#include "GridList.h"
#include "ExternalBoundary.h"
#include "Grid.h"
#include "Hierarchy.h"
#include "LevelHierarchy.h"
#include "TopGridData.h"
```

```
#include "communication.h"
#include "CommunicationUtilities.h"
```

6.2.6 Accessing BaryonField

Access data in the BaryonField array as is described in the page on *Accessing Data in BaryonField*.

6.2.7 Accessing the Hierarchy

The hierarchy should be traversed as described in *Getting Around the Hierarchy: Linked Lists in Enzo*.

6.2.8 enum

The enum construct in C has no standardized size, which can cause problems when using 64 bit integers. Direct integer assignment should be used instead. This also has the added advantage of making explicit the values of parameters that are also used in parameter files. The typical idiom should be:

```
#ifdef SMALL_INTS
typedef int hydro_method;
#endif
#ifdef LARGE_INTS
typedef long_int hydro_method;
#endif
const hydro_method
  PPM_DirectEuler      = 0,
  PPM_LagrangeRemap    = 1,
  Zeus_Hydro           = 2,
  HD_RK                = 3,
  MHD_RK               = 4,
  HydroMethodUndefined = 5;
```

6.3 Adding a new parameter to Enzo

If your parameter is only used for a problem initialization, this page is not relevant for you. You should just read it in during `ProblemInitialize.C` where `Problem` is replaced by the name of the problem type.

If you're extending Enzo for any reason, you'll probably need to add a new switch or parameter to the code. Currently, this page describes the simplest, most brute force method. There are four files you'll need to edit to make this happen.

- `global_data.h` holds all the global data. It's included in almost all Enzo files. Your parameter should be added like this:

  ```
  EXTERN int MyInt;
  EXTERN float MyFloat;
  ```

  EXTERN is a macro that either maps to extern if `USE_STORAGE` is defined, or nothing if `USE_STORAGE` is not defined. `USE_STORAGE` is defined in `enzo.C` before the inclusion of `global_data.h`, and undefined after.

- `SetDefaultGlobalValues.C` sets the default global values. Set your value here.

- `ReadParameterFile.C` reads the parameter file. In this routine, each line is read from the file and is compared to the given parameters with `sscanf()`. Your line should look like this:

```
ret += sscanf(line, "MyFloat       = %"FSYM, &MyFloat);
ret += sscanf(line, "MyInt         = %"ISYM, &MyInteger);
```

and should be inserted somewhere in the loop where line is relevant. Note that `ISYM` and `FSYM` are the generalized integer and float I/O macro, which exist to take care of the dynamic hijacking of 'float'. See this page for more information: *Variable precision in Enzo*. The `ret +=` controls whether the line has been read, or if Enzo should issue a warning about the line. Note also that `sscanf()` is neutral to the amount of consecutive whitespace in the format string argument.

- `WriteParameterFile.C` writes the restart parameter file. Somewhere before the end of the routine, you should add something that looks like

```
fprintf(fptr, "MyFloat       = %"GSYM"\n", MyFloat);
fprintf(fptr, "MyInt         = %"ISYM"\n", MyInt);
```

Note the use of quotes here and in the previous code snippet. This is correct.

For testing purposes you can verify that your new parameter is being correctly read in by adding a line like this at the bottom of `ReadParameterFile.C`:

```
  fprintf(stdout, "MyFloat %f MyInt %d\n", MyFloat, MyInt);
  return SUCCESS;
}
```

6.4 How to add a new baryon field

If you wish to add a new BaryonField array – for instance, to track the Oxygen fraction – you'll need to do a few things.

1. Add it to the `field_type` structure
2. Define it in your problem type. *Adding a new Test Problem.*.
3. Do something with it. *Adding a new Local Operator.*.
4. If you need to advect a species as a color field, you will have to investigate how that works. Specifically, the means of conservation – by fraction of by density – as well as the inputs into the hydro solver.

6.5 Variable precision in Enzo

In order to provide some global control over variable precision, Enzo uses a set of macros that control how the code treats integer and floating-point precision by overriding the float and int data types, and by introducing the `FLOAT` macro. This is a major sticking point for new users to the code, and this page is an attempt to clarify the issue as much as possible.

6.5.1 Floating-point precision

There are two different kinds of floating-point quantities in Enzo, those that explicitly deal with positional information (grid edges/locations, cell sizes, particle positions, and so on), and those that deal with non-position information (baryon density, temperature, velocity, etc.) Any variables that deal with position information should be declared as the `FLOAT` data type. For example:

```
FLOAT xpos, ypos, zpos;
```

A quantity that deals with non-positional information would be declared using the `float` data type:

```
float cell_HI_density, cell_H2I_density, cell_temperature;
```

The actual precision of `float` and `FLOAT` are controlled by the Makefile system (see *Obtaining and Building Enzo*.) To set the non-positional precision to 64-bit (double), you would issue this command:

```
make precision-64
```

before compiling the code. Similarly, to set the positional precision to 64-bit (double), you would issue this command:

```
make particles-64
```

The allowable values for non-positional precision are 32 and 64 bits, and for positional precision are 32, 64, and 128 bits. It is not recommended that you use particles-128 unless you need more than 30 or so levels of AMR, since `long double` arithmetic generally requires software libraries and can be very slow. Also note that the 128-bit precision code is not terribly stable, and only works on some systems (and with some sets of compilers). Use this with **extreme caution**.

Mixing ‘‘float‘‘ and ‘‘FLOAT‘‘: One can mix the `float` and `FLOAT` data types, but some care is required since the two are not necessarily the same precision. Compilers will generally promote the variables to the higher precision of the two, but this is not always true. The Enzo developers have chosen to make the assumption that the precision of `FLOAT` is always the same as, or greater than, the precision of `float`. So, when precision is critical or when mixing `float` and `FLOAT`, we recommend that you always promote all variables to `FLOAT`. Regardless, it is a good idea to check that your code is producing sensible results.

6.5.2 Integer precision

There is only one commonly-used type of integer in Enzo, which is `int`. This is controlled by the the integers-makefile command. For example,

```
make integers-64
```

would force all ints to be 64-bit integers (`long int`). The allowable integer values are 32 and 64 bit. In general, the only time one would need 64-bit ints is if you are using more than 2^{31} particles, since signed integers are used for the particle index numbers, and chaos will ensue if you have duplicate (or, worse, negative) particle indices.

6.5.3 Precision macros and printf/scanf

In order to keep the printf family of commands happy, Enzo uses several macros. `ISYM` is used for integers, `FSYM` and `ESYM` for `float`, and `PSYM` and `GSYM` for `FLOAT` (the latter of each pair outputs floats in exponential notation). Additionally, when writing `FLOAT` data to a file that will be read back in by Enzo (such as to the parameter or hierarchy file), it is wise to use `GOUTSYM`. In a printf or scanf statement, this macro will be replaced with the actual string literal statement.

An example of this usage macro in a printf statement to write out a float is:

```
printf("Hello there, your float value is %"FSYM".\n", some_float);
```

and to read in a set of three position coordinates using scanf out of a string named line:

```
sscanf(line,"PartPos  = %"PSYM" %"PSYM" %"PSYM, &XPOS, &YPOS, &ZPOS);
```

Note the somewhat counterintuitive use of quotation marks after the 3rd `PSYM`. For a large number of examples of how to use these macros, please refer to the files `ReadParameterFile.C` and `WriteParameterFile.C` in the Enzo source code.

6.5.4 The Fortran-C++ interface

It is critical to make sure that if you are interfacing Fortran and C/C++ code, the variable precision agrees between the two languages. Compilers do not attempt to ensure that calls from C/C++ to Fortran make any sense, so the user is manifestly on their own. To this end, when writing Fortran code, the data type `real` corresponds to `float`, and `REALSUB` corresponds to `FLOAT`. Mismatching these data types can cause misalignment in the data that is being passed back and forth between C/C++ and Fortran code (if the precision of `float` and `FLOAT` are not the same), and will often result in nonsense values that will break Enzo elsewhere in the code. This can be particularly tricky to debug if the values are not used immediately after they are modified!

6.5.5 If you need more details...

If you need more detailed information on this particular subject, there is no substitute for looking at the source code. All of these macros are defined in the Enzo source code file `macros_and_parameters.h`. Just look for this comment:

```
/* Precision-dependent definitions */
```

There are many examples of using the IO macros in `ReadParameterFile.C` and `WriteParameterFile.C`.

Also, please note that this set of macros may be replaced with a more robust set of macros in future versions.

6.6 Adding new refinement criteria

1. Add any new parameters you might need. (See *Adding a new parameter to Enzo*.)
2. Write your code to flag the cells
3. Call your method.

The first point has been discussed elsewhere.

6.6.1 Writing your code to flag cells

Your code needs to do a couple things:

1. Be named `FlagCellsToBeRefinedByXXXXXX`, where `XXXXXX` is your criterion.
2. Increment `FlaggingField[i]`
3. Count and return the number of flaggged cells.
4. Return -1 on error.

Your code to do the cell flagging can be a grid method.

A minimal code should look like this:

```
int grid::FlagCellsToBeRefinedByDensityOverTwelve(){

  int NumberOfFlaggedCells = 0;
  for( int i = 0; i< GridDimension[0]*GridDimension[1]*GridDimension[2]; i++ ){
     if( BaryonField[0][i] > 12 ){
       FlaggingField[i] ++;
       NumberOfFlaggedCells ++;
     }
  }
```

```
  return NumberOfFlaggedCells;

}
```

6.6.2 Call your method

Edit the file `Grid_SetFlaggingField.C` In this routine, there's a loop over the `CellFlaggingMethod` array. In this loop, you'll see code like this:

```
  /* ==== METHOD 47: By Density over 12 ==== */

case 47:

  NumberOfFlaggedCells = this->FlagCellsToBeRefinedByDensityOverTwelve();
  if (NumberOfFlaggedCells < 0) {
    fprintf(stderr, "Error in grid->FlagCellsToBeRefinedByDensityOverTwelve.\n");
    return FAIL;
  }
  break;
```

So we need to add a few things.

- Add a new case statement to the switch construct.
- Set `NumberOfFlaggedCells` via the method described above.
- Don't forget the break; statement.
- Check for errors.

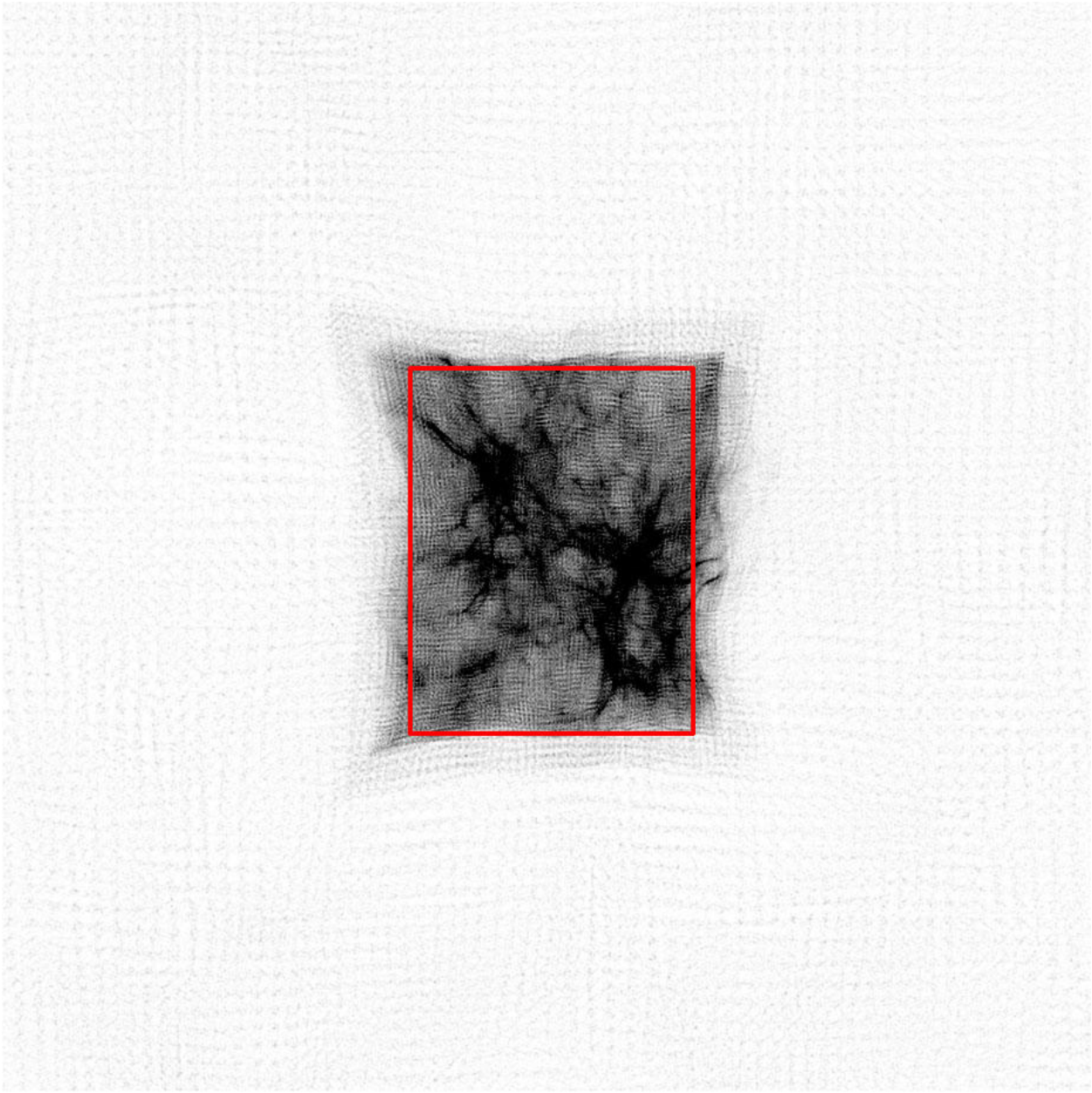
6.7 Auto adjusting refine region

6.7.1 Problem

In nested grid simulations, massive particles from outside the finest nested initial grid can migrate into the refine region. This may cause artifical collapses in halos whose potential is dominated by one or more massive particle. To avoid this in the past, the refine region was set to the Lagrangian volume of the halos of interest at the final redshift.

6.7.2 Solution

On every top-level timestep, we can search for these massive particles inside the current refine region and adjust the refine region to exclude these particles. The covering volume of the highest resolution particles may have been sheared and have an arbitrary shape. We adjust the refine region to have faces just inside of the innermost, relative to the center of the refine region, massive particles. Below is an illustration of this new region.



Here is the logic that we have taken to adjust the refine region because it is not a trivial min/max of the positions of the massive particles. If we were to take the maximum distance of the highest resolution particles from the refine region center, we would obtain a minimum covering volume that contains all high resolution particles, which is not desired. We will incrementally shrink the region by a cell width on the level with the finest nested initial grid.

1. Find the mass of the highest resolution particle, M_{min}.
2. Create a list of any particles with a mass > M_{min} inside the current refine region. This list is unique on each processor.
3. Because we will incrementally reduce the refine region by cell widths, it is convenient to convert the massive particle positions to integers in units of these cell widths.
4. Loop while any massive particles are contained in the refine region.
5. Originally the code looped over each face of the refine region to search for massive particles, but we found that this favored the first faces (x-dimension) in the loop. So we have randomized which face we will evaluate.

6. Search for any particles existing in the outermost slab (1 cell deep on the whole face) on the region face in question. If any massive particles exist in this slab, reduce the refine region by one cell width, e.g. -dy on the right face of the y-dimension.
7. Obtain the min/max of the left/right faces of the refine region from all processors.
8. Every 6 face loops, check if we have removed any particles (communication required).

    If we haven't and there still exists massive particles inside the region, there must be particles farther inside (greater than a cell width from the refine region boundary), we must reduce the refine region on a face to search for these particles. This is where the randomization comes into play, so we don't favor the x-faces. This could be improved by making an educated guess on which face to move inwards by searching for particles near the boundary. However, this might be difficult and time-consuming.

Below in the attachments (region.mov) is an animation showing the above process.

6.8 Accessing Data in BaryonField

For performance reasons, Enzo uses Fortran source to do all the important work. Because of this, it doesn't use the standard C/C++ data structure for the 3D `BaryonField` array, which stores all the Eulerian data.

`BaryonField` is stored as a one dimensional array. Typically C/C++ data is stored in row major order. **ENZO DATA IS STORED IN COLUMN MAJOR ORDER** because of its Fortran underpinnings.

To map between one and three dimensions, in column major order, use the following:

```
OneDindex = i + nx*j + nx*ny*k
```

in Enzo grid member functions, this can be done like this:

```
index = i + GridDimension[0]*(j + GridDimension[1]*k);
```

It should also be mentioned that it is always important to access data in 'stride 1' order. That means accessing data in the order it is stored in memory. So to set all `BaryonFields` to the number 12345.6:

```
int index;
for(int field=0;field<NumberOfBaryonFields;field++)
for(int k=0;k<GridDimension[2]; k++)
  for(int j=0;j<GridDimension[1]; j++)
    for(int i=0;i<GridDimension[0]; i++){
      index = i + GridDimension[0]*(j + GridDimension[1]*k);
      BaryonField[field][index] = 12345.6;
    }
```

This loops over the ghost zones as well as the active zones. To loop over only active zones, use `GridStartIndex` and `GridEndIndex`. Note that this loop must include `GridEndIndex`

```
int index;
for(int field=0;field<NumberOfBaryonFields;field++)
for(int k=GridStartIndex[2];k<=GridEndIndex[2]; k++)
  for(int j=GridStartIndex[1];j<=GridEndIndex[1]; j++)
    for(int i=GridStartIndex[0];i<=GridEndIndex[0]; i++){
      index = i + GridDimension[0]*(j + GridDimension[1]*k);
      BaryonField[field][index] = 12345.6;
    }
```

6.9 Grid Field Arrays

Field arrays are convenient ways (within code linked against the Enzo code base – including within Enzo itself!) to access grid data such as the baryon fields, or particle lists. They can also be used to get pre-defined derived fields, such as temperature. They are intended to be used by solvers, initializers, and analysis routines. The hope is provide a clean way for classes other than the grid to get to grid data, and to help make the current code base more modular.

6.9.1 Class Description

The array class is pretty simple: just enough to represent an N-dimensional grid, without any spatial information. Here is the heart of it, from `EnzoArray.h`:

```
template<typename T>
class EnzoArray
{
public:

  EnzoArray(int rank, int *dims, int *start, int *end,
        FLOAT *cell_size=NULL, int derived=FALSE){

...

  int Rank;                         // number of dimensions
  int Dimension[MAX_DIMENSION];     // total dimensions of all grids
  int StartIndex[MAX_DIMENSION];    // starting index of the active region
                                    //   (zero based)
  int EndIndex[MAX_DIMENSION];      // stoping index of the active region
                                    //   (zero based)
  FLOAT CellWidth[MAX_DIMENSION];

  T *Array;

  // used for velocities and positions
  T *Vector[MAX_NUMBER_OF_PARTICLE_ATTRIBUTES];

...
};

#define EnzoArrayFLOAT EnzoArray<FLOAT>
#define EnzoArrayFloat EnzoArray<float>
#define EnzoArrayInt   EnzoArray<int>
```

The array classes are really a single template, but the macros at the bottom of the header file will hide that from you.

Array vs. Vector

In the above code block, you'll notice two pointers: `T \*Array`; and `T \*Vector`. Here are the rules that these attributes follow: Only one of these will be used, and which one is used depends on the type of data you try to access. Namely, field data, such as density, will be pointed to by `Array`, and vector data, such as velocities or particle positions, will be pointed to by `Vector`.

Destructor (What Gets Deleted)

When the destructor is called, `Array` and `Vector` get deleted *only if* `derived` is TRUE. This is to keep the usage (declare and delete) similar for both derived and underived data. We really don't want to delete the density field on accident.

6.9.2 Access Methods

There are six accessor methods declared in `Grid.h`, two per data type (`float`, `int`, and `FLOAT`).

```
EnzoArrayInt *CreateFieldArrayInt(field_type field);
EnzoArrayInt *CreateFieldArrayInt(char *field_name);

EnzoArrayFloat *CreateFieldArrayFloat(field_type field);
EnzoArrayFloat *CreateFieldArrayFloat(char *field_name);

EnzoArrayFLOAT *CreateFieldArrayFLOAT(field_type field);
EnzoArrayFLOAT *CreateFieldArrayFLOAT(char *field_name);
```

These methods are defined in `Grid_CreateFieldArray.C`. Basically, they allocate a new `EnzoArray`, fill in the dimensions, attach the relevant pointers, and hand it back to. All you need to do is delete the return object.

6.9.3 Field Numbers and Names

The arguments to are either a field number, defined in `typedefs.h`, or the string version of the same. The string versions are defined in a `long` array, named `field_map` in `Grid_CreateFieldArray.C`. This means you can access something as

```
EnzoArrayFloat *density_array = mygrid->CreateFieldArrayFloat(Density);
```

or

```
EnzoArrayFloat *density_array = mygrid->CreateFieldArrayFloat("Density");
```

There are some fields which have names that are the same as grid attributes, like `ParticlePosition`. Rather than have a huge namespace conflict, these have field numbers prefixed with a "g", e.g., `gParticlePosition`. The string called is still just "ParticlePosition", like

```
EnzoArrayFloat *ppos = mygrid->CreateFieldArrayFloat(gParticlePosition);
```

or

```
EnzoArrayFloat *ppos = mygrid->CreateFieldArrayFloat("ParticlePosition");
```

The important part of the map is that it knows the data type of the fields, which you need to know, so you can call the right method. This is really pretty simple, since just about everything returned is a `float`. For a complete list of the (hopefully current) fields, see the section **Field List Reference**. For the best reference, check in `typedefs.h`, and `Grid_CreateFieldArray.C`.

6.9.4 Using the Methods

Here's a somewhat long-winded example of how to use the arrays. First, here's function to create a non-uniform grid

```
grid *Linear3DGrid(){
  // Create a new 3D grid
  float dens = M_PI, total_energy = 0.5, internal_energy = 0.0;
  float vel[3];
  int dims[3];
  FLOAT left[3], right[3];

  grid *lineargrid = new grid;
  int i, j, k, rank = 3;
  int index;

  for (i = 0; i < rank; i++) {
    dims[i] = 134;
    left[i] = 0.0;
    right[i] = 1.0;
    vel[i] = (i+1) * 0.125;
  }

  NumberOfParticleAttributes = 0;
  lineargrid->PrepareGrid(3, dims,
                          left, right, 2);

  int result = lineargrid->InitializeUniformGrid(dens, total_energy, internal_energy, vel);
  assert(result != FAIL);

  EnzoArrayFloat *dens_field = lineargrid->CreateFieldArrayFloat("Density");

  for (k = 3; k <= 130; k++) {
    for (j = 3; j <= 130; j++) {
      index =  k*(134)*(134) +
        j*(134) + 3;
      for (i = 3; i <= 130; i++, index++) {
        dens_field->Array[index] = (float)(i + 1000*j + 1000000*k);
      }
    }
  }

  delete dens_field;

  return lineargrid;
}
```

Notice how this function uses `CreateFieldArrayFloat` to set the values of the density array.

Now, here's a program that creates a uniform grid, and looks at some of the attributes:

```
Eint32 main(Eint32 argc, char *argv[]) {

  CommunicationInitialize(&argc, &argv);

  grid *agrid = Linear3DGrid();

  EnzoArrayFloat *dens = agrid->CreateFieldArrayFloat(Density);

  Eint32 index = 7 + 8*134 + 9*134*134;

  printf("density rank = %"ISYM"\n", dens->Rank);
  printf("density dim[0]  = %"ISYM"\n", dens->Dimension[0]);
  printf("density start[0]  = %"ISYM"\n", dens->StartIndex[0]);
```

```
  printf("density end[0]  = %"ISYM"\n", dens->EndIndex[0], 130);
  printf("density field[7 + 8*134 + 9*134*134] = %"FSYM"\n", dens->Array[index]);

  delete dens;
  delete agrid;

  // End the overall test suite
  CommunicationFinalize();

  return 0;
}
```

This is a complete program, `field_array_example.C`; what this snippet lacks is the fairly long list of header files that need to be included. You can compile this by calling `make field_array_example.exe` in source directory.

6.9.5 Field List Reference

Field Number	Field Name	Data Type	Array or Vector
Density	"Density"	float	Array
TotalEnergy	"TotalEnergy"	float	Array
InternalEnergy	"InternalEnergy"	float	Array
Pressure	"Pressure"	float	Array
Velocity1	"Velocity1"	float	Array
Velocity2	"Velocity2"	float	Array
Velocity3	"Velocity3"	float	Array
ElectronDensity	"ElectronDensity"	float	Array
HIDensity	"HIDensity"	float	Array
HIIDensity	"HIIDensity"	float	Array
HeIDensity	"HeIDensity"	float	Array
HeIIDensity	"HeIIDensity"	float	Array
HeIIIDensity	"HeIIIDensity"	float	Array
HMDensity	"HMDensity"	float	Array
H2IDensity	"H2IDensity"	float	Array
H2IIDensity	"H2IIDensity"	float	Array
DIDensity	"DIDensity"	float	Array
DIIDensity	"DIIDensity"	float	Array
HDIDensity	"HDIDensity"	float	Array
Metallicity	"Metallicity"	float	Array
ExtraType0	"ExtraType0"	float	Array
ExtraType1	"ExtraType1"	float	Array
GravPotential	"GravPotential"	float	Array
Acceleration0	"Acceleration0"	float	Array
Acceleration1	"Acceleration1"	float	Array
Acceleration2	"Acceleration2"	float	Array
gParticlePosition	"ParticlePosition"	FLOAT	Vector
gParticleVelocity	"ParticleVelocity"	float	Vector
gParticleMass	"ParticleMass"	float	Array
gParticleAcceleration	"ParticleAcceleration"	float	Vector
gParticleNumber	"ParticleNumber"	int	Array
gParticleType	"ParticleType"	int	Array

Continued on next page

Table 6.1 – continued from previous page

Field Number	Field Name	Data Type	Array or Vector
gParticleAttribute	"ParticleAttribute"	float	Vector
gPotentialField	"PotentialField"	float	Array
gAccelerationField	"AccelerationField"	float	Vector
gGravitatingMassField	"GravitatingMassField"	float	Array
gFlaggingField	"FlaggingField"	int	Array
gVelocity	"Velocity"	float	Vector

6.10 Adding a new Local Operator.

If you're adding new physics to Enzo, chances are you'll need to add some kind of new operator.

This page is only to describe new physics that is

- Operator split from everything else
- Completely local, so depends on the field value and their derivatives in a cell.
- Doesn't depend on the grid's position in the hierarchy.

Global operators, such as solution to Poisson's equation, are much more significant undertakings, and should be discussed with the Enzo development team.

1. Read all the supporting documents found in *Enzo Primary References*. This is not a simple piece of software.

It's really in your best interest to understand the basic algorithms before trying to write code to extend it. It's much more complex than Gadget or Zeus, and much much easier to break.

1. Open `EvolveHierarchy.C`
2. Read it, and understand the structure. The flowcharts can help, they can be found in *Enzo Flow Chart, Source Browser*.
3. Add a parameter to drive your code in *Adding a new parameter to Enzo*
4. Write your new routine. This can either be a grid member function (old style) or a non-member function that accesses the Enzo data using the *Grid Field Arrays* objects (prefered method.)
5. Locate this block of code:

```
if (Grids[grid1]->GridData->SolveHydroEquations(LevelCycleCount[level],
   NumberOfSubgrids[grid1], SubgridFluxesEstimate[grid1], level) == FAIL) {
  fprintf(stderr, "Error in grid->SolveHydroEquations.\n");
  return FAIL;
}

JBPERF_STOP("evolve-level-13"); // SolveHydroEquations()

//      fprintf(stderr, "%"ISYM": Called Hydro\n", MyProcessorNumber);

/* Solve the cooling and species rate equations. */
```

This is in the primary grid loop on this level.

1. Insert your new grid operation right before the last comment. It should look something like this:

```
if (Grids[grid1]->GridData->SolveHydroEquations(LevelCycleCount[level],
   NumberOfSubgrids[grid1], SubgridFluxesEstimate[grid1], level) == FAIL) {
  fprintf(stderr, "Error in grid->SolveHydroEquations.\n");
```

```
      return FAIL;
    }

    JBPERF_STOP("evolve-level-13"); // SolveHydroEquations()

//      fprintf(stderr, "%"ISYM": Called Hydro\n", MyProcessorNumber);

    /* Solve the cooling and species rate equations. */

    if( YourFlag ){
      if( Grids[grid1]->GridData->YourRoutine(YourArguments) == FAIL ){
        fprintf(stderr,"Error in grid->YourRoutine\n");
        return FAIL;
      }
```

If your code isn't a grid member, you can omit the `Grids[grid1]->GridData->` part.

6.11 Adding a new Test Problem.

This is the best place to start your Enzo Development Career. Even if you're not interested in actually writing a new problem generator, in this page I'll discuss the basic Enzo datastructures and programming patterns.

One deficiency in this tutorial is the lack of Particles. This is not an oversight, but due to the fact that the author of the article doesn't really use particles, as he's not a cosmologist. These will be added in the future, but particles are really not that big of a deal when it comes to the general Enzo data structures. All the information herein is still essential.

6.11.1 Overview

Essentially, you need two write files: `MyProblemInitialize.C` and `Grid_MyProblemInitializeGrid.C`. We'll be discussing these two files. `MyProblemInitialize` is the basic setup code that sets up parameters and the hierarchy, and `MyProblemInitializeGrid` is a member function of the grid class, and actually allocates and assigns data. There are several pitfalls to setting up these files, so read these pages carefully.

We strongly recommend reading everything that proceeds this page on the *Getting Started with Enzo* page and the page about version control and regression testing, *Introduction to Enzo Modification*.

Lastly, please give your problem a reasonable name. I'll be using `MyProblem` throughout this tutorial. Please change this to something that reflects the problem you're installing.

6.11.2 Setup and Installation

Please follow the general Enzo naming convention and call your routines `MyProblemInitialize` and store it in `MyProblemInitialize.C`, and `MyProblemInitializeGrid` and store it in `Grid_MyProblemInitializeGrid.C`

You'll need to install your code in three places.

1. **`Make.config.objects`** is the file that lists all the source file objects needed to build Enzo. Put

   ```
   MyProblemInitialize.o\
   Grid_MyProblemInitializeGrid.o\
   ```

   somewhere in the list of objects. If you want to make things really clean, you can add your own variable to the Makefile and have it driven by a command line switch, but this isn't necessary.

2. **Grid.h. You'll need to put** `MyProblemInitializeGrid` in this the grid class definition. Put it with the rest of the `*InitializeGrid` routines.

3. **InitializeNew.C. Put** `MyProblemInitialize` in `InitializeNew`. At the end of the large block of `*Initialize`, take the next unused ProblemType number and install your code. It should look something like this:

```
// 61) Protostellar Collapse
if (ProblemType == 61)
  ret = ProtostellarCollapseInitialize(fptr, Outfptr, TopGrid, MetaData);

// 62) My New Problem
if ( ProblemType == 62 )
  ret = MyProblemInitialize(fptr, Outfptr, TopGrid, MetaData);

// Insert new problem intializer here...

if (ret == INT_UNDEFINED) {
  fprintf(stderr, "Problem Type %"ISYM" undefined.\n", ProblemType);
  return FAIL;
}
```

To call your problem generator, make sure ProblemType = 62 is in your parameter file. (Or, if 62 is taken, whatever the next unused value is.)

The return value `ret` is used to check for errors and invalid values of `ProblemType`. The function signature will be discussed in the next section.

Also, don't forget to put the proto type at the top:

```
int MyProblemInitialize(FILE *fptr, FILE *Outfptr,
                                    HierarchyEntry &TopGrid,
                                    TopGridData &MetaData);
```

We will revisit `InitializeNew` at the end. For almost all problems, this will be all you do for these three files.

6.11.3 MyProblemInitialize

The primary drive routine is called `MyProblemInitialize`. It basically sets up some global values, problem specific values, and the hierarchy before calling `MyProblemInitializeGrid`.

Function Signature

The function signature of `MyProblemInitialize` is fairly rigid. It should look exactly like the prototype you installed in `InitializeNew`. There are 4 arguments that you'll almost certainly need, and one additional argument that only rare problems will need. You won't likely have any need to add any other arguments. In order, they are:

1. **`FILE *fptr` This is the pointer to the parameter file argument to** Enzo. It's opened and closed in InitializeNew You can read parameters if you like, see below.

2. **`FILE *Outfptr` This is the output pointer, a file called "amr.out."** This file contains the derived details of your problem setup for your record. There is no necessary output for this, it's for the users convenience.

3. **`HierarchyEntry &TopGrid` This is the pointer to the top of the** Hierarchy Linked List. For details of the linked list, *Getting Around the Hierarchy: Linked Lists in Enzo*. For most problem types, it points to the undivided root grid, which is a grid the full size of the top grid, where you will be initializing your data. For problems that are too large for the entire root grid to be allocated, we use the ParallelRootGridIO functionality, to be discussed later. (Please read everything between here and there.)

4. **`TopGridData &MetaData` This is the structure that contains the meta** data describing the Top Grid. Things like boundary condition, problem domain size, rank, and dimension are stored here. See `TopGridData.h` for a complete list of the contents.

If you want to write a problem with Dirichlet boundary conditions, for instance jet inflow, you will need to add a fifth argument to the function (and, of course, it's called in `InitializeNew`). This is the external boundary, `ExternalBoundary &Exterior`. This is the External Boundary object, which you will need to deal with. We will not be discussing this here. If you need to be doing a problem with boundary conditions other than the big 3 (periodic, reflecting, outflow) then we recommend you read the entirety of this tutorial, then follow what's done with the DoubleMach problem, which is problem type 4. You will also need to examine `Grid_SetExternalBoundaryValues.C`

Necessary Headers

The essential header files for `MyProblemInitialize` are the following:

```
#include <stdio.h>
#include <string.h>
#include "macros_and_parameters.h"
#include "typedefs.h"
#include "global_data.h"
#include "Fluxes.h"
#include "GridList.h"
#include "ExternalBoundary.h"
#include "Grid.h"
#include "Hierarchy.h"
#include "TopGridData.h"
```

These should be in this order, to ensure proper definitions across different header files. You should be familiar with the two standard headers <stdio.h> and <string.h>

In brief, these are:

- **`macros_and_parameters.h` The standard set of macros. This takes** care of the float promotion so its inclusion is **ABSOLUTELY ESSENTIAL**
- **`typedefs.h` This takes** care of enumerates for parameters like the hydro method.
- **`global_data.h` There** are a lot of global parameters in Enzo. This houses them.
- **`Fluxes.h` Definition of the** flux object. Not necessary for your objects, but I think its necessary for the later
- **`GridList.h` I don't think** this is necessary, but it's usually included.
- **`ExternalBoundary.h` This defines the external boundary object. Even** if you're not including the external boundary, it's necessary for the following headers.
- **`Grid.h` This defines the grid** class, which you'll definitely need.
- `Hierarchy.h` This defines the Hierarchy Entry linked list.
- `TopGridData.h` This defines the meta data object.

More information can be found in *Header files in Enzo*.

Necessary Assignments

There are two arrays that need to be filled in `MyProblemInitialize`. One of them is **ABSOLUTELY ESSENTIAL** for the functioning of the code. These are `DataLabel` and `DataUnits`. Both of these are arrays of strings

that will be used to label the HDF5 output files. Each element of the array corresponds to an element of the BaryonField array (more on this later) and MUST be defined in the same order. There is not a mechanism to ensure that you do this right, so don't screw it up.

DataLabel

This is the actual name of the field in the HDF5 file. Messing this up is asking for trouble. If you're not using chemistry, you'll want something that looks like this. If you change the actual names, you guarantee that an analysis tool somewhere will break, so don't do it. See `CosmologySimulationInitialize.C` for a more complete list, including extra chemical species.

```
char *DensName = "Density";
char *TEName   = "TotalEnergy";
char *GEName   = "GasEnergy";
char *Vel1Name = "x-velocity";
char *Vel2Name = "y-velocity";
char *Vel3Name = "z-velocity";
i = 0;
DataLabel[i++] = DensName;
DataLabel[i++] = TEName;
if (DualEnergyFormalism)
  DataLabel[i++] = GEName;
DataLabel[i++] = Vel1Name;
DataLabel[i++] = Vel2Name;
DataLabel[i++] = Vel3Name;
```

DataUnits

The units really don't matter very much. They're usually set to NULL

Reading from the Parameter File

You may want to read in problem specific parameters. PLEASE do not put problem specific parameters in the main parameter file reader.

The usual pattern reads each line of the parameter file, and tries to match each line with a parameter. This allows the parameter file to be independent of of order. The typical pattern looks like this:

```
float MyVelocity, MyDensity;
char line[MAX_LINE_LENGTH];
while (fgets(line, MAX_LINE_LENGTH, fptr) != NULL) {
 ret = 0;

  /* read parameters */

  ret += sscanf(line, "MyProblemVelocity      = %"FSYM,
                &MyVelocity);
  ret += sscanf(line, "MyProblemDensity       = %"FSYM,
                &MyDensity);
  if (ret == 0 && strstr(line, "=") && strstr(line, "MyProblem") &&
      line[0] != '#' && MyProcessorNumber == ROOT_PROCESSOR)
    fprintf(stderr,
       "warning: the following parameter line was not interpreted:\n%s\n",
            line);
}
```

If you're not familiar with these functions, here is a good list of standard C functions.

The last line checks for errors in parameters that start with `MyProblem`. Everything involving this routine should be prepended with `MyProblem`. In the file `ReadParameterFile.C`, the parameter file is read and any lines not recognized are thrown as errors; this is the section identified with

`\* check to see if the line belongs to one of the test problems \*/.`

You must add your prefix (in this case, `MyProblem`) to the list of test problem prefixes considered in this section:

```
if (strstr(line, "MyProblem")           ) ret++;
```

or else it will register as an error.

Calling the Grid Initializer: Unigrid

For a small, unigrid problem, the problem initializer is called using the standard Enzo function call procedure.

```
if( TopGrid.GridData->MyProblemInitializeGrid(MyVelocity, MyDensity) == FAIL ){
  fprintf(stderr,"MyProblemInitialize: Error in MyProblemInitializeGrid\n");
  return FAIL;
```

`TopGrid` is the `HierarchyEntry` that starts the hierarchy linked list. It's member `GridData` is a pointer to the actual grid object that you will be modifying.

We will be discussing AMR problems, and large problems that require parallel startup later.

6.11.4 MyProblemInitializeGrid

`MyProblemInitializeGrid` is the member function of the grid class. As a member function, it can access the private data, most importantly `BaryonField`. `BaryonField` is an array of pointers that stores the actual data that the simulator is interested in.

```
float *BaryonField[MAX_NUMBER_OF_BARYON_FIELDS];
```

Necessary Actions

There are four things that this routine ABSOLUTELY MUST do, and they MUST BE DONE IN THIS ORDER.

1. Set up the FieldType array and define NumberOfBaryonFields.

   The FieldType array is an array of type field_type, a type defined in `src/enzo/typedefs.h`. It is used to relate physics to the actual BaryonField element.

   `NumberOfBaryonFields` is the number of valid, allocated fields. This can be as little as 5 for pure fluid dynamics, or as many as you have chemistry to deal with.

   A typical pattern looks like this:

   ```
   NumberOfBaryonFields = 0;
   FieldType[NumberOfBaryonFields++] = Density;
   FieldType[NumberOfBaryonFields++] = TotalEnergy;
   if (DualEnergyFormalism)
     FieldType[NumberOfBaryonFields++] = InternalEnergy;
   FieldType[NumberOfBaryonFields++] = Velocity1;
   vel = NumberOfBaryonFields - 1;
   if (GridRank > 1)
     FieldType[NumberOfBaryonFields++] = Velocity2;
   ```

```
if (GridRank > 2)
  FieldType[NumberOfBaryonFields++] = Velocity3;
```

All the right hand side of those assigns can be found in `typedefs.h`.

Note that all processors must have this information defined for all grids, so this MUST come before step 2.

2. Exit for remote grids.

   Generally, grid member functions have two modes: things that all processors can do, and things that only processors that own the data can do. Usually, the routine simply exits if the processor doesn't own the data:

   ```
   if (ProcessorNumber != MyProcessorNumber)
     return SUCCESS;
   ```

   `ProcessorNumber` is a grid member that stores which processor actually has the data, and `MyProcessorNumber` is the global number of the processor.

   Processors that don't get this data need to not execute the rest of the code.

3. Allocate the BaryonFields

   ```
   this->AllocateGrids()
   ```

   does the trick. More details on this in the Parallel section below.

4. **Assign values to the `BaryonField`. See the page on Baryon Field Access for details.** Note that for problems that are perturbations on a homogenous background, the routine `InitializeUniformGrid` has been provided. See that routine for its function signature.

Initializing AMR problems

For problems that you want to initialize in an AMR fashion, all the previous steps apply. However, instead of simply calling the problem initializer on the Top Grid, one must now initialize a `HierarchyEntry` linked list (of which `TopGrid` is the head) and call the problem initializer on each subgrid. There are several ways to do this, depending on the complexity of the code. One first needs to understand the `HierarchyEntry` linked list. This Page gives a tutorial on the linked lists, and links to examples in the code.

Using ParallelRootGridIO

Main article: *Using Parallel Root Grid IO*

`ParallelRootGridIO` is a fairly complex piece of code. If you absolutely must do this in the code, it is recommended that you read the description of the inner workings of `ParallelRootGridIO` and then cloning what's done for the `CosmologyInitialize` routines.

6.12 Using Parallel Root Grid IO

First, read *Parallel Root Grid IO*. Come back when you're finished.

Parallel root grid IO (PRGIO) is necessary when initializing problems that don't fit in memory on one machine. A PRGIO problem generator needs to function in two passes. First it needs to set up the basic problem (see *Calling the Grid Initializer: Unigrid*) *without* allocating any data. This will create a temporary root grid that covers the entire domain. Then `CommunicationPartitionGrid` splits this grid into several pieces. Usually there is one partition per

MPI process unless the parameter `NumberOfRootGridTilesPerDimensionPerProcessor` is greater than 1. The temporary root grid is then deleted, leaving only the empty level-0 grids. Finally each processor **re**-initializes the newly created subgrids, this time allocating the data only when the grid belongs to it, i.e. `MyProcessorNumber == ProcessorNumber`. Both passes are done in `InitializeNew.C`.

For an example, see either the

- `CosmologySimulationInitialize` and `CosmologySimulationReInitialize`
- `TurbulenceSimulationInitialize` and `TurbulenceSimulationReInitialize`

routines in `InitializeNew.C`.

CHAPTER

SEVEN

REFERENCE INFORMATION

7.1 Enzo Primary References

The Enzo method paper is not yet complete. However, there are several papers that describe the numerical methods used in Enzo, and this documentation contains a brief outline of the essential physics in Enzo, in *Enzo Algorithms*. Two general references (that should be considered to stand in for the method paper) are:

- Introducing Enzo, an AMR Cosmology Application by **O'Shea et al.** In "Adaptive Mesh Refinement - Theory and Applications," Eds. T. Plewa, T. Linde & V. G. Weirs, Springer Lecture Notes in Computational Science and Engineering, 2004. Bibtex entry
- Simulating Cosmological Evolution with Enzo by **Norman et al.** In "Petascale Computing: Algorithms and Applications," Ed. D. Bader, CRC Press LLC, 2007. Bibtex entry

Three somewhat older conferences proceedings are also relevant:

- Simulating X-Ray Clusters with Adaptive Mesh Refinement by **Bryan and Norman.** In "Computational Astrophysics; 12th Kingston Meeting on Theoretical Astrophysics;" proceedings of meeting held in Halifax; Nova Scotia; Canada October 17-19; 1996, ASP Conference Series #123, edited by D. A. Clarke and M. J. West., p. 363. Bibtex entry
- A Hybrid AMR Application for Cosmology and Astrophysics by **Bryan and Norman.** In "Workshop on Structured Adaptive Mesh Refinement Grid Methods", Mar. 1997, ed. N. Chrisochoides. Bibtex entry
- Cosmological Adaptive Mesh Refinement by **Norman and Bryan.** In "Numerical Astrophysics : Proceedings of the International Conference on Numerical Astrophysics 1998 (NAP98)," held at the National Olympic Memorial Youth Center, Tokyo, Japan, March 10-13, 1998. Edited by Shoken M. Miyama, Kohji Tomisaka, and Tomoyuki Hanawa. Boston, Mass. : Kluwer Academic, 1999. (Astrophysics and space science library ; v. 240), p.19 Bibtex entry

The primary hydrodynamics methods are PPM and ZEUS, as described in the following two papers:

- The Piecewise Parabolic Method (PPM) for Gas-Dynamical Simulations by Colella, P.; Woodward, Paul R. Journal of Computational Physics (ISSN 0021-9991), vol. 54, April 1984, p. 174-201. Bibtex entry
- ZEUS-2D: A radiation magnetohydrodynamics code for astrophysical flows in two space dimensions. I - The hydrodynamic algorithms and tests. by Stone and Norman, Astrophysical Journal Supplement Series (ISSN 0067-0049), vol. 80, no. 2, June 1992, p. 753-790. Bibtex Entry

The extension of PPM to cosmology can be found here:

- A piecewise parabolic method for cosmological hydrodynamics. by Bryan et al. Comput. Phys. Commun., Vol. 89, No. 1 - 3, p. 149 - 168 Bibtex entry

The AMR method used in Enzo can be found here:

- Local adaptive mesh refinement for shock hydrodynamics by Berger, M. J. and Colella, P. Journal of Computational Physics (ISSN 0021-9991), vol. 82, May 1989, p. 64-84. Bibtex Entry.

The YT papers can be found here:

- M Turk, Analysis and Visualization of Multi-Scale Astrophysical Simulations Using Python and NumPy in Proceedings of the 7th Python in Science conference (!SciPy 2008), G Varoquaux, T Vaught, J Millman (Eds.), pp. 46-50 (Bibtex entry)
- yt: A Multi-code Analysis Toolkit for Astrophysical Simulation Data, by Turk, M. J.; Smith, B. D.; Oishi, J. S.; Skory, S.; Skillman, S. W.; Abel, T.; and Norman, M. L. The Astrophysical Journal Supplement, Volume 192, Issue 1, article id. 9 (2011) Bibtex Entry.

7.2 Enzo Algorithms

This section provides a very short overview of the algorithms used by the Enzo code. References to texts and journal articles providing a more complete discussion of the algorithms are included at the end of this page for the interested reader, or you can go to *Enzo Primary References*. for a more current list. As of this writing (October 2008), a formal Enzo method paper has not been published, but is in preparation. Much of the text and images on this page have been taken from one of the Laboratory for Computational Astrophysics contributions to the 2003 University of Chicago AMR conference [1] Enzo is written in a mixture of C++ and Fortran 77. High-level functions and data structures are implemented in C++ and computationally intensive lower-level functions are written in Fortran. Enzo is parallelized using the MPI message-passing library and uses the HDF5 data format to write out data and restart files in a platform-independent format.

7.2.1 Adaptive Mesh Refinement

Enzo allows hydrodynamics in 1, 2 and 3 dimensions using the structured adaptive mesh refinement (SAMR) technique developed by Berger and Collela [2]. The code allows arbitrary integer ratios of parent and child grid resolution and mesh refinement based on a variety of criteria, including baryon and dark matter overdensity or slope, the existence of shocks, Jeans length, and cell cooling time. The code can also have fixed static nested subgrids, allowing higher initial resolution in a subvolume of the simulation. Refinement can occur anywhere within the simulation volume or in a user-specified subvolume.

The AMR grid patches are the primary data structure in Enzo. Each individual patch is treated as an individual object, and can contain both field variables and particle data. Individual patches are organized into a dynamic distributed AMR mesh hierarchy using arrays of linked lists to pointers to grid objects. The code uses a simple dynamic load-balancing scheme to distribute the workload within each level of the AMR hierarchy evenly across all processors.

Although each processor stores the entire distributed AMR hierarchy, not all processors contain all grid data. A grid is a *real grid* on a particular processor if its data is allocated to that processor, and a *ghost grid* if its data is allocated on a different processor. Each grid is a real grid on exactly one processor, and a ghost grid on all others. When communication is necessary, MPI is used to transfer the mesh or particle data between processors. The tree structure of a small illustrative 2D AMR hierachy - six total grids in a three level hierarchy distributed across two processors - is shown on the left in Figure 1.

Each data field on a real grid is an array of zones with dimensionality equal to that of the simulation (typically 3D in cosmological structure formation). Zones are partitioned into a core block of *real zones* and a surrounding layer of *ghost zones*. Real zones are used to store the data field values, and ghost zones are used to temporarily store values from surrounding areas, ie, neighboring grids, parent grids or external boundary conditions, when required for updating real zones. The ghost zone layer is three zones deep in order to accomodate the computational stencil

[1] B. W. O'Shea et al. "Introducing Enzo, an AMR Cosmology Application." To be published in Adaptive Mesh Refinement - Theory And Applications, the proceedings from the 2003 University of Chicago AMR Workshop

[2] M. J. Berger and P. Colella. "Local adaptive mesh refinement for shock hydrodynamics," *J. Comp. Phys*, 82:64-84, 1989 link

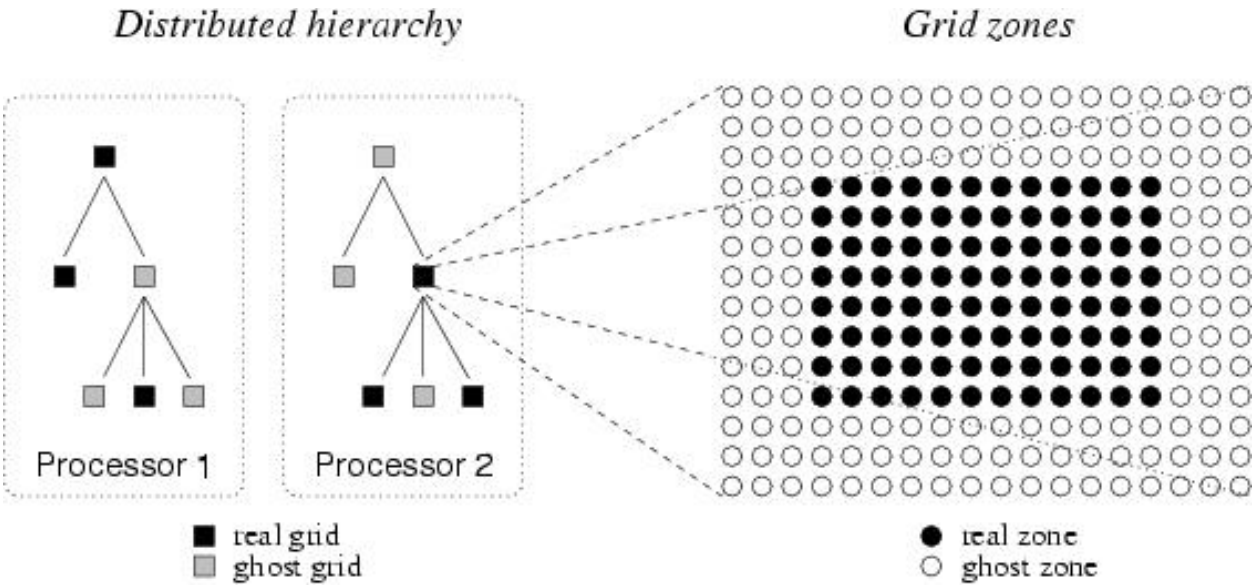


Figure 7.1: **Figure 1.** Real and ghost grids in a hierarchy; real and ghost zones in a grid.

in the hydrodynamics solver (See below), as indicated in the right panel in Figure 1. These ghost zones can lead to significant computational and storage overhead, especially for the smaller grid patches that are typically found in the deeper levels of an AMR grid hierarchy.

For more information on Enzo implementation and data structures, see references [3], [4], [5], and [6].

7.2.2 Dark Matter Dynamics

The dynamics of large-scale structures are dominated by dark matter, which accounts for approximately 85% of the matter in the universe but can only influence baryons via gravitational interaction. There are many other astrophysical situations where gravitational physics is important as well, such as galaxy collisions, where the stars in the two galaxies tend to interact in a collisionless way.

Enzo uses the Particle-Mesh N-body method to calculate collisionless particle dynamics. This method follows trajectories of a representative sample of individual particles and is much more efficient than a direct solution of the Boltzmann equation in most astrophysical situations. The particle trajectories are controlled by a simple set of coupled equations (for simplicity, we omit cosmological terms):

$$\frac{d\mathbf{x}_p}{dt} = \mathbf{v}_p$$

and

$$\frac{d\mathbf{v}_p}{dt} = -\nabla\phi$$

Where x_p and v_p are the particle position and velocity vectors, respectively, and the term on the right-hand side of the second equation is the gravitational force term. The solution to this can be found by solving the elliptic Poisson's

[3] G. L. Bryan. "Fluids in the universe: Adaptive mesh in Cosmology." *Computing in Science and Engineering*, 1:2 p.46, 1999 link

[4] G. L. Bryan and M. L. Norman. "A hybrid AMR application for cosmology and astrophysics." In *Workshop on Structured Adaptive Mesh Refinement Grid Methods*, p. 165. IMA Volumes in Mathematics #117, 2000 link

[5] G. L. Bryan and M. L. Norman. In D.A. Clarke and M. Fall, editors, Computational Astrophyiscs: 12th Kingston Meeting on Theoretical Astrophysics, proceedings of a meeting held in Halifax; Nova Scotia; Canada Oct. 17-19, 1996. ASP Conference Series #123, 1997 link

[6] M. L. Norman and G. L. Bryan. "Cosmological Adaptive Mesh Refinement." In Kohji Tomisaka, Shoken M. Miyama and Tomoyuki Hanawa, editors, Numerical Astrophysics: Proceedings of the International Conference on Numerical Astrophysics 1998, p. 19. Kluwer Academics, 1999

equation:

$$\nabla^2\phi = 4\pi G\rho$$

where ρ is the density of both the collisional fluid (baryon gas) and the collisionless fluid (particles).

These equations are finite-differenced and for simplicity are solved with the same timestep as the equations of hydrodynamics. The dark matter particles are sampled onto the grids using the triangular-shaped cloud (TSC) interpolation technique to form a spatially discretized density field (analogous to the baryon densities used to calculate the equations of hydrodynamics) and the elliptical equation is solved using FFTs on the triply periodic root grid and multigrid relaxation on the subgrids. Once the forces have been computed on the mesh, they are interpolated to the particle positions where they are used to update their velocities.

7.2.3 Hydrodynamics

The primary hydrodynamic method used in Enzo is based on the piecewise parabolic method (PPM) of Woodward & Colella [7] which has been significantly modified for the study of cosmology. The modifications and several tests are described in much more detail in [8], and we recommend that the interested reader look there.

PPM is a higher-order-accurate version of Godunov's method with third-order-accurate piecewise parabolic monotolic interpolation and a nonlinear Riemann solver for shock capturing. It does an excellent job capturing strong shocks and outflows. Multidimensional schemes are built up by directional splitting, and produce a method that is formally second-order-accurate in space and time and explicitly conserves energy, momentum and mass flux. The conservation laws for fluid mass, momentum and energy density are written in comoving coordinates for a Friedman-Robertson-Walker spacetime. Both the conservation laws and Riemann solver are modified to include gravity, which is calculated as discussed above.

There are many situations in astrophysics, such as the bulk hypersonic motion of gas, where the kinetic energy of a fluid can dominate its internal energy by many orders of magnitude. In these situations, limitations on machine precision can cause significant inaccuracy in the calculation of pressures and temperatures in the baryon gas. In order to address this issues, Enzo solves both the internal gas energy equation and the total energy equation everywhere on each grid, at all times. This *dual energy formalism* ensures that the method yields the correct entropy jump at strong shocks and also yields accurate pressures and temperatures in cosmological hypersonic flows. See reference [8] for more information about the dual energy formalism.

As a check on our primary hydrodynamic method, we also include an implementation of the hydro algorithm used in the Zeus astrophysical code [9], [10]. This staggered grid, finite difference method uses artificial viscosity as a shock-capturing technique and is formally first-order-accurate when using variable timesteps (as is common in structure formation simulations), and is not the preferred method in the Enzo code.

7.2.4 Cooling/Heating

The cooling and heating of gas is extremely important in astrophysical situations. To this extent, two radiative cooling models and several uniform ultraviolet background models have been implemented in an easily extensible framework.

The simpler of the two radiative cooling models assumes that all species in the baryonic gas are in equilibrium and calculates cooling rates directly from a cooling curve assuming Z = 0.3 Z_o. The second routine, developed by Abel,

[7] P. R. Woodward and P. Colella. "A piecewise parabolic method for gas dynamical simulations," *J. Comp. Phys*, 54:174, 1984 link

[8] G. L. Bryan, M. L. Norman, J. M. Stone, R. Cen and J. P. Ostriker. "A piecewise parabolic method for cosmological hydrodynamics," *Comp. Phys. Comm.*, 89:149, 1995 link

[9] J. M. Stone and M. L. Norman. "Zeus-2D: A radiation magnetohydrodynamics code for astrophysical flows in two space dimensions. I. The hydrodynamics algorithms and tests." *The Astrophysical Journal Supplement*, 80:753, 1992 link

[10] J. M. Stone and M. L. Norman. "Zeus-2D: A radiation magnetohydrodynamics code for astrophysical flows in two space dimensions. II. The magnetohydrodynamic algorithms and tests." *The Astrophysical Journal Supplement*, 80:791, 1992 link

Zhang, Anninos & Norman [11], assumes that the gas has primordial abundances (ie, a gas which is composed of hydrogen and helium, and unpolluted by metals), and solves a reaction network of 28 equations which includes collisional and radiative processes for 9 seperate species (H, H^+, He, He^+, He^{++}, H^-, H_2^+, H_2 and e^-). In order to increase the speed of the calculation, this method takes the reactions with the shortest time scales (those involving H^- and H_2^+) and decouples them from the rest of the reaction network and imposes equilibrium concentrations, which is highly accurate for cosmological processes. See [11] and [12] for more information.

The vast majority of the volume of the present-day universe is occupied by low-density gas which has been ionized by ultraviolet radiation from quasars, stars and other sources. This low density gas, collectively referred to as the Lyman-α Forest because it is primarily observed as a dense collection of absorption lines in spectra from distant quasars (highly luminous extragalactic objects), is useful because it can be used to determine several cosmological parameters and also as a tool for studying the formation and evolution of structure in the universe (see [13] for more information). The spectrum of the ultraviolet radiation background plays an important part in determining the ionization properties of the Lyman-α forest, so it is very important to model this correctly. To this end, we have implemented several models for uniform ultraviolet background radiation based upon the models of Haardt & Madau [14].

7.2.5 Star Formation and Feedback

One of the most important processes when studying the formation and evolution of galaxies (and to a lesser extent, groups and clusters of galaxies and the gas surrounding them) is the formation and feedback of stars. We use a heuristic prescription similar to that of Cen & Ostriker [15] to convert gas which is rapidly cooling and increasing in density into star *particles* which represent an ensemble of stars. These particles then evolve collisionlessly while returning metals and thermal energy back into the gas in which they formed via hot, metal-enriched winds.

7.2.6 Parallelization in Enzo

Enzo uses a grid-based parallelization scheme for load balancing. The root grid is partitioned up into N pieces (where N is the number of processors), and each processor is given a piece of the root grid, which it keeps for the duration of the simulation run. Subgrids are treated as independent objects and are distributed to the processors such that each level of grids is load-balanced across all processors. Boundary fluxes between neighboring grid patches and parent and children grids are passed back and forth using MPI commands.

The one portion of the code that is parallelized differently is the root grid gravity solver. As discussed above, the gravitational potential on the root grid is solved using a fourier transform method, which requires its own message-passing routines. The three-dimensional total density field (composed of the dark matter plus baryon density on the root grid) is decomposed into two-dimensional slabs (requiring one set of messages), which are then fourier transformed. The slabs are then transposed along another axis (requiring a second set of messages to be passed) and transformed again, and a third set of messages is required in order to obtain the original block decomposition. This is unavoidable when using a fourier transform scheme, and as a result the speed of the root grid gravity solver is very sensitive to the speed of the communication network on the platform that Enzo is being run on.

7.2.7 Initial Conditions Generator

A somewhat detailed description of the method Enzo uses to create initial conditions can be downloaded as a `postscript` or `PDF` document. To summarize: Dark matter particles and baryon densities are laid out on a uniform

[11] T. Abel, P. Anninos, Y. Zhang and M.L. Norman. "Modeling primordial gas in numerical cosmology." New Astronomy, 2:181-207, August 1997 link

[12] P. Anninos, Y. Zhang, T. Abel and M.L. Norman. "Cosmological hydrodynamics with multispecies chemistry and nonequilibrium ionization and cooling." New Astronomy, 2:209-224, August 1997 link

[13] M. Rauch. "The Lyman Alpha Forest in the Spectra of QSOs." Annual Review of Astronomy and Astrophysics, 36:267-316, 1998 link

[14] F. Haardt and P. Madau. "Radiative Transfer in a Clumpy Universe, II. The Ultraviolet Extragalactic Background." The Astrophysical Journal, 461:20, 1996 link

[15] R. Cen and J.P. Ostriker. "Galaxy formation and physical bias." The Astrophysical Journal Letters, 399:L113, 1992 link

Cartesian grid. Given a user-specified power spectrum P(k), the linear density fluctuation field is calculated at some initial time (typically z = 100 for high-resolution/small box simulations) by using P(k) to obtain the density fluctuations in k-space on a uniform Cartesian grid. P(k) is sampled discretely at each grid point, with the density fluctuations having a random complex phase and amplitude. The amplitude is generated such that the distribution of amplitudes is Gaussian. This cube is then fourier transformed to give physical density fluctuations. Particle positions and velocities and baryon velocities are calculated using the Zel'dovich approximate. See the document above, or read Bertschinger 1998 [16] for more information.

7.2.8 References

Note: Some of the links to references require a subscription.

7.3 Enzo Internal Unit System

The units of the physical quantities used in Enzo depend on the problem being run. For most test problems there is no physical length or time specified, so the units can be be simply scaled. For cosmology, there are a set of units designed to make most quantities of order unity so that single precision floating-point variables can be used. These units are defined in *Enzo Output Formats*. Additionally, discussion of how particle masses are stored in Enzo can be found at *Enzo Particle Masses*. However, with the broader use of Enzo for non-cosmological astrophysics applications, it has become necessary to add a new set of units into the code. This page describes how to set these units.

In order to have a self-consistent set of units, the user has to set appropriate length, time, and mass OR density scales. Simulations that include gravity also need to have a self-consistent gravitational constant that is scaled to the other variables. The four parameters that the user can set are `LengthUnits`, `TimeUnits`, `DensityUnits`, and `MassUnits`. Only one of `DensityUnits` or `MassUnits` needs to be set, since `MassUnits` = `DensityUnits` * `LengthUnits` ^{3} . Additionally, if the parameter `SelfGravity` is turned on (set to 1), the parameter `GravitationalConstant` must be set to 4*pi*G, where G is Newton's gravitational constant as a dimensionless quantity (that is, with all units scaled out).

The primary motivation for using a non-arbitrary set of units is to take advantage of Enzo's various chemistry and cooling algorithms, some of which have been scaled assuming CGS units. To do this, one chooses physical units assuming the simulation box size is unity in code units, and that a density/mass and time value of 1 in code units are something meaningful in CGS. For example, if one is interested in setting the box size to one parsec, a density of 1 in code units equivalent to to 10 hydrogen atoms per cubic centimeter (in CGS units), and the time unit to one million years, the appropriate settings of the parameters would be as follows:

```
DensityUnits = 1.67e-23     # 10 hydrogen atoms per cc in CGS (c/cm^3)
LengthUnits = 3.0857e+18    # one parsec in cm
TimeUnits = 3.1557e+13      # one megayear in seconds
```

If we then wish to use gravity, the gravitational constant must be set explicitly to 4*pi*G expressed in a unitless fashion. Since the gravitational constant in CGS has units of $cm^3/(g*s^2)$, this means that the value should be $4*pi*G_{cgs}$ * `DensityUnits` * `TimeUnits` ^{2}. So, in the units expressed above, that means the gravity parameters must be set as follows:

```
SelfGravity                = 1
GravitationalConstant      = 0.0139394          # 4*pi*G_{cgs}*DensityUnits*TimeUnits^2
```

Note that if gravity is turned on, the parameter `TopGridGravityBoundary` must also be set to either 0 (periodic) or 1 (isolated).

[16] E. Bertschinger. "Computer Simulations in Cosmology." Annual Review of Astronomy and Astrophysics, 36:599 link

If you set only `LengthUnits` and `DensityUnits` but not `TimeUnits` the code will calculate it for you using the actual gravitational constant. You see it printed out in the terminal when the code starts up and you can also find it towards the end of the parameter file of any output.

7.4 Enzo Particle Masses

A common problem for users who wish to manipulate Enzo data is understanding Enzo's internal unit system. This is explained in some detail in *Enzo Internal Unit System*. This page focuses specifically on the particle mass, which is one of the least intuitive pieces of the internal code notation. The most important thing to realize is that Enzo's `particle_mass` attribute ***is not a mass*** - it is actually a ***density***. This is done for a very good reason - Enzo calculates the gravitational potential by solving Poisson's equation using a grid-based density field, and when calculating the dark matter (or other particle) density, it is most efficient computationally to store it as a density rather than as a mass to avoid having to divide by volume or multiple by 1/V for every particle, on every timestep. So, the "mass" stored within the code is really this value in the cosmology calculations:

$$\text{mass} = \frac{\Omega_{m0} - \Omega_{b0}}{\Omega_{m0}} \left(\frac{\Delta x_p}{\Delta x_g}\right)^3$$

where Ω_{m0} is OmegaMatterNow, Ω_{b0} is OmegaBaryonNow, Δx_p is the mean separation between particles at the beginning of the simulation (in code units), and Δx_g is the grid spacing (in code units) of the grid that the particle resides in. Conversion to an actual mass is as follows:

$$\text{realmass} = \text{particlemass} \times \Delta x_g^3 \times \text{DensityUnits} \times \text{LengthUnits}^3$$

If one is using massive (non-zero mass) particles in a non-cosmology run, the formulation of the particle mass is analogous: it can be calculated as:

$$\text{mass} = \frac{\rho_{part}}{\text{DensityUnits}} \left(\frac{\Delta x_p}{\Delta x_g}\right)^3$$

where the upper and lower density values are the mean matter density of your particle field (so total particle mass divided by total volume, in your units of choice) divided by the DensityUnits (such that the fraction is completely unitless).

7.5 The Flux Object

This page is intended to document the creation, use, and destruction of the Fluxes object in Enzo. This will not be a complete description of the Flux Correction algorithm, see the primary references for that.

7.5.1 Purpose

In order to keep the change in zone size across grid boundaries consistent with the underlying conservation law, Flux Correction is used. Basically, it makes sure that the change in Total Energy inside a subgrid (or mass, momentum, or any other conserved quantitiy) is equal to the flux across the boundary **as seen by both levels**. This means that the coarse grid, which gets its solution in that space replaced by the fine grid data, also needs to have the zones right outside that space updated so they also see that same flux.

To facilitate this operation, the Fluxes object is used.

For each subgrid, there are two Fluxes objects, that store the flux computed in the solver (typically `Grid_[xyz]EulerSweep`). One stores the fluxes that the fine grid computes, and one stores the fluxes that the coarse grid computes. These are stored in two objects: a grid member fluxes `BoundaryFluxes` for the fine data, and fluxes `***SubgridFluxesEstimate` for the coarse data.

7.5.2 Fluxes.h

The actual object can be found in `src/enzo/Fluxes.h`.

```
struct fluxes
{
  long_int LeftFluxStartGlobalIndex[MAX_DIMENSION][MAX_DIMENSION];
  long_int LeftFluxEndGlobalIndex[MAX_DIMENSION][MAX_DIMENSION];
  long_int RightFluxStartGlobalIndex[MAX_DIMENSION][MAX_DIMENSION];
  long_int RightFluxEndGlobalIndex[MAX_DIMENSION][MAX_DIMENSION];
  float *LeftFluxes[MAX_NUMBER_OF_BARYON_FIELDS][MAX_DIMENSION];
  float *RightFluxes[MAX_NUMBER_OF_BARYON_FIELDS][MAX_DIMENSION];
};
```

This contains two sets of arrays for the actual flux values, and 4 arrays to describe the position of the flux in the computational domain. There is a flux on each face of the subgrid, and each flux has a vector describing its start and end. For instance, `LeftFluxStartGlobalIndex[0][dim]` describes the starting index for the X face left flux. `LeftFluxes[densNum][0]` describes the flux of density across the left x face.

7.5.3 SubgridFluxesEstimate

`SubgridFluxesEstimate` is a 2 dimensional array of pointers to Fluxes objects that a given grid patch will fill. Its indexing is like `*SubgridFluxesEstimate[Grid][Subgrid]` , where `Grid` goes over all the grids on a level, and `Subgrid` goes over that grid's subgrids PLUS ONE for the grid itself, as each grid needs to keep track of its own boundary flux for when it communicates with the parent. (This last element is used in conjunction with the `BoundaryFluxes` object, as we'll see later)

Allocation

Allocation of the pointer array for the grids on this level happens at the beginning of `EvolveLevel`:

```
fluxes ***SubgridFluxesEstimate = new fluxes **[NumberOfGrids];
```

At the beginning of the time loop, each grid has its subgrid fluxes array allocated, and a fluxes object is allocated for each subgrid (plus one for the grid itself)

```
while (dtThisLevelSoFar < dtLevelAbove) {
 ... timestep computation  ...
  for (grid = 0; grid < NumberOfGrids; grid++) {

     // The array for the subgrids of this grid
     SubgridFluxesEstimate[grid] = new fluxes *[NumberOfSubgrids[grid]];

     if (MyProcessorNumber ==
         Grids[grid]->GridData->ReturnProcessorNumber()) {

       for( Subgrids of grid ){
         SubgridFluxesEstimate[grid][counter] = new fluxes;
         ... Setup meta data ...
       }

       /* and one for the grid itself */
       SubgridFluxesEstimate[grid][counter] = new fluxes;
       ... and some meta data ...
```

```
    }
  } // end loop over grids (create Subgrid list)
```

Note that in older versions of Enzo are missing the processor check, so fluxes objects are allocated for each grid and subgrid on each processor, causing a bit of waste. This has been fixed since Enzo 1.5.

The `LeftFluxes` and `RightFluxes` are allocated in `Grid_SolveHydroEquations.C`

Assignment

After the `LeftFluxes` and `RightFluxes` are allocated in `Grid_SolveHydroEquations.C`, they are filled with fluxes from the solver. In v2.0, the C++ and FORTRAN interface with the hydrodynamics solver was improved to avoid the previous method that juggled pointers to a temporary array for the fluxes returned from the FORTRAN hydro solver. Now `Grid_[xyz]EulerSweep.C` allocates memory for each of the flux variables and passes them into each of the FORTRAN hydro routines. This removes any size limitations that the old wrappers had when the temporary array was too large.

Flux Correction

After being filled with coarse grid fluxes, `SubgridFluxesEstimate` is then passed into `UpdateFromFinerGrids`, where it is used to correct the coarse grid cells and boundary fluxes. For each grid/subgrid, `SubgridFluxesEstimate` is passed into `Grid_CorrectForRefinedFluxes` as `InitialFluxes`. The difference of `InitialFluxes` and `RefinedFluxes` is used to update the appropriate zones. (Essentially, the coarse grid flux is removed from the update of those zones ex post facto, and replaced by the average of the (more accurate) fine grid fluxes.

See the section below for the details of `SubgridFluxesRefined` and `RefinedFluxes`.

AddToBoundaryFluxes

The last thing to be done with `SubgridFluxesEstimate` is to update the `BoundaryFluxes` object for each grid on the current level. Since multiple fine grid timesteps are taken for each parent timestep, the **total** flux must be stored on the grids boundary. This is done in `Grid_AddToBoundaryFluxes`, at the end of the EvolveLevel timestep loop.

Deallocation

In the same grid loop that `BoundaryFluxes` is updated, the `SubgridFluxesEstimate` object is destroyed with `DeleteFluxes`, and the pointers themselves are freed.

```
for (grid = 0; grid < NumberOfGrids; grid++) {
   if (MyProcessorNumber == Grids[grid]->GridData->ReturnProcessorNumber()) {

    Grids[grid]->GridData->AddToBoundaryFluxes
        (SubgridFluxesEstimate[grid][NumberOfSubgrids[grid] - 1])


    for (subgrid = 0; subgrid < NumberOfSubgrids[grid]; subgrid++) {

     DeleteFluxes(SubgridFluxesEstimate[grid][subgrid]);

     delete SubgridFluxesEstimate[grid][subgrid];
    }
```

```
  delete [] SubgridFluxesEstimate[grid];
 }
```

7.5.4 grid.BoundaryFluxes

Each instance of each grid has a fluxes `BoundaryFluxes` object that stores the flux across the surface of that grid. It's used to correct it's Parent Grid.

Allocation

`BoundaryFluxes` is allocated immediately *before* the timestep loop in `EvolveLevel` by the routine `ClearBoundaryFluxes`.

Usage

For each grid, `BoundaryFluxes` is filled at the end of the `EvolveLevel` timestep loop by the last element of the array `SubgridFluxesEstimate[grid]` for that grid. This is additive, since each grid will have multiple timesteps that it must correct its parent for. This is done by `AddToBoundaryFluxes`, as described above.

`BoundaryFluxes` is used in `UpdateFromFinerGrids` to populate another fluxes object, `SubgridFluxesRefined`. This is done in `GetProjectedBoundaryFluxes`. The values in `SubgridFluxesRefined` are area weighted averages of the values in `BoundaryFluxes`, coarsened by the refinement factor of the simulation. (So for factor of 2 refinement, `SubgridFluxesRefined` has half the number of zones in each direction than `BoundaryFluxes`, and matches the cell width of the parent grid.)

`BoundaryFluxes` is also updated from subgrids in `CorrectForRefinedFluxes`. This happens when a subgrid boundary lines up exactly with a parent grid boundary. However, in many versions of Enzo, this is deactivated by the following code:

```
        CorrectLeftBoundaryFlux = FALSE;
        CorrectRightBoundaryFlux = FALSE;
#ifdef UNUSED
        if (Start[dim] == GridStartIndex[dim]-1)
          CorrectLeftBoundaryFlux = TRUE;
        if (Start[dim] + Offset == GridEndIndex[dim]+1)
          CorrectRightBoundaryFlux = TRUE;
#endif /* UNUSED */
```

It is unclear why this is, but removal of the UNUSED lines restores conservation in the code, and is essential for proper functioning of the MHD version of the code (which will be released in the future.) I have seen no problems from removing this code.

Many implementations of block structured AMR require a layer of zones between parent and subgrid boundaries. Enzo is not one of these codes.

Deallocation

`BoundaryFluxes` is only deleted once the grid itself is deleted. This happens mostly in `RebuildHierarchy`.

7.5.5 SubgridFluxesRefined

The final instance of a fluxes object is fluxes `SubgridFluxesRefined`. This object takes the fine grid fluxes, resampled to the coarse grid resolution, and is used to perform the flux correction itself. This section is short, as its existance has been largely documented in the previous sections.

Allocation

`SubgridFluxesRefined` is declared in `UpdateFromFinerGrids`. The actual allocation occurs in `Grid_GetProjectedBoundaryFluxes`, where it's passed in as `ProjectedFluxes`.

Usage

`SubgridFluxesRefined` is also filled in `Grid_GetProjectedBoundaryFluxes`, as the area weighted average of the subgrid boundary flux.

It is then passed into `Grid_CorrectForRefinedFluxes`, Here, it is used to update the coarse grid zones that need updating.

Deallocation

`SubgridFluxesRefined` is deleted after it is used in `Grid_CorrectForRefinedFluxes`.

7.6 Header files in Enzo

Here is a complete list of the Enzo 2.0 header files and a brief description of what they do.

`src/enzo/CoolData.h`

> Contains parameters for cooling tables and radiation fields. Most importantly this struct has the pointers to the tabulated cooling functions that are used in `cool1d_multi.src`. This type is used for the global variable CoolData.

`src/enzo/CosmologyParameters.h`

> Defines the global variables that are used in cosmology simulations, e.g. cosmological parameters, initial redshift, redshift outputs.

`src/enzo/ealFloat.h`

> Class for floating-point arrays that supports array arithmetic. Mainly used by the Enzo Analysis class.

`src/enzo/ealInt.h`

> Same as `ealFloat.h` but for integers.

`src/enzo/EnzoArray.h`

> Templated class that is a container for grid and particle quantities in the Enzo Analysis class.

`src/enzo/enzo_unit_tests.h`

> Framework for simple tests on Enzo. Not used in typical simulations.

`src/enzo/ExternalBoundary.h`

> The ExternalBoundary class definition.

`src/enzo/FastSiblingLocator.h`

Structure definitions for the chaining mesh and sibling lists.

`src/enzo/flowdefs.h`

Function prototypes and variables for FLOW_TRACE define. Currently not used.

`src/enzo/Fluxes.h`

The fluxes structure, used to contain the Coarse and Refined fluxes for each parent/subgrid pair.

`src/enzo/global_data.h`

This houses all global parameters for Enzo, which is most of them. Variables defined here are defined as extern in all routines but `src/enzo/enzo.C` (see the `DEFINE_STORAGE` #define there) and are initialized with `src/enzo/SetDefaultGlobalValues.C`.

`src/enzo/Grid.h`

This defines the primary God Class, `grid`.

`src/enzo/GridList.h`

Structure for a linked list of grids. Used when identifying new subgrids, `Grid_IdentifyNewSubgrids.C` and `Grid_IdentifyNewSubgridsSmall.C`.

`src/enzo/Hierarchy.h`

Defines the HierarchyEntry linked list structure. More can be found about this in *Getting Around the Hierarchy: Linked Lists in Enzo*.

`src/enzo/ImplosionGlobalData.h`

Contains global variables that have store the parameters in the Implosion problem type.

`src/enzo/LevelHierarchy.h`

Defines the `LevelHierarchyEntry` linked list structure. More can be found about this in *Getting Around the Hierarchy: Linked Lists in Enzo*.

`src/enzo/ListOfParticles.h`

Structure for a linked list of particle lists. Used in `OutputAsParticleData.C`.

`src/enzo/macros_and_parameters.h`

This is the home for all preprocessor directives, and is responsible for overloading floating point precision keywords.

`src/enzo/message.h`

Defines to handle error, warning, and debug messages.

`src/enzo/MTLPARAM.h`

Common variables for the Cen's metal cooling routines, `mcooling.src`

`src/enzo/performance.h`

Defines for the interface between Enzo and LCAperf.

`src/enzo/phys_constants.h`

Defines for physical constants

`src/enzo/ProtoSubgrid.h`

Defines the ProtoSubgrid class, used in `src/enzo/FindSubgrids.C`.

`src/enzo/RadiationFieldData.h`

Structure that contains the parameters and variables that describe the background radiation field. Only used for the global variable RadiationData in `global_data.h`.

`src/enzo/RateData.h`

Structure that holds all of the parameters and arrays of the rate equations for the non-equilibrium chemistry. Only used for the global variable RateData.

`src/enzo/region.h`

Structures that describe a region when computing the parallel FFT.

`src/enzo/SedovBlastGlobalData.h`

Contains global variables that have store the parameters in the Sedov blast problem type.

`src/enzo/ShockPoolGlobalData.h`

Contains global variables that have store the parameters in the shock pool problem type.

`src/enzo/SphericalInfall.h`

Contains global variables that have store the parameters in the spherical infall problem type.

`src/enzo/StarParticleData.h`

Global variables that store parameters about the star formation routines. It also has variables that keep track of the number of stars.

`src/enzo/TestGravitySphereGlobalData.h`

Contains global variables that have store the parameters in the test gravity sphere problem type.

`src/enzo/TestProblemData.h`

Structure that stores parameters that describe a problem initialization.

`src/enzo/TopGridData.h`

Defines the TopGrid structure, which houses the global parameters of the simulation.

`src/enzo/typedefs.h`

Has all the enumerate lists used to give words to parameters. Defines types for field (density, etc), interpolation method, hydro method, boundary type, gravity boundary type.

`src/enzo/units.h`

Global variables that store the units in CGS. Used when ComovingCoordinates is *off*.

`src/enzo/WavePoolGlobalData.h`

Contains global variables that have store the parameters in the wave pool problem type.

7.7 The `Enzo` Makefile System

The makefile system in `Enzo` is a bit complicated, because it's designed to work on many different platforms, allow many different compile-time configuration settings, and be usable by automated systems such as the `lcatest` parallel program testing environment.

To decouple machine-specific settings from configuration-specific settings, it's organized into separate files summarized below. Note that the files discussed on this page are found in the `src/enzo` subdirectory.

Makefile	The main makefile for compiling the `Enzo` executable `enzo.exe`
Make.mach.*	These files contain all machine-dependent settings
Make.config.*	These files contain all compile-time configuration settings

If there is already a `Make.mach.*` file present for the particular machine you want to compile on, and you just want to compile `Enzo` with the default configuration, then compiling is relatively straightforward. For example, to compile `Enzo` on NICS's Kraken platform (starting from the top-level `Enzo` directory):

```
./configure
cd src/enzo
gmake machine-nics-kraken
gmake
```

If all goes well, this should create the `enzo.exe` executable in the `src/enzo` subdirectory. Also, note that `gmake` is required, though `make` may work on your system as well.

7.7.1 Machine settings

If there is not already a `Make.mach.*` file present for your platform, you will need to create one. The easiest way to port `Enzo` to a new platform is to copy an existing `Make.mach.*` file to a new one and edit it accordingly. Generally, all variables prefixed by `MACH_` in `Make.mach.*` files should be assigned a value (even if that value is an empty string), and all variables that begin with `LOCAL_` (or anything else) are optional and only accessed within the `Make.mach.*` file itself.

The list of `MACH_` variables that can be set are listed below.

General variables:

MACH_FILE	Name of the make include file for the machine, e.g. `Make.mach.nics-kraken`
MACH_TEXT	Description of the platform, e.g. `"NICS Kraken"`
MACH_VALID	Should be set to 1, though not currently accessed

Paths to compilers:

MACH_CPP	The C preprocessor
MACH_CC_MPI	The MPI C compiler
MACH_CC_NOMPI	The C compiler
MACH_CXX_MPI	The MPI C++ compiler
MACH_CXX_NOMPI	The C++ compiler
MACH_F90_MPI	The MPI F90 compiler
MACH_F90_NOMPI	The F90 compiler
MACH_FC_MPI	The MPI F77 compiler
MACH_FC_NOMPI	The F77 compiler
MACH_CUDACOMPILER	The CUDA compiler
MACH_LD_MPI	The MPI linker (typically the MPI C++ compiler)
MACH_LD_NOMPI	The linker (typically the C++ compiler)

Compiler flags:

MACH_CPPFLAGS	Machine-dependent flags for the C preprocessor, e.g. `-P -traditional`
MACH_CFLAGS	Machine-dependent flags for the C compiler
MACH_CXXFLAGS	Machine-dependent flags for the C++ compiler
MACH_F90FLAGS	Machine-dependent flags for the F90 compiler
MACH_FFLAGS	Machine-dependent flags for the F77 compiler
MACH_LDFLAGS	Machine-dependent flags for the linker

Precision flags:

MACH_DEFINES	Machine-specific defines, e.g. `-DLINUX`, `-DIBM`, `-DIA64`, etc.
MACH_FFLAGS_INTEGER_32	Fortran flags for specifying 32-bit integers
MACH_FFLAGS_INTEGER_64	Fortran flags for specifying 64-bit integers
MACH_FFLAGS_REAL_32	Fortran flags for specifying 32-bit reals
MACH_FFLAGS_REAL_64	Fortran flags for specifying 64-bit reals

Paths to include header files:

MACH_INCLUDES	All required machine-dependent includes–should at least include HDF5.
MACH_INCLUDES_HYPRE	Includes for optional Hypre linear solver package
MACH_INCLUDES_MPI	Includes for MPI if needed
MACH_INCLUDES_CUDA	Includes for CUDA if needed
MACH_INCLUDES_PYTHON	Includes for Python if needed

Paths to library files:

MACH_LIBS	All required machine-dependent libraries–should at least include HDF5.
MACH_LIBS_HYPRE	Libraries for optional Hypre linear solver package
MACH_LIBS_MPI	Libraries for MPI if needed
MACH_LIBS_PAPI	Libraries for optional PAPI performance package (optionally called by `lcaperf`)
MACH_LIBS_CUDA	Libraries for CUDA if needed
MACH_LIBS_PYTHON	Libraries for Python if needed

Optimization flags:

MACH_OPT_AGGRESSIVE	Compiler/link flags for “aggressive” optimization
MACH_OPT_DEBUG	Compiler/link flags for debugging
MACH_OPT_HIGH	Compiler/link flags for standard optimizations
MACH_OPT_WARN	Compiler/link flags to generate verbose warning messages

Although it breaks from the `MACH_*` naming convention, there is also a **MACHINE_NOTES** variable for machine-specific information that is displayed whenever `Enzo` is compiled.

7.7.2 Makefile commands

The default action of typing `gmake` without a target is to attempt to compile `Enzo`. Other high-level makefile targets are `help`, and `clean`:

gmake	Compile and generate the executable `enzo.exe`
gmake help	Display this help information
gmake clean	Remove object files, executable, etc.

(For brevity we’ll omit the `gmake` portion for the remainder of the discussion.)

Configuration-related targets are `help-config`, `show-config`, `show-flags`, and `default`:

help-config	Display detailed help on configuration make targets
show-config	Display the current configuration settings
show-flags	Display the current compilers and compilation flags
default	Reset the configuration to the default values

Note that `gmake default` may also clear your machine setting, in which case you will need to rerun gmake machine-*platform*.

7.7.3 Configuration options

Other configuration targets, set using e.g. `gmake integers-32`, are listed below:

Free parameters

max-subgrids-N	Set the maximum number of subgrids to *N*.
max-baryons-N	Set the maximum number of baryon fields to *N*.
max-tasks-per-node-N	Set the number of tasks per node to *N*.
memory-pool-N	Set initial memory pool size (in number of photons).

Precision settings

integers-[32|64]	Set integer size to 32- or 64-bits.
precision-[32|64]	Set floating-point precision to 32- or 64-bits.
particles-[32|64|128]	Set particle position precision to 32-, 64-, or 128-bits.
inits-[32|64]	Set inits precision to 32- or 64-bits.
io-[32|64]	Set IO precision to 32- or 64-bits.
particle-id-[32|64]	Set integer size for particle IDs

Global settings

object-mode-[32|64]	Set address/pointer size to 32-bit or 64-bit object files. This is an obsolete setting and is no longer used.
testing-[yes|no]	Include hooks for the lcatest regression tests

Algorithmic settings

bitwise-[no|yes]	Turn on blocking-gravity for bitwise identical runs
emissivity-[no|yes]	Include emissivity field
fastsib-[no|yes]	Include fast sibling search
fluxfix-[no|yes]	Include sibling subgrid boundary fix
newgridio-[no|yes]	Use the new Grid IO routines
photon-[no|yes]	Include radiative transfer (adaptive ray tracing)

External libraries

use-mpi-[yes|no]	Set whether to use MPI.
isolated-bcs-[yes|no]	Set whether to compile in isolated boundary conditions code
tpvel-[yes|no]	Set whether to compile in tracer particle velocity information
lcaperf-[yes|no]	Set whether to call the optional lcaperf performance tool
papi-[yes|no]	Set whether to link in the PAPI library if required by lcaperf
hypre-[no|yes]	Include HYPRE libraries (implicit RT solvers)
cuda-[no|yes]	Set whether to use CUDA (GPU-computing)
python-[no|yes]	Set whether to use inline python
use-hdf4-[no|yes]	Set whether to use HDF4

Performance settings

opt-VALUE	Set optimization/debug/warning levels, where VALUE = [warn|debug|high|aggressive|cudadebug]
taskmap-[yes|no]	Set whether to use unigrid taskmap performance modification
packed-amr-[yes|no]	Set whether to use 'packed AMR' disk performance modification.
packed-mem-[yes|no]	Set whether to use 'packed memory' option: requires packed AMR.
unigrid-transpose-[yes|no]	Set whether to perform unigrid communication transpose performance optimization
ooc-boundary-[yes|no]	Set whether to use out-of-core handling of the boundary
load-balance-[yes|no]	Set whether to use load balancing of grids

7.7.4 The `Make.config.*` Files

The `Make.config.settings` and `Make.config.override` files

The default configuration settings and current configuration settings are stored in the two files `Make.config.settings` and `Make.config.override`.

The **Make.config.settings** file consists of assignments to the `CONFIG_*` make variables that define the default configuration settings in `Enzo`'s makefile. This file should not be modified lightly. If you type `gmake default`, then these will become the currently active settings.

The **Make.config.override** file, together with the `Make.config.settings` file, define the current configuration settings. This file should also not be edited (since misspelled configuration variable names may not be detected, leading to behavior that is unexpected and difficult to locate), though it will be modified indirectly through `gmake` when setting new configuration values. For example, if you were to type `gmake integers-32`, then the `Make.config.override` file would contain `CONFIG_INTEGERS = 32`. The values in the `Make.config.override` file essentially override the settings in `Make.config.settings`.

In summary:

default settings = **Make.config.settings**

current settings = **Make.config.settings + Make.config.override**

Typing `gmake default` will clear the `Make.config.override` file entirely, making the default settings in `Make.config.settings` the current settings.

The `Make.config.objects` file

This file is used simply to define the list of all object files, excluding the file containing `main()`. Only one variable needs to be set.

OBJS_CONFIG_LIB	List of all object files excluding the file containing `main()`

Dependencies are generated automatically using the makedepend command and stored in the `DEPEND` file, so dependencies don't need to be explicitly included. If it complains about missing files, such as `DEPEND` or `Make.config.override`, then try (re)-running the `./configure` script in the top-level `Enzo` subdirectory.

The `Make.config.targets` file

This file contains rules for all configuration-related make targets. It exists mainly to reduce the size of the top-level Makefile. When adding new configuration settings, this file will need to be modified.

The `Make.config.assemble` file

This file contains all the makefile magic to convert configuration settings (defined by `$(CONFIG_*)` make variables) into appropriate compiler flags (such as `$(DEFINES)`, `$(INCLUDES)`, etc.). When adding a new configuration setting, this file will need to be modified.

James Bordner (jobordner at ucsd.edu)

7.8 Parallel Root Grid IO

Parallel Root Grid IO (PRGIO) is a set of Enzo behaviors that allow the user to run problems that has a root grid larger than the available memory on a single node.

This page is intended for developers that need to write new problem generators that will be run at extremely large scale. Large problem size will need to utilize the PRGIO machinery in Enzo. As this brings a significant amount of added complexity, it isn't recommended for smaller problems. It is also recommended that you write the problem generator without this machinery first, and test on smaller problems, before adding the additional complexity. If you don't intend to write your own problem generator, this page is basically irrelevant.

7.8.1 Background: why it is how it is

PRGIO is an essential component of doing any simulations at large scale. In its initial inception, Enzo worked on shared memory machines. This meant that the total computer memory available dictated the problem size. Enzo would allocate the root grid on the root processor, then distribute spatially decomposed parts of the root grid to the other processors. When it came time to write the data, the root grid was collected back to the root processor, and written in a single file.

This worked fine until distributed computers were deployed in response to the limitations of a shared memory computer. This coincided with a growth of the desired root grid size for the Enzo simulation. Now, the total aggregate memory of a single shared memory computer and the memory required were vastly different. The old model broke down because you simply can't fit the $15x512^3$arrays you need in 512 Mb of RAM, but you can on 64 nodes if the memory is taken as an aggregate total. So out of necessity, PRGIO was born.

7.8.2 Short version

Essentially, PRGIO has three components (though not called in this order)

- *Input/Restart*
- *Output*
- *Initialization*

Input and Restarting

During initialization, the root grid is partitioned into tiles, and each processor reads the part, i.e. a HDF5 hyperslab, of the initial data files. For restarts, each grid is read by one processor that owns the data (`ProcessorNumber == MyProcessorNumber`) from the HDF5 file containing it.

Output

Unlike early versions of Enzo that collected all the grid data on one processor before writing to disk, with PRGIO each processor writes an HDF5 file for each grid it owns. In the packed AMR output mode, each processor writes one HDF5 file, and in it go all the grids it owns.

Initialization

This is the part that needs attention, because the details are not obvious from the code itself.

Initialization **BEFORE** PRGIO happens in three steps:

- Set up grid
- Allocate Data on the `TopGrid` object, on the Root Processor
- Partition `TopGrid` across processors.

WITH PRGIO, the order is different:

- Set up grid
- Partition `TopGrid`
- Allocate Data on the *working* grids.

Setup and Allocation

This is pretty straightforward in principle, but the implementation is a little confusing.

First grids need to be set up. There aren't very many things you need to do. See *MyProblemInitializeGrid* for a more comprehensive overview. Simplified, a count of the `NumberOfBaryonFields` is made and a record of which field is which goes in the `FieldType` array.

After the Partition (next section), you need to allocate the data.

The confusing bits are in the implementation. We'll describe this by way of example, using Cosmology simulations as our descriptor. `CosmologySimulationInitialize.C` contains two routines: `CosmologySimulationInitialize()` (CSI) and `CosmologySimulationReInitialize()` (CSRI). These are both called in `InitializeNew()`. The job of the first routine is to set up the hierarchy of grids and subgrids you'll need for your cosmology simulation, and call `CosmologySimulationInitializeGrid` (CSIG). Both CSI and CSIG are called whether or not PRGIO is on. CSRI is called from `InitializeNew()` after the Top Grid is partitioned. It is *only* called when PRGIO is on.

Stated a different way:

1. InitializeNew: reads the parameter file, then calls
2. CosmologySimulationInitialize: sets up the grid hierarchy. On each of those grids gets called
3. CosmologySimulationInitializeGrid: which sets NumberOfBaryonFields, and may allocate data.
4. PartitionGrid: breaks the root grid into parts, and sends those parts to the other processors.
5. CosmologySimulationReInitialize: If PRGIO is on, this is called. It loops over grids and calls CosmologySimulationInitializeGrid again, which allocates and defines the data.

CSI passes a flag, `TotalRefinement` to CSIG for each grid you initialize. This is equal to (refinement factor)$^{\text{(refinement level of this grid)}}$. So for the Top grid, this is equal to 1, and something that is greater than 1 on all other grids.

Inside of CSIG: if PRGIO is on **and** `TotalRefinement` == 1, then statements relating to reading data from disk, allocating memory, and accessing memory are **skipped.** (this is done by setting `ReadData = FALSE`) In all other cases, it's left on. (So if PRGIO is off, or **this grid** is not on the root level.) Thus at the first pass at initialization, the `TopGrid` doesn't get it's `BaryonFields` allocated.

The same procedure is done on the nested initial grids if `PartitionNestedGrids` == 1. If not, the root processor will read the entire nested grid, partition it into smaller subgrids, and finally send the data to different processors if `LoadBalancing > 0`. Regardless of the value of `PartitionNestedGrids`, the partitions of the static nested grids will never be re-combined for I/O, unlike the behavior of the root grid when PRGIO is off.

CSRI is called AFTER the root grid has been partitioned and sent off to the other processors. It does very little except call CSIG again. This time when CSIG is called, `TotalRefinement = -1`. This allows the data to be allocated.

7.8.3 Partition TopGrid and /* bad kludge */

The other confusing part the partition, specifically a line in `ExternalBoundary::Prepare()`.

```
if (ParallelRootGridIO == TRUE)
    TopGrid->NumberOfBaryonFields = 0; /* bad kludge! */
```

More on that in a moment.

`CommunicationPartitionGrid()` is the routine that takes the `TopGrid` (or, any grid) and breaks it across the processors. It first sorts out the layout of the processors with `MPI_Dims_create()`. It then evenly splits the initial grid over those processors by first creating a new grid on each tile, linking them to the Hierarchy linked list. It then (and here's the tricky part) allocates each grid on the Root processor and copies data from the Initial Grid to the new tile. Finally, it take these freshly created root grid tiles and sends them to their new processor home.

Here's where the **bad kludge!** comes in. You'll note that in the above description, there's an allocate on each of the newly created tiles *on the root processor*, which will allocate more than the root grid data. This is the problem we were trying to avoid. So `ExternalBoundary::Prepare()` sets `NumberOfBaryonFields` to zero, so when the allocate comes around it's allocating Zero fields.

Why is it in `ExternalBoundary::Prepare()`? A look at the lines immediately preceding the 'kludge' help:

```
BoundaryRank = TopGrid->GridRank;
NumberOfBaryonFields = TopGrid->NumberOfBaryonFields;
if (ParallelRootGridIO == TRUE)
  TopGrid->NumberOfBaryonFields = 0; /* bad kludge! */
```

In order to do its job properly, the `ExternalBoundary` objects need to know how many `BaryonFields` there are in the simulation. So `ExternalBoundary::Prepare()` records the data, and because that's the last place `NumberOfBaryonFields` is needed, sets it to zero.

When `CommunicationPartitionGrid()` gets to the point where it allocates the data, `NumberOfBaryonFields` is now zero, so it allocates no data. These empty root grid tiles are then distributed to the other processors.

Finally, `CosmologyReInitialize()` is called, which calls `CosmologyInitializeGrid()`. This code then resets `NumberOfBaryonFields` to its proper value, and since `TotalRefinement = -1` allocates all the data.

Then the simulation continues on, only aware of PRGIO when it comes time to not collect the data again.

7.9 Getting Around the Hierarchy: Linked Lists in Enzo

There are two primary linked lists in Enzo; `HierarchyEntry` and `LevelHierarchyEntry`. They're both used to traverse the hierarchy, but in very different ways. `HierarchyEntry` is used to traverse **down** the hierarchy, from

a parent to its children. `LevelHierarchyEntry` is used to traverse **across** the hierarchy, on a single level.

One of the primary things to note about the two lists is that `NextGridThisLevel` (which exists in both) serve different purposes.

In `LevelHierarchyEntry`, `NextGridThisLevel` links all the grids on a given level together.

In `HierarchyEntry`, `NextGridThisLevel` only counts things on a given level that share a parent.

Below we will present a description of the structures and their creation and usage in Enzo.

7.9.1 HierarchyEntry

The `HierarchyEntry` linked list is used for traversing *down* the hierarchy, from parents to children.

This is the contents of the definition of the structure, which you can find in `src/enzo/Hierarchy.h`.

```
struct HierarchyEntry
{
  HierarchyEntry *NextGridThisLevel; /* pointer to the next grid on level */
  HierarchyEntry *NextGridNextLevel; /* pointer to first child of this grid */
  HierarchyEntry *ParentGrid;        /* pointer to this grid's parent */
  grid           *GridData;          /* pointer to this grid's data */
};
```

`NextGridThisLevel` connects all children of a parent. `NextGridNextLevel` points to the first child of the given grid. `ParentGrid` connects to the parent, and `GridData` points to the actual grid structure.

Usage of HierarchyEntry lists

The `HierarchyEntry` list is used (among other things) whenever communication between child and parent grids needs to be done. The typical pattern for looping over all the children of a parent grid is as following:

```
1  HierarchyEntry * NextGrid = ParentGrid->NextGridNextLevel;
2  while (NextGrid != NULL ){
3    if (NextGrid->GridData->SomeFunctionOnChildren(args) == FAIL )
4      fprintf(stderr, "Error in your function\n");
5      return FAIL;
6    }
7    NextGrid = NextGrid->NextGridThisLevel;
8  }
```

Line 1 sets the pointer `NextGrid` to the "first" child of the parent grid.

Line 2 starts the while loop.

Lines 3-6 is the standard function call pattern in Enzo.

Line 7 advances the pointer to the next child on the *child* level.

This loop stops once all the children of `ParentGrid` have been accessed, because the last child grid of a given parent has NULL as `NextGridThisLevel`.

Generation of HierarchyEntry lists

The `HierarchyEntry` linked list is generated in several different points in the code. The details are slightly different for each place it's used, depending on the details of what that linked list is used for and the assumed structure of the hierarchy at that point. The list most used in the code is the one generated in `src/enzo/FindSubgrids.C`, called

in src/enzo/RebuildHierarchy.C. This code is called on a single 'Parent Grid' at a time. Paraphrased and annotated:

```
1  HierarchyEntry *, *ThisGrid;
2  PreviousGrid = &ParentGrid;
3   for (i = 0; i < NumberOfSubgrids; i++) {
4
5     ThisGrid = new HierarchyEntry;
6
7     if (PreviousGrid == &ParentGrid)
8       ParentGrid.NextGridNextLevel = ThisGrid;
9     else
10      PreviousGrid->NextGridThisLevel = ThisGrid;
11    ThisGrid->NextGridNextLevel = NULL;
12    ThisGrid->NextGridThisLevel = NULL;
13    ThisGrid->ParentGrid        = &ParentGrid;
14
15    ThisGrid->GridData = new grid;
16    ThisGrid->GridData = Setup Functions Skipped for clarity;
17
18    PreviousGrid = ThisGrid;
19 }
```

Line 1 starts the HierarchyEntry list with ParentGrid. (Called simply Grid in the source, changed here for clarity.)

Line 5 creates the next HierarchyEntry to be added to the list.

Line 7-8 attaches the new subgrid, and the ensuing subgrid chain, to the parent grid (note that this is only done for the first new subgrid)

line 10 attaches all subsequent new subgrids to the NextGridThisLevel chain.

Lines 11 and 12 ensure that both lists terminate with this new grid. NextGridThisLevel will be replaced if there is in fact a next grid. Since this routine is called only on a single Parent at a time, one can now see that for HierarchyEntry, the NextGridThisLevel list only links children that belong to the same Parent Grid.

Lines 13-17 finish setting up this grid.

If you're writing a new problem generator, and have been brought here by the AMR problem generation page, we advise that you examine one of the other code patterns that are used in Enzo. They look fairly similar to the above code, though have some details different. Some suggestions are:

For adding a single subgrid, visit src/enzo/SphericalInfallInitialize.C.

For adding a single stack of nested subgrids, see /src/enzo/ProtostellarCollapseInitialize.C.

For a completely general, though more complex setup, see src/enzo/CosmologySimulationInitialize.C.

Another notable routine that generates HierarchyEntry lists is src/enzo/CommunicationPartitionGrid.C, which breaks the TopGrid pointer across multiple processors.

7.9.2 LevelHierarchyEntry and LevelArray

The LevelHierarchyEntry Linked List is used for traversing all the grids on a given level. It's a simpler structure than HierarchyEntry. The source can be found in src/enzo/LevelHierarchy.h.

```
struct LevelHierarchyEntry
{
  LevelHierarchyEntry *NextGridThisLevel;  /* next entry on this level */
  grid                *GridData;           /* pointer to this entry's grid */
```

```
  HierarchyEntry      *GridHierarchyEntry; /* pointer into hierarchy */
};
```

`NextGridThisLevel` connects all grids on a given level. `GridData` points to the actual grid object, and `GridHierarchyEntry` points to the (unique) `HierarchyEntry` node discussed above.

The `LevelHierarchyEntry` lists, one for each populated level, are all bundled together in the `LevelArray` object. Both data structures will be discussed presently.

Usage of LevelHierarchyEntry and LevelArray

The main usage of the `LevelHierarchyEntry` list is quite similar to the main loop for `HierarchyEntry` lists.

```
LevelHierarchyEntry *Temp = LevelArray[level];
while (Temp != NULL) {
  if (Temp->GridData->MyCode(MyArgs) == FAIL) {
    fprintf(stderr, "Error in grid->SetExternalBoundaryValues.\n");
    return FAIL;
  }
  Temp = Temp->NextGridThisLevel;
}
```

This calls MyCode for each grid on level.

Generation of LevelHierarchyEntry and LevelArray

This is done in two places in the code: in `src/enzo/main.C` `main.C` and `src/enzo/RebuildHierarchy.C`. It's done by the code `src/enzo/LevelHierarchy_AddLevel.C`, which is described below.

The setup, prep in main.C:

```
for (int level = 0; level < MAX_DEPTH_OF_HIERARCHY; level++)
  LevelArray[level] = NULL;
```

The call in `main()`:

```
AddLevel(LevelArray, &TopGrid, 0);
```

The fill:

```
1  void AddLevel(LevelHierarchyEntry *LevelArray[], HierarchyEntry *Grid,
2                int level)
3  {
4     LevelHierarchyEntry *ThisLevel;
5
6    /* create a new LevelHierarchyEntry for the HierarchyEntry Grid
7       and insert it into the head of the linked list (LevelArray[level]). */
8
9    ThisLevel = new LevelHierarchyEntry;
10   ThisLevel->GridData = Grid->GridData;
11   ThisLevel->NextGridThisLevel = LevelArray[level];
12   ThisLevel->GridHierarchyEntry = Grid;
13   LevelArray[level] = ThisLevel;
14
15   /* recursively call this for the next grid on this level. */
16
```

```
17    if (Grid->NextGridThisLevel != NULL)
18      AddLevel(LevelArray, Grid->NextGridThisLevel, level);
19
20    /* ... and then descend the tree. */
21
22    if (Grid->NextGridNextLevel != NULL)
23      AddLevel(LevelArray, Grid->NextGridNextLevel, level+1);
24   }
```

This is a recursive function that takes `LevelArray` that's to be filled, the `HierarchyEntry` list that fills it, and a counter for the level. It's recursive in both `HierarchyEntry`'s lists, both `NextGridNextLevel` and `NextGridThisLevel`. The most notable lines are 11, 13, and 17. In lines 11 and 13, one can see that the current `HierarchyEntry` is attached to the HEAD of the list, but line 17 shows that the `HierarchyEntry` list is traversed from its head to its tail: so the `LevelArray` list is backwards from the `HierarchyEntry`. This is only really needed information on the top grid.

7.9.3 Traversing the Entire Hierarchy

Sometimes the user needs to traverse the entire hierarchy. This is done with a recursive function call on the `HierarchyEntry`. This should be done in a manner akin to the `AddLevel` code above.

7.10 Machine Specific Notes

Here we will mention some miscellaneous notes on specific machines. This is merely a list of pitfalls or things we have found useful, and by no means a replacement to the documentation.

7.10.1 NICS: Kraken

http://www.nics.tennessee.edu/computing-resources/kraken

Important

Serious errors have been found with a few Enzo routines when using -O2 and the PGI compilers on Kraken. Use with caution.

Trace Trap Flags

Useful for debugging, but slows the code down. You can find this info in the pgCC man page. (Not all compilers have decent trace trapping, so it deserves a mention here.)

```
-Ktrap=[option,[option]...]
       Controls the behavior of the processor when
       exceptions occur.  Possible options include
       -Ktrap=divz  Trap on divide by zero.
       -Ktrap=fp  Trap on floating point exceptions.
       -Ktrap=align Trap on memory alignment errors, currently ignored
       -Ktrap=denorm Trap on denormalized operands.
       -Ktrap=inexact Trap on inexact result.
       -Ktrap=inv Trap on invalid operands.
       -Ktrap=none (default)   Disable all traps.
```

```
-Ktrap=ovf Trap on floating point overflow.
-Ktrap=unf Trap on floating point underflow.
```

7.11 Particles in Nested Grid Cosmology Simulations

When running a nested grid cosmology simulation, not all the particles created by inits necessarily lie inside of the intended grid. This has to do with they way particle positions are calculated from the velocity field. This problem is not a flaw in the way inits makes initial conditions, but it can lead to unreliable results if it is not addressed.

Note: This effect does not always occur. But it should be checked for when doing nested initial conditions.

7.11.1 The Problem

Following the *cosmology tutorial* for *nested grids*, first inits is run, and then ring is run on the output of inits to prepare data for the Parallel Root Grid IO mode of Enzo. The contents of the initial conditions are easily inspected:

```
$ h5ls Particle*
ParticleMasses.0         Dataset {1, 2064384}
ParticleMasses.1         Dataset {1, 262144}
ParticlePositions.0      Dataset {3, 2064384}
ParticlePositions.1      Dataset {3, 262144}
ParticleVelocities.0     Dataset {3, 2064384}
ParticleVelocities.1     Dataset {3, 262144}
```

In this example, there are two initial grids. The root grid has 2,064,384 particles, and the nested grid has 262,144. After ring is run, a number of files with prefixes `PPos`, `PVel` and `PMass` are created. Using eight tasks, here are the contents of the PPos files for the top grid:

```
$ h5ls PP*0
PPos0000.0               Dataset {3, 258304}
PPos0001.0               Dataset {3, 258304}
PPos0002.0               Dataset {3, 257792}
PPos0003.0               Dataset {3, 257792}
PPos0004.0               Dataset {3, 258304}
PPos0005.0               Dataset {3, 258304}
PPos0006.0               Dataset {3, 257792}
PPos0007.0               Dataset {3, 257792}
```

And the nested grid:

```
$ h5ls PP*1
PPos0000.1               Dataset {3, 32743}
PPos0001.1               Dataset {3, 32665}
PPos0002.1               Dataset {3, 32767}
PPos0003.1               Dataset {3, 32844}
PPos0004.1               Dataset {3, 32715}
PPos0005.1               Dataset {3, 32151}
PPos0006.1               Dataset {3, 32749}
PPos0007.1               Dataset {3, 32692}
```

The sum of the particles in the top grid files is 2,064,384 particles, but in the nested grid files it is only 261,326, a deficit of 818 particles. The missing particles have been thrown out by ring because they lie outside the nested grid boundaries.

If the sum of the particles in the files after ring has been run is equal to the original total, the problem is not extant in the dataset.

7.11.2 The Solution

The solution to this problem is to introduce an extra step between inits and ring, where particles are moved to the correct grid. However, when a particle is moved to a grid with a different refinement, the mass of the particle must be modified. During this step, when a particle changes grid, this move must be tracked and its mass updated to reflect the different grid refinement. Please see *Writing your own tools, II - Enzo Physical Units* for more on why the particle mass must be changed when moving between grids.

One wrinkle to this solution is the `ParticleMasses` file *must* be created by inits, for all grids, along with the `ParticlePositions` and `ParticleVelocities files`. `CosmologySimulationParticleMassName` must therefore also be specified as an input in the Enzo parameter file.

Linked here is a simple Python script that will fix the initial condition files. After running the script, run ring on the new initial condition files. The script requires a Python installation that has both Numpy and h5py. A simple way to gain an installation of Python with these modules is to install yt, which is one of the *data analysis tools* available for Enzo.

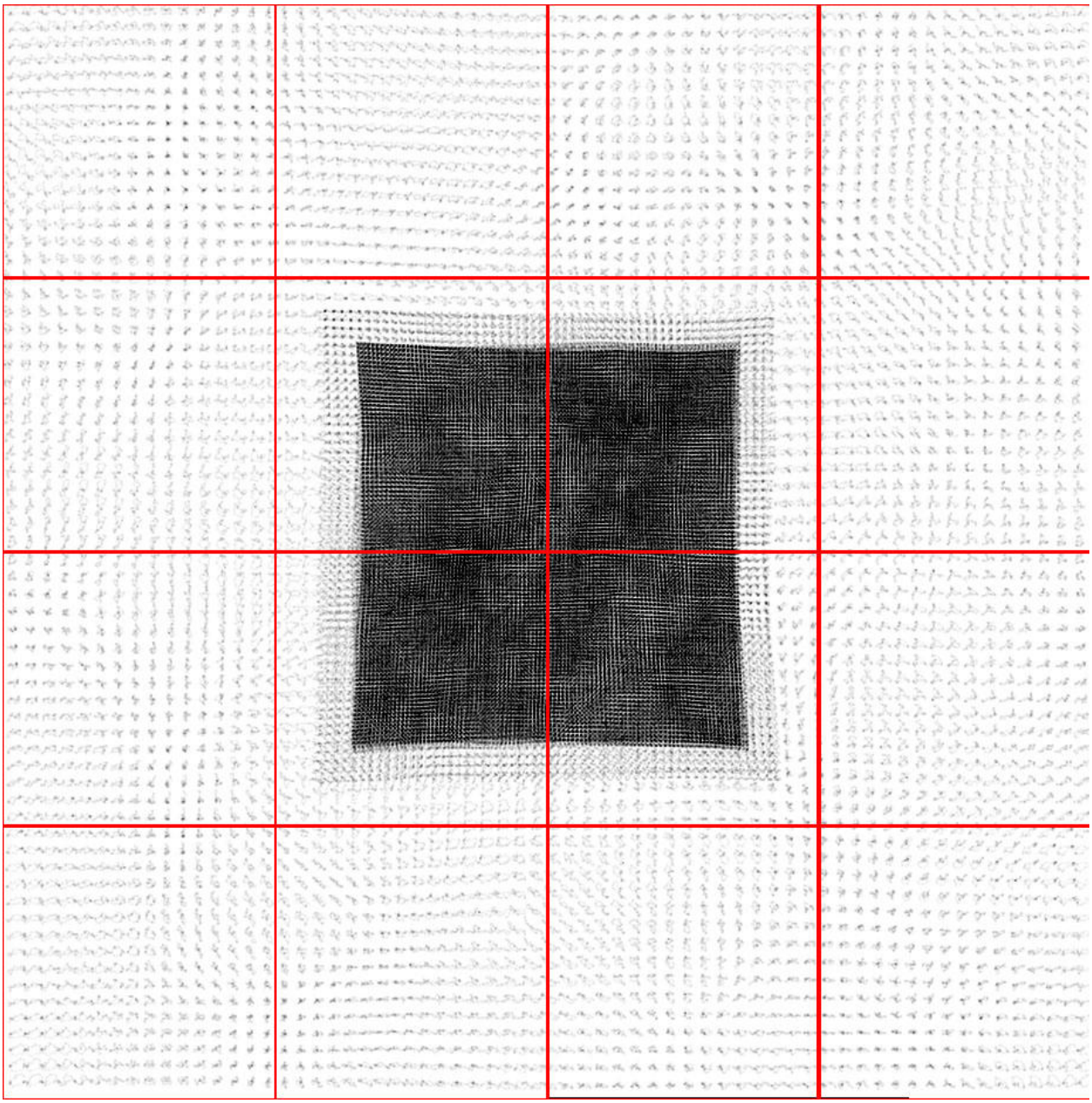
7.11.3 Procedure

Save a copy of the script to the same directory as your nested initial condition files. Edit the top of the file, where noted, to match your setup. Please note the order items should be entered. Once the settings are correct, invoke `python inits_sort.py`. The updated initial condition files will be placed inside the directory `new_ICs`. Then run ring on the new initial condition files, and use the results with Enzo.

7.12 Nested Grid Particle Storage in RebuildHierarchy

7.12.1 Problem

In the previous version of `RebuildHierarchy()`, all of the particles were moved to the parent on the level L_0 being rebuilt. This causes problems when running large simulations with nested initial grids because a small number of top-level grids cover the refine region, compared to the total number of top-level grids. This is illustrated in the figure below.



On distributed memory machines, only one (or more) top-level grid exists on one processor. The particles are stored only on the host processor, stored in `grid::ProcessorNumber`. This processor will run out of memory if a large number of particles are moved exclusively to a grid on this processor.

7.12.2 Solution

We can avoid this memory oversubscription by temporarily keeping the particles on the processor from the previous timestep, i.e. the processor of the original child grid, during the rebuild process. However, we still want to move the particles to the parent grid on level L_0 because we will be rebuilding this and finer levels from the data existing on these grids.

This is only necessary on levels with static subgrids because on levels with dynamics hierarchies the grids will be distributed across processors sufficiently to avoid this problem. On the levels with static subgrids, we depart from

the standard particle storage in Enzo, where the particles are stored on one processor and `NumberOfParticles` is the same on all processors. We adopt the strategy of storing particles on many processors for one grid, and `NumberOfParticles` denotes the number of particles actually stored on the local processor. Once we rebuild the coarsest level with a dynamical hierarchy, we move all of the particles to their host processor, i.e. `ProcessorNumber`, and synchronize `NumberOfParticles` to equal the total number of particles on the grid over all processors.

Below we will outline this method to distribute memory usage from particles during `RebuildHierarchy()` on level L. Pre-existing routines in `RebuildHierarchy()` are not included in the outline.

1. Set `NumberOfParticles` to zero on all grids on level >= L, except on the grid's host processor.
2. Find the finest level (L_{sub}) with static subgrids. In the code, this is called `MaximumStaticSubgridLevel`.
3. `grid::MoveAllParticles()` – Move all particles on grids on level > L to their parents on level L, but keep them on the same processor as before. Now the particles are on their parent, but distributed across many processors.
4. `CommunicationTransferParticles()` – Move any particles that have migrated across grid boundaries to their siblings.
5. `CommunicationCollectParticles(SIBLINGS_ONLY)` – If we are rebuilding a level > L_{sub}, move all particles to their host processor, as this new method is not needed. This was previously done in `grid::MoveAllParticles`. This routine is faster than before because we do the communication in one `MPI_Alltoallv()` call.
6. Loop over levels L_0 -> `MAX_DEPTH_OF_HIERARCHY`.
7. `DepositParticleMassFlaggingField()` – If level <= L_{sub}, then the particles are distributed across processor. This causes complications when creating the mass refinement flagging field for particles. Therefore, we must sum this particle mass field over these processors. For each grid, only processors with particles contribute to this sum to reduce the amount of computation and communication. In short, this routine performs a non-blocking `MPI_SUM` over a select number of processors.
8. `CommunicationCollectParticles(SUBGRIDS_LOCAL)` – This routine replaces `grid::MoveSubgridParticlesFast()`. It keeps the particles on the same processor, but this doesn't matter here because the children grids are always created on the same processor as its parent and then moved to another processor during load balancing.
9. `CommunicationCollectParticles(SIBLINGS_ONLY)` – After load balancing is complete on level L_{sub}, we can safely move the particles to their host processor without the worry of running out of memory.

7.13 Estimated Simulation Resource Requirements

Estimating problem sizes for most Enzo calculations is at best an inexact science, given the nature of Adaptive Mesh Refinement (AMR) simulations. The fundamental issue with an AMR calculation in cosmology or in many astrophysical situations where gravitational collapse is important has to do with memory. The amount of memory used at the beginning of the simulation (when you have a single grid or a handful of grids) is far, far less than the memory consumption at the end of the simulation, when there can be hundreds of grids per processor. The amount of memory required can easily grow by an order of magnitude over the course of a cosmological simulation, so it is very important to make sure to take this into account to ensure that enough memory is available in later stages of your simulation. It is also important to realize that in general one should try to keep the largest amount of data per processing core that you can so that individual cores are never data-starved. Data-starved processing units cause poor scaling, as your CPUs will then be sitting idle while waiting for data from other computing nodes. Computational fluid dynamics simulations are notoriously communication-heavy, making this a challenging corner of parameter space to operate in.

This page contains some rules of thumb that will help you along your way, based on data collected up to the release of Enzo v1.5 (so up to Fall 2008), when supercomputers typically have 1GB-2GB of memory per processing unit (a dual-processor node with two cores per processor would have 4-8 GB of memory, for example).

7.13.1 Cosmology or non-cosmology unigrid (non-AMR) simulations

These are actually quite straightforward to predict, given that in a unigrid simulation the grid is partitioned up in an approximately equal fashion and then left alone. Experimentation shows that, for machines with 1-2 GB of memory per core, one gets near-ideal scaling with 128^3 cells per core (so a 512^3cell calculations should be run on 64 processors, and a 1024^3 cell run should be done on 512 processors). This is comfortably within memory limits for non-cosmology runs, and there is no danger of running up against a node's memory ceiling (which causes tremendous slowdown, if not outright program failure). Unigrid cosmology runs have a further complication due to the dark matter particles - these move around in space, and thus move from processor to processor. Areas where halos and other cosmological structures form will correspond to regions with greater than average memory consumption. Keeping 128^3cells and particles per core seems to scale extremely efficiently up to thousands of processors, though if one is using a machine like an IBM Blue Gene, which typically has far less memory per core than other computers, one might have to go to 64^3 cells/particles per core so that nodes corresponding to dense regions of the universe don't run out of memory.

7.13.2 Cosmology adaptive mesh simulations

Scaling and problem size is much more difficult to predict for an AMR cosmology run than for its unigrid equivalent. As discussed above, the amount of memory consumed can grow strongly over time. For example, a 512^3 root grid simulation with seven levels of adaptive mesh refinement started out with 512 root grid tiles, and ended up with over 400,000 grids! This calculation was run on 512 processors, though memory consumption grew to the point that it had to be run on a system where half of the cores per node were kept this particle mass field over these processors. For each grid, only processors with particles contribute to this sum to reduce the amount of computation and communication. In short, this routine performs a non-blocking `MPI_SUM` over a select number of processors.

`CommunicationCollectParticles(SUBGRIDS_LOCAL)` – This routine replaces `grid::MoveSubgridParticlesFast()`. It keeps the particles on the same processor, but this doesn't matter here because the children grids are always created on the same processor as its parent and then moved to another processor during load balancing. `CommunicationCollectParticles(SIBLINGS_ONLY)` – After load balancing is complete on level L_{sub}, we can safely move the particles to their host processor without the worry of running out of memory.

7.14 SetAccelerationBoundary (SAB)

One of the minor bugs in Enzo that was uncovered by the addition of MHD-CT is the boundary on the gravitational acceleration field.

Enzo currently solves gravity in two phases: first by Fast Fourier Transform on the root grid, then by multigrid relaxation on the subgrids. Unfortunately, each subgrid is solved as an individual problem, and is not very concious of its neighbours.

The problem with this is the ghost zones. Enzo MHD-CT is not a divergence *free* method, but a divergence *preserving* method. There isn't a mechanism that reduces the divergence of the magnetic field. Unfortunately, inconsistencies in *any* fluid quantity can lead to divergence in the magnetic field. The magnetic field is stored on the faces of each computational zone, and are updated by an electric field that is stored on the edges. Since this data sits in the face of the zone, whenever two grids abut, they share a face, so it is vital that both grids describe everything in the stencil of the face centered fields identically, otherwise they will get different results for the magnetic field on that face, and divergence will be generated. It was noticed that in the case of the `AccelerationField` that due to the isolated nature of the gravity solver, the ghost zones of a subgrid didn't necessarily equal the active zones of grids that were next

to it. Thus the Magnetic fields in the shared face would ultimately be computed slightly differently, and divergence would show up.

The proper fix for this is replacing the gravity solver with one that is aware of the entire subgrid hierarchy at once, but this is quite costly in both programmer time and in compute time. Work has begun on this project at the LCA, but has not yet been finished.

As an intermediate step, Enzo was hacked a little bit. Initially, the main loop in `EvolveLevel.C` looked like this:

```
for( grid=0, grid< NumberOfGrids, grid++){
   Grid[grid]->SolvePotential
   Grid[grid]->SolveHydroEquations
}
```

Among, of course, many other physics and support routines. This was broken into two loops, and a call to `SetBoundaryConditions()` as inserted between the two.

```
for( grid=0, grid< NumberOfGrids, grid++){
   Grid[grid]->SolvePotential
}
SetBoundaryConditions
for( grid=0, grid< NumberOfGrids, grid++){
   Grid[grid]->SolveHydroEquations
}
```

However, since `SetBoundaryConditions()` doesn't natively know about the `AccelerationField`, another kludge was done. A new set of pointers `ActualBaryonField` was added to `Grid.h`, and the true pointers are saved here, while the `BaryonField` array is temporarily pointed to `AccelerationField`. This saved a substantial rewrite of the boundary setting routines, at the expense of some less-than-ideal code.

This is not a bug that makes much difference overall in cosmology simulations, and it does not solve the problem of artificial fragmentation that has been noticed by some groups. Cosmology tests have been done that compare solutions both with and without this fix, and only negligible changes appear. So for most runs, it simply adds the expense of an extra boundary condition set. However, with MHD-CT runs it is absolutely necessary, for explosive divergence will show up. Additionally, and other simulations that are extremely sensitive to overall conservation or consistency will require this flag. In any condition where the user is potentially concerned about we suggest running a test both with and without `SAB`, and comparing the answers. `SAB` brings the compuational expense of an additional boundary condition call, and the memory expense of three global fields, since without it the `AccelerationField` exists only on a single grid at a time, while with it all three fields must be created on the entire hierarchy at once. This is not a major expense on either count for most simulations.

This is controled by the preprocessor directive `SAB`. If this is defined, the necessary steps are taken to call the acceleration boundary. In the file machine make file, `Make.mach.machine-name`, this should be added to the variable `MACH_DEFINES`

7.15 Star Particle Class

7.15.1 Purpose

To give star particles more functionality and interaction with the grids, it was useful to create a new class for a generic particle type that can represent, e.g., stars, black holes, sink particles.

7.15.2 Main features

- merging

- accretion
- conversion to a radiation source
- adding feedback spheres to the grid, e.g. mass removal from accretion, supernovae.
- different behaviors for different star types
- multiple types of star particles
- "active" and "inactive" stars

7.15.3 Approach

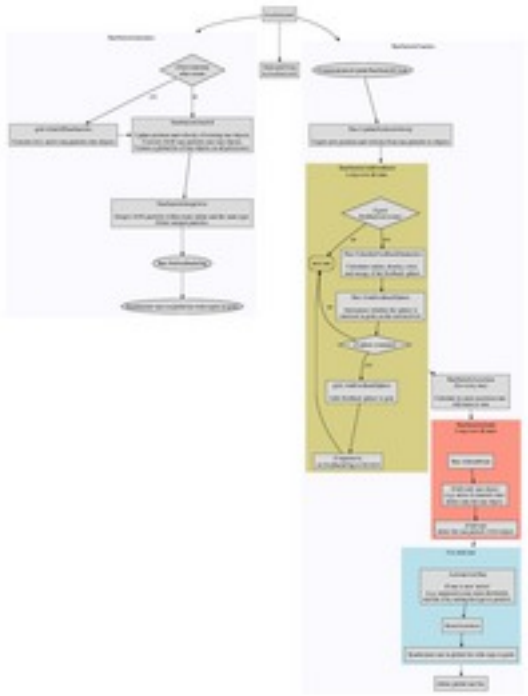


Figure 7.2: A flowchart of the logic of the star particle class. `View PDF`.

We keep the original implementation of the particles that are stored in the pointers, `ParticlePosition`, `ParticleVelocity`, `ParticleMass`, `ParticleNumber`, `ParticleType`, and `ParticleAttribute`. Star particles are still created in the FORTRAN routines, e.g. `star_maker2.F`. In the current version, the star class is a layer on top of these particles. Thus we must keep the particle pointers and objects synchronized when their quantities change.

Particles created in the FORTRAN routines that will be converted into a star object initially have a negative particle type. This indicates that the star is not "born" yet, which is also used to flag various feedback spheres, such as mass removal from the grid. The stars are activated, i.e. positive particle type, in `Star::ActivateNewStar()` after it has been checked for mergers, accretion, and feedback.

We store the star objects as a linked list in grid class. Because a star object can affect multiple grids (over multiple processors) when adding feedback sphere, processors other than the one hosting the star particle needs to know about this star object. Currently for convenience, we create a global list of star objects on all processors. For not many stars (< 100k), this does not consume that much memory. However in the future, we might have to reconsider how star particles are communicated across processors.

Feedback spheres

Any event can be set in `Star::SetFeedbackFlag` to add a feedback sphere. This sphere can be of any size, and its properties are set in `Star::CalculateFeedbackParameters()` and `grid::AddFeedbackSphere()`. Because they can cover grids on multiple levels, we have to ensure that they are all at the same time. In `Star::FindFeedbackSphere()`, we check if sphere is completely contained within grids on the current level. If true, we can safely add the sphere. If it's not imperative that the grids are completely synchronized, one can add the feedback sphere immediate after the star object is flagged for feedback.

Accretion / Mass Loss

Star objects can store up to 100 (`#define MAX_ACCR`) accretion rates as a function of time. Alternatively, currently in the black hole particles, they can have an instantaneous accretion rate. This is done in `Star::CalculateMassAccretion`. The actual accretion to the star object is done in `Star::Accrete()`.

7.15.4 How to add a new particle type

1. Set the particle type to the negative of the particle type in the star maker routine. Be sure not to overwrite the type like what's done in the regular `star_maker.F` routines.
2. Add the particle type to the if-statement in `grid::FindNewStarParticles`.
3. Then the particles merge if any exist within `StarClusterCombineRadius`. This is not restricted to only star cluster (radiating) particles. Even if there is any merging, the particle shouldn't disappear.
4. At the end of `StarParticleInitialize()`, the routine checks if any stars should be activated in `Star_SetFeedbackFlag`. This is where you should check first for errors or omissions. You'll have to add a new case to the switch statement. Something as simple as

   ```
   case NEW_PARTICLE_TYPE:
   if (this->type < 0)
      this->FeedbackFlag = FORMATION;
   else
      this->FeedbackFlag = NO_FEEDBACK;
   ```

   will work.

   After this, the particle is still negative but will be flipped after the feedback to the grid is applied in `Star_ActivateNewStar()` that's called from `StarParticleFinalize`. Here for Pop II and III stars, we use a mass criterion. For Pop III stars, we set the mass to zero in the `pop3_maker()` f77 routine, then only set the mass after we've applied the feedback sphere.
5. The grid feedback is added in `StarParticleAddFeedback` that is called in `StarParticleFinalize()`. In `Star_CalculateFeedbackParameters()`, you'll want to add an extra case to the switch statement that specifies the radius of the feedback sphere and its color (metal) density.
6. If the feedback sphere is covered by grids on the level calling `StarParticleAddFeedback()` (i.e. all of the cells will be at the same time), then `Grid_AddFeedbackSphere()` will be called. Here you'll have to add another if-block to add your color field to the grid.

7.16 Building the Documentation

The documentation for Enzo (including this very document) is built using the reStructuredText (ReST) syntax which is parsed into final formats using the Sphinx engine. Sphinx is a python package which may be installed using the pip

Python package installation tool like this:

```
$ pip install sphinx
```

Once that is installed, make sure that the binary `sphinx-build` is in your path (`$ which sphinx-build`). Relative to the top level of the Enzo package, the Enzo docs are in `doc/manual`. This directory contains a `Makefile` and a `source` directory. From within this directory, this command will parse the documents into a hierarchy of HTML files (identical what is on the web) into a new directory `build`:

```
$ make clean
$ make html
```

If that is successful, point your web browser to the file on disk (using the Open File... option of the File menu) `build/html/index.html` (this is relative to this same directory with the `Makefile`). On Mac OS X this command should work: `open build/html/index.html`. The docs should be nearly identical to what is online, but they are coming from the local machine.

7.16.1 Building a PDF of the Documentation

If (PDF)LaTeX is functional, is it possible to build a PDF of the Enzo documentation in one step. In the directory with the `Makefile`, use this command:

```
$ make pdflatex
```

If this is successful, the PDF will be `build/latex/Enzo.pdf`. The PDF might be preferred for some users, and can be searched all at once for a term, unlike a local copy of the HTML.

If PDFLaTeX is not working, `$ make latex` will not attempt to make the PDF. A PS or DVI (or whatever anachronistic thing your SPARCstation makes) can be made starting from `build/latex/Enzo.tex`.

7.16.2 Updating the Online Pre-Built Documentation

If you are an Enzo developer and need to update the current build of the documentation, these instructions should step you through that process. In the directory doc/manual, clone a copy of the docs repository and call it enzo-docs:

```
$ hg clone https://docs.enzo.googlecode.com/hg/ enzo-docs
```

The enzo-docs repository holds HTML and image files which are accessed over the web on the Google Code pages, and it is not modified manually.

In the directory doc/manual, execute these commands:

```
$ cd enzo-docs
$ hg pull
$ hg up
$ cd ..
$ rm -rf enzo-docs/*
$ make clean
$ make html
$ export CURRENT_REV="`hg identify`"
$ cp -Rv build/html/* enzo-docs/
$ cd enzo-docs
$ hg addremove --similarity=25
$ hg ci -m "Build from ${CURRENT_REV}"
$ hg push
```

The `addremove` extension will automatically update the current state of the files to be pushed with whatever is found in that directory. It will remove all files that it no longer sees and add all the files it currently does see. (You can supply `--dry-run` to see what it will do). The similarity argument just helps with keeping the size of the commits down, but because this repository is only to be used as a holding place for static content this should not be a problem.

CHAPTER

EIGHT

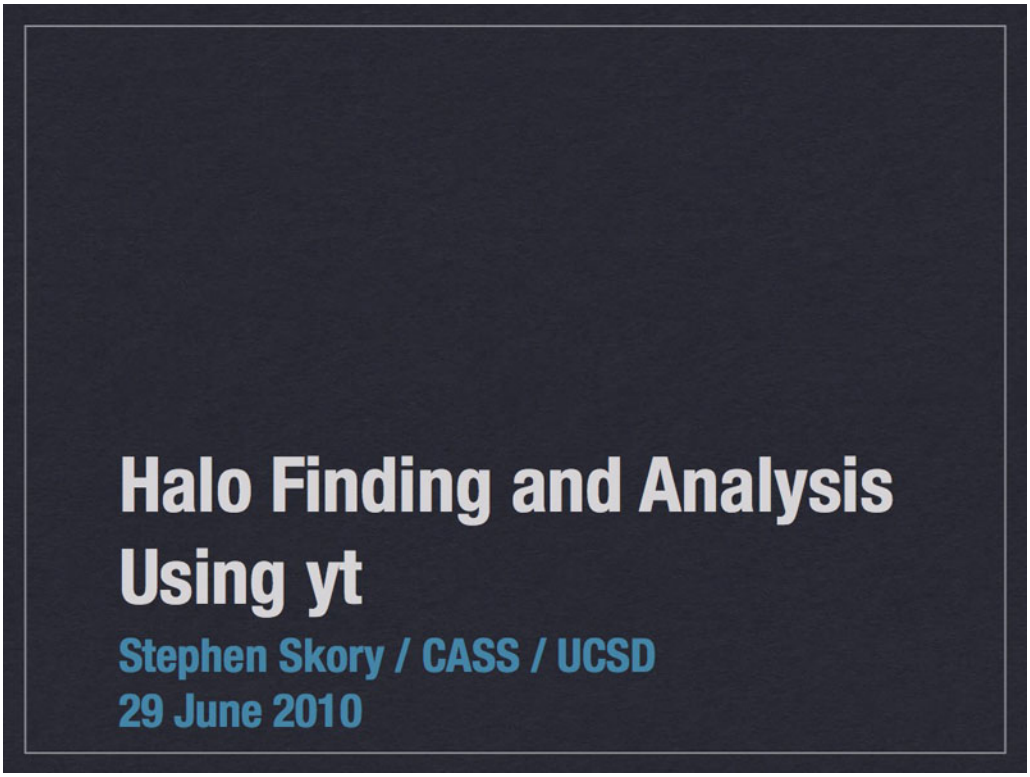
PRESENTATIONS GIVEN ABOUT ENZO

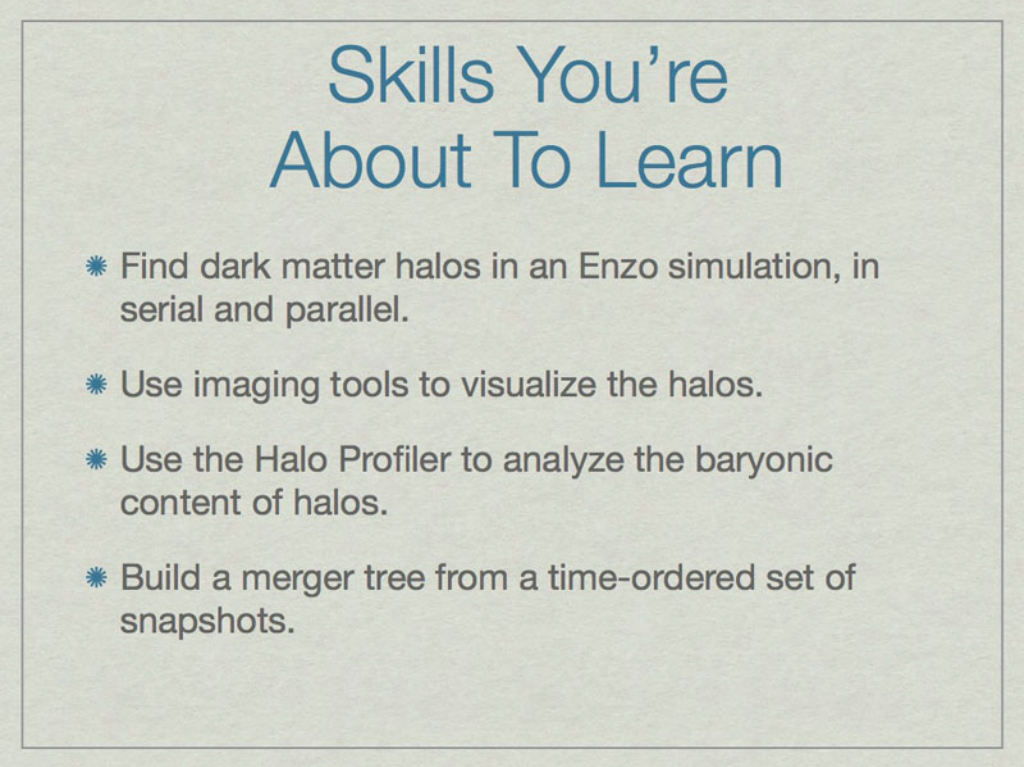
This is a collection of various presentations given about Enzo, or about work done with Enzo, or tools (with examples) to analyze Enzo data. There are some slides and videos available from the 2010 Enzo Workshop held in San Diego.

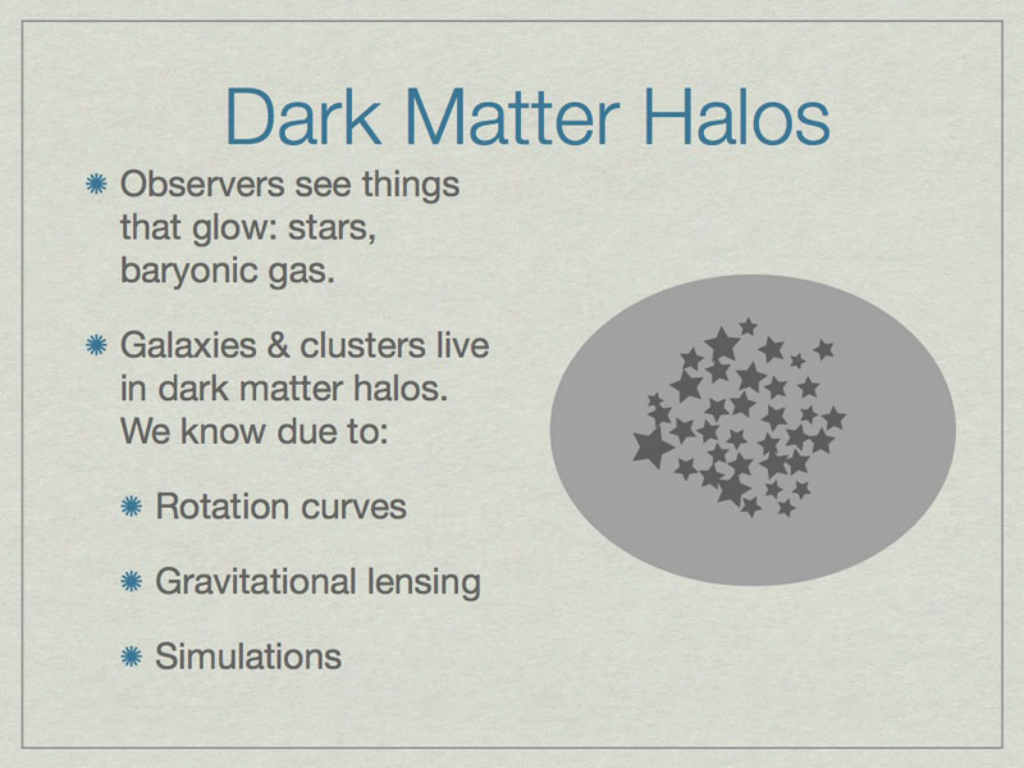
8.1 Halos and Halo Finding in yt

Below are the slides of a talk given by Stephen Skory at the 2010 Enzo Users Conference held June 28-30 at the San Diego Supercomputer Center. This talk introduces the three different methods of finding halos available in yt, and some of the other tools in yt that can analyze and visualize halos.

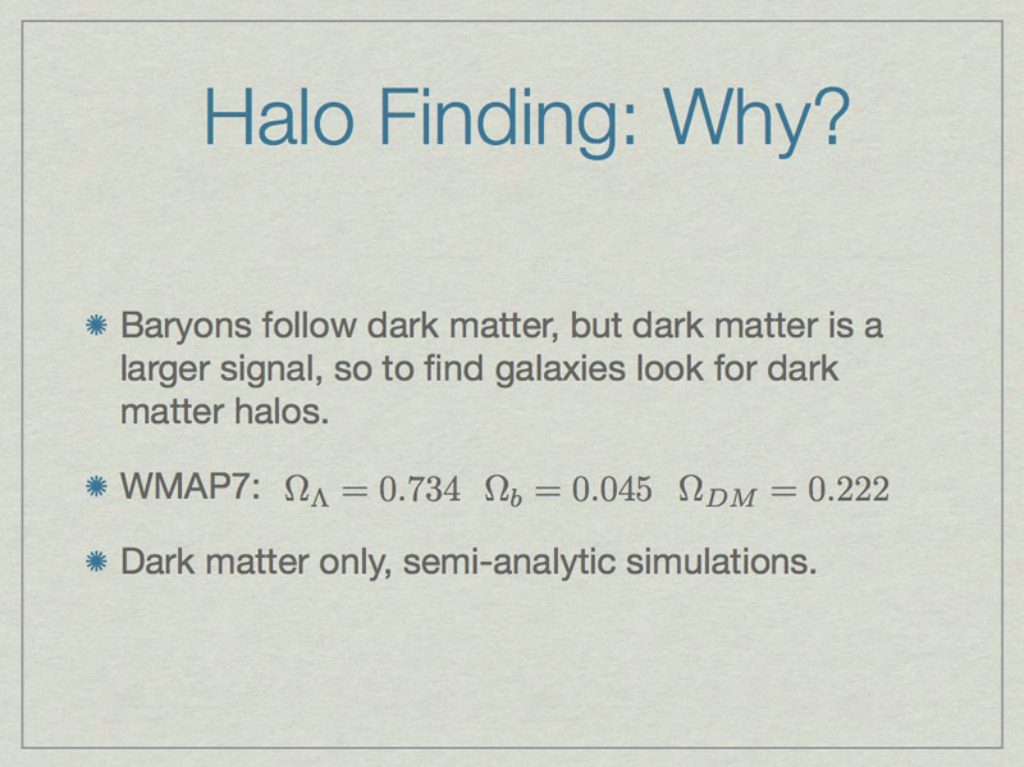
8.1.1 The Slides



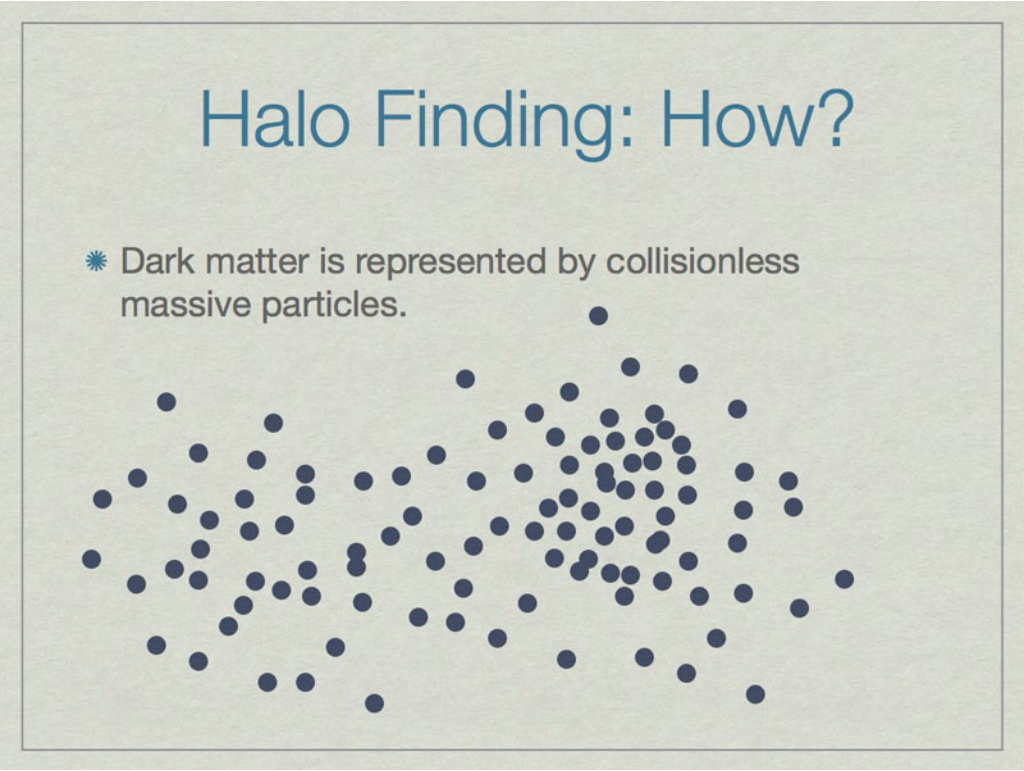


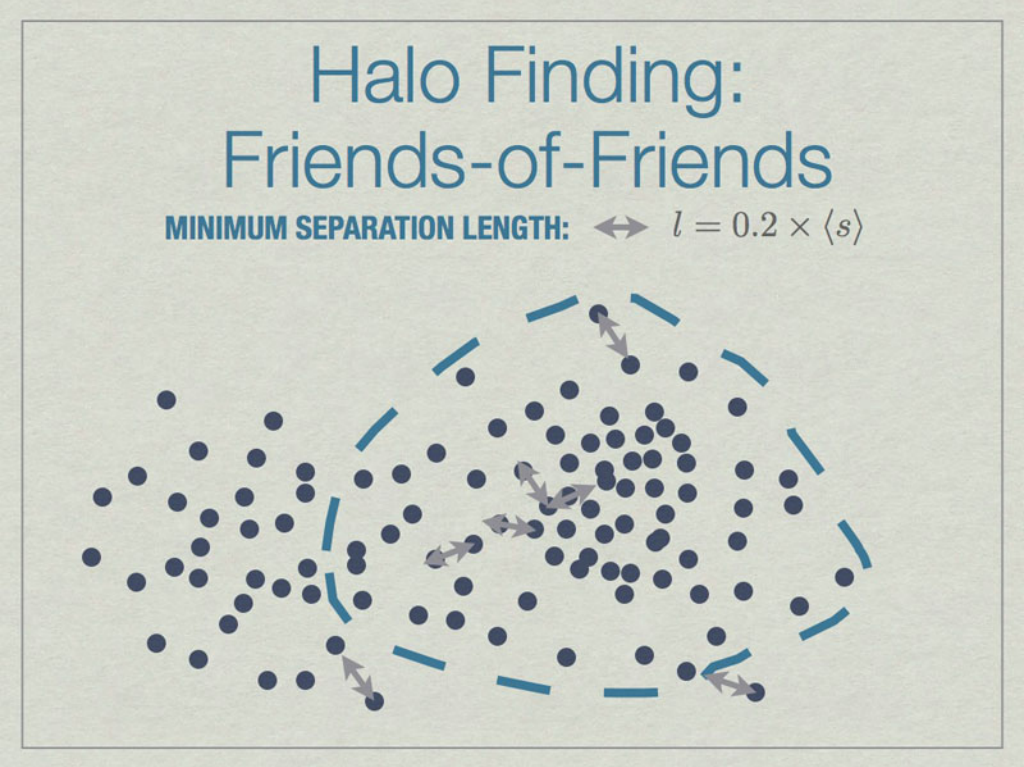


How do we know dark matter exists and surrounds galaxies? Here are some of the ways.

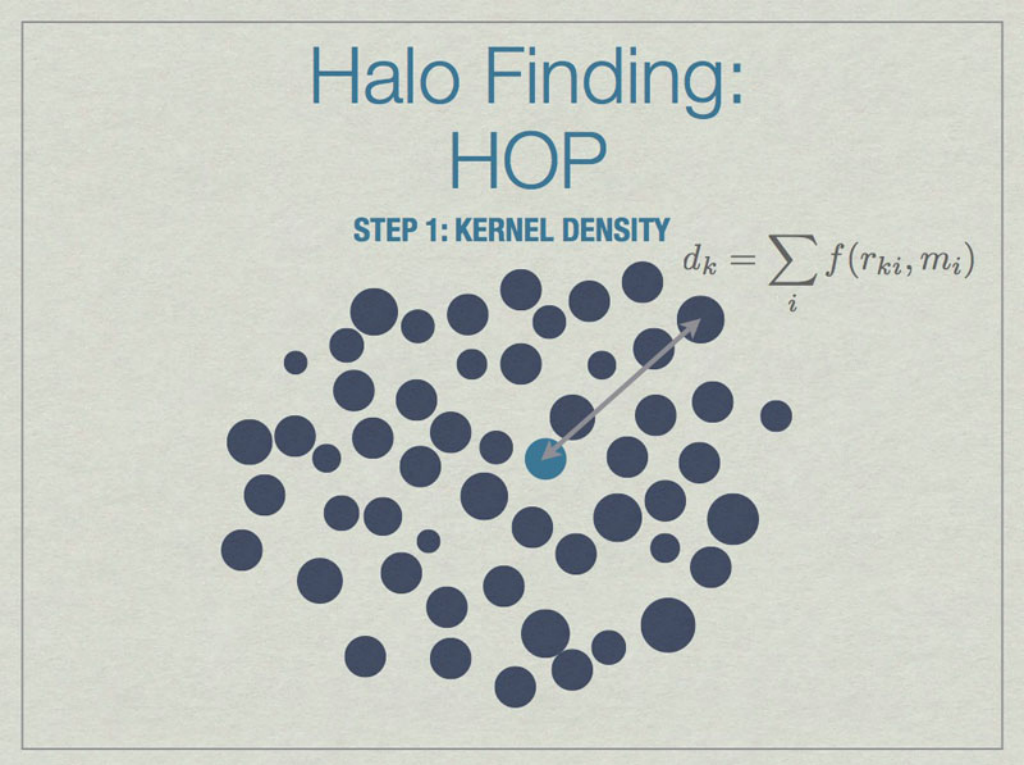


Observations look for things that glow, like stars, which live in galaxies. In simulations we want to find where the galaxies are, because that's where the interesting things are. It is better to look for dark matter rather than stars or gas because it is a stronger signal. Also, some simulations don't have stars or gas at all, like semi-analytic simulations.



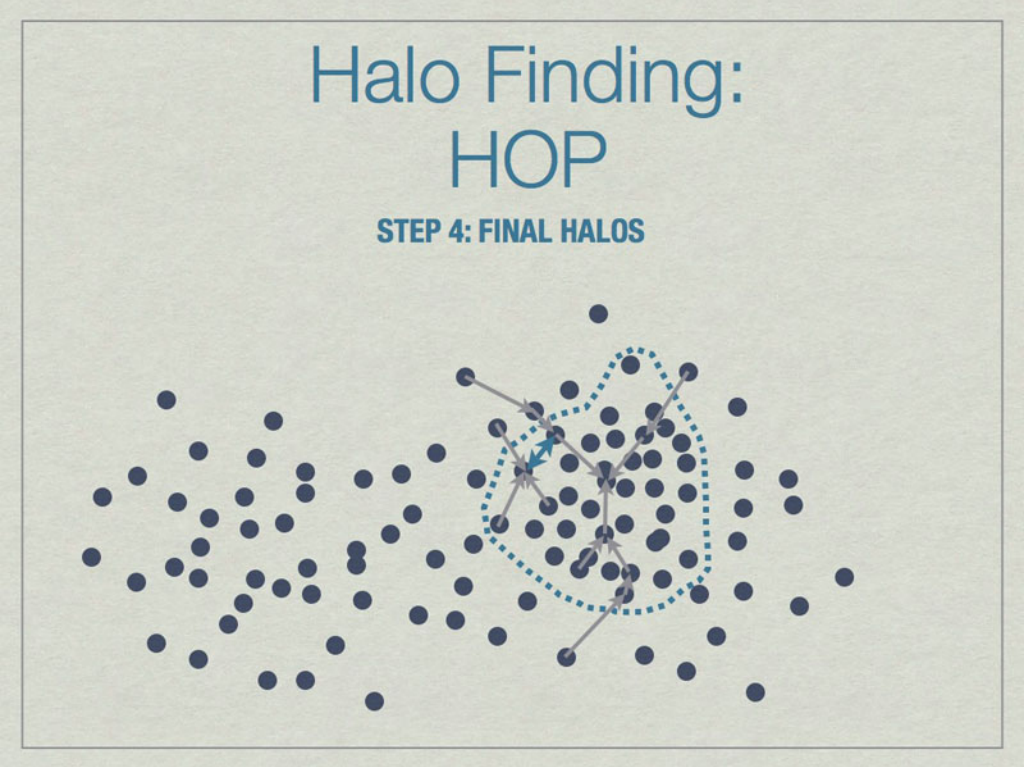


All particles closer than 0.2 of the mean inter-particle separation (s) are linked, and any all all links of particles are followed recursively to form the halo groups.



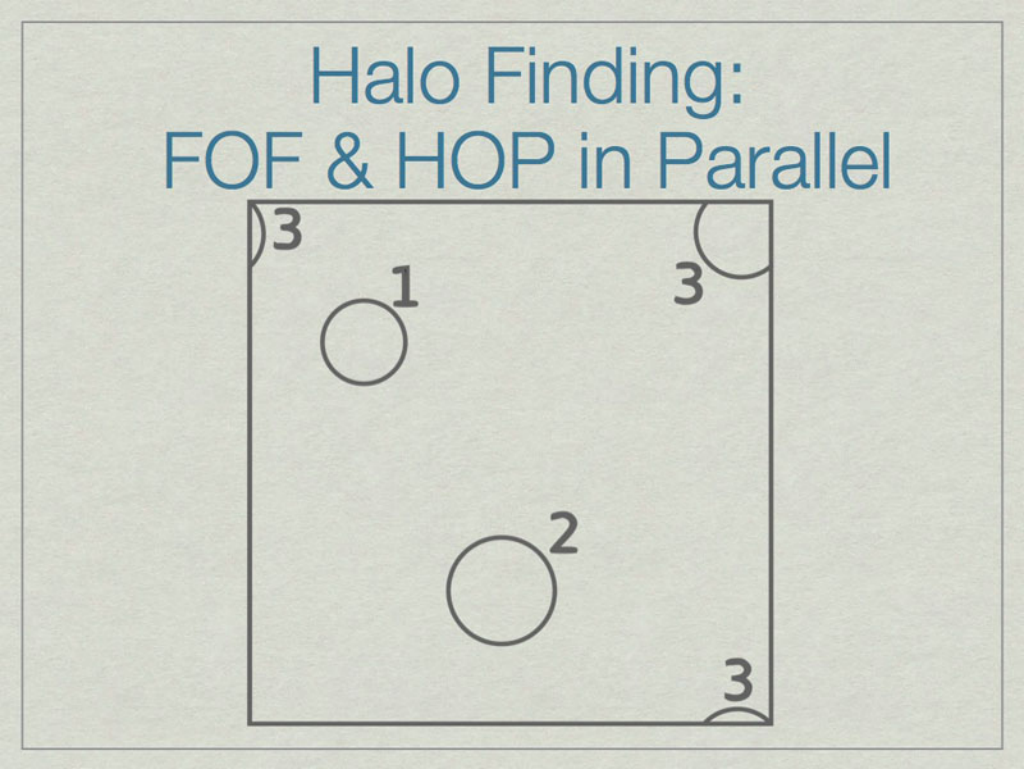
HOP starts by calculating a kernel density for each particle based on the mass of and distances to its nearest neighbors, the default is 64 of them.

Chains are built by linking particles uphill, from a particle with lower density to one that is higher, from the set of nearest neighbors. Particles that are their own densest nearest neighbors terminate the chains. Neighborinnearest neighbors, but in different chains.

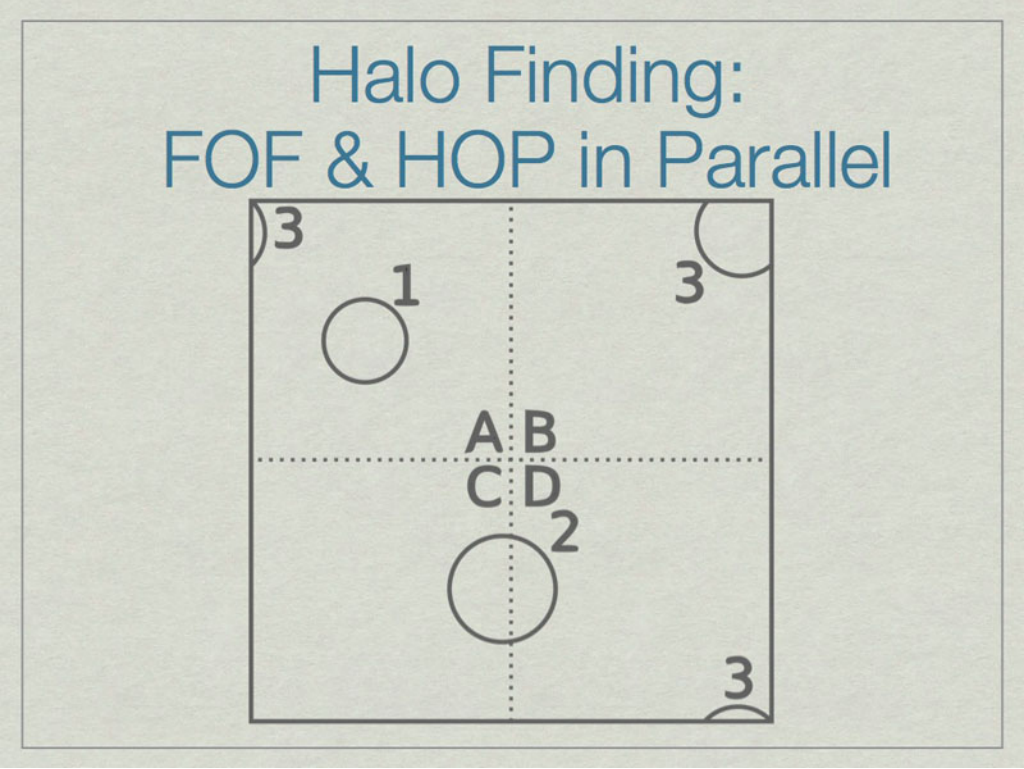


Neighboring chains are merged to build the final halos using various rules. The figure above shows the final halo

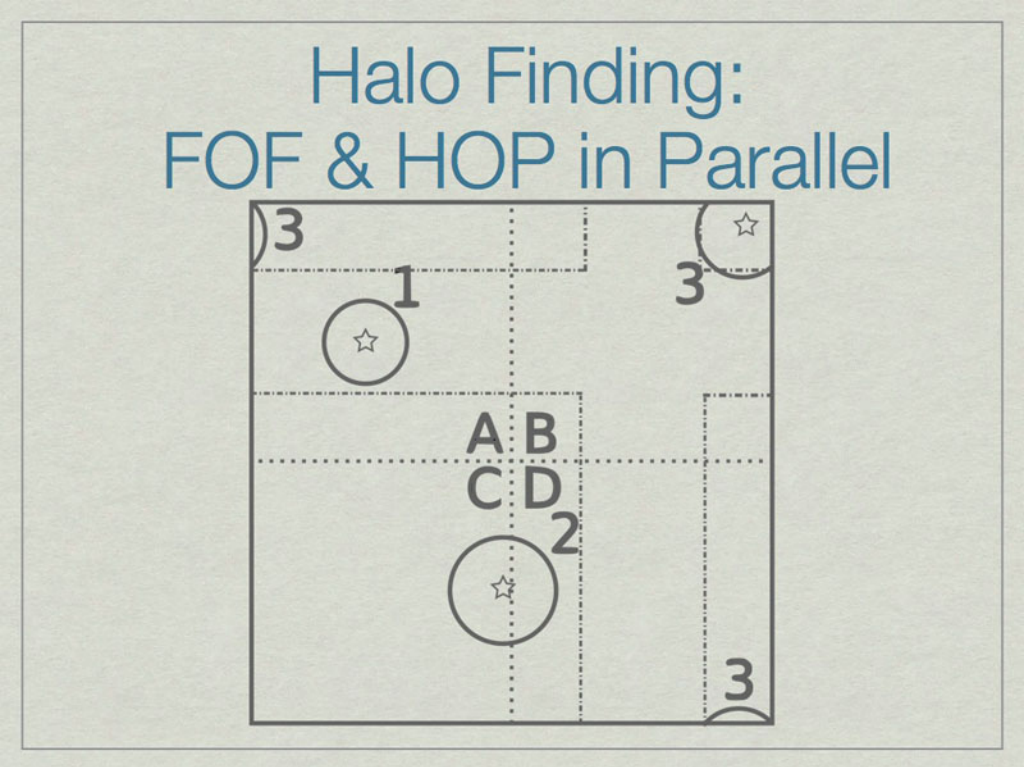
enclosed by a dashed line. A few particles have been excluded from the final halo because they are underdense.



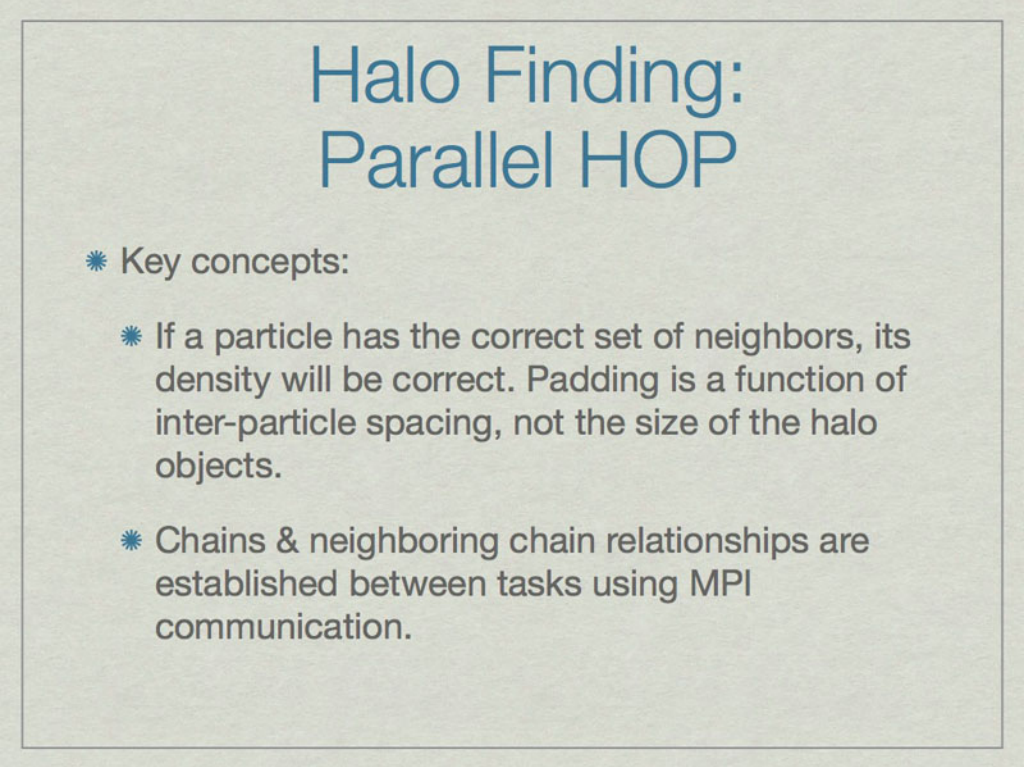
It is possible to run FOF & HOP in parallel. We start here with three halos in a volume, one of which (3) lies on the periodic boundary of the volume.



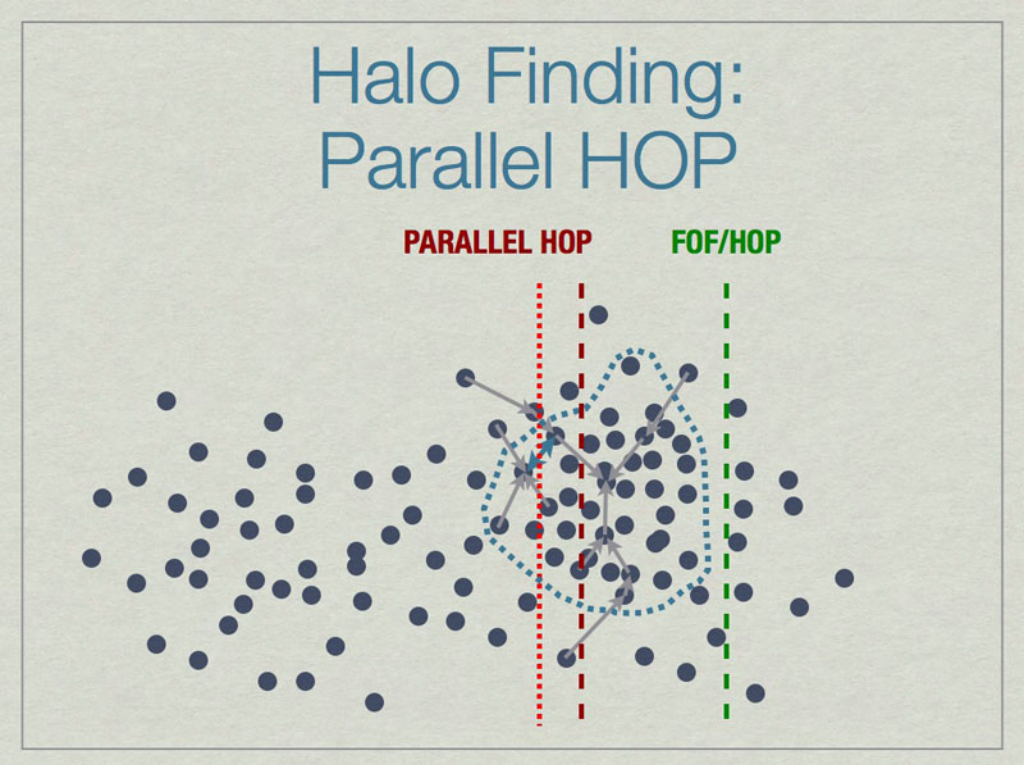
The dashed lines depict the subdivision of the full volume into subvolumes (A,B,C, and D) which define the sub-units for parallel analysis. Note that halos 2 & 3 lie in more than one subvolume.



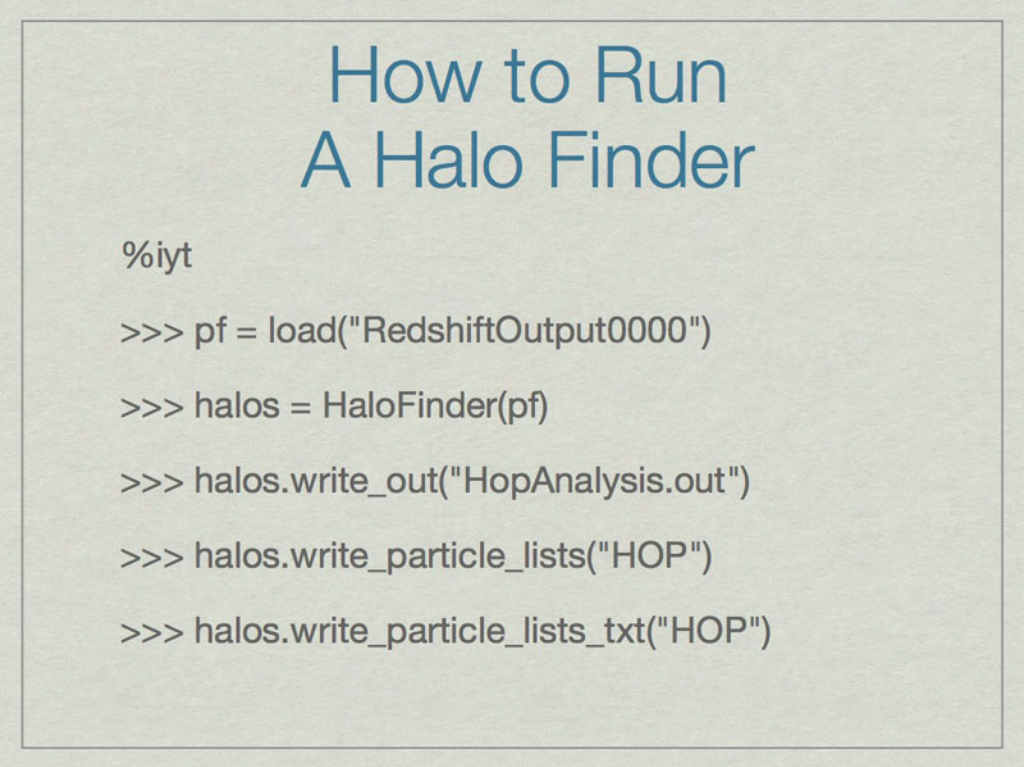
The solution is to add extra data on the faces of the subvolumes such that all halos are fully enclosed on at least one subvolume. Here subvolume C has been 'padded' which allows halo 2 to be fully contained in subvolume C. The centers of the halos, shown with stars, determine final ownership of halos so there is no duplication. However, this method breaks down when the halo sizes are a significant fraction of the full volume.



Parallel HOP is a fully-parallel implementation of HOP that allows both computation and memory load to be distributed using MPI parallelism.



Parallel HOP can reduce the padding by a substantial amount compared to FOF/HOP parallelism. This leads to many work- & memory-load advantages.

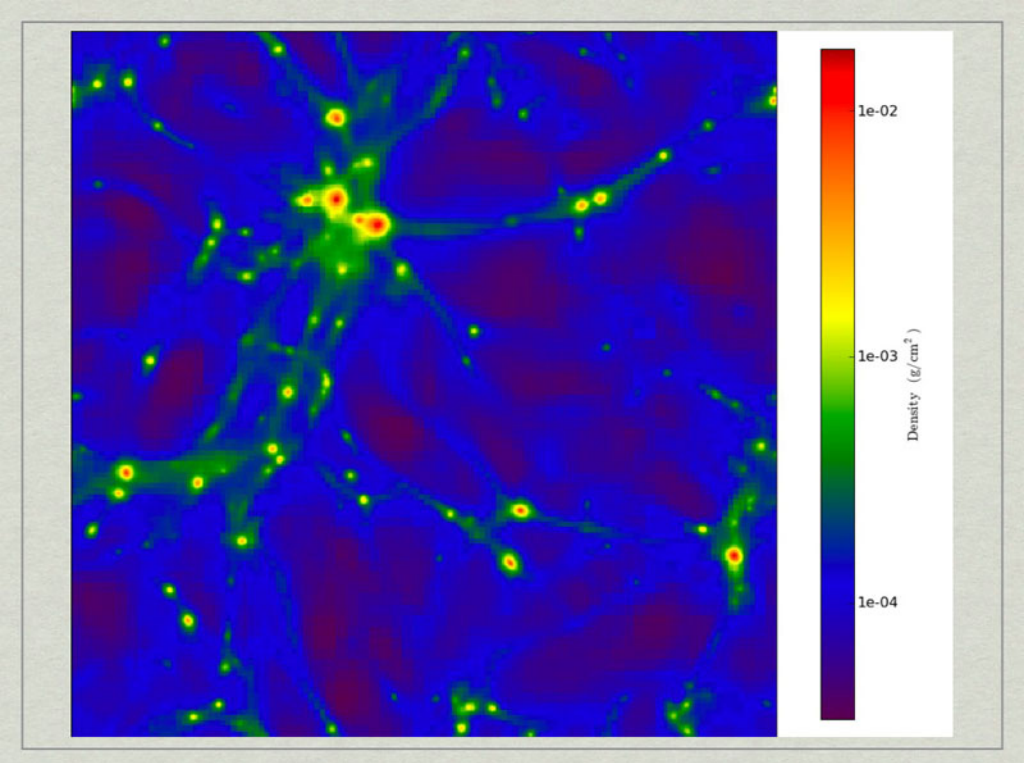


The first command builds a reference to an Enzo dataset. The second runs HOP on the particles in the dataset and stores the result in the `halos` object. The `write_out` command writes the halo particulars to a text file that contains the ID, mass, center of mass, maximum radius, bulk velocity and velocity dispersion for each halo. `write_particle_lists` and `write_particle_lists_txt` stores the information for the exact particles that are identified in each halo.

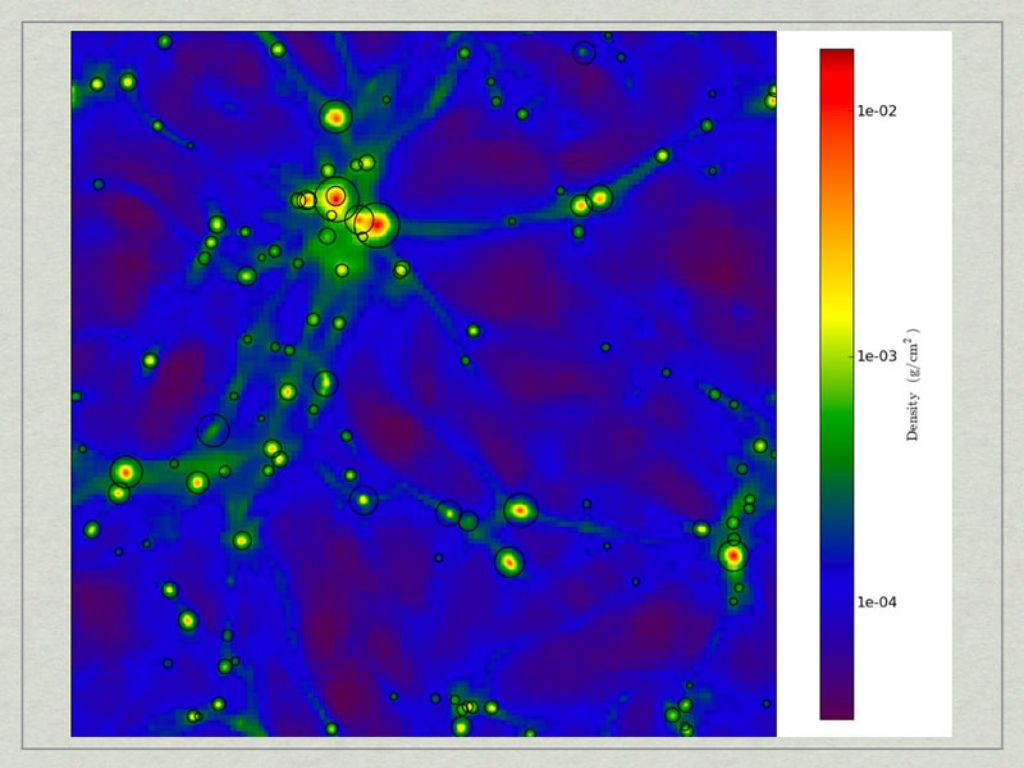
Visualize the Halos

```
>>> pc = PlotCollection(pf, center=[0.5, 0.5, 0.5])
>>> pc.add_projection("Density", 0)
>>> pc.save("test1")
>>> pc.plots[-1].modify["hop_circles"](halos)
>>> pc.save("test2")
```

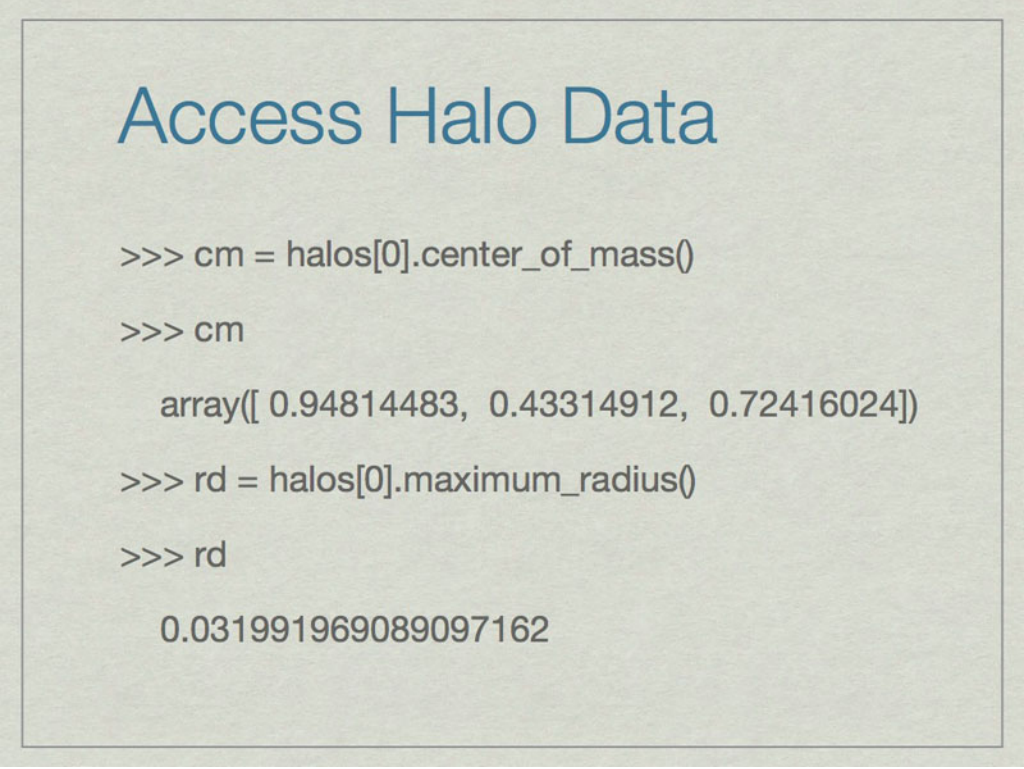
This shows how to find halos very simply and quickly using HOP in yt. First call 'iyt' from the command line. Next we reference the dataset, and then find the halos using HOP and the default settings. The next command writes out a text file with halo particulars, next the particle data for halos is written to a HDF5 file, and the last command saves a text file of where the particle halo data goes (important for parallel analysis).



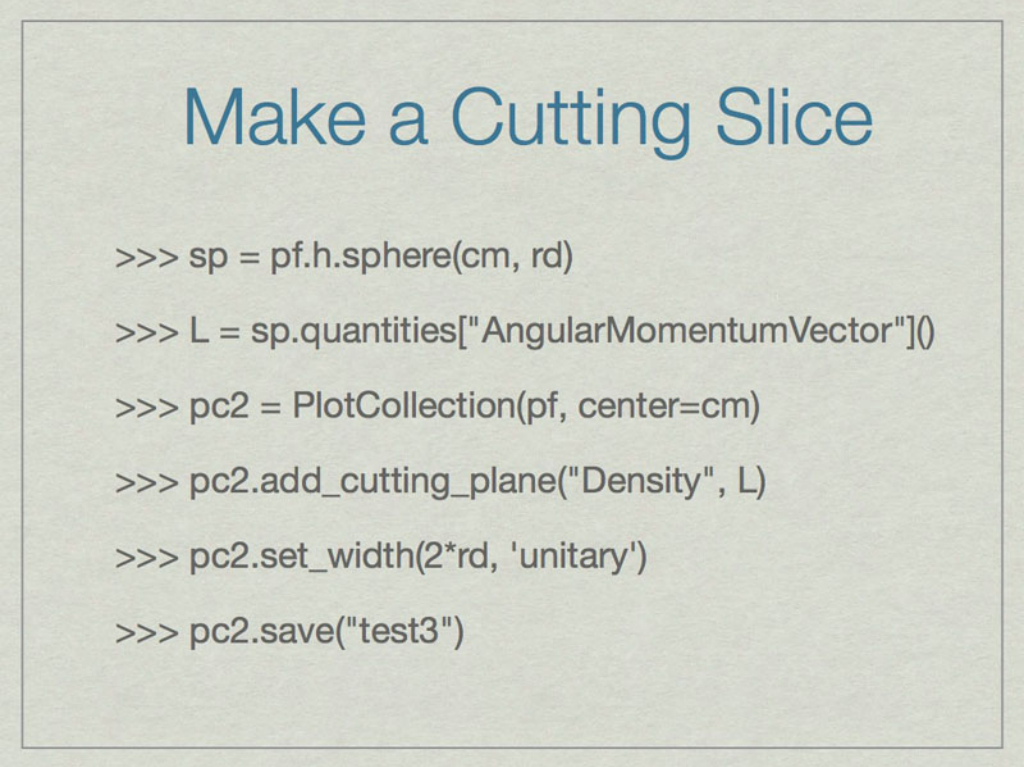
test1_Projection_x_Density.png. A density projection through a test dataset.



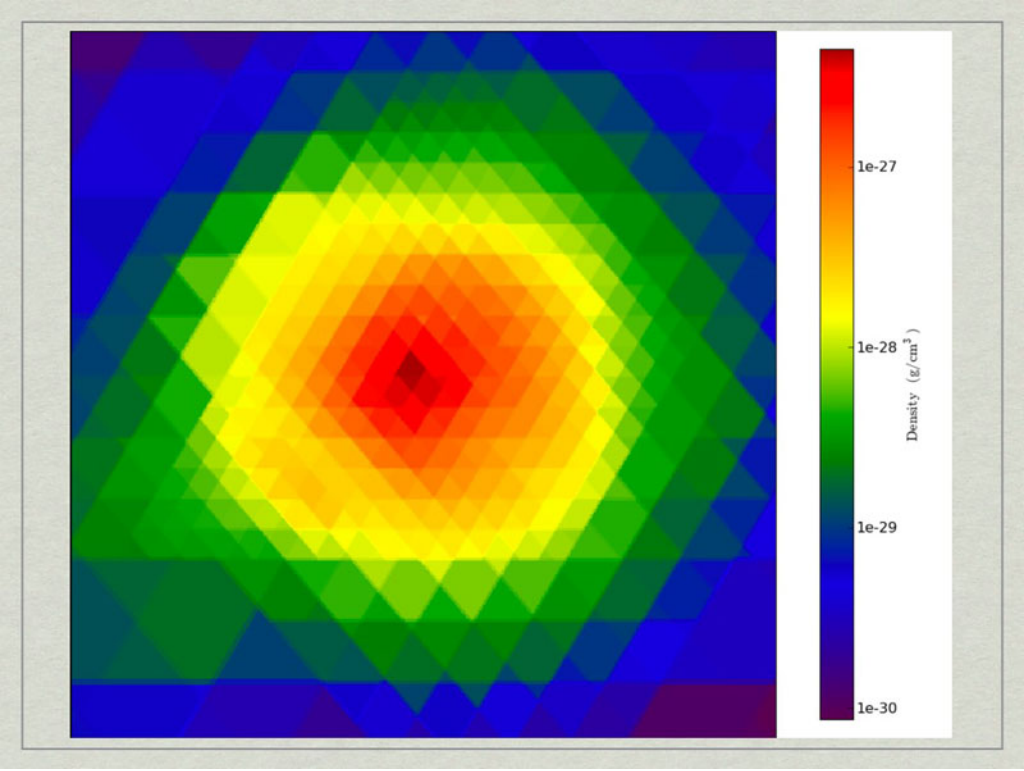
test2_Projection_x_Density.png. The halos have beecorresponds to the maximum radius of the halo.



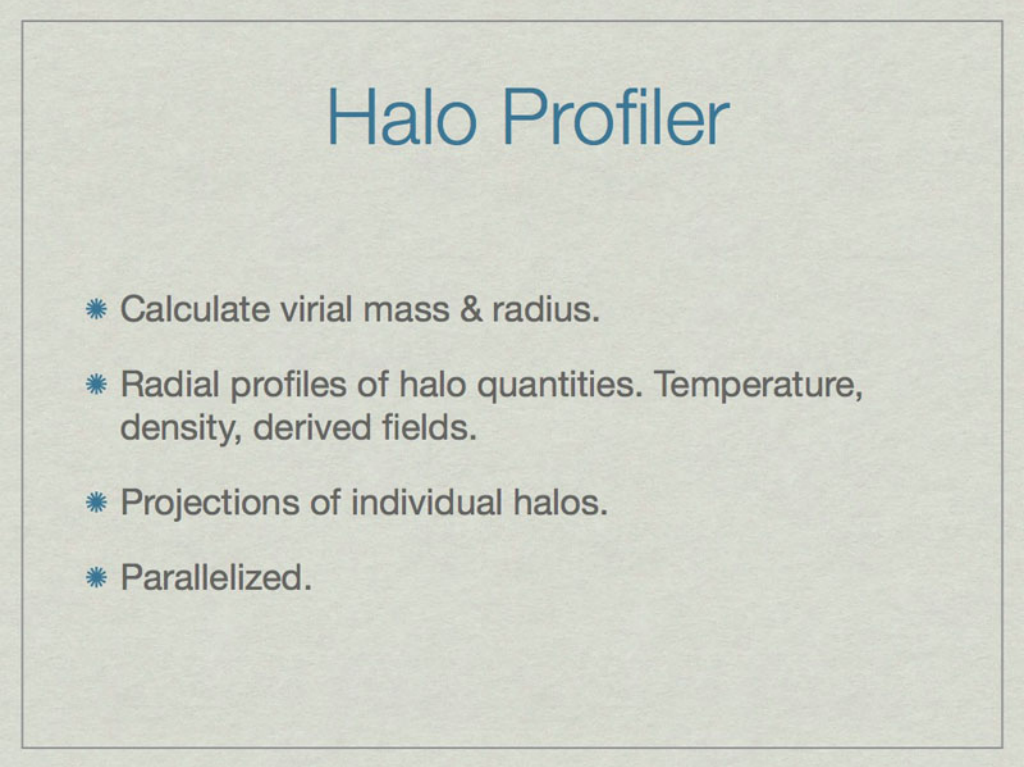
It is easy to access information about halos. All of these are in code units.



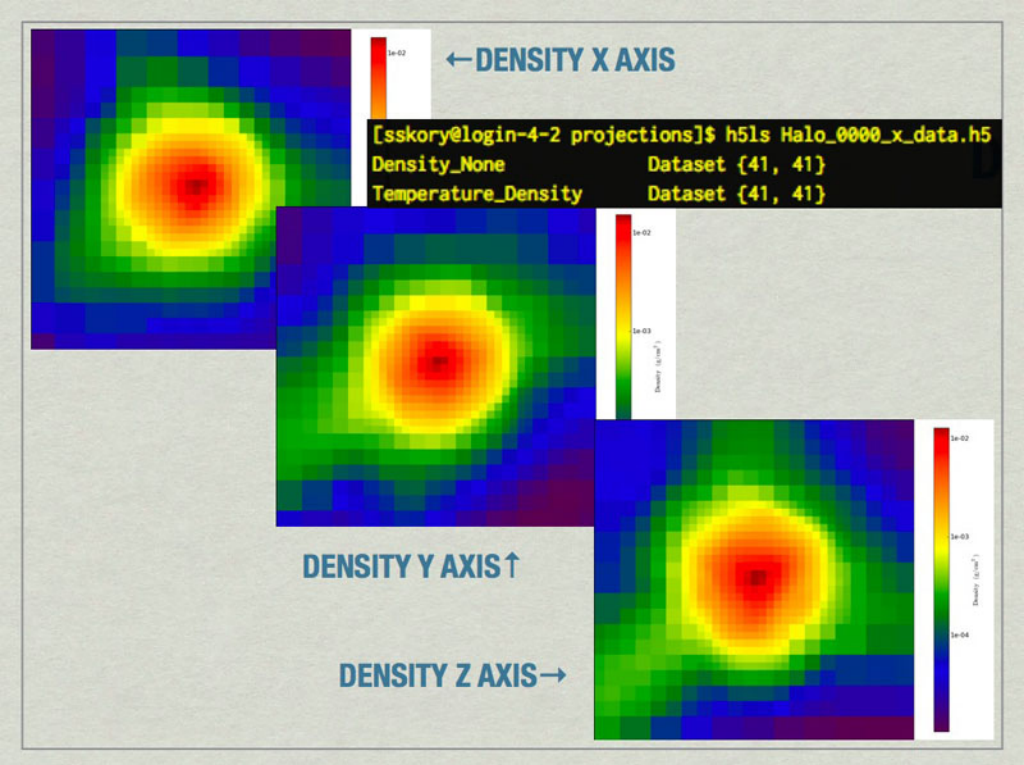
These commands will make a cutting slice through the center of the halo with normal vector oriented along the angular momentum vector of the halo.



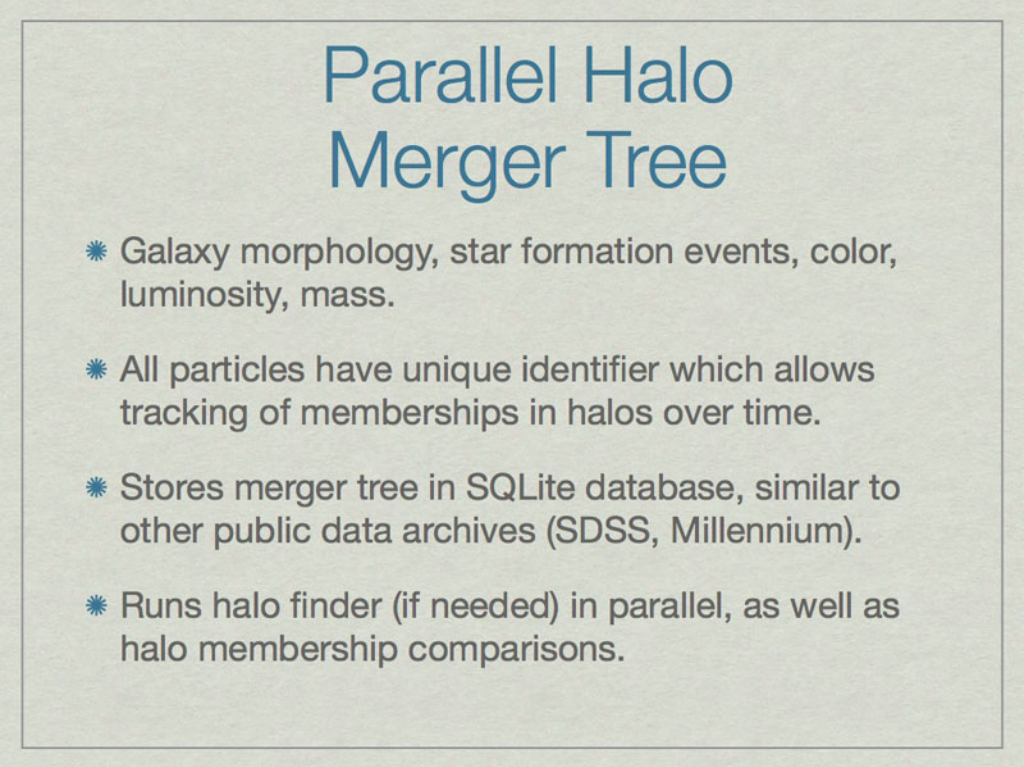
test3_CuttingPlane__Density.pngtest3_CuttingPlane__Density.png.



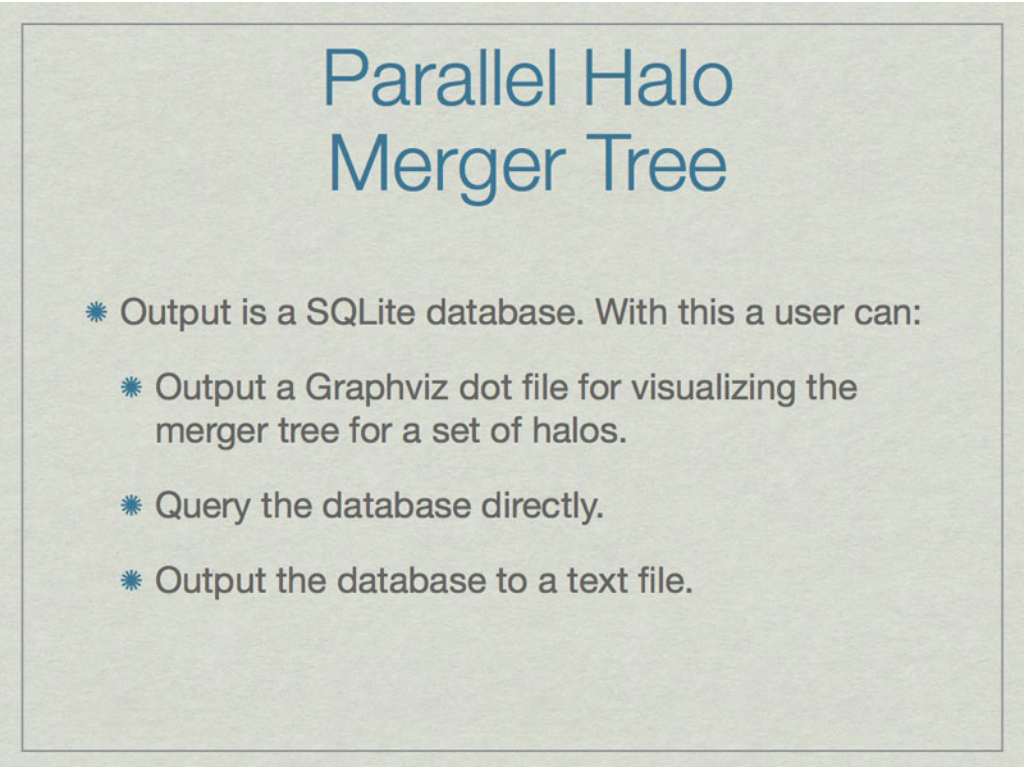
he halo profiler written by Britton Smith can analyze halos for various quantities. Given a HopAnalysis.out file, it can calculate many things on each halo.

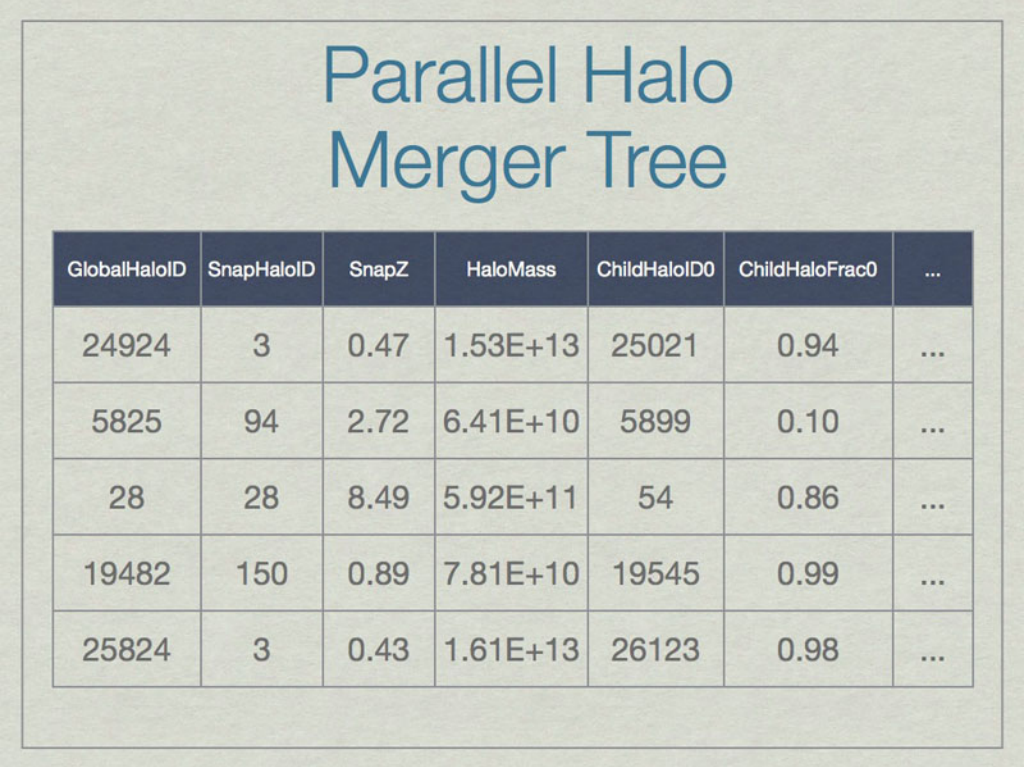


mages of the largest halo in the volume produced by the Halo Profiler. Also shown is the contents of the HDF5 files produced by the Halo Profiler.

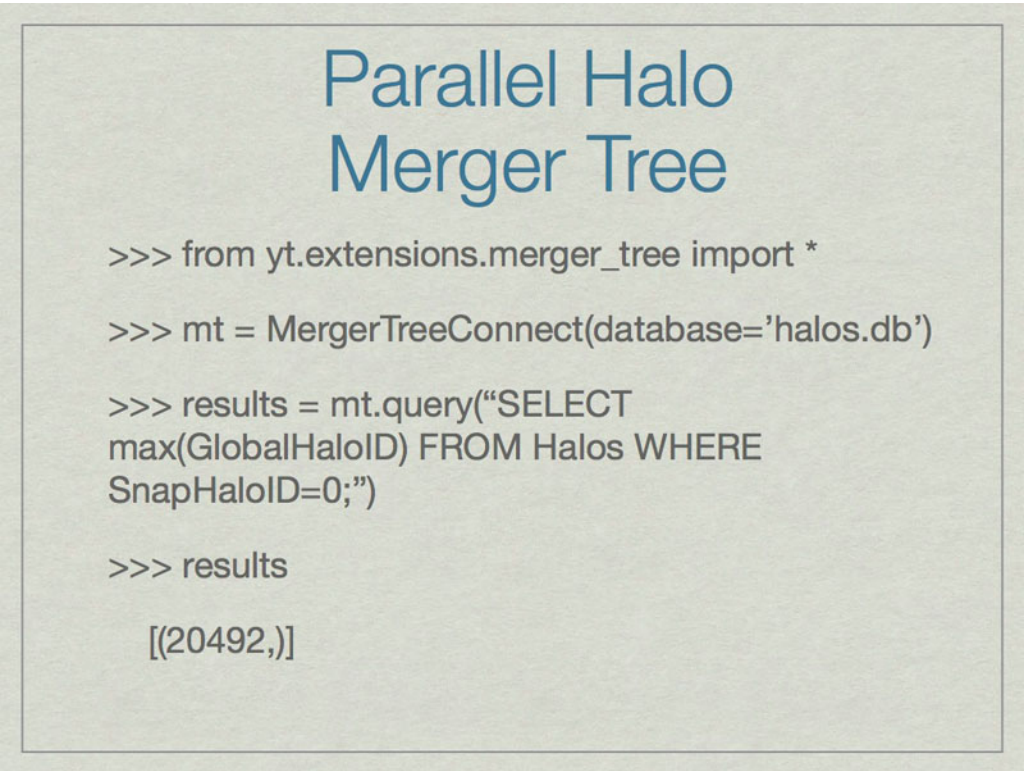


Merger trees are important when studying a halo because they affect many aspects of the halo. A merger tree tool analyzes a time-ordered series of datasets to build a comprehensive listing of the relationships between halos.

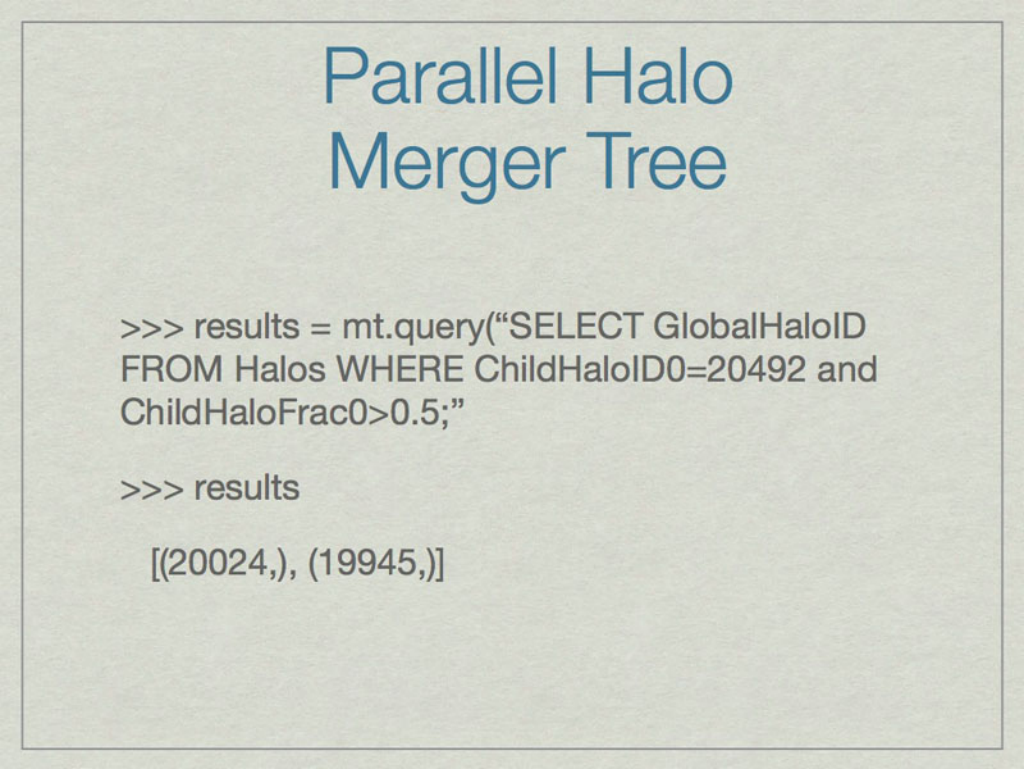




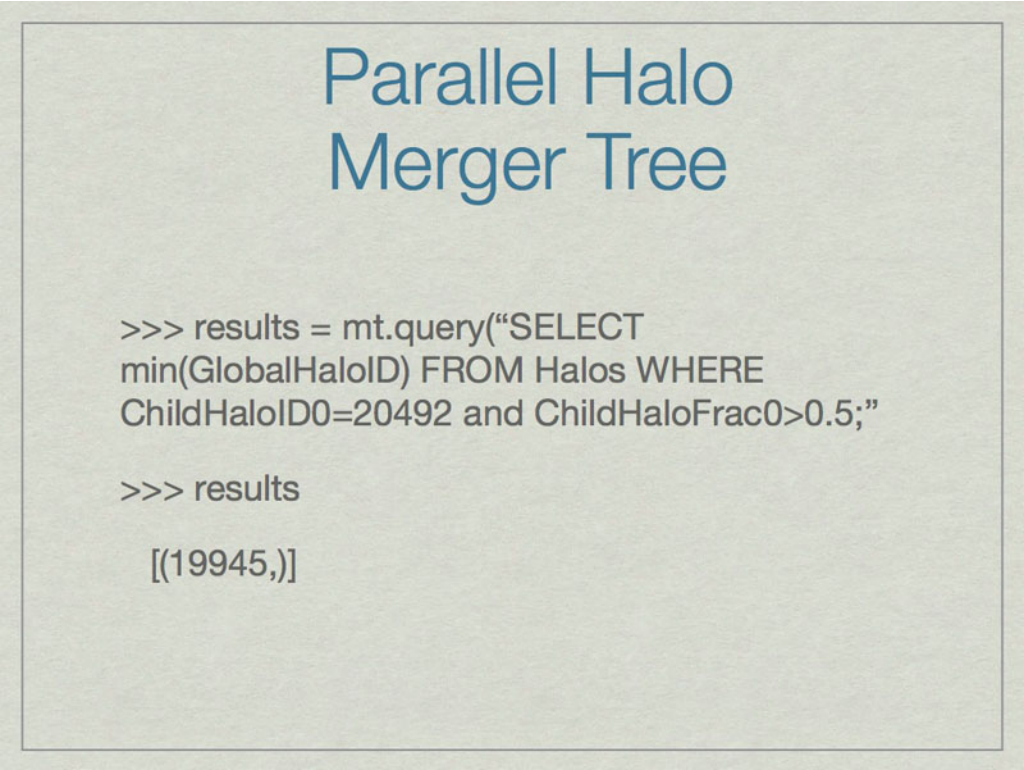
A SQL database can be thought of as a spreadsheet-like container, however entries are not ordered, unless the SQL query specifies that. This shows a few made-up example values in the database for a few real columns. Note that SnapHaloID is not unique. There are more columns in the database, but this is just an example. Columns not shown list the children for these halos.



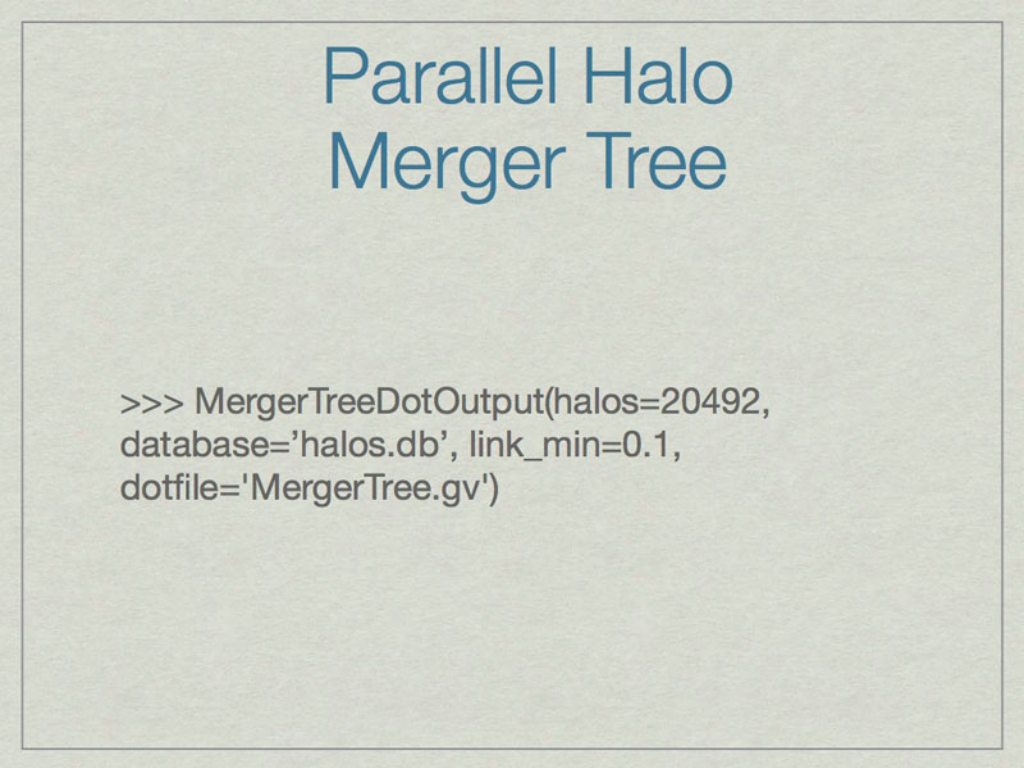
An example of how to find the GlobalHaloID for the most massive halo for the lowest redshift dataset.



Using the output of the previous slide, an example of how to find the parents that contribute the greatest fraction of their mass to the most massive halo at the lowest redshift.



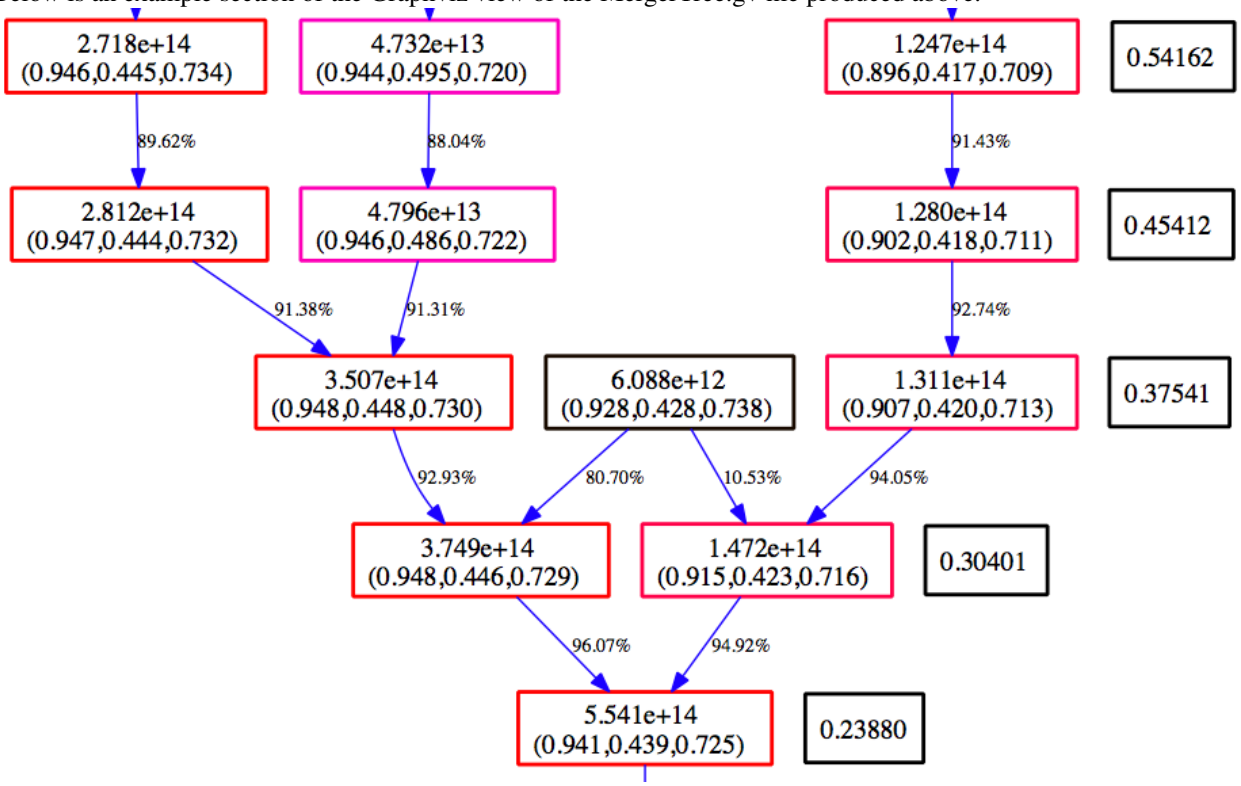
An example of how to find the most massive parent of the most massive halo at the lowest redshift.



An example of how to output the full merger tree for a given halo (20492) to a graphviz file (MergerTree.gv).

8.1.2 Merger Tree Graphviz Example

Below is an example section of the Graphviz view of the MergerTree.gv file produced above.



Time moves from the top to the bottom. The numbers in the black boxes give the redshift for each horizontal level of the merger tree. Each colored box corresponds to a halo that is in the merger tree for our final halo. The top number in each box gives the mass of the halo as determined by the halo finder. The second number is the center of mass for the halo in code units. The color of the box is scaled such that at each redshift, the most massive halo is red, and the smallest blue. The arrows connect a 'parent' halo to a 'child' halo, and the number next to each arrow gives the percentage of the mass of the parent halo that goes to the child halo.

CHAPTER

NINE

ENZO MAILING LISTS

There are two mailing lists for Enzo hosted on Google Groups, enzo-users and enzo-dev.

9.1 enzo-users

Everyone Enzo user should sign up for the enzo-users mailing list. This is is used to announce changes to Enzo, and sometimes major changes Enzo-related analysis tools. This list is appropriate for anything else Enzo-related, such as machine-specific compile problems, discussions of the science and physics behind what Enzo does, or queries about problem initialization. We recommend using the Enzo users mailing list liberally - by this we mean that any question asked on the list will educate everyone else on the list, and is manifestly not a stupid question. As long as a good effort has been made to try to figure out the answer before mailing the list, all questions about Enzo are welcome! Please follow the link below to sign up for this list and a link to discussion archives:

http://groups.google.com/group/enzo-users

To post a message to this list, send an email to:

enzo-users@googlegroups.com

The archives for the old Enzo users mailing list can be found linked below. A search of the list archives should be performed before emailing the list to prevent asking a question that has already been answered (using, for example, an advanced web search limited to that page).

https://mailman.ucsd.edu/pipermail/enzo-users-l/

9.2 enzo-dev

The second mailing is for developers of Enzo. This is for Enzo "old-hats", or anyone interested in adding new features to Enzo, or anyone who wants a deeper understanding of the internals of Enzo. Please follow the link below to sign up for the list and a link to the discussion archives:

http://groups.google.com/group/enzo-dev

To post a message to this list, send an email to:

enzo-dev@googlegroups.com

CHAPTER

TEN

REGRESSION TESTS

The Enzo trunk and select branches are checked out of Subversion and tested continuously using LCATest on ppcluster.ucsd.edu:

- Continuous Regression Test Results

For questions or suggestions related to the Enzo regression testing or lcatest, please contact James Bordner at jobordner at ucsd.edu.

CHAPTER

ELEVEN

CITING ENZO

If you use Enzo for a scientific publication, we ask that you cite the code in the following way in the acknowledgments of your paper:

Computations described in this work were performed using the Enzo code (http://enzo.googlecode.com), which is the product of a collaborative effort of scientists at many universities and national laboratories.

CHAPTER

TWELVE

SEARCH

- *search*