# Electrostatic PIC simulation
# of plasmas in one dimension

Daniel Martin

School of Physics and Astronomy,
The University of Manchester

Fourth Year M.Phys Project Report

January 2007

This experiment was performed in collaboration with P. Hughes
and under the supervision of Dr. P. Browning.

### Abstract

A computer program has been written to simulate plasmas in one dimension in the electrostatic limit. This was done using a Particle In Cell (PIC) method, which is detailed in this report. The validity of the program was tested by using it to simulate various known plasma behaviour including cold plasma oscillations, the two-stream instability and the four-stream instability which are documented in this report. Suggestions for enhancements to the program are given at the end of this report.

# Contents

# 1   Introduction

Plasma, or Ionised Gas, may be considered to be the fourth state of matter, a more energetic state than even a classical gas. It can be thought of as a sub-atomic gas; electrons have dissociated from their ions and move independently. Unlike the other states of matter (solids, liquids and gases) plasma contains free charge carriers (the free electrons and their ions), causing it to interact strongly with electromagnetic fields.

   Plasmas are more common than one might think and are estimated to make up over 99% of the visible universe. We see plasmas in the natural world, in the form of flames, lightning, the ionosphere[1] and the sun. Plasma is also used in modern technology such as in flourescent lamps, plasma televisions, and perhaps most notably in tokamaks, which magnetically confine plasma in order to achieve thermo-nuclear fusion.

   The technology required to confine and manipulate many plasmas (such as that used in attempts to achieve thermo-nuclear fusion - for example tokamaks) is very expensive and takes a long time to build. In contrast, computer simulation of plasmas is very cheap and programs can be written and altered far more quickly than physical equipment.

   Additionally, in a computer simulation, users can view all the information they desire, including information that would be difficult to obtain in a physical setup, such as the trajectory of a single particle. Perhaps most notably, the user can track the balance between kinetic and potential energy of the particles in the plasma.

   The most common methods of simulating plasmas are so called *Particle In Cell* (PIC) methods, which treat plasmas as a system of particles, and *Magneto-Hydro-Dynamic* (MHD) methods, which treat the plasmas as fluids. Hybrid models also exist, for instance models that treat some parts of the plasma as a fluid (using MHD methods) but others as particles (using PIC methods).

   A computer program has been written to simulate plasmas in the electrostatic limit, in one dimension, using a PIC method. This report will first explain the methods used in detail, then look at case studies of plasma behaviour that can be observed with the program. Finally, it will explain briefly how the program may be extended to simulate a wider variety of plasma behaviour.

# 2   General method

The program simulates plasmas by using a PIC method. In PIC methods of plasma simulation, one wishes to solve the equation of motion for each of the particles in the plasma. In general, these equations cannot be solved through linear analysis, and we are reduced to iterating over small, finite, time increments to achieve an approximate solution. At each time step, the electric field at each particle must be calculated. To calculate this exactly is too computationally expensive, so we resort to calculating it on a discretised grid before mapping the information back onto the individual particles.

   The program can be summed up as the repetition of four main operations as illustrated in figure 1. The program is started with a set of initial conditions, these being the positions, velocities and charges of all the particles that make up the plasma. Using the positions and charges of all the particles a charge density grid is calculated in the first 'weighting' stage. The electric field is then calculated on this grid. The program then uses a second weighting stage to map the electric field information from the grid onto the particles. Finally, the equations of motion can be 'integrated'[2], meaning that the particle positions and velocities are advanced by one time step. After this the

---

[1]Part of which reflects radio waves making long range radio communication on Earth possible.
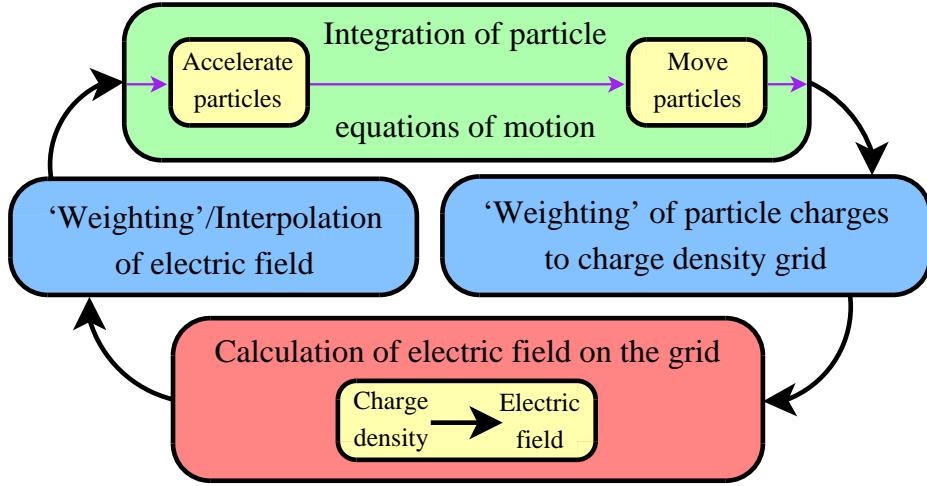[2]Note that here we have used the word 'integrated' rather loosely.

**Figure 1**: Flow diagram of the PIC method.

cycle is repeated to reach further time steps. The source-code for the main program loop can be found in section B.1 of the appendix.

Throughout the simulation, diagnostic information such as potential & kinetic energy, the electric field, charge density, and the particle positions & velocities at each time step, can be recorded for later examination by the simulator.

## 2.1   Integration of the particle equations of motion

The particles' motion is governed primarily by electromagnetism and thus the Lorentz force law:

$$\boldsymbol{F} = m\boldsymbol{a} = m\frac{d^2\boldsymbol{r}}{dt^2} = q(\boldsymbol{E} + \boldsymbol{v} \times \boldsymbol{B}) \tag{1}$$

$$\therefore \frac{d^2\boldsymbol{r}}{dt^2} = \frac{q}{m}(\boldsymbol{E} + \boldsymbol{v} \times \boldsymbol{B}) \tag{2}$$

Where, $\boldsymbol{F}$ is the force, $m$ is the particle's mass, $\boldsymbol{a}$ is the acceleration, *vecr* is the position, $t$ is time, $q$ is the charge, $\boldsymbol{E}$ is the electric field, $\boldsymbol{v}$ is the velocity and $\boldsymbol{B}$ is the magnetic field.

In this program, we are concerning ourselves only with plasmas in the electrostatic limit and are thus going to ignore the magnetic field. The particles' motion is therefore only dependent on the electric field. Additionally, we are only going to work in one dimension, so our equation of motion becomes:

$$\frac{d^2x}{dt^2} = \frac{q}{m}E_x(t) \tag{3}$$

(From here on we will drop the tedious $x$ sub-script.)

As the electric field, $E_x(t)$, is an arbitrary function of time, we cannot solve (3) by linear analysis. We must implement a numerical method to approximate a solution[3]. Explained below are some possible methods.

---

[3]The solution is only an approximation, due to a computer's inability to handle the infinitesimal quantities required for a true solution.

However we will see that in these methods it is useful split equation (3) (a $2^{\text{nd}}$ order differential equation) into two $1^{\text{st}}$ order equations:

$$\frac{dv}{dt} = \frac{q}{m}E(t) \tag{4}$$

$$\frac{dx}{dt} = v(t) \tag{5}$$

### 2.1.1   Euler's method

Euler's method is the simplest and most intuitive method for solving differential equations. As it is so simple we will derive it here.

As a computer cannot handle infinitesimal quantities, we are forced to approximate equations (4) and (5) as finite difference equations:

$$\frac{\Delta v}{\Delta t} = \frac{v_{n+1} - v_n}{\Delta t} = \frac{q}{m}E_n \tag{6}$$

$$\frac{\Delta x}{\Delta t} = \frac{x_{n+1} - x_n}{\Delta t} = v_n \tag{7}$$

Where, for instance $\Delta t$ represents a finite difference in time.

We split $\Delta v$ and $\Delta x$ into old and new information (we already know the old information and need to calculate the new). We can rearrange to find equations for our new velocity and position:

$$v_{n+1} = v_n + \frac{q}{m}E_n\Delta t \tag{8}$$

$$x_{n+1} = x_n + v_n\Delta t \tag{9}$$

For this approximation to work, $\Delta t$ must be small.[4] Euler's method is known as a $1^{\text{st}}$ order method, and is far from perfect. It only takes into account the derivative at the earlier end of the time step; it is far better to try to estimate the average of the derivative across the whole time interval.

### 2.1.2   Predictor-corrector methods

We can improve Euler's method to become a $2^{\text{nd}}$ order method. We do this by taking a trial step forward:

$$\widetilde{v}_{n+1} = v_n + \frac{q}{m}E_n\Delta t \tag{10}$$

$$\widetilde{x}_{n+1} = x_n + v_n\Delta t \tag{11}$$

We then calculate the new electric field, $\widetilde{E}_{n+1}$ at the new (trial) position and velocity. We can then average the two electric fields to obtain an estimate of average of the electric field across the total time step. We then use this to calculate our new position and velocity.

---

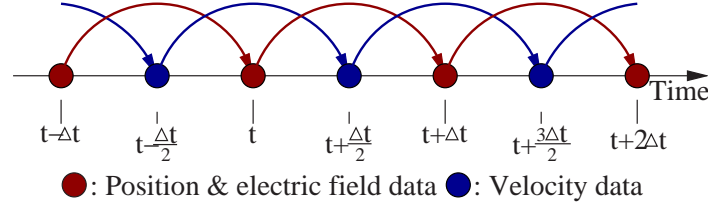[4]The effect of a large time step is shown in figure 2.8.1 of [1]:p33.

**Figure 2**: The leap-frog scheme. Each line represents an operation; when two lines overlap, the line underneath was performed first. Note that new position data is calculated by *leap-frogging* over known velocity data and vice-verse (electric field data is dependent on position data).

$$v_{n+1} = v_n + \frac{q}{m} \frac{E_n + \widetilde{E}_{n+1}}{2} \Delta t \tag{12}$$

$$x_{n+1} = x_n + \frac{v_n + \widetilde{v}_{n+1}}{2} \Delta t \tag{13}$$

It is obvious that this method is more computationally expensive than Euler's method, but more accurate. This method is known as the *Improved Euler's method* and is the simplest of a family of methods known as *predictor-corrector methods*. Runge-Kutta methods are often used to solve these problems and are closely related to predictor-corrector methods. As the order of the Runge-Kutta increases so does the computational expense. In practice the most popular is a 4th order Runge-Kutta.

As the calculation of the electric field from particle information is a non-trivial task,[5] Runge-Kutta and predictor-corrector methods represent a huge computational expense over Euler's method. So if we wish to increase the accuracy of this stage of the simulation, we should look for another method.

### 2.1.3   Leap-frog scheme

It is possible to achieve a 2nd order method with very little extra computational expense over Euler's method. As previously mentioned, we wish to calculate new velocity information using an average of the electric field (and also new position information using an average of the velocity) across the time step. The electric field halfway across the time step should be a good estimate (if the change is linear then it is exactly correct), it will only fail to be a good estimate when the simulator has set the time step to be too large.

We can implement such a scheme by 'staggering' the differential equations (8) & (9), so that the velocity is known at half integer time steps:

$$v_{n+\frac{1}{2}} = v_{n-\frac{1}{2}} + \frac{q}{m} E_n \Delta t \tag{14}$$

$$x_{n+1} = x_n + v_{n+\frac{1}{2}} \Delta t \tag{15}$$

Figure 2 illustrates the scheme. Schemes such as this one are known as *leap-frog schemes* as in the calculation of new position information we 'leap-frog' over known velocity information.

When implementing this scheme into a PIC method, one must recalculate the initial velocities of the particles so that they represent the hypothetical velocities of the particles half a time step before the start of the simulation.

---

[5]It forms three out of four of the operations of the main program loop, as shown in figure 1.

There are some disadvantages to this scheme. In the methods described earlier there was no reason for the time step to remain constant, allowing for 'real-time' simulations[6]; also an *adaptive time step* can be implemented for Runge-Kutta methods in order to maximise accuracy. In the leap-frog scheme however, the velocity must be *time-centred* (known halfway through the time-step). To change the time step mid-simulation requires the cumbersome re-calculation of the velocity to keep it time-centred.

Another disadvantage is the inconvenience of only knowing the velocities of the particles half-way through the time step. For instance, in calculation of the kinetic energy, $T_n = \frac{1}{2}mv_n^2$, which we would like to know at the same time as we know the potential energy, we need to find some way to calculate the velocity halfway through its natural time step. We can achieve this by simply making a half time step on the velocity information, but this is computationally expensive; alternatively we can store the previous value of the velocity and average the new and old information. In fact, for the kinetic energy, the following approximation works well:

$$T_n = \frac{1}{2}mv_{n-\frac{1}{2}}v_{n+\frac{1}{2}} \tag{16}$$

### 2.1.4   Comparison of methods

Of the three methods for time-integration described above, one must be chosen for the program. The leap-frog scheme is no more computationally expensive than Euler's method, aside from the initial half time step backwards, which is only performed once, and so is of little importance considering that a full simulation could be formed of hundreds or thousands of time steps. Euler's method whilst being simple and intuitive, rarely works well in practice, making the inconveniences of the leap-frog scheme a price worth paying. Our choice of time-integrator is therefore reduced to a choice between a predictor-corrector method and the leap-frog scheme.

As previously mentioned, the implementation of a predictor-corrector time-integrator in a PIC simulator would require the recalculation of the electric field, greatly increasing the computational cost, for example a $4^{\text{th}}$ order Runge-Kutta would require the electric field to be calculated 4 times per time step. For this reason the leap-frog scheme has been chosen for the program.

## 2.2   'Weighting' of the charge density and electric field

The 'weighting' stages of our program refer to the conversion between particle and grid based information. The electric field could be calculated without a grid through Coulomb's law, by adding together the contribution of each and every particle. However, given that plasma simulations involve a great many particles, this is a very computationally expensive method. Also the literal treatment of point-particles in one dimension would mean that particles with the same charge could not move past one another, it is more interesting to simulate a 1-dimensional projection of a real plasma. Projecting our particles on to a grid means that our particles behave as 'clouds' of charge rather than point-particles, and so can move past each other.

There are two weighting stages in our main program loop. The first is the charge density weighting, where the charge density is built up on the grid using the particle positions and charges. The second is the electric field weighting, where the electric field at each particle is estimated using the grid-based electric field information.

---

[6]The simulation would proceed at the same rate, regardless of the power of the machine it runs on. On slower machines accuracy is sacrificed rather than speed. Such simulations can be useful for graphical applications which aim to give insight to the user on the behaviour of plasmas, rather than being intended for serious research.
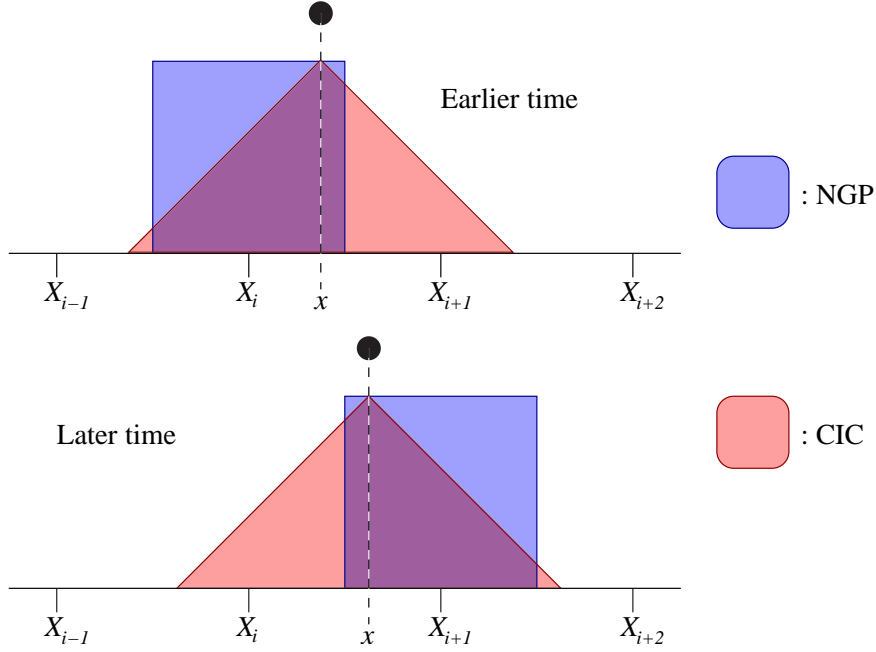
**Figure 3**: The effective particle shape of a particle at position $x$ at two different times.

There are many schemes for performing the weighting, each with their own 'order'. The higher the 'order', the more sophisticated the weighting, and the more computationally expensive it is. Therefore a compromise must be made. Also there are good reasons not to use weighting schemes of too great an order.

The source-code used for the charge density and electric field weighting can be found in the appendix, in sections B.2 and B.3 respectively.

### 2.2.1   $0^{\text{th}}$ order - Nearest Grid Point

*Nearest Grid Point*, or *NGP*, is the simplest of the weighting schemes. The charge of a particle is only assigned to a single grid point, the nearest grid point to the particle. The effective shape of the particle is rectangular with width $\Delta x$ as shown in figure 3, the charge density attributed to the nearest gridpoint, $\rho_i$, is:

$$\rho_i = \frac{q}{\Delta x} \tag{17}$$

Where $q$ is the charge of the particle and $\Delta x$ is the spacing between grid points.

Similarly, when performing the electric field weighting, the particle is exposed to the electric field at the nearest grid point to its current location.

One of the main problems with the NGP scheme is its poor handling of particles between grid points. If we consider a lone particle moving along the grid, the charge density will not change continuously but rather the single peak will jump discretely from one grid point to the next[7], leading to a noisy charge density and electric field. This problem also means that some plasma behaviour, such as cold plasma oscillations, cannot be simulated using the NGP weighting scheme. However NGP is the fastest of all the weighting schemes.

---

[7]This behaviour can be seen in figure 3 by comparing the two times.

### 2.2.2   1st order - Cloud In Cell

In the *Cloud In Cell*, or *CIC*, weighting scheme each particle is associated with two grid points. The effective shape of the particle is triangular as shown in figure 3, and the width is $2\Delta x$.

The charge density attributed to each grid point where $\rho_i$ is the nearest grid point to the left of the particle and $\rho_{i+1}$ is the nearest grid point to the right, is:

$$\rho_{i+1} = \frac{q}{\Delta x} \left[ \frac{x - X_i}{\Delta x} \right] \tag{18}$$

$$\rho_i = \frac{q}{\Delta x} \left[ \frac{X_{i+1} - x}{\Delta x} \right] = \frac{q}{\Delta x} \left[ \frac{X_i + \Delta x - x}{\Delta x} \right] \tag{19}$$

Where $q$ is the charge of the particle, $x$ is the position of the particle, $X_i$ is the position of the $i^{\text{th}}$ grid point and $\Delta x$ is the spacing between grid points.

These can be optimised to remove unnecessary mathematical operations:

$$\rho_{i+1} = \frac{q}{\Delta x} \left[ \frac{x}{\Delta x} - i \right] \tag{20}$$

$$\rho_i = \frac{q}{\Delta x} - \rho_{i+1} \tag{21}$$

(Assuming that $X_0 = 0$.)

When performing the electric field weighting, the field at the particle is given by:

$$E = \left[ 1 - \left( \frac{x}{\Delta x} - i \right) \right] E_i + \left[ \frac{x}{\Delta x} - i \right] E_{i+1} \tag{22}$$

The CIC weighting scheme produces far less noise than the NGP scheme as each particle is associated with two grid points which allows for improved handling of particles between grid points, and the movement of particles across the grid. The width of the effective particle shape is twice that of the NGP scheme and so even further from a point-like particle.

### 2.2.3   Higher order

It is possible to implement higher order weighting schemes such as *quadratic splines* (2nd order) and *cubic splines* (3rd order). However whilst these schemes increase smoothness and so reduce non-physical noise, they come at the price of increased computational expense. Also as the order of the weighting increases so does the width of the effective particle shape[8] meaning that the particles behave less like point-particles.

### 2.2.4   Mixed schemes

It is possible to use different weighting schemes for the different weighting stages in the main program loop (see figure 1). However, using the same weighting scheme for each has the benefit of conserving momentum. For instance, consider the possible self-force a particle could be exposed to when employing different weighting methods. Despite this problem, mixed schemes represent

---

[8]Figure 3 shows this effect for the low order weighting schemes.

an opportunity to save on computational expense.[9] They are also used in simulations that aim towards greater conservation of energy.

If one does implement a mixed scheme then one must be sure that the order of the electric field weighting is not greater than that of of the charge density weighting. As such, a scheme leads to unstable motion due to the self-force on the particles. If, on the other hand, the order is lower than that of the charge density then the self-force is such to induce simple harmonic oscillations which, whilst being non-physical, are at least stable.[10]

## 2.3   Electric field calculation

At the beginning of this stage, we have the charge density, $\rho_n$, defined on the grid and we wish to calculate the electric field, $E_n$ on the grid. The equation we need to solve is Poisson's equation:

$$\boldsymbol{\nabla}.\boldsymbol{E} = \nabla^2\phi = \frac{\rho(x)}{\epsilon_0} \tag{23}$$

Where, $\boldsymbol{E}$ is the electric field, $\phi$ is the electrostatic potential, $\rho(x)$ is the charge density and $\epsilon_0$ is the permittivity if free space.

Which in one dimension becomes:

$$\frac{dE_x}{dx} = \frac{d^2\phi}{dx^2} = \frac{\rho(x)}{\epsilon_0} \tag{24}$$

(From here on we will drop the $x$ sub-script.)

We can solve equation (24) using Fourier transforms. In order to simplify the algebra and to aid understanding we will define Fourier transforms in terms of operators:

$$\widetilde{f}(k) = \frac{1}{\sqrt{2\pi}} \int [f(x)]\, e^{ikx} dk = \widehat{FT}\left(f(x)\right) \tag{25}$$

$$f(x) = \frac{1}{\sqrt{2\pi}} \int \left[\widetilde{f}(k)\right] e^{-ikx} dx = \widehat{FT^{-1}}\left(\widetilde{f}(k)\right) \tag{26}$$

The electrostatic potential, $\phi_n$ on the grid is given by:[11]

$$\phi_n = \widehat{FT^{-1}}\left(-K^{-2}\widehat{FT}\left(\frac{\rho_n}{\epsilon_0}\right)\right) \tag{27}$$

Where $K$[12] is a finite difference term that takes the place of the wave-number, $k$ and is defined as:

$$K = k.sinc\left(\frac{k\Delta x}{2}\right) \tag{28}$$

For simplicity and to reduce computational expense we set $\epsilon_0 = 1$. Our method to calculate the electrostatic potential can now be summed up as a series of steps:

---

[9]Compare a CIC/CIC scheme to a CIC/NGP scheme for example.

[10]This issue is derived and discussed in greater detail in section 5-2-4 of [2]:p127-128.

[11]Proof is provided in section A of the appendix.

[12]The role of $K$ is discussed in greater length in appendix B of [3]:p431-436.

1. Fourier transform charge density.
   $$\rho_n \Rightarrow \widetilde{\rho_m}$$

2. Divide $\widetilde{\rho_m}$ by $(-K^2\epsilon_0)$.
   $$\widetilde{\rho_m} \Rightarrow \widetilde{\phi_m}$$

3. Inverse Fourier transform to obtain electrostatic potential.
   $$\widetilde{\phi_m} \Rightarrow \phi_n$$

Special care must be taken when $k = -K^2\epsilon_0 = 0$, in this case we set $\widetilde{\rho} = \widetilde{\phi} = 0$, this creates a 'neutralising background'.

Once the electrostatic potential has been calculated we must calculate the electric field itself by solving:

$$\boldsymbol{E} = -\boldsymbol{\nabla}\phi \tag{29}$$

Which in one dimension becomes:

$$E_x = -\frac{d\phi}{dx} \tag{30}$$

(From here on we will drop the $x$ sub-script.)

To solve this differential equation we formulate the following finite difference equation and iterate over the entire grid:

$$E_n = \frac{\phi_{n-1} - \phi_{n+1}}{2\Delta x} \tag{31}$$

The use of Fourier transforms means that the space in which the plasma is defined is 'periodic', meaning that if a particle moves out of the left hand side of the spacial grid it will appear on the right hand side. Special care must be taken when moving the particles to ensure that this happens.

This program uses an implementation of the Cooley-Tukey *Fast Fourier Transform* (*FFT*) algorithm to perform its Fourier transforms provided that the user sets the number of grid points to a power of 2, otherwise a standard (slow) algorithm is used instead. Given that most simulations use an order of magnitude fewer grid points than particles or less, this shouldn't be a problem.

The source-code for the electric field calculation can be found in section B.4 of the appendix.

# 3   Case studies

## 3.1   Cold-plasma oscillations

### 3.1.1   Method

A cold stationary plasma of charged particles is set up and perturbed in any[13] mode.

It is important to define what we mean by 'perturbed'. Consider a plasma consisting only of stationary, equally-spaced electrons. All the electrons are in their equilibrium positions and in the absence of any applied electric field they will remain stationary. Now consider slightly perturbing a single electron from its equilibrium position; it will now oscillate about its equilibrium position with amplitude equal to that of the initial displacement and frequency given by:

---

[13]Usually one of the lower frequency modes to minimise any non-physics introduced by the spacial grid.

$$f_p = \frac{\omega_p}{2\pi} = \frac{q}{2\pi}\sqrt{\frac{N}{L\epsilon_0 m}} \tag{32}$$

Where, $\omega_p$ is the plasma frequency, $q$ is the charge on a particle, $N$ is the number of particles, $L$ is the total length of grid and $m$ is the mass of a particle.

Now consider perturbing all the electrons, where a particle's position is now given by:

$$x = x_0 + x_1.cos\left(\frac{2\pi n}{L}x_0\right) \tag{33}$$

Where, $x$ is the particle position, $x_0$ is the particle's equilibrium position, $x_1$ is the amplitude of the displacement,[14] $n$ is the mode of excitation (a positive integer) and must be low enough that the grid resolves the spacial oscillations sufficiently well.

Such a perturbation will result in sinusoidal (not co-sinusoidal) density and in turn charge density.[15] The electrons will still oscillate about their equilibrium positions as before. This leads to behaviour in the charge density (and electric field) that is similar to that of a plucked string. As the electrons oscillate in phase with one another (but with differing amplitudes) temporal oscillations can be seen in the kinetic and potential energies of the plasma. From these it is possible to measure the frequency of the oscillations.

As previously mentioned, the amplitude of the perturbation, $x_1$ is usually very small compared with the particle spacing, and as we simulate with more particles than grid points, it is even smaller compared to the grid spacing, $\Delta x$. As we have seen in figure 3, $0^{th}$ order weighting (NGP) cannot accurately resolve small $(< \Delta x)$ differences in position. Therefore weighting of order 1 or higher must be used.

The simulation was conducted with a grid length of $2\pi$, 512 grid points, a time step of 0.001, $1^{st}$ order weight, 1024 particles, a charge-to-mass ratio of -1, and a perturbation amplitude of 0.001.

### 3.1.2  Analysis

Figure 4 shows the oscillations in the electric field for the first two modes. One full period is displayed for each and it can be seen that the initial and final electric fields are identical. Whilst these graphs aid understanding of cold plasma oscillations, it is difficult to measure the frequency of the oscillations from them. A better tool is to measure the frequency of the oscillations in the kinetic energy, as shown in the left-hand graph of figure 5.

In the case of point-particles the cold plasma oscillations will always have a frequency of $f_p$ as defined in equation (32). However this is not the case for finite size particles, where the frequency will change with wavenumber, $k$.[16] The right-hand graph of figure 5 shows the dispersion relation when using $1^{st}$ order weighting (CIC), both as predicted by theory and the programs output. After $k = 75$ the simulation breaks down as the spacial grid cannot adequately resolve the oscillations in the charge density, but up to this point the output is consistent with the theory.

---

[14]Usually very small, compared to the spacing between particles, if it is set to high the particle displacements will overlap and true sinusoidal waves will not be seen.

[15]Figure 5-2c in [3]:p84 shows the effect of such a perturbation on quantities such as the electric field.

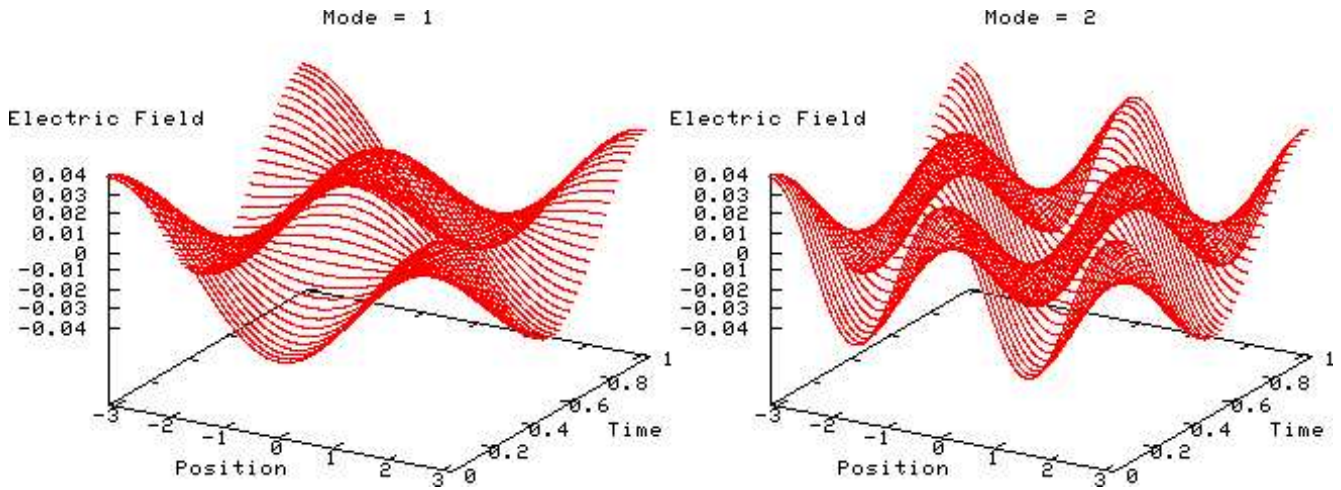[16]This is derived and discussed in greater detail in section 4-6 of [3]:p65-67.

**Figure 4**: The electric field at various times showing cold plasma oscillations.
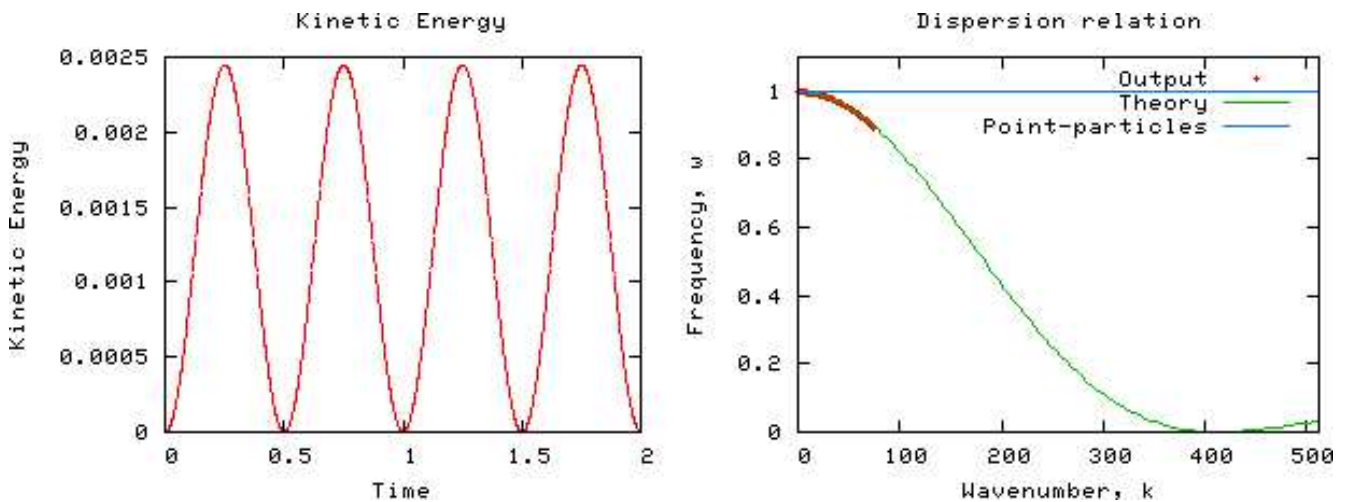


**Figure 5**: The left-hand graph is a kinetic energy graph of cold plasma oscillations for a given excited mode. The frequency of the energy oscillations is double the frequency of the cold plasma oscillations. The right-hand graph is the dispersion relation for cold plasma oscillations with 1st order (CIC) weighting.
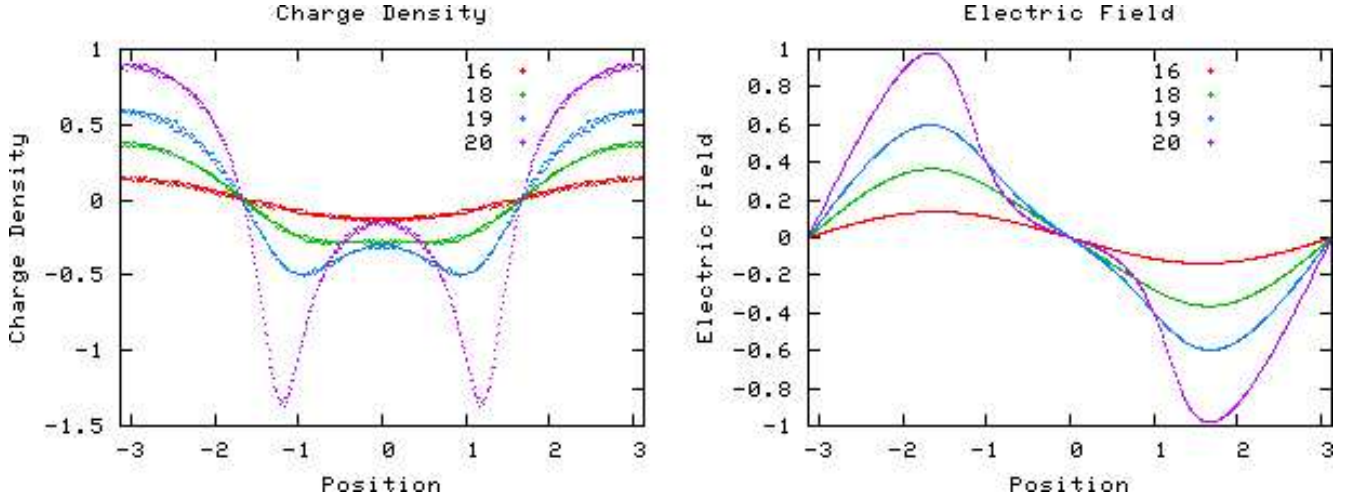
**Figure 6**: The charge density and electric field at various times as the two-stream instability builds up.

## 3.2    Two-stream instability

### 3.2.1    Method

Two initial beams of charged particles (electrons for example) are set up with equal and opposite velocities. The beams are perturbed (as defined in section 3.1.1) slightly in the mode of linear growth. The simulation used a drift velocity of $\pm 1$, a charge-to-mass ratio of -1, a plasma-frequency, $\omega_0$, of 1 and a grid length of $2\pi$, leading to a mode of linear growth of 1.

The instability was simulated using 512 grid points, a time step of 0.1, 1st order weighting, 2048 particles in each beam, and a perturbation amplitude of 0.0001.

### 3.2.2    Analysis

Figure 6 shows the charge density and the electric field for various times as the instability builds up. The first mode can be seen to build up until around $t = 18$, when the second mode also starts to appear, its growth being due to non-linear coupling. The two troughs that can be seen in the latest time on that graph rapidly become very deep and represent the 'bunching' of the particles.

Figure 7 shows the bunching effect. In the left-hand graphs, which show the instability for 2 streams of electrons, the bunches are found to be as far apart from each other as possible (remember that the space is periodic), note also that one beam dominates each bunch. In the right-hand graphs, which show the instability for 1 stream of electrons and 1 stream of positrons, the bunches are still dominated by one stream or the other (the charge density graph shows this particularly well), but this time the bunches are found very close to one another; this is as we expect, as the oppositely charged bunches attract.

Figure 8 contrasts the initial and final phase-space plots. In the first graph the two beams can be observed, with velocities +1 and -1. In the second graph, which is representative of the stable state the instability settles into, little remains of the original beams and many of the particles now follow oscillatory motion of varying amplitude (the 'swirl' in the centre of the graph is evidence of this). The plasma has heated up considerably since the start of the simulation (the plasma was originally cold), as can be seen from the wide spread of velocities. A small proportion of the particles have been accelerated by the instability rather than decelerated into oscillatory motion, with some particles travelling at around three times the speed of the original beams.
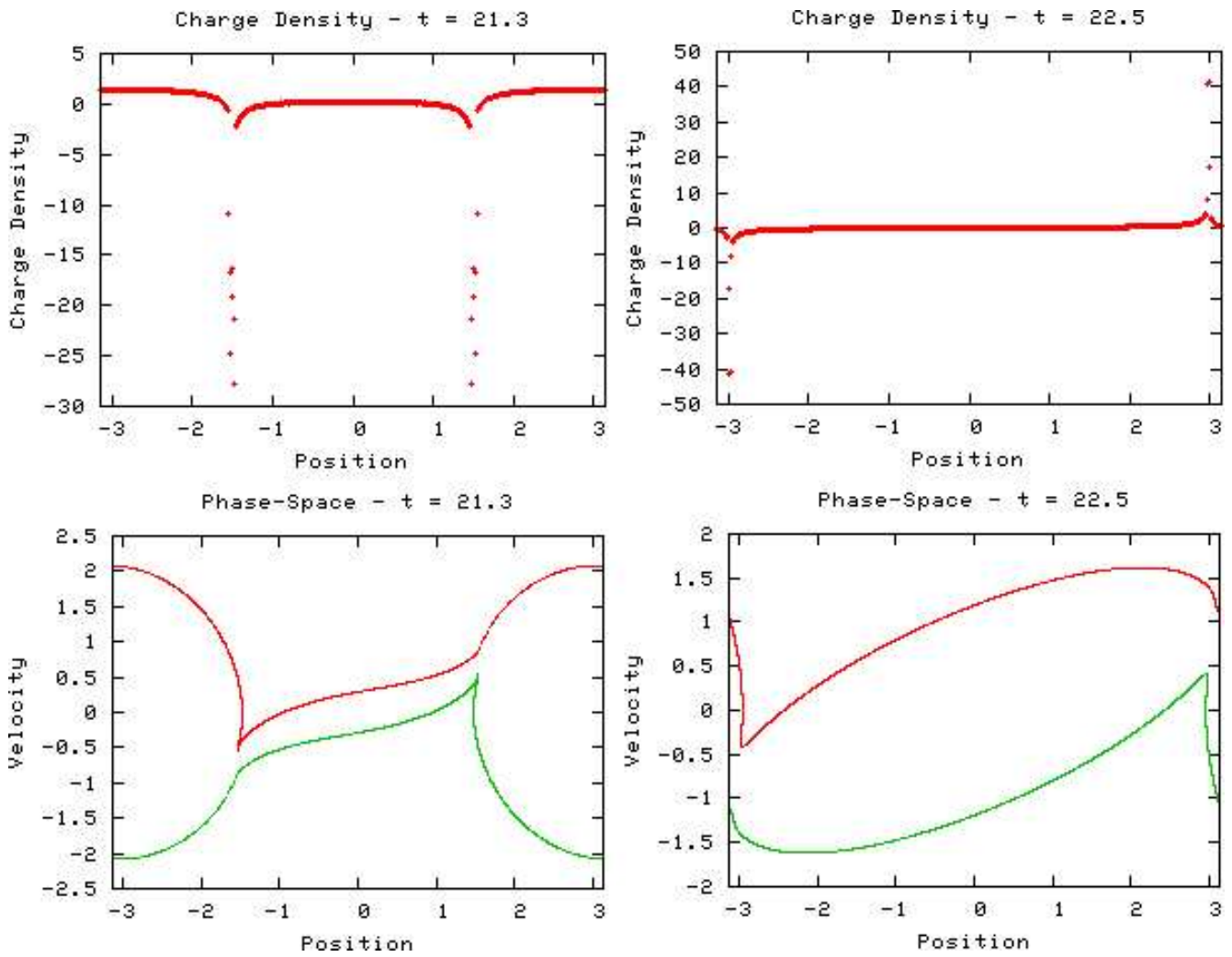
13

**Figure 7**: Bunching of particles in the two-stream instability. The graphs on the left-hand side show bunching for 2 beams of electrons, whereas the graphs on the right-hand side show bunching for 1 electron beam and 1 positron beam.
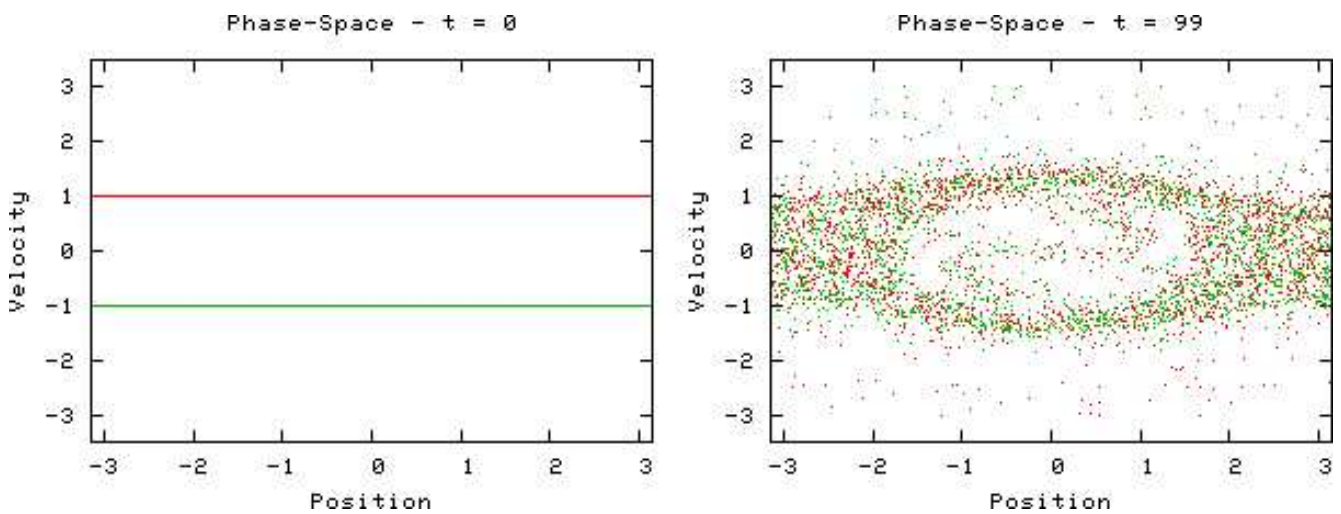


**Figure 8**: Initial and final state phase-space graphs of the two-stream instability.
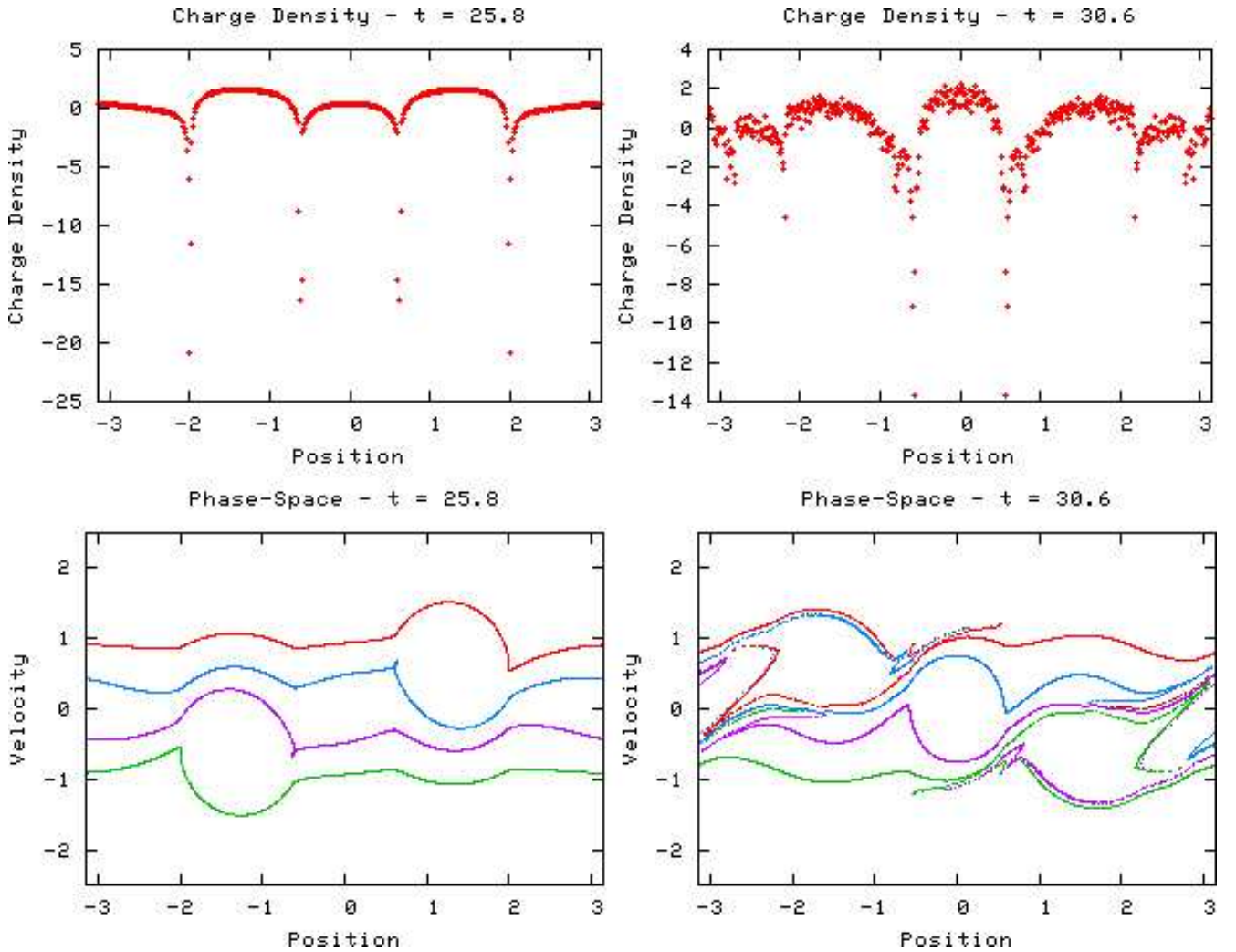
**Figure 9**: Bunching in the four-stream instability. Unlike the two-stream instability, the four-stream instability has several bunching stages.

## 3.3   Four-stream instability

### 3.3.1   Method

The two beams from the two-stream instability are set up as before, and two extra, slower, unperturbed beams are added.

The simulation was conducted with almost the same initial conditions as the two-stream instability, except for the extra beams, which had a drift velocity of $\pm 0.3$. Also the number of particles in each beam was reduced to 1536, and the amplitude of perturbation of the fast beams was increased to 0.0002.

### 3.3.2   Analysis

The four-stream instability is related to the two-stream instability as is evident from the phase-space graphs. Though unlike the two-stream instability it has more than one main bunching stage in its evolution. Figure 9 displays the first two bunching stages. The beams first interact most strongly with the other beam that is travelling in the same direction, ie. the fast beams interact with the slow beams, as is shown in the left-hand graphs of figure 9. After this the two slow
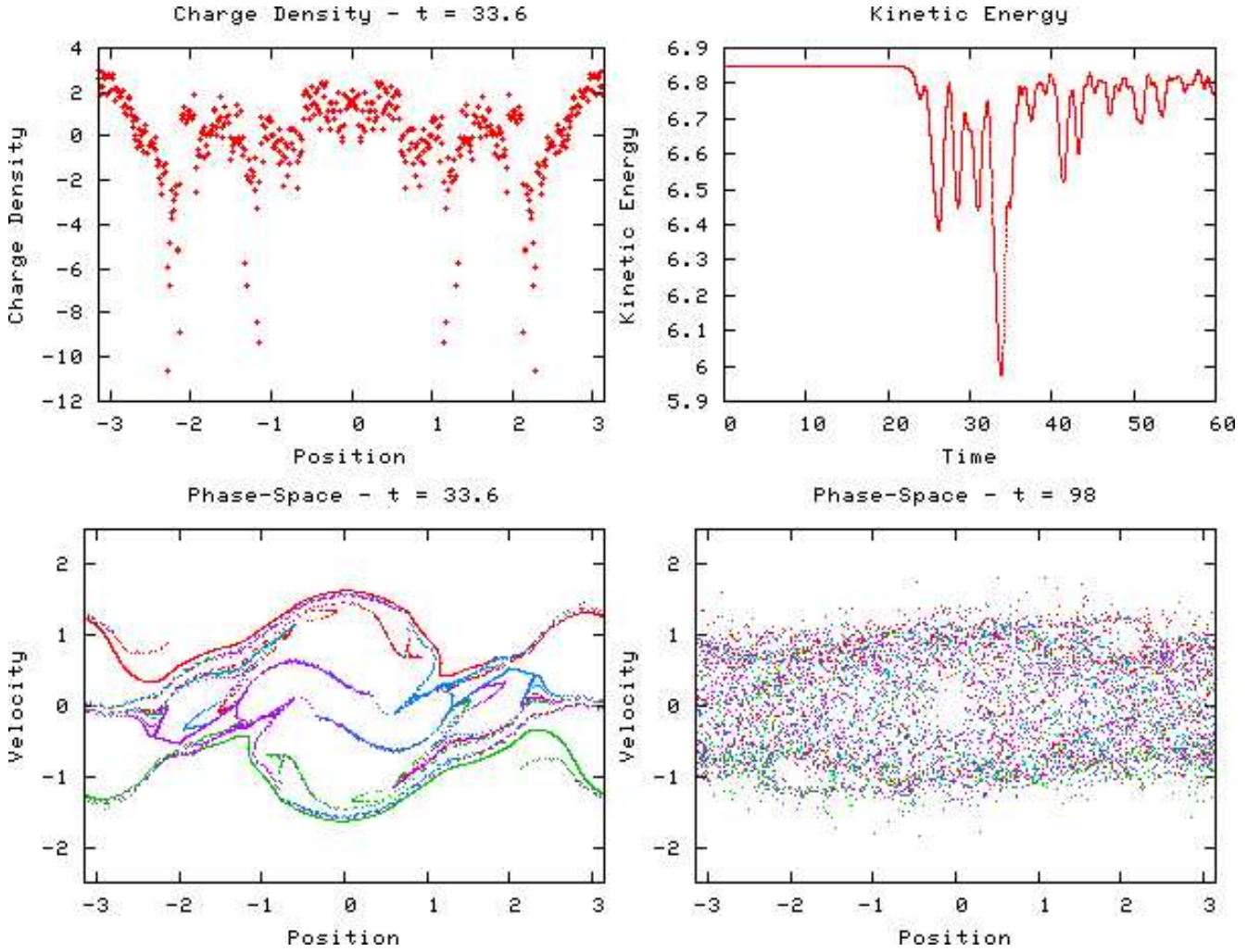
**Figure 10**: The left-hand graphs show another bunching stage in the four-stream instability. The top right graph shows kinetic energy of the plasma throughout the simulation. The bottom right graph shows the stable state the instability settles into.

beams interact as is shown in the right-hand graphs. The interactions appear to be mini-two-stream instabilities. As we expect, the bunches keep their distances from one another, due to their negative charge. At the first bunching stage the bunches are almost evenly distributed across the space. The remnants of the original phase-space vortices can be seen in the second phase-space graph and they appear to contribute to the second bunching stage.

Another bunching stage can be seen in in the left-hand graphs of figure 9, it is this bunching stage that is associated with the greatest drop in kinetic energy as can be seen from the top right graph. The bottom right graph is representative of the final state of the plasma and can be compared to the equivalent graph for the two-stream instability, found in figure 8. The most striking difference between the two is that whilst some particles in the two-stream instability were accelerated to three times their original speed, whereas in the four-stream instability no particles have more than double their initial speed.

As one might expect, if the slow beams are set very slow then the plasma behaviour approaches that of the three-stream instability.[17] On the other had if they are set to be close to the speed of

---

[17]The three-stream instability is not covered in this report. It uses the 2 beams from the two-stream instability and combines them with a static cold plasma.

the fast beams than the behaviour approaches that of the two-stream instability.

# 4    Conclusion

A computer program has been written to simulate plasmas in one dimension in the electrostatic limit, using a Particle In Cell (PIC) method. The method has been discussed in detail in the first half of this report. The validity of the program has then been tested through the study of various plasma behaviour, including cold plasma oscillations, the two-stream instability and the four-stream instability.

There is much that could be done to extend this project. Trivial but useful enhancements could be made to the program such as: the implementation of higher order weighting, support for warm initial plasmas and diagnostic improvements to make better use of Fourier analysis. A more challenging enhancement would be to add support for an applied magnetic field (this requires dealing with velocities in 2 dimensions rather than 1), which would allow the program to be used to study hybrid oscillations. The instabilities covered in this report could be investigated further, particularly if the new diagnostics for Fourier analysis were added. For instance, one could measure the growth rates of the modes in the two-stream instability and compare with theory. New instabilities could be observed such as the 3-stream instability.

Perhaps the most interesting extension to this project would be to write a new program to simulate plasmas electrodynamically in up to 3 dimensions. The weighting schemes for such a program would be more complicated, and rather than solving Gauss' law in the main program loop one must solve both Faraday's law and the Ampère-Maxwell law. A spacial leap-frog scheme could be implemented on the grid to aid their solution.[18]

The author's personal contribution to the project included writing the program from scratch and fixing its 'bugs', using the program to generate graphs of plasma behaviour, and studying the graphs, whilst he relied on his colleague, P. Hughes, to carry out background research to make this possible. Additionally the data for the dispersion relation graph in figure 5 was the result of P. Hughes work (involving over 100 invocations of the program). This project has been interesting as it has provided the author with the understanding required to implement more advanced PIC simulations, with more dimensions and which consider the effects of the magnetic field. The author would like to express his thanks to Dr P. Browning for her guidance over the course of this project.

# References

[1] S.H. Strogatz, "Nonlinear Dynamics and Chaos", Westview Press (1994)

[2] R.W. Hockney, J.W. Eastwood, "Computer Simulation Using Particles", IOP Publishing (1988)

[3] C.K. Birdsall, A.B. Langdon, "Plasma Physics via Computer Simulation", McGraw-Hill (1985)

[4] T. Tajima, "Computational Plasma Physics", Westview Press (2004)

[5] P. Browning, private correspondence

---

[18]As described in section 15-2 of [3]:p352-354.

# Appendix

# A   Proof of Fourier transform method

$$\frac{d^2 \phi_n}{dx^2} = \frac{\rho_n}{\epsilon_0}$$

$$= \frac{d^2}{dx^2}\left(\widehat{FT^{-1}}\left(\widetilde{\phi_m}\right)\right) = \widehat{FT^{-1}}\left(-K^2\widetilde{\phi_m}\right)$$

$$= \widehat{FT^{-1}}\left(\frac{\widetilde{\rho_m}}{\epsilon_0}\right) \tag{34}$$

$$\therefore \widetilde{\phi_m} = -K^{-2}\frac{\widetilde{\rho_m}}{\epsilon_0} \tag{35}$$

$$\Rightarrow \phi_n = \widehat{FT^{-1}}\left(\widetilde{\phi_m}\right) = \widehat{FT^{-1}}\left(-K^{-2}\frac{\widetilde{\rho_m}}{\epsilon_0}\right)$$

$$= \widehat{FT^{-1}}\left(-K^{-2}\widehat{FT}\left(\frac{\rho_n}{\epsilon_0}\right)\right) \tag{36}$$

$K$ is a finite difference term which replaces the wave-number, $k$, and is defined as:

$$K = k.sinc\left(\frac{k\Delta x}{2}\right) \tag{37}$$

# B   Relevant code

This section contains some of the most interesting source-code from the program. The full source-code is too long to be included here. It is written in C++.

## B.1   Main program loop

```cpp
int Simulation::Simulate()
{
 int     result    =  0;
 int     N         =  2 * Particles.Elements;
 int     i         =  0;
 int     j         =  0;
 double ThisTime   =  0;
 double dt         =  0;
 double test_dt    =  0;
 double yin[N];
 double yout[N];

 if(playing)
 {
  ThisTime = LastTime + TimeIncrement * 1000000; // Note the conversion to micro-seconds.
  dt = TimeIncrement;

  //
  // Move particles.
  //
  int n;
  i = 0;
  j = 0;
  while(i < Particles.Elements)
  {
```

```
    j = 2 * i;
    yin[j]     = Particles[i].r;
    yin[j + 1] = Particles[i].v;
    i++;
  }

  this->EulerLeapFrog(yout, yin, N, dt);

  i = 0;
  j = 0;
  double GridLength = GridFinish - GridStart;
  while(i < Particles.Elements)
  {
    j = 2 * i;
    while(yout[j] < GridStart) yout[j] += GridLength;
    while(yout[j] >= GridFinish) yout[j] -= GridLength;
    Particles[i].r = yout[j];
    Particles[i].v = yout[j + 1];
    i++;
  }

  //
  // Build charge distribution.
  //
  result += BuildChargeDensity();

  //
  // Calculate electric field.
  //
  result += BuildElectricField();

  //
  // Interpolate electric field.
  //
  result += InterpolateElectricField();

  //
  // All done.
  //
  LastTime = ThisTime;
  }

  return result;
}
```

## B.2   Charge density weighting

```
int Simulation::BuildChargeDensity()
{
  int    result = -1;                       // Return value of function, returns 0 on success.
  int    a;                                 // Grid point to the left of the particle.
  int    b;                                 // Grid point to the right of the particle.
  int    n        = 0;                      // Space index.
  int    N        = ChargeDensity.Elements; // Number of grid points.
  double dr       = GridIncrement;          // Space between grid points.
  double r;                                 // Position of particle.
  double rho;                               // Charge of particle / grid spacing.
  double rhob;                              // Charge density contribution from particle to point a.
  double rhoa;                              // Charge density contribution from particle to point b.
  double TotalCharge = 0;
  double NeutralisingBackground = 0;

  // Get Total Charge
  n = 0;
  while(n < Particles.Elements)
  {
    TotalCharge += Particles[n++].q;
  }
  NeutralisingBackground = -TotalCharge / (GridFinish - GridStart);

  // Wipe charge density clean.
  n = 0;
```

```
 while ( n < N)
 {
  ChargeDensity [ n++] = NeutralisingBackground ;
 }

 // Iterate over all particles .
 n = 0;
 while ( n < Particles . Elements )
 {
  r = Particles [ n ] . r ;
  rho = Particles [ n ] . q / dr ;
  assert ( GridStart <= r && r < GridFinish ) ;

  switch ( Weighting )
  {
   case 0:
     // 0th order weighting − NGP
     a = lround ( ( r − GridStart ) / dr ) ;
     if ( a >= N) a−= N;
     ChargeDensity [ a ] += rho ;
     break ;
   case 1:
     // 1st order weighting − CIC
     // Find grid points .
     a = ( int ) lround ( ( r − GridStart ) / dr ) ;
     if ( GridStart + a ∗ dr > r ) a−−;
     b = ( a < N − 1) ? a + 1 : a + 1 − N;  // Next point along , (minor complication at the end of
                                            //  the grid ).

     // Calculate charge contributions .
     rhob = rho ∗ ( ( r − GridStart ) / dr − a ) ;
     rhoa = rho − rhob ;

     // Assign density to grid points .
     ChargeDensity [ a ] += rhoa ;
     ChargeDensity [ b ] += rhob ;
     break ;
   default :
     CRASH( ) ;
     break ;
  }
  n++;
 }

 result = 0;

 return result ;
}
```

## B.3   Electric field weighting

```
int Simulation : : InterpolateElectricField ( )
{
 int result = 1;
 int i     = 0;

 while ( i < Particles . Elements )
 {
  Particles [ i ] . E = ElectricField ( Particles [ i ] . r ) ;
  i++;
 }

 result = 0;

 return result ;
}

double Simulation : : ElectricField ( double r )
{
 int a ;                         // The grid point to the left .
 int b ;                         // The grid point to the right .
 double result ;                 // The interpolated electric field at r .
```

```
double dr        = GridIncrement;  // The space between grid points.
double subterm;                    // Just used to speed this function up.

switch(Weighting)
{
 case 0:
  // 0th order weighting − NGP
  a = lround((r − GridStart) / dr);  if(a >= EField.Elements) a −= EField.Elements;
  result = EField[a];
  break;
 case 1:
  // 1st order weighting − CIC
  // Find nearest grid point to the left of our position, r.
  a = lround((r − GridStart) / dr);
  if(GridStart + a * dr > r) a−−;
  b = (a < EField.Elements − 1) ? a + 1 : a + 1 − EField.Elements;
  subterm = (r − GridStart) / dr − a;
  result = (1 − subterm) * EField[a] + subterm * EField[b];
  break;
 case 2:
  CRASH();
  break;
 default:
  CRASH();
  break;
}

 return result;
}
```

## B.4   Electric field calculation

```
int Simulation::BuildElectricField()
{
 int            result   = −1;
 int            n;                                   // Standard space index
 int            m;                                   // Fourier space index
 int            N        = ChargeDensity.Elements;   // Number of grid points
 double         dr       = GridIncrement;
 double         k;                                   // Wavenumber
 double         k_subterm = (2 * pi) / (N * dr);
 double         K;
 double         m_correct = −N / 2;
 Array<complex> ESPotentialK(N);
 Array<complex> Buffer(N);

 ESEnergy = 0;
 Transformer.Fast(ChargeDensityK.c_array(), ChargeDensity.c_array(), N, 1);
 m = 0;
 while(m < N)
 {
  k = (m + m_correct) * k_subterm;
  K = 2 * sin(k * dr / 2) / dr;
  ESPotentialK[m] = (K != 0.0) ? ChargeDensityK[m] / (K * K * epsilon_zero) : 0;
  ESEnergy += (0.5 * ChargeDensityK[m] * ESPotentialK[m].Conjugate()).Real;
  m++;
 }
 Transformer.Fast(ESPotential.c_array(), ESPotentialK.c_array(), N, −1);
 n = 0;
 while(n < N)
 {
  int a = (n == 0) ? N − 1 : n − 1;
  int b = (n == N − 1) ? 0 : n + 1;
  EField[n] = (ESPotential[a] − ESPotential[b]) / (2 * dr);
  n++;
 }

 result = 0;

 return result;
}
```