Magnetohydrodynamics Modeling in Python

Simulating Magnetic Fluids

by

Ryan Farber

A Study

Presented to the Faculty

of

Wheaton College

in Partial Fulfillment of the Requirements

for

Graduation with Departmental Honors

in Physics

May 11, 2015

Magnetohydrodynamics Modeling in Python

Simulating Magnetic Fluids

Ryan Farber

Abstract

The purpose of this work is to present a gentle first introduction to numerical magnetohydrodynamics at the advanced undergraduate level. We first present the necessary background physics, building up the magnetohydrodynamics equations piece by piece. Then, we present numerical methods to discretize the equations. In addition to the traditional finite difference method, we present stable and nonstandard discretization methods employed in THE_ARGO, a minimalist magnetohydrodynamics code written in Python by this author. A backtracking scheme is used to discretize advection and diffusion and is guaranteed to be stable. Our nonstandard discretization is an atypical use of Fast Fourier Transforms to find the fluid pressure. Last, we present test problems validating the physical accuracy of THE_ARGO's unusual methods. We hope this work will provide students with some foundation before delving into the nasty traditional codes written in C and Fortran. Additionally, we hope the scientific community may benefit from our presentation of the nonstandard techniques we used to discretize the magnetohydrodynamics equations. Unlike common methods, our schemes guarantee stability, allowing larger time steps than the Courant condition permits.

Acknowledgements

THE_ARGO is named after the mythic ship on which Jason and the Argonauts sailed to retrieve the Golden Fleece. It is so named in honor of my ever-enduring advisor Jason Goodman who has put up with my badgerings for many months and without whose wealth of knowledge this thesis would be impossible.

I dedicate this work to first students and perplexed tenderfoots of numerical magnetohydrodynamics.

Table of Contents

List of Figures

List of Tables

Part I

The Physics of Ideal MHD

Chapter 1

Introduction

Fluid dynamics transcended the realm of engineering to pique the interest of applied mathematics, and eventually physics, largely through the work of Ludwig Prandtl in the early 20^{th} century (Eckert 2006). Despite the hard work of a number of great minds, the nonlinearity of the fluid equations led many problems to be without a solution. The era of the computer in the last half century has revolutionized the field, allowing approximate solutions to be found for a great number of the analytically insoluble fluid dynamics problems. With increasing computer power in the past quarter century, global climate models and cosmological simulation models including the effects of fluid dynamics became possible. Today, numerical fluid dynamics is a large and continually invigorated field: gentle introductions and tutorials exist in modern programming languages such as Python (Barba 2013).

In contrast, the field of numerical magnetohydrodynamics (MHD) is still in ascension; introductions to MHD are sparse and the best textbooks typically do not include numerical methods. Numerical MHD is a very recent offshoot of numerical fluid dynamics due to its perceived complexity and computational expensiveness. As a result, the field of MHD is usually gazed on in abhorrence despite its supreme importance in astrophysics where magnetic fields are ubiquitous and the ionized population of gases is significant for many processes. We believe a further reason for the fear of numerical MHD is the lack of resources at an introductory level; most numerical fluid dynamics resources don't touch on MHD and there seems to be no intermediate step into MHD before stepping into the professional literature or fully developed but poorly documented codes.

We developed a simple numerical MHD model in Python called THE_ARGO both for the purpose of applying it in personal research and to provide a sorely needed resource to introduce students to numerical MHD. The student of MHD may utilize this work in two principal ways.[1] The student may use this text as a guide to writing their own MHD code, or the student may approach this work as a textbook, refraining from any actual programming. We strongly encourage the reader to choose the former approach, for it is easy to believe that one understands what is going on, yet much more difficult to understand at the level in which one can implement a working program.

This work is organized into three parts, each of which is meant to ease the uninitiated student of numerical MHD into the field. In the first part, we give a brief introduction to fluid dynamics, starting with the definition of a fluid and an informal derivation of the fluid equations. We next derive the effects of magnetic fields on fluid motion, as well as the equation governing the time evolution of the magnetic field itself. The resulting equations are called the MHD equations.

In the second part, we explain how the terms in the MHD equations can be translated to their computer-digestible counterparts (the process of this translation is called discretization). We begin by describing the standard technique of explicit finite differencing which forms the backbone of THE_ARGO and is also a very simple and comprehensible first step at discretization. We then improve upon the explicit approach by implementing stable and nonstandard techniques; we discretize advection and diffusion terms using a backtracking scheme, which guarantees numerical stability, and we substantially speed up execution time by implementing a Fourier transform technique to find the pressure.

All numerical models require tests to validate their physical accuracy and validation is especially important when the techniques are novel; after all, physics is interested in the nature of the physical world, not every obtuse mathematical possibility! In the last part, we present a suite of test problems to validate THE_ARGO and close this work by discussing what has been accomplished.

[1]There may be more, but I agree with (Scannapieco and Harlow 1995) that there are chiefly two ways.

Chapter 2

What is a Fluid?

The term fluid, far from being synonymous with liquid, applies to a broad range of materials depending upon the time scale of the study, including even glaciers on geophysical time scales. A simple fluid is defined to have absolutely no rigidity, meaning that a shear stress no matter how small will cause the fluid to deform continuously, or flow. In other words, although a fluid may resist its deformation, such resistance is futile; the resisting force is always less in magnitude than the deforming force, causing the fluid to change in volume. In contrast, a solid is a rigid body; a moderate shear stress applied to a hardcover book (a force tangential to the cover) will cause only a small deformation, and (for a good binding) the cover will return to its original position shortly after the shear stress is removed. In reality, materials may exhibit fluid-like or solid-like properties under different conditions, but the previous definition is wholly suitable to our materials of interest, chiefly water.

2.1 The continuum hypothesis

Any type of fluid is composed of atoms that are sharply discontinuous in their density; the predominant concentration of mass is in a nucleus of femtometer scale while approximately five magnitudes farther out light electrons race around as distended appendages of the atom. Nonetheless, we need not concern ourselves with the motion of the material atom by atom and instead suppose that we may treat the macroscopic dynamics of fluids as though that fluid was strictly continuous in composition, spread uniformly through any given small volume (the following exposition largely follows Batchelor 1967, pp. 4-6). This continuum hypothesis is borne out in everyday experiences with air and water, as well as from the measurements of suitably designed measuring devices. Reasonable measuring devices are sensitive to cubical volumes of 1 cm^3, and yet at this scale there is still about $3{\times}10^9$ molecules of air and magnitudes more molecules when submerged in water. Since the volume contains such a great number of molecules, statistical fluctuations from the material's actual particular structure for quantities such as density, temperature, *et cetera*, are sufficiently small to be immeasurable and consequently have no effect on the observed average. This "invariance" of the measurements holds for further reductions in the sensitive volume of the measuring device, within limits, except in special circumstances such as in shock waves.

At first thought, the continuum hypothesis may appear antithetical to the astrophysicist, for the interstellar medium contains about one atom per cm^3 (Chaisson and McMillan 2004). Nonetheless, fluid dynamics is an important field applied to astrophysics; the dilemma is resolved by noting that a centimeter scale cubical volume is far below observational limits and that astrophysicists are typically interested in scales sufficiently large for the continuum hypothesis to be applicable.

Now that we have defended the application of the continuum hypothesis to our treatment of fluids, we proceed with an informal derivation of the fluid equations.

Chapter 3

The Fluid Equations

3.1 Continuity

Imagine water flowing in a river. Imagine further that a poor fisherman suspends a cardboard box from a tree, so that the box stays in a fixed location in the river with respect to the tree. The fisherman has oriented the open face of the box to face upstream, hoping to catch some fish. Unfortunately for the fisherman, the face of the box facing downstream is unable to withstand the current and rips open. In this state, whatever water flows into the confines of the box must flow out of the box. The divergence operator $\nabla\cdot$ represents the net flux through an infinitesimal volume, meaning that if all the sides of the cardboard box get ripped off to allow water to enter and leave through any face, and we continuously shrink the cube to the limit of zero volume, we have

$$\nabla \cdot \mathbf{u} = 0 \tag{3.1}$$

where $\mathbf{u} = u\hat{\mathbf{i}} + v\hat{\mathbf{j}} + w\hat{\mathbf{k}}$ is the velocity vector.[2] Eq. 3.1 is called the equation of continuity for an incompressible fluid, a name befitting its physical interpretation, mentioned above, that any and all fluid entering the box at velocity $\mathbf{u}$ must leave the box at velocity $\mathbf{u}$. Note that real fluids are compressible, meaning that Eq. 3.1, in its most general form, should also have a term for the change in density over time. However, the effects of compressibility for water under normal conditions is negligible, so we continue considering only incompressible flows.

[2]We use the standard physics notation where vectors are boldfaced and $\hat{\mathbf{i}}$ is the x-component, $\hat{\mathbf{j}}$ is the y-component, and $\hat{\mathbf{k}}$ is the z-component of a vector.

3.2 Types of Acceleration

Next, imagine a river with a steady current. By steady, we mean the current does not change with time for a given location of the river with respect to the river bank. The antithesis of a steady current may occur in a river downstream from a suddenly broken dam; there, the current dramatically increases in strength as the water surges through that location before again reaching a steady state. Note that a steady current can flow at different speeds at different locations; if the river should encounter a gentle incline followed by a gentle decline, then the current will surely decrease along the incline and increase along the decline in the downstream direction. We have thus encountered two different notions of how the current of a river, or more generally the velocity of a fluid, may vary.

In classical particle dynamics, the (one-dimensional) acceleration of an object is typically represented by $\frac{du}{dt}$ denoting the total derivative of velocity u with respect to time t. Since a particle exists only at a point, the velocity is a function of time only, $u(t)$, and hence the total derivative of the velocity is the correct concept of acceleration for a particle. However, we wish to consider the acceleration of the whole velocity field of a fluid, which is a function of position as well as time, $u(x,t)$. In this case, the particle concept of acceleration can be applied to the fluid as the change in velocity over time at a fixed location of the fluid, denoted by $\frac{\partial u}{\partial t}$.

In addition to the acceleration from $\frac{\partial u}{\partial t}$, we also want to include the acceleration from spatial variations of the velocity field, such as from a gentle incline or decline. Consider a raft floating on a decline of the river, so that it is moving with the same velocity u as the river. At a time Δt later, the boat will have moved through a distance $\Delta x = u\Delta t$ and its speed will have increased to u^{new} since the strength of the current increases downstream along the decline. Following (Faber 1995) we find

$$u^{\text{new}} = u + \Delta x \frac{\partial u}{\partial x} = u + u\Delta t \frac{\partial u}{\partial x} \tag{3.2}$$

By dividing through by Δt and taking the limit as Δt goes to infinity, we have an acceleration of $u\,\frac{\partial u}{\partial x}$. The name of this type of acceleration is called advection, of which there are several forms. Linear advection has the form $u\,\frac{\partial C}{\partial x}$ where C is a static tracer such as food coloring and which describes the advection of the food coloring by the velocity field. However, we will chiefly be interested in nonlinear advection $u\,\frac{\partial u}{\partial x}$ which is also called self advection (the variable advects itself), especially in three dimensions which is[3]

$$u\frac{\partial \mathbf{u}}{\partial x} + v\frac{\partial \mathbf{u}}{\partial y} + w\frac{\partial \mathbf{u}}{\partial z} = (\mathbf{u}\cdot\nabla)\mathbf{u} \tag{3.3}$$

In the following we use the term advection to mean nonlinear or self advection; we will include the adjectives distinguishing linear from nonlinear advection only when both types are present.

Adding the contributions from $\frac{\partial u}{\partial t}$ and advection written in vector notation, we obtain the total acceleration of the velocity field

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u}\cdot\nabla)\mathbf{u} \equiv \frac{D\mathbf{u}}{Dt} \tag{3.4}$$

which is often represented by $\frac{D\mathbf{u}}{Dt}$, called the material derivative. Note that for a steady current, $\frac{\partial \mathbf{u}}{\partial t} = 0$ and the only acceleration is due to advection.

3.3 Pressure-Driven Acceleration

We next wish to determine contributions to $\frac{\partial u}{\partial t}$. Imagine again a cardboard box held in a fixed location underwater. We refer to the fluid contained in an infinitesimal cardboard box as a fluid element. Consider a nonuniform pressure applied to the box faces normal to the x-direction (say, upstream and downstream) as in Fig. 3.1. Recall that pressure $p = F/A$, for an applied force F per unit area A. If we consider the net force due to the two pressures acting on our cardboard

[3]Note that $(\mathbf{u}\cdot\nabla) \neq (\nabla\cdot\mathbf{u})$. The latter, the divergence of $\mathbf{u}$, results in a scalar whereas the former is a differential operator.

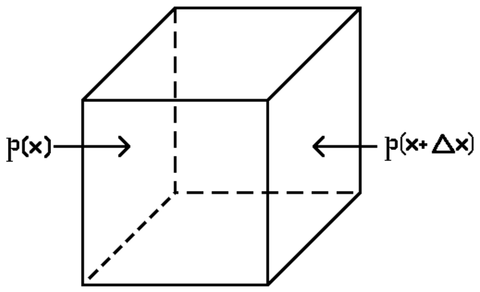


Figure 3.1: Downstream pressure $p(x)$ applied to the left face of the box, and upstream pressure $p(x + \Delta x)$ applied to the right face of the box. The difference in the two pressures causes a net acceleration of the cardboard box.

box, we have

$$p_{\text{net}} = -p(x + \Delta x) + p(x) = \frac{F_{\text{net}}}{A} \tag{3.5}$$

Newton's Second Law gives an expression for the force in terms of the acceleration, a, $F = ma = \frac{\rho a}{\Delta x \Delta y \Delta z}$ where m is the mass and ρ is the density of the fluid element. We use Newton's Second Law and substitute $A = \Delta y \Delta z$ to obtain

$$-p(x + \Delta x) + p(x) = \rho a \Delta x \tag{3.6}$$

from which we can find the net acceleration due to the change in pressure

$$a_{\text{net}} = -\frac{1}{\rho}\frac{p(x + \Delta x) - p(x)}{\Delta x} \tag{3.7}$$

Noting that Eq. 3.7 contains the definition of a derivative (with the limit of Δx approaching zero), we find

$$a_{\text{net}} = -\frac{1}{\rho}\frac{\partial p}{\partial x} \tag{3.8}$$

which generalizes in higher dimensions to the gradient of p:

$$\mathbf{a}_{\text{net}} = -\frac{\nabla p}{\rho} \tag{3.9}$$

Note that in general, ρ need not be uniform. However, we shall only concern ourselves with problems for which ρ has a constant value to minimize complications.

3.4 Viscous Diffusion of Momentum

A further contribution to $\frac{\partial u}{\partial t}$ occurs when fluid elements are in relative motion to one another, as in Fig. 3.2. The relative motion of the fluid elements above and below the middle box produce shearing stresses, and it is empirically observed that the shear stress is linearly proportional to the velocity gradient (Batchelor 1967). That is,

$$\begin{aligned} \sigma(y + \Delta y/2) &= \mu\tfrac{u(y)-u(y+\Delta y)}{\Delta y} \\ \sigma(y - \Delta y/2) &= \mu\tfrac{u(y-\Delta y)-u(y)}{\Delta y} \end{aligned} \tag{3.10}$$

where μ is a positive constant called the dynamic viscosity. Recalling that stress is a type of pressure, we know that the net stress σ_{net} is equal to the net force F per unit area A. Substituting $F = ma$ and $A = \Delta x \Delta z$ we have the new expression for the net stress

$$\frac{ma_{\text{net}}}{\Delta x \Delta z} = \sigma_{\text{net}} \tag{3.11}$$

We next multiply the left hand side (L.H.S.) by one in the form of $\frac{\Delta y}{\Delta y}$ and substitute the resulting $\frac{m}{\Delta x \Delta y \Delta z}$ with ρ to obtain

$$\rho a_{\text{net}} \Delta y = \sigma_{\text{net}} \tag{3.12}$$

Now, we substitute the change in σ for σ_{net}:

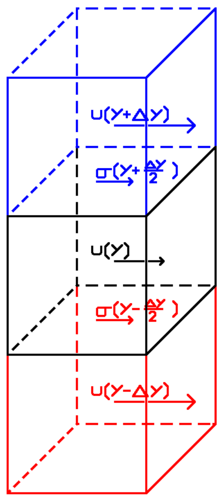


Figure 3.2: Shear stress applied to the middle cube's top face $\sigma(y+\Delta y/2)$ due to the relative velocity $u(y)-u(y+\Delta y)$ and shear stress applied to the middle cube's bottom face $\sigma(y-\Delta y/2)$ due to the relative velocity $u(y-\Delta y)-u(y)$.

$$\rho a_{\text{net}}\Delta y = \frac{-\sigma(y+\Delta y/2)+\sigma(y-\Delta y/2)}{\Delta y} \tag{3.13}$$

Next, we use Eq. 3.10, and divide both sides of the equation by $\rho\Delta y$ to obtain an expression for the net acceleration

$$a_{\text{net}} = \eta_\nu \frac{u(y+\Delta y)-2u(y)+u(y-\Delta y)}{\Delta y^2} \tag{3.14}$$

where $\eta_\nu = \mu/\rho$ is called the kinematic viscosity or also the coefficient of viscous diffusion. Noting the form of a second derivative in the limit that Δy approaches zero, we rewrite Eq. 3.14 as

$$a_{\text{net}} = \eta_\nu \frac{\partial^2 u}{\partial y^2} \tag{3.15}$$

The generalization in higher dimensions is

$$\mathbf{a}_{\text{net}} = \eta_\nu \nabla^2 \mathbf{u} \tag{3.16}$$

where ∇^2 is known as the Laplacian, which is shorthand for the divergence of the gradient. As with the fluid density ρ, we shall only consider problems with fluids of constant viscous diffusion η_ν for the remainder of this work.

3.5 The Fluid Equations

To recap, we found that the two types of acceleration a fluid can experience are due to forces bestowing a net acceleration over time $\frac{\partial \mathbf{u}}{\partial t}$ and by advection $(\mathbf{u}\cdot\nabla)\mathbf{u}$. The sum of the two accelerations is called the material derivative $\frac{D\mathbf{u}}{Dt}$. Additionally, we found two contributions to $\frac{\partial \mathbf{u}}{\partial t}$ by applying Newton's Second Law to pressure gradients normal to the face of a box or stresses shear (also called tangential) to the face of a box. History has named the resulting equation

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u}\cdot\nabla)\mathbf{u} = -\frac{\nabla p}{\rho} + \eta_\nu \nabla^2 \mathbf{u} \tag{3.17}$$

the Navier-Stokes equation after its founders, and it is typically referred to as the equation of motion of the fluid, a name befitting its contents. Typically, an equation of state is included as an additional fluid equation in order to find the fluid pressure. However, we instead use the pressure term as a means of removing divergence from the velocity field (to ensure the continuity equation is satisfied).

We do, however, include one additional equation which, along with the Navier-Stokes equation, forms our collection of fluid equations. The time evolution of an inert tracer concentration C is described by

$$\frac{\partial C}{\partial t} = -(\mathbf{u}\cdot\nabla)C + \eta_c \nabla^2 C \tag{3.18}$$

which says that the concentration is not affected (hence the befitting term "inert") by any process other than advection by the velocity field and diffusion.[4]

More physics, such as gravity, could easily be included by application of Newton's Second Law. We could also lift the restrictions of constant viscosity and density, and of incompressibility and isothermality; however, we restrict our focus

[4]η_c is the diffusion coefficient of the tracer concentration, which need not be the same as η_ν.

to minimalist numerical MHD so as to not overwhelm the reader. And for MHD, the most important missing pieces are the magnetic effects to which we now turn.

Chapter 4

The MHD Equations

4.1 Time Evolution of the Magnetic Field

In the previous chapter, we focused predominantly on the equation governing the time evolution of the velocity vector. Though magnetic effects must be added to the equation of motion, let us first consider the equation governing the time evolution of the magnetic field itself. We restrict our attention to highly conducting fluids and thus may neglect the Hall currents in Maxwell's equations. Maxwell's equations, with the magnetic permeability[5] μ, the electric permittivity ϵ, and the speed of light c set equal to one[6], are

$$\nabla \cdot \mathbf{B} = 0 \tag{4.1}$$

$$\nabla \cdot \mathbf{E} = \frac{q}{V} \tag{4.2}$$

$$\nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t} \tag{4.3}$$

$$\nabla \times \mathbf{B} = \mathbf{j} \tag{4.4}$$

[5]Note the repetition of the variable μ; it was used last chapter to signify dynamic viscosity! The author thinks repeated variable names are an abhorrence to be avoided at all costs. In this case, however, no harm is done; the dynamic viscosity was forever replaced with the viscous diffusion η_ν. Additionally, since the magnetic permeability is set equal to one, we shall not speak of it henceforth. On the other hand, we cannot use ρ for the charge density; ρ is already the fluid density! Thus we use q/V instead.

[6]Lorentz-Heaviside units

where $\mathbf{B}$ is the magnetic field vector, $\mathbf{E}$ is the electric field vector, q/V is the electric charge density, and $\mathbf{j}$ is the conduction current density. Note that the conduction current $\mathbf{j}$ is the only significant contribution to $\nabla \times \mathbf{B}$ for our materials of interest; one may recall other terms from a course in electricity and magnetism, but they are negligible until the fluid velocity approaches the speed of light (clearly not possible for water!) or as the frequency approaches that of visible light waves (Ferraro and Plumpton 1961).

From introductory electrodynamics one may remember that a linear relationship between the conduction current density and the electric field is empirically observed and is known as Ohm's Law

$$\mathbf{j} = \sigma \mathbf{E}^{'} \tag{4.5}$$

where σ is the conductivity of the fluid. We have suggestively written the electric field as $\mathbf{E}^{'}$ since the electric field is really the sum

$$\mathbf{E}^{'} = \mathbf{E} + \mathbf{u} \times \mathbf{B} \tag{4.6}$$

where the first term is the static electric field and the second term is an induction term generated by the motion of charged fluid parcels (Faraday's Law). Therefore, Ohm's Law may be written as

$$\mathbf{j} = \sigma(\mathbf{E} + \mathbf{u} \times \mathbf{B}) \tag{4.7}$$

Substituting Ohm's Law into Eq. 4.4, we find

$$\nabla \times \mathbf{B} = \sigma(\mathbf{E} + \mathbf{u} \times \mathbf{B}) \tag{4.8}$$

Next, we take the curl of both sides

$$\nabla \times (\nabla \times \mathbf{B}) = \sigma(\nabla \times \mathbf{E}) + \sigma(\nabla \times (\mathbf{u} \times \mathbf{B})) \tag{4.9}$$

so that we may eliminate the electric field using the expression for $\nabla \times \mathbf{E}$ in Eq. 4.3

$$\nabla \times (\nabla \times \mathbf{B}) = -\sigma \frac{\partial \mathbf{B}}{\partial t} + \sigma(\nabla \times (\mathbf{u} \times \mathbf{B})) \tag{4.10}$$

An identity[7] from vector calculus is handy at this point

$$\nabla \times (\nabla \times \mathbf{B}) = \nabla(\nabla \cdot \mathbf{B}) - \nabla^2 \mathbf{B} \tag{4.11}$$

Since $\nabla \cdot \mathbf{B} = 0$ by one of Maxwell's equations (Eq. 4.1), we can rewrite Eq. 4.10 as

$$-\nabla^2 \mathbf{B} = -\sigma \frac{\partial \mathbf{B}}{\partial t} + \sigma(\nabla \times (\mathbf{u} \times \mathbf{B})) \tag{4.12}$$

which can be solved to find the time evolution of $\mathbf{B}$

$$\frac{\partial \mathbf{B}}{\partial t} = \nabla \times (\mathbf{u} \times \mathbf{B}) + \eta_\rho \nabla^2 \mathbf{B} \tag{4.13}$$

where $\eta_\rho = 1/\sigma$ is the resistive diffusion. To break it down, the first term on the R.H.S. describes how the magnetic field is transported by the fluid flow (advected by the velocity), and the second term on the R.H.S simply says that magnetic field lines diffuse over time. When the second term on the R.H.S. can be neglected, Alfven's theorem of magentic flux applies. Also known as the "frozen in flux" condition or theorem, Alfven's theorem says that "the magnetic flux through a closed circuit moving with the fluid is constant" (Ferraro and Plumpton 1961, p. 25) and is hence thought of as magnetic field lines frozen into the flow of the fluid.

Unlike the equation of motion, we do not have any spare term with which we may remove divergence from the magnetic field. Instead, we calculate the magnetic field from the vector potential $\mathbf{A}$ which is defined to satisfy

$$\nabla \times \mathbf{A} = \mathbf{B} \tag{4.14}$$

[7]The author has proved this identity by hand; it is also used in (Batchelor 1967, p. 86).

and the derivatives of the curl operator are discretized via first order central differencing.

The vector identity $\nabla \cdot (\nabla \mathbf{f}) = 0$ for any vector $\mathbf{f}$ means that calculating $\mathbf{B}$ from Eq. 4.14 guarantees $\mathbf{B}$ will be divergence free. In this case, instead of a time evolution equation for magnetic field, we need a time evolution equation for the vector potential. We proceed by substituting Eq. 4.14 into Eq. 4.13

$$\frac{\partial(\nabla \times \mathbf{A})}{\partial t} = \nabla \times (\mathbf{u} \times (\nabla \times \mathbf{A})) + \eta_\rho \nabla^2 (\nabla \times \mathbf{A}) \tag{4.15}$$

By the orthogonality of spacetime, we can shift the curl outside of the time derivative, and by a vector calculus identity[8] we can similarly shift the curl outside the Laplacian

$$\nabla \times \frac{\partial \mathbf{A}}{\partial t} = \nabla \times (\mathbf{u} \times (\nabla \times \mathbf{A})) + \eta_\rho \nabla \times (\nabla^2 \mathbf{A}) \tag{4.16}$$

Next, we use three vector calculus identities[9] to obtain

$$\nabla \times \frac{\partial \mathbf{A}}{\partial t} = -\nabla \times \Big((\mathbf{v} \cdot \nabla)\mathbf{A} + \eta_\rho \nabla^2 \mathbf{A} \Big) \tag{4.17}$$

Finally, we remove the curl from both sides[10] to arrive at the time evolution equation for the vector potential.

$$\frac{\partial \mathbf{A}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{A} = \eta_\rho \nabla^2 \mathbf{A} \tag{4.18}$$

Note that our time evolution equation for $\mathbf{A}$ is just advection plus diffusion! By using this formulation, we both ensure the magnetic field is divergence free and

[8]To prove that the Laplacian may be shifted outside of a curl, use the curl of the curl identity (Eq. 4.11) and the definition of the vector potential $\nabla \times \mathbf{A} = \mathbf{B}$.

[9]First, $\mathbf{u} \times (\nabla \times \mathbf{A}) = \nabla(\mathbf{u} \cdot \mathbf{A}) - (\mathbf{u} \cdot \nabla)\mathbf{A}$ which was proved by hand (note that two terms drop out since both $\mathbf{u}$ and $\mathbf{A}$ are divergence free). Second, the curl of the gradient of any scalar field is identically zero, which removes the first term of the first identity mentioned. Third, we use the distributive property of curl $\nabla \times \mathbf{f} + \nabla \times \mathbf{g} = \nabla \times (\mathbf{f} + \mathbf{g})$ which holds for any vectors $\mathbf{f}$ and $\mathbf{g}$.

[10]By taking the path integral of both sides, one can find that $\nabla \times \mathbf{f} = \nabla \times \mathbf{g} \Rightarrow \mathbf{f} = \mathbf{g} + \mathbf{h}$ for any vector fields $\mathbf{f}$ and $\mathbf{g}$ and $\mathbf{h}$ is any vector field with zero curl. Since we use the vector potential solely to obtain a divergence free magnetic field, and any value of $\mathbf{h}$ will produce the same magnetic field ($\mathbf{B} = \nabla \times \mathbf{A}$), we set $\mathbf{h} = 0$.

make our system of equations more symmetric and easier to implement computationally.

Now that we have a method for evolving a divergence free magnetic field through time, we turn to incorporating magnetic force terms into the equation of motion.

4.2 Magnetic Force Terms in the Equation of Motion

Recall that the Lorentz force of a charge q in motion is

$$\mathbf{F} = q\mathbf{E} + q\mathbf{u} \times \mathbf{B} \tag{4.19}$$

A good conductor rearranges its distribution of charge nearly instantly to net neutrality, meaning that for our purposes:

$$\mathbf{F} = q\mathbf{u} \times \mathbf{B} \tag{4.20}$$

We next use Newton's second law to solve for the acceleration

$$\mathbf{a} = \mathbf{j} \times \mathbf{B} \tag{4.21}$$

where $\mathbf{j}$ is the current density. Recalling the expression for $\mathbf{j}$ in Ampere's Law (Eq. 4.4), we write

$$\mathbf{a} = (\nabla \times \mathbf{B}) \times \mathbf{B} \tag{4.22}$$

We next perform mystical vector calculus manipulations[11] to obtain:

[11] Again, the author has verified by hand the identity $(\nabla \times \mathbf{B}) \times \mathbf{B} = -\nabla\left(\frac{B^2}{2}\right) + (\mathbf{B} \cdot \nabla)\mathbf{B}$

$$\mathbf{a} = -\nabla\left(\frac{B^2}{2}\right) + (\mathbf{B}\cdot\nabla)\mathbf{B} \tag{4.23}$$

The first term on the R.H.S. is the so-called magnetic pressure, since it behaves similarly to the fluid pressure. The second term on the R.H.S. is the magnetic curvature term, which describes how the tension of magnetic field lines imparts an acceleration parallel to the direction of the magnetic field.

This chapter was certainly the most involved, but we achieved our goal: the MHD equations. In the next chapter, we review all the progress that we have made and preview what we will do next in Part 2.

Chapter 5

Looking to the Future

In the last few chapters, we have built up the equation of motion for a magnetic fluid term by term

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} - \nabla p + \eta_\nu \nabla^2 \mathbf{u} - \nabla\Big(\frac{B^2}{2}\Big) + (\mathbf{B} \cdot \nabla)\mathbf{B} \tag{5.1}$$

The first term on the R.H.S. called advection represents the acceleration in terms of spatial variations of the velocity field and is the only nonzero acceleration term when the net acceleration due to forces $\frac{\partial \mathbf{u}}{\partial t}$ is zero. The second term, the pressure gradient, is the acceleration experienced by a fluid element due to imbalanced pressures normal to two opposite faces of the element. The third term, the viscous diffusion, is an acceleration produced by the "rubbing" of two fluid elements, causing a net shear stress, and it is truly a frictional term. These first three terms comprise the totality of the fluid effects we consider while the last two terms include the magnetic effects. The fourth term is the magnetic pressure term, so-called in analogy to the fluid pressure term, and the fifth term is the acceleration experienced by the fluid element due to the tension of magnetic field lines.

Additionally, we have

$$\frac{\partial C}{\partial t} = -(\mathbf{u} \cdot \nabla)C + \eta_c \nabla^2 C \tag{5.2}$$

as the equation governing the time evolution of an inert tracer concentration. Henceforth, we drop the superfluous term "inert" since the equation says sufficiently that the tracer concentration is affected by advection by the velocity field and diffusion only and by no additional processes.

Through much effort, we found last chapter the equation governing the time evolution of the vector potential

$$\frac{\partial \mathbf{A}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{A} + \eta_\rho \nabla^2 \mathbf{A} \tag{5.3}$$

which can be used to calculate the magnetic field

$$\nabla \times \mathbf{A} = \mathbf{B} \tag{5.4}$$

in a manner which is guaranteed to be divergence free, required by one of Maxwell's equations. Again, we encounter familiar terms from the equation of motion; Eq. 5.3 consists solely of the velocity field advecting the vector potential (the first term on the R.H.S.) and diffusion of the vector potential (the second term on the R.H.S.).

Though the equation of motion has a number of terms, the other three equations are fairly straightforward, and four equations should not be too overwhelming for a student's first study in MHD. Additionally, the number of competing physical processes is not very many; diffusion, advection and pressure gradients are all the terms in the four equations.

As discussed in the previous two chapters, we have limited the scope of what THE_ARGO can solve. THE_ARGO is limited to incompressible, isothermal, irrotational, non-gravitational problems. THE_ARGO can be extended to include those pieces of additional physics but the current implementation is ideally suited to not overwhelm students being introduced to ideal MHD for the first time.

These four equations comprise the MHD equations our numerical model, THE_ARGO, must implement and solve. The MHD equations are a set of coupled non-linear partial differential equations. Consequently, very few analytical solutions are known and those found are typically uninteresting problems due to the simplifying assumptions made. In other words, the field of MHD is ideally suited to numerical investigations. In the next several chapters, we discuss the numerical methods by which THE_ARGO approximates and solves the MHD equations.

Part II

Discretizing the MHD Equations

Chapter 6

The Hollywood Way of Life

Our simulations are like motion pictures; we replace physical reality, in which time flows continuously, with a series of snapshots. Each snapshot includes a complete description of the system at that time, but any phenomena that occurs in time intervals less than the frequency between snapshots is lost. However, this loss of information is inevitable for many applications of planetary science and astrophysics, in which processes take thousands of years whereas we want our simulations to fit in reasonable research time spans. By making the change in time between those snapshots relatively small, we can describe a system to sufficient accuracy in a reasonable amount of time.

We refer to the snapshots of a simulated system as time steps; the notation we use for a variable of interest f at the current time step is f^n where n is called the time step. For one time step in the future we use f^{n+1} and for one time step in the past we use f^{n-1}. The change in time for any two consecutive time steps is Δt. Similarly, we simulate space as discrete blocks, which we call cells. Each cell has a width Δx in the first dimension, a breadth Δy in a second dimension, and a depth Δz in a third dimension. We indicate a particular cell for variable of interest f at time step n by $f^n_{i,j}$ where the subscript i specifies the cell in the x-dimension and j specifies the cell in the y-dimension.

The process of replacing continuous variables with discrete ones is called discretization. In the following several chapters, we discuss principally one dimensional and two dimensional discretization, both for ease of presentation and because THE_ARGO is not yet fully operational in three dimensions. Note, however, that the extension from two to three dimensions is very similar to the transition from one to two dimensions.

Chapter 7

Finite Differencing

One major technique in discretizing partial differential equations is known as the finite difference method (the following exposition is significantly influenced by Farlow 1993). To start, we write the Taylor expansion for a function of a single variable $f(x)$

$$f(x+\Delta x) = f(x) + f^{'}(x)\Delta x + \frac{f^{''}(x)}{2!}\Delta x^2 + ... \tag{7.1}$$

We truncate Eq. 7.1 to first order and solve for $f^{'}(x)$

$$f^{'}(x) = \frac{f(x+\Delta x) - f(x)}{\Delta x} \tag{7.2}$$

Eq. 7.2 is called the forward differencing approximation and is also known as an upwind scheme since the step Δx is taken in the positive x-direction. Note that in the limit that Δx approaches zero Eq. 7.2 contains the formal definition of a derivative as taught in a first course of calculus.

The backward difference approximation, or downwind scheme, is derived by Taylor expanding $f(x)$ with the step Δx taken in the negative x-direction

$$f(x-\Delta x) = f(x) - f^{'}(x)\Delta x + \frac{f^{''}(x)}{2!}\Delta x^2 - ... \tag{7.3}$$

truncating Eq. 7.3 to first order and solving for $f^{'}(x)$

$$f^{'}(x) = \frac{f(x) - f(x-\Delta x)}{\Delta x} \tag{7.4}$$

If we instead truncate Eq. 7.1 and Eq. 7.3 at second order, we can obtain the first order central difference approximation, by subtracting truncated Eqs. 7.1

from 7.3 and solving for $f^{'}(x)$

$$f^{'}(x) = \frac{f(x+\Delta x) - f(x-\Delta x)}{2\Delta x} \tag{7.5}$$

If we instead add the second order truncations of Eqs. 7.1 and 7.3, we obtain the second order central difference approximation by solving the resulting equation for $f^{''}(x)$

$$f^{''}(x) = \frac{f(x+\Delta x) - 2f(x) + f(x-\Delta x)}{\Delta x^2} \tag{7.6}$$

Note that Eq. 7.5 has a truncation error $O(x^2)$ compared to the other first order difference approximations. However, we must use a forward difference approximation for time partial derivatives (time flows forwards, not backwards or both ways!) or else the solution encounters a numerical instability and diverges; in general, we must use either forward or backward approximations for directional processes (such as advection). We can take advantage of the central difference approximations' second order truncation error only for isotropic processes (such as diffusion); when a discretization does not match the underlying physics of a process, numerical instabilities and generally erroneous solutions are prone to ensue.

Believe it or not, the first order forward difference, first order backward difference, first order central difference, and second order central difference approximations (and their two dimensional counterparts), are all the finite differencing[12] equations (just four!) we need to discretize the fluid equations.

We next present some examples of how the finite differencing is implemented in THE_ARGO to discretize terms of the MHD equations.

[12]For brevity, we henceforth typically refer to the act of applying a forward difference approximation as forward differencing, with the same form for the other finite difference approximations.

7.1 Curl of the Vector Potential

Recall that our method for keeping the magnetic field divergence free is to calculate the magnetic field from the vector potential

$$\mathbf{B} = \nabla \times \mathbf{A} \tag{7.7}$$

where $\mathbf{B}(x, y, z)$ has components[13] of B_x in the x-dimension, B_y in the y-dimension, and B_z in the z-dimension and $\mathbf{A}(x, y, z)$ has components of A_x in the x-dimension, A_y in the y-dimension, and A_z in the z-dimension. The curl operator in Eq. 7.7 is really differences of first order derivatives in each dimension. That is,

$$B_x = \frac{\partial A_z}{\partial y} - \frac{\partial A_y}{\partial z} \tag{7.8}$$

for the x-component of the magnetic field, and we discretize all derivatives with first order central differencing

$$B^n_{x_{i,j,k}} = \frac{A^n_{z_{i,j+1,k}} - A^n_{z_{i,j-1,k}}}{2\Delta y} - \frac{A^n_{y_{i,j,k+1}} - A^n_{y_{i,j,k-1}}}{2\Delta z} \tag{7.9}$$

Simple enough. But how do we find the vector potential in the first place!? Several chapters ago, we found that the time evolution equation for the vector potential is

$$\frac{\partial \mathbf{A}}{\partial t} + (\mathbf{v} \cdot \nabla)\mathbf{A} = \eta_\rho \nabla^2 \mathbf{A} \tag{7.10}$$

which we must discretize. In the next example, we discretize advection and diffusion of the velocity field. Note, however, that the exact same method may be applied to discretize advection and diffusion for the vector potential, so we omit the consequently redundant discretization of $\frac{\partial \mathbf{A}}{\partial t}$ here. Since the statements become cumbersome, we restrict ourselves to one dimension at first and then generalize to

[13]We work in three dimensions in this section since the expressions are brief enough.

two dimensions.[14]

7.2 Advection and Diffusion

7.2.1 In One Dimension

The equation of motion for advection and diffusion is a very important equation in fluid mechanics, and is called Burgers' equation. In one dimension, it is

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \eta_\nu \frac{\partial^2 u}{\partial x^2} \tag{7.11}$$

We discretize Eq. 7.11 by replacing the time derivative with the forward difference approximation, the first order space derivative with the backward difference approximation, and the second order space derivative with the second order central difference approximation to obtain

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} + u_i^n \frac{u_i^n - u_{i-1}^n}{\Delta x} = \eta_\nu \left(\frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2}\right) \tag{7.12}$$

We solve for u^{n+1} to obtain our time-stepping scheme for the advection plus diffusion equation in one dimension

$$u_i^{n+1} = u_i^n - \frac{\Delta t}{\Delta x} u_i^n (u_i^n - u_{i-1}^n) + \frac{\eta_\nu \Delta t}{\Delta x^2}(u_{i+1}^n - 2u_i^n + u_{i-1}^n) \tag{7.13}$$

In Part 3, we test the implementation of Eq. 7.13 compared to the analytical solution for a specially chosen test problem. Let us next generalize to two dimensions.

7.2.2 In Two Dimensions

In vector notation, advection plus diffusion is:

[14]The further generalization from two to three dimensions should be clear enough and we therefore omit such a further step.

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} = \eta_\nu \nabla^2 \mathbf{u} \tag{7.14}$$

which hides the fact that advection plus diffusion in two dimensions is actually a system of PDEs

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} = \eta_\nu\Big(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\Big) \tag{7.15}$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} = \eta_\nu\Big(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\Big) \tag{7.16}$$

We discretize both equations by finite differencing: forward differencing the time derivatives

$$\frac{\partial u}{\partial t} \rightarrow \frac{u_{i,j}^{n+1} - u_{i,j}^{n}}{\Delta t} \tag{7.17}$$

$$\frac{\partial v}{\partial t} \rightarrow \frac{v_{i,j}^{n+1} - v_{i,j}^{n}}{\Delta t} \tag{7.18}$$

backward differencing first order spatial derivatives, such as

$$\frac{\partial u}{\partial x} \rightarrow \frac{u_{i,j}^{n} - u_{i-1,j}^{n}}{\Delta x} \tag{7.19}$$

which we shall henceforth call $\frac{u_{i,j}^{n} - u_{i-1,j}^{n}}{\Delta x} \equiv \alpha_{u,x}$ for the sake of brevity, and central differencing second order derivatives, such as

$$\frac{\partial^2 u}{\partial x^2} \rightarrow \frac{u_{i+1,j}^{n} - 2u_{i,j}^{n} + u_{i-1,j}^{n}}{\Delta x^2} \tag{7.20}$$

which we shall henceforth call $\frac{u_{i+1,j}^{n} - 2u_{i,j}^{n} + u_{i-1,j}^{n}}{\Delta x^2} \equiv D_{u,x}$ for the sake of brevity. All together, the result is

$$\frac{u_{i,j}^{n+1} - u_{i,j}^{n}}{\Delta t} + u_{i,j}^{n}\alpha_{u,x} + v_{i,j}^{n}\alpha_{u,y} = \eta_\nu(D_{u,x} + D_{u,y}) \tag{7.21}$$

$$\frac{v_{i,j}^{n+1} - v_{i,j}^{n}}{\Delta t} + u_{i,j}^{n}\alpha_{v,x} + v_{i,j}^{n}\alpha_{v,y} = \eta_\nu (D_{v,x} + D_{v,y}) \tag{7.22}$$

which we solve for $u_{i,j}^{n+1}$ in the first equation and $v_{i,j}^{n+1}$ to obtain our time stepping scheme

$$u_{i,j}^{n+1} = u_{i,j}^{n} - \Delta t u_{i,j}^{n}\alpha_{u,x} - \Delta t v_{i,j}^{n}\alpha_{u,y} + \eta_\nu(\Delta t D_{u,x} + \Delta t D_{u,y}) \tag{7.23}$$

$$v_{i,j}^{n+1} = v_{i,j}^{n} - \Delta t u_{i,j}^{n}\alpha_{v,x} - \Delta t v_{i,j}^{n}\alpha_{v,y} + \eta_\nu(\Delta t D_{v,x} + \Delta t D_{v,y}) \tag{7.24}$$

In Part 3, we test the implementation of Eqs. 7.23 and 7.24 compared to an alternate discretization.

7.3 Limitations of Finite Differencing

In this chapter, we chose to discretize time derivatives with forward differences, with all but one term giving the value at the old time index. Such schemes in which the new value depends explicitly on the previous values are called explicit schemes. Explicit finite difference schemes are fairly easy to understand and reasonably fast; they do, however, face two supreme limitations.

It was mentioned earlier in this chapter that numerical instabilities may result from discretizations inappropriate to the underlying physics. For advection, the appropriate discretization depends upon the direction of the motion of the fluid, which for a chaotic flow, is constantly varying. The ugly solution is to check the direction of the fluid motion and implement the corresponding discretization.

An additional numerical instability plagues all explicit time stepping schemes. The instability occurs when the CFL condition (Courant, Friedrichs, Lewy; often called the Courant condition) is violated

$$u_{\max}\frac{\Delta t}{\Delta x} < \frac{1}{2} \tag{7.25}$$

where u_{max} is the magnitude of the maximum velocity simulated. Note that the condition requires increases in spatial resolution be compensated with increases in time resolution, meaning that a higher resolution simulation will need to perform more time steps to simulate the same total time as a lower resolution simulation. Additionally, it can be difficult to know beforehand what the maximum velocity achieved in a simulation may be, requiring either a guess or a dynamically varying Δt.

For these reasons, schemes that do not explicitly march values forward in time, befittingly called implicit schemes, are often used. Though implicit schemes are typically slower per time step than explicit schemes, implicit schemes can take larger time steps, as they are unburdened by the CFL condition, and hence can be faster than explicit schemes. In the next few chapters we present both implicit finite differencing schemes in addition to other implicit schemes.

Chapter 8

Backtracking Methods for Advection and Diffusion

In this chapter, we discretize advection and diffusion with a method guaranteeing stability (Stam 2003). The guaranteed stability allows us to take larger time steps for a certain spatial resolution than the CFL condition would allow. The technique is called backtracking and we'll first explain backtracking applied to advection.

8.1 Advection

To discretize advection via backtracking, we assign values to the new time step of each grid cell by determining from where that cell's material would have been carried (advected) by the velocity field, see Fig. 8.1. A linear interpolation of the old time step's value is then used to assign a value of the cell at the new time step. The linear interpolation is a weighted average of the field f's nearest neighbors at its backtracked position

$$f_{i,j}^{n+1} = s_0(t_0 f_{i0,j0}^n + t_1 f_{i0,j1}^n) + s_1(t_0 f_{i1,j0}^n + t_1 f_{i1,j1}^n) \tag{8.1}$$

where t_0 is how far the fluid particle was from the center of its bottom cell neighbor, and similarly t_1, s_0, and s_1 are distances from the top, left, and right neighboring cell center, respectively. We use f to emphasize the generality of the method; THE_ARGO applies the backtracking method to every instance of advection that occurs in the MHD equations. We next explain how we may backtrack diffusion.

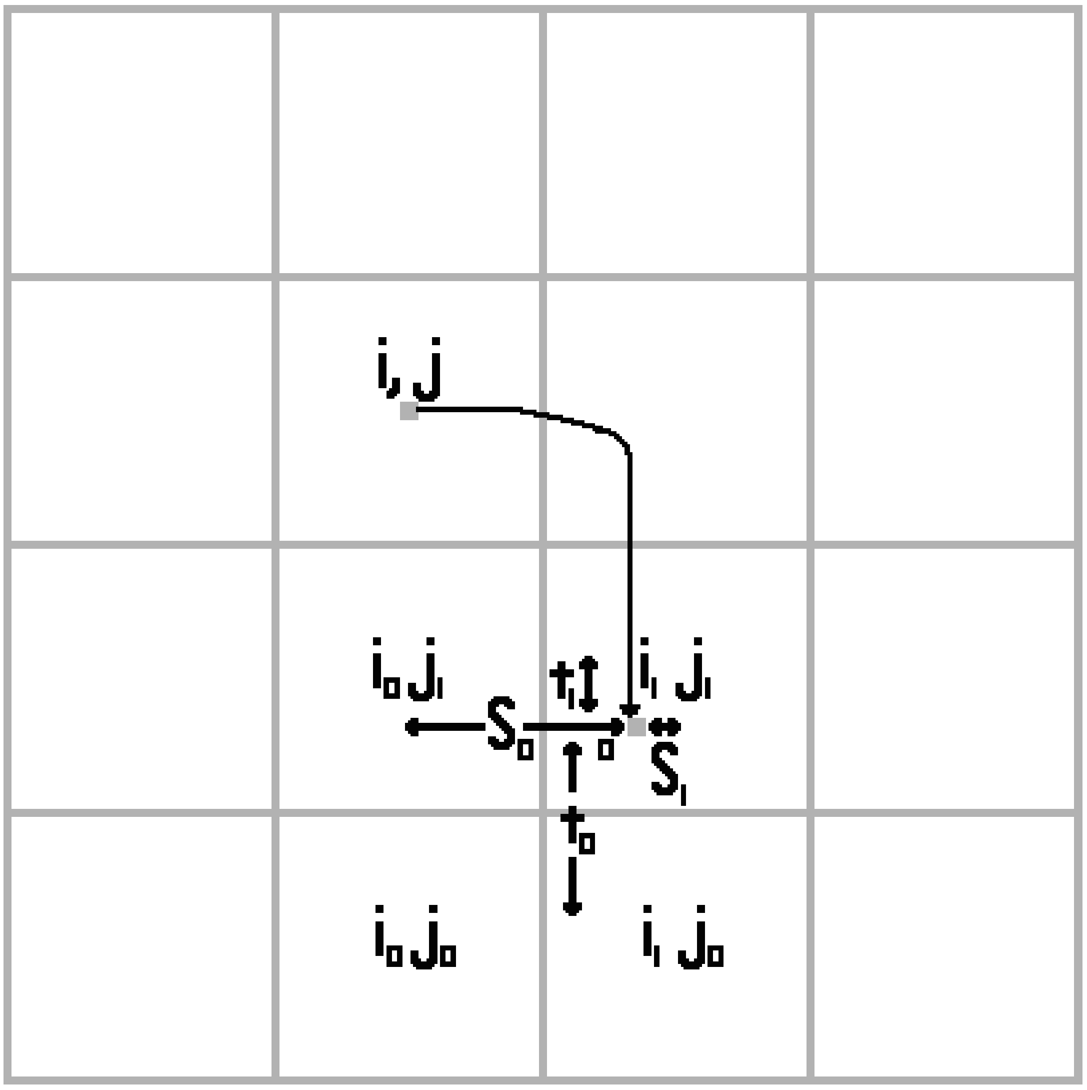


Figure 8.1: Schematic of the backtracking scheme for advection. We look at where the fluid particle, now at the center of a cell (upper left), was carried from (bottom right). We do so with a linear interpolation of how far the fluid particle was from the bottom wall of the cell t_0, and similarly t_1, s_0, and s_1 are distances from the top wall, left wall, and right wall of the cell, respectively.

8.2 Diffusion

Recall the diffusion equation for a field f

$$\frac{\partial f}{\partial t} = \eta_\nu \Big(\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \Big) \tag{8.2}$$

where we have again used f to denote the generality of the method; backtracking can be applied to every diffusion process in the MHD equations.

Backtracking applied to diffusion operates in a manner similar to advection, except that we consider only transfers between neighboring cells, see Fig. 8.2. To be more precise, we look for new values which satisfy

$$f^{\text{old}} = f^{\text{new}} - f^{\text{diffused}} \tag{8.3}$$

meaning that we require new values diffused backwards in time to return our old values.

We discretize Eq. 8.2 to satisfy Eq. 8.3 by backward differencing the time derivative and central differencing the space derivatives at the current time step

$$\frac{f^n_{i,j} - f^{n-1}_{i,j}}{\Delta t} = \eta_\nu \Big(\frac{f^n_{i+1,j} - 2f^n_{i,j} + f^n_{i-1,j}}{\Delta x^2} + \frac{f^n_{i,j+1} - 2f^n_{i,j} + f^n_{i,j-1}}{\Delta y^2} \Big) \tag{8.4}$$

Note that Eq. 8.4 is nearly identical to the explicit discretization presented last chapter; the chief difference is that Eq. 8.4 has the space derivative discretized at the current time step rather than the previous time step. Also note that for square cells, the result is simply an averaging of the nearest neighbors.

Next, we solve for $f^n_{i,j}$

$$f^n_{i,j} = \Big(f^{n-1}_{i,j} + \chi[\Delta y^2 (f^n_{i+1,j} + f^n_{i-1,j}) + \Delta x^2 (f^n_{i,j+1} + f^n_{i,j-1})] \Big) \Big/ \kappa \tag{8.5}$$

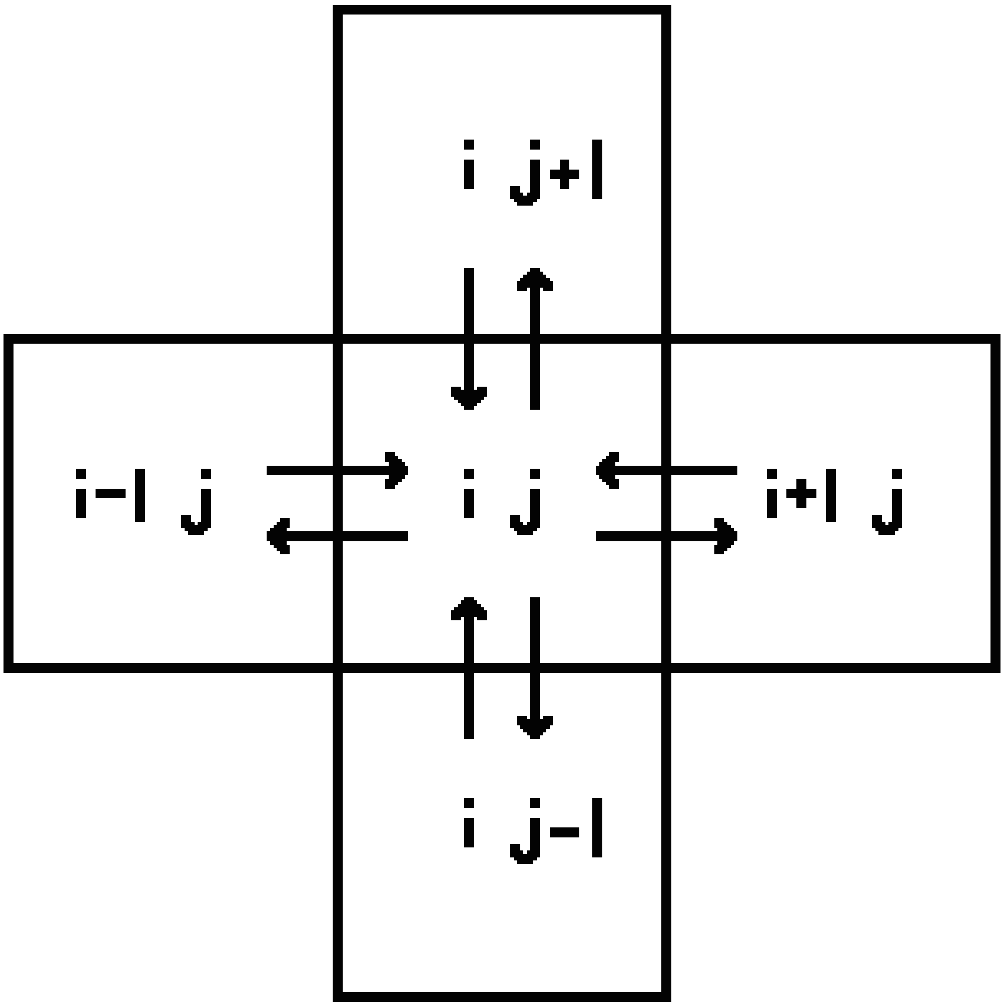


Figure 8.2: Schematic of diffusion as velocity exchanges between neighboring cells.

where $\chi = (\eta_\nu \Delta t)/(\Delta x^2 \Delta y^2)$ and $\kappa = (1 + 2\chi(\Delta x^2 + \Delta y^2))$ for brevity.

Eq. 8.5 is implemented in THE_ARGO by first assigning initial values to each cell, matching the required initial conditions. Then, the new value at each cell $f^n_{i,j}$ is computed by adding its current value at that cell $f^{n-1}_{i,j}$ to the weighted average of the current values of its neighboring cells.

It is not clear at present whether the stability gained by backtracking might not be traded for a loss of accuracy compared to explicit finite differencing. In Part 3, we will show that backtracking produces visually identical results to the tested and true finite differencing methods.

In the next chapter, we discuss a relaxation method used to find the pressure and how that pressure is used to remove divergence from the velocity field.

Chapter 9

Eliminating Velocity Divergence by Relaxing the Pressure

The relaxation method is similar to the backtracking method for diffusion described last chapter. First, the initial condition is applied to give each cell a starting value. Subsequently, each cell is assigned new values as a weighted average of the values of its neighboring cells. Whereas backtracking of diffusion stops at this point, the relaxation method continues, performing the weighted averaging re-assignments iteratively (over and over) until a steady state has sufficiently converged to the requisite precision; the method "relaxes" from its initial conditions to a steady state. For relaxation methods, the value of the steady state depends predominantly on boundary conditions and source terms, since memory of the initial condition is soon washed away by the averaging. Note that the backtracking method for diffusion is really a relaxation method; we just perform one iteration for diffusion since that is accurate enough for our purposes and is much faster.

9.1 Finding an Expression for the Pressure

Recall from Part 1 that we don't include the physical pressure (requiring an equation of state) in the fluid and MHD equations, but instead formulate ∇p to eliminate the divergence from the velocity field, as is required by the continuity equation for incompressible flow. Recall the fluid equation of motion

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} = -\frac{\nabla p}{\rho} + \eta_\nu \nabla^2 \mathbf{u} \tag{9.1}$$

The time derivative is discretized with a forward difference approximation but we retain vector notation since the same procedure is performed for both components of $\mathbf{u}$. We also rearrange terms to obtain

$$\mathbf{u}^{n+1} = \mathbf{u}^n - \Delta t(\mathbf{u}^n \cdot \nabla)\mathbf{u}^n - \Delta t\frac{\nabla p}{\rho} + \Delta t\eta_\nu \nabla^2 \mathbf{u}^n \tag{9.2}$$

where the spatial indices have been suppressed.

First, we discretize the advection and diffusion terms. Then, we apply the discretization to obtain $\mathbf{u}^{\text{new}}$, the updated velocity field

$$\mathbf{u}^{\text{new}} = \mathbf{u}^n - \Delta t(\mathbf{u}^n \cdot \nabla)\mathbf{u}^n + \Delta t\eta_\nu \nabla^2 \mathbf{u}^n \tag{9.3}$$

Due to unavoidable numerical errors (and possibly from some forcing terms and the boundary conditions), $\mathbf{u}^{\text{new}}$ is probably divergent! To eliminate this errant divergence, we rewrite Eq. 9.2 with the updated velocity field $\mathbf{u}^{\text{new}}$

$$\mathbf{u}^{n+1} = \mathbf{u}^{\text{new}} - \Delta t\frac{\nabla p}{\rho} \tag{9.4}$$

Next, we take the divergence of both sides and solve for ∇^2p

$$\nabla^2 p = -\frac{\rho}{\Delta t}(\nabla \cdot \mathbf{u}^{\text{new}}) \tag{9.5}$$

where we have used the fact that we are looking specifically for a pressure field to make $\nabla \cdot \mathbf{u}^{n+1} = 0$. Eq. 9.5 is called a Poisson equation, with the right hand side called the source term, b. Discretizing the divergence of $\mathbf{u}^{\text{new}}$ via second order central differencing, we arrive at the following expression for b

$$b_{i,j}^{\text{new}} \equiv -\frac{\rho}{\Delta t}\Big(\frac{u_{i+1,j}^{\text{new}} - u_{i-1,j}^{\text{new}}}{2\Delta x} + \frac{u_{i,j+1}^{\text{new}} - u_{i,j-1}^{\text{new}}}{2\Delta y}\Big) \tag{9.6}$$

where, as usual, the superscript indicates the time step and the subscripts indicate the particular cell.

In the next section we discretize $\nabla^2 p$.

9.2 Discretization of the Poisson Equation

Recall that the Laplacian operator is div of grad so that (for a Cartesian coordinate system) the Poisson equation is really just the sum of the second derivatives of p in each dimension

$$\nabla^2 p = \frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2} \tag{9.7}$$

We discretize Eq. 9.7 by second order central differencing

$$\frac{p_{i+1,j} - 2p_{i,j} + p_{i-1,j}}{\Delta x^2} + \frac{p_{i,j+1} - 2p_{i,j} + p_{i,j-1}}{\Delta y^2} = b_{i,j} \tag{9.8}$$

where b is the source term of form Eq. 9.6. Note that we have suppressed time indices, since all terms are at the same time step.

Solving for $p_{i,j}$ we find

$$p_{i,j} = \frac{(p_{i+1,j} + p_{i-1,j})\Delta y^2 + (p_{i,j+1} + p_{i,j-1})\Delta x^2 + b_{i,j}\Delta x^2 \Delta y^2}{2(\Delta x^2 + \Delta y^2)} \tag{9.9}$$

Note that Eq. 9.9 becomes the average of the four neighboring grid points of $p_{i,j}$ when $\Delta x = \Delta y$ and $b = 0$. Eq. 9.9 is solved by first applying the initial condition to give values for the pressure at each cell $p_{i,j}$. Then, the pressure at each cell $p_{i,j}$ is found by the weighted average of its neighboring cells. The weighted averaging is performed repeatedly, relaxing the initial condition of each cell until the values converge sufficiently. The value arrived at depends predominantly on the source term b.

Since the source term contains information on the divergence of the velocity field, now embedded in the pressure field, we can apply the gradient of the pressure to the updated velocity to ensure the velocity field at the next time step is divergence free. We do so by first order central differencing the gradient operator:

$$u^{n+1} = u^{\text{new}} - \frac{1}{\rho}\Big(\frac{p_{i+1,j} - p_{i-1,j}}{2\Delta x} + \frac{p_{i,j+1} - p_{i,j-1}}{2\Delta y}\Big) \tag{9.10}$$

We have now presented sufficient methods for the reader to implement their own MHD code. What about the magnetic pressure term in the equation of motion? It can be discretized by first order central differencing too. Additionally, recall that the magnetic field is calculated from the vector potential, ensuring it too is divergence free.

However, we have one additional method to present. The relaxation method is unfortunately quite slow; it took one minute per time step (!) to perform a coarse resolution 2D MHD simulation. In the next chapter, we speed things up with Fast Fourier Transforms.

Chapter 10

Fourier Transform Method

In this chapter we present the famously fast Fast Fourier Transform (FFT) method (and our presentation largely follows Gonsalves 2004).

Consider the one dimensional Poisson equation for a function f with source term g

$$\frac{\partial^2 f}{\partial x^2} = g \tag{10.1}$$

We rewrite f and g with their Fourier transforms

$$f(x) = \frac{1}{\sqrt{2\pi}} \int F(k_x) e^{ik_x x} dk_x \tag{10.2}$$

$$g(x) = \frac{1}{\sqrt{2\pi}} \int G(k_x) e^{ik_x x} dk_x \tag{10.3}$$

where F is the Fourier transform of f, G is the Fourier transform of g, and k_x is the wavenumber in the x-direction.

Next, we invoke a rule for transforming partial derivatives (Farlow 1993)

$$F\left(\frac{\partial^2 f}{\partial x^2}\right) = -k_x^2 F(f(x)) \tag{10.4}$$

where $F(f(x)) = F(k_x)$ (which may be verified using integration by parts).

Next, we take the Fourier transform of both sides of Eq. 10.1 and substitute Eq. 10.4 for $F\left(\frac{\partial^2 f}{\partial x^2}\right)$ to obtain

$$-k_x^2 F(k_x) = G(k_x) \tag{10.5}$$

where $G(k_x)$ is the Fourier transform of $g(x)$. We solve for $F(k_x)$ and take the inverse transform to obtain $f(x)$

$$f(x) = -\frac{1}{\sqrt{2\pi}} \int \frac{G(k_x)}{k_x^2} e^{ik_x x} dk_x \qquad (10.6)$$

We implement the two dimensional version of Eq. 10.6 in THE_ARGO by using the built-in functions of Python's scientific computing package numpy (The Scipy Community 2014). Specifically, we used fft2 to obtain $G(k_x, k_y)$, fftfreq to obtain k_x and k_y and ifft2 of the left hand side of the two-dimensional version of Eq. 10.6 to get $f(x, y)$, which in our case is the pressure p.

FFTs are not alien to fluid dynamics codes; indeed, most codes that incorporate FFTs evolve variables through time in frequency space, only transforming back to physical space in order to save the physical variable. In contrast, we take the atypical approach of operating predominantly in physical space, only transforming to frequency space to find the pressure, which we immediately transform back to physical space.

Whew! That was a lot of work. In the next chapter we summarize the methods presented in the last several chapters before moving on to the test problems.

Chapter 11

Discretization Methods Summary

After much work, it is helpful to see what we have accomplished and to remind ourselves of what remains to be done. We have two methods for discretizing every case of advection and diffusion in the MHD equations

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u}\cdot\nabla)\mathbf{u} - \frac{\nabla p}{\rho} + \eta_\nu \nabla^2 \mathbf{u} + (\mathbf{B}\cdot\nabla)\mathbf{B} - \nabla\Big(\frac{B^2}{2}\Big) \tag{11.1}$$

$$\frac{\partial C}{\partial t} = -(\mathbf{u}\cdot\nabla)C + \eta_c \nabla^2 C \tag{11.2}$$

$$\frac{\partial \mathbf{A}}{\partial t} = -(\mathbf{u}\cdot\nabla)\mathbf{A} + \eta_\rho \nabla^2 \mathbf{A} \tag{11.3}$$

$$\nabla \times \mathbf{A} = \mathbf{B} \tag{11.4}$$

Additionally, the pressure can be found by either a relaxation method or by Fourier transforms. The gradients of both fluid pressure and magnetic pressure, as well as the curl of **A** are discretized using second order central differencing (recall that we calculate **B** from $\nabla \times \mathbf{A}$ to ensure $\nabla \cdot \mathbf{B} = 0$).

Still, we must demonstrate that the numerical methods presented in this part yield physically accurate solutions. In the next part, we present a selection of problems to test the veracity of THE_ARGO.

Part III

Simulations Testing THE_ARGO

Chapter 12

Simulation Parameters

At the beginning of Part 2, it was mentioned that we discretize by dividing time and space into discrete chunks of width Δt and Δx, respectively. THE_ARGO represents Δt and Δx with the parameters DT and DX, respectively. THE_ARGO is designed to always set DX by

$$\mathrm{DX} = (\mathrm{XMAX} - \mathrm{XMIN})/(\mathrm{NX} - 1) \tag{12.1}$$

where XMAX (XMIN) is the maximum (minimum) of the domain and NX is the number of cells dividing the domain. Though THE_ARGO can use non-square domains, we only used square domains, so we henceforth only mention x-dimension parameters. For an example, a unit wide box divided into 51 cells assigns DX a value of

$$\mathrm{DX} = (1.0 - 0.0)/(51 - 1) = 0.02 \tag{12.2}$$

To keep the parameter tables of the following test problems as short as possible, we omit mention of DX since the presented parameters NX and DSIZE $\equiv$ XMAX - XMIN (domain size) can be used by the reader to calculate DX. Additional parameter names will be explained in the caption of the first table in which they appear.

The assignment of DT varies from problem to problem and hence we list its value in the table of parameters for each problem. We do, however, make sure to keep DT the same between implicit and explicit versions of the same test problem to facilitate comparisons. Usually, the specific value of DT is chosen to guarantee stability for an explicit version of the problem.

Note that we have made no mention of units; like most codes, all values are in non-dimensional units. This is possible by the law of dynamic similarity, which says that fluid (and also MHD) effects are the same at every scale when the ratio of forces are equal (Diogo Bolster and Donnelly 2011). This justifies our use of Lorentz-Heaviside units for the chapter on magnetism since we can set the electric permittivity, magnetic permeability, and speed of light equal to one without loss of information.

Without further ado, let us barrage THE_ARGO with test problems and see how it performs.

Chapter 13

Advection

We first present a test of how well THE_ARGO handles advection, without diffusion or any other process. We initialized the two dimensional velocity field as a hat function

$$\mathbf{u} = \begin{cases} 2, & \text{if } 0.5/\text{DX} < x < 1.0/\text{DX} \text{ and } 0.5/\text{DY} < y < 1.0/\text{DY}. \\ 1, & \text{otherwise.} \end{cases} \tag{13.1}$$

with periodic boundary conditions. The parameters used in this simulation are listed in Table 13.1.

We would expect a nonlinearly advected hat to move in a certain direction, with the velocity greatest in the direction it moves and the velocity tailing off at its rear. We will see how well our discretizations perform by how well the simulations live up to these expectations.

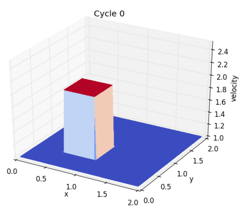
We solved this test problem using both the explicit finite differencing discretization and the implicit backtracking method in order to compare the relative accuracy of the two methods. See Figure 13.1 for plots of the first three hundred cycles; the output from the explicit finite differencing discretization is in the left column and the output from the implicit backtracking discretization is in the right column. As expected, the hat moves in a certain direction and tilts towards its leading side. Notice that the periodic boundary conditions are implemented well; as the hat hits a boundary wall it is carried through the wall on the opposite side. However, it can also be seen that the "hat" is signficantly diffused. How can this be? After all, we didn't include diffusion! Unfortunately, all numerical methods

NX	NT	DSIZE	DT
81	800	2.0	5E-3

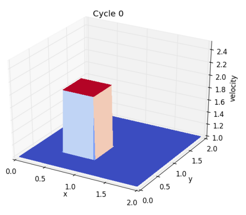
Table 13.1: Parameters for Advection Simulations. NT is the number of time steps.

suffer from "numerical diffusion" and THE_ARGO is no exception.

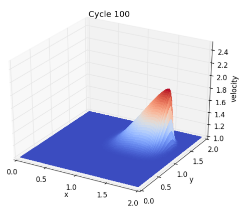
On the bright side, the agreement between the two methods is very good. There is still the possibility however that both are equally bad approximations to the exact solution. Therefore, we next solve a specific one-dimensional advection plus diffusion test problem, for which there exists an analytical solution, in order to determine the accuracy of our numerical solution compared to an exact solution.



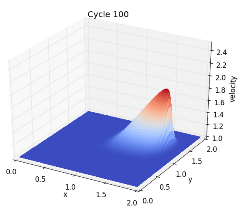
(a)



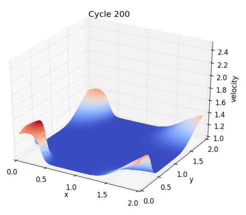
(b)



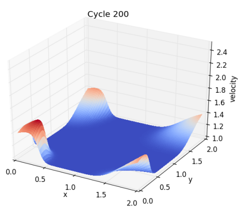
(c)



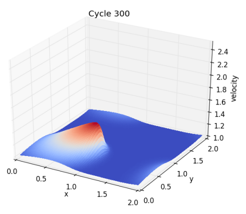
(d)



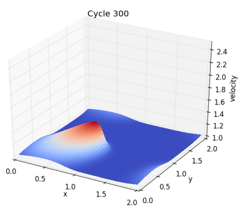
(e)



(f)



(g)



(h)

Figure 13.1: Advection Simulations. Left column: explicit finite differencing solver. Right column: backtracking solver.

Chapter 14

Advection Plus Diffusion in 1D

In the previous chapter, it was remarked that though the simulations of both discretizations look visually identical, they could both be equally bad approximations to the exact solution. In this chapter, we show that that is not the case, by solving a specific problem having a known analytical solution, with which we may compare our numerical solution (the knowledge of this analytical solution was found from Barba 2013).

The specific problem requires that we initialize the velocity as a saw-tooth function

$$u = -\frac{2\eta_\nu}{\phi}\frac{\partial \phi}{\partial x} + 4 \tag{14.1}$$

where ϕ is given by

$$\phi = \exp\left(\frac{-x^2}{4\eta_\nu}\right) + \exp\left(\frac{-(x-2\pi)^2}{4\eta_\nu}\right) \tag{14.2}$$

and implement periodic boundary conditions. This problem has an analytical solution given by

$$u = -\frac{2\eta_\nu}{\phi}\frac{\partial \phi}{\partial x} + 4 \tag{14.3}$$

where ϕ is

$$\phi = \exp\left(\frac{-(x-4t)^2}{4\eta_\nu(t+1)}\right) + \exp\left(\frac{-(x-4t-2\pi)^2}{4\eta_\nu(t+1)}\right) \tag{14.4}$$

The parameters used in this simulation are listed in Table 14.1. Though we have an analytical solution in this case, it is still a good idea to compare our results

NX	NT	DSIZE	DT	η_ν
101	100	2π	4.4E-3	0.07

Table 14.1: Parameters for Advection Plus Diffusion Simulation in 1D. η_ν is the viscous diffusion.

to what we expect from physical intuition. Nonlinear advection should cause the sawtooth to move to the right, with the leading side sharper and the rearside trailing off. Since diffusion is included, we expect the advected sawtooth to largely maintain its shape but to decrease in magnitude over time. Now let us see if our expectations are met.

In Figure 14.1, we have the initial and final time steps of the simulation plotted, with the analtyical solution superimposed on the numerical solution. As can be seen from Figure 14.1, the numerical solution has admirable accuracy and matches well with our physical expectations. As we move forward, we no longer have analytical solutions with which we may test the veracity of THE_ARGO; therefore, we must rely more deeply on our physical intuition of whether or not THE_ARGO produces trustworthy results. Additionally, since we have here shown that both advection schemes are accurate, we are optimistic that a cross-comparison of discretization methods should show discrepancies if there is numerical error. That is, we deem it unlikely that two different schemes will fail in the exact same manner.

In the next chapter, we include the same physics of advection and diffusion, but in two dimensions, initialized again with our old friend the hat function. The purpose of the test problem is to ensure that we scaled up our implementation correctly from one-dimensional diffusion to two-dimensional diffusion.

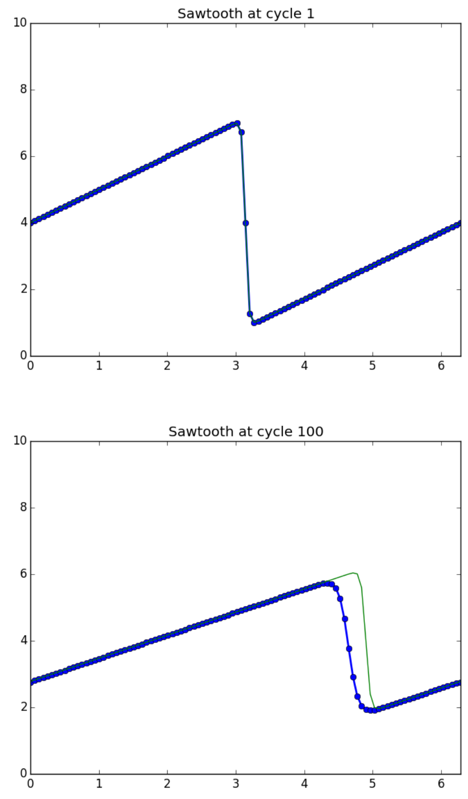


Figure 14.1: Advection Plus Diffusion Simulation in 1D. The analytical solution is the thin green line, and the numerical solution is the thicker blue line with points.

Chapter 15

Advection Plus Diffusion in 2D

In this chapter, we solve 2D advection plus diffusion, discretizing both with explicit finite differencing and backtracking, in order to compare how well the two methods incorporate diffusion.

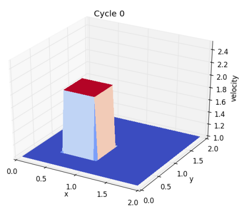
In this test problem, we initialize the velocity field with our old friend the hat function and use periodic boundary conditions. The parameters used in this simulation are listed in Table 15.1.

For these conditions, we physically expect the hat function to diffuse to lower magnitudes while move diagonally towards XMAX and YMAX, with the leading side a sharp face and the rearside trailing off. Again, THE_ARGO performs in accordance with our physical expections; see Figure 15.1 for plots from the first 1200 cycles. Output from the explicit finite differencing discretization is in the left column, and the output from the implicit backtracking and averaging discretization is in the right column.

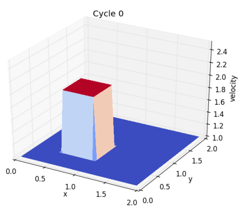
In the next chapter, we test including the pressure gradient term of the fluid equation of motion and see how well it keeps the velocity divergence free.

NX	NT	DSIZE	DT	η_ν
41	1600	2.0	0.05	1E-3

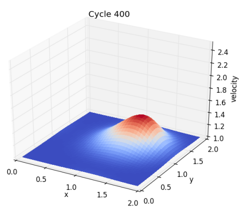
Table 15.1: Parameters for Advection Plus Diffusion in 2D Simulations.



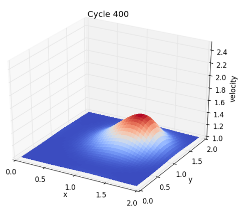
(a)



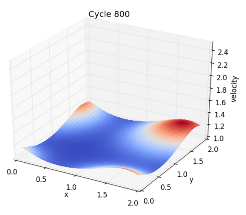
(b)



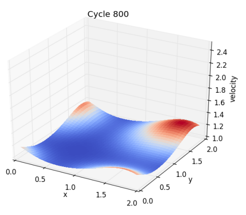
(c)



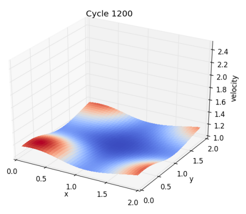
(d)



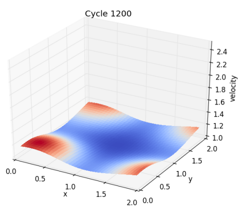
(e)



(f)



(g)



(h)

Figure 15.1: Advection Plus Diffusion in 2D Simulations. Right column: explicit finite differencing solver. Left column: backtracking solver.

Chapter 16

Lid-Driven Cavity Flow

In the previous several chapters, we have extensively tested advection and diffusion solvers of the velocity field. In this chapter, we present a problem which tests how well including the pressure gradient eliminates the divergence. We perform two simulations, in which both use the same relaxation pressure solver, while advection and diffusion are discretized with explicit finite differencing in one and backtracking in the other.

The parameters used in this simulation are listed in Table 16.1. We initialize the pressure, source, and velocity to be zero everywhere in the cavity except on the top wall (the "lid") where the x-component of the velocity is set to positive one (which is why this is a driven-lid cavity flow problem). Additionally, the boundary conditions are to set all variables to zero for all walls except the top wall in which the x-component of the velocity is set to positive one. In other words, the top wall is like a conveyor belt, continuously pushing material to the right at speed one.

In this situation, we would expect the amount of fluid to diminish on the left part of the top wall and pile up on the right part of the top wall, representing nonzero divergence. If the relaxation pressure solver is operating correctly, it should notice the divergence as a lower pressure on the left part of the top wall and the convergence as a higher pressure on the right part of the top wall. Subtracting the pressure gradient, then, should remove the convergence and divergence by causing the velocity at the top right wall to flow down and away from that corner and eventually flow to the left part of the top wall to balance what is pushed away from that region by the top wall conveyor belt. In other words, we expect the velocity field to form a closed circuit.

To check the output with our expectations, we now turn to Figure 16.1 in

NX	NT	NI	DSIZE	DT	η_ν	ρ
41	700	50	2.0	1E-3	0.1	1.0

Table 16.1: Parameters for Lid-Driven Cavity Flow Simulations. NI is the maximum number of iterations to relax the pressure. ρ is the fluid density.

which the ultimate cycle for each discretization is plotted, with the explicit finite difference scheme on top and the backtracking method on the bottom. The arrows represent the velocity with length signifying relative magnitudes and the color representing the pressure; we find our expectations to be wholly vindicated, demonstrating again the utility of THE_ARGO. In the next chapter, we test how well a tracer concentration is advected and diffused by the velocity field.

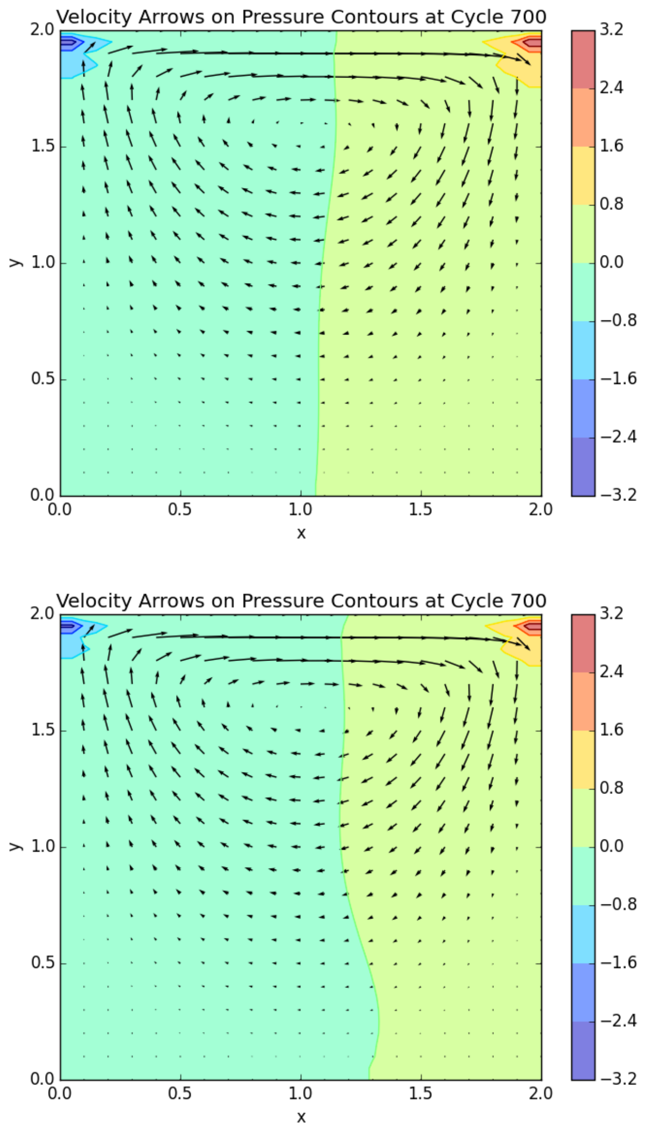


Figure 16.1: Lid-Driven Cavity Flow Simulations. The top plot displays the final time step for the explicit finite differencing scheme; the bottom plot shows the corresponding result from the backtracking scheme (both use the relaxation pressure solver).

Chapter 17

Double Vortex

The previous few chapters have tested all terms of the fluid equation of motion. Those test problems were borrowed from Loren A. Barba's online tutorial "12-Steps to Navier-Stokes" (Barba 2013). Comparing our plots to those displayed in her tutorial, we become espeically confident that THE_ARGO behaves properly in solving the Navier-Stokes equation. In the following two chapters, we move beyond the "12-Steps" and test the implementation of the time evolution of a tracer concentration in this chapter followed by a full MHD problem in the next chapter.

In this test problem, we begin by initializing all variables to zero except the velocity field, which is initialized in the form of two vortices (or vortexes if you're from Texas) and the tracer concentration which is initialized to an amount (the precise value being immaterial) at the four approximately central cells. The vortices are initialized by setting the center points of two initial circles, x_0 and y_0, equal to (25.0, 5.0) and (25.0, 45.0) and setting the components of the velocity field for the first vortex by

$$u = (\mathrm{Y} - y_0)/r^2 \tag{17.1}$$

$$v = -(\mathrm{X} - x_0)/r^2 \tag{17.2}$$

where $r^2 = (\mathrm{X} - x_0)^2 + (\mathrm{Y} - y_0)^2$ is the radius squared, X = [0,1,...NX-1] is the cell number in the x-dimension, and Y is similarly defined in the y-dimension. Note that the velocity field is set to zero at the center of the circle since otherwise a division by zero error would occur.

The second vortex is included by adding to the velocity field a reformulation of the R.H.S. of Eqs. 17.1 and 17.2 with the other values of x_0 and y_0 used and with the signs of u and v reversed. Reversing the signs of u and v reverses the direction (clockwise vs. counterclockwise) of the vortex flow; the reason for doing so is discussed momentarily.

Zero flux boundary conditions were applied to all variables, which set the derivative of the variable normal to the surface equal to zero to maintain "hard" boundary walls so no fluid can "spill out." The parameters used in this simulation are listed in Table 17.1. Note that we do not include any explicit diffusion in this problem; numerical diffusion is all the diffusion we need in this problem.

The terms in the fluid equation of motion and in the equation governing the time evolution of the tracer concentration were discretized using backtracking for all types of advection and diffusion, and relaxation to find the pressure.

We next want to consider what we expect THE_ARGO will produce based on physical intuition. To do so, it is necessary to know that the motion of vortices along a boundary wall obey a similar method of images as is encountered in undergraduate electrodynamics (Eggers n.d.). The method of images applied to vortices says that beyond each wall, there exists a "mirror" vortex. The vortex is said to be a "mirror" of the physical vortex since the direction of its vortical flow is reversed. The effect of two adjacent vortices of opposite sign is for each to push the other in the direction of its rotation with a force proportional to their inverse separation; consequently, an important aspect of the the mirror vortex that must be mentioned is that it and the physical vortex are equidistant from the wall. Thus, the vortex initialized at the bottom wall with a clockwise sense of direction will be pushed towards the left wall and then down that wall by its mirror vortex, and the vortex initialized at the top wall with a counterclockwise sense of direction will be pushed towards the left wall and then up that wall by its mirror vortex. Thus, our two oppositely signed vortices shall be pushed towards each other for a confrontation! As they approach, the real vortices begin acting

NX	NT	NI	DSIZE	DT	η_ν	ρ	η_c
51	6.4E4	200	1.0	0.2	0	1.0	0

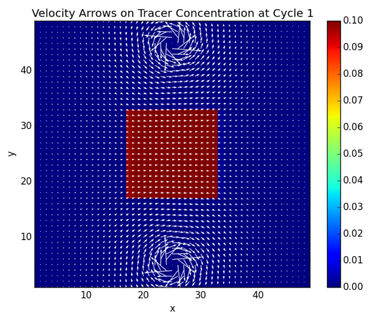
Table 17.1: Parameters for the Double Vortex Simulation.

on each other and push each other along the center of the box towards the right wall.

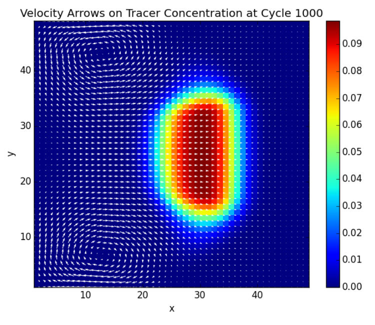
As can be seen from Figure 17.1, THE_ARGO once again accords with our expectations. Note that since NX=51, there is no exact center four zones; the initial tracer concentration is more towards the bottom and consequently the bottom vortex picks up more of the tracer concentration.

This double vortex problem also tested using a DT in excess of the CFL condition; Figure 17.1 shows the output for DT twenty times larger than the CFL condition would permit! Note that the simulation already took many cycles; it would have taken twenty times more cycles for an explicit solver. Though not shown here, we performed a much longer run with a DT satisfying the CFL condition. We found that the only change from the simulation presented in this chapter was the expected change in number of time steps to reach the same total time.

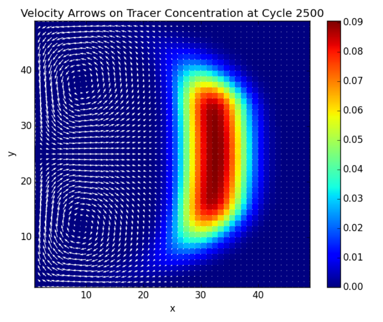
In the next chapter, we perform a test of including magnetism.



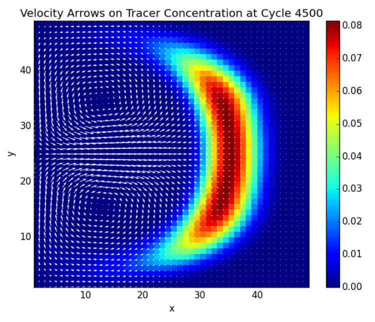
(a)



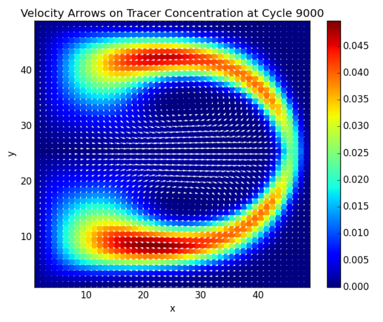
(b)



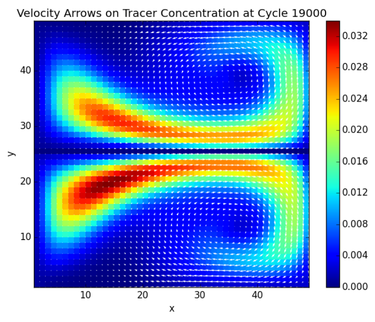
(c)



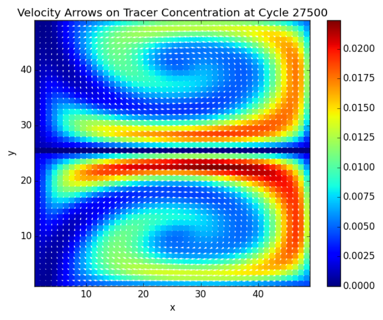
(d)



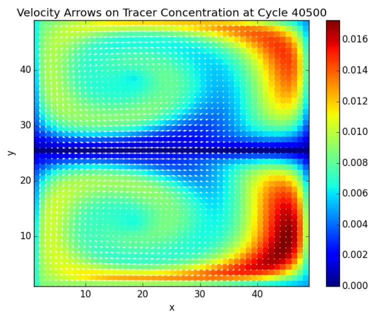
(e)



(f)



(g)



(h)

Figure 17.1: Double Vortex Simulation. Plotted are color meshes of the tracer concentration with superimposed arrows whose lengths are proportional to the magnitude of the velocity, alphabetically ordered. Regions colored red (blue) signify areas of high (low) concentrations. Note that the maximum concentration plotted (deepest red, look at the colorbar) decreases over time, as expected from numerical diffusion.

Chapter 18

The Orszag-Tang Vortex

In the past several chapters, we have extensively tested THE_ARGO's implementation of the fluid equation of motion as well as the time evolution of a tracer concentration. In this chapter, we test THE_ARGO's ability to include magnetic fields with the two-dimensional Orszag-Tang vortex, a full MHD test problem (Orszag and Tang 1979).

Our code is incompressible, which greatly simplifies the physics but is uncommon for MHD codes. In terms of validation, the best we can do is to compare our results with the battle-tested Athena MHD code, though it is fundamentally a compressible code (Stone et al. 2008).

To begin, we initialize all variables to zero. Then, we reset the velocity field to have a vortex at the center of the domain and the magnetic field to have four equally spaced "loops" (the "vortex equivalent" for magnetic fields) along the x-dimension and two equally spaced loops along the y-dimension.

The vortices were initialized with

$$u = -\sin(\mathrm{Y}) \tag{18.1}$$

$$v = \sin(\mathrm{X}) \tag{18.2}$$

$$A = \cos(\mathrm{Y}) + 0.5\cos(2\mathrm{X}) \tag{18.3}$$

where A is the z-component of the vector potential[15], and the magnetic field components were calculated by taking the curl of A in the manner presented in

[15]In two dimensions, we consider only the z-component of the vector potential, setting $A_x = A_y = 0$. Otherwise, the calculated magnetic field would have a non-zero third dimension.

Section 7.1

$$B_x = \frac{\partial A}{\partial y} \tag{18.4}$$

$$B_y = \frac{\partial A}{\partial x} \tag{18.5}$$

where the partials were discretized via first order central differencing. Additionally, the boundary conditions are periodic for all variables along all boundaries. The parameters used in this simulation are listed in Table 18.1.

Note the change from the method of initializing vortices (and loops) presented last chapter. For one thing, the present method is much more economical in initializing multiple vortices. But additionally, the vortices in the Double Vortex test problem were initialized with some divergence, which was acceptable since it was immediately removed. The magnetic field has no equivalent term to remove new divergence; whatever divergence it is initialized with will stay with it. That is why we changed the initialization method.

In this problem, speed became a limiting factor. We therefore replaced the relaxation pressure solver with the Fourier transform technique, yielding a six-fold increase in speed. Though not presented here, simulations performed with the Fourier transform technique versus the relaxation pressure solver produce the same results, validating our implementation of the Fourier transform technique. All other discretizations used the backtracking methods for advection and diffusion.

The purpose of the Orszag-Tang test problem is typically to determine how well a code handles supersonic turbulence and MHD shocks. Since our code is incompressible, it is only accurate for subsonic velocities. We therefore do not expect our velocity field to very well match Athena. However, we hope our magnetic fields are correctly perturbed by MHD shocks.

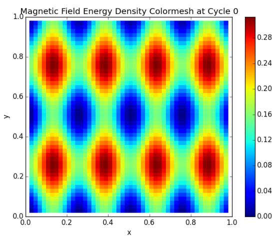
The evolution of the magnetic field can be seen in Figure 18.1, and the effects of the magnetic field on the velocity field's evolution can be seen in Figure 18.2. In

NX	NT	Domain Size	DT	η_ν	ρ	η_ρ
51	2100	2π	5E-4	0	$\frac{25}{36\pi}$	0

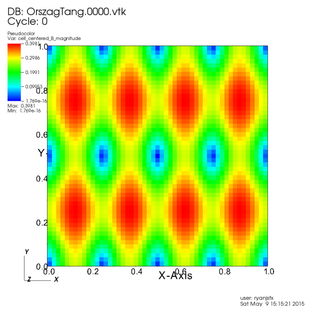
Table 18.1: Parameters for The 2D Orszag-Tang Vortex Simulation. η_ρ is the resistivity of the fluid.

both cases, the left column contains plots from THE_ARGO's simulation and the right column contains plots from Athena's simulation. To show how each evolved, we have plotted the initial time step and three later time steps, each spaced a hundred cycles apart (note that Athena displays only the dumped file name, not the cycle at which it dumped).

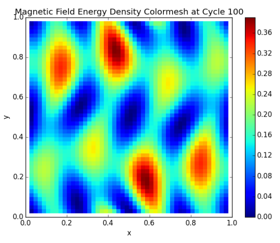
Our magnetic field energy density closely matches Athena's, which is very encouraging! However, as expected from the assumption of incompressibility, our velocity field is much more stagnant than Athena's. While Athena's velocity field recoils more uniformly from the buildup of overdensities and underdensities, THE_ARGO does not simulate this behavior. However, the degree of difference does allow for the worry that THE_ARGO is not yet including the effects of magnetic fields on the velocity field correctly.



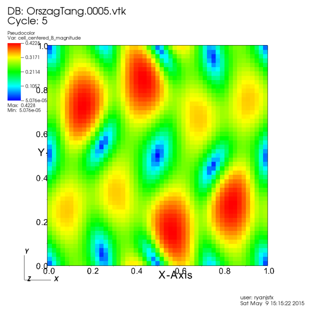
(a)



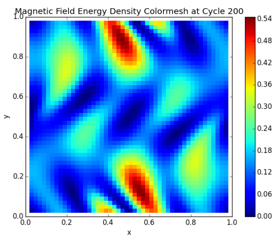
(b)



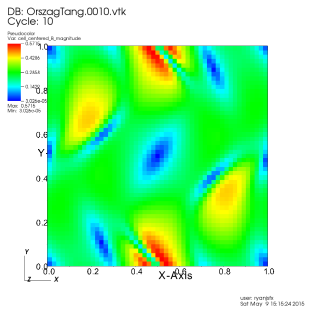
(c)



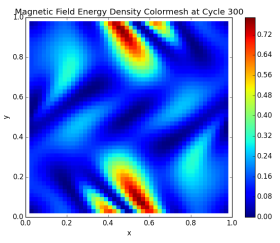
(d)



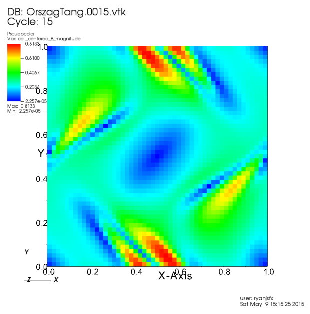
(e)



(f)

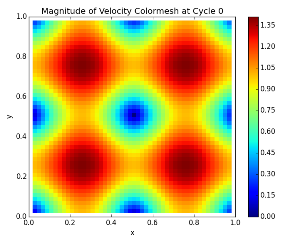


(g)

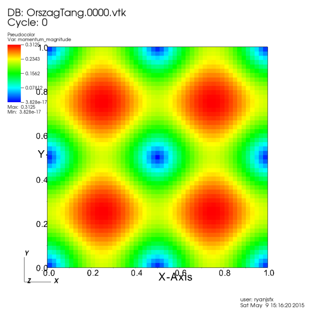


(h)

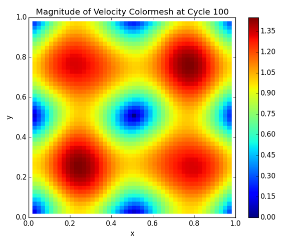
Figure 18.1: The Orszag-Tang Vortex Simulations: magnetic energy density. All plots are color meshes of the magnetic energy density; the left column contains THE_ARGO's simulations while the right column displays Athena's simulations.



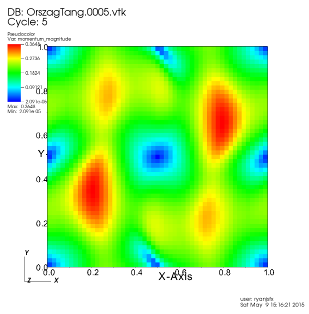
(a)



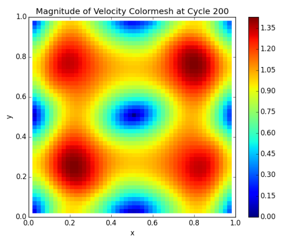
(b)



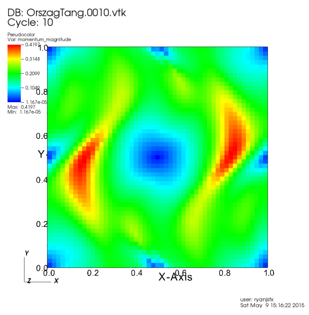
(c)



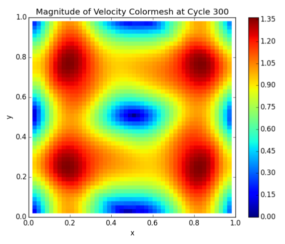
(d)



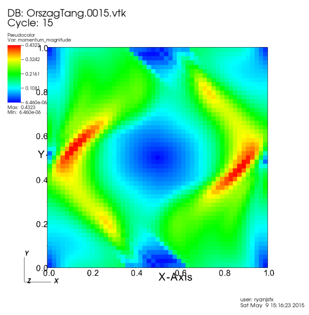
(e)



(f)



(g)



(h)

Figure 18.2: The Orszag-Tang Vortex Simulations: magnitude of the velocity. All plots are color meshes of the magnitude of the velocity field; the left column contains THE_ARGO's simulations while the right column displays Athena's simulations.

Chapter 19

Conclusions

We have presented a gentle introduction to numerical MHD for motivated students at the advanced undergraduate level. We began with informal derivations of the MHD equations and found that they may be written in a form that displays a high degree of symmetry; through the nine terms occurring in three equations, the MHD equations have only three distinct physical processes: advection, diffusion, and pressure.

We proceeded by discussing the numerical methods used in THE_ARGO to discretize these processes. First, we presented the standard finite differencing methods, which are still used in the latest version of THE_ARGO to discretize pressure gradients and to calculate the magnetic field from the vector potential. We then presented the technique of backtracking, which guarantees numerical stability, allowing for larger time steps to be used than the CFL condition would permit. Next, we presented the relaxation method as a means of finding the fluid pressure specifically to eliminate divergence from the velocity field. The last numerical method we presented, the FFT, was implemented in THE_ARGO to replace the intolerably slow relaxation pressure solver. Beyond the student of numerical MHD for whom this work is primarily aimed, we hope the scientific community may find utility in this work for the presentation of its novel use of working primarily in physical space and only using the FFT to find the pressure; typically all variables are Fourier transformed to frequency space and only transformed back to physical space for saving the physical value of the variables.

Physicists are interested in obtaining knowledge of the real world. As a result, the physicist desires validation of any code applied to physical problems in order to ensure its physical accuracy. We presented a slew of test problems, from the drab advection of a hat function, to the pretty double vortex, validating THE_ARGO's ability to accurately solve the fluid equations. Last, but certainly not least, we presented the Orszag-Tang Vortex simulation, a full MHD test problem. Though no outside facsimile was available, plots from the compressible MHD code Athena were consistent with our simulations to the degree expected. That is, the magnetic field evolution was very similar, but the velocity field (strongly affected from compressibility effects for Athena or perhaps by a programming bug) was quite different. For the interested reader, we present the source code of THE_ARGO in the following appendix.

Appendix A

THE_ARGO Source Code

```
## the_fluid.py
## by Ryan Farber 20 January 2015
## Last modified: 21 January 2015
"""
The_Fluid contains all the instance wide constants (attributes)
 that will be used by the_argo to solve fluid mechanics
problems; these attributes and The_Fluid instance are
instantiated by the input file of a problem.
"""
class The_Fluid(object):
    def __init__(self, NX,NY,NZ, DX,DY,DZ):

        self.NX = NX    # number of zones in x
        self.NY = NY
        self.NZ = NZ

        self.DX = DX    # width of a cell (zone) in x
        self.DY = DY
        self.DZ = DZ
    # end __init__
# end class The_Fluid

## end the_fluid.py
## the_solver_2d.py
## by Ryan Farber 28 January 2015
## Last modified: 03 May      2015

import sys; sys.path.insert(0, "../")
from the_fluid import The_Fluid

class The_Solver_2D(The_Fluid):
    """
    The_Solver_2D inherits __init__ from The_Fluid and requires
    minimally as input variables for instantiation:
    NX,NY,NZ (the number of zones in x,y, and z) and
    DX,DY,DZ (the width of a cell [zone] in x,y, and z).
    See an input file for further on instantiating a
    The_Solver_2D instance.

    The_Fluid_Solver_2D includes finite differencing methods
    for solving differential equations.
    """

    def back_diff_1st_2d(self, f, coeff1,coeff2):
        """Applies the 1st derivative of 2D field f (input) to
        a 2D field g (output), by backward differencing."""

        g  = coeff1*( f[ 1:-1, 1:-1 ] - f[  :-2, 1:-1 ] ) /
                                        self.DX
        g += coeff2*( f[ 1:-1, 1:-1 ] - f[ 1:-1,  :-2 ] ) /
                                        self.DY

        return g
    # end back_diff_2d
```

```
def central_diff_implicit_1st_2d(self, f, coeff1,coeff2):
    """Applies the 1st derivative of a 2D field f (input)
    to a 2D field g (output), by central differencing."""

    g  = coeff1*( f[ 2:  , 1:-1 ] + f[  :-2, 1:-1 ] ) /
                                (2*self.DX)
    g += coeff2*( f[ 1:-1, 2:   ] + f[ 1:-1,  :-2 ] ) /
                                (2*self.DY)

    return g
# end central_diff_implicit_1st_2d


def central_diff_1st_2dX(self, f, coeff):
    """Applies the 1st derivative of the x-component of a
    2D field f (input) to a 2D field g (output), by central
     differencing."""

    g = coeff*( f[ 2:  , 1:-1 ] - f[  :-2, 1:-1 ] ) /
                                (2*self.DX)

    return g
# end central_diff_1st_2dX


def central_diff_1st_2dY(self, f, coeff):
    """Applies the 1st derivative of the y-component of a
    2D field f (input) to a 2D field g (output), by central
     differencing."""
```

```
        g = coeff*( f[ 1:-1, 2:   ] - f[ 1:-1,  :-2 ] ) /
                                   (2*self.DY)


        return g
    # end central_diff_1st_2dY



    def central_diff_2nd_2d(self, f, coeff1,coeff2):
        """Applies the 2nd derivative of a 2D field f (input)
        to a
        2D field g (output), by central differencing."""

        g = (coeff1/self.DX**2 *
  (f[ 2:  , 1:-1 ] - 2*f[ 1:-1, 1:-1 ] + f[  :-2, 1:-1 ])

          +  coeff2/self.DY**2 *
  (f[ 1:-1, 2:   ] - 2*f[ 1:-1, 1:-1 ] + f[ 1:-1,  :-2 ])
              )
        return g
    # end central_diff_2nd_2d

# end The_Solver_2D

## end the_solver_2d.py
## the_fluid_solver_2d.py
## by Ryan Farber 20 January 2015
## Last modified: 10 May      2015

import numpy as np
from the_solver_2d import The_Solver_2D

class The_Fluid_Solver_2D(The_Solver_2D):
    """
    The_Fluid_Solver_2D inherits __init__ from The_Fluid and
    requires minimally as input variables for instantiation:
    NX,NY,NZ (the number of zones in x,y, and z) and DX,DY,DZ
    (the width of a cell [zone] in x,y, and z). See an input
    file for further on instantiating a The_Fluid_Solver_2D
    instance. The_Fluid_Solver_2D includes methods for solving
    the navier stokes equations in two dimensions; however, the
    flows must be: constant density, constant viscosity,
    incompressible, and isothermal fluid flows.
    """

    def linear_advect_explicit_2d(self, f):
        """Performs linear advection of a 2D field,
        by explicit backward differencing."""

        return self.DT*self.back_diff_1st_2d(f, self.C,self.C)
    # end linear_advect_explicit_2d



    def linear_advect_implicit_2d(self, f, XX,YY):
        """Performs linear advection of a 2D field (f)
        implicitly, keeping all values bounded within the
        domain, by cell-centered back-tracking and applying
        necessary weights."""

        x = XX - (self.DT/self.DX * self.C)
        y = YY - (self.DT/self.DY * self.C)

        x = np.where(x < 0.5, 0.5, x)
        y = np.where(y < 0.5, 0.5, y)

        x = np.where(x > (self.NX-2) + 0.5,
                     (self.NX-2) + 0.5, x)
        y = np.where(y > (self.NY-2) + 0.5,
                     (self.NY-2) + 0.5, y)

        i0 = x.astype(int); j0 = y.astype(int)
        i1 = i0 + 1; j1 = j0 + 1

        s1 = x - i0; t1 = y - j0
        s0 = 1 - s1; t0 = 1 - t1

        return (s0 * (t0 *  f[ i0, j0 ] + t1 *  f[ i0, j1 ])
             + s1 * (t0 *  f[ i1, j0 ] + t1 *  f[ i1, j1 ]))
    # end linear_advect_implicit_2d


    def linear_advect_implicit_periodic_2d(self, f, XX,YY):
        """Performs linear advection of a 2D field (f)
        implcitly, with periodic boundaries, by cell-centered
        back-tracking and applying necessary weights."""

        x = XX - (self.DT/self.DX * C)
        y = YY - (self.DT/self.DY * C)

        x = x % (self.NX - 2)
        y = y % (self.NY - 2)

        i0 = x.astype(int); j0 = y.astype(int)
        i1 = i0 + 1; j1 = j0 + 1

        s1 = x - i0; t1 = y - j0
        s0 = 1 - s1; t0 = 1 - t1

        return (s0 * (t0 *  f[ i0, j0 ] + t1 *  f[ i0, j1 ])
             +  s1 * (t0 *  f[ i1, j0 ] + t1 *  f[ i1, j1 ]))
    # end linear_advect_implicit_periodic_2d


    def nonlinear_advect_explicit_2d(self, f, fx,fy):
        """Performs nonlinear advection of 2D field (f), by 2D
        vector field (fx,fy) by explicit backward differencing.
        """

        return self.DT*self.back_diff_1st_2d(f,
                  fx[ 1:-1, 1:-1 ], fy[ 1:-1, 1:-1 ])
    # end nonlinear_advect_explicit_2d


    def nonlinear_advect_implicit_2d(self, f, fx,fy, XX,YY):
        """Performs nonlinear advection of 2D field (f) by 2D
        vector field (fx,fy), keeping all values bounded within
         the domain, by cell-centered back-tracking and
        applying necessary weights."""
        x = XX - (self.DT/self.DX * fx[1:-1,1:-1])
        y = YY - (self.DT/self.DY * fy[1:-1,1:-1])

        x = np.where(x < 0.5, 0.5, x)
        y = np.where(y < 0.5, 0.5, y)

        x = np.where(x > (self.NX-2) + 0.5,
                     (self.NX-2) + 0.5, x)
        y = np.where(y > (self.NY-2) + 0.5,
                     (self.NY-2) + 0.5, y)

        i0 = x.astype(int); j0 = y.astype(int)
        i1 = i0 + 1; j1 = j0 + 1

        s1 = x - i0; t1 = y - j0
        s0 = 1 - s1; t0 = 1 - t1

        return (s0 * (t0 *  f[ i0, j0 ] + t1 *  f[ i0, j1 ])
             + s1 * (t0 *  f[ i1, j0 ] + t1 *  f[ i1, j1 ]))
    # end nonlinear_advect_implicit_2d


    def nonlinear_advect_implicit_periodic_2d(self, f, fx,fy,
                                               XX,YY):
        """Advection by 2D field (fx,fy) of any component of a
        2D field (f), with periodic boundaries, by cell-
        centered back-tracking and applying necessary weights.
        """

        x = XX - (self.DT/self.DX * fx[1:-1,1:-1])
        y = YY - (self.DT/self.DY * fy[1:-1,1:-1])

        x = x % (self.NX - 2)
        y = y % (self.NY - 2)

        i0 = x.astype(int); j0 = y.astype(int)
        i1 = i0 + 1; j1 = j0 + 1

        s1 = x - i0; t1 = y - j0
        s0 = 1 - s1; t0 = 1 - t1

        return (s0 * (t0 *  f[ i0, j0 ] + t1 *  f[ i0, j1 ])
             +  s1 * (t0 *  f[ i1, j0 ] + t1 *  f[ i1, j1 ]))
    # end nonlinear_advect_implicit_periodic_2d


    def diffuse_explicit_2d(self, f):
        """Performs diffusion of a 2D field,
        by explicit central differencing;
        viscosity is assumed to be constant."""

        return self.DT*self.central_diff_2nd_2d(f, self.NU,
                                                    self.NU)
    # end diffuse_explicit_2d


    def diffuse_implicit_2d(self, f0,f, diff_coeff):
        """Performs diffusion of a 2D field implicitly;
        diff_coef (NU or ETA is assumed to be constant."""

        return ( (f0[1:-1,1:-1]
+ (diff_coeff*self.DT)/(self.DX**2 * self.DY**2)

* (self.DY**2 * (f[2:  , 1:-1] + f[ :-2, 1:-1])
+  self.DX**2 * (f[1:-1, 2:  ] + f[1:-1,  :-2])) )

/ (1 + (2*diff_coeff*self.DT) / (self.DX**2 * self.DY**2)
                          * (self.DY**2 + self.DX**2)) )
    # end diffuse_implicit_2d



    def apply_pressure_2dX(self, p, c):
        """Applies the pressure gradient to the x-component of
        a 2D field, by central differencing; c is assumed to be
        constant (density or magnetic permeability)."""

        return self.DT*self.central_diff_1st_2dX(p, c)
    # end apply_pressure_2dX



    def apply_pressure_2dY(self, p, c):
        """Applies the pressure gradient to the y-component of
        a 2D field, by central differencing; c is assumed to be
        constant (density or magnetic permeability)."""

        return self.DT*self.central_diff_1st_2dY(p, c)
    # end apply_pressure_2dY



    def apply_force_2d(self, g):
        """Applies the acceleration due to a force [such as
        gravity] (g) to a component of a 2D field (f)."""

        return self.DT*g[ 1:-1, 1:-1 ]
    # end apply_force_2d



    def calc_source_2d(self, u,v):
        """Calculates the source term of the pressure poisson
        equation; for the divergence terms."""

        return (self.central_diff_1st_2dX(u, self.RHO/self.DT)
               + self.central_diff_1st_2dY(v, self.RHO/self.DT)
               )
    # end calc_source_2d



    def relax_pressure_poisson_2d(self, p, src):
        """Solves the poisson equation for a 2D pressure field
        by central differencing in both dimensions. This solves
        the laplace equation for a 2D pressure field
        when src=0"""

        p[1:-1, 1:-1] = ((self.DY**2 *
                          (p[2:  , 1:-1] + p[ :-2, 1:-1])
                        +   self.DX**2 *
                          (p[1:-1, 2:  ] + p[1:-1,  :-2])

                        -   self.DX**2 * self.DY**2 *
                          src[1:-1,1:-1])

                        / (2*(self.DX**2 + self.DY**2)) )
```

```
        return p
    # end relax_pressure_poisson_2d



    def transform_pressure_poisson_2d(self, p, src):
        """Solves the poisson equation for a 2D pressure field
        by the Fast Fourier Transform (fft). This solves the
        laplace equation for a 2D pressure field when src=0"""

        srcTrans = np.fft.fft2(src[1:-1,1:-1])

        kx,ky = np.meshgrid(np.fft.fftfreq(self.NX-2,
                                           d=self.DX),
                            np.fft.fftfreq(self.NY-2,
                                           d=self.DY))

        denom = 1.0/(4 - 2*np.cos(2*np.pi*kx*self.DX)
                       - 2*np.cos(2*np.pi*ky*self.DY))
        denom[0,0] = 0

        p[1:-1,1:-1] = np.real_if_close(np.fft.ifft2(-srcTrans*
                                    denom*self.DX*self.DY))

        return p
    # end transform_pressure_poisson_2d



    def mag_curl_term_2dX(self, u,v, Bx,By):
        """Applies the x-component of curl(u x B) to evolve
        the x-component of the magnetic field."""
        return self.DT*(
        By[1:-1,1:-1]*( u[1:-1,2:] - u[1:-1,:-2])/self.DY
    +    u[1:-1,1:-1]*(By[1:-1,2:] - By[1:-1,:-2])/self.DY
    -   Bx[1:-1,1:-1]*( v[1:-1,2:] - v[1:-1,:-2])/self.DY
    +    v[1:-1,1:-1]*(Bx[1:-1,2:] - Bx[1:-1,:-2])/self.DY
                        )
    # end mag_curl_term_2dX


    def mag_curl_term_2dY(self, u,v, Bx,By):
        """Applies the y-component of curl(u x B) to evolve
        the y-component of the magnetic field."""

        return self.DT*(
       -By[1:-1,1:-1]*( u[2:,1:-1] - u[:-2,1:-1])/self.DX
    +    u[1:-1,1:-1]*(By[2:,1:-1] - By[:-2,1:-1])/self.DX
    -   Bx[1:-1,1:-1]*( v[2:,1:-1] - v[:-2,1:-1])/self.DX
    +    v[1:-1,1:-1]*(Bx[2:,1:-1] - Bx[:-2,1:-1])/self.DX
                        )
    # end mag_curl_term_2dY


    def mag_diffuse_2d(self, fx,fy):
        """Performs diffusion of a 2D field, by central
        differencing; resistivity and permeability are assumed
        to be constant."""

        return np.array([self.DT*self.central_diff_2nd_2d(fx,
                            self.ETA/self.MU, self.ETA/self.MU),
                            self.DT*self.central_diff_2nd_2d(fy,
                            self.ETA/self.MU, self.ETA/self.MU)])
    # end mag_diffuse_2d



    def mag_diffuse_implicit_2d(self, Bx,By):
        """Performs implicit diffusion of the magnetic field;
        resistivity and permeability are assumed to be constant
        """

        const = (self.ETA/self.MU*self.DT) /
                (self.DX**2 * self.DY**2)

        return np.array([Bx[1:-1,1:-1] / (
                          (1+2*const*(self.DY**2+self.DX**2)))
+ const / (      (1+2*const*(self.DY**2+self.DX**2)))
    *(self.DY**2*(Bx[2:,1:-1]+Bx[:-2,1:-1])
+ self.DX**2*(Bx[1:-1,2:]+Bx[1:-1,:-2])),

                                 By[1:-1,1:-1] / (
                          (1+2*const*(self.DY**2+self.DX**2)))
+ const / (      (1+2*const*(self.DY**2+self.DX**2)))
    *(self.DY**2*(By[2:,1:-1]+By[:-2,1:-1])
+ self.DX**2*(By[1:-1,2:]+By[1:-1,:-2]))])
    # end mag_diffuse_implicit_2d

# end The_Fluid_Solver_2D

## end the_fluid_solver_2d.py
## the_state_saver.py
## by Ryan Farber 20 January 2015
## Last modified: 21 January 2015
"""
The purpose of this file is to save the state of the fluid
processed by the_argo by pickling the data.

NOTE: using cpickle can be up to 1000x faster than pickle; also
 note pickle is not secure so do not deserialize from untrusted
 sources; also note that derserializing won't work if changes
to the class are made, so this is not good for long-term
storage.

To load data from a state file, use
    my_fluid = cPickle.load(open("file_name.p","rb"))
"""

import cPickle

def save_state(the_data, file_name):
    """Takes as input an array (alone or an array of arrays)
    and saves it by pickling to file of name 'file_name'."""

    cPickle.dump(the_data, open(file_name, "wb"))
# end save_state

## end the_state_saver.py
## INPUTS_ImplicitOrszagTang2D.py
## by Ryan Farber 11 February 2015
## Last modified: 04 May      2015
"""
This is an INPUTS file tailored for the ImplicitOrszagTang2D
problem.

All values chosen to imitate athena:
http://www.astro.princeton.edu/~jstone/Athena/tests/orszag-tang
/pagesource.html

NOTE: NX,NY,NZ,DX,DY, and DZ are required constants to
instantiate the my_fluid object; additional problem specific
constants are added after creating the my_fluid object below.
"""
import numpy as np
import os; cur_path = os.getcwd()
import sys; sys.path.insert(0, '../..'); sys.path.insert(0,
'../../..')
from the_fluid_solver_2d import The_Fluid_Solver_2D
from the_state_saver import save_state

##General Constants
NX =  51    # number of zones in x
NY =  51    # number of zones in y
NZ = "NA"   # not applicable to a 2D problem

XMIN = 0.0
XMAX = 1.0
YMIN = 0.0
YMAX = 1.0
```

```
DX = (XMAX - XMIN) / (NX-1)  # width of a cell in x
DY = (YMAX - YMIN) / (NY-1)  # width of a cell in y
DZ = "NA"
##End General Constants



my_fluid = The_Fluid_Solver_2D(NX,NY,NZ, DX,DY,DZ)



##Problem Constants
my_fluid.cycle_start = 1          # start cycle; use 1 as default
my_fluid.S    = 1E-1              # stability constant(CFL value)
my_fluid.NI   = int(2e2)          # number of iterations

my_fluid.RHO = 25.0/(36*np.pi)    # density of the fluid
my_fluid.MU  = 1.0                # magnetic permeability
my_fluid.NU  = 0                  # viscous diffusion
my_fluid.ETA = 0                  # resistivity (mag diffusion)

my_fluid.DT  = 5E-4
my_fluid.NT  = int(1.05/my_fluid.DT) # number of time steps
##End Problem Constants



my_fluid.SAVE_FREQ = int(0.01/my_fluid.DT)



##Plotting Constants
my_fluid.XMIN = XMIN
my_fluid.XMAX = XMAX
my_fluid.YMIN = YMIN
my_fluid.YMAX = YMAX

my_fluid.PLT_TYP       = """quiver pcolormesh concentration
                                                   and mag"""
my_fluid.LABEL         = "ot_vortex"
my_fluid.MAX_CYC_MAG = 9 # magnitude of max cycles
##End Plotting Constants



os.chdir(cur_path + "/StateFiles")
save_state(my_fluid, "my_fluid.p")



## end INPUTS_ImplicitOrszagTang2D.py
## the_file_name_getter.py
## by Ryan Farber 21 January 2015
## Last modified: 22 January 2015
"""
Prepends zeros to the cycle number to be used in the saved file
 name so that files show up in the correct order on drive.
"""
def get_file_name(cycles,max_cyc_mag, label,ext):
    """Requires as input the current cycle (as float) and the
    maximum number of cycles (as float) that will be run; also
    requires as input the my_fluid 'LABEL' attribute (as string
    ) as well as the file name extension (as string).
    Ouputs the cycle label as a string."""

    cyc_lbl = ""
    for i in xrange(1, max_cyc_mag+1):
        if cycles < 10**i:
            cyc_lbl   = "0"*(max_cyc_mag- i) + str(cycles)
            file_name = label + "_cycle_" + cyc_lbl + ext
            return file_name
        # end if
    # end for
# end get_file_name

## end the_file_name_getter.py
## implicit_orszag_tang_athena_2d.py
## by Ryan Farber 11 February 2015
## Last modified: 04 May       2015
"""
The purpose of this program is to apply the_argo to propagate
the orszag-tang vortex implicitly, a 2D ideal MHD problem,
in incompressible form.

I am attempting to imitate as closely as possible:
http://www.astro.princeton.edu/~jstone/Athena/tests/orszag-tang
/pagesource.html
"""
import glob
import numpy as np
import sys; sys.path.insert(0, "../../..")
from the_file_name_getter   import get_file_name
from the_state_saver        import save_state
from the_fluid              import The_Fluid
import os; os.chdir("./StateFiles")
import cPickle; my_fluid = cPickle.load(open("my_fluid.p", "rb"
))

##Setup
ext = ".p"  # filename extension for pickling
X = np.linspace(my_fluid.XMIN, my_fluid.XMAX, my_fluid.NX)
Y = np.linspace(my_fluid.YMIN, my_fluid.YMAX, my_fluid.NY)
Y,X = np.meshgrid(X,Y)

XX,YY = np.mgrid[1:my_fluid.NX-1, 1:my_fluid.NY-1] #imitates 2d
 for loop
```

```
if my_fluid.cycle_start == 1:
    u   = np.zeros((my_fluid.NX,my_fluid.NY)) # x-component of
    velocity
    v   = np.zeros((my_fluid.NX,my_fluid.NY)) # y-component of
    velocity
    p   = np.zeros((my_fluid.NX,my_fluid.NY)) # pressure
    src = np.zeros((my_fluid.NX,my_fluid.NY)) # src term for
    poisson eqn
    Bx  = np.zeros((my_fluid.NX,my_fluid.NY)) # x-comp of
    magnetic field
    By  = np.zeros((my_fluid.NX,my_fluid.NY)) # y-comp of
    magnetic field
    A   = np.zeros((my_fluid.NX,my_fluid.NY)) # magnet vector
    potential

    ##Initialize single vortex for velocity; double vortex for
    mag field
    u  = -np.sin(2*np.pi*Y)
    v  =  np.sin(2*np.pi*X)

    B0 = 1.0/np.sqrt(4*np.pi)
    A  = B0*np.cos(4*np.pi*X)/(4*np.pi) + B0*np.cos(2*np.pi*Y)
    /(2*np.pi)

    ##Update ghost zones so boundaries are periodic
    u[ 0, : ] = u[ -2,  : ]; u[ -1,  : ] = u[ 1, : ]
    u[ :, 0 ] = u[  :, -2 ]; u[  :, -1 ] = u[ :, 1 ]

    v[ 0, : ] = v[ -2,  : ]; v[ -1,  : ] = v[ 1, : ]
    v[ :, 0 ] = v[  :, -2 ]; v[  :, -1 ] = v[ :, 1 ]
    A[ 0, : ] = A[ -2,  : ]; A[ -1,  : ] = A[ 1, : ]
    A[ :, 0 ] = A[  :, -2 ]; A[  :, -1 ] = A[ :, 1 ]



    Bx[1:-1,1:-1] =  (A[1:-1,2:] - A[1:-1,:-2]) / my_fluid.DY
    By[1:-1,1:-1] = -(A[2:,1:-1] - A[:-2,1:-1]) / my_fluid.DX

    ##Update ghost zones so boundaries are periodic
    Bx[ 0, : ] = Bx[ -2,  : ]; Bx[ -1,  : ] = Bx[ 1, : ]
    Bx[ :, 0 ] = Bx[  :, -2 ]; Bx[  :, -1 ] = Bx[ :, 1 ]

    By[ 0, : ] = By[ -2,  : ]; By[ -1,  : ] = By[ 1, : ]
    By[ :, 0 ] = By[  :, -2 ]; By[  :, -1 ] = By[ :, 1 ]

    ##Save the state of the initial condition of the fluid
    cycles = 0; file_name = get_file_name(cycles, my_fluid.
    MAX_CYC_MAG,
                                                    my_fluid.
                                                    LABEL, ext)
    save_state([cycles, u,v, "NA", p,src, "NA","NA", Bx,By,A],
               file_name )
else:
    data_file = glob.glob("*" + str(my_fluid.cycle_start) + "*"
    )
    if data_file == []:
        print("Error! my_fluid.cycle_start data file not found.
        ")
        sys.exit()
    # end if
    data_file = data_file[0]
    the_data = cPickle.load(open(data_file))
    u  = the_data[1]; v   = the_data[2]
    p  = the_data[4]; src = the_data[5]; Bx = the_data[8]
    By = the_data[9]; A = the_datha[10]
    the_data = 0 # to save memory
# end if

##Solve!
for cycles in xrange(my_fluid.cycle_start , my_fluid.NT+1):
    u_old = u.copy()



    u[1:-1,1:-1]  = (my_fluid.
    nonlinear_advect_implicit_periodic_2d(u,u,v, XX,YY)
-   my_fluid.nonlinear_advect_implicit_periodic_2d(Bx,Bx,By,
                                                     XX,YY)
+   Bx[1:-1,1:-1]
-   my_fluid.apply_pressure_2dX(Bx**2+By**2, 0.5)
                       )
    v[1:-1,1:-1]  = (my_fluid.
    nonlinear_advect_implicit_periodic_2d(v,u_old,v, XX,YY)
-   my_fluid.nonlinear_advect_implicit_periodic_2d(By,Bx,By,
                                                     XX,YY)
+   By[1:-1,1:-1]
-   my_fluid.apply_pressure_2dY(Bx**2+By**2, 0.5)
                       )

    ##Update ghost zones
    u[ 0, : ] = u[ -2,  : ]; u[ -1,  : ] = u[ 1, : ]
    u[ :, 0 ] = u[  :, -2 ]; u[  :, -1 ] = u[ :, 1 ]

    v[ 0, : ] = v[ -2,  : ]; v[ -1,  : ] = v[ 1, : ]
```

```
v[ :, 0 ] = v[ :, -2 ]; v[ :, -1 ] = v[ :, 1 ]



src[1:-1,1:-1] = my_fluid.calc_source_2d(u,v)


p = my_fluid.transform_pressure_poisson_2d(p, src)


p[ 0, : ] = p[ -2, : ]; p[ -1, : ] = p[ 1, : ]
p[ :, 0 ] = p[ :, -2 ]; p[ :, -1 ] = p[ :, 1 ]

u[1:-1,1:-1] -= my_fluid.apply_pressure_2dX(p,
                                         1.0/my_fluid.RHO)
v[1:-1,1:-1] -= my_fluid.apply_pressure_2dY(p,
                                         1.0/my_fluid.RHO)

##Update ghost zones
u[ 0, : ] = u[ -2, : ]; u[ -1, : ] = u[ 1, : ]
u[ :, 0 ] = u[ :, -2 ]; u[ :, -1 ] = u[ :, 1 ]

v[ 0, : ] = v[ -2, : ]; v[ -1, : ] = v[ 1, : ]
v[ :, 0 ] = v[ :, -2 ]; v[ :, -1 ] = v[ :, 1 ]



A[1:-1,1:-1]  = my_fluid.
nonlinear_advect_implicit_periodic_2d(A,u,v, XX,YY)
A[ 0, : ] = A[ -2, : ]; A[ -1, : ] = A[ 1, : ]
A[ :, 0 ] = A[ :, -2 ]; A[ :, -1 ] = A[ :, 1 ]



Bx[1:-1,1:-1] =  ((A[1:-1, 2:  ] - A[1:-1,  :-2]) /
                    my_fluid.DY)
```

```
    By[1:-1,1:-1] = -((A[2:   , 1:-1] - A[ :-2, 1:-1]) /
                                  my_fluid.DX)


    Bx[ 0, : ] = Bx[ -2,  : ]; Bx[ -1,  : ] = Bx[ 1, : ]
    Bx[ :, 0 ] = Bx[  :, -2 ]; Bx[  :, -1 ] = Bx[ :, 1 ]


    By[ 0, : ] = By[ -2,  : ]; By[ -1,  : ] = By[ 1, : ]
    By[ :, 0 ] = By[  :, -2 ]; By[  :, -1 ] = By[ :, 1 ]



    if (cycles % my_fluid.SAVE_FREQ == 0):
        file_name = get_file_name(cycles, my_fluid.MAX_CYC_MAG,
                                              my_fluid.LABEL, ext)
        save_state([cycles, u,v, "NA", p,src, "NA","NA", Bx,By,
        A],file_name)
    # end if
# end for
file_name = get_file_name(cycles, my_fluid.MAX_CYC_MAG,
                                          my_fluid.LABEL, ext)
save_state([cycles, u,v, "NA", p,src, "NA","NA", Bx,By,A],
           file_name )

## end implicit_orszag_tang_athena_2d.py
```

References

Eckert, Michael (2006). *The Dawn of Fluid Dynamics: A Discipline between Science and Technology*. Weinheim: Wiley-VCH.

Barba, Loren A. (2013). *12 Steps to Navier-Stokes*. URL: `http://nbviewer.ipython.org/github/barbagroup/CFDPython/blob/master/lessons/05_Step_4.ipynb`.

Scannapieco, Evan and Francis H. Harlow (1995). *Introduction to Finite Difference Methods for Numerical Fluid Dynamics*. Los Alamos: Los Alamos National Laboratory.

Batchelor, G.K. (1967). *An Introduction to Fluid Dynamics*. Cambridge: Cambridge University Press.

Chaisson, Eric and Steve McMillan (2004). *Astronomy: A Beginner's Guide to the Universe*. Upper Saddle River: Pearson.

Faber, T.E. (1995). *Fluid Dynamics for Physicists*. Cambridge: Cambridge University Press.

Ferraro, V.C.A. and C. Plumpton (1961). *An Introduction to Magneto-fluid Mechanics*. London: Oxford University Press.

Farlow, Stanley J. (1993). *Partial Differential Equations for Scientists and Engineers*. New York: Dover.

Stam, J. (2003). "Real Time Fluid Dynamics for Games". In: *Proceedings of the Game Developer Conference* 18.10, pp. 891–921. DOI: `http://dx.doi.org/10.1002/andp.19053221004`.

Gonsalves, Richard J. (2004). *Chapter 6 Partial Differential Equations Lecture 2*. URL: `http://www.physics.buffalo.edu/phy410-505-2004/Chapter6/ch6-lec2.pdf`.

The Scipy Community (2014). *numpy.fft.fft*. URL: `http://docs.scipy.org/doc/numpy/reference/generated/numpy.fft.fft.html`.

Diogo Bolster, Robert E. Hershberger and Russell J. Donnelly (2011). “Dynamic similarity, the dimensionless science”. In: *Physics Today* 47.9, pp. 42–47. DOI: `http://dx.doi.org/10.1063/PT.3.1258`.

Eggers, Jens. *The method of images.* URL: `http://www.maths.bris.ac.uk/~majge/week9.pdf`.

Orszag, SA and CM Tang (1979). “Small-scale structure of two-dimensional magnetohydrodynamic turbulence”. In: *J. Fluid Mech.* 90.1, pp. 129–143. DOI: `http://dx.doi.org/10.1017/S002211207900210X`.

Stone et al. (2008). “Athena: A New Code for Astrophysical MHD”. In: *The Astrophysical Journal Supplement Series* 178.1, pp. 137–177. DOI: `http://dx.doi.org/10.1086/588755`.