1-D Simulation of the Alfvén Wave of Magnetohydrodynamics in a Cold Plasma

William H. Barndt

University of Alaska Fairbanks,
whbarndt@alaska.edu

April 27th, 2020

The Alfvén wave is a fundamental wave mode found in Magnetohydrodynamics (MHD) models of plasma. [1] It was first proposed by Hannes Alfvén in 1942 when he suggested the existence of what he called "electromagnetic-hydrodynamic waves" that propogate energy. In this project, in pursuit of gaining a further understanding of magnetohydrodynamics (MHD), plasma modeling, and partial differential equation solving, I created a simple 1-D simulation of the Alfvén wave in a cold plasma in Python using the numpy and matplotlib facilities.

1 Introduction

Plasma, the fourth state of matter, is a gas of composed of charged particles and free electrons. One of the models used to describe the behavior of plasma is the magnetohydrodynamics (MHD) model. In this model of plasma, the Alfvén wave is a fundamental wave mode found in the system. These waves correspond to the charged particles in the plasma oscillating in response to a magnetic tension restoring force caused by the magnetic field lines. They are dispersionless waves and travel parallel to the magnetic field lines and transverse to the direction of propogation. They were first proposed as "electromagnetic-hydrodynamic waves" by Hannes Alfvén in 1942 to explain why the photosphere (surface of the sun) was much colder than the solar corona, a few thousand Kelvin compared to about one million Kelvin. These waves are very important in the study of energy propogation in MHD systems. [2]

2 Description of Physics

The magnetohydrodynamics (MHD) model uses the Navier-Stokes equations coupled with Maxwell's Equations to describe the behavior of electrically conductive fluids. For this project, I simulated a cold plasma. This means that the the pressure in the system is zero and the energy continuity equation isn't neecessary in the system. [3] This was done to simplify the system. To further that goal, the flux conservative form for both the momentum and magnetic field continuity equations were used. [4] This didn't just allow me to keep track of less quantities but also allowed the numerical method implementation to be greatly streamlined. In addition, the equations were also normalized to the typical values of the system. [5] In this modified model, these are the necessary quantities for our system: Bulk plasma flow field $\mathbf{v}$, the mass density ρ, and the magnetic field $\mathbf{B}$.

With these quantities, the set of equations that relate them are:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0$$

$$\frac{\partial \rho \mathbf{u}}{\partial t} + \nabla \cdot [\rho \mathbf{u}\mathbf{u} + (p + B^2)\mathbf{I} - \mathbf{B}\mathbf{B}] = 0$$

$$\frac{\partial \mathbf{B}}{\partial t} + \nabla \cdot (\mathbf{u}\mathbf{B} - \mathbf{B}\mathbf{u}) = 0$$

The first equation representing the mass continuity equation, the second, the momentum continuity equation, and the third, the magnetic field continuity equation. [5]

An important thing to realize is that this simulation will be in one dimension, however, fluid dynamics and electrodynamics are intrinsically three dimensional phenomena. This means that we still must track each component of each field quantitiy. This grants a net total of seven quantities that will need to be accounted for in the system. To emulate a one-dimensional simulation, the derivatives in two of the dimensions should be set to zero. In this simulation I set the change in the x and y dimension to zero and only calculated the change in the z dimension.

To initialize the simulation of the Alfvén wave, we must start with a "pluck" of either the plasma flow flux or of the magnetic field flux. However, we need to initialize this perturbation perpendicular to the direction of propogation. If we only allow a change in the z-dimension the Alfvén wave will travel along the z-axis. We want to graph the wave propogation in 2D so plotting the perturbation of the plasma flux in either the x or y-dimension will show us the propogating Alfvén wave on a graph.

3 Numerical Methods

The Navier-Stokes equations and Maxwell's equations are classified as hyperbolic partial differential equations (HPDEs). Thus we need a numerical method for such a type of equation that will solve these for us. For this simulation, the finite difference method called the Lax-Wendroff method was chosen. [6] This is a two-step algorithm that is second-order accurate in both time and space: [7]

$$\mathbf{q}^*_{j+1/2} = \frac{1}{2}(\mathbf{q}^n_j + \mathbf{q}^n_{j+1}) - \frac{\Delta t}{2\,\Delta x}(\mathbf{F}^n_{j+1} - \mathbf{F}^n_j)$$

$$\mathbf{q}^{n+1}_j = \mathbf{q}^n_j - \frac{\Delta t}{\Delta x}(\mathbf{F}^*_{j+1/2} - \mathbf{F}^*_{j-1/2})$$

Because the system requires solving HPDEs, I have to solve both spacially and temporally. In order for the solution to converge, it must satisfy the Courant condition:

$$C = \frac{\nu_A \triangle t}{\triangle x} \leq C_{max},$$

where C is the Courant number, ν_A is the Alfvén velocity, and $C_{max} \leq 1$.

This condition states that you must have a time step small enough that you don't step over the next spacial grid point. If this condition isn't met, you lose information about the domain and the solution becomes unstable and diverges. [8] In order to avoid this, I first picked a spatial grid size then used the boundary conditions, that every variable in the system was normalized to one, to calculate the temporal grid size.

In the simulation, I used the numpy library in Python for the all of the math calculations. In particular, the most useful function used was the roll function. This was implemented in the Lax-Wendroff alogrithm to calculate all the changes in space by rolling the indices forward or backward depending on the type of step for each time step, thus not forcing a manual loop through each spatial index for each time step. This method also allowed the simulation to have periodic boundary conditions, so the Alfvén wave sould loop back around to the other side once an edge is hit.

4 Expected Results

As stated previously, the initial conditions of this simulation starts with a "pluck" of either the plasma's momentum or magnetic field. Although, since the simulation is only set up to allow changes in z, we must initialize this "pluck" perpendicular to the z-dimension in order to see the Alfvén wave propogate. I initialized this "pluck" as a Gaussian function in the x-dimension of the plasma momentum flux. You can also initialize a Gaussian in the x-dimension of the magnetic field flux and see the same graph if you plot the magnetic

field's x-dimension. (The y-dimension goes in analogy.) If the simulation exhibits this property, then the physics of the system was properly programmed.

Here is my graph of the initial setup of the x-dimension of the plasma's momentum flux:

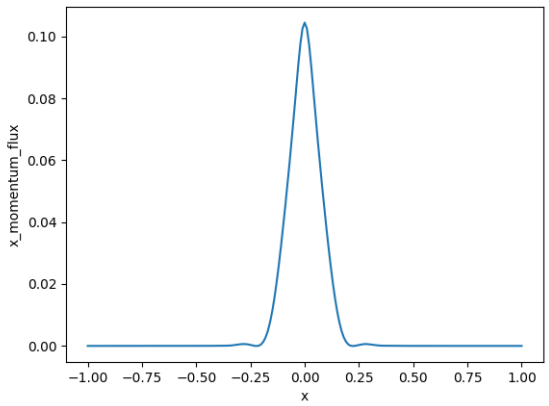

Figure 1: Initial Gaussian Perturbation

Once the simulation starts, the initial perturbation position should attempt to move towards equillibrium. This will cause the wave to be split into two, propogating outwards. Each wave's amplitude should be exactly half the initial perturbation amplitude and move at the same speed. More specifically, it should move at the Alfvén velocity, which equals one in this simulation since the I normalized the system and set every variable to one.

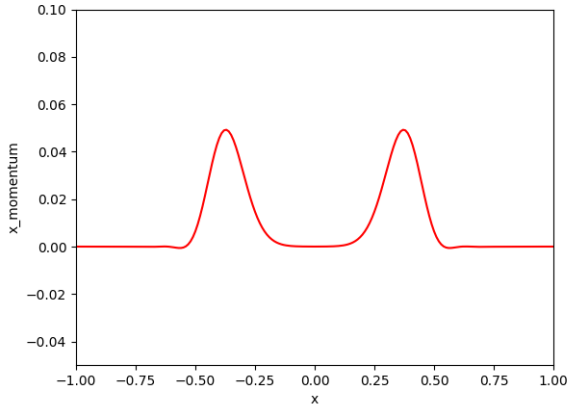

Figure 2: Simulation after roughly 200 time steps

This is the plot of the x-dimension magnetic field:

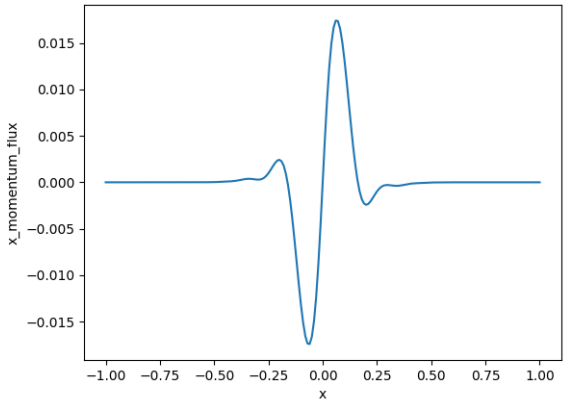

Figure 3: Initial magnetic field

This wave represents the change in the momentum flux, the derivative at all points of the momentum flux wave. Which is consistent with the relation of the flow to the magnetic field of the same dimension.

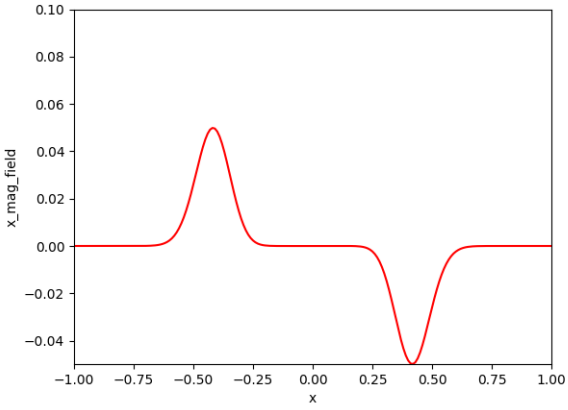

Figure 4: Simulation after roughly 200 time steps

5 Conclusion

In this project I learned much about MHD systems of plasmas and the Alfvén wave and what it represents in these systems. I've learned what it means to model an intrinsically three dimensional system in one dimension and how to interpret the results from it. I've also learned much more about numerically solving PDEs.

Moving forward, the next step would be to treat the multi-dimensional quantities as vectors and use numpy facilities to calculate the flux equations instead of hardcoding the equations for only the change in a certain dimension. This would allow me to condense some code but also provide more versitility in what I can simulate.

```
# Program ###################################################
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

class MHD_sim_var:
  def __init__(self, gauss_offset, stand_dev, gauss_amp):
      self.xt = np.arange(-1, xfinal+dx,dx) # 1-D Space (x)
      self.xt0 = gauss_offset               # Offset of the Initial Gaussian
      self.h = stand_dev                    # Standard Deviation of the Initial Gaussia
      self.A = gauss_amp                    # Amplitude of the Initial Gaussian
      # Animation Variables
      self.x_momentum = []
      self.y_momentum = []
      self.z_momentum = []
      self.x_mag_field = []
      self.y_mag_field = []
      self.z_mag_field = []
      self.tarr = []
      # Main MHD Variables
      self.d = np.ones(len(self.xt))        # Density
      self.dux = np.zeros(len(self.xt))     # Momentum
      self.duy = np.zeros(len(self.xt))
      self.duz = np.zeros(len(self.xt))
      self.ux = np.zeros(len(self.xt))      # Velocity
      self.uy = np.zeros(len(self.xt))
      self.uz = np.zeros(len(self.xt))
      self.Bx = np.zeros(len(self.xt))      # Magnetic Field
      self.By = np.zeros(len(self.xt))
      self.Bz = np.ones(len(self.xt))
      self.p = 0                            # Pressure
      # Flux Variables
      self.m_flx = np.zeros(len(self.xt))
      self.x_mom_flx = np.zeros(len(self.xt))
      self.y_mom_flx = np.zeros(len(self.xt))
      self.z_mom_flx = np.zeros(len(self.xt))
      self.x_mag_flx = np.zeros(len(self.xt))
      self.y_mag_flx = np.zeros(len(self.xt))
      self.z_mag_flx = np.zeros(len(self.xt))

# Initializes a Gaussian curve on the z-dimension of the Magnetic Field.
```

```
def initialize_simulation(mhd):
  mhd.dux = mhd.A*np.exp(-(mhd.xt-mhd.xt0)**2/mhd.h**2) # ???

# General Lax-Wendroff Discretizations (two-step)
def lax_wendroff_half(q, fl, dt, dx):
  qhalf = 0.5*(q + np.roll(q, -1)) - (0.5*dt/dx)*(np.roll(fl, -1) - fl)
  return qhalf

def lax_wendroff_full(qhalf, fl, dt, dx):
  qtemp = qhalf - (dt/dx)*(fl - np.roll(fl, 1))
  return qtemp

def calculate_flux_terms(bucket):
  bucket.m_flx = mass_flux(bucket)
  bucket.x_mom_flx = x_momentum_flux(bucket)
  bucket.y_mom_flx = y_momentum_flux(bucket)
  bucket.z_mom_flx = z_momentum_flux(bucket)
  bucket.x_mag_flx = x_mag_flux(bucket)
  bucket.y_mag_flx = y_mag_flux(bucket)
  bucket.z_mag_flx = z_mag_flux(bucket)

# Main Solver
def solver(mhd, tfinal, dt, dx):
  t = 0
  while t < tfinal:
    # Calculate Flux Terms
    calculate_flux_terms(mhd)

    # Container for the first half of Lax Wendroff
    mhd_half = MHD_sim_var(mhd.xt0, mhd.h, mhd.A)

    # Density
    mhd_half.d = lax_wendroff_half(mhd.d, mhd.m_flx, dt, dx)
    # Momentum
    mhd_half.dux = lax_wendroff_half(mhd.dux, mhd.x_mom_flx, dt, dx)
    mhd_half.duy = lax_wendroff_half(mhd.duy, mhd.y_mom_flx, dt, dx)
    mhd_half.duz = lax_wendroff_half(mhd.duz, mhd.z_mom_flx, dt, dx)
    # Divide to find velocity
    mhd_half.ux = np.divide(mhd_half.dux, mhd_half.d)
    mhd_half.uy = np.divide(mhd_half.duy, mhd_half.d)
    mhd_half.uz = np.divide(mhd_half.duz, mhd_half.d)
```

```
# Magnetic Field
mhd_half.Bx = lax_wendroff_half(mhd.Bx, mhd.x_mag_flx, dt, dx)
mhd_half.By = lax_wendroff_half(mhd.By, mhd.y_mag_flx, dt, dx)
mhd_half.Bz = lax_wendroff_half(mhd.Bz, mhd.z_mag_flx, dt, dx)

# Second half of Lax Wendroff
calculate_flux_terms(mhd_half)

# Density
dtemp = lax_wendroff_full(mhd.d, mhd_half.m_flx, dt, dx)
# Momentum
duxtemp = lax_wendroff_full(mhd.dux, mhd_half.x_mom_flx, dt, dx)
duytemp = lax_wendroff_full(mhd.duy, mhd_half.y_mom_flx, dt, dx)
duztemp = lax_wendroff_full(mhd.duz, mhd_half.z_mom_flx, dt, dx)
# Divide to find velocity
uxtemp = np.divide(duxtemp, dtemp)
uytemp = np.divide(duytemp, dtemp)
uztemp = np.divide(duztemp, dtemp)
# Magnetic Field
Bxtemp = lax_wendroff_full(mhd.Bx, mhd_half.x_mag_flx, dt, dx)
Bytemp = lax_wendroff_full(mhd.By, mhd_half.y_mag_flx, dt, dx)
Bztemp = lax_wendroff_full(mhd.Bz, mhd_half.z_mag_flx, dt, dx)

mhd.d = dtemp
mhd.dux = duxtemp
mhd.duy = duytemp
mhd.duz = duztemp
mhd.ux = uxtemp
mhd.uy = uytemp
mhd.uz = uztemp
mhd.Bx = Bxtemp
mhd.By = Bytemp
mhd.Bz = Bztemp

mhd.x_momentum.append(mhd.dux)
mhd.y_momentum.append(mhd.duy)
mhd.z_momentum.append(mhd.duz)
mhd.x_mag_field.append(mhd.Bx)
mhd.y_mag_field.append(mhd.By)
mhd.z_mag_field.append(mhd.Bz)
mhd.tarr.append(t)
```

```
    t += dt
  # Plot the wave
  #plt.figure(0)
  #plt.plot(mhdsim.xt, mhdsim.dux)
  #plt.figure(1)
  #plt.plot(mhdsim.xt, mhdsim.Bx)
  plt.show()

# Mass Continuity Equation
def mass_flux(mhd):
  return mhd.d*mhd.uz

# X-Momentum Continuity Equation
def x_momentum_flux(mhd):
  return mhd.d*mhd.ux*mhd.uz - mhd.Bz*mhd.Bx

# Y-Momentum Continuity Equation
def y_momentum_flux(mhd):
  return mhd.d*mhd.uz*mhd.uy - mhd.Bz*mhd.By

 # Z-Momentum Continuity Equation
def z_momentum_flux(mhd):
  return mhd.d*mhd.uz**2 + 0.5*(mhd.Bx**2 + mhd.By**2 + mhd.Bz**2) - mhd.Bz**2

# X-Magnetic Flux Continuity Equation
def x_mag_flux(mhd):
  return mhd.uz*mhd.Bx - mhd.Bz*mhd.ux

# Y-Magnetic Flux Continuity Equation
def y_mag_flux(mhd):
  return mhd.uz*mhd.By - mhd.Bz*mhd.uy

# Z-Magnetic Flux Continuity Equation
def z_mag_flux(mhd):
  return mhd.uz*mhd.Bz - mhd.Bz*mhd.uz

# Animate the simulation (Taken from Lab 9)
def animate(mhd, label, dim, nstep):
  fig, ax = plt.subplots()
  line, = ax.plot([],[],'r')
```

```
  ax.set_ylim(-mhd.A/2, mhd.A)
  ax.set_xlim(mhd.xt.min(), mhd.xt.max())
  ax.set_xlabel('x')
  ax.set_ylabel(label)
  def update_line(i):
    line.set_ydata(data[i])
    line.set_xdata(mhd.xt)
    return line,
  data = []
  for i in range(0, len(mhd.tarr), nstep):
    data.append(dim[i])
  nfrm = int(len(data))
  ani = animation.FuncAnimation(fig, update_line, frames=nfrm, interval=1, blit=True, r
  plt.show()
  return ani

# ------------------------------------------------------------
# Declaration of Program Parameters:
xfinal = 1
dx = 0.01
c = 0.1       # Courant Number
va = 1        # Alfven Speed
dt = c*dx/va
tfinal = 2000*dt
# Gaussian variables
h = 10.0*dx
gauss_amp = 0.1
xt0 = 0
# Animation variable
nstep = 5

# Main Program
mhdsim = MHD_sim_var(xt0, h, gauss_amp)
initialize_simulation(mhdsim)
solver(mhdsim, tfinal, dt, dx)
xm_ani = animate(mhdsim, 'x-momentum', mhdsim.x_momentum, nstep)
xm_ani
ym_ani = animate(mhdsim, 'y-momentum', mhdsim.y_momentum, nstep)
ym_ani
zm_ani = animate(mhdsim, 'z-momentum', mhdsim.z_momentum, nstep)
zm_ani
```

```
xb_ani = animate(mhdsim, 'x-mag_field', mhdsim.x_mag_field, nstep)
xb_ani
yb_ani = animate(mhdsim, 'y-mag_field', mhdsim.y_mag_field, nstep)
yb_ani
zb_ani = animate(mhdsim, 'z-mag_field', mhdsim.z_mag_field, nstep)
zb_ani
```

References

[1] Magnetohydrodynamics, Feb 2021.

[2] Alfvén wave, Feb 2021.

[3] Hunter Barndt. Discussions with dr. peter delamere.

[4] Peter Delamere. Hw4 solutions.

[5] Peter Delamere. Mhd simulation.

[6] Peter Delamere. Systems of equations.

[7] Lax-wendroff method, Feb 2021.

[8] Courant–friedrichs–lewy condition, Feb 2021.