भारतीय प्रौद्योगिकी संस्थान हैदराबाद
Indian Institute of Technology Hyderabad

Particle-in-cell simulations of Kinetic Alfven Waves

B-Tech Project Report

Srinikitha Bhagvati
EP18BTECH11003
Supervisor: Dr. Kiritkumar Makwana

28 April, 2021

Contents

Abstract

Kinetic Alfven Waves, the intermediate wave mode solution to the plasma wave dispersion relation, is a transverse wave with strong magnetic field variations perpendicular to the wave motion. Theoretically derived expressions for perturbations in the wave are used to initialize a simulation for the same. The simulation is done using iPIC3D, which is a software that uses implicit Particle-in-Cell methods and kinetic theory equations to run. The simulated Alfven frequency is compared to the theoretically derived frequency to verify the procedure.

1 Introduction

The kinetic Alfven wave (KAW) is the subject of immense interest and ongoing research due to its transverse nature with strong magnetic field variations perpendicular to the wave motion. Alfven wave turbulences occur commonly in space plasmas, especially while studying the solar wind, magnetosheath and auroral regions, and their nature and physics is still a subject of ongoing research. The motivation for this project is to simulate a clean Kinetic Alfven wave to pave the way for simulations of bigger, multiple-wave turbulent plasmas, that can aid in the study of understanding these systems.

The aim of this project is to simulate a pure, clean Kinetic Alfven wave using implicit particle-in-cell methods. In this report, Section 2 provides a summary of the analysis of the Kinetic Alfven wave using the Two-Fluid Model. Section 3 contains a brief introduction to iPIC3D, the software we will use to simulate our wave. Section 4 delves into the implementation of certain initial conditions in iPIC3D to simulate the required wave. Finally, in Section 5, we plot the simulated values of the magnetic field component to verify the Alfven wave frequency.

2 Summary: Expressions for KAW using the Two-Fluid Model

In this section, we will give a brief overview of the Kinetic Alfven Wave described using a quasi-neutral Two-Fluid plasma model with electron and ion species.

To fully describe the behaviour of a plasma wave, we need to specify its perturbations in electric field $\delta\vec{E}$, magnetic field $\delta\vec{B}$, density δn, pressure P, and velocities $\delta\vec{V_{xs}}$, $\delta\vec{V_{ys}}$ and $\delta\vec{V_{zs}}$ for species s. We use fluid theory only if we assume that the plasma is sufficiently collisional.

The perturbations are of the form $e^{\vec{k}.\vec{r}-\omega t}$ and we denote them by δ. The directions taken are as shown in Fig 1. We begin with the linearized Navier-Stokes momentum

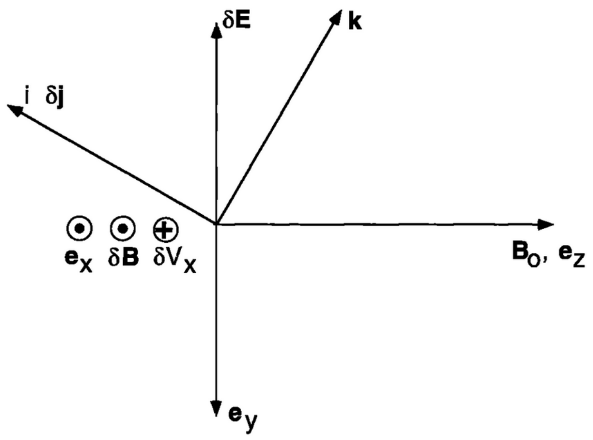

Figure 1: The directions of B_0 and k assumed without loss of generality

equation:

$$\delta \vec{V_\perp} - \frac{\omega^2}{\omega_c^2}\delta\vec{V} = (\frac{\delta\vec{E'} \times \vec{B_0}}{B_0^2})c - \frac{q}{m}\frac{i\omega}{\omega_c^2}\delta\vec{E'}$$

where $\delta\vec{E'} = \delta\vec{E} - \frac{\nabla\delta p}{qn_0}$, $V_0 = 0$ and $\omega_c = \frac{qB_0}{mc}$. This gives us expressions for the velocity perturbations in terms of density and electric field perturbations [1].
Taking the pressure perturbations as $\delta p = \gamma\kappa T_0 \delta n$, we can derive an expression for the density perturbation using the Continuity Equation:

$$\frac{\partial n}{\partial t} + n_0 \nabla.\delta\vec{V} = 0$$

Now, taking into consideration the quasi-neutrality condition ($\frac{\delta n_i}{n_{0i}} = \frac{\delta n_e}{n_{0e}}$), neglecting the term ω^2/ω_{ce}^2 and applying above assumptions, we get the following equation:

$$\delta E_y = i\frac{\omega}{\omega_c}\delta E_x + \frac{k_z\omega_c^2}{k_y\omega^2}\frac{(\omega_c^2(G_t k_z^2 - m_i\omega^2) - G_t k_y^2\omega^2)}{(G_e k_z^2 - m_e\omega^2)}\delta E_z \qquad (1)$$

where $G_t = G_i + G_e$, and $G = \gamma\kappa T$ [1].

Now, we know that Maxwell's Equations are:

Faraday's Law of Induction:	$\frac{\partial\vec{B}}{\partial t} = -c(\nabla \times \vec{E})$
Ampere's Law:	$(\nabla \times \vec{B}) = 4\pi\vec{j} + \frac{1}{c}\frac{\partial\vec{E}}{\partial t}$
Divergence Equation:	$\nabla.\vec{B} = 0$
Poisson's Equation:	$\nabla.\vec{E} = 4\pi q$

From these equations and Eq(1), we can derive expressions for the electric field $\delta\vec{E}$ and magnetic field $\delta\vec{B}$ perturbations in terms of $\delta\vec{E_x}$.

From the electric field equations and Eq(1), we derive the wave dispersion relation for ω in terms of wave-vector $\vec{k}$ to be as follows:

$$(\frac{\omega^2}{k_z^2 v_a^2} - 1)(\omega^2(\omega^2 - k^2v_a^2) - \beta k^2 v_a^2(\omega^2 - k_z^2 v_a^2)] = \omega^2(\omega^2 - k^2v_a^2)k_y^2[L^2 - \frac{\omega^2 L_e^2}{k_z^2 v_a^2}] \qquad (2)$$

where:

$$\begin{aligned}
v_a^2 &= \frac{B_0^2}{4\pi m_i n_0} \\
G_t &= \gamma_i \kappa_i T_{0i} + \gamma_e \kappa_e T_0 e \\
\beta &= \frac{G_t}{m_i v_a^2} \\
\omega_{pe}^2 &= 4\pi q_e^2 n_{0_e}/m_e \\
L &= \frac{1}{|\omega_{c_i}|}\sqrt{\frac{\gamma_i \kappa_i T_{0i} + \gamma_e \kappa_e T_0 e}{m_i}} = \frac{v_s}{|\omega_{c_i}|} \\
L_e &= \frac{c}{\omega_{pe}} = \frac{L}{\sqrt{\frac{m_i}{m_e}\beta}}
\end{aligned}$$

This is a cubic equation in ω^2 whose roots give the three wave modes of a plasma wave: the fast mode, the intermediate mode (Alfven), and the slow mode [1]. After rearranging the terms, the cubic equation can be written as:

$$\Omega^6 \frac{k_z^2}{k^2}(1 + k_y^2 L_e^2) - \Omega^4 [\frac{k_z^2}{k^2}(1 + k_y^2 L^2) + (1 + \beta + k_y^2 L_e^2)] + \Omega^2(1 + 2\beta + k_y^2 L^2) - \beta = 0 \quad (3)$$

where $\Omega = \frac{\omega}{k_z v_a}$

3 Introduction to iPIC3D

iPIC3D [2] is a software package written in C++ that employs implicit Particle-in-cell method to simulate a kinetic theory model (as opposed to fluid theory) of plasma. The code uses a uniform Cartesian grid to evaluate the electric fields $\vec{E}$, current densities $\vec{j}$ at the cell corners, the magnetic fields $\vec{B}$, the charge densities ρ at the cell-centers. Each cell further contains 64 'sub-cells' each, where each sub-cell represents one particle in the plasma. The velocity of each particle is given by $\vec{v} = v_{th} + u_0$ where v_{th} is the randomized thermal velocity of a particle about mean thermal velocity following a Maxwellian distribution and u_0 is the drift velocity.

To calculate the evolution of fields in a collision-less plasma, the code uses the Vlasov Equation and Maxwell's Equations [2]. For a distribution function f_s for a species s, defined as probability of finding a particle in $(d\vec{x}, d\vec{v})$ around a certain phase space point $(\vec{x}, \vec{y})$, the Vlasov equation gives

$$\frac{\partial f_s}{\partial t} + \vec{v}.\frac{\partial f_s}{\partial \vec{x}} + \frac{q_s}{m_s}(\vec{E} + \vec{v} \times \vec{B}).\frac{\partial f_s}{\partial \vec{v}} = 0$$

where q_s is the charge, m_s is the mass and $\vec{E}$, $\vec{B}$ are the electric and magnetic fields. We know that the Maxwell's Equations are

Faraday's Law of Induction:	$(\nabla \times \vec{E}) = -\frac{\partial \vec{B}}{\partial t}$
Ampere's Law:	$(\nabla \times \vec{B}) = \mu_0 j + \mu_0 \epsilon_0 \frac{\partial \vec{E}}{\partial t}$
Divergence Equation:	$\nabla.\vec{B} = 0$
Poisson's Equation:	$\epsilon_0 \nabla.\vec{E} = \rho$

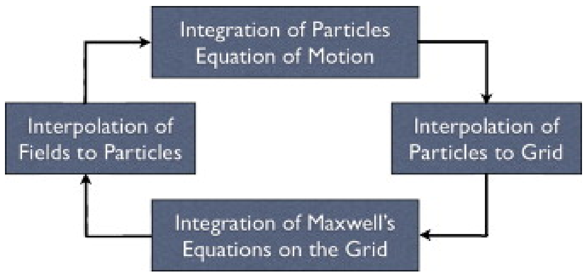
Integration of Particles Equation of Motion
Interpolation of Particles to Grid
Integration of Maxwell's Equations on the Grid
Interpolation of Fields to Particles

Figure 2: The iPIC3D computational cycle at each simulation time-step

Using charge density ρ and current density $\vec{j}$, we can establish a link between the Vlasov Equation and the Maxwell's Equations.

We start off by initializing the electric $\vec{E}$ and magnetic $\vec{B}$ fields on the grid. The code interpolates these fields to the position of each particle, and calculates the effect on those particles from the Lorentz force law $\frac{d\vec{v}}{dt} = q(\vec{E} + \frac{\vec{v}\times\vec{B}}{c})$. This calculated velocity causes the particles to move in the grid. We calculate the charge density ρ and current density $\vec{j}$ in each cell at the next time level, based on the newly 'moved' particle positions at that instant, and subsequently calculate new values for $\vec{E}$ and $\vec{B}$ from Maxwell's Equations. Thus, the code repeats this process over and over again until the limit we set on the number of timesteps is reached [2]. This process is summarized in the flow chart in Figure 2.

The output is given in the form of files containing the values of the plasma parameters at each cell in the grid. These files can further be plotted using python codes to visualize the wave.

4 iPIC3D-KAW

To simulate the Kinetic Alfven Wave using iPIC3D, we need to initialize the code with the perturbation fields, densities and velocities of a kinetic Alfven wave mentioned in Section 2. We plug the input parameters into the normalized equations given below to calculate these perturbations, thus initializing the required wave. We will discuss this in detail in the three following subsections.

4.1 Units

Before we proceed further, let us look at the system of units used in the code.
Length is normalized in units of ion-skin depth d_i, velocities are normalized to c and time is normalized in units of inverse of the plasma ion frequency ω_{pi}^{-1}, where $\omega_{pi} = c/d_i$. Charge of an electron e, mass of the ion m_i and magnetic permeability of vaccuum μ_0

are all set to unity, i.e.,

$$[e] == 1; m_i == 1; \mu_0 == 1$$

Since ion mass is set to unity, the electron mass ratio is a user defined quantity. The mass density ρ is $1/4\pi$ where ρ is normalized to some background density set to unity. The electron and ion pressures are thus defined as:

$$\begin{aligned} p_{gas} &= p_{ion} + p_{electron} \\ p_{ion} &= u_{th,i}^2 * \rho_i/(4 * pi) \\ p_{electron} &= (m_e/m_i) * u_{th,e}^2 * \rho_e/(4 * pi) \end{aligned}$$

Finally, the Alfven speed v_a equals the magnetic field B_0.

4.2 Initial Values for Fields, Densities and Velocities

As mentioned earlier, we need to specify the initial state of the plasma while running a simulation in iPIC3D. Now, we write down the normalized expressions for perturbations in electric and magnetic fields, velocities and density using the code units in Section 4.1. The constants used in these expressions are $\beta = p_{gas}/p_{mag}$ (where p_{gas} and p_{mag} are ion + electron pressure and magnetic pressure respectively), mr (electron mass ratio), v_a (Alfven speed) and γ (ratio of specific heats in fluid theory).

4.2.1 Density Perturbations

$$\frac{\delta n}{n_0} = \frac{i\delta E_z mr\omega_{c_i}}{v_a \frac{k_z L}{\sqrt{\beta}}(\gamma \frac{v_{th,e}^2}{3} - \omega^2 \frac{\sqrt{\beta}}{(k_z L)^2})} \tag{4}$$

4.2.2 Electric Field Perturbations

$$\frac{\delta E_z}{\delta E_x} = i \frac{k_z L}{k_y L} \frac{(\omega^2 - \frac{kL^2}{\beta^2} v_a^2)}{\omega\omega_{c_i}} \frac{(\gamma \frac{v_{th,e}^2}{3} - \omega^2 \frac{\beta^2}{(k_z L)^2})}{\gamma(mr \frac{v_{th,i}^2}{3} + \frac{v_{th,e}^2}{3})} \tag{5}$$

$$\frac{\delta E_y}{\delta E_x} = i \frac{(\omega^2 - \frac{(kL)^2}{\beta^2} v_a^2)(\omega^2 \frac{\beta^2}{(k_z L)^2}(1 - mr\gamma \frac{v_{th,i}^2}{3v_a^2}) - \gamma \frac{v_{th,e}^2}{3})}{\gamma[mr \frac{v_{th,i}^2}{3} + \frac{v_{th,e}^2}{3}][\frac{\omega^2}{v_a^2} \frac{\beta^2}{(k_z L)^2} - 1]]\omega\omega_{c_i}} \tag{6}$$

4.2.3 Magnetic Field Perturbations

$$\delta B_x = \frac{1}{\omega\sqrt{(\beta)}}((k_y L)\delta E_z - (k_z L)\delta E_y) \tag{7}$$

$$\delta B_y = \frac{1}{\omega\sqrt{(\beta)}}((k_z L)\delta E_x) \tag{8}$$

$$\delta B_z = \frac{-1}{\omega\sqrt{(\beta)}}(k_y L\delta E_x) \tag{9}$$

4.2.4 Velocity Perturbations (for ion and electron species)

$$\delta V_{x_i} = \frac{1}{v_a}(\delta E_y - i\frac{\omega}{\omega_{c_i}}\delta E_x) - i\frac{\gamma v_{th,i}^2(k_y L)}{3\omega_{c_i}\sqrt{\beta}}\frac{\delta n}{n_0} \tag{10}$$

$$\delta V_{x_e} = \frac{1}{v_a}(\delta E_y - i\frac{\omega}{\omega_{c_e}}\delta E_x) - i\frac{\gamma v_{th,e}^2(k_y L)}{3\omega_{c_e}\sqrt{(\beta)}}\frac{\delta n}{n_0} \tag{11}$$

$$\delta V_{y_i} = \frac{-1}{v_a}(\delta E_x + i\frac{\omega}{\omega_{c_i}}\delta E_y) - \frac{\gamma v_{th,i}^2(k_y L)}{3\omega_{c_i}\sqrt{(\beta)}}\frac{\omega}{\omega_{c_i}}\frac{\delta n}{n_0} \tag{12}$$

$$\delta V_{y_e} = \frac{-1}{v_a}(\delta E_x + i\frac{\omega}{\omega_{c_e}}\delta E_y) - \frac{\gamma v_{th,e}^2(k_y L)}{3\omega_{c_e}\sqrt{\beta}}\frac{\omega}{\omega_{c_e}}\frac{\delta n}{n_0} \tag{13}$$

$$\delta V_{z_i} = i\frac{\omega_{c_i}}{\omega v_a}\delta E_z + \frac{\gamma}{3}\frac{v_{th,i}^2}{\omega}\frac{(k_z L)}{\sqrt{(\beta)}}\frac{\delta n}{n_0} \tag{14}$$

$$\delta V_{z_e} = i\frac{\omega_{c_e}}{\omega v_a}\delta E_z + \frac{\gamma}{3}\frac{v_{th,e}^2}{\omega}\frac{(k_z L)}{\sqrt{(\beta)}}\frac{\delta n}{n_0} \tag{15}$$

The simulation will be initialised with the values of these perturbed quantities as calculated from these expressions.

4.3 Input Parameters

To implement the above equations, we used three different functions:

- calc-omegas[A.2]: To calculate the Alfven root from the dispersion relation. This is a simple cubic root solver that substitutes the values of wave-vector $\vec{k}$, β, ion electron mass ratio, γ and Alfven speed v_a in Eq 3, and returns its middle root, i.e., the Alfven frequency.

- put-dfields (in fields folder)[A.3]: To calculate the electric field perturbations $\delta\vec{E}$ (Eqs. 5, 6), magnetic field perturbations $\delta\vec{B}$ (Eqs. 7, 8, 9) and density perturbations δn (Eq. 4) using the output from calc-omegas.

- put-dfields (in particles folder)$^{A.4}$: To calculate the velocity perturbations $\delta\vec{V_s}$ for species s (Eqs. 10, 11, 12, 13, 14, 15) using the other quantities derived in the previous two functions.

As discussed in Section 3, each cell in the grid contains 64 particles. We initialize each of these particles with a velocity $\vec{v} = v_{th} + v_{drift}$, where v_{th} is the thermal velocity and $v_{drift} = V_0 + \delta V$ is the drift velocity calculated by the third function given above. Thus, the simulation is ready to run.

In the simulation we ran, the grid had 128×128 cells in the y- and z-axes, with $k_x = 0$ and $k_y = k_z = 1$ and electric field $\delta\vec{E} = 0.00002$. For other input parameters, see Appendix A.1.

5 Analysis and Results

After initializing the code with the necessary conditions, the code outputs the simulated plasma parameters of that instant after running for some user-specified number of time steps. These output values can further be plotted using python codes to visualize the wave. But this raises the question, how are we sure that output is the desired Kinetic Alfven wave? After all, we are not changing any of the codes used internally by iPIC3D to simulate our plasma waves after adding the initial conditions. To verify this, we calculate the frequency of the field perturbations at a future simulated time step.

At any future time-step, the perturbation for a kinetic Alfven wave is expected to be of the form $e^{\vec{k}.\vec{r}-\omega_a t}$, where ω_a corresponds to the Alfven root of the wave dispersion relation. Therefore, to verify that we are indeed simulating a kinetic Alfven wave, the value of ω_a must be equal to the intermediate solution of Eq. 3.

The frequency of the kinetic Alfven wave as calculated by the code from the dispersion relation is $\omega = 0.00705$.

Figure 3 shows the plot of the B_x field at $t = 0$ timesteps, $t = 100$ timesteps and $t = 500$ timesteps. By measuring the minimum at each column, and averaging the shift in the minimum, we find that the omega calculated is:
1. For 0 to 100 time steps, $\omega_a = 0.00784$
2. For 0 to 500 time steps, $\omega_a = 0.00864$

This is similar to the value obtained theoretically, even if it is not very good. The method we used to measure the frequency from the plots can be bettered. This is only a rough draft to check if the values we obtain are close to the theoretical values. In addition to this, we notice that as the number of time-steps is increased, the plots get more grainy, making it difficult to determine the crests and the troughs of the wave accurately. Having a higher resolution while simulating the wave can produce more accurate results.

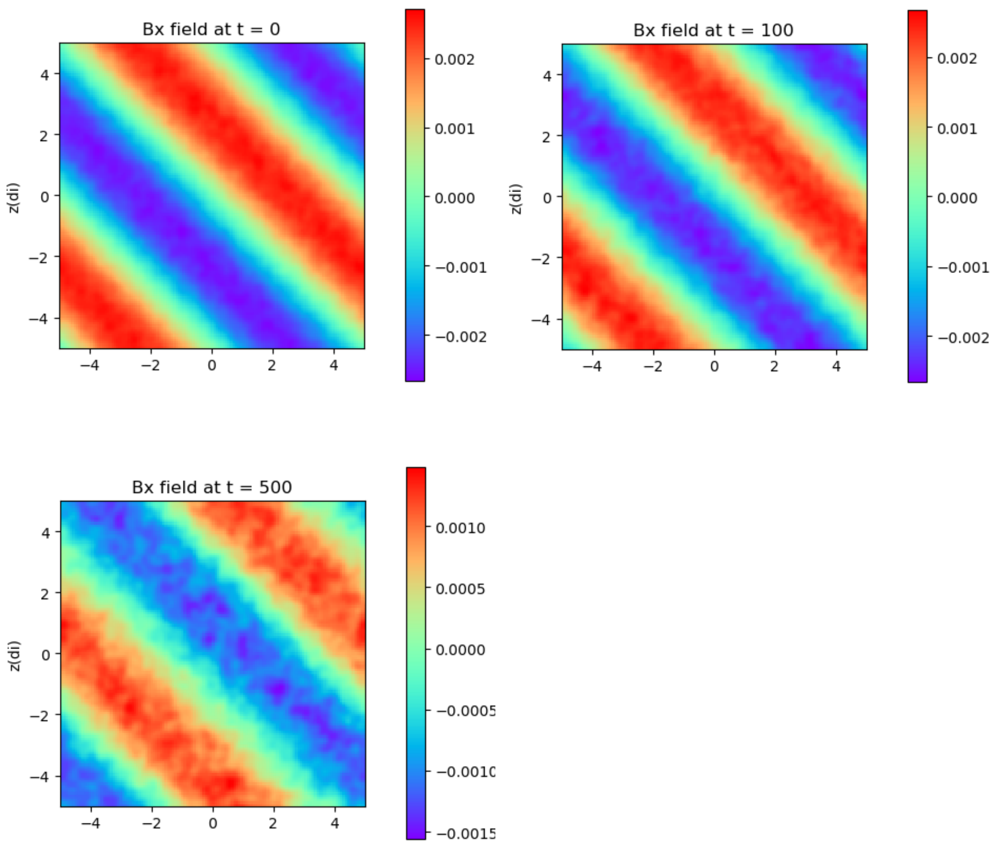

Figure 3: The plot the B_x field simulated at 0, 100 and 500 time-steps. The axes represent the y- and z- axes in the code. As $k_y = k_z = 1$, the plots are symmetric about the $y = z$ axis.

6 Conclusion

In this project, we simulated the Kinetic Alfven Wave we modelled last semester using the Two-Fluid model. We achieved this by using an implicit particle-in-cell simulator, iPIC3D, that works on the principles of kinetic theory of plasma. We initialised this simulator with the previously derived expressions for perturbations in the electric fields, the magnetic fields, the densities and the velocities. We plotted the results of the x-component of the magnetic field perturbation, and compared the frequency of the simulated wave to the theoretically predicted Alfven frequency. While the two values are still comparable, future simulations at higher resolutions and better methods to calculate the frequency of the simulated wave may improve the result further.

This project seeks to be the stepping stone for future endeavours to simulate such waves and environments. Kinetic Alfven waves are the subject of a lot of ongoing research, and, with the eventual simulation of a clean and accurate Alfven wave, we hope that this leads to insightful progress in the study of kinetic turbulences in plasma.

References

[1] J. V. Hollweg, "Kinetic alfven wave revisited," *JOURNAL OF GEOPHYSICAL RESEARCH*, vol. 104, no. A7, pp. 14,811–14,819, 1999.

[2] R.-u. Stefano Markidis, Giovanni Lapenta, "Multi-scale simulations of plasma with ipic3d," *Elsevier B.V*, 2009.

A Appendix

A.1 Input Values

A.1.1 Magnetic Reconnection

$B_{0x} = 0.00$
$B_{0y} = 0.00$
$B_{0z} = 0.01$

A.1.2 External magnetic field parameters

$B_{1x} = 0.000$
$B_{1y} = 0.000$
$B_{1z} = 0.000$

A.1.3 Time

$c = 1.0$	c = speed of light
$dt = 0.5$	dt = time step
$n_{cycles} = 501$	cycles

A.1.4 Box Size

$L_x = 1.0$	L_x = simulation box length - x direction
$L_y = 10.0$	L_y = simulation box length - y direction
$L_z = 10.0$	L_z = simulation box length - z direction

$x_{center} = 1.$	L_x = simulation box length-x direction in m
$y_{center} = 1.$	L_y = simulation box length-y direction in m
$z_{center} = 1.$	L_z = simulation box length-z direction in m
$L_{square} = .1$	
$nxc = 1$	nxc = number of cells - x direction
$nyc = 128$	nyc = number of cells - y direction
$nzc = 128$	nzc = number of cells - z direction
$rho_{INIT} = 1.01.0$	Initial density (make sure you are neutral)
$ns = 2$	Number of particles
$npcelx = 4 \quad 4$	Particles per cell in X
$npcely = 4 \quad 4$	Particles per cell in Y
$npcelz = 4 \quad 4$	Particles per cell in Z
$qom = -25.0 \quad 1.0$	Charge/mass ratio
$uth = 0.0289 \quad 0.0061$	Thermal velocity in X
$vth = 0.0289 \quad 0.0061$	Thermal velocity in Y
$wth = 0.0289 \quad 0.0061$	Thermal velocity in Z
$u0 = 0.0 \quad 0.0$	Drift velocity in X
$v0 = 0.0 \quad 0.0$	Drift velocity in Y
$w0 = 0.0 \quad 0.0$	Drift velocity in Z

A.2 Code: calc-omegas

```
void EMfields3D::calc_omegas(double * omegas, double vtherm[2], double vaa,

    int nks;
    int nx,ny,nz;
    double mr, Ex_amp;
    int index = 1;
    double kxL, kyL, kzL, beta,  L_var, alfroot , rootsf[3][2];
    const double PI = 4.0 * atan( 1.0 );

    fstream my_file;
    my_file.open("wave_input.txt", ios::in);
    if (my_file) {
        my_file >> nks;
    } else {
        cout << "File not found " << endl;
        exit(0);
    }
    mr = fabs(qom[0])/fabs(qom[1]);

    for (int i = 0; i < nks; i++) {
        my_file >> nx >> ny >> nz >> Ex_amp;

        //beta = (gamma/3.0)*(1/vaa*vaa)*((vtherm[1]*vtherm[1]) + ((vtherm
        beta = (gamma/3.0)*(1/(vaa*vaa))*((vtherm[1]*vtherm[1]) + ((vtherm
        L_var = sqrt(gamma)*(1/vaa)*sqrt(mr/(1 + mr))*sqrt(((vtherm[1]*vthe

        kxL = nx*(2*PI/Lx)*L_var;
        kyL = ny*(2*PI/Ly)*L_var;
        kzL = nz*(2*PI/Lz)*L_var;

        double kLsum = sqrt(kyL*kyL + kzL*kzL);

        //Finding coefficients of the cubic equation
double  coeff1 = ((kzL*kzL)/(kLsum*kLsum))*(1 + (kyL*kyL)/(mr*beta));
        double coeff2 = -(((kzL*kzL)/(kLsum*kLsum))*(1 + kyL*kyL) + (1 + be
        double coeff3 = 1 + 2*beta + kyL*kyL;
        double coeff4 =- beta;


        if (coeff1 ==0) {
                cout << "Inside calc_omegas: The equation can be reduced to
                index = 0;
        }

        if (coeff4 == 0){
                cout << "Inside calc_omegas: The equation can be reduced to
                index = 0;
        }

        //Solving above cubic equation
        if (index != 0){

        double dum1 = ( coeff2 * coeff2 - 3.0 * coeff1 * coeff3 ) / ( 9.0 *
        double dum2 = ( 9.0 * coeff1 * coeff2 * coeff3 - 27.0 * coeff1 * co
        double dum3 = coeff2 / ( 3.0 * coeff1 );

        double discriminant = dum1 * dum1 * dum1 - dum2 * dum2 ;

        if ( discriminant > 0 )
        {
                double theta = acos( dum2 / ( dum1 * sqrt( dum1 ) ) );
                double dum4 = 2.0 * sqrt( dum1 );
                for ( int n = 0; n < 3; n++ ) {
                        rootsf[n][0] = dum4 * cos( ( theta + 2.0 * n * PI )
                }
                index = 1;

        }

        else
        {
                double gamma1 = cbrt( dum2+ sqrt( -discriminant ) );
                double gamma2 = cbrt( dum2- sqrt( -discriminant ) );

                rootsf[0][0] = gamma1 + gamma2 - dum3;
                rootsf[0][1] = 0.0;

                double re = -0.5 * ( gamma1 + gamma2 ) - dum3;
                double imag = ( gamma1 - gamma2 ) * sqrt( 3.0 ) / 2.0;
                if ( discriminant == 0.0 )                    // Equal roots (h
                {

                        rootsf[1][0] = re; rootsf[1][1] = 0.0;

                        rootsf[2][0] = re; rootsf[2][1] = 0.0;
                        index = 1;

                }

                else
                {

                        rootsf[1][0] = re; rootsf[1][1] = imag;
                        rootsf[2][0] = re; rootsf[2][1] = -imag;
                        cout << "Two roots of Omega squared are imaginary.
                index = 0;
                }
        }
       for(int n=0; n<3; n++){
                if (rootsf[n][0] < 0){
                        index = 0;
                        break;
                }
        }
        //Finding Alfven root from the solutions of the cubic equation
        if (index != 0){
                double temp[3];
                temp[0] = sqrt(rootsf[0][0])*(vaa*(kzL/L_var));
                temp[1] = sqrt(rootsf[1][0])*(vaa*(kzL/L_var));
                temp[2] = sqrt(rootsf[2][0])*(vaa*(kzL/L_var));
                sort(temp, temp + 3);
                alfroot = temp[1];

                cout << "coeffs= " << coeff1 << " " <<coeff2<<" "<<coeff3<<
                cout <<" vaa vtherm10" << vaa <<" "<< vtherm[1]<<" "<<vther
                cout <<"beta= "<<beta<<" "<<endl;
                cout << "Capital Omega from fields= "<< temp[1]/(vaa*(kzL/L

        }
    }

        if (index == 0) cout << "Omega cannot be found because index is 0"
        omegas[i]= alfroot;
    }

    my_file.close();

}
```

A.3 Code: put-dfields (in the fields folder)

```
void EMfields3D::put_dfields(double arr[8], double xr, double yr, double zr

    int nks, nx, ny, nz;
    double kx_d, ky_d, kz_d;
    double Ex_amp[2], Ey_amp[2], Ez_amp[2], Bx_amp[2], By_amp[2], Bz_amp[2]
    const double PI = 4.0 * atan( 1.0 );
    fstream my_file;
    my_file.open("wave_input.txt", ios::in);
    if (my_file) {
        my_file >> nks;
    } else {
        cout << "File not found " << endl;
        exit(0);
    }
    cout << "In dfields: omegas is " << omegas[0] << " " << omegas[1] << en
    //arr[8] = {0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0};

    for (int i =0; i < 8; i++){
        arr[i] = 0.0;
```

```
}

for (int i = 0; i < nks; i++) {
    my_file >> nx >> ny >> nz >> Ex_amp[0];
    kx_d = nx*2*PI/Lx;
    ky_d = ny*2*PI/Ly;
    kz_d = nz*2*PI/Lz;
    Ex_amp[1] = 0.0;

    double mr = fabs(qom[0])/fabs(qom[1]);

    double beta = (gamma/3.0)*(1/(vaa*vaa))*((vtherm[1]*vtherm[1]) + ((
    double L_var = sqrt(gamma)*(1/vaa)*sqrt(mr/(1 + mr))*sqrt(((vtherm[

    double kxL = nx*(2*PI/Lx)*L_var;
    double kyL = ny*(2*PI/Ly)*L_var;
    double kzL = nz*(2*PI/Lz)*L_var;

    double kLsum = sqrt(kyL*kyL + kzL*kzL);
    double omega_ci = vaa*sqrt(1 + (1/mr));

    Ez_amp[0] = 0.0;
    Ez_amp[1] = Ex_amp[0]*(kzL/kyL)*(omegas[i]*omegas[i] - (kLsum*kLsum

    Ey_amp[0] = 0.0;
    Ey_amp[1] = Ex_amp[0]*(omegas[i]*omegas[i] - (kLsum*kLsum/(beta))*v

    Bx_amp[0] = (1/(omegas[i]*sqrt(beta)))*(kyL*Ez_amp[0] - kzL*Ey_amp
    Bx_amp[1] = (1/(omegas[i]*sqrt(beta)))*(kyL*Ez_amp[1] - kzL*Ey_amp
    By_amp[0] = (1/(omegas[i]*sqrt(beta)))*(kzL*Ex_amp[0]);
    By_amp[1] = (1/(omegas[i]*sqrt(beta)))*(kzL*Ex_amp[1]);
    Bz_amp[0] = (-1/(omegas[i]*sqrt(beta)))*(kyL*Ex_amp[0]);
    Bz_amp[1] = (-1/(omegas[i]*sqrt(beta)))*(kyL*Ex_amp[1]);

    //n_amp[0] = 0.0;
    //n_amp[1] = (Ez_amp[1]*mr*omega_ci)/(vaa*(kzL/sqrt(beta))*(gamma*(
    n_amp[0] = -(Ez_amp[1]*mr*omega_ci)/(vaa*(kzL/sqrt(beta))*(gamma*(v
    n_amp[1] = (Ez_amp[0]*mr*omega_ci)/(vaa*(kzL/sqrt(beta))*(gamma*(vt


    //cout << "In put_dfields kx, ky, kz, and Ex_amp are: "<< kx_d << "
    arr[0]= arr[0] + n_amp[0]*cos(kx_d*xr + ky_d*yr + kz_d*zr) - n_amp[
    arr[1]= arr[0];
    arr[2]= arr[2] + Ex_amp[0]*cos(kx_d*xr + ky_d*yr + kz_d*zr) - Ex_am
    arr[3]= arr[3] + Ey_amp[0]*cos(kx_d*xr + ky_d*yr + kz_d*zr) - Ey_am
```

```
        arr[4]= arr[4] + Ez_amp[0]*cos(kx_d*xr + ky_d*yr + kz_d*zr) - Ez_am
        arr[5]= arr[5] + Bx_amp[0]*cos(kx_d*xr + ky_d*yr + kz_d*zr) - Bx_am
        arr[6]= arr[6] + By_amp[0]*cos(kx_d*xr + ky_d*yr + kz_d*zr) - By_am
        arr[7]= arr[7] + Bz_amp[0]*cos(kx_d*xr + ky_d*yr + kz_d*zr) - Bz_am

    }
}
```

A.4 Code: put-dfields (in the particles folder)

```
void Particles3D::put_dfields(double arr[6], double xr, double yr, double z
double * omegas) {

    int nks, nx, ny, nz;
    double kx_d, ky_d, kz_d;
    double Ex_amp[2], Ey_amp[2], Ez_amp[2], Bx_amp[2], By_amp[2], Bz_amp[2]
    double Vxi_amp[2], Vyi_amp[2], Vzi_amp[2], Vxe_amp[2], Vye_amp[2], Vze_
    const double PI = 4.0 * atan( 1.0 );
    fstream my_file;
    my_file.open("wave_input.txt", ios::in);
    if (my_file) {
        my_file >> nks;
    } else {
        cout << "File not found " << endl;
        exit(0);
    }
    cout << "In dfields: omegas is " << omegas[0] << " " << omegas[1] << e
    //arr[6] = {0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0};

    for (int i =0; i < 6; i++){
        arr[i] = 0.0;
    }

    for (int i = 0; i < nks; i++) {
        my_file >> nx >> ny >> nz >> Ex_amp[0];
        kx_d = nx*2*PI/Lx;
        ky_d = ny*2*PI/Ly;
        kz_d = nz*2*PI/Lz;
        Ex_amp[1] = 0.0;

        double mr = fabs(qom_alt[0])/fabs(qom_alt[1]);

        double beta = (gamma/3.0)*(1/(vaa*vaa))*((vtherm[1]*vtherm[1]) + ((
        double L_var = sqrt(gamma)*(1/vaa)*sqrt(mr/(1 + mr))*sqrt(((vtherm[
double kxL = nx*(2*PI/Lx)*L_var;
double kyL = ny*(2*PI/Ly)*L_var;
double kzL = nz*(2*PI/Lz)*L_var;

double kLsum = sqrt(kyL*kyL + kzL*kzL);
double omega_ci = vaa*sqrt(1 + (1/mr));
double omega_ce = - mr*omega_ci;

Ez_amp[0] = 0.0;
Ez_amp[1] = Ex_amp[0]*(kzL/kyL)*(omegas[i]*omegas[i] - (kLsum*kLsum

Ey_amp[0] = 0.0;
Ey_amp[1] = Ex_amp[0]*(omegas[i]*omegas[i] - (kLsum*kLsum/(beta))*v

Bx_amp[0] = (1/(omegas[i]*sqrt(beta)))*(kyL*Ez_amp[0] - kzL*Ey_amp
Bx_amp[1] = (1/(omegas[i]*sqrt(beta)))*(kyL*Ez_amp[1] - kzL*Ey_amp
By_amp[0] = (1/(omegas[i]*sqrt(beta)))*(kzL*Ex_amp[0]);
By_amp[1] = (1/(omegas[i]*sqrt(beta)))*(kzL*Ex_amp[1]);
Bz_amp[0] = (-1/(omegas[i]*sqrt(beta)))*(kyL*Ex_amp[0]);
Bz_amp[1] = (-1/(omegas[i]*sqrt(beta)))*(kyL*Ex_amp[1]);

//n_amp[0] = 0.0;
//n_amp[1] = (Ez_amp[1]*mr*omega_ci)/(vaa*(kzL/sqrt(beta))*(gamma*(
n_amp[0] = -(Ez_amp[1]*mr*omega_ci)/(vaa*(kzL/sqrt(beta))*(gamma*(v
n_amp[1] = (Ez_amp[0]*mr*omega_ci)/(vaa*(kzL/sqrt(beta))*(gamma*(vt
Vxi_amp[0] = (1/vaa)*(Ey_amp[0] + Ex_amp[1]*(omegas[i]/omega_ci)) +
Vxi_amp[1] = (1/vaa)*(Ey_amp[1] - Ex_amp[0]*(omegas[i]/omega_ci)) -

Vxe_amp[0] = (1/vaa)*(Ey_amp[0] + Ex_amp[1]*(omegas[i]/omega_ce)) +
Vxe_amp[1] = (1/vaa)*(Ey_amp[1] - Ex_amp[0]*(omegas[i]/omega_ce)) -

//Vyi_amp[0] = (-1/vaa)*(Ex_amp[0] + Ey_amp[1]*(omegas[i]/omega_ci)
//Vyi_amp[1] = (-1/vaa)*(Ex_amp[1] - Ey_amp[0]*(omegas[i]/omega_ci)
Vyi_amp[0] = (-1/vaa)*(Ex_amp[0] - Ey_amp[1]*(omegas[i]/omega_ci))
Vyi_amp[1] = (-1/vaa)*(Ex_amp[1] + Ey_amp[0]*(omegas[i]/omega_ci))

//Vye_amp[0] = (-1/vaa)*(Ex_amp[0] + Ey_amp[1]*(omegas[i]/omega_ce)
//Vye_amp[1] = (-1/vaa)*(Ex_amp[1] - Ey_amp[0]*(omegas[i]/omega_ce)
Vye_amp[0] = (-1/vaa)*(Ex_amp[0] - Ey_amp[1]*(omegas[i]/omega_ce))
Vye_amp[1] = (-1/vaa)*(Ex_amp[1] + Ey_amp[0]*(omegas[i]/omega_ce))

Vzi_amp[0] = -(omega_ci/(omegas[i]*vaa))*Ez_amp[1] + (gamma*vtherm[
Vzi_amp[1] = (omega_ci/(omegas[i]*vaa))*Ez_amp[0] + (gamma*vtherm[1
        Vze_amp[0] = -(omega_ce/(omegas[i]*vaa))*Ez_amp[1] + (gamma*vtherm[
        Vze_amp[1] = (omega_ce/(omegas[i]*vaa))*Ez_amp[0] + (gamma*vtherm[0

        arr[0] = arr[0] + Vxe_amp[0]*cos(kx_d*xr + ky_d*yr + kz_d*zr) - Vxe
        arr[1] = arr[1] + Vye_amp[0]*cos(kx_d*xr + ky_d*yr + kz_d*zr) - Vye
        arr[2] = arr[2] + Vze_amp[0]*cos(kx_d*xr + ky_d*yr + kz_d*zr) - Vze
        arr[3] = arr[3] + Vxi_amp[0]*cos(kx_d*xr + ky_d*yr + kz_d*zr) - Vxi
        arr[4] = arr[4] + Vyi_amp[0]*cos(kx_d*xr + ky_d*yr + kz_d*zr) - Vyi
        arr[5] = arr[5] + Vzi_amp[0]*cos(kx_d*xr + ky_d*yr + kz_d*zr) - Vzi

        }

}
```