Enzo-E / Cello Documentation

Enzo Development Community

Mar 21, 2024

CONTENTS

1 Tutorial 3

2 User Guide 17

3 Reference Guide 57

4 Developer Guide 173

5 Design documents 245

6 Project documents 269

7 Testing 279

8 Getting started 315

9 Parameters 317

10 Using and developing Enzo-E / Cello 319

11 PDF User and Developer Guide (depreciated) 321

12 Indices and tables 323

Index 325

Cello is a highly scalable, fully-distributed array-of-octree parallel adaptive mesh refinement (AMR) framework, and **Enzo-E** is a scalable branch of the ENZO parallel astrophysics and cosmology application that has been ported to use Cello. Enzo-E / Cello is currently funded by the National Science Foundation (NSF) grant OAC-1835402, with previous funding through NSF grants PHY-1104819, AST-0808184, and SI2-SSE-1440709.

Two fundamental differences between Enzo-E and ENZO are their AMR design and code parallelization. Cello implements *array of octree* AMR, which has demonstrated scalability to date through 256K floating-point cores of the NSF Blue Waters supercomputer at the National Center for Supercomputing Applications. Unlike ENZO, which is parallelized using MPI, Enzo-E/Cello is parallelized using Charm++, an OOP parallel programming system, targeting the development of Exascale software applications, and actively developed at the Parallel Programming Laboratory at the University of Illinois, Urbana-Champaign.

Enzo-E currently has three hyperbolic solvers: PPM, an enhanced piecewise parabolic method solver that was migrated to Enzo-E from the ENZO code base, PPML, an ideal compressible MHD solver originally implemented in serial Fortran, and VL+CT, a dimensionally-unsplit compressible MHD solver implemented specifically for Enzo-E in C++. More recently, physics and infrastructure capabilities have been developed for particle methods, including an implementation of ENZO's CIC particle-mesh gravity solver, and cosmological expansion with comoving coordinates. Currently we are collaborating with Prof. Daniel Reynolds on developing and implementing a highly scalable multigrid-based linear solver.

CHAPTER

ONE

TUTORIAL

1.1 Getting started using Enzo-E

This page will help you get Enzo-E and Cello up and running. It covers downloading the source code, porting the code to new platforms, configuring and compiling the code, and running a sample test problem.

Other pages are available for helping to get started on specific architectures, including the "Frontera" supercomputer at TACC and the "Pleiades" supercomputer at NASA.

1.1.1 Getting Started: Pleiades

```
# Load more up-to-date software stack
module use -a /nasa/modulefiles/testing
module purge
module load pkgsrc/2021Q1 gcc/10.2 mpi-hpe/mpt.2.23 boost/1.76 comp-intel/2020.4.304␣
↪hdf5/1.12.0_serial pkgsrc/2021Q1
# Build Grackle (optional)
# Following https://grackle.readthedocs.io/en/latest/Installation.html
mkdir -p ~/src
cd ~/src
git clone https://github.com/grackle-project/grackle
cd grackle
git submodule update --init
./configure
# create build directory as install target later
mkdir build-icc
cd src/clib
# Adjust config to loaded modules, set install path, and optimization options
sed -i 's/1.8.18/1.12.0/;s/2020.2.254/2020.4.304/;s#$(HOME)/local#$(PWD)/../../build-icc
↪#;s/-axAVX -xSSE4.1 -ip -ipo/-axCORE-AVX512,CORE-AVX2/' Make.mach.nasa-pleiades
make machine-nasa-pleiades
make
make install

# Build Charm++
cd ~/src
git clone https://github.com/UIUC-PPL/charm.git
cd charm
git checkout v7.0.0
mkdir build-icc-mpi
```

(continues on next page)

(continued from previous page)

```
cd build-icc-mpi
cmake -DNETWORK=mpi -DSMP=OFF -DCMAKE_CXX_COMPILER=icpc -DCMAKE_C_COMPILER=icc -DCMAKE_
↪Fortran_COMPILER=ifort ..
make

# Build Enzo-E
cd ~/src
git clone https://github.com/enzo-project/enzo-e.git
cd enzo-e
# Custom environment override for the cmake call specific to Pleiades system as
# during the linking step (done with the `charmc` wrapper) the `mpicxx` wrapper is called,
# which, in turn, by default calls `g++` (but we need to link with `icpc` at the lowest␣
↪level).
export MPICXX_CXX=icpc
mkdir build-icc-mpi
cd build-icc-mpi
cmake -DCHARM_ROOT=${HOME}/src/charm/build-icc-mpi -DGrackle_ROOT=${HOME}/src/grackle/
↪build-icc -DEnzo-E_CONFIG=pleiades_icc ..
make
# To run Enzo-E simply call `mpiexec ./build-icc-mpi/bin/enzo-e input/HelloWorld/Hi.in`␣
↪as usual
```

1.1.2 Getting Started: Frontera

```
# Assuming the default Intel stack is loaded only boost and hdf5 modules are required
module load boost hdf5

# Build Grackle (optional)
# Following https://grackle.readthedocs.io/en/latest/Installation.html
mkdir -p ~/src
cd ~/src
git clone https://github.com/grackle-project/grackle
cd grackle
git submodule update --init
./configure
# create build directory as install target later
mkdir build-icc
cd src/clib
# Adjust config to  set install path, and optimization options
# TACC Stampede config works just fine for TACC Frontera
sed -i 's#$(HOME)/local#$(PWD)/../../build-icc#;s/-xCORE-AVX2/-xCORE-AVX512/' Make.mach.
↪tacc-stampede-intel
make machine-tacc-stampede-intel
make
make install

# Build Charm++
cd ~/src
git clone https://github.com/UIUC-PPL/charm.git
cd charm
```

(continues on next page)

(continued from previous page)

```
git checkout v7.0.0
mkdir build-icc-mpi
cd build-icc-mpi
cmake -DNETWORK=mpi -DSMP=OFF -DCMAKE_CXX_COMPILER=icpc -DCMAKE_C_COMPILER=icc -DCMAKE_
↪Fortran_COMPILER=ifort ..
make

# Build Enzo-E
cd ~/src
git clone https://github.com/enzo-project/enzo-e.git
cd enzo-e
mkdir build-icc-mpi
cd build-icc-mpi
cmake -DCHARM_ROOT=${HOME}/src/charm/build-icc-mpi -DGrackle_ROOT=${HOME}/src/grackle/
↪build-icc -DEnzo-E_CONFIG=frontera_icc ..
make
# To run Enzo-E simply call `ibrun ./build-icc-mpi/bin/enzo-e input/HelloWorld/Hi.in␣
↪+balancer GreedyLB` as usual
```

1.1.3 Dependency Installation

Before compiling `Enzo-E / Cello`, you may need to download and install 1. `CMake`, 2. `Charm++`, 3. `HDF5`, 4. `libpng`, 5. `libboost`, and (optionally) 6. `Grackle`:

1. Install `CMake`

Most systems nowaways have `CMake` already installed. If not, you can get the binary distribution from the CMake website.

2. Install `Charm++`

We generally recommend that you download *Charm++*, from the GitHub repository and then retrieve the latest version (7.0.0) known to build *Enzo-E/Cello*. This is demonstrated by the following command:

```
git clone https://github.com/UIUC-PPL/charm.git
cd charm
git checkout v7.0.0
```

Alternatively, if you don't have *git* installed, you can also download the appropriate release version of the code from the main website or from the Release page on GitHub.

***Charm++* can be configured to use different "backends" for handling communication, which include:**

- `mpi`: our recommended default choice for distributed machines. Under this configuration, communication occurs via calls to MPI commands (this obviously requires an MPI installation).
- `netlrts`: our recommended choice for development/testing on a local machine (this can make debugging far simpler). Under this configuration, communication occurs via standard networking protocols (that all modern computers are equipped with). This is generally slower than other options.

Charm++ uses *cmake* as the default build system. To build *Charm++* start from directory holding all downloaded files.

Invoke the following command to build with the *mpi* backend:

```
mkdir build-mpi # this directory name is arbitrary
cd build-mpi
cmake -DNETWORK=mpi -DSMP=OFF ..
make # you can specify make -j4 to compile with 4 threads
```

The following commands to build with the *netlrts* backend:

```
mkdir build-netlrts # this directory name is arbitrary
cd build-netlrts
cmake -DNETWORK=netlrts -DSMP=OFF ..
make # you can specify make -j4 to compile with 4 threads
```

As an aside *Charm++* can also be configured to directly use a machine's low-level communication primitives (that `MPI` implementations wrap). It can be complicated to compile with these backends, so additional details are provided on an architechture-specific basis.

3. Install `HDF5`

"HDF5 is a "data model, library, and file format for storing and managing data", and is the primary library used by Enzo-E / Cello for data output.

If HDF5 is not already installed on your machine, it may be available through your operating system distribution, otherwise it can be downloaded from the HDF5 website. Enzo-E / Cello currently uses the "serial" (non-MPI) version of HDF5.

4. Install `libpng`

"libpng is the official PNG reference library", and is the image format used by Enzo-E / Cello.

If `libpng` is not already installed on your machine, it may be available through your operating system distribution, otherwise it can be downloaded from the libpng website.

5. Install `libboost-dev`

"Boost provides free peer-reviewed portable C++ source libraries."

If `libboost-dev` is not already installed on your machine, it may be available through your operating system distribution, otherwise it can be downloaded from the libboost website.

6. Install Grackle (Optional)

By default, Enzo-E requires the Grackle chemistry and cooling library. If you do not need to use Grackle, you can simple disabling it by setting `-DUSE_GRACKLE=OFF` when you configure Enzo-E. See the Grackle documentation for installation instructions.

7. Install yt (Optional)

If you want to use the yt python package to analyse Enzo-E output data, you should install the latest version from source. This can be done with the following commands:

```
git clone https://github.com/yt-project/yt.git
cd yt
pip install -e .
```

This is **NOT** a requirement for building and running Enzo-E, but it is used in some tests.

1.1.4 Configuring/Building

`Enzo-E / Cello` is currently hosted on github.com (previously bitbucket.com). To obtain the latest version of the source code, you may clone it from the repository Enzo-E / Cello github repository:

```
git clone https://github.com/enzo-project/enzo-e.git
```

For a basic configuration on a linux system with the GNU compiler stack, the following command should work out of the box. If not, please report any problems. See the following subsection for more configuration options.

```
mkdir my-first-build # Again, the directory name can be anything
cd my-first-build
# replace <PATH/TO/charm/build-dir> with the path to the directory created
# when you built charm++ (either build-mpi or build-netlrts if you've been
# following along)
cmake -DCHARM_ROOT=<PATH/TO/charm/build-dir> \
      -DEnzo-E_CONFIG=linux_gcc -DUSE_GRACKLE=OFF ..
make -j4 # -j4 tells make to execute up to 4 commands in parallel
```

To build on a Mac, you should only need to replace `linux_gcc` with `darwin_clang`.

Note, if `ninja` is installed, the `ninja` build system can be used for faster build times. This is done by adding `-GNinja` to the `cmake` command (before the `..`) and calling `ninja` afterwards instead of `make`.

The Enzo-E executable is built within `bin/`.

Configuration options

Current `cmake` options are listed in the following subsubsections. Skip ahead to *Specifying Configuration Options* for details about how to specify the configuration.

General Configuration

These options are related to Enzo-E's general configuration.

Table 1: General Configuration

Name	Description	Default
`USE_DOUBLE_PREC`	Use double precision. Turn off for single precision.	ON
`new_output`	Temporary setting for using new Output implementation	OFF
`node_size`	Maximum number of procesess per shared-memory node (can be larger than needed)	64
`smp`	Use Charm++ in SMP mode (Charm++ must have been compiled to support SMP mode).	OFF
`balance`	Enable charm++ dynamic load balancing	ON
`balancer_included`	Charm++ load balancer included	"CommonLBs"
`balancer_default`	Charm++ load balancer to use by default	"TreeLB"

Configuring Dependencies

The parameters in the following table control whether `Enzo-E / Cello` should make use of certain external dependencies. These options all assume that `cmake` is succesfully able to find the location of these dependencies. In some cases, you may need to provide additional hints about the location of the dependencies (see *Specifying Configuration Options* for more details)

Table 2: External Dependencies

Name	Description	Default
`USE_GRACKLE`	Use Grackle Chemistry	ON
`use_jemalloc`	Use the jemalloc library for memory allocation	OFF
`use_papi`	Use the PAPI performance API	OFF

Floating-Point Optimization Options

Generally, `Enzo-E / Cello` will not be compiled with value-unsafe floating point optimizations unless they are explicitly enabled. This default behavior was chosen to support automated tests that check symmetry preservation in certain components of the code (mostly related to hydro/MHD). While this feature is not essential for all applications, it is very useful when debugging new features.

Note that the Intel compilers deviate from this behavior. By default, they enable more aggressive floating point optimizations by default (consequently the tests checking symmetry may fail). This is a large part of the reason that the Intel compilers have a reputation for producing faster code (in practice, this is not always the case).

Table 3: Floating point optimizations

Name	Description	Default
`OPTIMIZE_FP`	Enable value-unsafe floating point optimizations (for some compilers, this enables `-ffast-math`).	OFF
`USE_SIMD`	Enables compiler support for OpenMP SIMD directives (for `gcc`, `icc`, and `clang` compilers this will NOT enable other OpenMP directives and should not link the openmp runtime library). Enabling this requires that `OPTIMIZE_FP=ON`.	OFF

Profiling Options

These options control compilation choices that can be used to facillitate profiling in `Enzo-E / Cello`

Table 4: Profile-Related Configuration

Name	Description	Default
`memory`	Track dynamic memory statistics. Can be useful, but can cause problems on some systems that also override `new [] ()` / `delete [] ()`	OFF
`use_gprof`	Compile with -pg to use gprof for performance profiling	OFF
`use_performance`	Use Cello Performance class for collecting performance data (currently requires global reductions, and may not be fully functional) (basic time data on root processor is still output)	ON
`use_projections`	Compile the CHARM++ version for use with the Projections performance tool.	OFF

Testing Options

These options configure properties of automated tests. These options currently just affect tests in the *ctest framework* and don't affect tests in the *pytest framework*.

Table 5: Testing-Related Configuration

Name	Description	Default
`PARALLEL_LAUNCHER`	Launcher to use for parallel tests	charmrun
`PARALLEL_LAUNCHER_NPROC_ARG`	Argument to set number of processing elements for parallel launcher (for use with `charmrun`)	+p
`PARALLEL_LAUNCHER_NPROC`	Number of processors to run parallel unit tests	4
`BUILD_TESTING`	Whether to setup the CTest infrastructure and build unit test binaries (which are primarily built to be executed by the CTest infrastructure). This has no effect on the pytest infrastructure.	"ON"

Debugging Options

The following options are useful for debugging.

Table 6: Debug Options

Name	Description	Default
check	Do extra run-time checking. Useful for debugging, but can potentially slow calculations down	OFF
debug	Whether to enable displaying messages with the DEBUG() series of statements. Also writes messages to out.debug.<P> where P is the (physical) process rank. Still requires the "DEBUG" group to be enabled in Monitor (that is Monitor::is_active("DEBUG") must be true for any output)	OFF
debug_field		OFF
debug_field_face		OFF
debug_verbose	Print periodically all field values. See src/Field/field_FieldBlock.cpp	OFF
trace	Print out detailed messages with the TRACE() series of statements	OFF
trace_charm	Print out messages with the TRACE_CHARM() and TRACEPUP() series of statements	OFF
verbose	Trace main phases	OFF

Misc Options

The following options don't really belong in any other category

Table 7: Misc Options

Name	Description	Default
USE_PRECOMPILED_HEADERS	Precompile headers to try to reduce compile time	ON

Specifying Configuration Options

All variables can be set either on the command line by -D<variable>=<value> or in a machine config, see below. For example, a configure line may look like

```
cmake -DCHARM_ROOT=$(pwd)/../../charm/build-gcc-mpi-proj -DEnzo-E_CONFIG=msu_hpcc_gcc -
↪DGrackle_ROOT=${HOME}/src/grackle/build-gcc -Duse_projections=ON -Duse_jemalloc=ON -
↪Dbalance=ON  ..
```

To see all available (and selected) options you can also run ccmake . in the build directory (after running cmake in first place), or use the ccmake GUI directly to interactively configure Enzo-E by calling ccmake .. in an empty build directory.

If packages (external libraries) are not found automatically or if the wrong one was picked up, you can specify the search path by -D<package_name>_ROOT=/PATH/TO/PACKAGE/INSTALL, cf., the cmake example command just above. Note:

- these package locations are also picked up from the environment, i.e., an alternative option is export <package_name>_ROOT=/PATH/TO/PACKAGE/INSTALL .
- to specify the path to a libpng install, use -DPNG_ROOT=/PATH/TO/LIBPNG instead of -DLIBPNG_ROOT=....

The last option is a machine specific configuration file (see below).

In addition, the general *cmake* option is available to set basic optimization flags via CMAKE_BUILD_TYPE with values of:

- Release (typically -O3)

- `RelWithDebInfo` (typically `-O2 -g`)
- `Debug` (typically `-O0 -g`)

The first choice is generally fastest, while the second is a sensible choice during development (the compiler performs most optimizations and includes debugging information in the executable)

Note that setting `-DOPTIMIZE_FP=ON` will modify each of these build types to include value-unsafe floating point optimizations.

Machine files

Finally, for convenience we provide the option to set default value for your own machine/setup, see the `*.cmake` files in the `config` directory.

You can specify compilers and option in there that will be used a default when `cmake` is called with `-DEnzo-E_CONFG=my_config_name` where `my_config_name` requires a corresponding `config/my_config_name.cmake` to exist. The alternative directory for the machine configuration files is a `.enzo-e` directory in your home directory. If a file with the same name exists in your `${HOME}/.enzo-e` directory and in the `config` directory only the first one will be used. *Note*, all command line parameter take precedence over the default options. In other words, if `USE_DOUBLE_PREC` is `ON` in the machine file (or even automatically through the global default), the running `cmake -DEnzo-E_CONFIG=my_config_name -DUSE_DOUBLE_PREC=OFF ..` will result in a single precision version of Enzo-E.

Options in the machine file can also include the paths to external libraries and can be set via a "cached string", i.e., via

```
set(CHARM_ROOT "/home/user/Charm/charm/build-mpi" CACHE STRING "my charm build")
```

1.1.5 Running

In this section we run Enzo-E on a simple "Hello World" test program and take a look at Enzo-E's output.

1. Run Enzo-E

An included "Hello World" problem can be run using the following from the `$CELLO_HOME` directory:

```
charmrun +p4 bin/enzo-e input/HelloWorld/Hi.in
```

This assumes that the `charmrun` command is in your path. If it is not, then you will need to include the path name as well, e.g.:

```
~/Charm/bin/charmrun +p4 bin/enzo-e input/HelloWorld/Hi.in
```

This also assumes that local connections can be established passwordless. If errors like

```
Permission denied (publickey,gssapi-keyex,gssapi-with-mic,password,hostbased).
Charmrun> Error 255 returned from remote shell (localhost:0)
```

are displayed, a node local run (i.e., no "remote" connections even to the local host) could be used instead by adding `++local` to `charmrun`, e.g.:

```
~/Charm/bin/charmrun ++local +p4 bin/enzo-e input/HelloWorld/Hi.in
```

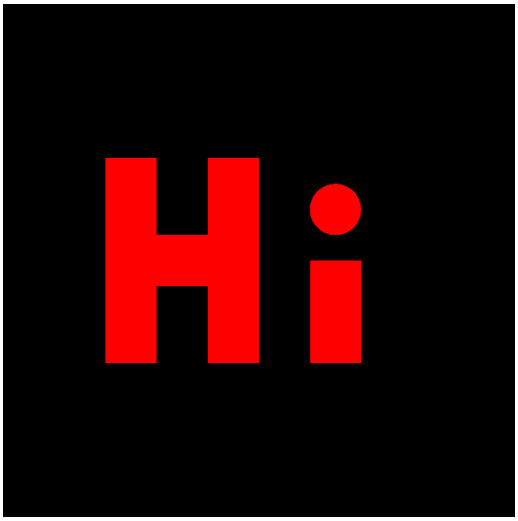
If you receive an error like

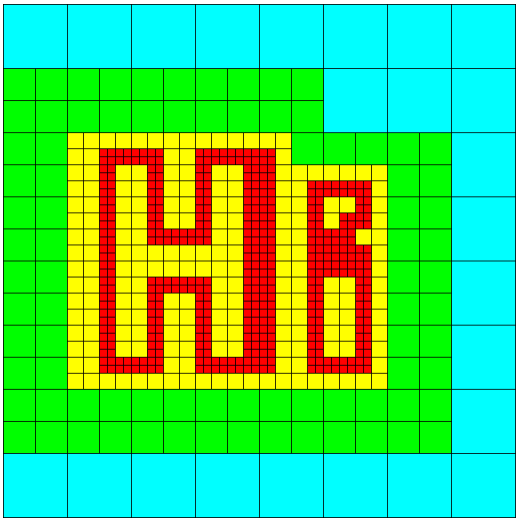
```
Charmrun> Timeout waiting for node-program to connect
```

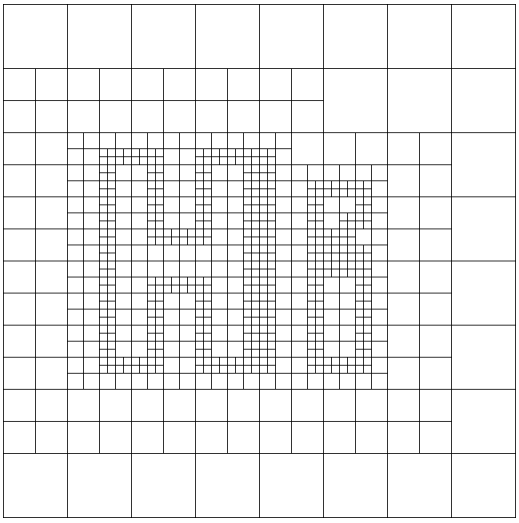
try running `./bin/enzo-e` without `charmrun` as crashes due to, e.g., libraries not being found may not be displaying.

If all goes well, Enzo-E will run the Hello World problem. Below are some of the generated images from the longer-running "HelloWorld.in" problem (note "HelloWorld.in" runs for about an hour, compared to a couple minutes for the shorter "Hi.in" input parameter file). These images show density and mesh hierarchy structure with blocks colored by level and by age.

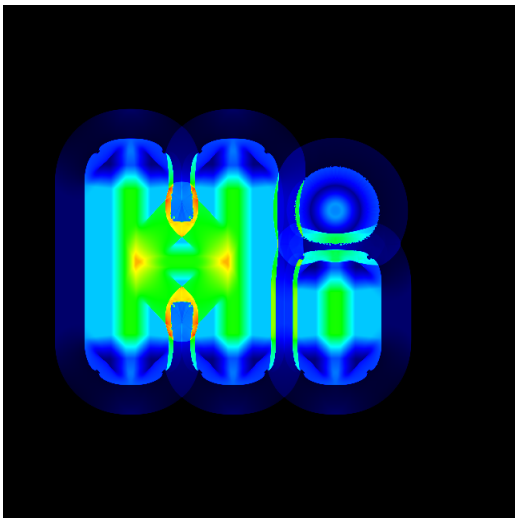
Time = 0.00

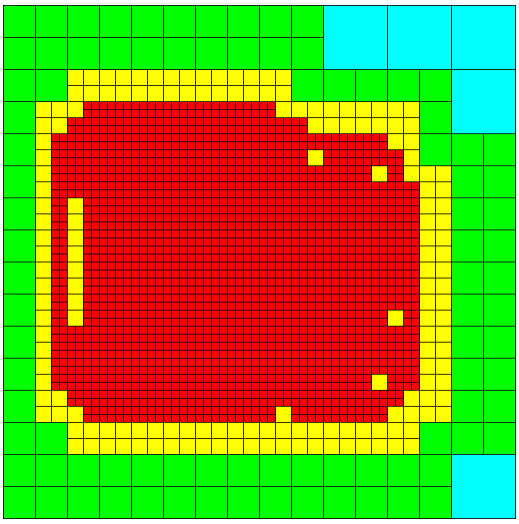


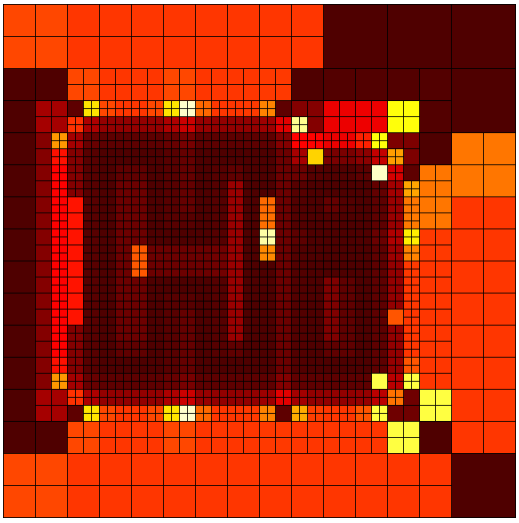




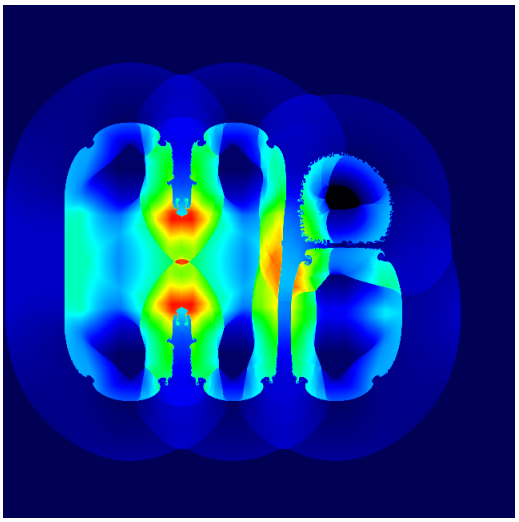
Time = 0.05

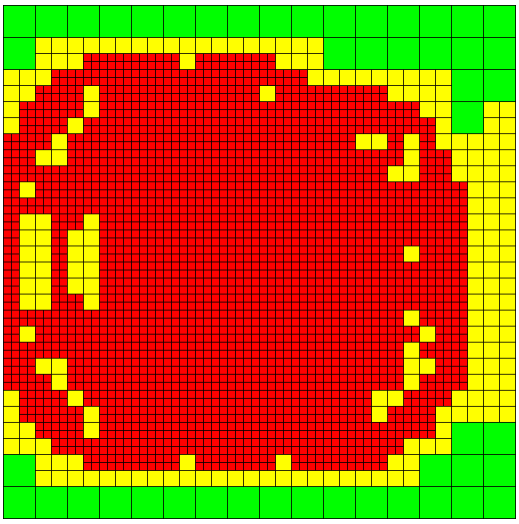


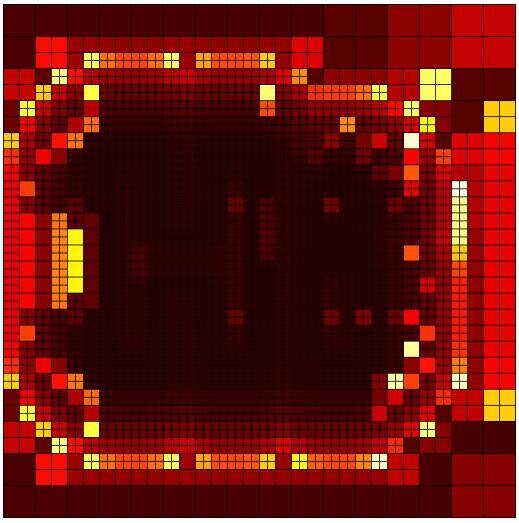




Time = 0.10







If you look at the `Hi.in` parameter file contents, you will notice that there are some `"include"` directives that include other files. When Enzo-E / Cello runs, it will generate a `"parameters.out"` file, which is the input file but with the included files inlined. This `"parameters.out"` file is itself a valid Enzo-E / Cello parameter file (though you may wish to rename it before using it as a parameter file to avoid it being overwritten.)

If you encounter any problems in getting Enzo-E to compile or run, please contact the Enzo-E / Cello community at the users' mailing list and someone will be happy to help resolve the problems.

2020-04-10: Updated with corrections from Joshua Smith.

CHAPTER

TWO

USER GUIDE

2.1 Parameter files

Whereas Enzo uses a flat parameter list with relatively simple data types, Cello uses a more "hierarchical" parameter list, and allows more complicated data types. The rationale is that the power of the application is ultimately limited by the expressiveness of its input; conversely, a rich and powerful input language can allow the application itself to be more powerful.

This page describes how Cello parameter files are structured, which is similar to configuration files readable by libconfig. For documentation on specific parameters in Enzo-E / Cello, please see the Enzo-E / Cello parameter reference page.

2.1.1 Groups

A **group** is a named collection of related parameters or subgroups enclosed in braces:

```
<group> ::= <group-name> "{" <parameter-assignment-list> "}"
```

All group names are capitalized, and all parameters must be contained in exactly one group. Below, `Domain` and `Mesh` are groups, and `lower`, `upper`, and `root_size` are parameters contained in the respective groups.

```
Domain {
  lower = [0.0, 0.0, 0.0];
  upper = [1.0, 1.0, 1.0];
}

Mesh {
  root_size = [128,128,128];
}
```

Parameters within a group can be specified in different parts of a file, or even in separate files. Ordering of parameters within a group only matters when a given parameter is listed more than once, in which case only its last value is used. Below, boundary conditions will be initialized to be "`reflecting`".

```
Boundary {
  type = "periodic";
}

Mesh {
  root_size = [128,128,128];
}
```

(continues on next page)

(continued from previous page)

```
Boundary {
  type = "reflecting";
}
```

2.1.2 Subgroups

A **subgroup** is a related collection of parameters within a group. Subgroups typically begin with a lower case letter, and may be nested.

Below, `Initial` is a group, `value` is a subgroup, and `cycle` and `density` are parameters.

```
Initial {

   value {
      density = [ x + y, x - y < 0.5, 1.0 ];
   }

   cycle = 10;

}
```

The shorthand used for naming parameters in the documentation is <group> : <subgroup> : <parameter>, e.g. "`Mesh : root_size`" or "`Initial : value : density`". Note that This shorthand is used only for documentation–it is not valid syntax within the parameter file.

2.1.3 Parameters

A **parameter** is a variable associated with a group and (optionally) a subgroup. A parameter assignment is the assignment of a value to a parameter. The scope of a parameter is its enclosing group and subgroup. Parameters may be either required or optional, and all optional parameters have default values that are specified in the API function call used to access the parameter value in the code. Some parameters may have more than one type, e.g. a scalar, or a list of scalars. Parameters may be assigned to multiple times, in which case the last value is used. All parameter assignments end with a semi-colon "`;`"

```
<parameter-assignment> ::= <parameter-name> '=' <value> ';'
```

2.1.4 Data types

Parameters can be assigned different *values*, depending on the parameter name and the group the parameter belongs to.

```
<value> ::= <value-integer>
          | <value-scalar>
          | <value-string>
          | <value-variable>
          | <scalar-expression>
          | <logical-expression>
          | <list>
```

A parameter value is one of several different basic data types:

```
==================  ============================
Type                Example
==================  ============================
integer             42
scalar              6.673e-8
string              "velocity_x"
variable            x, y, z, or t
scalar expression   2.0*sin(pi*x) + cos(pi*y)
logical expression  (x < y) || (y < z)
list                [ "velocity_x", 3.14, x < y ]
==================  ============================
```

Integer types are integers, and must be representable using a 32-bit integer.

Scalar types are any floating point or integral numerical values. The constant 'pi' is also recognized as a scalar.

> *Note that floating-point and integers are not interchangeable: if a floating point type is expected, one cannot use an integer.*

String types are enclosed in double-quotes.

Variables represent the position coordinates in space (x, y, and z) and time (t).

Scalar expressions are any "C-like" expression evaluating to a Scalar, and involving Scalar's, Variable's, operations '+' '-' '*' '/', '^' (for power), parenthesis, and (almost all) standard functions in math.h. Scalar expressions are typically used for specifying initial or boundary conditions, etc.

> *Note that "-" when used for subtraction must have blank space after it:* `x-1.0` *will not be parsed correctly, but* `x - 1.0` *will. Similarly, "-" when used for negation must not have a blank space after it.*

Logical expressions are any "C-like" expressions that evaluate to "true" or "false", and involve Scalars, Variabless, and at least one relational operator == != > < <= >=. Logical operators && and || are also recognized. Logical expressions are typically used for defining subregions of the domain for initial or boundary conditions.

Lists represent an ordered sequence of values of mixed types, separated by commas. Lists can be assigned a value, e.g. `list = ["dark","star"];`, or can be appended to, e.g. `list += ["trace"];` Appending to a parameter that has not been assigned to yet is allowed, and equivalent to assignment.

2.1.5 Comments

Comments begin with # and extend to the end of the line.

2.1.6 Include files

The `include` directive is used to include other parameter declarations from other files. For example, one can have a file of parameters for AMR that is maintained separately from problem specific declarations:

```
include "amr_defaults.incl"
include "hydro_defaults.incl"
```

The advantage of using `include` is that repetition between different parameter files can be reduced. However, a disadvantage is that parameters for a given run can be spread out among different files, making it difficult to understand what parameters are defined and their values. Because of this, Cello writes out its parameters to a single file `"parameters.out"`. Since it is a valid parameter file itself, it can even be used to rerun the simulation, though it should be renamed first to avoid it being overwritten.

2.1.7 Examples

Below is a list of sample input files used for developing Enzo-E parameters. Individual parameters are expected to evolve, though the underlying grammar and syntax are relatively fixed.

```
Boundary {
    type = "reflecting";
}

Domain {
    lower = [ 0.0, 0.0 ];
    upper = [ 0.3, 0.3 ];
}

Field {

    list = [ "density", "velocity_x", "velocity_y",
             "total_energy", "internal_energy", "pressure" ];

    courant = 0.8;
    gamma = 1.4;
    ghost_depth = 4;
}

Initial {
   list = ["value"];
   value {
      type = "value";
      density = [ 0.125, ( x  +  y ) <  0.1517 , 1.0 ];
      total_energy = [ 2.8, ( x  +  y ) <  0.1517 , 2.5 ];
      velocity_x = 0.0;
      velocity_y = 0.0;
   }
}

Adapt {
    list = [ "SLOPE" ];

    SLOPE {
        field_list = [ "density" ];
        min_refine = 2.0;
        max_coarsen = 0.5;
        type = "slope";
    }

}

Mesh {

    max_level = 3;
    root_blocks = [ 2, 2 ];
    root_rank = 2;
    root_size = [ 40, 40 ];
```

(continues on next page)

(continued from previous page)

```
}

Method {

    list = [ "ppm" ];

}

Output {

    list = [ "DENSITY", "MESH" ];

    DENSITY {
        name = [ "implosion-d-%03d.png", "count" ];
        type = "image";
        image_type = "data";
        field_list = [ "density" ];
        colormap = [ "black", "red", "yellow", "white"  ];
        schedule {
            step = 10;
            type = "interval";
            var = "cycle";
        }
    }

    MESH {
        name = [ "implosion-mesh-%03d.png", "count" ];
        type = "image";
        image_type = "mesh";
        image_reduce_type = "max";
        image_size = [ 513, 513 ];
        colormap = [ "black", "blue", "cyan", "green", "yellow", "red" ];
        schedule {
            step = 10;
            type = "interval";
            var = "cycle";
        }
    }
}

Stopping {
    cycle = 100;
    time = 2.50;
}
```

2020-04-10: Updated with corrections from Joshua Smith.

2.2 Parameter File Example

Below we describe the different sections of the "HelloWorld.in" input file we ran in the previous section:

2.2.1 Comments

The example input file begins with a comment header. Comments in Cello parameter files begin with # :

```
### Problem: Hello World
### Summary: Evolve high-pressure region shaped as an "E"
###    Date: 2011-10-27
```

Parameters are organized into "groups", which may be nested. Below we look in turn at each group in the HelloWorld.in sample parameter file.

2.2.2 Domain Group

```
Domain {
   lower = [ 0.0, 0.0 ];
   upper = [ 1.0, 1.0 ];
}
```

2.2.3 Mesh Group

```
Mesh {
   root_size      = [160, 160];
   root_blocks    = [  1,   1 ];
}
```

2.2.4 Field Group

```
Field {

   ghost_depth  = 3;
   courant = 0.8;

   fields = [
      "density",
      "velocity_x",
      "velocity_y",
      "total_energy",
      "internal_energy"
    ];
}
```

2.2.5 Method Group

```
Method {

   sequence = [ "ppm" ];

   ppm {
      diffusion   = true;
      flattening  = 3;
      steepening  = true;
      dual_energy = false;
   }
}
```

2.2.6 Physics Group

```
Physics {

   dimensions = 2;
   gamma = 1.4;

}
```

2.2.7 Initial Group

```
Initial {
    density       { value = [ 1.0,
    (0.35 < x && x < 0.40 && 0.25 < y && y < 0.70) ||
    (0.35 < x && x < 0.60 && 0.25 < y && y < 0.30) ||
    (0.35 < x && x < 0.60 && 0.45 < y && y < 0.50) ||
    (0.35 < x && x < 0.60 && 0.65 < y && y < 0.70),
                              0.125 ]; };
    total_energy  { value = [ 1.0  / (0.4 * 1.0) ,
    (0.35 < x && x < 0.40 && 0.25 < y && y < 0.75) ||
    (0.35 < x && x < 0.60 && 0.25 < y && y < 0.30) ||
    (0.35 < x && x < 0.60 && 0.45 < y && y < 0.55) ||
    (0.35 < x && x < 0.60 && 0.65 < y && y < 0.75),
                              0.14 / (0.4 * 0.1)]; };
    velocity_x    { value = [0.0]; };
    velocity_y    { value = [0.0]; }
}
```

2.2.8 Boundary Group

```
Boundary { type = "reflecting" }
```

2.2.9 Stopping Group

```
Stopping {
   cycle = 500;
}
```

2.2.10 Output Group

```
Output {

   file_groups = ["cycle_step"];

   cycle_step {
      field_list = ["density"];
      type     = "image";
      name     = ["E-%03d.png","cycle"];
      schedule = ["cycle","interval", 10];
      colormap = ["blue", "green", "yellow", "red"];
   }
}
```

2.3 Enzo-E Methods

This section decribes all Methods available in Enzo-E: what they do, what method-specific parameters they have, what fields / particles they access or update, and how interrelated Methods are intended to be used with each other.

Current Enzo-E methods available include those listed below.

2.3.1 `"accretion"` method

For cells within a spherical accretion zone around a sink particle, mass is removed (i.e., the values of the density field are reduced) and added to the sink particle. The momentum change of gas is in the accretion zone due to the mass loss is accounted for by changing the momentum of the sink particle, so that total momentum is conserved. The amount of mass removed is determined by which "flavor" of accretion is chosen (specified by the `"accretion:flavor"` parameter), as well as the values of the "density threshold" (specified by `"accretion:physical_density_threshold_cgs"`) and the "maximum mass fraction" (specified by `"accretion:max_mass_fraction"`).

In `"threshold"` flavor accretion, the change in density of each cell is zero if the current density is below the density threshold. If the current density is above the density threshold, the change in density is the current density minus the density threshold, or the maximum mass fraction times the current density, whichever is smaller.

In `"bondi_hoyle"` flavor accretion, the density change in each cell is calculated according to the method described in Mark R. Krumholz et al 2004, ApJ, 611, 399. Furthermore, the density change is limited in the same way as in `"threshold"` accretion.

In "flux" flavor accretion, the density change in each cell is calculated according to the method described in Andreas Bleuler & Romain Teyssier 2004, MNRAS, 445, 4015-4036. Furthermore, the density change is limited in the same way as in "threshold" accretion.

In "dummy" flavor accretion, no accretion is done (essentially, the accretion rate is zero). This can be useful for testing purposes.

This method can only be used if "merge_sinks" is also used, with "merge_sinks" preceding "accretion". In addition, this method requires the use of three spatial dimensions.

This method requires the following fields (in addition to the fields required by the hydro method): "density_source", "density_source_accumulate", "mom_dens_x_source", "mom_dens_x_source_accumulate", "mom_dens_y_source", "mom_dens_y_source_accumulate", "mom_dens_z_source", and mom_dens_z_source_accumulate". In addition, if sink particles have a "metal_fraction" attribute, there must be a "metal_density" field.

This method also requires sink particles to have the following attributes: "mass", "x", "y", "z", "vx", "vy", "vz", and "accretion_rate", which must all be of type "default".

parameters

Table 1: Method accretion parameters

Parameter	Type	Default	Description
"accretion_radius_cells"	*float*	*4.0*	*The accretion radius (i.e., the radius of the spherical accretion zone) in units of the minimum cell width (i.e., if the cell width along all the x, y, and z-axes are hx, hy, and hz, then the minimum cell width is the minimum of hx, hy, and hz), at the highest refinement level. Its value must be less than one fewer than the minimum ghost depth for "flux" accretion, and less than the minimum ghost depth for other flavors of accretion. The ghost depth is 4 (along all axes) by default.*
"flavor"	*string*	""	*The flavor of accretion used, which can be either "threshold", "bondi_hoyle", or "flux". If this parameter is not set in the parameter file, or if some other string is provided, then the accretion method will be called but will do nothing.*
"physical_density_threshold_cgs"	*float*	*1.0e-24*	*The value of the physical density threshold in cgs units. The density in each cell in the accretion zone cannot go below this value during the accretion process. The value of the density threshold in code units must be greater than or equal to the value of the density floor imposed by the hydro method.*
"max_mass_fraction"	*float*	*0.25*	*This parameter specifies the maximum fraction of mass which can be accreted from a cell in one timestep. This value of this parameter must be between 0 and 1.*

2.3.2 "balance" method

While the Charm++ parallel programming system supports many load balancers, Enzo-E also implements its own dynamic load balancer based on space-filling curves. As such, it relies on the "ordering_morton" Method to be called before "balance".

There are currently no method-specific parameters, though schedule parameters are likely to be useful, since one generally doesn't want or need to run the load balancer every cycle. Also, as further orderings are implemented beyond the "Morton" ordering, a parameter is likely to be introduced in the future for specifying the ordering to use.

restrictions

There are two known potential pitfalls when using the built-in Enzo-E load balancer.

First, there is a bug related to the ordering of Methods in `Method : list`, where the simulation can hang if `balance` is the last Method. To bypass this bug, please use `order_morton` and `balance` at the beginning of the `Method : list` parameter.

Second, the minimum level parameter `Adapt : min_level` should be set such that the coarsest level (which may be negative) is a single block. (This restriction is likely to be lifted in the near future when this particular parameter is removed.) (Also note this is a restriction related to the "order_morton" method, not the "balance" method.)

For example, to load balance a simulation with a 4^3 root-level blocking (4 root-level blocks of any size along each axis), one can use the following:

```
Mesh {
   root_blocks = [4, 4, 4];     # must be a regular blocking!
   # ...
}

Adapt {
   min_level = -2;              # must be log_2 of root_blocks!
   # ...
}

Method {
   list = ["order_morton", "balance", "ppm" ];
   order_morton { schedule { var = "cycle"; step = 20; } }
   balance      { schedule { var = "cycle"; step = 20; } }
   # ...
}
```

2.3.3 `"comoving_expansion"` method

Adds the comoving expansion terms to the physical variables.

2.3.4 `"grackle"` method

Calls methods provided by the external Grackle 3.0 chemistry and cooling library.

Compatability with hydro/mhd solvers

The `"grackle"` method is compatible with both the `"ppm"` and the `"mhd_vlct"` methods. The convention is to list the hydro method before `"grackle"` in the `Field:list` parameter. This configuration performs advection and radiative cooling in an operator-split manner (*Note: there isn't currently support for performing radiative cooling during the predictor step of the VL+CT solver*).

Integration with hydro-solvers is self-consistent when `Method:Grackle:primordial_chemistry` has values of `0` or `1`. However, the integration is somewhat inconsistent when the parameter exceeds `1`. While users shouldn't be too concerned about this latter scenario unless they are simulating conditions where H_2 makes up a significant fraction of the gas density, we describe the inconsistencies in greater detail below.

When `Method:Grackle:primordial_chemistry > 1`, the Grackle library explicitly models chemistry involving H_2 and how it modifies the adiabtic index. Grackle's routines treat γ_0, the "nominal adiabatic index" specified by

`Physics:fluid_props:eos:gamma`, as the adiabatic index for all monatomic species (this should be `5.0/3.0`). To that end, Grackle supplies functions that can effectively be represented as $\gamma(e, n_{\mathrm{H}_2}, n_{\mathrm{other}})$ and $p(\rho, e, n_{\mathrm{H}_2}, n_{\mathrm{other}})$. In these formulas:

- p, ρ and e correspond to the quantities held by the `pressure`, `density` and `internal_energy` fields. *(Note: the γ function's dependence on e accounts for the dependence of γ_{H_2} on temperature)*
- n_{H_2} specifies the number density of H_2. n_{other} specifies a selection of the other primordial species (that roughly approximate the total number density). In practice, these are computed from passively advected species fields.

There are a handful of locations within the `"ppm"` and `"mhd_vlct"` methods where this treatment is relevant:

1. **Computing the timestep:** each hydro/mhd method uses the $p(\rho, e, n_{\mathrm{H}_2}, n_{\mathrm{other}})$ function for the pressure values. However, they both use γ_0 in other places (such as the occurence of adiabatic index in the sound speed formula).
2. **Pre-reconstruction pressure calculation:** each hydro/mhd solver internally computes the pressure that is to be reconstructed with $p = (\gamma_0 - 1)e\rho$.
3. **Riemann Solver:** in each hydro/mhd solver, the Riemann Solver completely ignore the grackle supplied functions.
4. **VL+CT Energy floor and DE synchronization:** the internal energy floor is computed from the pressure floor using: $e_{\mathrm{floor}} = \frac{p_{\mathrm{floor}}}{(\gamma_0-1)\rho}$ (thus, p_{floor} may exceed $p(\rho, e_{\mathrm{floor}}, \ldots)$). Additionally, synchronizing the internal energy with total energy relies on γ_0.
5. **PPM reconstruction:** uses γ_0.

2.3.5 `"gravity"` method

Particle-mesh ("PM") method component to compute gravitational potential given a total density field, and calculate associated acceleration fields.

2.3.6 `"heat"` method

A sample Method for implementing forward-euler to solve the heat equation.

2.3.7 `"merge_sinks"` method

Merges together sink particles which are separated by less than a given "merging radius". This is done by copying all sink particles to / from all neighbouring blocks. A Friend-of-Friends algorithm is used to partition particles into groups, where all particles within a given group are separated by less than a merging radius. If a group has more than one particle, one of the particles has its properties changed: its position becomes that of the centre-of-mass of the group, and it takes the total mass, momentum and mass fraction of the whole group. In addition, its 'lifetime' attribute is set to be the maximum of the group, its 'creation_time' attribute is set to be the minimum of the group, and its 'id' attribute is set to the minimum of the group. Other particles in the group are marked for deletion. The final step is for each block to delete all the remaining sink particles which are 'out-of-bounds' of the block.

This method requires sink particles to have the following attributes: `"mass"`, `"x"`, `"y"`, `"z"`, `"vx"`, `"vy"`, `"vz"`, `"is_copy"`, `"id"`, `"lifetime"`, and `"creation_time"`. All these attributes must be of type `"default"`, except for `"is_copy"` and `"id"` which must be of type `"int64"`. Furthermore, `"is_copy"` must be initialized to 0 for all particles.

This method also requires that the number of root blocks across all axes is greater than 2, i.e., that `"Mesh:root_blocks" = [a,b,c]`, where `a`, `b`, and `c` are all greater than 2.

This procedure cannot handle the case where particles originally from non-neighbouring blocks are put into the same FoF group. If this is found to occur, the program stops and prints an error message. This situation is unlikely to happen, unless the merging radius is too large relative to the block size.

Currently this will only run in unigrid mode. This is because this method will only work correctly if all blocks containing sink particles are of the same size, or equivalently, on the same refinement level. For this reason, there is a check in the constructor of EnzoMethodMergeSinks for whether `"Adapt: max_level"` is equal to zero. In future, we plan to implement a refinement condition that any block containing a sink particle needs to be on the highest level of refinement. In this case, the assumption that blocks containing sink particles are all on the same level of refinement would be valid.

WARNING: there is currently a memory leak issue when running with this method which can cause Enzo-E to crash in mysterious ways. If this problem is encountered, it is advised to increase the batch size parameter (`"Particle:batch_size"`) by a factor of a few before attempting to run again. To be completely safe, the user can set a batch size larger than the total number of sink particles in the whole simulation, which should be feasible for small test problems.

parameters

Table 2: Method `merge_sinks` parameters

Parameter	Type	Default	Description
`"merging_radius_cells"`	*float*	*8.0*	*The merging radius in units of the minimum cell width (i.e., the minimum across all 3 dimensions), at the highest refinement level.*

2.3.8 `"mhd_vlct"` method

This implements the VL + CT (van Leer + Constrained Transport) unsplit Godunov method described by Stone & Gardiner (2009) . This solver operates in 2 modes (designated by the required `Method:mhd_vlct:mhd_choice` parameter):

1. a pure hydrodynamical mode (it cannot handle magnetic fields)
2. a MHD mode.

The algorithm is a predictor-corrector scheme, with attributes similar to the MUSCL-Hancock method. For both modes, the method includes two main steps. First the values are integrated to the half time-step. Then, the values at the half time-step are used to caluclate the change in the quantities over the full time-step.

In the MHD mode, the algorithm is combined with the constrained transport method. The primary representation of the magnetic field, $\vec{B}$, components are stored in 3 face-centered Cello fields. In more detail:

- B_x is stored on the x-faces
- B_y is stored on the y-faces
- B_z is stored on the z-faces

The method also tracks components of the magnetic fields at the cell-centers, which just store the average of the values from the cell's faces.

Currently, this method only support 3-dimensional problems. In the future, alternative modes supporting MHD could easily be implemented that use divergence-cleaning schemes instead of constrained transport.

parameters

Note that the courant factor (specified by the `"courant"` parameter) should be less than 0.5.

Table 3: Method `mhd_vlct` parameters

Parameter	Type	Default	Description
mhd_choice	*string*	*none*	*Specifies handling of magnetic fields (or lack thereof)*
time_scheme	*string*	*vl*	*Specifies time integration scheme (CURRENTLY JUST FOR TESTING)*
riemann_solver	*string*	*hlld*	*name of the Riemann Solver to use*
reconstruct_method	*string*	*plm*	*Reconstruction method*
theta_limiter	*float*	*1.5*	*controls dissipation of the "plm"/"plm_enzo" reconstruction method.*

fields

Table 4: Method `mhd_vlct` fields

Field	Type	Read/Write	Description
`"density"`	`enzo_float`	[rw]	
`"velocity_x"`	`enzo_float`	[rw]	
`"velocity_y"`	`enzo_float`	[rw]	
`"velocity_z"`	`enzo_float`	[rw]	
`"total_energy"`	`enzo_float`	[rw]	specific total energy
`"bfield_x"`	`enzo_float`	[rw]	Cell-centered bfield (average of the corresponding `bfieldi_x`)
`"bfield_y"`	`enzo_float`	[rw]	Cell-centered bfield (average of the corresponding `bfieldi_y`)
`"bfield_z"`	`enzo_float`	[rw]	Cell-centered bfield (average of the corresponding `bfieldi_z`)
`"bfieldi_x"`	`enzo_float`	[rw]	Primary representation of x-component of bfield (lies on x-faces).
`"bfieldi_y"`	`enzo_float`	[rw]	Primary representation of y-component of bfield (lies on y-faces).
`"bfieldi_z"`	`enzo_float`	[rw]	Primary representation of z-component of bfield (lies on z-faces).
`"pressure"`	`enzo_float`	[w]	computed from `total_energy` (`internal_energy` if dual-energy)
`internal_energy`	`enzo_float`	[rw]	if dual-energy

In hydro-mode, none of the 6 fields used to store the magnetic field should be defined.

At initialization the face-centered magnetic field should be divergence free. Trivial configurations (e.g. a constant magnetic field everywhere) can be provided with the `"value"` initializer. For non-trivial configurations, we have provide the `"vlct_bfield"` initializer which can initialize the magnetic fields (face-centered and cell-centered) from expression(s) given in the parameter file for component(s) of the magnetic vector potential.

`fluid_props` compatability

This method makes use of the `density` and `pressure` floor parameters that are set in the `physics:fluid_props:floors` section of the parameter file. See *Floors* for more details about specifying these parameters. This method requires that both parameters are specified and that they have positive values.

This method is also compatible with the `"modern"` dual-energy formalism. See *dual-energy formalism* for additional details.

reconstruction

This subsection details the available interpolation methods for reconstructing the left and right states of the cell-centered interfaces. Presently, all available methods perform reconstruction on cell-centered primitive quantites, $\mathbf{w} = (\rho, \mathbf{v}, p, \mathbf{B})$

To simplify the determination of the necessary number of ghost zones for a given combination of reconstruction algorithms on a unigrid mesh, we define the concepts of "stale depth" and "staling rate". We define a "stale" value as a value that needs to be refreshed. "Stale depth" indicates the number of field entries, starting from the outermost field values on a block, that the region encompassing "stale" values extends over. Every time quantities are updated over a (partial/full) timestep, the stale depth increases. We define the amount by which it increases as the "staling rate" (which depends on the choice of interpolation method).

For a unigrid simulation, the number of ghost zones is `1 + staling_rate` where `staling_rate` depends on the chosen reconstruction method.

For each available reconstruction method, the following table provides the name that must be passed to *reconstruct_method* in the input file, the associated staling depth, and a brief description.

Table 5: Available `mhd_vlct` reconstructors (and slope limiters)

Name	Staling Depth	Description
`"nn"`	*1*	*Nearest Neighbor - (1st order) reconstruction of primitives*
`"plm"` or `"plm_enzo"`	*2*	*Piecewise Linear Method - (2nd order) reconstruction of primitives using the slope limiter from Enzo's Runge–Kutta integrator. This is tuned by the* `"theta_limiter"` *parameter, which must satisfy* `1 <= "theta_limiter" <= 2`*. As in Enzo, the default value is 1.5. A value of 1 is the most dissipative and it is equivalent to the traditional minmod limiter. A value of 2 is the least dissipative and it corresponds to an MC limiter (monotized central-difference limiter).*
`"plm_athena"`	*2*	*Piecewise Linear Method - (2nd order) reconstruction of primitives using the slope limiter from Athena (& Athena++). For some primitive variable,* $\mathbf{w}_i$*, the limited slope is defined in terms of the left- and right-differences:* $\delta\mathbf{w}_{L,i} = \mathbf{w}_i - \mathbf{w}_{i-1}$ *and* $\delta\mathbf{w}_{R,i} = \mathbf{w}_{i-1} - \mathbf{w}_{i+1}$*. If the signs of the differences don't match (or at least 1 is 0), then the limited slope is 0. Otherwise the limited slope is the harmonic mean of the differences.*

We provide a few notes about the choice of interpolator for this algorithm:

- The recommended choices of reconstruction algorithm is piecewise-linear reconstruction (most test problems have been run using `plm` with `theta_limiter=2`, matching the integrator description in Stone & Gardiner 2009).
- It is supposed to be possible to reconstruct the characteristic quantities for this method or to use higher order reconstruction in place of `"plm"`
- Reconstruction is always performed on the cell-centered magnetic fields. After reconstructing values along a given axis, the values of the reconstructed magnetic field component for that axis are replaced by the face-centered magnetic field values.

Note: Under the hood, the default predictor-corrector temporal integration scheme **always** uses `"nn"` reconstruction to compute fluxes during the prediction stage (aka the partial timestep); the scheme breaks for any other choices.

This is why the number of required ghost zones is `1 + staling_rate`:

- the first term is the staling depth of the `"nn"` reconstructor
- the second term is the staling depth of the reconstructor specified by the *reconstruct_method* parameter; this is used for computing fluxes for the second stage (for the full timestep).

riemann solvers

This subsection details the available Riemann Solvers. Currently all available Riemann Solvers are defined to use magnetic fields, however, they all appropriately handle the cases where the magnetic fields are unformly 0. We provide a list of the names used to specify each Riemann Solver in the input file, and a brief description for each of them:

- `"hll"` The HLL approximate Riemann solver with wavespeeds bounds estimated as $S_L = \min(u_L - a_L, u_R - a_R)$ and $S_R = \max(u_L + a_L, u_R + a_R)$. This is one of the proposed methods from Davis, 1988, SIAM J. Sci. and Stat. Comput., 9(3), 445–473. The same wavespeed estimator was used in MHD HLL solver implemented for Enzo's Runge Kutta solver. Currently, this has only been implemented for MHD mode and it will raise an error as it isn't tested.
- `"hlle"` The HLLE approximate Riemann solver - the HLL solver with wavespeed bounds estimated according to Einfeldt, 1988, SJNA, 25(2), 294–318. This method allows the min/max eigenvalues of Roe's matrix to be wavespeed estimates. For a description of the procedure for MHD quantities, see Stone et al. (2008) . If using an HLL Riemann Solver, this is the recommended choice. Currently, this has only been implemented for MHD mode.
- `"hllc"` The HLLC approximate Riemann solver. For an overview see Toro, 2009, *Riemann Solvers and Numerical Methods for Fluid Dynamics*, Springer-Verlag. This is a solver for hydrodynamical problems that models contact and shear waves. The wavespeed bounds are estimated according to the Einfeldt approach. This can only be used in hydro mode.
- `"hlld"` The HLLD approximate Riemann solver described in Miyoshi & Kusano, 2005. JCP, 315, 344. The wavespeed bounds are estimated according to eqn 67 from the paper. This reduces to an HLLC Riemann Solver when magnetic fields are zero (the wavespeed bounds will differ from `"hllc"`). This can only be used in MHD mode.

Note: When the dual-energy formalism is in use, all of the solvers treat the internal energy as a passively advected scalar.

This is not completely self-consistent with the assumptions made by the HLLD solver. Unlike the other HLL-solvers which assume constant pressure in the intermediate regions of the Riemann Fan the HLLD solver assumes constant total pressure. It is unclear whether this causes any problems.

2.3.9 `"m1_closure"`: multigroup radiative transfer

Enzo-E's multigroup M1 Closure radiative transfer solver. The M1 Closure solver is implemented following Rosdahl et al. (2013).

This method adds the following capabilities:

1. **Photon injection:** Photons are sourced from star particles and CiC-deposited onto the mesh using a cloud radius of one cell width. Depositions occuring at block boundaries are communicated to neighboring blocks via ghost zone refresh with `set_accumulate = true`. This method accesses the `"luminosity` attribute for `"star"` particles, where luminosity is in units of s^{-1}. One can provide an SED using the `SED` parameter and specifying `m1_closure:radiation_spectrum="SED"`. Alternatively, one can specify `m1_closure:radiation_spectrum="blackbody"`, in which case a blackbody spectrum will be integrated. Radiation from recombination is also optionally included by setting the `recombination_radiation` parameter.
2. **Photon transport:** The transport equation is solved to update the radiation fields. Attenuation is optionally included with the `attenuation` parameter.

3. **Photoionization and heating:** This method calculates photoionization and heating rates, which can then be accessed by the `"grackle"` method with `Method:grackle:with_radiative_transfer=true`. Photoionization cross sections can either be provided in the parameter file or calculated inline. This is controlled using the `m1_closure:cross_section_calculator` parameter.

This is a reduced speed-of-light (RSOL) method, meaning that the value of the speed of light can be varied by setting the `m1_closure:clight_fraction` parameter. The radiation timescale is generally set by the speed of light, so a smaller speed of light will allow the simulation to progress faster. Some care must be taken when choosing the value for the RSOL, as the approximation is more valid in dense gas. The general rule of thumb is the the chosen RSOL must be much less than the typical I-front propagation speed in the medium. For densities typical for the interstellar medium $\left(10^{-2}\ \mathrm{cm}^{-3} - 10^{1}\ \mathrm{cm}^{-3}\right)$, fractions as low as $10^{-2}\ \mathrm{cm}^{-3}$ are valid. For simulations that seek to resolve the reionization of the intergalactic medium, however, the true value of speed of light must be taken in order for reionization to occur at the correct time. See Section 4 of Rosdahl et al. (2013) for a more detailed discussion.

This method **must** be used in tandem with the `"m1_closure"` initializer by appending `"m1_closure"` to `Initial:list`.

Note: Additional term in the radiative transfer equation corresponding to cosmological redshift not yet included. As such, redshifting of radiation into and out of groups is not captured.

The radiation timescale is set by the courant condition $\Delta t \leq \frac{\Delta x}{3c_r}$. For $c_r = c$, this will result in a timestep that is **very** small. Subcycling of radiative transfer with respect to hydrodynamics will be implemented soon!

Photochemistry is only supported for six-species: HI, HII, HeI, HeII, HeIII, and e^-.

Required Fields

The evolved fields are `"photon_density_i"`, `"flux_x_i"`, `"flux_y_i"`, and `"flux_z_i"`, where photon densities and fluxes are in units of $[L]^{-3}$ and $[L]^2[T]^{-1}$, respectively, and i denotes the group number. A set of fields must be defined for each group, and fluxes must be defined in the x, y, and z directions regardless of the dimensionality of the simulation. For technical reasons, an additional field called `"photon_density_i_deposit"` must be defined for each group.

For example, a simulation that evolves three groups must define these fifteen fields: `"photon_density_0"`, `"photon_density_0_deposit"`, `"flux_x_0"`, `"flux_y_0"`, `"flux_z_1"`, `"photon_density_1"`, `"photon_density_1_deposit"`, `"flux_x_1"`, `"flux_y_1"`, `"flux_z_1"`, `"photon_density_2"`, `"photon_density_2_deposit"`, `"flux_x_2"`, `"flux_y_2"`, `"flux_z_2"`.

Nine additional fields must be defined corresponding to the elements of the 3D radiation pressure tensor: `"P00"`, `"P01"`, `"P02"`, `"P10"`, `"P11"`, `"P12"`, `"P20"`, `"P21"`, `"P22"` Note that only these nine fields are required, regardless of the number of radiation groups specified.

Photionization and heating rates are calculated and stored in the following fields: `"RT_HI_ionization_rate"`, `"RT_HeI_ionization_rate"`, `"RT_HeII_ionization_rate"`, and `"RT_heating_rate"`.

2.3.10 "order_morton" method

This method is used to compute the "Morton-ordering" of blocks in the AMR hierarchy. This is a space-filling curve with moderate locality properties.

This method is typically used with other methods, including "check" and "balance". Output are long integer Block scalar data "order_morton:index" and "order_morton::count that give the unique index of the block in the ordering 0 <= index < CkNumPes(), and the total number of blocks (which is the same for all blocks).

See the *"balance" method* section for a code example. The "order_morton" method currently has no method-specific parameters, though is typically called with a schedule matching that of the methods that depend on the ordering.

Note: the name of this method may change in the future to "order", with "morton" being provided as a parameter to specify the ordering type.

restrictions

Currently, the minimum level parameter Adapt : min_level must be set such that the coarsest level (which may be negative) is a single block. This restriction is likely to be lifted in the near future since this parameter will soon be obsolete.

2.3.11 "pm_deposit" method

Particle-mesh ("PM") method component to deposit of field and particle mass into a "total density" field

parameters

Table 6: Method ppm parameters

Parameter	Type	Default	Description
"alpha"	*float*	*0.5*	*Deposit mass at time t + alpha * dt*

fields

particles

For a given particle type to be deposited to the total density field, it must be part of the "is_gravitating" group, and must have either an attribute called "mass", or a constant called "mass", but not both.

If "mass" is an attribute, we loop through the mass attribute array to get the mass of each particle; and if "mass" is a constant with a value specified in the input parameter file, the mass of each particle is equal to this value. In either case, the value of the divided by the cell volume to get a density quantity, which is deposited on to the grid via a CIC interpolation scheme.

2.3.12 "pm_update" method

Particle-mesh ("PM") method component to update particle positions given acceleration fields. Only particle types in the "is_gravitating" group are updated.

2.3.13 "ppm" method

This implements the modified piecewise parabolic method (PPM) in Enzo.

parameters

Table 7: Method ppm parameters

Parameter	Type	Default	Description
"diffusion"	*logical*	*false*	*PPM diffusion parameter*
"flattening"	*integer*	*3*	*PPM flattening parameter*
"minimum_pressure_support_parameter"	*integer*	*100Tiny*	*Enzo's MinimumPressureSupportParameter*
"pressure_free"	*logical*	*false*	*pressure-free flag*
"steepening"	*logical*	*false*	*PPM steepening parameter*
"use_minimum_pressure_support"	*logical*	*false*	*Minimum pressure support*

fields

Table 8: Method ppm fields

Field	Type	Read/Write	Description
"density"	enzo_float	[rw]	
"velocity_x"	enzo_float	[rw]	
"velocity_y"	enzo_float	[rw]	if rank 2
"velocity_z"	enzo_float	[rw]	if rank 3
"total_energy"	enzo_float	[rw]	
"acceleration_x"	enzo_float	[r]	if gravity
"acceleration_y"	enzo_float	[r]	if gravity and rank 2
"acceleration_z"	enzo_float	[r]	if gravity and rank 3
"pressure"	enzo_float	[w]	computed from total_energy

fluid_props compatability

This method is also compatible with the "bryan95" dual-energy formalism. See *dual-energy formalism* for additional details.

This method currently ignores all of the floor parameters that are set in the physics:fluid_props:floors section of the parameter file.

2.3.14 "ppml" method

PPML ideal MHD solver

2.3.15 "sink_maker" method

This method runs on blocks at the highest level of refinement, and forms sink particles in cells which satisy certain criteria.

First, the gas density in the cell must be larger than the density threshold, which is specified by the "sink_maker:physical_density_threshold_cgs" parameter. If so, the mass of the potential sink particle is $V_{cell} \times \max(\rho - \rho_{thresh}, f_{max}\,\rho)$, where V_{cell} is the cell volume, ρ is the cell gas density, ρ_{thresh} is the density threshold, and f_{max} is the maximum fraction of the cell mass which can be turned into a sink particle in one timestep, which is specified by the "sink_maker:maximum_mass_fraction" parameter. This mass must be greater than the minimum sink mass, which is specified by "sink_maker:min_sink_mass_solar" (in solar mass units).

Next, the local Jeans length λ_J is calculated, where $\lambda_J = \frac{\pi c_s^2}{G\rho}$, where c_s is the sound speed of the gas in the cell, and G is the gravitational constant. It is then checked whether $\lambda_J < N_J \times h_{max}$, where N_J is specified by "sink_maker:jeans_length_resolution_cells", and $h_{max} = max(h_x, h_y, h_z)$, where h_x, h_y, and h_z are the cell widths along the x-, y- and z-axes, respectively.

The next check is that the flow is converging. This is done by computing the strain tensor, given by $A_{ij} = \frac{1}{2}\left(\frac{dv_i}{dx_j} + \frac{dv_j}{dx_i}\right)$. Since this tensor / matrix is real and symmetric, it has three real eigenvalues, and the check is equivalent to checking that all three eigenvalues are negative.

The final check is optional, i.e., it is only done if "sink_maker:check_density_maximum" is "true", and a cell will pass this check if it is a local density maximum, that is, its density is larger than the density in all 26 neighboring cells.

If a cell passes all the checks that are performed, a sink particle is created. Its position is the coordinates of the center of the cell, plus a small random offset. The maximum size of the random offset is controlled by "sink_maker:max_offset_cell_fraction".

This method requires sink particles to have the following attributes: "mass", "x", "y", "z", "vx", "vy", "vz", and "creation_time", which must all be of type "default"; and "id" and "is_copy", which must be of type "int64". If sink particles have a "metal_fraction" attribute, there must be a "metal_density" field.

parameters

Table 9: Method `sink_maker` parameters

Parameter	Type	Default	Description
`"jeans_length_resolution_cells"`	*float*	*4.0*	*If the local Jeans length in a cell is less than this quantity multiplied by the maximum cell width, then the cell is a candidate for forming a sink. The maximum cell width is maximum value out of hx, hy, and hz, where hx, hy, and hz are the cell widths across the x-, y- and z-axes, respectively.*
`"physical_density_threshold_cgs"`	*float*	*1.0e-24*	*The value of the physical density threshold in cgs units. The density in a cell must be greater than the density threshold to be able to form a sink. The density in a cell after sink formation will be no less than the density threshold. The value of the density threshold in code units must be greater than or equal to the value of the density floor imposed by the hydro method.*
`"max_mass_fraction"`	*float*	*0.25*	*The mass of a newly-formed sink is bounded above by this parameter multiplied by the cell density multiplied by the cell volume. The value of this parameter must be between 0 and 1.*
`"min_sink_mass_solar"`	*float*	*0.0*	*The minimum mass of a newly-formed sink particle, in solar mass units.*
`"check_density_maximum"`	*logical*	*true*	*Determines whether a cell is required to be a local density maximum in order to form a sink particle.*
`"max_offset_cell_fraction"`	*float*	*0.0*	*When a cell creates a sink particle, the x/y/z coordinate of its initial position will be the x/y/z coordinate of the center of the cell, plus a random value generated from a uniform distribution on the interval [-A,A], where A is equal to this parameter multiplied by the cell width along the x/y/z axis.*
`"offset_seed_shift"`	*integer*	*0*	*When computing the random offset for the initial position of a sink particle, we compute an unsigned 64 bit integer value from the cycle number, the block index, and the cell index, and then add on this value to give the seed for the random number generator.*

2.3.16 `"trace"` method

Moves tracer particles given the velocity field.

2.3.17 `"turbulence"` method

Turbulence driving.

2.4 Enzo-E Physics Config-Groups

[This page is under development]

This section decribes the Physics subgroups that can be specified in the `Physics` Group of the parameter file. Each subgroup directly maps to a different type of `Physics` object in the codebase. These objects hold information that needs to be accessible across different Enzo-E methods and/or initializers.

In a parameter file, a user currently needs to explicitly list the names of all of the physics objects that they are configuring within the *Physics:list* parameter.

Note: Currently, the `"fluid_props"` and `"gravity"` don't need to be *Physics:list* for Enzo-E to parse and make use of parameters in the respective Physics subgroups. This choice is made purely for the sake of backwards compatability (so that parameter files designed for earlier versions of the code will continue to work properly).

With that said, this is mostly just an implementation detail (that is subject to change, especially as deprecated parameters are eventually removed). At this time, Users should still explicitly list these physics groups within the *Physics:list* parameter if they want to use/configure them.

2.4.1 `"cosmology"`

Specifies cosmological parameters

2.4.2 `"fluid_props"`

Specifies parameters related to the fluid properties. These parameters are largely into a few subcategories:

1. `dual_energy`: these parameters govern whether the simulation uses the dual-energy formalism and (if applicable) which formulation is used.
2. `floors`: these parameters specify floors for used for a small assortment of quantities.
3. `eos`: specifies parameters related to the (caloric) equation of state.

dual-energy formalism

A typical Godunov scheme might carry the specific total energy, E, as one of its primary variables (the `"total_energy"` field). To compute the pressure (e.g. for reconstruction), the scheme needs to first compute the specific internal energy, $e = p/((\gamma - 1)\rho)$, by subtracting the non-thermal energy components from E. Numerical problems arise under extreme conditions when $E_{\mathrm{non-thermal}}/e$ is extremely large (e.g. from some combination of high Mach number or magnetic energy) since e is the difference between 2 large numbers, $E - E_{\mathrm{non-thermal}}$.

The dual-energy formalism is a technique used to avoid these numerical issues. Under this technique we track an additional field called `"internal_energy"` that we frequently synchronize with with the `"total_energy"` field. Under these extreme conditions, then use the `"internal_energy"` field to provide extra precision for e.

In some sense the the dual-energy formalism can be considered an implementation detail of the hydro/MHD solver. However, the choice whether to use the dual-energy formalism has important implications for other methods and for initializers. This is the primary reason why the dual-energy formalism properties are tracked as part of the `fluid_props` physics object.

The relevant parameters are listed below:

Table 10: Method `fluid_props:dual_energy` parameters

Parameter	Type	Default	Description
type	*string*	`"disabled"`	Dual-energy formalism: `"disabled"`, `"bryan95"`, `"modern"`
eta	*list(float)*	-	Interpretation (and defaults) depend on value of *type*

When *type* is `"disabled"`, the simulation runs without the dual-energy formalism This is the default configuration and in this case, the *eta* parameter should **NOT** be specified.

The other two choices for *type* refer to two variants of the dual-energy formalism that we describe below.

"bryan95" variant

This is the original formulation of the dual-energy formalism described in Bryan et al (1995). It is used by the "ppm" solver and it is parameterized by two values: η_1 & η_2.

Physics:fluid_props:dual_energy:eta expects a list of 2 entries: $[\eta_1, \eta_2]$ (users are **NOT** permitted to provide a single entry). When this parameter isn't specified, it defaults to `[0.001, 0.1]`.

"modern" variant

This implementation is used by the "mhd_vlct" solver and it more closely resembles the implementation employed in Enzo's Runge–Kutta integrator than the "bryan95" variant. This variant is parameterized by a single value: η, and thus *Physics:fluid_props:dual_energy:eta* should only provide a single entry.

There are 3 primary differences from the "bryan95" variant:

1. the "internal_energy" field is always used to compute pressure. Under the bryan95 variant, pressure could be computed from "total_energy" or "internal_energy" (the decision was independent of synchronization).
2. pressure and "internal_energy" are not separately reconstructed. Instead, just the pressure is reconstructed. The "internal_energy" is computed at the left and right interfaces from the reconstructed quantities.
3. Synchronization of the total and internal energies is a local operation that doesn't require knowledge of cell neighbors. The "bryan95" variant requires knowledge of the immediate neighbors (each synchronization incremented the stale depth - so 3 extra ghost zones would have been required for the "mhd_vlct" solver).

For clarity, the conditions for synchronization are provided below. The specific internal_energy, e, is set to $e' = E - (v^2 + B^2/\rho)/2$ (where E is the specific total_energy) when the following conditions are met:

- $c_s'^2 > \eta v^2$, where $c_s'^2 = \gamma(\gamma - 1)e'$.
- $c_s'^2 > \eta B^2/\rho$ (this is always satisfied in hydro mode)
- $e' > e/2$

If the above conditions are not met, then total_energy is set to $e + (v^2 + B^2/\rho)/2$ in MHD mode (in hydro mode, it's set to $e + v^2/2$).

Note: in the future, the behavior described in difference 2, may change to achieve better compatibility with Grackle.

AMR Inconsistency

A minor inconsistency is present in our simulation when we use either dual-energy formalism in AMR simulations. Each formulation of the dual-energy formalism requires the hydro-solver to add a source term to the "internal_energy" field. This source term involves the velocity on each cell face (just the component normal to the face). In practice, we compute the velocity in the Riemann Solver.

When neighboring cells compute the source term, they should theoretically use consistent values for the velocity component on the shared face. While this is true for neighbors with the same refinement level, it's not true for neighbors with different refinement levels. In principle, a coarse cell should actually compute the source term from some kind of average of the velocity component on the faces of the neighboring finer cells (this could hypothetically be achieved with the equivalent of a "flux-correction").

We don't expect this inconsistency to be important given that the dual-energy formalism only affects highly supersonic flows and isn't conservative anyway.

EOS

The `"fluid_props:eos"` subgroup holds parameters that configure the nominal (caloric) equation of state. These parameters primarily affect the Hydro/MHD methods and the `"grackle"` method. It also affects the calculation of pressure and temperature fields.

At this time, all simulations are assumed to have an ideal EOS and the only configurable parameter is provided below. In the future, further EOS customization will be supported in this section

Table 11: Method `fluid_props:eos` parameters

Parameter	Type	Default	Description
gamma	*float*	`5.0/3.0`	Adiabatic index (a.k.a. the ratio of specific heats)

See *Compatability with hydro/mhd solvers* for further discussion about how the equation of state is handled when *Method:grackle:primordial_chemistry* exceeds 1 (under these conditions Grackle models a spatially varying adiabatic index).

Floors

The `"fluid_props:floors"` subsection is used for specifying the floors of different fluid quantities. A list of the quantities whose floors can be configured are provided below.

Table 12: Method `fluid_props:floors` parameters

Parameter	Type	Default	Description
density	*float*	-	The floor to apply to the `"density"` field.
pressure	*float*	-	The floor to apply to the `"pressure"` field.
temperature	*float*	-	The floor to apply to the `"temperature"` field.
metallicity	*float*	-	This multiplied by the `"density"` field and `enzo_constants::metallicity_solar` gives the floor for the `"metal_density"` field.

See *Enzo-E Methods* for discussions of the floors that are actually used by a given method. Be mindful that unlike the other parameters, the *metallicity* parameter doesn't directly specify the floor for a fluid field (the actual floor depends on other quantities).

Note: The `"pressure"` and `"temperature"` fields can be written to disk as derived quantities (if the fields are specified in the "derived" grouping). In these cases, these quantities are computed using `EnzoComputePressure` and `EnzoComputeTemperature`, respectively. You may want to check these classes to see if/when the floors get applied.

2.4.3 "gravity"

Specifies the gravitational constant. In the future, additional gravity-related parameters could be introduced.

Table 13: Physics `gravity` parameters

Parameter	Type	Default	Description
grav_const_codeU	*float*	-	The gravitational constant specified in code units. When not specified, it's automatically computed from the real-world reference value $G \approx 6.67 \times 10^{-8}\,\mathrm{cm}^3\,\mathrm{g}^{-1}\mathrm{s}^{-2}$ (see codebase for exact value).

In most cases, users should not specify *grav_const_codeU* at all (so that the appropriate default value is used). This parameter mostly exists to help simplify some test problems in non-cosmological simulations.

Users are **NOT** allowed to specify *grav_const_codeU* parameter in cosmological simulations. This is because cosmological code-units are defined such that $4\pi G\bar{\rho}$ has the value `1.0`, where $\bar{\rho}$ is the mean physical matter density of the universe.

We generally advise users to include `"gravity"` within *Physics:list* whenever they use any method involving gravity, event if they aren't explicitly assigning a value to *Physics:gravity:grav_const_codeU*.

Note: At the time of writing, if the user specifies both `"cosmology"` and `"gravity"` within *Physics:list*, it's important that `"cosmology"` comes first. In the future, we can hopefully relax these requirements.

2.5 Enzo-E initial conditions

[This page is under development]

Initial conditions define the initial setup of a problem. They include field values and particle values at the start of the simulation. They can be declared in the parameter file using Cello's "value" initializer, or specific problem initializers can be invoked, such as "implosion_2d" for the "Implosion test", or "sedov_array_3d" for a 3D array of Sedov blast waves.

`"accretion_test"`
: Initialize a sink particle with some mass, position, and velocity, in a background medium of constant density and pressure, possibly with a velocity directed towards the initial position of the sink particle.

`"bb_test"`
: Initialize a "BB Test" problem, following the setup described in Federrath et al 2010, ApJ, 713, 269.

`"cloud"`
: Initialize a spherical cloud embedded in a hot wind.

`"collapse"`
: Initialize a spherical collapse test.

`"file"`
: Initialize from input HDF5 file. Not implemented yet.

`"grackle_test"`
: Initialize for a grackle 2.0 chemistry and cooling library test (depreciated).

`"implosion_2d"`
: Initialize an "implosion" test problem.

"inclined_wave"
Initialize an inclined wave test problem. (Primarily used for testing the VL+CT MHD solver).

"m1_closure"
Initializer for M1 Closure radiative transfer. Initializes RT fields and reads eigenvalue table for the "HLL" flux function.

"merge_sinks_test"
Initialise sink particles with masses, positions, and velocities read from a text file specified in the parameter file.

"pm"
Initialize "dark" matter particles in either a regular uniform array with one particle per cell, or randomly following the "density" field distribution.

"ppml_test"
Initialize fields for the PPML solver for a high-pressure shpere in an anisotropic magnetic field.

"sedov"
Calls either "sedov_array_2d" or "sedov_array_3d" initializer depending on the problem rank.

"sedov_array_2d"

Initialize a regular 2D array of Sedov blast problems. Used for parallel-scaling studies without load balancing.

"sedov_array_3d"

Initialize a regular 3D array of Sedov blast problems. Used for parallel-scaling studies without load balancing.

"sedov_random" [Thomas Bolden]

Initialize a regular 3D array of Sedov blast problems. Used for parallel-scaling studies with load balancing.

"shock_tube"
Initialize an axis-aligned shock tube test problem (Primarily used for testing the VL+CT MHD solver).

"shu_collapse"
Initialize a Shu Collapse problem, following the setup described in Federrath et al 2010, ApJ, 713, 269.

"soup"

Similar to the "sedov" problem, but with letters instead of spheres.

"trace"

Initialize "trace" tracer particles in either a regular uniform array with one particle per cell, or randomly following the "density" field distribution.

"turbulence"

Initialize fields for driving turbulence, including "driving_[xyz]" fields.

"value"

Initialize fields using expressions directly from the parameter file.

> **Warning:** For technical reasons, "value" does not work reliably for multi-node problems.

"vlct_bfield"

Initialize the cell-centered magnetic fields for use by the VL + CT method. This initialization can be performed from expressions specified in the parameter file for each component of the magnetic vector potential (in this case, the face-centered magnetic fields are also initialized) OR from the face-centered

magnetic fields that have already been initialized by a separate initializer. Additionally, it provides the ability to update partially initialized `"total_energy"` fields with the specific magnetic energy computed from the newly computed cell-centered bfields and pre-initialized `"density"` fields.

2.6 Enzo-E Units

In non-cosmological simulations, the user is free to specify length, time, and either density or mass units (only one can be set). This is done by setting values for `Units:length`, `Units:time`, and either `Units:density` or `Units:mass`, which correspond to the unit length / time / density / mass in cgs units. If running with gravity, and if the user wants to use the standard value for the gravitational constant, the user must set a value for `Method:gravity:grav_const` which is consistent with their choice of units; i.e., its value must be $G_{cgs} \times M \times T^2 \times L^{-3}$, or equivalently, $G_{cgs} \times D \times T^2$, where M, D, T, L are the mass, density, time, and length units, and G_{cgs} is the value of the gravitational constant in cgs units.

In cosmological simulations, the code ignores any specified units and instead operates in a coordinate system which is comoving with the universal expansion, defining the length, time, velocity, and density units as given below (length and density units depend on time / redshift.)

The length unit is specified by `Physics:cosmology:comoving_box_size`, which gives the length unit in terms of comoving Mpc/h.

The density unit is defined so that the comoving mean matter density of the universe is 1, where the mean comoving matter density is given by $\frac{3H_0^2\Omega_m}{8\pi G}$.

The time unit is defined so that $\frac{3}{2}H_0^2\Omega_m(1+z_i)^3 = 1$, where z_i is the initial redshift of the simulation. This is the free-fall time at $z = z_i$, which has the effect of simplifying Poisson's equation.

The velocity unit is defined as $\frac{1+z_i}{1+z}L/T$, where L is the length unit, and T is the time unit.

For cosmological simulations, the value set for `Method:gravity:grav_const` is ignored.

In all simulations, the `"temperature"` field always has units of Kelvin.

2.7 Enzo-E Fields

[This page is under development]

Cello allows field data to be created and operated on in Blocks of an adaptive mesh hierarchy. Enzo-E defines specific fields, as well as groups of related fields. This page documents what fields are accessed by different Methods, recommended usage of Fields when writing Methods, and a reference of the Field API.

2.7.1 Using Fields in Methods

2.7.2 Field API

Method
Field::Field(FieldDescr *, FieldData *)

Summary
Create a Field object given a Field descriptor and a Field Data object

Return
none

This constructor creates a new Field object given a FieldDescr (field descriptor) and FieldData (field data) object.

Method
Field::Field(FieldDescr *, FieldData *)

Method
Field(const Field & field)

Summary
Copy constructor

Return
none

Method
Field::operator= (const Field & field)

Summary
Assignment operator

Return
Field &

Method
Field::~Field()

Summary
Destructor

Return
none

Method
Field::pup (PUP::er &p)

Summary
CHARM++ Pack / Unpack function

Return
void

Method
Field::field_descr()

Summary
Return the field descriptor for this field

Return
FieldDescr *

Method
Field::field_data()

Summary
Return the field data for this field

Return
FieldData *

Method
Field::set_field_descr(FieldDescr * field_descr)

Summary
Set the field descriptor for the Field object

Return
void

Method
Field::set_field_data(FieldData * field_data)

Summary
Set the field data object for the Field object

Return
void

Field Descriptor

Method
Field::set_alignment(int alignment)

Summary
Set alignment

Return
void

Method
Field::set_padding(int padding)

Summary
Set padding

Return
void

Method
Field::set_centering(int id, int cx, int cy=0, int cz=0)

Summary
Set centering for a field

Return
void

Method
Field::set_ghost_depth(int id, int gx, int gy=0, int gz=0)

Summary
Set ghost_depth for a field

Return
void

Method
Field::set_precision(int id, int precision)

Summary
Set precision for a field

Return
void

Method
Field::insert_permanent(const std::string & name)

Summary
Insert a new field

Return
int

Method
Field::insert_temporary(const std::string & name = “”)

Summary
Insert a new field

Return
int

Method
Field::field_count() const

Summary
Return the number of fields

Return
int

Method
Field::field_name(int id) const

Summary
Return name of the ith field

Return
std::string

Method
Field::is_field(const std::string & name) const

Summary
Return whether the field has been inserted

Return
bool

Method
Field::field_id(const std::string & name) const

Summary
Return the integer handle for the named field

Return
int

Properties

Method
Field::groups()

Summary
Return the grouping object for Fields

Return
Grouping *

Method
alignment() const

Summary
alignment in bytes of fields in memory

Return
int

Method
padding() const

Summary
padding in bytes between fields in memory

Return
int

Method
centering(int id, int * cx, int * cy = 0, int * cz = 0) const

Summary
centering of given field

Return
void

Method
is_centered(int id) const

Summary
return whether the field variable is centered in the cell

Return
bool

Method
ghost_depth(int id, int * gx, int * gy = 0, int * gz = 0) const

Summary
depth of ghost zones of given field

Return
void

Method
precision(int id) const

Summary
Return precision of given field

Return
int

Method
bytes_per_element(int id) const

Summary
Number of bytes per element required by the given field

Return
int

Method
is_permanent (int id_field) const

Summary
Whether the field is permanent or temporary

Return
bool

Method
num_permanent() const

Summary
Return the number of permanent fields

Return
int

History

Method
set_history (int num_history)

Summary
Set the history depth for storing old field values

Return
void

Method
num_history () const

Summary
Return the number of history generations to store

Return
int

Method
save_history (double time)

Summary
Copy "current" fields to history

Return
void

Copy "current" fields to history = 1 fields (saving time), and push back older generations up to num_history()

Method
history_time (int ih) const

Summary
Return time for given history

Return
double

Units

Method
units_scale_cgs (int id, double amount)

Summary
scale the field to cgs units given the unit scaling factor

Return
void

if it's already in cgs, then leave as-is except if it's in cgs but the scaling factor has changed (e.g. due to expansion) then adjust for the new scaling factor

---

Method
units_scale_code (int id, double amount)

Summary
convert the field to "code units" given the unit scaling factor

Return
void

if it's already in code units, leave it as-is warning if scaling factor has changed.

Method
units_scaling (const FieldDescr *, int id)

Summary
Return the current scaling factor of the given Field

Return
double

1.0 if in code units, or the scaling factor if in cgs

FieldData

Method
size(int * nx, int * ny = 0, int * nz = 0) const

Summary
Return size of fields on the data, assuming centered

Return
void

Method
dimensions(int id_field,int * mx, int * my = 0, int * mz = 0) const

Summary
Return dimensions of fields on the data, assuming centered

Return
void

Method
values (int id_field, int index_history=0)

Method
values (std::string name, int index_history=0)

Summary
Return full array of values for the corresponding field

Return
char *

Return array for the corresponding field, which may or may not contain ghosts depending on if they're allocated

---

Method
unknowns (int id_field, int index_history=0)

Method
unknowns (std::string name, int index_history=0)

Summary
Return array for the corresponding field

Return
char *

Return array for the corresponding field, which does not contain ghosts whether they're allocated or not

---

Method
permanent () const

Summary
Return the array of all permanent fields

Return
const char *

Method
cell_width(double xm, double xp, double * hx, double ym=0, double yp=0, double * hy=0, double zm=0, double zp=0, double * hz=0) const

Summary
Return width of cells along each dimension

Return
void

Method
clear (float value = 0.0, int id_first = -1, int id_last = -1)

Summary
Clear specified array(s) to specified value

Return
void

Method
permanent_allocated() const

Summary
Return whether array is allocated or not

Return
bool

Method
permanent_size() const

Summary
Return whether array is allocated or not

Return
size_t

Method
allocate_permanent(bool ghosts_allocated = false)

Summary
Allocate storage for the field data

Return
void

Method
allocate_temporary(int id)

Summary
Allocate storage for the temporary fields

Return
void

Method
deallocate_temporary(int id)

Summary
Deallocate storage for the temporary fields

Return
void

Method
reallocate_permanent(bool ghosts_allocated = false)

Summary
Reallocate storage for the field data

Return
void

Reallocate storage for the field data, e.g. when changing from ghosts to non-ghosts [costly for large blocks]

Method
deallocate_permanent()

Summary
Deallocate storage for the field data

Return
void

Method
ghosts_allocated() const

Summary
Return whether ghost cells are allocated or not.

Return
bool

Method
field_size (int id, int *nx=0, int *ny=0, int *nz=0) const

Summary
Return the number of elements (nx,ny,nz) along each axis

Return
int

Return the number of elements (nx,ny,nz) along each axis, and total number of bytes n

Debugging

Method
print (const char * message, bool use_file = false) const

Summary
Print basic field characteristics for debugging

Return
void

2.7.3 Enzo-E Fields

- acceleration_x
- acceleration_y
- acceleration_z
- B
- bfieldx
- bfieldx_rx
- bfieldx_ry
- bfieldx_rz
- bfieldy
- bfieldy_rx
- bfieldy_ry
- bfieldy_rz
- bfieldz
- bfieldz_rx
- bfieldz_ry

- bfieldz_rz
- cooling_time
- density
- density_total
- dens_rx
- dens_ry
- dens_rz
- DI_density
- DII_density
- driving_x
- driving_y
- driving_z
- e_density
- gamma
- H2I_density
- H2II_density
- HDI_density
- HeI_density
- HeII_density
- HeIII_density
- HI_density
- HII_density
- HM_density
- internal_energy
- metal_density
- potential
- pressure
- species_De
- species_DI
- species_DII
- species_H2I
- species_H2II
- species_HDI
- species_HeI
- species_HeII
- species_HeIII

- species_HI
- species_HII
- species_HM
- temperature
- total_energy
- velocity_x
- velocity_y
- velocity_z
- velox
- velox_rx
- velox_ry
- velox_rz
- veloy
- veloy_rx
- veloy_ry
- veloy_rz
- veloz
- veloz_rx
- veloz_ry
- veloz_rz
- X

2.8 Enzo-E Particles

[This page is under development]

This section describes all particle types in Enzo-E, including particle attributes, where each particle type is used and modified, and groups of related particles.

- dark
- trace
- star
- sink

IMPORTANT: If running a simulation with particles that have mass, and you want to analyse the output with yt, you must include a "mass_is_mass" parameter in the "Particle" group in the input parameter file. It can be set equal to anything, so long as it can be read by yt. It is recommended that the value is set to be "true".

In addition, you will need to install the latest development version of yt from source. This can be done with:

git clone https://github.com/yt-project/yt.git

cd yt

pip install -e .

2.9 Checkpoint / restart

Enzo-E can perform checkpoint dumps to save the current state, and can read them at initialization to continue a simulation where it left off. Setting up checkpointing involves including methods in the *Method* parameter group, and provided a schedule for them. Restarting requires adding a small number of parameters to a parameter file to turn on restart, and where the checkpoint files are located.

2.9.1 Checkpoint

```
Method {
  ...
  list = [ "order_morton", "check", ... ];
  order_morton {
     schedule { var = "cycle";  start = 5;  step = 5;  }
  }
  check {
     schedule { var = "cycle";  start = 5;  step = 5;  }
     dir = [ "Check-%02d", "cycle" ];
     num_files = 4;
     ordering = "order_morton";
  }
  ...
}
```

This example writes checkpoint dumps every 5 cycles, starting with the 5th cycle.

See the `input/Checkpoint/test_cosmo-checkpoint.in` parameter file for a working example of writing checkpoint directories.

Note: Currently, there is a restriction that the domain blocking must be square (2D) or cubical (3D), have a power-of-two blocking along the axes, and the `Adapt:min_level` parameter must be set such that the coarsest (negative) level is a single block. For example, if `Mesh:root_blocks = [ 16, 16, 16]`, then `Adapt:min_level` must be initialized to 4 ($= log_2 16$).

2.9.2 Restart

To restart from a checkpoint directory, edit a copy of the original parameter file to add two parameters, `Initial:restart` and `Initial:restart_dir`. For example, to restart the problem in the previous "Checkpoint" section at cycle 50, one would add the following:

```
Initial {

   restart = true;
   restart_dir = "Check-50";
}
```

Note that it's also possible to rerun with some parameters modified, for example different linear solver convergence criteria or different mesh refinement criteria. The main limitation is that the simulation data cannot change on restart: you cannot add new fields, new particle types or attributes, or change the number of root-level blocks or block sizes. You *can* restart with more (or fewer) processors, though the constraint that the number of processors must be at most the number of root-level blocks must still be satisfied.

The parameter file `input/Checkpoint/test_cosmo-restart.in` is the "restart" counterpoint to the "checkpoint" example mentioned in the previous section.

CHAPTER

THREE

REFERENCE GUIDE

3.1 Enzo-E / Cello Parameters

This page documents all current parameters implemented in Enzo-E / Cello. Each parameter is summarized, its type or types are listed, and the default value (if any) is provided. The scope of the parameter is also listed, which is either "Cello" or "Enzo", depending on whether the parameter is associated with Cello framework or Enzo-E application. Any assumptions associated with a parameter are also listed; for example, a parameter may only be valid if some other parameter is set to a certain value.

If you find any errors in the documentation, or have any specific suggestions, please contact the Enzo Project developers at github.

3.1.1 Adapt

Adapt parameters define how the mesh hierarchy dynamically adapts to the solution. It is closely related to the Mesh parameters, which defines the root grid size, number of blocks in the root grid, and size of blocks.

Parameter
Adapt : interval

Summary
Number of cycles between adapt steps

Type
integer

Default
1

Scope
Cello

The interval parameter is used to set the number of root-level cycles between mesh adaptation. The default is 1.

Parameter
Adapt : max_level

Summary
Maximum level in the adaptive mesh hierarchy

Type
integer

Default
0

Scope
Cello

This parameter specifies the level of the most highly refined Block in the mesh hierarchy. The default is 0, meaning there is no refinement past the initial root-level grid.

Parameter
Adapt : min_level

Summary
Minimum level in the adaptive mesh hierarchy

Type
integer

Default
0

Scope
Cello

This parameter specifies the coarsest level of "sub-root" Blocks, and must non-positive. This is used primarily for multigrid methods, such as in the "mg0" solver. The default is 0, meaning no sub-root Blocks are created. If multigrid is used, then both *Adapt:min_level* and Method:<mg-solver>:min_level must be set..

Parameter
Adapt : list

Summary
List of refinement criteria

Type
list (string)

Default
[]

Scope
Cello

List of mesh refinement criteria, each of which has its own associated Adapt:<criteria> parameters. When multiple criteria are used, if all refinement criteria evaluate to "coarsen", then the block will be tagged to coarsen; if any refinement criteria evaluate as "refine", then the block will be tagged to refine. (Note that a particular block will coarsen only if it and all other sibling blocks are tagged to coarsen as well.)

The items in the list need not be the same as the (required) *Adapt:<criterion>:type* parameter; they are solely used to identify and distinguish between different criteria in the simulation. This allows the user to use multiple criteria of the same type but with different parameters, e.g. "mask" with different masks:

```
Adapt {
   list = ["criterion_1", "criterion_2"];
   criterion_1 {
      type = "shock";
   }
   criterion_2 {
      type = "shear";
   }
}
```

Parameter
Adapt : min_face_rank

Summary
Minimum rank of Block faces to check for 2:1 refinement restriction

Type
integer

Default
0

Scope
Cello

Many numerical methods require a 2:1 refinement restriction on adaptive meshes, such that no Block in level i is adjacent to another Block in a level j with |i - j|>1. This assumption may be required across corners and edges as well as 2D faces. This parameter specifies the minimum rank (dimensionality) of Block faces across which to enforce the 2:1 refinement restriction.

Parameter
Adapt : <criterion> : field_list

Summary
List of field the refinement criterion is applied to

Type
[string | list (string)]

Default
[] (all fields)

Scope
Cello

This parameter specifies the fields that the refinement criteria is applied to. For example, if type = "slope" and field_list = ["density"], then the "refine by slope" refinement criterion is applied to the density field.

Parameter
Adapt : <criterion> : level_exponent

Summary
Level exponent parameter

Type
float

Default
0.0

Scope
Cello

Assumes
<criterion> is of type "mass"

The level exponent parameter is used in the "mass" refinement criterion type only. It is used as a scaling factor for the refinement criteria for different mesh levels.

Parameter
Adapt : <criterion> : max_coarsen

Summary
Cutoff value for coarsening a block

Type
[float | list (float)]

Default
0.5*min_refine

Scope
Cello

A block may coarsen if the refinement criterion applied to the block is smaller than this value everywhere in the block. A list is used for the "shock" refinement criterion type, in which case the first value is for pressure and the second is for the energy ratio.

Parameter
Adapt : <criterion> : include_ghosts

Summary
Whether to include ghost zones when applying the refinement criterion

Type
logical

Default
false

Scope
Cello

When applying a mesh refinement criterion, this parameter specifies whether to apply it to ghost zones in the block as well as non-ghost zones.

Parameter
Adapt : <criterion> : min_refine

Summary
Cutoff value for refining a block

Type
[float | list (float)]

Default
0.3

Scope
Cello

A block must refine if the refinement criterion applied to the block is larger than this value anywhere in the block. A list is used for the "shock" refinement criterion type, in which case the first value is for pressure and the second is for the energy ratio.

Parameter
Adapt : <criterion> : output

Summary
Name of a field in which to store the result of the refinement criterion

Type
string

Default
""

Scope
Cello

In addition to evolved field values, one may also output the refinement criteria. This may be useful for example for debugging or for finding appropriate values for :p:`max_coarsen and min_refine. A value of -1 specifies coarsening, +1 for refining, and 0 for staying the same.`

Parameter
Adapt : <criterion> : max_level

Summary
Maximum level to refine using this refinement criterion

Type
integer

Default
max (integer)

Scope
Cello

Adapt will not refine past max_level when using this refinement criterion. Note if the global Adapt:max_level is smaller, than that takes precidence; also, another criterion may refine past this if both Adapt:max_level and Adapt : <criterion> : max_level for the other criterion are both larger.

Parameter
Adapt : <criterion> : type

Summary
Type of mesh refinement criteria

Type
string

Default
"unknown"

Scope
Cello

Type of mesh refinement criteria. This is a required parameter, and must be one of "slope", "shear", "mask", "mass", "density", "shock", "particle_mass", or "particle_count".

3.1.2 Balance

Parameters for controlling dynamic load balancing are enclosed within the Balance group. Currently only one Balance parameter is available, which is used to control how frequently load balancing is performed.

Parameter
Balance : schedule

Summary
Scheduling parameters for dynamic load balancing

Type
subgroup

Default
none

Scope
Cello

See the *schedule* subgroup for parameters used to define when to trigger the dynamic load balancing operation.

3.1.3 Boundary

Boundary group parameters define boundary conditions. For simple (non-mixed) boundary conditions, only the type parameter is required, e.g. Boundary { type = "periodic"; }. For more complicated boundary conditions, the list parameter is used to define Boundary subgroups, where each subgroup specifies boundary conditions for some subset of the domain. The axis and face parameters are available to restrict boundary conditions to a subset of faces, whereas the mask parameter is available for even finer control of mixed boundary conditions, which may be time-dependent. Inflow boundary conditions use the value parameter/subgroup to specify field values at the boundary.

Parameter
Boundary : list

Summary
List of boundary condition subgroups

Type
[string | list (string)]

Default
[]

Scope
Cello

For mixed boundary conditions, the list parameter specifies the list of names of subgroups that define boundary conditions on each portion of the domain boundary. Boundary conditions in each subgroup are applied in the order listed. In the example below, two subgroups "one" and "two" are defined, which specify reflecting boundary conditions along the x-axis and outflow boundary conditions along the y-axis:

```
Boundary {
   list = ["one", "two"];
   one {
      type = "reflecting";
      axis = "x";
   }
   two {
      type = "outflow";
      axis = "y";
   }
}
```

Parameter
Boundary : <condition> : type

Summary
Type of boundary condition

Type
string

Default
"undefined"

Scope
Cello

Boundary conditions in Enzo-E include "reflecting" , "outflow" , "inflow" , and "periodic". Other boundary condition types can be implemented by either a) modifying the existing EnzoBoundary class or b) creating a new class inherited from the Boundary base class.

"inflow" boundary conditions additionally require the *value* parameter or subgroup and, in certain cases, the *field_list* parameter.

Parameter
Boundary : <condition> : axis

Summary
Axis along which boundary conditions are to be enforced

Type
string

Default
"all"

Scope
Cello

The axis parameter restricts the boundary conditions to the face orthogonal to the specified axis. axis must be "x" , "y" , "z" or "all". The axis parameter may be used in conjunction with the face parameter, or by itself.

Parameter
Boundary : <condition> : face

Summary
Face along which boundary conditions are to be enforced

Type
string

Default
"all"

Scope
Cello

The face parameter can restrict the boundary conditions to be applied only to the upper or lower faces. face orthogonal to the given face. face must be "upper" , "lower" or "all". The face parameter may be used in conjunction with the axis parameter, or by itself.

Parameter
Boundary : <condition> : mask

Summary
Subregion in which boundary conditions are to be enforced

Type
logical-expr

Default
none

Scope
Cello

The mask parameter specifies the subregion of the boundary on which to apply the boundary conditions. The logical expression may be a function of x, y, z, and t, and boundary conditions are restricted to where (and when) it evaluates to true:

```
Boundary {
   ...
   OUT {
      type = "outflow";
      mask = (x >= 4.0) ||
             (y >= 1.0 && (x >= 0.744017 + 11.547* t));
   }
}
```

Parameter
Boundary : <condition> : value

Summary
Value for inflow boundary conditions

Type
float

Type
float-expr

Type
list (float-expr [, logical-expr, float-expr [, ...]])

Default
[]

Scope
Cello

For inflow type boundary conditions, there are 2 ways to specify field values. In both cases, the parameter(s) are assigned "value-expressions", which may be of type float, float-expr, or a list of alternating float-expr and logical-expr types. Both float-expr and logical-expr may be functions of x, y, z, and t. When a list is specified, the logical-expr is treated as a mask, similar to an 'if-then-else' clause.

The two approaches include:

1. Under the historic approach, Boundary:<condition>:value is a parameter that is directly assigned a "value-expression". This "value-expression" specifies the value enforced by this boundary for all fields in the *field_list* parameter.

```
Boundary {
   ...
   VELOCITY_Y {
      type = "inflow";
      field_list = "velocity_y";
      value = [ -8.25*0.5,
                  ((x <= 0.166667) && (y <= 0.0) ) ||
                   (x <= 0.0) ||
                   ((x < 0.744017 + 11.547*t) && (y >= 1.0)),
                  0.0
               ];
   }
}
```

2. Under the second approach, Boundary:<condition>:value is a parameter-subgroup. Within the parameter-subgroup, "value-expressions" are assigned to parameters that are named after the fields that the "value-expressions" apply to. Under this approach, it is an error for the *field_list* parameter to be specified. This approach bears a lot of similarities to the way values are specified in the *"value" problem initializer.*

```
Boundary {
   ...
   x_upstream {
      type = "inflow";
      axis = "x";
      face = "lower";
      value {
         density    = [ 0.1, (y <= 0.5), 0.1 * (y - 0.5) ];
         velocity_x = 10.0;
         velocity_y = 0.0;
         velocity_z = 0.0;
         total_energy = 75.0;
      }
   }
}
```

Parameter
Boundary : <condition> : field_list

Summary
List of fields to apply boundary conditions to

Type
list (string)

Default
[]

Scope
Cello

The field_list parameter is used to restrict boundary conditions to the specified fields. An empty list, which is the default, is used to specify all fields.

3.1.4 Domain

Domain parameters specify the lower and upper extents of the computational domain, using the lower and upper parameters.

Parameter
Domain : lower

Summary
Lower domain extent

Type
list (float)

Default
[0.0, 0.0, 0.0]

Scope
Cello

Lower extent of the computational domain, [x_{min}], [x_{min}, y_{min}], or [x_{min}, y_{min}, z_{min}].

Parameter
Domain : upper

Summary
Upper domain extent

Type
list (float)

Default
[1.0, 1.0, 1.0]

Scope
Cello

Upper extent of the computational domain, [x_{max}], [x_{max}, y_{max}], or [x_{max}, y_{max}, z_{max}].

3.1.5 Field

Fields and their properties are defined using the Field parameter group. All fields must be explicitly defined using the list Field parameter, and must match the names expected by the respective Methods. Properties include the number of ghost zones, precision, and whether a field is centered or lies on some face, edge, or corner. Some performance-related parameters are available as well, including alignment in memory, and memory padding between fields.

Parameter
Field : list

Summary
List of fields

Type
list (string)

Default
[]

Scope
Cello

All fields must be explicitly listed in the *list* parameter. Field names depend on the Method(s) used; e.g., PPM uses "density", "velocity_x", "velocity_y", "total_energy", and "internal_energy".

Parameter
Field : gamma

Summary
Adiabatic exponent

Type
float

Default
5.0 / 3.0

Scope
Enzo

gamma specifies the ratio of specific heats for the ideal gas used by the hydrodynamics solvers.

> **Warning:** This parameter is deprecated and will be removed in a future version. Going forward, pass this value to *Physics:fluid_props:eos:gamma*.

Parameter
Field : alignment

Summary
Force field data on each block to start on alignment bytes

Type
integer

Default
8

Scope
Cello

Depending on the computer architecture, variables can be accessed from memory faster if they have at least 4-byte or 8-byte alignment. This parameter forces each field block array to have an address evenly divisible by the specified number of bytes.

Parameter
Field : <field> : centering

Summary
Specify the position of the given field variable within the computational cell.

Type
list (logical)

Default
[true, true, true]

Scope
Cello

By default, variables are centered within a computational cell. Some methods expect some variable, e.g. velocity components, to be positioned on a cell face. The effect of this parameter is to increase the dimension of the field block by one along each axis with a value of "false". Numerical method implementations like PPML that assume (NX,NY,NZ) sized blocks even for offset variables, as opposed to e.g. (NX+1,NY,NZ), should still define the variable as centered.

Parameter
Field : <field> : group_list

Summary
Specify a list of groups that the Field belongs to

Type
list (string)

Default
[ ]

Scope
Cello

Different Fields may belong to any number of different "groups". For example, Enzo uses "color fields", which Enzo-E implements as defining color fields to belong to the group "color".

Parameter
Field : ghost_depth

Summary
Field ghost zone depths

Type
[integer | list (integer)]

Default
[0, 0, 0]

Scope
Cello

The default storage patch / block ghost zone depths [gx, gy, gz] along each axis for fields. If an integer, then the same ghost zone depth is used for each axis. Currently this value needs to be 4 for PPM when AMR is used.

Parameter
Field : padding

Summary
Add padding of the specified number of bytes between fields on each block.

Type
integer

Default
0

Scope
Cello

If block sizes are large and a power of two, and if the computer's cache has low associativity, performance can suffer due to cache thrashing. This can be avoided by introducing padding between fields. A value of twice the cache line width is recommended. Since field blocks are usually small, this should not usually be an issue.

Parameter
Field : precision

Summary
Default field precision

Type
string

Default
"default"

Scope
Cello

Default precision for all fields. Supported precisions include "single" (32-bit) and "double" (64-bit). "quadruple" is accepted, but not implemented by most numerical methods (e.g. PPM). "default" is for compatibility with Enzo, and corresponds to either "single" or "double" depending on the CELLO_PREC configuration flag setting. This precision parameter must not conflict with the CELLO_PREC setting.

Parameter
Field : prolong

Summary
Type of prolongation (interpolation)

Type
string

Default
"linear"

Scope
Cello

For adaptive mesh refinement, field values may need to be transferred from coarser to finer blocks, either from coarse neighbor blocks in the refresh phase, or to fine child blocks during refinement in the adapt phase. Valid values include "linear" ; other values accepted but not implemented include "enzo" and "MC1" :e:` ; which are unfinished implementations of Enzo's` "InterpolationMethod" functionality.

Parameter
Field : restrict

Summary
Type of restriction (coarsening)

Type
string

Default
"linear"

Scope
Cello

For adaptive mesh refinement, field values may need to be transferred from finer to coarser blocks, either from fine neighbor blocks in the refresh phase, or to the parent block during coarsening in the adapt phase. Valid values include "linear" ; ;other values accepted but not implemented include "enzo".

Parameter
Field : history

Summary
How many generations of "old" fields to maintain

Type
integer

Default
0

Scope
Cello

Many problems may require field values from the previous timestep, e.g. for flux-correction, updating particles, etc. Cello supports this by allowing one or more generations of all fields to be stored and maintained. The default is 0, though 1 may be fairly common, and even more generations are supported if needed.

3.1.6 Group

The Group parameter group is used to differentiate between different types of Field's and Particles. For example, field groups may include "color" and "temporary", and particle groups may include "dark_matter" and "star".

```
Group {

   list = ["color", "temporary"];

   color {
      field_list = ["species_HI", "species_HII" ];
   }

   temporary {
      field_list = ["pressure", "temperature"];
   }

}
```

Field and Particle groups can analogously be defined in the respective Field and Particle parameter groups:

```
Field {

   list = ["density", "velocity_x", "species_HI"];

   species_HI {

      group_list = ["temporary"];

   }

}
```

Groups allow Cello applications to differentiate between these different types of fields and particles using the `Grouping` class (see `src/Cello/data_Grouping.?pp`).

Parameter
Group : list

Summary
List of groups

Type
list (string)

Default
[]

Scope
Cello

This parameter defines all groups.

Parameter
Group : <group> : field_list

Summary
List of fields belonging to the group

Type
list (string)

Default
[]

Scope
Cello

This parameter is used to assign fields to a given group.

Parameter
Group : <group> : particle_list

Summary
List of particle types belonging to the group

Type
list (string)

Default
[]

Scope
Cello

This parameter is used to assign particle groups to a given group.

3.1.7 Initial

The Initial group is used to specify initial conditions. cycle specifies the initial cycle number (usually 0), list specifies a list of initial conditions, which may include `"value"` for initializing fields directly, or other problem-specific initial condition generators.

Parameter
Initial : list

Summary
Sequence of initializers to apply.

Type
list (string)

Default
none

Scope
Cello

This parameter specifies the list of initializers to apply. Each initializer in the list is applied in the order specified. Not all initializers are meant to be used alongside other initializers. Possible initializers include:

- "accretion_test" Setup a test problem with a sink particle.
- "bb_test" Initialize a "BB Test" problem, following the setup described in Federrath et al 2010, ApJ, 713, 269.
- "cloud" Initialize a spherical cloud embedded in a hot wind.

- "collapse" Initialize a spherical collapse test.
- "grackle_test" Initialize a grackle chemistry and cooling test.
- "implosion_2d" Initialize an "implosion" test problem.
- "inclined_wave" Initialize an inclined wave test problem.
- "merge_sinks_test" Initialise sink particles with masses, positions, and velocities read from a text file specified in the parameter file.
- "pm" Initialize "dark" matter particles in either a regular uniform array with one particle per cell, or randomly following the "density" field distribution.
- "ppml_test" Initialize fields for the PPML solver for a high-pressure sphere in an anisotropic magnetic field.
- "sedov" Sets up sedov blast problems.
- "shock_tube" Initialize an axis-aligned shock tube test problem
- "shu_collapse" Initialize a Shu Collapse problem, following the setup described in Federrath et al 2010, ApJ, 713, 269.
- "soup" Similar to the "sedov" problem, but with letters instead of spheres.
- "trace" Initialize "trace" tracer particles in either a regular uniform array with one particle per cell, or randomly following the "density" field distribution.
- "turbulence" Initialize fields for driving turbulence, including "driving_[xyz]" fields.
- "value" Initialize fields using expressions directly from the parameter file.
- "vlct_bfield" Initialize the cell-centered magnetic fields for use by the VL + CT method. For more details, see *vlct_bfield*

Parameters specific to individual initializers are specified in subgroups.

Parameter
Initial : cycle

Summary
Initial cycle number

Type
list (integer)

Default
0

Scope
Cello

Initial value for the cycle number.

Parameter
Initial : time

Summary
Initial time

Type
float

Default
0.0

Scope
Cello

Initial time in code units.

Parameter
Initial : restart

Summary
Set to true to restart from a checkpoint directory created by the "check" method

Type
bool

Default
false

Scope
Cello

Restart the simulation to continue a previous run from a saved checkpoint. If true, the restart directory must be specified using the "restart_dir" parameter.

Parameter
Initial : restart_dir

Summary
Directory containing restart files from a checkpoint created using the "check" method

Type
string

Default
none

Scope
Cello

When restarting a simulation from a "check" checkpoint directory, this parameter must specify the path to that directory.

---

value

Parameter
Initial : value : <field>

Summary
Initialize field values

Type
list (float-expr, [logical-expr, float-expr, [. . .]])

Default
[]

Scope
Cello

This initialization approach allows initializing field values directly. The first element of the list must be a float expression, and may include arithmetic operators, variables "x", "y", "z", and most functions in the POSIX math library /include/math.h. The second optional list element is a logical expression, and serves as a "mask" of the domain. The third float expression parameter is required if a mask is supplied, and serves as the "else" case. Multiple such mask-value pairs may be used. For example:

```
Initial {

   list = ["value"];

   value {
      density = [ sin ( x + y ), x - y < 0.0, 1.0 ];
   }
}
```

is read as "Set the density field equal to sin (x + y) wherever x - y < 0.0 , otherwise set to 1.0 ".

accretion_test

The accretion_test Initial subgroup is used to initialize an accretion test problem. In particular, it initializes a single sink particle with a given initial mass, position, and velocity, in a background medium with constant density and pressure, and (possibly) an initial velocity field with constant magnitude, directed towards the sink particle's initial position. Running this test problem requires the use of the "mhd_vlct", "pm_update", "merge_sinks", and "accretion" methods. One can then test the accretion method by checking whether mass and momentum are conserved in this problem.

This initializer requires periodic boundary conditions, and three spatial dimensions.

Note: sink particles must have a `"mass"` attribute if this initializer is used.

Parameter
Initial : accretion : sink_mass

Summary
The initial mass of the sink particle.

Type
float

Default
0.0

Scope
Enzo

The initial mass of the sink particle

Parameter
Initial : accretion : sink_position

Summary
The initial position of the sink particle.

Type
list (float)

Default
[0.0,0.0,0.0]

Scope
Enzo

The initial (3D) position of the sink particle

Parameter
Initial : accretion : sink_velocity

Summary
The initial velocity of the sink particle.

Type
list (float)

Default
[0.0,0.0,0.0]

Scope
Enzo

The initial (3D) velocity of the sink particle

Parameter
Initial : accretion : gas_density

Summary
The initial uniform density of the gas.

Type
float

Default
1.0e-6

Scope
Enzo

The initial uniform density of the gas.

Parameter
Initial : accretion : gas_pressure

Summary
The initial uniform pressure of the gas.

Type
float

Default
1.0e-6

Scope
Enzo

The initial uniform pressure of the gas.

Parameter
Initial : accretion : gas_radial_velocity

Summary
The (inwards) radial velocity of the gas, with respect to the initial sink particle position.

Type
float

Default
0.0

Scope
Enzo

The gas velocity in every cell with have magnitude equal to the value of this parameter, directed towards the initial sink particle position.

---

bb_test

The bb_test Initial subgroup is used to initialize a "BB Test" problem, as described in Federrath et al 2010, ApJ, 713, 269. In particular, it initializes an isothermal sphere of gas with, with the gas having a constant (small) "external density" outside of the truncation radius. Within the truncation radius, the gas density has the following form:

$\rho(\phi) = \rho_0(1 + A\cos(2\phi)),$

where ρ is the gas density, ϕ is the azimuthal angle in the spherical polar coordinate system, ρ_0 is the mean density and A is the (small) fluctuation amplitude. The gas rotates around the z-axis as a solid-body, with an optional additional uniform "drift velocity".

Running this test problem requires the use of the `"pm_deposit"`, `"gravity"`, `"mhd_vlct"`, `"sink_maker"`, and `"pm_update"` methods.

This initializer requires periodic boundary conditions, three spatial dimensions, and that the gas adiabatic index (`"Field:gamma"`) is between 1.0 and 1.000001.

The following fields are required, and must be specified in the input parameter file: `"density"`, `"density_total"`, `"density_particle"`, `"density_particle_accumulate"`, `"density_gas"`, `"particle_mass"`, `"potential"`, `"potential_temp"`, `"potential_copy"`, `"total_energy"`, `"pressure"`, `"acceleration_x"`, `"acceleration_y"`, `"acceleration_z"`, `"velocity_x"`, `"velocity_y"`, `"velocity_z"`, `"X"`, `"X_copy"`, `"B"`, `"B_copy"`, `"density_source"`, `"density_source_accumulate"`, `"mom_dens_x_source"`, `"mom_dens_x_source_accumulate"`, `"mom_dens_y_source"`, `"mom_dens_y_source_accumulate"`, `"mom_dens_z_source"`, and `"mom_dens_z_source_accumulate"`.

Sink particles must have the following attributes: `"mass"`, `"x"`, `"y"`, `"z"`, `"vx"`, `"vy"`, `"vz"`, and `"is_copy"`. These must all be of type `"default"`, except for `"is_copy"`, which should be of type `"int64"`.

Sink particles must be in the `"is_gravitating"` particle group.

Parameter
Initial : bb_test : center

Summary
The coordinates of the center of the sphere.

Type
list (float)

Default
[0.0,0.0,0.0]

Scope
Enzo

The coordinates of the center of the sphere.

Parameter
Initial : bb_test : drift_velocity

Summary
The initial additional uniform velocity of the gas.

Type
list (float)

Default
[0.0,0.0,0.0]

Scope
Enzo

The initial additional uniform velocity of the gas. Inside the truncation radius, the initial gas velocity will be determined by solid body rotation, plus this additional velocity. Outside the truncation radius, the gas velocity is just this velocity.

Parameter
Initial : bb_test : truncation_radius

Summary
The truncation radius of the sphere.

Type
float

Default
1.0

Scope
Enzo

The truncation radius of the isothermal sphere. Within this radius, the gas density is determined by the azimuthal angle; outside of this radius, it is equal to "external_density".

Parameter
Initial : bb_test : nominal_sound_speed

Summary
The nominal uniform sound speed of the gas.

Type
float

Default
1.0

Scope
Enzo

Nominal uniform sound speed of the gas used to initialise the total specific energy. In practice the actual sound speed will be different since the adiabatic index is not exactly 1.

Parameter
Initial : bb_test : fluctuation_amplitude

Summary
The amplitude of the density fluctuation which determines the initial gas density.

Type
float

Default
0.0

Scope
Enzo

Within the truncation radius, the gas density has the following form:

$\rho(\phi) = \rho_0(1 + A\cos(2\phi)),$

where ρ is the gas density, ϕ is the azimuthal angle in the spherical polar coordinate system, ρ_0 is the mean density and A is the fluctuation amplitude.

Parameter
Initial : bb_test : external_density

Summary
The gas density outside of the truncation radius.

Type
float

Default
1.0e-6

Scope
Enzo

The gas density outside of the truncation radius. Should be set to something much smaller than the mean gas density inside the truncation radius.

Parameter
Initial : bb_test : mean_density

Summary
The mean gas density inside the truncation radius.

Type
float

Default
1.0e-6

Scope
Enzo

The mean gas density inside the truncation radius.

Parameter
Initial : bb_test : angular_rotation_velocity

Summary
The angular rotation velocity of gas inside the truncation radius in radians per second.

Type
float

Default
0.0

Scope
Enzo

Inside the truncation radius, the gas rotates as a solid body around the z-axis. This parameter determines the angular rotation velocity in units of radians per second.

---

cloud

The cloud Initial subgroup is used to setup a Spherical cloud embedded in a hot wind. The cloud and wind are assumed to be in pressure equilibrium with one another.

The presence of (or lack thereof) the "bfield_x", "bfield_y", and "bfield_z" fields indicate whether the setup is purely hydrodynamic or involves magnetic fields. Presently, only uniform magnetic fields are supported if they are constant across the entire domain. The values of the magnetic fields are specified in one of 2 ways:

1. If the *uniform_bfield* parameter is passed a list of 3 floats, the first, second, and third entries are used to initilize the x, y, and z components of a uniform magnetic field. If the "bfieldi_x", "bfieldi_y", and "bfieldi_z" face-centered fields are defined, then they will be correctly initialized for use with the VL+CT integrator.
2. (Deprecated) If the *uniform_bfield* parameter is not specified (or is passed a list containing 0 entries), then the cell-centered magnetic fields are assumed to have been initialized by another Initial subgroup (e.g. value) prior to the call of this subgroup.

The initial density of a cell entirely enclosed by the cloud is the spatial average taken over that cell of $\rho_{\rm cl}(1+\delta(\vec{x}))$, where $\delta(\vec{x})$ is a spatially varying perturbation term that we will return to momentarily. Outside of the cloud, we initialize the density to $\rho_{\rm w}$.

We use subsampling near the edge of cloud to estimate the fraction of the cell's volume occupied by the cloud, f_V. In more detail, we initialize the density of these cells to $f_V\rho_{\rm inside-avg} + (1-f_{\rm V})\rho_{\rm w}$, where $\rho_{\rm inside-avg}$ is the average of $\rho_{\rm cl}(1+\delta(\vec{x}))$ taken over all subcells that lie within the cloud.

Now, we consider the definition of the perturbation term that can be used to break symmetries in the cloud's initial density distribution. This is a sum of N planar waves or: $\delta(\vec{x}) = A\sum_{i=1}^{N}\cos\left(\frac{2\pi}{\lambda_i}\hat{e}_i\cdot\vec{x}+\phi_i\right)$, where

- N is set by *perturb_Nwaves*
- A is set by *perturb_amplitude*
- for each i we draw a random unit vector $\hat{e}_i$, as well as random values for λ_i and ϕ_i from $[\lambda_{\rm min},\lambda_{\rm max}]$ and $[0,\pi)$. $\lambda_{\rm min}$ and $\lambda_{\rm max}$ are set by the parameters *perturb_min_lambda* and *perturb_max_lambda*. As an aside, ϕ_i doesn't need to sample values within the interval $[\pi,2\pi)$ since $\hat{e}_i$ samples all directions.

The random values are all drawn from uniform distributions using a PRNG seeded by the value of *perturb_seed*.

Note: The randomly drawn values should be portable across different machines. But, this has not been rigorously tested.

Parameter
Initial : cloud : subsample_n

Summary
Determines the subsampling resolution

Type
integer

Default
0

Scope
Enzo

Subsampling is used to initialize the fields in regions of overlap between the cloud and the wind. For cells in this region, the fraction of the volume enclosed by the cloud is estimated by subdividing a cell into 2^n subcells along each axis (a value of 0, corresponds to no subsampling. The average density of the cells in this region are volume weighted and the average velocities are mass weighted. The total energy in a cell is currently computed by assuming constant internal energy density throughout the grid and using the average velocities and densities (and, if applicable, the magnetic fields).

Parameter
Initial : cloud : cloud_radius

Summary
Initial radius of the spherical cloud

Type
float

Default
none

Scope
Enzo

This must be a positive value.

Parameter
Initial : cloud : cloud_center_x

Summary
x coordinate of the cloud center

Type
float

Default
0.0

Scope
Enzo

Parameter
Initial : cloud : cloud_center_y

Summary
y coordinate of the cloud center

Type
float

Default
0.0

Scope
Enzo

Parameter
Initial : cloud : cloud_center_z

Summary
z coordinate of the cloud center

Type
float

Default
0.0

Scope
Enzo

Parameter
Initial : cloud : cloud_density

Summary
initial mass density of the cloud

Type
float

Default
none

Scope
Enzo

This must be a positive value.

Parameter
Initial : cloud : metal_mass_frac

Summary
initial fraction of the mass density contributed by metals

Type
float

Default
0.0

Scope
Enzo

If the ` `"metal_density_frac"`` field exists and is registered as a member of the ` `"colour"`` group, then the field is initialized by multiplying this value by the `"density"` field (this is done everywhere, regardless of proximity to the cloud center). Under these circumstances, this must have a positive value.

Parameter
Initial : cloud : uniform_bfield

Summary
initial uniform magnetic field values

Type
list (float)

Default
[ ]

Scope
Enzo

If specified, provides the values of the components of the initial magnetic field that are uniform throughout the entire domain. When employed this MUST have 3 entries. This will also initialize the face-centered fields magnetic fields (in addition to the cell-centered fields) if the appropriate fields have been defined. When this is not specified (i.e., when this has a list of 0 entries), the magnetic fields are assumed to have been pre-initialized by a separate problem initializer prior to the execution of the cloud initializer.

Parameter
Initial : cloud : wind_density

Summary
initial mass density of the wind

Type
float

Default
none

Scope
Enzo

This must be a positive value.

Parameter
Initial : cloud : wind_velocity

Summary
initial velocity of the wind along the x-axis

Type
float

Default
0.0

Scope
Enzo

Parameter
Initial : cloud : wind_total_energy

Summary
initial specific total energy of the wind

Type
float

Default
none

Scope
Enzo

This must be a positive value.

Parameter
Initial : cloud : wind_internal_energy

Summary
initial specific internal energy of the wind

Type
float

Default
0

Scope
Enzo

If the `"internal_energy"` field is defined, then this must be a positive value. In this case, the value is also used to help initialize the `"total_energy"` field for cells that overlap with the cloud. However, if the `"internal_energy"` field is not defined, then this must not have a specified value (i.e. it must have a value of 0).

Parameter
Initial : cloud : perturb_seed

Summary
Seeds the perturbations to cloud density

Type
integer

Default
0

Scope
Enzo

This must hold a value representable by a 32-bit unsigned integer. It is meaningless unless *perturb_Nwaves* and *perturb_amplitude* are positive.

Parameter
Initial : cloud : perturb_Nwaves

Summary
Number of planar waves to use in perturbation machinery

Type
integer

Default
0

Scope
Enzo

This should be zero or larger.

Parameter
Initial : cloud : perturb_amplitude

Summary
Planar wave amplitude to use in perturbation machinery

Type
float

Default
0.0

Scope
Enzo

This must be a non-negative value

Parameter
Initial : cloud : perturb_min_lambda

Summary
Lower bound on wavelengths used in perturbation machinery

Type
float

Default
none

Scope
Enzo

This must be a non-negative value and must be less than *perturb_max_lambda*. It is meaningless unless *perturb_Nwaves* and *perturb_amplitude* are positive.

Parameter
Initial : cloud : perturb_max_lambda

Summary
Upper bound on wavelengths used in perturbation machinery

Type
float

Default
none

Scope
Enzo

This must be a non-negative value and must exceed *perturb_min_lambda*. It is meaningless unless *perturb_Nwaves* and *perturb_amplitude* are positive.

Parameter
Initial : cloud : wind_total_energy

Summary
initial specific total energy of the wind

Type
float

Default
none

Scope
Enzo

This must be a positive value.

inclined_wave

The inclined_wave Initial subgroup is used to setup a HD, MHD, or Jeans wave at an angle inclined to the simulation domain for testing HD/MHD integrators. If applicable, magnetic fields fields are set to zero when a HD wave is initialized.

The initialization procedure was adopted from Gardiner & Stone (2008) . Specifically, a coordinate system "x0", "x1", "x2" is defined and the wave is initialized to travel along "x0". The transformation between "x", "y", "z" and "x0", "x1", "x2", is determined by the values of the *alpha* and *beta* parameters. They are explicitly related by

$$
\begin{aligned}
x &= x_0 \cos\alpha \cos\beta - x_1 \sin\beta - x_2 \sin\alpha \cos\beta \\
y &= x_0 \cos\alpha \sin\beta + x_1 \cos\beta - x_2 \sin\alpha \sin\beta \\
z &= x_0 \sin\alpha + x_2 \cos\alpha
\end{aligned}
$$

As in that paper, non-zero magnetic fields are initialized using the vector potential to ensure that they are divergence-free.

Parameter
Initial : inclined_wave : alpha

Summary
Angle used to help determine wave inclination

Type
float

Default
0

Scope
Enzo

The angle is assumed to have units of radians.

Parameter
Initial : inclined_wave : beta

Summary
Angle used to help determine wave inclination

Type
float

Default
0

Scope
Enzo

The angle is assumed to have units of radians.

Parameter
Initial : inclined_wave : wave_type

Summary
Specifies the type of wave to initialize.

Type
string

Default
alfven

Scope
Enzo

This value specifies the type of wave to initialize. We have provided more details about each option down below. Note, when using an MHD solver with a non-MHD wave, the mangetic fields are uniformly initialized to zero.

Hydro Waves

The values used to initialize hydrodynamical linear waves are taken from the columns of the matrix given in equation B3 of Stone et al. (2008) . Valid hydrodynamical waves include:

- `"sound"` A linear sound wave.
- `"hd_entropy"` A linear HD entropy wave with perturbations in v0 (velocity along the "x0"-axis).
- `"hd_transv_entropy_v1"` A linear HD entropy wave with perturbations in velocity component v1 (transverse to the direction of bulk motion).
- `"hd_transv_entropy_v2"` A linear HD entropy wave with perturbations in velocity component v2 (transverse to the direction of bulk motion).

MHD Waves

Each of the valid MHD waves are described in Gardiner & Stone (2008) . Valid MHD wave types include:

- `"alfven"` A linear Alfven wave with perturbations to the magnetic field along the "x2"-axis.
- `"circ_alfven"` A traveling circularly polarized Alfven wave.
- `"mhd_entropy"` A linear MHD entropy wave.
- `"fast"` A linear fast magnetosonic wave.
- `"slow"` A linear slow magnetosonic wave.

Jeans Wave

To initialize a Jeans wave, set this parameter to `"jeans"`. We use equations consistent with what Athena and (earlier versions of) Athena++ use. In detail, we use:

$$
\begin{aligned}
\rho &= \rho_{\rm bkg}\left(1 + A\sin\left(\frac{2\pi}{\lambda}x_0\right)\right) \\
\rho v_0 &= \rho_{\rm bkg}\frac{\sqrt{|\omega^2|}}{2\pi/\lambda}A\begin{cases} 0 & \omega^2 > 0 \\ \cos\left(\frac{2\pi}{\lambda}x_0\right) & \omega^2 < 0 \end{cases} \\
v_1 &= 0 \\
v_2 &= 0 \\
\rho E &= \frac{P_{\rm bkg}}{\gamma - 1} + \gamma A\sin\left(\frac{2\pi}{\lambda}x_0\right)
\end{aligned}
$$

in which:

- E specifies the specific internal energy
- γ is the adiabatic index.
- A is the amplitude, specified by Initial:inclined_wave:amplitude and λ is the wavelength, specified by Initial:inclined_wave:lambda
- $\rho_{\rm bkg} = 1$ and $P_{\rm bkg} = 1/\gamma$ in the appropriate code units
- $\omega^2 = (2\pi c_{s,\rm bkg}/\lambda)^2(1 - (\lambda/\lambda_J)^2)$ is the dispersion relation. In this equation, $c_{s,\rm bkg}^2 = \gamma P_{\rm bkg}/\rho_{\rm bkg}$ and $\lambda_J = c_{s,\rm bkg}\sqrt{\pi/(G\rho_{\rm bkg})}$. Note that the value of G is directly set by Method:gravity:grav_const.

Parameter
Initial : inclined_wave : amplitude

Summary
Sets the amplitudes of the waves.

Type
float

Default
1.e-6

Scope
Enzo

This must be a positive value. This has no effect for the circularly polarized Alfven wave (for that case, amplitude is fixed at 0.1).

Parameter
Initial : inclined_wave : lambda

Summary
The wavelength of the wave.

Type
float

Default
1.

Scope
Enzo

This must be a positive value.

Parameter
Initial : inclined_wave : positive_vel

Summary
Sets the sign of the wave speed.

Type
logical

Default
true

Scope
Enzo

Do not specify this parameter when initializing a circularly polarized Alfven wave or a Jeans wave. This is ignored for linear HD entropy waves when Initial:inclined_wave:parallel_vel is specified.

Parameter
Initial : inclined_wave : parallel_vel

Summary
optionally sets the background velocity for HD waves

Type
float

Default
none

Scope
Enzo

This can be used to specify a background velocity along v0 for HD linear waves. At present, this parameter should only be specified for the hydrodynamic waves.

merge_sinks_test

The merge_sinks_test Initial subgroup is used to read and initialise particle data from a text file. It is designed to be run with a small number of particles, in order to check conservation of mass and momentum (and potentially other quantities) when running with the *"merge_sinks"* method. The data are assumed to be arranged into seven columns, corresponding to mass, the x,y,z coordinates and the x,y,z-components of velocity, respectively. Each row corresponds to one sink particle.

Note: sink particles must have a "mass" attribute if this initializer is used.

Parameter
Initial : merge_sinks_test : particle_data_filename

Summary
Name of the file to read from.

Type
string

Default
none

Scope
Enzo

Must point to a valid text file, with data arranged in seven columns seperated by blank space

music

The music Initial subgroup is used to read block data from HDF5 files generated by MUSIC initial conditions generator. Parameters are used to specify the HDF5 files to read from, the names of the HDF5 datasets, what type of data the datasets contain ("field" or "particle"), field or particle names, and particle attributes. Additionally, a *coords* parameter is used to specify the axis ordering used. The music group has its own *list* parameter, one for each field or particle type and attribute.

The following example reads the "density" field from "GridDensity" file, and the "dark" particle "position_x" attributes from the "ParticleDisplacements_x" file:

```
Initial {

  list = ["music"];
  music {

     file_list = ["FD","PX"];
     FD {
       type      = "field";
       name      = "density";
       coords    = ".zyx";
       file      = "GridDensity";
       dataset   = "GridDensity";
     }
     PX {
       type      = "particle";
       name      = "dark";
       coords    = ".zyx";
       attribute = "position_x";
       file      = "ParticleDisplacements_x";
```

(continues on next page)

(continued from previous page)

```
      dataset   = "ParticleDisplacements_x";
    }
  }
}
```

Parameter
Initial : music : list

Summary
Name of the HDF5 to read from

Type
string

Default
none

Scope
Enzo

List of file identifiers, one for each field or particle type+attribute.

Parameter
Initial : music : <file> : type

Summary
Type of data to read in

Type
string

Default
none

Scope
Enzo

Type of data to read in, either "field" or "particle".

Parameter
Initial : music : <file> : file

Summary
Name of the HDF5 file to read from

Type
string

Default
none

Scope
Enzo

Name of the HDF5 file to read from.

Parameter
Initial : music : <file> : dataset

Summary
Name of the dataset to read from the the HDF5 file

Type
string

Default
none

Scope
Enzo

Name of the dataset to read from the the HDF5 file.

Parameter
Initial : music : <file> : name

Summary
Name of the field or particle type

Type
string

Default
none

Scope
Enzo

Name of the field or particle type.

Parameter
Initial : music : <file> : attribute

Summary
Name of the particle attribute to initialize

Type
string

Default
none

Scope
Enzo

Name of the particle attribute to initialize..

Parameter
Initial : music : <file> : coords

Summary
Ordering of axes in the HDF5 file

Type
string

Default
"zyx"

Scope
Enzo

String defining the axis ordering of 'x', 'y', and 'z' in the HDF5 file. For MUSIC initial conditions, which may have 4D datasets, "tzyx" can be used, where "t" is ignored and can be any character other than 'x', 'y', or 'z'.

sedov

Parameter
Initial : sedov : array

Summary
Size of array of Sedov blasts

Type
list (integer)

Default
[1, 1, 1]

Scope
Enzo

This parameter defines the size of the array of Sedov blast waves. The default is a single blast.

Parameter
Initial : sedov : radius_relative

Summary
Initial radius of the Sedov blast

Type
float

Default
0.1

Scope
Enzo

Todo
write

Parameter
Initial : sedov : pressure_in

Summary
Pressure inside the Sedov blast

Type
float

Default
1.0

Scope
Enzo

Todo
write

Parameter
Initial : sedov : pressure_out

Summary
Pressure outside the Sedov blast

Type
float

Default
1.0e-5

Scope
Enzo

Todo
write

Parameter
Initial : sedov : density

Summary
Density for the Sedov blast array problem

Type
float

Default
1.0

Scope
Enzo

Todo
write

shock_tube

The shock_tube Initial subgroup is used to setup axis-aligned shock tube test problems.

Generically, a shock tube get's set up to evolve along an axis given by the value of *aligned_ax*. The discontinuity is always placed at 0.5 along that axis (typically the domain should extend from 0.0 to 1.0).

Parameter
Initial : shock_tube : setup_name

Summary
Specifies the name of the shock tube problem to setup.

Type
string

Default
none

Scope
Enzo

Valid shock tube problems include:

- `"rj2a"` An MHD shock tube problem illustrated in Figure 2a of Ryu & Jones (1995) . The initialization assumes that the adiabatic index is 5/3.
- `"sod"` The hydrodynamical Sod shock tube test problem. The canonical adiabatic is 1.4 (although this is not required).

Parameter
Initial : shock_tube : aligned_ax

Summary
Specify the axis along which the shock tube evolves along.

Type
string

Default
x

Scope
Enzo

Allowed values are `"x"` , `"y"` , or `"z"` .

Parameter
Initial : shock_tube : axis_velocity

Summary
Value to add to velocity component along aligned_ax

Type
float

Default
0.

Scope
Enzo

This value is added throughout the entire domain.

Parameter
Initial : shock_tube : transverse_velocity

Summary
Value to add to a velocity component perpendicular to aligned_ax

Type
float

Default
0.

Scope
Enzo

This value is added throughout the entire domain. If aligned_ax is "x" , "y" , or "z" , then this value is added to the "velocity_y" , "velocity_z" , or "velocity_z" field.

Parameter
Initial : shock_tube : flip_initialize

Summary
Whether to mirror the initial condition across the discontinuity

Type
logical

Default
false

Scope
Enzo

When this is "true" the entire setup is mirrored across the discontinuity. Basically the left and right states are swapped AND all components of the magnetic field and velocity (including contributions from axis_velocity and transverse_velocity) are multiplied by -1.

shu_collapse

The shu_collapse Initial subgroup is used to initialize a Shu Collapse problem, as described in Federrath et al 2010, ApJ, 713, 269. In particular, it initializes a gravitationally unstable isothermal sphere of gas with an inverse-square density profile, with an optional uniform "drift velocity", and an optional sink particle at the center of the domain.

Running this test problem requires the use of the "pm_deposit", "gravity", "mhd_vlct", "sink_maker", and "pm_update" methods.

This initializer requires periodic boundary conditions, three spatial dimensions, and that the gas adiabatic index ("Field:gamma") is between 1.0 and 1.000001.

The following fields are required, and must be specified in the input parameter file: "density", "density_total", "density_particle", "density_particle_accumulate", "density_gas", "particle_mass", "potential", "potential_temp", "potential_copy", "total_energy", "pressure", "acceleration_x", "acceleration_y", "acceleration_z", "velocity_x", "velocity_y", "velocity_z", "X", "X_copy", "B", "B_copy", "density_source", "density_source_accumulate", "mom_dens_x_source", "mom_dens_x_source_accumulate", "mom_dens_y_source", "mom_dens_y_source_accumulate", "mom_dens_z_source", and "mom_dens_z_source_accumulate".

Sink particles must have the following attributes: "mass", "x", "y", "z", "vx", "vy", "vz", and "is_copy". These must all be of type "default", except for "is_copy", which should be of type "int64".

Sink particles must be in the "is_gravitating" particle group.

Parameter
Initial : shu_collapse : center

Summary
The coordinates of the center of the collapse.

Type
list (float)

Default
[0.0,0.0,0.0]

Scope
Enzo

The coordinates of the center of the sphere.

Parameter
Initial : shu_collapse : drift_velocity

Summary
The initial uniform velocity of the gas.

Type
list (float)

Default
[0.0,0.0,0.0]

Scope
Enzo

The initial uniform velocity of the gas.

Parameter
Initial : shu_collapse : truncation_radius

Summary
The truncation radius of the isothermal sphere.

Type
float

Default
1.0

Scope
Enzo

The truncation radius of the isothermal sphere. Within this radius, the gas has an inverse square density profile; outside of this radius, the gas density is determined by "external_density". Value must be at most a quarter of the domain width..

Parameter
Initial : shu_collapse : nominal_sound_speed

Summary
The nominal uniform sound speed of the gas.

Type
float

Default
1.0

Scope
Enzo

Nominal uniform sound speed of the gas used to initialise the total specific energy. In practice the actual sound speed will be different since the adiabatic index is not exactly 1.

Parameter
Initial : shu_collapse : instability_parameter

Summary
The instability parameter which determines the gas density profile.

Type
float

Default
2.1

Scope
Enzo

Instability parameter - sphere is gravitationally unstable if this is greater than 2.0. Determines density profile according to $\rho(r) = \frac{Ac_s^2}{4\pi G r^2}$, where rho is the gas density, r is the distance from the center of the sphere, A is the instability parameter, c_s is the nominal sound speed, and G is the gravitational constant.

Parameter
Initial : shu_collapse : external_density

Summary
The gas density outside of the truncation radius.

Type
float

Default
1.0e-6

Scope
Enzo

The gas density outside of the truncation radius. Should be set to something much smaller than the gas density just inside the truncation radius.

Parameter
Initial : shu_collapse : central_sink_exists

Summary
Controls whether a sink particle is placed at the center in the initial conditions.

Type
logical

Default
false

Scope
Enzo

If true, a sink particle is initialised with position at the center of the sphere, and velocity equal to "drift_velocity". Its mass is determined by "central_sink_mass".

Parameter
Initial : shu_collapse : central_sink_mass

Summary
The mass of the central sink particle, if it exists.

Type
float

Default
0.0

Scope
Enzo

If "central_sink_exists" is true, this determines the mass of the central sink particle. If false, this parameter is ignored.

---

turbulence

Parameter
Initial : turbulence : density

Summary
Initial density for turbulence initialization and method

Type
float

Default
1.0

Scope
Enzo

Initial density for initializing the turbulence problem.

Parameter
Initial : turbulence : pressure

Summary
Initial pressure for turbulence initialization and method

Type
float

Default
0.0

Scope
Enzo

Initial pressure for initializing the turbulence problem. Default is 0.0, meaning it is not used. Either *pressure* or *temperature* should be defined, but not both.

Parameter
Initial : turbulence : temperature

Summary
Initial temperature for turbulence initialization and method

Type
float

Default
0.0

Scope
Enzo

Initial temperature for initializing the turbulence problem. Default is 0.0, meaning it is not used. Either *pressure* or *temperature* should be defined, but not both.

vlct_bfield

This is used to compute the cell-centered magnetic field for the VL + CT MHD method. This initializer can be utilized in 2 ways:

1. Components of the vector potential (`"Ax"`, `"Ay"`, `"Az"`) can be specified as parameters of the subgroup (functions can be specified for each component in the same way as functions are specified for the value subgroup. The initializer operates in this mode as long as the values for one of the components of the vector potential is specified (any unspecified components are assumed to be zero everywhere). In this mode, both the cell-centered and face-centered magnetic field values get specified.
2. Initialize the cell-centered values of the magnetic fields after after another Initial subgroup (e.g. the value subgroup) has already to specified the face-centered magnetic fields (`"bfieldi_x"`, `"bfieldi_y"`, `"bfieldi_z"`). The cell-centered value is just the average of the corresponding face-centered component. The initializer operates in this mode if none of the components of the vector potential have specified values. (To properly use this mode, specify `"vlct_bfield"` in *Initial:list* **after** the name of the initializer that sets up the face-centered values.

In both modes, the option to update partially initialized `"total_energy"` fields with the specific magnetic energy computed from the newly computed cell-centered bfields and pre-initialized `"density"` fields.

It might be nice to eventually generalize this initializer to be able to initialize cell-centered B-fields from vector potentials for MHD integrators that don't require face-centered B-fields

Parameter
Initial : vlct_bfield : update_etot

Summary
update total energy with the initialized magnetic fields

Type
logical

Default
false

Scope
Enzo

If true, then the calculated cell-centered magnetic fields are used to update the specific total energy. This requires that the `"total_energy"` field has already been partially initialized (it just doesn't include the specific magnetic energy), and that the `"density"` field has been initialized.

Parameter
Initial : vlct_bfield : Ax

Summary
Expression for the x-component of the magnetic vector potential

Type
list (float-expr, [logical-expr, float-expr, [...]])

Default
[]

Scope
Enzo

This parameter allows for the direct specification of the x-component of the magnetic vector potential (which will be used to compute magnetic fields). The arguements for this parameter follow the same sets of rules as the parameters of Initial:value. If this parameter is not specified, but the values of the other components of the magnetic vector potential are, then this component is assumed to be zero everywhere.

Parameter
Initial : vlct_bfield : Ay

Summary
Expression for the y-component of the magnetic vector potential

Type
list (float-expr, [logical-expr, float-expr, [...]])

Default
[]

Scope
Enzo

This parameter allows for the direct specification of the y-component of the magnetic vector potential (which will be used to compute magnetic fields). The arguements for this parameter follow the same sets of rules as the parameters of Initial:value. If this parameter is not specified, but the values of the other components of the magnetic vector potential are, then this component is assumed to be zero everywhere.

Parameter
Initial : vlct_bfield : Az

Summary
Expression for the z-component of the magnetic vector potential

Type
list (float-expr, [logical-expr, float-expr, [...]])

Default
[]

Scope
Enzo

This parameter allows for the direct specification of the z-component of the magnetic vector potential (which will be used to compute magnetic fields). The arguements for this parameter follow the same sets of rules as the parameters of Initial:value. If this parameter is not specified, but the values of the other components of the magnetic vector potential are, then this component is assumed to be zero everywhere.

3.1.8 Memory

Parameters in the Memory group are used to define the behavior of Cello's dynamic memory allocation and deallocation.

Parameter
Memory : active

Summary
Whether to track memory usage

Type
logical

Default
true

Scope
Cello

This parameter is used to turn on or off Cello's build-in memory tracking. By default it is on, meaning it tracks the number and size of memory allocations, including the current number of bytes allocated, the maximum over the simulation, and the maximum over the current cycle. Cello implements this by overloading C's new, new[], delete, and delete[] operators. This can be problematic on some systems, e.g. if an external library also redefines these operators, in which case this parameter should be set to false. This can be turned off completely by setting "memory" OFF (default value) as a cmake option.

3.1.9 Mesh

Parameter
Mesh : root_blocks

Summary
Number of Blocks used to tile the coarsest refinement level

Type
list (integer)

Default
[1, 1, 1]

Scope
Cello

This parameter specifies the number of Blocks along each axis in the mesh "array". The product must not be smaller than the number of processors used.

Parameter
Mesh : root_rank

Summary
Physical dimensionality of the problem

Type
integer

Default
0

Scope
Cello

Number of physical dimensions in the problem, 1, 2, or 3.

Parameter
Mesh : root_size

Summary
Coarsest Patch size

Type
list (integer)

Default
[1, 1, 1]

Scope
Cello

This parameter specifies the total size of the root-level mesh. For example, [400, 400] specifies a two dimensional root-level discretization of 400 x 400 zones, excluding ghost zones.

3.1.10 Method

Parameter
Method : list

Summary
Sequence of numerical methods to apply.

Type
list (string)

Default
none

Scope
Cello

This parameter specifies the list of numerical methods to use, and is analagous to "EvolveLevel" routine in ENZO. Each method in the list is applied in the order specified. Possible methods include:

- "comoving_expansion" adds comoving expansion terms to the physical variables.
- "cosmology" for writing redshift to monitor output.
- "flux_correct" for performing flux corrections when using AMR.
- "grackle" for heating and cooling methods in the Enzo Grackle library
- "gravity" solves for the gravitational potential given gas and particle density fields.
- "heat" for the forward-Euler heat-equation solver, which is used primarily for demonstrating how new Methods are implemented in Enzo-E
- "pm_deposit" deposits "dark" particle density into "density_particle" field using CIC for "gravity" method.

- "pm_update" moves cosmological "dark" particles based on positions, velocities, and accelerations. **This will be phased out in favor of a more general "move_particles" method.**
- "ppm" for Enzo-E's PPM hydrodynamics method. *This may be phased out in favor of using a more general "hydro" method instead, with a specific hydro solver specified.*
- "ppml" for the PPML ideal MHD solver. *This may be phased out in favor of using a more general "mhd" method instead, with a specific mhd solver specified.*
- "mhd_vlct" for the VL + CT (van Leer + Constrained Transport) MHD solver.
- "trace" for moving tracer particles. **This will be phased out in favor of a more general "move_particles" method.**
- "turbulence" computes random forcing for turbulence simulations.

Parameters specific to individual methods are specified in subgroups, e.g.:

```
Method {
   list = ["ppm"];
   ppm {
      diffusion   = true;
      flattening  = 3;
      steepening  = true;
      dual_energy = false;
   }
}
```

There are a subset of parameters that can be specified for all methods. For example, a *schedule* subgroup can be defined for any method object (to dictate when the method is executed).

For more detailed documentation on Methods, see *Enzo-E Methods*

Parameter
Method : courant

Summary
Global Courant safety factor

Type
float

Default
1.0

Scope
Cello

The global Courant safety factor is a multiplication factor for the time step applied on top of any Field or Particle specific Courant safety factors.

accretion

Parameter
Method : accretion : accretion_radius_cells

Summary
The radius of the spherical accretion zone around each sink particle, in units of the minimum cell width.

Type
float

Default
4.0

Scope
Enzo

The accretion radius (i.e., the radius of the spherical accretion zone) in units of the minimum cell width (i.e., if the cell width along all the x, y, and z-axes are hx, hy, and hz, then the minimum cell width is the minimum of hx, hy, and hz),, at the highest refinement level. Its value must be less than one fewer than the minimum ghost depth for "flux" accretion, and less than the minimum ghost depth for other flavors of accretion. The ghost depth is 4 (along all axes) by default.

Parameter
Method : accretion : flavor

Summary
The flavor of accretion used.

Type
string

Default
""

Scope
Enzo

The flavor of accretion used, which can be either "threshold", "bondi_hoyle", "flux", or "dummy". If this parameter is not set in the parameter file, or if some other string is provided, then Enzo-E will exit with an error message.

Parameter
Method : accretion : physical_density_threshold_cgs

Summary
The value of the accretion (physical) density threshold in cgs units.

Type
float

Default
1.0e-24

Scope
Enzo

The value of the (physical) density threshold in cgs units. The density in each cell in the accretion zone cannot go below this value during the accretion process. The value of this parameter in code density units must be greater than or equal to the value of the density floor imposed by the hydro method (either "ppm" or "mhd_vlct". In cosmological

simulations, the density unit is the mean matter density of the universe which decreases with time, which means that the value of a density quantity expressed in these units will increase with time, while the density floor is fixed in comoving units. The consequence is that is is sufficient for the density threshold to be above the density floor at the start of the simulation to guarantee that it will be above the floor at all subsequent times.

Parameter
Method : accretion : max_mass_fraction

Summary
The maximum fraction of mass which can be accreted from a cell in one timestep.

Type
float

Default
0.25

Scope
Enzo

This parameter specifies the maximum fraction of mass which can be accreted from a cell in one timestep. This value of this parameter must be between 0 and 1.

check

The `"check"` method is used for writing HDF5 data files used by Cello for restart. The `"check"` method can also be used for data dumps, though currently all data (all fields and all particles on all blocks) are written to disk. This method should typically be called with a schedule.

Warning 1: Currently requires the `"order_morton"` method to be called beforehand, with a matching schedule.

Warning 2: If a file or directory corresponding to a requested checkpoint dump already exists, no data will be written. This is to avoid over-writing checkpoint files, but may not necessarily be what is expected. This behavior may be revised in the future.

Parameter
Method : check : dir

Summary
Directory in which to write the checkpoint files

Type
list (string)

Default
“”

Scope
Enzo

This parameter specifies the subdirectory for the output file. The first element is the file name, which may contain printf-style formatting fields. Subsequent values correspond to variables for the formatting fields, which may include “cycle”, “time”, “count” (a counter incremented each time output is performed), “proc” (the process rank), and “flipflop” (alternating 0 and 1).

Note warning 2 above: if the directory pre-exists, no data will be written!

Parameter
Method : check : num_files

Summary
The number of HDF5 files in which to store restart data

Type
integer

Default
1

Scope
Enzo

This parameter specifies the number of HDF5 files to use for writing checkpoint data. Since an ordering is used, files will be close to the same size. For large runs, using a value close to the number of compute nodes is generally a reasonable value. Note that parallel HDF5 is not (currently) used, so the only parallelism available is from writing to multiple files. Do not use the default of 1 for large runs!

Parameter
Method : check : ordering

Summary
Block-ordering used for determining block-to-file mapping

Type
string

Default
"order_morton"

Scope
Enzo

This parameter defines the method used for ordering blocks. Currently, the default "order_morton" is the only allowed value.

Parameter
Method : check : include_ghosts

Summary
Whether to include ghost zones in checkpoint files

Type
logical

Default
false

Scope
Enzo

This parameter specifies whether to include ghost zones when writing field data to HDF5 files. While this should likely be left as false to save disk storage (e.g. a factor of about 3.375 when 16^3 blocks are used), earlier versions of Enzo-E always included ghost zones, so if the original behavior is required this should be set to true.

Parameter
Method : check : monitor_iter

Summary
How often to write progress updates

Type
integer

Default
0

Scope
Enzo

This is a debugging parameter, used to periodically write progress updates to stdout. The value indicates how often an update gets written, with 0 meaning no output, and k>0 meaning output every time k blocks get written. This can produce a lot of output for large problems and k=1.

feedback

Method:feedback parameters.

Parameter
Method : feedback : supernovae

Summary
Whether to turn on supernova feedback in EnzoMethodFeedbackSTARSS

Type
logical

Default
true

Scope
Enzo

Whether to turn on supernova feedback in EnzoMethodFeedbackSTARSS

Parameter
Method : feedback : unrestricted_sn

Summary
Whether to turn on supernova feedback in EnzoMethodFeedbackSTARSS

Type
logical

Default
true

Scope
Enzo

Allow for > 1 supernova event per star particle in a timestep.

Parameter
Method : feedback : stellar_winds

Summary
Whether to turn on stellar winds in EnzoMethodFeedbackSTARSS

Type
logical

Default
true

Scope
Enzo

Whether to turn on stellar wind feedback in EnzoMethodFeedbackSTARSS

Parameter
Method : feedback : radiation

Summary
Whether to turn on radiation in EnzoMethodFeedbackSTARSS

Type
logical

Default
true

Scope
Enzo

Whether to turn on ionizing radiation in EnzoMethodFeedbackSTARSS. If true, updates the "luminosity" attribute of "star" particles using piecewise rates taken from Appendix A of Hopkins et al. (2018).

Parameter
Method : feedback : analytic_SNR_shell_mass

Summary
Calculates supernova remnant shell mass via analytic formulae

Type
logical

Default
true

Scope
Enzo

Calculates supernova remnant shell mass via analytic formulae

---

Parameter
Method : feedback : fade_SNR

Summary
Allow coupling to fading phase of SNe in EnzoMethodFeedbackSTARSS

Type
logical

Default
true

Scope
Enzo

Allow coupling to fading phase of SNe in EnzoMethodFeedbackSTARSS if cell width is greater than fading radius.

Parameter
Method : feedback : NEvents

Summary
Manually set off supernovae in test problems

Type
integer

Default
-1

Scope
Enzo

If -1, will probabilistically model supernova. If "NEvents" > 0, will set off N-supernovae per particle, (1 per timestep for each particle). If NEvents = 0, no supernovae will go off. Mostly for testing purposes.

flux_correct

Parameter
Method : flux_correct : group

Summary
Name of group of fields to apply flux correction to

Type
string

Default
"conserved"

Scope
Cello

Flux correction must be applied to conserved fields in AMR simulations to maintain conserved quantities across mesh resolution jumps. This parameter selects the group of fields to which the "flux_correct" method will be applied.

Fields that store a conserved quantity divided by density (e.g. `"total_energy"`, `"velocity_x"`) have special handling. Such fields must be included in both the group specified by this parameter AND the `"make_field_conservative"` group. Flux corrections are applied to an element of such fields according to the following procedure:

1. The element is multiplied by the corresponding element of the `"density"` field (before the flux corrections are applied to the `"density"` field).
2. Flux corrections are applied to the product from step 1.
3. Finally, the element in the original field is assigned the value computed in step 2 divided by the corresponding element from the `"density"` field (after flux corrections are applied to the `"density"` field).

An error will be raised if these special fields are detected, and the `"density"` field is not included in the group specified by this parameter.

Parameter
Method : flux_correct : min_digits

Summary
Number of digits expected to be conserved by fields in tests

Type
list

Default
[]

Scope
Cello

Specifies the minimum number of digits that are expected to be conserved by fields. This is used for testing purposes (the simulation will check at each timestep whether this expectation has been met). Entries of this list should alternate between the name of fields (a string) and the expected number of conserved digits for that field (a float).

The example provided below indicates that the `"density"` field and the product of the `"density"` & `"velocity_x"` fields are expected to be conserved to `7.1` and `4.9` digits, respectively:

```
Method {
   flux_correct {
      min_digits = ["density", 7.1,
                    "velocity_x", 4.9];
   }
}

Group {
   list = [ "conserved", "make_field_conservative" ];
   conserved {
      field_list = [ "density", "velocity_x"];
   }
   make_field_conservative {
      field_list = [ "velocity_x"];
   }
}
```

For the sake of backwards compatibility, this parameter can be assigned a single float (that is not in a list). In this case, the value is assumed to be the expected minimum number of digits conserved by the `"density"` field. *(Support for this type of parameter may be removed in the future)*

grackle

"Grackle is a chemistry and radiative cooling library for astrophysical simulations. It is a generalized and updated version of the chemistry network of the Enzo simulation code."

While most of the parameters come directly from Grackle, there are a few notable exceptions. These generally affect how Enzo-E uses Grackle and don't have direct counterparts listed on the Grackle parameters section of the Grackle website. These parameters include:

Parameter
Method : grackle : courant

Summary
Courant safety factor

Type
float

Default
1.0

Scope
Enzo

The method-specific courant safety factor. This is meaningless unless *use_cooling_timestep* has been set to `true`. In that case, the timestep associated with the Grackle method is this value mutliplied by the minimum timestep.

Parameter
Method : grackle : use_cooling_timestep

Summary
Whether to limit the timestep by the minimum cooling time

Type
logical

Default
false

Scope
Enzo

By default, usage of Grackle does not limit the timestep. When this parameter is set to `true`, the timestep is limited by the product of the minimum cooling time and *Method:grackle:courant*.

Parameter
Method : grackle : radiation_redshift

Summary
redshift of the UV background in non-cosmological simulations

Type
float

Default
-1.0

Scope
Enzo

In non-cosmological simulations, this parameter is used to specify the redshift of the UV background. The default value, `-1.0`, is used to indicate that this parameter is unset.

- When this parameter has a value other than the default value in a cosmological simulation, the program will abort with an error message.
- When this parameter has a default value in a non-cosmological simulation, the radiation redshift is set to 0.0, internally.

All of the other allowed parameters are used to directly configure the grackle parameters stored in Grackle's configuration object, which are each listed on the Grackle parameters section of the Grackle website. In general, to configure a given parameter on that page, `<grackle-param>`, just assign your desired value to Method:grackle:<grackle-param>. The primary exceptions to this guideline are for the following grackle parameters:

- `use_grackle`: when the grackle method is in use, this is always set to `1`.
- `Gamma`: this grackle parameter is instead initialized by *Physics:fluid_props:eos:gamma*
- `grackle_data_field`: this grackle parameter is initialized with the value of *Method:grackle:data_file*

If you choose not to specify a value for a given grackle parameter, the default value is selected by the Grackle library.

For brevity (and to avoid having out-of-date documentation), we omit descriptions for the vast majority of recognized parameters that are directly used to initialize a corresponding grackle parameter. With that said, we make exceptions for a small handful of these parameters (primarily in cases where a parameter's value may necessitate the existence of a field) and provide descriptions for them down below:

Parameter
Method : grackle : data_file

Summary
Path to the data file containing the metal cooling and UV background tables.

Type
string

Default
""

Scope
Enzo

Path to the data file containing the metal cooling and UV background tables. This parameter is directly used to initialize Grackle's `grackle_data_file` parameter. The only reason this isn't called Method:grackle:grackle_data_file is for the sake of maintaining backwards compatability.

Parameter
Method : grackle : primordial_chemistry

Summary
Flag to control which primordial chemistry network is used

Type
integer

Default
0

Scope
Enzo

Flag to control which primordial chemistry network is used (this directly corresponds to Grackle's `primordial_chemistry` parameter).

0: no chemistry network. Radiative cooling for primordial species is solved by interpolating from lookup tables calculated with Cloudy. A simplified set of functions are available (though not required) for use in this mode. For more information, see Pure Tabulated Mode.

1: 6-species atomic H and He. Active species: H, H^+, He, He^+, $^{++}$, e^-.

2: 9-species network including atomic species above and species for molecular hydrogen formation. This network includes formation from the H^- and $H_2{}^+$ channels, three-body formation (H + H + H and H + H + H^2), H^2 rotational transitions, chemical heating, and collision-induced emission (optional). Active species: above + H^-, H^2, $H_2{}^+$.

3: 12-species network include all above plus HD rotation cooling. Active species: above plus D, D^+, HD.

Note: In order to make use of the non-equilibrium chemistry network (primordial_chemistry options 1-3), you must add and advect baryon fields for each of the species used by that particular option.

Parameter
Method : grackle : metal_cooling

Summary
Flag to enable metal cooling using the Cloudy tables

Type
logical

Default
false

Scope
Enzo

Flag to enable metal cooling using the Cloudy tables. If enabled, the cooling table to be used must be specified within the table specified by the `Method:grackle:data_file` parameter.

Note: In order to use the metal cooling, you must add and advect a metal density field.

gravity

Parameter
Method : gravity : solver

Summary
Name of the linear solver to use

Type
string

Default
"unknown"

Scope
Enzo

Identifier for the linear solver to use, which must be included in the "Solver:list" parameter.

Parameter
Method : gravity : order

Summary
Order of accuracy discretization to use for the discrete Laplacian

Type
integer

Default
4

Scope
Enzo

Second, fourth, and sixth order discretizations of the Laplacian are available; valid values are 2, 4, or 6.

Parameter
Method : gravity : accumulate

Summary
Whether to add one layer of ghost zones when refreshing particle density

Type
logical

Default
true

Scope
Enzo

This should be true for all runs with particles, since particle mass deposited in the "density_particle" field may bleed into the first layer of ghost zones. This parameter ensures that that mass will be included in "density_total".

Parameter
Method : gravity : dt_max

Summary
The maximum timestep returned by EnzoMethodGravity::timestep

Type
float

Default
1.0e10

Scope
Enzo

The timestep returned by EnzoMethodGravity::timestep (when called on a block) is calculated as follows. First, the geometric mean of the cell-widths in all dimensions is found, which we call the "mean cell width". Next, the quantity "epsilon" is calculated, as the mean cell width divided by the square of dt_max. Then, the maximum acceleration magnitude across all cells in the block is found, which we call "a_mag_max". We then calculate the mean cell width divided by the sum of a_mag_max and epsilon. The timestep is then the square root of this quantity. This means that

if all the accelerations are zero (such as at the first time step), the timestep is equal to dt_max. Defining the timestep in this way also means that the value of the timestep is independent of how the acceleration vectors are oriented relative to the mesh.

Parameter
Method : gravity : grav_const

Summary
Gravitational constant

Type
float

Default
none

Scope
Enzo

> **Warning:** This parameter is deprecated and will be removed in the future. The user should use *Physics:gravity:grav_const_codeU* instead. These 2 parameters have identical behavior when using the `"gravity"` method, but the new parameter influences other gravity-related calculations such as (but *NOT* limited to)
>
> - calculation of acceleration from a static potential
> - enforcement of minimum pressure support
> - star formation
> - certain initial conditions

heat

Parameter
Method : heat : alpha

Summary
Parameter for the forward euler heat equation solver

Type
float

Default
1.0

Scope
Enzo

Thermal diffusivity parameter for the heat equation.

merge_sinks

Parameter
Method : merge_sinks : merging_radius_cells

Summary
The distance within which sink particles merge with each other, in units of the minimum cell width

Type
float

Default
8.0

Scope
Enzo

The distance within which sink particles merge with each other, in units of the minimum cell width, i.e., the minimum of the cell widths in all 3 dimensions, at the highest level of refinement.

mhd_vlct

Method:mhd_vlct parameters are used to initialize parameters for Enzo-E's VL (+ CT) (magneto)hydrodynamic integrator.

The Method:mhd_vlct:mhd_choice determines whether the method is used as a pure hydrodynamic integrator or a MHD integrator that uses constrained transport.

Parameter
Method : mhd_vlct : mhd_choice

Summary
Denotes handling of bfields (or lack thereof)

Type
string

Default
none

Scope
Enzo

Denotes how the integrator handles magentic fields. This must be specified. Valid choices include:

- `"no_bfield"` The integrator acts as a pure hydrodynamical integrator; magnetic fields are ignored entirely.
- `"constrained_transport"` Magnetic fields are evolved using constrained transport. The primary representation of the magnetic fields are stored in face-centered cello fields and cell-centered cello-fields are used to store a secondary representation.

This may be updated to include additional options in the future. For more details see *"mhd_vlct" method*

For debugging purposes, there is technically one last choice, `"unsafe_constant_uniform".` This is NOT meant for science runs. When this option is selected, the magnetic field is treated as a cell-centered conserved quantity and the magnetic fluxes computed in the Riemann solver are directly added to to the magnetic fields (magnetic field values are only stored in cell-centered Cello fields). Outside of very specific cases, this will NOT enforce the divergence-free constrain of the magnetic fields to grow. To use this option, you need to explicitly comment out an error in `"enzo_EnzoMethodMHDVlct.cpp"`.

Parameter
Method : mhd_vlct : courant

Summary
Courant safety factor

Type
float

Default
1.0

Scope
Enzo

The method-specific courant safety factor. The method's minimum timestep is the minimum value of the following expression (which is computed for all cells):

$$C \times \min\left(\frac{\Delta x}{c_f + |v_x|} + \frac{\Delta y}{c_f + |v_y|} + \frac{\Delta z}{c_f + |v_z|}\right)$$

in which:

- C is the courant factor
- c_f is the local fast magnetosonic speed (it reduces to the local sound speed in the absence of magnetic fields)
- $v_x,\ v_y,\ v_z$ are the velocity components
- $\Delta x,\ \Delta y,\ \Delta z$ are cell widths

A value of 0.5 or smaller is generally recommended.

Warning: The way that parsing of the courant safety factor is currently handled, the default value is fixed to 1.0 for all values (which is too large for this method). Some near-term modifications are planned that will resolve this issue.

Parameter
Method : mhd_vlct : time_scheme

Summary
name of the time-integration scheme to use

Type
string

Default
vl

Scope
Enzo

Name of the time integration scheme to use. The recommended choice is `"vl"`, which corresponds to the default 2-stage predictor-corrector scheme. This should generally be used with a courant factor satisfying $C \leq 0.5$.

At present, the only other option is `"euler"`, which just updates the MHD fields in a single-stage. It is **ONLY INTENDED FOR TESTING PURPOSES** and at the time of writing this documentation, it has not been rigorously

tested. When using this choice, make sure to pass `"nn"` to the *reconstruct_method* parameter. This scheme should generally be compatible with courant factors satisfying $C \leq 1$.

In the future, we may add additional options to this parameter to support higher order Runge-Kutta integration schemes.

Parameter
Method : mhd_vlct : riemann_solver

Summary
name of the Riemann solver to use

Type
string

Default
hlld

Scope
Enzo

Name of the Riemann solver to use. For a list of options, see *riemann solvers*

Parameter
Method : mhd_vlct : reconstruct_method

Summary
name of the reconstruction method

Type
string

Default
plm

Scope
Enzo

Name of the interpolation method used to reconstruct face-centered primitives for computing the fluxes. For a list of options, see *reconstruction*

Note: When *time_scheme* is `"vl"`, this has no effect on the predictor stage (aka the half-timestep); it only affects the reconstruction method in the second stage. The predictor stage **MUST** use nearest-neighbor reconstruction.

Parameter
Method : mhd_vlct : theta_limiter

Summary
controls the dissipation of certain slope limiters.

Type
float

Default
1.5

Scope
Enzo

Modifies the disipation of the slope limiter of the "plm"/"plm_enzo" piecewise linear reconstruction algorithm. For more details, see *reconstruction*

Parameter
Method : mhd_vlct : half_dt_reconstruct_method

Summary
name of the reconstruction method to use for the full timestep

Type
string

Default
nn

Scope
Enzo

Name of the interpolation method used to reconstruct face-centered primitives for the first half timestep. "nn" is recommended for this method (problems arise if "plm" or "plm_athena" are used). For a list of options, see *reconstruction*

> **Warning:** This parameter is deprecated and will be removed in a future version. It only carried meaning while using the original predictor-corrector time integration scheme.
>
> Furthermore, this parameter only conveyed the illusion of choice. In reality, the integrator ONLY worked when this was set to "nn". For that reason, this parameter is not replaced.

Parameter
Method : mhd_vlct : full_dt_reconstruct_method

Summary
name of the reconstruction method to use for the full timestep

Type
string

Default
plm

Scope
Enzo

Name of the interpolation method used to reconstruct face-centered primitives for the full timestep. For a list of options, see *reconstruction*

> **Warning:** This parameter is deprecated and will be removed in a future version. The *reconstruct_method* parameter is a direct replacement.

Deprecated mhd_vlct parameters

The following parameters have all been deprecated and will be removed in a future version of Enzo-E. Going forwards, the corresponding parameters in Physics:fluid_props should be used instead.

Parameter
Method : mhd_vlct : dual_energy

Summary
Whether to use dual-energy formalism

Type
logical

Default
false

Scope
Enzo

Whether to use the dual-energy formalism.

Parameter
Method : mhd_vlct : dual_energy_eta

Summary
Dual energy parameter eta

Type
float

Default
0.001

Scope
Enzo

Dual-energy formalism parameter.

Parameter
Method : mhd_vlct : density_floor

Summary
Lower limit on density

Type
float

Default
none

Scope
Enzo

Density floor, which must exceed 0. This is applied during reconstruction and quantity updates.

Parameter
Method : mhd_vlct : pressure_floor

Summary
Lower limit on thermal pressure

Type
float

Default
none

Scope
Enzo

Thermal pressure floor, which must exceed 0. This is applied during reconstruction and quantity updates.

m1_closure

Method:m1_closure parameters are used to initialize parameters for Enzo-E's multigroup M1 Closure radiative transfer solver.

Parameter
Method : m1_closure : N_groups

Summary
The number of groups to define

Type
integer

Default
1

Scope
Enzo

This parameter specifies the number of energy groups to use for radiative transfer. The transport equation will be solved separately for each group. N_groups must be >= 1.

Parameter
Method : m1_closure : energy_lower

Summary
Lower bin edges of groups in eV

Type
list (float)

Default
range(1.0, 100.0, 100.0/N_groups)

Scope
Enzo

This parameter specifies the lower bounds for energy groups in eV. Bins are not required to be contiguous. Lower bounds are inclusive.

Parameter
Method : m1_closure : energy_upper

Summary
Upper bin edges of groups in eV

Type
list (float)

Default
range(100.0/N_groups, 100.0 + 100.0/N_groups, 100.0/N_groups)

Scope
Enzo

This parameter specifies the lower bounds for energy groups in eV. Bins are not required to be contiguous. Upper bounds are non-inclusive.

Parameter
Method : m1_closure : energy_mean

Summary
Group mean energy in eV

Type
list (float)

Default
0.5 * (energy_lower[i] + energy_upper[i])

Scope
Enzo

This parameter specifies the group mean energies in eV.

Parameter
Method : m1_closure : clight_frac

Summary
Speed-of-light fraction

Type
float

Default
1.0

Scope
Enzo

Speed of light fraction to use for radiative transfer.

Parameter
Method : m1_closure : photon_escape_fraction

Summary
Escape fraction of photons from gas around star particles

Type
float

Default
1.0

Scope
Enzo

Calculated photon densities for star particle radiation are multiplied by this value before being deposited onto the mesh.

Parameter
Method : m1_closure : courant

Summary
Courant number for radiative transfer

Type
float

Default
1.0

Scope
Enzo

The radiation timestep is calculated as dt = courant * dx/(3*clight_frac*C).

Parameter
Method : m1_closure : cross_section_calculator

Summary
Choose calculator for group-mean cross-sections

Type
string

Default
"vernier"

Scope
Enzo

Specifies which group-mean photoionization cross section to use. Options include:

- `"vernier"` Calculates the group-mean cross section using fits from `Vernier et al. (1996) <https://ui.adsabs.harvard.edu/abs/1996ApJ...465..487V/abstract>. The midpoint energy is used for each group (e.g. `energy = 0.5 * (E_lower + E_upper)` ).`
- `"vernier_average"` Same as `"vernier", with the added step of averaging the cross section over all star particles in the simulation, weighted by mass * luminosity, where luminosity in this case is converted to units of photons/s.`
- `"custom"` Specify cross sections explicitly in the parameter file using the *sigmaN* and *sigmaE* parameters.

Parameter
Method : m1_closure : flux_function

Summary
Choose flux function for radiative transfer

Type
string

Default
"GLF"

Scope
Enzo

Specifies which flux function to use for converting cell-centered fluxes to face-centered fluxes in the transport step. For the following functions, $\mathcal{F}$ and $\mathcal{U}$ are vectors of unknowns defined such that $\frac{\partial \mathcal{U}}{\partial t} + \nabla \mathcal{F}(\mathcal{U})$.

- `"GLF"` $\mathcal{F}_{i+1/2} = \frac{1}{2}(\mathcal{F}_i + \mathcal{F}_{i+1}) + \frac{c}{2}(\mathcal{U}_{i+1} + \mathcal{U}_i)$
- `"HLL"` $\mathcal{F}_{i+1/2} = \frac{\lambda^+ \mathcal{F}_i - \lambda^- \mathcal{F}_{i+1} + \lambda^+ \lambda^- (\mathcal{U}_{i+1} + \mathcal{U}_i)}{\lambda^+ - \lambda^-}$, where λ^+ and λ^- are eigenvalues of the Jacobian $\frac{\partial \mathcal{F}}{\partial \mathcal{U}}$. *Requires path to eigenvalue text file using the :p:`hll_file*` parameter.

Broadly speaking, "GLF" is more diffusive than "HLL", but is better at handling radiation from isotropic sources. On the other hand, "HLL" is better for simulating beams of radiation and shadows. See` Rosdahl et al. (2013) for a more detailed comparison.

Parameter
Method : m1_closure : hll_file

Summary
Path to text file containing table of eigenvalues

Type
string

Default
"hll_evals.list"

Scope
Enzo

Path to text file containing table of eigenvalues. A table with eiganvalues calculated by Gonzalez, Audut, & Huynh (2007) is provided in the `input/RadiativeTransfer` directory.

Parameter
Method : m1_closure : min_photon_density

Summary
Minimum photon density in CGS units

Type
float

Default
0.0

Scope
Enzo

Minimum photon density in units of cm^{-3}.

Parameter
Method : m1_closure : particle_luminosity

Summary
User-specified luminosity for star particles in erg/s

Type
float

Default
-1.0

Scope
Enzo

If particle_luminosity >= `0.0`, all star particles will be given the emission rate specified using this parameter. Otherwise, the *"luminosity"* particle attribute will be checked unless *radiation_spectrum* equals `"blackbody"` .

Parameter
Method : m1_closure : radiation_spectrum

Summary
Type of radiation spectrum for star particle

Type
string

Default
"custom"

Scope
Enzo

Options include:

- `"blackbody"` Calculates emission rate into each radiation group by integrating over a Planck function
- `"custom"` Specify SED explicitly in the parameter file using the SED parameter

---

Parameter
Method : m1_closure : attenuation

Summary
Whether to attenuate radiation

Type
bool

Default
true

Scope
Enzo

Whether to include attenuation in the radiative transport equation. Requires color fields to be defined for six-species chemistry (HI, HII, HeI, HeII, HeIII, and e^-.) If no density fields are defined, the attenuation calculation will be skipped by default.

Parameter
Method : m1_closure : thermochemistry

Summary
Whether to include thermochemistry

Type
bool

Default
true

Scope
Enzo

Whether to include thermochemistry. If thermochemistry == true, photoionization and heating rates are calculated and stored in the following fields: *"RT_HI_ionization_rate"*, *"RT_HeI_ionization_rate"*, *"RT_HeII_ionization_rate"*, and *"RT_heating_rate"*. The actual updates to species fields and the solving of the energy equation are handled by Grackle. As such, this method must be run in tandem with Grackle using Method:grackle:with_radiative_transfer = 1.

Parameter
Method : m1_closure : recombination_radiation

Summary
Whether to include recombination radiation

Type
bool

Default
"false"

Scope
Enzo

Whether to source photons from recombination radiation. Ignoring recombination radiation is known as the on-the-spot approximation. This is valid in gas that is optically thick to ionizing radiation.

Parameter
Method : m1_closure : lyman_werner_background

Summary
Whether to include a Lyman-Werner background

Type
bool

Default
"false"

Scope
Enzo

Whether to include an H2-photodissociating Lyman-Werner background. Requires radiation group 0 to be defined corresponding to energies in the Lyman-Werner band (11.18-13.6 eV). A constant intensity can be specified using the *LWB_J21* parameter. If *LWB_J21* is not set, the intensity will be calculated using the redshift-dependent polynomial fit defined in Equation 16 of Wise et al. (2012).

Parameter
Method : m1_closure : LWB_J21

Summary
Intensity of the LW background in units of 1e-21 erg s^-1 cm^-2 Hz^-1 sr^-1

Type
float

Default
-1.0

Scope
Enzo

Intensity of the LW background in units of 1e-21 erg s^-1 cm^-2 Hz^-1 sr^-1.

Parameter
Method : m1_closure : H2_photodissociation

Summary
Whether to include H2 photodissociation from LW radiation

Type
bool

Default
"false"

Scope
Enzo

Whether to include an H2 photodissociation from Lyman-Werner radiation. If true, H2 photodissication rates are calculated and stored in a field called "RT_H2_dissociation_rate". Requires radiation group 0 to be defined corresponding to energies in the Lyman-Werner band (11.18-13.6 eV).

Parameter
Method : m1_closure : SED

Summary
User-specified SED for radiating point sources

Type
list (float)

Default
Flat spectrum, where each group is given an emission rate of 1.0/N_groups

Scope
Enzo

User-specified SED for radiating point sources. This is a list, where each entry corresponds to an energy fraction to inject into each group every timestep. For example, if SED = [0.1, 0.6, 0.3], the emitted radiation will be split into three groups such that L1 = 0.1*L, L2 = 0.6*L, and L = 0.3*L, where L is the total luminosity of the particle.

Parameter
Method : m1_closure : sigmaE

Summary
User-specified group-mean cross sections in cm^2, averaged by energy

Type
list (float)

Default
0.0

Scope
Enzo

User-specified group-mean cross sections, averaged by energy. Requires *cross_section_calculator* to be set to *"custom"*. If this parameter is set, *sigmaN* must also be set. This list has length *N_groups* * 3 (*N_groups* * 4 if *H2_photodissociation* is `true`), where the number 3 (or 4) represents the number of chemical species (HI, HeI, HeII, and optionally H2I). For example, a simulation could use three radiation groups with *energy_mean* = [21.62, 30.0, 60.0]. In this case, setting sigmaE = [1.78e-18,0.0,0.0, 7.03e-19,5.36e-18,0.0, 9.19e-20,1.37e-18,1.22e-18] would produce the same cross sections as the fits from *Vernier et al. (1996) <https://ui.adsabs.harvard.edu/abs/1996ApJ…465..487V/abstract>_*.

Parameter
Method : m1_closure : sigmaN

Summary
User-specified group-mean cross sections in cm^2, averaged by photon density

Type
list (float)

Default
0.0

Scope
Enzo

User-specified group-mean cross sections, averaged by photon number. Requires *cross_section_calculator* to be set to *"custom"*. If this parameter is set, *sigmaE* must also be set. This list has length *N_groups* * 3 (*N_groups* * 4 if *H2_photodissociation* is `true`), where the number 3 (or 4) represents the number of chemical species (HI, HeI, HeII, and optionally H2I). For example, a simulation could use three radiation groups with *energy_mean* = [21.62, 30.0, 60.0]. In this case, setting *sigmaE* = [1.78e-18,0.0,0.0, 7.03e-19,5.36e-18,0.0, 9.19e-20,1.37e-18,1.22e-18] would produce the same cross sections as the fits from *Vernier et al. (1996) <https://ui.adsabs.harvard.edu/abs/1996ApJ…465..487V/abstract>_*.

Parameter
Method : m1_closure : temperature_blackbody

Summary
User-specified blackbody temperature for radiating point sources

Type
float

Default
0.0

Scope
Enzo

User-specified blackbody temperature for radiating point sources. Requires *radiation_spectrum* to be set to `"blackbody"`.

---

null

Parameter
Method : null : dt

Summary
Set the time step for the "null" Method

Type
float

Default
max (float)

Scope
Enzo

Sets the time step for the null Method. This is typically used for testing the AMR meshing infrastructure without having to use any specific method. It can also be used to add an additional maximal time step value for other methods.

pm_deposit

Parameter
Method : pm_deposit : alpha

Summary
Compute the total gravitating density field at time t + alpha*dt

Type
float

Default
0.5

Scope
Enzo

Sets the factor defining at what time to deposit mass into the density_total field. The default is 0.5, meaning density_total is computed at t + 0.5*dt.

ppm

Method:ppm parameters are used to initialize parameters for Enzo-E's PPM hydrodynamics method.

Parameter
Method : ppm : courant

Summary
Courant safety factor

Type
float

Default
1.0

Scope
Enzo

The method-specific courant safety factor. The method's minimum timestep is the minimum value of the following expression (which is computed for all cells):

$$C \times \left(\frac{c_s + |v_x|}{a\,\Delta x} + \frac{c_s + |v_y|}{a\,\Delta y} + \frac{c_s + |v_z|}{a\,\Delta z} \right)^{-1}$$

in which:

- C is the courant factor
- a is the cosmological scale factor
- c_s is the local sound speed
- $v_x,\ v_y,\ v_z$ are the velocity components
- $\Delta x,\ \Delta y,\ \Delta z$ are cell widths

A value of 0.8 or smaller is generally recommended.

Warning: The way that parsing of the courant safety factor is currently handled, the default value is fixed to 1.0 for all values (which is too large for this method). Some near-term modifications are planned that will resolve this issue.

Parameter
Method : ppm : diffusion

Summary
PPM diffusion parameter

Type
logical

Default
false

Scope
Enzo

PPM diffusion parameter.

Parameter
Method : ppm : flattening

Summary
PPM flattening parameter

Type
integer

Default
3

Scope
Enzo

PPM flattening parameter.

Parameter
Method : ppm : minimum_pressure_support_parameter

Summary
Enzo's MinimumPressureSupportParameter

Type
integer

Default
100

Scope
Enzo

Enzo's MinimumPressureSupportParameter parameter. This is meaningless unless *use_minimum_pressure_support* is set to `true`.

Parameter
Method : ppm : pressure_free

Summary
Pressure-free flag

Type
logical

Default
false

Scope
Enzo

Pressure-free flag.

Parameter
Method : ppm : steepening

Summary
PPM steepening parameter

Type
logical

Default
false

Scope
Enzo

PPM steepening parameter.

Parameter
Method : ppm : use_minimum_pressure_support

Summary
Minimum pressure support

Type
logical

Default
false

Scope
Enzo

Enzo's UseMinimumPressureSupport parameter.

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, they may undergo artificial fragmentation and angular momentum non-conservation. This parameter can be used to turn on a very simple fudge described in Machacek, Bryan & Abel (2001) in order to alleviate this problem.

When using minimum-pressure support, a floor is applied to the specific internal energy (or equivalently, temperature) to blocks with a refinement level equivalent to *Adapt:max_level*. A floor is applied in order to satisfy the inequality: $\lambda_J \geq \sqrt{K}\,\Delta x$, where:

- $\lambda_J = c_s\sqrt{\pi/(G\rho)}$ is the Jeans-length
- K encodes the value assigned to *minimum_pressure_support_parameter*
- Δx encodes the cell-width.

In other words, the minimum pressure-support ensures that the Jeans length is larger than the cell-width by a factor that is at least the square-root of *minimum_pressure_support_parameter*.

Deprecated ppm parameters

The following parameters have all been deprecated and will be removed in a future version of Enzo-E. Going forwards, the corresponding parameters in Physics:fluid_props should be used instead.

Parameter
Method : ppm : density_floor

Summary
Lower limit on density

Type
float

Default
1.0e-6

Scope
Enzo

Density floor, which replaces Enzo's "tiny_number".

Parameter
Method : ppm : dual_energy

Summary
Whether to use dual-energy formalism

Type
logical

Default
false

Scope
Enzo

Whether to use the dual-energy formalism.

Parameter
Method : ppm : dual_energy_eta_1

Summary
Dual energy parameter eta 1

Type
float

Default
0.001

Scope
Enzo

First dual-energy formalism parameter.

Parameter
Method : ppm : dual_energy_eta_2

Summary
Dual energy parameter eta 2

Type
float

Default
0.1

Scope
Enzo

Second dual-energy formalism parameter.

Parameter
Method : ppm : mol_weight

Summary
Enzo's Mu parameter

Type
float

Default
0.6

Scope
Enzo

Enzo's Mu molecular weight parameter.

Parameter
Method : ppm : number_density_floor

Summary
Lower limit on number density

Type
float

Default
1.0e-6

Scope
Enzo

Number density floor, which replaces Enzo's "tiny_number".

Parameter
Method : ppm : pressure_floor

Summary
Lower limit on pressure

Type
float

Default
1.0e-6

Scope
Enzo

Pressure floor, which replaces Enzo's "tiny_number".

Parameter
Method : ppm : temperature_floor

Summary
Lower limit on temperature

Type
float

Default
1.0e-6

Scope
Enzo

Temperature floor, which replaces Enzo's "tiny_number".

sink_maker

Parameter
Method : sink_maker : jeans_length_resolution_cells

Summary
Determines how many cell widths are required to resolve the local Jeans length for a cell not to form a sink.

Type
float

Default
4.0

Scope
Enzo

If the local Jeans length in a cell is less than this quantity multiplied by the maximum cell width, then the cell is a candidate for forming a sink. The maximum cell width is maximum value out of hx, hy, and hz, where hx, hy, and hz are the cell widths across the x-, y- and z-axes, respectively.

Parameter
Method : sink_maker : physical_density_threshold_cgs

Summary
The minimum physical density required for a cell to form a sink particle in cgs units.

Type
float

Default
1.0e-24

Scope
Enzo

The value of the physical density threshold in cgs units. The density in a cell must be greater than the density threshold to be able to form a sink. The density in a cell after sink formation will be no less than the density threshold. The value of the density threshold in code units must be greater than or equal to the value of the density floor imposed by the hydro method.

Parameter
Method : sink_maker : max_mass_fraction

Summary
The maximum fraction of a cell's gas mass which can be turned into a sink particle in one timestep.

Type
float

Default
0.25

Scope
Enzo

The mass of a newly-formed sink is bounded above by this parameter multiplied by the cell density multiplied by the cell volume. The value of this parameter must be between 0 and 1.

Parameter
Method : sink_maker : min_sink_mass_solar

Summary
The minimum mass of a newly-formed sink particle, in solar mass units.

Type
float

Default
0.0

Scope
Enzo

The minimum mass of a newly-formed sink particle, in solar mass units. If there is not enough gas mass in a cell to form a sink with at least this mass, no sink is formed.

Parameter
Method : sink_maker : check_density_maximum

Summary
Determines whether a cell is required to be a local density maximum in order to form a sink particle.

Type
logical

Default
true

Scope
Enzo

If true, then a cell will only form a sink particle if its density is larger than the density in all 26 neighboring cells.

Parameter
Method : sink_maker : max_offset_cell_fraction

Summary
Controls the size of the random displacement of a sink particle's initial position relative to the center of the cell

Type
float

Default
0.0

Scope
Enzo

When a cell creates a sink particle, the x/y/z coordinate of its initial position will be the x/y/z coordinate of the center of the cell, plus a random value generated from a uniform distribution on the interval [-A,A], where A is equal to this parameter multiplied by the cell width along the x/y/z axis.

Parameter
Method : sink_maker : offset_seed_shift

Summary
Seed used to generate the random displacement of a sink particle's initial position relative to the center of the cell

Type
integer

Default
0

Scope
Enzo

When computing the random offset for the initial position of a sink particle, we compute an unsigned 64 bit integer value from the cycle number, the block index, and the cell index, and then add on this value to give the seed for the random number generator.

star_maker

Method:star_maker parameters.

Parameter
Method : star_maker : flavor

Summary
Which star_maker method to use

Type
string

Default
STARSS

Scope
Enzo

Options: "STARSS", "stochastic"

Parameter
Method : star_maker : use_density_threshold

Summary
Use number density threshold for star formation

Type
logical

Default
false

Scope
Enzo

Flag to enable number density threshold for star formation.

Parameter
Method : star_maker : number_density_threshold

Summary
Use number density threshold for star formation

Type
logical

Default
false

Scope
Enzo

Set number density threshold for star formation in units of cm^-3. Requires "use_density_threshold"=true.

Parameter
Method : star_maker : use_overdensity_threshold

Summary
Use overdensity threshold

Type
logical

Default
false

Scope
Enzo

Flag to enable overdensity threshold for star formation. Currently only valid for cosmology simulations.

Parameter
Method : star_maker : overdensity_threshold

Summary
Use overdensity threshold for star formation

Type
float

Default
0.0

Scope
Enzo

Set overdensity threshold for star formation. Requires "use_overdensity_threshold"=true.

Parameter
Method : star_maker : use_velocity_divergence

Summary
Use converging flow criterion for star formation

Type
logical

Default
false

Scope
Enzo

Flag to check whether div(V) < 0

Parameter
Method : star_maker : use_cooling_time

Summary
Check if cooling_time < dynamical_time for star formation

Type
logical

Default
false

Scope
Enzo

Flag to check if cooling_time < dynamical_time

Parameter
Method : star_maker : use_temperature_threshold

Summary
Use temperature threshold for star formation

Type
logical

Default
false

Scope
Enzo

Flag to enable temperature threshold check for star formation

---

Parameter
Method : star_maker : temperature_threshold

Summary
Temperature threshold for star formation

Type
float

Default
1e4

Scope
Enzo

Set temperature threshold required for star formation. Requires "use_temperature_threshold"=true.

Parameter
Method : star_maker : use_self_gravitating

Summary
Use FIRE2 virial parameter criterion for star formation

Type
logical

Default
false

Scope
Enzo

Checks that alpha < 1, where alpha is the virial parameter calculated using the FIRE-2 prescription. See Appendix C of Hopkins et al. (2018).

Parameter
Method : star_maker : use_altAlpha

Summary
Use alternate virial parameter criterion for star formation

Type
logical

Default
false

Scope
Enzo

Checks that alpha < 1, where alpha is the virial parameter calculated as "potential/total_energy". Currently only accessed by EnzoMethodStarMakerSTARSS.

Parameter
Method : star_maker : use_h2_self_shielding

Summary
Use H2 self-shielding criterion for star formation

Type
logical

Default
false

Scope
Enzo

Checks that f_shield < 0, where f_shield is the H2 self-shielded fraction calculated using fits from Krumholz & Gnedin (2011).

Parameter
Method : star_maker : use_jeans_mass

Summary
Use Jeans mass criterion for star formation

Type
logical

Default
false

Scope
Enzo

Checks that cell_mass > max(jeans_mass, 1000 Msun) in a cell.

Parameter
Method : star_maker : critical_metallicity

Summary
Metallicity threshold for star formation

Type
float

Default
0.0

Scope
Enzo

Set metallicity threshold required for star formation

Parameter
Method : star_maker : maximum_mass_fraction

Summary
Max fraction of gas in a cell that can be converted into a star particle per formation event.

Type
float

Default
0.05

Scope
Enzo

Max fraction of gas in a cell that can be converted into a star particle per formation event.

Parameter
Method : star_maker : min_level

Summary
Minimum AMR level required for star formation.

Type
integer

Default
0

Scope
Enzo

Set minimum AMR level required for star formation.

Parameter
Method : star_maker : minimum_star_mass

Summary
Minimum star particle mass

Type
float

Default
0.0

Scope
Enzo

Set minimum star particle mass.

Parameter
Method : star_maker : maximum_star_mass

Summary
Maximum star particle mass

Type
float

Default
-1.0

Scope
Enzo

Set maximum star particle mass. For no limit, set "maximum_star_mass" < 0.

Parameter
Method : star_maker : turn_off_probability

Summary
Turn off probablistic elements of EnzoMethodStarMakerSTARSS.

Type
logical

Default
false

Scope
Enzo

Turn off probablistic elements of EnzoMethodStarMakerSTARSS. Mostly meant for debugging.

turbulence

Parameter
Method : turbulence : edot

Summary
Initial value for edot for turbulence Method

Type
float

Default
-1.0

Scope
Enzo

Todo
write

Parameter
Method : turbulence : mach_number

Summary
Value for Mach number in turbulence problem

Type
float

Default
0.0

Scope
Enzo

Todo
write

3.1.11 Monitor

3.1.12 Output

Output parameters are used to specify what types of disk output to perform and on what schedule.

Parameter
Output : list

Summary
List of output file sets

Type
list (string)

Default
[]

Scope
Cello

List of active file sets, each of which has its own associated Output : <file_set> : parameters. Any file set parameters associated with a file set not in the *list* parameter are ignored.

Parameter
Output : <file_set> : axis

Summary
Axis of projections for image output

Type
string

Default
none

Scope
Cello

Assumes
<file_set> is of type "image"

For the "image" output type, the axis along which to project the data for 3D problems. Values are *"x", "y", :e:`or* "z". See the associated type parameter.

Parameter
Output : <file_set> : schedule

Summary
Output schedule for the given file set

Type
subgroup

Default
none

Scope
Cello

See the *schedule* subgroup for parameters used to define when to perform output for the given file set.

Parameter
Output : <file_set> : colormap

Summary
Color map for image output

Type
list (float | string)

Default
["black", "white"]

Scope
Cello

Assumes
<file_set> is of type "image"

For the "image" output type, this parameter defines the colormap as a list of RGB values, such that the minimum field value is assigned the first color in the list, the maximum the last, and linear interpolated color values in between. If image_min or image_max parameters are defined, those are used instead for the respective min and max field values. The list must contain at least two colors, and the default is "black" and "white".

RGB values are specified in one of several ways. The original (and depreciated) way is as a triad of floating point numbers between 0.0 and 1.0, such that 1.0, 0.0, 0.0 represents red, 0.5, 0.5, 0.5 represents gray, etc. Another preferred way is named colors, e.g. "black" , "white" , "azure", etc. Any CSS3 extended color name should be accepted (lowercase). See CSS3 Extended Color list for a list of accepted color names. Also accepted are color specifications of the form #rrggbb, e.g. "#000000" for black, "#ffffff" for white, etc. (case independent).

RGB values in a given colormap can mix-and-match color representations; for example, ["black", 0.5, 0.5, 0.5, "#Abacab"] is a valid colormap.

Parameter
Output : <file_set> : field_list

Summary
List of fields to output

Type
list (string)

Default
[]

Scope
Cello

List of fields for this output file set. For "image" field types, the field list must contain exactly one field.

Parameter
Output : <file_set> : particle_list

Summary
List of particle types to output

Type
list (string)

Default
[]

Scope
Cello

List of particles types for this output file set..

Parameter
Output : <file_set> : name

Summary
File names

Type
list (string)

Default
“”

Scope
Cello

Assumes
<file_set> is *not* of type “restart”

This parameter specifies the names of files in the corresponding file_group. The first element is the file name, which may contain printf-style formatting fields. Subsequent values correspond to variables for the formatting fields, which may include “cycle”, “time”, “count” (a counter incremented each time output is performed), “proc” (the process rank), and “flipflop” (alternating 0 and 1, which can be useful for checkpoint directories). The file name should include an appropriate extension, e.g. “.png” for “image” output, and “.h5” or “.h5” for “data” output. Example: [“projection-%04d.png”, “cycle”].

Parameter
Output : <file_set> : dir

Summary
Name of the subdirectory for the output file

Type
list (string)

Default
“”

Scope
Cello

This parameter specifies the subdirectory for the output file. The first element is the file name, which may contain printf-style formatting fields. Subsequent values correspond to variables for the formatting fields, which may include “cycle”, “time”, “count” (a counter incremented each time output is performed), “proc” (the process rank), and “flipflop” (alternating 0 and 1, which can be useful for checkpoint directories). Example: [“Checkpoint-%d”, “flipflop”].

This parameter is required for file groups of type “checkpoint”. While optional for other file types, the behavior is different for groups of type “data”. In that case, two extra files are output: <DIR>.file_list, which contains a list of all data files output, and <DIR>.block_list, which contains a list of all names of Blocks and the corresponding data file containing each Block.

Parameter
Output : <file_set> : stride_write

Summary
Subset of processors to perform write

Type
integer

Default
1

Scope
Cello

Assumes
<file_set> is of type "data"

Status
DEPRECIATED

This parameter allows for a strict subset of physical processors to output data, which is especially helpful for large process counts to reduce the load on parallel file systems. **This parameter is depreciated, since the "output" method is preferered for writing data output.**

Parameter
Output : <file_set> : stride_wait

Summary
Stride for sequencing processor data writes

Type
integer

Default
1

Scope
Cello

Assumes
<file_set> is of type "data"

Status
DEPRECIATED

This parameter allows for processes to write sequentially to prevent too many processes overloading the file system. A good starting point would be the number of processes in a shared memory node, in which case at most one process per node will be writing at any point in time. **This parameter is depreciated, since the "output" method is preferered for writing data output.**

Parameter
Output : <file_set> : type

Summary
Type of output files

Type
string

Default
"unknown"

Scope
Cello

The type of files to output in this output file set. Supported types include "image" (PNG file of 2D fields, or projection of 3D fields) and "data". For "image" files, see the associated colormap and axis parameters.

Parameter
Output : <file_set> : image_min

Summary
Data value associated with the first color in the colormap

Type
float

Default
0.0

Scope
Cello

Assumes
<file_set> is of type "image"

This parameter specifies the Field value associated with the first color in the file set's colormap.

Parameter
Output : <file_set> : image_max

Summary
Data value associated with the last color in the colormap

Type
float

Default
0.0

Scope
Cello

Assumes
<file_set> is of type "image"

This parameter specifies the Field value associated with the last color in the file set's colormap.

Parameter
Output : <file_set> : image_lower

Summary
Lower bound on domain to be output in image

Type
list (float)

Default
[min (float), min (float), min (float)]

Scope
Cello

Assumes
<file_set> is of type "image"

This parameter specifies the lower limit of the domain to include in the image. This can be used for imaging "slices" of 3D data, or zeroing in on interesting region of the domain.

Parameter
Output : <file_set> : image_upper

Summary
Upper bound on domain to be output in image

Type
list (float)

Default
[max (float), max (float), max (float)]

Scope
Cello

Assumes
<file_set> is of type "image"

This parameter specifies the upper limit of the domain to include in the image. This can be used for imaging "slices" of 3D data, or zeroing in on interesting region of the domain.

Parameter
Output : <file_set> : image_ghost

Summary
Whether to include ghost zones in the image

Type
logical

Default
false

Scope
Cello

Assumes
<file_set> is of type "image"

Setting the image_ghost to true will include ghost zone values in the image output. This is typically used only when debugging. The default is false.

Parameter
Output : <file_set> : image_reduce_type

Summary
How to handle 3D field data orthogonal to the image

Type
string

Default
"sum"

Scope
Cello

Assumes
<file_set> is of type "image"

When images are generated for 3D problems, multiple data values will be associated with each pixel in the image. This parameter defines how to handle these multiple values, including "sum", "min", "max", and, "avg". For field data the default of "sum" is appropriate, though for images of meshes "max" should be used.

Parameter
Output : <file_set> : image_face_rank

Summary
Whether to include neighbor markers in the mesh image output

Type
integer

Default
3

Scope
Cello

Assumes
<file_set> is of type "image"

This parameter is primarily used for debugging. Internally, each node in the mesh keeps track of the mesh level of its neighbors. This parameter includes a marker on each face colored according to the neighbor's level. The value of this parameter specifies the lower limit on the face "rank" (0 for corners, 1 for edges, 2 for faces). The default of 3 means no markers are displayed.

Parameter
Output : <file_set> : image_size

Summary
Set the size of the image

Type
list (integer)

Default
[0,0]

Scope
Cello

Assumes
<file_set> is of type "image"

Specify the size of the output image. By default it is sized to be one pixel per field value at the finest mesh level. This is useful to keep images from being to big for large problems, or too small for small problems (e.g. for mesh images which could otherwise be too small).

Parameter
Output : <file_set> : image_log

Summary
Whether to output the log of the data

Type
logical

Default
false

Scope
Cello

Assumes
<file_set> is of type "image"

If true, then the natural logarithm of the field value is used for mapping values to the colormap, otherwise use the original field value.

Parameter
Output : <file_set> : image_type

Summary
Type of image to write

Type
string

Default
"data"

Scope
Cello

Assumes
<file_set> is of type "image"

This parameter is used to control whether field values are used to generate the image, whether it's an image of the mesh structure, or a combination of both. Valid values are "data", "mesh", or "data+mesh".

Parameter
Output : <file_set> : image_block_size

Summary
Number of pixels for fine-level blocks in a mesh image

Type
integer

Default
1

Scope
Cello

Assumes
<file_set> is of type "image"

Status
DEPRECIATED

For images of meshes, this parameter defines how many pixels wide each finest-level block is in the image. This parameter and the image_size parameter should not both be set.

Parameter
Output : <file_set> : image_mesh_color

Summary
How to color blocks in a mesh image

Type
string

Default
"level"

Scope
Cello

Assumes
<file_set> is of type "image"

By default, blocks in mesh images are colored according to the level of the block. In addition to "level", other possible ways to assign colors to blocks include "process" and "age".

3.1.13 Particle

Cello supports any number of particle types–e.g. *"dark"* for dark matter particles, or *"trace"* for tracer particles. Each particle type in turn may have any number of attributes–e.g. *"x"* or *"position_x"* for position, *"vx"* or *"velocity_x"* for velocity, *"mass"*, *"id"*, etc. Attributes can have any basic floating-point or integer type.

All particle types must have at least attributes for position, defined using the *position* parameter. This allows Cello to know whether particles have moved off of a Block, and if so to relocate them to the correct new block.

Particle positions may be defined as integer types instead of floating-point. When a particle position attribute is defined as an integer, then the coordinate value is defined relative to the enclosed Block instead of a global coordinate system. This can be useful both to reduce memory usage, and to simultaneously improve accuracy–it avoids possible catastrophic cancellation errors that are especially large in "deep" Blocks in an AMR hierarchy whose position is far from 0. When positions are defined as integers, 0 is defined to be the center of the block, and [*-min-int* / 2 , *max-int* / 2) are the bounds of the Block, where *min-int* is the minimum value of the signed integer of the corresponding size. Integer types allowed include *"int8"*, *"int16"*, *"int32"*, and *"int64"*. Two byte integers *"int16"* should be sufficient for most simulations: it has a range of [-16384, 16384) within the particle's containing Block, and ranges [-32768, -16384) and [16384, 32768) on either side of the associated Block.

Particles are allocated and operated on in "batches". The *batch_size* parameter defines how many particles are in a batch. By operating on particles in batches, the frequency of memory operations is greatly reduced, and functions operating on particle attributes can be more efficient due to reduced overhead. It should also simplify writing particle methods to be executed on accelerators, such as NVIDIA or AMD GPU's.

Just as with fields, particle types can be assigned to *groups*.

Parameter
Particle : list

Summary
List of particle types

Type
list (string)

Default
[]

Scope
Cello

Cello allows arbitrary parameter types (dark matter particles, tracer particles, star particles, etc.), each with arbitrary attributes (position, velocity, etc.). The list parameter defines which types of particles to use.

```
Particle {

    list = ["dark", "trace"];

}
```

Parameter
Particle : batch_size

Summary
Number of particles in a "batch" of particles

Type
integer

Default
1024

Scope
Cello

Particles are allocated and operated on in *batches*. The number of particles in a batch is set using the batch_size parameter. The default batch size is 1024.

Parameter
Particle : particle_type : attributes

Summary
List of attribute names and data types

Type
list (string)

Default
none

Scope
Cello

Each particle type can have multiple attributes of varying types, which are defined by the attributes parameter. The attributes parameter is a list of strings, alternating between the name of the parameter, and its type. Names may include "position_x", "velocity_z", "mass", "id", etc. Types may include "single", "double", "quadruple", "int8", "int16", "int32", or "int64". Ordering of attributes in memory is as in the attributes parameter.

```
Parameter {

    list = ["trace", "dark"];

        trace {
```

(continues on next page)

(continued from previous page)

```
            attributes = ["id", "int64",
                          "x",  "single",
                          "y",  "single",
                          "z",  "single"];
        }

        dark {

            attributes = ["id",         "int64",
                          "mass",       "double",
                          "velocity_x", "single",
                          "velocity_y", "single",
                          "velocity_z", "single",
                          "position_x", "int16",
                          "position_y", "int16",
                          "position_z", "int16"];
      }
  }
```

Note that when attributes of multiple sizes are included in the same parameter type, it can be helpful to order the attributes so that larger-sized attributes are listed first, followed by smaller-sized attributes. This can help prevent allocating more memory than necessary, since attributes may be padded with unused bytes for correct memory alignment.

Parameter
Particle : particle_type : interleaved

Summary
Format of output files

Type
logical

Default
false

Scope
Cello

Particle attributes within a batch of particles may be stored in memory either particle-by-particle, or "interleaved" (attribute-by-attribute). If $a_{i,j}$ represents the jth attribute of particle i, then with interleaved = false, attributes would be stored as $a_{0,0} \ldots a_{m,0}, a_{0,1} \ldots a_{m,1} \ldots a_{0,n} \ldots a_{m,n}$. If, however, interleaved = true, then attributes would be stored as $a_{0,0} \ldots a_{0,n}, a_{1,0} \ldots a_{1,n} \ldots a_{m,0} \ldots a_{m,n}$. Non-interleaved particle attributes have array accesses of stride 1 and minimal storage overhead, but may not utilize cache well. Interleaved particle attributes *may* have improved cache utilization, but will have stride > 1, and may require memory padding for correct alignment of attributes in memory. The default is false.

Parameter
Particle : particle_type : group_list

Summary
Specify a list of groups that the Particle type belongs to

Type
list (string)

Default
[ ]

Scope
Cello

Different Particle types may belong to any number of different "groups", which allows simulation code to loop over multiple related particle types.

```
Particle {
    list = ["trace","dark","star"];

    dark { group_list = ["is_gravitating"]; }
    star { group_list = ["is_gravitating"]; }
}
```

This example can be rewritten as follows, which is completely equivalent:

```
Particle
    list = ["trace","dark","star"];
}

Group {
    list = ["is_gravitating"];
    is_gravitating {
       particle_list = ["dark","star"];
    }
}
```

Parameter
Particle : particle_type : position

Summary
Format of output files

Type
string

Default
""

Scope
Cello

Cello needs to know which particle attributes represent position, so that it can determine when particles migrate out of a Block and need to be moved to a neighboring Block. This is done using the position parameter:

```
Particle {

   list = ["trace"];

   trace {

      attributes = ["id",
                    "x","single",
                 "y","single",
```

(continues on next page)

(continued from previous page)

```
                  "z","single"];

      position = ["x","y","z"];
   }
}
```

Parameter
Particle : particle_type : velocity

Summary
Format of output files

Type
string

Default
“”

Scope
Cello

Enzo may need to know which particle attributes represent velocity, for example for kick() or drift() operations. This is done using the velocity parameter, whose usage is analogous to the position parameter. While specifying position is required, specifying velocity is optional.

```
Particle {

   list = ["dark"];

   trace {

      attributes = [ "x","single",   "y","single",   "z","single",
                     "vx","single",  "vy","single",  "vz","single",
                  "mass","single"];

      velocity = ["vx","vy","vz"];
   }
}
```

Parameter
Particle : mass_is_mass

Summary
Flag for whether particle masses should be handled as a quantity with dimensions of mass.

Type
any

Default
none

Scope
Cello

This parameter is not used by Enzo-E, but is rather used by yt to indicate whether particle "mass" should be treated as a quantity with dimensions of mass. If this parameter is absent, then "mass" is treated as a quantity with dimensions of density, which has a value equal to the true mass divided by the root level cell volume. The value of this parameter is not used: yt only checks whether this parameter is present, however, it must have some value which can be read in by yt. It is recommended to set the velue to be "true".

3.1.14 Performance

Parameter
Performance : warnings

Summary
Whether to output performance-related warnings

Type
logical

Default
true

Scope
Cello

If calls to the Performance API are incorrect, e.g. if stop_region() is called on a region that has not been started, then this parameter specifies whether or not to display warning messages

---

Parameter
Performance : papi : counters

Summary
List of PAPI counters

Type
list (string)

Default
[]

Scope
Cello

List of PAPI hardware performance counters to trace, e.g. 'counters = ["PAPI_FP_OPS", "PAPI_L3_TCA"];'. For a list of available counters, use the PAPI "papi_avail" utility.

3.1.15 Physics

Parameter
Physics : list

Summary
List of physics-sets

Type
list (string)

Default
[]

Scope
Cello

Specifies list of physics-sets, each of which has its own associated: Physics : <physics-set> : parameters. Any parameters associated with a physics-set subgroup that is not in this *list* parameter are ignored. Allowed values include:

- "cosmology" specifies cosmological constants.
- "fluid_props" specifies parameters related to gas properties.
- "gravity" specifies parameters related to gravity.

cosmology

Parameter
Physics : cosmology : comoving_box_size

Summary
Enzo's CosmologyComovingBoxSize parameter

Type
float

Default
64.0

Scope
Enzo

Enzo's CosmologyComovingBoxSize parameter.

Parameter
Physics : cosmology : hubble_constant_now

Summary
Hubble constant for Z=0

Type
float

Default
0.701

Scope
Enzo

Hubble constant for Z=0.

Parameter
Physics : cosmology : initial_redshift

Summary
Enzo's CosmologyInitialRedshift parameter.

Type
float

Default
20.0

Scope
Enzo

Enzo's CosmologyInitialRedshift parameter.

Parameter
Physics : cosmology : max_expansion_rate

Summary
Maximum expansion rate

Type
float

Default
0.01

Scope
Enzo

Maximum expansion rate.

Parameter
Physics : cosmology : omega_lamda_now

Summary
Omega lambda for Z=0

Type
float

Default
0.721

Scope
Enzo

Omega lamda for Z=0.

Parameter
Physics : cosmology : omega_matter_now

Summary
Omega matter for Z=0

Type
float

Default
0.279

Scope
Enzo

Omega matter for Z=0.

fluid_props

Physics:fluid_props parameters are used to specify general fluid properties for the gas. The majority of these parameters are further divided into the following 3 subgroups: `dual_energy`, `floors`, `eos`.

An example configuration is described below:

```
Physics {
   list = ["fluid_props"];
   fluid_props {

      dual_energy { type = "disabled"; }

      eos {
         gamma = 1.4;
      }

      floors {
         density  = 1e-10;
         pressure = 1e-10;
      }

      mol_weight = 0.6;
   }
}
```

Parameter
Physics : fluid_props : mol_weight

Summary
Specifies the gas's nominal mean molecular weight

Type
float

Default
0.6

Scope
Enzo

Specifies the gas's nominal mean molecular weight. This is used in operations like computing the temperature field. When the `"grackle"` method is in use, this parameter may not be used.

---

dual_energy

Physics:fluid_props:dual_energy parameters specify details about the dual-energy formalism (or the lack thereof). Significantly more detail about the different variants are provided in *dual-energy formalism*.

Parameter
Physics : fluid_props : dual_energy : type

Summary
specifies formulation of dual-energy formalism (if any)

Type
string

Default
disabled

Scope
Enzo

Specifies the formulation of the dual-energy formalism that the simulation uses (if any). Valid choices include:

- `"disabled"` The dual-energy formalism is not used. This is the default option.
- `"bryan95"` Enables the formulation of the dual-energy formalism that is described in Bryan et al (1995)
- `"modern"` Enables the formulation of the dual-energy formalism that is described in *Overview*

Parameter
Physics : fluid_props : dual_energy : eta

Summary
eta parameter(s) for the dual-energy formalism

Type
list (float)

Default
[]

Scope
Enzo

Specifies parameters used in the dual energy formalism. It expects a different number of parameters based on the value of Physics:fluid_props:dual_energy:type. The expectations are summarized below:

- `"disabled"`: This should always be an empty list.
- `"bryan95"`: The list should always contain two entries corresponding to `[eta_1, eta_2]`. An empty list (the default value) is treated as though it has the value `[0.001, 0.1]`.
- `modern"`: The list should always one entry corresponding to `[eta]`. An empty list (the default value) is treated as though it has the value `[0.001]`.

eos

Physics:fluid_props:dual_energy parameters specify details about the (caloric) equation of state. At this time, this only supports an ideal gas. But, in the future this will support alternative equations of state (like an isothermal equation of state).

See *Compatability with hydro/mhd solvers* for further discussion about how the equation of state is handled when *Method:grackle:primordial_chemistry* exceeds 1 (under these conditions Grackle models a spatially varying adiabatic index).

Parameter
Physics : fluid_props : eos : gamma

Summary
Adiabatic exponent

Type
float

Default
5.0 / 3.0

Scope
Enzo

gamma specifies the ratio of specific heats for the ideal gas used by the hydrodynamics solvers. This is used in a number other places like the `"grackle"` method, and various problem-initializers.

floors

Physics:fluid_props:floors parameters specify the floors that should be applied to different fluid quantities. Valid parameter values must be positive. Different methods selectively apply these floors. For more details, see *Floors*.

Parameter
Physics : fluid_props : floors : density

Summary
Lower limit on density field

Type
float

Default
none

Scope
Enzo

Floor to be applied to the mass density field.

Parameter
Physics : fluid_props : floors : pressure

Summary
Lower limit on thermal pressure

Type
float

Default
none

Scope
Enzo

Thermal pressure floor.

Parameter
Physics : fluid_props : floors : temperature

Summary
Lower limit on temperature

Type
float

Default
none

Scope
Enzo

Temperature floor, which must exceed 0. Note that this is specified with units of Kelvin (since the `"temperature"`` field is always measured in Kelvin in Enzo-E)`

Parameter
Physics : fluid_props : floors : metallicity

Summary
Minimum metallicity in solar units

Type
float

Default
0.0

Scope
Enzo

Minimum metallicity in solar units. This parameter is multiplied by the `enzo_constants::metallicity_solar` constant and the local value of the `"density"` field to give the local floor for the `"metal_density"` field

gravity

Parameter
Physics : gravity : grav_const_codeU

Summary
The gravitational constant in code units

Type
float

Default
none

Scope
Enzo

When this parameter is not specified, the gravitational constant is automatically taken to be the real-world reference value $G \approx 6.67 \times 10^{-8}\,\mathrm{cm}^3\,\mathrm{g}^{-1}\mathrm{s}^{-2}$ (the actual value used within the codebase is not quoted here in case we ever update it). In this case, the conversion between cgs units and code-units are handled internally.

Note: In most cases, users should not need to specify this parameter at all (so that the appropriate default value is used). This parameter mostly exists to help simplify some test problems in non-cosmological simulations.

Users are **NOT** allowed to specify this parameter in cosmological simulations. This is because cosmological code-units are defined such that $4\pi G\bar{\rho}$ has the value `1.0`, where $\bar{\rho}$ is the mean physical matter density of the universe.

At the time of writing this documentation, users can force the usage of the default value by assigning this parameter a non-positive value. However this is an implementation-detail that may change in the future (if it does change, an error will be raised).

3.1.16 schedule

"schedule" is a parameter *subgroup* that defines when to do something, such as perform output, apply a method, or to apply the dynamic load balancer. Schedules can be specified as either

- a *list* of values
- an interval of values specified using some subset of *start*, *schedule:stop*, and *step*.

The associated variable, set using *var*, can be "cycle", "time", or "seconds". Here "time" refers to simulation time, and "seconds" to wall-clock time. At each cycle, all schedules are checked to see if the cycle number, simulation time or wall-clock seconds match the list or interval of values. If there is a match, the associated output or is performed; otherwise, it is skipped.

Note that when simulation "time" is specified, then the simulation's time step may be reduced so that the corresponding output occurs exactly at the specified time.

```
Output {


   list = ["check", "dump", "image"];

   check {

      # **** write a checkpoint every 100.0 seconds ****

      schedule {
         var = "seconds";
         start = 100.0;
         step =  100.0;
      }
       ...
   }

   dump {

      # **** perform a data dump every 50 cycles until cycle 1000 ****

      schedule {
         var = "cycle";
         step =   50;
         stop = 1000;
       }
        ...
   }

   image {
```

(continues on next page)

(continued from previous page)

```
      # **** write an image at times t = 1.0,  2.0, and 5.0 ****

      schedule {
         var = "time";
         list = [1.0, 2.0, 5.0];
       }
        ...
   }
}
```

Parameter
schedule : var

Summary
Variable associated with scheduling for the given file set

Type
string

Default
"none"

Scope
Cello

The var parameter specifies what value is checked at each cycle, which may be "cycle", "time", or "seconds" Here "time" refers to simulation time, and "seconds" to wall-clock time. Note that when simulation "time" is specified, the simulation's time step may be reduced such that the corresponding output occurs exactly at the specified time.

Parameter
schedule : list

Summary
List of scheduled values for the specified variable

Type
[list (integer) | list (float)]

Default
[]

Scope
Cello

This parameter specifies a list of values to check against for output with respect to cycle, time, or seconds. If the *var* parameter associated with the schedule is "cycle", then value must be a list of integers; otherwise, value must be a list of float's The default is an empty list.

Parameter
schedule : start

Summary
Starting value for scheduled interval

Type
[integer | float]

Default
0 | 0.0

Scope
Cello

Todo
write

Parameter
schedule : stop

Summary
Last value for scheduled interval

Type
[integer | float]

Default
max (integer) | max (double)

Scope
Cello

Todo
write

Parameter
schedule : step

Summary
Stepping increment for interval

Type
[integer | float]

Default
1 | 1.0

Scope
Cello

Todo
write

3.1.17 Solver

Parameter
Solver : solver : iter_max

Summary
Iteration limit for the CG solver

Type
int

Default
100

Scope
Enzo

Maximum number of CG iterations to take.

Parameter
Solver : solver : res_tol

Summary
Residual norm reduction tolerance for the CG solver

Type
float

Default
1e-6

Scope
Enzo

Stopping tolerance on the 2-norm of the residual relative to the initial residual, i.e. CG is defined to have converged when $||R_i||_2 / ||R_0||_2 < res_tol$.

Parameter
Solver : solver : grav_const

Summary
Gravitational constant

Type
float

Default
6.67384e-8

Scope
Enzo

Gravitational constant used in place of G. The default is G in cgs units.

Parameter
Solver : solver : diag_precon

Summary
Whether to apply diagonal preconditioning

Type
logical

Default
false

Scope
Enzo

Whether to diagonally precondition the linear system A*X = B in EnzoSolverGravityCg by 1.0 / (h^2).

Parameter
Solver : solver : monitor_iter

Summary
How often to display progress

Type
integer

Default
1

Scope
Enzo

The current iteration, and minimum, current, and maximum relative residuals, are displayed every monitor_iter iterations. If monitor_iter is 0, then only the first and last iteration are displayed.

3.1.18 Stopping

Parameter
Stopping : cycle

Summary
Stopping cycle

Type
integer

Default
max (integer)

Scope
Cello

Stopping cycle.

Parameter
Stopping : time

Summary
Stopping time

Type
float

Default
max (double)

Scope
Cello

Stopping time.

Parameter
Stopping : seconds

Summary
Stop after this number of seconds (wall-clock time)

Type
float

Default
max (double)

Scope
Cello

End the calculation after this many seconds of wall-clock time.

Parameter
Stopping : interval

Summary
Stopping interval

Type
integer

Default
1

Scope
Cello

Number of cycles between applying the stopping criteria.

3.1.19 Testing

Parameter
Testing : cycle_final

Summary
Enzo-E unit test parameter for expected final cycle number

Type
integer

Default
0

Scope
Cello

Enzo-E unit test parameter for expected final cycle number.

Parameter
Testing : time_final

Summary
Enzo-E unit test parameter for expected final time

Type
float

Default
0.0

Scope
Cello

Enzo-E unit test parameter for expected final time.

Parameter
Testing : time_tolerance

Summary
Tolerance on the absolute error between actual final time and time_final

Type
float

Default
1.0e-6

Scope
Cello

Enzo-E unit test parameter for tolerance on the expected final time.

3.1.20 Units

Parameter
Units : length

Summary
Units scaling factor for length

Type
double

Default
1.0

Scope
Cello

Units scaling factor for length.

Parameter
Units : mass

Summary
Units scaling factor for mass

Type
double

Default
1.0

Scope
Cello

Units scaling factor for mass. Only one of mass and density Units parameters can be initialized to 0.

Parameter
Units : time

Summary
Units scaling factor for time

Type
double

Default
1.0

Scope
Cello

Units scaling factor for time.

Parameter
Units : density

Summary
Units scaling factor for density

Type
double

Default
1.0

Scope
Cello

Units scaling factor for density. Only one of mass and density Units parameters can be initialized to 0.

CHAPTER

FOUR

DEVELOPER GUIDE

[This page is under development]

This "Developing with Cello" section describes how to develop applications to use Cello's scalable adaptive mesh refinement, including adding new computational methods, initial conditions, refinement criteria, etc.

4.1 Enzo-E Coding Guidelines

[This page is under development: *last updated 2023-07-13* **]**

This page describes current coding guidelines for Enzo-E development. It is a working document and always open to suggestions for changes.

4.1.1 Naming files

All Enzo-E-related files should be placed in an appropriate subdirectory of `src/Enzo` (the Enzo-E source code directory). File paths have the form `src/Enzo/`*component*`/` *class*`.[hc]pp`, where

- *component* is the organizing directory from the Enzo-Layer. You can investigate
- *class* is the class-name (see below for Enzo-E class naming guidelines)
- and the `.hpp` extension is for header (declaration) files while `.cpp` is for source (definition) files.

Some examples are shown below for classes that nicely fall into the Cello class-hierarchy:

Table 1: Sample Enzo-E file-naming

File type	Enzo-E file name	Other comments [1]
Physics method	`src/Enzo/`*component*`/EnzoMethod`*Name*`.[hc]pp`	these are split out over a number of different components
Linear solver	`src/Enzo/`*component*`/EnzoSolver`*Name*`.[hc]pp`	currently, all solvers are located in the `gravity/solvers` (since they are currently just used by the Gravity Solver)
Initial conditions	`src/Enzo/`*component*`/EnzoInitial`*Name*`.[hc]pp`	currently, these are mostly located in `initial` or `tests` (the latter is for initializers that are just used by test-problems)
Boundary conditions	`src/Enzo/`*component*`/EnzoBoundary`*Name*`.[hc]pp`	currently, there is one instance of this class within `enzo-core`
Interpolation	`src/Enzo/`*component*`/EnzoProlong`*Name*`.[hc]pp`	currently, these are all located inside of `mesh`
Restriction	`src/Enzo/`*component*`/EnzoRestrict`*Name*`.[hc]pp`	currently, these are all located inside of `mesh`

[1] The comments in this table may become somewhat outdated over time. Your best bet is to always look at the actual file structure in the source code

All Cello-related files should be in the `src/Cello` subdirectory (it is organized has a flat directory structure). File names should have the form *component_class*`.[hc]pp`, where *component* is the high-level code component (e.g. "`problem`", "`io`", "`mesh`"), *class* is the class-name, and extension `.hpp` for header (declaration) files and `.cpp` for source (definition) files.

4.1.2 Naming classes

Enzo-E classes all begin with `Enzo`.

Subclasses of the Cello-Hierarchy

For functionality to be executed within a simulation, that functionality must be executed by a subclass of a Cello class. For that reason, most classes in the Enzo-layer are subclasses of a Cello class. Such subclasses all begin with `Enzo`, followed by the (capitalized) general type indicating the type of Cello class ("`Method`", "`Solver`", "`Initial`" for initial conditions, etc.), followed by the specific name for the class type. For example, the Conjugate Gradient (CG) linear solver class is named `EnzoSolverCg`. Note that all code implementing an Enzo-E class should (generally) live in the *class-name*`.hpp` and *class-name*`.cpp` files (more discussion about file naming can be found in the previous section)

Class type	Enzo-E class name	Cello base class
Physics method	`EnzoMethod`*Name*	`Method`
Linear solver	`EnzoSolver`*Name*	`Solver`
Initial conditions	`EnzoInitial`*Name*	`Initial`
Boundary conditions	`EnzoBoundary`*Name*	`Boundary`
Interpolation	`EnzoProlong`*Name*	`Prolong`
Restriction	`EnzoRestrict`*Name*	`Restrict`

Classes outside of the Cello-Hierarchy

There's a temptation to try to squeeze everything into a subclass extending the Cello-class hierarchy. For that reason, it's worth emphasizing that developers should feel free to implement classes that are not part of this hierarchy. In these cases, the convention is generally for the class name to start with `Enzo` and for the rest of the name to avoid causing confusion with names in the above table.

This is especially useful to consider when implementing complex functionality that involve a lot of code. There are certain cases where this can this can lead to massive classes (the implementation is well over 1000 lines). In such cases, it can be hard to keep track of everything that is going on (especially if you are not the original developer of that code). In these cases, there are sometimes opportunities where you can define a separate helper class that does a good job abstracting some subset of this functionality.

The `EnzoMethodM1Closure` class provides examples of where this is done. This class requires a data table to be read in from an external file. Rather than implementing the functionality to read the table and access the table data (after it has been read into memory) within the `EnzoMethodM1Closure` class, this functionality is part of the `M1Tables` class. Other examples arise in `EnzoMethodMHDVlct` and `EnzoInitialInclinedWave`.

Note: Developers are encouraged to put classes into their own files. (Of course, exceptions can be made - especially if the class is really tiny).

However, there are a handful of examples in the codebase where this is not the case. Sometimes the header and source file dedicated to a class inheriting from a Cello-class will contain declarations/definitions of other classes (that are used to help implement the primary class). Such examples arose for historical reasons: the enzo-layer was previously organized with a flat directory structure, which made it hard to identify groups of files that implemented functionality that was used together.

In some of these cases, the names of the secondary classes are not prefixed with `Enzo`. This should be avoided going forward (especially if the class is defined/implemented in its own file/header pair).

The primary reason for the `Enzo` prefix is to make it easier to differentiate between classes from the `Cello` and `Enzo` layers at a glance.

4.1.3 Naming class methods

Methods (functions associated with a specific class) are generally named beginning with a lower-case letter, and underscores for spacing. Public methods end in an underscore _.

Method type	Method name
public methods	`a_public_thing()`
private methods	`a_private_thing_()`
Charm++ entry methods	`p_blah()`
Charm++ reduction entry methods	`r_reduce()`

Note that Charm entry methods have very different behavior than regular C++ methods–they are (usually) asynchronous, return immediately, and are called using a Charm++ proxy to a class typically residing on a different processing element in a different memory space. So it's important to name entry methods in a way that they are obviously entry methods! See the Charm++ manual for more details.

4.1.4 Headers and File Organization

The Cello and Enzo layers of the codebase are both organized into subcomponents, but there are slightly different guidelines dictating the file organization.

Cello

The files are all organized in a flat directory structure. Every file is prefixed by the name of the component that it belongs to. Individual headers are not intended to be used. Instead, each component defines 2 standard header files.

1. *_component*`.hpp`: This is a private header that is used to aggregate the headers for all functions/classes defined as part of the component.
2. *component*`.hpp`: This is the public header. It contains includes the *_component*`.hpp`. It also contains all of the necessary include statements for the headers from other components that are necessary for the declarations/definitions present in the current headers (usually it includes the _-prefixed header).

Because the directory structure is flat all include-directives just specify filenames (there are no paths).

Enzo

Over the last several years, the Enzo layer has grown significantly (it is now comparable in size to the Cello layer), and it is likely to continue growing. Due to the large (and increasing) size of the Enzo layer, we take some steps to improve build-times. We are in the midst of rolling-out an updated policy.

Traditional Approach

Historically, the Enzo layer was structured just like one of the components in the Cello Layer in a flat structure. All individual header files were aggregated inside of the private `src/Enzo/_enzo.hpp` header and there was a public header called `src/Enzo/enzo.hpp`. The files were also originally organized in a flat directory structure, but that is no longer the case. Additionally, just about every source file started with:

```
#include "cello.hpp"
#include "enzo.hpp"
```

While this approach has been **very** successful and there's nothing wrong with it *per se*, it does trigger a full rebuild of the entire Enzo layer any time any header file changes.

New Approach

Under our new approach, the contents of the Enzo layer are now organized into subdirectories, which corresponds to a subcomponents. Each subcomponent has an aggregate header named after the component (e.g. the `Enzo/mesh` subcomponent should have an associated header called `Enzo/mesh/mesh.hpp`).

Note: Nested subdirectories are handled on a case-by-case basis. In some cases, each nested subdirectory is treated as a separate subcomponent. In other cases, nested subdirectories are only used for organization-purposes.

Unlike with the Cello layer, there is currently no distinction between a private and public header: the aggregate header **is** the public header. Each public header file should be self-contained. In other words, the header should compile on its own, without requiring it to be included alongside other header files (or requiring a particular order of include-directives).

To be self-contained, each public header needs to have the necessary include directives so that all of the other aggregated headers within the public header have access to the necessary symbols.

- Within the enzo-layer, all include-directives should specify paths to relative to the `src` directory. Thus the include directives may look like:

```
#include "Cello/cello.hpp"
#include "Cello/mesh.hpp"
#include "Enzo/enzo.hpp"
```

  This ensures that there is no ambiguity if there are similarly named subcomponents in the Enzo- and Cello-layers.

- A public header in a given subcomponent should generally only include the public headers from other subcomponents. Because including the public header from another component exposes symbols beyond what is strictly necessary, we recommend adding a comment next to the include directive that specifies the name of the symbol (e.g. a class, type, function)that is required from the header. By doing this, future developers can more easily remove unnecessary `#include` directives if a particular dependency is no longer necessary OR is moved
- It is important to be mindful that include-directives introduce transitive dependencies, and any time a header file changes, all `.cpp` files that depend on that header (whether directly or transitively) needs to be recompiled.
- Do your best to ensure that the public header files only include what is necessary and avoid unnecessary include statements (to keep compile-times shorter). With that said, it's better to explicitly write out include-directives to other headers, even if that header would transitively be included by some other unrelated header.

Note: In the long-term, it may make sense to make each individual header self-contained, which is the recommended strategy by the Google C++ Style Guidelines

Ideally, each header would include just the headers defining other symbols (classes/types/functions) that it needs. Under this approach, inclusion order would become less problematic and it would further speed up incremental compilation. Transitioning to this kind of approach all at once would be intractable since there are currently over 125 header files in the Enzo layer. However, this alternative approach is definitely worth exploring and could be implemented on a subcomponent-by-subcomponent basis in the future.

What is actually necssary to include in a header file?

It is useful to have a brief discussion about what is actually necessary to include in a header file.

As noted in the Mozilla Style-Guide, a full definition of a type is required for the type to be used

- as a Base class,
- as a member/local variable
- in a function declaration, where an instance of the type is passed as an argument, by-value, OR is returned from the function, by value.
- with `delete` or `new`
- as a template argument (in certain cases)
- if it refers to a non-scoped enum type (in all cases)
- when refering to the values of a scoped enum

The Mozilla Style-Guide also notes that a forward declarations of a type will suffice when the type is used

- for declaring a member/local variable that holds a reference or pointer to that type.

- in a function declaration that accepts reference or pointer to that type as an argument or returns a reference or pointer to that type.
- in the definition of a type-alias.

Note: In general, there is a difference of opinion about whether forward declarations should be used. For example, the Mozilla Style-Guide and the LLVM Style Guide generally encourage usage of forward declarations. In contrast, the Google Style Guide discourages this practice.

We don't currently take a strong position on this. In the codebase's current state, there are definitely cases where using forward declarations is **VERY** useful (it may be helpful to leave a comment explaining why its useful/necessary in a particular case).

If we do shift to making **all** header-files self-contained, it may make the codebase more readable if we avoided forward declarations in the future. But, we can cross that bridge, when we get to it.

Finally, it's worth noting that there is a large temptation to implement functions or a class's member-functions (aka methods) inside of the header where they are declared. Sometimes this is unavoidable - it's necessary when defining templates and sometimes it's necessary for performance-purposes (to facilitate inlining of a function called within a tight for-loop). However, in a lot of cases (especially when implementing a `virtual` method), this can and should be avoided. This practice has a tendency to introduce additional include-directives into a header file that could otherwise just be located within a source file.

For example, a handful of functions in Enzo-E make use of functionality defined in the `<algorithm>`, `<random>`, and `<sstream>` headers without needing to pass around types defined in these headers between functions. In these cases, by implementing such functions in `.cpp` files, we can directly include these headers in the `.cpp` source files and avoid including them in the header files (this can actually save a lot of time during compiling since standard library headers can be large).

As we finish transitioning the Enzo layer to using separate subcomponents, additional opportunities will arise for including headers in source files rather than inside of headers. For example, most times when you access instances of `EnzoPhysicsCosmology`, `EnzoPhysicsFluidProps`, or `GrackleChemistryData` in the method of a class, `MyClass`, the header defining `MyClass`, doesn't actually require knowledge about the definitions of these other classes. Often times, a method `MyClass` will simply use `!enzo::cosmology()`, `!enzo::fluid_props()`, or `!enzo::grackle_chemistry()` to retrieve an instance of one of these classes. Then the method will query a piece of information stored in the retrieved instance and it will never touch the instance again.

Questions and Answers about introducing new files to the Enzo Layer

How do I add a new file to the Enzo Layer?

Usually you will introduce a header (`.hpp`) and source (`.cpp`) file at the same time.

1. Find an appropriate subdirectory to put the new file in.
2. *Identify the relevant aggregate header file.* First, check for a header file that shares the name of the current subdirectory (with a `.hpp` suffix). If that doesn't exist, check for the aggregate header file in the parent directory. Repeat the process until you find subdirectory has an aggregate header file (you should only really need to go up 1 level).
3. Once you find the appropriate aggregate header file, add an include statement to your header file (if there aren't include-statements to other header files in the same subdirectory, you're probably in the wrong place).
4. *Identify the relevant* `CMakeLists.txt` *file for other files in the current subdirectory.* First, check for this file in the subdirectory where you have placed your new files. If it doesn't exist, check the parent directory. Repeat the process until you find it.

5. Add the paths to your new header and source file to the existing list of other files that are located in the same subdirectory as your new files. (If no such list exists, but you see something about `GLOB`, you may not need to do anything).

How do I delete a file from the Enzo Layer?

This is pretty self-explanatory. Just make sure to delete entries in the aggregate header file and (if applicable) the `CMakeLists.txt` file that specify the path(s) to your deleted file.

How do I move files between subdirectories in the Enzo Layer??

This is also pretty easy. Just make sure to delete the old paths from the aggregate header and the `CMakeLists.txt` files where they were originally listed. And, make sure to add the new paths to the appropriate aggregate header and the `CMakeLists.txt` files.

Enzo Header Guards

To avoid issues with a header being included multiple times (i.e. if it is a transitive dependency of another header), we make use of `#define` header guards in every header. In general, the header guard symbol looks something like `ENZO_<PATH>_<FILE>_HPP`, where `<PATH>` is replaced by the path to the file and `<FILE>` is the filename (excluding the `.hpp` suffix).

Note: A number of headers have guard symbols that are unchanged from before the Enzo subdirectory was reorganized. As an example, `EnzoMethodPpm` was previously defined in `src/Enzo/enzo_EnzoMethodPpm.hpp` and had a header guard symbol called `ENZO_ENZO_METHOD_PPM_HPP` (the underscore was used as a separator between Camel-Case separated words and in place of the period``.

4.1.5 General Coding Practices

We recommend some of the following coding practices:

- Prefer Composition over Inheritance
  - In C++, inheritance is the primary way to implement runtime polymorphism.[1] It is most useful to implement an interface as an abstract base class (e.g. `Method` or `Initial`). Then all derived classes can be used interchangably (after they've been constructed). This is a **great** uses of inheritance (that are used throughout Cello). The Google C++ Style Guide calls this "Interface Inheritance".
  - Inheritance can also be used in other cases as a mechansim for reusing code.
    * For example, one might want to make slight changes to the behavior of a base class by subclassing it.
    * Unfortunately, this practice easily/commonly produces code where the control flow is less explicit, which makes the code harder to reason about. If the base class has a chain of methods, where some can be selectively overwritten, there is a lot more to think about at any given time.
    * An alternative approach for code reuse in these circumstances involves composition. Essentially, you compose your objects out of self-contained (possibly reusable) components. These components may be stored as temporary variables in a function call or as attributes of a class. This typically produces much more explicit code. (Note: Interface inheritance may be used to make it possible to switch between different components).

[1] There are other approaches for acheiving runtime polymorphism in c++. The main alternative used in the codebase is the idea of a tagged-union (this is used inside of the `Parameters` class and the `ViewCollec` class template. It is also leveraged by the *EOS functionality*). This can provide certain speed advantages over inheritance, but the tradeoff is that you need to declare the closed set of all allowed types as part of the tagged union. In contrast, inheritance lets you freely introduce new types in other sections of the code.

    * A simple Google search of "composition over inheritance" will produce lots of additional discussion about this topic. The wikipedia article about this topic can be found here

- Where possible, use `const` to denote that a function doesn't mutate an argument or a member function doesn't mutate the members of a class. The concept of immutability helps make code much easier to think about (especially when you aren't the original author). The C++ core guidelines talk a little about why this can be useful.
- In the arguments of a function or the arguments of a member-function, prefer to pass C++ references instead of regular C pointers. There are a few exceptions to this rule: * You are passing in an array of data (as a pointer) * You explicitly want to allow the argument to be a `nullptr`.
- When declaring enumerations, prefer to use a scoped enum. The concept of a scoped enums was introduced in C++11. An example of a scoped enum from the codebase is the `ghost_choice` enumeration. The declaration of this type looks something like the following snippet:

```
enum class ghost_choice {exclude, include, permit};
```

  In the above snippet, if the `class` keyword is replaced by the `struct` keyword, the result is completely equivalent. If neither the `class` nor `struct` keyword were present, then `ghost_choice` would be an unscoped enum (or a C-style enum).

  There are three main differences to be aware of when using a scoped enum:

  1. The enumerators must be specified as `ghost_choice::exclude`, `ghost_choice::include`, and `ghost_choice::permit`. This is much more explicit (and consequently better) than the alternative. If `ghost_choice` were instead defined as an unscoped enum, `exclude`, `include`, and `permit` would be introduced as symbols in the global namespace.[2]
  2. Because `ghost_choice` is a scoped enum, `ghost_choice` is directly recognized as a type of a variable. If it were an unscoped enum, you must If it were an unscoped enum, you could instead declare a variable of type `enum ghost_choice`.
  3. Perhaps most importantly, scoped enums have better type safety. Specifically, integer values cannot be implicitly converted to an enum. Any conversions from an integer value requires an explicit cast. This generally leads to more explicit code. Furthermore, it lets the compiler identify cases where the order of an integer argument and a scoped enum argument are accidentally permuted.

- **ERROR REPORTING:** We don't use C++ exceptions in the codebase. In general, when an error arises, we generally abort the program with an informative error message. You can signal that an error occured by using the `ERROR` family of macros or you can use the `ASSERT` family of macros to have the program conditionally abort if some condition is not satisfied.
  - When you are implementing new functionality, you are encouraged to liberally use the `ERROR` and `ASSERT` families of macros to ensure that Enzo-E loudly fails and aborts when the functionality is used in unexpected ways. When running a really expensive simulation, a user should generally prefer that a simulation loudly fails. The extreme alternative case is for the simulation to run to completion while silently having problems, which likely invalidates the results (and these problems may not be detected until MUCH later). Furthermore, it's easy enough for a user to comment out an error message that they wish to ignore.
  - We briefly describe the arguments of `ERROR`, `ERROR1`, `ERROR2`, ..., `ERROR8`, `ASSERT`, `ASSERT1`, `ASSERT2`, ..., `ASSERT8` down below:
    * The first argument is always the name of the function where the macro is being invoked (to assist debugging in the future).
    * The second argument is a c-string that provides the error message. Printf formatting specifiers can be used within the error message, but the number of specifiers must match the integer at the end of the macro (e.g. `ERROR2` or `ASSERT2` expects 2 printf specifiers while `ERROR` or `ASSERT2` expects none).

[2] You can mimic the scoping behavior to some degree with an unscoped enum if you place the definition of the unscoped inside of a class or struct.

* The next arguments specifiy the variables used by the formatting specifier (if there are any).
* The `ASSERT` macro-family expects 1 last argument: the boolean condition dictating whether the program aborts.

4.1.6 Accessing Field data

There are 2 approaches for accessing Cello Field block arrays.

1. The traditional approach is to directly manipulate the raw pointers managed by Cello. These can be accessed with the `Field::values` method.
2. The preferred approach is to use the `CelloView` multidimensional array templates that wrap the raw pointers managed by Cello. Using `CelloViews` improve code clarity and safety at the cost of very marginal losses in performance (according to benchmarks). `Field::view` is used to help facillitate this approach. For more details about `CelloViews`, see *Using CelloView*

Below, we provide a table that summarizes the suggested names for Field-related variables used to access the array elements. We additionally provide brief descriptions of relevant `Field` methods. Sample code is also included for both approaches. In each case, the sample code initializes active zones (non-ghost zones) of the `"density"` field to be equal to `0.0`.

Summary of field attributes

Field-related variable	suggested names
Array dimensions	`mx,my,mz`
Active region size	`nx,ny,nz`
Ghost zone depth	`gx,gy,gz`
Loop variables	`ix,iy,iz`

Relevant `Field` methods

The traditional approach for accessing fields relies on:

```
char * Field::values (int id_field, int index_history=0) throw ();
```

This method returns the pointer to the data for the corresponding field (specified by `id_field`). If no field can be found, this returns a `nullptr`. The inclusion of ghost zones is determined by whether or not they've been allocated. While invoking this method as part of a `Method` object's implementation, it's fairly safe to assume that the returned array will indeed include the ghost zones.

The `index_history` argument is used to optionally specify the generation of field values that you want to load (higher values correspond to older generations of values). The default value, `0`, will give you the current generation of values (this is usually what you want).

The preferred approach for accessing fields relies upon:

```
template<class T>
CelloView<T,3> Field::view (int id_field,
                            ghost_choice choice = ghost_choice::include,
                            int index_history=0) throw ();
```

This returns a *CelloView* that acts as a view of the specified field. The meaning of the `id_field` and `index_history` arguments are unchanged from `Field::values`. Unlike `Field::values`, when an invalid `id_field` is specified, the program will abort with an error.

The template parameter `T` specifies the expected datatype of the field. If the field does not have the expected datatype, the program will abort with an explanatory error message. While implementing a `Method` object in the `Enzo` layer, this parameter is frequently `enzo_float`.

By default, the returned `CelloView` **always** include ghost zones. This behavior is controlled by the `choice` argument, which can be passed the following values:

- `ghost_choice::include`: the returned view always includes ghost zones (the program aborts with an error message if ghost zones aren't allocated). This is the default value.
- `ghost_choice::exclude`: the returned view always excludes ghost zones.
- `ghost_choice::permit`: the returned view includes ghost zones if they are allocated (replicating the behavior of `Field::values`).

Overloads are also provided for `Field::values` and `Field::view` that:

- provide read-only access to field arrays from a `const Field` instance
- let you replace the first argument with a string holding the field's name

Sample code for clearing active density zones

Traditional Approach

```
Field field = cello::field();
int id = field.field_id("density");
int mx, my, mz;
field.dimensions (id, &mx, &my, &mz);
int gx, gy, gz;
field.ghost_depth(id, &gx, &gy, &gz);

enzo_float * d = (enzo_float *) field.values(id);

for (int iz=gz; iz<mz-gz; iz++) {
  for (int iy=gy; iy<my-gy; iy++) {
    for (int ix=gx; ix<my-gx; ix++) {
      int i = ix + mx*(iy + my*iz);
      d[i] = 0.0;
    }
  }
}
```

Preferred Approach

A shorter version of the following snippet is also possible where we pass `ghost_choice::exclude` as the second argument to `field.view` to entirely exclude the ghost zone from the array.

```
Field field = cello::field();
int id = field.field_id("density");
int gx, gy, gz;
field.ghost_depth(id, &gx, &gy, &gz);

CelloView<enzo_float,3> d = field.view<enzo_float>(id);

// we can get the array shape directly from the array (a minor design quirk
// may make the arguments for the shape method seem a little unintuitive)
int mz = d.shape(0);
int my = d.shape(1);
int mx = d.shape(2);

for (int iz=gz; iz<mz-gz; iz++) {
  for (int iy=gy; iy<my-gy; iy++) {
    for (int ix=gx; ix<my-gx; ix++) {
      d(iz, iy, ix) = 0.0;
    }
  }
}
```

More code examples that use this functionallity can be found in the implementation of the `EnzoInitialCloud` and `EnzoInitialBCenter` classes

4.1.7 Accessing Particles

4.1.8 Preprocessor Macros

In case you add a new preprocessor macro or definition that is being used in a `.ci` file, you have to add it to the `CHARM_PREPROC_DEFS` list in the main `CMakeLists.txt` so that it is being used/forwarded when processing the `.ci` files.

For example, for Grackle we set `set(CHARM_PREPROC_DEFS ${CHARM_PREPROC_DEFS} "-DCONFIG_USE_GRACKLE ")`, which appends to the existing list of defintions alredy stored in the `CHARM_PREPROC_DEFS` variable.

4.2 Writing initial conditions

[This page is under development]

4.3 Writing refinement criteria

[This page is under development]

4.4 Writing Methods

[This page is under development]

All method classes are subclasses of the `Method` class. The virtual methods that concrete classes override are shown below:

virtual void `Method::compute`(Block *block) = 0

Apply the method to advance a block one timestep

The `Block::compute_done()` method MUST be invoked on all `blocks` passed to this member function by the end of the control flow that this function launches.

- In simple cases, that should be done just before this function returns.
- The placement will varies in more complex cases. For example, if this function invokes a single reduction, the call to `Block::compute_done()` should be performed after completing the reduction (e.g. in the `compute_resume` member function)

virtual std::string `Method::name`() = 0

Return the name of this Method.

inline virtual double `Method::timestep`(Block *block)

Compute maximum timestep for this method

The default implementation returns the maximum finite value of `double`

inline virtual void `Method::compute_resume`(Block *block, CkReductionMsg *msg)

Resume computation after a reduction

This member function only typically needs to be implemented by Method classes that employ reductions.

Note: This page is very incomplete. Among other things, we have not discussed `pup` routines.

4.4.1 Overview

When writing a `Method` class, it's useful to understand how it is used by `Cello/Enzo-E`.

Recall that when you launch `Enzo-E`, you specify how many PEs (processing elements) should be used on the command line. During startup, a list of `Method` objects are constructed on each PE, based on the parameter file (this list is managed by the PE's `Problem` instance). It's also useful to remember simulation data (e.g. fields and particles) are associated with `Block` objects. Throughout the simulation each PE is responsible for evolving a local set of 1 or more `Block` objects (note that load-balancing can theoretically migrate `Block` objects).

Before each compute cycle, `Cello/Enzo-E` determines the current timestep. For each local `Block`, a PE invokes the `timestep` for each of its `Method` instances to determine constraints on the next timestep. Some other considerations (e.g. user-specified scheduling of operations/stopping at certain simulation times) may alter the duration of the timestep. Finally, a reduction is performed to pool together the constraints from all blocks.

During the compute cycle, each PE executes a control flow similar to the following code snippet. Be mindful, that the following snippet doesn't actually exist anywhere in the codebase.

```
void call_compute_on_all_methods(std::vector<Method*> &method_l,
                                 std::vector<Block*> &local_block_l){
  std::size_t num_methods = method_l.size();
  std::size_t num_local_blocks = local_block_l.size();

  for (std::size_t method_ind = 0; method_ind < num_methods; num_methods++){
    Method* cur_method = method_l[method_ind];

    for (std::size_t j = 0; j < num_local_blocks; j++){
      /* do some fancy stuff related to refreshing fields with data from
         neighboring blocks */

      // call compute on the method
      cur_method->compute(local_block_l[j])
    }

    /* apply a synchronization barrier to make sure that all blocks across
     * all processes have finished completing the current Method.
     *
     * (essentially, wait for block->compute_done() to be called on every
     * block...)
     */
  }

}
```

Pitfalls

The previous section should have made it clear that a given `Method` instance generally has its `timestep` and `compute` method invoked on one or more `Block` per cycle. Consequently, problems can arise if you mutate the attributes of a `Method` instance based on data from a given `Block` instance.

A good rule-of-thumb for new developers is that you should generally avoid mutating attributes `Method` object outside of the constructor. If you need to associate data with a given `Block`, you should consider using one of the specialized data interfaces that exist for:

- field data (managed by `Field`)
- particle data (managed by `Particle`)
- scalar data (managed by `Scalar`)

An advantage of using these interfaces is that the associated data will be appropriately migrated if a `Block` migrates between PEs. In certain cases one might alternatively add an attribute to `EnzoBlock`, but that's generally discouraged if it can be avoided (the `Scalar` interface is usually a better choice).

As an aside, there may be times where it makes sense to violate this guideline (e.g. to facillitate optimizations).

Standard properties tracked in base class

All `Method` classes provide some standardized properties that are managed through the base class.

In some of the following cases, we will talk about how the parameter gets specified for a hypothetical method called `"my_method"` (in this hypothetical scenario, subclass's implementation of *`name()`* would return `"my_method"`).

Courant Number

All `Method` classes have an associated courant condition. For a method named `"my_method"`, the courant value is specified via the parameter called `Method:my_method:courant`.

This parameter is automatically parsed by machinery in the `Cello` layer and the machinery will update the `Method` objects with this value right after the constructor is called. The value of this parameter can be accessed in a `Method` subclass with the following function:

inline double `Method::courant()` const

Query the associated courant factor.

This parameter is usually accessed in the subclass's implementation of *`timestep()`*.

At this time, developers should avoid parsing and tracking the courant value separately within the subclass.

Note: This should not be confused with the `Method:courant` parameter. This parameter specifies a global courant factor that should never be touched by a `Method` subclass. Instead, this parameter is entirely handled by the rest of the `Cello` infrastructure.

Scheduling

All `Method` classes support the ability to be scheduled. For a method named `"my_method"`, the schedule is specified via a subgroup called `schedule`. The rules for specifying a schedule are fairly standard and are described elsewhere in the documentation.

The initialization and usage of an associated schedule are all handled by external `Cello` machinery. A `Method` subclass should never need to interact with it (in fact, interacting with it improperly could cause problems).

Refresh Machinery

`Cello` provides some standardized machinery for specifying requirements related to the fields and particles that need to be refreshed. The refresh operations are automatically handled by the Cello machinery prior to calls to the `method` class.

Configuration of this machinery is typically handled in the constructor of a `Method` subclass.

[This section is incomplete]

4.5 Parameter Parsing and Configuration

[This page is under development]

4.5.1 Overview

Most of the parsing work is handled by the `Parameters` class from the Cello layer and stored internally. All parameters are parsed after startup on a single process and then the parsed information is communicated between processes.

We are currently in the process of migrating between approaches for accessing the parsed parameters in order to use them to configure Enzo-E/Cello classes. Given the large scope of this transition, we decided to gradually migrate between the approaches, rather than try to do it one shot.

- At the time of writing, we have migrated a large number of `Method` classes.
- Our intention is to eventually migrate as much as possible to this new approach (e.g. all `Method` classes, all `Physics` classes, all `Initial` classes, etc.). Ultimately we would like to remove the `Config` and `EnzoConfig` classes.

First, we briefly review properties of Enzo-E/Cello parameters and talk about the idea of a "parameter-path" Next, we describe how to actually access a parameter value (using the new approach). Then, we talk through an example of how you might add a new parameter (using the new approach). Afterwards, we provide some more details about the design of the new-approach. Finally, we briefly discuss the older approach (and some of its shortcomings).

4.5.2 Parameter Files and Parameter-Paths

Todo: Consider merging the content of this section into *Parameter files*.

As explained in *Parameter files*, Enzo-E/Cello makes use of a hierarchical parameter file (this documentation assumes you are familiar with the basics from that section).

Essentially, parameters are organized within groups (and possibly subgroups). In other words, the parameters can be thought of as leaf nodes in a tree-hierarchy of "groups". The following snippet illustrates the organization of parameters from a hypothetical parameter file:

```
Method {
   list = [ "mhd_vlct", "grackle"];

   mhd_vlct {
      # other assorted parameters...
      courant = 0.3;
   }

   grackle {
      # other assorted parameters...
      courant = 0.4;
   }
}
```

The organization of parameters in a group hierarchy is analogous to the organization of files in a directory hierarchy. Continuing this analogy, we have devised a shorthand for naming parameters in the documentation (and throughout the codebase) that is similar to a file path. One can think of these names as a "parameter-path".

- The parameter-paths associated with the above snippet (that you would see in the documentation or as strings in the codebase) would include Method:list, Method:mhd_vlct:courant, or Method:grackle:courant. Please note: precise parameter names are subject to change over time.
- In general, a parameter-path for a given parameter lists the names of ancestor "groups", separated by colons, and lists the name of the parameter at the end (i.e. the string that directly precedes an assignment).

4.5.3 How To Access Parsed Parameter Values

As mentioned above, the nitty-gritty details of parsing are handled by Enzo-E automatically and the results are stored within an instance of the `Parameters` class.

Values associated with parameters can be queried by invoking methods directly provided by the `Parameters` class or `ParametersGroup` class.

- a `Parameters` instance provides access to **all** parameters
- a `ParametersGroup` instance is a light-weight object that provides access to parameters within a particular group

Basic API for Accessing Parameters

Both classes define a common set of methods for querying the values associated with the parameters. We'll now describe some of the most commonly used methods. Consider a reference to a `Parameters` instance or a `ParametersGroup` instance called `p`. To access the value associated with a parameter `s` one might invoke one of the following methods (based on the expected type of `s`):

- `p.value_logical(s, false)` if the parameter is expected to specify a boolean value and if it defaults to a value of `false` when the parameter is not specified.
- `p.value_integer(s, 7)` if the parameter is expected to be an integer and if it defaults to a value of `7` when the parameter is not specified.
- `p.value_float(s, 2.0)` if the parameter is expected to be a floating-point value and if it defaults to a value of `2.0` when the parameter is not specified.
- `p.value_string(s, "N/A")` if the parameter is expected to be a string and if it defaults to a value of `"N/A"` when the parameter is not specified.

If the user specified the desired parameter, but with an unexpected type, the program will abort with an error message (another function is provided to query the parameter type)

In each of these snippets, `s` is **always** a parameter path, but the precise interpretation depends on how `p` is defined. When `p` references a `Parameters` instance, `s` must specify an absolute parameter-path. In contrast, when `p` references a `ParameterGroup` instance, `s` must specify the path relative to the group associated with the `ParameterGroup`.

For completeness, consider the following parameter-file snippet:

```
Physics {
  fluid_props {
    eos    {  gamma = 1.6666666666666667; }
    floors {  density  = 1.0e-10;         }
  }
}
```

The table below clarifies the value of `s` that should be used to specify Physics:fluid_props:eos:gamma for different choices of `p`.

if p is a refers to a	then s is
`Parameter` instance	`"Physics:fluid_props:eos:gamma"`
`ParameterGroup` instance associated with Physics:fluid_props	`"eos:gamma"`
`ParameterGroup` instance instance associated with Physics:fluid_props:eos	`"gamma"`
`ParameterGroup` instance associated with Physics:fluid_props:floors	unable to specify the desired parameter

Common Patterns in the Codebase

Enzo-E and Cello define many classes descended from the Cello-class-hierarchy (e.g. `Method` subclasses) that are directly initialized from the parameters in a single parameter-group. These classes are commonly initialized with a constructor that accepts a `ParametersGroup` instance (associated with the appropriate parameter-group) as an argument.

For the sake of example, let's consider the `EnzoMethodHeat` class. This class is configured by parameters like the ones in the Method:heat group from the following parameter-file snippet:

```
Method {
   list = [ "heat"];

   heat {
      alpha = 0.6;
      courant = 0.3;
   }
}
```

Here's we present (an edited) example of what the class's constructor might look like:

```
EnzoMethodHeat::EnzoMethodHeat (ParameterGroup p)
 : Method(),
   alpha_(p.value_float("alpha",0.7)) // access alpha param & use it to initialize
                                      // this->alpha_ (for the sake of example,
                                      // it defaults to 0.7 if not specified)
{
  // parse the courant value
  double parsed_courant_val = p.value_float("courant", 1.0);
  this->set_courant(parsed_courant_val); // <- this a convenience method provided by
                                         //    the Method base class

  // for debugging purposes or for printing out informative error messages:
  //   - you can use p.get_group_path() to get the current the std::string
  //     specifying the parameter-group's path that p is associated with.
  //   - you can use the p.full_name(s) to get the absolute path for a parameter
  //     (s is the parameter-path relative to the current group)
  //   - when initializing the class from the above parameter-file snippet
  //       - p.get_group_path()   would return std::string("Method:heat")
  //       - p.full_name("alpha") would return std::string("Method:heat:alpha")

  // we have omitted a bunch of other code that is required for initialing a Method
  // class...
}
```

PLEASE NOTE: that adding a new parameter to Cello/Enzo-E involves a few additional steps beyond just modifying the constructor. These steps are described in the next section.

4.5.4 How to add a new parameter

Let's walk through an example where we want to introduce a new parameter to `EnzoMethodHeat`. Suppose we want to add a new parameter called my_param. The full name of this parameter would be Method:heat:my_param.

The steps are as follows:

1. Introduce a new member-variable (aka an attribute) to `EnzoMethodHeat` (in the `EnzoMethodHeat.hpp` file). For the sake of example, let's imagine that we want to directly store the value specified in the parameter-file in a member-variable (a.k.a. an attribute) named `my_param_`.
   - The convention is to declare all member-variables as `private` or `protected` (if the value of that attribute is needed outside of the class, you should define a `public` accessor-function).
   - Relatedly, the names of all `private` & `protected` member-variables or member-functions should generally be suffixed with an underscore. An underscore should **NEVER** be the first character in the name of a member-variable or member-function.
   - **NOTE:** the value of the parameter doesn't necessarily need to initialize a variable with a matching name (or type), this is just a convenience in this example (although, it does make the code a little easier to follow)
2. Modify the pup routine of `EnzoMethodHeat` and the `PUP::able` migration constructor to properly handle the newly added member-variable
3. Modify the main constructor of `EnzoMethodHeat` to initialize `my_param_` based on the value parsed from the parameter file.
   - The constructor of `EnzoMethodHeat` is passed a copy of an instance of `ParameterGroup`, in an argument `p`.
   - In the simplest case, you might use one of the methods described *here* to access the value specified in the parameter-file, and store the result in `my_param_`.
   - Alternative methods of `p` or more advanced logic (than a simple assignment) may be needed in slightly more sophisticated cases (for example if the parameter expects a list of values or if you want to abort the program if the parameter can't be found).

4.5.5 Design Overview (new approach)

Our new approach revolves around the usage of the `ParameterGroup` class for accessing/querying parameters stored in an instance of the `Parameters` class. Instances of the `ParameterGroup` class are light-weight and are expected to have a short-lifetime (akin to `Field` or `Particle`).

As illustrated above, instances of the `ParameterGroup` class are expected to be passed to the constructor of classes that inherit from Cello class-hierarchy.

The main feature of the `ParameterGroup` class is that it provides methods for querying/accessing parameters with parameter-paths that share a common root.

- The root parameter-path is specified during the construction of a `ParameterGroup` instance and cannot be changed over the lifetime of the instance.
  - The immutable nature of the root parameter-path is a feature: whenever a `ParameterGroup` instance is passed to a function, you ALWAYS know that the root parameter-path is unchanged (without needing to check the helper function's implementation).

  - If a developer is ever tempted to mutate the root-path, they should just initialize a new `ParameterGroup` (since the instances are lightweight)
  - The root path can be queried with `ParameterGroup::get_group_path()`

- When a string is passed to one of the accessor methods, that string is internally appended to the root pararameter-path and the result represents the full name of the queried parameter. (You can think of this string as specifying the relative path to the parameter). You can use `ParameterGroup::full_name(s)` to see the full parameter name that a string, `s`, maps to.

Why do we even need `ParameterGroup`?

To motivate the existence of the `ParameterGroup` class, it's useful to consider alternative approaches. The most obvious option is to simply pass instances of the `Parameters` class to constructors (insteading of passing a `ParameterGroup` instance).

To flesh out this alternative case more, let's consider the following snippet of a hypothetical parameter file.

```
Method {
  list = [
     "output", # ...
  ];

  output {
    file_name = # ...
    all_fields = true;
    all_particles = true;
    # other parameters ...

  }
}
```

This particular snippet can easily be parsed if we pass a reference to the `Parameters` object to `MethodOutput`'s constructor. An example code block is included here, to show (roughly) what that constructor might look like:

```
// NOTE:
// - MethodOutput is a special case. Historically, it has needed to accept
//   an argument other than just the parameters
// - a delegating constructor is only used as a matter of convenience
// - We have made a number of simplifications here compared to what the
//   source code actually looks like...

MethodOutput::MethodOutput(/* ... */, Parameters &p)
  : MethodOutput(/* ... */,
                 p.value_string("Method:output:file_name", ""),
                 p.value_logical("Method:output:all_fields", false),
                 p.value_logical("Method:output:all_particles", false),
                 /* ... */)
{ }
```

There is nothing wrong with the above snippet, and it will work in a lot of cases. However, we will encounter issues when we want to set up a simulation that makes use of multiple `MethodOutput` instances. To illustrate how this is done in Enzo-E, see the following snippet from a hypothetical parameter file:

```
Method {
  list = [
     "output_field", "output_particle", # ...
  ];

  output_field {
    file_name = # ...
    all_fields = true;
    all_particles = false;
    # other parameters ...

    type = "output";
  }

  output_particle {
    file_name = # ...
    all_fields = false;
    all_particles = true;
    # other parameters ...

    type = "output";
  }

}
```

As you can see from the above snippet, a parameter-subgroup carrying the configuration of the `MethodOutput` instance is no longer called `"output"`.

- now 2 `MethodOutput` instances should be initialized, using the configuration from the parameter-subgroups called `"output_field"` and `"output_particle"`.
- Cello/Enzo-E determines the `Method` subclass that a given parameter-subgroup, Method:<subgroup>, is meant to describe based on the value of Method:<subgroup>:type. In both above subgroups, we have specified the type as `"output"`. (In the common case where Method:<subgroup>:type is omitted, the type parameter defaults to the string-value of <subgroup>)

Importantly, the absolute paths of the parameters that are used to initialize the `MethodOutput` instances are different in the second parameter file compared to the first. The main difference is in the the root-path to the subgroup.

To gracefully handle both scenarios, we now make use of the of the `ParameterGroup` class. A code snippet using our new approach is shown below:

```
// NOTE:
// - MethodOutput is a special case. Historically, it has needed to accept
//   an argument other than just the parameters
// - a delegating constructor is only used as a matter of convenience
// - We have made a number of simplifications here compared to what the
//   source code actually looks like...

MethodOutput::MethodOutput(/* ... */, ParameterGroup p)
  : MethodOutput(/* ... */,
                 p.value_string("file_name", ""),
                 p.value_logical("all_fields", false),
                 p.value_logical("all_particles", false),
                 /* ... */)
```

(continues on next page)

(continued from previous page)

```
{ }
```

Note: Historically, the `Parameters` class has also had the capability to track a common root-path. However, the code was not very explicit about whether that capability was being used or not (although, most of the time you could safely assume that the feature wasn't being)

It's our intention to eventually remove this capability from the `Parameters` class, since the `ParameterGroup` class can be used for the same purpose (and it's more explict)

Note: The main disadvantage of this approach is that we no longer specify the full, absolute parameter names, when accessing the values. However, this is mostly unavoidable if we want to gracefully accomodate initialization of multiple instances of the same `Method` subclass. Hopefully, this page of documentation will help to offset this disadvantage.

The *only* other alternative is have `ParameterGroup` instances "auto-magically" redirect absolute parameter-paths, but I think that will generally be more confusing.

Hypothetical Question: How do I used `ParameterGroup` to query the parameter specified to configure some other `Method` subclass?

The short answer is "you don't". The `ParameterGroup` class is designed to restrict access to parameters within the associated parameter-group/root-path. This is a **feature** that discourages the design of classes that are configured by parameters scattered throughout the parameter file.

Let's be more concrete: let's imagine that while configuring an instance of a class called `MethodX`, and we want to access a special parameter value stored outside of `MethodX`'s associated parameter-group. That special parameter might instead be part of a parameter-group associated with a different `Method` subclass, a `Compute` subclass, an `Initial` subclass, etc.

Experience tells us it is usually an anti-pattern to directly access that parameter value (via `ParameterGroup` or `Parameter` instance). This problems with this kind of code include:

1. It makes refactoring of that parameter much more difficult.
2. It can lead to cases where you are trying to access parameter-values for `Method` subclasses regardless of whether the subclass is even being used in the simulation.

Preferred alternatives to include:

1. Introducing an accessor method to access the special parameter-value from the `Method` subclass (or `Compute` subclass or `Initial` subclass or etc.) that the parameter is associated with.
2. Altering the way in which the parameter is specified and store it within a `Physics` class.

The tradeoffs of these approaches are discussed in greater detail *here*.

In rare cases (e.g. during refactoring when we convert a previously Method-specific parameter to a Physics parameter and want to retain backwards compatability), exceptions to this philosophy need to be made. Thus, an "escape-hatch" is provided to directly access the global `Parameter` object: call the `cello::parameters()`. Please, avoid using this "escape-hatch" unless it's truly necessary.

Todo: We could consider extending the analogy between a parameter-path and a file path. For example, one could imagine interpreting a path that begins with a `:` as an absolute parameter-path and all other strings as relative parameter-paths.

This would probably streamline the documentation to some degree.

If we were to do that, we would need to modify the code to recognize this convention. We would probably also want to modify the various parameter-accessor methods of the `ParameterGroup` to continue to restrict access to parameters within the common root-path that a `ParameterGroup` is configured with.

4.5.6 Historical Approach

Historically, all parameters were parsed shortly after startup and then the results were stored as variables in the `Config` and `EnzoConfig` classes. However, this approach had a number of warts:

- Adding a new parameter "properly" was laborious. Let's imagine that we want to add a parameter, `<param>`, to class `<Klass>`. This class might be a subclass of `Method`, `Initial`, `Physics`, etc. To add this new parameter, we need to
  - (i) define a new member-variable (aka attribute) to `EnzoConfig` to hold the value of the parameter
  - (ii) ensure that the new member-variable of `EnzoConfig` is properly serialized
  - (iii) add the logic to retrieve the value associated with the parameter from the `Parameters` object and store that value in the newly defined member variable of `EnzoConfig`
  - (iv) modify the line of code where `EnzoProblem` calls the constructor of `<Klass>`, in order to pass the parameter-value stored in the newly-defined member-variable of `EnzoConfig`.
  - (v) add a newly-defined member variable on `<Klass>` in order to store the value of the parsed parameter
  - (vi) ensure that the new member-variable of `EnzoConfig` is properly serialized
  - (vii) modify the primary constructor of `<Klass>` to actually initialize the new member-variable
- Because of how laborious this is, developers have a tendency to just skip the last for steps and access the attributes of the global `EnzoConfig` instance. This has all the short-comings of global variables (it makes things hard to refactor)
- If you want to do error-checking of the parameter-values, it's not always clear where to do that (within `EnzoConfig` vs within the constructor of the class that uses the parameter)
- complications arise if multiple instances of a class can be initialized with different configurations.

Our new practice takes inspiration from Athena++. Essentially, the new approach's intention is to have every `Method/Initial/Physics` class just directly access the needed values from the parameter file. We skip the whole step of storing the parsed values in an `Config` or `EnzoConfig` instance and then forwarding those values. We essentially "cut out the middleman".

4.6 Writing Physics Classes

[This page is under development]

This page tries to provide an overview for how to write `Physics` classes and documents some quirks about existing classes.

All physics classes are subclasses of the `Physics` class. Unlike other classes in the Enzo-E layer that descend from the Cello class-hierarchy (`Method`, `Initial`, `Prolong`, `Refine`, etc.), the Cello-layer doesn't really interact much with instances of the `Physics` classes - beyond storing them.

In practice, `Physics` classes are commonly used to store problem-specific configuration information that needs to be accessed by multiple different `Method` classes and/or initializers. Some functions that make use of this information are also sometimes introduced to these classes.

4.6.1 When to write a `Physics` class

Other ways to store data

To understand when it may be useful to write a `Physics` class, it's first useful to discuss some of the ways one could access/store cross-cutting configuration data:

1. store this information in a global variable (DON'T DO THIS)
   - From a coding-style perspective, this is almost always the **wrong** way to store such information.
   - Moreover, Charm++ does not support global variables unless they are properly declared in the .CI (note: the code will still compile, but it should be avoided all the same).
2. access information stored in `EnzoConfig`, which is accessible through `enzo::config()` (TRY NOT TO DO THIS)
   - this class already stores a lot of values parsed from parameter files.
   - while this approach can be convenient in simple cases, it generally leads to brittle code that is hard to refactor (challenges can come up if you want to alter the way that different options are stored in the parameter file).
   - This approach has been used a fair amount in the past, but we are actively moving away from it (and discourage this approach).
3. Cross-cutting configuration information is commonly only relevant when you are using a particular `Method` class. In this case, you can store the configuration information within that `Method` class and define public instance-methods on that particular class that accesses the information.
   - This approach is strongly prefered over the preceding approaches 1 and 2. Refactoring is generally easier in this approach. Moreover, this can be easier than defining a `Physics` class.
   - For this approach, it's important to understand how to access an instance of a particular kind of `Method` at an arbitrary point in the code (after all `Method` classes have been constructed). One can use `enzo::problem()->method("<name>")` to return a pointer to the instance of the `Method` class for which *`Method::name()`* returns `"<name>"`. If no such instance can be found, the expression returns a `nullptr`. You then need to cast that pointer to the appropriate `Method` subclass before you access the information.
   - At the time of writing, this approach is commonly used to store information encoded within the `EnzoMethodGrackle` class. To access such information, one could write

     ```
     const EnzoMethodGrackle *ptr = static_cast<const EnzoMethodGrackle*>
       (enzo::problem()->method("grackle"));
     if (ptr != nullptr) {
       // maybe do stuff with ptr->try_get_chemistry() ...
     }
     ```

     In practice, some convenience functions have been written to help with these sorts of operations like `enzo::grackle_method()` or `enzo::grackle_chemistry()`
   - In principle, one could do something analogous involving subclasses of `Initial`, but that could potentially introduce problems during a simulation restart.

Storing information in a `Physics` class

It's often most useful to encode configuration-information within a `Physics` class when there isn't an obvious single `Method` class where it should be stored.

A particular scenario where this is relevant is when separate (somewhat-interchangable) `Method` classes implement different algorithms to model the same set of physics. For example, consider the storage/access of the dual-energy formalism configuration. Since this is mostly relevant in the context of a hydro-solver it may make sense to store this information in the `Method` class that encapsulates a hydro-solver. However, because Enzo-E has `Method` classes that implement different hydro-solvers (that use the dual-energy formalism), we instead encode this information in a `Physics` class.

There are also scenarios where some configuration information isn't really associated with any singular `Method`. For example the Equation-Of-State is important to a number of different methods. Another example includes the Gravitational Constant - this is important in self-gravity solvers and external-potential solvers, which are implemented in different `Method` classes.

4.6.2 General Tips

The general advice is to implement a `Physics` class so that it is immutable (after construction the instance's state doesn't change).

This makes the behavior of `Physics` classes much easier to reason about because a single PE (processing element)

- only has one instance of a given `Physics` class
- AND is responsible for evolving one or more instances of `EnzoBlock`.

4.6.3 Quirky Implementations

`EnzoPhysicsCosmology` currently tracks some mutable state (e.g. the current scale-factor, the current rate of expansions, current redshift). This is just something to be mindful of.

It's worth noting that the initialization of `EnzoPhysicsFluidProps` and `EnzoPhysicsGravity` are a little quirky. These objects are ALWAYS initialized, regardless of whether a user specifies the names of these objects in the *Physics:list* configuration-file parameter. This choice was made for the sake of maintaining backwards compatability with older versions of parameter-files that were created before these classes were invented (since they encode some information that was previously stored elsewhere).

Note: This means that all simulations have an instance of `EnzoPhysicsGravity` (regardless of whether or not gravity is actually in use).

4.7 Using CelloView

4.7.1 Background

Historically, to access elements at a given location, `(ix, iy, iz)` of an array in Enzo, the developer would need to explicitly calculate the index of the pointer using knowledge of the underlying shape of the array represented by the pointer.

To simplify and enhance the readability of code in Enzo-E, we have implemented `CelloView`, which encapsulates the operations associated with Multi-dimensional data. This class template draws loose inspiration from Athena++'s

`AthenaArray` and numpy's `ndarray`. This class was initially called `CelloArray`, but the name was changed for reasons described in *Why is it named CelloView*.

See the first two cases listed in *Examples* for comparisons of snipets written using `CelloView` and traditional pointer operations. These examples reflect operations performed in Enzo-E.

Throughout the Enzo portion of the codebase, we extensively use the type `EFlt3DArray` which acts is an alias for `CelloView<enzo_float,3>`.

4.7.2 Design Goals

The design of the class was primarily driven by the following specifications:

- Emphasize fast access to array elements by passing the index along each dimension to the `operator()` method.
  - This method can be inlined within for-loops and for 3D arrays it has the same complexity as that of `AthenaArray`.
  - Simple benchmarks show that the current implementation achieves performance comparable to c-style array access
- The `CelloView` needs to be able to allocate and manage its own memory AND wrap existing pointers (namely the pointers allocated by the Cello's Field framework)
  - This allows code using the `CelloView` to coexist alongside code which use pointers in a more conventional way.
- `CelloView` needs be able to represent a view of a (mostly contiguous) subarray of a pre-existing instance of `CelloView`. This facilitates the encapsulation of a directional mesh operation in a single generalized function (e.g. writing a single flux function for all directions rather than separate functions to compute flux along the x, y, and z directions).

4.7.3 User Interface

The class template is formally defined as `CelloView<T,D>` where `T` is the element type (frequently `enzo_float`) and `D` is the number of dimensionsions of the array.

At a high-level, this class template has semantics like a pointer or `std::shared_ptr` (there are also similarities to numpy's `ndarray`). These objects serve as a smart-pointer with methods for treating the data as a specialized array. These semantics explicitly differ from the C++ standard library containers (like `std::vector`).

In other words, `CelloView<T,D>` acts as an address for the underlying data. The copy constructor and copy assignment operation effectively make shallow copies and deepcopies are made by explicitly invoking special methods. A consequnce of this is that any modifications made to the elements of an array within a function, where the array had been passed by value, will affect the value of the array outside of the function. We will return to this topic below in *Pointer Semantics*

To provide a more detailed description of `CelloView<T,D>`'s user interface it is most straightforward to describe the different operations with examples (rather than providing a detailed API).

Array Creation

Simplest initialization:

- Use the constructor `CelloView(Args... args)` to construct an array of 0s of shape `(arg0, arg1, ... arg{D-1})`. The resulting array owns the underlying memory and deallocation is entirely taken care of
- Examples:
  - Construct an array of shape `(2,3,4)` that holds doubles:

```
CelloView<double,3> arr(2,3,4);
```

  - Construct an array of shape `(5,)` that holds ints:

```
CelloView<int,1> arr(5);
CelloView<int,1> arr2 = CelloView<int,1>(5); // yields same result
```

Wrap a pre-existing pointer:

- Use the constructor `CelloView(T* array, Args... args);` to wrap the pointer `array` which represents an array with shape `(arg0, arg1, ... arg{D-1})`.
- Example: Construct an array representing `[[0,1,2],[3,4,5]]`:

```
int data[] = {0,1,2,3,4,5};
CelloView<int,2> arr(data,2,3);
```

We can also forward declare an array and assign values to it later.

```
int data[] = {0,1,2,3,4,5};
CelloView<int,2> arr;
arr = CelloView<int,2>(data,2,3);
```

Dimension Size

To get the length along a dimension (or axis), call `arr.shape(unsigned int dim)`, where `dim` is the number of the dimension. Dimensions numbers start at `0` and are ordered with increasing indexing speed (`dim=D-1` is the dimension with fastest indexing).

Element Access

To access an element pass indices to the `operator()(Args... args)` method. As many indices should be specified as there are dimensions in the array (the number of args **must** match the number of dimensions.

The `operator()(Args... args)` method returns a reference or copy (depending on the circumstance) of the element.

Example: print element `(0,2)` of the array `[[0,1,2],[3,4,5]]`:

```
int data[] = {0,1,2,3,4,5};
CelloView<int,2> arr(data,2,3);
printf("%d\n", arr(0,2)); // prints "2"
// printf("%d\n", arr(2));       This would fail to compile
// printf("%d\n", arr(0,0,2));   This would fail to compile
```

Simple Assignment - Shallow/Deep Copies

Shallow copies are produced via ordinary assignment.

```
int data[] = {0,1,2,3,4,5};
CelloView<int,2> a(data,2,3);
CelloView<int,2> b = a; // b is now a shallow copy of a
CelloView<int,2> c(2,2); // c represents [[0,0],[0,0]]
CelloView<int,2> d = c; // d is now a shallow copy of c
c = a; // c is now a shallow copy of a
```

When `c` is assinged the contents of `a`, `c` becomes a shallow copy of `a`. However the contents of `d` are unaffected. It still represents the array `[[0,0],[0,0]]`.

To perform a deepcopy, assign the the results of the `deepcopy` method.

```
int data[] = {0,1,2,3,4,5};
CelloView<int,2> a(data,2,3);
CelloView<int,2> e = a.deepcopy(); // e is now a deep copy of a
```

Modifications to the contents of `e` will not be reflected in `a` or `data` (and vice-versa)

Creating Subarrays

Calling `arr.subarray(Args... args)` returns a (mostly contiguous) view of a subarray specified by `args`, where `args` represent the slices along each dimension. Each `arg` should be an instance of `CSlice` and the number of `args` **must** match the number of dimensions of the array. `CSlice` is a class that represents the start and stop points along a given dimension. The standard constructor is simply: `CSlice(int start, int stop)`.

As an aside, when `arr` has 2 or more dimensions, `arr.subarray` has an overload that accepts a single integer argument `i`. The returned subarray is roughly equivalent to the view returned by `arr.subarray(CSlice(i,i+1), ...)` where the omitted arguments are slices that include all of the elements along the corresponding dimensions. The *only* difference is that the resulting array has 1 fewer dimensions than `arr`.

Subarray Examples

We present an extended example below. We start by defining a subarray, `sub` of an array `arr` (which wraps an existing pointer of data and represents the array `[[0,1,2],[3,4,5]]`).

```
int data[] = {0,1,2,3,4,5};
CelloView<int,2> arr(data,2,3);
CelloView<int,2> sub = arr.subarray(CSlice(0,2),CSlice(1,3));
printf("%d\n", sub(1,0)) // prints "4";
```

At this point `sub` represents the subarray `[[1,2],[4,5]]` of the full array held by `arr`. `sub` is truly a "view" of `arr`. Modifications to the elements of `sub` and modifications to elements in `arr` (if it lies in the subarray), are reflected in both locations.

```
arr(1,3) *= -3;
sub(0,0) = -100;
```

After executing the above block of code, `arr` now represents `[[0,-100,2],[3,4,-15]]` and `sub` represents the subarray `[[-100,2],[4,-15]]`.

`CelloView` also provides support for taking subarrays of subarrays (or taking subarrays of shallow copies). If we define a subarray of `sub` the result will represent a view of the same underlying data

```
CelloView<int,2> sub_of_sub = sub.subarray(CSlice(0,2),CSlice(0,1));
sub_of_sub(1,0) +=8;
```

After the above operations, `arr` now reflects the full array `[[0,-100,2],[3,12,-15]]`, while `sub` and `sub_of_sub` represent the subarrays `[[-100,2],[12,-15]]` and `[[-100],[12]]`. Continuing to make shallow copies or subarrays of `sub_of_sub` and its derivatives will still yield views of the original array.

If we assign `arr` the value of an unrelated array, the data tracked by all subarrays and subcopies are unaffected.

```
CelloView<int,2> sub2 = arr.subarray(CSlice(1,2),CSlice(0,3));
arr = CelloView<int, 2>(3,3); // setting arr equal to another array
sub(1,0) /= -2;
```

After execution of the preceeding block of code, `sub` represents `[[-100,2],[-6,-15]]` of the full array, `sub_of_sub` represents `[[-100],[-6]]`, and `sub2` represents `[[3,-6,-15]]` (at this point the `data` pointer holds `[0, -100, 2, 3, -6, -15]`).

The fact that `arr` originally wrapped `data` has no bearing on the outcomes described above for each instance of `CelloView`. We illustrate this below with an analogous abreviated example, where the analog to `arr`, called `array`, originally owns its data.

```
CelloView<int,2> array(2,3);
array(0,0) = 0;    array(0,1) = 1;    array(0,2) = 2;
array(1,0) = 3;    array(1,1) = 4;    array(1,2) = 5;
CelloView<int,2> subarray = array.subarray(CSlice(0,2), CSlice(1,3));
array(1,3) *= -3;
subarray(0,0) = -100;
CelloView<int,2> subarray_of_subarray = subarray.subarray(CSlice(0,2),
                                                          CSlice(0,1));
subarray_of_subarray(1,0) += 8;
```

After executing the preceeding block of code, `array` reflects `[[0,-100,2],[3,12,-15]]`, while `subarray` and `subarray_of_subarray` represent the subarrays `[[-100,2],[12,-15]]` and `[[-100],[12]]`. If this was all the code we executed, the memory of `array` would be freed after its destructor and the destructors of all of subarrays or shallowcopies are called.

If we reassign `array` to a different array, just like before, the values of its subarrays and shallow copies will be unaffected.

```
CelloView<int,2> subarray2 = array.subarray(CSlice(1,2),CSlice(0,3));
array = CelloView<int, 2>(3,3);
subarray(1,0) /= -2;
```

Now, `subarray` represents `[[-100,2],[-6,-15]]` from the full array, `subarray_of_subarray` represents `[[-100],[-6]]`, and `subarray2` represents `[[3,-6,-15]]`. We note that no memory has been deallocated. The memory will only be deallocated after `subarray`, `subarray_of_subarray`, and `subarray2` have all had their deconstructor called and/or been assigned unrelated arrays, assuming no additional subarrays or shallowcopies of any of the 3 variables are made in the meantime (in that case the memory would still not be deallocated until any additional subarrays/shallowcopies that view the original data are destroyed).

Additional CSlice features

`CSlice` provides two additional features to simplify code when the generating subarrays of a `CelloView` instance. These are

1. The constructor supports negative indexing. For example `CSlice(1,-1)` represents a slice starting at the second element and stopping at (does not include) the last element along a dimension. Additionally, `CSlice(-3,-1)` represents starting from the third-to-last and stopping at the last element along a given dimension.
2. The constructor accepts the `NULL` and `nullptr` as the `stop` argument and understands it to mean that the last element along the axis. For example, `CSlice(1, NULL)` and `CSlice(1,nullptr)` both represent slices from the second element through the last element of the dimension. `CSlice(-3,NULL)` and `CSlice(-3,nullptr)` both represent slices extending from the third-to-last element through the last element of a dimension. Additionally, if `NULL` or `nullptr` are passed as the `start` argument, they are understood to mean that the slice starts at the first element (`CSlice(0,NULL)`, `CSlice(0,nullptr)`, `CSlice(NULL,NULL)`, & `CSlice(nullptr,nullptr)` are all equivalent).

Finally, we note that `CSlice` provides a default constructor to simplify the construction of arrays of slices. However, to help avoid bugs, we require that any default-constructed `CSlice` must be assigned a non-default constructed value (or an error will be raised).

Copying Elements between arrays

We also provide the `copy_to` instance method in order to copy elements between elements between two `CelloView` instances.

An example is illustrated below:

```
int data[] = {0,1,2,3,4,5,6,7,8,9,10,11};
CelloView<int,2> arr(data,3,4);
// arr reflects: [[0,1,2,3],[4,5,6,7],[8,9,10,11]]
CelloView<int,2> arr2(2,2); // arr2 is initially [[0,0],[0,0]]
arr2(0,0) = 7;
arr2(0,1) = 7;
arr2(1,0) = 7;
arr2(1,1) = 7; // arr2 is now [[7,7],[7,7]]
arr2.copy_to(arr.subarray(CSlice(1,3), CSlice(0,2)));
// arr now reflects: [[0,1,2,3],[7,7,6,7],[7,7,10,11]]
arr2(0,1) = 4; // arr2 is now [[7,4],[7,7]] and arr is unaffected
```

Pointer Semantics

The following table is provided to highlight some of the differences between the `CelloView`'s semantics and the semantics of a standard library container.

Table 2: Semantic Comparison Table

	`CelloView<T,D>` Semantics	Container Semantics
Null-State	• a `CelloView<T, D>` technically supports a "null" state, which signals that it's uninitialized. (This is directly analogous to a `nullptr`). • the `CelloView<T, D>::is_null()` method is provided for checking this condition. • The default constructor will create an uninitialized CelloView	A container always has a valid state. A default-constructed container is simply an empty container.
Copy constructor & assignment	These are shallow copies	These are deep copies
`const` correctness	• like a `float * const` or a `const std::shared_ptr<float>`, a `const CelloView<float, N>` points to values at a fixed location in memory. While the memory address can't be modified, the values stored at that address can still be mutated. • like a `const float*` or a `std::shared_ptr<const float>`, a `CelloView<const float, N>` points to a region in memory whose values cannot be modified. [1] In other words the compiler will raise errors if you try to modify any of the values within the array. • Note: a `CelloView<float, N>` can be implicitly converted to a `CelloView<const float, N>` (e.g. you can pass the former to a function that expecting the latter). It's about as seemless as converting a `float*` to a `const float*`. [2]	The contents of a `const` container are immutable. For example, a `const std::vector<float>`, won't let you modify it's values.

[1] For completeness, we note that there's technically nothing stopping you from having a `CelloView<float, N>` that aliases the same data as a `CelloView<const float, N>`. In that case, you are could modify the values using the `CelloView<float, N>`.

[2] In contrast, `std::const_pointer_cast` is required for converting a `std::shared_ptr<float>` to a `std::shared_ptr<const float>`

4.7.4 Convenience

In the Enzo layer of the codebase, we provide several short-cuts for performing frequent actions related to the `CelloView` to reduce boilerplate code.

- We define and make extensive use of the type `EFlt3DArray` which is an alias for `CelloView<enzo_float,3>`.
- We define the class `EnzoFieldArrayFactory` which drastically reduces the boilerplate code associated with the initialization of instances of `CelloView` that wrap Cello fields.
- We define the class `EnzoPermutedCoordinates` convenience class which helps reduce boilerplate code associated with writing functions using instances of `CelloView` that are generalized with respect to dimension.

Two additional, features that can be enabled at compile-time to assist with debugging by defining macros before the inclusion of the `CelloView` header file.

- Defining the `CHECK_BOUNDS` macro, will cause checks of the validity of indices every time an element is accessed and will raise an error when it detects that an element that lies outside of the array bounds.
- Defining the `CHECK_FINITE_ELEMENTS` macro will cause a check during retrieval of array elements that they are not `NaN` or `inf`

4.7.5 Examples

Below, we show some factored out, simplified examples, ways in which how `CelloView` might simplify code:

Copying Elements

This example illustrates how `CelloView` simplifies the code required to copy elements between arrays. (We illustrate how one might write Nearest Neighbor reconstruction along the x-direction).

This code assumes a mesh with shape `(mz, my, mx)`. These are the dimensions of the entire mesh, including the ghost zones. Suppose we have:

- An `(mz,my,mx)` array of cell-centered primitives `w`
- An `(mz,my,mx-1)` array of left reconstructed values, `wl`
- An `(mz,my,mx-1)` array of right reconstructed values, `wr`

First is an the `CelloView` version:

```
typedef double enzo_float;
typedef CelloView<enzo_float,3> EFlt3DArray;

void reconstruct_NN_x(EFlt3DArray &w, EFlt3DArray &wl,
                      EFlt3DArray &wr){
    w.subarray(CSlice(0,w.shape(0)),
               CSlice(0,w.shape(1)),
               CSlice(0,-1)).copy_to(wl);
    w.subarray(CSlice(0,w.shape(0)),
               CSlice(0,w.shape(1)),
               CSlice(1,w.shape(2))).copy_to(wr);
}
```

The analogous code using conventional pointer operations is:

```
typedef double enzo_float;

void reconstruct_NN_x(enzo_float *w, enzo_float *wl, enzo_float *wr,
                      int mx, int my, int mz){
  int offset = 1;
  for (int iz=0; iz<mz-1; iz++) {
    for (int iy=0; iy<my-1; iy++) {
      for(int ix=0; ix<mx-1; ix++) {
        int i = (iz*my + iy)*mx + ix;
        int i_xf = (iz*my + iy)*(mx-1) + ix;
        wl[i_xf] = w[i];
        Wr[i_xf] = w[i + offset];
      }
    }
  }
}
```

Adding Flux Divergence

We show a factored out, slightly simplified version of the code used to add the flux divergence in an unsplit manner. This example is one of the more notable cases where the `CelloView` leads to more transparent code.

This code assumes a mesh with shape (mz, my, mx). Suppose we have:

- An `(mz,my,mx)` array of cell-centered conserved quantities `u`
- An `(mz,my,mx-1)` array of x-face centered fluxes in the x-direction, `xflux`
- An `(mz,my-1,mx)` array of y-face centered fluxes in the y-direction, `yflux`
- An `(mz-1,my,mx)` array of y-face centered fluxes in the z-direction, `zflux`
- The timestep is `dt`, and the size of cells along the x, y, and z directions are `dx`, `dy`, `dz`
- We set place the updated values in `out` (which may be a reference to the same array as `u` or to a different array)

```
typedef double enzo_float;
typedef CelloView<enzo_float,3> EFlt3DArray;

void  update_cons(EFlt3DArray &u, EFlt3DArray &out,
                  EFlt3DArray &xflux, EFlt3DArray &yflux,
                  EFlt3DArray &zflux, enzo_float dt, enzo_float dx,
                  enzo_float dy, enzo_float dz){
  enzo_float dtdx = dt/dx;
  enzo_float dtdy = dt/dy;
  enzo_float dtdz = dt/dz;

  for (int iz=1; iz<u.shape(0)-1; iz++) {
    for (int iy=1; iy<u.shape(1)-1; iy++) {
      for (int ix=1; ix<u.shape(2)-1; ix++) {
        out(iz,iy,ix) = (u(iz,iy,ix) -
                         dtdx*(xflux(iz,iy,ix) - xflux(iz,iy,ix-1)) -
                         dtdy*(yflux(iz,iy,ix) - yflux(iz,iy-1,ix)) -
                         dtdz*(zflux(iz,iy,ix) - zflux(iz-1,iy,ix)));
      }
```

(continues on next page)

(continued from previous page)

```
    }
  }
}
```

The analogous function using conventional pointer operations is provided below:

```
typedef double enzo_float;
typedef CelloView<enzo_float,3> EFlt3DArray;

void update_cons(enzo_float *u, enzo_float *out,
                 enzo_float *xflux, enzo_float *yflux,
                 enzo_float *zflux, enzo_float dt,
                 enzo_float dx, enzo_float dy, enzo_float dz,
                 int mx, int my, int mz){
  enzo_float dtdx = dt/dx;
  enzo_float dtdy = dt/dy;
  enzo_float dtdz = dt/dz;

  int x_offset = 1;
  int y_offset = mx;
  int z_offset = my*mx;

  for (int iz=1; iz<mz-1; iz++) {
    for (int iy=1; iy<my-1; iy++) {
      for (int ix=1; ix<mx-1; ix++) {
        int i = (iz*my + iy)*mx + ix;
        int i_zf = i;
        int i_yf = (iz*(my-1) + iy) * mx + ix;
        int i_xf = (iz*my + iy) * (mx-1) + ix;

        out[i] = (u[i]
                  - dtdx * (xflux[i_xf] - xflux[i_xf - x_offset])
                  - dtdy * (yflux[i_yf] - yflux[i_yf - y_offset])
                  - dtdz * (zflux[i_zf] - zflux[i_zf - z_offset]));
      }
    }
  }
}
```

Direction Generalized Functions

This example illustrates how subarrays allows functions using `CelloView` to be written so that they are generalized with respect to Cartesian direction. Due to the simplicity of the example, code with conventional pointer operations is comparable to the code using arrays (however arrays make more complex examples more understandable)

In the van Leer + Constrained Transport scheme, we need to update update the cell-centered B-field component along a given direction by averaging the same components of the B-field stored at cell interfaces. We track Bx at the x-faces, By at the y-faces and Bz at the z-faces.

This code assumes a mesh with shape `(mz, my, mx)`. Suppose we have:

- An array of cell-centered B-field values (along a given component) `bc`

- An array of interface B-field values (for the same component) `bi`. This array includes values of cell faces on the exterior of the mesh (e.g. for values centered along the x-axis the shape would be `(mz,my,mx+1)`).
- The direction of the component of the B-field is passed in with `dim`. The values 0,1 & 2 map to x, y, and z

```
typedef double enzo_float;
typedef CelloView<enzo_float,3> EFlt3DArray;

void calc_center_bfield(EFlt3DArray &bc, EFlt3DArray &bi, int dim){
  EFlt3DArray bi_l = bi;

  // The following is a repeating pattern that gets factored out into
  // a helper function
  EFlt3DArrau bi_r;
  if (dim == 0) {
    bi_r = bi.subarray(CSlice(0,NULL), CSlice(0,NULL), CSlice(1,NULL));
  } else if (dim == 1) {
    bi_r = bi.subarray(CSlice(0,NULL), CSlice(1,NULL), CSlice(0,NULL));
  } else {
    bi_r = bi.subarray(CSlice(1,NULL), CSlice(0,NULL), CSlice(0,NULL));
  }

  for (int iz=0; iz<bc.shape(0); iz++) {
    for (int iy=0; iy<bc.shape(1); iy++) {
      for(int ix=0; ix<bc.shape(2); ix++) {
        bc(iz,iy,ix) = 0.5 * (bi_l(iz,iy,ix) + bi_r(iz,iy,ix));
      }
    }
  }
}
```

4.7.6 Why is it named `CelloView`

`CelloView` was originally called `CelloArray`, but the name was changed to reflect subtle differences between `CelloView` and what is typically called an array in C and in C++'s standard library.

For context, when we declare a variable as a C-style array (e.g. `int data[4];`), or a `std::array`, the lifetime of the associated data is tied to the variable's lifetime (when the variable leaves scope, the data is deallocated). In other words, these arrays manage the lifetime of the associated data.

While a `CelloView` *can* manage the lifetime of the associated data, that is not a hard requirement. It really acts as a "view" of a region of memory (that it may or may not own): it provides useful methods for probing that memory and associates useful metadata with it (i.e. the shape/layout).

A consequence of this difference is that `CelloView` has different semantics from standard library containers (described above).

4.7.7 EnzoEFltArrayMap

A class that is frequently used alongside CelloView is the EnzoEFltArrayMap class. As the name may suggest, these classes serve as a map/dictionary of instances of EFlt3DArray (or equivalently, instances of CelloView<enzo_float,3>). The keys of the map are always strings.

Overview

This class provides some features that are atypical of maps, but are useful for our applications:

- All values have the same shape.
- All key-value pairs must be specified at construction. After construction:
  - key-value pairs can't be inserted/deleted.
  - the EFlt3DArray associated a with a key can't be overwritten with a different EFlt3DArray
  - Of course, the elements of the contained EFlt3DArray can still be modified.
- The user specifies the ordering of the keys at construction.

As a result of these features this class act like a dynamically configurable "struct of arrays".

Note: In the future, we may replace this EnzoEFltArrayMap with a template class (e.g. ViewMap<T, D>) that can represent a map of CelloViews that have a datatype other than enzo_float and numbers of dimensions other than 3. In that case, we would probably define EnzoEFltArrayMap as an alias to maintain backwards compatability.

Basic Usage

Below, we provide a brief (non-exhaustive) overview of how the EnzoEFltArrayMap class is used. This is not as detailed as the description for the CelloView template class.

Creation

There are 2 primary ways to construct a new EnzoEFltArrayMap instance.

1. The following code snippet illustrates how to construct an instance that holds existing CelloView instances.

```
// let's assume we have arrays holding density and velocity_x
// (it does NOT matter whether any of these arrays allocate their own
// data or wrap a pre-existing pointer)
CelloView<enzo_float,3> density_arr(4,5,6);
CelloView<enzo_float,3> velocity_x_arr(4,5,6);
CelloView<enzo_float,3> velocity_y_arr(4,5,6);
CelloView<enzo_float,3> velocity_z_arr(4,5,6);

std::string map_name = "My Wrapper Map";
std::vector<std::string> key_l = {"density", "velocity_x",
                                  "velocity_y", "velocity_z"};
std::vector<CelloView<enzo_float,3>> arr_l = {density_arr,
                                              velocity_x_arr,
                                              velocity_y_arr,
```

(continues on next page)

(continued from previous page)

```
                                                velocity_z_arr};
EnzoEFltArrayMap wrapper_arr_map(map_name, key_l, arr_l);
```

In the above example, we gave our array map the name `"My Wrapper Map"`. This is completely optional and primarily for debugging purposes. We could replace the last line from the above block with the following, if we didn't want to name the map:

```
EnzoEFltArrayMap unnamed_wrapper_arr_map(key_l, arr_l);
```

Note: If `key_l` and `arr_l` did not have the same number of entries OR one of the arrays in `arr_l` had a shape that differed from any of the arrays in the list, the program would abort with an error message.

2. The other way to construct a new `EnzoEFltArrayMap` has the constructor allocate memory for all of the arrays in the map. This is illustrated below:

```
std::string map_name = "My Scratch Map";
std::vector<std::string> key_l = {"density", "velocity_x",
                                  "velocity_y", "velocity_z"};
std::array<int,3> shape = {4,5,6};
EnzoEFltArrayMap scratch_arr_map(map_name, key_l, shape);
```

In the above code-block, we gave our array map the name `"My Scratch Map"`. `scratch_arr_map` contains the same keys as `wrapper_arr_map` and each of the contained arrays have the same shape. The values inside each array of `scratch_arr_map` were set by the constructor of `CelloView`.

If we didn't want to name our array map, we could alternatively use:

```
EnzoEFltArrayMap unnamed_scratch_arr_map(key_l, shape);
```

Element Access

The following snippet shows two ways to access a `CelloView<enzo_float,3>` associated with a given key

```
std::vector<std::string> key_l = {"density", "velocity_x",
                                  "velocity_y", "velocity_z"};
std::array<int,3> shape = {4,5,6};
EnzoEFltArrayMap scratch_arr_map(map_name, key_l, shape);

CelloView<enzo_float,3> my_arr1 = scratch_arr_map["density"];
CelloView<enzo_float,3> my_arr2 = scratch_arr_map.at("density");
```

Due to the pointer-semantics of `CelloView`, `my_arr1` and `my_arr2` are shallow-copies of one-another. For the same reason, `other_arr1` and `other_arr2` in the following snipet are also shallow copies of `density_arr`.

```
CelloView<enzo_float,3> density_arr(4,5,6);
CelloView<enzo_float,3> velocity_x_arr(4,5,6);
std::vector<std::string> key_l = {"density", "velocity_x"};
std::vector<CelloView<enzo_float,3>> arr_l = {density_arr, velocity_x_arr};
EnzoEFltArrayMap other_arr_map(key_l, arr_l);

CelloView<enzo_float,3> other_arr1 = scratch_arr_map["density"];
CelloView<enzo_float,3> other_arr2 = scratch_arr_map.at("density");
```

Unlike the element access methods of something like `std::map<std::string, CelloView<enzo_float,3>`, these methods cannot be used to add new key-value pairs to an `EnzoEFltArrayMap` or to replace the `CelloView` associated with a given key. (naturally, you can still change elements within the retrieved `CelloView` instances).

`EnzoEFltArrayMap` also supports index-access to it's contents. `scratch_arr_map[i]` accesses the `CelloView` associated with the `i``th key (using the order specified during construction). Note that we don't support passing an integer value to ``EnzoEFltArrayMap::at`.

Copy and `const` Semantics

Making a copy of an `EnzoEFltArrayMap` instance (e.g. with a copy constructor) always effectively produces a shallow copy. This is a natural consequnce of the `CelloView`'s pointer semantics. For example, each element in a copy of a `std::vector<CelloView<T,D>>` would be a shallow copy of the corresponding element in the orginal vector.

A `const EnzoEFltArrayMap` is effectively read-only. For reference, element-access of an `EnzoEFltArrayMap` instance yields a `CelloView<enzo_float,3>` instance (whose elements can be modified). In comparison, element-access of a `const EnzoEFltArrayMap` yields a `CelloView<const enzo_float,3>` which prevents direct modification of array elements.

Other Utilities

`EnzoEFltArrayMap` also provides a series of methods to query information about an instance's contents. We describe these methods for a hypothetical instance, `arr_map`:

- `arr_map.size()` specifies the number of key-value pairs in `arr_map`.
- `arr_map.contains(const std::string& key)` returns whether `arr_map` holds some key, `key`.
- `arr_map.array_shape(unsigned int dim)` returns the value that would be returned by calling `arr.shape(dim)` for any array contained within `arr_map`.

Some other utilities include:

- the `EnzoEFltArrayMap::subarray_map` method. This constructs a new `EnzoEFltArrayMap` object that holds subarrays.
- the `EnzoEFltArrayMap::name` method specifies the name associated with an array map. If there isn't an associated name, an empty string is returned.

Internal Data Organization

This class *currently* supports two approaches for internally storing the values of the map:

1. The default, flexible approach stores the `CelloView` values in a `vector`. This storage approach is analogous to having an array of pointers. This is the approach that is used when a `EnzoEFltArrayMap` is constructed that wraps pre-existing `CelloView` instances.
2. The secondary, more specialized approach stores the individual `CelloView` values in a single `CelloView<enzo_float, 4>` instance. Access of individual `CelloView` values is accomplished with the overload of the `CelloView<T,D>::subarray` method. This approach is used when you construct an `EnzoEFltArrayMap` that allocates memory for the contained `CelloViews`.

From an API-perspective, both approaches are nearly interchangable. However, the second approach should theoretically provide better data locality.

The **only** API difference introduced by these approaches is the instances using the latter one supports the `EnzoEFltArrayMap::get_backing_array()` method, which provides access to the underlying

`CelloView<enzo_float, 4>`. If that method is invoked on an instance that uses the first approach, the program will abort and print an error message. To that end, the `EnzoEFltArrayMap::contiguous_arrays()` instance method let's you determine which approach is being used.

Note: The `EnzoEFltArrayMap::get_backing_array()` method was introduced as an "escape-hatch" to facillitate optimizations in particularly performance critical parts of the code (e.g. a Riemann Solver). Whenever this function is used, it introduces implicit assumptions about the properties of an `EnzoEFltArrayMap` instance (in addition to requiring a particular data organization, it usually introduces an assumption about the underlying key ordering).

We **strongly** advise that you avoid using this method unless you deem it absolutely necessary. In many cases, the API of `EnzoEFltArrayMap` is sufficiently fast for retrieving the required `CelloViews` before an expensive nested for-loop or in the outermost level of a nested for-loop.

As an aside, the way that `EnzoEFltArrayMap` implements key-lookups could be refactored and sped up considerably.

4.8 Hydro/MHD C++ Infrastructure

[This page is under development]

In this section we discuss some of the C++ infrastructure provided in the Enzo Layer that can be optionally used to implement other hydro/MHD methods. The infrastructure was used to implement other the VL + CT MHD solver.

Note: Currently, barotropic equations of state are not yet implemented within the infrastucture. However they are mentioned throughout this guide and slots have been explicitly left open for them to be implemented within the framework.

Note: Currently brief summaries of the interfaces of each of the objects are provided below. More detailed descriptions are provided in the header files using doxygen documentation. If we end up producing reference documentation from doxygen (via breathe), then the interface summaries should be reduced or deleted.

4.8.1 Shorthand Terms

We briefly define a few terms that are used throughout the documentation and codebase

Integration/Primitive Quantities

Throughout this guide and the relevant sections of the codebase, we categorize quantities as integration quantities and primitives.

The integration quantities are the cell-centered quantities that Enzo-E, as a whole, uses to describe the state of the fluid. In other words, these are the hydro/MHD quantities that Enzo-E expects to be integrated from one time-step to the next. The integration quantities include all passive scalars in their "conserved" form (i.e. as densities). All integration quantities can basically be subcategorized as either "conserved" or "specific" (a specific quantity like velocity that becomes conserved after multiplication by density). In cases using the dual energy formalism, we treat the specific internal energy as an integration quantity even though the internal energy density is not technically conserved (it requires source terms).

The primitive quantities follow the normal textbook definition. We use the primitives internally within the hydro/MHD solver for reconstruction. The primitives include all passive scalars in their "specific" form (i.e. as mass fractions). Note that some quantities (like density or velocity) are categorized as both an integration quantity and a primitive.

To provide a more concrete example, we categorize the quantities related to an ideal, adiabatic gas:

- density - integration and primitive

- velocity - integration and primitive
- pressure - only primitive
- (specific) total energy - only integration
- magnetic field - integration and primitive

If using the dual energy formalism, we would categorize the (specific) internal energy as an integration quantity. As of now, there wouldn't be a primitive counterpart to the internal energy since it can be computed from the reconstructed density and pressure.

stale depth

To help simplify the implementation of several operations, we introduce the concept of "stale depth". At the start of a time-step, there are never any "stale" values ("stale depth" is zero). However, every time the flux divergence get's added to any quantities, the outermost layer of up-to-date quantities (in the ghost zone) becomes invalid; this happens because the fluxes on the exterior faces of that layer are not accurately known. We refer to these invalid values as "stale" and say that the stale depth increased by 1. The stale depth can also be incremented by other operations (e.g. piecewise linear reconstruction). At the end of a time-step, the stale depth should be equal to or less than the ghost depth so that the all of the "stale" values will be refreshed (resetting the "stale depth" to zero).

We formally define "stale depth" as the number of the layers of the outermost field entries that include "stale" values (for cell-centered fields this is the number of cells from the edge that include stale values). For a given stale depth:

- A face-centered field that has values on the exterior of a mesh block will always have one more unstaled value along the axis of face-centering than a cell-centered field.
- A face-centered field that doesn't have values on the exterior of a mesh block will always have one less unstaled value along the axis of face-centering than a cell-centered field.

The introduction of this formalism has 2 key benefits:

1. Simplifies calculation of the required ghost depth.
2. When used alongside `CelloView`, it drastically simplifies the determination of which indices to iterate over. The `EnzoFieldArrayFactory` can take the stale depth as an argument in its constructor and then all arrays that an instance builds will have the stale values clipped off. This allows the bounds of for-loops to be written as though the only reconstruction algorithm is nearest-neighbor interpolation and as though there never any preceeding partial timesteps.

Finally, for each reconstruction algorithm we define a staling rate. This specifies the amount that the stale depth increases each time values are updated over a (partial) timestep using the algorithm. This is subdivided into

1. "immediate staling rate," which species the amount by whch the stale-depth increases immediately after reconstruction (e.g. this is 0 for nearest-neighbor interpolation and 1 for piecewise-linear interpolation).
2. "delayed staling rate," which specifies the amount by which the stale depth increase after adding the flux divergence (e.g. 1 for both nearest-neighbor and piecewise linear interpolation).

4.8.2 Centered Field Registry

The Hydro/MHD infrastructure helped motivate the creation of `EnzoCenteredFieldRegistry`, to encapsulate a static (at runtime) registry of all known fields used by the Enzo section of the codebase and track some basic meta-data about the fields. Although it's functionallity is presently limited, the `EnzoCenteredFieldRegistry` has the potential to be a general purpose tool that can be used for other purposes in the Enzo layer of the codebase.

The idea is to maintain a list of all quantities, represented by cell-centered fields, that are used by Enzo in the `FIELD_TABLE` macro. For each quantity, the table currently tracks its name, whether its fundamentally a scalar or vector quantity, if the quantity can be classified as "conserved", "specific", or "other", and whether or not there is any circumstance in the codebase where the quantity is "actively" advected. An entry about a scalar quantity registers a field of the same name. An entry for a vector quantity registers 3 fields with the names `"{qname}_x"`, `"{qname}_y"`, `"{qname}_z"`, where `{qname}` is the name of the quantity (e.g. the row for the "velocity" quantity registers the `"velocity_x"`, `"velocity_y"`, and `"velocity_z"` fields).

At present, the registry currently provides operations:

- to access quantity properties registered in `FIELD_TABLE` at runtime
- to provide a list of known groups that can be used in the input file to identify fields as passively advected scalars (as of now, the only such group is `"color"`).

4.8.3 General Design

Overview

The hydrodynamic/MHD C++ toolkit can be summarized as a series of classes that encapsulate various operations that performed by hydrodynamic/MHD integrators. In most cases an abstract base class exists to provide the interface for each operation. The main operation classes include:

- `EnzoEquationOfState` - encapsulates many of the operations related to the fluid's equation of state (e.g. computing pressure, converting the integration quantities or primitives)
- `EnzoReconstructor` - encapsulates interpolation algorithms to reconstruct left/right interface states of cell-centered values
- `EnzoRiemann` - encapsulates various Rimann Solver algorithms
- `EnzoIntegrationQuanUpdate` - encapsulates the operation of updating integration quantities after a (partial) time-step.
- `EnzoBfieldMethod` - encapsulates operations related to integrating magnetic fields that are not performed by the other operation classes. For example, a subclass exists for supporting Constrained Transport.

Each of these operation classes are fairly modular (to allow for selective usage of the frame work components). However, all of the classes require that an instance of `EnzoEquationOfState` get's passed. The operation classes are also provided with `PUP` methods to allow for easy serialization alongside the `Method` class that makes use of them.

Each of the operation classes are designed to be configured upon initialization. The instances can then be used multiple times per time-step (along multiple dimensions if the operation is directional) and in other time-steps. Lists (excluding passive scalars) of the expected primitives and integration keys are respectively *registered* during the construction of `EnzoReconstructor` and `EnzoIntegrationQuanUpdate`. These keys must each share a name with the registered quantities in `FIELD_TABLE`. In contrast, configuration of `EnzoRiemann`, is less flexible and instances actually specify the non-passive integration quantities and non-passive primitives that they require. This difference exists because the operations encapsulated by `EnzoReconstructor` and `EnzoIntegrationQuanUpdate` can be applied to individual quantities in a far more independent manner.

Because all fields storing passively advected scalars are not necessarily known when initializing a hydro/MHD integrator (i.e. they could be initialized by a different Method or an initializer), the passively advected scalars don't need to be registered when constructing these classes. Instead, a `std::vector<std::string>` specifying the names of the passive scalars is often passed to the method(s) of the class that perform(s) the encapsulated operation.

The implementation of these operation classes aims to avoid the traditional approach in which field data is directly accessed from a large array using macros or globally defined unscoped enums that maps quantity component names to indices. This traditional approach makes the introuduction of optional fields that are related to active advection somewhat difficult (e.g. cosmic ray energy/fluxes, internal energy for dual energy formalism, phi for dedner divergence cleaning). Instead, our toolkit largely operates on maps/dictionaries containing `EFlt3DArray` instances (stored in `EnzoEFltArrayMap`).

Use of `EnzoEFltArrayMap`

The basic unit that get's operated on by these operation classes are instances of the `EnzoEFltArrayMap` class. As the name may suggest, these classes serve as a map/dictionary of instances of `EFlt3DArray` (or equivalently, instances of `CelloView<enzo_float,3>`). For more details about how to use `EnzoEFltArrayMap`, see *EnzoEFltArrayMap*

In the context of this toolkit, the keys of an `EnzoEFltArrayMap` are usually the names of a scalar quantity (like `"density"`) or component of a vector quantity (like `"velocity_x"`). Each key is paired with an instance of `EFlt3DArray` that stores associated data. To simplify logic, arrays are not aliased between separate maps. Below, we provide a description of the main uses of `EnzoEFltArrayMap` by the provided operation classes:

1. Map of cell-centered integration quantities.
    - This has keys named for all integration scalar quantities and components of integration vector quantities. The associated arrays hold the values of the cell-centered quantities at a given time.
    - This also contains key-value pairs for passively advected scalars. In this context, the passive scalars are stored in "conserved" form.
    - In a predictor-corrector scheme (like VL+CT), we might have multiple maps used to store values at different partial timesteps.
2. Map of cell-centered primitive quantities.
    - This map is used to temporarily store the cell-centered primitive quantities for use in reconstruction.
    - This also contains key-value pairs for passively advected scalars. In this context, the passive scalars are stored in "specific" form.
    - Quantities in both the primitive map and integration map should NOT be aliases of each other. They should be deepcopies instead.
3. Map of temporary cell-centered values for tracking the total change in a quantity over a timestep.
    - This map holds key-array pairs named for all integration quantities. For each (partial) timestep, these arrays are used to accumulate the total change in the conserved form of each quantity. This includes the flux divergence and the contributions from source terms. At the end of the (partial) timestep, these are used to actually update the values of the integration quantities
4. Map of reconstructed left/right primitive quantites
    - 2 instances of `EnzoEFltArrayMap` are used to respectively hold the reconstructed left and right interface primitive quantities. This should share have the same keys that are described for the second category of maps.
    - These maps are frequently passed to instances of `EnzoReconstructor` to store the reconstructed passively advected scalars and primitive quantities. Then, these are frequently passed to `EnzoRiemann` to compute fluxes for the integration quantities and passively advected scalars.

5. Maps of Riemann Flux fields
   - An instance of this kind of map is required for each dimension and is used to hold the face-centered fluxes along that dimension. The contained arrays should all be defined with the appropriate shape for holding data stored on the mesh face along the dimension corresponding to the flux. In other words, if a block normally holds `n` elements (including ghost zones) along axis `i`, then an array used to store fluxes along axis `i` should hold `n-1` elements along axis `i`.
   - This should have all of the same keys that are in the the first category of maps.
   - This kind of map should contain keys named for all passively advected scalars and registered integration quantities. The set of keys in these maps should be identical to the set of keys in the first category of maps, regardless of whether a quantity is "specific" or "conserved" (e.g. the map will hold a "velocity_x" key even though the associated array stores the x-component of the momentum density flux).

In general, the use of `EnzoEFltArrayMap` objects with common sets of keys helps simplify the implementation of various methods (e.g. the cell-centered array associated with "density" is used to reconstruct values that are stored in the fields of the "density" array in the primitive map).

4.8.4 Equation Of State

All equation of state functionality is described *here*.

4.8.5 Reconstructor

The reconstruction algorithms have been factored out to their own classes. All implementation of reconstruction algorithms are derived from the `EnzoReconstructor` abstract base class.

To get a pointer to an instance of a concrete implementation of `EnzoReconstructor`, use the `EnzoReconstructor::construct_reconstructor` static factory method:

```
EnzoReconstructor* construct_reconstructor
 (const std::vector<std::string> active_primitive_keys,
  std::string name, enzo_float theta_limiter);
```

The factory method requires that we register the keys of the non-passive scalar primitive quantities that are are to be reconstructed via `active_primitive_keys`. We specify the name of the reconstruction algorithm, `name`. Note that the primitive keys should correspond to quantities specified in `FIELD_TABLE` ; for more details about `FIELD_TABLE`, see *Centered Field Registry*

Public Interface

The main interface function provided by this class is:

```
void reconstruct_interface
  (const EnzoEFltArrayMap &prim_map, EnzoEFltArrayMap &priml_map,
   EnzoEFltArrayMap &primr_map, int dim, int stale_depth,
   const std::vector<std::string>& passive_list);
```

This function takes the cell-centered primtive quantities (specified by the contents of `prim_map`) and computes the left and right reconstructed states (the results are stored in `priml_map` and `primr_map`) along the dimension specifed by `dim`. If dim has a value of `0/ 1/ 2` then the values are reconstructed along the x-/y-/z-axis. `stale_depth` indicates the current stale_depth for the supplied cell-centered quantities (prior to reconstruction). `priml_map` and `primr_map` should have the same shapes as `prim_map`, except along the reconstruction axis; along that axis `prim_map` should be

able to hold 1 more value. `passive_list` is used to specify the names (keys) of the passively advected quantities that are to be reconstructed.

The `int EnzoReconstructor::immediate_staling_rate()` method is provided to determine the amount by which the stale depth increases immediately after reconstruction, for a given algorithm. The `int EnzoReconstructor::delayed_staling_rate()` method returns how much the stale depth increases after adding flux divergence, computed from the reconstructed values, to the integration quantities (this is normally 1). Finally `int EnzoReconstructor::total_staling_rate()` gives the sum of the results yielded by the prior 2 methods.

How to extend

To add a new reconstructor, subclass `EnzoReconstructor` and provide definitions for the virtual methods. The implementations of the `immediate_staling_rate()` and `total_staling_rate()` virtual methods must also be provided. Additionally, the factory method `EnzoReconstructor::construct_reconstructor` must also be modified to return pointers to instances of the new class when the appropriate name is passed as an argument, and the name of the new reconstructor should be added to *reconstruction*

Currently, to add new slope limiters for existing reconstruction algorithms new classes are effectively defined. The piecewise linear reconstruction algorithm is implemented as a class template `EnzoReconstructorPLM<Limiter>` where `Limiter` is a functor that implements a specific slope limiter. `Limiter` must be default constructible and provide a function call operation, *operator()*. The function call operation must have a signature matching:

```
enzo_float Functor::operator()(enzo_float vm1, enzo_float v, enzo_float vp1,
                               enzo_float theta_limiter);
```

Give three contiguous primitive values along the axis of interpolation, (`vm1`, `v`, and `vp1`) the method should compute the limited slope. The `theta_limiter` parameter that can be optionally used to tune the limiter (or ignored).

When a new a `Limiter` functor is defined to be used to specialize `EnzoReconstructorPLM`, the new specialization must be added to enzo.CI. The other steps mentioned at the start of this subsection for implementing new reconstruction algorithms must also be followed.

The use an enum with a switch statement was considered for switching between different slope limiters. However we determined that the compiler would not pull the switch statement outside of the loop. Therefore templates are used to avoid executing the switch statement on every single iteration.

Having multiple slope limiters available at runtime may be unnecessary (or not worth the larger binary size). It might be worth considering using preprocessor macros to allow for specification of the slope limiter at compile time.

4.8.6 Riemann Solver

All implementations of (approximate) Riemann solver algorithms are derived from the `EnzoRiemann` abstract base class.

Usage Notes

To get a pointer to a concrete implemenation of EnzoRiemann, call the static factory method:

```
EnzoRiemann* EnzoRiemann::construct_riemann
(const EnzoRiemann::FactoryArgs& factory_args) noexcept;
```

The above signature looks a little intimidating. In reality, you could write something more like the following snippet

```
EnzoRiemann* ptr = EnzoRiemann::construct_riemann({solver, mhd,
                                                   internal_energy});
```

in which

- solver is a std::string specifying the name of the solver
- mhd is a boolean specifying whether magnetic fields are present
- internal_energy is a boolean specifying if the internal energy flux must be computed.

This weird indirection only currently exists to accomodate charm++'s pup functionality. Once Enzo-E transitions to its custom restart functionallity, we'll remove this indirection.

An instance of EnzoRiemann specifies the expected non-passive keys (and key-order) that the flux_map argument should have when passed to its solve method (these keys correspond to integration quantities).

```
const std::vector<std::string> integration_quantity_keys() const;
```

The following method specifies the expected non-passive keys (and key-order) that the priml_map and primr_map arguments should have when passed an EnzoRiemann's solve method (these keys correspond to primitive quantities).

```
const std::vector<std::string> primitive_quantity_keys() const;
```

The main interface function of EnzoRiemann is:

```
void solve(const EnzoEFltArrayMap &prim_map_l,
           const EnzoEFltArrayMap &prim_map_r,
           EnzoEFltArrayMap &flux_map, int dim,
           int stale_depth, const str_vec_t &passive_list,
           const CelloView<enzo_float,3> * const interface_velocity) const;
```

In this function, the prim_map_l and prim_map_r arguments are references to the EnzoEFltArrayMap objects holding the arrays of reconstructed left/right primitive quantities. The flux_map argument holds the face-centered arrays where the computed fluxes for each integration quantity are written. dim indicates the dimension along which the flux should be computed (0,1,2 corresponds to x,y,z). interface_velocity is an optional argument used to specify a pointer to an array that can be used to store interface velocity values computed by the Riemann Solver (this is primarily used for computing internal energy source terms when the dual energy formalism is in use).

Some additional notes:

- The first EnzoRiemann::primitive_quantity_keys().size() keys of prim_map_l and prim_map_r should match the values and order of EnzoRiemann::primitive_quantity_keys().
- Likewise, the first EnzoRiemann::integration_quantity_keys().size() keys of flux_map should match the values and order of EnzoRiemann::integration_quantity_keys().
- prim_map_l, prim_map_r, and flux_map should also each contain keys for each of the passive scalars in passive_list (the order of these is not currently enforced).

- All of the arrays in `prim_map_l`, `prim_map_r`, and `flux_map` should have the same shape. If `interface_velocity` is specified, it should also have that shape.
- Calling the `contiguous_arrays()` instance method for `prim_map_l`, `prim_map_r`, and `flux_map` must return `true` in each case (in other words, each array in the array maps should be stored in nearly-contiguous 4D arrays).

Implementation Notes: Kernels

At the time of writing, the calculations specific to each Riemann Solver are implemented as *kernels* (inspired by *Kokkos*). More precise requiements are detailed in *Kernel Requirements*, but in short each kernel:

- defines a `operator()(int iz, int iy, int ix) const` method for computing the flux at the specified cell-interface
- specifies a specialization of the template class `EnzoRiemannLUT<InputLUT>`. This both indicates the expected actively advected integration (and primitive) quantities **AND** acts as a compile-time lookup table (that maps quantity names to unique array indices). See *EnzoRiemannLUT* for more details.
- Specifies the expected type of the equation of state by specifying the expected corresponding EOS class. Follow *this link* for more details about EOS Struct objects.
- are configured by an instance of `KernelConfig<EOSStructT>` (see *KernelConfig<EOSStructT>* for more details).

The `EnzoRiemannImpl<Kernel>` class template is used to wrap each kernel. This class template subclasses the `EnzoRiemann` abstract base class and is used to implement the interface described above, in *Usage Notes*, for each kernel (i.e. in other words, we use it to implement "type erasure"). In more detail, `EnzoRiemannImpl<Kernel>::integration_quantity_keys()` and `EnzoRiemannImpl<Kernel>::primitive_quantity_keys()` methods specify the fields (and field ordering) required by a given kernel.

The steps of `EnzoRiemannImpl<Kernel>::solve` are fairly straight-forward:

1. An instance of `KernelConfig<Kernel::EOSStructT>` is created.
2. The `Kernel` is constructed and executed at each cell-interface
3. The fluxes for passively advected scalars are computed (this step is completely independent of the choice of `Kernel`).

EnzoRiemannLUT

As described above in the *Overview*, we sought to avoid the common approach of hydro codes that map actively advected quantities indices with macros or globally defined unscoped enums. The `EnzoRiemannLUT<InputLUT>` template class basically serves as a compromise between this traditional approach approach and using a hash table (which introduce unacceptable overhead) for organizing quantities in the main loop of `EnzoRiemannImpl<Kernel>`. Alternatively it can be thought of as a scoped version of the traditional approach.

This is a template class that provides the following features at compile time:

- a lookup table (LUT) that maps the names of components of a subset of the actively advected integration quantities defined in `FIELD_TABLE` to unique, contiguous indices.
- the number of integration quantity components included in the table
- a way to iterate over just the conserved or specific integration quantities values that are stored in an array using these mapping

- a way to query which of the actively advected integration quantities in FIELD_TABLE are not included in the LUT

These feature are provided via the definition of publicly accessible integer constants in every specialization of the template class. All specializations have:

- a constant called `num_entries` equal to the number of integration quantity components included in the lookup table
- a constant called `specific_start` equal to the number of components of conserved integration quantities included in the lookup table
- `qkey` constants, which include constants named for the components of ALL actively advected integration quantities in FIELD_TABLE. A constant associated with a SCALAR quantity, `{qname}`, is simply called `{qname}` while constants associated with a vector quantity `{qname}` are called `{qname}_i`, `{qname}_j`, and `{qname}_k`.

The `qkey` constants serve as both the keys of the lookup table and a way to check whether a component of an actively advected quantity is included in the table. Their values are satisfy the following conditions:

- All constants named for values corresponding to quantities NOT included in the lookup table have values of `-1`
- All constants named for conserved integration quantities have unique integer values in the internal `[0, specific_start)`
- All constants named for specific integration quantities have unique integer values in the interval `[specific_start, num_entries)`
- Constants must be defined for all 3 components (or none of the components) of a vector quantity (e.g. velocity or magnetic fields). Additionally, the `k`th component of a vector quantity is expected to have a value that is 1 larger than that of the `j`th component and 2 larger than the `i`th component.

The lookup table is always expected to include density and the 3 velocity components.

This template class also provides a handful of helpful static methods to programmatically probe the table's contents at runtime and validate that the above requirements are specified.

For the sake of providing some concrete examples about how the code works, let's assume that we have a class `MyIntegLUT` that is defined as:

```
struct MyIntegLUT {
  enum vals { density=0, velocity_i, velocity_j, velocity_k,
              total_energy, num_entries, specific_start = 1};
};
```

The template specialization `EnzoRiemannLUT<MyIntegLUT>` assumes that all undefined `qkey` constants omitted from `MyIntegLUT` are not included in the lookup table and will define them within the template specialization to have values of `-1`.

To access the index associated with density or the jth component of velocity, one would evaluate:

```
int density_index = EnzoRiemannLUT<MyIntegLUT>::density; //=0
int vj_index = EnzoRiemannLUT<MyIntegLUT>::velocity_j;   //=2
```

Additionally, the value of `EnzoRiemannLUT<MyIntegLUT>::bfield_k` would be `-1`.

It makes more sense to talk about the use of this template class when we have a companion array. For convenience, the alias template `lutarray<LUT>` type is defined. The type, `lutarray<EnzoRiemannLUT<InputLUT>>` is an alias of the type `std::array<enzo_float, EnzoRiemannLUT<InputLUT>::num_entries>;`.

As an example, imagine that the total kinetic energy density needs to be computed at a single location from an values stored in an array, `integ`, of type `lutarray<EnzoRiemannLUT<MyIntegLUT>>`:

```
using LUT = EnzoRiemannLUT<MyIntegLUT>;
enzo_float v2 = (integ[LUT::velocity_i] * integ[LUT::velocity_i] +
                 integ[LUT::velocity_j] * integ[LUT::velocity_j] +
                 integ[LUT::velocity_k] * integ[LUT::velocity_k]);
enzo_float kinetic = 0.5 * integ[LUT::density] * v2;
```

`EnzoRiemannLUT<InputLUT>`, makes it very easy to write generic code that can be reused for multiple different lookup table by using by passing its concrete specializations as a template argument to other template functions/classes. Consider the case where a single template function is desired to compute the total non-thermal energy density at a single location for an arbitrary lookup table:

```
template <class LUT>
enzo_float calc_nonthermal_edens(lutarray<LUT> prim)
{
  enzo_float v2 = (prim[LUT::velocity_i] * prim[LUT::velocity_i] +
  prim[LUT::velocity_j] * prim[LUT::velocity_j] +
  prim[LUT::velocity_k] * prim[LUT::velocity_k]);

  enzo_float bi = (LUT::bfield_i >= 0) ? prim[LUT::bfield_i] : 0;
  enzo_float bj = (LUT::bfield_j >= 0) ? prim[LUT::bfield_j] : 0;
  enzo_float bk = (LUT::bfield_k >= 0) ? prim[LUT::bfield_k] : 0;
  enzo_float b2 = bi*bi + bj*bj + bk*bk;

  return 0.5(v2*prim[LUT::density] + b2);
}
```

`KernelConfig<EOSStructT>`

The `KernelConfig` template class simply groups the configuration parameters for kernel (the alternative would be to require that the members of `KernelConfig` are listed as members for every class implementing a kernel).

The relationship between this class and a kernel is modelled after the relationship between captured values and a lambda function. We made this class as lightweight as possible to encourage the compiler to make similar optimizations. The declaration of `KernelConfig` is reproduced below:

```
template<typename EOSStructT>
struct KernelConfig{
  /// @class    KernelConfig
  /// @ingroup  Enzo
  /// @brief    [\ref Enzo] Stores the configuration options, input arrays, and
  /// output arrays used by a Riemann Solver Kernel.
  ///
  /// To help facilitate optimizations, we avoid introducing methods and
  /// declare all member variables to be ``const``
  ///
  /// Indices passed to the trailing 3 axes of ANY array (3D & 4D) held by this
  /// struct are used to specify spatial position of values.
  /// - In practice, it's sufficient to understand that a given set of spatial
  ///   indices, `(iz,iy,ix)`, represent the same spatial location in each array
  /// - For completeness, we note that integer indices map to the center of the
  ///   x-faces, y-faces, & z-faces for `dim` values of 0, 1, & 2, respectively.
  ///   The extents of these trailing spatial axes `(LENz, LENy, LENx)` are:
```

(continues on next page)

(continued from previous page)

```
///   - when `dim=0`: `LENz = mz`,      `LENy = my`,      `LENx = mx-1`
///   - when `dim=1`: `LENz = mz`,      `LENy = my - 1`, `LENx = mx`
///   - when `dim=2`: `LENz = mz - 1`, `LENy = my`,      `LENx = mx`
///   where `(mz,my,mx)` gives the number of cells along each axis of a
///   cell-centered field (including the ghost zone)

/// @name ConfigOptions
/**@{*/
/// dimension along which the flux is being computed (0 -> x, 1 -> y, 2 -> z)
const int dim;
/// EOS Struct Object
const EOSStructT eos;
/**@}*/

/// @name PrimaryArrays
/// `prim_arr_l` and `prim_arr_r` provide a kernel's primary input data. The
/// primary outputs get written to `flux_arr`. All 3 arrays share the shape:
/// `(LUT::num_entries, LENz, LENy, LENx)`
///
/// The index passed to axis 0 correspond to different quantities (a kernel's
/// `LUT` specifies the precise mapping between quantities & index values).
/// There are 2 relevant points to be mindful of:
/// 1. for these arrays, when accessing components of vector quantities
///    (e.g. `LUT::velocity_i`, `LUT::velocity_j`, or `LUT::bfield_k`),
///    the i, j, & k components ALWAYS map to the x, y, and z components.
/// 2. For performance purposes, the length of the arrays along axis 0 is
///    allowed to exceed `LUT::num_entries`. But kernels should never
///    access these indices (they don't map to quantities in LUT)
/// For completeness, we provide the following examples:
/// - `flux_arr(LUT::density,...)` & `flux_arr(LUT::velocity_k,...)`
///   are where the computed density & momentum_z fluxes should be stored
/// - `prim_arr_l(LUT::density,...)` & `prim_arr_l(LUT::velocity_i,...)` hold
///   the reconstructed density & velocity_x values on the left side of a
///   cell interface
/**@{*/
/// array to store the computed fluxes
const CelloView<enzo_float,4> flux_arr;
/// array of primitives reconstructed on the left side of cell interfaces
const CelloView<const enzo_float,4> prim_arr_l;
/// array of primitives reconstructed on the right side of cell interfaces
const CelloView<const enzo_float,4> prim_arr_r;
/**@}*/

/// @name DualEnergyArrays
/// These arrays store outputs used in the dual-energy formalism and have the
/// shape `(LENz, LENy, LENx)`.
///
/// For any Dual-Energy compatible EOS, kernels should ALWAYS fill these
/// arrays with the relevant values. If the dual energy formalism isn't used
/// outside of the Riemann Solver, these are initialized with scratch-space.
/// This is done to avoid unnecessary branching and code generation.
/**@{*/
```

(continues on next page)

(continued from previous page)

```
  /// array to store the computed flux for the specific internal energy
  ///
  /// @note
  /// When `flux_arr.shape(0) > LUT::num_entries`, this can technically alias
  /// one of `flux_arr`'s subarrays without a corresponding `LUT` entry. In
  /// practice, kernels will NEVER be affected by this.
  const CelloView<enzo_float,3> internal_energy_flux_arr;
  /// array to store the velocity (along `dim`) computed at the cell interface
  const CelloView<enzo_float,3> velocity_i_bar_arr;
  /**@}*/
};
```

Note: In the near-future, the above snippet will be replaced with nicer looking documentation that will be auto-generated using doxygen and breathe

More details will be given below about internal energy calculations.

Kernel Requirements

The basic skeleton for defining a kernel is defined below. For concreteness, we've assumed that this kernel uses the `MHDLUT` and an ideal EOS (but those could easily be changed).

```
struct MyKernel
{

public: // typedefs
  using LUT = EnzoRiemannLUT<MHDLUT>;
  using EOSStructT = EOSStructIdeal;

public: // fields
  const KernelConfig<EOSStructT> config;

public: // methods
  FORCE_INLINE void operator()(const int iz,
                               const int iy,
                               const int ix) const noexcept
  {
    // compute the fluxes at (iz, iy, ix) & store result in config.flux_arr
    //
    // For concreteness:
    //   - the left and right reconstructed density values (for the same
    //     cell-interface) can be retrieved using:
    //       config.prim_arr_l(LUT::density, iz, iy, ix)
    //       config.prim_arr_r(LUT::density, iz, iy, ix)
    //   - we want to write computed density flux to:
    //       config.flux_arr(LUT::density, iz, iy, ix)
    //
    // It may be helpful to point out that:
```

(continues on next page)

(continued from previous page)

```
    //   when (config.dim == 0) -> this should computes on x-faces
    //   when (config.dim == 1) -> this should computes on y-faces
    //   when (config.dim == 2) -> this should computes on z-faces
    //
    // For more information, including how exactly indices map to spatial
    // locations, check the docstrings for KernelConfig. In practice, it's
    // sufficient to understand that (iz,iy,ix) refers to the same spatial
    // location in ALL of the arrays.
    //
    // assuming it makes sense for the EOS (it's non-barotropic), this should
    // also compute values needed for the dual energy formalism and store
    // results in config.internal_energy_flux_arr(iz,iy,ix) &
    // config.velocity_i_bar_arr(iz,iy,ix)

  }
};
```

There are a couple of things to note:

- As explained in *KernelConfig<EOSStructT>*, when you access vector quantities from `config.prim_arr_l`, `config.prim_arr_r`, or `config.flux_arr` using `LUT` (e.g. `LUT::velocity_i`, `LUT::velocity_j`, `LUT::bfield_k`), the `i`, `j`, & `k` components **always** map to the `x`, `y`, & `z` components, respectively.
  - To be concrete: `config.flux_arr(LUT::velocity_j, iz, iy, ix)` always refers to the flux of the `y` velocity component.
  - To try preserve symmetry, we often use `config.dim` to remap the `i`, `j`, and `k` components (like what is done in Athena++). The current convention is to include the following code block at the start of `operator()`:

    ```
    const int external_velocity_i = config.dim + LUT::velocity_i;
    const int external_velocity_j = ((config.dim+1)%3) + LUT::velocity_i;
    const int external_velocity_k = ((config.dim+2)%3) + LUT::velocity_i;
    const int external_bfield_i = config.dim + LUT::bfield_i;
    const int external_bfield_j = ((config.dim+1)%3) + LUT::bfield_i;
    const int external_bfield_k = ((config.dim+2)%3) + LUT::bfield_i;
    ```

    and to use `external_velocity_i`, `external_velocity_j`, `external_velocity_k` instead of `LUT::velocity_i`, `LUT::velocity_j`, `LUT::velocity_k` when accessing values from `prim_arr_l`, `prim_arr_r`, and `flux_arr`.
  - We have given some thought to trying to abstract this vector permutation, but it unfortunately remains unclear how to do this in a way that preserves performance across different compilers, without requiring template specializations of `operator()` for computing fluxes along different dimensions.
- It's also possible to make `config` a private field or to introduce additional fields. However, if you do either of those things, you will need to define a constructor that accepts `config` as a single argument.
- `FORCE_INLINE` is a macro that uses compiler-specific extensions to force inlining of functions. Please use this sparingly outside, only in cases where you see a noticable speedup (inlining everything can actually slow code down from inflating the binary's size).

Dual Energy Formalism Treatment

The dual-energy formalism requires that a Riemann Solver computes fluxes for the internal energy, and a velocity component at the cell-interfaces.

Computing these quantities doesn't change the Riemann Solver calculation (or the required reconstructed primitives), it just involves an extra calculation. To avoid unnecessary template code generation or introducing branching:

- `EnzoRiemannLUT` generally does not include `internal_energy` as an entry (there are other reasons why this is convenient)
- Kernels **always** compute the necessary quantities for the dual-energy formalism (assuming the EOS is compatible with the dual-energy formalism)

Instead, `EnzoRiemannImpl<Kernel>` manages the dual-energy configuration:

- `EnzoRiemannImpl<Kernel>::solve` will **ALWAYS** make sure that `config.internal_energy_flux_arr` and `config.velocity_i_bar_arr` are valid arrays. If arrays aren't provided (i.e. the dual-energy formalism is not in use), scratch data will be used for this explicit purpose.
- `EnzoRiemannImpl<Kernel>` is responsible for including `"internal_energy"` in the output of `integration_quantity_keys()`, based on how it's configured.

Adding new quantites

To add support for new actively advected integration cell-centered quantities (e.g. cosmic ray energy/flux), the table of cell-centered quantities (`FIELD_TABLE`) must be updated. See *Centered Field Registry* for more details. To add support for computing fluxes for such quantities, modifications must be made to either `EnzoRiemannImpl` or the `Kernel` of an existing solver. Alternatively, for certain quantities, a brand new solver may need to be introduced.

When adding a new integration vector quantity, you also need to add a few lines to the main for-loop of `EnzoRiemannImpl` for copying values to `wl`/`wr` and from `fluxes` (The existing code doing this for the velocity and magnetic fields should be used as a guide).

Adding new solvers

New Riemann Solvers can currently be added to the infrastructure by either subclassing `EnzoRiemann` or defining a new specialization of `EnzoRiemannImpl<Kernel>`. In either case, the `EnzoRiemann::construct_riemann` factory method must be modified to return the new solver and *riemann solvers* should be updated.

The additional steps for implementing a new Riemann solver by speciallizing `EnzoRiemannImpl<Kernel>` are as follows:

1. Define a new `Kernel` class (e.g. `HLLDKernel`)
2. *(optional)* define an alias name for the specialization of `EnzoRiemannImpl` that uses the new `Kernel` class (e.g. `using EnzoRiemannHLLD = EnzoRiemannImpl<HLLDKernel>;`).

Note: Due to the template-heavy nature of our implementation, the Riemann Solvers been separated from the rest of the Enzo-layer into their own sub-library.

This choice was made because the convention in the Enzo-layer (at least before we made this separation) is to effectively include every header file in every source file. Since template code usually needs to be written in header files, changes to the Riemann Solver algorithms used to frequently trigger rebuilds of the entire Enzo-layer. This separation has significantly sped up incremental rebuilds during the development workflow for the Riemann Solvers.

4.8.7 Updating integration quantities

The `EnzoIntegrationQuanUpdate` class has been provided to encapsulate the operation of updating integration quantities after a (partial) time-step. The operation was factored out of the `EnzoMethodMHDVlct` class since it appear in all Godunov solvers.

The constructor for `EnzoIntegrationQuanUpdate` has the following signature:

```
EnzoIntegrationQuanUpdate(std::vector<std::string> integration_quantity_keys,
                          bool skip_B_update)
```

The function requires that we:

- register the keys of the integration quantities (with `integration_quantity_keys`)
- indicate whether the update to the magnetic field should be skipped.

The integration quantity keys should match the names specified in `FIELD_TABLE`; see *Centered Field Registry* for more details. The update to the magnetic field should be skipped when Constrained Transport is in use (since the magnetic field update is handled separately). If the magnetic field is not specified as an integration quantity, then the value specified for `skip_B_update` is unimportant

The following method is used to compute the change in (the conserved form of) the integration and passively advected quantites due to the flux divergence along dimension `dim` over the (partial) timestep `dt`. The arrays in `dUcons_map` are used to accumulate the total changes in these quantities. `passive_list` lists the names (keys) of the passively advected scalars.

```
void accumulate_flux_component
  (int dim, double dt, enzo_float cell_width,
  const EnzoEFltArrayMap &flux_map,
   EnzoEFltArrayMap &dUcons_map, int stale_depth,
   const std::vector<std::string> &passive_list) const;
```

The method used to clear the values of the arrays used for accumulation is provided below. This sanitization should be performed before starting to accumulate flux divergence or source terms. The `passive_list` argument is used in the same way as the previous function.

```
void clear_dUcons_group(EnzoEFltArrayMap &dUcons_map, enzo_float value,
                        const std::vector<std::string> &passive_list) const;
```

The method used to actually add the accumulated change in the integration (specified in `dUcons_map`) to the values of the integration quantities from the start of the timestep (specificed by `initial_integration_map`) has the following signature:

```
void update_quantities
  (EnzoEFltArrayMap &initial_integration_map,
   const EnzoEFltArrayMap &dUcons_map,
   EnzoEFltArrayMap &out_integration_map,
   EnzoEFltArrayMap &out_conserved_passive_scalar,
   int stale_depth,
   const std::vector<std::string> &passive_list) const;
```

The fields included in `dUcons_map` should include contributions from both the flux divergence AND source terms. The results for the actively advected quanties are stored in `out_integration_map` and the results for the passively advected scalars are stored in conserved form in the arrays held by `out_conserved_passive_scalar` (note that the initial values of the passive scalars specified in `initial_integration_map` are in specific form).

4.8.8 Magnetic Field Integration

Subclasses of the abstract base class, `EnzoBfieldMethod` are used to implement magnetic field integration-related operations. While operations like reconstruction and flux calculations of relevant quantities are expected to be carried out with `EnzoReconstructor` and `EnzoRiemann`, all other magnetic field integration-related operations should be encapsulated by `EnzoBfieldMethod`.

Currently, the only subclass is `EnzoBfieldMethodCT`, which implements operations related to Constrained Transport. Other subclasses could be implemented in the future that encapsulate other integration methods (e.g. divergence cleaning).

From the perspective of an integrator that employs `EnzoBfieldMethod`, the primary result of each operation is to modify the values cell-centered/reconstructed quantities, since that's all the integrator directly needs to know about. In reality, side effects performed by these operations can be equally as important. For example, `EnzoBfieldMethodCT` implicitly needs to update face-centered magnetic field values (given that the face-centered values serve as the primary representation, and the cell-centered values are derived directly from them).

To accomplish these goals, `EnzoBfieldMethod`, basically implements a state machine. It basically provides 3 classes of methods: (i) state machine-methods, (ii) physics methods, and (iii) descriptor methods.

State Machine Methods

When the `EnzoBfieldMethod` is first constructed, it has an uninitialized state. During construction the number of partial timesteps (`num_partial_timesteps`) involved per cycle must be specified.

At the beginning of an integration cycle (when an `EnzoBfieldMethod` object is unitialized), the cello `Block` that is going to be integrated block that must be specified using the following method.

```
void register_target_block(Block *block) noexcept;
```

This method will correctly set the internal state and will invoke the virtual `register_target_block_` method, which is used by subclasses to preload relevant data from `block` and for the delayed initialization of scratch arrays (since the shapes may not be known at construction).

Once a target block has been registerred, the `EnzoBfieldMethod` object is now ready to perform integration-related operations for the first partial timestep (the physics methods can now be called). The following method is used to increment the partial timestep:

```
void increment_partial_timestep() noexcept;
```

The target block is unregistered once this method to has been called `num_partial_timesteps` times. Any calls to `register_target_block` while a block is still registered will currently cause an error.

There are couple of things to keep in mind:

- Any calls to physics methods or other state machine or other when no target block is registered are not allowed.
- It's EXTREMELY important that `increment_partial_timestep` is always invoked `num_partial_timesteps` after a target block is registered and before there is chance for blocks to migrate between nodes. In other words, the a target block should always be registerred and unregistered during a single call to the cello `Method` object that represents the integrator.

Physics Methods

These methods are actually used to perform the relevant magnetic field integration operations. Each method is a pure virtual method that must be implemented by a subclass (even if the method immediately returns). These methods were all written and named based on the operations of Constrained Transport (CT). In the future, additional methods may need to be introduced to facillitate the implementation of other magnetic field integration schemes.

These methods are listed below with brief description. For more details, please see the docstring. The methods are expected to generally be called in the general order that they are listed. While this isn't currently enforced, incorrect results may arise if they aren't called in the proper order.

In the context of CT, the following method is used to overwrite the reconstructed value magnetic field component that corresponds to the axis of reconstruction with the (internally tracked) face-centered value.

```
void correct_reconstructed_bfield(EnzoEFltArrayMap &l_map,
                                  EnzoEFltArrayMap &r_map, int dim,
                                  int stale_depth) noexcept;
```

The following method is used by `EnzoBfieldMethodCT` to take note of the upwind direction after computing the Riemann Fluxes along a dimension `dim`.

```
void identify_upwind(const EnzoEFltArrayMap &flux_map, int dim,
                     int stale_depth) noexcept;
```

Finally, the following method is used to actually update the cell-centered magnetic field values.

```
void update_all_bfield_components
  (EnzoEFltArrayMap &cur_prim_map, const EnzoEFltArrayMap &xflux_map,
   const EnzoEFltArrayMap &yflux_map, const EnzoEFltArrayMap &zflux_map,
   EnzoEFltArrayMap &out_centered_bfield_map, enzo_float dt,
   int stale_depth) noexcept;
```

In `EnzoBfieldMethodCT` this will also update the face-centered magnetic field values (it assumes that `identify_upwind` was called once for each dimension and uses the stored data). When using this alongside `EnzoIntegrationQuanUpdate`, care needs to be taken about the order in which this method is called relative to `EnzoIntegrationQuanUpdate::update_quantities` that accounts for the time when floors are applied to the total energy.

Descriptor Methods

These are virtual methods that can be invoked at any time after the `EnzoBfieldMethod` object has been constructed. These are used to describe requirements of the given magnetic field integration method.

Currently, only one such method exists:

```
void check_required_fields() const noexcept;
```

These may change in the future.

How to extend

Implementing a new method for magnetic field integration is fairly straight-forward. Basically all you have to do is implement a subclass of `EnzoBfieldMethod`. In addition to providing implementations for each each physics and descriptor method, the subclass also needs to implement:

```
void register_target_block_(Block *target_block,
                            bool first_initialization) noexcept;
```

As mentioned earlier, this method is called by `register_target_block` while registering a new target block. In this call the subclass should preload any data it will need from the `target_block`. The `first_initialization` argument indicate whether this is the first time a `target_block` is being registered after the instance has been constructed (this includes the first time following deserialization after a restart). It can be used to help with lazy intialization of scratch space.

Once a second concrete subclass of `EnzoBfieldMethod` *is provided, it may be worthwhile to introduce a factory method.*

4.9 Fluid Properties and EOS

[This page is under development]

Fluid properties are centralized in the `EnzoPhysicsFluidProps`. The user documentation is described *here*.

As an aside, this section uses terms like "integration" and "primitive" quantities or "stale depth" that are defined in the *Hydro/MHD Infrastructure section*.

4.9.1 Fluid props

The primary goal of the `EnzoPhysicsFluidProps` class is to hold general data about fluid properties (that may affect multiple parts of the codebase) in a central location and provide methods for accessing this information. An important guiding principle is that the class is immutable: once it's initialized it should **not** change.

With that said, it does provide some other miscellaneous useful methods.

Primary methods

All of the primary methods return references to immutable objects that can be used to query information related to the fluid properties.

const EnzoDualEnergyConfig &`EnzoPhysicsFluidProps::`**`dual_energy_config`**`()` const noexcept

Access a constant reference to the contained dual energy configuration object.

const EnzoFluidFloorConfig &`EnzoPhysicsFluidProps::`**`fluid_floor_config`**`()` const noexcept

Access a constant reference of the object that encodes the fluid floor properties.

const EnzoEOSVariant &`EnzoPhysicsFluidProps::`**`eos_variant`**`()` const noexcept

Access a constant reference to the contained `EnzoEOSVariant` class. It holds an object that represents the caloric EOS used by the simulation. See the *EOS* section for more details about the design of the EOS functionality (and how to use it).

Misc useful methods

The functionality described in this subsection are only defined as methods of `EnzoPhysicsFluidProps` for a lack of better place to define them. In the future, it might make sense to move them around.

It's important that none of these functions actually mutate the contents of `EnzoPhysicsFluidProps`. As in the Hydro/MHD Infrastructure section, many of these function operations act on the contents of `EnzoEFltArrayMap` rather than directly on `Block` objects for additional flexibility.

void `EnzoPhysicsFluidProps::primitive_from_integration`(const EnzoEFltArrayMap &integration_map, EnzoEFltArrayMap &primitive_map, int stale_depth, const std::vector<std::string> &passive_list, bool ignore_grackle = false) const

This method is responsible for computing the primitive quantities (to be held in `primitive_map`) from the integration quantities (stored in `integration_map`). Non-passive scalar quantities appearing in both `integration_map` and `primitive_map` are simply deepcopied and passive scalar quantities are converted from conserved-form to specific form. If `EnzoPhysicsFluidProps` holds a non-barotropic EOS, this method also computes pressure (by calling `EnzoEquationOfState::pressure_from_integration()`).

void `EnzoPhysicsFluidProps::pressure_from_integration`(const EnzoEFltArrayMap &integration_map, const CelloArray<enzo_float, 3> &pressure, int stale_depth, bool ignore_grackle = false) const

This method computes the pressure from the integration quantities (stored in `integration_map`) and stores the result in `pressure`. This wraps the `EnzoComputePressure` object whose default behavior is to use the Grackle-supplied routine for computing pressure when the simulation is configured to use `EnzoMethodGrackle`. The `ignore_grackle` parameter can be used to avoid using that routine (the parameter is meaningless if the Grackle routine would not otherwise get used). This parameter's primary purpose is to provide the option to suppress the effects of molecular hydrogen on the adiabatic index (when Grackle is configured with `primordial_chemistry > 1`).

void `EnzoPhysicsFluidProps::apply_floor_to_energy_and_sync`(EnzoEFltArrayMap &integration_map, const int stale_depth) const

This method applies the pressure floor to the `"total_energy"` array specified in `integration_map`. If using the dual-energy formalism the floor is also applied to the `"internal_energy"` (also specified in `integration_map`) and synchronizes the `"internal_energy"` with the `"total_energy"`.

If `EnzoPhysicsFluidProps` holds a barotropic EOS, this method should do nothing.

Note:

In the future, it may make sense to directly pass the pressure floor.

4.9.2 EOS

Overviews

This section documents how different caloric/isothermal equations of state are supported in Enzo-E. For reference, these govern the relationship between density, pressure, and internal (or thermal) energy.

Currently, Enzo-E supports relatively few equations of state. Over time, Hydro codes have a tendency to add support for multiple different types of equations of state. It's therefore important to have a solid strategy in place early on to

support multiple equations of state. Unlike other simulation codes (e.g., Athena++) that partially configure physics-features (like the choice of EOS) at compile-time, Enzo-E tends to takes the approach of compiling all physics at once. Thus, Enzo-E needs to support the selection of the EOS at runtime.

The obvious strategy (and the original approach that we took) is to use inheritance with virtual methods. However, virtual methods are not well-suited for being used within compute kernels (i.e., in the body of a for-loop). Issues arise because: (i) there is overhead associated with virtual method calls and (ii) there are problems with invoking the virtual methods on GPUs. While we can get around this to some degree by designing virtual methods to be called outside of the for-loop, there will always be cases where EOS details must be known within a for-loop (e.g., in a Riemann Solver).

For this reason, we choose a different approach for achieving polymorphism, which involves using a tagged union. The idea is that we represent each type of EOS as a stand-alone class and the type-safe union holds an instance of one of those classes (unlike a typical union, the type-safe union explicitly prohibits unsafe access of any union member other than the one that is currently stored) This is an approach popular in functional programming, in modern languages (e.g., Rust), and that has received support in C++17. While this approach still incurs some overhead analogous to that of a virtual function, it provides much greater flexibility in choosing where/when we pay this overhead. For example, we can choose to pay this cost just before a for-loop.

High-Level Design

As mentioned above, `EnzoEOSVariant`, is simply a container that holds an EOS object. An EOS object is an instance of one of a few (unrelated, standalone) classes like `EnzoEOSIdeal` or `EnzoEOSIsothermal`. that acts as a typesafe union which always holds an instance of one of these classes.

When the `EnzoPhysicsFluidProps` physics object is constructed, it creates an instance of the EOS class and holds it internally within an instance of `EnzoEOSVariant`. Throughout the remainder of the simulation, the `EnzoPhysicsFluidProps` physics object prevents mutation of the `EnzoEOSVariant` instance that it owns (and consequently to the contained EOS object). Users can access this object with *`EnzoPhysicsFluidProps::eos_variant()`*.

`EnzoEOSVariant` implements a type-safe union that is modelled after the `std::variant` template class introduced in C++17. This class was designed in a way that the internals can easily be replaced with `std::variant` when Enzo-E eventually transitions to using C++17.

Note: The choice to model `EnzoEOSVariant` after `std::variant` leads to slightly more complicated code than is strictly necessary.

EOS Classes

These classes are supposed to be lightweight struct/classes that encapsulate an equation of state. It's also important that these objects are cheap to copy. They are all entirely defined in header files to facilitate inlining.

We currently expect an EOS class, `EOSClass`, to provide the following methods:

- `constexpr static const char* EOSClass::name() noexcept` This returns the name of the EOS class (it should match the name a user would specify in a parameter file)
- `constexpr static bool EOSClass::is_barotropic() noexcept` This should return true when the pressure field is just a function of density (e.g., in an isothermal gas).
- `std::string EOSClass::debug_string() const noexcept` This should return a string (for debugging purposes) that represents the internal state of the EOS object.

Other methods supported by an EOS may include calculation of sound speed, fast magnetosonic speed, internal energy, etc. Essentially all (non-static) methods of an EOS-object are declared as `const` (i.e. there's no reason for them to mutate internal state).

One of the perks of using tagged unions is that different types of EOS objects don't NEED to implement the same methods. For example, it doesn't make much sense for an isothermal eos to support methods that compute the thermal energy.

Currently, two EOS classes exist: `EnzoEOSIdeal` and `EnzoEOSIsothermal`. The `EnzoEOSIdeal` class implements methods that, given the density and pressure, will compute the following quantities:

- specific internal energy
- internal energy density
- sound speed
- fast magnetosonic speed (this requires magnetic field values to also be specified)

At the time of writing this section, `EnzoEOSIsothermal` is mostly just a placeholder that is used alongside the PPML method (it's not actually used within the PPML method, but it indicates the choice of EOS when other methods are used alongside PPML).

Note: At this time, temperature-related stuff is handled entirely outside of the EOS. The rationale for this choice is that this functionality is somewhat unrelated to hydro-solvers (but this is something that can be revisited in the future).

Grackle has also **NOT** been integrated with the EOS solver at this time. (this may need to be revisited in the future).

Note: Currently, to ensure that they are lightweight, all of the EOS classes are "aggregates", which means that they are classes with:

1. no user-provided or explicit constructors
2. no private or protected data members (attributes)
3. no default member initializers (this can be relaxed in C++ 14)
4. no base classes or virtual methods

Invariants that might be enforced in a constuctor are instead enforced by a factory method (e.g. `EnzoEOSIdealt::construct()`).

In reality, it would probably simplify the code quite a bit, without sacrificing much/any performance, if we just required that the class was trivially copyable (that's possible without it being an aggregate)

Using `EnzoEOSVariant` (accessing stored EOS object)

The `EnzoEOSVariant` class is a type-safe union that ALWAYS holds an instance of one of the types representing an EOS. EOS objects instances are lightweight structs.

When discussing how to use `EnzoEOSVariant`, it is most instructive to describe different operations with examples (rather than providing a detailed API).

Retrieving the EOS Object

Let's first imagine we want to write some code that assumes that Enzo-E is configured with an ideal EOS and requires knowledge of the adiabatic index, `gamma`. If Enzo-E is configured with a different type of EOS the codebase should terminate with an error.

The following snippet shows a verbose approach for accomplishing this:

```
void my_func(/* args... */) {

  // 1. retrieve pointer to the PE's EnzoPhysicsFluidProps instance
  const EnzoPhysicsFluidProps* fluid_props = enzo::fluid_props();

  // 2. fetch a const reference to the EnzoEOSVariant instance held within
  //    the object pointed to by fluid_props
  const EnzoEOSVariant& eos_variant = fluid_props->eos_variant();

  // 3. fetch a const reference to the eos within eos_variant, while
  //    enforcing the assumption that it's an EnzoEOSIdeal instance
  const EnzoEOSIdeal& eos = eos_variant.get<EnzoEOSIdeal>();

  // fetch the value of gamma
  enzo_float gamma = eos.get_gamma();

  // do work with gamma...
}
```

Now, let's break this down in slightly more detail.

1. `enzo::fluid_props()` returns a pointer to the instance of the `EnzoPhysicsFluidProps` that is configured for the Processing Element (PE). This pointer can't be a `nullptr` (if it is, the function will abort with an error).
2. fetch a const reference to the `EnzoEOSVariant` instance held within the object pointed to by `fluid_props`
3. fetch a const reference to the eos within `eos_variant` if it currently holds an `EnzoEOSIdeal`. In other cases, the program aborts with an error.

We can write a much more concise form of the above function:

```
void my_func(/* args... */) {

  // the program aborts with an error if Enzo-E was not configured with an
  // ideal eos
  const EnzoEOSIdeal& eos = enzo::fluid_props()->eos_variant().get<EnzoEOSIdeal>();
  enzo_float gamma = eos.get_gamma();
  // do work with gamma...
}
```

In both of these snippets we make use of the method:

template<typename **T**>
const *T* &`EnzoEOSVariant::`**`get`**`()` const

Accessor method that returns a reference to the contained EOS object if `this` currently holds the EOS object of type `T`. Otherwise, the program aborts with an error message. A non-`const`-qualified version of this method also exists.

This is a counterpart of the `std::get` template function.

Retrieving the EOS Object with Detailed Error Message

Now let's consider a variation on the last case. In this situation let's imagine that we want to write a more detailed error message in the case where it is executed and Enzo-E is not configured with an ideal EOS:

```
void my_func(/* args... */) {

  const EnzoEOSIdeal* eos
    = enzo::fluid_props()->eos_variant().get_if<EnzoEOSIdeal>();

  if (eos == nullptr) {
    ERROR("my_func",
          "my_func only works when Enzo-E is configured with an ideal EOS");
  }
  enzo_float gamma = eos->get_gamma();
  // do work with gamma...
}
```

This snippet makes use of

template<typename **T**>
const *T* *`EnzoEOSVariant::`**`get_if`**`()` const

> Accessor method that returns a pointer to the contained EOS object, if `this` curently holds the EOS object of type `T`. Otherwise, a `nullptr` is returned. The program aborts with an error if `T` is a type that `EnzoEOSVariant` is incapable of holding. A non-`const`-qualified version of this method also exists.
>
> This is a counterpart of the `std::get_if` template function.

Alternatively we could also accomplish the above by writing:

```
void my_func(/* args... */) {

  const EnzoEOSVariant& eos_variant = enzo::fluid_props()->eos_variant();
  if (!eos_variant.holds_alternative<EnzoEOSIdeal>()) {
    ERROR("my_func",
          "my_func only works when Enzo-E is configured with an ideal EOS");
  }
  enzo_float gamma = eos_variant().get<EnzoEOSIdeal>().get_gamma();
  // do work with gamma...
}
```

This last snappet employs the following method:

template<typename **T**>
T *`EnzoEOSVariant::`**`holds_alternative`**`()` const

> Returns whether `this` currently holds the alternative EOS type, `T`. The program aborts with an error, if `T` is a type that `EnzoEOSVariant` is incapable of holding.
>
> This acts as a backport for one of C++17's `std::holds_alternative`

Using EnzoEOSVariant (General semantics)

The EnzoEOSVariant class has semantics just like std::variant (albeit, slightly more limited).

For example, EnzoEOSVariant is never empty. If you call:

```
EnzoEOSVariant my_eos_variant;
```

Then the variable my_eos_variant holds a default-constructed instance of EnzoEOSVariant. At the time of writing this documentation this object will contain an instance of an EnzoEOSIsothermal, but that is an implementation detail that may change over time.

Like std::variant, EnzoEOSVariant also has value-like semantics. This means that any time you perform a copy on an instance of EnzoEOSVariant it's a deepcopy.

```
const EnzoEOSVariant& eos_variant = enzo::fluid_props()->eos_variant();

// make a copy of eos_variant
EnzoEOSVariant my_eos_variant = eos_variant;
```

Any mutations to the contents of my_eos_variant will not affect the contents of enzo::fluid_props()->eos_variant(). Examples might include:

- changing the type of object stored within my_eos_variant (if it initially stores an instance of EnzoEOSIsothermal, we could replace it with an instance of EnzoEOSIdeal)
- mutating the attributes of a stored object within my_eos_variant (one could imagine mutating the value of gamma stored within a EnzoEOSIdeal instance)

As an aside, the API of EnzoPhysicsFluidProps is designed so that the user can't accidentally mutate the PE's EOS object (you can only mutate copies of that object).

Using EnzoEOSVariant (A concrete example)

Many hydro methods need to determine the maximum timestep that they allow. In the process, they may need to compute:

$$C_0 \times \min\left(\frac{\Delta x}{|c_{s,ijk} + v_{x,ijk}|}, \frac{\Delta y}{|c_{s,ijk} + v_{y,ijk}|}, \frac{\Delta z}{|c_{s,ijk} + v_{z,ijk}|}\right)$$

where C_0 is the courant factor (a constant between 0 and 1) and Δx, Δy, Δz specify cell widths.

The following code snippet shows a somewhat simplified example of how you might perform this calculation. This snippet will abort with an error if each Processing Element's global EnzoPhysicsFluidProps instance was configured to hold anything other than an ideal EOS.

```
double timestep(CelloArray<const enzo_float, 3> density,
                CelloArray<const enzo_float, 3> velocity_x,
                CelloArray<const enzo_float, 3> velocity_y,
                CelloArray<const enzo_float, 3> velocity_z,
                CelloArray<const enzo_float, 3> pressure,
                double dx, double dy, double dz,
                double courant_factor)
{
  // the program aborts with an error if Enzo-E was not configured with an
```

(continues on next page)

(continued from previous page)

```
  // ideal EOS (as an aside, we are technically making a copy of the EOS
  // here - that should be totally fine sinces it's just a memcpy)
  const EnzoEOSIdeal eos = enzo::fluid_props()->eos_variant().get<EnzoEOSIdeal>();

  const int mx = density.shape(2);
  const int my = density.shape(1);
  const int mz = density.shape(0);

  double dt = std::numeric_limits<double>::max();
  for (int iz = 0; iz < mz; iz++) {
    for (int iy = 0; iy < my; iy++) {
      for (int ix = 0; ix < mx; ix++) {

        double cs = (double) eos.sound_speed(density(iz,iy,ix),
                                             pressure(iz,iy,ix));
        double abs_vx = std::fabs((double) velocity_x(iz,iy,ix));
        double abs_vy = std::fabs((double) velocity_y(iz,iy,ix));
        double abs_vz = std::fabs((double) velocity_z(iz,iy,ix));
        double tmp = std::min(std::min(dx/(abs_vx + cs),
                                       dy/(abs_vy + cs)),
                              dz/(abs_vz + cs));
        dt = std::min(dt, tmp);
      }
    }
  }

 return courant_factor * dt;
}
```

Using `EnzoEOSVariant` (visitor pattern)

The `EOSVariant::visit()` method can be used to dispatch code based on the type of the EOS that is stored within the EOSVariant. This method effectively implements the visitor design pattern. While this is generally most helpful when you have a collection of objects, it may be helpful in simplifying some code in Enzo-E.

The method that is used to accomplish this is defined below, but it's most useful to consider example cases

template<class **Visitor**>
`EnzoEOSVariant::`**`visit`**`(`*Visitor* &&vis) const noexcept

> invokes the callable visitor, `vis`, by passing the EOS instance held by `this`. The visitor must accept any of the EOS variants passed as an argument, by value, and return an output with a consistent type for all of them.
>
> This acts like a very crude port of `std::visit()` from C++17.

Query whether the EOS is barotropic

Let's consider an example where we want to query whether the EOS is barotropic. We will make use of the `is_barotropic` method that is defined for all EOS classes.

Now the obvious way to write this is:

```
bool is_barotropic_eos(const EOSVariant& eos_variant) {

  if (eos_variant.holds_alternative<EnzoEOSIdeal>()) {
    return eos_variant.get<EnzoEOSIdeal>().is_barotropic();
  } else if (eos_variant.holds_alternative<EnzoEOSIsothermal>()) {
    return eos_variant.get<EnzoEOSIsothermal>().is_barotropic();
  } else {
    ERROR("is_barotropic_eos", "eos_variant holds an unknown eos");
  }
}
```

The code snippet shown above is fine, but it could get tedious to have to modify that code every time that we introduce a new type of EOS. Instead we can write the following snippet, which accomplishes the same thing (but without the caveat):

```
struct IsBarotropicVisitor {
  template <typename T>
  bool operator()(T eos) const { return T::is_barotropic(); }
};

bool is_barotropic_eos(const EOSVariant& eos_variant) {
  return eos_variant.visit(IsBarotropicVisitor());
}
```

This second snippet is still a little verbose. It can further simplify in C++14 to:

```
bool is_barotropic_eos(const EOSVariant& eos_variant) {
  return eos_variant_.visit([](auto eos) { return eos.is_barotropic(); });
}
```

One could imagine that this example generalizes to any case where all EOS classes provide a common interface (e.g., calling the `name` method or the `debug_string` method).

More Sophisticated cases

One could also apply the visitor design pattern in more sophisticated cases, like our *timestep-example*.

Note: It's a little unclear how well this visitor design pattern works with compute kernels. At the end of the day, it may make sense to just drop the `visit` method. (The method's complexity may not be worthwhile)

How to extend this machinery

When you introduce a new EOS class, you need to do three things:

1. You need to update a small subsection of the declaration of `EnzoEOSVariant` where the names of the EOS classes are listed.
2. You need to update the `pup()` routine implemented in the source file for `EnzoEOSVariant`.
3. You need to update the `EnzoConfig::read_physics_fluid_props_()` method to allow the user to specify a new type of EOS.

4.10 Tips for Debugging

This page contains a collection of useful tips for debugging.

4.10.1 General Advice

In general, if you find simple calls to `CkPrintf` are inadequate, you should consider whether the GDB debugger can help. A common scenario where the GDB debugger is useful is when you encounter a segmentation fault (this can occur when you dereference a `NULL` pointer). Essentially, the debugger runs the program until it crashes and then it will let you inspect variables in the stack frames just before the segmentation fault occurred. It can also be very useful to inspect variables in stack frames when the program aborted early during a call to the `ERROR` or `ASSERT` macros.

Debugging a program with GDB is easiest when debugging a problem: replicated in a version of Enzo-E/Cello that was built on top of a netlrts-based build of charm++ on a local machine with xterm windows. In this scenario, you simply need to append `++debug-no-pause` to the arguments of the `charmrun` launcher (e.g. near the arguments specifying the number of nodes). This will cause charm++ to open an xterm window for each node and within each window the program is launched under GDB.

Note: To make optimal use of GDB, you should be sure that you compiled Enzo-E/Cello with debugger symbols. Usually, setting `-DCMAKE_BUILD_TYPE=RELWITHDEBINFO` during the build process is adequate, however some compiler optimizations will prevent you from querying the values of some local variables. This can be addressed by recompiling with `-DCMAKE_BUILD_TYPE=DEBUG`.

You generally don't need to worry about having symbols for other libraries (like charm++).

Note: TODO: add instructions for using GDB on a remote cluster using an MPI-based build of charm++

TODO: it may be useful to provide a link to documentation describing how to use GDB to investigate stack frames

TODO: it may be useful to talk a little about coredumps. Specifically, how do we enable them and how do we use gdb to inspect them?

4.10.2 Iterating on a bug captured by a unit test

When addressing a bug in in the Cello Layer that can be captured by a unit test, it can be a lot faster to simply rebuild the file containing the unit test each time you make a modification, rather than rebuilding all targets.

To be concrete, imagine we were fixing a bug in a subclass of `Schedule`, that was reproduced in the `test_schedule` unit-test. Each time we want to check whether a change to a source/header file fixes the bug, we can rebuild `test_schedule` by executing `make test_schedule` (or `ninja make_schedule`, depending on how your build is set up) from your build directory. In comparison, calling `make` (or `ninja`) would rebuild all binaries and can take a lot more time.

4.10.3 Dumping data to disk

For certain classes of problems, it can be useful to dump data (like field data) to disk (exactly when the problem arises) and investigate the values at a later time

We provide 2 functions for doing this:

template<typename **T**>
void `disk_utils::`**`dump_view_to_hdf5`**(const std::string &fname, CelloView<*T*, 3> view)

create an HDF5 file called `fname` and then save a copy of `view` to it

the view is saved in a group called "data-dump" with the name "data"

Parameters

- **`fname`** – **[in]** is the file name where data will be saved
- **`view`** – **[in]** is the view that is saved to disk

template<typename **T**>
void `disk_utils::`**`dump_views_to_hdf5`**(const std::string &fname, const std::vector<std::pair<std::string, CelloView<*T*, 3>>> &pairs)

create an HDF5 file called `fname` and then save copies of views to it

the view is saved in a group called "data-dump"

Parameters

- **`fname`** – **[in]** is the file name where data will be saved
- **`pairs`** – **[in]** is a vector of (view-name, view) pairs that will be saved to disk. No view-name should be duplicated

4.11 Primer on CMake

There is a lot of excellent documentation for CMake available online. However, some of that documentation can seem overwhelming.

This primer was written to provide some information at a level between the *Getting started using Enzo-E* tutorial and that documentation.

As an aside, when searching for online CMake documentation and guides, you should generally make sure that the guide was written for version 3.0 or later (sometimes called "Modern CMake").

4.11.1 History of Enzo-E's build system

Historically, Enzo-E made use of the SCons build system. However, in 2022 Enzo-E migrated to the CMake build system.

While there are a number of viable build systems, there has been a general convergence among C++ developers towards using CMake (this is evidenced by large projects like Boost and Charm++). Given the very large userbase of CMake, bugs are found and fixed relatively quickly. Moreover, CMake makes the process of locating and linking against external dependencies relatively easy (especially if they are popular libraries OR were themselves built with CMake).

In contrast, SCons is relatively less popular. This means that certain areas of the codebase receive less attention (e.g. less widely used compiler toolchains), which can lead to some issues that are more difficult to work around. Additionally, support staff at clusters are less likely to be familiar with SCons (so they are generally less equipped to help support users)

4.11.2 How Does a CMake Build Work?

For people mostly familiar with build systems like makefiles, CMake may seem a little unintuitive. It may be insightful to walk through some commands step-by-step that would be used in a fresh Enzo-E build.

1. Unlike some other build systems, CMake specializes in out-of-source builds (see *What is an Out-Of-Source Build?* for some additional details/benefits). In other words, for a given build CMake only creates/modifies a file in some user-specified directory "build-directory." Consequently, the first step in initiating a fresh build is to create this "build-directory" and change the terminal's working directory to your newly created directory. Most tutorials suggest that you create a directory called `build` in the root directory of the repository. In reality this directory's name is arbitrary and you can pretty much put it wherever you want.

```
# these commands are typically executed from the root directory of the
# Enzo-E repository
mkdir build
cd build
```

2. The next step is to actually invoke CMake. This step involves the configuration of your build (more details about various configuration options for Enzo-E are provided in *Configuring/Building*). It's important to understand that CMake does not generally compile any code[1]. Instead, this step generates build-files (stored within the build-directory) that are natively understood by the user's platform. For most Enzo-E user/developers (on unix-like systems), CMake will generate makefiles or (optionally) ninja files. When you invoke cmake on the command-line you typically include variables related to the configuration and the final argument specifies the path to the root of the Enzo-E directory. If you placed your build file at the root of your Enzo-E directory, this step might look something like:

```
cmake -DCHARM_ROOT=<PATH/TO/charm/build-dir> \
      -DEnzo-E_CONFIG=linux_gcc -DUSE_GRACKLE=OFF ..
```

3. The last step is to actually invoke the generated build files. If you've been following along, this may correlate to a command like:

```
make -j4 # -j4 tells make to execute up to 4 commands in parallel
```

[1] Technically, CMake's configuration step may include compilation of some short C-programs. This is almost always done in service of testing feature availability of a compiler/platform or testing something about the configuration of a dependency. Files that contribute to the main executable/libraries/tests aren't compiled during this stage.

4.11.3 How are CMake Projects Configured?

CMake makes use of an interpreted scripting language that is typically found in `CMakeLists.txt` text files that are distributed in different directories throughout the project's repository. The primary `CMakeLists.txt` file is found in Enzo-E's root directory.

The scripting language supports many features that include variables, arrays, functions/macros, loops, importing from other files, etc. Care should be taken when using the scripting features. Sometimes the scope rules and caching of variables can lead to somewhat unintuitive behavior.

CMake provides built-in commands that are used to provide standardized information. Examples include `project` (define the name of the project), `add_library` (define/declare a library that is to be built), `add_executable` (define/declare an executable that is to be built), etc.

output targets

The CMake build system revolves around the idea of "targets."

The simplest types of targets are an executable and a static or shared library. The former can be created with `add_executable` and while the latter are created with `add_library`. The source files are usually listed when one of these targets is declared.

After targets are declared, then other properties about the targets are defined. Examples of such properties include:

- Macro definitions that are passed directly through the compiler (via `target_compile_definitions`)
- include directories (via `target_include_directories`)
- dependencies on other libraries (via `target_link_libraries` – and yes, this does indeed apply to static libraries)

These properties are typically defined with different "scopes" (`INTERFACE`, `PUBLIC`, or `PRIVATE`). The scope determines if a property affects the target and/or its dependent(s). When scopes are defined correctly, dependency management becomes straight-forward.

There are also other types of library targets that don't directly correspond to a compiled library. Examples include `INTERFACE` libraries and `IMPORTED` libraries. The former might be used to represent a header-only library or a collection of compiled libraries that you use all at once. Typically `IMPORTED` libraries are used to represent external dependencies.

Note, the usage of Charm++ introduces some additional complexity to this project that is not described here.

`cmake` directory

Following a common convention among CMake projects, we have included a directory called `cmake` at the root level of the repository. This directory holds scripts used for optimizations, building charm++ modules, and locating external dependencies.

`config` directory

Following a convention from Enzo, the `config` directory holds CMake scripts that each define useful variables for the build that are specific to different platforms.

4.11.4 Questions

What is an Out-Of-Source Build?

As stated above CMake specializes in out-of-source builds. In fact, steps have been taken to prevent users from creating in-source builds.

For some context, in-source builds store files created during the build-process (e.g. object files) in the same directories that hold the source files. It's likely that you encountered this kind of build in a tutorial explaining how to use Make. For concreteness, Grackle is a software project that employs in-source builds. Projects that support in-source builds typically need to provide specialized logic for cleaning up from a build (e.g. they usually support `make clean`).

In contrast, out-of-source builds typically put all files created during the configuration/compilation steps into some build directory. Because CMake only mutates the contents of the build directory, and it stores all configuration information within the build directory, cleaning up from a build (to start from scratch), is as simple as deleting the build directory. Out-of-source builds also allow you to maintain multiple different builds at once (that each support incremental recompilation). For example, you may want to have a separate build directory for each branch that you are actively developing.

The fact that CMake stores a given build's configuration within the build directory also makes it possible to maintain separate build directories dedicated to different configurations. This might be useful in the following cases: - while optimizing you might want to have separate build directories dedicated to different compilers (e.g. gcc and icc) - while implementing a new regression test, you may want to have a build directory supporting single precision and another supporting double precision. - while working with code related to an optional dependency (e.g. Grackle), it might be helpful to have one build with the dependency and one build without it

How do I start a new build from scratch?

Just delete your current build directory and make a new one.

As explained in *What is an Out-Of-Source Build?*, you don't actually need to delete an older build to make a new one from scratch. You can just make new build directory with a different name.

Creating a fresh build is tedious. How do I make this easier?

It can be a little tedious to start a fresh build if you have defined a number of parameters on the command line. If it has been a while since your last fresh build it may be difficult to remember all of the configurations. One way to make this easier is to create a custom machine configuration file.

As is detailed below, *here*, a small oversight previously increased how frequently fresh builds were required. At the time of writing this page, this should now be less of an issue.

What do I need to change when I add a new file to the source code?

The answer to this question is: "usually nothing". The CMake system for Enzo-E is configured to use globbing expressions to locate source files that are included in the build. As long your new files are added to one of the main source directories and follows standard naming conventions, the build system should automatically find it.

As an aside, our usage of globbing expressions to locate sources needed by CMake is considered a bad practice by the CMake developers. Their recommendation is that all files used in the build are explicitly listed. This is because CMake generally only adjusts the build system if a `CMakeLists.txt` file has been updated (e.g. if you append a new entry to the list of source files). Historically, if you globbed for source files, CMake wouldn't know that it needed to rebuild the list of source files when you added a new one (since the `CMakeLists.txt` file wasn't touched).

To mitigate some of these issues we make use of a relatively new flag called the `CONFIGURE_DEPENDS` Flag (it's specified with the globbing expressions). When this flag is present, CMake tells the generator (e.g. makefile, ninja) to rerun the glob expression at build time. If there are any differences in the recovered set of files, CMake reconstructs the build system. The CMake developers generally view this unfavorably because:

- globbing expressions can be expensive (this may be less of an issue on Linux compared to windows)
- "[t]he `CONFIGURE_DEPENDS` flag may not work reliably on all generators"

For that reason, we may want to reconsider our approach in the future.

What is Ninja?

Ninja is an alternative build-system to something like Make. Ninja was designed to be a smaller, less-feature-rich alternative to Make that is intended to be used with tools (like CMake) that generate its input files. These design goals can facilitate faster build-times in large projects.

To use `Ninja`, you need to make sure it is installed on your machine and you need to specify `-G Ninja` as one of the configuration arguments in the second step described in *How Does a CMake Build Work?*. Then in the build step, replace `make` with `ninja`.

Does CMake save a log of compiler outputs to any files?

The short answer is no. Unlike the old build system, the current system does not save the compilation outputs to disk. If this is something that would be broadly useful, we might be able to hack something together that accomplishes this.

4.12 Writing Documentation

The Enzo-E/Cello documentation is written in a plaintext markup language called reStructuredText (reST). The Sphinx documentation generator is used to convert the plaintext files into HTML.

The Sphinx documentation provides an excellent primer on reStructuredText here. This is a great place to start if you are interesting in contributing to Cello/Enzo-E's documentation.

This page describes some additional extensions that introduce special features for Enzo-E/Cello's documentation.

4.12.1 The `breathe` extension

To reduce some duplication, we make use of Breathe. It is a Sphinx plugin that we use to selectively integrate documentation generated with Doxygen from docstrings in Enzo-E/Cello's source code.

4.12.2 The `par` extension

We have also written an extension for Sphinx to assist with formatting Cello/Enzo-E's parameters, called `par`. At present, this extension provides 3 primary features:

1. A directive called *`par:parameter`* that is to be used when defining a new parameter in the reference section. Parameters defined with this directive automatically provide an anchor point to facillitate cross-referencing (i.e. using the *`par:param`* role).
2. Pretty-formatting of parameter names. This is done by the *`par:parameter`* directive, as well as the *`par:param`* and *`par:paramfmt`* roles. Note, this functionallity supports special formatting of components of parameter names enclosed by angle brackets (e.g. `:par:paramfmt:`Boundary:<condition>:type`` and `:par:paramfmt:`Initial:value:<field>`` are respectively rendered as Boundary:<condition>:type and Initial:value:<field>)
3. Pretty-formatting of parameter types with the *`par:typefmt`* role. For example, `:par:typefmt:`list (float)`` gets rendered as list (float).

Defining a Parameter

When defining a new parameter in the reference section, you should use the following directive:

`.. par:parameter::`

This directive should be used when defining a new Cello/Enzo-E parameter.

The argument passed to the directive is the name of the parameter being documented. After the line `.. par:parameter:: ParamName`, you should leave a blank line, and then you should define a list of fields (e.g. `Summary`, `Type`, `Default`, `Scope`) that provide an overview of the parameter. Finally, you generally provide an extended description after the field-list.

Example

It is insightful to consider a concrete example. Below we show how to do this for the Method:heat:alpha parameter:

```
.. par:parameter:: Method:heat:alpha

   :Summary:    :s:`Parameter for the forward euler heat equation solver`
   :Type:       :par:typefmt:`float`
   :Default:    :d:`1.0`
   :Scope:      :z:`Enzo`

   :e:`Thermal diffusivity parameter for the heat equation.`
```

This snippet will be rendered as the following (cross-referencing to this output has been disabled):

Parameter
Method : heat : alpha

Summary
Parameter for the forward euler heat equation solver

Type
float

Default
1.0

Scope
Enzo

Thermal diffusivity parameter for the heat equation.

Note: Other potential benefits of *par:parameter*

Defining parameters with this directive could also facillitate other benefits in the future such as:

- the automated populating of parameter tables displayed in the User Guide section
- the automated formatting of the text in the parameter fields
- the displayed format can be easily updated

Formatting parameters names in text

The extension also defines two roles that can be used when mentioning parameters in text.

:par:param:

This formats the name of a parameter nicely and creates a cross reference to an existing parameter definition. For example, `:par:param:`Mesh:root_size`` renders as *Mesh:root_size*.

You can modify this role's behavior by prefixing the content with a ~ or !:

- when the role's content is prefixed with a ~, the rendered link text only shows the final component of the parameter. For example, `:par:param:`~Mesh:root_size`` renders as *root_size* (it links to the same place as *Mesh:root_size*).
- when the role's content is prefixed with a !, no link is constructed (e.g. `:par:param:`!Mesh:root_size`` renders as Mesh:root_size). This does the same thing as *par:paramfmt*

Note: Parameter names that also serve as links are intentionally styled different from non-linked parameter names (to provide readers with a visual cue to the difference). Additionally, unlinked parameter names rendered with this role are more compactly than the titular parameter name in *par:parameter* (since this role is commonly used inline)

:par:paramfmt:

This role is provided as a convenience. It effectively aliases the behavior of the *par:param* role when the content is prefixed by ! (i.e. the name is formatted nicely, but no link is created). For example, `:par:paramfmt:`Mesh:root_size`` renders as Mesh:root_size.

Formatting parameter types in text

We have defined the following role to nicely format the names of parameter types.

:par:typefmt:

Some examples include:

- `:par:typefmt:`float`` renders as float
- `:par:typefmt:`list (float)`` renders as list (float)
- `:par:typefmt:`list (float-expr, [logical-expr, float-expr, [...]])`` renders as list (float-expr, [logical-expr, float-expr, […]])

CHAPTER

FIVE

DESIGN DOCUMENTS

This section describes some of the lower-level designs of Enzo-E and Cello, including flux-correction, IO, and ghost zone refresh.

5.1 Adapt Design

In the *adapt phase*, blocks may refine or coarsen to adapt to the evolving resolution requirements of a simulation. The main complication is enforcing the "level-jump" condition, which prohibits adjacent blocks from being in non-consecutive mesh refinement levels. (Blocks are partitioned into "levels" based on how refined they are: more highly-refined blocks are in higher-numbered levels, with the "root-level" of the simulation defined as "level 0". The difference in resolution between any pair of successive levels L and L+1 (the "refinement factor") is always 2 in Enzo-E.)

Maintaining the level-jump conditions may require refining blocks that would not otherwise be refined, or may require not coarsening blocks that would otherwise be coarsened. The process of refining blocks in a mesh hierarchy solely to maintain the level-jump condition across block faces is called *balancing* the mesh (not to be confused with *dynamic load balancing*)

Figure 1. illustrates the steps used in the adapt phase. Suppose we begin with the mesh hierarchy at the left, which contains seven blocks: three in a coarse level and four in the next finer level. The first step involves applying local refinement criteria to each block; in this particular example, the center-most fine block is tagged for refinement, here indicated by a "+" in the left-most image.

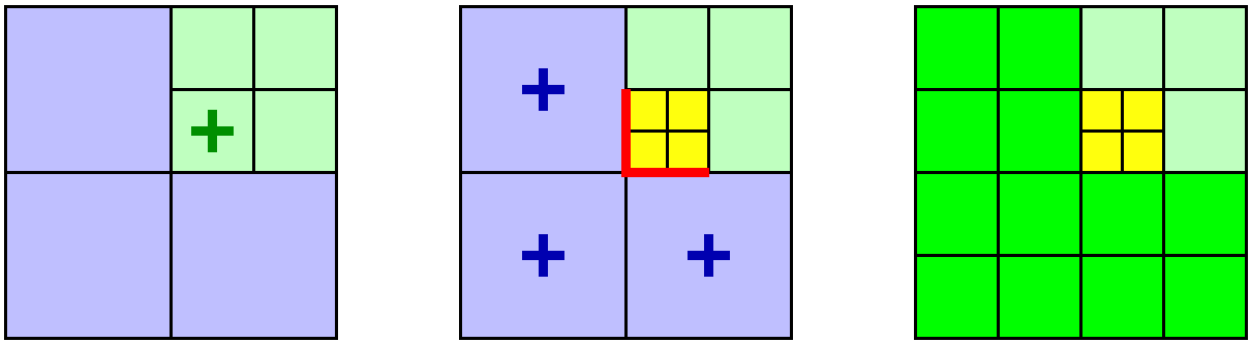


Fig. 1: **Figure 1.** Refining a block (left) may trigger further refinements (center) to maintain the level-jump condition.

If we were to only refine this block, however, level jumps would be introduced across the faces marked by red lines in the center image (here we optionally include corners as "faces"). By refining the coarse blocks, these level jumps are removed. This final mesh after completing the balancing step is shown on the right.

We note that blocks marked for refinement solely to maintain the level-jump condition may themselves trigger further refinement in neighboring blocks. While such cascades can repeat multiple steps, each block in the cascade is in a

coarser level than its predicessor, so cascades are always guaranteed to terminate. However, cascades still complicate parallelizing the algorithm, since any given block may not immediately know whether it needs to refine (or not coarsen) so determining when the balancing step of the adapt phase is actually complete is non-trivial.

5.1.1 Revised adapt algorithm description

In this section we describe a revised algorithm for the adapt phase in Enzo-E/Cello. This algorithm was first developed by Phil Miller, and is presented in his Ph.D. Dissertation, Reducing synchronization in distributed parallel programs (University of Illinois at Urbana-Champaign, 2016).

The previous parallel algorithm implemented in Cello relied on Charm++'s support for *"quiescence detection"*, which is defined as *"the state in which no processor is executing an entry point, no messages are awaiting processing, and there are no messages in-flight"* (see The Charm++ Parallel Programming System) Getting this algorithm to work correctly required considerable effort and debugging, and even after several years of development on Enzo-E / Cello, users still occasionally ran into issues of level-jumps in the resulting mesh hierarchy.

Miller's algorithm avoids using quiescence detection in favor of a more direct approach. First, each block evaluates its local adapt criteria to determine whether it needs to refine, stay in the same level, or can coarsen. Next, both lower and upper bounds on mesh levels are determined for each block and communicated with neighbors. Bounds for a block may be adjusted as newer updated bounds arrive from neighboring blocks. When a block's minimum and maximum levels match, the block's next level is decided. All leaf blocks are guaranteed to reach this state, which can be proven by induction on the mesh level starting with the finest level (See Miller 2016).

Before presenting the algorithm, we define the following notation:

- B_i *block i*
- B_j *a block adjacent to block i*
- L_i^k *the level of Block i in cycle k*
- $\hat{L}_i^{k+1}$ *block i's desired next level as locally-evaluated from refinement criteria*
- $\underline{L}_{i,s}^{k+1} \leq L_i^{k+1} \leq \bar{L}_{i,s}^{k+1}$: *current lower and upper level bounds (for step s), which are dynamically updated*
- L_i^{k+1} *the next level which is decided when* $\underline{L}_{i,s}^{k+1} = \bar{L}_{i,s}^{k+1}$

We can now write the two main conditions that we use to initialize and update the level bounds:

- $|L_i^k - L_i^{k+1}| \leq 1$ *the (temporal) level-jump condition: a block can refine or coarsen at most once per adapt cycle*
- $|L_i^k - L_j^k| \leq 1$ *the (spacial) level-jump condition: refinement levels of adjacent blocks can differ by at most one*

Level bounds are initialized to be $\underline{L}_{i,0}^{k+1} \leftarrow \hat{L}_i^{k+1}$ and $\bar{L}_{i,0}^{k+1} \leftarrow L_i^k + 1$. That is, the minimum level is initially the level determined by the local refinement criteria, and the maximum level is initially one level of refinement more than the current level (or the maximum allowed level in the simulation.)

The balancing step of the algorithm proceeds by alternately sending a block's level bounds to its neighbors, and, having received updated bounds from its neighbors, updating the block's own level bounds. Bounds are updated according to the following:

$\underline{L}_{i,s+1}^{k+1} \leftarrow \max(\underline{L}_{i,s}^{k+1}, \max_j(\underline{L}_{j,s}^{k+1} - 1))$

$\bar{L}_{i,s+1}^{k+1} \leftarrow \max(\underline{L}_{i,s}^{k+1}, \max_j(\bar{L}_{j,s}^{k+1} - 1))$

The lower bound is updated if any neighbor's minimum bound is greater than one plus the block's current minimum bound. The maximum bound, which is used to determine when the balancing algorithm terminates, is defined as the maximum of the minimum bound, and the maximum of *all* neighboring maximum bounds minus one. Note that in general the maximum bound can only be updated after all neighboring blocks have been heard from. Additional synchronization is required for a block to coarsen, since a block can coarsen only if all of its siblings can as well.

5.1.2 Revised adapt algorithm implementation

To reduce the complexity of the already over-burdened Block classes, we introduce an `Adapt` class to maintain and update level bounds for a Block and its neighbors. The `Adapt` class keeps track of the current level bounds of all neighboring blocks, which is redundantly stored as a list of `LevelInfo` objects for each neighboring Block, and a `face_level_` vector of the current level in the direction of each face. (The *face_level_* representation is a carry-over from the previous algorithm, but was retained because it simplifies code that needs to access a neighbor's level given the neighbor's relative direction rather than absolute Index). Below summarizes the API for the newer `LevelInfo` section, which is used to collectively determine the next level for all blocks in the mesh.

`void set_rank (int rank)`

Set dimensionality of the problem $1 \leq \text{rank} \leq 3$. Only required for initialization in test code, since Cello initializes it using *cello::rank()*.

`void set_valid (bool valid)`

Set whether the Adapt object is "valid" or not. Set to false when the corresponding Block is refined. "valid" is accessed internally when a block is coarsened to identify the first call triggered by child blocks. It's set to true internally after the first call to coarsen().

`void set_periodicity (int period[3])`

Set the periodicity of the domain, so that the correct neighbors can be identified on domain boundaries.

`void set_max_level (int max_level)`

Set the maximum allowed mesh refinement level for the problem.

`void set_min_level (int min_level)`

Set the minimum allowed mesh refinement level for the problem.

`void set_index (Index index)`

Set the index of the Adapt object's associated block.

`void insert_neighbor (Index index)`

Insert the given Index into the list of neighbors. This is a lower-level routine and should generally not be called–use `refine_neighbor()` instead.

`void insert_neighbor (Index index, bool is_sibling)`

Insert the given Index, and specify that the Block is a sibling. This version is used exclusively in test code in test_Adapt.cpp.

`void delete_neighbor (Index index)`

Delete the specified neighbor. This is a lower-level routine and should generally not be called–use `coarsen_neighbor()` instead.

`void reset_bounds ()`

Reset level bounds for this block and neighbor blocks in preparation for a new adapt phase.

`void refine_neighbor (Index index)`

Update the list of neighboring blocks associated with refining the specified neighbor block.

`void coarsen_neighbor (Index index)`

Update the list of neighboring blocks associated with coarsening the specified neighbor block.

`void refine(Adapt adapt_parent, int ic3[3])`

Update the Adapt object for a recently refined block. The block's parent adapt object is passed in to update the neighbor lists accordingly, and which child this block is in its parent block is specified by ic3[].

`void coarsen(Adapt adapt_child)`

Update the adapt object for a recently coarsened block. Must be called exactly once for each coarsened child (in any order), specified by the child block's associated Adapt object. This is required to update the neighbor lists correctly.

`void initialize_self(Index index, int level_min, int level_now)`

Initialize the adapt object with the given Block index and level bounds.

`void update_neighbor(Index index, int level_min, int level_max, bool can_coarsen)`

Update the specified neighbor block's level bounds and "can_coarsen" attribute.

`void update_bounds()`

Reevaluate the block's level bounds given the current level bounds of all neighbors.

`bool is_converged()`

Return whether the level bounds of this block have converged to a single value (that is min_level == max_level).

`bool neighbors_converged()`

Return whether all neighboring block's level bounds have converged.

`void get_level_bounds(int * level_min, int * level_max, bool * can_coarsen)`

Get the current level bounds and "can_coarsen" attribute for this Block. Must be preceeded by a call to "update_bounds()".

`bool get_neighbor_level_bounds(Index index, int * level_min, int * level_max, bool * can_coarsen)`

Return the level bounds and "can_coarsen" attribute for the specified neighbor.

`int level_min()`

Return the current lower bound on this block's refinement level.

`int level_max()`

Return the current upper bound on this block's refinement level.

`bool can_coarsen()`

Return the current value of "can_coarsen" for this block.

`int num_neighbors()`

Return the number of neighbors for this block.

`int is_sibling(int i)`

Return whether the ith neighbor is a sibling of this block (whether the neighbor block and this block share the same parent.

`Index index()`

Return the Block index associated with this Adapt object.

`Index index(i)`

Return the Block index for the ith neighbor block.

5.2 Flux Correction Design

5.2.1 Requirements

Flux correction involves updating field values along faces between neighboring blocks to ensure that conserved quantities are conserved across jumps in spacial and temporal resolution. The update involves adding correction factors to all coarse values that lie along coarse-fine interfaces, where the correction factors are computed using the coarse and fine fluxes across the interface. While the basic update is straightforward, care must be taken to ensure that the correction factors are computed correctly, especially for non-centered field variables or when adaptive time stepping is used. Additional corrections will be required for MHD, and expansion terms in cosmological problems may also need to be considered.

Basic operations involved include the following:

1. allocating and deallocating flux data
2. setting and accessing flux values
3. communicating fluxes between neighboring blocks (fine-to-coarse)
4. computing correction factors given coarse and fine fluxes
5. correcting coarse-level field values given correction factors

Relative to ENZO's structured AMR grids, flux correction for Enzo-E / Cello is simpler due to Cello's array-of-octree refinement: blocks only share fluxes at block faces (Enzo-E blocks do not contain sub-blocks), and the topology of fine- and coarse-level block intersections is simpler (Enzo-E coarse-level block faces are adjacent to exactly four fine-level block faces). The first simplification removes the need to loop over sub-grids or store fluxes internal to a block (i.e. the "projection step" in ENZO), and the second removes the need to explicitly store loop indices for flux arrays.

We assume fluxes for all conserved fields are provided by the hydrodynamics solver along all required block faces at each time step.

Method requirements

The Method component is responsible for storing, accessing, communicating, and operating with fluxes, to implement flux correction using support provided by Cello flux classes. Specific requirements are listed below:

- **RE-1.** Initialize and store fluxes, provided by the hydro solver, at each time step for all conserved fields across all coarse-fine interfaces
- **RE-2.** Request required fluxes from neighboring Blocks
- **RE-3.** Compute correction factors for all required conserved field face values given a block face and fluxes from both the block and its face-sharing neighbor
- **RE-4.** Correct conserved field face values given computed correction factors
- **RE-5.** Identify which fields require flux correction
- **RE-6.** Support adaptive time stepping
- **RE-7.** Support MHD
- **RE-8.** Ensure conservation is not lost through any other operation, e.g. interpolation

Data requirements

Cello's flux data classes provide the flux-correction method with sufficient support for storing fluxes, computing flux correction factors, and applying the flux correction to conserved field values.

Specific requirements include the following:

- **RC-1.** Store fluxes of conserved fields that lie along block faces
- **RC-2.** Store associated fluxes computed on adjacent blocks
- **RC-3.** Communicate fluxes between adjacent blocks when (and ideally only when) needed
- **RC-4.** Store the time interval along with each collection of fluxes
- **RC-5.** Allow multiple fluxes to be stored for the same field but different time steps
- **RC-6.** Provide support for a block to compute correction factors along a block face given the block's fluxes and the corresponding neighboring block fluxes
- **RC-7.** Provide support for correcting field values along a block face given the computed correction factors
- **RC-8.** Allow for dynamic allocation and deallocation of fluxes (optional)
- **RC-9.** Ensure conservative inter-grid interpolation and coarsening

5.2.2 Design

Our design is developed top-down, starting with a Cello Method for implementing flux correction, *MethodFluxCorrect*.

To support coding MethodFluxCorrect, our design uses a *FluxData* class, whose responsibility is to manage all flux data on a block. This FluxData class is analagous to the existing FieldData and ParticleData classes. Like FieldData and ParticleData objects, FluxData is contained in the Blocks Data object. Communication is handled by the refresh mechanism by augmenting the existing communication of field and particle data between Blocks to include communicating fluxes.

While flux data could be implemented directly as arrays (e.g. std::vector, EnzoArray, etc.) other attributes need to be associated with each collection of flux data, specifically the Block the fluxes were computed on, which Block face the fluxes are associated with, the time interval for the fluxes, etc. For this we introduce a lower-level class *FaceFluxes* to store the array of flux data for an individual face, along with its defining attributes. The FaceFluxes class is in turn implemented usingt a simple *Face* class that defines the block face on which the fluxes are defined.

Interfaces

We develop the interfaces for the flux-related classes below, starting with the top-level MethodFluxCorrect, then the progressively higher-level Face, FaceFluxes, and FluxData classes.

MethodFluxCorrect class

The MethodFluxCorrect class is a Cello Method, whose main virtual method is compute(Block). This operates on some subset of data types on a Block. The MethodFluxCorrect method initiates communication between processes, and computes and applies appropriate flux-correction operations on required Field values along block interfaces.

Since the MethodFluxCorrect class is inherited from the Cello Method class, the public interface for this class is already prescribed. The two methods in the interface are the constructor MethodFluxCorrect() used to initialize the required communication, and MethodFluxCorrect::compute(Block) which implements flux correction on a given Block. (The other virtual function in the Method interface is timestep(), which is not required for flux-correction.)

`MethodFluxCorrect::MethodFluxCorrect()`
Create a new MethodFluxCorrect object, and define its refresh communication requirements

---

`virtual void MethodFluxCorrect::compute (Block * block)`
Request Cello to refresh its flux data, then apply flux correction

- **block**: *Block that flux correction is being applied to*

Face class

A block Face is any facet, edge, or corner of a block. For flux correction in hydrodynamics, one typically only deals with facets, or (d-1)-dimensional faces; however, for MHD, edge faces may be used as well.

A face is determined by its "center" (ix,iy,iz), -1 <= ix,iy,iz <= +1, assuming the block corners are at (+/-1,+/-1,+/-1). As examples, the positive Y-axis facet is (0,+1,0), the edge along X=+1 and Z=-1 is (+1,0,-1), and the entire block as a 3-D "face" is (0,0,0).

Each Face is associated with a normal vector defining the direction of the fluxes. This direction is given by the axis (x=0, y=1, z=2), and face (lower=0, upper=1).

`Face::Face (int ix, int iy, int iz, int axis, int face)`
Create a Face object for a Block associated with face (ix,iy,iz), -1 <= ix,iy,iz <= +1, with fluxes in the direction (axis, face), 0 <= axis < rank, 0 <= face <= 1.

`void Face::get_face (int *ix, int *iy, int *iz)`
Return the tuple (ix,iy,iz), -1 <= ix,iy,iz <= 1, identifying the block's face, which may be a corner, edge, facet, or the entire block.

`int Face::axis()`
Return the axis associated with the normal direction: x=0, y=1, or z=2.

`int Face::face()`
Return whether the normal direction is towards the lower (0) or upper (1) face direction.

FaceFluxes class

Face fluxes represent an array of fluxes of a given conserved Field through a Block's face or subset of a face. Operations available for fluxes including *coarsening*, for summing fluxes in a finer block to match the resolution of a neighboring coarser block, and *summing*, for accumulating a sequence of fluxes associated with a block with a finer time step to match a neighboring block's coarser time step.

`FaceFluxes::FaceFluxes (Face face, int index_field, int nx, int ny, int nz, int cx, int cy, int cz)`
Create a FaceFluxes object for the given face, field, and block size. Optionally include centering adjustment (0 <= cx,cy,cz <= 1) for facet-, edge-, or corner-located field values

---

void FaceFluxes::allocate()
Allocate the flux array and initialize values to 0.0.

---

void FaceFluxes::deallocate()
Deallocate the flux array.

---

void FaceFluxes::clear()
Set flux array values to 0.0.

Face FaceFluxes::face()
Return the face associated with the FaceFluxes.

---

int FaceFluxes::index_field()
Return the associated field index.

---

void FaceFluxes::get_size (int * mx, int * my, int * mz)
Return the array dimensions of the flux array, including any adjustments for centering. Indexing is ix + mx(iy +my*iz).*

---

void FaceFluxes::set_flux_array (std::vector<double> array, int dx, int dy, int dz)
*Copy flux values from an array to the FluxFaces flux array. Array element array[ix*dx + iy*dy + iz*dz] should correspond to flux value (ix,iy,iz), where (0,0,0) <= (ix,iy,iz) < (mx,my,mz).*

---

std::vector<double> & FaceFluxes::flux_array (int * dx=0, int * dy=0, int * dz=0)
*Return the array of fluxes and associated strides (dx,dy,dz) such that the (ix,iy,iz) flux value is fluxes[ix*dx + iy*dy + iz*dz], where (0,0,0) <= (ix,iy,iz) < (mx,my,mz).*

---

void FaceFluxes::coarsen(int cx, int cy, int cz, int rank)
Used for coarsening fine-level fluxes to match coarse level fluxes. Arguments (cx,cy,cz) specify the child indices of the block within its parent (not to be confused with centering (cx_,cy_,cz_); flux array size is kept the same, with offset determined by child indices.

---

void FaceFluxes::accumulate (FaceFluxes & ff, int cx, int cy, int cz, rank)
Add the FaceFluxes object to this one. Used for accumulating fluxes with finer time steps until they match the coarser time step. Assumes spacially-conforming FaceFluxes objects.

FaceFluxes & FaceFluxes::operator *= (double weight)
Scale the fluxes array by a scalar constant.

FluxData class

The FluxData class defines a collection of all FluxFaces required by a Block to perform flux corrections. This includes all flux arrays on Faces whose neighboring Block differs in either mesh refinement level or time step. FluxFaces for faces that require flux-correction come in conforming pairs, one set of fluxes corresponding to the Block, and one corresponding to the block's neighbors. Flux arrays for the neighboring block are received in the flux refresh operation. Support for coarsening, adding, and differencing fluxes is the responsibility of the FaceFluxes class; FluxData is primarily a container.

FluxData::FluxData()
Create an empty FluxData() object

---

void FluxData::allocate(int nx, int ny, int nz, std::vector<int> field_list, std::vector<int> * cx_list=nullptr, std::vector<int> * cy_list=nullptr, std::vector<int> * cz_list=nullptr)
Allocate all flux arrays for each field in the list of field indices. Optional arrays to indicate the centering of fields may also be provided.

---

void FluxData::deallocate()
Deallocate all face fluxes for all faces and all fields.

---

int FluxData::num_fields()
Return the number of field indices.

---

int FluxData::index_field(int i_f)
Return the i'th field index.

---

void FluxData::block_fluxes(int axis, int face, int i_f)
Return the face fluxes object associated with the given facet and field. Note 0 <= i_f < num_fields() is an index into the field_list vector, not the field index itself.

---

void FluxData::neighbor_fluxes(int axis, int face, int i_f)
Return the neighboring block's face fluxes associated with the given facet and field. Note 0 <= i_f < num_fields() is an index into the field_list vector, not the field index itself.

---

void FluxData::set_block_fluxes(FaceFluxes * ff, int axis, int face, int i_f)
Set the block's face fluxes associated with the given facet and field. Note 0 <= i_f < num_fields() is an index into the field_list vector, not the field index itself.

---

void FluxData::set_neighbor_fluxes(FaceFluxes * ff, int axis, int face, int i_f)
Set the neighboring block's face fluxes associated with the given facet and field. Note 0 <= i_f < num_fields() is an index into the field_list vector, not the field index itself.

---

void FluxData::sum_neighbor_fluxes(FaceFluxes * ff, int axis, int face, int i_f)
Accumulate (sum) the neighboring block's face fluxes associated with the given facet and field. Note 0 <= i_f < num_fields() is an index into the field_list vector, not the field index itself.

Flux Communication

Communicating fluxes between adjacent blocks is similar to communicating field face data and migrating particles, so it makes sense to augment the existing refresh mechanism for communicating FieldData and ParticleData to also refresh FluxData. We briefly outline a couple possible differences below.

First, flux data is generally only communicated from coarse blocks to neighboring fine blocks, and from blocks with a smaller time step to neighboring blocks with larger time steps. Thus an additional communication pattern and associated synchronization could be introduced. Examples of existing synchronization patterns are sync_neighbor type for communicating between all pairs of adjacent leaf blocks, and sync_level between all adjacent blocks in the same level. For fluxes, we could introduce a sync_fluxes synchronization type, which by definition includes

1. leaf blocks only
2. *usually* only d-1 faces (may need block edge- or corner-adjacent flux data for MHD)
3. only from finer resolution to coarser resolution (temporal as well as spacial)

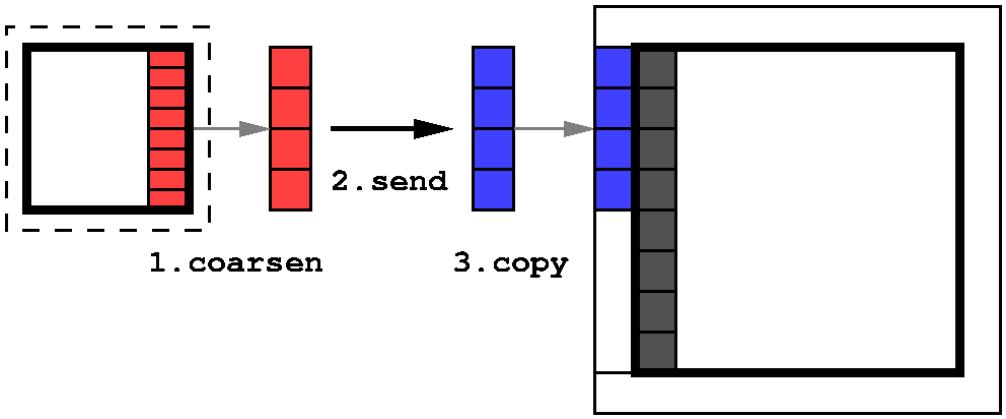


Fig. 2: Communicating fluxes assuming constant time steps.

Testing

Multiple levels of testing are used, including unit tests for the lower level Face, FaceFluxes, and FluxData classes, and application testing for MethodFluxCorrect.

Application tests include varying difficulties of meshes, physics, boundary conditions, and floating-point precision. Different levels are summarized below:

- **Mesh**
  - **M0** single block
  - **M1** unigrid
  - **M2** one additional refinement level
  - **M3** two additional refinement levels
- **Physics**
  - **PH** hydro

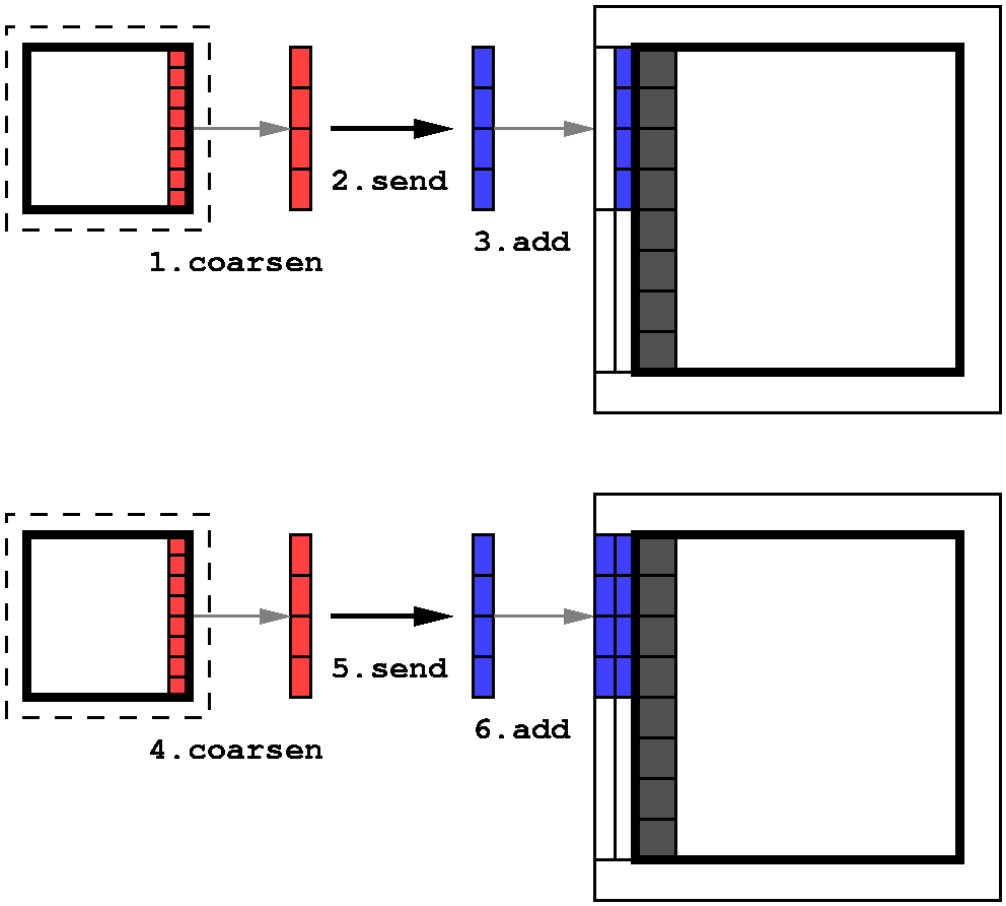


Fig. 3: Communicating fluxes assuming adaptive time steps.

  - **PG** gravity
  - **PP** gravitating particles
  - **PC** cosmological expansion
- **Boundary**
  - **BP** periodic
  - **BR** reflecting
- **Floating-point precision**
  - **F4** single
  - **F8** double
- **Parallelism**
  - **p1**: single processor
  - **pc**: parallel across cores
  - **pn**: parallel across nodes

Documentation

- **DD-1.** *Design*: add flux correction design to design/design-flux.rst
- **DD-2.** *Method*: add MethodFluxCorrect method documentation to user/problem_method.rst
- **DD-3.** *Testing*: add testing/testing_flux.rst test documentation
- **DD-4.** *Parameters*: update doc/source/param/ parameter documentation

5.2.3 Milestones

- **M-1.** MethodFluxCorrect demonstrated working for hydrodynamics
- **M-2.** MethodFluxCorrect demonstrated working for MHD

5.2.4 Tasks

- **T-1.** FaceFluxes class design and implementation
- **T-2.** FluxData class design and implementation
- **T-3.** MethodFluxCorrect class design and implementation

5.3 Checkpoint/Restart Design

5.3.1 Requirements

Three code functional requirements of I/O in Cello are:

1. writing data dumps for subsequent reading by external analysis/visualization applications
2. writing checkpoint files, and
3. reading checkpoint files to restart a previously run simulation

(While writing image files such as "png" files is also included in the I/O component of Cello, here we focus on HDF5 files containing actual block data.)

Additionally, writing and reading disk files must be scalable to the largest simulations runnable on the largest HPC platforms available, which necessarily include the largest parallel file systems available.

This scalable I/O approach has been implemented for checkpoint/restart, and will be adapted for use with data dumps in the near future.

5.3.2 Approach

The approach used includes determining a block ordering to aid mapping blocks to files, what data are written to the files, and how file I/O is parallelized.

Ordering

The approach involves a generalization of the previous `MethodOutput` method, but enables load-balancing of data between disk files through the use of block *orderings* to define how blocks are mapped to files. Currently, the ordering used in `MethodOutput`, which is implicit and embedded in the code, is based on a regular partitioning of root-level blocks together with their descendents. The updated implementation factors out this ordering into an `Ordering` class, provides a Morton space-filling curve ordering, and allows enables defining other orderings, such as Hilbert curves

File content

The content of the data files must be augmented to include all state data required to recreate a previously saved AMR block array on restart. Some information such as block connectivity are generated as blocks are inserted into the mesh hierarchy. Other information such as method or solver parameters are not stored, but are taken from the parameter file. This allows for "tweaking" of parameters on restart, for example to adjust refinement criteria or solver convergence criteria.

Control flow

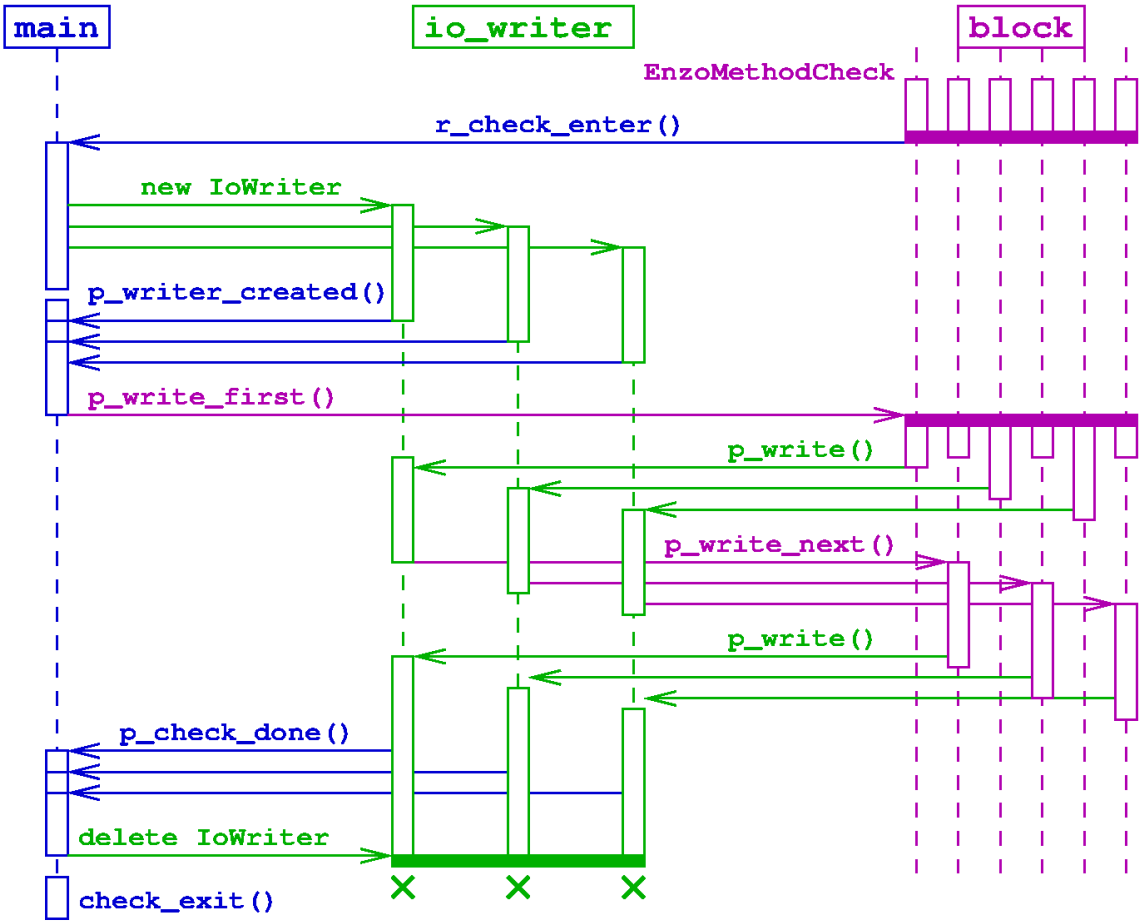
Control flow is handled by separate `IoWriter` or `IoReader` chare arrays, where each element is associated with a single HDF5 file. Advantages over previous approaches are better load-balancing of I/O operations, and decoupling of I/O operations from the Block chare array. For Enzo-E checkpoint/restart data in particular, `IoEnzoReader` and `IoEnzoWriter` chare arrays are used.

5.3.3 Design

Components of the new I/O approach include

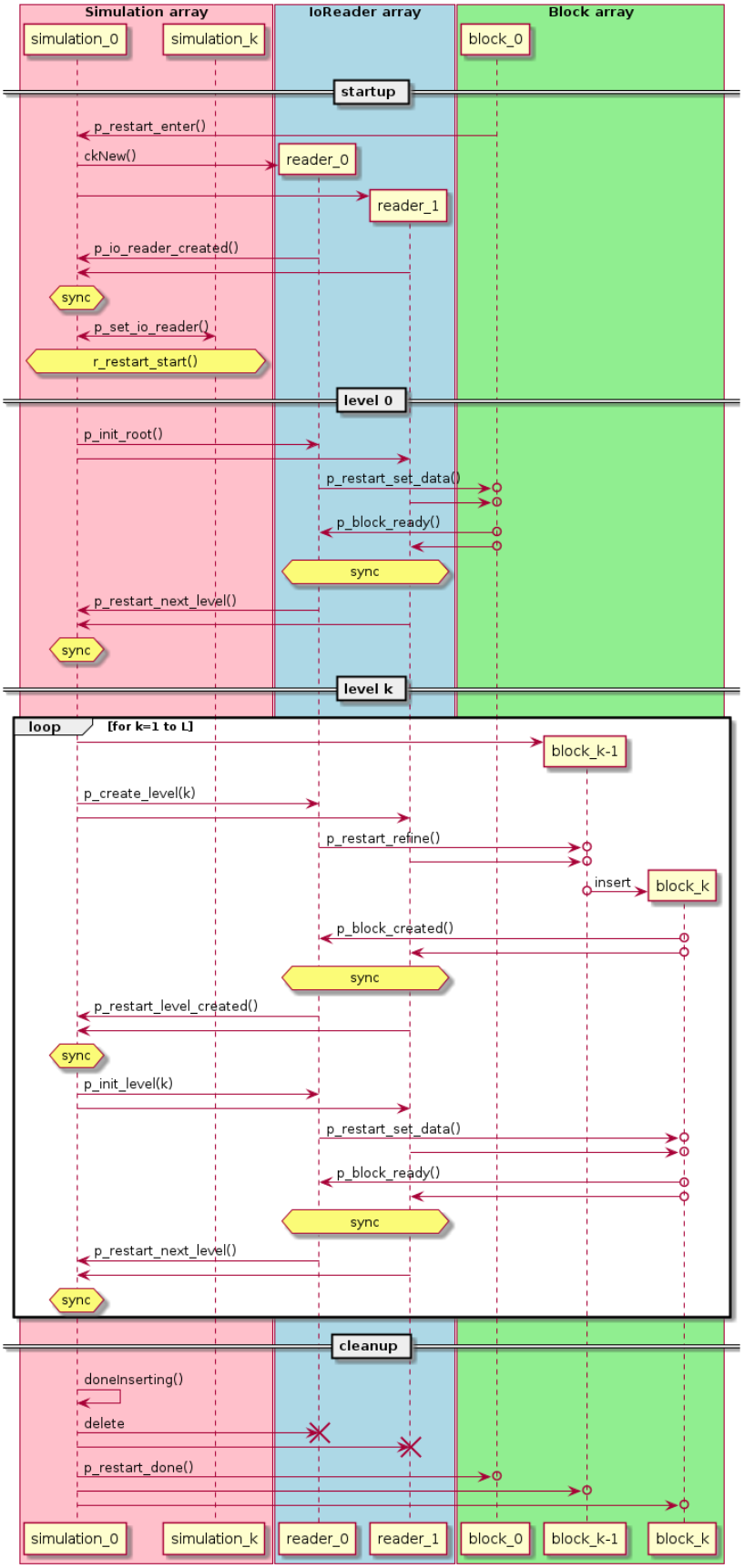
1. Control management
   - `control_restart.cpp`
     - `Main::r_restart_enter()`
     - `Main::p_restart_done()`
     - `Main::restart_exit()`
2. New Classes
   - `EnzoMethodCheck`
   - **`IoEnzoReader`**
     - `IoEnzoReader::IoEnzoReader()`
   - **`IoEnzoWriter`**
     - `IoEnzoWriter::IoEnzoWriter()`
   - **`IoReader`**
     - `IoReader::IoReader()`
   - **`IoWriter`**
     - `IoWriter::IoWriter()`
   - `MethodOrderMorton`

Output: checkpoint



Input: restart

The UML sequence diagram below shows how the `Simulation` group, IoReader chare array, and Block chare array interoperate to read data from a checkpoint directory. Time runs vertically starting from the top, and the three Charm++ group/arrays are arranged into three columns. Code for restart is found in the `enzo_control_restart.cpp` file.


Simulation array
IoReader array
Block array
simulation_0
simulation_k
block_0
startup
p_restart_enter()
ckNew()
reader_0
reader_1
p_io_reader_created()
sync
p_set_io_reader()
r_restart_start()
level 0
p_init_root()
p_restart_set_data()
p_block_ready()
sync
p_restart_next_level()
sync
level k
loop
[for k=1 to L]
block_k-1
p_create_level(k)
p_restart_refine()
insert
block_k
p_block_created()
sync
p_restart_level_created()
sync
p_init_level(k)
p_restart_set_data()
p_block_ready()
sync
p_restart_next_level()
sync
cleanup
doneInserting()
delete
p_restart_done()
simulation_0
simulation_k
reader_0
reader_1
block_0
block_k-1
block_k

startup

Restart begins in the "startup" phase, with the unique root block for the (0,0,0) octree in the array-of-octrees calling the `Simulation` entry method `p_restart_enter()`.

The `p_restart_enter()` entry method reads the number of restart files from the top-level *file-list* file, initializes synchronization counters, and creates the `IoEnzoReader` chare array, one element for each file.

The `IoEnzoReader` constructors calld the `p_io_reader_created()` entry method in the root `Simulation` object to notify it that they've been created.

`p_io_reader_created` counts the number of calls, and after it has received the last `IoEnzoReader` notification, it distributes the `proxy_io_enzo_reader` array proxy to all other `Simulation` objects by calling `p_set_io_reader()`.

`p_set_io_reader()` stores the incoming proxy, then calls the `r_restart_start()` barrier across `Simulation` objects, which is used to guarantee that all proxy elements will have been initialized before any are accessed in subsequent phases.

level 0

In the level-0 (root-level) phase, the root `Simulation` object reads the file names from the *file-list* file, and calls the `p_init_root()` entry method in all `IoEnzoReader` objects, sending the checkpoint directory and file names.

The `p_init_root()` entry method opens the *block-data* (HDF5) file and reads global attributes. It also opens and reads tho *block-list* (text) file, reading in the list of blocks and organizing them by mesh refinement level. It reads in each block data, saving data in blocks levels greater than 0, and sending data to level-0 blocks. Note level-0 blocks exist at the beginning of restart, but no blocks in levels higher than 0 do. Data are packed and sent to blocks in levels <= 0 using the `EnzoBlock::p_restart_set_data()` entry method.

The `EnzoBlock::p_restart_set_data()` method unpacks the data into the Block, then notifies the associated `IoEnzoReader` file object that data has been received using the `p_block_ready` entry method.

`IoEnzoReader::p_block_ready()` counts the number of block-reday acknowledgements, and after the last one calls `Simulation::p_restart_next_level()` to process the next refinement level blocks.

level k

The level-k phase for k=1 to L is more complicated than level-0 because the level k > 0 blocks must be created first.

Assuming blocks up through level k-1 have been created, the root `Simulation` object calls `IoEnzoReader::p_create_level(k)` for each `IoEnzoReader`.

In `p_create_level()`, synchronization counters are initialized for counting the k-level blocks, and then each block in the list of level-k blocks is processed. To reuse code from the adapt phase, level-k blocks are created by refining the *parent* block, via a `p_restart_refine()` entry method.

In `p_restart_refine()`, the parent level k-1 block creates a new child block, inserts the new block in its own child list, and recategorizes as a non-leaf.

In the `EnzoBlock` constructor, the newly created block checks if it's in a restart phase, and if so sends an acknowledgement to the associated `IoEnzoReader` object using the `p_block_created()` entry method.

In `p_block_created` the `IoEnzoReader` object counts the number of acknowledgements from newly-created level-k blocks, and after it receives the last one it calls `p_restart_level_created()` on the root-level `Simulation` object. After this, the rest of the level-k phase mirrors that of the level-0 phase.

cleanup

In the cleanup section, after all blocks up to the maximum level have been created and initialized, the `p_restart_next_level()` entry method calls the Charm++ call `doneInserting()` on the block chare array, then calls `p_restart_done()` on all the blocks, which completes the restart phase.

Classes

EnzoMethodInput

5.3.4 Data format

Data for a given checkpoint dump are stored in a single checkpoint directory, specified in the user's parameter file using the `Method:check:dir` parameter.

The number of data files in the directory is specified using the `Method:check:num_files` parameter. A rule-of-thumb is to use the same number of files as (physical) nodes in the simulation.

Data files are named `block_data-` *x* `.h5`, where 0 <= x < `num_files`. The format of data files is given in the next section.

Each data file has an associated *block-list* text file named `block_data-` *x* `.block_list`. The block-list file contains a list of all block names in the associated data file, together with each block's mesh refinement level. There is one block listed per line, and the block name and level are separated by a space.

A `check.file_list` text file is also included, which includes the number of data files, and a list of the file prefixes `block_data-` *x*.

Note all blocks are included in the files, not just leaf-blocks, and including blocks in "negative" refinement levels.

Data file contents

The HDF5 data files are used to store all block state data, as well as some global data.

Simulation attributes

Metadata for the simulation are stored in the top-level "/" group. These include the following:

- *cycle*: Cycle of the simulation dump.
- *dt*: Current global time-step.
- *time*: Current time in code units.
- *rank*: Dimensionality of the problem.
- *lower*: Lower extents of the simulation domain.
- *upper*: Upper extents of the simulation domain.
- *max_level*: Maximum refinement level.

Block attributes

Block attributes and data are stored in HDF5 groups with the same name as the block, e.g. "B00:0_00:0_00:0".

Block attribute data include the following:

- *cycle*: Cycle of this block.
- *dt*: Current block time-step.
- *time*: Current time of this block.
- *lower*: Lower extents of the block.
- *upper*: Upper extents of the block.
- *index*: Index of the block, specified using three 32-bit integers.
- *adapt_buffer*: Encoding of the block's neighbor configuration.
- *num_field_data*: currently unused.
- *array*: Indices identifying the octree containing the block in the "array-of-octrees".
- *enzo_CellWidth*: Corresponds to the EnzoBlock `CellWidth` parameter.
- *enzo_GridDimension*: Corresponds to the EnzoBlock `GridDimension` parameter.
- *enzo_GridEndIndex*: Corresponds to the EnzoBlock `GridEndIndex` parameter.
- *enzo_GridLeftEdge*: Corresponds to the EnzoBlock `GridLeftEdge` parameter.
- *enzo_GridStartIndex*: Corresponds to the EnzoBlock `GridStartIndex` parameter.
- *enzo_dt*: Corresponds to the EnzoBlock `dt` parameter.
- *enzo_redshift*: Corresponds to the EnzoBlock `redshift` parameter.

Block data

Block data are stored as HDF5 datasets.

Fields are currently stored as arrays of size `(mx,my,mz)`, where `mx`, `my`, and `mz` are the dimensions of the field data *including* ghost data. (Note that future checkpoint versions may only include non-ghost data to reduce disk space.) Dataset names are field names with `"field_` prepended, for example `"field_density"`.

Particles are stored as one-dimensional HDF5 datasets, one dataset per attribute per particle type. Datasets are named using `"particle"` + *particle-type* + *particle attribute*, delimited by underscores. For example, `"particle_dark_vx"` for the x-velocity particle attribute `"vx"` values of the `"dark"` type particles in the block. The length of the arrays equals the number of that type of particle in the block.

5.4 Interpolation Design

5.4.1 Requirements

This document describes incorporating "ENZO interpolation" into Enzo-E; in particular, the "SecondOrderA" method. This method requires an extra layer of coarse-level cells compared to the basic "trilinear interpolation" previously implemented in Enzo-E. Implementing this requires additional communication, since this augmented coarse-grid array overlaps multiple additional blocks.

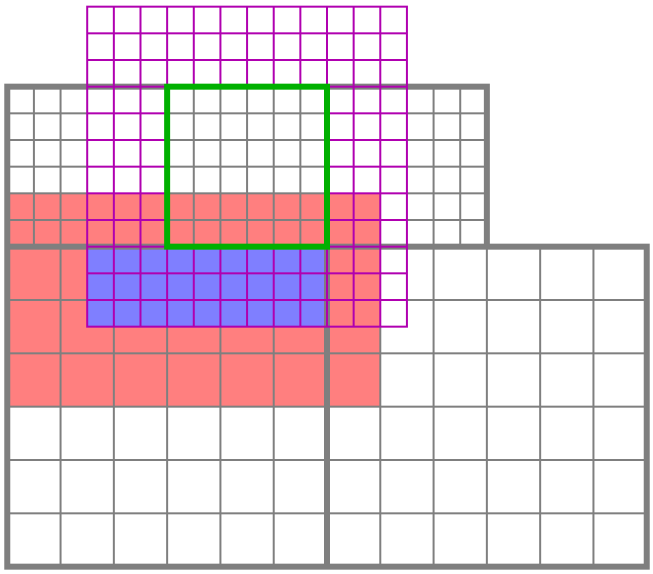


Fig. 4: In ENZO's SecondOrderA interpolation method, multiple additional blocks intersect the extended array of coarse values (red) required for computing the interpolated ghost zone values (blue) for the fine block (green) that intersects a coarse block.

5.4.2 Design

We decompose the operations required according to the role of blocks for each interpolation. Note a given block may participate in multiple roles for different block faces. There are three main block roles involved in an interpolation:

1. **Sending block**
    a. sends its overlapped cell values to the receiver
2. **Extra block**
    a. sends its overlapped cell values to the receiver
3. **Receiving block**

a. receives data from sending and extra blocks

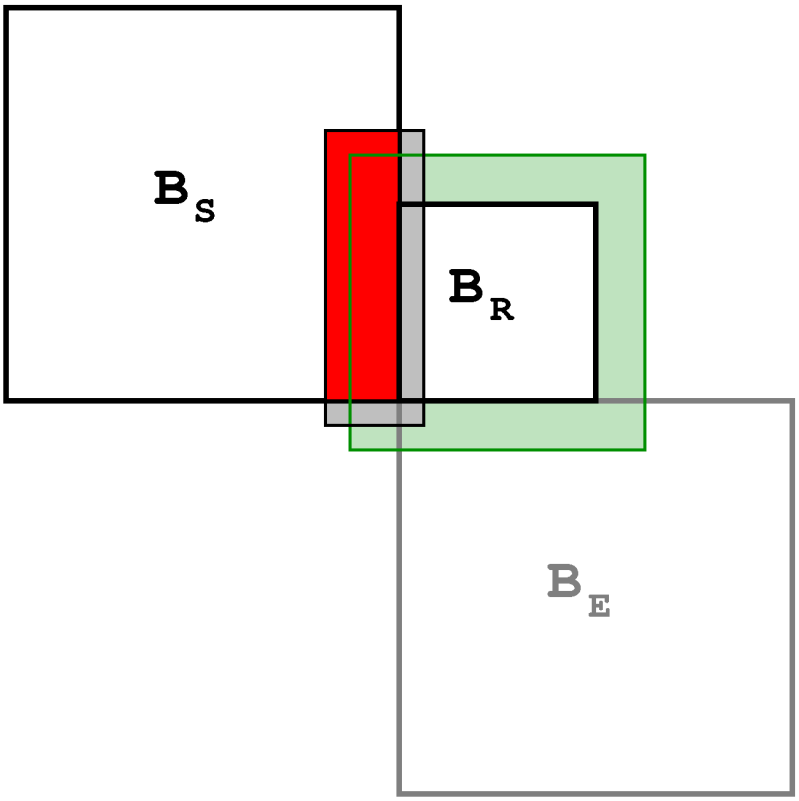
b. copies incoming data to padded array

c. copies its own overlapped cells to the padded array

d. calls the interpolation operation using the padded array to compute the interpolated ghost cell values

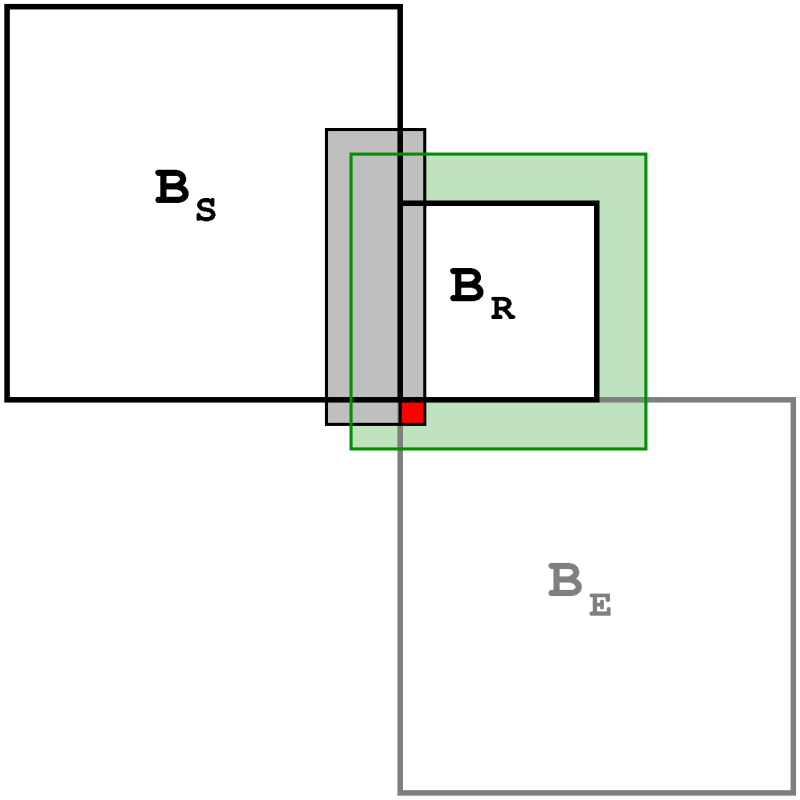
Some minor operations are not included, such as extra blocks that are in the same level as the receiving block need to "coarsen" their values before sending to the receiving block in operation 3.

Sending Block



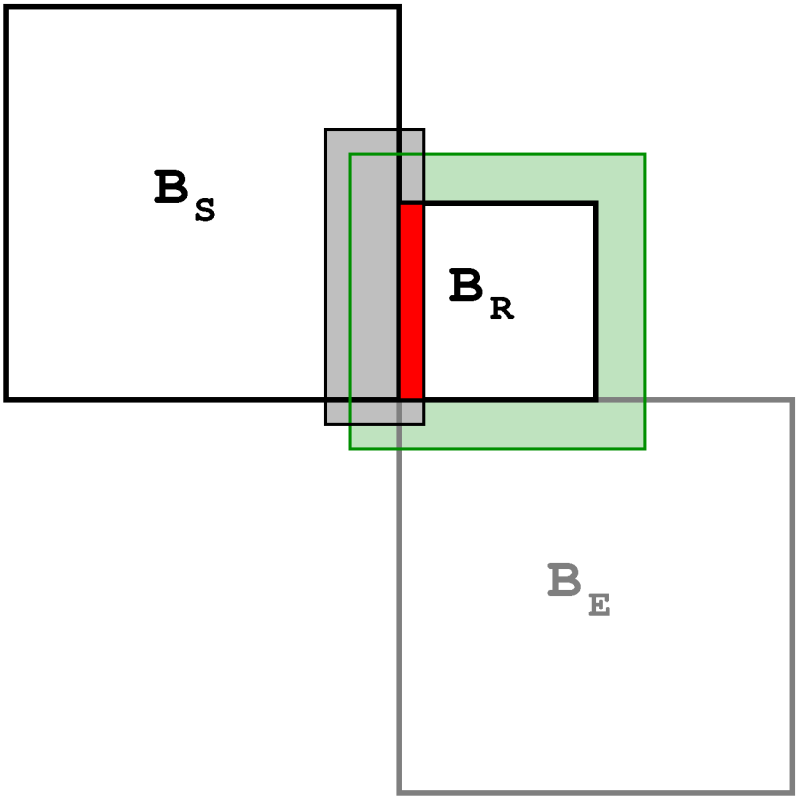
The sending block must send its data to the receiving block, as it does for all other refresh operations. The main difference is it requires an additional layer of padding when sending. This is handled by the method `Block::refresh_load_field_face_()` in `control_refresh.cpp`

Extra Block



In addition to the sending block, some extra blocks must also send their overlapped cell data to the receiving block. This requires an extra loop and some additional logic for a block to determine that it must participate in an interpolation operation not as the sender or receiver. This is handled in the `Block::refresh_load_extra_face_()` method, which can be found in the `control_refresh.cpp` source file. Depending on the relative mesh refinement level, an extra block may be handled

Receiving Block



5.4.3 Milestones

5.4.4 Tasks

5.5 Refresh Design [EMPTY]

CHAPTER

SIX

PROJECT DOCUMENTS

6.1 Project Plan

6.2 Enzo-E / Cello Guiding Principles

Key guiding principles in Enzo-E / Cello design and construction are performance, scalability, usability, reliability, and flexibility.

6.2.1 Performance

Cello enables high single-node performance through the design of its data classes for fields (Eularian) and particles (Lagrangian)

The **Field** class represents arrays of field variables (density, velocity, etc.) that are associated with blocks of the AMR hierarchy. The size of these block-local arrays is specified by the user in Enzo's configuration file, allowing flexibility for optimizing for cache size and parallel task granularity. Cello's field data also supports improving performance through data alignment and data structure padding. *Data alignment*—aligning array elements along 128-bit boundaries—can greatly improve performance of a CPU's SIMD instructions. *Data structure padding*—separating field arrays by some small multiple of the cache line length—can reduce cache conflict and improve cache reuse. Block size, data alignment, and data structure padding are all controllable through Enzo-E's parameter file.

The **Particle** class represents particles associated with blocks in the AMR hierarchy. It can represent multiple particle types, with particles in each type having arbitrary attributes (though a minimal particle type must have a position). For efficiency, particles are allocated and operated on a *batch* at a time, where the number of particles in each batch is specified by the user. Particles within a batch can be stored in two different ways: by particle, or by attribute, which again can affect efficiency. Additionally, storing particles in fixed-sized batches should aid in the implementation of efficient accelerator-based code (CUDA, OpenACC, etc.) for particles.

6.2.2 Scalability

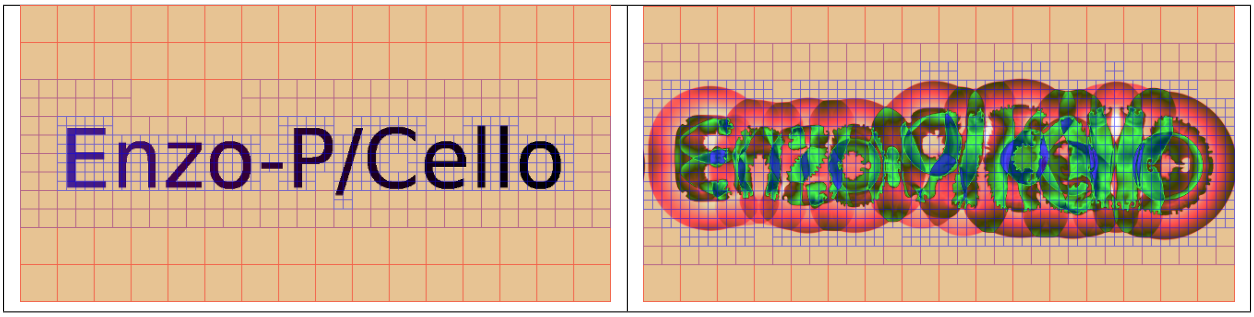
Cello enables high scalability in part through the innovative combination of Charm++ and Cello's array-of-octree AMR. Charm++, the externally-developed parallel programming system on which Cello is built, is a data-driven, asynchronous, programming language based on C++. Charm++ incorporates state-of-the-art dynamic load balancing, its asynchronous execution model is naturally latency-tolerant, and it provides automatic overlap of computation and communication. Emergent scalability issues such as fault-tolerance and energy efficiency are also being researched and incorporated into Charm++, allowing Enzo-E / Cello to benefit automatically as the parallel programming system continues to improve.

The *array-of-octree* AMR approach that is implemented in Cello has been shown to be among the most scalable known AMR approaches to date. The specific AMR remeshing algorithm is based on that developed and prototyped by

the Charm++ group (see Langer, et al). As in their approach, our mesh refinement algorithm proceeds mostly asynchronously, using only a lightweight barrier-like operation provided by the Charm++ runtime system to detect consensus on refinement levels between blocks.

6.2.3 Usability

Table 1: A simple hydrodynamics problem



Cello is designed to be both *user-friendly* and *developer-friendly*. The integrated Enzo-E / Cello structured configuration files are easy to read and write, and allow for much more powerful yet easy-to-understand problem initialization than in Enzo. As an example, the simple test problem whose output is shown above used the combination of a PNG image file created using a paint program, together with the following configuration to initialize the density field:

```
Initial {
   density { value = [ 1.0 + x, "input/density_mask.png", 0.125 ]; }
}
```

This reads naturally as "initialize density with the value 1.0 wherever the input image mask is set (i.e. not black), and 0.125 elsewhere."

Logical expressions such as `x + sin(y) < 0.5` can be used in place of image masks, and multiple masks and logical expressions can be combined to generate arbitrarily complex initial conditions. This capability greatly simplifies the user experience of setting up a test problem, making Enzo-E not only useful for researchers to run scientifically viable simulations, but also for students and educators to learn and experiment with concepts in computational fluid dynamics and astrophysics.

Migrating new functionality from Enzo or other science applications to Enzo-E / Cello is also very developer-friendly. Only the serial code implementing a numerical method on a single grid patch in Enzo is required, and frequently no modifications of the method code are even necessary. The bulk of the migration effort required is writing an EnzoBlock class that contains static variables to replace Enzo's global variables, but this has already been done. Even Enzo's troublesome redefinition of *float* to *double* can be easily handled, by globally replacing *float* with the macro *enzo_float* in the code being migrated.

6.2.4 Reliability

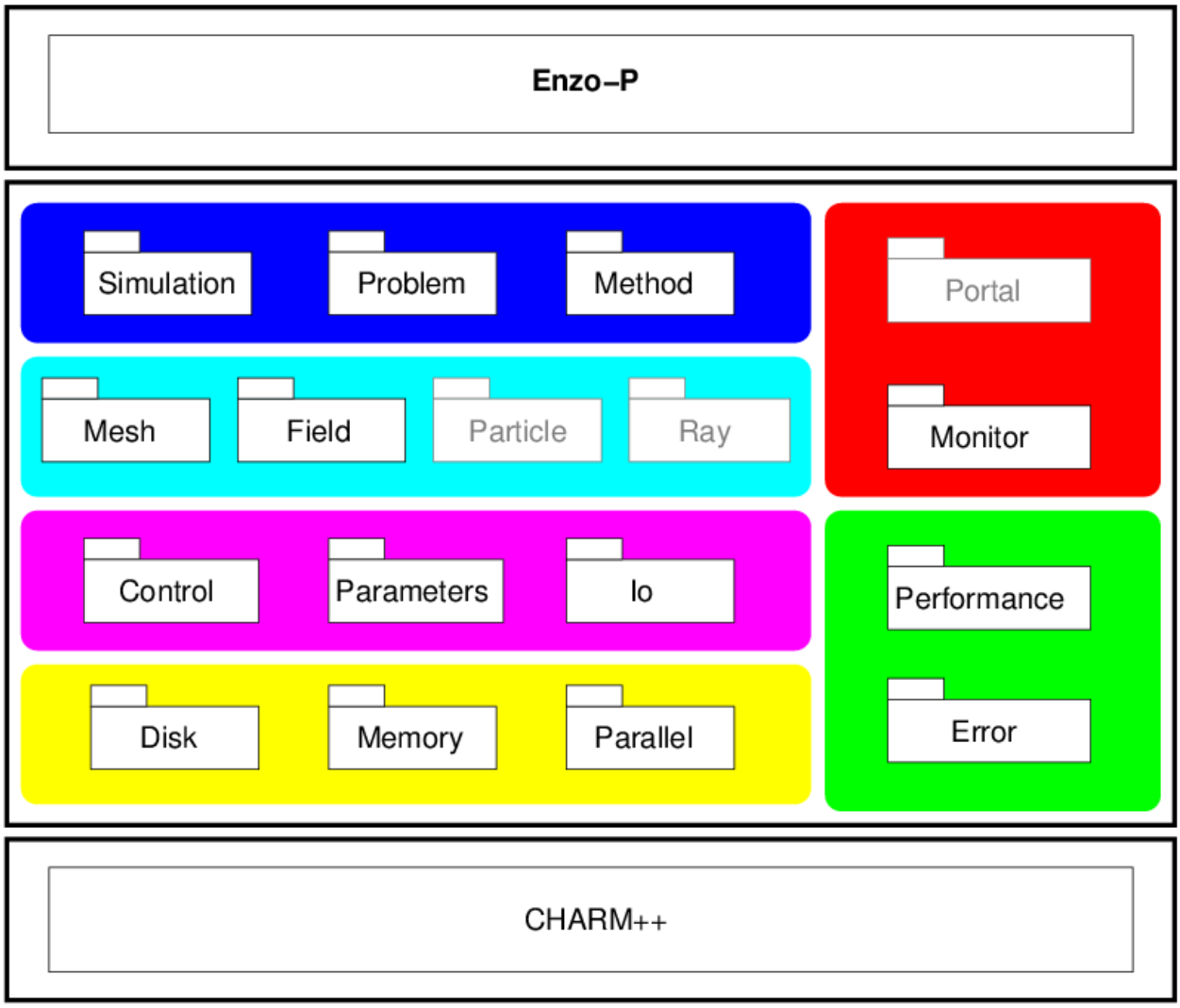
Cello currently contains a large and growing collection of test problems, together with built-in support for both running tests and visualizing results. The running of tests is incorporated into the software build system and invoked by executing the provided *./regression.sh* script, and test results are visualized through a supplied PHP hypertext preprocessor web page. Cello also has support for testing invariants during code execution, helping to identify numerical anomalies and reporting them to the user before they may otherwise make themselves known, which reduces time debugging and leads to overall higher software quality. Defect tracking is coordinated through Bugzilla to ensure that users have a means to report bugs, and that developers have the means to obtain the information required to fix them.

6.2.5 Flexibility

We anticipate new and possibly disruptive developments in hardware, as well as new developments in numerical methods and parallel data structures, so flexibility is a high priority in our design and development. Object-oriented programming provides a natural flexibility through polymorphism, encapsulation, and separation of interface and implementation. Also, we have adopted an agile iterative test-driven development approach with continuous refactoring, which improves flexibility by reducing code complexity and improving code readability.

6.3 Cello Components

The software design of Enzo-E / `Cello` is based on object-oriented programming (OOP), a proven software design and implementation paradigm that helps reduce software complexity and improves code maintainability.



The `Cello` framework is functionally decomposed into a collection of software components. The `Enzo-E` application is implemented on top of Cello, which is comprised of the components within the central boxed region. `Cello` is in turn implemented using the Charm++ parallel programming system. Components in `Cello` are organized into multiple layers, with software dependencies directed primarily downwards; cross-cutting software concerns are also implemented as components. Each component is composed of one or more inter-operating C++ classes.

6.3.1 Enzo-E

`Enzo-E` is the astrophysics and cosmology application built on top of the `Cello` scalable AMR framework. `Enzo-E` interfaces with `Cello` primarily through C++ class inheritance of `Cello` classes in the `Simulation`, `Problem`, `Method`, and `Mesh` components. Due to the separation of concerns between `Enzo-E` and Cello, development of `Enzo-E` requires no knowledge or awareness of parallel programming, and development of `Cello` requires no knowledge of specific numerical methods. This allows relatively independent software development of `Enzo-E` by physical science and numerical methods experts, and of `Cello` by computer science and parallel programming experts.

6.3.2 High-level components

Top level components of `Cello` include `Simulation`, `Problem`, and `Method`. A `Simulation` defines and manages a computational astrophysics `Problem`, which defines physics, method, and data structure `Parameters`, initial conditions, and boundary conditions. The `Simulation` class, which is implemented as a `Charm` *process group*, initializes and begins parallel execution of the simulation. `Method` classes are used to implement individual numerical methods that compute on `Field` (current) data on a block, as well as `Particle` and `Ray` data (proposed).

6.3.3 Data structure components

Data structure components include the existing `Mesh` and `Field` components, and the proposed `Particle` and `Ray` components, for implementing the distributed computational data containers. The `Mesh` component includes classes for representing and operating on an adaptive mesh hierarchy, implemented as a fully-distributed array-of-octrees. Octree nodes are associated with Blocks, which are containers for the `Field` and other data, and implemented as a Charm++ *chare array*.

6.3.4 Middle-level components

Middle-level components include `Control`, `Parameters` and `Io` `Control` handles the time stepping of `Method` s to advance the problem forward in time, as well as sequencing adaptive mesh refinement data structure operations, including remeshing, scheduling dynamic load balancing (which will be delegated to Charm++), and refreshing ghost zones on Block boundaries. The `Parameters` component serves to read, store, and provide access to parameters defined in an input configuration file. To improve usability over Enzo, configuration files are more structured, and support floating-point and logical expressions to greatly simplify initializing problems with complex initial conditions. The `Io` component serves as a layer to coordinate the disk output of data structure components, such as `Simulation` `Hierarchy` and `Field` data. It calls the `Disk` component to handle actual file operations.

6.3.5 Hardware-interface components

The lower-level hardware-interface components include `Disk`, `Memory` and `Parallel`. The `Disk` component implements basic disk operations, isolating the specific file format from the higher-level `Io` component. `Disk` currently supports HDF5, and we propose to support the Adaptable IO System (ADIOS) in the future to enhance transfer of data to and from other HPC software components. The `Memory` component controls dynamic memory allocation and management. Currently `Memory` handles allocating and monitoring heap memory usage; proposed functionality includes allocating, deallocating, and transferring data between main memory, hardware accelerator (GPU) memory, and many-core coprocessors (e.g.~the Intel Xeon Phi). As with the `Disk` component, this serves to isolate lower-level details from higher-level components. The `Parallel` component currently supplies basic access to core rank and core count, and is being depreciated.

6.3.6 Interface components

Interface components include `Monitor` (current) and `Portal` (proposed). The `Monitor` component controls the user-readable summary of progress to stdout, and the proposed `Portal` component will control the interaction of `Enzo-E` with external applications running concurrently, such as inline analysis or real-time visualization. One particular such analysis and visualization application is yt, which we will use to help drive the design and development of the `Portal` component.

6.3.7 Cross-cutting components

Some `Cello` components can in principle be called from any software layer—these include `Performance` and `Error`. The `Performance` component dynamically collects performance data for the running `Enzo-E` simulation, and provides a holistic summary of performance data to the user, as well as to software components that can adapt to optimize desired performance metrics. Current metrics measured include memory usage (via the `Memory` component), and computation amount and memory access amount (via the Performance Application Programming Interface (PAPI). Future support will include metrics for monitoring parallel communication, dynamic load balancing, and disk usage. The `Error` component will be used to detect, evaluate, and decide what to do about software errors; higher-level error detection and recovery will be handled by Charm++, which supports both simple checkpoint to disk, as well as double in-memory checkpoint with automatic restart.

6.4 Gravity Solver Optimisation

This document will outline the results of optimisations on the gravity solver performed by a team of scientists consisting of Dr. John Regan and Dr. Stefan Arridge from Maynooth University and Sophie Wenzel-Teuber and Dr. Buket Benek Gursoy from the Irish Centre for High End Computing (ICHEC, part of NUI Galway).

6.4.1 Summary

We investigated test problems with 16^3, 32^3, 64^3, 256^3 root grid, varying number of blocks and number of PEs, with/without AMR, for around 200 cycles. Furthermore, we analysed the bottleneck of the simulations with a profiler and present a runtime analysis. Specifically, for dark-matter-only cosmology runs, and 256^3 unigrid, the optimal setup was found to be 8^3 blocks, each of size 32^3. The effects of changing parameters such as `min_level` and the number of V-cycles on code performance were also analysed. It was observed that it is unlikely to achieve much speed-up just from changing some parameters in the configuration file. A new feature was implemented in the code to allow refresh operations to have different values for `min_face_rank` and by using a smaller `ghost_depth` a decrease in runtime of 30% could be achieved for a unigrid simulation.

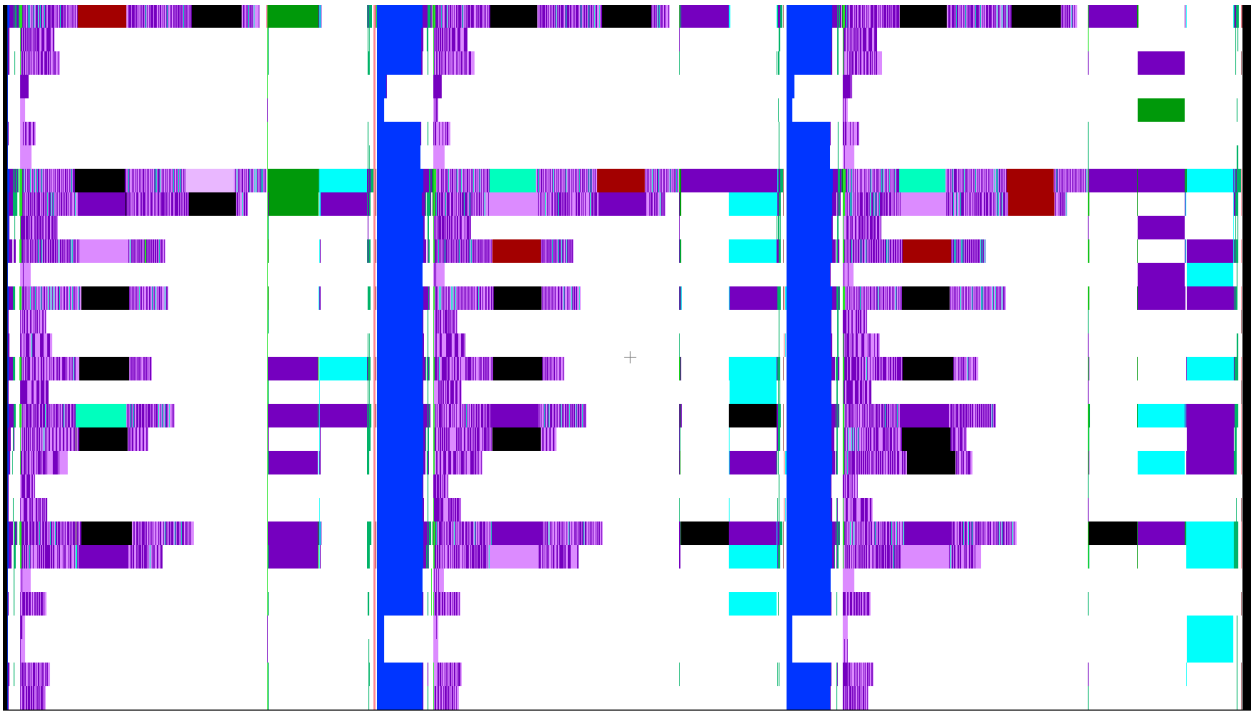
6.4.2 Profiling with Projections

Charm++ has the possibility to gather tracing data of all entry functions and a visualisation tool called Projections.

The tracing data logs which processes executes which entry function at which point in time and for how long. This creates one log file per PE. The size of these log files can be set with the `+logsize` parameter. Whenever this size is exceeded the logs are flushed to disk. This I/O can heavily influence the tracing results as it will appear as a long execution of an entry function in the data.

The Projections visualisation tool provides many possibilities to view this tracing data. The following image is a screenshot from a gravity simulation with 30 PEs on a grid of 64x64x64 cells, divided into 4 blocks per dimension.

All tracing and runtime measurements have been performed on ICHEC's supercomputer on one or two nodes consisting of 40 Intel Skylake processors each and 192GiB (1.5TiB) RAM.



The X axis shows the time of 3 cycles (203-205) and the Y axis has one line per PE of colour codes for the entry functions this PE executed. White signals idle time.

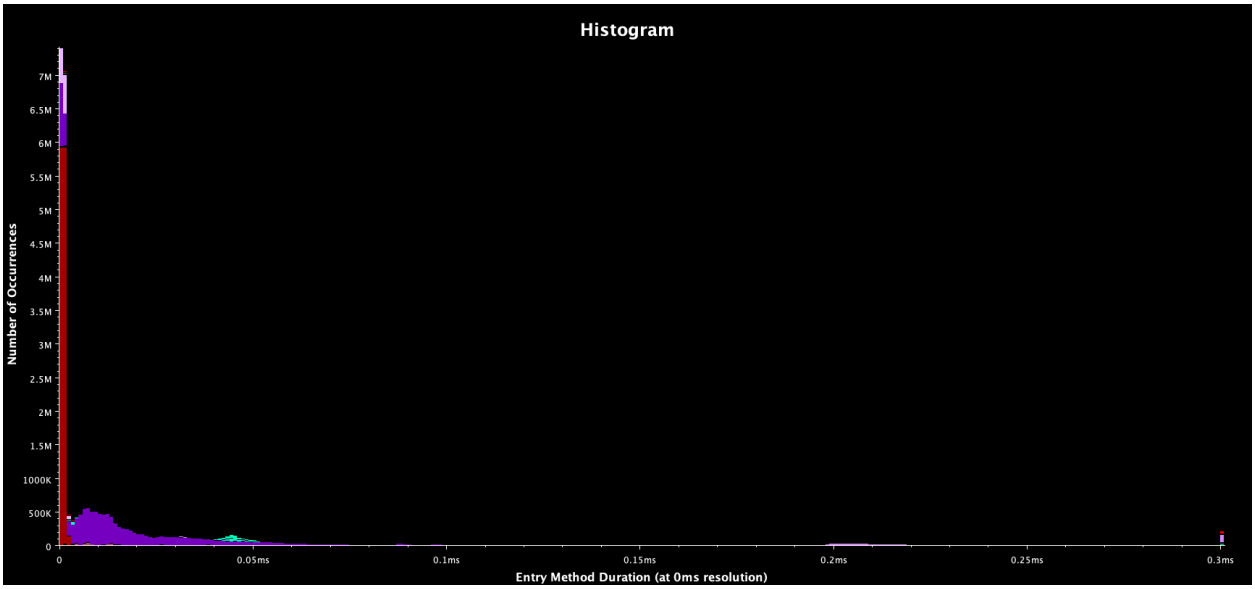
The purple coloured lines are the refresh routine (functions `p_dot_recv_children` in the brighter and `p_new_refresh_recv` in the darker shade).

The blue columns represent the `r_stopping_compute_timestep` function and signal the end of one and beginning of a new timestep. The equal length of this function for almost all PEs suggest that the log data was flushed during the execution of this function. Therefore, the execution time shown in the image is not representable for the real runtime. The same is true for equally sized time spans in other functions throughout the plot.

Especially the black blocks that represent "overhead" probably have to be neglected as well as this overhead almost never appears in plots without the problem of flushing log files to disk.

However, what can safely be said is that the refresh routine plays a vital role for the performance of this Enzo-E simulation.

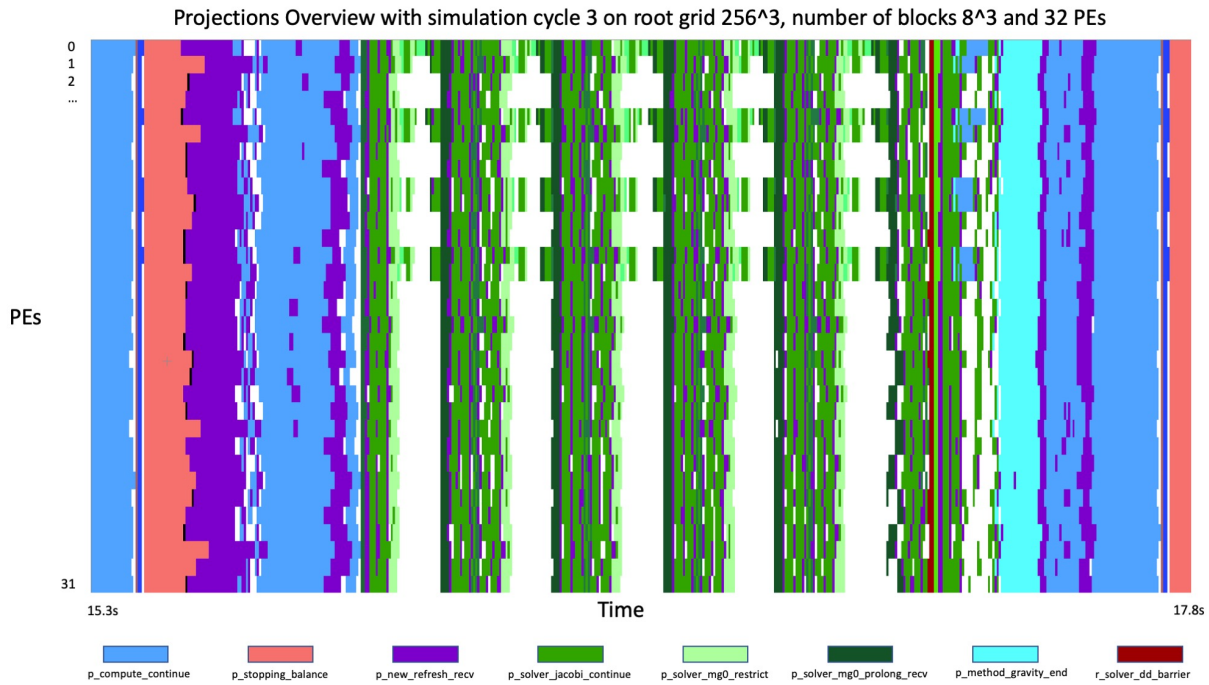
Other plots of Projections show that the refresh routine alone accounts for more than 24 million messages in the 3 cycles above of which most result in an entry function execution time of less than 0.02s.



This image shows a histogram of the execution times of the messages for the entry functions.

Purple coloured are the refresh functions as above. Red represents `p_control_sync_count`, which is also called in this routine to query the refresh state of a blocks neighbours.

When disabling the adaptive mesh refinement and running in a unigrid mode the effect of the refresh routine is much smaller and the stages of the simulation can be identified in the overview.



This simulation was run on a grid of 256 cells cubed, divided into 8 blocks per direction and parallelised over 32 PEs.

The five cycles of the Multigrid solver are clearly distinguishable because all PEs except one are idle. The number of V-cycles can be adjusted in the parameter file and will automatically reduce due to a smaller residual later in the simulation. Less coarsening of the grid such that more than only one processor is solving the problem and therefore reducing the idle time of the other processors is possible by adjusting the `min_level` parameter but results in longer

runtimes (presumably due to communication).

The function `p_compute_continue` also has a large impact on the simulation time and is accountable for starting all solvers that are linked to the simulation.

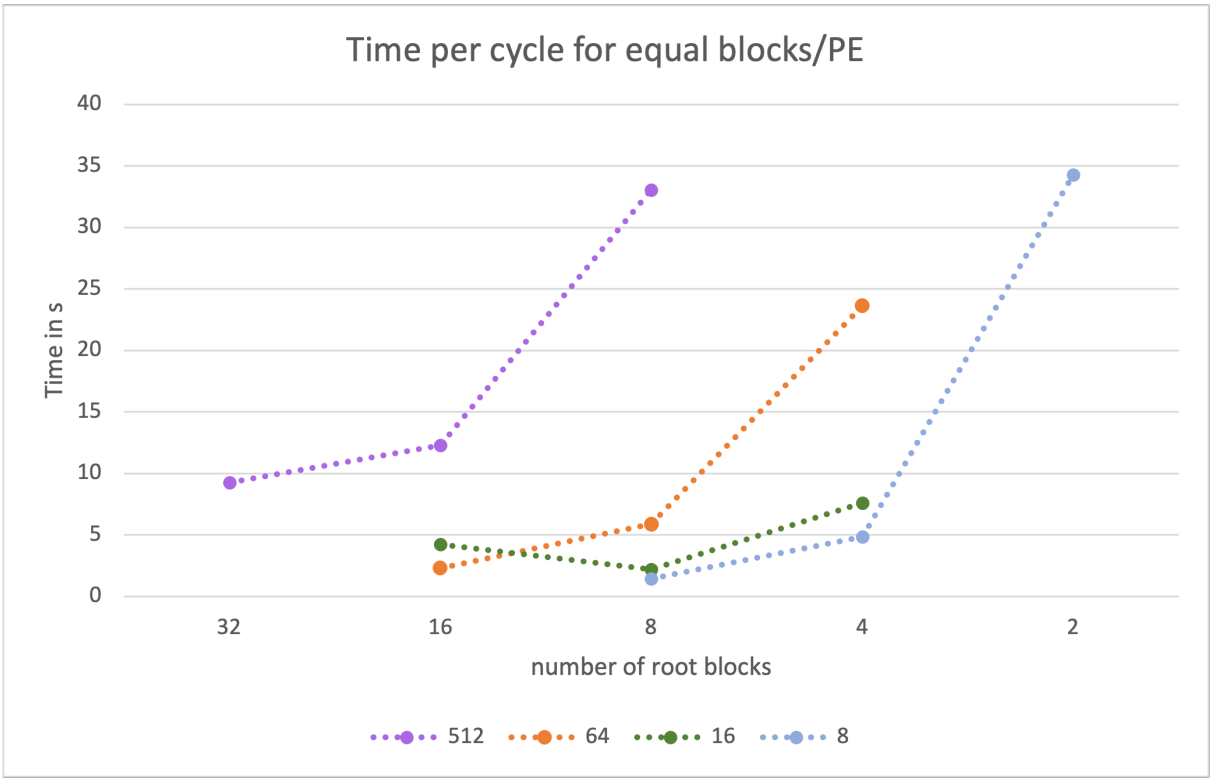
6.4.3 Runtime analysis for unigrid on a single node

To compare parameters and runtime settings in future analysis a good setup has to be found first that can yield the baseline for performance observations.

All combinations of the grid sizes 32, 64 and 256 cubed, the number of blocks of 2, 4, 8, 16, 32 and 64 per direction (where applicable) and the number of PEs of 2, 4, 8, 16, 32 and 64 (also where applicable) were run for a simulation without adaptive mesh refinement for clarity.

This provides a lot of numbers that are difficult to compare. The goal is to find metrics that cover a beneficial balance between a good workload per PE and therefore a performance utilisation of the resources, and a high parallelisation.

What was used is a mixture of weak and strong scaling: The following plot shows the runtime per cycle (`total runtime / number of cycles`) for a simulation size of 256 cubed for four constant numbers of blocks per PE.

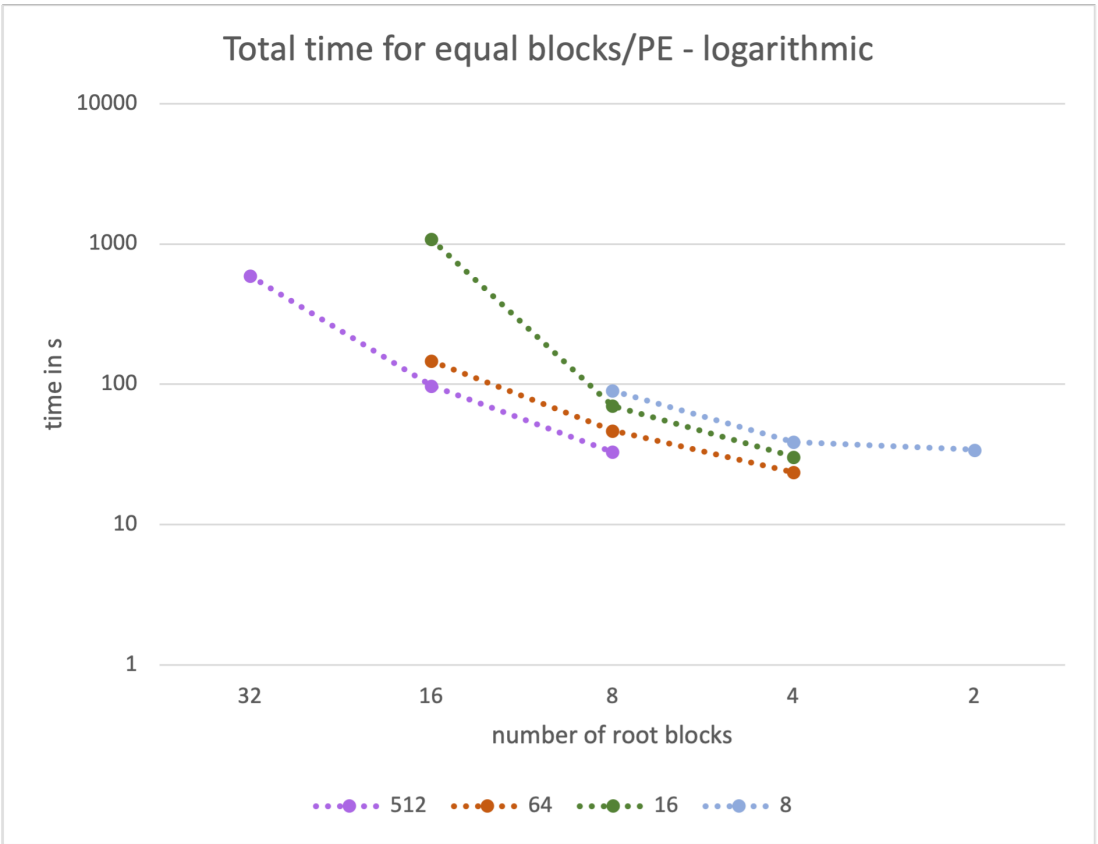


For example for a number of 64 blocks per PE the runtimes were calculated for

- 16^3 blocks on 64PEs
- 8^3 blocks on 8 PEs
- 4^3 blocks on 1 PE.

This means in the graph above from left to right the blocks become larger but with less parallelisation. Therefore, each PE has more to calculate on the right side of the graph than on the left, but less to communicate.

A similar plot can be created for the accumulated time of all PEs to perform one cycle - this is the value from above multiplied by the number of PEs. This relates to the CPU time needed to complete one step of the simulation.



From this graph it is clear that it is more efficient to compute larger blocks on fewer PEs.

Clearly, both representations have to be combined as each one is insufficient for an educated guess of a balanced setup. From this combination we concluded that a good initial setup of the simulation for further tests is using 32PEs and 8 blocks on a grid of 256 cubed cells (which relates to the mid point of the green line in the two plots above).

These metrics fail at smaller simluation sizes for two reasons: The amount of computational work can be viewed as too small compared to the overhead and the parameter space is not sufficient to produce enough results for meaningful graphs of the metrics mentioned above. It is still the case however that fewer but larger blocks and fewer PEs result in less CPU time per cycle (again, due to the reduced overhead for communication).

6.4.4 Refresh routine

As mentioned above the refresh routine is the most limiting factor in a single node AMR run.

How large the ghost zones are that are copied and how many neighbours take part in the exchange is parameterised by two variables.

The `ghost_depth` that determines the number of cells that are copied in a direction was set to 4 in the runs above which is a necessity at the moment. Significant code changes would be required to adapt this value for the distinctive refresh routines started by the different parts of Enzo-E.

Only in a simulation on a uniform grid this value can be decreased. Comparing the runtime of simulations with a ghost depth of 2 and 4 resulted in a decrease from 2.7872s to 2.1578s (speedup of 1.292) for a simulation with the setting determined in the method above (256^3 cells in 8^3 blocks and 32 PEs).

Another parameter are the number of neighbours that exchange their ghost zones. In a three dimensional grid a cell has 26 neighbours. 6 of them share a face, 12 share an edge and 8 only a corner.

Depending on the algorithm only the data from neighbouring faces is used. Therefore, it is enough for the refresh routine to only exchange data from these neighbour blocks.

The variable to determine how many neighbours are used for the refresh is called `min_face_rank`. A small code change allowed for this to be set when constructing refresh objects. For the same simulation parameters as above this resulted in a difference of from 2.0589s to 1.9297s (speedup of 1.067) per cycle for a unigrid and from 89.0964s to 89.5893s per cycle for an AMR run. The runtime of the AMR simulation has been averaged over all 160 cycles even though the simulation the mesh is not refined in the first ~100 cycles. Therefore, these numbers have to be treated with caution.

Combining both approaches is as mentioned only possible in a unigrid simulation but adds up to a decrease in runtime of roughly 30% (calculated from a decrease of the time per cycle from 2.8122s to 1.9586s).

6.4.5 Future Work

While this document summarises successful outcomes on the gravity solver optimisation there is still further work that could potentially be carried out. However, further optimisations are likely to be challenging and would require extensive code changes like implementing a more effective smoother such as red-black Gauss-Seidel, doing multiple Jacobi smoothings per refresh, or implementing full-multigrid method instead of V-cycles. Another potential work would be to investigate an alternative strategy for computing gravitational forces e.g. through the use of 3D parallel FFT.

CHAPTER

SEVEN

TESTING

Enzo-E currently uses two different testing frameworks for its test suite, *ctest* and *pytest*. Both frameworks are run on all pull requests and each has its advantages. You are welcome to use either when creating new tests. The `ctest` framework is used primarily for unit tests (i.e., tests that run from self-contained binaries), but can also be used for tests that run the `enzo-e` binary. The `pytest` framework is mainly useful for answer tests, where results from two versions of an `enzo-e` simulation are compared.

7.1 The ctest Framework

The `ctest` infrastructure in Enzo-e is comprised of unit tests (which test individual functions and functionality) and integration tests which test a more holistic portion of the codebase. In this documentation, we detail the layout of the ctest infrastructure. All input files are located in the input directory and the tests themselves are defined in `test/CMakeLists.txt`

7.1.1 How to Run Tests

To run all the tests currently in the testing infrastructure run `ctest` in the build directory.

Basic parameters to control `ctest` are:

- `ctest -N` to see all available tests
- `ctest -V` to run all tests with verbose output
- `ctest -R someregex` to run all tests matching `someregex`, e.g., `ctest -R heat`
- `ctest -L somelabel` to run all test with label matching `somelabel`. For `Enzo-E` we currently only set `serial` and `parallel` for serial and parallel (i.e., using multiple PEs) tests, respectively

Note, you may need to adjust the parallel launch configuration to your environment for the parallel test. By default, `charmrun` with 4 PEs will be used to launch parallel tests. To use `srun` instead (e.g., for a pure Charm++ MPI build) with 8 ranks you have to set `-DPARALLEL_LAUNCHER=srun -DPARALLEL_LAUNCHER_NPROC_ARG="-n" -DPARALLEL_LAUNCHER_NPROC=8` when configuring your build.

In order to run a test separately from the main `ctest` infrastructure write `bin/enzo-e` followed be the location of the test to be run in the main enzo directory. For example, `bin/enzo-e input/Cosmology/method_cosmology-1.in` runs the method_cosmology-1 test. Some tests are intended to run on multiple processors, for these tests the number of processors must be specified like so `charmrun +p4 bin/enzo-e input/Cosmology/method_cosmology-8.in`, this runs the method_cosmology-8 test on four processors, a test designed to be run on multiple processors in parallel. If charmrun command is not in your path charmrun's path must also be included: `~/Charm/bin/charmrun +p4 bin/enzo-e input/Cosmology/method_cosmology-8.in`.

In order to exclude the `"shu_collapse"` and `"bb_test"` tests (as is done when running the tests on CircleCI, execute the following command: `ctest -E "(shu_collapse)|(bb_test)"`.

7.1.2 How to Analyse the Test Results

By default all tests will be run and the output is stored in the *test* directory of the build.

`ctest` will automatically tell you which tests passed and which tests failed. For a more verbose output of the test, you can call `ctest` with `--output-on-failure`.

If an integration test fails in start-up this implies that there is something wrong with how the test is set up either as an error in the `test/CMakeLists.txt` file that calls the test or that the test has an undefined parameter. If a test fails while running this indicates that the feature being tested does not work.

In order to see what happened during the test, you can look at the output directory of the test, which is located in a subdirectory (named after the test) in the `test` directory of the build directory, for example `method_cosmology-1.in` the test results are stored in `test/MethodCosmology/Cosmology-8`. This directory also contains all outputs (image and data files).

7.2 How to create a new ctest test

7.2.1 Introduction

This document provides instruction for creating tests for use with *The ctest Framework*. To make an answer test, where results of a simulation can be run and compared from two versions of `enzo-e`, it is recommended to use *The pytest Framework*.

When adding a new function to the enzo-e project there is a possibility that it may affect the rest of the code, for this reason new code must be able to pass all existing tests as well as tests built for the new code. This means that integration tests are required. These tests are designed to stress the system. When making a new test these steps must be followed:

- Create input parameters
- Create a new test subdirectory
- Integrate the test into test infrastructure
- Running your new test

7.2.2 Create Input Parameters for the New Test.

Create a new subdirectory for the new tests in the input directory. Create new `.in` file that contains input parameters for testing the new feature. This should be a simple problem such as a 2D implosion problem (See enzo-e/input/Hydro/test_implosion.in). More details on how to set input parameters see *Parameter files* . As an example of a parameter file see load-balance-4.in in the enzo-e/input/Balance directory. This file will be used later on to set the parameters of the test in the running the test section

7.2.3 Integrating the Test into Test Infrastructure

To add a test to the `ctest` infrastructure, you have to add it to the `test/CMakeLists.txt` file. Several different options exit depending on the type of the test and how it is being run.

Direct Enzo-E tests with input file

For tests that only execute the `enzo-e` binary with a given parameter file and determine pass/fail by the exit code, you should use the `setup_test_serial` or `setup_test_parallel` function (in case the test should be run on more than one compute unit). Note that the number of compute units to run the parallel tests is set when the build is configured via the `PARALLEL_LAUNCHER_NPROC` option. Both functions take three arguments: (1) a short test name, (2) the output directory (i.e., the directory within the `build/test` folder where potential output from the test case is being stored), and (3) the input file. A sample test case may look like `setup_test_parallel(Bound-Periodic-2D BoundaryConditions/Periodic-2D input/Boundary/boundary_periodic-2d.in)`.

Enzo-E test with output being analyzed/processed by Python script

For tests that execute `enzo-e` but determine pass/fail by Python postprocessing, e.g., using `yt`. To add this kind of test use either the `setup_test_serial_python` or `setup_test_parallel_python` function. Again these functions have three mandatory arguments: (1) a short test name, (2) the output directory, and (3) the Python script to be called. Note that the Python script now includes the input file(s) that should be called and the exit code of the script is used to determine pass/fail of the entire test. Also note that these two function take additional arguments that are being passed to the Python script. A sample case is `setup_test_serial_python(merge_stars_serial merge_stars/serial "input/merge_stars/run_merge_stars_test.py" "--prec=${PREC_STRING}")`. We strongly encourage you to follow the overall structure used in the existing Python scripts when setting up a new test case.

Unit tests

Unit tests are self-contained test that are build as separate binaries. In order to add these kind of test use the `setup_test_unit` function. Again this function takes three arguments: (1) a short test name, (2) the output directory, and (3) the binary name to be called. A sample case is `setup_test_unit(Array ArrayComponent/Array test_cello_array)`.

Note, several units test are currently commented out as there are linking issues. For more information, see GitHub issue #176.

7.2.4 Running Your New Test

In order to run your new test you can call it directly, e.g., through the name `ctest -R my-short-test-name`.

Note that in order for the test to show up in the build directory (if your build has already been configured), you need to reconfigure the build so that the new test case is being picked up.

To run all the tests use `ctest` in the build directory.

7.3 The pytest Framework

The `pytest` framework is primarily used for running answer tests, where a simulation is run with two versions of `enzo-e` and their results are compared. This is useful for testing problems with no analytical solution or generally verifying that results from commonly run simulations don't drift.

It is also useful in for testing problems that do have analytic solutions (the answer test might quantify how close a simulation result is to the analytic expected solution). While such tests do exist in the ctest-framework, they often involve more boiler-plate code.

pytest is a Python-based framework for detecting and running a series of tests within a source code repository. When running `pytest`, the user can provide a directory in which `pytest` will look for files named *test_*.py* and run all

functions within those files whose names start with "test". `pytest` will run all tests and present a summary of which ones passed and failed. All functions that run without producing an error will be marked as passed.

7.3.1 Installation

`pytest` can be installed with `pip` or `conda`.

```
$ pip install pytest
```

```
$ conda install pytest
```

7.3.2 Answer Testing

Within `enzo-e`, we make use of the TestCase class to define a general `EnzoETest` class that will run a given simulation within a temporary directory and delete that directory once finished. This class and other useful answer testing functionality are located in the source in *test/answer_tests/answer_testing.py*. All answer tests are located in the other files within the *test/answer_tests* directory.

Some other functionality, that may be reused in other unrelated scripts provided in the Enzo-E repository, are provided in the `test_utils` subdirectory.

Running the Answer Test Suite

The answer test suite is run in two stages. First, test answers must be generated from a version of the code known to function correctly. A git tag associated with the main repository marks a changeset for which the code is believed to produce good results. This tag is named `gold-standard-#`. To pull tags from the main repository and see which tags exist, do the following:

```
$ git fetch origin --tags
$ git tag
```

To generate test answers, use the highest numbered gold standard tag.

Configuring the Answer Test Suite

The behavior of the test can be configured by passing command line arguments to `pytest` or by setting environment variables (or by mixing both).

When invoking `pytest`, the command line flags discussed here should be passed **after** the path to the *test/answer_tests* directory has been provided. For the sake of example (the meaning of flags are explained below), one might invoke:

```
$ pytest test/answer_tests  \
         --build-dir ./build \
         --answer-store
```

The following table lists command line flags, and where applicable, the environment variables that they are interchangable with. In cases where both are set, the command line argument is given precedence.

Table 1: Configuring pytest behavior

flag	env var	description
`--build-dir`	N/A	points to the build-directory where the target enzo-e binary was built (that binary has the path: BUILD_DIR/bin/enzo-e). The path to the charmrun launcher will be inferred from the *BUILD_DIR/CMakeCache.txt* file, but can be overwritten by the `--charm` flag or the `CHARM_PATH` environment variable. This precedence was chosen in case a user causes a change to relevant cached build-variables, but have not rebuilt Enzo-E (i.e. *CMakeCache.txt* may not be valid for the binary). When this flag isn't specified, the test infrastructure searches for the enzo-e binary at ENZOE_ROOT/build/bin/enzo-e, but doesn't try to infer charmrun's location from *CMakeCache.txt*.
`--local-dir`	`TEST_RESULTS_DIR`	points to a directory in which answers will be stored/loaded
`--charm`	`CHARM_PATH`	points to the directory in which `charmrun` is located
`--answer-store`	`GENERATE_TEST_RESULTS`	When the command line flag is specified, test results are generated. Otherwise, results are compared against existing results (unless the environment variable is specified). The environment variable can be be set to `"true"` to generate test results or `"false"` to compare with existing results.
`--grackle-input-data-dir`	`GRACKLE_INPUT_DATA_DIR`	points to the directory where `Grackle` input files are installed. If not specified, then all tests involving `Grackle` will be skipped.

Earlier versions of the tests also required the `"USE_DOUBLE"` environment variable to be set to `"true"` or `"false"` to indicate whether the code had been compiled in double or single precision.

```
$ export TEST_RESULTS_DIR=~/enzoe_tests
$ export CHARM_PATH=~/local/charm-v7.0.0/bin
```

Generating Test Answers

First, check out the highest numbered gold standard tag and compile `enzo-e`.

```
# in the future, you will need to subsitute 004 for a higher number
$ git checkout gold-standard-004
$ ...compile enzo-e
```

Then, run the test suite by calling `pytest` with the answer test directory (make sure to configure behavior correctly with command-line arguments or environment variables). In the following snippet, we assume you are currently at the root of the Enzo-E repository and that you will replace `<build-dir>` with the directory where you build enzo-e (this is commonly `./build`)

```
$ pytest test/answer_tests --local-dir=~/enzoe_tests --build-dir=<build-dir> --answer-
↪store
========================== test session starts ==========================
platform linux -- Python 3.9.13, pytest-7.1.2, pluggy-1.0.0
rootdir: /home/circleci/enzo-e
collected 1 item

test/answer_tests/test_vlct.py .                                    [100%]
```

(continues on next page)

(continued from previous page)

```
========================== 1 passed in 13.26s ==========================
```

Assuming there are no errors, this will run the simulations associated with the tests, perform the analysis required to produce the answers, save the answers to files, and report that all tests have passed.

Comparing Test Answers

Once test answers have been generated, the above steps need not be repeated until the gold standard tag has been updated. Now, any later version of the code can be run with the test suite to check for problems. To configure the test suite to compare with existing answers, omit the `--answer-store` flag and ensure that the `GENERATE_TEST_RESULTS` variable is either unset or set to `"false"`.

```
$ git checkout main
$ ...compile enzo-e
$ pytest test/answer_tests --local-dir=~/enzoe_tests --build-dir=<build-dir>
```

Helpful Tips

By default, most output printed by `enzo-e` or the test scripts will be swallowed by `pytest`. When tests fail, the Python traceback may be shown, but not much else. There are various flags to increase the verbosity of `pytest`, but the `-s` flag will show all output, including from the simulation itself. The `enzo-e` answer test suite will also print out the values of all configuration variables when this flag is given.

```
$ pytest -s test/answer_tests # other args...
```

When debugging an issue it's sometimes helpful to force pytest to run a subset of tests. This can be accomplished with the `-k` flag. For example, to only run a subset of tests with `"grackle"` in the test name, one might execute

```
$ pytest test/answer_tests -k "grackle" # other args...
```

When investigating a failing test or prototyping a brand-new test, it can sometimes be helpful to run the tests against multiple versions of enzo-e. Rather than rebuilding Enzo-E each time you want to do that, you can instead build the different versions of Enzo-E in separate build-directories, and direct `pytest` to use the different builds with the `--build-dir` flag.

Creating New Answer Tests

This section follows the example of `TestHLLCCloud` in *test/answer_tests/test_vlct.py*. Answer tests can be created by making a new Python file in the *test/answer_tests* directory with a name starting with 'test_' or by adding to an existing file if the test falls within the theme given by its name. If your test requires configuring a new simulation parameter file, see *Create Input Parameters for the New Test.* for information on setting that up.

The answer testing framework exists in *test/answer_tests/answer_testing.py*. New test files created in the same directory can directly import from this file.

Creating a New Test Class

To make a new test, one must create a new Python class that subclasses the `EnzoETest` class. Three attributes must be defined within the class:

- parameter_file: the relative path to the simulation parameter file from within the input directory.
- max_runtime: the maximum runtime of the simulation in seconds. The simulation will be stopped and the test marked as failed if this is exceeded. Set this to something a bit longer than the typical runtime to detect when new changes have significantly altered the runtime. If not given, the max runtime is infinity.
- ncpus: the number of processes with which to run the simulation.

```
from answer_testing import EnzoETest

class TestHLLCCloud(EnzoETest):
    parameter_file = "vlct/dual_energy_cloud/hllc_cloud.in"
    max_runtime = 30
    ncpus = 1
```

Tests involving `Grackle`

If the class is associated with a test simulation that invokes `Grackle`, you need to annotate the class declaration with the `uses_grackle` decorator.

```
from answer_testing import EnzoETest, uses_grackle

@uses_grackle
class TestGrackleGeneral(EnzoETest):
    ...
```

For all classes annotated with this decorator:

- the framework knows that it must make symbolic links to all files in the directory run by `GRACKLE_INPUT_DATA_DIR` before it runs the simulation associated with this class.
- the testing framwork also knows to skip the associated test(s) if the `GRACKLE_INPUT_DATA_DIR` environment variable is unset.

If you forget to add this label, `Enzo-E` will not be able to locate the data file needed for Grackle (in a portable way). Thus, the associated simulation and test will fail.

Creating the Test Function

The code above configures the simulation associated with the test. The next step is to write a function which will be run after the simulation completes successfully. This is done by creating a class method within the test class. This function should only take the argument `self` (because it's a class method) and nothing else. The function will be run from within the directory where the simulation was run, so it will be able to load any files that were output.

```
def test_hllc_cloud(self):
    fn = "hllc_cloud_0.0625/hllc_cloud_0.0625.block_list"
    assert os.path.exists(fn)
```

Tests are typically implemented with an `assert` or related statement. In the above example, we check for the existence of a file that should have been created by the simulation. This is not specifically an answer test as we are not comparing

with results from another version of the code. However, these sorts of assertion checks can be included in your test function if they are useful for verifying proper running of the code.

Creating an Answer Test Function

To create an answer test that will automatically save data to files and compare with other files, we make use of the `ytdataset_test` Python decorator, also located in *test/answer_tests/answer_testing.py*.

```
from answer_testing import \
    EnzoETest, \
    ytdataset_test, \
    assert_array_rel_equal
```

We also import an assertion function that will check for relative closeness of values in an array.

The `ytdataset_test` decorator can then be put immediately above the definition of a test function. This wraps the test function in additional code that will save test files and run comparisons. With the `ytdataset_test`, one must also provide a function that will perform the comparison of results.

```
@ytdataset_test(assert_array_rel_equal, decimals=8)
def test_hllc_cloud(self):
    ds = yt.load("hllc_cloud_0.0625/hllc_cloud_0.0625.block_list")
    ad = ds.all_data()

    wfield = ("gas", "mass")
    data = {field[1]: ad.quantities.weighted_standard_deviation(field, wfield)
             for field in ds.field_list}

    return data
```

When using `ytdataset_test` decorator, **a test function must return a dictionary of values.** The values in the dictionary can be anything, e.g., numbers, string, arrays, etc. In the above example, we load a snapshot with `yt` and compute the weighted average and standard deviation (the `weighted_standard_deviation` function returns both) of all the fields on disk. We now only need to return that and the `ytdataset_test` wrapper will save a file named after the test function (in this case, 'test_hllc_cloud.h5' and will use the `assert_array_rel_equal` function to check that results agree to within 8 decimal places. Note, the NumPy testing module defines several other assertion functions which may be useful.

Including Additional Configuration Options

The easiest way to communicate additional configuration options is through environment variables. Once an environment variable is set (i.e., with `export` in bash), it can be seen by your test using the `os.environ` dict. Below, we use the USE_DOUBLE environment variable to determine whether `enzo-e` was compiled in single or double precision, and adjust the tolerance on the tests accordingly.

```
import os

use_double = os.environ.get("USE_DOUBLE", "false").lower() == "true"
if use_double:
    decimals = 12
else:
    decimals = 6
```

(continues on next page)

(continued from previous page)

```
# inside the TestHLLCCloud class
@ytdataset_test(assert_array_rel_equal, decimals=decimals)
def test_hllc_cloud(self):
    ...
```

Note: The above code is primarily for the sake of example. In practice, we now automatically detect the code's precision from the enzo-e executable.

Alternatively, additional configuration options can be configured through new command-line flags, which are introduced and parsed by the *conftest.py* file in the *answer_test* directory. This is generally more robust than adding environment variables (since the flags are more easily discovered and are more explicit). But, in practice it's made slightly more complicated by the fact that flags are parsed with pytest hooks. Flags added in this way work best with `pytest` fixtures, while our tests mostly leverage features from Python's unittest module.

Caveats

Below are a few things to keep in mind when designing new tests.

Defining Multiple Test Functions within a Class

Multiple test functions can be implemented within the same answer test class. However, the test simulation will be run **for each test**. If you want to perform multiple checks on a long running simulation, it is a better idea to implement them all with separate asserts inside a single function.

Answer Test Functions Must Have Unique Names

Answer test functions that use the `ytdataset_test` wrapper must all have unique names. This is because each results file will be named with the name of the function itself.

7.4 Existing ctest Tests

Currently, the Enzo-e `ctest` infrastructure includes:

7.4.1 Accretion Tests

Six tests of the enzo-e "accretion" method.

- the threshold accretion serial test
- the threshold accretion parallel test
- the Bondi-Hoyle accretion serial test
- the Bondi-Hoyle accretion parallel test
- the flux accretion serial test

- the flux accretion parallel test

  All the files necessary for running the tests can be found in input/accretion.

Each test involves setting up initial conditions with a sink particle with some initial mass, position, and velocity, in a background medium of gas with a uniform density and pressure. In the "threshold" and "Bondi-Hoyle" tests, the gas has an initial velocity of zero everywhere, and inthe "flux" tests, the gas in every cell has a velocity with a uniform magnitude, directed towards the initial position of the sink particle. In the "Bondi-Hoyle" tests, the parameters are given values so that the initial Bondi-Hoyle radius is approximately equal to one cell width.

For the serial tests, Enzo-E runs in serial mode.

For the parallel tests, Enzo-E is run in parallel on four PEs.

The methods used are "mhd_vlct", "pm update", "merge sinks", and "accretion".

The python script input/accretion/mass_momentum_conservation.py tests if mass and momentum are conserved across all the output snapshots, to within some tolerance, which depends on the floating-point precision with which Enzo-E was compiled (1.0e-6 for single precision, 1.0e-12 for double precision). If all quantities are indeed conserved, the test passes, if not, the test fails.

These tests can be executed by running "ctest -R accretion" in the build directory. See input/accretion/README for further information on these tests.

7.4.2 BB Test

Runs the "BB Test" problem as described in Federrath et al 2010, ApJ, 713, 269, and tests for mass conservation.

The initial conditions include isothermal gas, with the gas having a constant small "external density" outside of the truncation radius. Within the truncation radius, the gas density has the following form:

$\rho(\phi) = \rho_0(1 + A\cos(2\phi)),$

where ρ is the gas density, ϕ is the azimuthal angle in the spherical polar coordinate system, ρ_0 is the mean density and A is the (small) fluctuation amplitude. In addition, the gas rotates around the z-axis as a solid-body.

When the problem is left to run, the gas collapses into a disk, and sink particles form near the center, which accrete gas and merge together.

This test can be executed by running "ctest -R bb_test" in the build directory.

All the files required for running the test can be found in input/bb_test. See input/bb_test/README for further information.

7.4.3 Array Test

Under construction

test_cello_array

Under construction

7.4.4 Error Test

Under construction

test_error

Under construction

7.4.5 Memory Test

Under construction

test_memory

Under construction

7.4.6 Monitor Test

Under construction

test_monitor

Under construction

7.4.7 Adapt Tests

Tests run 2D implosion problem on a slope mesh refinement with square initial values at different `max_level`. This specifies the most highly defined block in the mesh hierarchy.

adapt-L5-P1

tests adapt at `max_level=5`

7.4.8 Boundary Condition Tests

Runs an implosion problem on various boundary conditions in 2D and 3D.

boundary_outflow-2d

Tests boundary `type=outflow` in a 2D enviroment

boundary_outflow-3d

Tests boundary `type=outflow` in a 3D enviroment

boundary_periodic-2d

Tests boundary `type=periodic` in a 2D enviroment

boundary_periodic-3d

Tests boundary `type=periodic` in a 3D enviroment

boundary_reflecting-2d

Tests boundary `type=reflecting` in a 2D enviroment

boundary_reflecting-3d

Tests boundary `type=reflecting` in a 3D enviroment

7.4.9 FluxCorrect tests

Tests for FluxCorrect method. This currently runs a 3D simulation with a pure-hydro inclined entropy/contact wave and checks how well the various fields are conserved.

inclined_contact_smr_ppm

Integrates the entropy/contact wave using the PPM method. This effectively uses static mesh refinement.

inclined_contact_smr_vl

Integrates the entropy/contact wave using the VL+CT method in hydro mode. This effectively uses static mesh refinement.

7.4.10 Gravity Tests

Tests for gravity methods. Runs gravity method using cg solver. Outputs mesh, phi, rho, x acceleration, y acceleration image files.

method_gravity_cg-1

Tests `EnzoMethodGravityCg` at P=1 in 2D.

method_gravity_cg-8

Tests `EnzoMethodGravityCg` at P=8 in 2D.

7.4.11 Heat Tests

Tests for heat methods

method_heat-1

2D Heat Diffusion problem with P=1

method_heat-8

2D Heat Diffusion problem with P=8

7.4.12 HelloWorld

Tests of full code to demonstrate functionality. These are done as a demostration of enzo's capabilities.

initial_png

Evolves high-pressure region Hello World using Cello.png mask file

7.4.13 Initial MUSIC Tests

Tests for reading MUSIC HDF5 initial conditions at different mesh root_blocks

Solves simple cosmology problem using initial conditions from MUSIC HDF5 initial condition geneerator

initial_music-111

Tests reading MUSIC HDF5 initial conditions at `mesh` *root_blocks = [1,1,1]*

initial_music-112

Tests reading MUSIC HDF5 initial conditions at `mesh` *root_blocks = [1,1,2]*

initial_music-114

Tests reading MUSIC HDF5 initial conditions at `mesh` *root_blocks = [1,1,4]*

initial_music-121

Tests reading MUSIC HDF5 initial conditions at `mesh` *root_blocks = [1,2,1]*

initial_music-141

Tests reading MUSIC HDF5 initial conditions at `mesh` *root_blocks = [1,4,1]*

initial_music-211

Tests reading MUSIC HDF5 initial conditions at `mesh` *root_blocks = [2,1,1]*

initial_music-222

Tests reading MUSIC HDF5 initial conditions at `mesh` *root_blocks = [2,2,2]*

initial_music-411

Tests reading MUSIC HDF5 initial conditions at `mesh` *root_blocks = [4,1,1]*

initial_music-444

Tests reading MUSIC HDF5 initial conditions at `mesh` *root_blocks = [4,4,4]*

7.4.14 Merge Sinks Tests

Four tests of the enzo-e "merge sinks" method.

- the stationary serial test
- the stationary parallel test
- the drifting serial test
- the stationary parallel test

All the files necessary for running the tests can be found in input/merge_sinks.

Each test involves setting up initial conditions with 1000 sink particles randomly distributed in a sphere, with velocities directed towards the centre of the sphere. In the "drifting" tests, an additional uniform velocity is added to all the particles, and in the "stationary" tests, there is no additional velocity.

For the serial tests, Enzo-E runs in serial mode.

For the parallel tests, Enzo-E is run in parallel on four PEs.

The methods used are "pm update" and "merge sinks", so that the particles all move with constant velocity until they are merged together.

The python script input/merge_sinks/mass_momentum_conservation.py then tests if mass and momentum are conserved across all the output snapshots, to within some tolerance, which depends on the precision specified by the $CELLO_PREC environment variable (1.0e-4 for single precision, 1.0e-6 for double precision). If all quantities are indeed conserved, the test passes, if not, the test fails. See input/merge_sinks/README for further details on running the tests.

These tests can be executed by running "ctest -R merge_sinks_*" in the build directory. See input/merge_sinks/README for further information on these tests.

7.4.15 Output Tests

Tests checking for correct output

output-header

Under construction

output-stride-1

Runs a simple 2D implosion. Tests output stride where `stride_write = 1`. This means testing where specifying that exactly one physical processor is used to output data.

output-stride-2

Similar to above test but on two physical processors where `stride_write = 2`

output-stride-4

Similar to above test but on four physical processors where `stride_write = 4`

7.4.16 Particle Tests

Tests on particle motion. Uses tracer particles.

test_particle-x

Tests particle motion in x direction only

test_particle-y

Tests particle motion in y direction only

test_particle-xy

Tests particle motion in x and y directions together

test_particle-circle

Tests circular particle motion

test_particle-amr-static

Tests static particle in adaptive mesh refinement

test_particle-amr-dynamic

Tests dynamic particle in adaptive mesh refinement

7.4.17 PPM tests

Tests for PPM method. Runs a 2D implosion problem using ppm method and outputs density image file and a density data file.

method_ppm-1

Tests PPM method at P=1

method_ppm-8

Tests PPM method at P=8

7.4.18 Shu Collapse Test

Runs the Shu Collapse problem as described in Federrath et al 2010, ApJ, 713, 269, and tests for mass conservation.

The initial conditions include static isothermal gas with an inverse square density profile within a truncation radius, with the gas having a constant small "external density" outside of the truncation radius. When the problem is left to run, a sink particle forms at the center, and the gas collapses towards the center to be accreted by the sink particle.

This test can be executed by running "ctest -R shu_collapse" in the build directory.

All the files required for running the test can be found in input/shu_collapse. See input/shu_collapse/README for further information.

7.4.19 VLCT Tests

Under construction

dual_energy_cloud

Under construction

dual_energy_shock_tube

Under construction

HD_linear_wave

Under construction

MHD_linear_wave

Under construction

MHD_shock_tube

Under construction

passive_advect_sound_wave

Under construction

7.5 Existing Test Simulations

Currently, Enzo-e has the following test simulations in the input folder:

7.5.1 Accretion Tests

Six tests of the enzo-e "accretion" method.

- the threshold accretion serial test
- the threshold accretion parallel test
- the Bondi-Hoyle accretion serial test
- the Bondi-Hoyle accretion parallel test
- the flux accretion serial test
- the flux accretion parallel test

  All the files necessary for running the tests can be found in input/accretion.

Each test involves setting up initial conditions with a sink particle with some initial mass, position, and velocity, in a background medium of gas with a uniform density and pressure. In the "threshold" and "Bondi-Hoyle" tests, the gas has an initial velocity of zero everywhere, and inthe "flux" tests, the gas in every cell has a velocity with a uniform magnitude, directed towards the initial position of the sink particle. In the "Bondi-Hoyle" tests, the parameters are given values so that the initial Bondi-Hoyle radius is approximately equal to one cell width.

For the serial tests, Enzo-E runs in serial mode.

For the parallel tests, Enzo-E is run in parallel on four PEs.

The methods used are "mhd_vlct", "pm update", "merge sinks", and "accretion".

The python script input/accretion/mass_momentum_conservation.py tests if mass and momentum are conserved across all the output snapshots, to within some tolerance, which depends on the floating-point precision with which Enzo-E was compiled (1.0e-6 for single precision, 1.0e-12 for double precision). If all quantities are indeed conserved, the test passes, if not, the test fails.

These tests can be executed by running "ctest -R accretion" in the build directory. See input/accretion/README for further information on these tests.

7.5.2 Adapt Tests

Tests run 2D implosion problem on a slope mesh refinement with square initial values at different `max_level`. This specifies the most highly defined block in the mesh hierarchy.

adapt-L0-P1

tests adapt at `max_level=0`

adapt-L1-P1

tests adapt at `max_level=1`

adapt-L2-P1

tests adapt at `max_level=2`

adapt-L3-P1

tests adapt at `max_level=3`

adapt-L4-P1

tests adapt at `max_level=4`

adapt-L5-P1

tests adapt at `max_level=5`

7.5.3 Balance Tests

load-balance-4

Runs 2D implosion problem on slope adaptive mesh. Tests the load balancing on different processors. Outputs two images.

mesh-balanced

Under construction

mesh-unbalanced

Under construction

7.5.4 BB Test

Runs the "BB Test" problem as described in Federrath et al 2010, ApJ, 713, 269, and tests for mass conservation.

The initial conditions include isothermal gas, with the gas having a constant small "external density" outside of the truncation radius. Within the truncation radius, the gas density has the following form:

$\rho(\phi) = \rho_0(1 + A\cos(2\phi)),$

where ρ is the gas density, ϕ is the azimuthal angle in the spherical polar coordinate system, ρ_0 is the mean density and A is the (small) fluctuation amplitude. In addition, the gas rotates around the z-axis as a solid-body.

When the problem is left to run, the gas collapses into a disk, and sink particles form near the center, which accrete gas and merge together.

This test can be executed by running "ctest -R bb_test" in the build directory.

All the files required for running the test can be found in input/bb_test. See input/bb_test/README for further information.

7.5.5 Boundary Condition Tests

Runs an implosion problem on various boundary conditions in 2D and 3D.

boundary_mixed-2d

Tests boundary set to list of different boundary condition subgroups in a 2D enviroment. Boundary set to periodic on the x-axis and reflecting on the y-axis.

boundary_outflow-2d

Tests boundary `type=outflow` in a 2D enviroment

boundary_outflow-3d

Tests boundary `type=outflow` in a 3D enviroment

boundary_periodic-2d

Tests boundary `type=periodic` in a 2D enviroment

boundary_periodic-3d

Tests boundary `type=periodic` in a 3D enviroment

boundary_reflecting-2d

Tests boundary `type=reflecting` in a 2D enviroment

boundary_reflecting-3d

Tests boundary `type=reflecting` in a 3D enviroment

7.5.6 Checkpoint Tests

Tests for restart checkpoints. Runs 2D implosion using PPM method with slope mesh refinement.

checkpoint_ppm-1

Tests checkpoint/restart for serial run methods

checkpoint_ppm-8

Tests checkpoint/restart for parallel run methods

checkpoint_boundary

Under construction

checkpoint_grackle

Under construction

checkpoint_vlct

Under construction

7.5.7 Collapse Tests

Tests of Collapse methods

collapse-pm-2d

Tests 2D DM collapse problem using particle mesh method. Uses gravity solver type bicgstab. Outputs nine image files and one data file.

7.5.8 Cosmology Tests

Tests of the enzo-e cosmology methods using simple cosmology problems.

cosmo-dd-multispecies-short

A simple test problem that runs the PPM, Cosmology, Gravity (with DD-solver), and Grackle (with multiple species) for a little more than 100 cycles (so that the domain has the opportunity for refinement).

method_cosmology-1

Runs PPM and Cosmology methods with cosmology variables set in the Physics parameter. This outputs image files for: density field, x velocity, y velocity, z velocity, x acceleration, y acceleration, z acceleration, total density and potential image files. It also outputs an hdf5 data file.

This is a deprecated problem that will be removed in the near future.

method_cosmology-8

This is the same as the preeceding problem, but uses 8 root blocks instead of 1.

This is a deprecated problem that will be removed in the near future.

7.5.9 FluxCorrect tests

Tests for FluxCorrect method. This currently runs a 3D simulation with a pure-hydro inclined entropy/contact wave and checks how well the various fields are conserved.

inclined_contact_smr_ppm

Integrates the entropy/contact wave using the PPM method. This effectively uses static mesh refinement.

inclined_contact_smr_vl

Integrates the entropy/contact wave using the VL+CT method in hydro mode. This effectively uses static mesh refinement.

7.5.10 Grackle Tests

Tests for the method that invokes Grackle. These tests set up a cooling test without hydrodynamics to run many one-zone models in Grackle, fully sampling the density, temperature, and metallicity parameter space over which the chemistry and cooling/heating tables are valid.

method_grackle_general

This test compares the summary statistics computed for several grackle fields after a certain period of time to previously archived values.

The simulation timesteps are much larger that the cooling/heating. This makes it more likely that separate processing elements will execute grackle routines at the same time (thus increasing the chances of exposing hypothetical problems related to Grackle & SMP mode).

This test is somewhat fragile given that upgrading Grackle versions could conceivably alter the field values. In the future it would be better to replace this with a test that:

1. checks out and builds a previous commit of Enzo-E
2. runs the simulation and saves the exact field values after running the simulations
3. checks out and builds a newer commit of Enzo-E (while leaving the build of Grackle unchanged)
4. runs the simulation and confirms that the Grackle related field values are identical to the field values from the earlier simulation.

method_grackle_cooling_dt

This test runs Grackle for a fixed number of cycles, and compares the final simulation time to a reference value. Each simulation timestep is set fraction of the minimum magnitude of the cooling/heating timestep.

7.5.11 Gravity Tests

Tests for gravity methods.

Completion-Time Tests

Runs gravity method using cg solver. Outputs mesh, phi, rho, x acceleration, y acceleration image files. Tests succeed or fail based on the final completion time of the simulation.

method_gravity_cg-1

Tests `EnzoMethodGravityCg` at P=1 in 2D.

method_gravity_cg-8

Tests `EnzoMethodGravityCg` at P=8 in 2D.

Analytic-Tests

These tests are associated with a dedicated python-script that runs a test problem with a known analytic solution.

stable_jeans_wave

The basic setup for this test is to initialize a 1D stable Jeans wave in a 3D box that is inclined with respect all 3 Cartesian axes. Enzo-E writes the initial snapshot to disk, evolves the wave over a complete period, and then writes the final snapshot result to disk. Then the python script computes the L_1 error norm computed by comparing values in the initial and final snapshots (the analytic solution expects them to be identical) and compares it with a precomputed value.

For this test to pass, the gravity solver and hydro solver's gravity source terms must each be implemented correctly. At this time, this test has only been defined to use the VL+CT solver (without magnetic fields) and the cg-solver. The python script invokes the test at 2 resolutions (16 cells per wavelength and 32 cells per wavelength).

To invoke this test, invoke the following command from the root directory of the repository:

```
python input/Gravity/run_stable_jeans_wave_test.py \
  --launch_cmd <path-to-enzoe-binary>
```

where `<path-to-enzoe-binary>` is replaced with the path to the Enzo-E binary that you want to test. The test prints a summary of the results to stdout and an exit code of of 0 indicates that all subtests passed.

7.5.12 Heat Tests

Tests for heat methods

test_heat

2D Heat Equation solved with Forward Euler

method_heat-1

2D Heat Diffusion problem with P=1

method_heat-8

2D Heat Diffusion problem with P=8

7.5.13 HelloWorld

Tests of full code to demonstrate functionality. These are done as a demostration of enzo's capabilities.

HelloWorld

Demonstration test of Enzo-E, generates high pressure region in shape of Hello World

Hi

Shorter version of HelloWorld. Takes few minutes as opposed to several hours

initial_png

Evolves high-pressure region Hello World using Cello.png mask file

nsf-demo

Demo for nsf (national science foundation)

test_soup-2d

tests a sedov type problem with letters instead of spheres in 2D

test_soup-3d

tests a sedov type problem with letters instead of spheres in 3D

7.5.14 Hierarchy Tests

Tests Hierarchy of enzo

enzop-cello-amr

Tests enzo cello amr hierarchy

7.5.15 Hydro tests

Tests on hydrodynamic methods

advect2u

Tests advection methods

test_double_mach

2D double mach reflection problem

From *broken link <https://ww.astro.virginia.edu/VITA/ATHENA/dmr.html>_*

test_implosion

Runs a 2D implosion problem. Tests Initial:<field>:value. Test outputs density image file and mesh image file.

test_implosion-code

Runs a 2D implosion problem. Tests Initial:code.

test_kelvin_helmholtz

Sample Kelvin Helmholtz instability test. Test outputs density image file and mesh image file.

test_turbulence3d

Tests turbulence method

7.5.16 Initial MUSIC Tests

Tests for reading MUSIC HDF5 initial conditions at different mesh root_blocks

Solves simple cosmology problem using initial conditions from MUSIC HDF5 initial condition geneerator

initial_music-111

Tests reading MUSIC HDF5 initial conditions at `mesh` *root_blocks = [1,1,1]*

initial_music-112

Tests reading MUSIC HDF5 initial conditions at `mesh` *root_blocks = [1,1,2]*

initial_music-114

Tests reading MUSIC HDF5 initial conditions at `mesh` *root_blocks = [1,1,4]*

initial_music-121

Tests reading MUSIC HDF5 initial conditions at `mesh` *root_blocks = [1,2,1]*

initial_music-141

Tests reading MUSIC HDF5 initial conditions at `mesh` *root_blocks = [1,4,1]*

initial_music-211

Tests reading MUSIC HDF5 initial conditions at `mesh` *root_blocks = [2,1,1]*

initial_music-222

Tests reading MUSIC HDF5 initial conditions at `mesh` *root_blocks = [2,2,2]*

initial_music-411

Tests reading MUSIC HDF5 initial conditions at `mesh` *root_blocks = [4,1,1]*

initial_music-444

Tests reading MUSIC HDF5 initial conditions at `mesh` *root_blocks = [4,4,4]*

7.5.17 Isolated Galaxy Tests

Under construction

method_isolatedgalaxy

Under construction

method_isolatedgalaxy-particles

Under construction

WLM_small

Under construction

7.5.18 Merge Sinks Tests

Four tests of the enzo-e "merge sinks" method.

- the stationary serial test
- the stationary parallel test
- the drifting serial test
- the stationary parallel test

All the files necessary for running the tests can be found in input/merge_sinks.

Each test involves setting up initial conditions with 1000 sink particles randomly distributed in a sphere, with velocities directed towards the centre of the sphere. In the "drifting" tests, an additional uniform velocity is added to all the particles, and in the "stationary" tests, there is no additional velocity.

For the serial tests, Enzo-E runs in serial mode.

For the parallel tests, Enzo-E is run in parallel on four PEs.

The methods used are "pm update" and "merge sinks", so that the particles all move with constant velocity until they are merged together.

The python script input/merge_sinks/mass_momentum_conservation.py then tests if mass and momentum are conserved across all the output snapshots, to within some tolerance, which depends on the floating-point precision with which Enzo-E was compiled (1.0e-4 for single precision, 1.0e-6 for double precision). If all quantities are indeed conserved, the test passes, if not, the test fails.

These tests can be executed by running "ctest -R merge_sinks" in the build directory. See input/merge_sinks/README for further information on these tests.

7.5.19 Methods Tests

Tests for general methods

test_null

Runs null method. This tests AMR meshing infrastructure without having a specific method.

7.5.20 Output Tests

Tests checking for correct output

output-header

Under construction

output-stride-1

Runs a simple 2D implosion. Tests output stride where `stride_write = 1`. This means testing where specifying that exactly one physical processor is used to output data.

output-stride-2

Similar to above test but on two physical processors where `stride_write = 2`

output-stride-4

Similar to above test but on four physical processors where `stride_write = 4`

output-data

Output test of 2D implosion problem

7.5.21 Parse Tests

Under construction

parse_groups

Under construction

parse_include

Under construction

parse_integer

Under construction

parse_list

Under construction

parse_logical

Under construction

parse_scalar

Under construction

7.5.22 Particle Tests

Tests on particle motion. Uses tracer particles.

test_particle-x

Tests particle motion in x direction only

test_particle-y

Tests particle motion in y direction only

test_particle-xy

Tests particle motion in x and y directions together

test_particle-circle

Tests circular particle motion

test_particle-amr-static

Tests static particle in adaptive mesh refinement

test_particle-amr-dynamic

Tests dynamic particle in adaptive mesh refinement

7.5.23 Performance Tests

Tests with performance monitoring

performance-papi

2D implosion problem with Performance monitoring

perf

7.5.24 PPM tests

Tests for PPM method. Runs a 2D implosion problem using ppm method and outputs density image file and a density data file.

method_ppm-1

Tests PPM method at P=1

method_ppm-8

Tests PPM method at P=8

7.5.25 PPML Tests

Tests for PPML methods. Each test outputs density image files for each axis and a data dump file.

method_ppml-1

Tests PPML method with P=1

method_ppml-8

Tests PPML method with P=8

7.5.26 Sedov Tests

sedov

Tests Sedov method

7.5.27 Shu Collapse Test

Runs the Shu Collapse problem as described in Federrath et al 2010, ApJ, 713, 269, and tests for mass conservation.

The initial conditions include static isothermal gas with an inverse square density profile within a truncation radius, with the gas having a constant small "external density" outside of the truncation radius. When the problem is left to run, a sink particle forms at the center, and the gas collapses towards the center to be accreted by the sink particle.

This test can be executed by running "ctest -R shu_collapse" in the build directory.

All the files required for running the test can be found in input/shu_collapse. See input/shu_collapse/README for further information.

7.5.28 VLCT Tests

Under construction

dual_energy_cloud

Under construction

dual_energy_shock_tube

Under construction

HD_linear_wave

Under construction

MHD_linear_wave

Under construction

MHD_shock_tube

Under construction

orszang-tang

Under construction

passive_advect_sound_wave

Under construction

7.5.29 Other Tests

Under construction

bb_unigrid_SF_FB

Under construction

method_feedback

Under construction

test_collapse-bcg2

Under construction

test_collapse-bcg3

Under construction

test_collapse-dd2

Under construction

test_collapse-dd3

Under construction

test_collapse-gas-bcg2

Under construction

test_collapse-gas-dd2

Under construction

test_collapse-gas-hg2

Under construction

test_collapse-hg2

Under construction

test_collapse-hg3

Under construction

test_cosmo-bcg-fc0

Under construction

test_cosmo-bcg-fc1

Under construction

test_cosmo-bcg

Under construction

test_cosmo-cg-fc0

Under construction

test_cosmo-cg-fc1

Under construction

test_cosmo-cg

Under construction

test_cosmo-dd-fc0

Under construction

test_cosmo-dd-fc1

Under construction

test_cosmo-dd

Under construction

test_cosmo-hg-fc0

Under construction

test_cosmo-hg-fc1

Under construction

test_cosmo-hg

Under construction

test_cosmo-mg-fc0

Under construction

test_cosmo-mg-fc1

Under construction

test_cosmo-mg

Under construction

test-flux2-xm

Under construction

test-flux2-xp

Under construction

test-flux2-ym

Under construction

test-flux2-yp

Under construction

test-flux3-xm

Under construction

test-flux3-xp

Under construction

test-flux3-ym

Under construction

test-flux3-yp

Under construction

test-flux3-zm

Under construction

test-flux3-zp

Under construction

7.6 Existing Answer Tests

The answer test suite currently covers the following simulations:

- *cosmo-dd-multispecies-short*
- *dual_energy_cloud*

7.6.1 Grackle Tests

Tests for the method that invokes Grackle. These tests set up a cooling test without hydrodynamics to run many one-zone models in Grackle, fully sampling the density, temperature, and metallicity parameter space over which the chemistry and cooling/heating tables are valid.

method_grackle_general

This test compares the summary statistics computed for several grackle fields after a certain period of time to previously archived values.

The simulation timesteps are much larger that the cooling/heating. This makes it more likely that separate processing elements will execute grackle routines at the same time (thus increasing the chances of exposing hypothetical problems related to Grackle & SMP mode).

This test is somewhat fragile given that upgrading Grackle versions could conceivably alter the field values. In the future it would be better to replace this with a test that:

1. checks out and builds a previous commit of Enzo-E
2. runs the simulation and saves the exact field values after running the simulations
3. checks out and builds a newer commit of Enzo-E (while leaving the build of Grackle unchanged)
4. runs the simulation and confirms that the Grackle related field values are identical to the field values from the earlier simulation.

method_grackle_cooling_dt

This test runs Grackle for a fixed number of cycles, and compares the final simulation time to a reference value. Each simulation timestep is set fraction of the minimum magnitude of the cooling/heating timestep.

CHAPTER

EIGHT

GETTING STARTED

The Getting started using Enzo-E section should cover everything one needs to know to download Enzo-E / Cello and its dependent software, configure, build, and run a small example test problem.

CHAPTER

NINE

PARAMETERS

The Enzo-E / Cello parameter reference section is a reference page for all Enzo-E and Cello parameters, which are used to write parameters files defining simulations to run.

How parameters are organized within a parameter file is described in Parameter files, which covers parameter groups, subgroups, parameters, and data types recognized.

An example parameter file is described in detail in Parameter file example.

CHAPTER

TEN

USING AND DEVELOPING ENZO-E / CELLO

The new Using Enzo-E section will describe in detail how to use Enzo-E, including what methods, fields, particle types etc. are available. While only an outline exists now, we are working on getting this section completed soon.

The new Developing with Cello section will describe how to add new functionality to an application such as Enzo-E built on Cello, including adding new computational methods, initial conditions, and refinement criteria. As with the Using Enzo-E section, this documentation is under active development, and is mostly an outline at this point.

CHAPTER

ELEVEN

PDF USER AND DEVELOPER GUIDE (DEPRECIATED)

An Enzo-E / Cello User and Developer Guide is available from the link below. Warning, it's big (0.2GB), and currently written as a presentation, and some sections are somewhat outdated. This document, while currently still useful, is being phased out in favor of the above online content.

`Using and Developing Enzo-E/Cello`

Presentations given at the 2018 Enzo Days Workshop are also available below. They are more up-to-date, but are also still rather large.

`Enzo-E / Cello Status and What's New?`

`Introduction to Enzo-E`

`First Steps with Enzo-E`

CHAPTER

TWELVE

INDICES AND TABLES

- genindex
- search

INDEX

D

disk_utils::dump_view_to_hdf5 (*C++ function*), 237
disk_utils::dump_views_to_hdf5 (*C++ function*), 237

E

EnzoEOSVariant::get (*C++ function*), 231
EnzoEOSVariant::get_if (*C++ function*), 232
EnzoEOSVariant::holds_alternative (*C++ function*), 232
EnzoEOSVariant::visit (*C++ function*), 234
EnzoPhysicsFluidProps::apply_floor_to_energy_and_sync (*C++ function*), 228
EnzoPhysicsFluidProps::dual_energy_config (*C++ function*), 227
EnzoPhysicsFluidProps::eos_variant (*C++ function*), 227
EnzoPhysicsFluidProps::fluid_floor_config (*C++ function*), 227
EnzoPhysicsFluidProps::pressure_from_integration (*C++ function*), 228
EnzoPhysicsFluidProps::primitive_from_integration (*C++ function*), 228

F

Face FaceFluxes::face(), **252**
Face::Face (*int ix, int iy, int iz, int axis, int face*), **251**
FaceFluxes & FaceFluxes::operator *= (*double weight*), **252**
FaceFluxes::FaceFluxes (*Face face, int index_field, int nx, int ny, int nz, int cx, int cy, int cz*), **251**
FluxData::FluxData(), **253**

I

int Face::axis(), **251**
int Face::face(), **251**
int FaceFluxes::index_field(), **252**
int FluxData::index_field(int i_f), **253**
int FluxData::num_fields(), **253**

M

Method::compute (*C++ function*), 184
Method::compute_resume (*C++ function*), 184
Method::courant (*C++ function*), 186
Method::name (*C++ function*), 184
Method::timestep (*C++ function*), 184
MethodFluxCorrect::MethodFluxCorrect(), **251**

P

par:param (*role*), 243
par:parameter (*directive*), 242
par:paramfmt (*role*), 243
par:typefmt (*role*), 244

S

std::vector<double> & FaceFluxes::flux_array (*int * dx=0, int * dy=0, int * dz=0*), **252**

V

virtual void MethodFluxCorrect::compute (*Block * block*), **251**
void Face::get_face (*int *ix, int *iy, int *iz*), **251**
void FaceFluxes::accumulate (*FaceFluxes & ff, int cx, int cy, int cz, rank*), **252**
void FaceFluxes::allocate(), **252**
void FaceFluxes::clear(), **252**
void FaceFluxes::coarsen(int cx, int cy, int cz, int rank), **252**
void FaceFluxes::deallocate(), **252**
void FaceFluxes::get_size (*int * mx, int * my, int * mz*), **252**
void FaceFluxes::set_flux_array (*std::vector<double> array, int dx, int dy, int dz*), **252**
void FluxData::allocate(int nx, int ny, int nz, std::vector<int> field_list, std::vector<int> * cx_list=nullptr, std::vector<int> * cy_list=nullptr, std::vector<int> * cz_list=nullptr), **253**

void FluxData::block_fluxes(int axis, int face, int i_f), 253
void FluxData::deallocate(), 253
void FluxData::neighbor_fluxes(int axis, int face, int i_f), 253
void FluxData::set_block_fluxes(FaceFluxes * ff, int axis, int face, int i_f), 253
void FluxData::set_neighbor_fluxes(FaceFluxes * ff, int axis, int face, int i_f), 253
void FluxData::sum_neighbor_fluxes(FaceFluxes * ff, int axis, int face, int i_f), 253