The `athena2.0` *Programmer'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 being developed with support of the NSF Information Technology Research (ITR) program. This *Programmer's Guide* describes version 2.0 (hereafter referred to as `athena2.0`); the second publicly released version of the code. This guide is intended as a very basic introduction into the data structures and organization of `athena2.0`, it will be useful to any user interested in modifying or extending the code. The *User's Guide* should be consulted for basic directions on how to install, configure, compile, and run `athena2.0`.

The source code development of `athena2.0` is managed using CVS (Concurrent Versions System)[1]. The genealogy of the code is kept clearly visible by the branching-mechanism in CVS (see also Figure **??**). Code development proceeds in the MAIN line of CVS. Once the code was deemed stable in 1D, it was branched with tagname `ONE_DIM` and prepared for a 1.0 release. The 2D algorithm was developed in the main line, and once fully tested, it was branched with tagname `TWO_DIM` and prepared for a 2.0 release. Development of the 3D version continues in the main line. Since the full 3D version can be used in 2D or 1D simply by setting the appropriate dimension(s) to 1, no more development will occur in `ONE_DIM` and `TWO_DIM` branches, other than important bug fixes, with the intent to keep these lower dimension releases simple and educational.

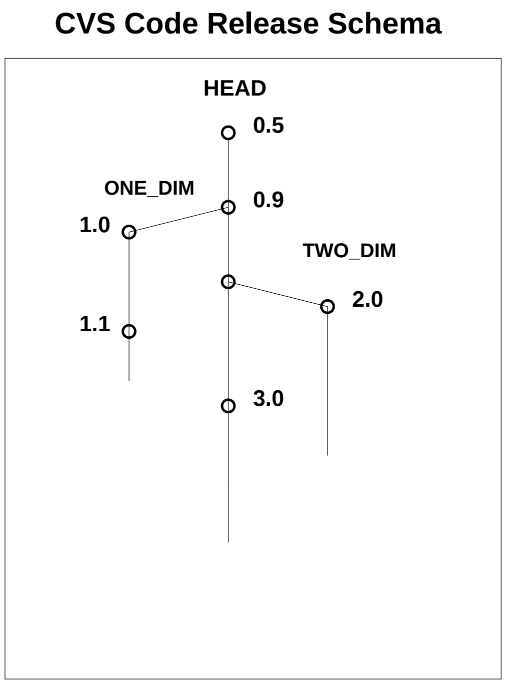

Figure 1: Structure of the CVS code development tree. Ongoing development occurs in the MAIN line of CVS. Released versions are the branches `ONE_DIM` and `TWO_DIM`.

2 The Computational Grid

The definition of the computational grid in `athena2.0` is fairly simple, since it uses a single uniform mesh. Figure **??** diagrams the transformation between the continuous spatial dimensions x_1 and

[1] `http://www.cvshome.org`, see also *Open Source Development with CVS* by Karl Frogel (Coriolis Press, 1999), and our own CVS project page `http://www.astro.umd.edu/ teuben/miriad/cvs.html`

x_2 and the discrete coordinates i, j. The origin of the computational domain has the coordinates $(x1\_0, x2\_0)$ in the continuous coordinates. These need not be the same as the origin $x_1 = 0$ and $x_2 = 0$. The origin of the discrete coordinate system (located at $(x1\_0, x2\_0)$) is $(ix1, ix2) = (0, 0)$. The computational domain can be further subdivided into patches; each patch is offset from $(ix1, ix2)$ by an integer number of grid cells $(idisp, jdisp)$. The grid coordinates within a given patch are referenced to the location of the point $(idisp, jdisp)$ (which is treated as the origin for *that* patch). The first active cell in the grid has the coordinates (is, js), which are both equal to the number of ghost cells used to specify the boundary conditions, that is $is = js = nghost$.

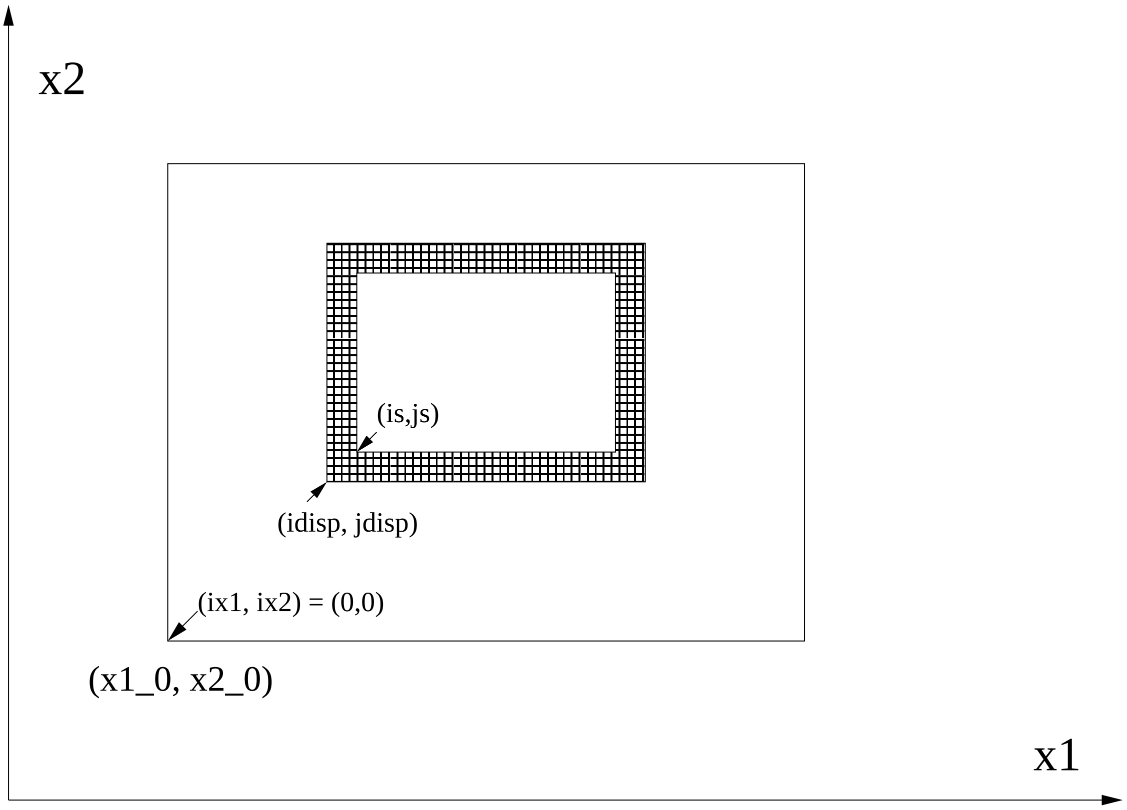

Figure 2: Relation between the continuous spatial dimensions x_1 and x_2 and the discrete coordinates i, j in `athena2.0`. See the text for details.

Mathematically, the spatial coordinate of the *center* of any grid point on the mesh is given by

$$x1_i = x1\_0 + (i + idisp - 0.5) * (\Delta x1)$$

$$x2_j = x2\_0 + (j + jdisp - 0.5) * (\Delta x2)$$

where $\Delta x1$ and $\Delta x2$ are the grid spacing on the patch. These formulae are implemented in the function `cc_pos.c`.

The use of patches and the offsets $(idisp, jdisp)$ are useful for nested grids, or tiling a single grid on parallel processors. However since neither capability has been implemented yet in `athena2.0`, the offsets are always set to zero.

Figure **??** shows the centering of variables within a cell on the computational grid. The vector **U** stores volume-averaged values of the conserved variables (density, momentum, total energy, and cell centered magnetic field). The fundamental representation of the magnetic field are the *area-averaged* components stored at cell faces.

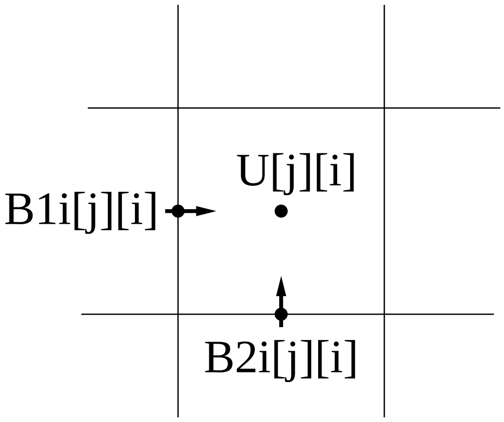

Figure 3: Centering of the magnetic field components at cell faces, and conserved variables at cell centers.

2.1 The Grid Structure

We organize the information about a grid patch, as well as the dependent variables on that patch, into a C structure called a Grid. The dependent variables *at cell center* are organized into another C structure we call Gas. Thus, a Grid contains the Gas structure, the face centered magnetic field, plus information about coordinates. These structures are defined in the header file `athena.h`, the Grid structure from that file is reproduced below.

```
typedef struct Grid_s{
  Gas **U;                      /* pointer to a 2D array of Gas'es */
#ifdef MHD
  Real **B1i,**B2i;             /* pointer to a 2D array of interface B's */
#endif /* MHD */
  Real x1_0;                    /* x1-position of coordinate ix = 0 */
  Real x2_0;                    /* x2-position of coordinate jx = 0 */
  Real dx1,dx2;                 /* cell size */
  Real dt,time;                 /* time step, absolute time */
  int nstep;                    /* number of integration steps taken */
  int Nx1,Nx2;                  /* number of zones in x1,x2 direction */
  int is,ie;                    /* start/end cell index in x1 direction */
  int js,je;                    /* start/end cell index in x2 direction */
  int idisp;                    /* coordinate ix = index i + idisp */
  int jdisp;                    /* coordinate jx = index j + jdisp */
  char *outfilename;            /* basename for output files */
}Grid;
```

Note that both the Gas structure and the face-centered fields are 2D arrays. The advantage of making the cell centered variables a 2D array of type Gas, rather than making the Gas structure itself contain 2D arrays, is that we guarantee that different variables at the same grid cell will be stored contiguously in memory, which can improve cache performance.

2.2 The Gas Structure

The Gas structure stores cell centered values of each of the dependent variables, it is defined in `athena.h` and reproduced below.

```
typedef struct Gas_s{
  Real d;                       /* density */
  Real M1;                      /* Momenta in 1,2,3 */
  Real M2;
  Real M3;
#ifndef ISOTHERMAL
  Real E;                       /* Total energy density */
#endif /* ISOTHERMAL */
#ifdef MHD
  Real B1c;                     /* cell centered magnetic fields in 1,2,3 */
  Real B2c;
  Real B3c;
#endif /* MHD */
}Gas;
```

The vector **M** denotes the linear momentum. Note the Gas structure contains all 3 components of the cell-centered magnetic field, that is it contains 8 variables in all. The longitudunal component of the field is dropped in the one-dimensional system of equations; we define new structures for the dependent variable in one-dimension, each of which only contains 7 variables. For the conserved variables, the structure is Cons1D

```
typedef struct{
  Real d;                       /* density (cell averaged) */
  Real Mx;                      /* Momenta in X,Y,Z */
  Real My;
  Real Mz;
#ifndef ISOTHERMAL
  Real E;                       /* Total energy density */
#endif /* ISOTHERMAL */
#ifdef MHD
  Real By;                      /* grid centered magnetic fields in Y,Z */
  Real Bz;
#endif /* MHD */
}Cons1D;
```

while for the primitive variables the structure is Prim1D

```
typedef struct{
  Real d;                       /* density (cell averaged) */
  Real Vx;                      /* Velocity in X,Y,Z */
  Real Vy;
  Real Vz;
#ifndef ISOTHERMAL
  Real P;                       /* pressure */
#endif /* ISOTHERMAL */
#ifdef MHD
  Real By;                      /* grid centered magnetic fields in Y,Z */
  Real Bz;
#endif /* MHD */
}Prim1D;
```

Note that we have used the subscripts x,y,z to denote the components of vectors in the one-dimensional variables contained in Cons1D and Prim1D, whereas we used the subsripts 1,2,3 to denote the components of vectors in the Gas structure. The variables in the Gas structure are fixed with respect to the coordinates of the grid (e.g., M1 corresponds to the momentum in the 1-direction). However, since the one-dimensional variables may represent a slice in any direction, the x,y,z components are not fixed with respect to the grid (that is, Mx corresponds to M1 along an a slice in the 1-direction, but Mx corresponds to M2 along an a slice in the 2-direction).

The order of the variables in the Gas, Prim1d, and Cons1D structures is extremely important and cannot be changed for several reasons. Firstly, this order determines the order of the elements used in the eigensystem of the linearized equations (computed in the functions contained in the files `esystem_prim.c` and `esystem_roe.c`). Secondly, in several functions we set a pointer to the first element in the structure, and then address successive elements (variables) by incrementing the pointer, as in the following code fragment from the function `lr_states_prim2.c`

```
pWl = (Real *) &(Wl[i+1]);

qx = 0.5*MAX(ev[NWAVE-1],0.0)*dtodx;
for (n=0; n<NWAVE; n++) {
  pWl[n] = Wrv[n] - qx*dW[n];
}
```

Here, `Wl`, `Wrv`, and `dW` are all structures of type Prim1D, and `NWAVE` is the number of components of these structures (which depends on whether the problem is hydrodynamic or MHD, and adiabatic or isothermal). Using pointers in this way leads to more compact coding, is more efficient (since it allows vectorization of the loop), and by using structures to ensure the components of the vectors are stored contiguously, it is more cache efficient. However, *be warned!*. It also means the order of the variables is hardwired in the code, and cannot be changed, which has the potential to be the source of some nasty bugs.

3 Memory Allocation and Management

`athena2.0` uses dynamic memory allocation, meaning that all arrays must be created at run time using `malloc` or `calloc`. This makes the coding slightly more complex, but has the great advantage that problems of different dimensions (1D or 2D), and problems of different sizes can be run without the need for recompiling.

The indexing convention used in `athena2.0` is that all multidimensional arrays must have the $x1$ index as the fastest incrementing index. In this way, data in cell $i-1$ should be contiguous with data in cell i, whereas data in cell $j-1$ will be $Nx1 + 2 * nghost$ data elements away from the data in cell j. In order to achieve this in C, arrays must be referenced as `A[j][i]`. The file `real_array.c` contains functions for creating (allocating) and destroying (deallocating) 2D arrays.

4 Basic Code Structure

The basic organization of the `athena2.0` code is shown below (using psuedo-code to simply represent the actual algorithm implemented in `main.c`).

```
main{
  show_config
  par_open
  par_cmdline
  par_close

  if (ires) {              /* For restarts (not implemented) */
    restart_grid_block
    integrate_init
    set_bvals
  }
  else {                   /* For new jobs */
    set_eos_param
    init_grid_block
    integrate_init
    problem
    set_bvals_init
    set_bvals
    init_dt
  }

  init_output
  lr_states_init
  Userwork_before_loop

  while(t<tlim && n<nlim) {
    data_output
    integrate
    set_bvals
  }

  Userwork_after_loop
  data_output
  lr_states_destruct
  integrate_destruct
  data_output_destruct

}
```

The first block of four functions opens, parses, and closes the input file, as well as reading and responding to any command line options.

The second block (contained in the `if (ires) { ... }` statement) initializes the simulation, including allocation of memory needed for the Gas and Grid structures, as well as any work arrays needed by the integration routines. The function problem is actually a soft link to one of the problem generators in the directory `/src` which is set during the configure step (before the code is compiled).

The next block of three lines outputs the initial conditions, and does any final work before the main integration loop.

The main loop is contained in the `while` statement. The function `integrate` calls the appropriate integration routine either `integrate_1d` or `integrate_2d` at run time based on the dimensions of the grid specified in the input file. Since the code uses dynamic memory allocation, the dimensions of the grid can be changed at run time without recompiling the code.

Finally, the last 5 lines after the main loop deallocates memory, outputs the final data, and terminates execution.

The data_output routine calls a variety of other functions:

```
data_ouput
  dump_binary
  dump_hdf4
  dump_history
  dump_table
  output_fits
  output_pgm
  output_ppm
  output_tab
```

Output functions named dump_* generally write out all the data in the grid in the given format, whereas output functions named output_* generally write out selected variables only. The *User's Guide*, section 3 has more information about data output formats.

The integrator for one-dimensional problems has a very simple structure:

```
integrate_1d
  lr_states
    esystem_prim
  GET_FLUXES
    esystem_cons
```

The function lr_states is also a soft link set by configure to be one of lr_states_prim1, lr_states_prim2, or lr_states_prim3 depending on the order of reconstruction specified by the user (see the User's Guide, §2.3). The keyword GET_FLUXES is set by configure to be one of flux_hlle, flux_hllc, or flux_roe depending on the Riemann solver to be used (see the User's Guide, §2.3).

The integrator for two-dimensional problems is much more complex, because of the need to use an unsplit integration method (consisting of a predict and correct steps), and the need to add "source terms" to balance the transverse flux gradients in the predict step. Details of the algorithm in multidimensions are given in the Method Paper.

5 Final Words about Code Development

We have used mostly the GNU development tools (gmake, gcc, autoconf) to develop the code. There is no reason, once the configure script has been generated, for it to fail on other platforms (although it almost certainly *will* fail on some machines). To date we have tested the code using Linux and MacOSX.

Coding style is very personal and can be a source of many fruitless discussions at Café Nero and/or Small World Cofee, so we adopt the following policy: we recommend the default emacs style (for example, indenting by 2 K&R style). See doc/template.c for guidelines.