

The athena3.0 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 developed with support of the NSF Information Technology Research (ITR) program. This *Programmer's Guide* describes version 3.0 (hereafter referred to as **athena3.0**); the third publicly released version of the code. This guide is intended as a very basic introduction into the data structures and organization of **athena3.0**, it will be useful to any user interested in modifying or extending the code. The *User's Guide* should be consulted for directions on how to install, configure, compile, and run **athena3.0**.

2 The Computational Grid

Figure 1 diagrams the transformation between the continuous spatial dimensions x_1 , x_2 , and x_3 and the discrete coordinates i, j, k . The origin of the computational domain has the coordinates $(x1_0, x2_0, x3_0)$ in the continuous coordinates. These need not be the same as the origin $(x_1, x_2, x_3) = (0, 0, 0)$. The discrete coordinate system is indexed by the integers $(ix1, ix2, ix3)$. The origin of the discrete coordinates $(ix1, ix2, ix3) = (0, 0, 0)$ is located at $(x1_0, x2_0, x3_0)$. The computational domain can be further subdivided into patches; each patch is offset from the origin of the discrete coordinate system by an integer number of grid cells $(idisp, jdisp, kdisp)$. The grid coordinates within a given patch are referenced to the location of the point $(idisp, jdisp, kdisp)$ (which is treated as the origin for *that* patch). The first active cell in the grid has the coordinates (is, js, ks) , which are all equal to the number of ghost cells used to specify the boundary conditions, that is $is = js = ks = nghost$. The last active cell in each dimension is labeled by $i = ie$, $j = je$, and $k = ke$.

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

$$x1_i = x1_0 + (i + idisp - 0.5) * (\Delta x1)$$

$$x2_j = x2_0 + (j + jdisp - 0.5) * (\Delta x2)$$

$$x3_k = x3_0 + (k + kdisp - 0.5) * (\Delta x3)$$

where $\Delta x1$, $\Delta x2$, and $\Delta x3$ are the grid spacing on the patch. These formulae are implemented in the function `cc_pos.c`. The use of patches and the offsets $(idisp, jdisp, kdisp)$ are useful for nested grids, or tiling a single grid on parallel processors.

Figure 2 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. We label the *left* interface value with the same index as the cell-center.

3 Data Structures

The entire computational domain is divided into grid blocks. Each block represents a patch of the domain updated by a separate processor (for calculations on parallel systems), or a region in which the grid is refined (for nested or adaptive grids), or both. Data on the grid is organized in the following C structures (defined in the header file **athena.h**).

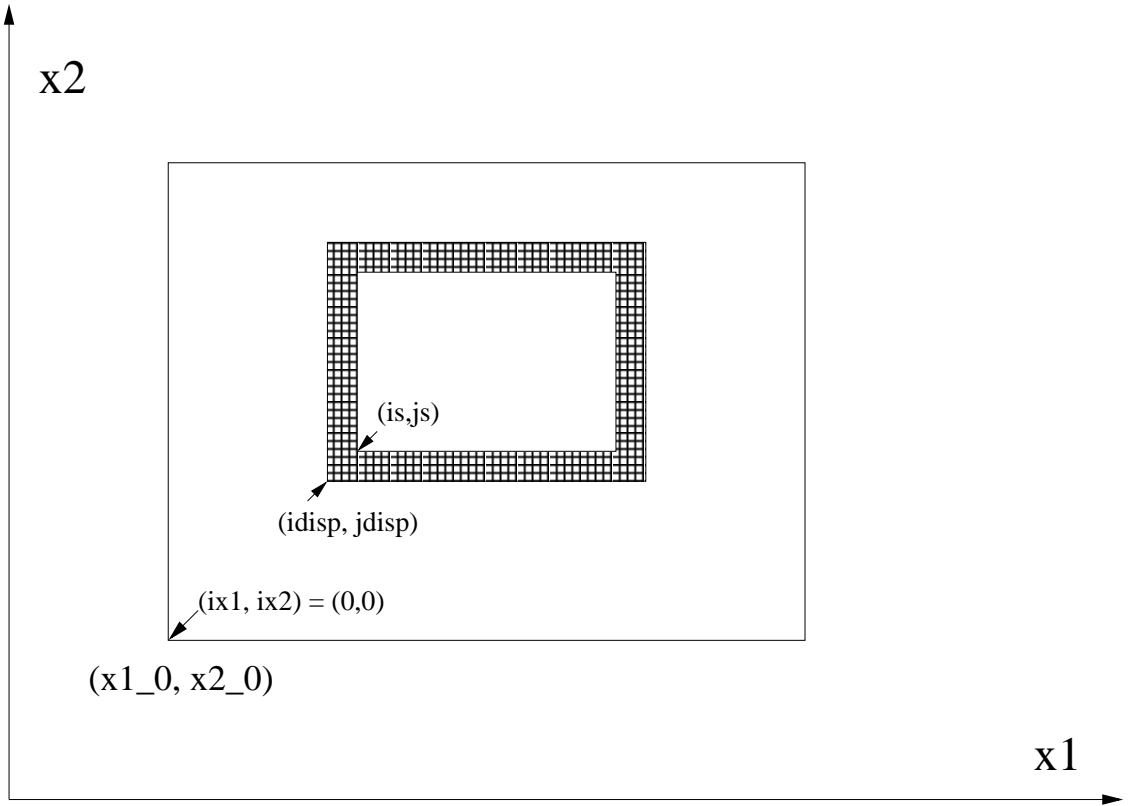


Figure 1: Relation between the continuous spatial dimensions x_1 and x_2 and the discrete coordinates i, j in a two-dimensional slice at a fixed value of x_3 (k) in **athena3.0**. The relations in the $x_2 - x_3$ and $x_1 - x_3$ planes are identical. See the text for details.

3.1 The Domain Structure

Information about the entire computational domain is stored in the Domain structure, reproduced below.

```
typedef struct Grid_Block_s{
    int ixs, jxs, kxs;           /* Minimum coordinate of cells in this block */
    int ixe, jxe, kxe;           /* Maximum coordinate of cells in this block */
    int my_id;                   /* process ID (rank in MPI) */
}Grid_Block;
```

```
typedef struct Domain_s{
    Grid_Block ***grid_block;    /* 3D array of grid blocks tiling this domain */
    int ixs, jxs, kxs;           /* Minimum coordinate of cells over entire domain */
    int ixe, jxe, kxe;           /* Maximum coordinate of cells over entire domain */
}Domain;
```

Note the Domain contains an array of Grid_Blocks, which in turn contain information about the coordinates of that patch in the continuous space (see §2.1).

3.2 The Grid Structure

We organize the information about each grid patch in the Domain, as well as the dependent variables on that patch, into a C structure called a Grid. The dependent variables *at cell center* are organized

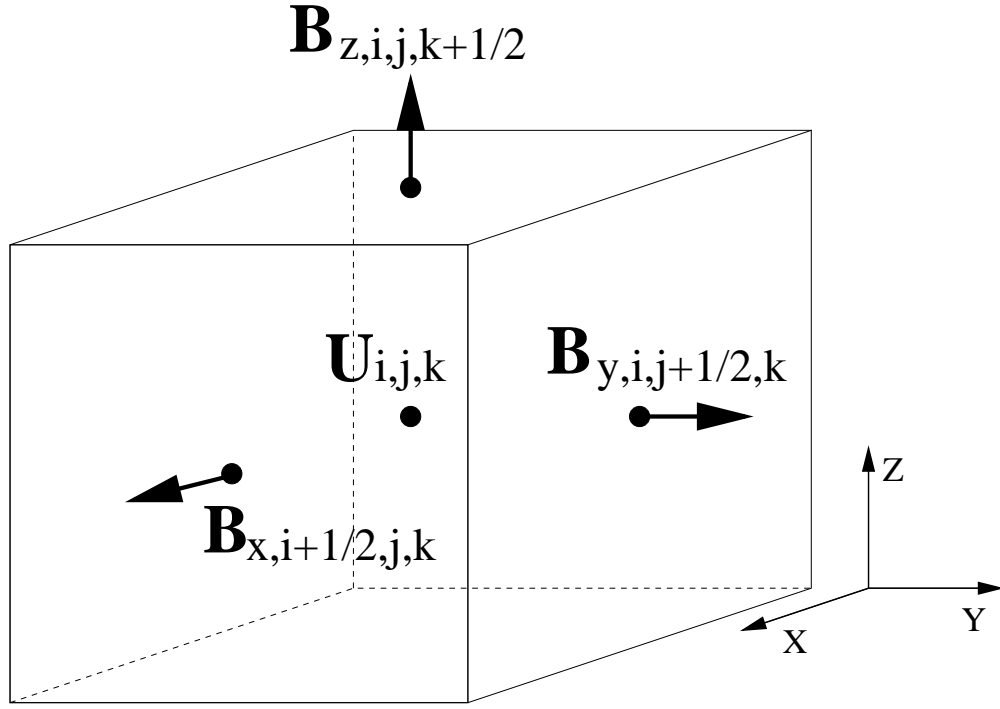


Figure 2: Centering of the magnetic field components at cell faces, and conserved variables at cell centers.

into another C structure we call Gas. Thus, a Grid contains the Gas structure, the face centered magnetic field, plus information about coordinates. The Grid structure is reproduced below.

```
typedef struct Grid_s{
    Gas ***U;                                /* pointer to a 3D array of Gas'es */
#ifdef MHD
    Real ***B1i,***B2i,***B3i;              /* pointer to a 3D 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 x3_0;                               /* x3-position of coordinate kx = 0 */
    Real dx1,dx2,dx3;                       /* cell size */
    Real dt,time;                           /* time step, absolute time */
    int nstep;                              /* number of integration steps taken */
    int Nx1,Nx2,Nx3;                        /* number of zones in x1, x2, x3 direction */
    int is,ie;                             /* start/end cell index in x1 direction */
    int js,je;                             /* start/end cell index in x2 direction */
    int ks,ke;                             /* start/end cell index in x3 direction */
    int idisp;                              /* coordinate ix = index i + idisp */
    int jdisp;                              /* coordinate jx = index j + jdisp */
    int kdisp;                              /* coordinate kx = index k + kdisp */
    char *outfilename;                     /* basename for output files */
    int my_id;                             /* process ID (or rank in MPI) updating this Grid */
    int nproc;                             /* total number of processes in the calculation */
    int rx1_id, lx1_id;                    /* ID of grids to R/L in x1-dir (default = -1) */
    int rx2_id, lx2_id;                    /* ID of grids to R/L in x2-dir (default = -1) */
    int rx3_id, lx3_id;                    /* ID of grids to R/L in x3-dir (default = -1) */
}Grid;
```

Note that both the Gas structure and the face-centered fields are 3D arrays. The advantage of making the cell centered variables a 3D array of type Gas, rather than making the Gas structure itself contain 3D arrays, is that we guarantee that different variables at the same grid cell will be stored contiguously in memory, which can improve cache performance.

3.3 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. Use 1,2,3 to label */
    Real M2;               /* directions in anticipation of */
    Real M3;               /* covariant coordinate in future */
#ifdef 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 longitudinal 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 Cons1D_s{
    Real d;                /* density */
    Real Mx;               /* Momenta in X,Y,Z. Use X,Y,Z now instead */
    Real My;               /* of 1,2,3 since this structure can contain */
    Real Mz;               /* a slice in any dimension: 1,2,or 3 */
#ifdef ISOTHERMAL
    Real E;                /* Total energy density */
#endif /* ISOTHERMAL */
#ifdef MHD
    Real By;               /* cell centered magnetic fields in Y,Z */
    Real Bz;
#endif /* MHD */
}Cons1D;
```

while for the primitive variables the structure is Prim1D

```
typedef struct Prim1D_s{
    Real d;                /* density */
    Real Vx;               /* Velocity in X,Y,Z */
    Real Vy;
```

```

    Real Vz;
#ifndef ISOTHERMAL
    Real P;                                /* pressure */
#endif /* ISOTHERMAL */
#ifdef MHD
    Real By;                                /* cell centered magnetic fields in Y,Z */
    Real Bz;
#endif /* MHD */
}Prim1D;

```

Note that we have used the subscripts x,y, and z to denote the components of vectors in the one-dimensional variables contained in Cons1D and Prim1D, whereas we used the subscripts 1,2, and 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, and 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.

3.4 The Output Structure

A final important data structure defined in `src/athena.h` is used for outputs. Each <output> block in an athena3.0 input file creates a new element in an array whose members are the following structure:

```

typedef struct Output_s{
    int n;                /* the N from the <outputN> block of this output */
    Real dt;              /* time interval between outputs */
    Real t;               /* next time to output */
    int num;              /* dump number (0=first) */
}

```

```

char *out;      /* variable (or user fun) to be output */
char *id;       /* filename is of the form <basename>[.idump][.id].<ext> */

/* variables which describe data min/max */
Real dmin;      /* user defined min for scaling data */
Real dmax;      /* user defined max for scaling data */
Real gmin;      /* computed global min (over all data output so far) */
Real gmax;      /* computed global max (over all data output so far) */
int sdmin;      /* 0 = auto scale, otherwise use dmin */
int sdmax;      /* 0 = auto scale, otherwise use dmax */

/* variables which describe coordinates of output data volume */
int ndim;       /* 3=cube 2=slice 1=vector 0=scalar */
int ix1l, ix1u; /* lower/upper x1 indices for data slice -1 = all data */
int ix2l, ix2u; /* lower/upper x2 indices for data slice -1 = all data */
int ix3l, ix3u; /* lower/upper x3 indices for data slice -1 = all data */
int Nx1;        /* number of grid points to be output in x1 */
int Nx2;        /* number of grid points to be output in x2 */
int Nx3;        /* number of grid points to be output in x3 */
Real x1_0, dx1; /* origin and grid spacing of output slice in x1 */
Real x2_0, dx2; /* origin and grid spacing of output slice in x2 */
Real x3_0, dx3; /* origin and grid spacing of output slice in x3 */

/* variables which describe output format */
char *out_fmt;  /* output format = {bin, tab, hdf, hst, pgm, ppm, ...} */
char *dat_fmt;  /* format string for tabular type output, e.g. "%10.5e" */
char *palette;  /* name of palette for RGB conversions */
float *rgb;     /* array of RGB[256*3] values derived from palette */
float *der;     /* helper array of derivatives for interpolation into RGB */

/* pointers to output functions; data expressions */
VGFunout_t fun; /* function pointer */
Gasfun_t expr;  /* expression for output */
} Output;

```

As can be seen, all of the information associated with the output (variables to be written, format, frequency of output, etc.) are stored in this structure. The functions in the file `output.c` should be consulted for more details.

4 Memory Allocation and Management

Athena3.0 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, 2D, or 3D), and problems of different sizes can be run without the need for recompiling.

The indexing convention used in **athena3.0** is that all multidimensional arrays must have the x_1 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 $Nx_1 + 2 * nghost$ data elements away from the data in cell j , and the data in cell $k - 1$ will be $(Nx_1 + 2 * nghost) * (Nx_2 + 2 * nghost)$ data elements away from the data in cell k . In order to achieve this in C, arrays must be referenced as `A[k][j][i]`. The file `real_array.c` contains functions for creating (allocating) and destroying (deallocating) 2D and 3D arrays.

Recall (from §2.1) that on each grid patch, the first active zone is labeled is and the last is labeled ie (similarly for js and je , and ks and ke). Thus, to stride across all active zones requires triply nested `for` loops:

```

for (k=ks; k<=ke; k++) {
    for (j=js; j<=je; j++) {
        for (i=is; i<=ie; i++) {
            ....
        }
    }
}

```

Note the inner loop should always be over the i -index.

5 Code Structure

The main integration loop is contained in `main.c`, which all initializes the domain and grid patches, and orchestrates I/O. Which integrator is actually used is determined by a function pointer set at runtime by `integrate_init()` in the file `integrate.c`.

The 1D, 2D, and 3D integrators are very different in structure. However, much effort has been put into keeping them as modular as possible. Thus, they all share the same reconstruction functions (e.g., `lr_states_prim2.c`, etc.) and flux functions (e.g., `flux_roe.c`, etc.). Eigensystems in the conserved and primitive variables, needed by some of the Riemann solvers and reconstruction algorithms respectively, are contained in the `esystem_roe.c` and `esystem_prim.c` files.

The input file is parsed by the functions in `par.c`, and I/O is coordinated by the functions in `output.c`.

Users who wish to make substantial modification or extension of the algorithms should start by understanding the 1D integrator (`integrate_1d.c`) and the functions it calls, before moving on to the 2D and 3D integrators.

6 Final Words about Code Development

We have used mostly the GNU development tools (gmake, gcc, autoconf) to develop the code. It should work on any platform that supports these tools (although experience shows that undoubtedly there will be some machines for which it *will* fail). To date we have tested the code on Linux, Solaris, and MacOSX.

Coding style is very personal and has been the source of many animated discussions at Café Nero and Small World Coffee. We have only agreed on the following: we recommend the default emacs style (for example, indenting by 2 as in K&R style).