

The Fortran 95 version of Athena

John F. Hawley, Jacob B. Simon
Department of Astronomy
University of Virginia
PO Box 400325
Charlottesville, VA 22904

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

I don't know what language I will be programming in in the future, but I know it will be called Fortran.

1 Introduction

Athena is a grid-based code for astrophysical gas dynamics developed with support of the NSF Information Technology Research (ITR) program. This document describes the Fortran 95 (F95) version, called *Athena3D*. Further information about the code can be found in the *Athena Programmer's Guide*, and the *Athena User's Guide* which describe the publicly released version of *Athena* written in the C language. The algorithm is described in Gardiner & Stone 2005, *Journal of Computational Physics*, vol. 205, 509-539.

Why a F95 version? If you are asking this question, well then perhaps this particular implementation isn't for you. But for a large portion of the research community, Fortran remains the language of choice for scientific applications and is likely to be the language that many astrophysicists are most comfortable with. Note, however, that this implementation of *Athena* is written in F95, which may seem almost as unfamiliar to aging Fortran 77 users as a language that doesn't share the Fortran name.

F95 is a powerful, modern language well-suited to vector mathematics. This makes it ideal for the implementation of a finite-difference physics code. F95's array syntax allows for compact, easy to read code. The addition of dynamic memory management removes many of the shortcomings of F77 compared to, say, C. F95 permits the creation of user-defined types, and F95's module construct allows for a simple and highly adaptable method to compartmentalize tasks and ensure interoperability. The Fortran freeformat style breaks the rigid rules of column number that date back to the era of punchcards.

The C version of *Athena* has been the primary development version of the *Athena* algorithm. In developing the Fortran version of *Athena*, we have tried to implement the best of the algorithms developed in the C version, but not all of the C version's functionality has been duplicated. The F95 version, like the C version, computes the solution of a 3-D Euler system with source terms using either hydrodynamics or MHD, with either an adiabatic or isothermal equation of state. The geometry is Cartesian (x, y, z) , and the grid spacing (dx, dy, dz) is constant, although dx , dy , and dz need not

be equal. Source terms (e.g. forces) can be applied in any of the three directions. Precision (4 byte or 8 byte words) can be specified with ease.

The *Athena* project was conceived with several motivations in mind. First and foremost was the perceived need for a second-order multidimensional code for MHD using the flux-conservative Godunov approach. Such a code could complement the widely-used *ZEUS* code which is based on operator-split finite differencing.

A second aim was to develop a code to serve not as a black box, but as a starting point for groups wishing to construct their own research tools. In this respect, it should be noted that the research potential is somewhat limited at the present by the restriction that the coordinates be Cartesian and equally spaced. Using this code for research applications will entail some amount of adaptation; at the very least users are expected to construct their own problem-specific module.

Another use of this code is mainly pedagogy. Students of computational science should have available a code that is relatively easy to use, likely to compile on their system without struggle, and clearly written. Because of the pedagogical nature of the code, certain goals were adopted for the F95 version of *Athena*. These include:

- Clarity in coding style for ease of understanding.
- No pre-processing of the fortran files. Instead of `#IFDEF` blocks in the source which tend to make the code difficult to read, options are chosen explicitly by the user and assembled at compile time.
- Strict adherence to F95 standards, to try to ensure the code will compile and run on any platform that has an f90/95 compiler.
- Ease of use for computing a variety of standard 1D, 2D, and 3D hydrodynamics and MHD test problems.
- Consistency in algorithm with the C version of the *Athena*. However, the emphasis is on the “best practices” of that code, rather than reproducing all the options or aspects available.
- Ability to run in either serial or parallel mode. Parallel mode is implemented with MPI calls and is designed for ease of use and understanding.

The following characteristics are of secondary importance for this implementation:

- Code speed: clarity and simplicity are valued over performance.
- Implementing all of the options in the C version: not all the flux routines have been implemented, nor all of the energy conservation options, the EMF integration options, or source functions.
- Consistency in input/output, code structure, etc. with the C version has largely been abandoned, except in broad terms.

Thus this is not strictly speaking an F95 implementation of the C version of *Athena*. Rather it is an independent implementation of the algorithm found therein. Since the goal of this project is not to provide a black box for computation, but to provide a resource for learning and further independent code development, a user can work with whatever suits them best, and, if they so desire, code their own implementation in whatever language they prefer.

2 Quick guide

This section will provide a brief summary of some of what one needs to use the *Athena3D* code. It is not comprehensive and probably can't be completely understood without independent reference to the code itself.

We begin with a brief description of the serial version of the code.

System requirements: You need an f90/95 compiler. That is really all you need for the serial version. The parallel version will require a parallel system and the MPI libraries. A Makefile is supplied to ease the process of compiling. It is assumed you are reasonably familiar with simple makefiles. User choices in code configuration are made prior to compiling. This includes choosing whether to compile the serial version or the parallel version. A perl script, `buildathena` is also supplied to further simplify the build operation if one is so inclined.

Select Precision: First, decide if you want single or double real precision. In `MODULE athenadefs` set `r_kind = real8` for double precision, or `r_kind = real4` for single precision. This module also specifies how many ghost zones the grid will use, and defines the types `gas` and `grid`.

Select Physics: There are four “flavors” of physics available to use: hydrodynamics or magnetohydrodynamics, and adiabatic or isothermal equation of state. To select the desired physics, enter the desired physics modules in `MODULE physics`. You must enter the “defs” module and “eigen” module

corresponding to the physics you wish to implement. For example, if you wish to use an adiabatic equation of state for magnetohydrodynamics, then type `USE mhddefs` and `USE mhdeigen` in `MODULE physics`. Trying to use more than one type will result in a compile error.

The “defs” module defines basic types and functions associated with a particular type of physics. The fundamental conserved fluid vector for isothermal hydrodynamics consists of density and three momenta; adiabatic hydro adds the total energy. MHD requires three components of magnetic field. In *Athena*, we also include face-centered as well as cell-centered magnetic field components. The work in the code is done on vectors whose right-most rank index corresponds to the variable. There are conserved variables and primitive variables and routines to shift between the two. For example, for adiabatic MHD there are 8 variables (conserved or primitive) and 7 eigenfunctions. Typically *U* is used to label the conserved variables and *W* for the primitive. The code also makes use of two logical variables, `MHD` and `ADIABATIC`. These are set to `.true.` or `.false.` as appropriate within the “defs” modules used in the physics module.

The “eigen” modules contain routines for calculating eigenvalues and eigenmatrices for the specified type of physics.

Select integration options: The main integration routine has a name like `integrate3D_1step`; there is a routine for each dimensionality. This routine evolves the fluid variables over one timestep. Schematically, the routine computes a left and right value at every interface, uses those values to compute a flux, then updates the conservation equation using those fluxes. Inside this routine, one can choose the order of interpolation and the type of emf integration. Specifically, the interpolation of the left and right states is carried out in `MODULE lr_states`. To select first order interpolation, say

```
USE lr_states, get_states => lr_states_1storder
```

To get second order interpolation,

```
USE lr_states, get_states => lr_states_2ndorder
```

and to get third order interpolation,

```
USE lr_states, get_states => lr_states_3rdorder
```

The emf integration is selected from `MODULE integrate_emf` as

```
USE integrate_emf, integrate_emf_corner => integrate_emf_corner_upwind
```

One other emf integration routine is currently available (`integrate_emf_corner_zero`). There is also a selection given for the fluxes module in the `USE fluxes` statement. The default flux routine (the Roe scheme) works well for hydro or mhd. Currently no other flux routine is implemented.

Select Problem: The initialization routine is interfaced to the main routine through the call

```
call problem(u,grid0)
```

where the arguments are

```
type (gas) :: u
type (grid) grid0
```

The problem subroutine is part of the `MODULE problem_definition`. This module contains all of the elements that are unique to a particular problem, including source terms and boundary conditions.

Specify Source Terms: Many problems require the addition of source terms. For example, a Rayleigh-Taylor instability calculation requires one to include a constant gravity in the vertical direction. *Athena* has a mechanism for including the user defined source terms. In the Fortran version, the source functions are located in the `PROBLEM_DEFINITION` module and are thus associated with the particular problem one is solving. Within this modules there are logical variables, e.g., `xsource`, which can be set to `.true.` to activate the use of the source functions. There are two forms of the source functions, one appropriate for the primitive variables, the other appropriate for the conservative variables.

As a specific example, consider the Rayleigh-Taylor problem. The file that contains the `PROBLEM_DEFINITION` is `rayleigh-taylor.f90`. This assumes that the vertical direction is to be the y direction, and there is a constant gravitational force g in this direction. In this specific case we wish to add the primitive source term $-g$ to the v_y term, and a conservative source term $-\rho g$ to the y -momentum equation, and $-\rho v_y g$ term to the conserved energy equation (assuming that the adiabatic equation of state is employed). At the top of the `PROBLEM_DEFINITION` module we find

```
logical :: ysource = .true.
```

Near the end of this module, we find the functions that define the source terms. The nonzero elements of the source term array for the 3D Rayleigh-Taylor problem are set as

```

SUBROUTINE prim_source_y
source(:,2) = -gravity

SUBROUTINE cons_source3D_y
source(:,:,:,3) = -cons(:,:,:,1)*gravity
if (adiabatic) source(:,:,:,5) = -cons(:,:,:,3)*gravity

```

Note that the primitive source routine acts on the 1D primitive vector which is ordered so that the velocity in the direction of the sweep comes in the 2nd slot, whereas the conservative source term acts on the full conserved vector. Also note, there are separate 1D, 2D, and 3D versions of the source term routines.

This example is fairly straightforward, but others might not be. One often might wish to add a force out of the plane, or a coriolis force, or some other term that isn't obviously oriented in the x or the y direction. The question of whether a term is an x source or a y source is not whether it is operating in one direction or another but how it is incorporated into the computation of the fluxes that update the conserved state vector. How the force terms should be included is not always clear, and some choices can lead to nonphysical solutions. The main point is that one must choose only one direction in which to add the source term so as to not include the same force twice.

NOTE: The present version of the *Athena3D* code does not incorporate the “conservative potential” force term in the *integrate_1step* routines. This method is an alternative approach to adding source terms, appropriate for, as the name implies, forces that derive from a conservative potential. See the C version to see how this implementation is done.

Compiling: The easiest way to compile the code is to use the `buildathena` script provided. Its usage is as follows:

```

Usage: buildathena --prob=problem name --type=serial or parallel
--fluid=hydro or mhd --therm=iso or ad
      (Defaults are serial, hydro, and iso)

```

The script invokes `make`. Note that for this script to work, the problem definition file must be in the `prob/` directory, which is separate from the `src/` directory where the *Athena3D* source code is contained. You may need to edit the Makefile to work on your system.

The Makefile uses the environment variable `PROB` to name the file to use for the initialization routine and the variable `TYPE` for either a serial or

parallel code. Therefore, if one chooses not to use the `buildathena` script to compile the code, the `PROB` and `TYPE` variables must be defined in the shell. For example, to compile the Brio-Wu shock tube test one would have (in, for example, bash)

```
export PROB=briowu
export TYPE=serial
```

Note that for this compilation procedure to work, the problem definition file (`briowu.f90` in our example) must be in the `src/` directory.

Runtime Input: Runtime input is done through namelists. The main namelists are assumed to be in the file `athena.input`. As nice as it might be to have command line input, routines such as `iargc()` and `getarg()` are not part of the F95 standard, and are therefore not implemented. There are, however, frequent extensions on many systems. You may implement them to suit. As an example, here is the content of the `athena.input` file appropriate for the Sod shock tube:

```
&JOB
  problem_id      = 'SOD'
  ,restart_flag   = 0
  ,restart_file   = 'SOD.0000000'
/
&TIME
  CourNo         = 0.4
  ,dt_res        = 1.0
  ,nlim          = 200
  ,tlim          = 0.2
  ,wlim          = 24.0
  ,dt_hst        = 0.1
  ,dt_bin        = 0.1
  ,dt_tab        = 0.1
/
&GRD
  nxzones        = 100
  ,xmin          = 0.0
  ,xmax          = 1.0
  ,ibc_x         = 2
  ,obc_x         = 2
  ,nyzones       = 1
```

```

,ymin          = 0.0
,ymax          = 1.0
,ibc_y         = 2
,obc_y         = 2
,nzzones       = 1
,zmin          = 0.0
,zmax          = 1.0
,ibc_z         = 2
,obc_z         = 2
/
&PARALLEL
  Ngrid_x      = 1
  Ngrid_y      = 1
  Ngrid_z      = 1
/
&PROB
  gamma        = 1.4
  ,dl          = 1.0
  ,pl          = 1.0
  ,vxl         = 0.0
  ,vyl         = 0.0
  ,vzl         = 0.0
  ,dr          = 0.125
  ,pr          = 0.1
  ,vxr         = 0.0
  ,vyr         = 0.0
  ,vzr         = 0.0
/

```

The block `JOB` sets the name of the problem, which will be used in all the files generated by the job, the restart flag (1 for restart, 0 for new problem), and a restart file name used if the restart flag is 1.

The block `TIME` sets the Courant number, the time frequency for restart file output (`dt_res`), history file output (`dt_hst`), binary file output (`dt_bin`), and table file output (`dt_tab`), as well as the limit to the number of timesteps before stopping (`nlim`) and the endtime of the problem (`tlim`). `wlim` is the wall-clock limit in hours, which is useful if you want the code to run for a certain length of time and then exit gracefully with a restart dump. The Courant number should be equal to 0.4 or less for multidimensional problems; this restriction is associated with the “6-way” CTU implementation

for 3D integration. In 1D one can use a larger Courant number, but it must always be less than one.

The block `GRD` defines the grid that will be used in the problem: number of zones in the x , y and z direction (`nxzones`, `nyzones`, and `nzzones` respectively), minimum and maximum boundaries of the grid in each direction (e.g., `xmin` and `xmax`), and the boundary condition flags. The physical boundary conditions (as opposed to internal boundaries in parallel domain decomposition) are implemented in the `PROBLEM_DEFINITION` module via subroutines with names like `set_bvalx`. The type of physical boundary conditions are set by the flags in the input file. The names and supported values of the flags are:

```
ibc_x = Inner Boundary Condition for x
obc_x = Outer Boundary Condition for x
ibc_y = Inner Boundary Condition for y
obc_y = Outer Boundary Condition for y
ibc_z = Inner Boundary Condition for z
obc_z = Outer Boundary Condition for z
```

with values 0 = do nothing, 1 = reflecting; 2 = outflow; 4 = periodic, 5 = special (user defined).

The namelist `PROB` can be used for the problem subroutine, if desired. Here the `PROB` block provides the left and right initial values used in the `sod.f90` initialization routine. This is the only block in the `athena.input` namelist that should change. Normally, it is read in by the `PROBLEM_DEFINITION` module. There is nothing to prevent you from adding additional namelist blocks in a `PROBLEM_DEFINITION`.

Output Routines: Obviously, getting one's data out from the code in a usable form is an important thing, but the form that one might prefer the data to be written in can be both problem and user dependent. In *Athena3D* all the output routines have been included in `MODULE data_out`. These routines are called at the time frequencies specified in the `TIME` block of the namelist file. The forms of the possible output are tabular form (ascii file) written by `SUBROUTINE tabout`, full binary output files written by `SUBROUTINE binout`, and a time history file written by `SUBROUTINE history`. The latter file, at present, just contains a record of time, timestep, and global measures of conserved values, but one can modify it to write out any value whose time dependence one wishes to study. The `tabout` routine is mainly set up to write 1D slices of code variables. This routine currently only works for the serial version of the code. The binary output is the most generic, writing out

the spatial grid coordinates (x , y , and z), the conserved state variables, and the primitive variables. For convenience, an IDL procedure file is included which will read this binary file into IDL. A manual page for this routine is included as an appendix.

Restart and Checkpointing: In the namelist block JOB, the `restart_flag` variable is set to 1 for restart, and 0 for new problem. The `restart_file` is the name of the restart file to be used if the restart flag is 1. In the TIME block, the term `dt_res` is the time interval at which restart files will be written. These will accumulate and they can be rather large. So, don't write too many restarts if you don't have to.

Parallel Version: The *Athena3D* code is designed to employ domain decomposition using MPI. Because of the modularity of the code, only a few routines are different between the serial and the parallel version, most notably the main program routine and a parallel boundary routine. The code must be compiled as the parallel version and the MPI libraries linked in. On most systems, there is a special fortran compiler command such as `mpif90` that handles the library linking automatically.

When running a problem in parallel, the total number of physical zones in the problem remains designated by the `nxzones`, `nyzones`, `nzzones` parameter in the GRD block of the namelist file. The problem is divided into subdomains according to the PARALLEL block in this file, which specifies the number of separate grids to use in each direction. For example,

```
&PARALLEL
  Ngrid_x      = 2
  Ngrid_y      = 4
  Ngrid_z      = 8
```

will break up the total volume twice in the x direction, 4 times in the y direction and 8 times in the z direction for a total of 64 subdomains, requiring 64 MPI processors. The decomposition is done automatically. The code tries to allocate the same number of zones to each subdomain, but it is not necessary to ensure that, say, `nzzones` can be evenly divided by `Ngrid_z`. The code will distribute any "leftover" zones among the processors.

The binary restart files produced by the parallel version make no reference to how the problem was broken up into subdomains. This means that a simulation can be restarted with a different number of processors, if desired. Similarly, the binary data output file looks the same whether the parallel or serial version is run.

Note that this procedure won't scale to huge problems since it requires gathering data onto one processor that writes the file. For really big problems, the size of a single variable could well exceed the available memory on that processor. It was decided that the simplicity of this approach outweighs the drawbacks, since, for most routine applications, there should be no problem. If you are planning to run truly large problems, you are probably capable (or should be) of rewriting these routines to another form.

3 Modules

F95 codes are based on modules. A module is a collection of declarations and subprograms designed to perform certain specific tasks and to associate data with that specific task. In this section, we list the characteristics of each of the modules that make up the *Athena3D* code.

MODULE athenadevs: This module contains the globally required parameter declarations for the code. The parameter `r_kind` allows the user to choose between single and double precision reals. `NGHOST` specifies the number of boundary “halo” or “ghost” zones required at a grid edge. The minimum number depends on the order of the interpolation used. Third-order interpolation requires 4 ghost zones. Since computing values in ghost zones does require cpu time, one wants to use no more than is necessary. Using fewer than required, however, will produce garbage.

The defined type `gas` consists of a collection of rank-3 pointers for the fundamental conserved variables of the code, specifically, density, `d`; energy, `e`; momenta, `mx`, `my`, `mz`; cell-centered magnetic fields, `bx`, `by`, `bz`; and interface-centered magnetic fields, `bxi`, `byi`, `bzi`. Which pointers are allocated will depend on the type of physics used. For example, the magnetic field pointers are not allocated for hydrodynamic problems.

The defined type `grid` is composed of information associated with the grid. This includes the grid itself, `x`, `y`, `z`; the grid spacing and its inverse, `dx`, `dy`, `dz`, `dxi`, `dxi`, `dzi`; the timestep, `dt`; simulation time; the number of physical zones in each dimension, `xzones`, `yzones`, `zzones`; the beginning and ending indices for the physical zones, `is`, `ie`, `js`, `je`, `ks`, `ke`; and the boundary value flags for each direction. Note that we hold to the convention that indices `i`, `j`, `k` are used for the `x`, `y`, `z` directions respectively.

MODULE boundary: This module contains code to implement the standard boundary conditions: reflecting, periodic and outflow. One of the design aims for the code was to extract all treatment of boundaries from the main components of the code. That requires that boundary conditions be applied to the fundamental conserved variables rather than applied to the fluxes at a boundary, for example. Note that “outflow” conditions could be implemented several ways; the method implemented here is very simple and assumes zero slope at the boundary. There are two choices in how one can implement “reflecting” boundary conditions on the magnetic field. These are reflection or antireflection. The reflection conditions implemented here have the field component perpendicular to the boundary set to zero.

MODULE data_out: This module handles the output and restart functions for the *serial version* of the code. SUBROUTINE `data_dump` is called from the main routine and controls what output subroutines are called. SUBROUTINE `tabout` produces tabular ascii data (only works for the serial version). SUBROUTINE `binout` dumps the entire grid into a binary file. Note that the logical parameter `DUMP_GHOST` can be set to `.true.` if one wants to dump the ghost zones as well as the physical zones (only works for the serial version). This option can be useful for debugging. SUBROUTINE `history` is designed to give globally integrated values (e.g., total energy, mass, magnetic energy, etc.) as a function of time in an ascii columnar data format. SUBROUTINE `read_restart` and `write_restart` do what their names say: they handle the input and output for restart dumps. MODULE `pdata_out` basically has the same function as **MODULE data_out** but is used for the parallel version. In particular, it includes all of the MPI calls necessary to assemble data from all the processors.

MODULE fluxes: This module takes input left and right states and returns a flux. The module could contain any number of alternative flux routines which would be selected by the following statement in the *integrate_1step* routines:

```
USE fluxes, get_fluxes => flux_roe
```

At present the `flux_roe` routine is the only one implemented. But if one is so inclined, alternatives can be placed in this module.

MODULE integrate_emf: This module takes the face-centered emfs computed from the flux routine and returns a corner-centered emf for use in the constrained transport evolution. The module can contain any number of alternative routines to do this. In the present code, only two are implemented, the “corner upwind” scheme and the “corner zero” scheme. Tests have shown the corner upwind scheme to be the best in most circumstances. The choice is selected by the `USE` statement in the *integrate_1step* routines. In 3D for example, one specifies the routine for each corner:

```
USE integrate_emf &
  , integrate_emfx_corner => integrate_emfx_corner_upwind &
  , integrate_emfy_corner => integrate_emfy_corner_upwind &
  , integrate_emfz_corner => integrate_emfz_corner_upwind
```

MODULE lr_states: This module takes input of zone-centered primitive variables and returns a left and right value for each zone interface. Three

versions are implemented, including first order upwind, 2nd order linear, and 3rd order parabolic. The choice is selected through the `USE` statement in the `integrate_1step` routines:

```
USE lr_states, get_states => lr_states_3rdorder
```

Again, other algorithms can be implemented and inserted here to compute left and right states.

MODULE parallel_mpi: This module handles the MPI messaging required for domain decomposition. MPI calls require knowing the real type which is designated by integers `MPI_REAL8` or `MPI_REAL4`. The values of these integers are implementation specific. So, we use the variable `MY_REAL` which is set according to the type of real designated by the user at compile time. Buffers are allocated and packed with the variables to be swapped at internal boundaries. The messaging is done with `MPI_SendRecv`. A given domain has the identity of its neighbor processors stored in integer variables `left`, `right`, `back`, `front`, `above`, `below`, in the x , y , and z directions respectively. If the boundary is a physical rather than an internal boundary, then the value of the neighbor identifier is set equal to `MPI_PROC_NULL`. Physical boundaries will have their ghost zones set by the boundary condition routines in the `MODULE problem_definition`.

Note that this implementation does a certain amount of copying back and forth and uses blocking sends and receives. The object here was simplicity and safety rather than speed.

MODULE physics: This module is simply a place holder for the choice of physics to be used. There are four physics choices: isothermal hydrodynamics, adiabatic hydrodynamics, isothermal MHD, and adiabatic MHD. Each choice has two modules associated with it. For example, adiabatic hydrodynamics has `MODULE hydrodefs` and `MODULE hydroeigen`. The “defs” type of module contains the information that defines the type of data structure required. It includes various routines for converting primitive to conservative variables and back again, extracting a 1D or 2D slice from the full 3D array, and setting various standard source functions. The “eigen” type of module contains the routines that compute the primitive and conservative eigenvalues. One sets the physics for the whole code by including the appropriate `USE` statements in the physics module.

MODULE problem_definition: This module contains everything specific to a particular problem, including the initialization, boundary condition handling, and source functions. For many problems, these things might be

very simple. For others, they might be very complicated. It is recommended that one builds a new problem using the preexisting problem definition files as templates.

4 Routines

main and **pmain**: The main program for *Athena3D*, a serial and a parallel version.

athena_abort and **pathena_abort**: These routines provide for a clean exit from the code when an error condition is identified. This is more important for the MPI version since a call to `MPI_ABORT` is made to ensure all processes shut down.

evolve1d, **evolve2d**, and **evolve3d**: These routines take the full gas array and arrange them as needed to integrate forward in time. Since the gas array is a collection of rank three pointers, these routines reduce the rank to the appropriate value for 1D and 2D problems. They rely on `gas_to_cons` conversions that are defined in the “defs” modules, e.g., `MODULE hydrodefs`.

init_dt: This subroutine computes the initial timestep using the zone centered variables to compute maximum wavespeeds. Thereafter, the code computes the new timestep using the maximum wavespeeds returned by the eigenvalue routines.

integrate1d_1step, **integrate2d_1step**, and **integrate3d_1step**: These routines take an input array of conservative gas variables and evolve the variables forward by one timestep. The algorithm is the CTU scheme for multidimensional problems. Specifically, the “6-way” scheme for 3D is implemented.

setup_grid and **psetup_grid**: These routines initialize the grid zone information contained in the `TYPE grid` variable using the information read in by the main program in the `GRD` namelist block. The `SUBROUTINE psetup_grid` is for the parallel version of the code and initializes a particular processor’s subdomain.

5 Example

As a concrete example, let's consider running the Rayleigh-Taylor problem in 2D using adiabatic hydrodynamics. The easiest way to set this up is to run the `buildathena` script as follows:

```
./buildathena --prob=rayleigh-taylor --type=serial --fluid=hydro --therm=ad
```

The file `rayleigh-taylor.f90` must exist in the `prob/` directory. The script creates a run directory, `bin/`, creates the file, `athena3d` which is a serial executable for this problem, and copies the default `athena.input` file for the Rayleigh-Taylor problem to the `bin/` directory. The `athena3d` file is also copied to `bin/`. The input file will look like:

```
&JOB
  problem_id      = 'RT'
  ,restart_flag   = 0
  ,restart_file   = 'RT.0000000'
/
&TIME
  CourNo         = 0.4
  ,dt_res        = 1.0
  ,nlim          = 1000000
  ,tlim          = 10.
  ,wlim          = 24.0
  ,dt_hst        = 0.1
  ,dt_bin        = 0.1
  ,dt_tab        = 1.0
/
&GRD
  nxzones        = 300
  ,xmin          = -0.25
  ,xmax          = 0.25
  ,ibc_x         = 4
  ,obc_x         = 4
  ,nyzones       = 900
  ,ymin          = -0.75
  ,ymax          = 0.75
  ,ibc_y         = 1
  ,obc_y         = 1
  ,nzzones       = 1
```

```

,zmin          = 0.0
,zmax          = 1.0
,ibc_z         = 4
,obc_z         = 4
/
&PARALLEL
  Ngrid_x      = 2
  Ngrid_y      = 8
  Ngrid_z      = 1
&PROB
  gamma        = 1.4
,amp           = 0.01
,b0            = 0.25
,iprob         = 1
/

```

This is set up for a 2D problem because `nzzones` is set equal to 1. The y direction is the one that is in the direction of the gravitational acceleration. In the `PROB` block, the `iprob` parameter is set equal to 1. This variable determines whether to initialize the problem with random perturbations (`iprob=0`) or a fixed sine wave as is selected here. Finally, note that while the `Ngrid_x`, `Ngrid_y`, `Ngrid_z` variables are not necessarily set equal to 1, these variables will be ignored since the serial version of the code is compiled.

The Rayleigh-Taylor problem does well with a lot of resolution and is a good example to run in parallel (the above example has `nxzones = 300`, `nyzones = 900`, which is a large resolution). To do so, rebuild the code with the `--type=parallel` option in `buildathena`. This will produce the executable, `pathena3d`, in the `bin/` directory. The parallel block is set up to break the problem into 16 subdomains, 2 in the x direction and 8 in the y . The code could then be run with a command like

```
mpirun -np 16 ./pathena3d
```

The way to run an mpi job varies from system to system. The command `mpiexec` may be preferred on your system.

The test problems currently included with the *Athena3D* release are:

- 1D Sod hydro shock tube
- 1D Brio-Wu MHD shock tube
- 1D interacting blast waves

- 1D, 2D, or 3D linear waves in hydro or MHD
- 1D, 2D, or 3D circularly polarized, nonlinear Alfvén waves in MHD
- 2D double mach reflection
- 2D or 3D blast wave in hydro or MHD
- 2D Kelvin-Helmholtz instability in hydro or MHD
- 2D or 3D Rayleigh-Taylor instability in hydro or MHD
- 2D implosion problem in hydro
- 2D bow shock problem in hydro
- 2D MHD Orszag-Tang vortex
- 2D current sheet oscillation
- 2D or 3D magnetic field loop advection

6 Known Limitations

The *Athena3D* code can do quite a lot, but there is also a lot that it can't do. These limitations fall into several categories: (1) Things the C version can do that this version does not do, (2) Things that neither version does that would be nice to have and would be straightforward to implement if the authors' sole job was to develop this code, (3) Things that this code can't do and which would require considerable development effort to get it to do.

1. Things that this code doesn't do that the C version does (although they would be straightforward to implement in Fortran and may be some day):

- Other flux calculation routines (only Roe is implemented).
- Forces implemented in terms of a conserved cell-centered potential (Forces implemented as x , y , or z sources).
- Various alternative data output options, e.g. fits, vtk, pre-defined images (only binary, history and tabular data outputs defined).
- Simple way to add user defined output expressions (user must tinker with `data_out` module).
- H-correction to fix carbuncle problem.
- Self-gravity using a Fast Fourier Transform method.
- An arbitrary number of passive scalars that can be advected with the flow.
- Compute timestep from cell-centered values (timestep set from maximum wavespeed returned from eigen routines in previous integration step).
- Other integration algorithms (only "6-way" CTU scheme is implemented).
- Command line specification of input file (command line specifications not part of current Fortran standard).

2. Things neither code does that would be straightforward to implement:

- HDF5 format option for data dumps, particularly for parallel.
- Additional fluxes, interpolations, monotonic slopes, or integration schemes.

- Adding simple physics through operator splitting (e.g. cooling).
3. Things requiring more development effort or extensive rewriting:
- Graded, or non-constant mesh spacing.
 - Noncartesian grid geometry.
 - Static or adaptive mesh refinement.
 - Any equation of state except for constant γ adiabatic equation of state or isothermal.
 - Any new conserved quantity other than those already specified.
 - Radiative transfer, resistive and viscous terms.
 - Any additional physics handled through modifying the eigen solution.

Appendix: Manual Page for IDL readbin3d command which can be used with the binary dump files from the Fortran *Athena3D* code.

```
READBIN3D,filename, x, y, z, u, w, [GAMMA=variable], [ISOC=variable],  
[OFFSET=offset], [ENDIAN=string]
```

Arguments

filename

A scalar string containing the name of the binary file to be read by READBIN3D.

x

A real array, $x(nx)$, of grid cell locations, x , for nx zones returned by READBIN3D.

y

A real array, $y(ny)$, of grid cell locations, y , for ny zones returned by READBIN3D.

z

A real array, $z(nz)$, of grid cell locations, z , for nz zones returned by READBIN3D.

u

A real array, $u(nx, ny, nz, nvar)$, of conserved variables returned by READBIN3D. The $nvar$ dimension depends on the physics of the problem. For isothermal hydrodynamics, $nvar = 4$. For adiabatic hydro, $nvar = 5$. For MHD, $nvar = 7$ (isothermal) or $nvar = 8$ (adiabatic). $U = (\rho, \rho v_x, \rho v_y, \rho v_z, E, B_x, B_y, B_z)$.

w

A real array, $w(nx, ny, nz, nvar)$, of primitive variables returned by READBIN3D. The $nvar$ dimension depends on the physics of the problem. For isothermal hydrodynamics, $nvar = 4$. For adiabatic hydro, $nvar = 5$. For MHD, $nvar = 7$ (isothermal) or $nvar = 8$ (adiabatic). $W = (\rho, v_x, v_y, v_z, P, B_x, B_y, B_z)$.

Keywords

GAMMA

When set, READBIN3D will return the value of the adiabatic gamma to the named *variable*.

ISOC

When set, READBIN3D will return the value of the isothermal sound speed to the named *variable*.

OFFSET

Set this keyword if the binary file to be read was not produced by a Fortran program or needs an offset other than the default of 4 bytes for the header and footer of a Fortran binary write.

ENDIAN

Set this keyword to one of three string values: “big”, “little” or “native”. This keyword specifies the byte ordering of the file to be read. If the computer running IDL uses byte ordering that is different from that of the file, this will swap the order of the bytes when the file is read. (Default: “native” = perform no byte swapping.)