PHYS 449: Undergraduate Honours Thesis

Comparison of Landau Damping in Two Computer Models

by

Aaron Froese

A.Sc., University College of the Fraser Valley, 2002

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

BACHELOR OF SCIENCE

in

The Faculty of Science

(Physics)

THE UNIVERSITY OF BRITISH COLUMBIA

March 31, 2005

Abstract

The Vlasov-Maxwell model of a plasma is introduced and simplified to two-dimensional phase-space. Unstable growth and Landau damping of the electric field occurring in this model are developed for three different initial velocity distributions, the Maxwellian, two cold streams, and two Lorentzian streams. The particle-mesh scheme and the finite-difference approximation scheme for numerical simulation of the model are compared. Then the velocity distributions are run in two programs that employ the schemes, and the data is compared with theory.

Contents

List of Tables

List of Figures

List of Algorithms

Acknowledgements

Thanks go to Matt Choptuik and the UBC Numerical Relativity Group for their assistance and the use of their computing facilities.

Chapter 1

Introduction

Plasma is the fourth state of matter. A solid has few internal degrees of freedom; all its constituent atoms are locked in place, relative to each other. If one adds energy to a solid, the bonds holding those atoms together will break, and the material will gain internal degrees of freedom, eventually crossing a phase boundary and becoming a liquid. The atoms in a liquid are free to move independently, but face constraints on that movement. They must remain in contact, but can slide across each other. Heating a liquid to produce gas will eliminate these final constraints on the position of the atoms. They will fly into free space and eventually achieve a completely random distribution. As the heating continues, the atoms in the gas will speed up, eventually hitting one another with such force that they break into ions, the components of a plasma.

All these phase transitions, of course, do not take place instantly. There are states of equilibrium between each complete phase. The lattice in a solid begins to collapse first at points of weakness. The stronger bonds follow suit only when sufficient "incentive" is provided. Once a liquid starts evaporating at its surface, initially gas will be re-condensing as well. And when a gas begins to ionize, not every atom will lose all of its electrons. As the heat increases, so does the level of disintegration. For this reason, the actual point at which a gas becomes plasma is open to debate. However, as soon as the gas becomes weakly ionized, it does begin to exhibit the characteristic behaviour of a plasma.

Plasma is one of the few systems whose components interact by long-range forces, the other notable one being groups of massive bodies in gravitational interplay. Each free charge in the plasma is affecting, and subsequently being affected by, every other free charge. The charges produce electric fields and the charge currents produce magnetic fields. Then the electric field induces more magnetic field and the magnetic field induces more electric field; the fields propagate as electromagnetic waves. These fields produce forces on the particles that give rise to a wide range of complex behaviour. While this electromagnetic interaction accounts for the basic coherence of everyday objects, atoms are charge-neutral, and so it is only the residual fields that play a role. Therefore, the observed range of the force is microscopic compared to that in a plasma, and the resulting behaviour is instead simple and familiar.

Plasma physics, the study of the complex interplay between ions, has a number of broad applications. Ninety-nine percent of the visible matter in the universe is plasma. The stars are completely ionized and are constantly emitting charged particles as solar wind. The interstellar space is full of a sparse population of ions, and a denser grouping is found in the magnetospheres

around planets. An understanding of plasma physics is therefore fundamental to elucidate the mechanisms behind most space phenomena.

Plasma has applications in the lab as well. It is used to create laser beams and etch semiconductors. However, the most appealing use for plasma, and therefore the most convincing reason for delving into its secrets, is the possibility of controlled fusion.

Fusion generators promise potentially limitless clean energy. The fuels are only deuterium, stripped from heavy water molecules, and tritium, produced from lithium ores. Both are plentiful and easily accessible, and the process requires far less fuel than chemicals reactions, such as burning coal or oil. Unlike coal, oil, and even nuclear fission, fusion energy produces no harmful byproducts. There is no carbon dioxide nor particulate matter to be released into the atmosphere, and no radioactive waste to store. The stray neutrons that escape the reactor core are sparser than their fission counterparts, and so, while the reactor materials do become radioactive, the effect is orders of magnitude weaker than in fission reactors. In fact, it is safe to enter the reactor chamber within an hour after its deactivation.

1.1 Computer Simulations

The computer-modelled plasma that I study is less complex than a complete reactor simulation. This simulation is collisionless and non-relativistic, and variations in the plasma are limited to one dimension. Only the Coulomb interaction is invoked to give structure. The plasma is analyzed in phase-space, so that the simulation domain is two-dimensional. One dimension gives position and the other gives velocity. Technically, since the position of each component particle of the plasma is specified in phase-space, the number of dimensions required to specify a single state is $2N$-dimensional.

The computer can only simulate a finite domain size, so assumptions have to be made about what happens on the edges. This is easy for the velocity direction, the program just needs to ascertain that there are no particles with very large velocity, that is, a velocity larger than the domain boundary. In the spatial direction, since the plasma should not disappear from the field of view, the boundary conditions must be assumed to be periodic. That way, when particles fly off one side of the domain, they (or rather, their counterparts from another interval) reappear on the opposite side.

I study the behaviour of two different programs, one which employs a finite-differencing approximation (FDA) scheme and the other a particle-mesh (PM) scheme. The PM simulation keeps track of the positions and velocities of a number of representative particles. It creates a density function on a 1D grid based on the spatial distribution of those particles. Then it calculates the electric field from the density and finally attributes an electric force to each of the particles, depending on where they are. This force and the velocity of each particle determines how the particle moves through phase-space.

Instead of using particles, which are conceptually closer to an actual plasma,

the FDA simulation assumes that the plasma is a 'charge fluid.' The program keeps track of how much fluid there is at each point of a fixed grid. Fluid can flow from grid point to grid point, but the amount depends on the velocity and the electric force at that grid point. The electric force is calculated in much the same way as in the PM method, but without the difficulty of interpolating back and forth between the particles and the grid.

The physical problems that will be attempted with each program are simulation of the Maxwellian velocity distribution and the two-stream instability. The former exhibits Landau damping, and the latter an instability, which, surprisingly enough, are two sides of the same coin. In one case, the energy in the fields is damped, imparting it to the particles as kinetic energy. In the other, the kinetic energy is taken from the particles and produces growth in the fields. At some point in both cases, a balance is achieved and the effect saturates out.

1.2 Motivation

This project is intended to improve my understanding of plasma behaviour and allow me to gain more experience in computational physics. The physical problems I will be studying with my programs are standard examples, at least in the linearized regime, in most introductory plasma physics texts [2][3][4][5]. However, plasma physics has settled on particles as the de-facto standard for computational experimentation, and FDA simulations do not appear in those same texts. Comparing the two methods is a facet that is rarely covered, and as such, is novel research.

In any simulation, there are many parameters that can be modified. An experienced simulator will know what each one does and only adjust the ones that produce interesting variations in the physics. It is my hope that I will be able to vary many different aspects of the simulation, and thereby gain a better grasp of the balance between plasma behaviour and computational artifacts in both simulations.

Chapter 2

Mathematics

2.1 Development of the Vlasov-Maxwell Model

Plasmas typically contain a large number of ions, from 10^{15} in the Van Allen belts to 10^{20} in a fusion reactor to 10^{22} in an interstellar gas cloud. For this reason, it is definitely not feasible to keep track of each individual particle. Therefore, we introduce the phase-space density function $f(\vec{x}, \vec{v}, t)$, which is measured in units of $\mathrm{s}^3\ \mathrm{m}^{-6}$. It specifies the number of particles in a box of volume $dxdydz$ that have a velocity in a range $dv_x dv_y dv_z$ at each point in time. One can recover the particle number density and the average velocity by integrating over the velocity dimensions of the phase-space density.

$$n(x,t) = \int\int\int f(\vec{x}, \vec{v}, t) dv_x dv_y dv_z = \int f(\vec{x}, \vec{v}, t) d^3v \tag{2.1}$$

$$n\vec{u} = \int\int\int \vec{v} f(\vec{x}, \vec{v}, t) dv_x dv_y dv_z = \int \vec{v} f(\vec{x}, \vec{v}, t) d^3v \tag{2.2}$$

2.1.1 The Boltzmann Equation

Now, given a starting $f(\vec{x}, \vec{v}, t)$ and a set of forces, we should theoretically be able to completely determine the state of the system at any future time. This evolution should be unique, so that two different sets of initial conditions will never produce identical results, unless, of course, when one of the starting conditions is a future state of the other. Neither should one starting condition yields multiple possible outcomes.

Since each unique state causes the system to evolve along a unique path, the phase space density at any two points along that path should be equal. For example, a particle starts at $x(t_1)$ with velocity $v(t_1)$, and after some time ends up at $x(t_2)$ with velocity $v(t_2)$. Likewise, all the particles that are infinitesimally close to the first with a velocity infinitesimally similar to its own at t_1 should end up at much the same point by t_2. This is essentially the definition of the phase-space density function, so it is reasonable to say that

$$f(\vec{x}(t_2), \vec{v}(t_2), t_2) = f(\vec{x}(t_1), \vec{v}(t_1), t_1) \tag{2.3}$$

We will rewrite the equation by moving both terms to one side and adding a certain arbitrary constant.

$$\frac{f(\vec{x}(t_2), \vec{v}(t_2), t_2) - f(\vec{x}(t_1), \vec{v}(t_1), t_1)}{t_2 - t_1} = 0 \tag{2.4}$$

Since this relation is true for any value of t_1 and t_2, we can let t_2 get close to t_1 and rewrite each as $t_1 \equiv t$ and $t_2 \equiv t + \Delta t$.

$$\lim_{\Delta t \to 0} \frac{f(\vec{x}(t+\Delta t), \vec{v}(t+\Delta t), t+\Delta t) - f(\vec{x}(t), \vec{v}(t), t)}{\Delta t} = 0 \tag{2.5}$$

It is apparent that the result of assuming that systems evolve uniquely is simply a total derivative.

$$\frac{df(\vec{x}, \vec{v}, t)}{dt} = 0 \tag{2.6}$$

This result is known as Louville's theorem.

Louville's theorem provides an equation to describe the evolution of the system. The total derivative is expanded to

$$\begin{aligned}
0 &= \frac{df}{dt} \\
0 &= \frac{\partial f}{\partial t} + \frac{\partial x}{\partial t}\frac{\partial f}{\partial x} + \frac{\partial y}{\partial t}\frac{\partial f}{\partial y} + \frac{\partial z}{\partial t}\frac{\partial f}{\partial z} + \frac{\partial v_x}{\partial t}\frac{\partial f}{\partial v_x} + \frac{\partial v_y}{\partial t}\frac{\partial f}{\partial v_y} + \frac{\partial v_z}{\partial t}\frac{\partial f}{\partial v_z} \\
0 &= \frac{\partial f}{\partial t} + v_x\frac{\partial f}{\partial x} + v_y\frac{\partial f}{\partial y} + v_z\frac{\partial f}{\partial z} + a_x\frac{\partial f}{\partial v_x} + a_y\frac{\partial f}{\partial v_y} + a_z\frac{\partial f}{\partial v_z} \\
0 &= \frac{\partial f}{\partial t} + \vec{v}\cdot\nabla_x f + \vec{a}\cdot\nabla_v f
\end{aligned} \tag{2.7}$$

This result is called the Boltzmann equation. One can see that the characteristic equations are

$$\frac{d\vec{x}}{dt} = \vec{v} \qquad \frac{d\vec{v}}{dt} = \vec{a}, \tag{2.8}$$

which are the basic equations of Newtonian motion.

2.1.2 The Vlasov Equation

According to Newton's 2$^{\text{nd}}$ Law, in order to determine the acceleration for the Boltzmann equation, we must know both the mass of the object and the force applied to it. Since the aim is for an electromagnetic model, the only force acting on the particles is the Lorentz force.

$$\vec{a} = \frac{\vec{F}}{m} = \frac{q}{m}(\vec{E} + \vec{v}\times\vec{B}) \tag{2.9}$$

Plugging this into (2.7) gives the Vlasov equation, which depends on the charge-to-mass ratio, rather than just the mass.

$$\frac{\partial f}{\partial t} + \vec{v}\cdot\nabla_x f + \frac{q}{m}(\vec{E} + \vec{v}\times\vec{B})\cdot\nabla_v f = 0 \tag{2.10}$$

Certainly this in itself is not a revolutionary enough change to warrant a new name for the equation, but A A Vlasov in 1938 was the first to solve it to find the plasma dispersion function and apply it to various problems.

2.1.3 Maxwell's Equations

We have an equation to calculate the phase-space density function, but it depends on the electric and magnetic fields. These are not given, since they change in time depending on the movement of the charged particles. They can be solved using Maxwell's equations, given here in the Coulomb gauge,

$$\nabla^2 \vec{A} = -\mu_0 \vec{J} \tag{2.11a}$$

$$\nabla^2 \phi = -\frac{\rho}{\epsilon_0} \tag{2.11b}$$

$$\vec{E} = -\nabla\phi - \frac{\partial \vec{A}}{\partial t} \tag{2.11c}$$

$$\vec{B} = \nabla \times \vec{A} \tag{2.11d}$$

where the currents and charge densities can be extracted from the phase-space density function.

$$\rho = \sum_{species} qn = \sum_{species} q \int f(\vec{x}, \vec{v}, t) d^3v \tag{2.12a}$$

$$\vec{J} = \sum_{species} qn\vec{u} = \sum_{species} q \int \vec{v} f(\vec{x}, \vec{v}, t) d^3v \tag{2.12b}$$

Up until now, it has been assumed that there is just one phase-space density function, but if there is more than one type of particle, they must be kept track of separately. However, the electric and magnetic fields depend on the movement of all charged particles collectively. Therefore, all species must be summed over when calculating the charge and current densities.

The combination of (2.10), (2.11), and (2.12) constitute the Vlasov-Maxwell system of equations. They completely describe a collisionless plasma which interacts only electromagnetically, and despite such a simple description, encompass a very rich range of behaviour. However, for the purposes of this paper, they are quite complex, and it is helpful to make some simplifying assumptions to explore a more basic system.

2.2 Down to One Dimension

Simulating a plasma in the full six dimensions of phase-space (x, y, z, v_x, v_y, v_z) is computationally prohibitive and conceptually difficult. We must make some assumptions about the behaviour of the plasma so that the model is simplified to only one space dimension and its corresponding velocity dimension. Firstly, the plasma is assumed to be homogeneous in two directions, so that we can integrate over them.

$$f(x, \vec{v}, t) = \int \int f(\vec{x}, \vec{v}, t) dv_y dv_z \tag{2.13}$$

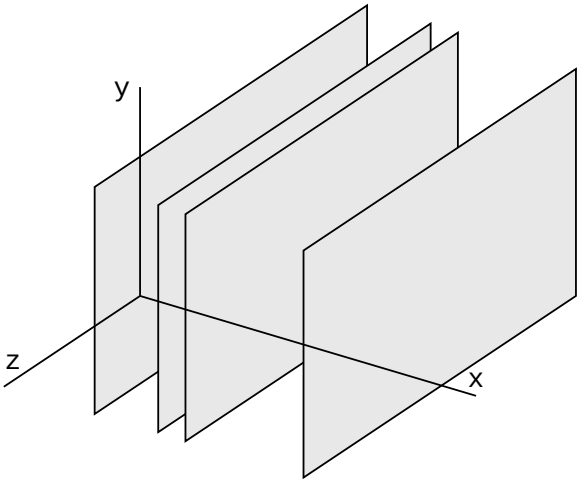


Figure 2.1: Infinite sheets of charge interact in a slab geometry where two spatial dimensions are homogeneous.

This condition is known as the slab geometry (Figure 2.1), as all 'particles' appearing in the density function are actually infinite sheets in the plane perpendicular to $\hat{x}$.

2.2.1 Motion of a Charged Particle in a Magnetic Field

A single charged particle in electric and magnetic fields experiences the Lorentz force due to the fields.

$$\vec{F} = q(\vec{E} + \vec{v} \times \vec{B}) \tag{2.14}$$

If we impose an external magnetic field that points in the $\hat{z}$-direction, such that $\vec{B} = B\hat{z}$, then the force due to that magnetic field is

$$\begin{aligned} m\dot{\vec{v}} = \vec{F_B} &\equiv q(\vec{v} \times \vec{B}) \\ &= qB(v_y\hat{x} - v_x\hat{y}). \end{aligned} \tag{2.15}$$

Since superposition is valid, we can analyze the component of the path of the particle that is only dependent on the magnetic field.

$$\dot{v}_x = \left(\frac{qB}{m}\right) v_y \tag{2.16a}$$

$$\dot{v}_y = -\left(\frac{qB}{m}\right) v_x \tag{2.16b}$$

$$\dot{v}_z = 0 \tag{2.16c}$$

This shows that the magnetic field does not affect the component of the velocity that is parallel to it. Taking the derivative of (2.16a) and substituting in (2.16b)

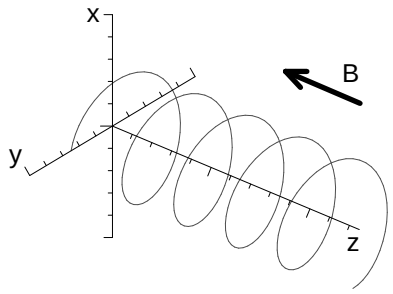


Figure 2.2: A particle travels a helical trajectory in a constant magnetic field.

produces an ODE that can be solved for v_x.

$$\ddot{v}_x = -\left(\frac{qB}{m}\right)^2 v_x \tag{2.17}$$

$$v_x = A\sin(\omega_c t + \phi) \tag{2.18}$$

where $\omega_c \equiv qB/m$ is called the 'cyclotron frequency' and A and ϕ are dependent upon initial conditions. Substituting (2.18) back into (2.16b) shows that v_y is

$$v_y = A\cos(\omega_c t + \phi). \tag{2.19}$$

This suggests that we define the velocity perpendicular to the magnetic field,

$$\begin{aligned} v_\perp &= \sqrt{v_x^2 + v_y^2} \\ &= \sqrt{A^2\cos^2(\omega_c t + \phi) + A^2\sin^2(\omega_c t + \phi)} \\ &= A, \end{aligned} \tag{2.20}$$

which turns out to be the amplitude of the oscillations in v. Integrating (2.18) and (2.19) gives the path of the particle due to the magnetic field.

$$x = -\frac{v_\perp}{\omega_c}\cos(\omega_c t + \phi) \tag{2.21a}$$

$$y = \frac{v_\perp}{\omega_c}\sin(\omega_c t + \phi) \tag{2.21b}$$

$$z = v_z t \tag{2.21c}$$

Thus, without a $\hat{z}$-component to the velocity, the particle traces out a circle in the xy-plane. Taking a non-zero v_z will cause the particle to travel in a helical trajectory along the magnetic field lines (Figure 2.2). The radius of the circle

or helix is

$$\begin{aligned} r &= \sqrt{x^2 + y^2} \\ &= \sqrt{\left(\frac{v_\perp}{\omega_c}\right)^2 \cos^2(\omega_c t + \phi) + \left(\frac{v_\perp}{\omega_c}\right)^2 \sin^2(\omega_c t + \phi)} \\ &= \frac{v_\perp}{\omega_c} = \frac{m v_\perp}{qB}. \end{aligned} \tag{2.22}$$

What is important to note is that the radius is inversely proportional to the strength of the magnetic field. If we impose a very strong field, then the radius goes to zero, and the particles are constrained to move in the same direction as the field. Therefore, the second condition we will apply to the model is that there must be a strong external magnetic field imposed in the $\hat{x}$-direction, such that magnetic fields in the other directions are negligible.

2.2.2 Electrostatic Equations

Now that motion has been restricted to one dimension, the Vlasov-Maxwell system of equations can be significantly streamlined. The Vlasov equation (2.10) becomes

$$\frac{\partial f}{\partial t} + v\frac{\partial f}{\partial x} + \frac{q}{m}(E_x + v_y B_z - v_z B_y)\frac{\partial f}{\partial v} = 0. \tag{2.23}$$

But since the perpendicular magnetic fields are assumed to be negligible, this simplifies further to

$$\frac{\partial f}{\partial t} + v\frac{\partial f}{\partial x} + \frac{q}{m}E_x\frac{\partial f}{\partial v} = 0. \tag{2.24}$$

Now we only need to calculate E_x, which from (2.11) is

$$\frac{\partial^2 \phi}{\partial x^2} = -\frac{\rho}{\epsilon_0} \tag{2.25a}$$

$$E_x = -\frac{\partial \phi}{\partial x} \tag{2.25b}$$

where the charge density is found just as before (2.12).

$$\rho = \sum_{species} q \int f(x, v, t) dv \tag{2.26}$$

2.2.3 Final Assumptions

In this model, we will assume that the electrons are free to move, while the protons are fixed, creating a uniform neutralizing background. This is useful because now we only need one phase-space density function, and since protons are 1837 times heavier than electrons, it is not an unreasonable assumption. Therefore, the charge density is more simply

$$\rho = -e \int f(x, v, t) dv + \rho_0, \tag{2.27}$$

where ρ_0 is the charge density of the protons, which remains to be found.

Plasmas contain electromagnetic components, in the form of 'plasma waves', which suggest that the problem will contain periodic solutions. Assuming that the spatial boundary conditions are periodic has two benefits, the first being that a finite simulated domain can imitate an infinite spatial domain, Additionally, charges reaching the border do not have to be artificially stopped or reflected, thereby breaking momentum conservation. This is preferable, since conserved quantities are useful for program diagnostics, and should be retained whenever possible.

Periodic boundary conditions can only exist if there are no diverging fields in the period. Periods outside some 'origin period' would experience the divergent fields and the effect would be compounded with greater distance. Charges in each period would experience different forces, but they are the same charges! This can also be seen by noting that in a non-neutral period, the potential on the right edge would not return to its value on the left edge, breaking the periodicity.

Therefore, the proton charge density is easy to calculate. To achieve neutrality, it must be equal to the total electron charge density, which is the total charge in one period divided by its length.

$$\rho_0 = \frac{e \int_L dx \int dv f(x,v,t)}{L} \tag{2.28}$$

where L is the length of the period and $f(x,v,t)$ is still the *electron* density function.

2.2.4 Summary of the Continuous Model

For convenience, I will collect all the equations that make up the simplified one-dimensional model. The assumptions that have been made to reach this point are as follows.

1. The charge distribution is homogeneous in the $\hat{y}$ and $\hat{z}$ directions.
2. There exists a very strong magnetic field pointing in the $\hat{x}$ direction.
3. The system is periodic in the $\hat{x}$ direction.
4. The protons are fixed, uniform and neutralizing.

This model is far simpler than the complete Vlasov-Maxwell system of equations, but it still produces very interesting behaviour.

$$\frac{\partial f}{\partial t} = -v\frac{\partial f}{\partial x} + \frac{e}{m_e}E\frac{\partial f}{\partial v} \tag{2.29a}$$

$$E = -\frac{\partial \phi}{\partial x} \tag{2.29b}$$

$$\frac{\partial^2 \phi}{\partial x^2} = -\frac{\rho}{\epsilon_0} \tag{2.29c}$$

$$\rho = -e\int f(x,v,t)dv + \rho_0 \tag{2.29d}$$

$$\rho_0 = \frac{e}{L}\int_L dx \int dv f(x,v,t) \tag{2.29e}$$

2.3 Landau Damping

To solve the system in the linearized regime, we will follow Landau's solution as presented in [4]. We assume that the phase-space density function is described by a time-independent, spatially uniform component and a transient component with a small amplitude oscillation in space. Likewise, the electric field can change in time, but has a regular oscillation in space.

$$f(x,v,t) = f_0(v) + f_1(v,t)e^{ikx} \tag{2.30a}$$

$$E(x,t) = \hat{E}(t)e^{ikx} \tag{2.30b}$$

Plugging these into (2.29a) gives the linear Vlasov equation.

$$\begin{aligned}\frac{\partial f_1}{\partial t} &= -ikvf_1 + \frac{e}{m_e}E\left(\frac{\partial f_0}{\partial v} + \frac{\partial f_1}{\partial v}\right)\\ \frac{\partial f_1}{\partial t} &\approx -ikvf_1 + \frac{e}{m_e}E\frac{\partial f_0}{\partial v}\end{aligned} \tag{2.31}$$

The final term has been dropped, since the electric field is first-order and f_1 is also first-order, making their combination second-order. We can also simplify the electrostatic equations. Putting them all together by taking the derivative of (2.29b) and then plugging in (2.29c) and (2.29d) gives Gauss's Law.

$$\epsilon_0\frac{\partial E(x,t)}{\partial x} = -e\int_{-\infty}^{\infty}\left(f_0(v) + f_1(v,t)e^{ikx}\right)dv + \rho_0 \tag{2.32}$$

Substituting in the simplified forms for E and f allows the derivative to be easily evaluated. The integral over f_0 is the total electron charge. It cancels with ρ_0, the total proton charge.

$$ik\epsilon_0\hat{E}(t)e^{ikx} = -e\int_{-\infty}^{\infty} f_1(v,t)e^{ikx}dv \tag{2.33}$$

The exponential on the right can come out of the integral and cancel the one on the left, giving the final form

$$ik\epsilon_0\hat{E}(t) = -e\int_{-\infty}^{\infty} f_1(v,t)dv. \tag{2.34}$$

The system has thus been reduced to a linearized version of the Vlasov equation (2.31) and a linearized version of Poisson's equation (2.34).

2.3.1 The Laplace Transform

Since f_1 is transient, the Laplace transform is useful for solving this problem. The transformation is defined as

$$\mathcal{L}\{f_1(v,t)\} \equiv \tilde{f}_1(v,s) = \int_0^{\infty} f_1(v,t)e^{-st}dt \tag{2.35}$$

and the inverse is a contour integral around the complex left half-plane encompassing all singularities.

$$\begin{aligned} f_1(v,t) = \mathcal{L}^{-1}\{f_1(v,s)\} &= \frac{1}{2\pi i}\oint_C \tilde{f}_1(v,s)e^{st}ds \\ &= \sum \text{Res}\left\{\tilde{f}_1(v,s)e^{st}\right\} \end{aligned} \tag{2.36}$$

The transform of a derivative is

$$\mathcal{L}\left\{\frac{df_1}{dt}\right\} = s\tilde{f}_1(v,s) - f_1(v,0) \tag{2.37}$$

2.3.2 The Dispersion Relation

Taking the Laplace transform of the linearized Vlasov equation (2.31) gives

$$(s+ikv)\tilde{f}_1(v,s) - \frac{e}{m_e}\tilde{E}(s)\frac{\partial f_0}{\partial v} = f_1(v,0) \tag{2.38}$$

while the transform of Poisson's equation (2.34) is

$$ik\epsilon_0\tilde{E}(s) = -e\int \tilde{f}_1(v,s)dv. \tag{2.39}$$

Solving the first for $\tilde{f}_1(v,s)$ and plugging it into the second gives an equation that can be solved for $\tilde{E}(s)$.

$$\begin{aligned} ik\epsilon_0\tilde{E}(s) &= -\frac{e^2}{m_e}\tilde{E}(s)\int_{-\infty}^{\infty}\frac{\partial f_0/\partial v}{(s+ikv)}dv - e\int_{-\infty}^{\infty}\frac{f_1(v,0)}{s+ikv}dv \\ \left(1 - \frac{ie^2}{km_e\epsilon_0}\int_{-\infty}^{\infty}\frac{\partial f_0/\partial v}{s+ikv}dv\right)\tilde{E}(s) &= \frac{ie}{k\epsilon_0}\int_{-\infty}^{\infty}\frac{f_1(v,0)}{s+ikv}dv \end{aligned} \tag{2.40}$$

The term inside the braces is the dispersion relation for the plasma, describing how fast waves propagate depending upon their frequency. In the simple situation where a sheet of electrons interacts with a sheet of protons, the frequency of small oscillations is

$$\omega_p^2 = \frac{ne^2}{m_e \epsilon_0} \tag{2.41}$$

This is called the plasma frequency, since it often appears in plasma dispersion relations as the first-order term. It nearly appears in this dispersion relation, missing only the particle number density n, which is contained in f_0. By defining a normalized function $\hat{f}_0 \equiv f_0/n$, we can set the Landau dispersion relation to

$$D(k,s) = 1 - i\frac{\omega_p^2}{k}\int_{-\infty}^{\infty} \frac{\partial \hat{f}_0/\partial v}{s+ikv} dv. \tag{2.42}$$

Finally, in all the cases I will analyze, at the initial time $f_1(v) = 0$, so the term on the LHS of (2.40) is just a constant. This means that the Laplace transform of the electric field is only

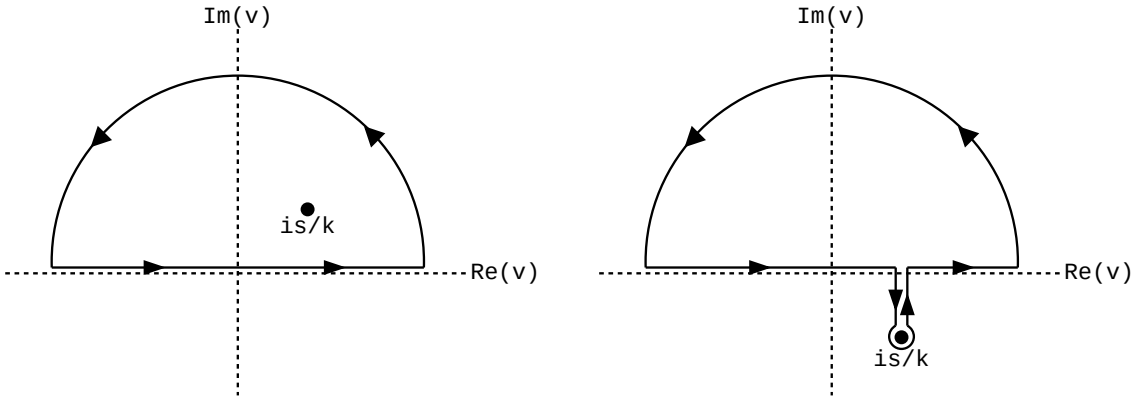
$$\tilde{E}(s) = \frac{iA}{D(k,s)}. \tag{2.43}$$

2.3.3 The Inverse Transform

Now, to find the electric field, there are two integrations that must be performed. The first one is in the dispersion function, but it has a singularity at $is/k = \omega/k \equiv v_p$. Physically, this singularity is caused by resonance at the phase-velocity of the plasma. The closer electrons are to the phase-velocity, the more strongly they will be affected by the waves. An electron that is not traveling near the phase-velocity will spend roughly equal time being accelerated on the front of the wave and being decelerated on the back of the wave, and experience a net force near zero. However, an electron that is going only slightly slower than the wave will spend a long time in front of it. Just like a surfer, the wave will push it along, making its journey to the crest take a long time. Then it will fall down the other side quickly and ride the next wave.

Conversely, an electron that is moving slightly faster than the wave will try to rise over the wave from the back, but will be constantly hindered by climbing the potential. When it reaches the peak, it quickly falls down the front of the wave, spending little time in the accelerating potential. Overall, the average force it experiences will decelerate it. Since particles moving slightly faster than resonance are slowed and particles moving slightly slower than resonance are accelerated, the effect is that the velocity distribution near the phase-velocity will flatten out.

To calculate the integral in the dispersion function, it is simplest to use contour integration (Figure 2.3a). By residue theory, the integral is the sum of all the residues with positive imaginary components. However, the singularity caused by the phase-velocity resonance must be treated with special care. As the system evolves, the phase-velocity changes. If it changed from being positive



(a) (b)

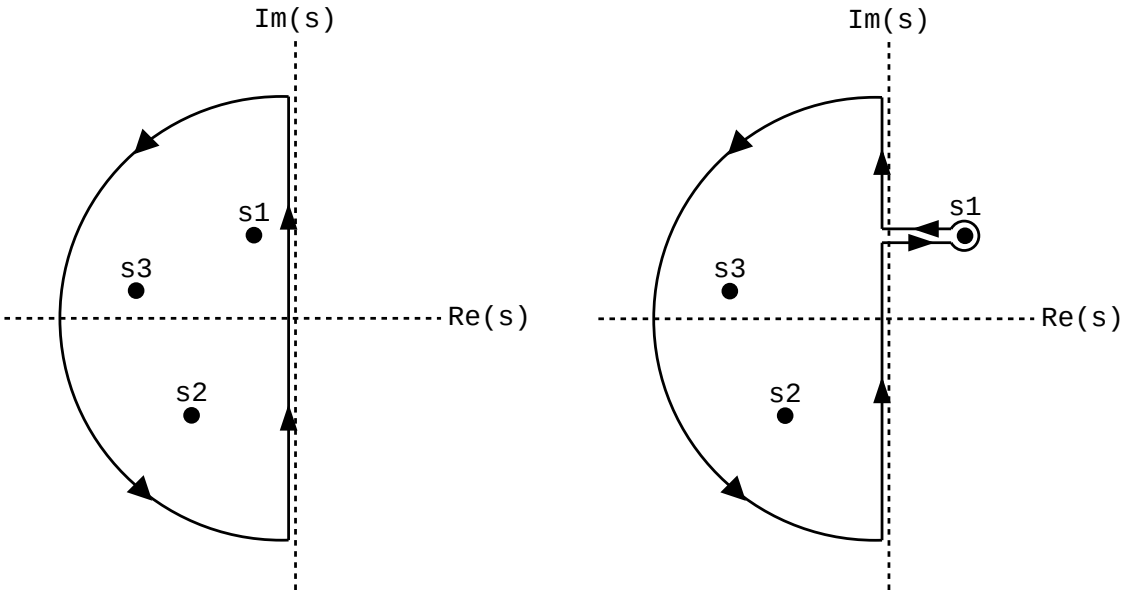
Figure 2.3: (a) The contour for integrating the dispersion function when its singularity is positive and (b) the analytic continuation when the singularity is negative.

to negative, the dispersion relation would suffer an unphysical discontinuity. Therefore, the contour must be deformed (Figure 2.3b) to always include the singularity so as to provide a proper analytic continuation.

$$\begin{aligned} D(k,s) &= 1 - i\frac{\omega_p^2}{k}\left[\oint_C \frac{\partial \hat{f}_0/\partial v}{s+ikv}dv\right] \\ &= 1 - i\frac{\omega_p^2}{k}\left[2\pi i \sum \mathrm{Res}\left\{\frac{\partial \hat{f}_0/\partial v}{s+ikv}\right\}\right] \\ &= 1 + 2\pi\frac{\omega_p^2}{k}\left[\left.\frac{\partial \hat{f}_0}{\partial v}\right|_{v=is/k} + \sum_j \mathrm{Res}\left\{\frac{\partial \hat{f}_0/\partial v}{s+ikv}; v=v_j\right\}\right] \end{aligned} \tag{2.44}$$

Alternatively, in certain cases the dispersion function can be handled by series expansion. However, the phase-velocity singularity must still be analyzed through contour integration. This method is used in Section 2.4 when dealing with the Maxwellian distribution.

Once the dispersion relation has been found, the electric field can be determined. The only singularities in the transform of the electric field occur at the



(a) (b)

Figure 2.4: The Laplace inversion contour for the electric field in the case of (a) Landau damping when all zeros have $\mathrm{Re}(s) < 0$ and (b) an instability when one or more zeros of the dispersion function have $\mathrm{Re}(s) > 0$.

zeros of the dispersion function.

$$\begin{aligned}\hat{E}(t) &= \oint_C \tilde{E}(s)e^{st}ds \\ &= \sum_{j=1}^{N} \mathrm{Res}\left\{\frac{iAe^{st}}{D(k,s)}; s = s_j\right\} \\ &= \sum_{j=1}^{N} e^{s_j t}\mathrm{Res}\left\{\frac{iA}{D(k,s)}; s = s_j\right\} \end{aligned} \tag{2.45}$$

In the last step, the exponential does not create any singularities, so it can be removed from the residue calculation. For an inverse Laplace transform, *every* singularity must lie inside the contour (Figure 2.4). In some cases, this would be prohibitive, since N could be a very large number. However, at large times, the residue whose singularity contains the largest real component (s_1) will become the dominant term.

$$\lim_{t\to\infty} \hat{E}(t) = e^{s_1 t}\mathrm{Res}\left\{\frac{iA}{D(k,s)}; s = s_1\right\} \tag{2.46}$$

When s_1 has a positive real component, the electric field grows with time, so there is an instability. But when $\mathrm{Re}(s_1)$ is negative, the electric field experiences Landau damping. Because of the resonance with the phase velocity, the general rule is that an instability will occur when there are more particles moving slightly faster compared to the phase-velocity than there are moving slightly slower. When the velocity distribution flattens out, there is a loss of kinetic energy, which is imparted to the fields. Landau damping will occur when there is a larger population moving slightly slower than the phase-velocity. Flattening the velocity distribution requires an addition of energy, which is taken from the fields.

2.4 Maxwellian Distribution

The Maxwellian distribution characterizes a completely thermalized gas. However, in the case of a plasma, we shall see that it is not the state of highest entropy. The Maxwell velocity distribution in one dimension is

$$\hat{f}_0(v) = \frac{1}{\sqrt{2\pi}v_t} \exp\left(-\frac{v^2}{2v_t^2}\right). \tag{2.47}$$

When plugging this into the dispersion relation, it is apparent that there is only one singularity at $v = is/k$.

$$\begin{aligned} D(k,s) &= 1 - \frac{\omega_p^2}{k} \oint_C \frac{1}{s+ikv} \frac{-v}{\sqrt{2\pi}v_t^3} \exp\left(-\frac{v^2}{2v_t^2}\right) \\ &= 1 - \sqrt{2\pi}\frac{\omega_p^2 s}{k^3 v_t^3} \exp\left(\frac{s^2}{2k^2 vt^2}\right) \end{aligned} \tag{2.48}$$

The dispersion function is resolved easily enough. However, we now have a problem, because it has infinitely many zeros in the positive complex plane (Figure 2.5). The solution is to calculate the integral without resorting to residue theory.

The Maxwellian distribution is an example of weak Landau damping. The first zero of the dispersion function lies just to the left of the imaginary axis. Therefore, a semicircular detour has to be put in the contour to accommodate the singularity that will arise when calculating the electric field. The rest of the integral can be found by a principal value calculation. Substituting $s = -i\omega$ into the original dispersion relation (2.42) gives

$$D(k,s) = 1 + \frac{\omega_p^2}{k}\left(\text{p.v.}\int_{-\infty}^{\infty} \frac{\partial f_0/\partial v}{\omega - kv} - \frac{\pi i}{k}\left.\frac{\partial f_0}{\partial v}\right|_{v=\omega/k}\right) \tag{2.49}$$

Assuming that the phase-velocity is larger than a majority of the particle velocities, we can expand the integrand in powers of kv/ω.

$$D(k,s) = 1 + \frac{\omega_p^2}{k}\left(\int_{-\infty}^{\infty} \frac{\partial f_0}{\partial v}\left(\frac{1}{\omega} + \frac{kv}{\omega^2} + \frac{k^2v^2}{\omega^3} + \frac{k^3v^3}{\omega^4} + \ldots\right) dv - \frac{\pi i}{k}\left.\frac{\partial f_0}{\partial v}\right|_{v=\omega/k}\right) \tag{2.50}$$

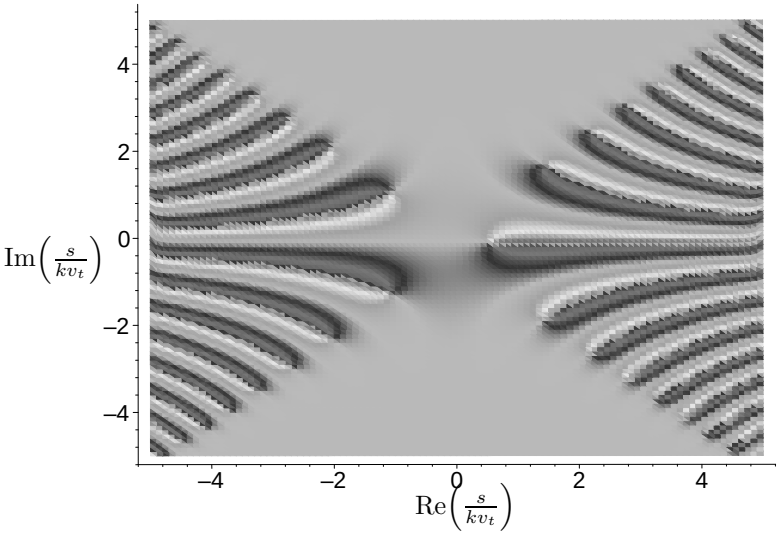


Figure 2.5: The complex plot of the dispersion function for a Maxwellian distribution. There is a zero at the tip of each finger-like projection.

Each component of the integrand can then be analyzed individually for the Maxwell distribution.

$$\int_{-\infty}^{\infty} \frac{\partial f_0}{\partial v} dv = 0 \qquad \int_{-\infty}^{\infty} \frac{\partial f_0}{\partial v} v dv = -1 \tag{2.51a}$$

$$\int_{-\infty}^{\infty} \frac{\partial f_0}{\partial v} v^2 dv = 0 \qquad \int_{-\infty}^{\infty} \frac{\partial f_0}{\partial v} v^3 dv = -3v_t^2 \tag{2.51b}$$

These values make the approximation to the dispersion function

$$D(k,s) = 1 - \frac{\omega_p^2}{\omega^2}\left(1 + \frac{3k^2v_t^2}{\omega^2}\right) + \frac{i}{2}\left(\frac{\pi}{2}\right)^{1/2} \frac{\omega_p^2 \omega}{k^3 v_t^3} \exp\left(-\frac{\omega^2}{2k^2v_t^2}\right) \tag{2.52}$$

The zero of this equation occurs at

$$\omega^2 = \omega_p^2\left(1 + \frac{3k^2v_t^2}{\omega^2}\right) - \frac{i}{2}\left(\frac{\pi}{2}\right)^{1/2} \frac{\omega_p^2 \omega^3}{k^3 v_t^3} \exp\left(-\frac{\omega^2}{2k^2v_t^2}\right) \tag{2.53}$$

Since both the second real term and the imaginary component are small corrections, the factors of ω can be approximated with the plasma frequency ω_p.

$$\omega^2 = \omega_p^2 + 3k^2v_t^2 - \frac{i}{2}\left(\frac{\pi}{2}\right)^{1/2} \frac{\omega_p^4 \omega}{k^3 v_t^3} \exp\left(-\frac{\omega_p^2}{2k^2v_t^2}\right) \tag{2.54}$$

Then, by using the binomial theorem on the real part and dividing both sides of the imaginary part by a factor of ω, the components of the frequency are

$$Re(\omega) \approx \omega_p + \frac{3}{2}k^2 v_t^2 \tag{2.55a}$$

$$Im(\omega) \approx -\frac{1}{2}\left(\frac{\pi}{2}\right)^{1/2} \frac{\omega_p^4}{k^3 v_t^3} \exp\left(-\frac{\omega_p^2}{2k^2 v_t^2}\right) \tag{2.55b}$$

One can see that the oscillation frequency is indeed very close to the plasma frequency, so the earlier substitutions were appropriate. Interestingly, the imaginary part is negative, so the system is always damped.

2.5 Two-Stream Instability

When two beams of electrons pass through each other, the resulting electric field causes a loss of free-streaming energy and sets up phase-space vortices as the electrons fall under the influence of the growing plasma waves. The velocity distribution for the two streams can take any number of shapes, but usually they are assumed to be individually Maxwellian.

2.5.1 Cold Streams

In the limit that the temperature of the streams approaches zero, the distribution is just two delta functions.

$$\hat{f}_0(v) = [\delta(v - v_0) + \delta(v + v_0)]/2 \tag{2.56}$$

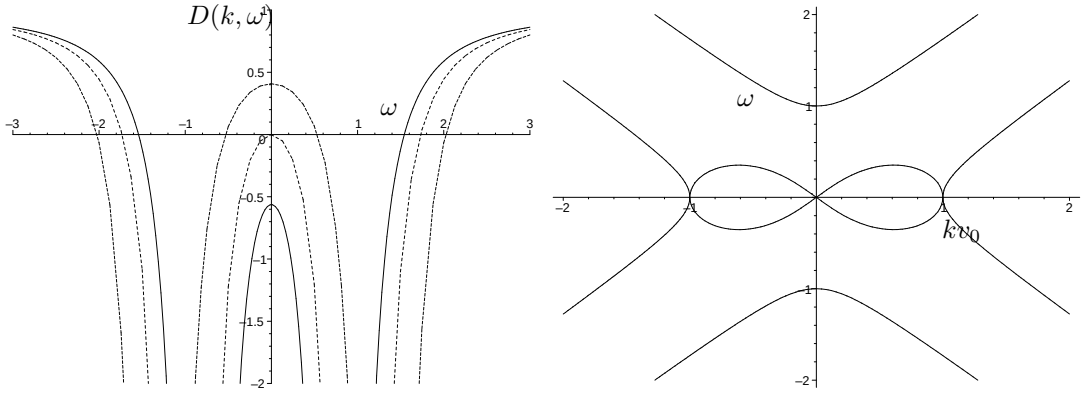
In this case, the dispersion function (2.42) takes the form

$$\begin{aligned} D(k,s) &= 1 + \frac{\omega_p^2}{k}\int_{-\infty}^{\infty} \frac{\partial f_0/\partial v}{\omega - kv} dv \\ &= 1 + \frac{\omega_p^2}{k}\left\{ -\int_{-\infty}^{\infty} f_0 \frac{\partial}{\partial v}\left(\frac{1}{\omega - kv}\right) dv + \left[\frac{f_0}{\omega - kv}\right]_{-\infty}^{\infty} \right\} \\ &= 1 - \omega_p^2 \int_{-\infty}^{\infty} \frac{f_0}{(\omega - kv)^2} dv \\ &= 1 - \frac{\omega_p^2}{2}\left(\frac{1}{(\omega - kv_0)^2} + \frac{1}{(\omega + kv_0)^2}\right) \end{aligned} \tag{2.57}$$

This is a quartic equation with the solutions

$$\omega = \pm\sqrt{k^2 v_0^2 + \omega_p^2 \pm \sqrt{4k^2 v_0^2 + \omega_p^2}} \tag{2.58}$$

The inner root contains only positive values, but the quantity in the outer root could be positive or negative. Therefore, there are either two real roots and two complex roots, or all four roots are real (Figure 2.6).



(a) (b)

Figure 2.6: (a) The dispersion function for the cold two-stream instability where the solid, dashed, and dotted lines correspond to the cases where $kv_0 < \omega_p$, $kv_0 = \omega_p$, and $kv_0 > \omega_p$, respectively. (b) The values of ω/ω_p which are zeros of the dispersion function. ω is imaginary on the infinity shape, but real on the four hyperbolas.

$$\left|\frac{kv_0}{\omega_p}\right| < 1 \implies \begin{cases} \text{two real roots} \\ \text{two complex roots} \end{cases}$$

$$1 < \left|\frac{kv_0}{\omega_p}\right| \implies \text{all four roots real}$$

$$\frac{kv_0}{\omega_p} = \frac{\sqrt{6}}{4} \implies \max\{\text{Im}(\omega)\} = \frac{\omega_p}{\sqrt{8}}$$

Complex solutions correspond to growth and real solutions correspond to stable oscillations. In this model, there can only be one or the other (at the initial time). One can see from these observations that as the wavenumber of the initial perturbation is increased, eventually a threshold will be crossed, preventing the instability from occurring.

2.5.2 Warm Streams

The Coulomb forces in a two stream instability thermalize the streams, spreading the peaks in the velocity distribution. Therefore, the cold stream approximation is not valid for long. Completely cold streams are impossible to create anyway; there will always be some variance in momentum between particles. It is important to compare the cold stream approximation to the behaviour of a pair of warm streams. Ideally, the warm streams would be two Gaussians. However, Gaussians have the disadvantage of error functions appearing in the

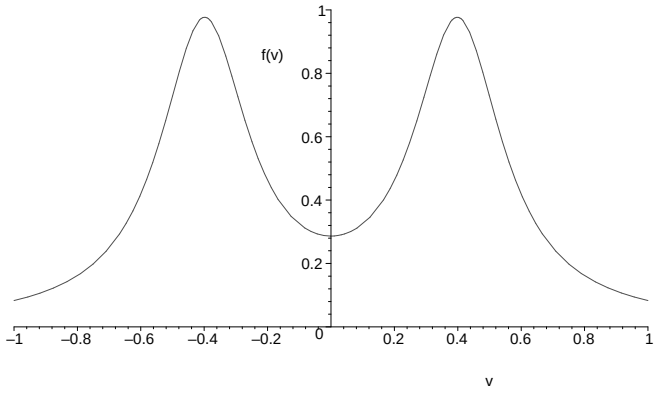


Figure 2.7: The warm stream velocity distribution is two Lorentzians.

calculations, so the zeros must be calculated numerically. Instead, I will use a pair of Lorentzians, which yield an analytic dispersion function.

$$\hat{f}_0(v) = \frac{b}{2\pi}\left[\frac{1}{(v-a)^2+b^2} + \frac{1}{(v+a)^2+b^2}\right] \tag{2.59}$$

This velocity distribution can fortunately be treated completely with the contour integration method described in Section 2.3, without having to resort to approximations.

The dispersion relation is

$$D(k,s) = 1 - i\frac{\omega_p^2}{k}\left\{\oint_C \frac{b}{\pi}\left[\frac{v-a}{((v-a)^2+b^2)^2} + \frac{v+a}{((v+a)^2+b^2)^2}\right]\frac{dv}{s+ikv}\right\} \tag{2.60}$$

The singularities occur at four symmetric points, $v = \pm a \pm ib$, as well as the phase-velocity, $v = is/k$. Only the residues in the positive complex plane are included in the calculation. These are

$$\text{Res}\left\{\frac{is}{k}\right\} = \frac{b}{\pi}\left[\frac{is/k-a}{((is/k-a)^2+b^2)^2} + \frac{is/k+a}{((is/k+a)^2+b^2)^2}\right] \tag{2.61a}$$

$$\text{Res}\left\{a+ib\right\} = \frac{-1}{4\pi k(is/k+a-ib)^2} \tag{2.61b}$$

$$\text{Res}\left\{-a+ib\right\} = \frac{-1}{4\pi k(is/k-a-ib)^2}, \tag{2.61c}$$

which make the dispersion relation take the following relatively simple form.

$$D(k,s) = 1 + i\frac{\omega_p^2}{k^2}\frac{(s/k+b)^2-a^2}{((s/k+b)^2+a^2)^2} \tag{2.62}$$

Like the cold streams, this is also a quartic equation with four roots. However, the equation for the roots does not have the visual appeal of that in the

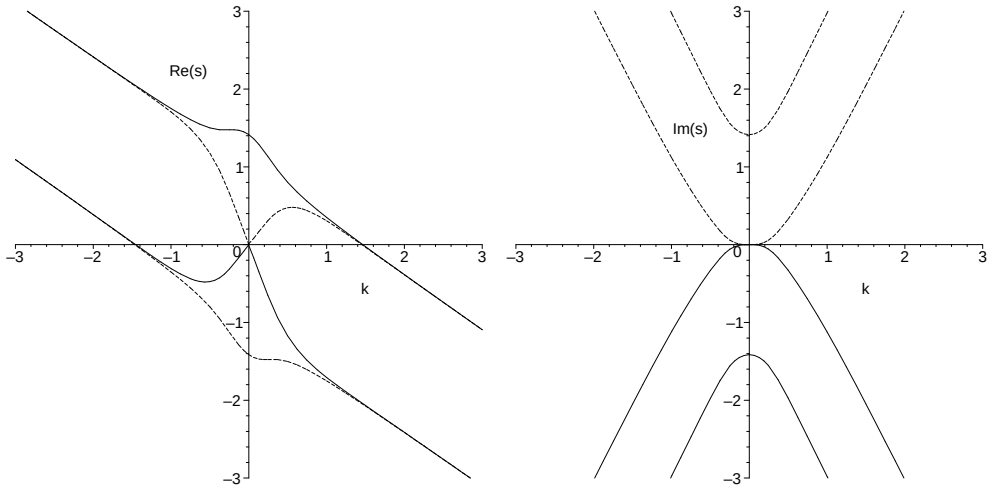


Figure 2.8: The (a) real and (b) imaginary components of the four zeros of the dispersion function for the double Lorentzian velocity distribution.

case of the cold stream.

$$s = -bk \pm \frac{1}{2}\sqrt{-i2\omega_p^2 - 4a^2k^2 \pm 2\omega_p\sqrt{i8a^2k^2 - \omega_p^2}} \tag{2.63}$$

Figure 2.8 shows how the positions of the zeros vary with the wavenumber. It is apparent that there are two positive and two negative roots until a threshold value of k is reached, after which all the roots are negative. Physically, like the cold streams, as the length of the domain is reduced, there becomes insufficient room for vortices to form, so the instability disappears.

By modifying some of the parameters in the velocity distribution, we can form a qualitative picture of the requirements for instability. For the root with the largest real component, the $k = 0$ intercept is $s = \sqrt{2}\omega_p(1 - i)$. The asymptotic behaviour at large k is $s = \omega_p/2 - bk - i2ak$. The separation of the peaks a is proportional to the rate of decay to the asymptotic behaviour. Assuming they are sufficiently far apart $a > \omega_p/k$, the last positive root becomes negative at approximately $k = \omega_p/2b$ and the instability disappears. As the peaks are brought closer together, the maximum wavenumber that can still support an instability becomes very slightly larger, but it never grows to be more than $k = \omega_p/\sqrt{2}b$.

Since the roots have a complicated form, it is simplest to calculate the growth of the electric field numerically. As an example, using the parameters of the initial distribution shown in Figure 2.7 and setting both the plasma frequency

and wavenumber to unity, the four roots of the dispersion fall at

$$\left.\begin{array}{rl} a = & 0.4 \\ b = & 0.17 \\ \omega_p = & 1 \\ k = & 1 \end{array}\right\} \Longrightarrow \left\{\begin{array}{rl} s_1 = & 0.497 - i0.718 \\ s_2 = & 0.297 - i0.718 \\ s_3 = & 0.269 - i0.752 \\ s_4 = & -0.131 - i0.752 \end{array}\right. \tag{2.64}$$

At a large time, the rate of growth is proportional to the root with the largest real component. It is easy to see that in this example, we should expect a growth rate approximately equal to one half.

Interestingly, my derivation of the conditions for stability using only contour integration does not agree with the standard approximation. From the approximation, [4] derives a simple expression to find the stability of an arbitrary symmetric double-peak distribution. It states that instabilities occur when

$$\frac{\omega_p^2}{k^2}\int_{-\infty}^{\infty}\frac{f_0(v) - f_0(0)}{v^2}dv > 1 \tag{2.65}$$

which for this case of two Lorentzians is

$$\frac{\omega_p^2}{k^2}\frac{a^2 - b^2}{(a^2 + b^2)^2} > 1 \tag{2.66}$$

However, I found qualitatively that

$$a \ll \omega_p/k \implies \frac{\omega_p^2}{k^2}\frac{1}{2b^2} > 1 \tag{2.67}$$

$$a \gg \omega_p/k \implies \frac{\omega_p^2}{k^2}\frac{1}{4b^2} > 1. \tag{2.68}$$

These limits are obviously not the same as those of (2.66). Therefore, this will make a good test of theory.

2.6 Properties of Interest

Other than the growth or damping rate of the electric field, there are a number of interesting and important physical quantities that can be watched in any particular experiment. A more extensive list is found in [2]. Assuming that the simulation is occurring in the phase-space of one dimension, one can find three different types of data.

Phase-Space Snapshots (2D): At various times, one can output the raw phase-space density functions. This would provide a lot of detail, but can take up much more space than performing some simplification before dumping the data to disk.

- phase-space density function - $f(x, v, t)$

Projection Snapshots (1D): Phase-space data is edifying, but no real experiment can sample it. Therefore, projecting the data into measurable dimensions is preferable. Grid quantities to generate are the velocity and speed distributions in velocity space, the particle number density, charge density, electric potential, and electric field in real space, and the electrostatic energy in momentum space.

- velocity distribution - $f(v,t) = \int_{-\infty}^{\infty} f(x,v,t)dx$
- speed distribution - $f(v,t) = \int_{0}^{\infty} f(x,v,t) + f(x,-v,t)dx$
- particle number density - $n(x,t) = \int_{-\infty}^{\infty} f(x,v,t)dv$
- charge density - $\rho(x,t) = q \cdot n(x,t)$
- electric potential - $\phi''(x,t) = -\rho(x,t)/\epsilon_0$
- electric field - $E(x,t) = -\phi'(x,t)$
- charge density in p-space - $\rho(k,t) = 2\left|\int_{-\infty}^{\infty} \rho(x,t)e^{2\pi ikx}dx\right|$
- electric potential in p-space - $\phi(k,t) = 2\left|\int_{-\infty}^{\infty} \phi(x,t)e^{2\pi ikx}dx\right|$
- electrostatic energy p-space distribution - $E_{es}(k,t) = \frac{1}{4}\rho(k,t)\phi(k,t)$

Histories (scalar): Global scalar quantities have the advantage of using up little disk space, so they can be output frequently to provide a detailed picture of system evolution in time. Since most are conserved in time, their conservation (or lack thereof) is useful as a diagnostic tool. These include total energy, momentum, and charge.

- electrostatic energy - $E_{es}(t) = \frac{1}{2}\int_{-\infty}^{\infty} \rho(x,t)\phi(x,t)dx$
- kinetic energy - $KE(t) = \frac{1}{2}m\int_{-\infty}^{\infty} f(x,v,t)v^2dv$
- total energy - $E(t) = KE(t) + E_{es}(t)$
- total momentum - $p(t) = \int_{-\infty}^{\infty} f(x,v,t)|v|dv$
- total charge - $Q(t) = q\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x,v,t)dxdv$

Monitoring these physical values can paint a detailed picture of what the simulation is doing. Of course, the choice of output depends on what the experimenter is attempting to prove. In my case, I was just exploring, so I implemented all of them.

Chapter 3

Numerical Methods

3.1 Particle-Mesh

To simulate a plasma, a particle-mesh simulation keeps track of a number of representative particles. It records each one's position and velocity at each time step, and then calculates their new positions for the next time step. The number of simulated particles is, of course, much smaller than the number in the physical plasma. To justify this, we assume that if a number of ions start very close together with nearly the same velocity, then after some time, they will still be close together with another velocity. Therefore, we can clump each set of close particles together into a superparticle.

However, computing the interaction forces between N particles is not terribly feasible. There are $\mathcal{O}(N^2)$ operations required, so that even clumping particles together does not help much. After all, making superparticles reduces the resolution of the simulation, so we wish to make them as small as possible, keeping their number as large as possible. To reduce the number of operations, we will calculate the charge density and electric field on a one-dimensional mesh.

In a single time step of a PM simulation, the computer performs the following operations.

1. Use particle positions to assign an equivalent charge density on the mesh.
2. Use Poisson's equation to calculate the electric potential on the mesh.
3. Find the electric field, and equivalently, the force on the mesh.
4. Move each particle based on the force it feels and its velocity.

The first and last processes, converting between the particles and the mesh, are just linearly proportional to the particle number. The middle two processes take place on the M mesh points and, while calculating the electric field is a simple derivative and linear in M, solving Poisson's equation is a matrix problem requiring $M \log_2 M$ operations. Therefore, the PM method requires $\mathcal{O}(N + M \log_2 M)$ operations. This is much more efficient than the particle-to-particle interaction method, assuming that $M \log_2 M$ is not close to N^2.

Of course, the number of mesh points should be much less than the number of particles. The particles must reasonably cover two-dimensions in phase-space, while the mesh spans only one. Otherwise, it would be likely that each mesh point would have either one or no particles associated with it. The resulting

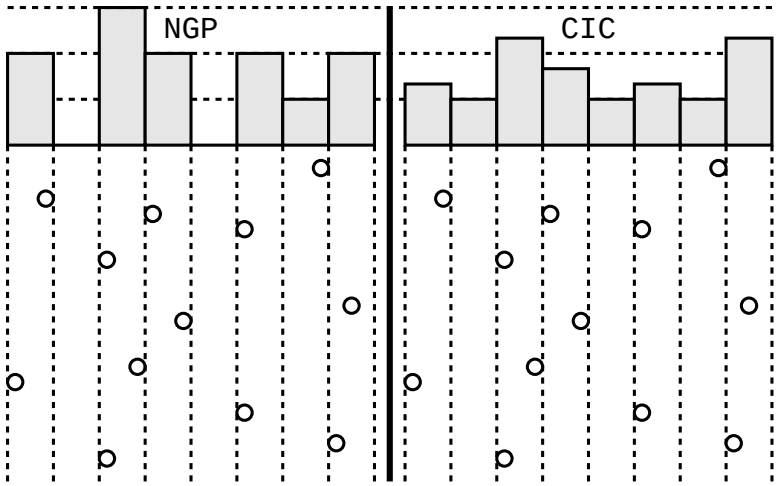


Figure 3.1: A visual comparison of the NGP and CIC charge assignment schemes. Given the same particle distribution, NGP produces large peaks and troughs compared to the smoother result from CIC.

charge density would be extremely discontinuous and noisy. Therefore, for better performance, the number of charges should be much larger than the mesh size.

3.1.1 Assigning Charge to the Mesh

At the start of each time step, we have a distribution of superparticles in phase-space. We wish to find the one-dimensional charge density (2.29d). A simple way to do this would be to cycle through all the particles and add the charge for each one to the nearest grid point. Letting the charge and spatial position of each particle be q and x_i, respectively, and the nearest mesh point be p, the charge accumulation follows Algorithm 3.1. However, this method creates a very rough function for sparse distributions, as shown in Figure 3.1.

Algorithm 3.1 NGP Charge Accumulation

for $i = 1$ to N **do**
  $p = \text{int}(x_i + 0.5)$
  $\rho_p = \rho_p - e$
end for

To smooth the assigned charge out, one can use a first-order correction, called cloud-in-cell (CIC) interpolation. For this, each particle's charge is divided between the two mesh points between which it lies. The division depends upon how close the particle is to each point. This time, the charge accumulation

follows Algorithm 3.2, where p and $p+1$ are the surrounding mesh points and the neutralizing effect of the stationary protons is included.

Algorithm 3.2 CIC Charge Accumulation

for $i = 1$ to N **do**
  $p = \text{int}(x_i)$
  $f = x_i - \text{real}(p)$
  $\rho_p = \rho_p - e(1-f)$
  $\rho_{p+1} = \rho_{p+1} - ef$
end for
for $p = 1$ to M **do**
  $\rho_p = \rho_p + \frac{Ne}{M}$
end for

Higher order corrections can be included, using elaborate charge allocation functions, but I found this amount of smoothing to be sufficient.

3.1.2 Poisson's Equation

Now that we are working on a grid, we can use a finite-difference equation instead of the continuous version of Poisson's equation (2.29c). To find the second-order FDA equation, we take Taylor series of the three points around the point of interest.

$$\phi_{p-1} = \phi_p - \Delta x \frac{\partial \phi}{\partial x} + \Delta x^2 \frac{\partial^2 \phi}{\partial x^2} + \mathcal{O}(\Delta x^3) \tag{3.1a}$$

$$\phi_p = \phi_p \tag{3.1b}$$

$$\phi_{p+1} = \phi_p + \Delta x \frac{\partial \phi}{\partial x} + \Delta x^2 \frac{\partial^2 \phi}{\partial x^2} + \mathcal{O}(\Delta x^3) \tag{3.1c}$$

$$\tag{3.1d}$$

where Δx is the separation between mesh points. If we weight them properly, after all three equations have been summed, the only remaining term will be the second derivative. Substituting this into Poisson's equation gives the sought-after result.

$$\phi_{p-1} - 2\phi_p + \phi_{p+1} = \Delta x^2 \frac{\partial^2 \phi}{\partial x^2} + \mathcal{O}(\Delta x^3) \tag{3.2}$$

$$= -\Delta x^2 \frac{\rho_p}{\epsilon_0} \tag{3.3}$$

There are M such finite-difference equations, one for each of the M mesh points. All the equations can together be expressed in matrix form.

$$\mathbf{D}^2 \vec{\phi} = -\frac{\Delta x^2}{\epsilon_0} \vec{\rho} \tag{3.4}$$

$$\begin{bmatrix} -2 & 1 & 0 & 0 & \dots & 0 & 1 \\ 1 & -2 & 1 & 0 & \dots & 0 & 0 \\ 0 & 1 & -2 & 1 & \dots & 0 & 0 \\ 0 & 0 & 1 & -2 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & -2 & 1 \\ 1 & 0 & 0 & 0 & \dots & 1 & -2 \end{bmatrix} \begin{Bmatrix} \phi_1 \\ \phi_2 \\ \phi_3 \\ \phi_4 \\ \vdots \\ \phi_{N-1} \\ \phi_N \end{Bmatrix} = -\frac{\Delta x^2}{\epsilon_0} \begin{Bmatrix} \rho_1 \\ \rho_2 \\ \rho_3 \\ \rho_4 \\ \vdots \\ \rho_{N-1} \\ \rho_N \end{Bmatrix}$$

However, this is not a tridiagonal matrix, because of the non-zero values in the top-right and bottom-left corner. These arise because equations for $i = 1$ and $i = M$ follow periodic boundary conditions. Therefore, the problem needs to be modified before canned software can be used to solve it.

We use the method described in [6]. $\mathbf{D}_*^2$ is defined to be the submatrix of $\mathbf{D^2}$ containing rows 1 to M-1 and columns 1 to M-1. Similarly, $\vec{\phi}_*$ and $\vec{\rho}_*$ are subvectors of $\vec{\phi}$ and $\vec{\rho}$ less only the M^{th} value. Also define $\vec{k_*}$ as $k_1 = k_{N-1} = 1$ and $k_i = 0$ for $i = 2..N-2$. Then the matrix equation becomes two coupled equations.

$$\mathbf{D}_*^2\vec{\phi}_* + \phi_N\vec{k_*} = \vec{\rho}_* \tag{3.5a}$$

$$\phi_{N-1} - 2\phi_N + \phi_1 = \rho_N \tag{3.5b}$$

$$\begin{bmatrix} -2 & 1 & 0 & 0 & \dots & 0 \\ 1 & -2 & 1 & 0 & \dots & 0 \\ 0 & 1 & -2 & 1 & \dots & 0 \\ 0 & 0 & 1 & -2 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \dots & -2 \end{bmatrix} \begin{Bmatrix} \phi_1 \\ \phi_2 \\ \phi_3 \\ \phi_4 \\ \vdots \\ \phi_{N-1} \end{Bmatrix} = -\frac{\Delta x^2}{\epsilon_0} \begin{Bmatrix} \rho_1 \\ \rho_2 \\ \rho_3 \\ \rho_4 \\ \vdots \\ \rho_{N-1} \end{Bmatrix} + \begin{Bmatrix} \phi_N \\ 0 \\ 0 \\ 0 \\ \vdots \\ \phi_N \end{Bmatrix}$$

This matrix equation is now tridiagonal, but it is not unique. Since the electric potential can vary by a constant, we can choose to define ϕ_N to be zero.

$$\begin{bmatrix} -2 & 1 & 0 & 0 & \dots & 0 \\ 1 & -2 & 1 & 0 & \dots & 0 \\ 0 & 1 & -2 & 1 & \dots & 0 \\ 0 & 0 & 1 & -2 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \dots & -2 \end{bmatrix} \begin{Bmatrix} \phi_1 \\ \phi_2 \\ \phi_3 \\ \phi_4 \\ \vdots \\ \phi_{N-1} \end{Bmatrix} = \frac{\Delta x^2}{\epsilon_0} \begin{Bmatrix} \rho_1 \\ \rho_2 \\ \rho_3 \\ \rho_4 \\ \vdots \\ \rho_{N-1} \end{Bmatrix} \tag{3.6a}$$

$$\phi_{N-1} + \phi_1 = \rho_N \tag{3.6b}$$

The separate equation (3.6b)is not required to solve the system. It can be used as a check that the matrix solver produced a consistent solution.

Once the electric potential has been found, it is a simple matter to find the electric field, and subsequently, the force on each particle. The discrete version of (2.29b) is

$$E_p = -\frac{\phi_{p+1} - \phi_{p-1}}{\Delta x}. \tag{3.7}$$

3.1.3 Moving the Particles

The particles are updated by using Newton's equations. The discrete versions that apply to the electrons are

$$\frac{v_i^{n+1/2} - v_i^{n-1/2}}{\Delta t} = \frac{e}{m_e} E(x_i^n) \tag{3.8}$$

$$\frac{x_i^{n+1} - x_i^n}{\Delta t} = v_i^{n+1/2} \tag{3.9}$$

where n is the number of the time step. The positions are recorded at whole time steps, while the velocities are recorded at half-time steps. This keeps each calculation symmetric in time, improving the stability, and is the reason for naming the scheme "leapfrog." In practice, the half-step does not affect the calculations, and is purely an entity of theory.

There is a difficulty, however. The electric field is only defined at the mesh points E_p, but it is needed at the particle positions $E(x_i^n)$, which are likely to lie in between the points. Therefore, an interpolation must be performed back into phase-space. This takes the same form as the interpolation to get the charge onto the mesh, as seen in Algorithm 3.3.

Algorithm 3.3 Update Particle Placement

for $i = 1$ to N **do**
  $p = \text{int}(x_i)$
  $f = x_i - \text{real}(p)$
  $v_i = v_i + \frac{e}{m_e}[E_p(1-f) + E_{p+1}f]\Delta t$
  $x_i = x_i + v_i \Delta t$
end for

After the particle coordinates have been updated, the boundary conditions are applied. If any particle has an x position greater than 1 or less than 0, it is adjusted to fall between them, while properly conforming to the periodic boundary conditions. The program will also check that no particle exceeds the maximum velocity. This completes a single time step of the simulation. At this point, diagnostics on the state of the particles are performed, and then the whole set of instructions is repeated.

3.1.4 Stability

It is important to be sure that any time integration scheme is not amplifying the truncation errors so that they eventually swamp the real physics. To ensure this does not happen, we look at whether the error incurred during a given step grows or decays in time. Combining (3.8) and (3.9) gives a function of only positions.

$$x^{n+1} - 2x^n + x^{n-1} = \frac{e}{m_e} E(x^n) \Delta t^2 \tag{3.10}$$

Let the exact solution at each time step be given by X^n, so that the error is

$$\epsilon^n = x^n - X^n \tag{3.11}$$

Substituting this into (3.10) gives

$$(X^{n+1} + \epsilon^{n+1}) - 2(X^n + \epsilon^n) + (X^{n-1} + \epsilon^{n-1}) = \frac{e}{m_e} E(X^n + \epsilon^n)\Delta t^2 \tag{3.12}$$

Taking the Taylor series of the electric field and collecting terms makes the equation

$$X^{n+1} - 2X^n + X^{n-1} + \epsilon^{n+1} - 2\epsilon^n + \epsilon^{n-1} = \frac{e}{m_e}\left(E(X^n) + \epsilon^n \left.\frac{\partial E}{\partial x}\right|_{x=X^n}\right)\Delta t^2 \tag{3.13}$$

Since (3.10) is most certainly true for its own exact solution, the terms that contain X can be subtracted out, which leaves

$$\epsilon^{n+1} - 2\epsilon^n + \epsilon^{n-1} = \frac{e}{m_e}\left(\epsilon^n \left.\frac{\partial E}{\partial x}\right|_{x=X^n}\right)\Delta t^2 \tag{3.14}$$

Since we are only interested in whether the error grows or decays, and not by how much, it is sufficient to replace the $\partial E/\partial x$ by its maximum amplitude value. We will choose the negative maximum, since the electric field is oscillatory.

$$\epsilon^{n+1} - 2\epsilon^n + \epsilon^{n-1} = -\epsilon^n \frac{e}{m_e}\left|\frac{\partial E}{\partial x}\right|_{max}\Delta t^2 \tag{3.15}$$

If we convert the derivative of the electric field to the charge density, we can see that in a completely uniform spatial distribution, the constants are equivalent to the plasma frequency squared.

$$\epsilon^{n+1} - 2\epsilon^n + \epsilon^{n-1} = -\epsilon^n \frac{e\rho_{max}}{m_e\epsilon_0}\Delta t^2 \tag{3.16}$$

However, since they are greater than ω_p^2 in all other cases, we will call them Ω^2. Dividing through by ϵ^{n-1} produces a quadratic equation.

$$\epsilon^2 - 2\epsilon + 1 = -\epsilon\Omega^2\Delta t^2 \tag{3.17}$$

The solution is

$$\epsilon_\pm = 1 - \frac{\Omega^2\Delta t^2}{2} \pm \frac{\Omega^2\Delta t^2}{2}\sqrt{1 - \frac{4}{\Omega^2\Delta t^2}} \tag{3.18}$$

Since the error is a linear combination of both values, we need the modulus of each to be less than or equal to one for there to be no growth in time. The condition for this is $\Omega\Delta t < 2$. For all my particle simulations, I use a time step of $\Delta t = 0.01$ seconds and a plasma frequency of $\omega_p < 5$. To break the stability condition, there would need to be an overdensity of at least $\Omega/\omega_p = 40$, which is unlikely.

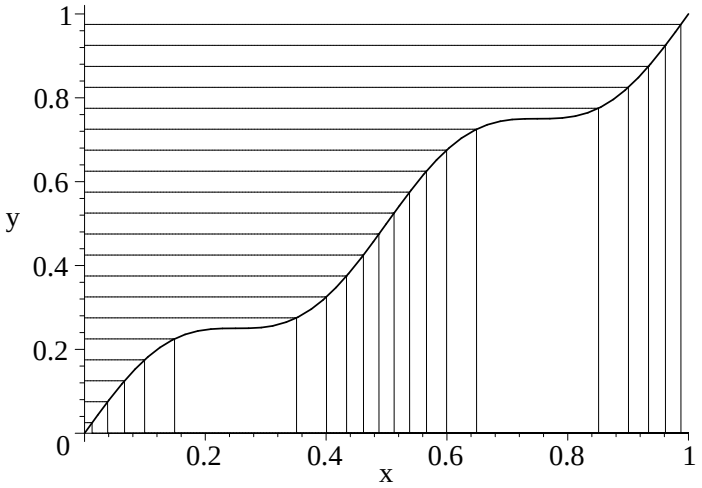


Figure 3.2: To generate particle positions with a sinusoidal distribution, the integral of the function is found. Then a uniform distribution in y will intersect the plot so that the density of the x points match the sinusoid. Here, the desired function is $f(x) = 1 + \cos(4\pi x)$.

3.1.5 Initialization

To properly distribute the particles, the positions in space are generated independently of the velocities. Then a particle is placed at each combination of coordinates. Given a distribution $f(x)$ in the range $x = 0..1$, the positions are given by the solution to

$$y = \int_0^x f(x')dx'. \tag{3.19}$$

where y is taken from a uniform distribution between 0 and 1, sampled either randomly or at equal intervals. To find the solutions, Newton's method is appropriate, since there is only one solution for each value of y. The basic method can be found in Algorithm 3.4. However, in practice, significant tweaking is required for each individual distribution function. As an example,

Algorithm 3.4 Compute points for $f(x)$ distribution on $x = 0..1$

Require: $\int_0^1 f(x)dx = 1$
  $x_{n+1} = y$
  while $|x_{n+1} - x_n| > \epsilon$ **do**
    $x_n = x_{n+1}$
    $x_{n+1} = x_n - (\int_0^{x_n} f(x)dx - y)/f(x_n)$
  end while

in all cases I studied, the placement in space is a sinusoidal distribution function

$f(x) = 1 + A\cos(2\pi kx)$. Its integral is plotted in Figure 3.2. One can see that a uniform set of y values will generate the appropriate distribution of x values.

There is a choice between two evils when generating the initial particle positions. One option is to place the particles randomly. However, when I tried a random sampling, there were not enough particles to remove the noise sufficiently. The extra fluctuations changed the amount of spatial perturbation I wished to start with, making it greater than specified. The alternative is to generate particles on a grid. This makes the electric potential exact, since it is an integral, but the initial charge density becomes jagged when it is close to zero. The velocity distribution is also discontinuous, actually being composed of a number of independent streams, and not truly the function specified. However, the latter method provided the desired global initial conditions more accurately, so it is implemented in the code.

3.2 Finite-Difference Approximation

The finite-difference simulation, rather than maintaining the particle picture of a plasma, makes use of the Vlasov equation (2.29a). The phase-space density function is recorded on a fixed grid, with each grid point holding an amount of "charge fluid." Advantages to this method over the PM method include the ability to move smalls amount of charge, rather than just integer amounts, as well as the ability to sample the enter domain, irrespective of where the action is taking place. It does not, however, scale as well when the number of dimensions is increased.

The operations that take place in the FDA simulation are similar to those in the PM simulation. In fact, steps (2) and (3) are identical. In a single time step, the computer performs the following operations.

1. Sum phase-space density over velocity space to get spatial charge density.
2. Use Poisson's equation to calculate the electric potential.
3. Find the electric field.
4. Update the phase-space density with the Vlasov equation.

Since we have the phase-space density specified at each point on a 2-D mesh $f^n_{i,j}$, the first operation is easy to perform. It is just the discrete form of (2.29d) and (2.29e).

$$\rho_i = -e\sum_{j=1}^{N_v} f^n_{i,j}\Delta x + \rho_0 \tag{3.20a}$$

$$\rho_0 = e\sum_{i=1}^{N_x}\sum_{j=1}^{N_v} f^n_{i,j}\Delta x\Delta v \tag{3.20b}$$

where $i = 1..N_x$ is the index and Δx the step size of the x dimension discretization, and $j = 1..N_v$ is the index and Δv the step size of the v dimension discretization. Now, calculating Poisson's equation and the electric field follows exactly the form discussed for the PM case in Section 3.1.2.

Once the electric field is known, only one step remains to complete the time step, updating the phase-space density function with the Vlasov equation. I tried two different methods to create a finite-difference equation for (2.29a), the Crank-Nicholson scheme and the Leapfrog scheme.

3.2.1 Crank-Nicholson Scheme

The second-order Crank-Nicholson equation for the Vlasov equation is centered in time by making it depend implicitly upon its own solutions. There is a picture of the stencil used to calculate each point in Figure 3.3a.

$$\frac{f_{i,j}^{n+1} - f_{i,j}^{n}}{\Delta t} + \frac{1}{2}\left[v_j \frac{f_{i+1,j}^{n} - f_{i-1,j}^{n}}{2\Delta x} + v_j \frac{f_{i+1,j}^{n+1} - f_{i-1,j}^{n+1}}{2\Delta x} + \frac{eE_i}{m_e} \frac{f_{i,j+1}^{n} - f_{i,j-1}^{n}}{2\Delta v} + \frac{eE_i}{m_e} \frac{f_{i,j+1}^{n+1} - f_{i,j-1}^{n+1}}{2\Delta v} \right] = 0 \tag{3.21}$$

To enforce the spatially periodic boundary conditions, we identify $f_{N_x+1,j}^{n} = f_{1,j}^{n}$. The velocity boundary conditions are set to zero at a specified maximum velocity, since it is statistically probable that the energy is equally partitioned among the particles in the plasma.

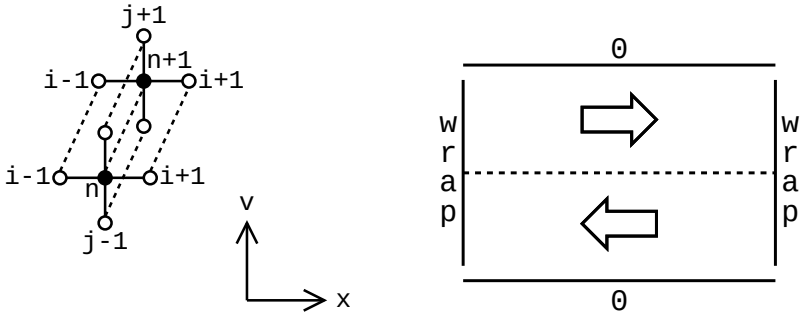
$$f_{i,1}^{n} = f_{i,N_v}^{n} = 0 \tag{3.22}$$

These are shown in Figure 3.3b.

To solve the equation for each point on the grid, we employ Gauss-Seidel relaxation. First we assume that solution at the advanced time step is close to the current solution, $f_{i,j}^{n+1} = f_{i,j}^{n}$. Then we cycle over the whole grid, solving each point based on the adjacent points, even though their values are not yet correct. The adjustment δ applied to each point at each iteration k will bring it closer to its correct solution. Then it will provide a more accurate basis to solve for its neighbouring points. The adjustment is found just like in Newton's method; it is the Vlasov finite-difference equation (3.21) divided by the derivative with respect to $f_{i,j}^{n}$.

$$\delta = \frac{\frac{f_{i,j}^{n+1} - f_{i,j}^{n}}{\Delta t} + \frac{1}{2}\left[v_j \frac{f_{i+1,j}^{n} - f_{i-1,j}^{n}}{2\Delta x} + v_j \frac{f_{i+1,j}^{n+1} - f_{i-1,j}^{n+1}}{2\Delta x} + \frac{eE_i}{m_e} \frac{f_{i,j+1}^{n} - f_{i,j-1}^{n}}{2\Delta v} + \frac{eE_i}{m_e} \frac{f_{i,j+1}^{n+1} - f_{i,j-1}^{n+1}}{2\Delta v} \right]}{\frac{1}{\Delta t} + \frac{v_j}{4\Delta x} + \frac{a_i}{4\Delta v}}. \tag{3.23}$$

When it falls below a given threshold, the process is concluded. This is summarized in Algorithm 3.5.



(a) (b)

Figure 3.3: (a) The Crank-Nicholson stencil and (b) the domain and boundary conditions for the FDA simulation. The dotted line indicates zero velocity, and the white arrows indicate the direction of flow in phase space.

Algorithm 3.5 Gauss-Seidel Relaxation for Vlasov equation

$f^{n+1} = f^n$
repeat
  $\delta_{max} = 0$
  for $i = 1$ to N_x **do**
    for $j = 1$ to N_v **do**
      $\delta = F(f)/F'(f)$
      $f_{i,j}^{n+1} = f_{i,j}^n - \delta$
      if $|\delta| > \delta_{max}$ **then**
        $\delta_{max} = |\delta|$
      end if
    end for
  end for
until $\delta_{max} < \epsilon$

Using von Neumann's method to analyze the stability of this scheme, we assume that the error takes the form

$$\epsilon_{i,j}^{n} = \zeta^{n} e^{Ik\Delta x i} e^{Ih\Delta v j} \tag{3.24}$$

where $I \equiv \sqrt{-1}$ to avoid confusion with the indices. For the error to not grow in time, the modulus of ζ must be less than or equal to 1. Plugging $\epsilon_{I,J}^{n}$ into the Vlasov equation (3.21) and then dividing by it gives

$$\frac{\zeta - 1}{\Delta t} + \frac{I}{2}(\zeta + 1)\left[\frac{v_j}{\Delta x} sin(k\Delta x) + \frac{eE_i}{m_e \Delta v} sin(h\Delta v)\right] = 0 \tag{3.25}$$

Collecting the ζ terms suggests a substitution

$$\frac{\zeta - 1}{\zeta + 1} = -\frac{I\Delta t}{2}\left[\frac{v_j}{\Delta x} sin(k\Delta x) + \frac{eE_i}{m_e \Delta v} sin(h\Delta v)\right] \equiv IG(\Delta x, \Delta v, \Delta t) \tag{3.26}$$

Solving for zeta gives

$$\begin{aligned} \zeta &= \frac{1 + IG}{1 - IG} = \frac{(1 + IG)^2}{1 + G^2} \\ |\zeta|^2 &= \frac{(1 - G^2)^2 - 4G^2}{(1 + G^2)^2} = \frac{(1 + G^2)^2}{(1 + G^2)^2} = 1 \end{aligned} \tag{3.27}$$

Since the modulus is always 1, the Crank-Nicholson scheme is unconditionally stable.

3.2.2 Leapfrog Scheme

An alternative to the Crank-Nicholson scheme is the Leapfrog scheme. To achieve a centered time derivative, the $n - 1$ time step is kept in memory. This is much simpler, as the value of $f_{i,j}^{n}$ is given explicitly and, therefore, no iterative behaviour is required.

$$\frac{f_{i,j}^{n+1} - f_{i,j}^{n-1}}{2\Delta t} + v_j \frac{f_{i+1,j}^{n} - f_{i-1,j}^{n}}{2\Delta x} + \frac{eE_i}{m_e} \frac{f_{i,j+1}^{n} - f_{i,j-1}^{n}}{2\Delta v} = 0 \tag{3.28}$$

Applying the von Neumann stability analysis to this case, it is apparent that the equation for error propagation is

$$\zeta - \zeta^{-1} = -\frac{\Delta t}{\Delta x} v_j 2I \sin(k\Delta x) - \frac{\Delta t}{\Delta v} \frac{eE_i}{m_e} 2I \sin(k\Delta x) \equiv IG(\Delta x, \Delta v, \Delta t) \tag{3.29}$$

Solving for ζ produces the equation

$$\zeta = -I\frac{G}{2} \pm \sqrt{1 - \left(\frac{G}{2}\right)^2} \tag{3.30}$$

When the value inside the square root is positive, then $|G| \leq 2$ and the modulus of ζ is

$$|\zeta|^2 = \left(\frac{G}{2}\right)^2 + 1 - \left(\frac{G}{2}\right)^2 = 1. \tag{3.31}$$

Otherwise, when $|G| > 2$, the modulus of ζ is

$$|\zeta_\pm|^2 = \left(\frac{G}{2}\right)^2 + \left(\frac{G}{2}\right)^2 - 1 \pm 2\frac{G}{2}\sqrt{\left(\frac{G}{2} - 1\right)^2} \tag{3.32}$$

$$|\zeta_+|^2 > \frac{G^2}{2} - 1 > 1 \tag{3.33}$$

Since $|\zeta_+|$ is always greater than 1, the condition for stability is that $|G| \leq 2$. This means that the leapfrog scheme is only stable when

$$\left|\frac{\Delta t}{\Delta x} v_j 2\sin(k\Delta x) + \frac{\Delta t}{\Delta v}\frac{eE_i}{m_e} 2\sin(k\Delta x)\right| \leq 2$$

$$\frac{\Delta t}{\Delta x} v_j + \frac{\Delta t}{\Delta v}\frac{eE_i}{m_e} \leq 1. \tag{3.34}$$

One must include error checking in all leapfrog simulations to ensure that this condition is not broken, otherwise the results will be meaningless.

Despite the fact that the Crank-Nicholson scheme is unconditionally stable and requires less memory than the leapfrog scheme, it is much slower than the leapfrog scheme. For these simulations, the memory required is quite low, so an increase of 50% is not significant. But the Gauss-Seidel method must iterate about 6 or 7 times per time step, which is many times longer than the single iteration leapfrog. I found no difference in the solutions, so I chose to use the leapfrog method in my program.

3.3 Physical Diagnostics

It is appropriate to discuss the discretized equations for the properties of interest listed in Section 2.6. However, most of them are trivial conversions. For instance, the speed distribution is

$$f(v,t) = \int_0^\infty f(x,v,t) + f(x,-v,t)dx \rightarrow f_j^n = \sum_{i=0}^{N_x} (f_{i,j_0+j}^n + f_{i,j_0-j}^n)\Delta x$$

where j_0 is the row of $v = 0$.

There is one set of equations that do need a short explanation: those that give the frequency dependence of the energy. Performing the discrete Fourier transform in the continuous regime uses complex exponentials, but they need to be recast as sinusoidal functions for the numerical transform. The equations

for the charge density, electric potential, and electrostatic energy are as follows:

$$\rho_k = 2\sqrt{\left\{\sum_{i=1}^{N_x} \rho_i \cos(2\pi k \Delta x i)\right\}^2 + \left\{\sum_{i=1}^{N_x} \rho_i \sin(2\pi k \Delta x i)\right\}^2} \tag{3.35}$$

$$\phi_k = 2\sqrt{\left\{\sum_{i=1}^{N_x} \phi_i \cos(2\pi k \Delta x i)\right\}^2 + \left\{\sum_{i=1}^{N_x} \phi_i \sin(2\pi k \Delta x i)\right\}^2} \tag{3.36}$$

$$E_k = \frac{1}{4}\rho_k \phi_k \tag{3.37}$$

I only record histories for the first 8 modes in my programs.

3.4 Convergence Testing

With all these numerical approximations, we need to be sure that the solution we are getting converges as we increase the resolution. Presumably, if the programming is correct, when the spacing between the grid points reaches zero, the continuous solution should be reached. Since no computer would ever be able to do this, we need to check that at least some solution is being approached as the grid spacing is reduced. If the behaviour changes drastically with different resolutions, there is something wrong.

According to [7] as described in Matt's FDA manual, the solutions of second-order FDAs take the form

$$u^h(x,t) = u(x,t) + h^2 e_2(x,t) + h^4 e_4(x,t) + \dots \tag{3.38}$$

where h is the parameter that scales the grid, $u(x,t)$ is the continuum solution, and the $e_n(x,t)$ are unknown error functions. To make use of this, we calculate the solution to our FDA at three different levels, so that each point of the lowest level matches every second point of the middle level and every fourth point of the highest level. Then we have

$$\text{low} \qquad u^{4h}(x,t) = u(x,t) + (4h)^2 e_2(x,t) + (4h)^4 e_4(x,t) + \dots \tag{3.39}$$

$$\text{mid} \qquad u^{2h}(x,t) = u(x,t) + (2h)^2 e_2(x,t) + (2h)^4 e_4(x,t) + \dots \tag{3.40}$$

$$\text{high} \qquad u^{h}(x,t) = u(x,t) + h^2 e_2(x,t) + h^4 e_4(x,t) + \dots \tag{3.41}$$

If we calculate the difference between the first two and the second two, a course of action suggests itself.

$$|u^{4h}(x,t) - u^{2h}(x,t)| = 12h^2|e_2(x,t)| + 48h^4|e_4(x,t)| + \dots \tag{3.42}$$

$$|u^{2h}(x,t) - u^{h}(x,t)| = 3h^2|e_2(x,t)| + 15h^4|e_4(x,t)| + \dots \tag{3.43}$$

If we define the convergence factor as (3.42) divided by (3.43), then this value should approach 4 as $h \to 0$ and the higher terms disappear.

$$Q(t) = \frac{|u^{4h}(x,t) - u^{2h}(x,t)|}{|u^{2h}(x,t) - u^{h}(x,t)|} \tag{3.44}$$

Of course, $u^{4h}(x,t), u^{2h}(x,t)$, and $u^h(x,t)$ are not scalar values. They include points spanning a space, so their norm must be properly defined. One possibility is the l_2 norm.

$$|u^h(x,t)| = \left(J^{-1} \sum_{j=1}^{J} [u^h(x,t)]^2 \right)^{1/2} \tag{3.45}$$

Chapter 4

Projects

4.1 Two-Particle Test

The first project is a test to see whether the particle-mesh simulation behaves properly for a simple case. We place two stationary electrons at positions $x_1 = 0.4$ and $x_2 = 0.6$. Since there are periodic boundary conditions, the left electron will see an infinite series of other electrons at distances $x = x_2 - x_1 + n$ and $x = n$. Of course, its own mirror images will have no effect because the copies on each side cancel. But when $x_2 - x_1 < 0.5$, the first electron is closer to the second electron on the right, so it is repelled in the negative x direction. Eventually, $x_2 - x_1 > 0.5$, and it will be closer to the second electron on the left, so it is repelled back in the positive direction. Therefore, there should be sinusoidal oscillation.

The plasma frequency ω_p is defined as the frequency that electrons oscillate past stationary protons. This is exactly the case we are testing here, and therefore, by setting $\omega_p = 2\pi$, we expect to see a period of 1 (Figure 4.1a). This is exactly what occurs and one can also see that the electric potential and electric field have exactly the form expected (Figure 4.1c+d). The potential has peaks at the particle positions and slopes off as r^2, while the electric field is linear with discontinuities at the particle positions.

While changing the time step, the energy conservation can be seen to be linear. This is not as good as the quadratic convergence that the leapfrog scheme claims, but it is sufficient for our needs. These results fill us with confidence and we proceed to a more complicated problem.

4.2 Maxwellian Distribution

When observing the generated velocity distribution (Figure 4.2a+b), it is apparent that the FDA approach has significant advantages. The PM distribution exhibits discontinuities where the particle density is low, while the FDA distribution remains smooth everywhere. For this reason, we will often defer to the FDA data, as it provides cleaner results.

Linear theory indicates that the Landau damping of the electric field will impart energy to the particles near the phase-velocity. When subtracting the initial velocity distribution from the distribution at early times (Figure 4.2c), there is an obvious node at the phase-velocity. What is surprising is how quickly the situation becomes non-linear.

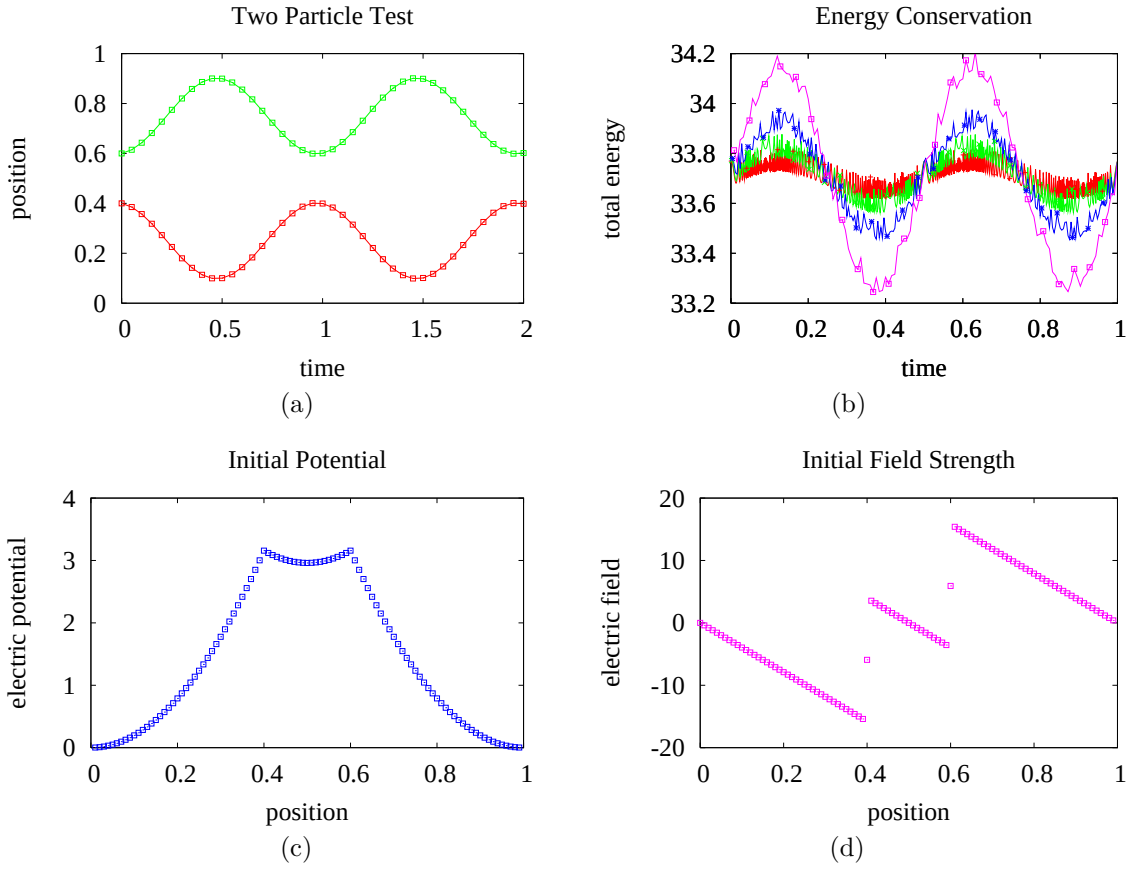


Figure 4.1: (a) Particle positions over two periods of the two particle test. (b) Energy conservation for time steps of 0.001 [r], 0.002 [g], 0.004 [b], and 0.008 [p]. (c) The potential and (d) the electric field at $t = 0$.

At $t = 0.3$, the phase-velocity is very close to its initial value, but by $t = 0.9$ it has shifted outwards and a new, unstable phase-velocity has appeared. At $t = 1.7$, both of these have shifted outwards and another stable phase-velocity appears. New phase-velocities are created at a rate of $\Delta t \approx 0.9$ at first. This number does drop as time progresses, but very slowly. Eventually, the velocity distribution looks like a terraced Maxwellian. Unfortunately, this effect is rather small (note the vertical scales), so it is only readily observable on the difference graph (Figure 4.2d).

Going back to comparing the simulation schemes, an interesting test is how they handle progressively lower perturbation amplitudes. While the FDA program is limited only by the size of the subdivisions of a double floating-point variable, the PM program must deal with integer sized particles. One can see in Figure 4.3a that the FDA program runs small amplitude perturbations flawlessly; there is no change in the initial damping rate between a perturbation of 0.1 and one of 10^{-8}.

The same cannot be said of the PM program Figure 4.3b. Indeed, one questions whether any damping is occurring at all. What is the worst part is that an initial perturbation of only 10^{-3} appears to hit the limit of machine epsilon; near $t = 0$, the blue points (10^{-3}) inhabit the same region as the gray points, which indicate no perturbation at all.

Let us look, instead, at the computed convergence for each program. The convergence factor for the FDA scheme is quadratic (4), as promised, although it does begin to drop as time progresses (Figure 4.3c). On the other hand, the PM scheme appears to have a convergence factor midway between linear and quadratic, which eventually drops to linear (Figure 4.3d). However, scaling the particle number up raises it rapidly and the period of high convergence lasts longer, so one expects that the scheme is indeed quadratic. Notice also that the time scale on the PM graph is significantly longer than that on the FDA graph, so the FDA convergence may exhibit the similar poor behaviour at later times.

A complete list of damping rates and oscillation frequencies for the Maxwellian tests can be found in Tables A.1 and A.2. The plasma frequency ω_p, thermal velocity v_t, and perturbation mode k were all varied. Despite the results from the PM scheme being difficult to analyze, there was still excellent agreement between both programs. However, the results were not corroborated by the theory. The theory indicates oscillation rates almost identical to the plasma frequency and infinitesimal Landau damping. However, both programs produced oscillation frequencies larger than the plasma frequency and significant Landau damping.

Since both programs agree with each other within error and give the same visual results as a third-party program, I suspect a simple change of the theory is all that is required. The variation in the results while changing the parameters exhibited the same behaviour as the theory. There may be a constant of proportionality that needs to be included with some of the parameters.

There is an interesting effect that I noticed in all damped trials of the FDA simulation. If the electrostatic energy is analyzed by Fourier modes (Figure 4.4), then at early times, only the damped perturbation mode k contributes

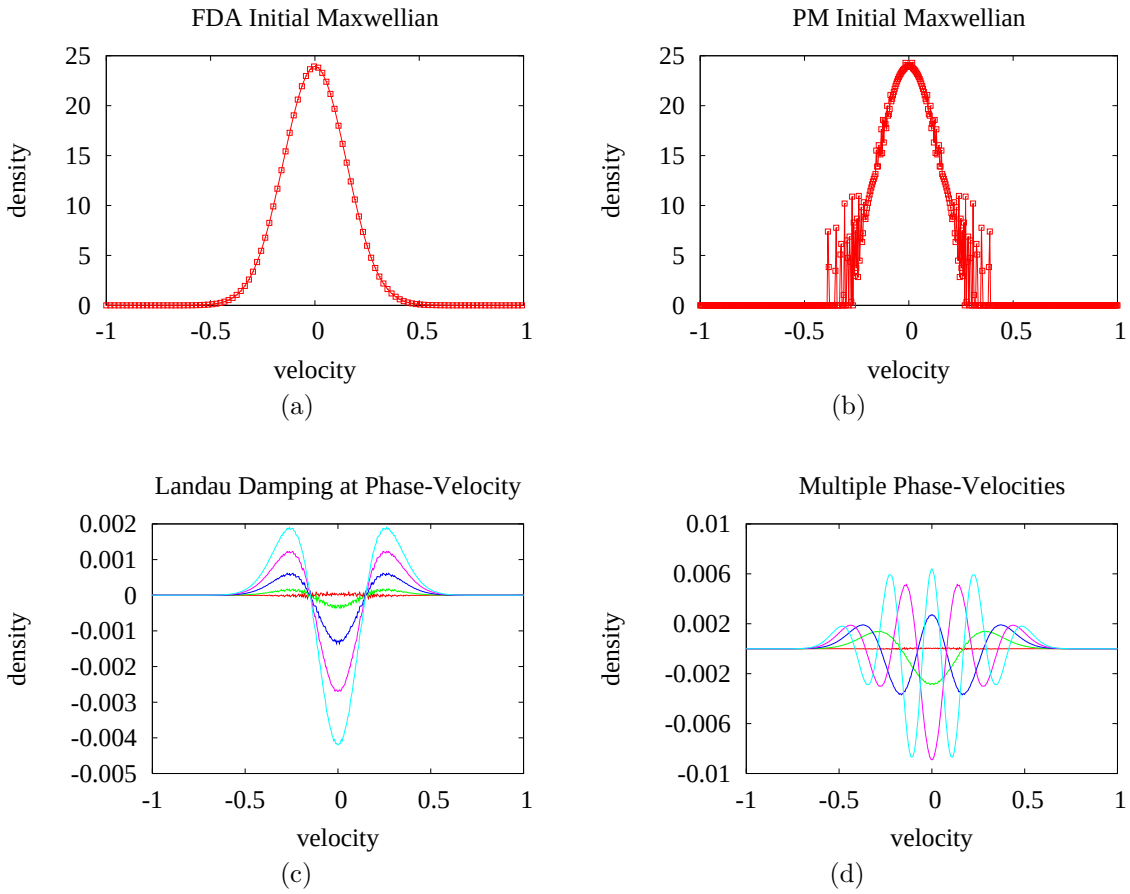


Figure 4.2: Comparison of the initial velocity distributions in (a) the FDA simulation and (b) the PM simulation. (c) Initial flattening occurs at $v = 0.15$, the phase-velocity. Times shown are $t = 0$ [r], 0.8 [g], 0.16 [b], 0.23 [p], 0.31 [a]. (d) Later, multiple flattening points appear. Times shown are $t = 0$ [r], 7.8 [g], 1.6 [b], 2.3 [p], 3.1 [a].

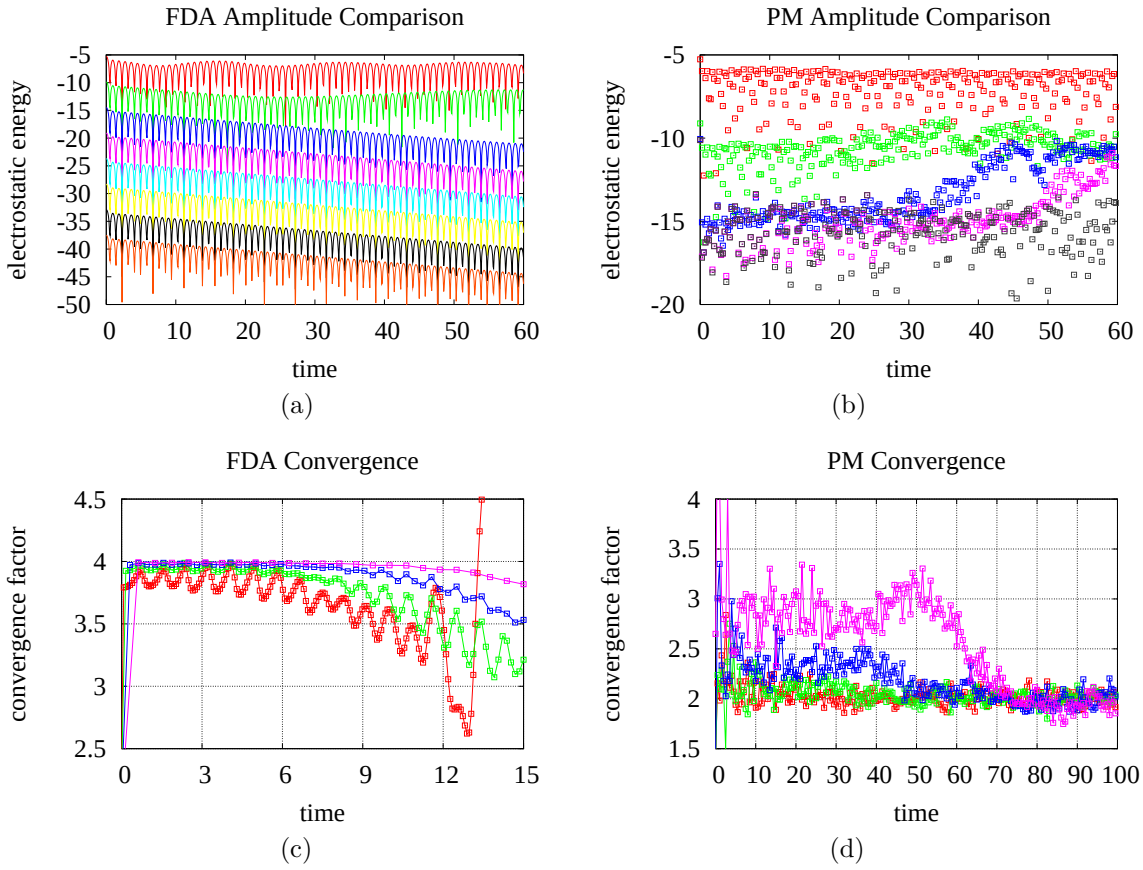


Figure 4.3: Different magnitude initial perturbations in (a) the FDA simulation and (b) the PM simulation with a Maxwellian velocity distribution. The colours in both graphs correspond, except the gray points in the PM plot indicate no perturbation. (c) Convergence factors between grid sizes of $(N_x, N_v) = (16,32)$, (32,64), (64,128) [r], (128,256) [g], (256,512) [b], and (512,1024) [p]. (d) Convergence factors between particle numbers of 2.5k, 10k [r], 40k [g], 160k [b], and 640k [p].

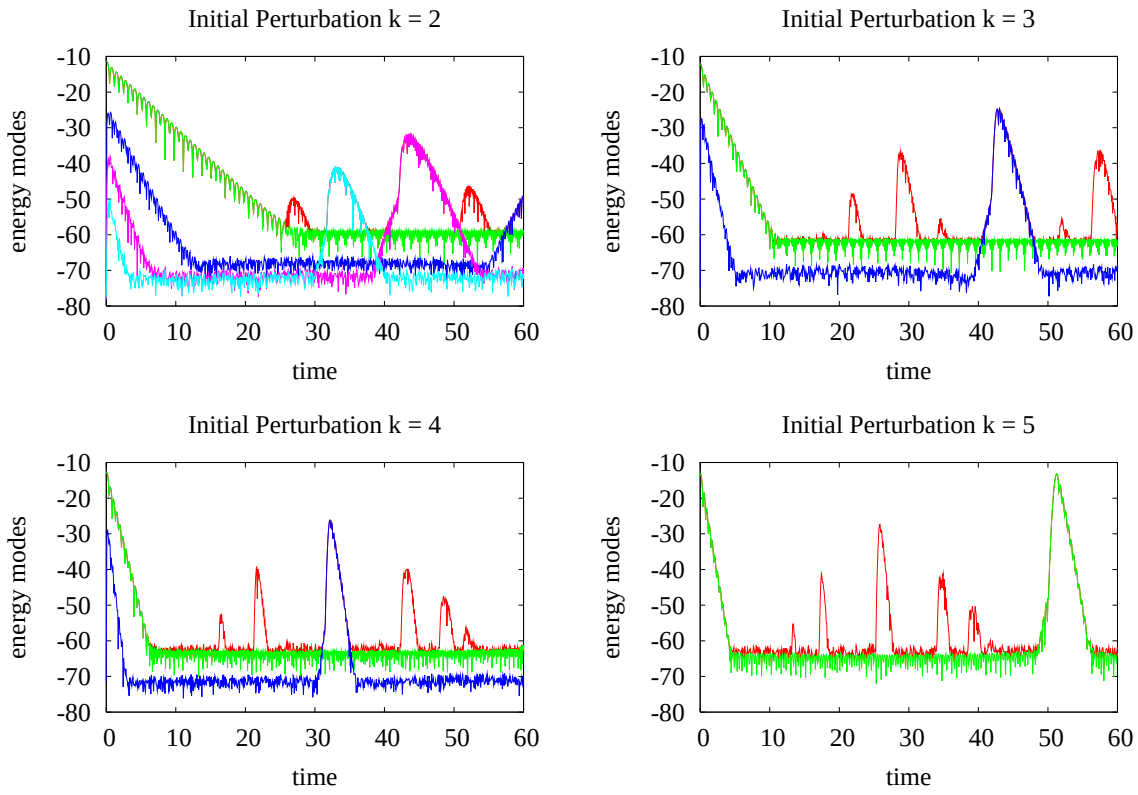


Figure 4.4: Electrostatic energy mode logplots for different modes of initial perturbation k in the FDA simulation with a Maxwellian velocity distribution. The red line in each is the total energy. The energy with mode k is green, $2k$ is blue, $3k$ is purple, and $4k$ is aqua.

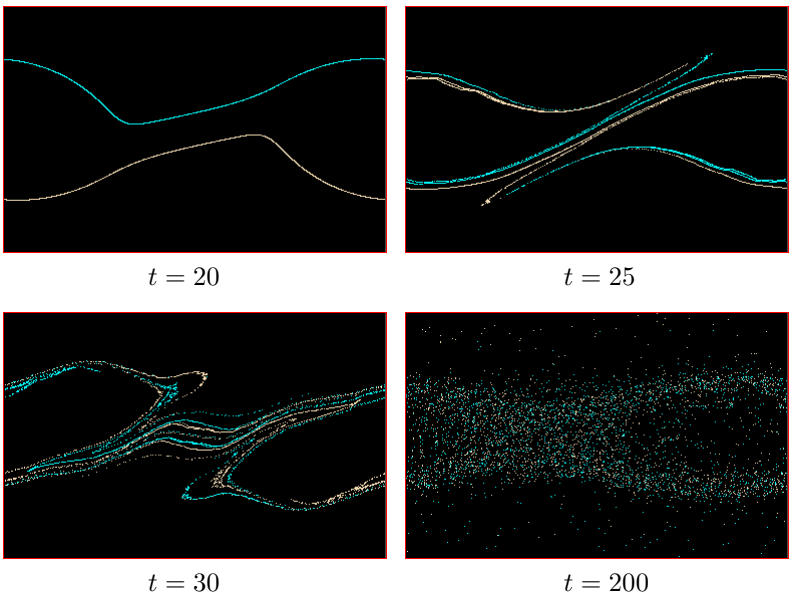


Figure 4.5: Phase-space plots of the particles in a cold two-stream distribution. Images from ES1 [2].

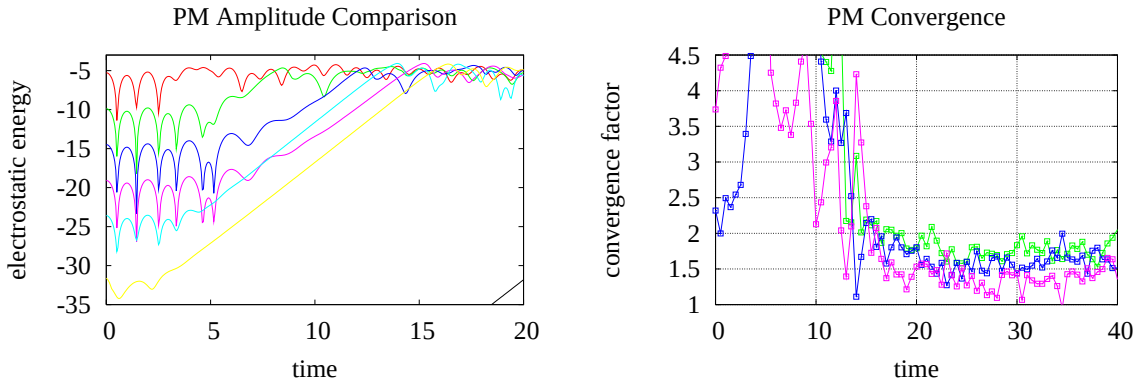
significantly. Multiples of k also appear, and it is interesting to see how they are damped at a rate proportional to their individual modes. Eventually, the damping ceases, but the electrostatic energy does not remain constant. It exhibits spikes as certain modes that are multiples of k are suddenly given a lot of energy. The spiking occurs at a frequency proportional to the mode. This is obviously an artifact of the simulation, and not physical, but I am curious to investigate its cause.

4.3 Two-Stream Instability

4.3.1 Cold Streams

The cold two-stream velocity distribution starts with two delta function streams. As time progresses, the streams will wrap around each other with a 'kneading' action, eventually forming vortices (Figure 4.5). Because of the discontinuous initial conditions, it cannot be simulated using the FDA program; finite-difference techniques require smooth functions to be reliable. This made it difficult to study, since the PM simulation produced noiser data, and there was no corroboration possible between programs.

Unlike the Maxwellian, one can see that varying the amplitude of the initial perturbation is consistent, although the rate of growth is not completely identical in all cases (Figure 4.6a). There does not appear to be a point where



(a) (b)

Figure 4.6: (a) Different magnitude initial perturbation in the cold two-stream velocity distribution. The black line in the bottom-right is the no perturbation solution. It intersects the vertical axis at 10^{-70}. (b) Convergence factors between particle numbers of 2.5k, 10k, 40k [g], 160k [b], and 640k [p].

decreasing the perturbation does not lower the initial electrostatic energy. This cannot be completely true, however, since initial conditions without any perturbation will eventually become unstable. Round-off errors incurred while moving the particles around seed the instability. The initial electrostatic energy of the zero perturbation case is about 10^{-70}. Fortunately, this lower limit is well below anything we would bother to simulate.

The convergence of this distribution is worse than the Maxwellian. The time before the convergence factor falls off is shorter, and it stabilizes below 2 (Figure 4.6b). I suspect this is because of the kneading that causes filamentation in velocity space. Also, if the vortices do not form in exactly the same place, the convergence will be low, despite the behaviour being identical.

To compound the problem of poor convergence, the data is very dissimilar to the theory (Table A.3). Furthermore, the theory indicates that varying parameters does not produce monotonic results, so that a missing constant of proportionality would change the expected values drastically.

4.3.2 Warm Streams

The warm two-stream velocity distribution has features that make it very difficult to measure. Its behaviour is extremely dependent on the initial conditions. Sometimes, after the transients disappear from the electrostatic energy, the remainder of the plot has no oscillations. The instability growth rate is then very easy to measure, but the oscillation frequency is impossible to find. Alternatively, one would expect that if the two Lorentzians were very wide and close,

then the distribution should act like the Maxwellian. However, this only occurs when $a \approx b$ (Figure 4.7a). If $b > a$, then the electrostatic energy quickly settles to a constant value and doesn't oscillate at all.

The amplitude comparison for the warm two-stream distribution is similar to the Maxwellian case (Figure 4.7a+b), with the FDA handling all magnitudes equally well, but the PM hitting machine error at 10^{-3}. The convergence of the PM simulation is midway between the Maxwellian and the cold streams. However, the FDA convergence is very bad, with the highest resolution levels always below 1. This causes me to suspect that unstable initial conditions are slightly chaotic; even if a minute change produces a solution with identical behaviour, it might be in a different place.

I have analyzed some data from this case, which appears in Table A.4, but because of its erratic behaviour, I was unable to make many conclusions. There is corroboration between programs when the instability exists, but there is disagreement in the stable case. Since the known cases, the Maxwellian and the cold two-stream distribution, did not match their theory exactly, I am currently unable to confirm which theory for the warm two-strema distribution with my data. Fortunately, I was able to determine that the transition point of the instability occurs when a is close to b, and therefore, that (2.66) is likely correct. This case requires further simulation to find initial parameters that give clearer results.

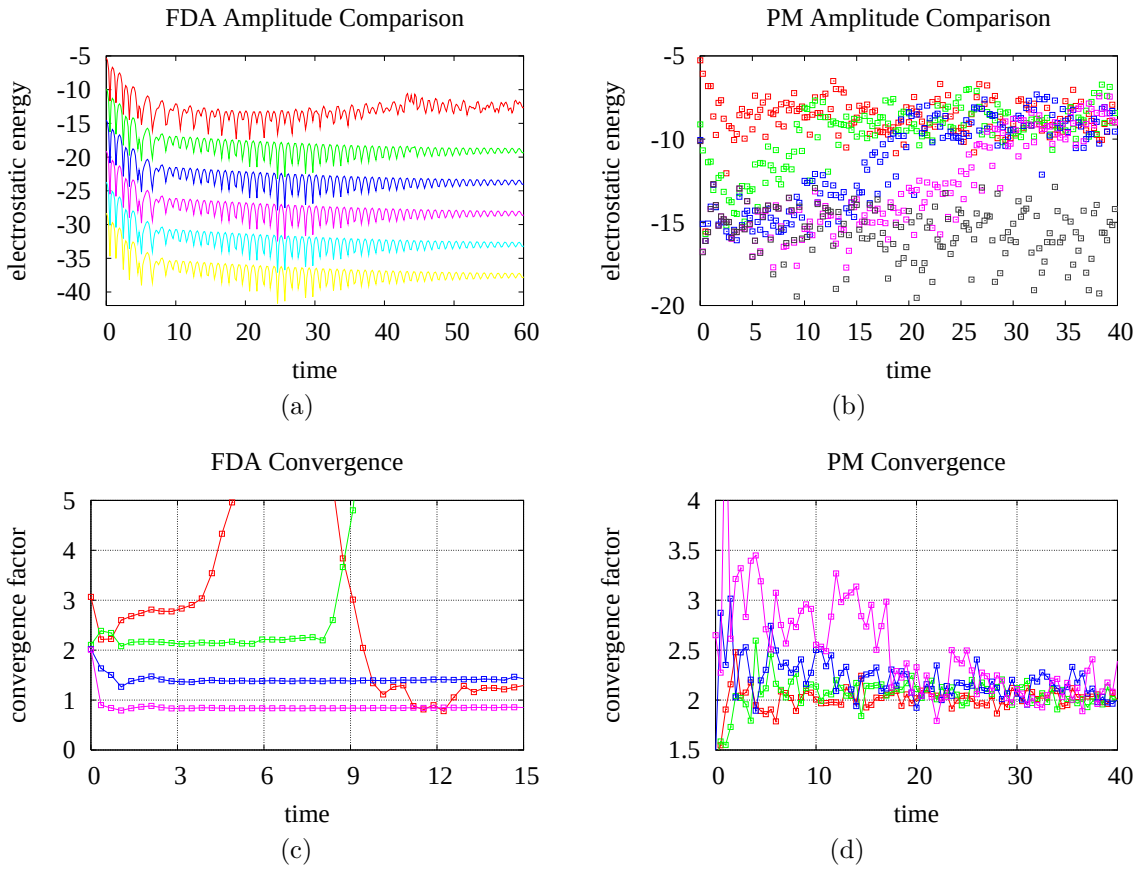


Figure 4.7: Different magnitude initial perturbation amplitudes in (a) the FDA simulation and (b) the PM simulation with a warm two-stream velocity distribution. The colours in both graphs correspond, except the gray points in the PM plot indicate no perturbation. (c) Convergence factors between grid sizes of (N_x, N_v) = (16,32), (32,64), (64,128) [r], (128,256) [g], (256,512) [b], and (512,1024) [p]. (d) Convergence factors between particle numbers of 625, 2500, 10k [r], 40k [g], 160k [b], and 640k [p].

Chapter 5

Conclusion

Plasma exhibits a complex set of behaviours that are usually not measurable in physical experiments. Therefore, one must turn to computer simulations to elucidate what is occurring inside. Even in one dimension, I experienced how complicated a 'simple' four-equation model can be.

I created two programs, one running a particle-mesh simulation and the other a finite-difference approximation, to emulate the interaction of charged electrons in a neutralizing background. In initial tests, both programs visually corroborated the results of third-party programs. They also performed well on the simple two particle test.

Three initial velocity distributions were used to generate data: the Maxwellian, the cold two-stream distribution (two delta functions), and the warm two-stream distribution (two Lorentzians). The Maxwellian data exhibited flattening near the phase-velocity as described by theory. It also showed how the nonlinear behaviour causes the creation of new phase-velocities. In this case, both programs agreed quantitatively with each other and were qualitatively similar to the theory.

Both two-stream distributions produced qualitatively correct behaviour. However, in each case, there were only two sets of data to compare. For the cold streams, only the PM simulation could be used, and like the Maxwellian, the theory was not well matched to the results. The warm streams are undocumented, and so, were supposed to be a test of new theory. However, the results were unclear and the only conclusion that could be drawn is that the perturbation disappears near the point where the separation of the Lorentzian peaks is twice the width of each one.

Despite the inconclusive evidence derived from the programs, I did succeed in my objective to become more experienced at simulating plasma. I have learned about Landau damping and seen the effect of different parameters on my results. And, most importantly, I have discovered that FDA simulations are easier to implement and yield much cleaner data than particle simulations, and when the dimensionality of the problem is low, FDA programs run at a speed competitive with their PM counterparts.

Bibliography

[1] D. Anderson, R. Fedele, and M. Lisak. A tutorial presentation of the two stream instability and Landau damping. *American Journal of Physics*, 69(12):1262, 2001.

[2] Charles K. Birdsall and A. Bruce Langdon. *Plasma Physics Via Computer Simulation*. McGraw-Hill, New York, 1985.

[3] Francis F. Chen. *Introduction to Plasma Physics and Controlled Fusion*, volume 1. Plenum Press, New York, 1984.

[4] Robert J. Goldston and Paul H. Rutherford. *Introduction to Plasma Physics*. Institute of Physics Publishing, London, 1995.

[5] Roger W. Hockney and James W. Eastwood. *Computer Simulation Using Particles*. McGraw-Hill, New York, 1981.

[6] A. Mangeney, F. Califano, C. Cavazzoni, and P. Travnicek. A Numerical Scheme for the Integration of the Vlasov-Maxwell System of Equations. *Journal of Computational Physics*, 179:495, 2003.

[7] L. F. Richardson. The Approximate Arithmetical Solution by Finite Differences of Physical Problems involving Differential Equations, with an Application to the Stresses in a Masonry Dam. *Phil. Trans. Roy. Soc.*, 210:307, 1910.

[8] Marc Spiegelman. Myths and methods in modeling. http://www.ldeo.columbia.edu/ mspieg/mmm/, 2000.

Appendix A

Tabulated Project Results

Table A.1: Oscillation frequency for Maxwellian velocity distribution

ω_p	v_t	k	Re(ω_{PM})	Re(ω_{FDA})	Re(ω_{theory})
1.0	0.15	1	2.0 ± 0.1	1.936 ± 0.001	1.034
1.5	0.15	1	2.4 ± 0.1	2.338 ± 0.001	1.534
2.0	0.15	1	2.8 ± 0.1	2.722 ± 0.001	2.034
2.5	0.15	1	3.1 ± 0.1	3.109 ± 0.001	2.534
3.0	0.15	1	3.47 ± 0.01	3.512 ± 0.001	3.034
3.5	0.15	1	3.90 ± 0.01	3.93 ± 0.005	3.534
4.0	0.15	1	4.35 ± 0.01	4.37 ± 0.01	4.034
4.5	0.15	1	4.80 ± 0.01	4.82 ± 0.01	4.534
5.0	0.15	1	5.28 ± 0.01	5.28 ± 0.01	5.034
3.0	0.05	1	3.05 ± 0.01	3.050 ± 0.001	3.004
3.0	0.10	1	3.20 ± 0.01	3.212 ± 0.001	3.015
3.0	0.15	1	3.47 ± 0.01	3.512 ± 0.001	3.034
3.0	0.20	1	3.9 ± 0.1	3.884 ± 0.001	3.060
3.0	0.25	1	4.5 ± 0.1	4.29 ± 0.01	3.094
3.0	0.30	1	4.8 ± 0.2	4.74 ± 0.01	3.135
3.0	0.15	1	not clear	3.05 ± 0.01	3.034
3.0	0.15	2	not clear	4.71 ± 0.01	3.135
3.0	0.15	3	not clear	5.90 ± 0.01	3.304
3.0	0.15	4	not clear	6.95 ± 0.01	3.540
3.0	0.15	5	not clear	8.1 ± 0.1	3.844
3.0	0.15	6	not clear	9.1 ± 0.1	4.215
3.0	0.15	7	not clear	10.2 ± 0.1	4.654
3.0	0.15	8	not clear	11.2 ± 0.1	5.160

Table A.2: Damping rate for Maxwellian velocity distribution

ω_p	v_t	k	Im(ω_{PM})	Im(ω_{FDA})	Im(ω_{theory})
1.0	0.15	1	-1.6 ± 0.1	-1.50 ± 0.05	-4e-8
1.5	0.15	1	-0.9 ± 0.1	-0.88 ± 0.01	-1e-19
2.0	0.15	1	-0.39 ± 0.01	-0.48 ± 0.01	-7e-36
2.5	0.15	1	-0.17 ± 0.01	-0.25 ± 0.01	-3e-57
3.0	0.15	1	stable	-0.11 ± 0.005	-2e-83
3.5	0.15	1	stable	-0.032 ± 0.001	-2e-114
4.0	0.15	1	stable	stable	-2e-150
4.5	0.15	1	stable	stable	-3e-191
5.0	0.15	1	stable	stable	-6e-237
3.0	0.05	1	stable	stable	-8e-777
3.0	0.10	1	stable	stable	-2e-190
3.0	0.15	1	stable	-0.10 ± 0.01	-2e-83
3.0	0.20	1	-0.36 ± 0.05	-0.48 ± 0.01	-9e-46
3.0	0.25	1	-0.95 ± 0.05	-1.06 ± 0.01	-2e-28
3.0	0.30	1	-1.65 ± 0.05	-1.70 ± 0.02	-4e-19
3.0	0.15	1	not clear	-0.11 ± 0.01	-2e-83
3.0	0.15	2	not clear	-1.80 ± 0.05	-4e-19
3.0	0.15	3	not clear	-4.54 ± 0.05	-1e-7
3.0	0.15	4	not clear	-7.81 ± 0.05	-9e-4
3.0	0.15	5	not clear	-11.4 ± 0.1	0.04
3.0	0.15	6	not clear	-15.3 ± 0.1	0.27
3.0	0.15	7	not clear	-19.5 ± 0.1	0.74
3.0	0.15	8	not clear	-23.5 ± 0.1	1.3

Table A.3: Results for cold two-stream velocity distribution

ω_p	v_t	k	ω^1_{PM}	ω^1_{th}	ω^2_{PM}	ω^2_{th}
1.0	0.10	1	0.25 ± 0.1	1.01	0.5714 ± 0.0006	0.0980
1.5	0.10	1	0.31 ± 0.1	1.51	0.894 ± 0.004	0.0991
2.0	0.10	1	0.39 ± 0.1	2.01	1.188 ± 0.002	0.0995
2.5	0.10	1	0.45 ± 0.1	2.50	1.675 ± 0.009	0.0997
3.0	0.10	1	0.49 ± 0.1	3.00	1.83 ± 0.01	0.0998
3.5	0.10	1	0.60 ± 0.1	3.50	2.116 ± 0.006	0.0998
4.0	0.10	1	0.67 ± 0.1	4.00	2.57 ± 0.02	0.0999
4.5	0.10	1	0.71 ± 0.1	4.50	3.10 ± 0.02	0.0999
5.0	0.10	1	0.80 ± 0.1	5.00	3.32 ± 0.02	0.0999
3.0	0.05	1	0.50 ± 0.1	3.00	1.886 ± 0.008	0.050
3.0	0.10	1	0.49 ± 0.1	3.00	1.83 ± 0.01	0.100
3.0	0.15	1	0.58 ± 0.1	3.01	1.862 ± 0.006	0.149
3.0	0.20	1	0.48 ± 0.1	3.02	1.85 ± 0.01	0.198
3.0	0.25	1	0.68 ± 0.1	3.03	1.93 ± 0.01	0.246
3.0	0.30	1	0.75 ± 0.1	3.04	1.810 ± 0.007	0.294
3.0	0.10	1	0.49 ± 0.1	3.00	1.83 ± 0.01	0.100
3.0	0.10	2	0.50 ± 0.1	3.02	1.66 ± 0.02	0.198
3.0	0.10	3	0.57 ± 0.1	3.04	1.76 ± 0.01	0.294
3.0	0.10	4	0.63 ± 0.1	3.08	1.681 ± 0.004	0.386
3.0	0.10	5	0.85 ± 0.1	3.12	stable	0.474
3.0	0.10	6	0.94 ± 0.1	3.17	stable	0.557
3.0	0.10	7	1.04 ± 0.1	3.22	stable	0.634
3.0	0.10	8	1.12 ± 0.1	3.28	stable	0.705

Table A.4: Results for warm two-stream velocity distribution

ω_p	a	b	k	Re(ω_{PM})	Re(ω_{FDA})	Im(ω_{PM})	Im(ω_{FDA})
1.0	0.10	0.10	1	0.43 ± 0.1	±	0.1111 ± 0.0004	±
1.5	0.10	0.10	1	0.47 ± 0.1	±	0.184 ± 0.001	±
2.0	0.10	0.10	1	0.54 ± 0.1	±	0.295 ± 0.002	±
2.5	0.10	0.10	1	0.55 ± 0.1	±	0.382 ± 0.004	±
3.0	0.10	0.10	1	0.66 ± 0.1	±	0.521 ± 0.008	±
3.5	0.10	0.10	1	0.72 ± 0.1	±	0.69 ± 0.01	±
4.0	0.10	0.10	1	0.78 ± 0.1	±	0.75 ± 0.01	±
4.5	0.10	0.10	1	0.90 ± 0.1	±	1.50 ± 0.02	±
5.0	0.10	0.10	1	1.06 ± 0.1	±	1.29 ± 0.04	±
3.0	0.05	0.10	1	0.62 ± 0.1	0.62 ± 0.1	0.79 ± 0.01	stable
3.0	0.10	0.10	1	0.66 ± 0.1	0.66 ± 0.1	0.521 ± 0.008	stable
3.0	0.15	0.10	1	0.60 ± 0.1	0.71 ± 0.1	0.516 ± 0.007	0.289 ± 0.003
3.0	0.20	0.10	1	0.43 ± 0.1	0.75 ± 0.1	0.570 ± 0.005	0.596 ± 0.005
3.0	0.25	0.10	1	0.42 ± 0.1	1.00 ± 0.1	0.760 ± 0.003	0.771 ± 0.008
3.0	0.30	0.10	1	0.44 ± 0.1	1.07 ± 0.1	0.855 ± 0.002	0.851 ± 0.007
3.0	0.10	0.01	1	0.51 ± 0.1	0.58 ± 0.1	1.13 ± 0.02	1.66 ± 0.07
3.0	0.10	0.04	1	0.60 ± 0.1	0.51 ± 0.1	0.72 ± 0.02	0.69 ± 0.02
3.0	0.10	0.07	1	0.64 ± 0.1	0.51 ± 0.1	0.55 ± 0.01	0.234 ± 0.003
3.0	0.10	0.10	1	0.66 ± 0.1	0.64 ± 0.1	0.521 ± 0.008	stable
3.0	0.10	0.15	1	0.68 ± 0.1	0.67 ± 0.1	0.547 ± 0.007	stable
3.0	0.10	0.20	1	0.78 ± 0.1	0.75 ± 0.1	0.529 ± 0.006	stable

Appendix B

Code for Particle-Mesh Program

These appendices contain the important parts of the code used to generate the data for this thesis. Many components, such as error checking and I/O commands, have been omitted to save space, except where they were appropriate substitutions for commenting. For complete digital copies, please see my website, which is now at 'http://laplace.physics.ubc.ca/People/aaron/'.

The PM simulation was written in C++. There are two programs to run, the initial data generating program and the actual PM program. The former takes a generation file that contains parameters and control values and creates an input file with the initial positions for each and every particle. The latter accepts that input file and runs the simulation, producing the diagnostic information.

B.1 Initial File Generator

```
//*********************************************************************
// Get setup parameters to generate
//*********************************************************************

//setup parameters
unsigned short setup;
unsigned long numParticles;
double amp, k, vmax, vc, vwid;

//for calculation
const double epsilon=1e-10;
double uniform, y, w, area, x_n, x_np1, v_n, v_np1;
unsigned long side;

if( interactive )
  std::cout << "Please choose a setup.\n"
    << " 1. Two Particles\n"
    << " 2. Single Cold Stream Dist\n"
    << " 3. Cold Two Stream Dist\n"
    << " 4. Maxwellian Dist [exp(-v^2/vt^2)]\n"
    << " 5. Assymetric Two Stream Dist [v^2*exp(-v^2/vt^2)]\n"
```

```
    << " 6. Lorentzian Two Stream Dist
              [1/((v-b)^2+a^2) + 1/((v+b)^2+a^2)]\n"
    << "Choice: ";
std::cin >> setup;

if( interactive ) std::cout << "Number of particles: ";
std::cin >> numParticles;
if( interactive ) std::cout << "Perturbation amplitude (amp): ";
std::cin >> amp;
if( interactive ) std::cout << "Perturbation wavenumber (k): ";
std::cin >> k;
if( interactive ) std::cout << "Maximum velocity (vmax): ";
std::cin >> vmax;
if( interactive ) std::cout << "Median velocity (vc): ";
std::cin >> vc;
if( interactive ) std::cout << "Velocity decay rate (vwid): ";
std::cin >> vwid;

//*********************************************************************
// Two Particles
//*********************************************************************
if( setup == 1 ) {
  outFile << "0.4, 0.0\n"
          << "0.6, 0.0\n";
}

//*********************************************************************
// Cold Two Stream distribution
//*********************************************************************
else if( setup == 3) {

  if( numParticles % 2 != 0 )
    errorMsg << "Number of particles not even.";

  for( i=0; i<numParticles/2; i++ ) {

    //f(x) = 1 + A*cos(2*PI*k*x)
    //int(f) = x + A/(2*PI*k)*sin(2*PI*k*x)    0<int(f)<1
    uniform = (double(i)+0.5)/double(numParticles/2);
    y = uniform;
    x_np1 = y;

    do {
      x_n = x_np1;
      if( fabs(amp*2.*PI*k*sin(2.*PI*k*x_n)) > epsilon )
        x_np1 = x_n - ( x_n + amp/(2.*PI*k)*sin(2.*PI*k*x_n) - y )/
```

```
                    ( 1 + amp*cos(2.*PI*k*x_n) );
    } while( fabs(x_np1-x_n) > epsilon );

    outFile  << x_np1 << "\t" <<  vc << "\n"
             << x_np1 << "\t" << -vc << "\n";
  }
}
//*********************************************************************
// Maxwellian distribution
//*********************************************************************
else if( setup == 4 ) {

  side = (unsigned short)( sqrt(numParticles) );
  if( side*side != numParticles )
    errorMsg << "Number of particles not a square.";

  for( i=0; i<numParticles; i++ ) {

    //f(x) = 1 + A*cos(2*PI*k*x)
    //int(f) = x + A/(2*PI*k)*sin(2*PI*k*x)    0<int(f)<1
    uniform = (double(i)+0.5)/double(numParticles/2);
    y = uniform;
    x_np1 = y;

    do {
      x_n = x_np1;
      if( fabs(amp*2.*PI*k*sin(2.*PI*k*x_n)) > epsilon )
        x_np1 = x_n - ( x_n + amp/(2.*PI*k)*sin(2.*PI*k*x_n) - y )/
                      ( 1 + amp*cos(2.*PI*k*x_n) );
    } while( fabs(x_np1-x_n) > epsilon );

    //g(x) = exp(-v^2/vwid^2)
    //int(g) = 0.5*sqrt(PI)*vwid*erf(v/vwid)
    area = 0.5*erf( vmax/sqrt(2.)/vwid );
    uniform = (double(i/side)+1.)/double(side+1);
    w = area*(uniform*2.-1.);
    v_np1 = w;

    do {
      v_n = v_np1;
      v_np1 = v_n - ( 0.5*erf(v_n/sqrt(2.)/vwid) - w )/
                    ( exp(-0.5*pow(v_n/vwid, 2))/(sqrt(2.*PI)*vwid) );
    } while( fabs(v_np1-v_n) > epsilon );

    outFile << x_np1 << "\t" << v_np1 << std::endl;
  }
```

```
}

//*********************************************************************
// Lorentzian Two Stream Dist [1/((v-b)^2+a^2) + 1/((v+b)^2+a^2)]
//*********************************************************************
else if( setup == 6 ) {

  side = (unsigned short)( sqrt(numParticles) );
  if( side*side != numParticles )
    errorMsg << "Number of particles not a square.";

  std::vector<double> x, v;

  //compile list of x values
  for( i=0; i<side; i++ ) {
    //f(x) = 1 + amp*cos(2*PI*x)
    //int(f) = x + amp/(2*PI)*sin(2*PI*x)    0<int(f)<1
    uniform = ( double(i)+0.5 )/double(side);
    y = uniform;
    x_np1 = y;

    do {
      x_n = x_np1;
      if( fabs(amp*2.*PI*k*sin(2.*PI*k*x_n)) > epsilon )
        x_np1 = x_n - ( x_n + amp/(2.*PI*k)*sin(2.*PI*k*x_n) - y )/
                      ( 1 + amp*cos(2.*PI*k*x_n) );
    } while( fabs(x_np1-x_n) > epsilon );

    x.push_back(x_np1);
  }

  //compile list of v values for centred Lorentzian
  for( j=0; j<side/2; j++ ) {

    //g(x) = exp(-v^2/vwid^2)
    //int(g) = 0.5*sqrt(PI)*vwid*erf(v/vwid)
    area = atan(vmax/vwid)/PI;
    uniform = (double(j)+1.)/double(side/2+1);
    w = area*(uniform*2.-1.);
    v_np1 = 0.;

    do {
      v_n = v_np1;
      v_np1 = v_n - ( atan(v_n/vwid)/PI - w )/
                    ( vwid/PI/(pow(v_n,2)+pow(vwid,2)) );
    } while( fabs(v_np1-v_n) > epsilon );
```

```
    v.push_back(v_np1);
  }

  for( i=0; i<side; i++ ) {
    for( j=0; j<side/2; j++ ) {
      outFile << x[i] << "\t" << vc+v[j] << "\n"
              << x[i] << "\t" << -vc-v[j] << "\n";
    }
    if( side % 2 == 1 ) {
      outFile << x[i] << "\t0\n";
    }
  }
}
```

B.2 Main Function

The main function of the particle generator takes care of setup, IO, and general organization, such as the main time-stepping loop.

```
//*************************************************************************
// Define Phase
//*************************************************************************
class Phase
{
 public:
  inline Phase( double setX, double setV )
    : x(setX), v(setV) {};
  inline Phase()
    : x(0.), v(0.) {};

  double x, v;
};

//*************************************************************************
// Read initial conditions
//*************************************************************************

//get the constants
std::string filename = "pm";
double meshSpacing, startTime, timeStep, plasmaFreq;
unsigned long numMeshPoints, numTimeSteps;
unsigned short substepLevel;

if( interactive ) std::cout << "Enter file ID: ";
```

```
std::cin >> filename;
if( interactive ) std::cout << "Enter mesh spacing: ";
std::cin >> meshSpacing;
if( interactive ) std::cout << "Enter number of mesh points: ";
std::cin >> numMeshPoints;
if( interactive ) std::cout << "Enter initial time: ";
std::cin >> startTime;
if( interactive ) std::cout << "Enter base time step: ";
std::cin >> timeStep;
if( interactive ) std::cout << "Enter number of base time steps: ";
std::cin >> numTimeSteps;
if( interactive ) std::cout << "Enter level of substepping: ";
std::cin >> level;
if( interactive ) std::cout << "Enter plasma frequency: ";
std::cin >> plasmaFreq;
if( interactive ) std::cout << "Enter phase coordinates:\n";

std::cout << "Running ID '" << filename << "'\n";

//get the initial sampling point conditions
std::vector<Phase> initialConditions;
double currX, currV;

while( std::cin.peek() != EOF ) {
  std::cin >> currX >> currV;
  initialConditions.push_back( Phase(currX, currV) );
}

std::cout << "Number of particles read: "
          << initialConditions.size() << "\n";

//*********************************************************************
// Setup
//*********************************************************************
//create the plasma object
Experiment plasma( meshSpacing, numMeshPoints, startTime,
                   timeStep/pow(2., level), level, plasmaFreq,
                   initialConditions, outputFormat );

//output initial data
outputFile << plasma.writeHeader();
outputFile << plasma.writeData();

//time the calculation
Stopwatch timer;
timer.start();
```

```
//*********************************************************************
// Perform Simulation
//*********************************************************************

//base step loop
for( i=0; i < numTimeSteps*pow(2, level); i++ ) {
  plasma.step();
  //output data for high memory usage formats only at rate user sets
  if( outputFreq != 0 and i % outputFreq*pow(2, level) == 0 ) {
  outputFile << plasma.writeData();
  snapshotFile << plasma.writeData("snapshot");
  velocityFile << plasma.writeData("velocity");
  }
  historyFile << plasma.writeData("history");
  fourierFile << plasma.writeData("fourier");

  //save progress every 5 minutes
  if( timer.alarm(300) ) {
  stateFile << filename << " file ID\n"
          << plasma.writeData("save-state");
  std::cout << "State saved at step " << i << ".\n"
          << "Time elapsed - " << timer.writeString() << std::endl;
  }
}

//*********************************************************************
// Clean up
//*********************************************************************
//output elapsed time
timer.stop();

stateFile << filename << " file ID\n"
        << plasma.writeData("save-state");
std::cout << "Simulation complete.\n"
        << "Time elapsed - " << timer.writeString() << std::endl;
```

B.3 Experiment Class

The experiment class performs time steps and analyzes the particles to produce physical data.

```
//*********************************************************************
// Set up class with parameters and constants
//*********************************************************************
```

```
Experiment::Experiment( double meshSpacing, unsigned long numMeshPoints,
double startTime, double timeStep,
unsigned short substepLevel, double plasmaFreq,
const std::vector<Phase>& initialConditions,
const std::string& outputFormat )
  : time( startTime ),
    QtoM(1.), eps0(1.), level( substepLevel ),
    H( meshSpacing ), DT( timeStep ), freq( plasmaFreq ),
    Nmesh( numMeshPoints ), L( double(Nmesh)*H ),
    Ncell( initialConditions.size()/double(Nmesh) ),
    rhoFactor( freq*freq/Ncell*eps0/QtoM ), //gives charge per particle
    density( Nmesh ), potential( Nmesh ), Efield( Nmesh ),
    format( outputFormat )

{
  particles.resize( initialConditions.size() );

  unsigned long i;
  for( i=0; i<initialConditions.size(); i++ ) {
    particles[i].x = initialConditions[i].x;
    particles[i].v = initialConditions[i].v;
  }

//output total charge as a check
  std::cout << "Q = " << Ncell*L*rhoFactor << "\n";
}

void Experiment::step()
{
  unsigned long i, loMeshPt, hiMeshPt;
  double hiFrac;

  //*********************************************************************
  // UpdateDensity - Assign charge to the mesh
  //*********************************************************************

  //double neutral = 0;

  for( i=0; i<Nmesh; i++ )
    density[i] = rhoFactor*Ncell;

  for( i=0; i<particles.size(); i++ ) {
    loMeshPt = (unsigned long)(particles[i].x/H);
    hiMeshPt = loMeshPt+1;
    hiFrac = particles[i].x/H - double(loMeshPt);
    if( loMeshPt == Nmesh ) {
```

```
    loMeshPt = 0;
    hiMeshPt = 1;
  }
  else if( hiMeshPt == Nmesh ) {
    hiMeshPt = 0;
  }

  density[loMeshPt] -= rhoFactor*(1.0-hiFrac);
  density[hiMeshPt] -= rhoFactor*hiFrac;
}

//*********************************************************************
// UpdatePotential - Calculate the potential from the charge density
//*********************************************************************

//compute potential at mesh point 1
potential[1] = Nmesh*density[0];
for( i=1; i<Nmesh; i++ )
  potential[1] += i*density[i];
potential[1] *= H*H/eps0/double(Nmesh);

//compute potential at mesh point 2
potential[2] = (H*H/eps0*density[1] + 2*potential[1]);

//compute rest of potentials
for( i=3; i<Nmesh; i++ )
  potential[i] = (H*H/eps0*density[i-1] + 2*potential[i-1]
                  - potential[i-2]);

//compute potential at mesh point 0
potential[0] = (H*H/eps0*density[Nmesh-1] + 2*potential[Nmesh-1]
                - potential[Nmesh-2]);

//*********************************************************************
// UpdateEfield - Calculate the electric field
//*********************************************************************

//boundary condition
Efield[0] = (potential[1] - potential[Nmesh-1])/(2*H);

//internal conditions
for( i=1; i <= Nmesh-2; i++ )
  Efield[i] = (potential[i-1] - potential[i+1])/(2*H);

//boundary condition
Efield[Nmesh-1] = (potential[Nmesh-2] - potential[0])/(2*H);
```

```
  //*********************************************************************
  // UpdateParticles - Find force at each particle and
  //                   perform equations of motion
  //*********************************************************************

  for( i=0; i<particles.size(); i++ ) {

    loMeshPt = (unsigned long)(particles[i].x/H);
    hiMeshPt = loMeshPt+1;
    hiFrac = particles[i].x/H - double(loMeshPt);
    if( loMeshPt == Nmesh ) {
      loMeshPt = 0;
      hiMeshPt = 1;
    }
    else if( hiMeshPt == Nmesh ) {
      hiMeshPt = 0;
    }

    particles[i].v += QtoM*(Efield[loMeshPt]*(1.0-hiFrac) +
                      Efield[hiMeshPt]*hiFrac)*DT; //negative charges
    particles[i].x += particles[i].v*DT;

    //do wrap-around and check that it only happens once
    if( particles[i].x < 0 ) {
      particles[i].x += L;
      if( particles[i].x < 0 )
      throw std::runtime_error( "Speeds have become excessive." );
    }
    if( particles[i].x > L ) {
      particles[i].x -= L;
      if( particles[i].x > L )
      throw std::runtime_error( "Speeds have become excessive." );
    }
  }

  //*********************************************************************
  // Increment the time
  //*********************************************************************
  time += DT;
}

//*********************************************************************
// Output header data for pp2d
//*********************************************************************
std::string Experiment::writeHeader() const
```

```
{
  std::stringstream ss;

  if( format == "pp2d-xy" || format == "pp2d-xv" ) {

    //give number of particles and their sizes
    ss << particles.size() << "\n";

    unsigned long i;
    for( i=0; i < particles.size(); i++ ) {
      ss << 0.1 << "\n";
    }
  }

  return ss.str();
}

//*********************************************************************
// Output phase-space data for pp2d
//*********************************************************************
std::string Experiment::writeData( const char* outputFormat ) const
{
  std::string fmt( outputFormat );
  std::stringstream ss;
  unsigned long i, j;

  if( fmt == "pp2d-xy" ) {
    ss << time << "\n";

    for( i=0; i < particles.size(); i++ ) {
      ss << particles[i].x << "\t0\t0\n";
    }
  }
  else if ( fmt == "pp2d-xv" ) {
    ss << time << "\n";

    for( i=0; i < particles.size(); i++ ) {
      ss << particles[i].x << "\t" << particles[i].v << "\t0\n";
    }
  }
  else if ( fmt == "gnuplot-xt" ) {
    ss << time << "\t";

    for( i=0; i < particles.size(); i++ )
      ss << particles[i].x << "\t";
```

```
  ss << std::endl;
}

//*********************************************************************
// Output history data
//*********************************************************************
else if ( fmt == "history" ) {
  ss << time << "\t";

  double KE = 0., Ees = 0., mom = 0., absMom = 0.;

  for( i=0; i < particles.size(); i++ ) {
    KE += particles[i].v*particles[i].v;
    mom += particles[i].v;
    absMom += fabs( particles[i].v );
  }
  //Q = (Ncell*L*rhoFactor)
  //Q/Nparticles = H*rhoFactor
  KE *= 0.5*H*rhoFactor/QtoM;
  mom *= H*rhoFactor/QtoM;
  absMom *= H*rhoFactor/QtoM;

  for( i=0; i < Nmesh; i++ )
    Ees -= density[i]*potential[i];
  Ees *= 0.5*H;

  ss << mom << "\t" << absMom << "\t"
     << Ees << "\t" << KE << std::endl;
}
else if ( fmt == "snapshot" ) {
  ss << "# " << time << "\n";

  //unsigned long count = 0;
  double position = 0., totalDen = 0., totalPot = 0., totalEfd = 0.;

  for( i=0; i < Nmesh; i++ ) {
    position += H*i;
    totalDen += -density[i];
    totalPot += potential[i];
    totalEfd += Efield[i];
    //if( count == 25 ) {
    ss << position << " " << totalDen << " "
     << totalPot << " " << totalEfd << "\n";
    position = 0.;
    totalDen = 0.;
    totalPot = 0.;
```

```
    totalEfd = 0.;
  }

  ss << std::endl << std::endl;
}

//*********************************************************************
// Output velocity distribution data
//*********************************************************************
else if ( fmt == "velocity" ) {

  unsigned long loMeshPt, hiMeshPt;
  double hiFrac;
  std::vector<double> vDist(1000);
  const double VMAX = 2.0, DV = 2.*VMAX/vDist.size();

  for( i=0; i<particles.size(); i++ ) {

    loMeshPt = (unsigned long)((particles[i].v+VMAX)/DV);
    hiMeshPt = loMeshPt+1;
    if( loMeshPt < 0 or hiMeshPt >= vDist.size() ) continue;

    hiFrac = (particles[i].v+VMAX)/DV - double(loMeshPt);

    vDist[loMeshPt] += (1.0-hiFrac)/double(particles.size());
    vDist[hiMeshPt] += hiFrac/double(particles.size());
  }

  ss << "# " << time << "\n";

  for( i=0; i < vDist.size(); i++ )
    ss << -VMAX + i*DV << " " << vDist[i] << "\n";

  ss << std::endl << std::endl;
}

//*********************************************************************
// Output data on first 8 terms of discrete Fourier series
//*********************************************************************
else if ( fmt == "fourier" ) {
  ss << time << "\t";

  double cosCoeff, sinCoeff, density_k, potential_k;

  //find total energy (j=0) and amplitude of first 8 modes
  for( j=0; j<9; j++ ) {
```

```
      //fourier series of density
      cosCoeff = 0;
      sinCoeff = 0;
      for( i=0; i<Nmesh; i++ ) {
      cosCoeff += density[i]*cos(i*H*2*PI*j);
      sinCoeff += density[i]*sin(i*H*2*PI*j);
      }
      density_k = sqrt( cosCoeff*cosCoeff + sinCoeff*sinCoeff )*H;

      //fourier series of potential
      cosCoeff = 0;
      sinCoeff = 0;
      for( i=0; i<Nmesh; i++ ) {
      cosCoeff += potential[i]*cos(i*H*2*PI*j);
      sinCoeff += potential[i]*sin(i*H*2*PI*j);
      }
      potential_k = sqrt( cosCoeff*cosCoeff + sinCoeff*sinCoeff )*H;


      //find electrostatic energy
      //if you want to output density_k and potential_k, multiply them by 2.
      ss << density_k*potential_k << "\t";
    }

    ss << std::endl;
  }
  //*********************************************************************
  // Output an input format file as a save-state
  //*********************************************************************
  else if ( fmt == "save-state" ) {
    ss << H << " mesh spacing\n"
       << Nmesh << " number of mesh points\n"
       << time << " initial time\n"
       << DT << " base time increment\n"
       << "10000 number of base time steps\n"
       << level << " level of substepping\n"
       << freq << " plasma frequency\n";

    for( i=0; i < particles.size(); i++ )
      ss << particles[i].x << " " << particles[i].v << "\n";
  }

//return the data to the main function
  return ss.str();
}
```

Appendix C

Code for FDA Program

The finite difference approximation program was built using FORTRAN and the high-level language RNPL, developed by Robert Marsa and Matt Choptuik at Center for Relativity, The University of Texas in Austin. Three user defined files were required: the RNPL description, the initialization code, and the update code. Their names are relatively self-explanatory. Please see RNPL documentation for further description of the features used.

C.1 RNPL Description File

```
############################################################
# RNPL program to solve the Vlasov equation in 1D
#
#    f_t + v*f_x + a*f_v = 0
#
# The acceleration 'a' is given by the Lorentz force.
#
#    a = q*E
#
############################################################

#-----------------------------------------------------------
# Definition of memory size (only needed for Fortran)
#-----------------------------------------------------------
system parameter int memsiz := 10000000

#-----------------------------------------------------------
# Definition of parameters and associated default values.
#-----------------------------------------------------------
parameter int first  :=   1

#-----------------------------------------------------------
# Specify domain
#-----------------------------------------------------------
parameter float xmin :=   0.0
parameter float xmax :=   1.0
parameter float vmin :=  -5.0
```

```
parameter float vmax :=   5.0

#-------------------------------------------------------------
# Physical values
#-------------------------------------------------------------
parameter float wp   :=   1.0
parameter float qtot0:=   1.0
parameter float qnorm:=   1.0
constant parameter float eps0:= 1.0
constant parameter float qtom :=   1.0
constant parameter float twopi := 6.283185307179586

#-------------------------------------------------------------
# The following parameters are used in the
# specification of the initial data.
#-------------------------------------------------------------
parameter int type   :=   1
parameter float amp  :=   1.0
parameter float knum :=   1.0
parameter float xc   :=   0.5
parameter float xwid :=   0.05
parameter float vc   :=   0.0
parameter float vwid :=   0.05
parameter float const:=   0.0

parameter float dissip:=  0.0

#-------------------------------------------------------------
# Definition of coordinate system
#-------------------------------------------------------------
phase coordinates t, x, v

#-------------------------------------------------------------
# Definition of finite-difference grids
#-------------------------------------------------------------
uniform phase grid g1 [1:Nx][1:Nv] {xmin:xmax} {vmin:vmax}

#-------------------------------------------------------------
# Definition of u as the phase-space density function
#-------------------------------------------------------------
float f on g1 at -1,0,1 {out_gf = 1}

#-------------------------------------------------------------
# Initializations
#-------------------------------------------------------------
```

```
initialize f { [1:Nx][1:Nv] := 0 }

initializer0.inc initializer1 initializes f
    header f, x, v, dx, dv, xmin, xmax, qtot0, qnorm, twopi, knum,
   wp, qtom, eps0, type, amp, vc, vwid, xc, xwid, const

#-------------------------------------------------------------
# Updates
#-------------------------------------------------------------

looper standard

update0.inc update0 updates f
  header f, t, x, v, dt, dx, dv, xmin, xmax, twopi,
first, qtom, dissip, qtot0, qnorm, const
```

C.2 Update Function

```
integer i,j
integer Nx, Nv
real*8 Ncell

Nx = g1_Nx - 1 //last point is identified with first
Nv = g1_Nv

C-----------------------------------------------------------------------
C Maxwellian velocity distribution
C-----------------------------------------------------------------------
if (type .eq. 2) then
  do i=1, Nx
    do j=1, Nv
      if ( j .eq. 1 .or. j .eq. Nv ) then
        f_n(i,j) = const
      else
        f_n(i,j) = exp(-0.5d0*v(j)**2/vwid**2)/
&                  (sqrt(twopi)*vwid)*(1d0+amp*cos(twopi*knum*x(i)))
&                  + const
      end if
    end do
  end do
C-----------------------------------------------------------------------
C Two Lorentzian velocity distribution
C-----------------------------------------------------------------------
else if (type .eq. 4) then
  do i=1, Nx
```

```
    do j=1, Nv
      if ( j .eq. 1 .or. j .eq. Nv ) then
        f_n(i,j) = const
      else
        f_n(i,j)= 2d0*vwid/twopi*( 1d0/((v(j)-vc)**2 + vwid**2)
&                + 1d0/((v(j)+vc)**2 + vwid**2) )*
&               (1d0+amp*cos(twopi*knum*x(i))) + const
      end if
    end do
  end do
end if

C-----------------------------------------------------------------------
C Normalize charge
C-----------------------------------------------------------------------
Ncell = 0d0
do j=1, Nv
  do i=1, Nx
    Ncell = Ncell + f_n(i,j)
  end do
end do
Ncell = Ncell*dx*dv
qnorm = ((xmax-xmin)*wp**2/Ncell*eps0/qtom)

write(*,*) "Q_initial = ", Ncell, " qnorm = ", qnorm

do j=1, Nv
  do i=1, Nx
    !((xmax-xmin)*wp**2*eps0/qtom) is total charge
    f_n(i,j) = f_n(i,j) * qnorm
  end do
  f_n(g1_Nx, j) = f_n(1,j)
end do

qtot0 = 0d0
do j=1, Nv
  do i=1, Nx
    qtot0 = qtot0 + f_n(i,j)
  end do
end do
qtot0 = qtot0*dx*dv
write(*,*) "Q = ", qtot0
```

C.3 Initialization Function

```
C-----------------------------------------------------------------------
C Define variables
C-----------------------------------------------------------------------
!output global from 'other_glbs.inc'
integer       output(41)
common      / com_oglb_int / output

real*8  a(g1_Nx-1), phi(g1_Nx-1)
integer i, ip1, ip2, im1, im2, j, jp2, jm2
integer Nx, Nv

real*8 qtot, test

!lapack variables
real*8 dl(g1_Nx-2), d(g1_Nx-2), du(g1_Nx-2) !diagonals
integer nrhs, info
parameter (nrhs = 1)

!distribution variables
integer  ret, gft_out_full
real*8  rhox(g1_Nx-1), rhoxB(g1_Nx-1), rhov(g1_Nv)
integer rank, shape(1)
character*5 cnames(1)

!IO variables
integer fh, skip
save skip
character*50 historyFile, rhoXFile, rhoVFile, rhoWFile, fmt

!history variables
real*8 ptot, absptot, Ees, KE, rmserr
real*8 fa, fb
real*8 rhok(0:8), phik(0:8)

Nx = g1_Nx - 1 //last point is identified with first
Nv = g1_Nv

C-----------------------------------------------------------------------
C Update Density - Integrate over the velocity dimension
C  charge density is placed in 'phi'
C-----------------------------------------------------------------------

qtot = 0d0
do i=1, Nx
```

```
  do j=1, Nv
    qtot = qtot + f_n(i,j)
  end do
end do
qtot = qtot*dx*dv

!sum over all velocities
do i=1, Nx
  rhox(i) = -qtot/(xmax-xmin)
  do j=1, Nv
    rhox(i) = rhox(i) + f_n(i,j)*dv
  end do

  !prepare for potential calculation
  phi(i) = rhox(i)*dx**2 / eps0
end do

C-----------------------------------------------------------------------
C Update Potential - Use tridiagonal solver for Poisson's equation
C    charge potential is placed in 'phi'
C-----------------------------------------------------------------------

!create the submatrix for d2phi/dx2
do i=1, Nx-1
  du(i) = -1d0
  d(i)  =  2d0
  dl(i) = -1d0
end do

!calculate the Nx-1 by Nx-1 submatrix
call dgtsv( Nx-1, nrhs, dl, d, du, phi, Nx, info )

!set the end value to zero
phi(Nx) = 0d0

C-----------------------------------------------------------------------
C Update Acceleration - coefficient of du/dv in Vlasov equation
C-----------------------------------------------------------------------

!acceleration proportional to -dphi/dx
a(1) = qtom*(phi(Nx) - phi(2))/(2*dx)
do i=2, Nx-1
  a(i) = qtom*(phi(i-1) - phi(i+1))/(2*dx)
end do
a(Nx) = qtom*(phi(Nx-1) - phi(1))/(2*dx)
```

```
do i=1, Nx
  if( (dt/dx)**2+(a(i)*dt/dv)**2  .ge. 1d0 ) then
    write(*,*) "Caution: a(", i, ") =", a(i),
&              " exceeds the instability criterion."
  end if
end do

C-----------------------------------------------------------------------
C Solve equation
C-----------------------------------------------------------------------

do i=1, g1_Nx
  f_np1(i,1) = const * qnorm
  f_np1(i,Nv) = const * qnorm
end do

do j=2, Nv-1
  do i=1, Nx

    ip1 = i+1
    im1 = i-1
    if( i .eq. 1 ) then
      im1 = Nx
      im2 = Nx-1
    else if( i .eq. Nx ) then
      ip1 = 1
      ip2 = 2
    end if

    jp2 = j+2
    jm2 = j-2
    if( j .eq. 2 ) then
      jm2 = 1
    else if( j .eq. Nv-1 ) then
      jp2 = Nv
    end if

    !standard leapfrog
    f_np1(i,j)= f_nm1(i,j) - dt*(
&        v(j)*(f_n(ip1,j)-f_n(im1,j))/dx +
&        a(i)*(f_n(i,j+1)-f_n(i,j-1))/dv )
&        - dissip/16d0 *
&        ( 6d0*f_n(i,j) + f_n(ip2,j) + f_n(im2,j)
&        - 4*(f_n(ip1,j) + f_n(im1,j))
&        + 6d0*f_n(i,j) + f_n(i,jp2) + f_n(i,jm2)
&        - 4*(f_n(i,j+1) + f_n(i,j-1)) )
```

```
  end do
  f_np1(g1_Nx,j) = f_np1(1,j)
end do

C-----------------------------------------------------------------------
C Output diagnostics
C-----------------------------------------------------------------------
if( first .eq. 1 ) then
  skip = output(13)
else if( t .gt. 0d0 ) then
  skip = skip + 1
end if

if( skip .eq. output(13) ) then
  skip = 0

  !find velocity distribution
  do j=1, Nv
    rhov(j) = 0d0
    do i=1, Nx
      rhov(j) = rhov(j) + f_n(i,j)
    end do
    rhov(j) = rhov(j)*dv
  end do

  !find total momentum and kinetic energy
  ptot = 0
  absptot = 0
  KE = 0
  do j=1, Nv
    ptot = ptot + v(j)*rhov(j)
    absptot = absptot + abs(v(j))*rhov(j)
    KE = KE + v(j)**2*rhov(j)
  end do
  ptot = ptot/qtom*dv
  absptot = absptot/qtom*dv
  KE = 0.5d0*KE/qtom*dv

  !find electrostatic energy
  Ees = 0
  do i=1, Nx
    Ees = Ees + rhox(i)*phi(i)
  end do
  Ees = 0.5d0*Ees*dx

  !find RMS error
```

```
  rmserr = 0d0
  do i=1, Nx
    do j=2, Nv-1

    ip1 = i+1
    im1 = i-1
    if( i .eq. 1 ) then
      im1 = Nx
    else if( i .eq. Nx ) then
      ip1 = 1
    end if

    rmserr = rmserr + ( ( f_np1(i,j)-f_nm1(i,j) )/(2d0*dt) +
&                       v(j)*( f_n(ip1,j)-f_n(im1,j) )/(2d0*dx) +
&                       a(i)*( f_n(i,j+1)-f_n(i,j-1) )/(2d0*dv) )**2
    end do
  end do
  rmserr = sqrt( rmserr / (Nx * (Nv-2)) )

  !find amplitude of first 8 modes
  do j=0, 8
    fa = 0d0
    fb = 0d0
    do i=1, Nx
      fa = fa + rhox(i)*cos(twopi*real(j)*x(i))
      fb = fb + rhox(i)*sin(twopi*real(j)*x(i))
    end do
    rhok(j) = sqrt( fa**2 + fb**2 )*dx

    fa = 0d0
    fb = 0d0
    do i=1, Nx
      fa = fa + phi(i)*cos(twopi*real(j)*x(i))
      fb = fb + phi(i)*sin(twopi*real(j)*x(i))
    end do
    phik(j) = sqrt( fa**2 + fb**2 )*dx
  end do

  !write histories
  if( first .eq. 1 ) then
    open(fh, FILE=historyFile, STATUS='UNKNOWN')
  else
    open(fh, FILE=historyFile, STATUS='UNKNOWN',
&        ACCESS='APPEND')
  end if
  write(fh,900) t, (qtot-qtot0)/qtot0, ptot, absptot,
```

```
&      Ees, KE, rmserr
 900      format( 10000E14.6 )
  close(fh)

  !write spatial snapshots
  if( first .eq. 1 ) then
    open(fh, FILE=rhoXFile, STATUS='UNKNOWN')
  else
    open(fh, FILE=rhoXFile, STATUS='UNKNOWN',
&        ACCESS='APPEND')
  end if
  write(fh,*) "#", t
  do i=1,Nx
    write(fh,900) x(i), rhox(i), phi(i), a(i)/qtom
  end do
  write(fh,*)
  write(fh,*)
  close(fh)

  !write velocity snapshots
  if( first .eq. 1 ) then
    open(fh, FILE=rhoVFile, STATUS='UNKNOWN')
  else
    open(fh, FILE=rhoVFile, STATUS='UNKNOWN',
&        ACCESS='APPEND')
  end if
  write(fh,*) "#", t
  do j=1,Nv
    write(fh,900) v(j), rhov(j), f_n(1,j)
  end do
  write(fh,*)
  write(fh,*)
  close(fh)

  !write fourier history
  if( first .eq. 1 ) then
    open(fh, FILE=rhoWFile, STATUS='UNKNOWN')
  else
    open(fh, FILE=rhoWFile, STATUS='UNKNOWN',
&        ACCESS='APPEND')
  end if
  write(fh,900) t, (rhok(j)*phik(j), j=0,8)
  close(fh)
end if
```

Appendix D

Code for Convergence Test

This program is written in C++ with Qt I/O objects. It accepts Gnuplot-style data files and computes the convergence factor (3.44) for the column specified.

```
//prepare variables
double diff12, diff23a, diff23b, sum12, sum23, convergence;

QString buffer;
unsigned short blankLine, ratio, column = 1;
unsigned long i, j, record = 0, lastSize;
bool ok;
double time1, time2, time3, yValue;
std::vector<double> set1, set2, set3;

//*********************************************************************
// Loop through data files until one ends
//*********************************************************************
while( !stream1.atEnd() and !stream2.atEnd() and !stream3.atEnd() ) {

  record++;

  //load record from file1
  blankLine = 0;
  set1.clear();
  do {
    buffer = stream1.readLine().simplifyWhiteSpace();

    if( buffer.isEmpty() )
      blankLine++;
    else if( buffer.startsWith("#") ) {
      buffer.remove(0,1);
      time1 = buffer.toDouble(&ok);
    }
    else {
    yValue = buffer.section( ' ', column, column ).toDouble(&ok);
    set1.push_back(yValue);
    }
```

```
} while( blankLine != 2 and !stream1.atEnd() );

//load record from file2
blankLine = 0;
set2.clear();
do {
  buffer = stream2.readLine().simplifyWhiteSpace();

  if( buffer.isEmpty() )
    blankLine++;
  else if( buffer.startsWith("#") ) {
    buffer.remove(0,1);
    time2 = buffer.toDouble(&ok);
  }
  else {
  yValue = buffer.section( ' ', column, column ).toDouble(&ok);
   set2.push_back(yValue);
  }
} while( blankLine != 2 and !stream2.atEnd() );

//load record from file3
blankLine = 0;
set3.clear();
do {
  buffer = stream3.readLine().simplifyWhiteSpace();

  if( buffer.isEmpty() )
    blankLine++;
  else if( buffer.startsWith("#") ) {
    buffer.remove(0,1);
    time3 = buffer.toDouble(&ok);
  }
  else {
    yValue = buffer.section( ' ', column, column ).toDouble(&ok);
    set3.push_back(yValue);
  }
} while( blankLine != 2 and !stream3.atEnd() );

//*********************************************************************
// Check that data conforms to expectations
//*********************************************************************

//check that sets have correct sizes
if( set1.size() <= 0 or set2.size() <= 0 or set3.size() <= 0 ) {
 std::cerr << "converge> record " << record << ": Empty set.";
return 1;
```

```
}
else if( set1.size() == set2.size() and
         set2.size() == set3.size() ) {
  ratio = 1;
}
else if( set2.size() == 2*set1.size() and
         set3.size() == 2*set2.size() ) {
  ratio = 2;
}
else if( set2.size()-1 == 2*(set1.size()-1) and
         set3.size()-1 == 2*(set2.size()-1) ) {
  set1.pop_back();
  set2.pop_back();
  set3.pop_back();
  ratio = 2;
}
else if( set2.size() == 10*set1.size() and
         set3.size() == 10*set2.size() ) {
  ratio = 10;
}
else {
  std::cerr << "converge> record " << record
            << ": Set sizes do not match.";
  return 1;
}

//check that times correspond
if( time1 != time2 or time2 != time3 ) {
  std::cerr << "converge> record " << record
            << ": Times do not match.";
  return 1;
}

//check that the number of entries has not changed
if( record > 1 and set1.size() != lastSize ) {
  std::cerr << "converge> record " << record
            << ": Array size has changed.";
}
lastSize = set1.size();

//*********************************************************************
// Perform calculation
//*********************************************************************
sum12 = 0.;
sum23 = 0.;
if( ratio == 1 ) {
```

```
    for( i=0; i<set1.size(); i++ ) {
      diff12 = set2[i] - set1[i];
      diff23a = set3[i] - set2[i];
      sum12 += diff12*diff12;
      sum23 += diff23a*diff23a;
    }
  }
  else if( ratio == 2 ) { //ratio = 2
    for( i=0; i<set1.size(); i++ ) {
      diff12 = set2[2*i] - set1[i];
      diff23a = set3[4*i] - set2[2*i];
      diff23b = set3[4*i+2] - set2[2*i+1];
      sum12 += diff12*diff12;
      sum23 += (diff23a*diff23a + diff23b*diff23b)/2.;
    }
  }
  else { //ratio > 2
    for( i=0; i<set1.size(); i++ ) {
     diff12 = set2[ratio*i] - set1[i];
     diff23b = 0;
     for( j=0; j<ratio; j++ ) {
     diff23a = set3[ratio*(ratio*i+j)] - set2[ratio*i+j];
     diff23b += diff23a*diff23a;
   }
   sum12 += diff12*diff12;
   sum23 += diff23b/ratio;
  }
}
convergence = sum23 ? sqrt(sum12 / sum23) : 0;

std::cout << time1 << "\t" << convergence << std::endl;
}
```