APC/AST 523 Final Project : Zeus 2D

Evan Yerger and Valentin Skoutnev

May 15, 2017

1 Overview

This project attempted to lay the foundation for a 2D simulation of the MRI instibility in astrophysical acceretion disks. Our goal was to complete the hydrodynamic algorithms and tests for the MHD equations with no electric or magnetic fields. We closely followed Stone and Norman "ZEUS-2D: A radiation magnetohydrodynamics code for astrophysical flows in two space dimensions. I-The hydrodynamic algorithms and tests" (1992).

2 The Physical Setup and Equations

The MHD equations with out electric and magnetic terms are simply the equations of hydrodynamics. The domain of these equations in our program is a 2D cartesian mesh with periodic boundary conditions. Fluid equations are a reasonable approximation to real systems when the mean free path of particles compromising the fluid is much smaller than the length scale of the fluctions of macroscopic variables describing the fluid. Systems that fall under this category include most gases at STP and denser regions of an accreting accertion around its host body.

The hydrodynamic can be arrived at by taking moments of the Boltzmann equation for a fluid and closing the system with an equation state relating the pressure to density and energy. This should be equivilant to doing the same procedure for the kinetic equations for different plasma species

to obtain the MHD equations and then setting all electric and magnetic fields to zero. The set of hydrodynamic equations are:

$$\frac{D\rho}{Dt} + \rho\nabla\cdot v = 0 \tag{1}$$

$$\rho\frac{Dv}{Dt} = -\nabla p - \rho\nabla\phi \tag{2}$$

$$\rho\frac{D}{Dt}(\frac{e}{\rho}) = -p\nabla\cdot v \tag{3}$$

where $\frac{D}{Dt} = \frac{\partial}{\partial t} + v\cdot\nabla$ is the comoving derivative. ϕ is determined from the Poission equation for gravity:

$$\nabla^2\phi = 4\pi G\rho \tag{4}$$

3 Numerical Methods

For each time interval, updating the dependent variables is done using the operator splitting procedure. The splitting is done in two steps, the source and transport steps which updates equations in the following two groups:

Source:

$$\rho\frac{\partial v}{\partial t} = -\nabla p - \rho\nabla\phi - \nabla\cdot Q \tag{5}$$

$$\frac{\partial e}{\partial t} = -p\nabla\cdot pv - Q : \nabla v \tag{6}$$

Transport:

$$\frac{d}{dt}\int_v \rho dV = -\int_{dV} \rho(v - v_g)dS \tag{7}$$

$$\frac{d}{dt}\int_v \rho v dV = -\int_{dV} \rho v(v - v_g)dS \tag{8}$$

$$\frac{d}{dt}\int_v e dV = -\int_{dV} e(v - v_g)dS \tag{9}$$

v_g here is the grid velocity which we take to be a constant. Also a viscous term has been added in the source term that was not in the true equations.

This term is added for the purpose of dissipating unphysical effects on shock boundaries where the finite difference method breaks down. The transport step uses an intergral form because this allows the numerical method to conserve the total quantity of the advected variables to within round off error.

3.1 Source Step

The source step uses finite differences to compute new dependent variables. We will not write out all the numerical sets in their gory detail because they are written out in generality in Stone and Norman. We implemented their steps for cartesian coordinates. The source step is split into three parts. The first part updates the velocity due to the pressure gradients and graviational forces as shown above. The second step computes the viscosity terms. The third step computes the compressional heating term which updates the internal energy. As an example, an update for the x velocity in the first substep looks like:

$$\frac{{v_x}_{i,j}^{n+1} - {v_x}_{i,j}^{n}}{\Delta t} = -\frac{p_{i,j}^n - p_{i-1,j}^n}{\Delta x(\rho_{i,j} - \rho_{i-1,j})/2} - \frac{\phi_{i,j}^n - \phi_{i-1,j}^n}{\Delta x} \qquad (\mathbf{10})$$

where pressure is computed from the equation state $p = (\gamma - 1)e$ for an ideal gas.

3.2 Transport Step

The transport step uses finite differencing to advect the dependent variables (density, energy, and x,y,z momenta) as quantities that live inside the cell volumes. The rate of change of a quantity living in a cell volume is equal to the flux of the variable out of the cell surface. In two dimensions, the advection is done in two steps, one for each direction. The step first computes fluxes in the x direction in the entire domain using upwinded values of the dependent variables. Then these x fluxes are used to compute partially updated values of the dependent variables. Computation of the fluxes in the y direction in the entire domain then uses the upwinded value of the

partially updated dependent variables. The final dependent variables are then updated using these y fluxes.

We again don't write out the full numerical steps since they are in Stone and Norman. We adapt their technique for cartesian coordinates. As an example, updating the density looks like:

$$Fx_{i,j} = \rho^*_{i,j}(v_{xi,j} - v_g)dy \tag{11}$$

$$\frac{\rho_{partial}{}_{i,j} - \rho_{i,j}}{\Delta t} = Fx_{i,j} - Fx_{i+1,j} \tag{12}$$

$$Fy_{i,j} = \rho_{partial}{}^*_{i,j}(v_{y_{i,j}} - v_g)dy \tag{13}$$

$$\frac{\rho_{new}{}_{i,j} - \rho_{partial}{}_{i,j}}{\Delta t} = Fy_{i,j} - Fy_{i,j+1} \tag{14}$$

where $q^*_{i,j}$ means the interpolated upwind value of the q variable to the cell surface.

By computing fluxes in the entire domain before using them, one is able to conserve the dependent variable to numerical precision since the sum of all the fluxes is equal to the flux out of the boundary of the domain. The interpolation upwinded method that we currently implement is the 1st order Donor Cell method. We plan to soon incorporate the Van Leer 2nd order method.

3.3 Poisson Solver

Since we only use periodic boundary conditions we decided to use the fourier series method to solve Possions equation for ϕ. The Poisson solver first Fourier transforms the density ρ. Then the Fourier transformed $\hat{\phi}$ is computed from the solving the fourier transform of the Poisson equation:

$$\hat{\phi} = \frac{\Delta x^2 \hat{\rho}}{2(cos(2\pi i/N_x) + cos(2\pi j/N_y) - 2)}$$

Finally, ϕ is obtained by taking the inverse Fourier transform of $\hat{\phi}$.

4 Code Structure

The program on a high level is run from *zeus_main.cpp*. We wrote several helper classes to help keep track of time keeping (Timekeeper class), accessing arrays (Grid class), and program constants (Constants Class). Overall, *zeus_main.cpp* first initializes the Timekeeper, Constants, and Grid obejcts (which intiilaizes all the grids we need for the source and tranport steps). Then the main function runs the time iteration. Each time step calculates Δt using Timekeeper, then calls the poission solver, then calls the source step, and then calls the transport step, and lastly updates boundary conditions before starting over at the next time step. At the end of the entire copmutation it destructs all allocated memory.

We now describe in more detail our three classes followed by the poisson solver, source step, transport step, boundary conditions, and data I/O.

4.1 TimeKeeper class

The Timekeeper class (in *timestep.cpp*) has one primary function: compute the Δt for each step such that the CFL condition is satisfied in all orthogonal directions. It also keeps track of total time elapsed. As done in Zeus 2D, at the beginning of each time iteration Timekeeper takes the maximum over all grid cells of $\frac{1}{t_{max}} = max_{gridcells}(\frac{1}{(\delta t_1)^2} + \frac{1}{(\delta t_2)^2} + \frac{1}{(\delta t_3)^2} + \frac{1}{(\delta t_4)^2})^{\frac{1}{2}}$ and sets $\Delta t = C_o * t_{max}$ where C_o is the safety number (Courant number). δt_1 is the minimum sound speed crossing time across a cell, δt_2 is the minimum sound flow speed crossing time across a cell in the x direction, δt_3 is the minimum flow speed crossing time across a cell in the y direction, and δt_4 is the minimum diffusion time across a cell.

4.2 Constants class

The Constants class (in *constants.cpp*) simply holds the constants used throughout the program, most of which are initialized at the very beginning. The most important constants include the number of active grid cells in the x and y directions (N_x and N_y), the size of the spacings (dx, dy, dz), the number of ghosts cells (2), the value of γ (by default set to monotonic

gas value $\frac{5}{3}$), and the safety factor C_o (by default $C_o = .5$).

4.3 Grid class

The Grid class (in *grids.cpp*) is the most important one. Grid stores the current values of the pressure, density, energy, x,y,z velocities, and potential. Grid also has several helper functions such as initializing and detroying arrays and taking finite differences of arrays in a given direction. This approach to storing arrays was chosen because for all of our functions that operate on the dependent variables, we can simply pass a pointer to our Grid object to each function and then retrieve what we need from Grid in each function. Modifiying and debugging code with this approach is vastly easier and more flexible than if we were to hard code which arrays to pass into which functions.

4.4 Poisson Solver

We numerically implement the method described in the *Numerical Methods* section. To compute Fourier transforms we use the FFTW library. FFTW reference: Matteo Frigo and Steven G. Johnson, "The Design and Implementation of FFTW3," Proceedings of the IEEE 93 (2), 216231 (2005).

4.5 Source Step

The source step (in *source.cpp*) is fairly straight forward and is done as described in the *Numerical Methods* section. *source.cpp* has a main function *source_step*() and several helper functions. Each substep has its own function for the X and Y direction as needed that gets called in *source$_s$tep*(). This makes the main source step function easy to read and makes the overall step modular and thus easy to modify or debug. For reference, our commented *source_step*() function is shown below:

```
void source_step(Consts* c, Grid* g, double dt)
{
  //   calculate pressure from internal energy
```

```
  pressure(c, g, c->gamma-1);

  //   First substep
  subOneX(c, g, dt);
  subOneY(c, g, dt);

  //   Second substep
  subTwoQ(c, g, dt);
  subTwoVVE(c, g, dt);

  //   Third substep
  subThree(c, g, dt);

  //   Update old velocities in Grid g with newly computed ones
  replaceV(c, g);
}
```

4.6 Transport Step

The transport step (in *transport.cpp*) ended up having roughly 5 functions to carefully compute each transport substep for each of the 5 advected variables for a total of roughly 25 functions. The transport step calculates interpolated values of dependent variables and stores them in temporary arrays for later use in the calculation of the fluxes. This is done for every dependent variable. Then the fluxes are computed using separate functions for each variable. Then each variable is updated using these fluxes using separate functions. This is done separately in the X and Y directions. Extra care had to be taken to store temporary arrays such as interpolated density and partially updated density because these values were reused in later parts of the transport step.

The commented main function *transport_step*() that does all the high level computation organization is shown below:

4.7 Boundary Conditions

The boundary conditions (in *bcs.cpp*) are updated via 4 functions for the left, right, bottom, and top rows of ghost cells (periodicL(), periodicR()...). Each function updates the respective 2 rows or columns of ghosts cells of the all dependent variables that live in Grid g. For example, for periodicL(), the far right 2 columns of the active grid are copied into the 2 columns of ghost cells on the left.

4.8 Data I/O

5 References

Stone, J. M., and Norman, M. L. (1992). ZEUS-2D: A radiation magneto-hydrodynamics code for astrophysical flows in two space dimensions. I-The hydrodynamic algorithms and tests. The Astrophysical Journal Supplement Series, 80, 753-790.