The athena3.1 *User's Guide*

James M. Stone, Thomas A. Gardiner
Department of Astrophysical Sciences
Princeton University
Princeton, NJ 08540

Peter J. Teuben
Department of Astronomy
University of Maryland
College Park MD 20742-2421

and

John F. Hawley
Department of Astronomy
University of Virginia
PO Box 3818 University Station
Charlottesville, VA 22903

1 Introduction

Athena is a grid-based code for astrophysical gas dynamics developed with support of the NSF Information Technology Research (ITR) program. This *User's Guide* describes version 3.1 (hereafter referred to as `athena3.1`); the third publicly released version of the code.

The `athena3.1` code contains algorithms for the following:

- compressible hydrodynamics and ideal MHD in one, two, or three spatial dimensions (Cartesian coordinates only),
- ideal gas equation of state with arbitrary γ (including $\gamma = 1$, an isothermal EOS),
- second- or third-order reconstruction using the characteristic variables (first-order interpolation is also available),
- numerical fluxes computed using a variety of Riemann solvers (including various solvers based on HLL fluxes, Roe's linearization, and exact solvers in certain simple cases),
- source terms due to a static gravitational potential,
- self-gravity computed using FFTs,
- an arbitrary number of passive scalars advected with the flow,
- parallelization based on MPI.

It is strongly advised to use the latest release of the code for research problems (even those in one-dimension) to take advantage of improvements and potential bug fixes.

There are four basic sources of documentation for `athena3.1`:

1. *The Method Papers:* There are four in all: Gardiner & Stone (2005; 2008), Stone & Gardiner (2008), and Stone et al. (2008). By the end of 2008, each of these are available on ADS, astro-ph, or with the code distribution.
2. *The User's Guide:* (this document) gives an overview of how to install, configure, compile, and run `athena3.1` and visualize the resulting output.
3. *The Programmer's Guide:* gives a basic introduction to the data structures, variable names, grid definitions, and code structure.
4. *Web-based Tutorials:* contains instructions[1] for running problems from the `athena3.1` test suite, including examples of results.

Users of `athena3.1` should have a basic, working knowledge of the Unix operating system, access to a C compiler, and a graphics package for plotting or animating one-, two-, and three-dimensional data. Some familiarity with code management using Makefiles is helpful but not necessary.

The code has been developed using the GNU development tools, maintaining strict adherence to ANSI standards, thus it should be possible to configure, compile, and run the code on any platform that supports these standards. It has been tested on Linux, MacOSX, and Solaris.

[1] `http://www.astro.princeton.edu/~jstone/athena.html`

1.1 Changes from `athena3.0`

The last publicly released version of the code was v3.0. The current version differs primarily in that it has more physics, in particulat the advection of passive scalars, and self-gravity using FFTs. Like v3.0, this version implements two different unsplit, three-dimensional algorithms; one based on CTU (Gardiner & Stone 2008), and one based on an algorithm due to van Leer (Stone & Gardiner 2008).

In addition, there are a number of structural changes to the code, including: (1) some files in `/src` and `src/prob` are renamed, and some new files are added, (2) some options have changed in the configure script, (3) there have been some bugs fixed in this version.

1.2 Future versions

The current developmental version of the code includes many more algorithmic extensions than in this release, such as static and adaptive mesh refinement, and radiation transfer. It is likely that some of these extensions will be released in the future, albeit on an irregular schedule.

2 Quick Start

To install, configure, compile, and run the code do the following.

1. Download[2], uncompress, and untar the source code distribution file.
2. Create the configure script by running `autoconf` in the `athena3.1` directory.
3. Test the install by running `configure`; `make all`; `make test`.
4. If there are no errors in the previous step the install was successful. The code can now be re-configured, re-compiled, and used to run any of the test problems in `/src/prob`.

3 Getting Started

3.1 Obtaining and Installing `athena3.1`

The source code for `athena3.1` can be downloaded as a gzipped tar file from the web[2]. To install and run the code requires only a C compiler.

After downloading the tar file, uncompressing and untarring it, the following directory structure should be created:

```
/athena3.1
          /doc            documentation, manuals
          /src            source code, and include files
              /prob       problem files (see /src/problem.c for a symbolic link)
          /tst            various input files for tests
              /1D-hydro
              /1D-mhd
              /2D-hydro
              /2D-mhd
```

[2]from `http://www.astro.princeton.edu/∼jstone/athena.html`

```
            /3D-hydro
            /3D-mhd
        /vis            visualization tools and scripts
            /dx         OpenDX scripts
            /idl        IDL (rsinc.com) scripts
            /sm         Super Mongo scripts
            /vtk        code to join VTK legacy files
```

In addition to the directories listed above (which are created when you untar the source code), another directory will be created by the Makefile when `athena3.1` is compiled for the first time, using `make all`.

```
/athena3.1
          /bin           contains executable, created by Makefile
```

(Trying to compile the code with `make compile` before `make all` will result in an error since the `bin` directory will not yet exist.)

The `athena3.1/doc` directory contains .pdf files of this document, the *Programmer's Guide*, and the Method papers. These documents serve as the primary source of reference for the code, and should be consulted for complete information.

3.2 Configuring `athena3.1`

After you have installed `athena3.1` the next step is to configure the code for a specific problem. To generate the `configure` shell script using the GNU `autoconf` toolkit[3], type `autoconf` in the `athena3.1` directory. If you wish to by-pass the configure script and work with the Makefiles directly, see §3.4 below.

There are several basic functions served by the `configure` script. The first is to enable or disable *features* in the code, and to choose between different *packages* implemented in `athena3.1`. Configure features are used when a particular option has only two choices: enabled ("on") or disabled ("off"): examples include choosing whether the executable uses single or double precision, or enabling or disabling various data output formats. Configure packages are used when an option may have more than one choice. Examples of packages include the basic physics options (such as hydrodynamics or MHD, adiabatic or isothermal equation of state, etc.), as well as the algorithm options (order of accuracy for the reconstruction step, choice of Riemann solver, etc.). These package options are controlled at the source code level using C precompiler macros. The `configure` script provides a powerful way of setting these macros through a command line interface that does not require the user to edit any special files.

More advanced features of the `configure` script are to set compiler and linker options and flags using environment variables, and to query the system automatically to see that a C-compiler, linker, and any external libraries necessary for compilation are installed and accessible. Currently, only the former of these features are utilized.

Once the `configure` script has been created, it can be used with the following syntax

`configure [--enable-`*feature*`] [--disable-`*feature*`] [--with-`*package*`=`*choice*`]`,

where *feature* and *package* are valid options in `athena3.1`, and *choice* is the value to which *package* is to be set. The valid optional features in `athena3.1` are given in Table 1, and the valid optional packages (including all possible choices and their default values) are given in Table 2[4].

Table 1: Optional features controlled by `configure` in `athena3.1`

FEATURE	DEFAULT	Comments
single	disabled	computations performed in single precision (default is double)
debug	disabled	compiles code with flags needed for debugger
ghost	disabled	causes ghost zones to be written during output
mpi	disabled	parallelization using MPI library
h-correction	disabled	H-correction to fix carbuncle problem
fft	disabled	compile and link with FFTW block decomposition

Only one choice can be made for each optional package given in Table 2 (the choices are mutually exclusive). The "problem" package should be set to the file name in the directory `athena3.1/src/prob` that is to be used by the code to initialize the problem of interest. A variety of choices are included for the test problems that can be run by `athena3.1` (the default problem file is `athena3.1/src/prob/linear_wave1d.c`, *i.e.* `--with-problem=linear_wave1d`), these are described in more detail in §6. Note that `configure` creates a symbolic link between `problem.c` in `athena3.1/src` and the appropriate file in `athena3.1/src/prob`.

The `configure` script should be run in the root directory `/athena3.1`. Running `configure` with the help option (`configure --help`) gives more information, including a list of all optional features and packages.

For example, to configure `athena3.1` to run an isothermal hydrodynamical shocktube problem initialized with the function `shkset1d.c` problem using Roe fluxes and third-order interpolation in single precision, use the following:

```
configure --with-problem=shkset1d --enable-single --with-eos=isothermal
                         --with-gas=hydro --with-order=3
```

To configure `athena3.1` to run the linear wave test problem in 3D adiabatic MHD using HLLD fluxes, the van Leer integrator, and second-order interpolation in double precision parallelized with MPI, use the following:

```
configure --with-flux=hlld --with-problem=linear_wave3d --with-integrator=vl
                                   --enable-mpi
```

When `configure` runs, it creates a custom `Makefile` for the problem in the directory `athena3.1/src` using the file `athena3.1/src/Makefile.in` as a template. After successful execution, `configure` will echo the options that have been set (including all the default values).

3.3 Compiling `athena3.1`

After running `configure`, all that is required to compile the code is to type `make all` in the top level directory `/athena3.1` (within which the configure script is run). This automatically creates the directory `athena3.1/bin`, which will contain the executable, and runs make in the `athena3.1/src` directory to compile and link the code. The top-level Makefile in the `/athena3.1` directory also contains other targets listed in Table 3.

[3]autoconf: see `http://www.gnu.org/software/autoconf/`

[4]The file `configure.ac` contains the information auto-conf needs to generate the configure script with these options.

Table 2: Optional packages controlled by configure in `athena3.1`

PACKAGE	CHOICE[a]	Comments
problem	*file-name*	use *file-name* in directory `/src/prob` for initial conditions
gas	hydro	create code for hydrodynamics
	mhd	create code for MHD
eos	**adiabatic**	use adiabatic equation of state
	isothermal	use isothermal equation of state
nscalars	#	add # passively advected scalars (default is 0)
gravity	fft	enable self-gravity using FFTs
flux	**roe**	Roe's Riemann solver
	force	FORCE flux
	hlle	HLLE Riemann solver
	hllc	HLLC Riemann solver (hydrodynamics only)
	hlld	HLLD Riemann solver (MHD only)
order	1	use first-order spatial reconstruction
	2	use second-order (piecewise-linear) spatial reconstruction
	3	use third-order (piecewise-parabolic) spatial reconstruction
integrator	**ctu**	corner transport upwind unsplit integrator in 3D
	vl	van Leer unsplit integrator in 3D

[a] Default choices shown in bold.

Table 3: Some Makefile targets in `athena3.1`

TARGET	Comments
all	creates `/athena3.1/bin` directory and compiles code
compile	compiles code
clean	removes .o files in `/athena3.1/src`
help	prints help message
test	runs install test (see §10.1)

Normally the Makefiles should never be edited by hand. However, it is possible to edit the Makefile in the `/athena3.1/src` directory to change compiler flags (to improve optimization, for example) rather than using environment variables. Just remember that the `Makefile` is always overwritten every time `configure` is run, so any custom changes may be lost.

To make changes to the Makefile in the `/athena3.1/src` that are permanent, edit the template file `/athena3.1/src/Makefile.in`. For example, the best way to customize the compiler, options, flags, and the path of local libraries is through the `MACHINE=` option on the make command line. Currently, the options for several target machines are included in `Makefile.in`. If no value for the `MACHINE` option is provided, the default is to use the `gcc` compiler with an optimization level of `-O3`. By using

```
% make all MACHINE=ophir
```

the Intel C compiler (icc) with the options `-O2 -xW -tpp7 -ipo -i_dynamic -gcc-name=gcc32` will be used. To add a new target machine, copy one of the machine targets in `Makefile.in`, rename it, and edit as appropriate.

3.4 By-passing configure

Most of the physics options in `athena3.1` are controlled by precompiler macros. The complete set of valid macros (some of which are mutually exclusive) are defined in the file `/src/defs.h.in`. The configure script creates a header file `/src/defs.h` which contains the appropriate set of macro definitions required for the given problem. However, one can by-pass the configure script by creating and editing the `defs.h` file by hand (be warned that running configure at some later time will overwrite this file).

Similarly, the compilation step is controlled by a Makefile generated by the configure script from `/src/Makefile.in`. To bypass the configure script, a custom Makefile must be created by hand from this template. Again, remember that the `Makefile` is always overwritten every time `configure` is run, so any custom changes may be lost.

3.5 Running `athena3.1`

After configuring and compiling `athena3.1`, there should be an executable called `athena` in the directory `athena3.1/bin`. There are two steps to running the code: (1) editing parameters in the input file, and (2) running the executable. Editing the input file is described in more detail in the subsections below.

The `athena3.1` executable can be run using the `-i` option to specify the name of the input file. For example, to run the Brio & Wu shocktube use the command

```
  % athena -i ../tst/1D-mhd/athinput.brio-wu
```

The code will first echo the values of all input parameters to stdout. During the main integration loop it will print the cycle number and timestep, and when it concludes it will print final diagnostic information.

A variety of command line options have been implemented in `athena3.1`. A list is given by the `-h` switch:

```
  % athena -h
Athena version 3.1 - 01-JAN-2008
  Last configure: Wed Jan  9 09:25:53 EST 2008

Usage: athena [options] [block/par=value ...]

Options:
  -i <file>       Alternate input file [athinput]
  -d <directory>  Alternate run dir [current dir]
  -h              This Help, and configuration settings
  -n              Parse input, but don't run program
  -c              Show Configuration details and quit
  -r <file>       Restart a simulation with this file

Configuration details:

 Problem:                 linear_wave3d
 Gas properties:          MHD
 Equation of State:       ADIABATIC
 Passive scalars:         0
 Self-gravity:            none
 Order of Accuracy:       2 (SECOND_ORDER)
 Flux:                    hlld
 Unsplit 3D integrator:   ctu
 Precision:               DOUBLE_PREC
 Output Modes:
   Ghost Cells:           disabled
 Parallel Modes:
   MPI:                   MPI_SERIAL
 H-correction:            disabled
 FFT:                     disabled
```

The `-d` option can be used to create a new directory in which `athena3.1` will run and write the output files. The `-n` option is useful for debugging any parsing errors, as it will dump the contents of all parsed block/parameters.

A value for any of the valid parameter names in the input file can also be input from the command line, this over-rides the values in the input file. This, in combination with the `-d` option, is useful for parameter surveys. The `-c` option is useful for checking the configuration parameters with which the executable was compiled.

3.5.1 Editing the input file

Run time parameters are set in an input file, usually given the name `athinput.`*problem-name*, where *problem-name* is a string identifier. Often this string is the same as the name of the problem-generator, i.e. the function in `athena3.1/src/prob` which is used to initialize the data. In some cases, a name which is more specific to the problem at hand is used (since some problem-generators can be used to initialize more than one problem).

As an example of an `athena3.1` input file, the file `/athena3.1/tst/1D-mhd/athinput.brio-wu` is reproduced below. Other examples can be found in the subdirectories in `/athena3.1/tst/`.

```
<comment>

problem = Brio & Wu shock tube
author  = M. Brio & C. C. Wu
journal = J. Comp. Phys. 75, 400-422 (1988)
config  = --with-problem=shkset1d

<job>

problem_id      = Brio-Wu   # problem ID: basename of output filenames
maxout          = 3         # Output blocks number from 1 -> maxout

<output1>
out_fmt = tab               # Tabular data dump
dt      = 0.0025            # time increment between outputs

<output2>
out_fmt = hst               # History data dump
dt      = 0.0025            # time increment between outputs

<output3>
```

```
out_fmt = bin               # Binary data dump
dt      = 0.0025            # time increment between outputs

<time>

cour_no         = 0.8       # The Courant, Friedrichs, & Lewy (CFL) Number
nlim            = 10000     # cycle limit
tlim            = 0.1       # time limit

<grid>

Nx1             = 800       # Number of zones in X1-direction
x1min           = 0.0       # minimum value of X1
x1max           = 1.0       # maximum value of X1
ibc_x1          = 2         # inner (X1) boundary flag
obc_x1          = 2         # outer (X1) boundary flag

Nx2             = 1         # Number of zones in X2-direction
x2min           = 0.0       # minimum value of X2
x2max           = 1.0       # maximum value of X2
ibc_x2          = 2         # inner (X2) boundary flag
obc_x2          = 2         # outer (X2) boundary flag

Nx3             = 1         # Number of zones in X3-direction
x3min           = 0.0       # minimum value of X3
x3max           = 1.0       # maximum value of X3
ibc_x3          = 2         # inner (X3) boundary flag
obc_x3          = 2         # outer (X3) boundary flag

<parallel>

NGrid_x1 = 1
NGrid_x2 = 1
NGrid_x3 = 1

<problem>

gamma           = 2.0       # gamma = C_p/C_v

shk_dir         = 1         # Shock Direction -- (1,2,3) = (x1,x2,x3)

dl              = 1.0       # density on left half of grid
pl              = 1.0       # pressure
v1l             = 0.0       # X-velocity
v2l             = 0.0       # Y-velocity
v3l             = 0.0       # Z-velocity
b1l             = 0.75      # X-magnetic field
b2l             = 1.0       # Y-magnetic field
b3l             = 0.0       # Z-magnetic field

dr              = 0.125     # density on right half of grid
pr              = 0.1       # pressure
v1r             = 0.0       # X-velocity
v2r             = 0.0       # Y-velocity
v3r             = 0.0       # Z-velocity
b1r             = 0.75      # X-magnetic field
b2r             = -1.0      # Y-magnetic field
b3r             = 0.0       # Z-magnetic field
```

Note the syntax of the parameter specification used in this file. Parameters are grouped into named *blocks*, with the name of each block appearing on a single line within angle brackets. Block names must always appear in angle brackets on a separate line (although the blank lines above and below the block names are not required).

Below each block name is a list of parameters, with syntax

$$\textit{parameter-name} = \text{value [\# comments]}$$

White space after the parameter name, after the '=', and before the '#' character is ignored. Everything after (and including) the '#' character is also ignored. Only one parameter value can appear per line. Comment lines (i.e. a line beginning with '#') are allowed for documentation purposes. A maximum number of 256 characters is permitted per line in the input file. Both block names and parameter names are case sensitive.

The input file is read by a very flexible parser written for `athena3.1` (`/athena3.1/src/par.c`). The entire input file is read at the very beginning of the main program, and the parameter names and their values stored in memory. Thereafter, these values can be accessed as necessary by any function at any time during execution. The parser allows the parameter names to appear in any order within a named block, extra (or misspelled) parameter names will be parsed and never used. There are no default values for any of the run-time parameters in the input file; each parameter must be supplied a value through the input file. If a value is requested from the parser but its name does not exist, the parser will print an error message and terminate the execution of the program. In this way, missing parameter names will be detected at run time. The parameters may be integers, floating point numbers, or strings. The parser will do automatic type conversion, for example converting floating point numbers to double precision if necessary (though the user is expected to know the basic difference between real, integer, and string data types). Parameter values can also be set at run time through the command line, which provides a very flexible way of testing the code and running parameter searches, using the syntax "*block/parameter=value*".

Below we describe each of the parameter blocks in the input file, and the parameters they contain.

3.5.2 The `<comment>` block

Provides self-documentation of the file. The variables in this block are not used in the code.

3.5.3 The `<job>` block

Parameters in this block control properties of the jobs run by `athena3.1`. They are accessed by the function `main.c`.

- **problem_id:** string added as basename of output filenames (see §4). Usually same as *problem-name* used in input file. There is no maximum length for this name[5].
- **maxout:** Specifies how many output blocks will be read from input file. Output blocks `<output1>` through `<outputN>` where `N` = maxout are scanned for valid output descriptions. Missing output blocks are permitted.

[5]Although remember that the maximum length of a line in the input file is 256 characters.

3.5.4 The `<output#>` block

Parameters in these blocks control the writing of "outputs", e.g. data dumps, images, etc.

- **out:** Variable to image for `pgm, ppm, fits` output formats. Currently accepted values are: `d, M1, M2, M3, E, B1c, B2c, B3c, ME, V1, V2, V3, P, S, cs2`. If variable is set to `all`, then output will include `d, M1, M2, M3` and may also include `E, B1c, B2c, B3c` depending on the configuration, and the output format must be one of `bin, dx, hst, tab, rst, vtk` (which can only dump all rather than selected variables).
- **out_fmt:** Output format, e.g. `bin, dx, hst, tab, rst, vtk, fits, pdf, pgm, ppm`. See §4 for more description of these formats.
- **dat_fmt:** Optional field for controlling the format string used to write tabular output files, e.g. `%12.5e`. This value should not appear in quotes and no white space should be present.
- **dt:** Time increment between outputs (in problem time).
- **time:** Time of next output (in problem time). If not set, the default will be the initial problem time (for new runs), or the current problem time (for restarts).
- **id:** Any string, added to label output filenames.
- **dmin/dmax:** max/min applied to output (useful for images).
- **palette:** Color palette for images. Currently available palettes are `rainbow, jh_colors, idl1, idl2, step8, step32, heat`.
- **ix1, ix2, ix3:** Range of indices in x1, x2, or x3 directions over which data is averaged. For example `ix1=:` will average over whole x1 axis and dump a 2D array. `ix1=5:` will average from 5 to end. `ix1=:10` will average from start to 10. `ix1=5:10` will average from 5 to 10. `x1=5` will extract the single plane at i index 5.
- **usr_expr_flag:** Set to 1 if a user-defined expression is to be used to compute output quantity, see §4.2.

3.5.5 The `<time>` block

Parameters in this block control times in a job (such as ending time). They are accessed by the function `main.c`.

- **tlim:** Time to stop integration, in units defined by problem
- **nlim:** Maximum number of cycles of the main loop before stopping. Set to -1 to stop only on time limit `tlim`.
- **cour_no:** CFL number, must be less than 1.0 for 1D and 2D, 0.5 for 3D.

3.5.6 The `<grid>` block

Parameters in this block control the properties of the grid. They are accessed by the function `init_grid_block.c`.

- **Nx1, Nx2, Nx3:** number of grid cells in the x1-, x2-, and x3-directions.
- **x1min, x2min, x3min:** x1-, x2-, x3-coordinate of left-edge of first cell
- **x1max, x2max, x3max:** x1-, x2-, x3-coordinate of right-edge of last cell. The computational domain in the x1-direction spans $x1_{max} - x1_{min}$, the grid spacing is $\triangle x1 = (x1_{max} - x1_{min})/Nx1$, and the center of the first cell is located at $x1 = x1_{min} + \triangle x1/2$. Also, $x1_{max} > x1_{min}$ is required. Similarly for the computational domain in the x2- and x3-direction.
- **ibc_x1, obc_x1:** integer flags for boundary conditions applied at “inner” (left) and “outer” (right) edges of grid. Currently three values are implemented: 1 = reflecting, 2 = outflow (projection), and 4 = periodic. See §5 for more information.
- **ibc_x2, obc_x2:** Analogous parameters for the x2-direction.
- **ibc_x3, obc_x3:** Analogous parameters for the x3-direction.

3.5.7 The `<parallel>` block

Parameters in this block control the decomposition of the computational domain into MPI blocks. The domain can be decomposed in any coordinate direction, and into an arbitrary number of blocks. This allows slab, pencil, and block decompositions.

- **NGrid_x1:** Number of MPI blocks in the x1-direction.
- **NGrid_x2:** Number of MPI blocks in the x2-direction.
- **NGrid_x3:** Number of MPI blocks in the x3-direction.

This block can be omitted for serial jobs, or the number of MPI blocks can be set to one in each dimension (as in the example).

3.5.8 The `<problem>` block

Parameters in this block are accessed by the problem-generator, and therefore depend on the problem being run. For example, the problem-generator `athena3.1/src/prob/shkset1d.c` (which is used for the Brio & Wu test problem) requires the following parameter values in the `<problem>` block:

- **gamma:** ratio of specific heats used in equation of state
- ***l:** values of variable * in left-state
- ***r:** values of variable * in right-state

The ***l** and ***r** parameter names are specific to the Brio & Wu shocktube problem. In general, the input file for other problems will have different variable names in the problem block.

4 Data output formats

As described above in §3.5.4, data output in the `athena3.1` code is controlled by the `<output>` blocks in the input file. There should be one block for each type of data output required. There is no limit on the total number of outputs. Output filenames use a naming convention *basename-id#.dumpid.outid.type*, where the *basename* is inherited from the `<job>/problem_id` parameter, the *-id#* labels the processor id for jobs run with MPI with # an integer equal to the rank of the MPI process (the root process does not contain an *id#*, nor is it present for serial jobs), the *dumpid* is a zero filled unsigned integer with `<job>/numdigits`[6], the *outid* is the string specified in `<output>/id` (if not specified, the string will be `out-#`, where `#` denotes the block number in the input file which generated the output), and the *type* denotes the output format (`bin, tab, hst, vtk, rst, pdf, pgm, ppm, fits`). Note that history dump filenames do not include a *dumpid* or *outid*. Also note that none of the "dump" formats (which output all variables, that is any of `bin, vtk, hst`) include the *outid* string.

The meaning of the parameters in the `<output>` block has already been described in §3.5.4. Below we provide more information about each of the output types.

1. **History dumps:** (*type* = `hst`) Formatted table of a variety of volume integrated values, with one line in the table created every `<output>/dt`. Thus, at the end of execution, the output file contains $tlim/dt$ lines which form a time-history of these quantities. The file is created by the function `dump_history.c`; more (or problem specific) quantities can be added by editing this file. The data is appended to the file each time the `dump_history()` function is called.

2. **Binary dumps:** (*type* = `bin`) Unformatted write of all dependent variables over all active zones. If the `dx` option is enabled by configure, then an OpenDX header file with the same name as the corresponding binary file but with the extension `.dx` will be created. This header file allows binary dumps to be read by OpenDX networks (see §9.3). A new file is created with a time interval of `<output>/dt`. Created by the function `dump_binary.c`.

3. **Tabular dumps:** (*type* = `tab`) Formatted table of all dependent variables overall all zones. A new file is created with a time interval of `<output>/dt`. Created by the function `dump_table.c`. Useful for making 1D plots.

4. **ppm output:** (*type* = `ppm`) Two dimensional images of the variable set in the output block using the `out` variable name. Global scaling can be set using the parameters `dmin` and `dmax` in the output block, otherwise each image is scaled independently. A default color palette (`rainbow`) is used, but several others are available[7]. The image could be a slice or an average over a range of grid cell indices whose orientation is determined by the values of `<output>/ix1` (or `<output>/ix2`, `<output>/ix3`), see §3.5.4. Created by the function `output_ppm.c`.

5. **pgm output:** (*type* = `pgm`) Grayscale images, written in pgm format. Scaling, orientation, and averaging used to create slice is controlled in the same way as ppm outputs. Created by the function `output_pgm.c`.

6. **Probability distribution functions:** (*type* = `pdf`) Outputs PDF of selected variables in tabular form. Created by the function `output_pdf.c`.

[6] currently fixed at four

[7] see `http://www.astro.princeton.edu/~jstone/athena_color_tables.html` for an index.

7. **Flexible image transport system output:** (*type* = `fits`) Same as ppm images, but written in FITS[8] format. Created by the function `output_fits.c`.

8. **Visualization Toolkit dumps:** (*type* = `vtk`) Similar to binary dumps, but output written in VTK[9] legacy format. Useful for 3D simulations. Created by the function `dump_vtk.c`.

9. **Restart dumps:** (*type* = `rst`) Creates a binary dump of all variables (in double precision if necessary) that can be used to restart a simulation. The start of the restart file contains the entire input file in ASCII. For jobs run in parallel with MPI, there will be one restart file per process. See the next subsection for further details.

It is important to note that **output files in `athena3.1` will always be silently overwritten!**

4.1 More on Restarts

Restart dumps (sometimes called *checkpoints*) are useful when a calculation must be continued from a previous point. The files contain enough information, and with the necessary accuracy, that a restart calculation generates *identical* data (to all significant digits) to a calculation run continuously. `Athena3.1` defines its own format for restart files. The file `restart.c` contains all the functions needed to read and write these files.

To write a restart file, add the following `<output>` block to the input file:

```
<output2>
out_fmt = rst             # Restart dump
dt      = 1.0             # time increment between outputs
```

Note that `<job>/maxout` must be greater than or equal to two in this example. The time increment `<output>/dt` is measured in problem time, and should be set to give the desired output frequency of files (usually writing one restart dump every 6 hours of wall clock time is useful). For jobs run in parallel with MPI, there will be one restart file per process, and the restarted job must use the same number of processors as there are restart files.

If the problem contains special, user-defined data, these must be added to the restart dumps. `Athena3.1` provides a mechanism for automatically adding such data. In the problem generator, two functions are provided:

```
void problem_write_restart(Grid *pG, Domain *pD, FILE *fp)
{
  return;
}

void problem_read_restart(Grid *pG, Domain *pD, FILE *fp)
{
  return;
}
```

Generally these functions are empty, but if necessary they can be used to read and write extra parameters, or set problem-specific boundary conditions on restart, etc. The problem generator `src/prob/rt.c` contains an example of usage. See the *Programmer's Guide* for more information about the structures in the argument list to these functions.

To read a restart file, the `-r` command is used on the command line:

[8] `http://fits.gsfc.nasa.gov/`

[9] `http://www.vtk.org`

```
% athena -r myfile.rst
```

Note that an input file, specified by `-i myinput`, is not needed for restarts. This is because the restart file contains the original input file, in ASCII format, at the beginning, from which all the necessary parameters are read by `par.c` on restart. This also makes restart files self-documenting: the values of input parameters used in the calculation that generated the restart file can be read with an editor. If an input file is specified along with a restart,

```
% athena -r myfile.rst -i myinput
```

then the values in `myinput` overwrite the values stored in the restart file itself. Alternatively, values in the input file can be overwritten using the command line, for example:

```
% athena -r myfile.rst time/tlim=20.0
```

Usually the `time/tlim` parameter needs to be modified on restart. For parallel jobs run with MPI, only the name of the restart file for the root (rank 0) process needs to be specified, all other processes will create their own appropriate restart filename based on this name.

4.2 Adding user-defined output expressions

Often it is useful to output a variable other than one defined in the `<output>/out` type (see §3.5.4) using one of the valid formats listed in §4. For example, one might like to create ppm images of the kinetic energy density. This can be easily accomplished using the `<output>/usr_expr_flag`. The following steps are required.

Firstly, write a function that computes the desired variable. It must be of type `Real`, and the argument list must contain the `Gas` structure and the indices of the grid cell. (For details on the information contained in the `Gas` structure, see the *Programmer's Guide*.) The following example, taken from `src/prob/field_loop.c`, computes the $z-$component of the current density at cell i, j, k.

```
static Real current(const Grid *pG, const int i, const int j, const int k)
{
  return ((pG->B2i[k][j][i]-pG->B2i[k][j][i-1])/pG->dx1 -
          (pG->B1i[k][j][i]-pG->B1i[k][j-1][i])/pG->dx2);
}
```

Next, use the function `get_usr_expr()`, included in every problem generator, to return the value computed by this function if the string in `<output>/out` has the appropriate value. As an example, the current density is computed using the function given above if the `<output>/out` string is 'J3', with the following code

```
Gasfun_t get_usr_expr(const char *expr)
{
  if(strcmp(expr,"J3")==0) return current;
  return NULL;
}
```

To create a movie of the current density, use an output block in the input file in which `<output>/out` is 'J3', and `<output>/usr_expr_flag=1`, with the other valid parameters set as appropriate (to control, for example, the time interval between outputs, min/max scaling, etc.).

4.3 Adding user-defined output formats

It is also fairly easy to add entirely new data output formats, beyond what is provided for in `athena3.1` and described in §4. This can be accomplished in two steps. The first step is to write a new output function in the file containing the problem generator. Suppose for example that the new output function is called `special_output`. It must have the following prototype in the problem generator file

```
void special_output(Grid *pGrid, Domain *pDomain, Output *pOut);
```

(For details on the information contained in the `Grid`, `Domain`, or `Output` structures, see the *Programmer's Guide.*) The second step is to enroll this function by adding a call to `data_output_enroll` anywhere in the `problem` routine. The `data_output_enroll` function has the following prototype.

```
void data_output_enroll(Real time, Real dt, int num, const VGFunout_t fun,
                        const char *fmt, const Gasfun_t expr, int n,
                        const Real dmin, const Real dmax, int sdmin, int sdmax);
```

The arguments to this function serve the following purpose.

- **time:** Time of next output, usually the current simulation time.
- **dt:** The time interval between outputs.
- **num:** The initial data output number.
- **fun:** The name of the output function (a function pointer). In the example above this is `special_output`, but it could also be say `output_ppm` for making images of some quantity.
- **fmt:** This is an optional format string used, for example, by `dump_table`.
- **expr:** The name of the function (a function pointer) of the quantity to be imaged when using an image type output routine, e.g. {`output_ppm, output_pgm, output_fits`}.
- **n**: Currently, image type outputs contain the string "out#" in their file-name where the number "#" is replaced with the argument **n**.
- **dmin, dmax:** When making image type outputs, the data can either be auto-scaled to the min/max of the each image, or scaled to the fixed values **dmin / dmax**.
- **sdmin, sdmax:** Logical flags which indicate whether to use auto-scaling (**sdmin / sdmax** = 0) or to use the fixed scales (**sdmin / sdmax** != 0).

In the simplest case the call to `data_output_enroll` could take this form, where unused arguments are set to 0, or `NULL`.

```
data_output_enroll(pGrid->time,0.1,0,special_output,NULL,NULL,0,0.0,0.0,0,0);
```

5 Specifying Boundary Conditions

As described in §3.5.6, integer flags can be used to specify a limited set of boundary conditions automatically in `athena3.1`. The actual implementation of the boundary conditions uses function pointers. The flags are used to enroll the appropriate default functions from the complete list in `/src/set_bvals.c`. Each of these functions sets quantities in the ghost zones according to the algorithm selected by the value of the flag.

The use of function pointers makes adding new boundary conditions for specific problems quite easy. For example, to add a new problem-specific boundary condition along the inner X1 boundary, the user would (1) write a new function which sets the values in the ghost zones and include it in the same file as the problem generator, and (2) enroll this new function by adding the following line at the end of the problem generator

```
set_bvals_fun(right_x1,special_bc_function_name);
```

where `special_bc_function_name` is the name of the special function written in step (1). The first argument of `set_bvals_fun` specifies the boundary on which the special function is enrolled; use `left_x1` or `right_x1` for the inner or outer x1-boundary, and similarly `left_x2` or `right_x2` specifies the inner or outer x2-boundary respectively, and `left_x3` or `right_x3` specifies the inner or outer x3-boundary respectively. As examples, users should look in the `dmr.c`, `noh.c`, and `shkset3d.c` problem generators; each contains special boundary functions enrolled in this fashion.

Users should also note the following:

1. Boundary condition flags in the input file are only required for directions in which the grid is integrated. That is, if `Nx1>1` and `Nx2=Nx3=1`, then only ibc_x1 and obc_x1 are required in the input file. The parameters ibc_x2, obc_x2, ibc_x3, and obc_x3 may be present, but their value will not be checked.

2. If the user enrolls a boundary condition routine for say the inner x1-boundary, the boundary condition flag ibc_x1 in the parameter file is not required. Again it may be in the parameter file, but its value will not be checked.

6 Problem Generators Included in `athena3.1`

A large number of problem generators are included in the `/src/prob` directory. The complete list is

```
blast.c       dmr.c             linear_wave3d.c  README       shkset2d.c
carbuncle.c   field_loop.c      lw_implode.c     rotor.c      shkset3d.c
cpaw1d.c      kh.c              noh.c            rt.c         shu-osher.c
cpaw2d.c      linear_wave1d.c   orszag-tang.c    shk_cloud.c  twoibw.c
cpaw3d.c      linear_wave2d.c   pgflow.c         shkset1d.c
```

The `README` file in this directory describes the purpose of each problem.

7 Gravity in `athena3.1`

7.1 Source terms due to a static potential

The treatment of source terms in `athena3.1` is significantly different than previous versions. Now, only source terms due to a static gravitational potential are allowed, and this treatment strictly conserves the total energy in the flow.

To include a static gravitational potential in calculations, a separate user-defined function which computes the gravitational potential given the (x_1, x_2, x_3) coordinates as an input argument must be specified in the problem generator file, for example

```
static Real grav_pot(const Real x1, const Real x2, const Real x3);
```

The arguments are the $x-$, $y-$, and $z-$coordinates at which the potential is to be evaluated. This function is then enrolled into the integrator using a function pointer, called `StaticGravPot`. This pointer is defined in `src/globals.h`. To enroll the user-defined function described in the example above, add the line

```
  StaticGravPot = grav_pot;
```

anywhere in the problem generator file. If `StaticGravPot`, is not specified in the problem generator, then no gravitational acceleration will be included in the integrator.

The problem files `pgflow.c` and `rt.c` are good examples of how to include source terms in your own applications. In particular, `pgflow.c` sets up a stringent test of gravitational source terms.

7.2 Self-gravity using FFTs

`athena3.1`includes the ability to include self-gravity of the fluid, computed using FFTs. A unique block decomposition of the FFT algorithm is used, based on the FFTW libraries. To use this feature, you must have the FFTW3.x libraries installed, you must modify the options to the compiler in the `athena3.1/Makeoptions.in` file to link to these libraries, and you must configure the code with the `--with-gravity=fft --enable-fft` options. The numerical algorithm implemented in `athena3.1`conserves the total momentum of the fluid exactly. The gravitational constant is read as a parameter in the `<problem>` block in the input file. The linear wave test problems included in the code distribution can be used to test the self-gravity option. Note the algorithm only works for periodic boundary conditions.

8 Running `athena3.1` on multiple processors using MPI

`athena3.1` is parallelized using domain decomposition based on the Message Passing Interface[10] (MPI). The code can be run on any distributed memory cluster (or any multiple processor system) on which MPI is installed using the following steps.

Firstly, during the configure step, the MPI option must be enabled via

```
% configure --enable-mpi
```

This sets precompiler macros to include the appropriate MPI code.

Next, during the compile step, the appropriate MPI libraries must be linked. Perhaps the easiest way to achieve this is to add an option to `src/Makefile.in` specifying the compiler, compiler options, linker, and libraries specfic to the the target machine. For example, using

```
% make all MACHINE=hydra
```

includes the appropriate compiler and MPI libraries for a Beowulf cluster at Princeton University called 'hydra'. By copying the options for hydra to a new target machine 'mymachine', editing them as appropriate, and then compiling with

[10]http://www-unix.mcs.anl.gov/mpi/

```
% make all MACHINE=mymachine
```

the appropriate executable should be produced.

Next, the input file for the problem of interest must be edited to add a `<parallel>` block with that specifies the desired domain decomposition. For example, the following block in the input file

```
<parallel>
NGrid_x1 = 1
NGrid_x2 = 10
NGrid_x3 = 1
```

will result in a slab decomposition with 10 slabs in the $y-$direction (a total of 10 processors are needed); while

```
<parallel>
NGrid_x1 = 1
NGrid_x2 = 2
NGrid_x3 = 3
```

will result in a pencil decomposition with two pencils in the $y-$direction and three in the $z-$direction (a total of 6 processors are needed); while

```
<parallel>
NGrid_x1 = 4
NGrid_x2 = 4
NGrid_x3 = 4
```

will result in block decomposition with four blocks in each direction (a total of 64 processors are needed). Any decomposition is allowed, although there can be no fewer than four active zones along any direction in any MPI block.

Finally, the MPI job must be run using the `mpiexec` or `mpirun` command. The number of processors used must be specified e.g. through the command line using `-np #` (where `#` specifies the number of processors to be used). The number of processors used at run time must agree with the number of MPI blocks specified in the `<parallel>` block in the input file, or `athena3.1` will print an error message and terminate. A useful script for the Parallel Batch System (PBS) which is often used to schedule jobs on parallel clusters is included in the `athena3.1/doc` directory.

Note that data generated by MPI parallel jobs will be written to separate files for each process (except for history or pdf files, which contain the appropriate MPI calls to do global sums). A useful program for joining together multiple vtk files generated by a parallel job is included in `athena3.1/vis/vtk`.

9 Visualizing output

`Athena3.1` does not come with a default graphics package. Instead, the user must decide which visualization package is best suited to their needs, output the data in a format which can be read by this package, and then proceed. As a start, rudimentary scripts for several different graphics packages are supplied with the source code; future versions may incorporate more sophisticated visualization tools. The following subsections describe useful visualization packages for `athena3.1` data files (the discussion assumes the code has already been run to produce output).

9.1 Supermongo

A popular package for making publication-quality one-dimensional plots is SM[11]. A simple SM macro that can read tabular output from `athena3.1` is provided in `athena3.1/vis/sm`.

9.2 IDL procedures

IDL (Interactive Data Language)[12] procedures that can read both the binary and VTK dump files and make plots are included in `athena3.1/vis/idl`. To run these procedures IDL must be installed on the system. From the `athena3.1/bin` directory, use the following to read a binary file and make some one-dimensional plots:

```
% idl
IDL> .run ../vis/idl/pltath.pro
IDL> nine_plot,'Brio-Wu.0040.bin',1
```

A variety of potentially useful procedures are included in the `pltath.pro` file.

9.3 OpenDX networks

The OpenDX[13] package can read `athena3.1` binary dump files, provided the `.dx` header files exist. This requires `athena3.1` be configured with the dx option enabled and of course OpenDX must be installed on the system. v2.0 of Athena included an example network to read binary files, currently this network has yet to be extended to `athena3.1`.

9.4 VTK

We have found the VisIt package[14] useful for plotting 3D data sets. The VTK legacy format produced by `athena3.1` can be read by VisIt. For data created with executables parallelized with MPI, a C code that joins multiple files into one is provided in `athena3.1/vis/vtk`. This is useful for jobs run on massively parallel clusters.

9.5 2D animations

`Athena3.1` can output 2D images that can be displayed directly, or easily turned into animations. For example, if a series of ppm images of a single variable have been created, they can be displayed either using ImageMagick:

```
 % animate *.ppm
```

or, alternatively, `xanim` (which requires converting the ppm images to FLI format):

```
 % ls Wind*ppm > list1
 % ppm2fli -g80x80 list1 Wind.fli
 % xanim Wind.fli
```

[11] `http://www.astro.princeton.edu/~rhl/sm/`
[12] `http://www.rsinc.com`
[13] `http://www.opendx.org`
[14] `http://www.llnl.gov/visit`

10 Examples of Running `athena3.1`

10.1 The `athena3.1` Benchmark

To test the installation of `athena3.1`, a benchmark can be run automatically using the Makefile. This benchmark consists of a linear wave convergence test on a grid of 512 zones; it computes and outputs the L1 error norm in the fast magnetosonic wave compared to the analytic solution. If the benchmark fails to run, or if the resulting error norm is large, then something has gone wrong in the installation or in the compilation of the code.

To run the benchmark, use the following commands (in the `athena3.1` root directory; these steps assume the configure script has already been generated with `autoconf`).

```
% configure
% make all
% make test
(cd tst/1D-mhd; ./run.test)
zone-cycles/cpu-second = 3.067215e+05
zone-cycles/wall-second = 3.055470e+05
L1 norm for density: 6.333390e-11
```

The `zone-cycles/cpu-second` is a useful measure of the code performance, it corresponds to the number of grid cells updated per cpu second. This of course depends on the physics included, the geometry of the problem, as well as the processor used for the calculation. The values reported above are for a 3.08Ghz Intel Xeon processor. The error norm is absolute, and should never be larger than 10^{-10}.

10.2 Running a 1D test problem with `athena3.1`: The Brio & Wu shocktube

As an example of how to configure, compile, and run `athena3.1` and visualize the output for a one-dimensional test problem, we show the steps required to run the Brio & Wu shocktube. Assuming the code has already been installed (see §2 or §3), the first step is to configure:

```
% cd athena3.1
% configure --with-problem=shkset1d
```

The configure script will print a variety of diagnostic statements during this step. Next, the code must be compiled:

```
% make all
```

The default compiler options will print diagnostic statements. Then the code can be run in the `athena3.1/bin` directory:

```
% cd bin
% athena -i ../tst/1D-mhd/athinput.brio-wu
```

The code will print information about every timestep as it executes. It should generate 40 binary dumps named `Brio-Wu.*.bin`, 40 tabular dumps named `Brio-Wu.*.tab`, as well as a history file `Brio-Wu.hst`. There are a variety of ways that the data in these files can be visualized; one way is to use the IDL scripts included in `athena3.1/vis/idl`.

```
% idl
IDL> .r ../vis/idl/pltath.pro
IDL> four_plot,'Brio-Wu.0040.bin'
```

The resulting plots which should now appear on the screen is shown in Figure 1.

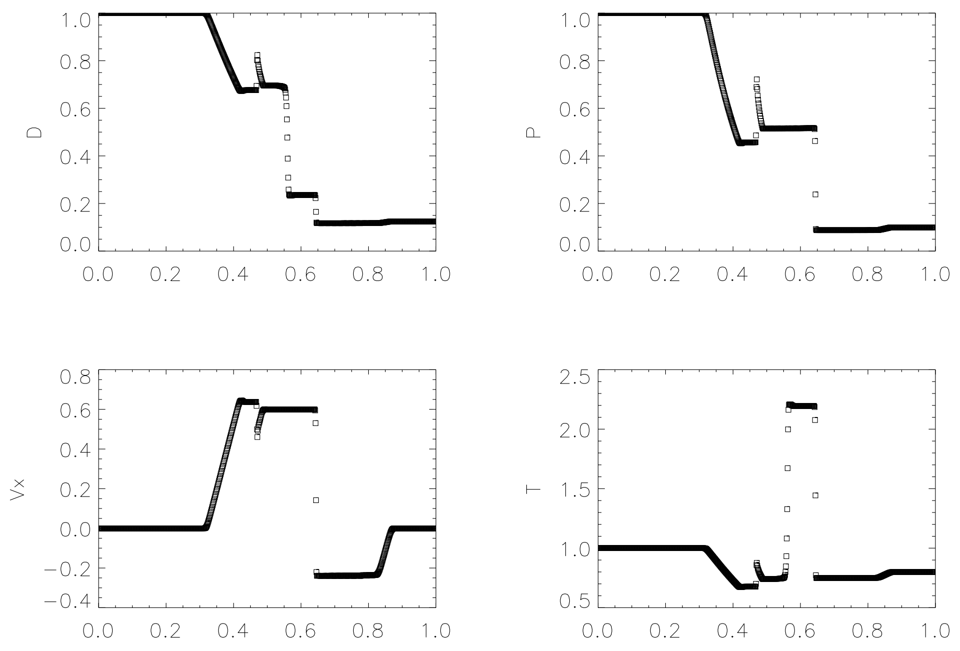

Figure 1: Results from Brio & Wu shocktube test problem plotting the using IDL procedure `four_plot`

10.3 Running a 2D test problem with `athena3.1`: The Orszag-Tang vortex

As an example of how to configure, compile, and run `athena3.1` and visualize the output for a two-dimensional test problem, we show the steps required to run the Orszag-Tang vortex test using the third order algorithm and the HLLD fluxes. Again, assuming the code has already been installed (see §2 or §3), the first step is to configure:

```
% cd athena3.1
% configure --with-order=3 --with-problem=orszag-tang --with-flux=hlld
```

A variety of diagnostic statements will be printed during this step. Next, the code must be compiled:

```
% make all
```

The default compiler options will print diagnostic statements. Then the code can be run in the `athena3.1/bin` directory:

```
% cd bin
% athena -i ../tst/2D-mhd/athinput.orszag-tang
```

The code will print information about every timestep as it executes. On a 3.08 GHz Xeon processor, the calculation takes about 4 minutes to complete. It should generate 100 binary dumps named `OrszagTang.*.bin`, 250 ppm images of the gas pressure named `OrszagTang.*.P.ppm`, 250 ppm images of the density named `OrszagTang.*.d.ppm`, as well as a history file `OrszagTang.hst`. There are a variety of ways that the data in these files can be visualized; one way is to use the IDL scripts included in `athena3.1/vis/idl`. To make a contour plot of the pressure at time $t = 0.5$, use the following:

```
% idl
IDL> .r ../vis/idl/pltath.pro
IDL> readbin,'OrszagTang.0050.bin'
IDL> contour,p,nlevels=30,/isotropic,xstyle=1,ystyle=1
```

The resulting plot which should now appear on the screen is shown in Figure 2.

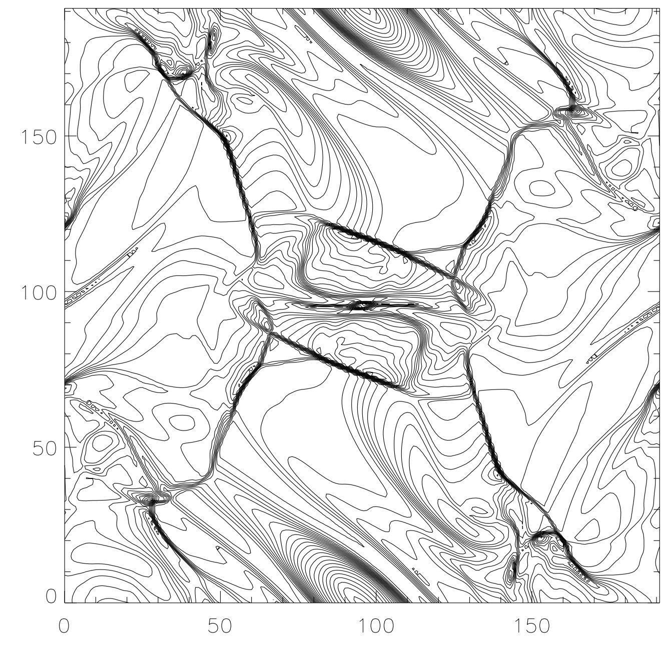

Figure 2: Contour plot of the gas pressure at time $t = 0.5$ from the Orszag-Tang test problem plotted using IDL.

It is also interesting to watch an animation of the density or pressure in the problem. This can be done a variety of ways. The simplest is to use the ppm images generated directly by the code. For example, if ImageMagick is installed on the system, try

```
% animate *.P.ppm
```

The density can be animated in a similar fashion.

10.4 Running a 3D test problem with `athena3.1`: Advection of a field loop

As an example of how to configure, compile, and run `athena3.1` and visualize the output for a three-dimensional test problem, we show the steps required to run the advection of a field loop test. Again, assuming the code has already been installed (see §2 or §3), the first step is to configure:

```
% cd athena3.1
% configure --with-order=3 --with-problem=field_loop --with-flux=hlld
```

A variety of diagnostic statements will be printed during this step. Next, the code must be compiled:

```
% make all
```

The default compiler options will print diagnostic statements. Then the code can be run in the `athena3.1/bin` directory:

```
% cd bin
% athena -i ../tst/3D-mhd/athinput.field_loop4 grid/Nx1=64 grid/Nx2=64 grid/Nx3=64
parallel/NGrid_x1=1 parallel/NGrid_x2=1 parallel/NGrid_x3=1
```

Note the input file for the field loop test rotated at an angle is run (`iprob=4` in the `<problem>` block). Since the default input file for this problem uses a 128^3 grid, with a block decomposition on 8 processors using MPI, the command line is used to overwrite the input parameters to use a 64^3 grid and only one processor.

It will take about 30 minutes for this calculation to complete on a 3.08 GHz Xeon processor. The run should generate three VTK files, 250 ppm images of the current density on a $x-y$ slice, and a history file. A plot of an isosurface of the magnetic energy density made with the VisIt package is shown in Figure 3. Alternatively, the IDL procedure `readvtk` in `vis/idl/pltath.pro` could be used to to read the data, and then one of the plotting routines in IDL could be used to display the data.

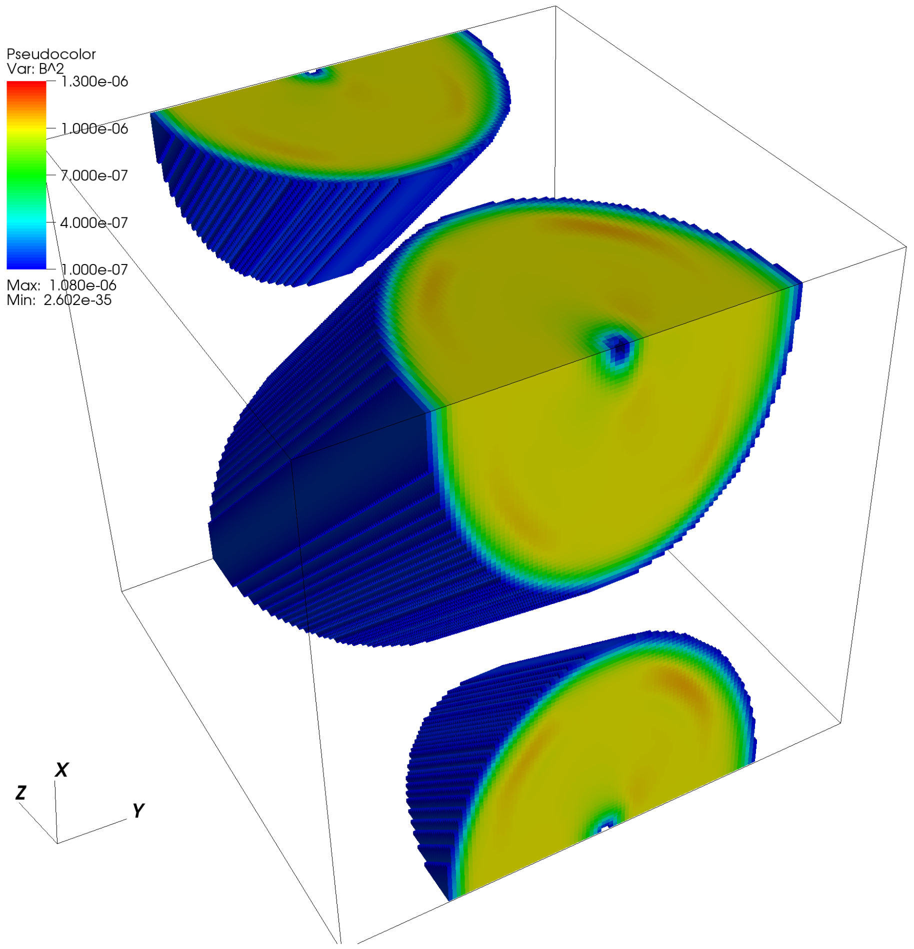

Figure 3: Surface plot of the magnetic energy, generated by VisIt, for the advection of a rotated field loop in 3D.

10.5 The `athena3.1` Test Suite

Each of the problem generators in the `/src/prob` subdirectory sets up another test problem for `athena3.1`. Further descriptions of these tests, and examples of results from running `athena3.1`, can be found in the Athena code web pages, and in the Method papers.

11 Running New Problems

The real utility of the `athena3.1` code is as a solver for new problems (i.e. problems that are not initialized by the set of problem generators included in the source code distribution). For new problems, the following steps are required.

1. Write a new function that initializes the problem. This function must be of type `void`, have the name `problem`, and have as arguments pointers to the `Gas` and `Domain` structures. The function must be contained in a file in the `/src/prob` directory.

2. Write new functions called `Userwork_in_loop` and `Userwork_after_loop` (which may be no-ops if not needed) and include them in the file containing `problem`. As the names suggest, these functions can be used to perform special problem-dependent work in or after the main loop (see `linear_wave.c` for an example).

3. If special purpose boundary conditions are needed, write special functions that implement them, and enroll them using the function `set_bvals_fun` (see §5).

4. If special purpose data output is needed, write special functions that implement them, and enroll them using the function `data_output_enroll` (see §§4.2 and 4.3).

5. Once the above is complete, configure and compile the code using the appropriate physics options, and including the new problem generator using `--with-problem=`*new-name*.

It is likely the *Programmer's Guide* will be needed to write a new problem generator to understand the data structures and names used in `athena3.1`. As a start, the problem generators in `/src/prob` can be used as templates.