Particle Charge Determination in a Magnetized Dusty Plasma Flow

by

Dylan J. Funk

A dissertation submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
May 6, 2023

Keywords: Plasma Physics, Dust Charge, Dust Coupling

Abstract

Dusty plasmas consist of components typically found in a plasma (electrons, ions and neutral particles) as well as micrometer sized dust particles. The structural and dynamic properties of a dusty plasma system are governed by the dust particle charging state and the interaction of these particles with each other as well as the surrounding plasma and as a result of this, the knowledge of the exact charging state of the dust particles is very important. Theories such as Orbital Motion Limited (OML) and Allen-Boyd-Reynolds (ABR) theories as well as modified versions of these have been used to theoretically determine dust charge value in the past. Some recent experiments to determine particle charge indicate differences from theoretical models. This is particularly the case in the presence of a magnetic field. A molecular dynamic simulation has been created to study dust particle dynamics in the presence of a magnetic field. In a flowing system, a dust particle density gradient can build up due to the Lorentz force (similarly to the classical Hall effect). This dissertation will show multiple theories which have been developed to utilize this gradient to determine the particle charge in different coupling regimes. This is a new method for determining dust charge value which will be useful in many future experiments.

Due to the interactive properties of dust particles, coupling of dust particles becomes an important factor. The coulomb coupling parameter Γ is defined as a ratio of interactive potential energy to thermal kinetic energy. With this coupling parameter, regimes of liquid, solid and gas-like can be defined. Using a similar experimental system to that of a classical Hall effect, separate theories for calculation of dust charge were developed for the low coupling (”gas-like”) and high coupling (”solid-like”/crystalline) regimes. These two new methods will help increase the accuracy of dust charge determination in future dusty plasma research.

Acknowledgments

First, I'd like to thank my advisor, Dr. Uwe Konopka. It has been many years since I started on this long journey and many times I was afraid I wasn't going to find it's successful end. It was always through the great advice (and unmatched patience) from you that I am able to find this dissertation completed. Thank you for giving expert advice and allowing me to discover the project and research that fit me best. I hope you're as proud of this work as I am.

Next, I'd like to thank my secondary advisor, Dr. Edward Thomas Jr. Thank you for always being available for a question at a moments notice whether it was related to my research or not. Over the years this has become even more impressive as you have become more and more busy and ''wearing all the hats''.

I'd also like to thank my committee members, Dr. Stuart Loch and Dr. David Maurer. From your classes, I learned a great deal about physics in a way that made me even more interested. From your advice, I have always found it helpful and appreciated.

To my physics friends, it's been exciting to see you grow over the years and to those who haven't graduated yet, I look forwards to seeing you do so. Dr. Lori McCabe, Brandon Doyle, Ellie Williamson, Dr. Ami DuBois, Dr. Ivan Arnold and Dr. Spencer Leblanc, I am glad to have had you as my colleagues throughout the years and I look forward to seeing you at conferences in the future and to see how your careers progress. I know you'll all do great things. You already have.

To my mom and dad, thanks for always being there when I needed a place to go or to help me out of tight spots, I wouldn't have gotten here without you. To their partners, Bernard and Lisha, my parents made the right choice in choosing you both. And of course to my true family, my pets, I'd have failed a thousand times without you fluffy babies. I know you'd be proud of me if you could read.

Lastly, I want to thank my friends, my second family, Julia, Amber, Kat, Dan, Steve, Kayla, Kelly, Gabriela, Dayanna, Claire and many many more. You know as well as I how hard this has been, I couldn’t have done it without you all. There’s no words to describe how grateful I am for every last one of you. Thank you, thank you, thank you.

Contents

List of Figures

List of Tables

List of Abbreviations

Δ	Average inter-particle spacing
Γ	Coupling coefficient
γ	Dust drag coefficient $[s^{-1}]$
γ_m	Modified drag coefficient $[kg \cdot s^{-1}]$
λ_D	Debye length
$\overrightarrow{B}$	Magnetic field
$\overrightarrow{E}$	Electric field
ϕ_s	Surface potential
c_s	Sound speed
f	Force density
P	Pressure

Plasma Component Parameters

$k = e, n, i, d$	k represents plasma components (electron, neutral, ion or dust)
m_k	Component mass
n_k	Component density

Q_k, q_k	Component charge value
R_k, r_k	Component radius
T_k	Component temperature

Notation

$< x >$	Average of a variable
$\overrightarrow{x}$	Vector notation

Physics Constants

c	Speed of light in a vacuum
e	Elementary charge ($e = 1.602 * 10^{-19}$ C)
k_B	Boltzmann Constant ($k_B = 1.381 * 10^{-23}\ J \cdot K$)

Chapter 1

Introduction

1.1 Motivation and Scope

The goal of this dissertation is to develop a new approach to the problem of dust particle charge determination. This will be done by first showing previous theoretical models and analytical approaches which have been used to determine dust charge as well as the experimental results using these methods. Next, a new theory will be introduced for the case of low inter-particle coupling (”gaseous”-like dusty plasma) as well as a new theory for the case of high inter-particle coupling (”solid”-like dusty plasma). A new molecular dynamic simulation will then be introduced which has been created in order to test this theory as well as the validity of this simulation for this research. The results of this simulation will be shown as well as the analytical tools which have been created in order to test these theories. Lastly, a future experiment will be proposed which can be utilized to test these theories further.

1.2 Plasmas

The field of plasma physics is one which has existed for almost one hundred years. Some of the most important terms in plasma physics such as ”plasma” and ”sheath” were originally coined by Irving Langmuir in 1928 when he wrote ”...Except near the electrodes, where there are *sheaths* containing very few electrons, the ionized gas contains ions and electrons in about equal numbers so that the resultant space charge is very small. We shall use the name *plasma* to describe this region containing balanced charges of ions and electrons” [1]. Since this time, the field of plasma physics has advanced significantly to become one of the most interesting and relevant fields of physics. Plasmas are commonly found in nature in stars, in nebulae as well as on Earth in the form of the Aurora Borealis as shown in figure 1.1. Knowledge of plasmas in these system gives us information about the Earth and the universe which would have been otherwise difficult to interpret. It has been shown, for example, that plasma characteristics observed by low-altitude satellites can be used to map the upper atmosphere of the Earth [2].

Plasmas are known as the ”fourth state of matter” and are created by taking a gas (for example, Argon or Helium) and adding a large amount of energy into the system. This energy input is accomplished through various different methods in practice. In a laboratory system, energy is typically input into the system via an applied direct current (DC), alternating current (AC) or radio frequency (RF) current. As this energy increases, the electrons separate from neutral particles which forms a standard plasma. This standard plasma consists of positively charged ions, negatively charged electrons and non-energized neutral particles which are seen in plasma systems [3].

One common property of plasmas is the release of excess energy in the system in the

Figure 1.1: An example of plasma in the Aurora Borealis above North America (Image from NOAA space weather).

form of light. This is referred to often as the ”plasma glow.” The characteristics of this light emission can be analyzed to find information about the plasma itself. This is typically done by using a spectrometer to analyze the wavelengths of light being emitted from the energetic electrons. Specific wavelengths of light will then correspond to specific energetic decays as the bound electrons drop to lower energy levels. The difference between these energy levels and therefore the emitted light is unique and can be used to determine information about the plasma itself and which atoms make up the plasma. The intensity of the spectral lines emissions can are also used to diagnose plasma parameters such as temperature and density. The wavelengths help identify the emitting elements and are used to diagnose plasma flows and velocities. This technique, known as optical emission spectroscopy has been used to analyze many different types of plasmas [4] and is one of many techniques to determine plasma parameters in a system. An example of laboratory plasma systems are shown in Figure 1.2.

In nature, plasmas which occur are often times influenced by external contaminants not included in our definition of a ”standard plasma”. Any new particles in the system will change the nature of the plasma. This new system which contains more than the three (ions, electrons and neutrals) plasma components is commonly referred to as a ”complex” or ”dusty” plasma.

1.3 Dusty Plasmas

A dusty plasma (sometimes called a complex plasma) is typically defined as a four component plasma composed of ions, electrons, neutral particles and an additional component. There is no set definition for the specifics of this fourth component however it typically refers to a component of larger scale compared to other components which exists within the system. This fourth component is referred to as a ”dust particle” and hence the name ”dusty plasma”. Dusty plasmas are present in nature in environments such as the upper atmosphere, volcanic clouds, the interstellar media, planetary nebulae and planetary rings [5, 6]. These dust particles can have a wide range of varying shapes, sizes and material properties each of which will affect the way a dusty plasma behaves.

In terms of size, dusty plasmas can be made from particles on the order of a few nanometers [7] up to almost a millimeter in size [8]. It has also been shown that it is possible for a dusty plasma to have a distribution of many sizes within the same system [10, 9, 11].

Dust particles can be made of many different types of many different materials such as melamine formaldehyde [12], silicon [8], ice (as is the case for some of saturns rings) [14, 13],

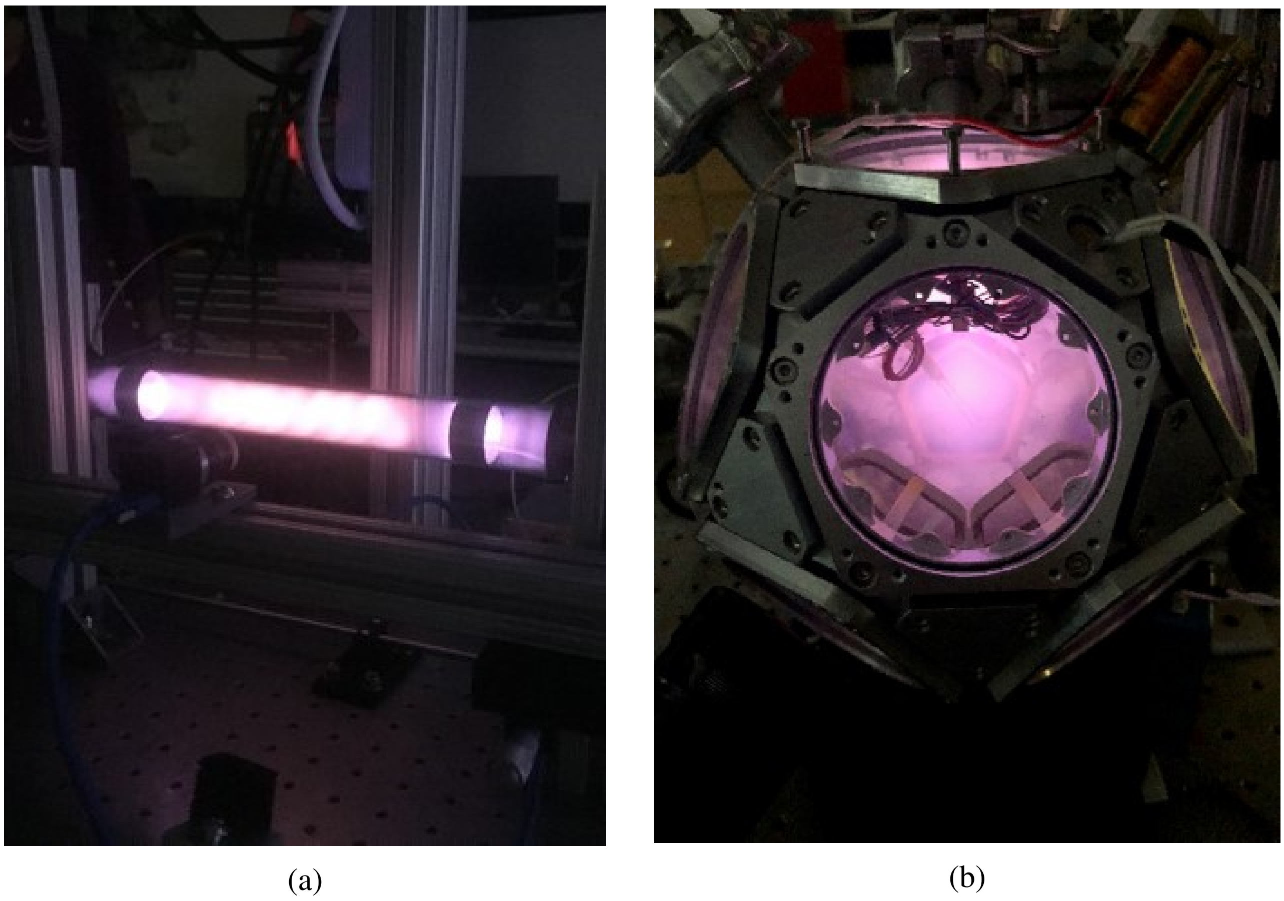


Figure 1.2: Two examples of laboratory plasmas (a) An example of a typical laboratory DC discharge chamber and (b) an example of a laboratory RF generated plasma in the dodecahedral plasma chamber at Auburn University). Both of these plasmas are glowing with a "pink" hue, analysis of the frequency of this light can show what gas is inside of these chambers. In both of these cases, Argon is used.

to the much more complicated systems of microorganisms [15] as well as various metals such as tungsten which has been used to study ablation in fusion reactors [16].

Beyond the size and material of the dust particles is their shape. While many of these particles are assumed to be spherical [12], there are cases where this is not the case. Some examples of this are particles which result from particle growth [8]. These non-spherical dusty plasmas have even been theorized to exist within Saturn's rings [13]. The assumption of a spherical shape however is generally accepted to have minimal effect for cases where the particle size $\ll$ Debye length.

As a result of this wide range of parameter spaces it is useful to have a definition of a

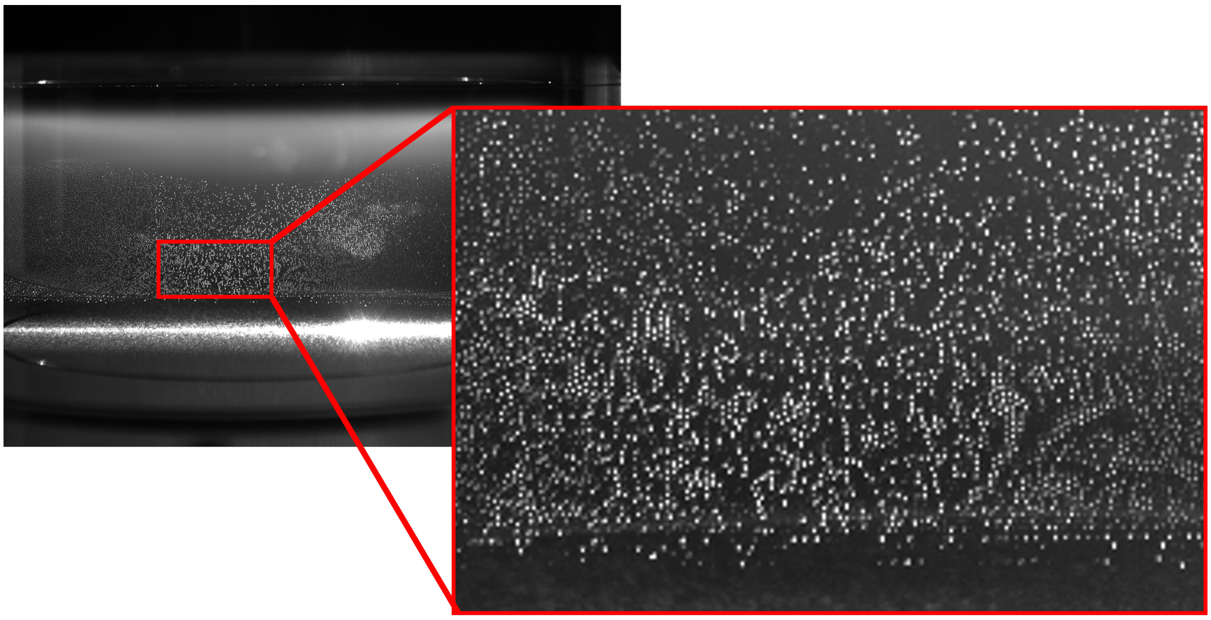


Figure 1.3: An example of a dusty plasma created by in a Zyflex chamber (Image provided by Uwe Konpoka). Dust particles shown in zoomed in portion beginning to form structure.

”common” laboratory dusty plasma. In a common laboratory dusty plasma, particles have a radius on the scale of a micrometer and are made of plastic such as melamine-formaldehyde (which has a mass density of 1510 kg/m^3 [17]).

One major benefit of a dusty plasma system is that it can be used as a macro scale system which emulates more complex micro scale systems (such as electrons and atoms) due to the charged nature of the particles. Some of the behavior which has been observed in dusty plasmas is similar to that of various states of matter through the interaction and coupling between particles [18].

1.3.1 Dust Charging

As the dust particles interact with the surrounding plasma, they accumulate charge based on their physical properties such as characteristic size, shape and the material properties of the dust. The determination of an accurate value for this quantity has proven to be a very difficult task. While many methods have been postulated for calculating dust charge value, there are many circumstances such as the presence of magnetic fields [19] in which the accuracy of these theories decrease. In the following section, various theoretical, computational and experimental methods that have been used to approximate, calculate or measure dust charge value in a system will be introduced.

The first introduction to dust charge accumulation on a dust particle was shown in the 1926 paper by Mott-Smith and Langmuir [20] as well as (coincidentally) the paper by Langmuir [1]. This was done by first assuming the particle will behave similarly to a small spherical probe. Next, showing the currents incident upon a ”small” spherical probe in a plasma with the assumption that the changes to the surrounding plasma did not expand beyond the sheath region. This ”small” spherical probe is defined as dust radius, $r_d \ll$ Debye length λ_D. Debye length is defined in equation 1.1 and can be described as a charge carrier’s net electrostatic effect and the approximate length to which this electrostatic effect persists [21].

$$\lambda_D = \sqrt{\frac{k_B T_e}{4\pi n_e e^2}} \tag{1.1}$$

where n_e is plasma electron density, T_e is the electron temperature, k_b is Boltzmann’s constant and e is electron charge.

This theory of charging was further expanded upon with Allen-Boyd-Reynolds (ABR) theory [22] which assumes radial motion of the ions and electrons about the dust particle as well as Orbital-motion limited (OML) theory [23, 24, 3, 17] which assumes orbital motion of the incident ions about the dust particle.

Both OML and ABR theories calculate the ion and electron current incident upon the probe based on assumptions about the particle motion. Some of the limitations of ABR and OML theories appear in the case where $r_D/\lambda_D < 2.8$ at which point electron current is measured to be above expected values as shown in the paper by Sonin et al. [25]. In terms of dusty plasmas, the inaccuracy of ABR and OML theories appear as dust particle size increases and therefore result in an inaccuracy in the calculation of dust charge value using these theories. Further calculations of mixed mode theories were done by Bryant et al. [26] which show that values for dust charge lie somewhere between these two theories as shown by the plots in this paper.

One of the most important aspects which affects the nature of dusty plasmas is the charge on individual dust particles. The dust particles interact with the plasma components (ions, electrons, neutral particles) through collisions which results in some of this charge being deposited on the dust particles. This charge deposition over time can be represented as a current from each of the plasma components (k) incident upon the dust particle.

$$\frac{dq_k}{dt} = \sum I_k \tag{1.2}$$

These currents incident upon the dust particle will eventually result in the particles becoming charged. The determination of this dust charge is very important as this property will have an effect on the surrounding plasma. An example of a dusty plasma is shown in figure 1.3.

An assumption is made first that, in the case of a lab, $T_e > T_i$ and that mobility will be limited. The currents on this particle are assumed to be dominated by the electron and ion currents. These currents are assumed to come to an equilibrium on the dust particle over time to reach a constant value which balance.

$$\frac{dQ}{dt} = \sum I_k = I_i + I_e = 0 \tag{1.3}$$

This is known as the floating condition where I_e is the electron current and I_i is the ion current. This condition can be used to calculate the dust particle charge value once the incident currents have reached an equilibrium. In general, when the dust charge value is calculated, the dust particles are found to have a negative dust charge value. This is the case as the condition which charges the dust particle is the interaction with the surrounding plasma. The movement of plasma components around the dust changes over time as the particle becomes charged. The external power input mainly acts upon the electrons in the system as they are smaller and will move faster than the ions and neutrals in the system. This energy is then transferred to the neutral particles in the system through the highly energized electrons and subsequently ionization occurs. Electrons are initially more energetic than ions and neutrals and will have more collisions with the dust particles. This results in a more negative charge on the dust particles [17, 3]. Eventually the dust particles will become more negative and repel the incident electrons. When this occurs a balance of flux is established.

OML theory is commonly used as an assumption for many laboratory dusty plasmas as a typical dusty plasma falls within the valid parameter space of this theory and has been shown experimentally to fit theory [24]. The limits of OML theory appear as the ratio of r_d/λ_D increases. It is shown in the paper by Zobnin et al. [27] that as this ratio increases, the absolute value of the dust surface potential increases. Further deviation of dust charge value from OML theory comes with the inclusion of magnetic fields.

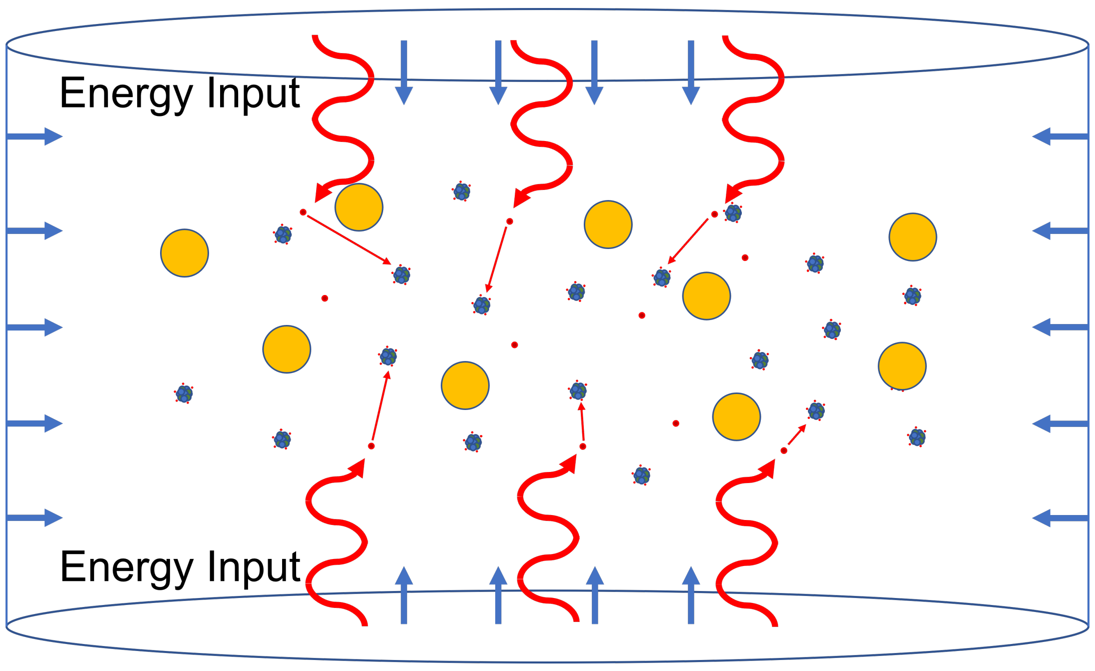
The charge value on a dust particle has been theorized to change as magnetic field becomes larger. It is shown in the papers by Tsytovich et al. [19, 28] that this deviation from expected value begins to occur as magnetic field becomes larger than a critical value. This critical magnetic field B_{cr} arises as the dust radius compares to quantities called the ion and electron gyro-radii which are given by

$$r_{B,k} = \frac{m_k v_{T,k}}{q_k B} \tag{1.4}$$

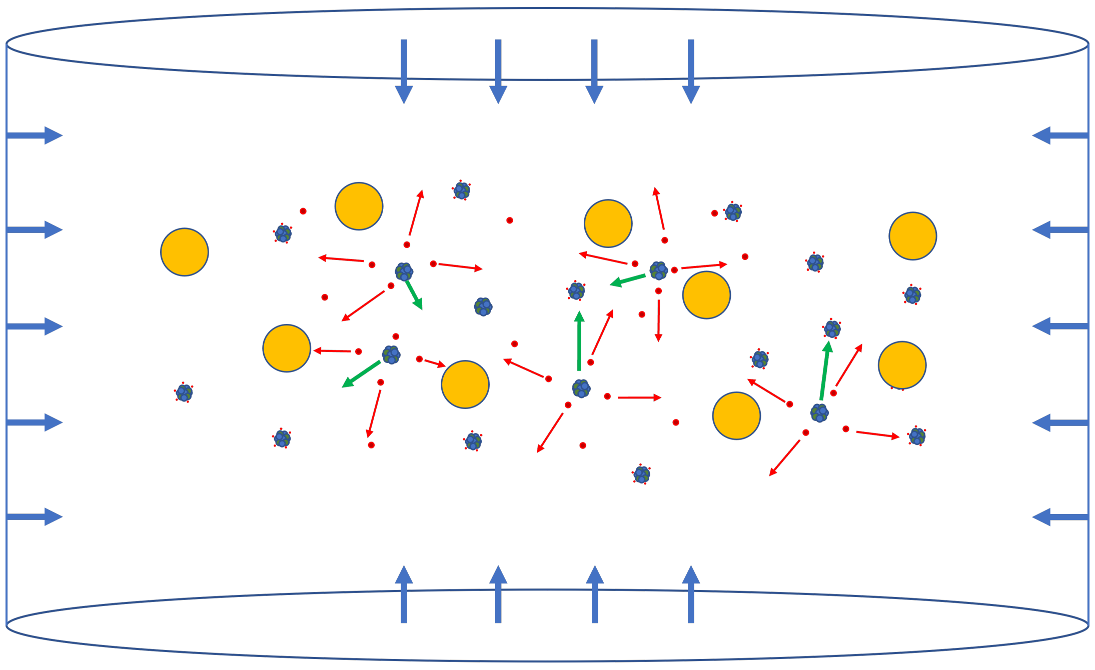
where the subscript k can refer either to electron (e) or ion (i). Here thermal velcity v_T is given by

$$v_T \propto \sqrt{\frac{k_B T_k}{m_k}} \tag{1.5}$$

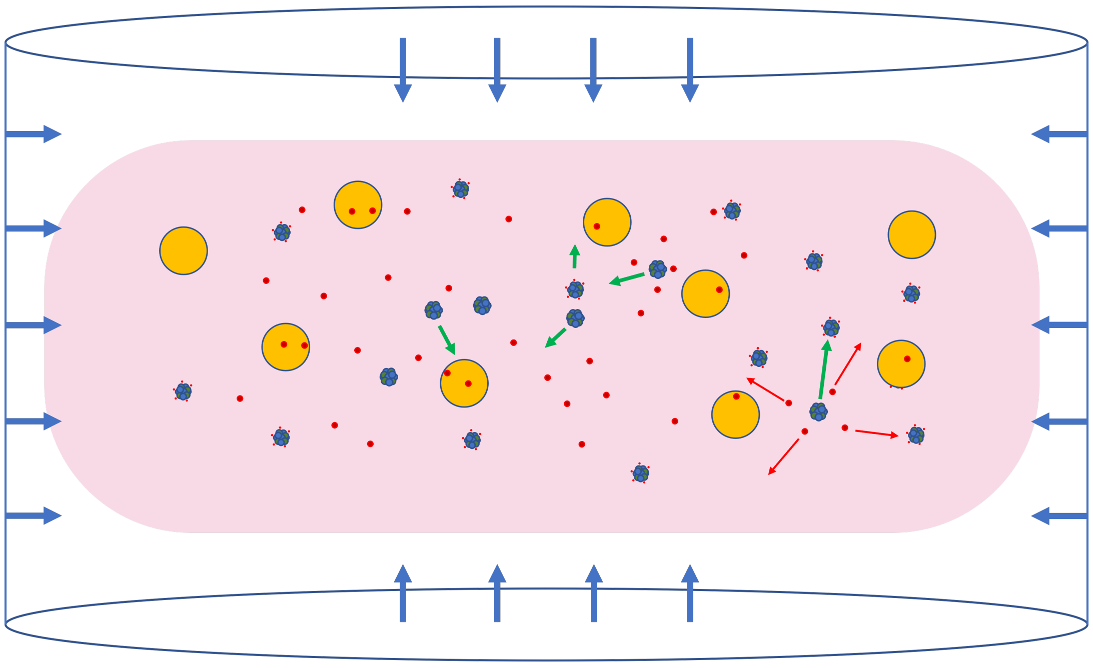
According to equation 1.4, these radii decrease as the magnetic field increases. For particles of dust radius $r_d \sim 10\ \mu m$ this change begins to occur at relatively small magnetic fields (for example, the critical magnetic field $B_{cr}^{e} = 4kG$ for a dust particle size of a $\sim 10\ \mu m$). In the case when $B_0 > B_{cr}$, electron gyroradius becomes smaller than dust radius and results in a decrease of dust charge value. As the magnetic field becomes even higher and ion gyro-radius becomes smaller than the dust radius, the dust charge value starts to increase greatly compared to values calculated with OML theory without a magnetic field. A computational analysis of the critical magnetic field at which dust charge value begins to decrease has been shown in the paper by Kodanova et al.[29].



(a)



(b)



(c)

Figure 1.4: Figure (a) A cartoon showing input of thermal energy into the dusty plasma system of neutral dust particles and neutral atoms. Energized free electrons move to collide with neutral particles. Figure (b) Cartoon showing ionization of particles. Ions move to collide with dust particles and neutral particles in system. Figure (c) Plasma ignites and all particles move in system with higher electron thermal energy than ion and neutral thermal energy due to interaction with wall

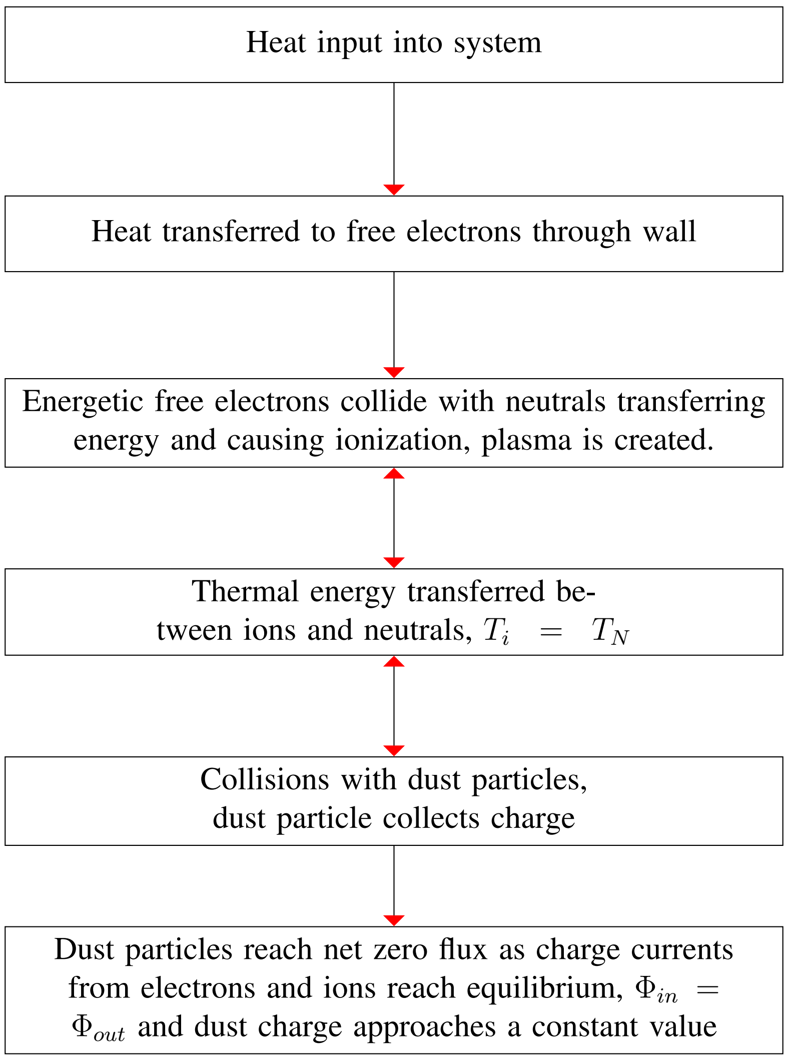

Heat input into system
Heat transferred to free electrons through wall
Energetic free electrons collide with neutrals transferring energy and causing ionization, plasma is created.
Thermal energy transferred between ions and neutrals, $T_i = T_N$
Collisions with dust particles, dust particle collects charge
Dust particles reach net zero flux as charge currents from electrons and ions reach equilibrium, $\Phi_{in} = \Phi_{out}$ and dust charge approaches a constant value

Figure 1.5: Flow diagram showing generation of dusty plasma

1.3.2 Dust Charge Analysis

While analysing the charging process is a useful method for calculating dust charge value, there are many other methods of finding this quantity. For the case of an isolated dust particle, a common method for calculating dust charge is achieved by establishing a connection between the floating potential of dust grains ϕ_s. dust capacitance C and their charge value Q_d [30, 31, 3].

$$Q_d = C\phi_s \tag{1.6}$$

The equation above requires an assumption for the capacitance of a dust particle. For this we consider the dust particle as a small spherical capacitor with the opposing charges being at a distance equal to the screening length λ_D [32]. This yields a dust capacitance of

$$C = 4\pi\epsilon_0 r_d e^{-\frac{r_d}{\lambda_D}} \approx 4\pi\epsilon_0 r_d(1 + \frac{r_d}{\lambda_D}) \qquad (\frac{r_d}{\lambda_D} \text{ is small}) \tag{1.7}$$

When $r_d \ll \lambda_D$ Equation 1.7 becomes [3]

$$C = 4\pi\epsilon_0 r_d \tag{1.8}$$

This is a common assumption for many laboratory dusty plasmas with micron sized particles and debye lengths typically between 50 - 500 μm [5].

For dust particles sized below 50 μm, charge value begins to fluctuate as a result of the discrete nature of the charging process via interactions with the surrounding plasma components. This is shown numerically in the paper by Cui and Goree [33] then analytically by Matsoukas and Russel [34]. These charge fluctuations reveal a power law [30, 27].

$$\frac{\Delta Q}{< Q >} = 0.5| \langle N \rangle |^{-\frac{1}{2}} \tag{1.9}$$

where Q refers to dust charge value and N refers to number of dust particles. This means that there will be further uncertainty in the dust charge as the average charge becomes smaller.

There have been many experimental methods for measuring dust charge. One of the simplest among these methods would be to look at the balance of forces for a dust particle. This has been done in the case where electric field balances with gravitational force on the dust particle. Equation 1.10 would allow the calculation of dust charge value provided a value of electric field could be found [35]. The drawback of this is that electric field is not a quantity that can be directly measured easily without disrupting the plasma. An example of this force balance is shown in figure 1.6

$$q_d E_z = mg \tag{1.10}$$

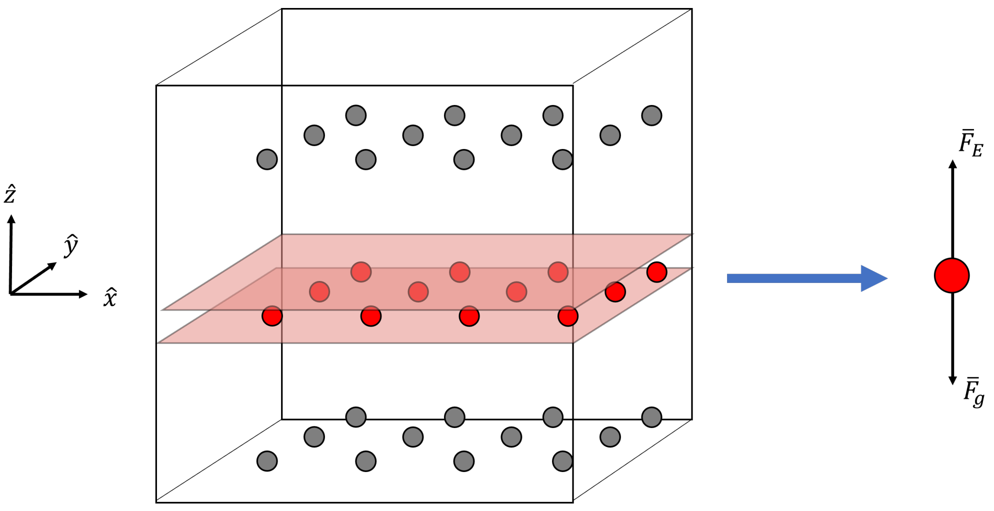


Figure 1.6: A cartoon showing a dust particle in a plasma in the case where the electric field force, $F_E = qE$ balances with the gravitational force, $F_g = mg$ to cause levitation of the particle.

Similarly, another experiment by Barkan et al. [36] uses the force balance between the electric field force F_E and force of gravity F_g for a dropped particle in order to calculate dust charge once the particle has reached a constant velocity

$$\sum F_y = F_E - F_g = 0 \tag{1.11}$$

Another method for experimental determination of the dust charge value is to use a Langmuir probe to measure the drop in plasma potential between a system without dust to that of a system with dust. This difference can then be compared to the quasi-neutrality condition [37, 36]

$$en_+ - en_e - N_d Q_d = 0 \tag{1.12}$$

which could then be used to calculate dust charge value Q_d provided the dust, electron and ion densities are all known quantities. One major drawback of using a probe to measure dust value is that it will disrupt the plasma and therefore has the possibility of changing the conditions of the dust-plasma system (and unfortunately the charge of the dust particles). Furthermore, knowledge of the ion, electron and dust densities are difficult to determine accurately [22].

The calculation of dust surface potential can be found through the analysis of dust trajectories during particle-particle collisions and the radial distances between particles. By assuming a screened Coulomb potential, dust plasma charge value can be calculated. The calculation for the effective dust charge value as well as screening length of these partricles has been shown in the papers by Konopka et al. to have an uncertainty of 10-20 percent [12, 38]. While this method is rather unperturbative, it does become difficult in a high dust density situation when the influence of neighbor particles cannot be ignored. The measurement of dust charge in this system is also dependent upon the dynamics of the dust particles. With the inclusion of a magnetic field the dynamics of the particles change and this will subsequently have an increase in the uncertainty of the measurement.

It has also been shown that it is possible to use vertical oscillations and waves in crystallized dust systems as a diagnostic. This is done by using a laser to input energy in the system in the form of an acoustic wave. By observing the wave as it moves through the system the height of the vertical oscillations of the dust can be measured. This can then be used to determine different properties in the dusty plasma such as coupling and dust charge value, Q_d [39].

A few experiments have been conducted for the case of a magnetized dusty plasma. An investigation into the dust charge value for this case was conducted by Lynch at Auburn University [40]. This was done in a magnetized dusty plasma which has been theorized by Tsytovich et al. (among others) [19] to alter the dust charge value in the system. An experiment was conducted by dropping individual, isolated dust particles into plasma with a perpendicular external magnetic field as shown in figure 1.7 While the dust particles are falling, they become charged as a result of the surrounding plasma. The Lorentz force resulting from the magnetic field acting upon the dust particle causes the particle to deflect. The magnitude of this deflection can be utilized to then measure the dust charge value on each particle. The dust charge value was measured to be lower than expected values calculated using OML charging theory [40]. A drawback of this method is that it is difficult to use in a more dense, collisional dusty plasmas.

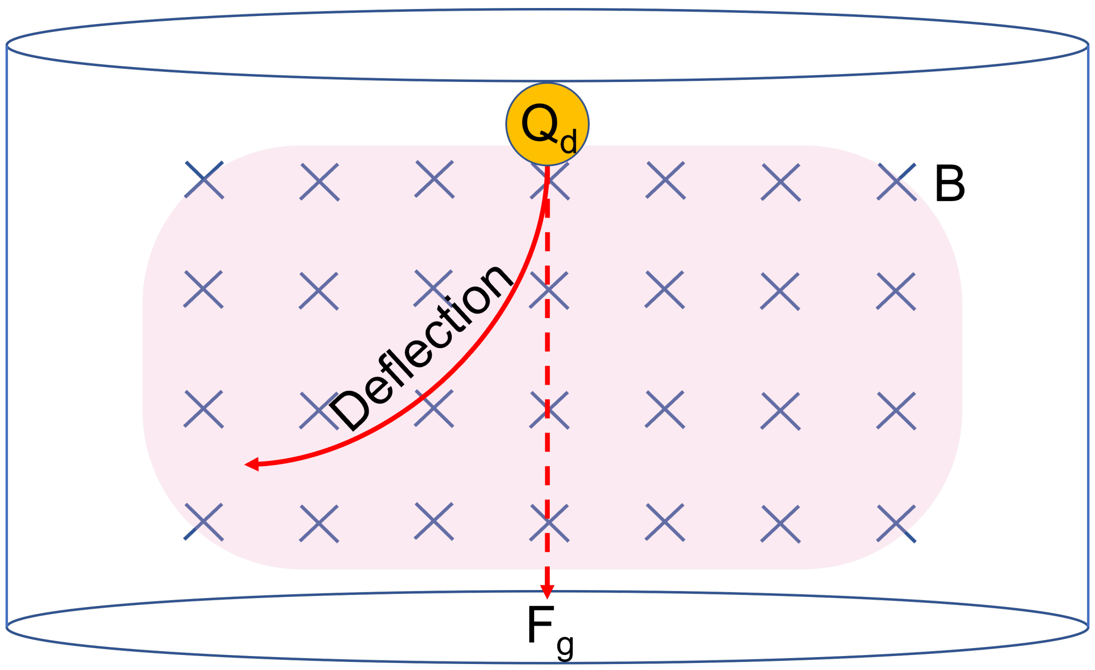


Figure 1.7: Experiment conducted by Lynch at Auburn University [40] which utilizes the deflection due to the Lorentz force of a single isolated dust particle in a perpendicular magnetic field to calculate dust charge value

1.3.3 Debye Shielding

As a result of the plasma surrounding dust particles, the effect of the interactive force is diminished. For charged particles, the interaction between particles can be modeled with a Coulomb potential.

$$\phi_{coulomb} = \frac{q_k}{4\pi\epsilon_0}\frac{1}{r_k} \tag{1.13}$$

A result of the screening of this potential by the surrounding plasma, the use of a screened Coulomb potential can be used for evaluating the interaction between these particles [41]. This theory has been tested experimentally by Konopka et al. [38] to have excellent agreement with the theory of screening in a dusty plasma. This screening effect is referred to as a Debye shielded potential

$$\phi_{shielded} = \frac{q_k}{4\pi\epsilon_0}\frac{e^{\frac{-r_k}{\lambda_D}}}{r_k} \tag{1.14}$$

1.3.4 Coupling Between Charged Particles

Dusty plasmas have also been used to study states of matter on a macro scale. A parameter known as the Coulomb coupling parameter can be introduced which correlates to the different state of matter of the dusty plasma. This coupling parameter Γ which is quantified by the ratio of inter-particle potential energy compared to the thermal kinetic energy of the dust particles [42].

$$\Gamma_f = \frac{\text{Potential Energy}}{\text{Kinetic Energy}} = \frac{q_d^2}{4\pi\epsilon_0 k_B T_d \Delta}\frac{f}{2} \tag{1.15}$$

Where here Δ is a parameter representing the inter-particle spacing of the dust particles (in meters) and f is the degrees of freedom in the system. It can be noted here that for this case, Γ_f given in equation 1.15 takes into account the degrees of freedom of the system. When referencing literature on the subject, the degrees of freedom are often times omitted or the system is assumed to match the description given in the paper.

$$\Gamma = \frac{q_d^2}{4\pi\epsilon_0 k_B T_d \Delta} \tag{1.16}$$

In the case of a screened potential (which is useful in the case of a Debye shielded system) this parameter can be modified by [43]

$$\Gamma_s = \frac{\text{Potential Energy}}{\text{Kinetic Energy}} = \frac{q_d^2}{4\pi\epsilon_0 k_B T_d \Delta} e^{\frac{-\Delta}{\lambda_D}} \tag{1.17}$$

A dusty plasma system is said to be strongly coupled when the coupling coefficient $\Gamma \gg 1$, or in other words when Coulomb interaction potential energy is much greater than dust thermal kinetic energy. In the case of a strongly coupled dusty plasma, the particles will no longer be isolated from one another. Subsequently, the dust charge values in the highly coupled case have been found to be lower than expected values calculated using various charging theories [18, 44].

For cases when $\Gamma \ll 1$, the system will be dominated by the thermal energy of the dust particles and therefore will behave similarly to a system in a gas-like state. Most dusty plasmas will be in the weakly coupled regime where $\Gamma < 1$. For dusty plasmas where Γ is on the order of 1, the system can be approximated as being in a ”liquid-like” state. For cases where the coupling coefficient $\Gamma \gg 1$ the system begins to form into a solid-like, Coulomb crystal [45, 18, 30].

Crystals such as this have been postulated as far back as 1938 and have been proposed as macro-scale models for more complex crystalline systems [46]. For dust particles in plasmas, Coulomb crystals were more specifically theorized in a 1986 paper by Ikezi et al. [47]. The crystallization of the system and the coulomb crystals result in a carbon-like hexagonal structure as shown in figure 1.8. Dusty plasma crystallization have also been studied experimentally and have been found to have dust charge values which are reduced compared to those calculated in the case of isolated dust particles using charging theory [18].

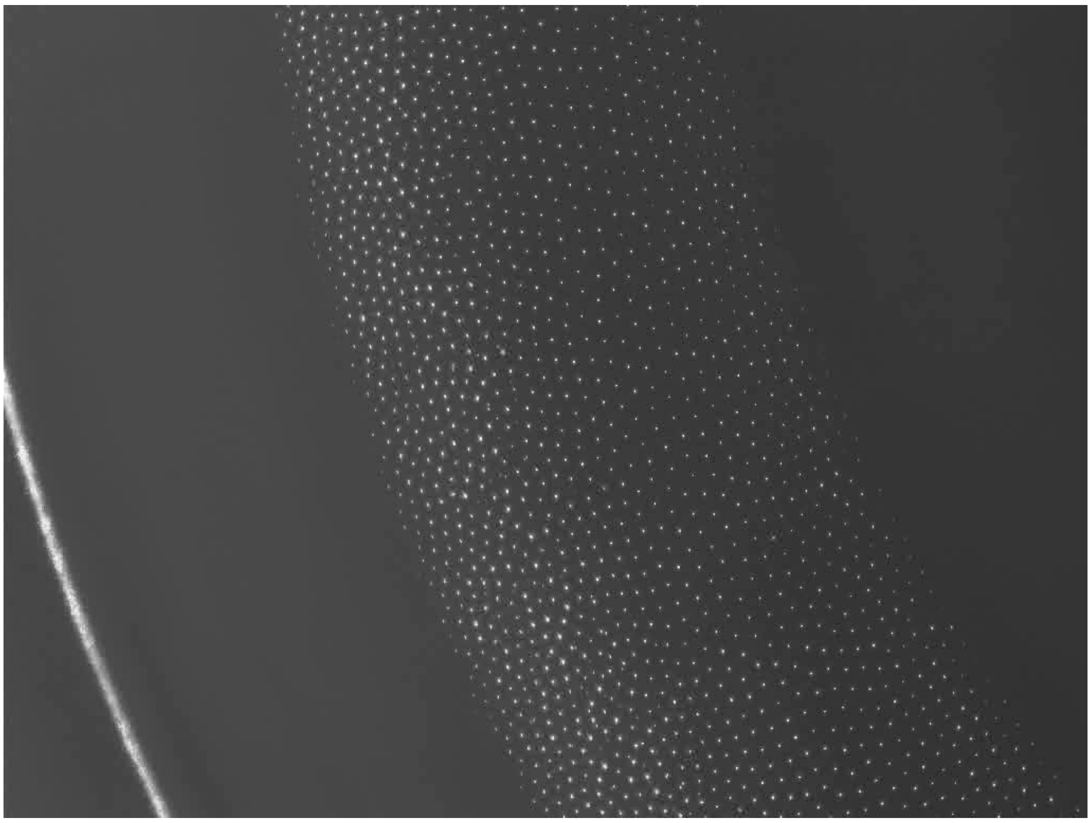


Figure 1.8: An example of crystallization in a dusty plasma system as shown in an experimental setup created at Max Planck Institute (Image provided by Uwe Konopka)

Much of the behaviour of dusty plasmas is related to how the dust particles interact with each other as well as the surrounding plasma. This interaction is largely dominated by the dust charge value therefore it is important to have an accurate method of determining this dust charge value. The determination of this particle charge is difficult to ascertain accurately as there are currently limited methods for calculating this value to a high degree of precision. Further complications arise with the addition of a magnetic field which many current methods have shown to find difficulty compensating for [19]. Due to these constraints, a new measurement technique which can measure this value precisely without modifying the dust or surrounding plasma is required. This is especially relevant in the case of high dust density, high coupling or in the presence of a magnetic field where the uncertainty in dust charge value is even higher.

In order to address the problem of charge value determination, a theory to accurately determine dust charge value has been created which takes both magnetic field and coupling into account. The theory will be expanded upon in the next chapter in the cases of both high and low coupling coefficient. These theories are then tested using a molecular dynamic simulation which has been developed for this purpose. A potential experimental investigation to verify the results collected from the simulation data has also been developed.

In chapter 2, the theoretical approach will be presented for the cases of both high and low dust particle coupling. In chapter 3, a simulation will be presented which will be used to analyze the methods introduced in chapter 2. In chapter 4, the results of the simulation are discussed as well as the analytical codes which have been developed to test the theory. In chapter 5, an experiment is proposed which can be constructed to test this theory. Finally, chapter 6 discusses the summary of this dissertation as well as the future work which has been theorized to continue this research.

Chapter 2

Theory

2.1 Theoretical Approach

In this chapter, the general theoretical approach will be introduced. This will then be expanded upon for both the case of a fluid system (low coupling) as well as a crystalline system (high coupling). For both cases, the theory requires a system in which an external force compresses the system along a set direction. The external force is directed opposite the direction of an electric field confinement. Experimentally this confinement will be the result of the walls of the system or the walls of an imposed potential well in the system. The result of these forces is a gradient in the density of the dust particles. This is very similar to a classical Hall effect system.

A classical Hall effect system initially has uniformly distributed electrons along a two dimensional plane. These electrons are given a velocity along a specific flow direction which is parallel to the plane. This plane is perpendicular to an external magnetic field. Having charge, the incident magnetic field gives the electrons a $\overrightarrow{v} \times \overrightarrow{B}$ drift perpendicular both to the direction of movement as well as the magnetic field. An electron density gradient builds up establishing a Hall electric field that opposes the Lorentz force. As a result, a Hall voltage can be measured across the conductor perpendicular to both the electron motion as well as the magnetic field. A similar system can then be envisioned in which negatively charged dust acts as the particles in the system in a similar way to the electrons in the case of the classical Hall effect system, the main difference in this case being the shielding effect of the surrounding plasma. Shielding of the dust particles has a result that the compensation force as a result of a Hall electric field is no longer a global effect and now is instead a result of the pressure force from the dust-dust collisions. This is shown in Figure 2.1 as it applies to our similar ”Dusty Plasma” Hall effect

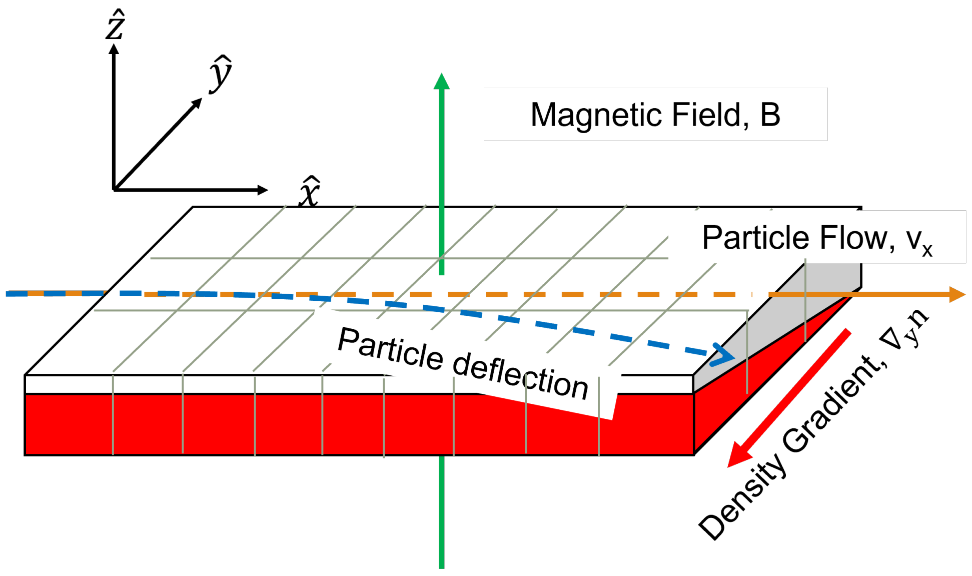


Figure 2.1: Dust particles should behave similarly to the classical Hall effect, this is a result of the charged nature of the dust particles

system. As with the electrons, these charged dust particles will also have a resultant $\vec{v} \times \vec{B}$ drift due to the magnetic field.

2.2 Low Coupling Theory

When looking at low coupling theory it is helpful to view the system as a fluid. Along with this a few assumptions will be made:

1. The system is isothermal (i.e., $\vec{\nabla} T = 0$)
2. The system has reached a steady state and is independent of time ($\frac{dQ}{dt} = 0, v =$ constant, $\vec{\nabla} n =$ constant)
3. Magnetic field, flow direction and density gradient are all perpendicular
4. External electric field is approximately zero in region of evaluation
5. Density is non-uniform along y-direction

As we are viewing the system as a fluid, it is useful at this point to start solving the equation for momentum.

$$mn\left[\cancelto{0}{\frac{\partial \vec{u}}{\partial t}} + \cancelto{0}{(\vec{u}\cdot\vec{\nabla})\vec{u}}\right] = \sum_i f_i \tag{2.1}$$

where m is dust mass, n is dust density, u is system velocity. It should be noted that for the sake of this derivation, f_i refers to a force density and F_i will refer to forces. Here, $\frac{\partial \vec{u}}{\partial t} = 0$ because we assume that our system has reached steady state. From Lorentz force, this speed perpendicular to the magnetic field results in a deflection of the dust particles perpendicular to the flow direction. By the time the system has reached steady state, a density gradient will have formed as shown in Figure 2.1.

The forces which act upon a dust particle in our system are shown in Figure 2.2. The drag and driving forces shown in this diagram oppose and result in a constant flow velocity once the system has reached a steady state. Furthermore, the force balance between the vertical confinement electric field and the gravitational force results in a monolayer of particles which allows the system to be viewed of as two dimensional. An example of this monolayer is shown in figure 2.3. Lastly, the Lorentz force in the system will become balanced with the force of the pressure gradient will also balance and this is what allows the calculation of the particle charge proportional to the density gradient.

Listing the force densities in the system is necessary now to solve equation 2.1, it is useful however to remember that the density is non-uniform along our y direction $n = n(y)$

$$\vec{f}_{Lor} = qn\left(\vec{E} + \vec{u}\times\vec{B}\right) \tag{2.2}$$

Lorentz force for this system will have an electric field component along our $\hat{z}$ direction as well as a magnetic field component along the $\hat{y}$ direction. The next force to look into is the force resulting from the dust pressure, as this force is a result of the gradient of temperature and density, it can be shown this will only appear along the y-direction since the pressure will be strongest when the dust particle density is highest (i.e. anti-parallel to the magnetic force

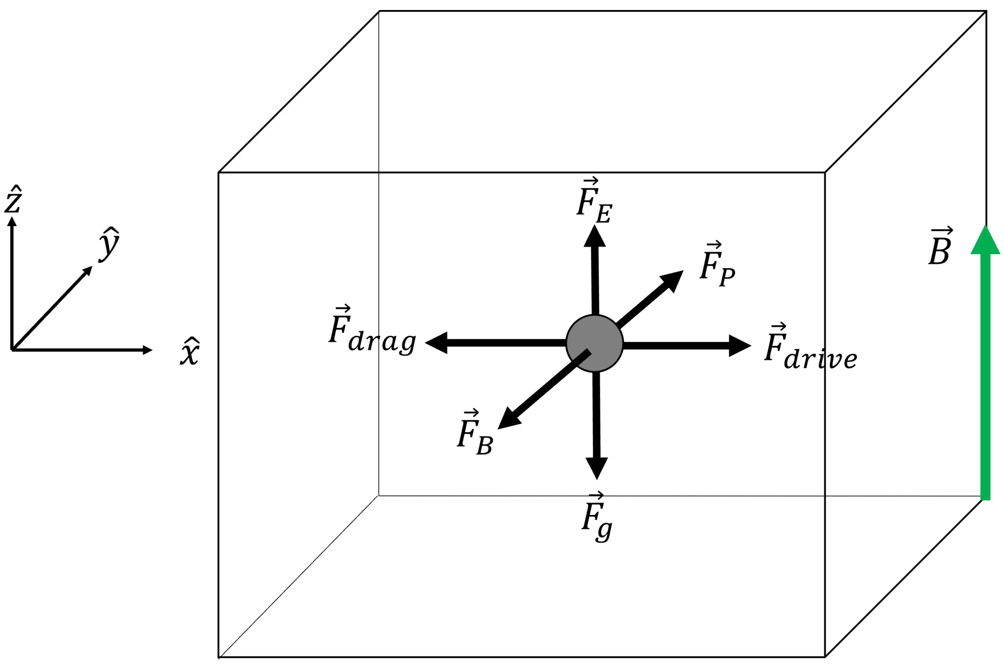


Figure 2.2: Forces on a dust particle in a dusty plasma. Among these are driving force, drag, electric field (F_E), gravity (F_g), pressure force (F_P), and lorentz force (F_B)

and perpendicular to the magnetic field.

$$\overrightarrow{f}_{Pressure} = -\overrightarrow{\nabla} P = -\overrightarrow{\nabla}(n_d k_B T_d)$$

$$= -k_B T_d \overrightarrow{\nabla} n_d - k_B \cancelto{0}{n \overrightarrow{\nabla} T_d} \tag{2.3}$$

$$\overrightarrow{f}_{drive} = n\overrightarrow{F}_{drive} = (n)(const)\,\hat{x} \tag{2.4}$$

One of our assumptions for this system is a constant driving force. Experimentally this can either be generated in a number of different ways some of which will be discussed further in this dissertation. An important piece of note is that this driving force can not be proportional to charge as this will result in charge value cancelling.

$$\overrightarrow{f}_{drag,n} = -n\gamma\overrightarrow{u} \tag{2.5}$$

The framework for neutral drag on a charged particle in a plasma was originally theorized in the 1924 paper by Paul Epstein [48]. Neutral drag on the dust particles where drag coefficient

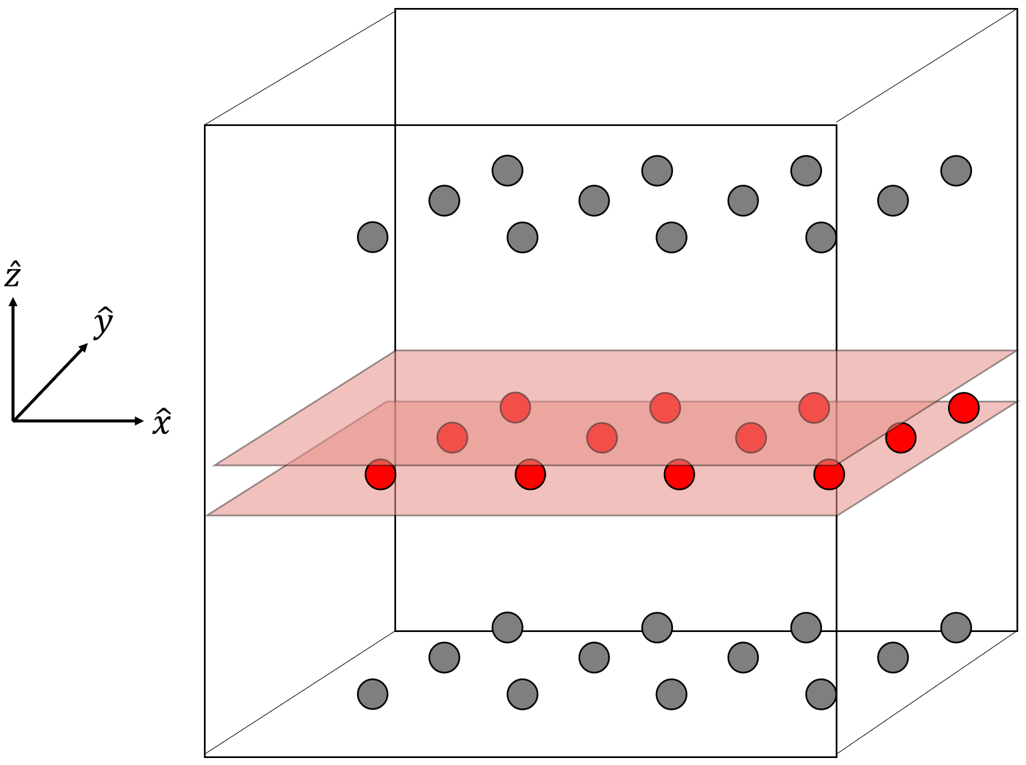


Figure 2.3: A diagram showing the monolayer of dust particles. The area between the red boxes is what we will be looking at (i.e. along the direction of gravitational force shown in figure 2.2). This is useful as it allows our system to be viewed as two dimensional

γ is given by

$$\gamma = \delta \frac{4\pi}{3} n_n m_n c_s r_d^2 \tag{2.6}$$

Here, c_s is the sound speed which is a result of the neutral temperature and neutral mass shown in equation 2.7.

$$c_s = \sqrt{\frac{8k_B T_n}{\pi m_n}} \tag{2.7}$$

Equation 2.8 shows δ which is the reflection coefficient. The reflection coefficient δ relies on how a charged dust particle moves through the plasma and the resulting reflection of the plasma components. This value for the reflection coefficient is chosen as it represents diffuse reflection for the case of a perfect spherical thermal non-conductor as is the case for a commonly used melamine-formaldehyde dust particle [48, 49].

$$\delta = \left(1 + \frac{9\pi}{64}\right) \tag{2.8}$$

Using these equations we can then solve equation 2.1 further by assuming that gravity and electric field along the z direction will cancel

$$\begin{aligned} 0 &= \overrightarrow{f}_{Lor} + \overrightarrow{f}_{p} + \overrightarrow{f}_{eps} + \overrightarrow{f}_{drive} \\ &= qn\left(\overrightarrow{E} + \overrightarrow{u} \times \overrightarrow{B}\right) - \overrightarrow{\nabla} p - n\gamma \overrightarrow{u} + \overrightarrow{f}_{drive} \end{aligned} \tag{2.9}$$

Reiterating the assumptions of perpendicular magnetic field, zero electric field (within the region of simulation) and constant flow direction yields equation 2.10:

$$\begin{aligned} &\overrightarrow{B} = B_o\hat{z}, \quad E_x = 0 \\ &E_y \approx 0, \qquad \overrightarrow{u} = u_x\hat{x} \end{aligned} \tag{2.10}$$

And applying these assumptions gives an equation for the system flow velocity.

$$\begin{aligned} \underline{x:} \quad 0 &= -k_B T_d \cancelto{0}{\left(\frac{\partial}{\partial x} n\right)} + f_{drive} - n\gamma u_x \\ \rightarrow u_x &= \frac{f_{drive}}{n\gamma} = \frac{F_{drive}}{\gamma} \end{aligned} \tag{2.11}$$

This shows that the velocity of the system is a function of the driving force F_{drive} as well as the drag coefficient. As both of these values are constant, the velocity of the particles along the x direction is constant. Solving for forces along the y direction,

$$\begin{aligned} \underline{y:} \quad 0 &= qn\left(\overrightarrow{u} \times \overrightarrow{B}\right)_y - k_B T_d \nabla_y n \\ \rightarrow 0 &= qnu_x B_o - k_B T_d \nabla_y n \end{aligned} \tag{2.12}$$

By using equation 2.2 dust charge value can be evaluated for the case of low coupling

$$q = \frac{k_B T_d \gamma \nabla_y n}{B_o f_{drive}} \tag{2.13}$$

Or, by inserting the result from equation 2.2 into equation 2.2 charge can be calculated with this average system velocity, u_x

$$q = \frac{k_B T_d}{B_o u_x} \frac{\nabla_y n}{n} \tag{2.14}$$

Equation 2.14 gives an value for dust charge dependent upon the dust temperature T_d, dust density gradient $\nabla_y n$ and flow velocity u_x. These are all parameters which can be evaluated by using positional data over multiple time frames in either an experiment using particle tracking software or in a simulation.

This charge calculation is useful when dust thermal energy is much higher than the inter-particle energy between dust particles. Another method of evaluation therefore is required when the opposite is true. When a set of particles is crystallized or in other words has a high coefficient of coupling, a separate dust charge calculation theory is necessary. For this case, a high coupling charge calculation theory has been developed.

2.3 High Coupling Theory

For the case of high coupling in a dusty plasma, the inter-particle potential energy will be larger than the thermal potential energy of the dust particles. The result of this is that the system will begin to behave in a solid-like manner where crystalline structure can begin to form at high values of the coupling coefficient, Γ_s. In order to evaluate this coupling coefficient an assumption must be made regarding the interaction between dust particles. For an independent set of charged particles in a vacuum, a coulomb potential shown in equation 1.13 is assumed as the interaction potential between particles. Since each particle is screened by the plasma in the system, it is necessary to instead use a screened coulomb potential as an assumption. This is apparent from the solution of Poisson's equation in spherical coordinates for the potential given a space charge around a sphere given by [32]

$$\rho = n_o e[1 - \exp(e\phi/kT)] \tag{2.15}$$

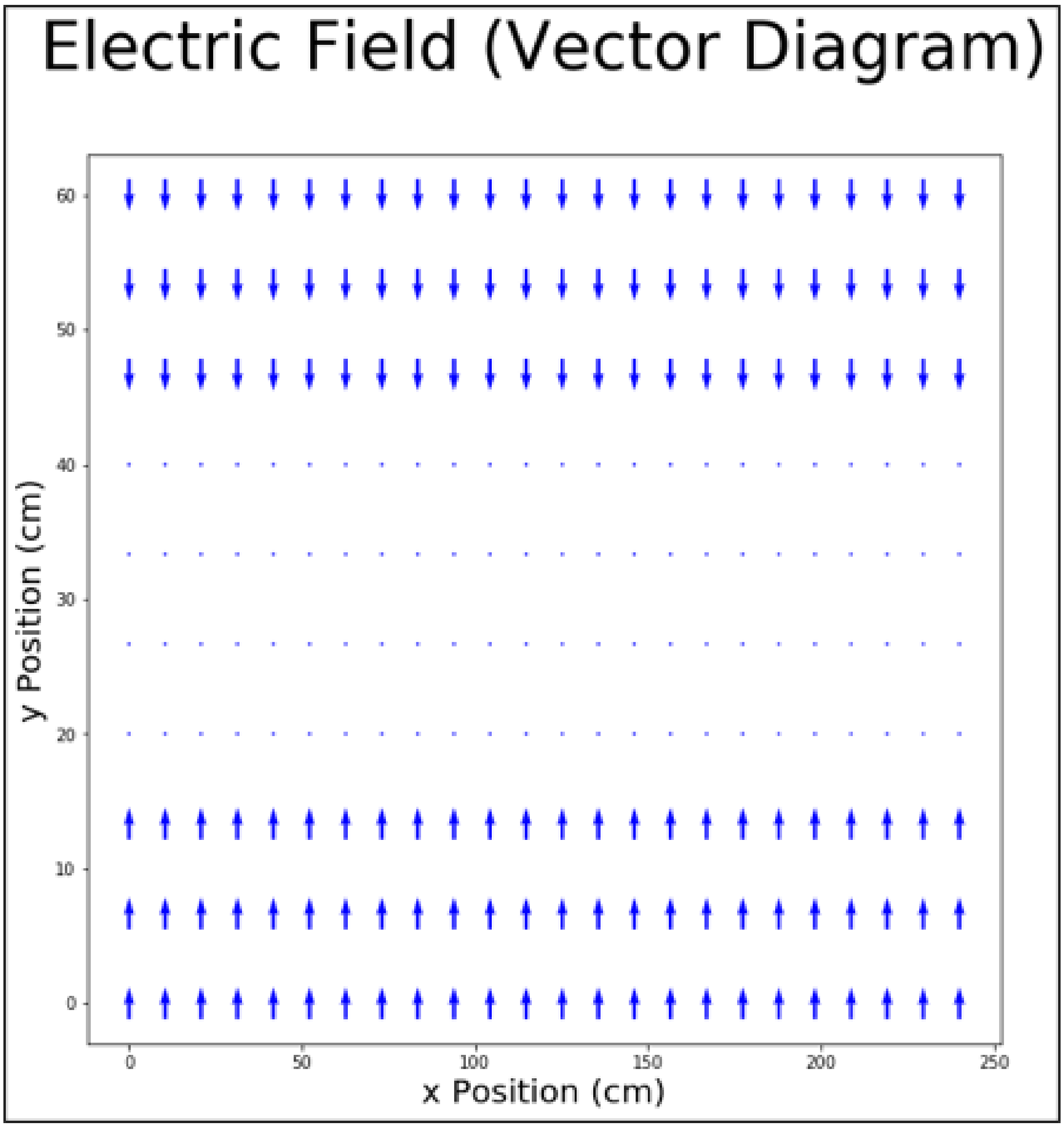


Figure 2.4: Electric field diagram for the case of a positive dust particle. In this system, the particle will be confined within the center region along the y-direction

The solution of poisson's equation, $\nabla^2\phi = -\frac{\rho}{\epsilon_0}$, for charge density from equation 2.15 subsequently yields an equation for inter-particle potential ϕ

$$\phi = \frac{q_d}{4\pi\epsilon_0}\frac{e^{\frac{-r}{\lambda_D}}}{r_d} \tag{2.16}$$

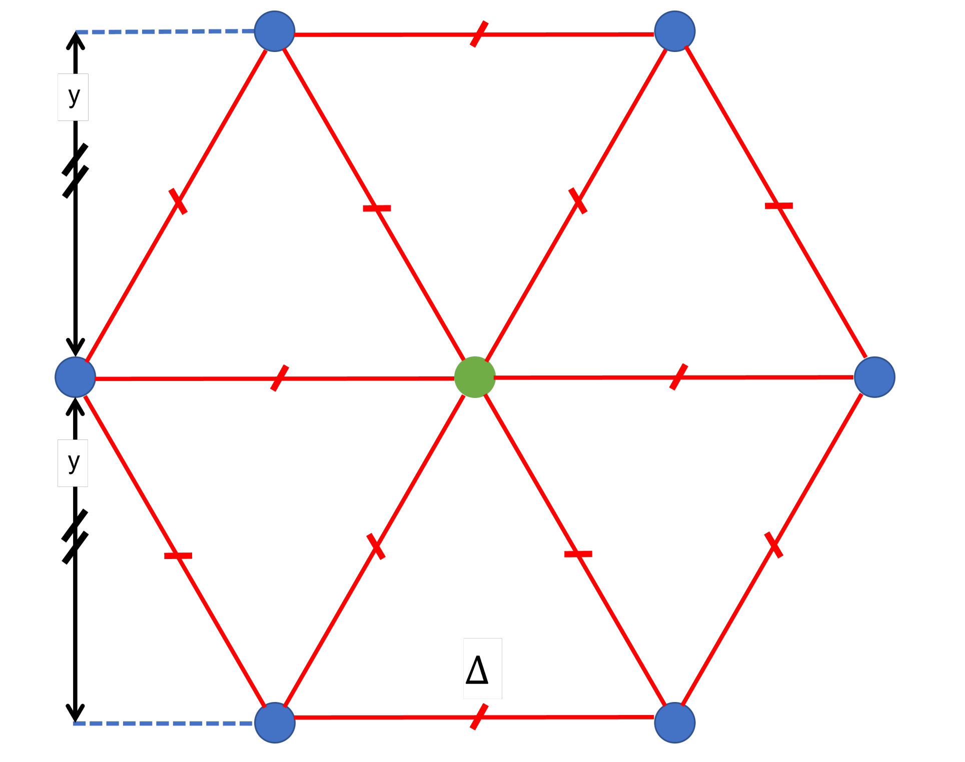
Taking the gradient of this force gives an equation for the interaction force between dust particles. This interaction force is similar to the coulomb force but includes the effect of screening as a result of the surrounding plasma from equation 2.16.

$$\overrightarrow{F}_{interaction} = -q\overrightarrow{\nabla}\phi = \frac{q_d^2}{4\pi\epsilon_0}\frac{e^{\frac{-r}{\lambda_D}}}{\lambda_D r_d^2}(\lambda_D + r_d)\hat{r} \tag{2.17}$$

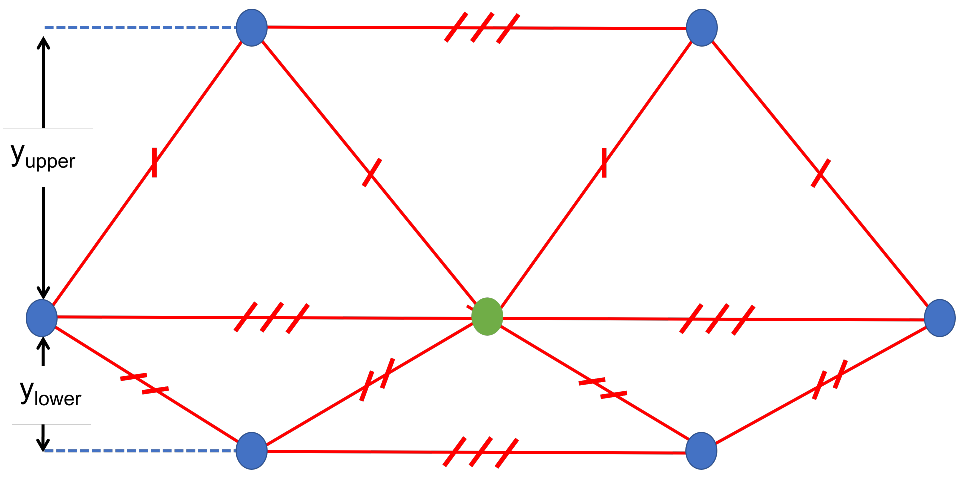
As a result of the interaction forces the particles will all repel one another. This repulsion combined with the confinement within the system by an external electric field shown in figure 2.4 causes the particles to organize into a hexagonal shape. This results has been observed in experiments by H. Thomas et al. [18] as well as Chu et al. [45] which goes on to show the hexagonal form these crystals will take. This hexagonal structure is shown in figure 2.5a. This has been observed many times and is a true crystalline structure. When the compressional force (in this case Lorentz force) is included a more complex system arises.

Under the effects of a compressional Lorentz force a ”crystal” similar to that shown in figure 2.5b forms. A note should be made here about the validity of the use of the word ”crystal”. Since the definition of a crystal is ”a piece of a homogeneous solid substance having a natural geometrically regular form with symmetrically arranged plane faces” from this is technically no longer a crystal as the symmetry and homogeneity has been lost, however for the sake of simplicity I will continue to refer to this as a ”crystal-like” or ”pseudo-crystalline” structure. Due to the discreteness of bond lengths of crystalline structures in nature, it is uncommon to have this sort of structure. A similar geometry can be found for the case of sedimentation layers in charged colloidal particles as shown in [50]. This work shows a similar system with slightly different three dimensional geometry. Also in the field of colloid chemistry, the paper by Philipse et al. [51] shows unique diffusion and sedimentation profiles for the case of sedimentation in colloids where the Debye screening between adjacent particles is relevant, which is the case for our dusty plasma system.

Taking into account the forces incident upon the central particle in figure 2.5b, figure 2.6 is formed. This figure includes both the external forces (drag, driving force, compressional Lorentz force) as well as the interaction forces from each of the six nearest neighbors. Including



(a)



(b)

Figure 2.5: This figure shows an example of (a) a particle with no compressional force which creates a perfect hexagon with constant inter-particle distance shown here as Δ and subsequently a uniform y-directional distance. (b) Shows a figure which is under a compressional force balanced by the boundary conditions which holds the particles stationary. Note here that the distances between particles is constant however the inter-particle spacing will change for each layer. The x-direction distances are all constant however regardless of layer.

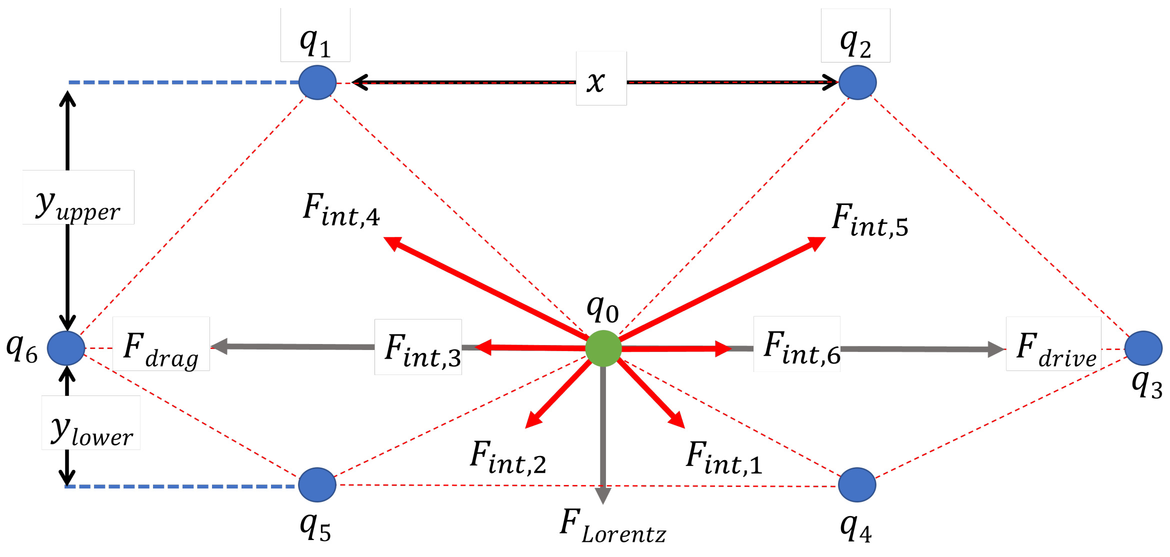


Figure 2.6: Asymmetric dust ”crystal” due to compressional force. This shows the interaction forces (shown in red) as well as the external forces (shown in gray). Note here that the upper and lower y distances are not equal but the distance between particles in the x direction are the same. Here, the green particles is the one being investigated and can be viewed as particle i with charge value q_i

this interaction force for a calculation of forces from Newton’s second law

$$\sum \overrightarrow{F} = (F_{drive} - F_{drag} + \sum F_{int,x})\hat{x} + (F_{lorentz} + \sum F_{int,y})\hat{y} = m\ddot{\overrightarrow{x}} \tag{2.18}$$

Figure 2.7shows a plot of the density in the x direction over time. Since this density is relatively constant, it can be said that the particles are equally spaced along the x direction. From this, $\ddot{\overrightarrow{x}} = 0$ and the system can said to be in a steady state. Newtons second law $\sum F = ma$ can then be solved along the x direction.

As shown in figure 2.6, the forces which we will be taking into account are those of the Epstein drag, driving force as well as the interaction forces in the system for the x direction (Lorentz force only points along y).

$$F_{int,5,x} - F_{int,4,x} + F_{int,1,x} - F_{int,2,x} + F_{int,6} - F_{int,3} + F_{drive} - F_{drag} = 0 \tag{2.19}$$

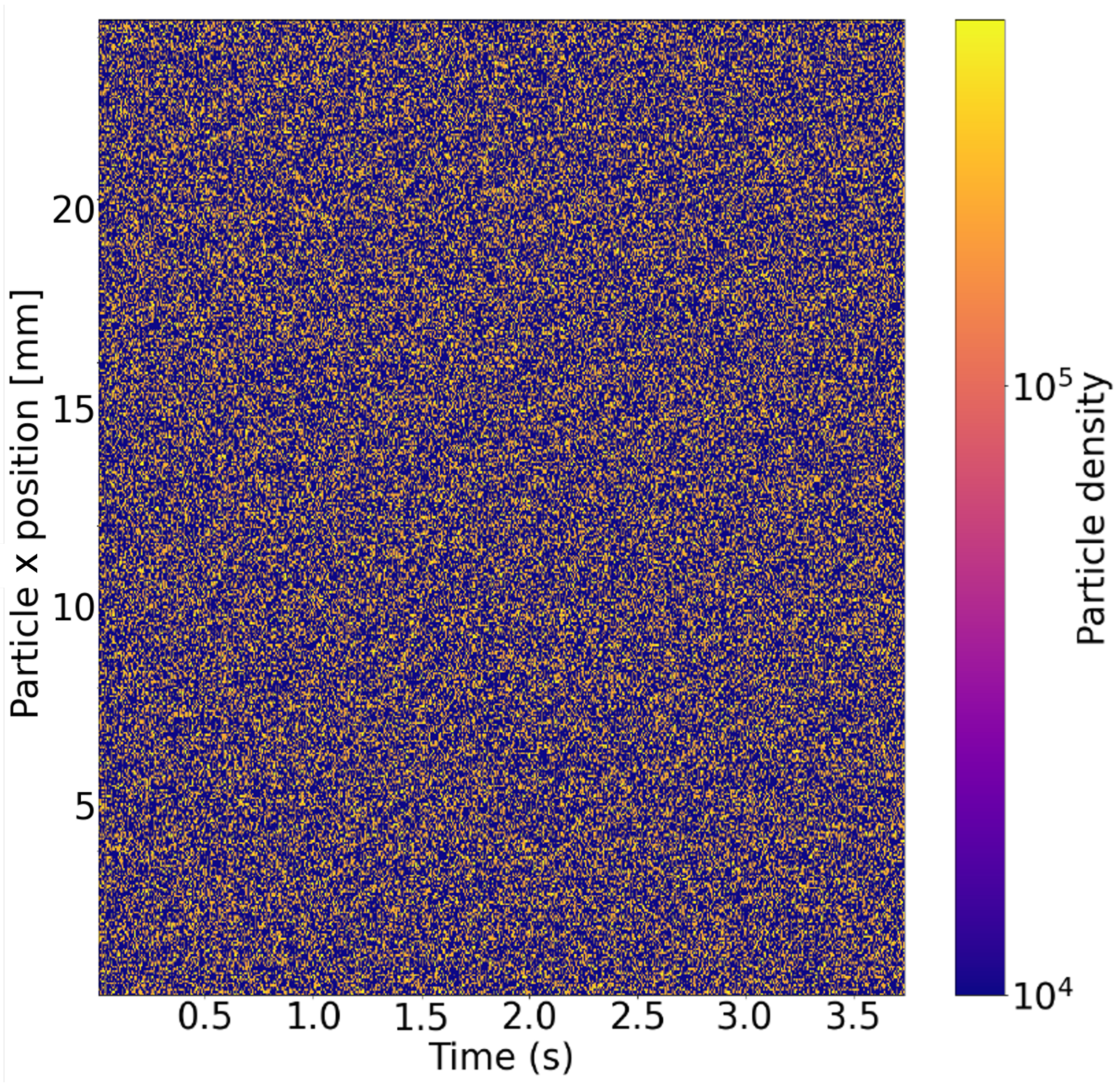


Figure 2.7: This figure shows a plot of the dust particle density over time in a simulation. Each frame in x is plotted on a histogram and this is shown as a single vertical slice. For the particle position along x there is no preferential direction or structure for the density of the particles. As time progresses it can be seen that the density is random. This is why the image appears to look like ”white noise”, this noise is because of the movement of the particles in the system in random Stochastic motion as a result of the temperature in the system.

As a result of the uniformity along the x direction, the interaction forces in the system will all cancel and this simply yields

$$F_{drive} = F_{drag} \tag{2.20}$$

This matches the assumption in the system that the drag and driving forces are independent of the interaction forces.

$$F_{drive} = F_{drag}, \qquad \sum F_{int,x} = 0 \tag{2.21}$$

In order to avoid confusion, it should be pointed out here the labeling of forces in this section. The forces follow those shown in figure 2.6. Each of the particles is given a number 1 through 6 and the interaction forces are corresponding to the force between the center particle and the number give. Each of these forces will be repulsive forces as all dust particles are assumed to have the same dust particle charge. With this in mind, Newton's second law can be solved along the y-direction to yield

$$F_{int,4,y} + F_{int,5,y} - F_{int,2,y} - F_{int,1,y} - F_{i,Lorentz} = 0 \tag{2.22}$$

Where $F_{i,Lorentz}$ and similarly $v_{i,x}$ refer respectively to the lorentz force on particle i (in figure 2.6 particle i would be the green central particle) and the velocity of particle i. $F_{int,y}$ refers to the assumed interaction force in the system along the y-direction (perpendicular to that of the driving force of the system as well as perpendicular to the magnetic field in the system). In the case of this dissertation, the assumed interaction force is that of a Debye shielded potential as shown in equation 2.17.

Due to the uniformity in x as well as the symmetry in the system,

$$\begin{aligned} F_{int,lower} &= F_{int,4,y} = F_{int,5,y} \\ F_{int,upper} &= F_{int,1,y} = F_{int,2,y} \end{aligned} \tag{2.23}$$

Continuing with the y-direction and rewriting equation 2.22 with terms defined in 2.23 yields

$$2F_{int,lower}\frac{y_{lower}}{r_{lower}} - 2F_{int,upper}\frac{y_{upper}}{r_{upper}} - q_d v_{i,x} B_z = 0 \tag{2.24}$$

Continuing and, for the sake of ease of view, rewriting r_{lower} as r_L and the same for forces and r_{upper}

$$\frac{2q_d^2}{4\pi\epsilon_0}\left[\frac{(r_U+\lambda_D)e^{-r_U/\lambda_D}}{\lambda_D r_U^2}\frac{y_U}{r_U}-\frac{(r_L+\lambda_D)e^{-r_L/\lambda_D}}{\lambda_D r_L^2}\frac{y_L}{r_L}\right]=q_d v_{i,x}B_z \tag{2.25}$$

This equation can be subsequently solved for the dust particle charge as a result of the geometry of the system:

$$q_d=\frac{4\pi\epsilon_0 v_{i,x}B_z}{\frac{2(r_U+\lambda_D)e^{-r_U/\lambda_D}}{\lambda_D r_U^2}\frac{y_U}{r_U}-\frac{2(r_L+\lambda_D)e^{-r_L/\lambda_D}}{\lambda_D r_L^2}\frac{y_L}{r_L}}=\frac{v_x B_z}{\sum G_k} \tag{2.26}$$

Where G_k refers to the geometric terms from the interaction forces further defined in equations 2.28 and 2.29. While equation 2.26 is true for the ideal case where the system is truly symmetric along the x-direction this is not always the case, by assuming $F_{int,left} \neq F_{int,right}$ the following equation results.

$$q_d=\frac{4\pi\epsilon_0 v_{i,x}B_z}{G_4+G_5-G_1-G_2} \tag{2.27}$$

where for the sake of simplicity a geometric term, G_k is introduced. Here, this geometric term

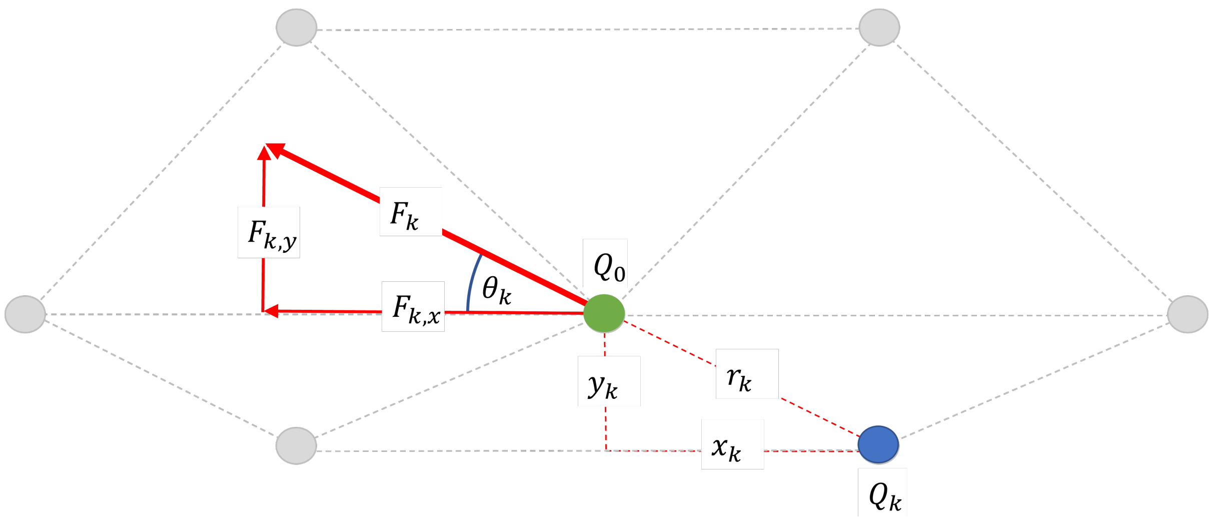


Figure 2.8: Visualization of a single interaction force and it's components and angle. This angle is used in equation 2.28

is simply the y-component of the interaction force without the charge. Looking at an individual interaction force is shown in figure 2.8.

$$G_k=\frac{F_k}{q_d^2}\sin\theta_k=\frac{F_k}{q_d^2}\frac{y_k}{r_k}=\frac{F_{k,y}}{q_d^2} \tag{2.28}$$

Or in other words, rewriting equation 2.28 with the values of force from equation 2.17 yields,

$$G_k = \frac{(r_k + \lambda_D)e^{-r_k/\lambda_D}}{\lambda_D r_k^2} \frac{y_k}{r_k} \tag{2.29}$$

Equation 2.26 can be further generalized for a separate external non-Lorentz force which is not dependent upon q_d by substituting $q_d v_x B_z = F_{external}$. This results in a more general equation:

$$q_d = \sqrt{\frac{F_{external}}{\sum G_k}} \tag{2.30}$$

Equation 2.30 is useful for cases such as compressive force due to gravity. As a result of the previous derivations, equations 2.26 and 2.30 can be used in the case of a crystal-like or highly-coupled dusty plasma system. Combining this with equation 2.13 for the case of a fluid or lowly-coupled system means that dust charge value can now theoretically be determined for a wide range of cases. This can further be validated through analysis of a simulation which will be shown in the next chapter.

Chapter 3

Molecular Dynamics Simulation

3.1 Approach

The theory from the previous chapter was tested using a computational approach. In order to achieve this, a molecular dynamic (MD) simulation was written and the body of this code is included in appendix A. This code was developed in C++ and is a molecular dynamics simulation the purpose of which is to analyze the movements of the individual particles in the system. This is useful for simulation of dusty plasmas due to the discrete nature of the macro-sized system particles.

The forces which are applied to the particles in this simulation consist of Epstein drag, Lorentz force, gravitational force, an external driving force, a Yukawa or screened Coulomb interaction force derived from equation 2.16 as well as a force resulting from the thermal dust temperature. A flow diagram of the simulation is shown in figure 3.1

Two separate integration methods were tested for this simulation. The first of which was Runge-Kutta 4th-order (RK4) integration. This is an extremely rigorous integration method which decreases error at the cost of longer simulation times compared to other integration methods. While this integration method was originally implemented, a more efficient integration method was eventually implemented which is referred to as the BAOAB method. While most integration methods have a tendency to increase error as drag coefficient increases, BAOAB has the lower error when compared to methods such as Stochastic Position Verlet (SPV) and Brunger-Brooks-Karplus (BBK) as shown in the papers by Leimkuhler et al. [52, 53].

In order to make the simulation more efficient, the integration method was changed from

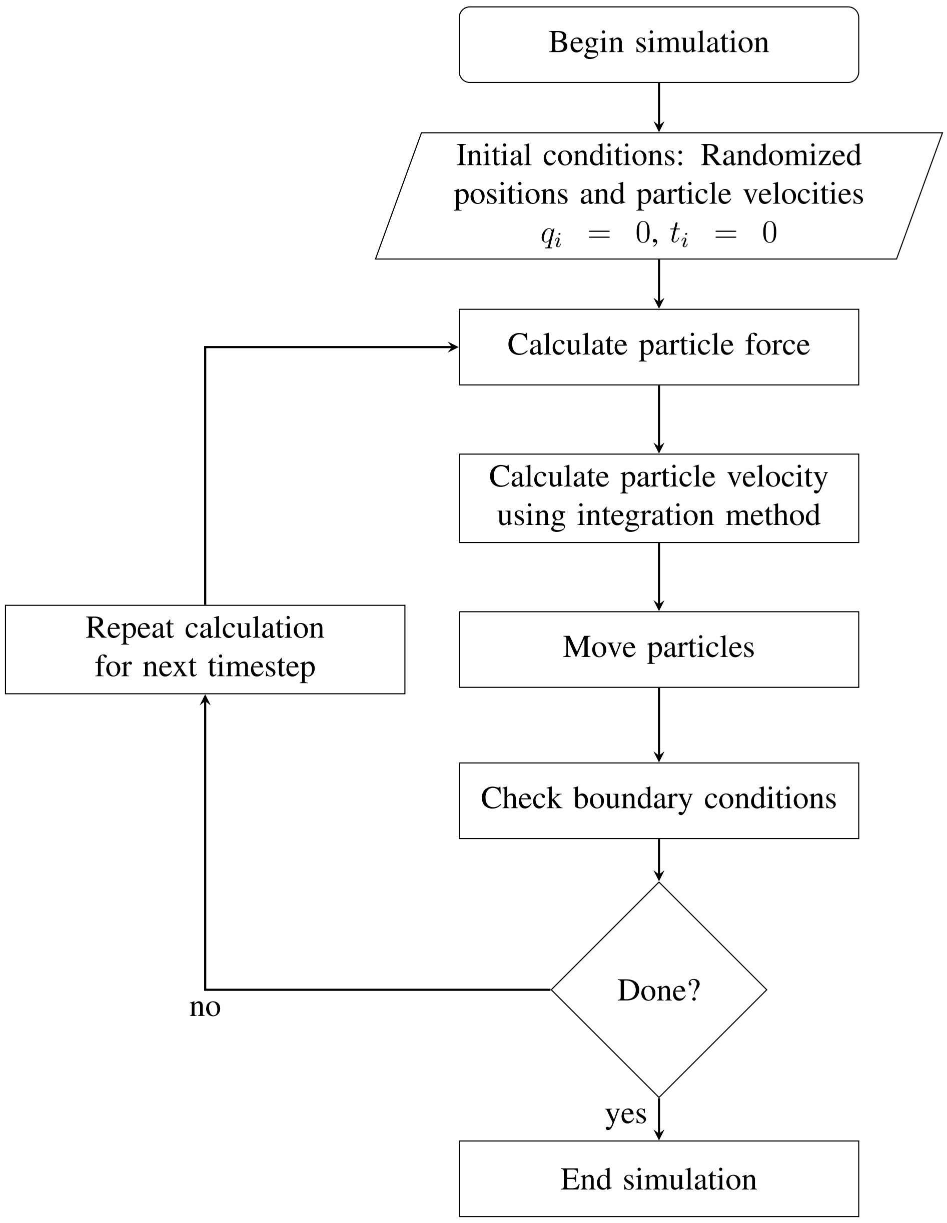

Begin simulation
Initial conditions: Randomized positions and particle velocities $q_i = 0, t_i = 0$
Calculate particle force
Calculate particle velocity using integration method
Move particles
Check boundary conditions
Done?
no
yes
Repeat calculation for next timestep
End simulation

Figure 3.1: Flow diagram for simulation

RK4 to this new method which is similar to a velocity-verlet integrator that can more easily and efficiently function in our simulation. This simulation is implemented by first analyzing the simple set of equations which refer to the change of position as well as change of momentum are shown in equation 3.1.

$$d\begin{bmatrix}\overrightarrow{x}\\ \overrightarrow{p}\end{bmatrix} = \underbrace{\begin{bmatrix}m^{-1}\overrightarrow{p}\\ 0\end{bmatrix}dt}_{\mathcal{A}} + \underbrace{\begin{bmatrix}0\\ -\overrightarrow{\nabla}U\end{bmatrix}dt}_{\mathcal{B}} + \underbrace{\begin{bmatrix}0\\ -\gamma_m\overrightarrow{p}\,dt + \sigma m^{1/2}dW\hat{x}\end{bmatrix}}_{\mathcal{O}} \tag{3.1}$$

where we use position x, momentum p, an infinitesimal unit of the Weiner process dW (which is a stochastic process which includes Brownian motion in the system, in other words this includes the thermal properties of the system), potential energy U, drag coefficient γ_m [units of $kg \cdot s^{-1}$] and variance σ. It is important to note here that the drag coefficient is calculated using drag on a charged particle as shown in the paper by Epstein [48]. From equation 3.1, $\mathcal{A}$ refers to a simple euler integration method, $\mathcal{B}$ refers to the forces acting on the particle and $\mathcal{O}$ refers to the drag on the particle as well as the Langevin thermostat. This thermostat introduces the thermal energy and therefore temperature into the system [54].

Some useful constants can be defined which incorporate mass m, temperature T and friction coefficient γ_m and a number R which is chosen randomly from a normal distribution which is centered at zero [52, 53].

$$c_1 = e^{-m\gamma_m \delta t}, \quad c_2 = \gamma_m^{-1}(1-c_1), \quad c_3 = \sqrt{k_B T(1-c_1^2)} \tag{3.2}$$

$$\begin{aligned}
\mathcal{B}: \quad & p_{n+1/2} &=& \quad p_n - \delta t \nabla U(x_n)/2 \\
\mathcal{A}: \quad & x_{n+1/2} &=& \quad x_n + \delta t m^{-1} p_{n+1/2}/2 \\
\mathcal{O}: \quad & \hat{p}_{n+1/2} &=& \quad c_1 p_{n+1/2} + c_3 m^{1/2} R_{n+1} \\
\mathcal{A}: \quad & x_{n+1} &=& \quad x_{n+1/2} + \delta m^{-1} \hat{p}_{n+1/2}/2 \\
\mathcal{B}: \quad & p_{n+1} &=& \quad \hat{p}_{n+1/2} - \delta \nabla U(x_{n+1})/2
\end{aligned} \tag{3.3}$$

An alternate way of viewing this method is to see each term in equation 3.3 corresponds to a step of $\mathcal{B} \to \mathcal{A} \to \mathcal{O} \to \mathcal{A} \to \mathcal{B}$. Where $\mathcal{B}$ is the momentum, $\mathcal{A}$ is the position, $\mathcal{O}$ is the momentum operator (which includes the temperature in coefficient c_3). Once the method of integration has been analyzed, the evaluation of an appropriate time step is necessary before running the simulation.

The value of the time step is very relevant when running a simulation. A shorter time step will correspond to a higher level of precision. The drawback of a shorter time step is a longer simulation time. The most efficient time step is calculated based on the movement of particles in our system with regard to their interaction with nearby particles [55]. The relevant time step will correspond to the smallest of the values calculated from equation 3.4

$$\Delta t \sim \min(\frac{l_p \gamma_m}{|F_{ext}|}, \frac{l_p^2 \gamma_m}{6 k_B T_d}) \tag{3.4}$$

where $l_p = \Delta$ is the characteristic system length represented by the average inter-particle spacing in our system. It can be noted that F_{ext}/γ_m will be the drift velocity of our particles, this shows that the first is the average amount of time it would take for a particle to collide with another particle based on the density of the dust. The second term in equation 3.4 is a characteristic amount of time it would take for the thermal energy to make the particles collide. More concisely, equation 3.4 helps choose a time-step depending on the larger of either thermal kinetic energy or kinetic energy relating to particle velocity.

One of the largest benefits of designing a new simulation is the variability of the conditions of the system. For our experimental system we would like to be able to freely change a number of parameters such as external magnetic field, dust particle parameters (such as dust charge, size, density and mass), external electric field, drag coefficient, initial velocity, temperature, simulation size and number of dust particles. The variation of these parameters facilitates in the testing of the theories derived previously in Chapter 2.

The general simulation system is that which is shown in figure 2.1 of a constant particle flow with a perpendicular magnetic field causing the compressional Lorentz force. This is shown in Figure 3.2 where the resulting density gradient can be seen. A secondary simulation system has also been used with no driving force and instead a constant compressional external force is considered. This is similar to a gravitational compression which can be seen in many experiments.

The confinement of the system is also important to take note of. In terms of boundary conditions, the simulation uses a circular boundary condition along the x direction and an electric field confinement along the y direction. As a result it is important to note the external electric field acting upon our system. This can be seen in Figure 3.3

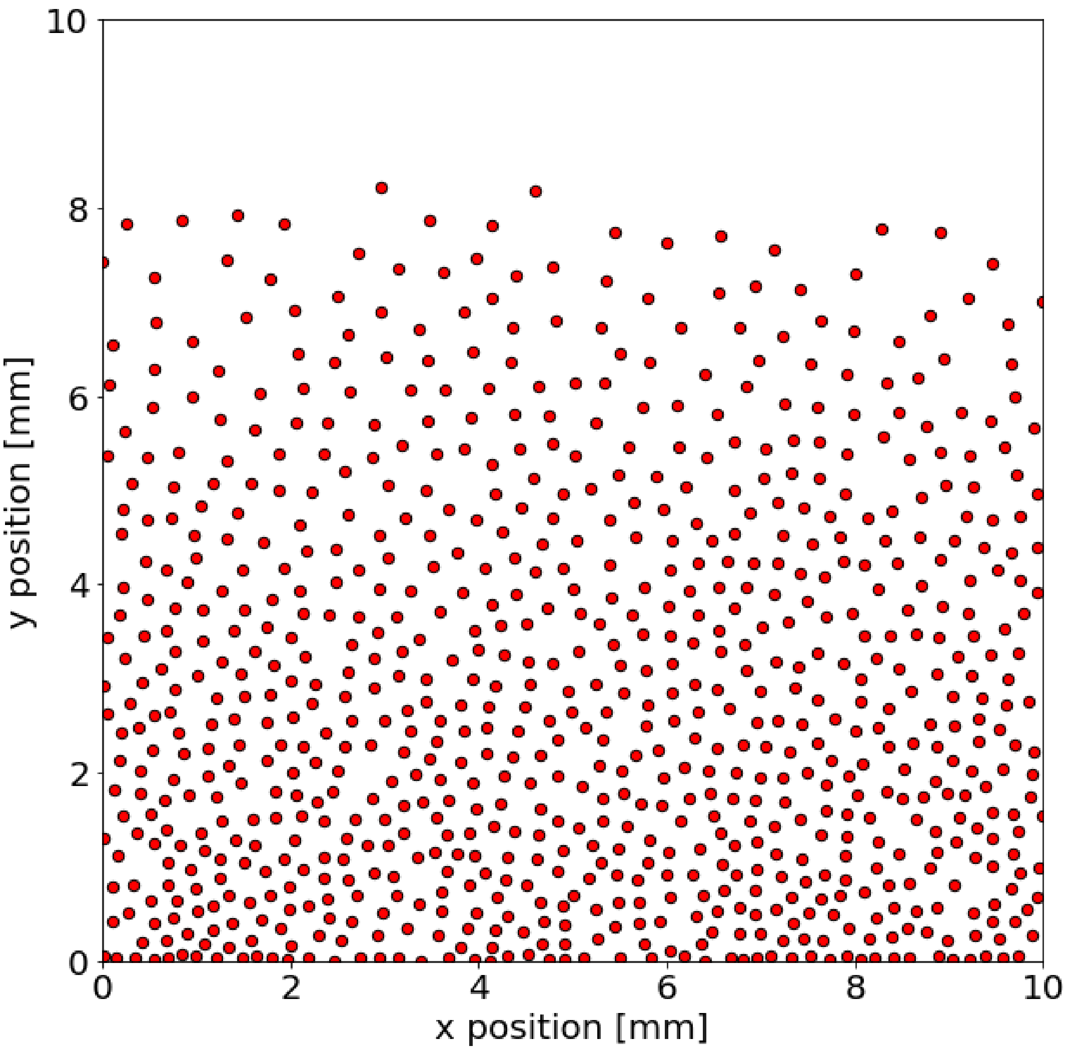


Figure 3.2: A plotted frame of the simulation in the case of an incident magnetic field and charged particles. In this image, the density gradient is visible as there are more particles along the lower portion of the simulation space than there are on the upper region.

3.2 Validity of Simulation

Various assumptions have been made in our theory. In this section these assumptions will be tested and validated in the MD simulation. The assumptions made in our system are shown below:

1. The system is isothermal (i.e., $\vec{\nabla} T = 0$)

2. The system has reached a steady state and is independent of time

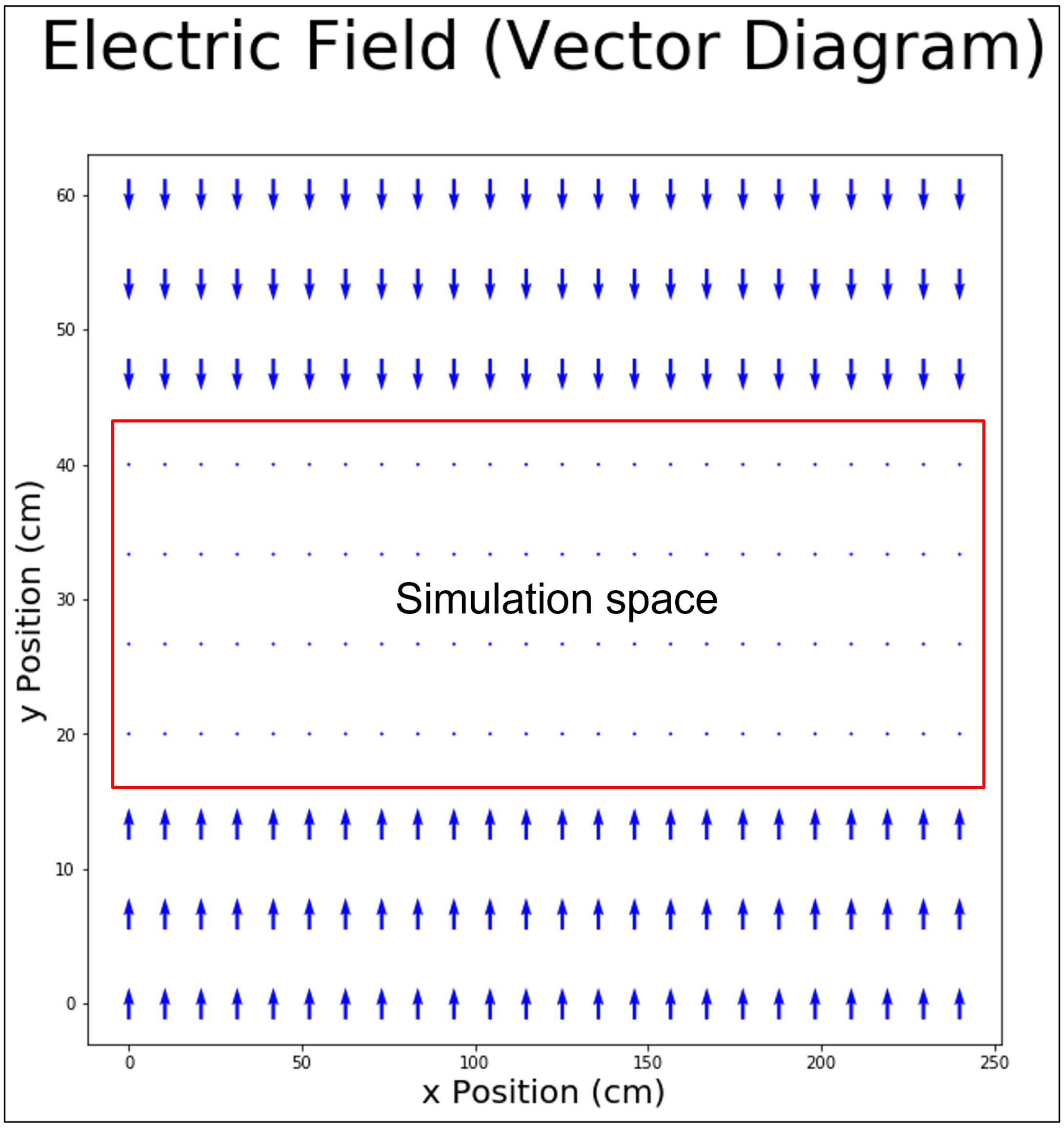


Figure 3.3: External electric field incident upon system. Experimentally, this is similar to the electric field which appears as a result of the chamber walls. Note that the external electric field in the center of the simulation region is zero which fits with our assumptions. This electric field aids in containment as the negatively charged dust particles will have a resultant force which pushes them back into the center region of the system which is where the analysis takes place

3. Magnetic field, flow direction and density gradient are all perpendicular

4. External electric field is approximately zero in region of evaluation

5. Interaction force is modeled from a shielded Debye potential

For the first assumption of an isothermal system the dust particle temperature is assumed to be constant in each dimension. In order to test this it is useful to calculate the temperature of each particle compared to both the x and y direction. An example of this calculation is shown in figure 3.4. This figure shows that the input temperature of the system matches the assumption of an isothermal system as it does not vary along our y position

Steady state is a necessary assumption for both high and low coupling theories. There

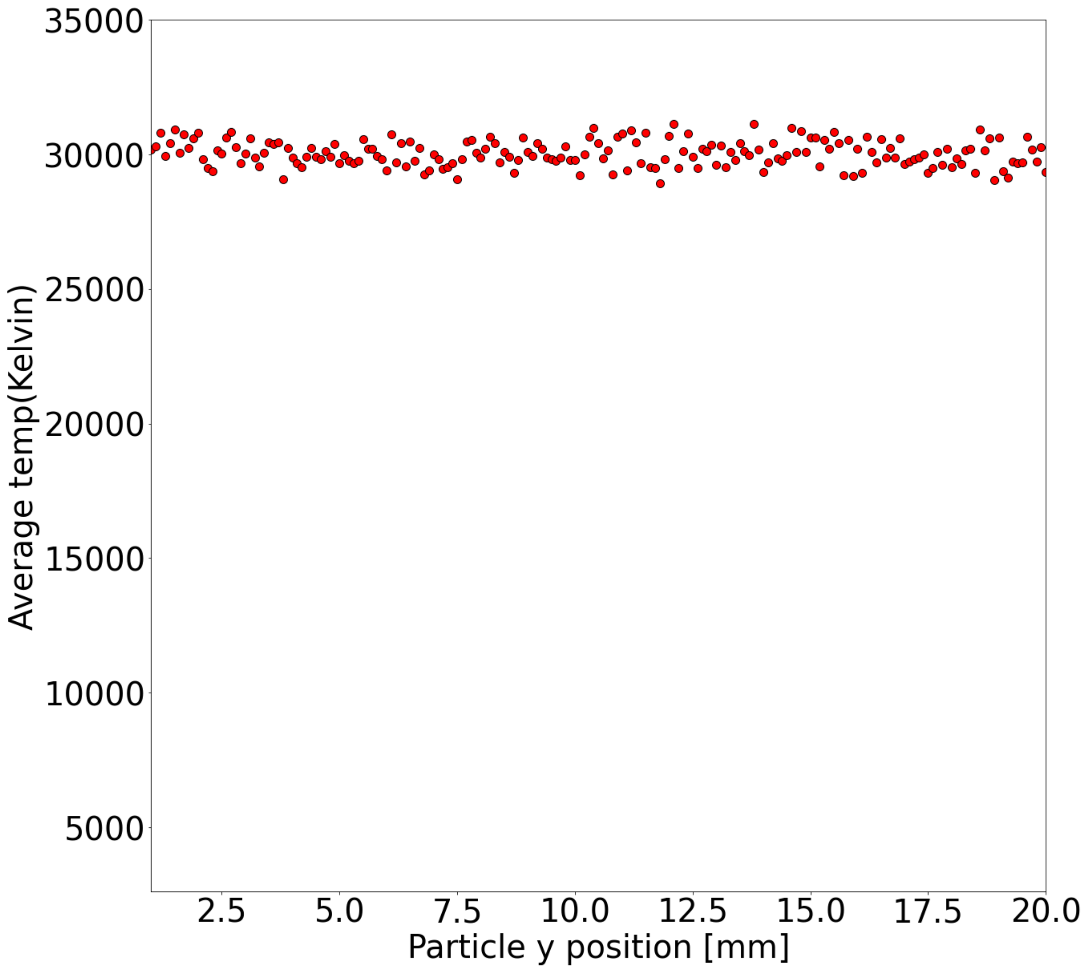


Figure 3.4: Average temperature vs y position. This shows a nearly constant temperature for each y position. The uncertainty in this calculation is evaluated by using the velocities of the dust particles and is further in analyzed in chapter 4

are multiple time scales relevant in the steady state assumption. The fastest of the relevant time scales is that of the dust charging process. This is derived in the paper by J. Goree [30] and the dust particle charging time is given by

$$\tau_q = K_\tau \frac{\sqrt{k_B T_e}}{r_d n_{plasma}} \tag{3.5}$$

where T_e is electron temperature and K_τ is a constant on the order of $10^3\ s\ \mu m\ cm^{-3}\ eV^{-1/2}$ found by a numerical solution of the continuous charging model with the assumption of no electron emission and non-drifting Maxwellians. Since n_{plasma} is very large, the dust charging time is on the order of 10^{-6} seconds which is very short compared to the simulation time (on the order of seconds). This means this time scale can be safely ignored for both our simulation and the experiment shown in chapter 5.

The next time scale is that of velocity. With a constant driving force, the velocity of these particles will reach steady state velocity depending on the magnitude of the driving force and the drag coefficient. A similar result has been shown in chapter 2 equation 2.2, $u_x = F_{drive}/\gamma_m$.

In order to calculate the characteristic time, an equation for the velocity as a function of time must be found. To start, Newton's second law is used

$$\sum F = F_{drive} - \gamma_m v = m\dot{v} \tag{3.6}$$

Integrating this equation,

$$\int_{v_0}^{v} \frac{dv}{F_{drive} - \gamma_m v} = \int_0^t \frac{dt}{m} \tag{3.7}$$

$$\Rightarrow \left. \frac{\ln\left(F_{drive} - \gamma_m v\right)}{-\gamma_m} \right|_{v_0}^{v} = \frac{t}{m} \tag{3.8}$$

rewriting and remembering that $\frac{\gamma_m}{m} = \gamma$

$$\Rightarrow \ln\left(\frac{F_{drive} - \gamma_m v}{F_{drive} - \gamma_m v_0}\right) = -\frac{\gamma_m t}{m} = -\gamma t \tag{3.9}$$

$$\Rightarrow \left(\frac{F_{drive} - \gamma_m v}{F_{drive} - \gamma_m v_0}\right) = e^{-\gamma t} \tag{3.10}$$

$$\Rightarrow v(t) = \left(v_0 - \frac{F_{drive}}{\gamma_m}\right) e^{-\gamma t} + \frac{F_{drive}}{\gamma_m} \tag{3.11}$$

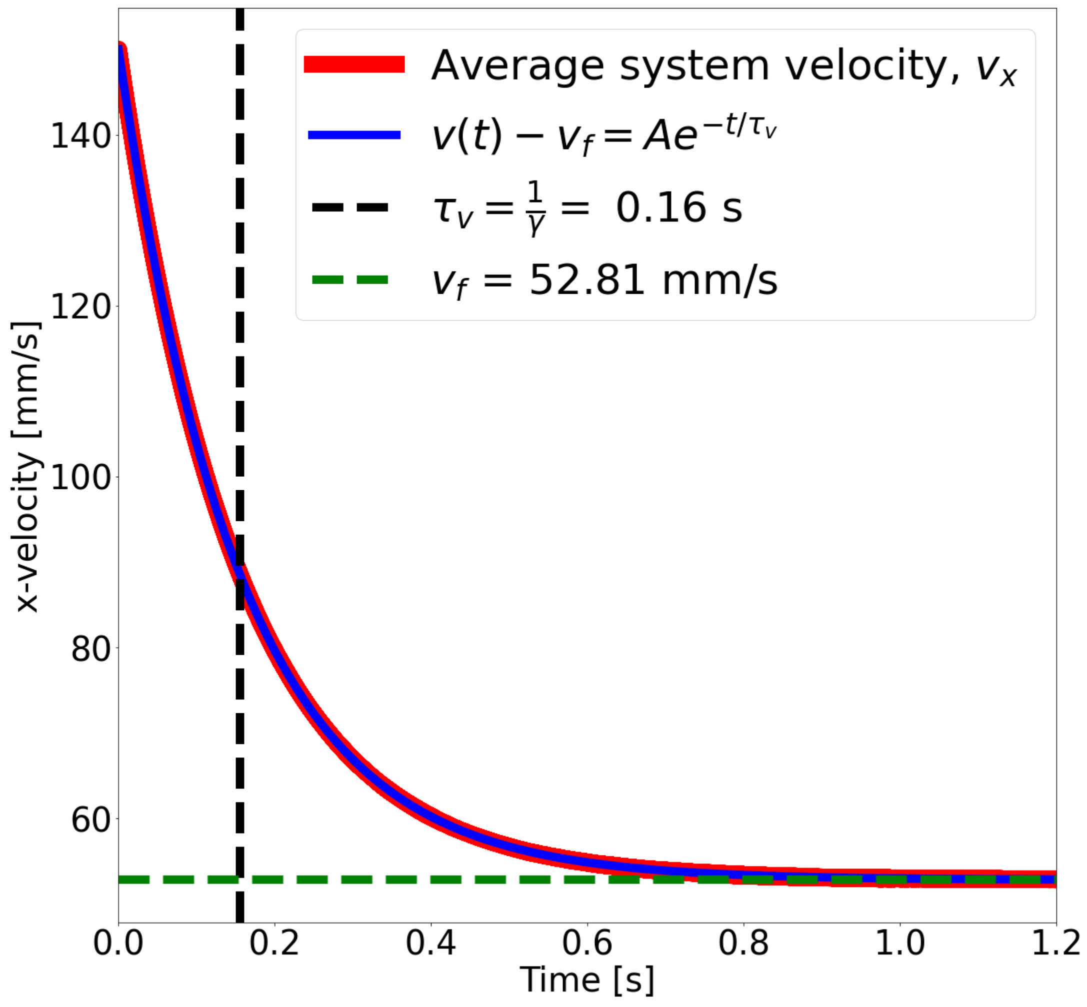


Figure 3.5: This plot shows a plot of simulation x-velocity data as well as an exponential fit of the data. Following equation 3.12, A = $v_0 - v_f$ where here $u_x = v_f$.

.

which, combining with equation 2.2 shown in chapter 2, $u_x = \gamma_m / F_{drive}$ yields

$$v(t) = (v_0 - u_x)\, e^{-\gamma t} + u_x \tag{3.12}$$

This allows the definition of a characteristic time constant τ_v

$$\tau_v = \frac{1}{\gamma} \tag{3.13}$$

The time constant shown in equation 3.13 as well as the function in equation 3.12 are both plotted in figure 3.5. From this it can be seen that the drag coefficient determines the time scale in the system as well as the final system velocity. It is valid to assume that, after a certain amount of time depending on system parameters, the system will reach a steady state with a constant velocity which is independent of the initial velocity of the system. The independence from initial velocity can be seen in figure 3.6. This time scale can also be seen while calculating the charge value for a low coupling system as is shown in Figure 3.7

The last, and most complicated, of the time scales is that of sedimentation time. This

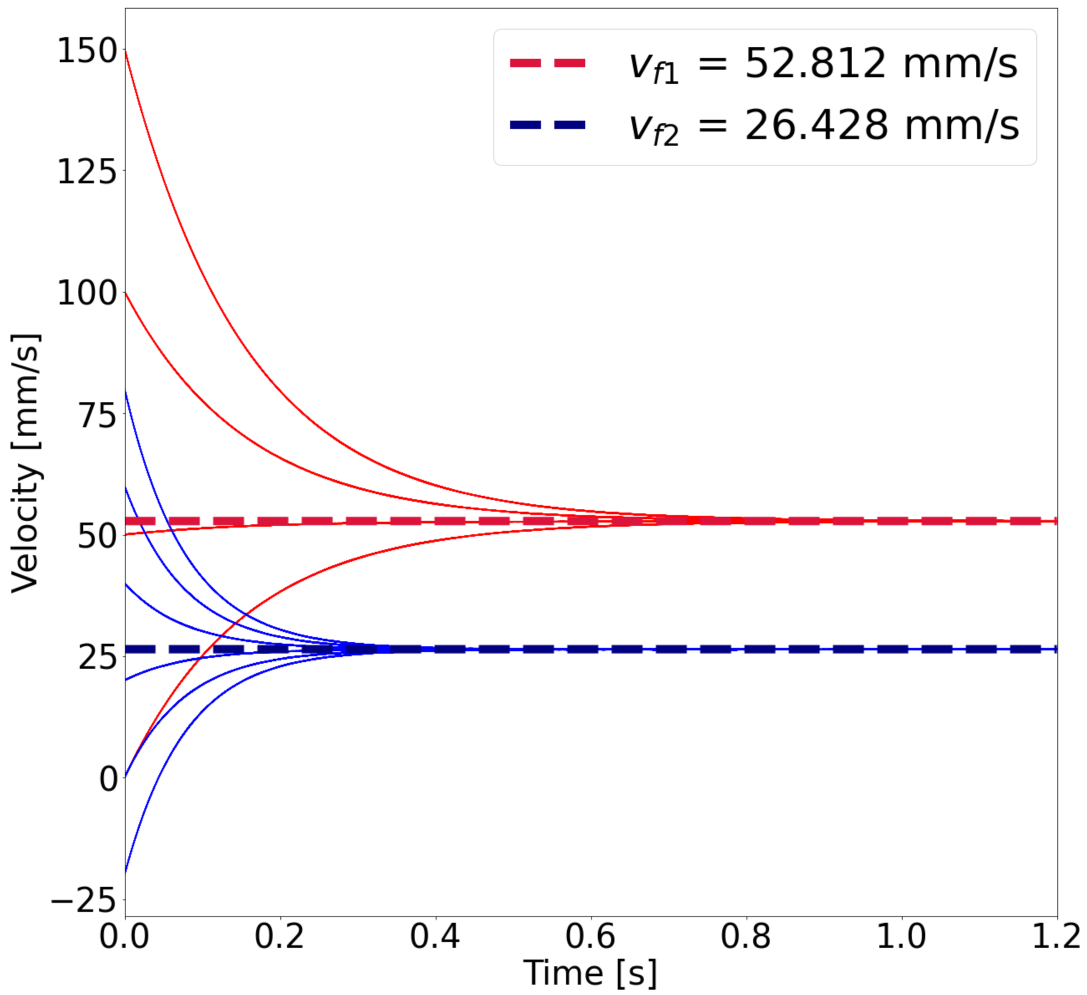


Figure 3.6: This diagram shows multiple simulation runs varying both initial velocity and drag coefficient. A lower drag coefficient (Red) corresponds to a higher final velocity. A high drag coefficient (Blue) corresponds to a lower final velocity. This is independent of initial velocity as multiple initial velocities converge to the same velocity value

.

time scale will be relevant mainly for the high coupling theory. After reaching a constant velocity (also known as the drift velocity), the dust particles in the system will still need to reorganize

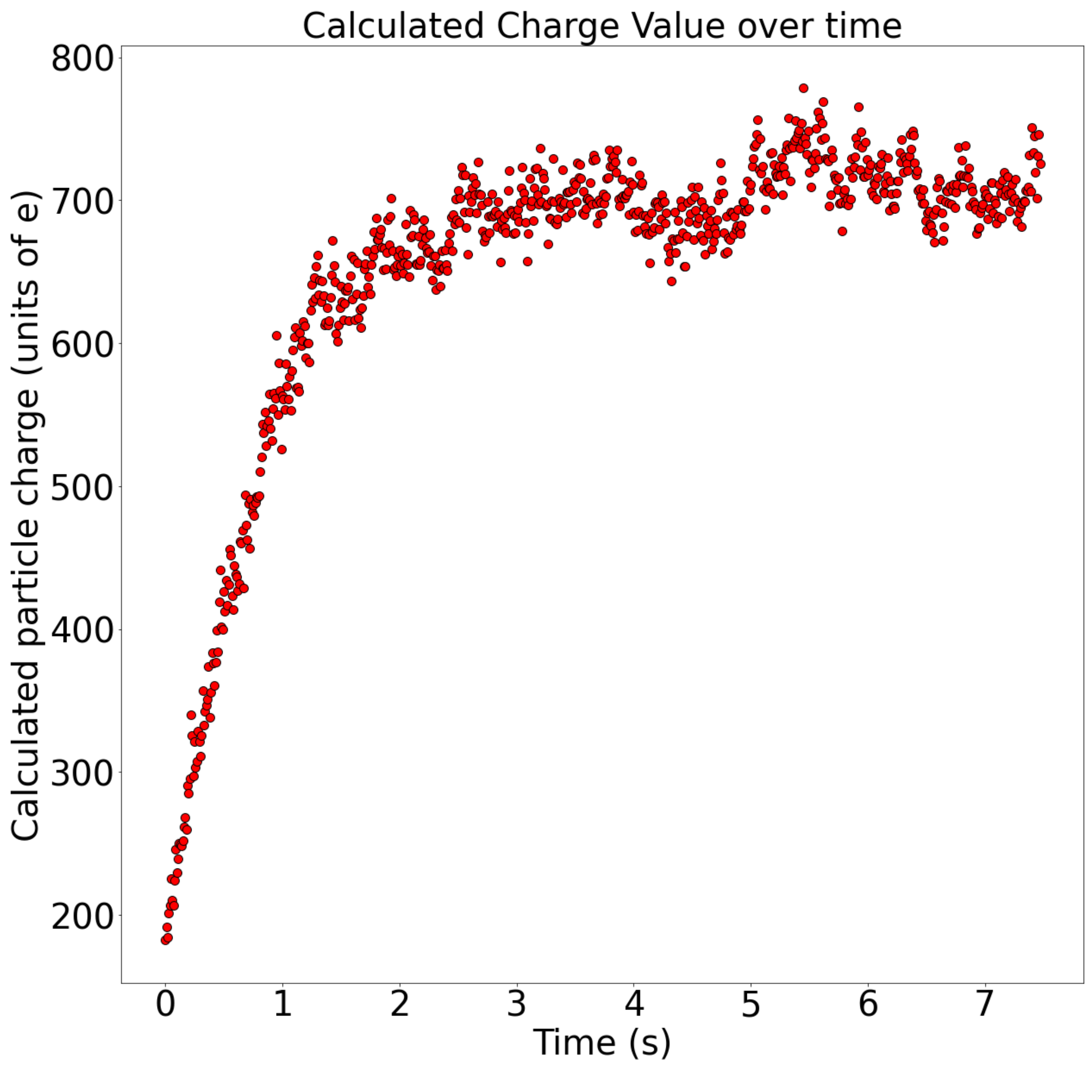


Figure 3.7: This diagram shows the calculation of dust charge value for a lowly coupled system. For a single simulation with an input charge value of 700 electron charges, the calculation becomes or accurate over time as the system comes to steady state.

to achieve the pseudo-crystalline state. This sedimentation can be seen from the density versus time plot shown in figure 3.8. The most important thing to note here is that the time scale of sedimentation depends upon the compression force (here being Lorentz force) and as a result, the particles which are lower on the y axis organize faster than those at a higher y position. This means that the sedimentation time will be proportional to the coupling coefficient of the system. This sedimentation time is generally longer than the other time scales and therefore can not be ignored. The exact physics of this system is left as future work and is accommodated by allowing the system charge to reach a constant value as shown in figure 3.7.

Another assumption that is made is a Maxwellian velocity distribution for the case of

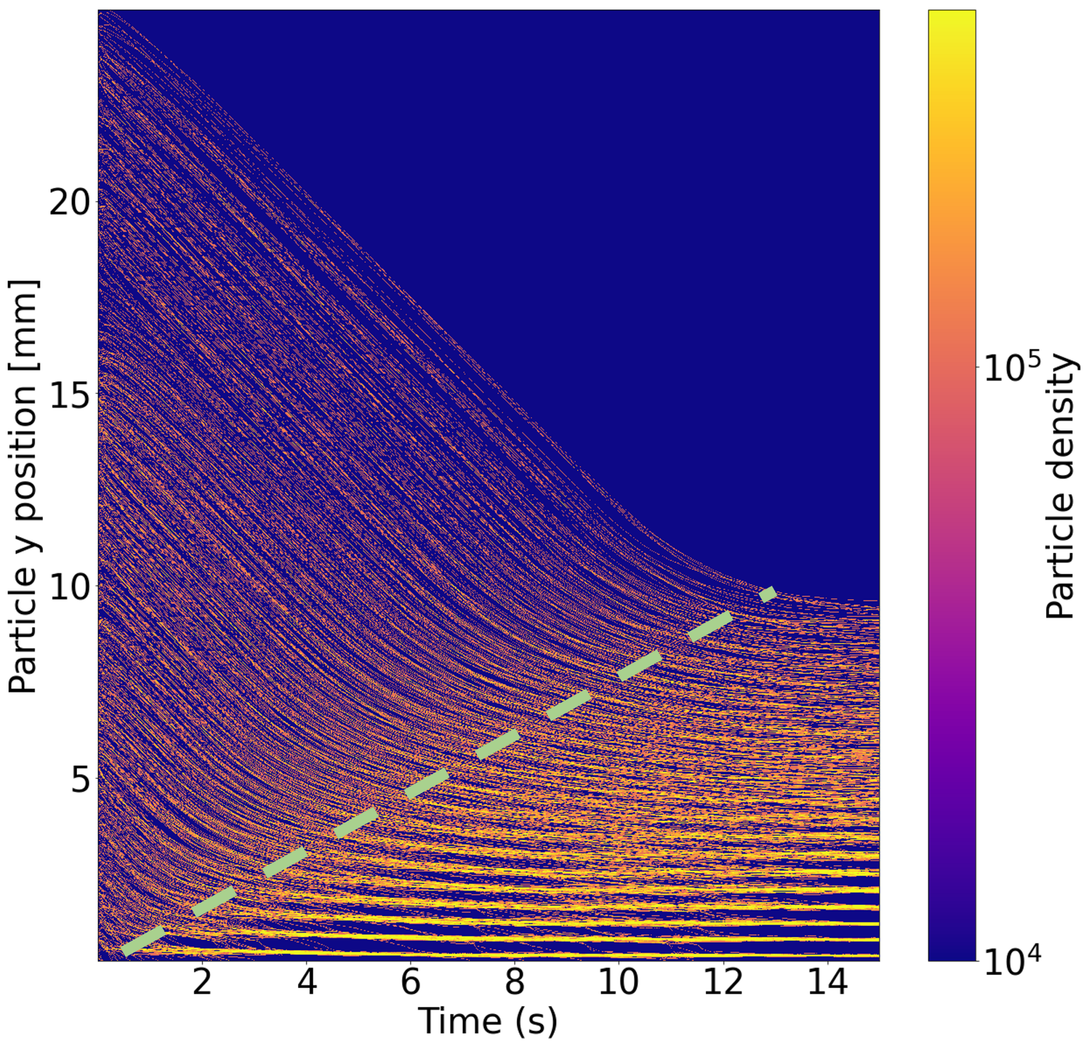


Figure 3.8: Results shown here are from a system with a high compression force resulting in a highly coupled system. The amount of time to reach a steady state in terms of density and therefore sedimentation is proportional to the y position and therefore the density and coupling coefficient at that location

low coupling. By calculating the velocity of each particle it can be shown that this follows a Maxwell-Boltzmann curve as is shown in Figure 3.9. In this figure it is shown that the expected value of the standard deviation matches the value of standard deviation calculated from the velocity data in the system. This plot also shows the average velocity of the system which is necessary for charge calculations. In the case of the agreement of the experimental standard deviation to standard deviation calculated from the input temperature of the system shows the validity of the assumption of a uniform distribution.

In order to evaluate the importance of the standard deviation in the system, two equations must be compared. The first equation is that of a standard Maxwell-Boltzmann distribution.

$$f(v)d^3v = \left(\frac{m}{2\pi k_B T}\right)^{3/2} e^{-\frac{mv^2}{2k_B T}} d^3v \tag{3.14}$$

Or, in the case of one dimension,

$$f(v_x)dv_x = \sqrt{\frac{m}{2\pi k_B T}} e^{-\frac{mv^2}{2k_B T}} dv_x \tag{3.15}$$

The second equation that this will be compared to is that of a normal distribution in a single dimension is given as

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \tag{3.16}$$

Where here, μ is the average of the variable x. Comparing these two equations yields an equation for the standard deviation σ in terms of the dust temperature and dust mass.

$$\sigma = \sqrt{\frac{2k_B T_d}{m_d}} \tag{3.17}$$

Utilizing this standard deviation allows the calculation of the system temperature as is shown in figure 3.9. It is also shown in this figure that by finding the standard deviation by fitting a Maxwellian distribution to the system and comparing this to the standard deviation by using equation 3.17 yields nearly identical values. Knowing the assumptions in our system and the validity of the simulation allows us then to begin analysis of the theories through use of this simulation.

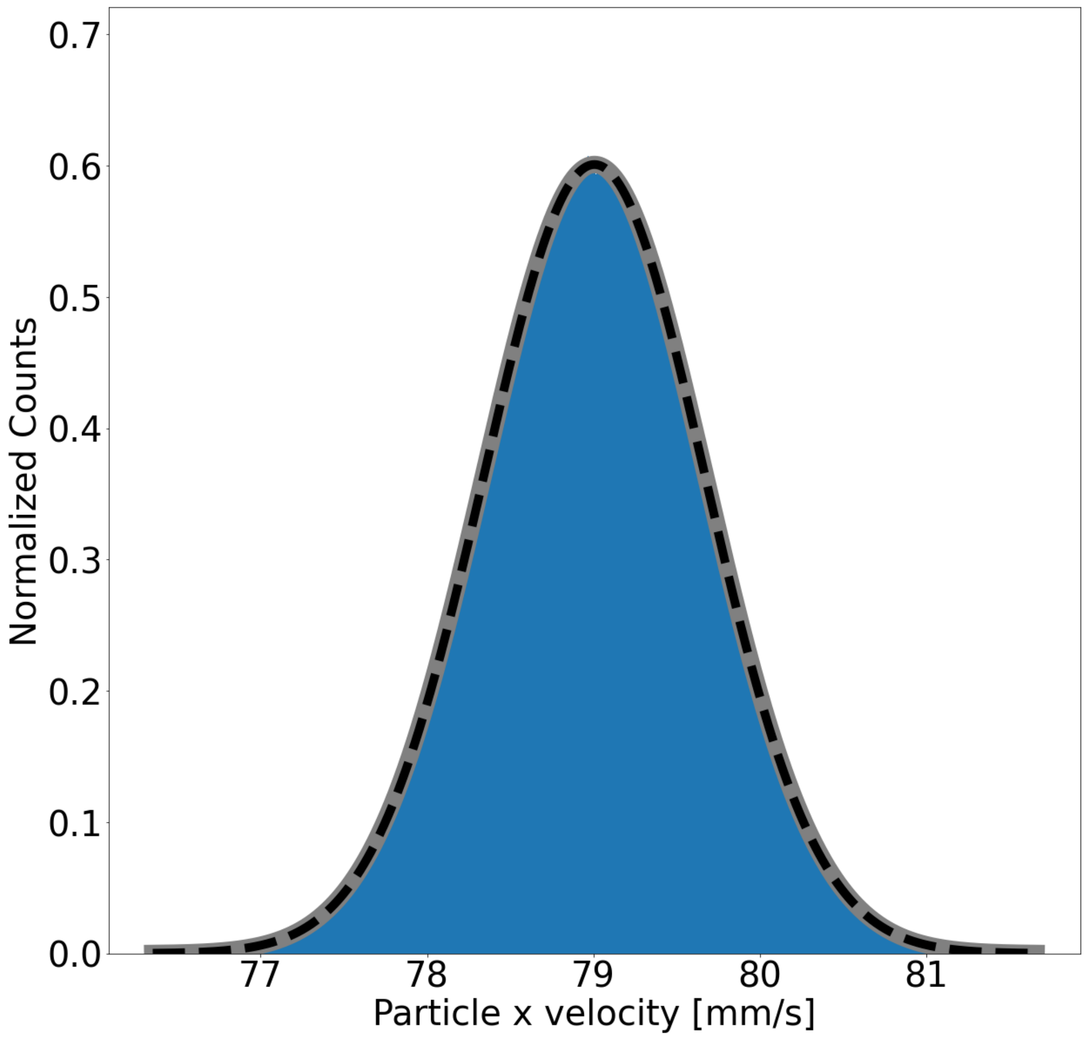


Figure 3.9: Particle x velocity histogram. The grey line corresponds to the fit from the input temperature corresponding to standard deviation $\sigma = \sqrt{\frac{2k_B T_{input}}{m_d}} = 0.664679$ while the black dashed line corresponds to a numerical fit of the data to a Maxwellian distribution which corresponds to $\sigma = 0.666408$. This distribution also has an average velocity of 79 mm/s

Chapter 4

Results and Analysis

In this chapter we will analyze data which is a result of various simulation runs carried out using the molecular dynamic simulation discussed in chapter 3. These simulation runs vary system parameters such as dust density, magnetic field, dust temperature and input charge value. Data collected from these simulation runs will be used to show the analytical techniques used to verify the theories derived previously in chapter 2 for cases of high and low coupling as well as the parameter spaces for which each theory is valid.

4.1 Analytical Codes

In order to study the data, analytical codes first needed to be developed. The data which has been collected to be analyzed was the output of this simulation. The simulation produced both positional and velocity data for the dust particles which are similarly obtainable in the case of experimental data. Both particle positions and velocity data can be found in an experimental system by analyzing particle positions from video data. This has been achieved in previous experiments using particle image velocimetry (PIV) [56, 57, 58] as well as by tracking individual particle positions using particle tracking velocimetry (PTV) [59].

Show below are the various analytical codes needed to evaluate our simulations. These codes were all written in Python and can be found in appendix B.

1. Low coupling analytical codes:

(a) Dust particle temperature evaluation from dust velocity data:

Dust particle velocity u_{array} $\rightarrow$ Standard deviation of velocity, σ_{u_x} $\rightarrow$ Dust temperature, T_d

(b) Dust particle average velocity from dust velocity data:

Dust particle velocity, u_{array} $\rightarrow$ Average of dust velocity u_x

(c) Dust density as a function of y position and fit from positional data:

Dust particle position $\rightarrow$ histogram of data for each time step $\rightarrow$ sum over all time steps after steady state $\rightarrow$ Fit of function to histogram data

(d) Dust charge calculation:

Dust velocity, u_x, dust temperature, T_d, and dust density $N(y)$ $\rightarrow$ Average dust charge value, q_d

(e) Code for analysis of calculated parameters as a function of y position using N(y)

(f) Coupling parameter vs charge calculation

2. High coupling analytical codes:

(a) High coupling charge calculation from positional data:

Select particle and find particle position $\rightarrow$ find nearest neighbors $\rightarrow$ use nearest neighbors position to find particle charge $\rightarrow$ average over particles in system

(b) Charge calculation over time

(c) Code for analysis of calculated parameters as a function of y position Find particle individual particle charge q_i $\rightarrow$ count y position $\rightarrow$ find equation for charge as a function of y

(d) Coupling parameter vs charge calculation

3. Additional analytical codes:

(a) Density over time analysis:

Calculate histogram of positions for a single time step $\rightarrow$ plot heat map for each time step

(b) Code for comparison of charge calculation of both high and low coupling methods

4.1.1 Low Coupling Analytical Codes

The result of the low coupling theory is given by equation 2.13 which was previously derived in Chapter 2 is shown again below

$$q_d = \frac{k_B T_d}{B_o u_x} \frac{\nabla_y n}{n} \tag{4.1}$$

In order to find the dust charge value, the quantities of dust temperature T_d, density n and average particle velocity u_x must be found. In order to evaluate this result, analysis techniques were developed to find these quantities from the available data sets.

The first code which was developed is that which evaluates the dust particle temperature. In order to calculate this, the velocity data is used and the x velocity of each dust particle is taken. The standard deviation of the velocity distribution is then calculating using this velocity data, dust temperature can subsequently be found as shown in equation 4.3. This calculation assumes a Maxwellian distribution for the data which is verified in figure 3.9 which shows a plot of both the best fit of a Maxwellian distribution the input temperature as well as the plot of the distribution which fits the histogram data.

$$\frac{1}{2} m\sigma^2 = k_B T_d \tag{4.2}$$

$$T_d = \frac{1}{2} \frac{m\sigma^2}{k_B} \tag{4.3}$$

These two values show almost no deviation (less than 0.3 percent difference on average for a lowly coupled system) as shown in figure 3.9 this also demonstrates that the assumption of a Maxwellian distribution is valid. The second code is a simple one which takes the average of the velocity data in each time step then determines the average of each of these values over the entire simulation after the system reaches steady state. This gives the value of the average velocity of the dust particles in the system.

The next analytical code for the low coupling theory is that which determines the dust density in the system, N(y). Similar to the previous two methods, this can be done for each time step and is accomplished by plotting a histogram of the particle position for each direction. This plot of the histogram yields the dust particle density for each time step and is shown in figure 4.3 and data is used after the system reaches this steady state. The values of density can be further averaged over all time steps to yield the figure shown in 4.4. This figure also gives the result of an exponential fit to the data. As a result of the compressional force (in this case Lorentz force), the density will be non-uniform along the direction of the compression force. This is shown in figure 4.4

The determination of a valid fit for this data is done best by comparing the distribution of particles to that of air in the atmosphere. The similarities between these two systems lie in the fluid state of both. The equations for pressure and density are shown in U.S. Standard Atmosphere (1976) [60] which shows that the pressure and density follow an equation shown given by

$$P = P_b \left[\frac{T_{M,b}}{T_{M,b} + L_{M,b}(H - H_b)} \right]^{\left(\frac{g_0' M_0}{R{*}L_{M,b}}\right)} \tag{4.4}$$

where the quantities H_b, $L_{M,b}$, $R*$, M_0 and g_0' refer respectively to geopotential height, molecular-scale temperature gradient, gas constant, molecular weight and gravitational acceleration at sea level. For the case of $\nabla T = 0$ and therefore $L_{M,b} = 0$, equation 4.4 evaluates to

$$P = P_b \text{exp} \left[\frac{-g_0' M_0 (H - H_b)}{R * T_{M,b}} \right] \tag{4.5}$$

$$\Rightarrow \rho = \rho_b \text{exp} \left[\frac{-g_0' M_0 (H - H_b)}{R^* T_{M,b}} \right] \tag{4.6}$$

The equation for density shown in equation 4.6 can be further evaluated where density ρ in the case of atmosphere is equivalent to dust density n for the case of the dusty plasma system. In the atmospheric case, the force of compression is characterized by gravitational force. Similarly, $TR^* = k_b T$ and replacing quantities used in the dusty plasma system this equation can be rewritten to fit a more appropriate to this research form

$$\rho = \rho_b \text{exp}\left[\frac{F_{compression}\Delta y}{nk_bT_d}\right] \tag{4.7}$$

This equation shows the relevance of the exponential fit as a function of y coordinates. Equation 4.7 is therefore similar to the equation of fit which is shown in figure 4.4.

Lastly, average dust charge value in the system is calculated. This is accomplished by first running the previous three analytical codes to get dust velocity, u_{ave}, dust temperature, T_d, and the dust number density distribution, $N(y)$. Using these values, the final code utilizes equation 2.13 to calculate the dust charge value for a single time step. This is then averaged over all time steps after steady state has been reached to find the average dust charge value in the system. It is important to note that this has been averaged both over all particles and particle positions as well as for all time steps.

4.1.2 High Coupling Analytical Codes

The main difference between these two methods is the discrete nature of the highly coupled crystal-like structure compared to the more fluid and continuous nature of the low coupling system. For the case of high coupling, the dust charge value found by analyzing the crystal-like structure seen in 4.1.

The first high coupling analytical code calculates the charge of a single particle by using the positional data in the system. In order to calculate dust charge for a highly coupled system, equation 2.26 is used. This necessitates the measurement of particle positions from the simulation.

In order to find the inter-particle forces, a single particle is chosen. The distance between this particle and all particles are found and the six nearest neighbors are selected. The positions of these nearest neighbors are then plugged into equation 2.26. The result of this first code is the charge value of a single chosen particle. A sample of this calculation is shown in table 4.1. This charge is then averaged over all particles in the system (particles are assumed to have the same particle charge). A flow diagram of this calculation is shown in figure 4.2.

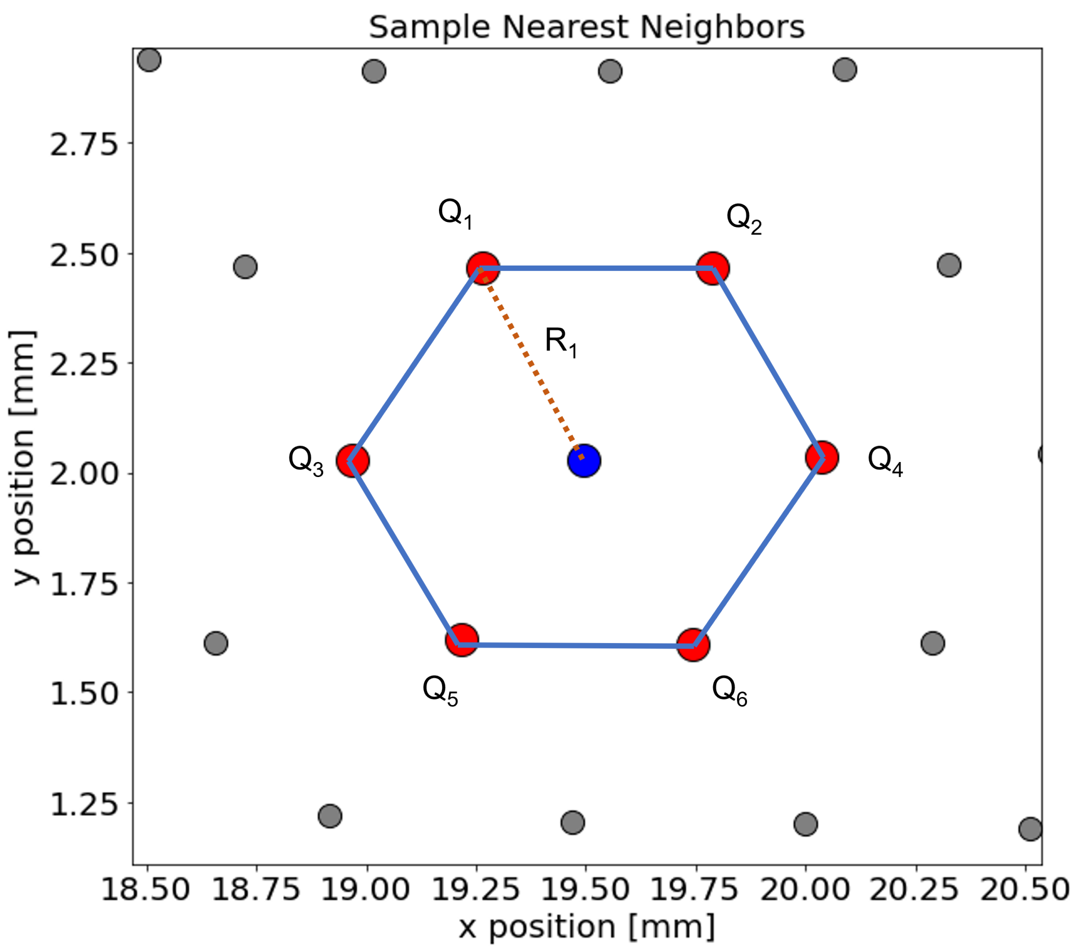


Figure 4.1: Example of hexagonal structure in a highly coupled dusty plasma with $\Gamma \gg 1$

The second analytical code calculates the density of the system for a single time step. The density is then plotted on a heat map to show the evolution of the system over time. A sample of this code is shown in figures 4.3 and 4.5 and the analytical code is given in the appendix.

Similarly, for the third analytical code, charge is calculated for each time step and plotted as a function of time. A sample calculation of this code is shown in figure 3.7

4.2 Results

4.2.1 Low Coupling

For the case of low coupling, the assumptions must once again be taken into account, these assumptions are:

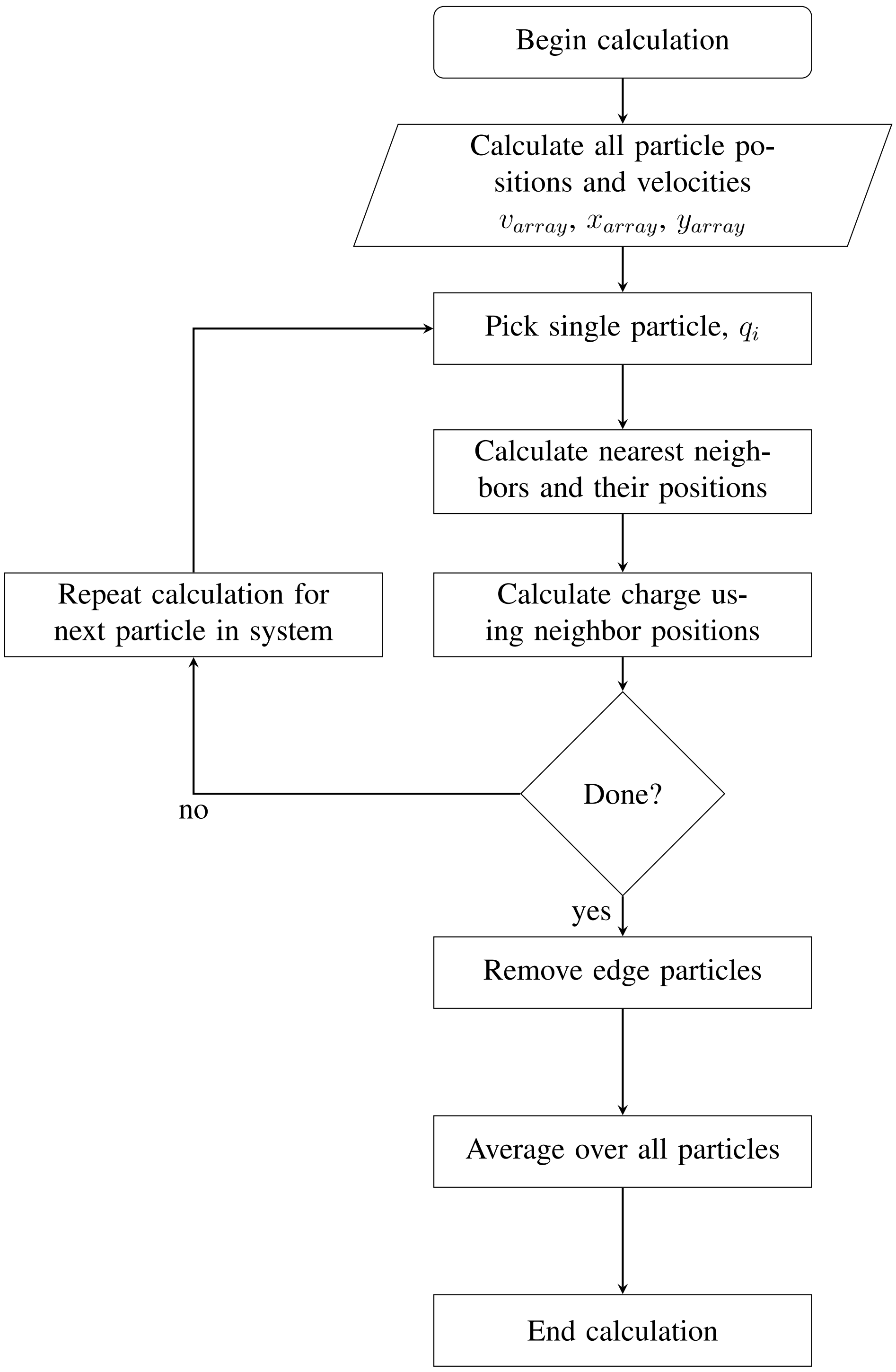

Begin calculation
Calculate all particle positions and velocities v_{array}, x_{array}, y_{array}
Pick single particle, q_i
Calculate nearest neighbors and their positions
Calculate charge using neighbor positions
Done?
no
Repeat calculation for next particle in system
yes
Remove edge particles
Average over all particles
End calculation

Figure 4.2: Flow diagram showing calculation method for average particle charge for the case of high coupling in the system

1. The system is isothermal (i.e., $\nabla T = 0$)

2. The system has reached a steady state and is independent of time

3. Magnetic field, flow direction and density gradient are all perpendicular

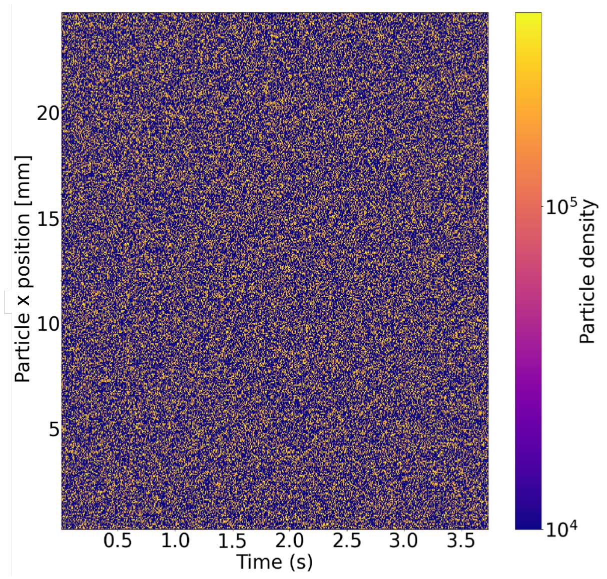
4. External electric field is approximately zero in region of evaluation

5. Interaction force is modeled from a shielded Debye potential

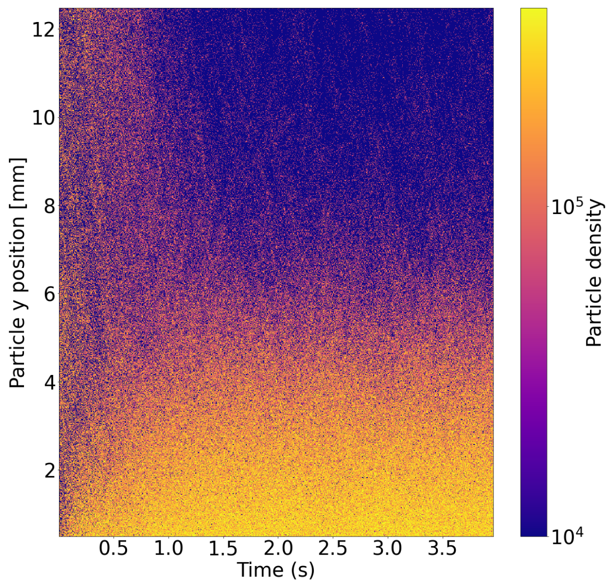
6. Density is non-uniform perpendicular to magnetic field and flow, and uniform along the flow direction

While most of these assumptions have been previously verified in the Simulation chapter, the case of density must be analyzed for each of the coupling cases. Figure 4.3 shows the particle density along both the x and y direction in the case of low coupling. From these figures it can be seen that along the x direction there is uniform particle distribution while along the y direction (direction of compressional force) there is a clear gradient which forms.

Calculating the quantities of dust temperature, density, and drift velocity using the previously mentioned analytical codes, particle charge is calculated using equation 2.13. A sample



(a)



(b)

Figure 4.3: This figure shows an example of (a) the particle density along the x direction as a function of time. This shows that for our system, the particle density in x is uniform in time as well as uniform in x. Or, in other words, $\nabla_x x = 0$ and $\partial x/\partial t = 0$ (b) Shows a plot of particle density as a function of the y direction. From this plot it can be seen that the system reaches a steady state after about one second for this sample as well as being non-uniform in the y direction (or $\nabla_y \neq 0$).

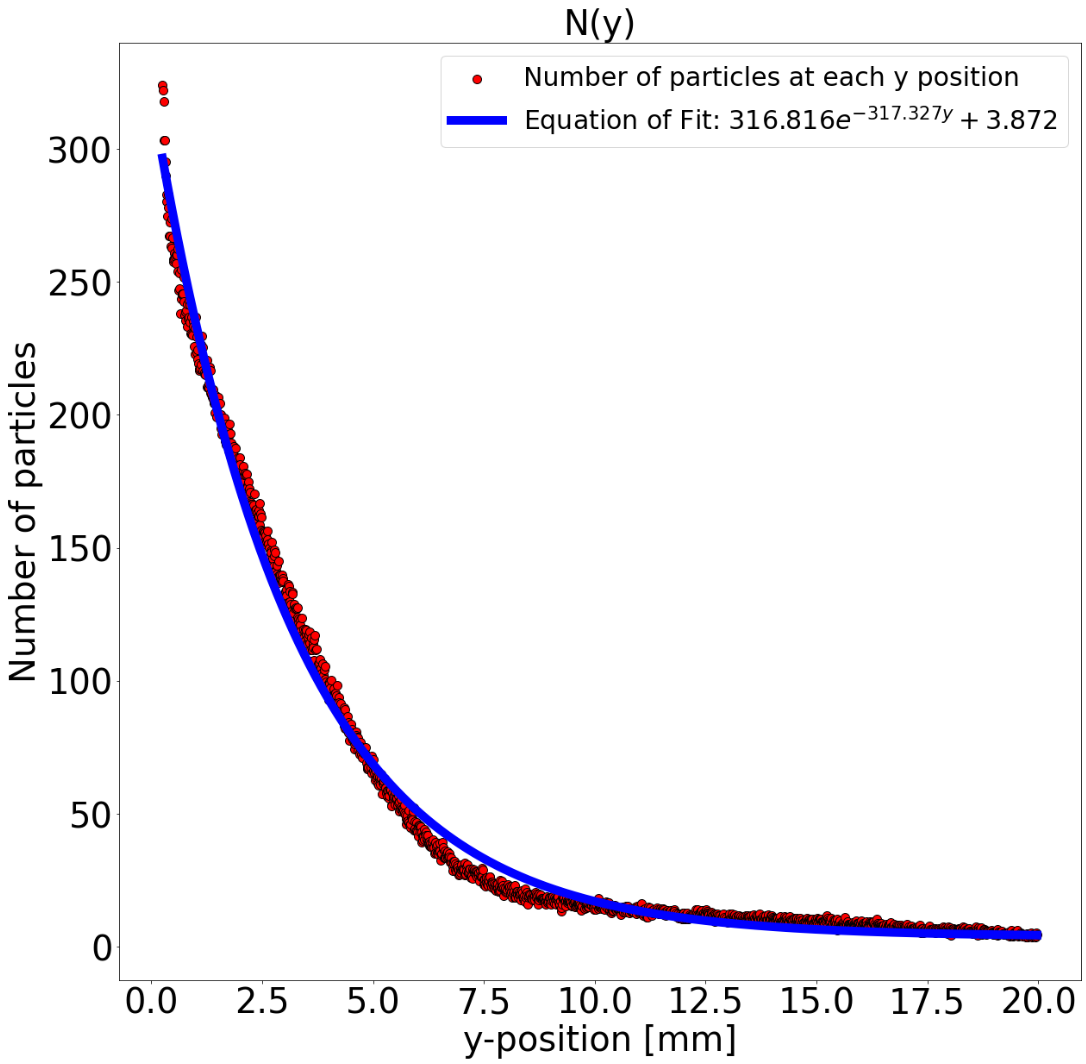


Figure 4.4: Exponential fit for number of particles in each bin (histogram) in the system for the case of low coupling. This exponential fit gives the equation for N(y) in our system. Using this result and the values for $u_x = 79cm/s$ and $T = 30,000K$ a dust charge value of $q = 974.15e$ which, when compared to $q_{input} = 1000e$ gives very reasonable agreement with only 2.63 percent difference.

calculation of this method is shown in figure 4.4.

In terms of error analysis, this comes down to the parameters which have been used to calculate the charge. For equation 2.13 this refers specifically to the error in finding dust temperature, dust density, and dust drift velocity.

For the average dust drift velocity, the standard error involved in this calculation follows the equation

$$SE = \frac{\sigma}{\sqrt{N}} \tag{4.8}$$

where σ is the standard deviation of the distribution and N is the number of data points. For the calculation of velocity, standard deviation is dependent upon the temperature in the system by the equation.

$$\sigma = \sqrt{\frac{k_B T_d}{m}} \tag{4.9}$$

The result of this is that for a lower temperature system, the standard deviation of the velocity (and therefore standard error) is smaller. Inadvertently, because of the equation for coupling coefficient shown in equation 1.16, provided the two have the same charge and inter-particle energy, a highly coupled system would have a lower error than a lowly coupled system. The result of this is that the error can be minimized by having more data points as well as a lower temperature.

For the calculation of dust density, this value is calculated from the positional data in the system. This means that the uncertainty in calculating dust density is coupled with the uncertainty in finding the the positions of the particles in the system. This uncertainty can be reduced by more accurately determining particle positions in the system.

4.2.2 High Coupling

In the case of a high coefficient of coupling in the system (Where $\Gamma_s \gg 1$ [5]) a crystalline structure begins to form and the dusty plasma is said to be in a solid-like state. This crystal-like structure is visible in figure 4.5 which is generated from the data created by the molecular dynamic simulation previously discussed.

The assumptions made for the case of High Coupling are the same as for low coupling with one additional assumption:

1. The system is isothermal (i.e., $\nabla T = 0$)
2. The system has reached a steady state and is independent of time
3. Magnetic field, flow direction and density gradient are all perpendicular

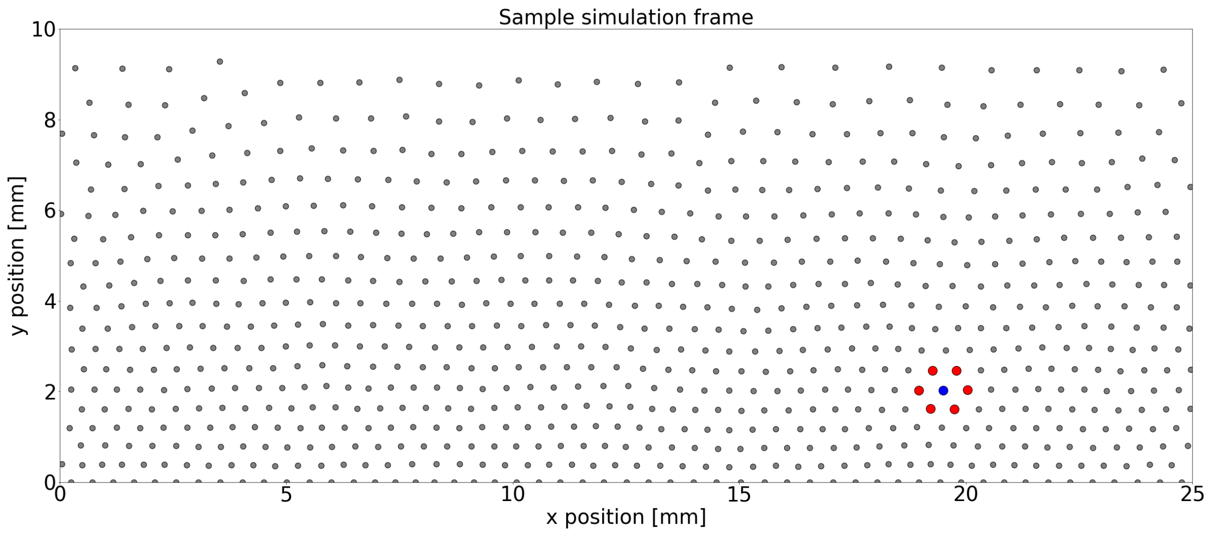


Figure 4.5: A frame taken from a single run of the simulation shown in chapter 3. This frame shows a visible crystal-like structure in a regime of high coupling.

4. External electric field is approximately zero in region of evaluation

5. Density is non-uniform along y-direction

6. **Interaction force is modeled from a shielded Debye potential**

The analysis for the high coupling case allows for similar methods to verify these assumptions as shown in the previous section. Similarly, the assumption of a Debye shielded potential has been shown to be valid for the case of dusty plasmas.

Here each of the particles are labeled individually and the inter-particle forces are calculated using each of their radial distances from the center particle of this asymmetric hexagon. This is then used to calculate the dust charge value. A sample of this calculation is shown in table 4.1.

Though it is difficult to discern from figure 4.1, the differences in y coordinates between each of the layers can be seen in table 4.1. This shows that $\Delta y_{upper} > \Delta y_{lower}$ which is predicted from the theory on the effect of compressional force on the solid-like pseudo-crystalline system. The charge value for the data from the molecular dynamic simulation fits well with the input data. In order to further analyze this theory, this same calculation can be done for every particle in the system.

Particle	x_n [mm]	y_n [mm]	R_n [mm]
Center	19.4940	2.0297	-
1	19.7876	2.4641	0.5242
2	19.2658	2.4668	0.4931
3	18.9675	2.0273	0.5432
4	20.0372	2.0344	0.5266
5	19.2174	1.6186	0.4956
6	19.7444	1.6090	0.4895
	$Q_{calculated}$ =	8277.3e	
	Q_{actual} =	8000e	
	Percent Difference =	3.403$\%$	

Table 4.1: Positional data calculated from image given in 4.1. Each individual force is calculated and then used to find the particle charge using equation 2.26

This same calculation is done on a larger scale by following the flow diagram shown in figure 4.2. This calculation is done by finding first the charge for a single particle and then averaging over all particles in the system. This is further averaged for each time step in order to ensure a more accurate calculation.

For the calculation of standard error in this system, equation 4.8 can once again be used. Here, this error is largely dependent upon the knowledge of the particle positions in the system.

Similarly to the case of low coupling, the calculation for charge can be calculated and compared to coupling parameter to show the validity of this theory for various values of coupling parameter.

4.3 Discussion

4.3.1 Comparison to Previous Methods and Accuracy

As shown in the previous section, both high coupling and low coupling methods have percent differences which are less than 5$\%$. This percent difference can be compared to previous experimental charge measurement methods such as that shown in the paper by Konopka et al. [12] which show a percent difference of between 10-20 $\%$ from expected values. This increase in precision for charge determination is a very valuable addition to the field of dusty plasmas.

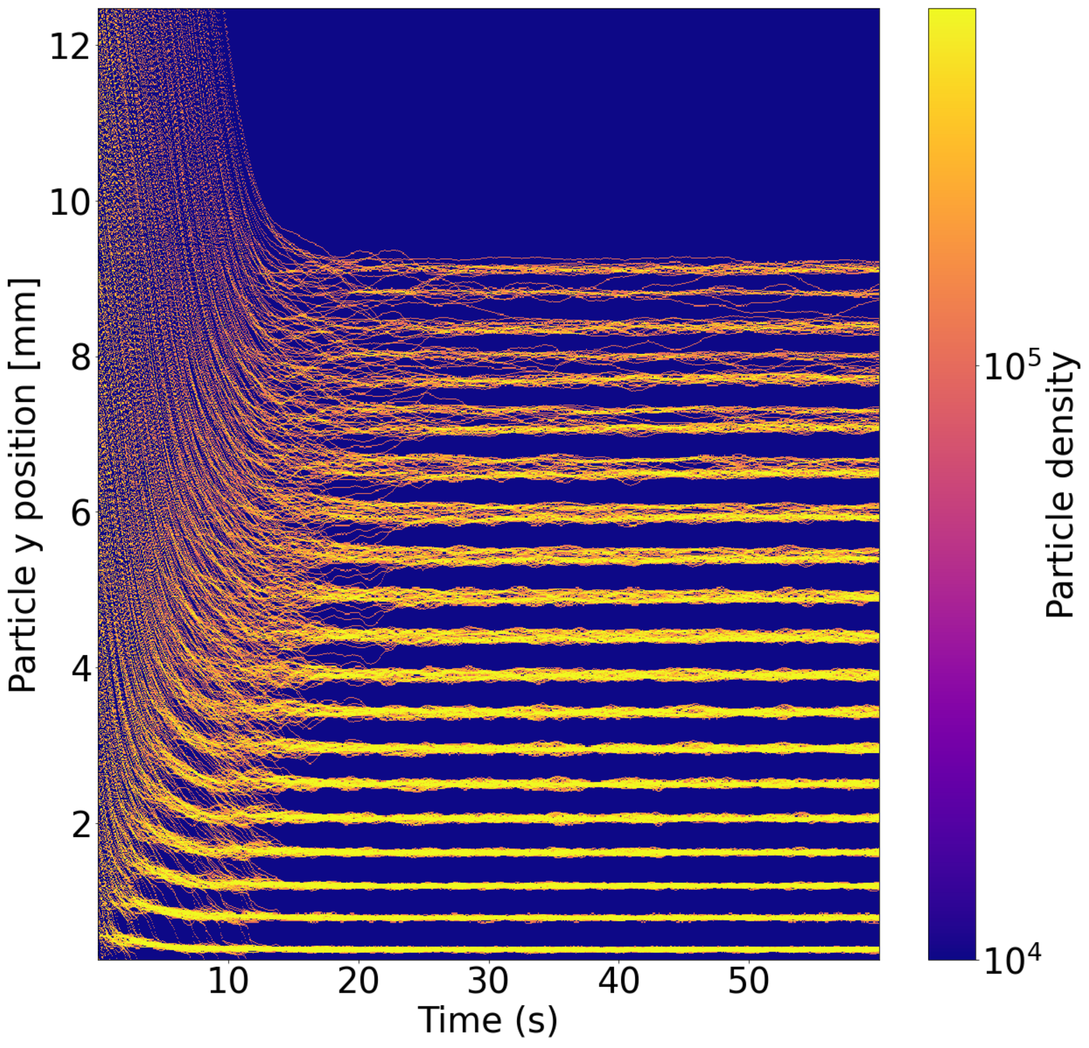


Figure 4.6: Plot of density vs time for the case of a highly coupled system. This plot shows how the particles start with a randomized particle position on the left hand side of the diagram and organize themselves into discrete rows over time. This plot confirms that the particles organize into a relatively constant y position based on their heigh (since the spread of positions along y is relatively small for a single layer compared to the distance between the layers.

Furthermore this method can be applied to the case of magnetized dusty plasmas which have so far had limited experimental determination methods.

4.3.2 Limits of Theories

The accuracy of the measurement for each theory however is dependent upon the coupling in the system.The percent difference between input and expected values in this system is valid for approximately $\Gamma_s < 1$ (Low coupling) and for $\Gamma_s > 100$ (High coupling). In the parameter space where $1 < \Gamma_s < 100$, the system is said to be in a ”liquid-like” state. As many dusty plasmas exist in this regime, it will be important to develop a separate method for calculation of dust particle charge while the dusty plasma system is in this parameter space. This intermediate, liquid-like regime shows both a pseudo-crystalline state (low y position and high coupling coefficient) and a more gaseous state (high y position and low coupling coefficient). This intermediary regime will be discussed further in chapter 6. This region between the two extremes of high and low coupling theories is one which can be investigated further. Hopefully, the investigation of a unifying mixed mode theory can be found in the future. Though this intermediary region may have higher percent error than the high or low coupling modes, it will allow further investigation of these regions and at the very least allow for a comparative analysis method to measure dust charge value.

Chapter 5

Experimental Investigation

5.1 Experimental Approach

As part of future work, an experiment has been designed in order to test these theories discussed in previous chapters. There are a few unique requirements that designate a design which is different that those previously used in experiments. These requirements are as follows:

1. The dust has reached a steady state
2. Particles are confined within our system
3. Magnetic field perpendicular to flow direction
4. External electric field is approximately zero

Due to these requirements, a unique experimental design is needed. The chamber which was chosen is one similar to the ”Zyflex” chamber developed at DLR. This chamber is a cylindrical chamber designed for dusty plasmas in microgravity as it has a great deal of visibility and allows for changes in the future [61]. One benefit of using this chamber are the very large windows on the sides as well as on top of the chamber. Another benefit is the large amount of space and variability of the inside of the system. An example of this chamber is shown in figure 5.1.

For the requirement of steady state, multiple methods have been proposed such as using gravity as a driving force along a tilted axis or using a laser to drive the dust particles. The method which was investigated is that of using a varying potential well to move the particles

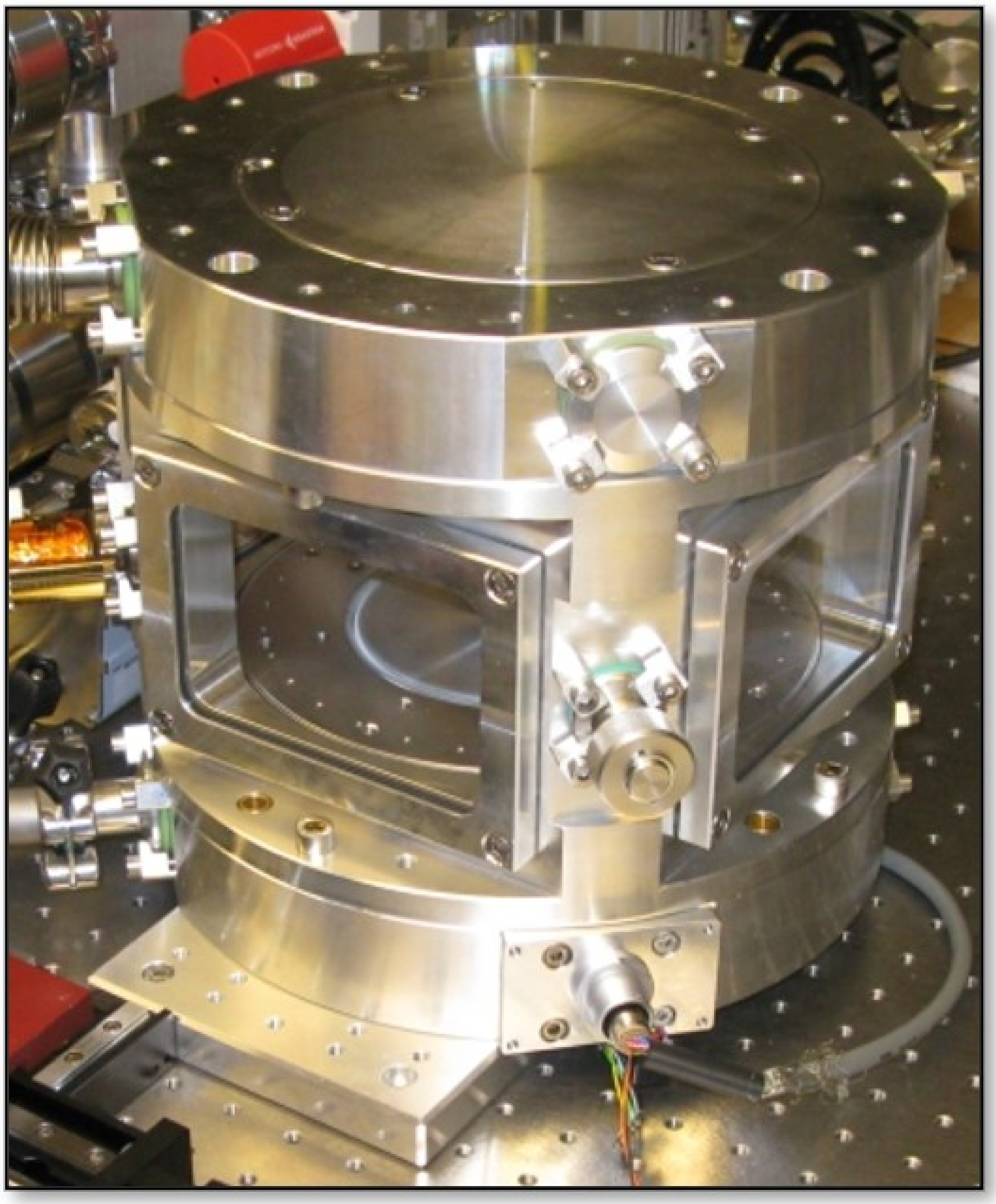


Figure 5.1: Image of the zyflex chamber

along a specified direction. This method has shown to move particles along at a constant velocity as is shown from simulations in the paper by Jiang et al. [62]. In this paper, a set of electrodes is arranged in a grid. Each cell of this grid of electrodes (called ”stripes”) is given a voltage which is offset by a phase from its neighbors. It is further shown that a phase difference between each stripe of $\phi = \pi/3$ yields the most efficient transport. An example of the transport method of a moving potential well shown in figure 5.2.

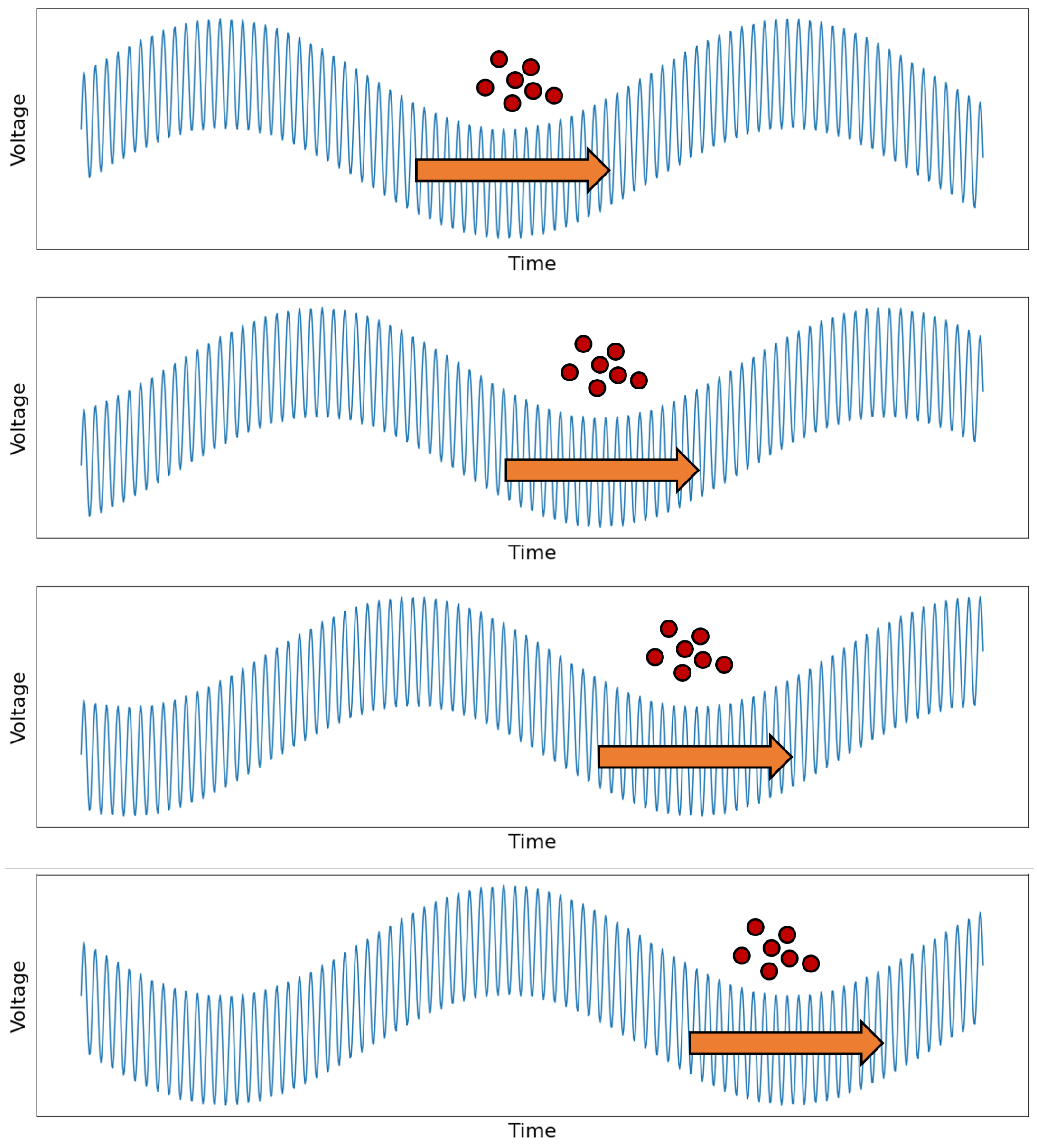


Figure 5.2: This figure shows the moving potential well which can carry the dust particle along a set direction. Each plot represents an adjacent electrode plate.

There are two separate signals necessary to generate this signal. The first of which is a Radio Frequency (RF) signal on the order of 13 MHz which is typical for plasma generation. The second signal is that of the moving potential well, this is an AC signal which is proportional to the distance between adjacent electrodes divided by the drift velocity of the particles. This is shown in figure 5.4.

To generate these signals, a unique electronic setup was designed combining both an RF

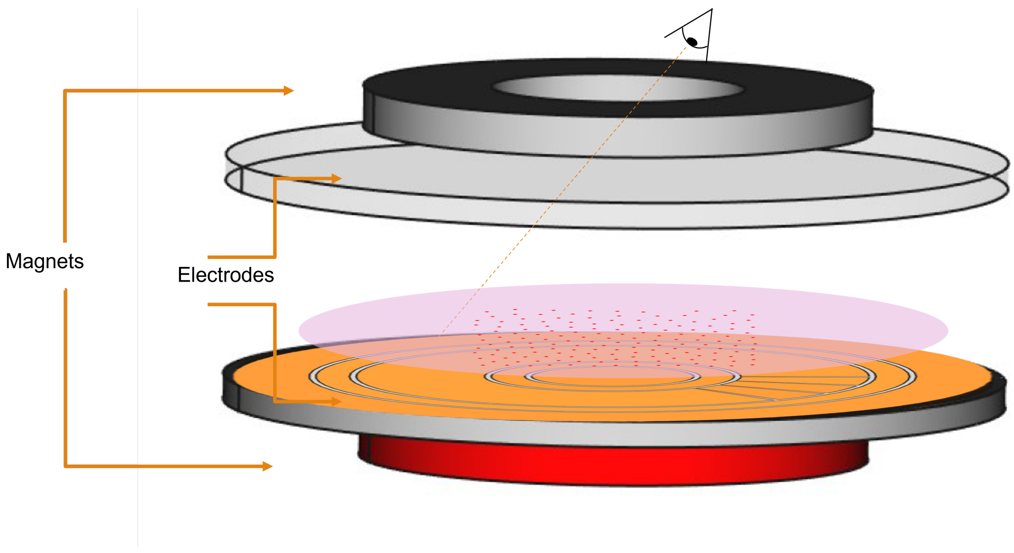


Figure 5.3: Proposed experimental setup

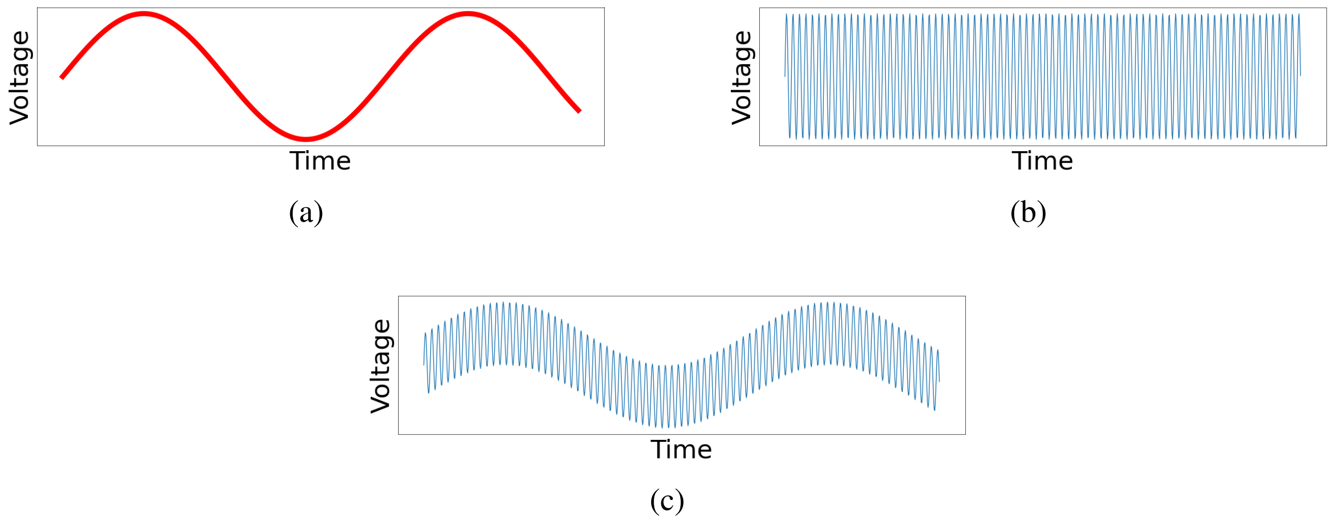


Figure 5.4: In order to create the signal required to move the particles in the experimental system, multiple signals must be combined. (a) Represents the AC input signal generated from a micro-controller which has a frequency proportional to the velocity of the dust particles, (b) represents the RF signal used to create the plasma in the system and finally (c) shows the combination of these two signals

voltage as well as a DC offset. This design is shown in the flow diagram in figure 5.5 and has a special requirement of using a micro-controller to control the signals in the system.

5.2 Electronic Design

For the electronic design, multiple circuits were required. This electronic design has been split into two parts, the high voltage RF circuit and the AC circuit which is controlled by a micro controller. Each of the design of this circuit is shown in figure 5.5.

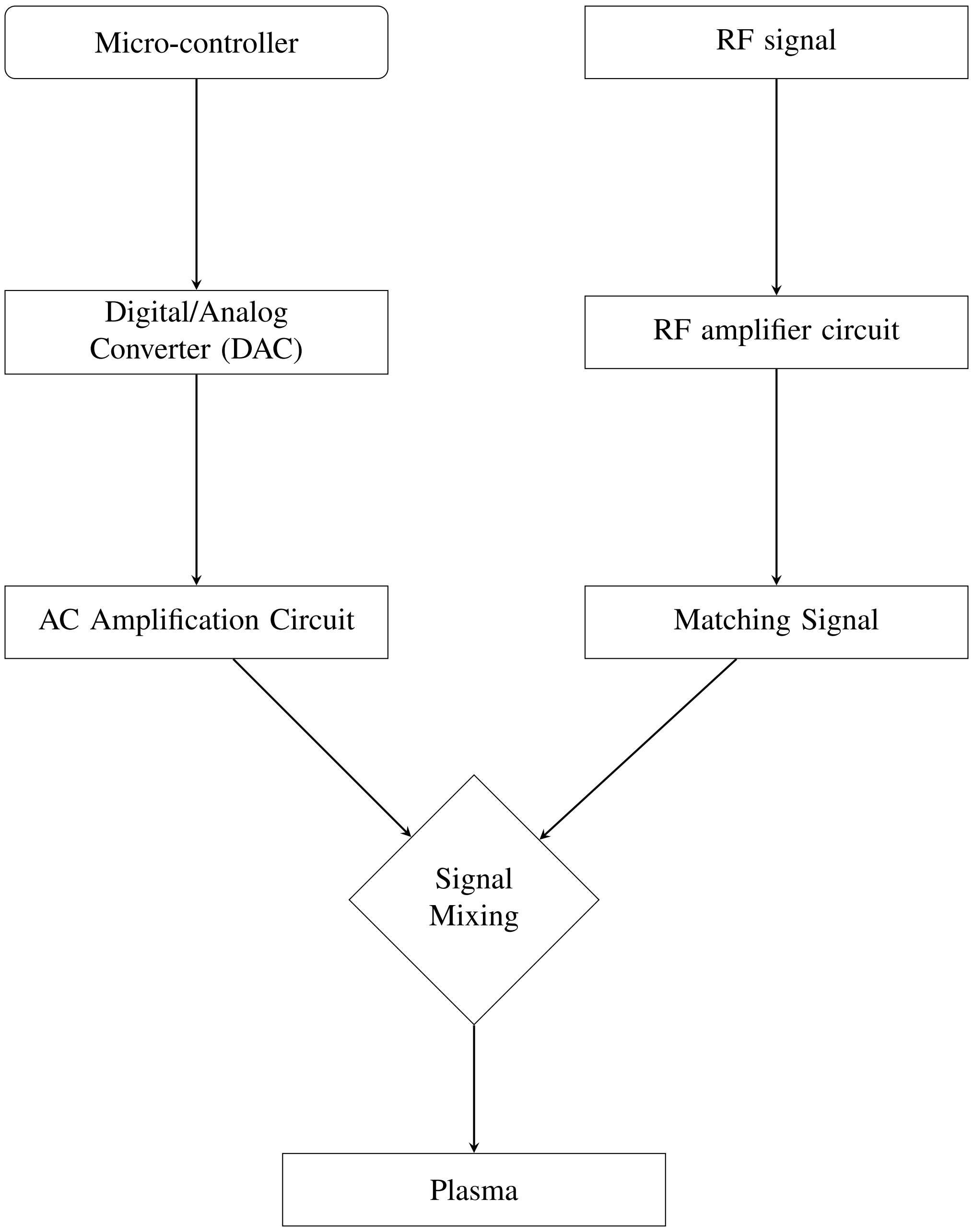


Figure 5.5: Flow diagram for experimental electronics

5.2.1 Radio Frequency Circuit Design

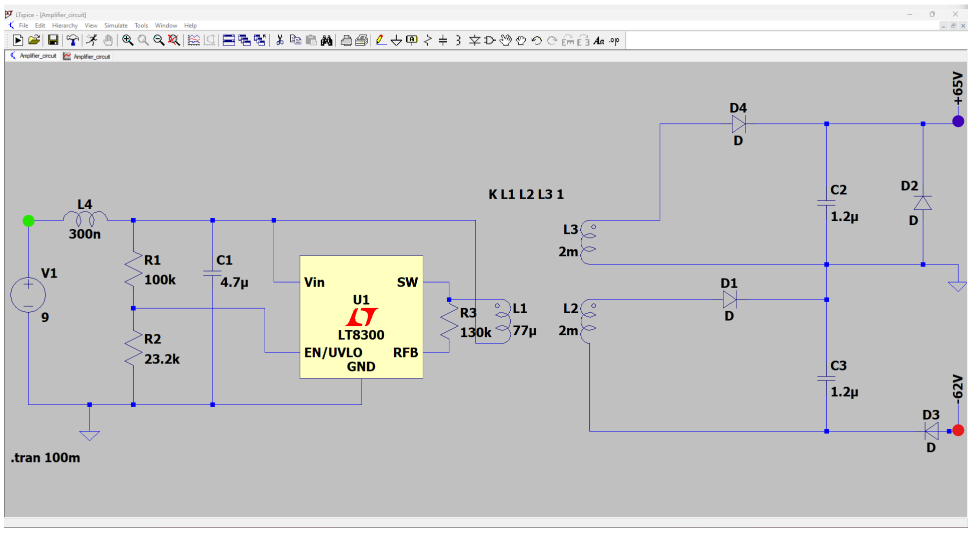
In order to generate a plasma, a high voltage radio frequency (RF) signal must be generated. This signal is designed to have a frequency of 13 MHz with a DC voltage offset of 70 Volts. The purpose of this DC offset is to serve as the walls of the potential well as shown in figure 5.8. The 13 MHz signal was generated by a function generator at 5 volts. This function generator creates a 5 volt signal which is then amplified to $\pm$ 70 volt DC offset. A circuit was created to accomplish this DC offset which is shown in figure 5.6. Using this signal, a plasma can be generated. Next, the alternating current signal needs to be created.

5.2.2 Alternating Current Signal Generation

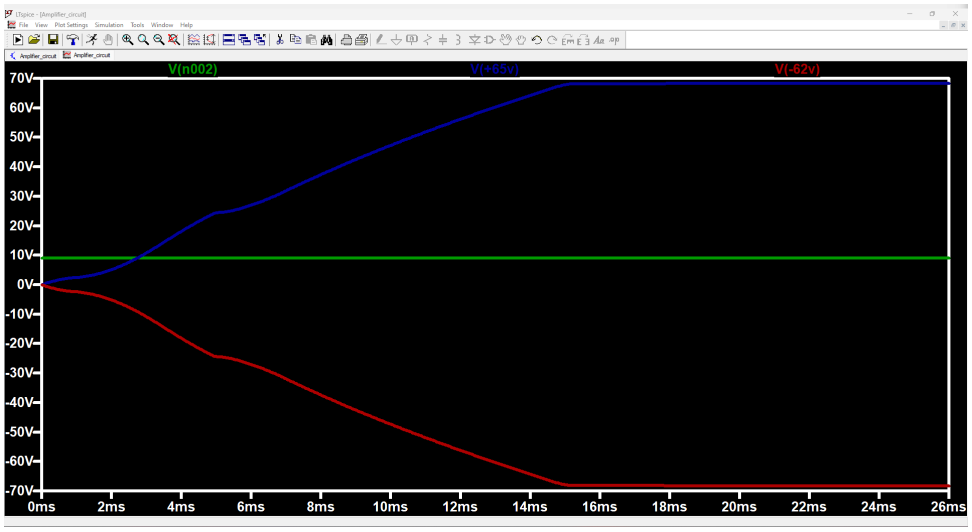
For the AC current, an Arduino Due was used as a micro-controller and code was written in the Arduino programming language, Arduino C, for the purpose of controlling the AC signal in the system. The arduino code is shown in section C.2 of the appendix of this dissertation. The purpose of the micro-controller is to generate a digital signal in a sawtooth pattern at a chosen ratio. This signal was subsequently converted into a sinusoidal analog signal using a digital/analog converter (DAC). This analog signal was then amplified in order to generate the AC current potential well shown in figure 5.4.a. By applying these alternating potential wells to electrode labeled in figure 5.7 as pads P1, P2 and P3, this will create the driving force in our system.

5.3 Electrode Design

The electrode design is similar to that which is shown in the paper by Jiang et al. [62] was chosen to be used. There are two main problems with using an exactly identical design. Firstly, the dimensions would be required would be very large along one dimension in order to allow the moving dust particles to reach steady state or alternatively have the particles move slowly in which case they would be unable to reach steady state. This would require a unique chamber design which therefore would not fit in experiments such as the Magnetized Dusty Plasma eXperiment (MDPX).



(a)



(b)

Figure 5.6: Shown in this figure is the RF amplifier circuit which takes a 9V signal and amplifies it to ± 70V DC offset. Figure (a) shows the circuit to create the voltage amplification. Figure (b) shows the output of the simulation of this circuit. It should be noted here that the colored dots on diagram (a) correspond to the similarly colored plots in figure (b).

Secondly, the loss of particles at the end of their path (i.e., non-circular boundary condition) would require a constant replenishment of dust particles in the system. This would complicate things further as it can be difficult to uniformly disperse dust particles in an experimental system and this non-uniformity might disrupt the steady state of the system. In order to avoid these requirements a circular system was preferable so as to confine the particles (without needing to replenish the dust in the system) as well as allow the system to reach a steady state in terms of both dust charge and dust velocity as discussed in previous chapters.

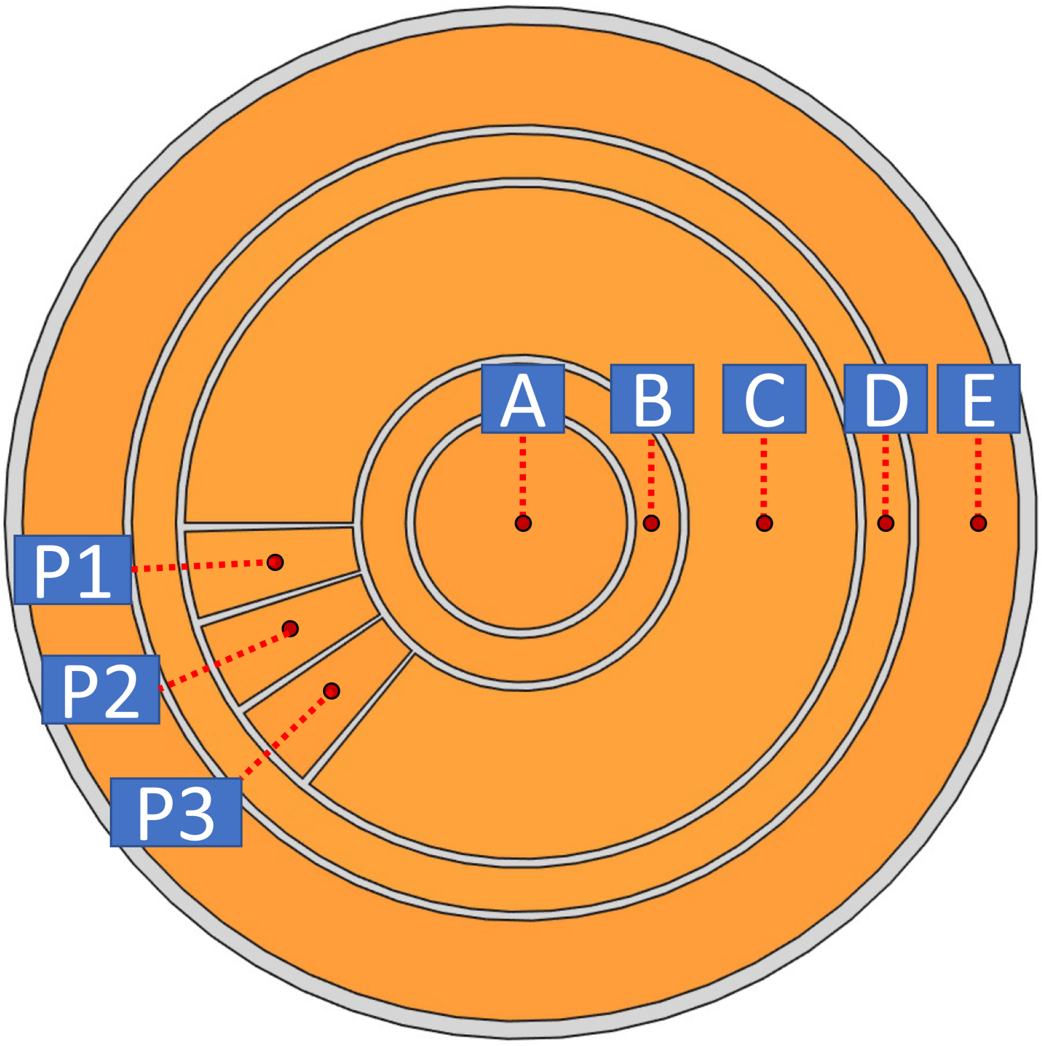


Figure 5.7: Experimental electrode design consisting of 8 individual electrodes controlled separately. There are two separate sets of electrodes, A-E confine the dust particles while P1-P3 create a constant driving force

The electrodes labeled B and D in Figure 5.8 are the confinement rings in this system. These are given a high voltage so as to create a potential well to keep the dust particles within the region of flow (which is all of electrode C as well as P1, P2 and P3). Further, pads A and E are held at a much higher voltage in the case that any dust comes near to falling out of the system. This flow region will allow the dust particles to be confined within our system and remove the requirement of replenishing the number of dust particles as the number of particles will remain constant. As previously stated, the purpose of this is to allow the dust in the system to reach a steady state. Figure 5.9 shows an example of the voltage on plates A through E in this system. These voltages on the electrodes both generate the plasma in the system as well as create a potential well for the particles to be confined within.

This experimental design once built will allow for the testing of the theoretical frame-

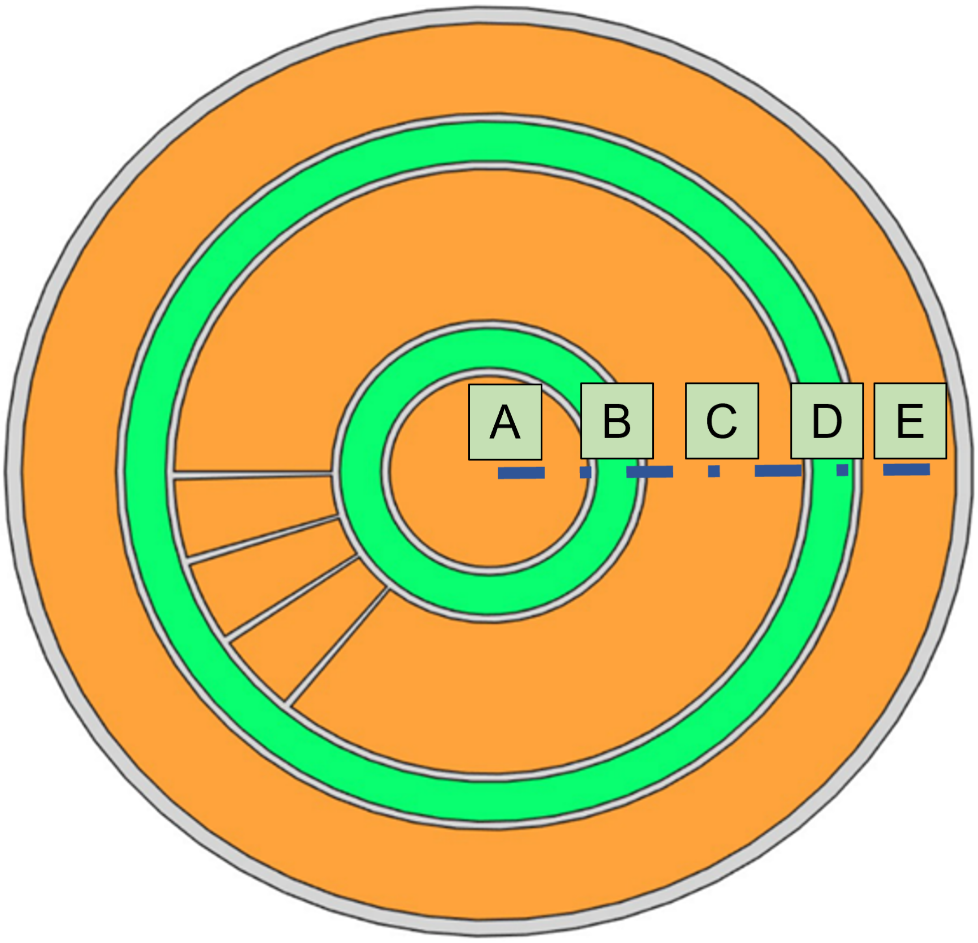


Figure 5.8: Electrode rings labeled B and D and highlighted in green in this image will act as potential walls in our system.

work shown in this dissertation in chapter 2. While much of this experiment has been created and tested, due to the constraints of time as well as the expansion of the theory to include high coupling, this experiment was never fully realized. The framework of this experiment will be very useful for future graduate students and I hope that this experiment is completed and this theory can be further verified.

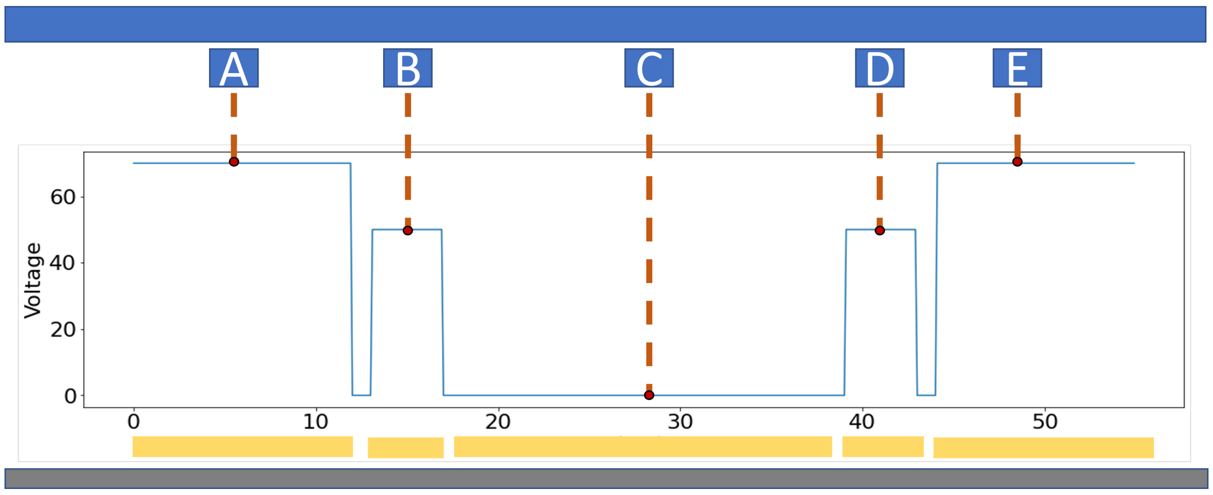


Figure 5.9: A sample electrode configuration for electrodes A through E. This Voltage configuration will allow for a flow region along electrode C. The inclusion of electrodes B and D can allow for further physics to be tested (for example, setting voltage to 0 V on these electrodes will create a region of shear around C)

Chapter 6

Summary and Future Work

6.1 Summary

One of the particular reasons why dusty plasmas have been such an interesting field of study is their behaviour with regard coupling coefficient. The behaviour of dust particles and its similarity to that of various states of matter is a unique way of viewing both states of matter and Coulomb interactions on the macro scale. With the importance of the Coulomb interaction in the study of dusty plasmas the value of dust charge becomes very important. The complication of the dust-plasma system makes finding this value incredibly difficult. The goal of this dissertation from the onset has been to find a new method for accurately determining dust charge in a dusty plasma system. Initially this was intended to be done by comparing molecular dynamic simulations to experiments. This was however expanded to include further analysis of dust charge value determination and its correspondence to the coupling parameter with two separate theories. Utilizing the molecular dynamic simulation to analyze these theories was successful in showing the accuracy of these new methods of analysis.

Chapter 1 began with an introduction to dusty plasmas and the relevance of their charge values. This was accomplished by first showing the Coulomb potential and its variation into the Debye shielded potential in the case of a dusty plasma. The interactions between particles were further expanded upon by the introduction of the coupling parameter and its relation to states of matter and crystallization. The discussion of charge determination techniques which have been previously used in dusty plasmas and their limitations have been split into two categories, theoretical and experimental. Among the theoretical methods, these include charging theories such as ABR and OML theory which are commonly used in the field of dusty plasmas. For the experimental theories, dust charge was shown to be calculated using dust capacitance, vertical oscillations, dust acoustic waves as well as force balance in levitation of particles. One important note is the limitations of these methods with regards to magnetic field as shown by Tsytovich et al. [19].

In Chapter 2, two new theories for charge determination were introduced. Both theories use the assumption of a system similar to that in the case of the classical Hall effect. This system requires a constant flow of particles (i.e., particles with a constant velocity) as well as a perpendicular magnetic field in order to generate a loretz force which acts as a compressional force in the system. The first of the two theories is for the case of a lowly coupled system with an incident magnetic field. Among the assumptions for this system is that for a lowly coupled system, the dust will behave as a fluid which is true when $K_{thermal} \gg U_{inter-particle}$. As a result, fluid equations are used for this theory and dust charge q_d is found in equation 2.13 to be proportional to the dust temperature T_d, density gradient ∇n and average particle velocity along the flow direction u_x.

Next, a charge determination theory was introduced for the case of a highly coupled dusty plasma. These dusty plasmas can be said to be crystalline-like. An interaction force is solved for the case of a shielded Debye potential. This force is utilized in the calculation of the force balance on a particle. This is unique as these dust pseudo-crystals are asymmetric as a result of the compressional force. An equation for the dust charge value in the case of a highly coupled dusty plasma is shown in equation 2.26.

Chapter 3 introduces the simulation which was created to run these experiments. This chapter begins by introducing the integration method which is used in this simulation and it's benefits. Furthermore, equation 3.4 shows the method for calculation of time step compared to the the scale of either the energy from the driving force in the system or the thermal energy in the system. This is chosen based on which is the smaller time step (i.e., whether driving force or stochastic force is stronger). The molecular dynamic simulation is validated with a comparison of the analysis results to the assumptions made in the system. Assumptions such as the system being time-independent, isothermal, and having a perpendicular density gradient are all verified here.

Chapter 4 shows the results and analysis of the data in the system. For the case of a lowly coupled dusty plasma, the quantity is calculated using equation 2.13. This is done by determining the dust temperature, density and average velocity from positional and velocity data. Next, a calculation for the case of a highly coupled dusty plasma is conducted by using equation 2.26 by using positional and velocity data from the simulation. A sample calculation is provided for a single particle and further calculations are shown for the ensemble of the dust system.

In Chapter 5, experimental design is introduced to test the two previous theories. First the experimental setup is discussed with the proposed use of the Zyflex chamber. The electrode is next discussed with its unique method for control of dust particles. Also shown here is the design of the electrode to incorporate a circular boundary condition. Lastly, the electronic control of the system is introduced which includes both electronic design for controlling multiple unique signals as well as microcontroller design.

6.2 Future Work

In the investigation of this work, many interesting areas of study have been discovered which may increase the accuracy and validity of these methods as well as giving more information about dusty plasmas overall.

6.2.1 Experimental Testing

Chapter 5 discussed an experimental system which can analyze the results of these theories. This system can be used therefore to test both the high and low coupling theories simply by modifying the system parameters to achieve the coupling necessary. This experiment has been entirely designed from the chamber to the electrode to the electronics. This design allows for further work on studying flowing dusty plasma systems in the future provided this system is built. Preliminary experimental setup has been done at Auburn University and will be followed up in future work. Should this experiment be completed it will allow for further analysis of the theoretical method by providing error data in a real system.

One major benefit of this experiment being at Auburn University will be the availability of the magnetized dusty plasma experiment (MDPX). The experiment which has been designed and described in chapter 5 has been created in a way which is compatible with MDPX. This will allow study of both the low and high coupling cases. MDPX (shown in figure 6.1) is an multi-user device which consists of a superconducting magnet as well as a plasma chamber. This magnet has a range of up to 4 T and would be able to test both the low and high coupling charge determination theories.

Further work has been proposed which will analyze data from experiments of similar

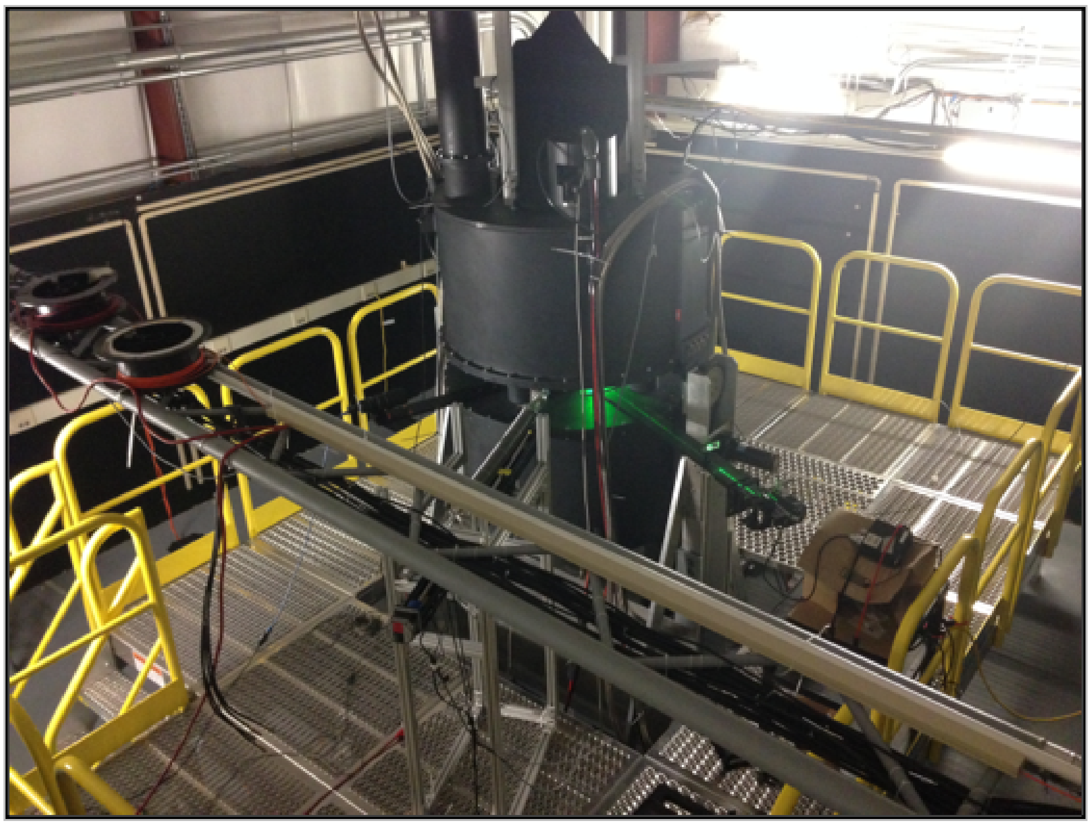


Figure 6.1: Image showing Magnetized Dusty Plasma eXperiment (MDPX) user device at Auburn University (courtesy of Uwe Konopka)

systems to what has been seen in simulations. Since compression by gravitational force occurs in all ground based dusty plasma systems, this compressed crystalline-like structure has been previously observed and can be seen evaluated using this new method. In order to study this data, positional and velocity data can be taken from images of experimental research and analyzed using previously existing particle analysis software such as ImageJ or COPLA. Analyzing this data and using it to find dust charge will help expand upon the validity of these theories. Figure 6.2 shows a dusty plasma system in which compression and density gradient can be seen.

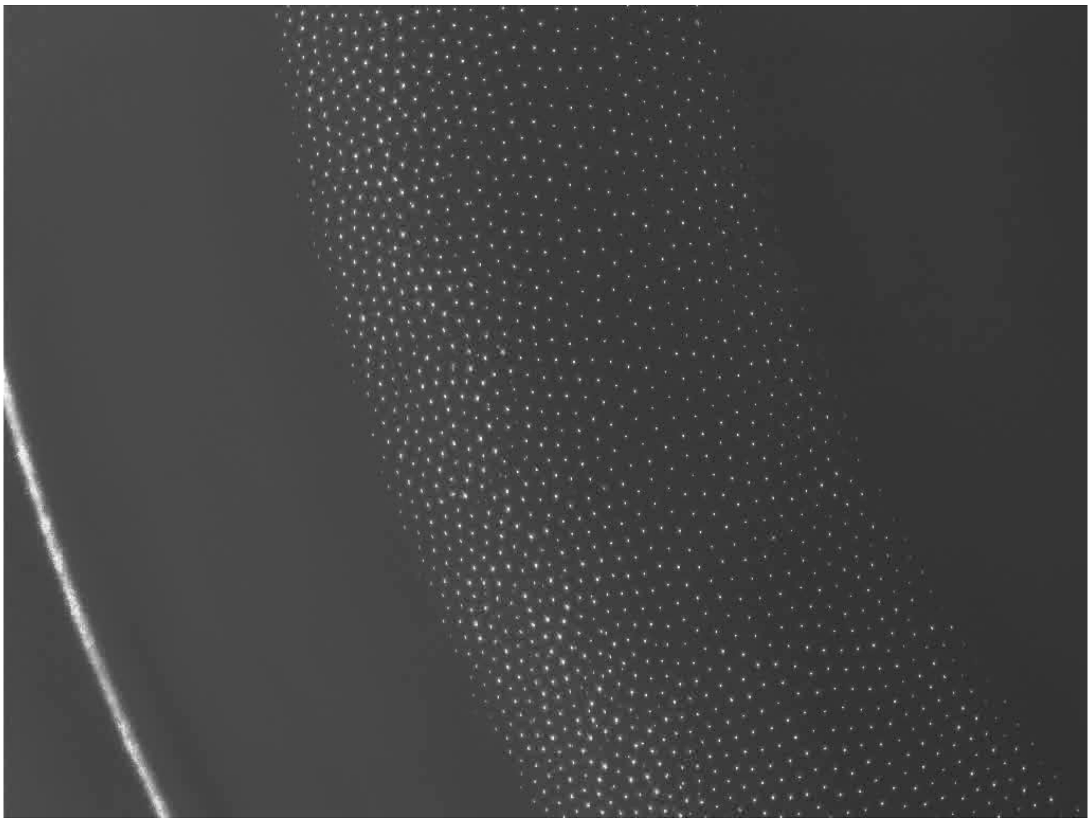
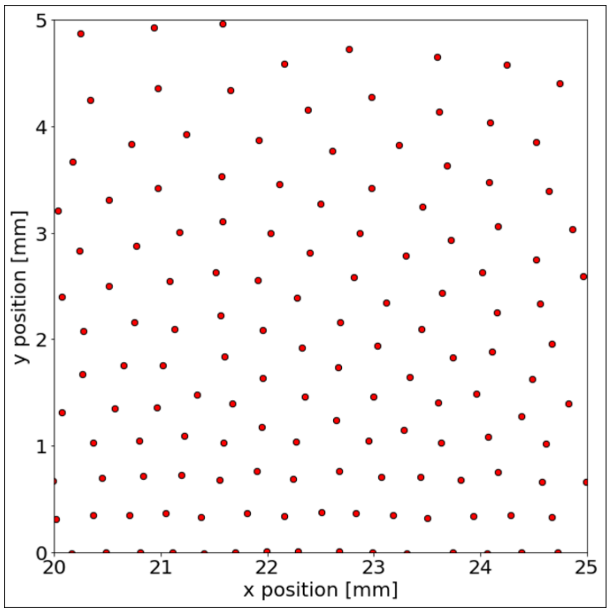


Figure 6.2: Image of a magnetized dusty plasma showing density gradient in Zyflex chamber. This shows at least two distinctive regimes, the first being on the rightmost side where the system can be said to be crystallized, on the left side of the system, the particles are less organized and therefore more liquid-like in nature. on the crystalline side, the distance between layers can be seen to be increasing (image courtesy of Uwe Konopka)

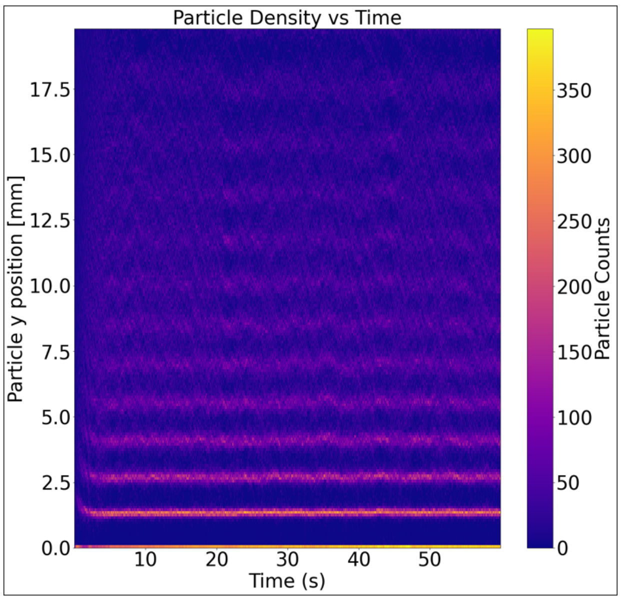
6.2.2 Mixed Coupling Theory and Phase Transitions

An interesting result of the high and low coupling theories is the existence of phase transitions as a function of position. Since the phase a dusty plasma exists in is dependent upon the coupling coefficient, the relevance of a non-constant coupling coefficient creates some interesting effects as shown in figure 6.3.

Figure 6.3 shows an example case where the lower portion of the simulated system has a high coupling coefficient where crystallization occurs. This can be seen in the existence of



(a)



(b)

Figure 6.3: This figure shows an example of (a) Corresponding simulation frame showing both highly coupled crystal-like and lowly coupled liquid states (b) Density vs Time showing both crystal-like and liquid states

hexagon-like structure along the bottom few rows. The upper portion of this figure shows a more fluid state with less uniform structure. This kind of system may occur when coupling coefficient $\Gamma \sim 1$ which means the energy of the system is balanced between thermal and inter-particle potential energy. This is relevant for many systems and falls in the range where the accuracy of each of the two respective theories drops. A theory for this boundary range would be helpful to increase upon the accuracy of the theories of compressed dusty plasma systems.

6.2.3 Density analysis

The density of the system is non-uniform and there is a reliance upon the assumption of an appropriate fit for the density of the system. As shown in figure 6.4 the assumption of an exponential fits reasonably well for for this system. It can be noted however while the exponential assumption of fit matches what is expected from a compressed gas similar to that found in air in the atmosphere [60], there is slight deviation from this fit. By plotting the residuals of this data, an interesting structure is found as shown in figure 6.5. This structure appears in the case of a lowly coupled system. With further analysis of this, information about

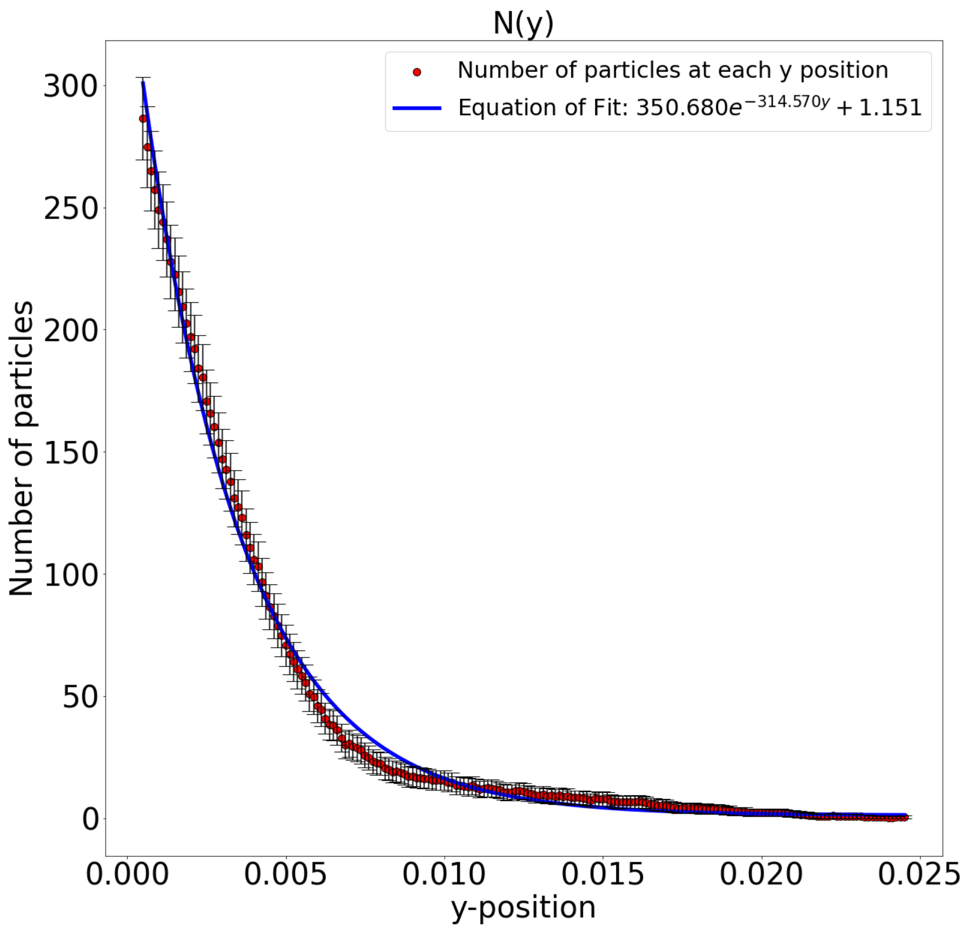


Figure 6.4: Plot showing the number density of particle as well as the exponential fit to the data

the system and possibly the charge may be extrapolated. This may be useful in either error correction (in the case of finding a better fit) or couple possibly show more information about the system which may be held in the structure of these dust particles.

6.2.4 Time Scale Analysis

The time scales involved in these simulations come in three separate varieties. The first is that of the charging time scale which is on the order of approximately 10^{-6} seconds as shown by equation 3.5. This is minimal compared to the simulation times in the system. Next is that of acceleration time which is a result of the particles coming to a drift velocity. The last and most interesting is the time scale of structural organization or sedimentation. This is relevant for the case of a highly coupled system as shown in figure 6.6. This varying sedimentation time is an

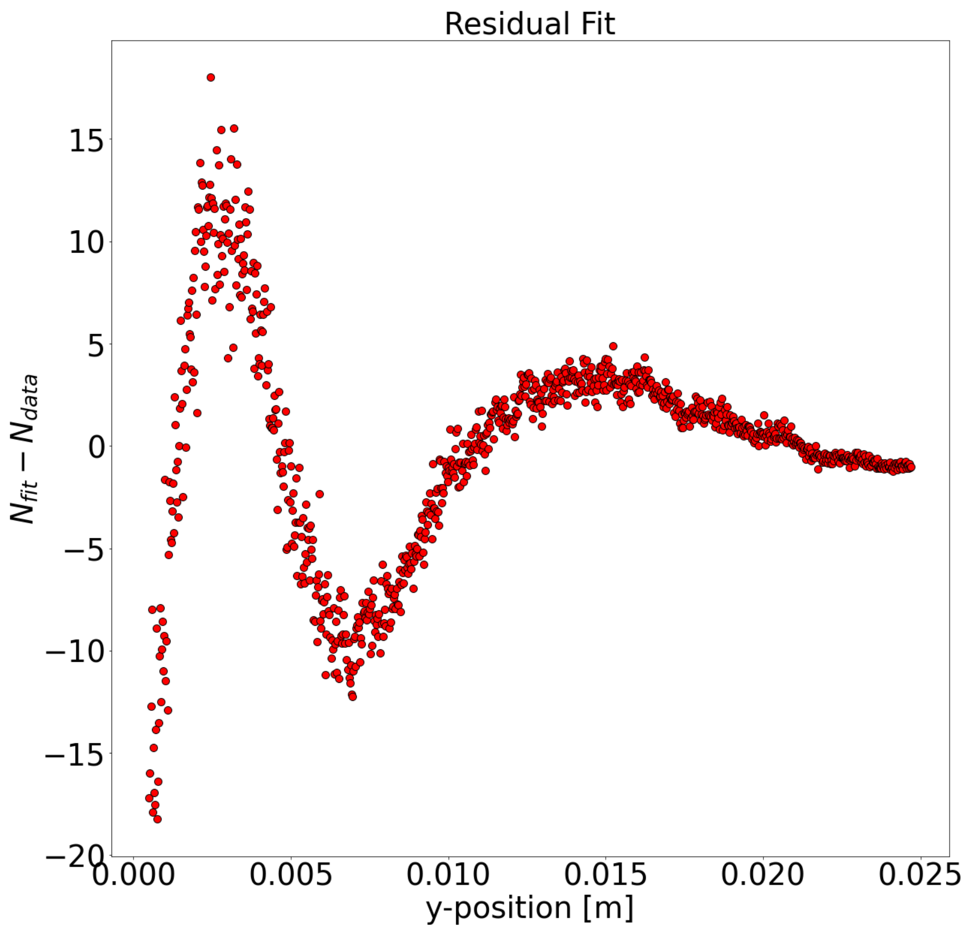


Figure 6.5: Plot showing the number density of particle as well as the exponential fit to the data

interesting result. Analysis of this may result in further information regarding the dust particles themselves such as another method of determining dust charge value. Furthermore, the value of this information increases as dust parameters change. This is due to the fact that for, for example, a different dust particle size the charging time on various time scales will change.

6.3 Conclusion

There are a few major benefits to this research: firstly, the molecular dynamics code which was developed for the purpose of studying this system is widely applicable. This is not only the case for dusty plasmas but for any system which might require a molecular dynamics code. When this simulation was developed it was developed for the purposes of being usable by not only myself but users in general. It is my hope that once this work is completed, others

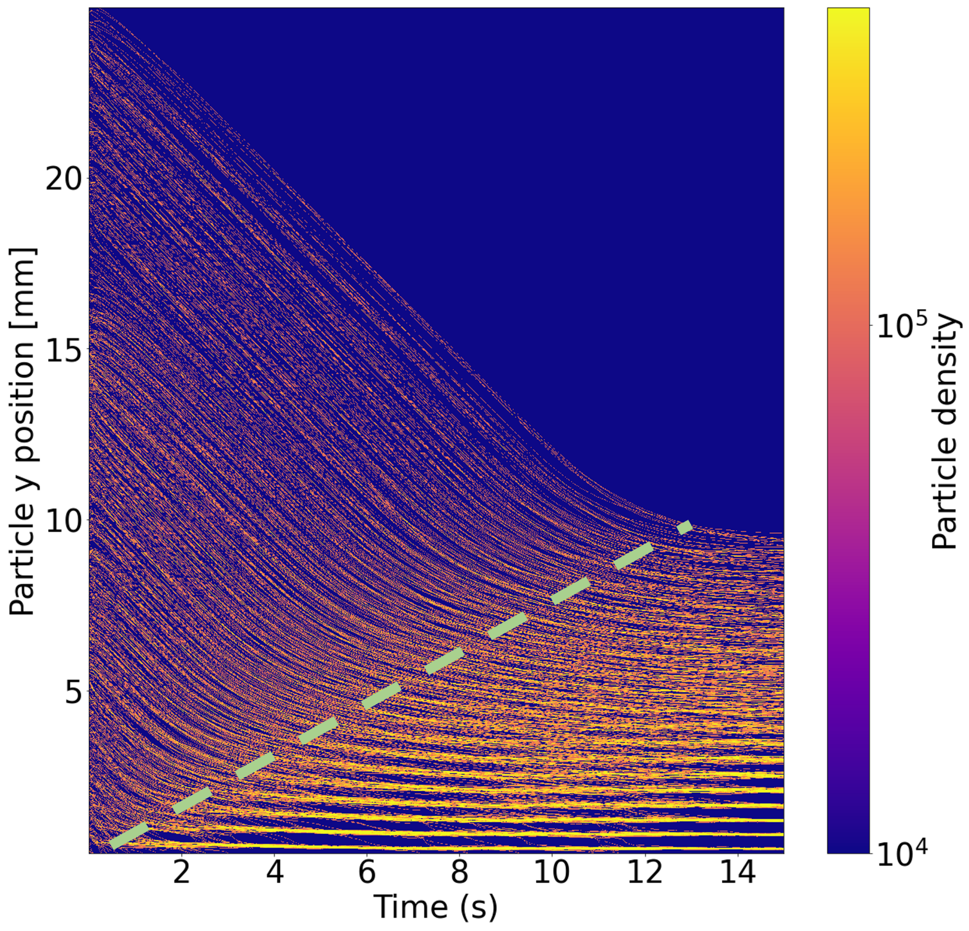


Figure 6.6: Density as a function of time for a highly coupled system. The dotted line in this plot shows an approximation of the time at which the system can be said to be steady state. This varies with the y position in the system and therefore will also vary with the coupling in the system.

will be able to utilize this code and adapt it to their own needs.

The second major benefit is that of the applicability of the research itself. This work introduces two new methods (High and low coupling charge calculation theories) which can be used for magnetized dusty plasma systems to calculate dust particle charge to a high degree of precision as shown in chapter 4. The added benefit is that this requires no other equipment experimentally than image processing techniques and a camera (Neither of which will disturb the plasma itself). As well, the high coupling charge calculation theory can be used for compressed dusty plasma systems without magnetic field. These results are enough to show the value of this research project.

Lastly, in this work many interesting branches developed which were shown in the previous ”Future work” section. This will allow this work to be valuable for future researchers as a resource to develop new ideas and to perfect this method beyond my time and capabilities. I greatly look forward to seeing the continuation of my work and how it can be applied to research both by myself and others in the future.

Bibliography

[1] I Langmuir. “Oscillations in ionized gases”. en. In: *Proc. Natl. Acad. Sci. U. S. A.* 14.8 (Aug. 1928), pp. 627–637.

[2] Patrick T. Newell and Ching-I. Meng. “Mapping the dayside ionosphere to the magnetosphere according to particle precipitation characteristics”. In: *Geophysical Research Letters* 19.6 (Mar. 1992), pp. 609–612. DOI: 10.1029/92gl00404. URL: https://doi.org/10.1029/92gl00404.

[3] Alexander Piel. *Plasma physics*. en. 2010th ed. Berlin, Germany: Springer, June 2010.

[4] C Aragón and J A Aguilera. “Characterization of laser induced plasmas by optical emission spectroscopy: A review of experiments and methods”. en. In: *Spectrochim. Acta Part B At. Spectrosc.* 63.9 (Sept. 2008), pp. 893–916.

[5] Robert Merlino. “Dusty plasmas: from Saturn’s rings to semiconductor processing devices”. In: *Advances in Physics: X* 6.1 (2021). ISSN: 23746149. DOI: 10.1080/23746149.2021.1873859. URL: https://doi.org/10.1080/23746149.2021.1873859.

[6] Hannes Alfvén. “The Plasma Universe”. In: *Phys. Today* 39.9 (Sept. 1986), pp. 22–27.

[7] M Coppins. “Electrostatic breakup in a misty plasma”. en. In: *Phys. Rev. Lett.* 104.6 (Feb. 2010), p. 065003.

[8] Gary S. Selwyn, John E. Heidenreich, and Kurt L. Haller. “Particle trapping phenomena in radio frequency plasmas”. In: *Applied Physics Letters* 57.18 (Oct. 1990), pp. 1876–1878. DOI: 10.1063/1.104021. URL: https://doi.org/10.1063/1.104021.

[9] M S Sodha, S K Mishra, and S Misra. "Kinetics of Complex Plasmas Having Dust of Different Materials With Corresponding Size Distribution". In: *IEEE Transactions on Plasma Science* 39.4 (Apr. 2011), pp. 1141–1149. DOI: 10.1109/tps.2011.2112673. URL: https://doi.org/10.1109/tps.2011.2112673.

[10] Harry L. F. Houpis and Elden C. Whipple. "Electrostatic charge on a dust size distribution in a plasma". In: *Journal of Geophysical Research* 92 (A11 1987), p. 12057. ISSN: 0148-0227. DOI: 10.1029/ja092ia11p12057.

[11] Ove Havnes, Torsten K. Aanesen, and Frank Melandsø. "On dust charges and plasma potentials in a dusty plasma with dust size distribution". In: *Journal of Geophysical Research* 95 (A5 1990), p. 6581. ISSN: 0148-0227. DOI: 10.1029/ja095ia05p06581.

[12] U. Konopka, L. Ratke, and H. M. Thomas. "Central collisions of charged dust particles in a plasma". In: *Physical Review Letters* 79.7 (1997), pp. 1269–1272. ISSN: 10797114. DOI: 10.1103/PhysRevLett.79.1269.

[13] Kil-Byoung Chai and Paul M Bellan. "Spontaneous formation of nonspherical water ice grains in a plasma environment". en. In: *Geophys. Res. Lett.* 40.23 (Dec. 2013), pp. 6258–6263.

[14] M. Horányi et al. "Dusty plasma effects in Saturn's magnetosphere". In: *Reviews of Geophysics* 42.4 (Dec. 2004). DOI: 10.1029/2004rg000151. URL: https://doi.org/10.1029/2004rg000151.

[15] Akio Sanpei et al. "Levitation of microorganisms in the sheath of an RF plasma". In: *IEEE Trans. Plasma Sci. IEEE Nucl. Plasma Sci. Soc.* 46.4 (Apr. 2018), pp. 718–722.

[16] S K Kodanova et al. "The effect of magnetic field on dust dynamic in the edge fusion plasma". In: *IEEE Trans. Plasma Sci. IEEE Nucl. Plasma Sci. Soc.* 46.4 (Apr. 2018), pp. 832–834.

[17] Paul M Bellan. *Fundamentals of plasma physics*. Cambridge, England: Cambridge University Press, July 2008.

[18] H. Thomas et al. “Plasma crystal: Coulomb crystallization in a dusty plasma”. In: *Physical Review Letters* 73.5 (1994), pp. 652–655. ISSN: 00319007. DOI: 10.1103/PhysRevLett.73.652.

[19] V. N. Tsytovich, N. Sato, and G. E. Morfill. “Note on the charging and spinning of dust particles in complex plasmas in a strong magnetic field”. In: *New Journal of Physics* 5 (2003). ISSN: 13672630. DOI: 10.1088/1367-2630/5/1/343.

[20] H. M. Mott-Smith and Irving Langmuir. “The theory of collectors in gaseous discharges”. In: *Physical Review* 28.4 (1926), pp. 727–763. ISSN: 0031899X. DOI: 10.1103/PhysRev.28.727.

[21] Peter Debye and Erich Hückel. “The theory of electrolytes. I. Freezing point depression and related phenomena [Zur Theorie der Elektrolyte. I. Gefrierpunktserniedrigung und verwandte Erscheinungen]”. de. In: 24 (1923), pp. 185–206.

[22] J E Allen, R L F Boyd, and P Reynolds. “The Collection of Positive Ions by a Probe Immersed in a Plasma”. In: *Proceedings of the Physical Society. Section B* 70.3 (Mar. 1957), pp. 297–304. DOI: 10.1088/0370-1301/70/3/303. URL: https://doi.org/10.1088/0370-1301/70/3/303.

[23] J. E. Allen. “Probe theory - the orbital motion approach”. In: *Physica Scripta* 45.5 (1992), pp. 497–503. ISSN: 14024896. DOI: 10.1088/0031-8949/45/5/013.

[24] Francis F. Chen. “Langmuir probes in RF plasma: Surprising validity of OML theory”. In: *Plasma Sources Science and Technology* 18.3 (2009). ISSN: 09630252. DOI: 10.1088/0963-0252/18/3/035012.

[25] Ain A. Sonin. “Free-molecule Langmuir probe and its use in flow-field studies”. In: *AIAA Journal* 4.9 (1966), pp. 1588–1596. ISSN: 00011452. DOI: 10.2514/3.3740.

[26] P. Bryant, A. Dyson, and J. E. Allen. “Langmuir probe measurements of weakly collisional electropositive RF discharge plasmas”. In: *Journal of Physics D: Applied Physics* 34.10 (2001), pp. 1491–1498. ISSN: 00223727. DOI: 10.1088/0022-3727/34/10/309.

[27] A V Zobnin et al. “Dust – Acoustic Instability in an Inductive Gas-Discharge Plasma”. In: 95.3 (2002), pp. 429–439.

[28] V. N. Tsytovich and J. Koller. “Collective dust-dust attraction in strong magnetic field”. In: *Contributions to Plasma Physics* 44.4 (2004), pp. 317–326. ISSN: 08631042. DOI: 10.1002/ctpp.200410048.

[29] S K Kodanova et al. “Dust particle evolution in the divertor plasma”. In: *IEEE Trans. Plasma Sci. IEEE Nucl. Plasma Sci. Soc.* 44.4 (Apr. 2016), pp. 525–527.

[30] J. Goree. “Charging of particles in a plasma”. In: *Plasma Sources Science and Technology* 3.3 (1994), pp. 400–406. ISSN: 09630252. DOI: 10.1088/0963-0252/3/3/025.

[31] C K Goertz. “Dusty plasmas in the solar system”. en. In: *Rev. Geophys.* 27.2 (1989), p. 271.

[32] E. C. Whipple. “Potentials of surfaces in space”. In: *Reports on Progress in Physics* 44.11 (1981), pp. 1197–1250. ISSN: 00344885. DOI: 10.1088/0034-4885/44/11/002.

[33] Chunshi Cui and J. Goree. “Fluctuations of the Charge on a Dust Grain in a Plasma”. In: *IEEE Transactions on Plasma Science* 22.2 (1994), pp. 151–158. ISSN: 19399375. DOI: 10.1109/27.279018.

[34] Themis Matsoukas and Marc Russell. “Particle charging in low-pressure plasmas”. In: *Journal of Applied Physics* 77.9 (1995), pp. 4285–4292. ISSN: 00218979. DOI: 10.1063/1.359451.

[35] V E Fortov et al. “Micron-sized particle-charge measurements in an inductive rf gas-discharge plasma using gravity-driven probe grains”. en. In: *Phys. Rev. E Stat. Nonlin. Soft Matter Phys.* 70.4 Pt 2 (Oct. 2004), p. 046415.

[36] A Barkan, N D’Angelo, and R L Merlino. “Charging of dust grains in a plasma”. en. In: *Phys. Rev. Lett.* 73.23 (Dec. 1994), pp. 3093–3096.

[37] Wenjun Xu, Nicola D'Angelo, and Robert L Merlino. "Dusty plasmas: The effect of closely packed grains". en. In: *J. Geophys. Res.* 98.A5 (May 1993), pp. 7843–7847.

[38] U. Konopka, G. E. Morfill, and L. Ratke. "Measurement of the interaction potential of microspheres in the sheath of a rf discharge". In: *Physical Review Letters* 84.5 (2000), pp. 891–894. ISSN: 10797114. DOI: 10.1103/PhysRevLett.84.891.

[39] A. Piel et al. "Waves and oscillations in plasma crystals". In: *Journal of Physics B: Atomic, Molecular and Optical Physics* 36.3 (2003), pp. 533–543. ISSN: 09534075. DOI: 10.1088/0953-4075/36/3/311.

[40] B Lynch. "Microparticle Dynamics in Strongly Magnetized Low Temperature Plasmas". PhD thesis. Auburn University, 2017.

[41] C Castaldo, U de Angelis, and V N Tsytovich. "Screening and attraction of dust particles in plasmas". en. In: *Phys. Rev. Lett.* 96.7 (Feb. 2006), p. 075004.

[42] E Thomas Jr. "Dust clouds in dc-generated dusty plasmas: Transport, waves, and three-dimensional effects". en. In: *Contrib. plasma phys.* 49.4-5 (June 2009), pp. 316–345.

[43] G. S. Dragan and V. V. Kutarov. "Correlation function of the coupling parameter in dusty plasmas". In: *AIP Conference Proceedings* 1925.January (2018). ISSN: 15517616. DOI: 10.1063/1.5020415.

[44] A. Melzer, T. Trottenberg, and A. Piel. "Experimental determination of the charge on dust particles forming Coulomb lattices". In: *Physics Letters A* 191.3-4 (1994), pp. 301–308. ISSN: 03759601. DOI: 10.1016/0375-9601(94)90144-9.

[45] J H Chu. *Direct Observation of Coulomb Crystals and Liquids in Strongly Coupled rf Dusty Plasmas*. 1994, p. 25.

[46] E. Wigner. "Effects of the electron interaction on the energy levels of electrons in metals". en. In: *Transactions of the Faraday Society* 34 (1938), pp. 678–685.

[47] H Ikezi. "Coulomb solid of small particles in plasmas". In: *Phys. Fluids* 29.6 (1986), p. 1764.

[48] Paul S. Epstein. "On the resistance experienced by spheres in their motion through gases". In: *Physical Review* 23.6 (1924), pp. 710–733. ISSN: 0031899X. DOI: 10.1103/PhysRev.23.710.

[49] Bin Liu et al. "Radiation pressure and gas drag forces on a melamine-formaldehyde microsphere in a dusty plasma". In: *Physics of Plasmas* 10.1 (2003), pp. 9–20. ISSN: 1070664X. DOI: 10.1063/1.1526701.

[50] D.N. Petsev, V.M. Starov, and I.B. Ivanov. "Concentrated dispersions of charged colloidal particles: Sedimentation, ultrafiltration and diffusion". In: *Colloids and Surfaces A: Physicochemical and Engineering Aspects* 81 (Dec. 1993), pp. 65–81. DOI: 10.1016/0927-7757(93)80235-7. URL: https://doi.org/10.1016/0927-7757(93)80235-7.

[51] Albert P Philipse and Gijsberta H Koenderink. "Sedimentation–diffusion profiles and layered sedimentation of charged colloids at low ionic strength". In: *Advances in Colloid and Interface Science* 100-102 (Feb. 2003), pp. 613–639. DOI: 10.1016/s0001-8686(02)00078-7. URL: https://doi.org/10.1016/s0001-8686(02)00078-7.

[52] Benedict Leimkuhler and Charles Matthews. "Rational construction of stochastic numerical methods for molecular sampling". In: *Applied Mathematics Research eXpress* 2013.1 (2013), pp. 34–56. ISSN: 16871200. DOI: 10.1093/amrx/abs010. arXiv: 1203.5428.

[53] Benedict Leimkuhler and Charles Matthews. "Efficient molecular dynamics using geodesic integration and solvent-solute splitting". In: *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* 472.2189 (2016). ISSN: 14712946. DOI: 10.1098/rspa.2016.0138.

[54] Nawaf Bou-Rabee. "Time integrators for molecular dynamics". In: *Entropy* 16.1 (2014), pp. 138–162. ISSN: 10994300. DOI: 10.3390/e16010138.

[55] Vikram Suresh and Ranganathan Gopalakrishnan. “Tutorial: Langevin Dynamics methods for aerosol particle trajectory simulations and collision rate constant modeling”. In: *Journal of Aerosol Science* 155.December 2020 (2021), p. 105746. ISSN: 18791964. DOI: `10.1016/j.jaerosci.2021.105746`. URL: `https://doi.org/10.1016/j.jaerosci.2021.105746`.

[56] Edward Thomas et al. “New developments in particle image velocimetry (PIV) for the study of complex plasmas”. In: Garmisch-Partenkirchen (Germany): AIP, 2011.

[57] Edward Thomas Jr and Michael Watson. “First experiments in the Dusty Plasma Experiment device”. In: *Phys. Plasmas* 6.10 (Oct. 1999), pp. 4111–4117.

[58] E Thomas and R L Merlino. “Dust particle motion in the vicinity of dust acoustic waves”. In: *IEEE Trans. Plasma Sci. IEEE Nucl. Plasma Sci. Soc.* 29.2 (Apr. 2001), pp. 152–157.

[59] Brian Lynch, Uwe Konopka, and Edward Thomas. “Real-time particle tracking in complex plasmas”. In: *IEEE Trans. Plasma Sci. IEEE Nucl. Plasma Sci. Soc.* 44.4 (Apr. 2016), pp. 553–557.

[60] *U.S. Standard Atmosphere*. NASA, 1976.

[61] C. A. Knapek et al. ““Zyflex”: Next generation plasma chamber for complex plasma research in space”. In: *Review of Scientific Instruments* 92.10 (Oct. 2021), p. 103505. DOI: `10.1063/5.0062165`. URL: `https://doi.org/10.1063/5.0062165`.

[62] Ke Jiang et al. “Controlled particle transport in a plasma chamber with striped electrode”. In: *Physics of Plasmas* 16.12 (Dec. 2009), p. 123702. DOI: `10.1063/1.3273074`. URL: `https://doi.org/10.1063/1.3273074`.

[63] E. C. Shipple. “Potentials of surfaces in space”. In: *Reports on Progress in Physics* 44.11 (1981), pp. 1197–1250. ISSN: 00344885. DOI: `10.1088/0034-4885/44/11/002`.

[64] Themis Matsoukas, Marc Russell, and Matthew Smith. “Stochastic charge fluctuations in dusty plasmas”. In: *Journal of Vacuum Science and Technology A: Vacuum, Surfaces, and Films* 14.2 (1996), pp. 624–630. ISSN: 0734-2101. DOI: `10.1116/1.580156`.

[65] Giovanni Lapenta. “Simulation of charging and shielding of dust particles in drifting plasmas”. In: *Physics of Plasmas* 6.5 II (1999), pp. 1442–1447. ISSN: 1070664X. DOI: 10.1063/1.873395.

[66] Vadim N. Tsytovich et al. “Elementary processes in complex plasmas”. In: *Lecture Notes in Physics* 731.1 (2008), pp. 67–140. ISSN: 00758450. DOI: 10.1007/978-3-540-29003-2_3.

[67] Jan Carstensen, Franko Greiner, and Alexander Piel. “Ion-wake-mediated particle interaction in a magnetized-plasma flow”. In: *Physical Review Letters* 109.13 (2012), pp. 1–4. ISSN: 00319007. DOI: 10.1103/PhysRevLett.109.135001.

[68] Xian Zhu Tang and Gian Luca Delzanno. “Orbital-motion-limited theory of dust charging and plasma response”. In: *Physics of Plasmas* 21.12 (2014). ISSN: 10897674. DOI: 10.1063/1.4904404. arXiv: 1503.07820. URL: http://dx.doi.org/10.1063/1.4904404.

[69] M. Puttscher et al. “Vertical oscillations of dust particles in a strongly magnetized plasma sheath induced by horizontal laser manipulation”. In: *Physics of Plasmas* 24.1 (2017). ISSN: 10897674. DOI: 10.1063/1.4973231. URL: http://dx.doi.org/10.1063/1.4973231.

[70] E. Thomas et al. “Pattern formation in strongly magnetized plasmas: Observations from the magnetized dusty plasma experiment (MDPX) device”. In: *Plasma Physics and Controlled Fusion* 62.1 (2020). ISSN: 13616587. DOI: 10.1088/1361-6587/ab410c.

[71] Songfen Liu et al. “The structure of a two-dimensional magnetic dusty plasma”. In: *Journal of Physics A: Mathematical and General* 38.13 (2005), pp. 3057–3063. ISSN: 03054470. DOI: 10.1088/0305-4470/38/13/016.

[72] Mártin Lampe et al. “Trapped ion effect on shielding, current flow, and charging of a small object in a plasma”. In: *Physics of Plasmas* 10.5 (2003), pp. 1500–1513. DOI: 10.1063/1.1562163. eprint: https://doi.org/10.1063/1.1562163. URL: https://doi.org/10.1063/1.1562163.

[73] E Thomas Jr, R L Merlino, and M Rosenberg. “Magnetized dusty plasmas: the next frontier for complex plasma research”. In: *Plasma Phys. Control. Fusion* 54.12 (Dec. 2012), p. 124034.

[74] A Barkan and R L Merlino. “Confinement of dust particles in a double layer”. In: *Phys. Plasmas* 2.9 (Sept. 1995), pp. 3261–3265.

[75] E C Whipple, T G Northrop, and D A Mendis. “The electrostatics of a dusty plasma”. en. In: *J. Geophys. Res.* 90.A8 (Aug. 1985), pp. 7405–7413.

[76] Sergey A Khrapak, Alexey V Ivlev, and Gregor E Morfill. “Momentum transfer in complex plasmas”. en. In: *Phys. Rev. E Stat. Nonlin. Soft Matter Phys.* 70.5 Pt 2 (Nov. 2004), p. 056405.

Appendices

Appendix A

Molecular Dynamics Simulation Code

This first appendix includes the molecular dynamics simulation which is used for analysis of the theory. This is broken up into multiple different sections

1. md_sim.c includes the main simulation (This includes the integration method, BAOAB as well as an alternate RK4 method)
2. md_sim_class.c includes all of the functions which are used in the main portion of the simulation
3. md_sim_class.h includes the implementation and naming of all functions
4. md_sim_ext.h includes all of the constants used in the simulation

The main piece of the simulation which is absent here is the CO-OPTIONS functions which come from the COPLA software developed by uwe konopka. The main pieces which are missing are those which implement the running of the software. This software is available fully on git-hub and has included version control

A.1 md_sim.c

```
//==============================================================================
// This code is a simple molecular dynamic simulation for use with dusty
// plasmas. It is mathematically implemented using the Runge-Kutta 4 or
// RK4 method.
```

```
//
// Written by: Dylan Funk at Auburn University, 2018-2022
//
//===========================================================================

#include "md_sim_ext.h"
#include "COPLA-Common/co_options.h"       // option headers and structures
#include "md_sim_class.c"

#include <stddef.h>                         // needed for NULL
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <map>
#include <random>
#include <cmath>
#include <iostream>
#include <iomanip>
#include <ctype.h>
#include <windows.h>
#include <unistd.h>
#include <sys/stat.h>
#include <limits.h>
#include <sstream>
#include <string>

```

```
#define MD_SIM_VERSION    001
// #define DEBUG

//=========================================================================
//Commandline parameters for the program including physical parameters

_BOOL_ md_sim_help = _FALSE_;                  // set, if help output is needed
_UINT_ md_sim_num_particles = 100;             // number of particles to simulate
double  md_sim_drag_coef = drag_coef;          // need replacement [units] xxxx
double  md_sim_size = size;                    // box size [Debye lengths]
double  md_sim_magnetic_field_Bz = B_field;     // field in Bz direction [T]
double  md_sim_simulation_interval_t1 = 10; // total interval of simulation[s]
double  md_sim_time_step_dt = 0.001;           // time steps size [s]
double  md_sim_E_max_x = E_max_x;
double  md_sim_E_max_y = E_max_y;
double  md_sim_temperature = md_temp;
double  md_sim_q_val = q_val;
double  md_sim_start_velocity = start_velocity;
int     md_sim_num_time_steps = 1 +
        int(floor((md_sim_simulation_interval_t1)/md_sim_time_step_dt));
_BOOL_ md_sim_flag_save_parameter = _FALSE_;   // flag, if to save parameters
_BOOL_ md_sim_flag_load_parameter = _FALSE_;   // flag, if to use prev. values
_BOOL_ md_sim_test_set_parameter = _FALSE_;



char *q_input = NULL;
char *size_input = NULL;
char *B_input = NULL;
char *temp_input = NULL;
char *total_time_input = NULL;
char *timestep_input = NULL;
char *drag_input = NULL;
char *vel_input = NULL;

char    *fname_buffer = NULL;
_UINT_ fname_buffer_MAX = 1024;

const char    *fname_option_default = "md_sim";
char    *fname_option = NULL;
co_options md_sim_dict = co_options("md_sim", MD_SIM_VERSION);
md_sim_class md_sim_functions = md_sim_class();





double calculateSD(double data[], int num_vals)
{
  double sum = 0.0, mean, sd = 0.0, Differ, varsum = 0;
  int i;
  for(i = 0; i < num_vals; i++){
    sum += data[i];
  }
  mean = sum/num_vals;
  for(i = 0; i < num_vals; i++){
    Differ = data[i] - mean;
    varsum = varsum + pow(Differ,2);
  }
  double variance = varsum/num_vals;
  sd = sqrt(variance);
  return sd;
}





//----------------------------------------------------------------------------
//  rate_x
//
//  Rate from the force equation (or more precicesly,
//  the rate = dv/dt = F/mass of moving charge)
//
//  The way that rk4 works is that it solves the equation dx/dt = f(x) where
//  for this situation x is our velocity v
//
//  Variables : x, y = position of moving particle
//                   v = component of velocity for current calculation
//                                    q_x, q_y = position of all stationary particle
//
//  Note these rate calculations view a particle k as moving at time i
//  while viewing all others as stationary
//----------------------------------------------------------------------------
double rate_x(double t, double vx, double vy, double *qx, double *qy, int k,
                int  num_particles ,double  box_size ,  double  md_sim_q_val)
{

  //”””NOTE:  REMOVE  THIS  FOR  STANDARD  RUNS”””
  //This  section  is  for  runs  using  external  constant  force
  //”””-----------------------------------””””
  //double  f_drive  =  0;
  //double  f_external_y  =  -3E-15;
  //”””-----------------------------------””””

  double  rate  =  0;
  double  drag  =  0;//NOTE:Drag  accounted  for  in  sim  steps
  md_sim_class::vector  Efield  =
        md_sim_functions.electric_field(qx[k],  qy[k],  box_size);
  double  Ex  =  Efield.x;

  double  EcrossB  =  md_sim_q_val  *  (Ex  +  vy  *  B_field)  /  m;
  double  boundx  =  0;
        rate  =  f_drive  /  m  +  EcrossB;
  //printf(”v  cross  B  x  =  %.5f\n”,  md_sim_q_val  *  (vy  *  B_field)  /  m);
        for  (int  j  =  0;  j  <  num_particles;  j++)
        //  This  loop  cycles  through  each  available  charge  and
        //  includes  the  effect  in  the  rate
        {
                        if(j  !=  k)
                        {
                        double  delta_x  =  qx[k]  -  qx[j];
      if(qx[k]  <=  box_size/2)
```

```
        {
          if(fabs(delta_x) >= box_size/2) delta_x = box_size - fabs(delta_x);
        }
        if(qx[k] > box_size/2)
        {
          if(fabs(delta_x) >= box_size/2) delta_x = -box_size + fabs(delta_x);
        }
                          double delta_y = qy[k] - qy[j];
              // Note: these two are distances from the particle
              //  to the adjacent charges
        double r_square = fabs(delta_x * delta_x + delta_y * delta_y);
                          double r = sqrt(r_square);
              // Distance from particle to stationary charges

        if(r < 1E-5)
        {
          r = 1E-5;
        }
                    rate=rate+kappa*md_sim_q_val*md_sim_q_val*(1/(lambda*r*r))*(r+l
             *(exp(-r/lambda))*(delta_x/r)/m;
             // Add the effect from the current stationary charge
                          }
          }
          return rate;
}


//------------------------------------------------------------------------
```

```
// Rate_y
//
// Rate_y is much the same as ratex except there is no driving force
// The forces at work here are the electric field, magnetic field as
// well as epstein drag and inter-particle coulomb forces.
//------------------------------------------------------------------------
double rate_y(double t, double vx, double vy, double *qx, double *qy,
              int k, int num_particles, double box_size, double md_sim_q_val)
{
  //"""NOTE: REMOVE THIS FOR STANDARD RUNS"""
  //"""------------------------------------"""
  //double f_drive = 0;
  //double f_external_y = -3E-15; CHANGE THIS IN .h FILE
  //"""------------------------------------"""
  md_sim_class::vector Efield =
        md_sim_functions.electric_field(qx[k], qy[k], box_size);
  double Ey = Efield.y;
  if (qy[k] > box_size)
  {
    Ey = -1000;
    //printf("Upper bound, (%.4f, %.4f)\n", qy[k]/lambda, md_sim_q_val*Ey);
  }
  else if (qy[k] <= 0)
  {
    Ey = 1000;
    //printf("Lower bound, (%.4f, %.4f)\n", qy[k]/lambda, md_sim_q_val*Ey);
  }
  double drag = 0;// Accounted for in simulation integration
  double rate =  md_sim_q_val * (Ey - vx * B_field) / m + f_external_y / m;

   //REMOVE FOR NON-COLUMN SIMULATION
  //rate = rate - 1*md_sim_q_val/(1000*electron_charge);
  //double F_Rand = ((double)(rand() % 1000000)/1000000.0*2 - 1);
  //printf("v cross B y = %.5f\n", md_sim_q_val * ( - vx * B_field) / m);

        for (int j = 0; j < num_particles; j++)
  // This loop cycles through each available
  // charge and includes the effect in the rate
        {
                if(j != k)
                {
                double delta_x = qx[k] - qx[j];
      if(qx[k] >= box_size/2)
      {
        if(fabs(delta_x) >= box_size/2) delta_x = box_size - fabs(delta_x);
      }
      if(qx[k] < box_size/2)
      {
        if(fabs(delta_x) >= box_size/2) delta_x = -box_size + fabs(delta_x);
      }
                        double delta_y = qy[k] - qy[j];
            // Note: these two are distances from the particle
            //  to the adjacent charges

      double r_square = fabs(delta_x * delta_x + delta_y * delta_y);
                        double r = sqrt(r_square);
```

```
              // Distance from particle to stationary charges

         if(r < 1E-5)
         {
           r = 1E-5;
             // Do not allow particles to overlap
         }
                   rate=rate+kappa*md_sim_q_val*md_sim_q_val*(1/(lambda*r*r))*(r+l
         *(exp(-r/lambda))*(delta_y/r)/m;
              // Add the effect from the current stationary chare
                   }
           }
  //printf("rate y: %.5f\n", rate);
           return rate;
}

//-----------------------------------------------------------------------
// commandline help message header
//-----------------------------------------------------------------------

int md_sim_usage(void)
{
  fprintf(stderr,"=====================================================\n");
  fprintf(stderr,"This␣program␣is␣a␣molecular
␣␣␣␣␣␣␣␣␣␣dynamic␣simulation␣for␣use␣with\n");
  fprintf(stderr,"dusty␣plasmas␣and␣was␣written␣by␣Dylan␣Funk␣at␣Auburn\n");
  fprintf(stderr,"University␣in␣2018\n");
  fprintf(stderr,"=====================================================\n");

  return(0);
}

//----------------------------------------------------------------------------
// Output simulation parameters
//----------------------------------------------------------------------------

int md_sim_print_simulation_parameters()
{
  printf("----------------------------------------------------------------\n");

  printf("Number of Particles: %i\n",              md_sim_num_particles
);
  printf("Number of Time Steps: %i\n",             md_sim_num_time_steps
);
  printf("Time interval: %f\n",                    md_sim_time_step_dt
);
  printf("Box Size (in debye lengths): %f\n",  md_sim_size
);
  printf("Electric Field Max (x-dir): %f\n",    md_sim_E_max_x
);
  printf("Electric Field Max (y-dir): %f\n",    md_sim_E_max_y
);
  printf("Magnetic Field: %f\n",                   md_sim_magnetic_field_Bz  );
  printf("Drag Coefficient: %.10e\n",              md_sim_drag_coef
);
```

```
  printf("Dust Charge: %.2e\n",                    md_sim_q_val);
  printf("Dust radius: %.5e\n",                    dust_radius);
  printf("Dust mass: %.5e\n",                      m);
  printf("Dust Temperature: %.5e\n",               md_sim_temperature);
  printf("Start velocity: %.5e\n",                 md_sim_start_velocity);
  printf("\nFile Name: %s_****.txt\n",             fname_option);


  if(md_sim_flag_save_parameter == 0)
    printf("Flags: Do NOT save particle positions\n");
  if(md_sim_flag_save_parameter == 1)
    printf("Flags: Save last particle positions\n");
  if(md_sim_flag_load_parameter == 0)
    printf("Flags: Random initial particle positions\n");
  if(md_sim_flag_load_parameter == 1)
    printf("Flags: Initial particle positions loaded from file\n");

  printf("---------------------------------------------------------------\n");

  return(0);
}

```

```
//----------------------------------------------------------------------
//  Main
//----------------------------------------------------------------------

int main(int argc, char *argv[])
{

  double tic = clock();
  fname_buffer = (char*)malloc(fname_buffer_MAX);
  *fname_buffer = 0;

  //--------------------------------------------------------------------
  //  Section for Options
  //--------------------------------------------------------------------
  md_sim_dict.add_option("help/h/01/show␣help/1/",
    &md_sim_help);
  md_sim_dict.add_option("num_particles/n/01/Set␣Number␣of␣Particles/U/",
    &md_sim_num_particles);
  md_sim_dict.add_option("Drag/d/01/Set␣drag␣[default␣2E-13]/S/",
    &drag_input);
  md_sim_dict.add_option("vel/v/01/Set␣start␣velocity
␣␣␣␣[default␣driving␣force␣over␣drag]/S/", &vel_input);
  md_sim_dict.add_option("Size/z/01/Set␣Box␣Size
␣␣␣␣[default␣100␣debye␣lengths]/S/",&size_input);
  md_sim_dict.add_option("temp/k/01/Set␣Dust␣Temp/S/",&temp_input);
  md_sim_dict.add_option("charge/q/01/Set␣Dust␣charge␣(number␣of␣electron
␣␣␣␣charges)/S/",&q_input);
  md_sim_dict.add_option("Bfield/b/01/Set␣Magnetic␣Field␣[default␣0␣T]/S/",
    &B_input);
  md_sim_dict.add_option("Time/t/01/Set␣total␣time␣[default␣2.0␣s]/S/",
    &total_time_input);
  md_sim_dict.add_option("tInt/i/01/Set␣time␣interval␣[default␣0.001␣s]/S/",
    &timestep_input);
  md_sim_dict.add_option("save/s/01/Set␣save␣options␣[default␣0␣(dont␣save),
␣␣␣1␣(save␣last␣position)]/1/",&md_sim_flag_save_parameter);
  md_sim_dict.add_option("load/l/01/Set␣load␣options
␣␣␣␣[default␣0␣(random␣particle␣positions),␣1␣(load␣particle␣position)]/1/",
    &md_sim_flag_load_parameter);
  md_sim_dict.add_option("test/x/01/Make␣test␣values(1␣for␣yes,␣0␣for␣no)/1/",
    &md_sim_test_set_parameter);
  md_sim_dict.add_option("filename/f/01/Set␣file␣name/S/",&fname_option);
  md_sim_dict.register_detailed_help_callback(&md_sim_usage);


  //printf("\n\n\n-- %.3e -- \n\n\n",md_sim_q_val);
  int pos = 1, res = 0;
  if (argc > pos) res = md_sim_dict.evaluate_comline(argc,argv,&pos);


  if (q_input != NULL)          {md_sim_q_val = atof(q_input);}
  if (temp_input != NULL)       {md_sim_temperature = atof(temp_input);}
  if (size_input != NULL)       {md_sim_size = atof(size_input);}
  if (B_input != NULL)          {md_sim_magnetic_field_Bz = atof(B_input);}
  if (total_time_input != NULL){md_sim_simulation_interval_t1 =
atof(total_time_input);}
  if (timestep_input != NULL)  {md_sim_time_step_dt = atof(timestep_input);}
  if (drag_input != NULL)        {md_sim_drag_coef = atof(drag_input);}
  if (vel_input != NULL)         {md_sim_start_velocity = atof(vel_input);}


  int     md_sim_num_time_steps = 1 +
          int(floor((md_sim_simulation_interval_t1)/md_sim_time_step_dt));

  if (md_sim_help)
  {
md_sim_dict.print_full_options_help();
exit(0);
  }

  if(fname_option == NULL)
      strcpy(fname_option, fname_option_default);
  else if(sizeof(fname_option) < fname_buffer_MAX - 24)
      strcpy(fname_buffer, fname_option);
  else if(sizeof(fname_option) > fname_buffer_MAX - 24)
  {
printf("ERROR: File name must be less that %d characters",
      (fname_buffer_MAX - 24));
exit(1);
  }

  md_sim_class::vector v;
  double *vx_old = (double*)malloc(md_sim_num_particles * sizeof(double));
  double *vy_old = (double*)malloc(md_sim_num_particles * sizeof(double));
  double *vy_new = (double*)malloc(md_sim_num_particles * sizeof(double));
  double *vx_new = (double*)malloc(md_sim_num_particles * sizeof(double));
  double *x_old = (double*)malloc(md_sim_num_particles * sizeof(double));
  double *x_new = (double*)malloc(md_sim_num_particles * sizeof(double));
  double *y_old = (double*)malloc(md_sim_num_particles * sizeof(double));
  double *y_new = (double*)malloc(md_sim_num_particles * sizeof(double));
  int   *particle_flag = (int*)malloc(md_sim_num_particles * sizeof(int));
  int   **grid;

  double md_sim_box_size = md_sim_size*lambda;

  const int num_grid_x = (int)floor(md_sim_box_size/GRID_SIZE);
  const int num_grid_y = (int)floor(md_sim_box_size/GRID_SIZE);
  int cross_count = 0;
  //double vx_max = 0, vx_min = 0, vy_max = 0, vy_min = 0;
  //box_size = md_sim_size*lambda;
  //box_size = md_sim_size*lambda;
  //dens = (double*)malloc(num_grid_y * sizeof(double));
  grid = (int**)malloc(num_grid_y * sizeof(int *));
  //double *grad = (double*)malloc(md_sim_num_particles * sizeof(double));;
  for(int i = 0; i < num_grid_y; i++)
grid[i] = (int*)malloc(num_grid_x * sizeof(int));

  if(md_sim_flag_load_parameter == 0)
md_sim_functions.init_parameters(x_old, y_old, vx_old, vy_old, x_new,
                        y_new, vx_new, vy_new, md_sim_num_particles,
                        md_sim_box_size,md_sim_start_velocity);
  else if(md_sim_flag_load_parameter == 1)
  {
//std::ifstream infile("start_positions.txt");
FILE* f1 = fopen("start_positions.txt", "r");

printf("\nLoading␣Particle␣Positions...\n");
int num_particles = 0;
char parse[1], del_1[1], del_2[1];
double tmp_x, tmp_y,tmp_vx,tmp_vy;
fscanf(f1, "%d\n", &num_particles);


/*for(int j=0;j<num_particles+1;j++) x_old[j]=atof(fgets(str,8,f1));
printf("%.6f", x_old[num_particles-1]);
for(int j = 0; j < num_particles-1; j++) fscanf(f1, "%lf", y_old[j]);
fscanf(f1, "%lf\n", &y_old[num_particles]);
for(int j = 0; j < num_particles-1; j++) fscanf(f1, "%lf", vx_old[j]);
fscanf55:()(f1, "%lf\n", &vx_old[num_particles]);
for(int j = 0; j < num_particles-1; j++) fscanf(f1, "%lf", vy_old[j]);
fscanf(f1, "%lf\n", &vy_old[num_particles]);
  */
for(int k = 0; k < num_particles; k++)
{
  fscanf(f1, "%lf", &tmp_x);
  x_old[k] = tmp_x*lambda;
  x_new[k] = tmp_x*lambda;
  //printf("%.3e ", x_new[k]);
}
for(int k = 0; k < num_particles; k++)
{
      fscanf(f1, "%lf", &tmp_y);
      y_old[k] = tmp_y*lambda;
      y_new[k] = tmp_y*lambda;
  }
  for(int k = 0; k < num_particles; k++)
  {
      fscanf(f1, "%lf", &tmp_vx);
      vx_old[k] = tmp_vx;
      vx_new[k] = tmp_vx;
  }
  for(int k = 0; k < num_particles; k++)
  {
      fscanf(f1, "%lf", &tmp_vy);
      vy_old[k] = tmp_vy;
      vy_new[k] = tmp_vy;
  }

  fclose(f1);
  printf("Loading Complete!");
  }



  //---------------------------------------------------------------------------
  // Section for main simulation operation
  //---------------------------------------------------------------------------
  chdir("Data");
  mkdir(fname_option);
```

```
    chdir(fname_option);
    strcpy(fname_buffer,fname_option);
    printf("====\n\n%s\n\n====\n\n",fname_buffer);
    FILE *md_sim_f_grad = fopen(strcat(fname_buffer,"_gradient.txt"), "w");
    strcpy(fname_buffer,fname_option);
    FILE *md_sim_f_vals = fopen(strcat(fname_buffer,"_values.txt"), "w");
    strcpy(fname_buffer,fname_option);
    FILE *md_sim_f_count = fopen(strcat(fname_buffer,"_count.txt"), "w");
    strcpy(fname_buffer,fname_option);
    FILE *md_sim_f_vavg = fopen(strcat(fname_buffer,"_vx_ave.txt"),"w");
    strcpy(fname_buffer,fname_option);
    FILE *md_sim_f_vx = fopen(strcat(fname_buffer,"_vx.txt"),"w");
    strcpy(fname_buffer,fname_option);
    FILE *md_sim_f_vy = fopen(strcat(fname_buffer,"_vy.txt"),"w");

      //md_sim_functions.load_parameters(x_old, y_old, vx_old, vy_old, x_new,
        //                   y_new, vx_new, vy_new);
    md_sim_print_simulation_parameters();

    md_sim_q_val = md_sim_q_val * electron_charge;

    printf("Start Simulation.\n");
    md_sim_functions.init_files(md_sim_num_time_steps, md_sim_num_particles,
                         md_sim_f_vals, md_sim_f_grad, md_sim_f_count,
                         md_sim_f_vavg, md_sim_time_step_dt,
                         md_sim_simulation_interval_t1,md_sim_box_size);//,
                         //fname_option,fname_buffer);
    printf("Files Initialized\n");
```

```
md_sim_functions.print_particle_positions(x_new, y_new,
  md_sim_num_particles, md_sim_f_vals);
//grid = md_sim_functions.grid_count(x_old, y_old, md_sim_time_step_dt,
// 0, md_sim_num_particles, dens, md_sim_f_grad);
//md_sim_functions.dens_count(grid, md_sim_time_step_dt, 0,
// dens, md_sim_f_grad);

if(md_sim_test_set_parameter == 1) {
  printf("\n ----- NOTE: CREATING TEST VALUES ----- \n");
}
//==============================================================================
// MAIN TIME LOOP:
//    For each time step, i, the second loop cycles through each particle
// and calculates its forces on all other particles using the rk4
// method defined above
//
// NOTE: The first index of each array is the timestep
//        (int i is the time step), the second index,
//        k, refers to each individual particle.
//==============================================================================
      for (int i = 1; i < md_sim_num_time_steps-1; i++)
      {
    //Report on % complete
    md_sim_functions.progress_report(i, md_sim_num_time_steps);

    //For each charge, calculate the new velocity and position
              for(int k = 0; k < md_sim_num_particles; k++)
              {
```

```
            if(md_sim_test_set_parameter == 0)
            {
/*                              v = md_sim_functions.rk4(&rate_x ,
                 &rate_y ,
                  md_sim_time_step_dt ,
                  md_sim_simulation_interval_t1 + md_sim_time_step_dt * (i - 1),
                  vx_old[k],
                  vy_old[k],
                  x_old ,
                  y_old ,
                 k,
                  md_sim_num_particles ,
                  md_sim_box_size );


              //Scaling factor is damping time/dt (damping time =
                m/drag coefficient)
               vx_new[k] = md_sim_functions.random_velocity_update(v.x,
                md_sim_time_step_dt );
                     vy_new[k] = md_sim_functions.random_velocity_update(v.y,
                md_sim_time_step_dt );
                     x_new[k] = vx_new[k] * md_sim_time_step_dt + x_old[k];
                     y_new[k] = vy_new[k] * md_sim_time_step_dt + y_old[k];
*/

              //baoab method:
              //Step B:
              double vx_half = vx_old[k] + rate_x(md_sim_simulation_interval_t1
            + md_sim_time_step_dt * (i - 1),vx_old[k],vy_old[k],x_old,y_old,k,
            md_sim_num_particles,md_sim_box_size,
            md_sim_q_val)*md_sim_time_step_dt/2;
        double vy_half = vy_old[k] + rate_y(md_sim_simulation_interval_t1 +
            md_sim_time_step_dt * (i - 1),vx_old[k],vy_old[k],x_old,y_old,k,
            md_sim_num_particles,md_sim_box_size,
            md_sim_q_val)*md_sim_time_step_dt/2;
        //Step A: position_update
        double x_half = x_old[k] + vx_half*md_sim_time_step_dt/2;
        double y_half = y_old[k] + vy_half*md_sim_time_step_dt/2;
        //Step O:
        vx_old[k] = md_sim_functions.random_velocity_update(vx_half,
            md_sim_time_step_dt,md_sim_drag_coef);
        vy_old[k] = md_sim_functions.random_velocity_update(vy_half,
            md_sim_time_step_dt,md_sim_drag_coef);
        //Step A: position_update
        x_new[k] = x_half + vx_old[k]*md_sim_time_step_dt/2;

        //x_new[k] = md_sim_box_size/2;
        // *$*$ THIS SECTION IS USED FOR STATIONARY COLUMN

        y_new[k] = y_half + vy_old[k]*md_sim_time_step_dt/2;
        //Step B:
        vx_new[k] = vx_old[k] + rate_x(md_sim_simulation_interval_t1 +
            md_sim_time_step_dt * (i - 1),vx_old[k],vy_old[k],x_new,y_new,k,
            md_sim_num_particles,
            md_sim_box_size,md_sim_q_val)*md_sim_time_step_dt/2;
        vy_new[k] = vy_old[k] + rate_y(md_sim_simulation_interval_t1 +
            md_sim_time_step_dt * (i - 1),vx_old[k],vy_old[k],x_new,y_new,k,
            md_sim_num_particles,
            md_sim_box_size,md_sim_q_val)*md_sim_time_step_dt/2;
        vx_old[k] = vx_new[k];
        vy_old[k] = vy_new[k];




        md_sim_functions.bound_check(&x_new[k], &y_new[k], &vx_new[k],
                                    &vy_new[k],md_sim_box_size);
      }
      else if(md_sim_test_set_parameter == 1)
      {
        v = md_sim_functions.test_gen(md_sim_num_particles,f_drive);

                                    vx_new[k] = v.x;
                                    vy_new[k] = v.y;

        x_new[k] = vx_new[k] * md_sim_time_step_dt + x_old[k];
        md_sim_functions.bound_check(&x_new[k], &y_new[k], &vx_new[k],
                                     &vy_new[k],md_sim_box_size);
      }

      //if(vx_new[k] > vx_max) vx_max = vx_new[k];
      //if(vx_new[k] < vx_min) vx_min = vx_new[k];
      //if(vy_new[k] > vy_max) vy_max = vy_new[k];
      //if(vy_new[k] < vy_min) vy_min = vy_new[k];
```

```

                    }
        cross_count=md_sim_functions.update_count(x_old, x_new, y_new, vx_new,
                                vy_new, md_sim_num_particles, md_sim_f_count,
                                md_sim_f_vx, md_sim_f_vy,cross_count,md_sim_box_size);
        //grid = md_sim_functions.grid_count(x_old, y_old, md_sim_time_step_dt,
        //i, md_sim_num_particles, dens, md_sim_f_grad);
        md_sim_functions.update_arrays(x_old,y_old,vx_old,vy_old,x_new,y_new,
                                            vx_new, vy_new,md_sim_num_particles);
        md_sim_functions.print_average_velocities(vx_new,md_sim_num_particles,
                                md_sim_f_vavg);
        if(plot_count == plot_interval-1)
        {
          md_sim_functions.print_particle_positions(x_new, y_new,
                                md_sim_num_particles, md_sim_f_vals);
          plot_count = 0;
        }
        else plot_count++;

        //------------------------------------------------------------------
        //       Update the file that saves start positions of the Particles
        //------------------------------------------------------------------
        if(md_sim_flag_save_parameter == 1 && i % save_interval == 0)
        {
          chdir("..");
          chdir("..");
          printf("Saving...");
          FILE *f3 = fopen("start_positions.txt","w");
```

```
            fprintf(f3, "%i\n", md_sim_num_particles);
            md_sim_functions.print_particle_positions(x_new, y_new,
              md_sim_num_particles, f3);
            md_sim_functions.print_particle_velocities(vx_new, vy_new,
              md_sim_num_particles, f3);
            fclose(f3);
            chdir("Data");
            chdir(fname_option);
          }

          //md_sim_functions.dens_count(grid, md_sim_time_step_dt, i,
          //dens, md_sim_f_grad);
          md_sim_functions.grid_zero(grid, md_sim_box_size);
            }

      fclose(md_sim_f_vavg);
      fclose(md_sim_f_count);
      fclose(md_sim_f_vx);
      fclose(md_sim_f_vy);
      md_sim_functions.final_update(cross_count,fname_option,fname_buffer,
        md_sim_num_particles,md_sim_time_step_dt, md_sim_q_val);

      double toc = clock();
      printf("\n\nDone!\n");
      double time_elapsed = ((double) (toc - tic))/ CLOCKS_PER_SEC;
      printf("Elapsed␣Time:␣%f␣seconds␣\n", time_elapsed);
  //   double sd_calc = calculateSD(vx_new, md_sim_num_particles);
  //   printf("Standard Deviation = %f", sd_calc);
      //------------------------------------------------------------------------
      // Section for cleaning up memory
      //------------------------------------------------------------------------

      // free string variables, if aquired by co_options
      md_sim_dict.cleanup_memory();

      return 0;
  }
```

A.2 md_sim_class.c

```
//========================================================================
// This code is a simple molecular dynamic simulation for use with dusty
// plasmas. It is mathematically implemented using the Runge-Kutta 4 or
// RK4 method. This is the CLASS!
//
// Written by: Dylan Funk at Auburn University, 2022
//========================================================================

#include "md_sim_class.h"                  // my header file :)
#include "md_sim_ext.h"
#include "COPLA-Common/co_options.h"

#include <stddef.h>                        // needed for NULL
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
```

```
#include <time.h>
#include <iostream>
#include <string>
#include <random>
#include <ctime>
#include <windows.h>
#include <sstream>
#include <fstream>


std::default_random_engine generator;
std::normal_distribution<double> distribution(0.0,sqrt(kb*md_temp/m));
std::normal_distribution<double> dist2(0.0,1.0);
//==========================================================================
// Public
//==========================================================================

//--------------------------------------------------------------------------
// md_sim_class
//
// Constructor for md_sim_class.
//
//--------------------------------------------------------------------------

md_sim_class::md_sim_class()
{

}
```

```

//--------------------------------------------------------------------------
// ~md_sim_class
//
// Destructure for md_sim_class.
// Keep care to free memory yourself that might be occupied.
//--------------------------------------------------------------------------

md_sim_class::~md_sim_class()
{

}

//--------------------------------------------------------------------------
// initialize
//
// initialize variables for the simulation. Use this at least once before
// staring any simulations!!!
//--------------------------------------------------------------------------

int md_sim_class::initialize()
{
  return(0);
}

//==========================================================================
// Private
//==========================================================================
```

```cpp


//---------------------------------------------------------------------------
//  bound_check
//
//  This checks the particle positions with the boundary conditions listed
//  Currently boundary conditions are:
//  y-direction, rigid boundaries
//  x-direction, cyclical boundaries
//---------------------------------------------------------------------------

void md_sim_class::bound_check(double* x, double* y, double* vx, double* vy,
                               double box_size)
{
  double box_size_y = box_size/x_y_ratio;
  if(*x < 0)
  {
    //*x = 0;                           // Rigid boundary
    *x = box_size + fmod(*x,box_size); //Circular boundary condition
    //printf("Particle x < 0:  %f", *x);
  }

  if(*x > box_size)
  {

    //*x = box_size - (double)(rand() % 20000)/2000000000.0;
    *x = fmod(*x,box_size); //Circular boundary condition
    //printf("Particle x > box_size:  %f", *x);
  }

  if(*y < -box_size_y/(30))
  {
    //if(*vy < 0) *vy = 0;
    //printf("\n%f", *y);
    //*y = -box_size_y/(30);
    //printf("Particle y < 0:  %f", *y);
  }

  if(*y > 33*box_size_y/(30))
  {
    //if(*vy > 0) *vy = 0;
    //printf("\n%f", *y);
    //*y = 33*box_size_y/(30) - (double)(rand() % 20000)/2000000000.0;
    //printf(" Particle y > box_size: %f", *y);
  }
}

//------------------------------------------------------------------------
//  dens_count
//
//  This comes in for density gradient calculates later by summing (in the left
//  and the right halves) the total number of particles for each y cell
//  coordinate in each time step
//------------------------------------------------------------------------

void md_sim_class::dens_count(int** grid, double dt, int time_step, double* den
                                FILE *f_grad, double box_size)
{
  const int num_grid_x = (int)floor(box_size/GRID_SIZE);
  const int num_grid_y = (int)floor(box_size/GRID_SIZE);
  int grad_y = 0;
  double sumx = 0, sumx2=0,sumy=0,sumy2=0,sumxy=0;
  for(int j = 0; j < num_grid_y; j++)
  {
    dens[j] = 0;
    for(int k = 0; k < num_grid_x; k++)
    {
      dens[j] = dens[j] + grid[k][j];
      //Total number of particles on grid space
    }
    dens[j] = dens[j]/num_grid_y;
    //Average overall x values
  }

  for(int i = 0; i <num_grid_y; i++)
  {
    sumx += i;
    sumy += dens[i];
    sumx2 += pow(i,2);
    sumy2 += pow(dens[i],2);
    sumxy += i*dens[i];
  }

  double denom = num_grid_y * sumx2 - pow(sumx,2);
  grad_y = (num_grid_y * sumxy - sumx*sumy)/denom;
  fprintf(f_grad, "%f %f", grad_y, dt*time_step);
  fprintf(f_grad, "\n");
}

//------------------------------------------------------------------------
// electric_field
//
// This function returns the proper electric field in x and y based on the
// position of the particle, the complicated sign_x and E-field functions come
// from the geometric position of each particle to return the proper (linearly
// linearly increasing) electric field magnitude and direction (sign_x and
// sign_y)
//------------------------------------------------------------------------
struct md_sim_class::vector md_sim_class::electric_field(double x, double y,
  double box_size)
{
  struct vector Efield;
  double E_bound_x = box_size*0.99;
  double E_bound_y = box_size*0.99/x_y_ratio;
  double box_size_y = box_size/x_y_ratio;
  double sign_x = -(x - box_size/2 - E_bound_x/2)/
   fabs(x - box_size/2 - E_bound_x/2);
  double sign_y = -(y - box_size_y/2 - E_bound_y/2)/
   fabs(y - box_size_y/2 - E_bound_y/2);
  //These two yield either -1 or +1 depending on x or y

  if(fabs(x - box_size/2) < E_bound_x/2)
```

```
        Efield.x = 0;
  if(fabs(x - box_size/2) >= E_bound_x/2)
        Efield.x = E_max_x*2*sign_x*(fabs(fabs(x - box_size/2) - E_bound_x/2)/
           (box_size - E_bound_x));
  if(fabs(y - box_size_y/2) < E_bound_y/2)
        Efield.y = 0;
  if(fabs(y - box_size_y/2) >= E_bound_y/2)
        Efield.y = 0;
      /*
        //E_max_y*2*sign_y*(fabs(fabs(y - box_size_y/2) - E_bound_y/2)/
        (box_size_y - E_bound_y));
        //EDIT 2/13/23: Changed below section to be 10*box_size
        (relevance is the variance in y direction)
      */
  if(y > 10*box_size_y)
        Efield.y = -E_max_y;
  if(y < 0*box_size_y)
        Efield.y = E_max_y;

  return Efield;
}


void md_sim_class::final_update(int cross_count,char *fname_option,
  char* fname_buffer,int num_particles,double time_step_dt,double q)
{
  strcpy(fname_buffer,fname_option);
  //printf("\n\nAlmost There...\n\n");
```

```
    FILE *f_count_read = fopen(strcat(fname_buffer,"_count.txt"),"r");
    strcpy(fname_buffer,fname_option);
    FILE *f_vx_read = fopen(strcat(fname_buffer,"_vx.txt"),"r");
    strcpy(fname_buffer,fname_option);
    FILE *f_vy_read = fopen(strcat(fname_buffer,"_vy.txt"),"r");

    FILE *f_tmp = fopen(strcat(fname_buffer,"tmp.txt"),"w");
    strcpy(fname_buffer,fname_option);
    double *tmp_count;
    double *tmp_vx;
    double *tmp_vy;
    tmp_count = (double*)malloc(sizeof(double)*cross_count);
    tmp_vx = (double*)malloc(sizeof(double)*cross_count);
    tmp_vy = (double*)malloc(sizeof(double)*cross_count);

    printf("\n%d %s\n", cross_count, fname_buffer);
    for(int i = 0; i < cross_count; i++)
    {
      fscanf(f_count_read, "%lf", &tmp_count[i]);
      fscanf(f_vx_read, "%lf", &tmp_vx[i]);
      fscanf(f_vy_read, "%lf", &tmp_vy[i]);
    }
    fclose(f_count_read);
    fclose(f_vx_read);
    fclose(f_vy_read);

    //==================================================================
    //    Every time a particle is within a counting region, its positional and
```

```
// velocity data is saved and here it is copied to a file that
// ends "_count.txt"
// The first few lines of this are information about the sim:
// (Magnetic Field)
// (Input charge value)
// (Temperature)
// (Box size)
// (Time step)
// (Number of time steps)
// dens[0] dens[1]... dens[num_grid_y] (at t = 0)
// dens[0] dens[1]... dens[num_grid_y] (at t = 1)
// ...
// dens[0] dens[1] ... dens[num_grid_y] (at t = n)
//=====================================================================
FILE *f_count_write = fopen(strcat(fname_buffer,"_count.txt"),"w");
strcpy(fname_buffer,fname_option);


//=====================================================================
// To plot the x velocity, we will output the x velocity into a data file
// named "*_vx.txt" this will be used to plot in python. The data
// should look similar to this:
// (Number of crossings)
// vx[0] vx[1]... vx[num_particles] (at t = 0)
// vx[0] vx[1]... vx[num_particles] (at t = 1)
// ...
// vx[0] vx[1]... vx[num_particles] (at t = n)
//=====================================================================
FILE *f_vx_write = fopen(strcat(fname_buffer,"_vx.txt"),"w");
strcpy(fname_buffer,fname_option);
//==========================================================================
// To plot the y velocity, we will output the y velocity into a data file
// named "*_vy.txt" this will be used to plot in python.
// The data should look similar to this:
// (Number of crossings)
// vy[0] vy[1]... vy[num_particles] (at t = 0)
// vy[0] vy[1]... vy[num_particles] (at t = 1)
// ...
// vy[0] vy[1]... vy[num_particles] (at t = n)
//==========================================================================
FILE *f_vy_write = fopen(strcat(fname_buffer,"_vy.txt"),"w");
strcpy(fname_buffer,fname_option);


float n_q = q/electron_charge;
fprintf(f_count_write, "%f\n", B_field);
fprintf(f_count_write, "%e\n", n_q);
fprintf(f_count_write, "%f\n", md_temp);
fprintf(f_count_write, "%f\n", size);
fprintf(f_count_write, "%f\n", time_step_dt);

fprintf(f_count_write, "%d\n", cross_count);
fprintf(f_vx_write, "%d\n", cross_count);
fprintf(f_vy_write, "%d\n", cross_count);

for(int i = 0; i < cross_count-1; i++)
```

```
  {
    if(i % save_interval==0)
    {
      fprintf(f_count_write, "%.4f ", tmp_count[i]);
      fprintf(f_vx_write, "%.4f ", tmp_vx[i]);
      fprintf(f_vy_write, "%.4f ", tmp_vy[i]);
    }
  }

  fprintf(f_count_write, "%.4f", tmp_count[cross_count]);
  fprintf(f_vx_write, "%.4f", tmp_vx[cross_count]);
  fprintf(f_vy_write, "%.4f", tmp_vy[cross_count]);


  fclose(f_count_write);
  fclose(f_vx_write);
  fclose(f_vy_write);
}

//---------------------------------------------------------------------------
// grid_zero
//
// Zero out the grid variable, maybe unnecessary
//---------------------------------------------------------------------------
void md_sim_class::grid_zero(int** grid, double box_size)
{
  const int num_grid_x = (int)floor(box_size/GRID_SIZE);
  const int num_grid_y = (int)floor(box_size/GRID_SIZE);
  for(int p = 0; p < num_grid_x; p++)
     {
       for(int q = 0; q < num_grid_y; q++) grid[p][q] = 0;
     }
}

//---------------------------------------------------------------------------
//  grid_count
//
//  Take all the points and assign them to grid points
//---------------------------------------------------------------------------
int** md_sim_class::grid_count(double* x, double* y, double dt, int time_step,
                               int num_particles, double* dens, FILE *f_grad,
                               double box_size)
{
  const int num_grid_x = (int)floor(box_size/GRID_SIZE);
  const int num_grid_y = (int)floor(box_size/GRID_SIZE);
  int** grid = (int**)malloc(num_grid_y * sizeof(int *));
  for(int i = 0; i < num_grid_y; i++)
      grid[i] = (int*)malloc(num_grid_x * sizeof(int));
  //grid_zero(grid);

  int x_pos = 0;
  int y_pos = 0;
  for(int k = 0; k < num_particles; k++)
  {
    x_pos = (int) floor(x[k]/GRID_SIZE);
    y_pos = (int) floor(y[k]/GRID_SIZE);
```

```
        grid[x_pos][y_pos]++;
        //printf("%d", grid[x_pos][y_pos]);
    }

    int grad_y = 0;
    double sumx = 0, sumx2=0,sumy=0,sumy2=0,sumxy=0;
    for(int j = 0; j < num_grid_y; j++)
    {
        dens[j] = 0;
        for(int k = 0; k < num_grid_x; k++)
        {
            dens[j] = dens[j] + grid[k][j];
            //Total number of particles on grid space
        }
        dens[j] = dens[j]/num_grid_y;
        //Average overall x values
    }

    for(int i = 0; i <num_grid_y; i++)
    {
        sumx += i;
        sumy += dens[i];
        sumx2 += pow(i,2);
        sumy2 += pow(dens[i],2);
        sumxy += i*dens[i];
    }
    double denom = num_grid_y * sumx2 - pow(sumx,2);
    grad_y = (num_grid_y * sumxy - sumx*sumy)/denom;
```

```
  fprintf(f_grad, "%f %f", grad_y, dt*time_step);
  fprintf(f_grad, "\n");

  return grid;
}

//------------------------------------------------------------------------
//  init_files
//
//  Initialize the files to be written to
//------------------------------------------------------------------------
void md_sim_class::init_files(int num_time_steps, int num_particles,
                              FILE *f_vals, FILE *f_grad, FILE *f_count,
                              FILE *f_vave, double dt, double t1,
                              double box_size)//,
                              //char *fname_option,char* fname_buffer)
{
  const int num_grid_x = (int)floor(box_size/GRID_SIZE);
  const int num_grid_y = (int)floor(box_size/GRID_SIZE);
  //======================================================================
  // Once we calculate the positions of the particles, it is then output
  // to a data file called "values.txt", this is then input into a Python
  // code which plots our data at certain timesteps and produces
  // various image files which we can later use. The GDL code requires a
  // specific format:
  //
  //  (number of particles)
  //  (num_time_steps/plot_interval)
  //  (Display/Box width)
  //  (Display/Box height)
  //  x_0 x_1 x_2 ... x_k    (for t = t0)
  //  y_0 y_1 y_2 ... y_k    (for t = t0)
  //  x_0 x_1 x_2 ... x_k    (for t = t0 +dt)
  //  y_0 y_1 y_2 ... y_k    (for t = t0 +dt)
  //  ...
  //  x_0 x_1 x_2 ... x_k    (for t = t0 + n*dt)
  //  y_0 y_1 y_2 ... y_k    (for t = t0 + n*dt)
  // This format will be taken care of in various fprintf functions
  //====================================================================
  fprintf(f_vals,"%i\n%i\n%g\n%g\n",num_particles,
          (num_time_steps-1)/plot_interval+1,
          box_size/lambda, box_size/(lambda*x_y_ratio));

  //====================================================================
  //   To plot the density gradient we will output them into a data file
  // "_gradient.txt" this will be used to plot in python. The data should look
  // similar to this:
  // (Number of times steps)
  // (Number of grid points in y)
  // dens[0] dens[1]... dens[num_grid_y] (at t = 0)
  // dens[0] dens[1]... dens[num_grid_y] (at t = 1)
  // ...
  // dens[0] dens[1] ... dens[num_grid_y] (at t = n)
  //======================================================================
  fprintf(f_grad, "%d\n", num_time_steps);
  fprintf(f_grad, "%d\n", num_grid_y);
```

```

  fprintf(f_vave, "%.4f\n", dt);
  fprintf(f_vave, "%.4f\n", t1);
  fprintf(f_vave, "%.4e\n", drag_coef);
/*
  //==========================================================================
  //    To plot the density gradient we will output them into a data file named
  // "gradient.txt" this will be used to plot in GDL. The data should look
  // similar to this:
  // (Number of times steps)
  // (Number of grid points in y)
  // dens[0] dens[1]... dens[num_grid_y] (at t = 0)
  // dens[0] dens[1]... dens[num_grid_y] (at t = 1)
  // ...
  // dens[0] dens[1] ... dens[num_grid_y] (at t = n)
  //==================================================================
  FILE *f_vx_write = fopen(strcat(fname_buffer,"_vx.txt"),"w");
  strcpy(fname_buffer,fname_option);
  //==================================================================
  //    To plot the density gradient we will output them into a data file
  // "gradient.txt" this will be used to plot in GDL. The data should look
  // similar to this:
  // (Number of times steps)
  // (Number of grid points in y)
  // dens[0] dens[1]... dens[num_grid_y] (at t = 0)
  // dens[0] dens[1]... dens[num_grid_y] (at t = 1)
  // ...
  // dens[0] dens[1] ... dens[num_grid_y] (at t = n)
  //===========================================================================
  FILE *f_vy_write = fopen(strcat(fname_buffer,"_vy.txt"),"w");
  strcpy(fname_buffer,fname_option);
  */
}

//---------------------------------------------------------------------------
// load_parameters
//
// Loads data points from a previous file
//---------------------------------------------------------------------------
void md_sim_class::load_parameters(double* x_old, double* y_old,
                                   double* vx_old,double* vy_old,
                                   double* x_new, double* y_new,
                                   double* vx_new, double* vy_new)
{
  //std::ifstream infile("start_positions.txt");
  FILE* f1 = fopen("start_positions.txt", "r");

  printf("\nLoading␣Particle␣Positions...\n");
  int num_particles = 0;
  char parse[1], del_1[1], del_2[1];
  double *tmp_x, *tmp_y,*tmp_vx,*tmp_vy;
  fscanf(f1, "%d\n", &num_particles);

  tmp_x = (double*)malloc(sizeof(double)*num_particles);
  tmp_y = (double*)malloc(sizeof(double)*num_particles);
  tmp_vx = (double*)malloc(sizeof(double)*num_particles);
  tmp_vy = (double*)malloc(sizeof(double)*num_particles);

  fscanf(f1, "%lf", &tmp_x);
  fscanf(f1, "%lf", &tmp_y);
  fscanf(f1, "%lf", &tmp_vx);
  fscanf(f1, "%lf", &tmp_vy);
  /*for(int j = 0; j < num_particles+1; j++) x_old[j] = atof(fgets(str,8,f1));
  printf("%.6f", x_old[num_particles-1]);
  for(int j = 0; j < num_particles-1; j++) fscanf(f1, "%lf", y_old[j]);
  fscanf(f1, "%lf\n", &y_old[num_particles]);
  for(int j = 0; j < num_particles-1; j++) fscanf(f1, "%lf", vx_old[j]);
  fscanf(f1, "%lf\n", &vx_old[num_particles]);
  for(int j = 0; j < num_particles-1; j++) fscanf(f1, "%lf", vy_old[j]);
  fscanf(f1, "%lf\n", &vy_old[num_particles]);
*/
  for(int k = 0; k < num_particles; k++)
  {
    x_new[k] = tmp_x[k];
    y_new[k] = tmp_y[k];
    vx_new[k] = tmp_vx[k];
    vy_new[k] = tmp_vy[k];

    printf("%.3f", x_old);
  }

  fclose(f1);
  printf("Loading␣Complete!");
}
```

```

//----------------------------------------------------------------------
// init_parameters
//
// Initialization of parameters. This includes random distribution of dust
// particle locations as well as setting all charge velocities to zero
//----------------------------------------------------------------------
void md_sim_class::init_parameters(double* x_old, double* y_old,
                                   double* vx_old,double* vy_old,
                                   double* x_new, double* y_new,
                                   double* vx_new, double* vy_new,
                                   int num_particles,
                                   double box_size, double v_start)
{
  srand((int)time(0));
  for (int k = 0; k < num_particles; k++)
  {
    x_old[k] = (double)(rand() % 10000)/10000.0*box_size;
    // Position all new particles randomly
    y_old[k] = (double)(rand() % 10000)/10000.0*box_size/x_y_ratio;
    vx_old[k] = distribution(generator) + v_start;
    vy_old[k] = distribution(generator);
    x_new[k] = x_old[k];
    y_new[k] = y_old[k];
  }
}

//----------------------------------------------------------------------
```

```
// print_average_velocities
//
// Print the positions of the particles to a data file. Uses the format listed
// under the function init_files
//--------------------------------------------------------------------
void md_sim_class::print_average_velocities(double* vx,int num_particles,
  FILE *f_out)
{
  double vx_ave = 0;
  for(int k = 0; k < num_particles; k++)
  {
    vx_ave = vx_ave + vx[k]/lambda;
  }
  vx_ave = vx_ave/num_particles;

  printf("%.4f␣", vx_ave);
  fprintf(f_out, "%.4f␣", vx_ave);
}


//--------------------------------------------------------------------
// print_particle_positions
//
// Print the positions of the particles to a data file. Uses the format listed
// under the function init_files
//--------------------------------------------------------------------
void md_sim_class::print_particle_positions(double* x, double* y,
                                            int num_particles,FILE *f_out)
```

```
{
  for(int k = 0; k < num_particles; k++)
  {
    fprintf(f_out, "%.4f␣", x[k]/lambda);
  }
  fprintf(f_out,"\n");
  for(int k = 0; k < num_particles; k++)
  {
    fprintf(f_out, "%.4f␣", y[k]/lambda);
  }
  fprintf(f_out, "\n");
}

//----------------------------------------------------------------------------
// print_particle_velocities
//
// Print the velocities of the particles to a data file.Uses the format listed
// under the function init_files
//----------------------------------------------------------------------------
void md_sim_class::print_particle_velocities(double* vx, double* vy,
                                              int num_particles, FILE *f_out)
{
  for(int k = 0; k < num_particles; k++)
  {
    fprintf(f_out, "%.3f␣", vx[k]);
  }
  fprintf(f_out,"\n");
  for(int k = 0; k < num_particles; k++)
  {
    fprintf(f_out, "%.3f ", vy[k]);
  }
  fprintf(f_out, "\n");
}



//----------------------------------------------------------------------------
// progress_report
//
// This function simply updates the user with what percentage the
// simulation is complete
//----------------------------------------------------------------------------
void md_sim_class::progress_report(int i, int num_time_steps)
{
  double digits = 0.1;
  //if(i % ((num_time_steps-1)/100) < digits)
    printf("\r%.3f %% Complete", (i*100.0/num_time_steps));
}


//----------------------------------------------------------------------------
// random_velocity_update
//
// This function updates velocity in a distribution based on the dust temp
// This comes from Langevin Dynamics based on the paper by Lemkuhler and
// Matthews (2012) and the BAOAB method
```

```
//------------------------------------------------------------------------
double md_sim_class::random_velocity_update(double v,double time_step,
  double drag)
{
    double norm = dist2(generator);
    double c1 = exp((-drag/m)*time_step);
    double c2 = sqrt(1-c1*c1)*sqrt(kb*md_temp/m);
    double v_new = c1*v + norm*c2;
    return v_new;
}

//------------------------------------------------------------------------
// A simple function to generate a test set of data based on the input data
//------------------------------------------------------------------------
struct md_sim_class::vector md_sim_class::test_gen(int num_particles,
  double f_drive)
{
  md_sim_class::vector v;
  v.x = f_drive/drag_coef + distribution(generator);
  v.y = distribution(generator);
  return v;
}

//------------------------------------------------------------------------
// update_arrays
//
//     This next loop is to (after all caculations) move the charges
// to their new position for the calculation in the next time step
```

```
//---------------------------------------------------------------------------
void md_sim_class::update_arrays(double* x_old, double* y_old,
                                 double* vx_old,double* vy_old,
                                 double* x_new, double* y_new,
                                 double* vx_new, double* vy_new,
                                 int num_particles)
{
  for(int k = 0; k < num_particles; k++)
  {
    vx_old[k] = vx_new[k];
    x_old[k] = x_new[k];
    vy_old[k] = vy_new[k];
    y_old[k] = y_new[k];
  }
}

int md_sim_class::update_count(double* x_old,double* x_new, double* y_new,
                               double* vx_new, double* vy_new,
                               int num_particles,
                               FILE *f_count, FILE *f_vx, FILE *f_vy,
                               int cross_count, double box_size)
{
  for(int k = 0; k < num_particles; k++)
  {
    if(1)                                          //Use all particles
    //if(x_old[k] > (box_size - center_region)/2
    //  && x_old[k] < (box_size + center_region)/2)
    //Use particles in center region
```

```
      //if(x_old[k] < box_size/2 && x_new[k] > box_size/2 && vx_new[k] > 0)
      {
        fprintf(f_count, ”%.4f␣”, y_new[k]/lambda);
        fprintf(f_vx, ”%.4f␣”, vx_new[k]/lambda);
        fprintf(f_vy, ”%.4f␣”, vy_new[k]/lambda);
        cross_count = cross_count + 1;
      }
    }
    return cross_count;
}

//-------------------------------------------------------------------------
// rk4
//
// Runge-Kutta 4th order, this function integrates the rate (force/m) equation
// to give the velocities at the current time step
//-------------------------------------------------------------------------
struct md_sim_class::vector md_sim_class::rk4(double (*fx)(double, double,
    double, double *, double *, int, int, double),
                                        double (*fy)(double, double, double, do
    double *, int, int, double),
                                        double dt, double t, double vx, double
    double *qy, int k, int num_particles, double box_size)
{
        struct vector v1;
    //double norm = dist2(generator);
    //double const_D = 2*m*drag_coef*kb*md_temp/dt;
        double
    k1 = fx(t, vx, vy, qx, qy, k, num_particles, box_size),
                l1 = fy(t, vx, vy, qx, qy, k, num_particles, box_size),

    k2 = fx(t + dt / 2, vx + k1 * dt / 2, vy + l1 * dt / 2, qx, qy,
      k, num_particles, box_size),
                l2 = fy(t + dt / 2, vx + k1 * dt / 2, vy + l1 * dt / 2, qx, qy,
      k, num_particles, box_size),

        k3 = fx(t + dt / 2, vx + k2 * dt / 2, vy + l2 * dt / 2, qx, qy,
      k, num_particles, box_size),
                l3 = fy(t + dt / 2, vx + k2 * dt / 2, vy + l2 * dt / 2, qx, qy,
      k, num_particles, box_size),

        k4 = fx(t + dt, vx + dt * k3, vy + dt * l3, qx, qy, k, num_particles,
      box_size),
                l4 = fy(t + dt, vx + dt * k3, vy + dt * l3, qx, qy, k, num_part
      box_size),

        kf = vx + dt * (k1 + 2 * k2 + 2 * k3 + k4) / 6,
                lf = vy + dt * (l1 + 2 * l2 + 2 * l3 + l4) / 6;
                v1.x = kf;
                v1.y = lf;
        return v1;
}
```

A.3 md_sim_class.h

The code for md_sim_class.h

```
//=============================================================================
// This code is a simple molecular dynamic simulation for use with dusty
// plasmas. It is mathematically implemented using the Runge-Kutta 4 or
// RK4 method. This is the CLASS!
//
// Written by: Dylan Funk at Auburn University, 2022
//=============================================================================

#ifndef _MD_SIM_CLASS_H
#define _MD_SIM_CLASS_H

#include "COPLA-Common/co_options.h"
#include <stddef.h>
// needed for NULL
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>

class md_sim_class {

  public:

  struct simulation_parameters{
        _UINT_ num_particles;
```

```
   } simulation_parameters;

struct vector{
     double x, y;
   };

private:

   struct particles{
     double x, y, gx,gy;
   };


public:

md_sim_class();                          // Constructor
~md_sim_class();                         // Destructor

void   init_files(int num_time_steps, int num_particles, FILE *f_vals, FILE *
void   bound_check(double* x,double* y,double* vx,double* vy,double box_size)
void   dens_count(int** grid,double dt,int time_step,double* dens, FILE *f_gr
struct vector electric_field(double x,double y,double box_size);
void   final_update(int cross_count,char* fname_option,char* fname_buffer,int
void   grid_zero(int** grid, double box_size);
int**  grid_count(double* x, double* y, double dt, int time_step, int num_pa
void   load_parameters(double* x_old,double* y_old,double* vx_old,double* vy_
void   init_parameters(double* x_old,double* y_old,double* vx_old,double* vy_
void   print_particle_positions(double* x,double* y,int num_particles,FILE* f
  void    print_average_velocities(double* vx,int num_particles, FILE* f_out);
  void    print_particle_velocities(double* vx,double* vy,int num_particles,FILE
  void    progress_report(int i, int num_time_steps);
  double  random_velocity_update(double v, double time_step, double drag);
  struct vector test_gen(int num_particles, double f_drive);
  void    update_arrays(double* x_old,double* y_old,double* vx_old,double* vy_ol
  int     update_count(double* x_old, double* x_new, double* y_new, double* vx_n
  struct vector rk4(double (*fx)(double,double,double,double*,double*,int,int,d
  //static double rate_x(double t,double vx,double vy,double *qx,double *qy,dou
  //static double rate_y(double t,double vx,double vy,double *qx,double *qy,int
  int     initialize();                    // initialize simulation parameters

  private:



};

#endif
```

A.4 md_sim_ext.h

```
//==================================================================================
// This includes the external constants included in the md simulation Written
// by Dylan Funk
//==================================================================================

#ifndef _MD_SIM_EXT_H
#define _MD_SIM_EXT_H

#include "COPLA-Common/co_options.h"
#include <stddef.h>                  // needed for NULL
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>

//------------------------------------------------------------------------------------
//Default globals for the program including physical parameters
//------------------------------------------------------------------------------------

double md_sim_ext_g = 9.8;
double electron_charge = 1.6E-19;
double kappa = 9E9;                  //  1/4*pi*epsilon_0 ~ 9*10^9

//------------------------------------------------------------------------------------
// to be converted to commandline parameters
//------------------------------------------------------------------------------------
```

```

double x_y_ratio = 1;
double lambda = 250E-6;                                        // Debye Length
double E_max_x = 0;                  // Electric field found on edge of containme
double E_max_y = 100000;                // in units of V/m
double B_field = 4;//1;                 // External magnetic field in z direction
double dust_mass_density = 1510;    // in kg/m^3 (1510 for melamine formaldehyde
double dust_radius = 1.7E-6;         // [m]
double q_val = 8000;    // Dust particle charge value [C]
double m = (4*3.14/3)*dust_mass_density*dust_radius*dust_radius*dust_radius;
    // m ~ 3.1E-14 Dust particle mass in kg
double kb = 1.381E-23;               //Boltzmann constant
double f_drive = 100*2.64E-13;

double md_temp = 1000;
double n_ions = 5E14;               // ion number density (units = m^-3)
double m_ions = 6.63E-26;           // ion mass (assumed argon) in kg
double T_ions = 0.026;              // ion temp in eV (T~300 K)
double v_ave_ions = sqrt(8*(T_ions*electron_charge)/(3.14*m_ions));
  // average ion speed (note AVERAGE not rms)
  // [NOTE: T_ions*electron_charge is a conversion to joules]
double m_neutrals = m_ions;
double P_neutrals = 30;             // in Pascals, typically about 0.3 mBar
double T_neutrals = 300;            // In kelvin

double n_neutrals = P_neutrals/(kb*T_neutrals);       // ~7.24E21 m^-3
double v_ave_neutrals = sqrt(8*(kb*T_neutrals)/(3.14*m_neutrals));

double alpha_eps = 1 + 9*3.14159/64;  // See eps 1924
double drag_coef = 0.2*alpha_eps*(4*3.14/3)*n_neutrals*m_neutrals*v_ave_neutral
  // F_epstein_drag = drag_coef*v_particles, from epstein paper 1924

double start_velocity = f_drive/drag_coef;
double size = 80;
double BOX_WIDTH = size*lambda, BOX_HEIGHT = size*lambda;
double center_region = size*lambda/4;
  // Define parameters of box to plot
//double E_bound_x = BOX_WIDTH*0.8, E_bound_y = BOX_HEIGHT*0.8;
  // Soft boundary limits where E becomes greater than zero
double GRID_SIZE = 2*lambda;

int plot_interval = 20;          // number of time steps between plotting
int save_interval = 20;
int plot_count = 0;
int flag_start_parameter = 0;
int flag_save_paramter = 0;

//const int num_grid_x = (int)floor(BOX_WIDTH/GRID_SIZE);
//const int num_grid_y = (int)floor(BOX_HEIGHT/GRID_SIZE);
  //Grid spacing

double f_external_y = 0;//-3E-15;

#endif
```

Appendix B

Analytical Code

This is the analytical code as discussed in chapter 4. These have been broken up into two general analytical codes (both written in python), one for low coupling and one for high coupling. These are the main pieces of code which can be used to analyze both positional and analytical data. Further analytical pieces are excluded from this section

B.1 Low coupling analytical code

```
# -*- coding: utf-8 -*-
"""
Created on Sun Aug 29 20:17:52 2021

@author: dil_e
"""

import matplotlib.pyplot as plt
import numpy as np
import gc
import pickle
import scipy
from scipy import optimize
import matplotlib.colors as colors
import scipy.special as sp
#import ffmpy
#import os

debl = 250E-6
#Boltzmann Constant
kb = 1.381E-23
mass = 3.105E-14
f_drive = 2.64E-13
drag_coef = 3.3417023489E-12

#hist parameters, box size, left bound and right bound
box_width_y = 0.1*250E-6
lb1 = 40*250E-6
rb1 = 200*250E-6

lb2 = 120*250E-6
rb2 = 150*250E-6

#This creates an array that will hold our histogram boxes
val_range = np.arange(lb1,rb1,box_width_y)
val_range2 = np.arange(lb2,rb2,box_width_y)

plot_start = 5
filename = input("Please Enter Filename: ")

#variable for holding our plot choice
plot_flag = int(input("Choose a plot fit (0 for linear, 1 for exponential): "))
```

```


#==============================================================================
#   func_exp is our exponential assumption for exp fit later
#==============================================================================
def func_exp(x, a, b, c):
    return a * np.exp(-b * x) + c


#==============================================================================
#   func_exp2 is our exponential assumption for exp fit later
#==============================================================================
def func_exp2(x, a, b, c):
    return a * np.exp(b * x) + c

#==============================================================================
#   func_test is our test fit function
#==============================================================================
def func_test(x, a, b, c):
    return a / x **2

def func_test2(x,a,b,c):
    return a*(x+b)**2 + c

#==============================================================================
#   func_residual is to test our residuals
#==============================================================================
def func_residuals(x, a, b, c):
    return a*x*sp.jve(0,b/x)+c

#======================================================================
#   plot_func is a simple plotter function to generalize all of our plots
#======================================================================
def plot_func(x_vals, y_vals, size = 100,fontsize = 40, *args, **kwargs):
    title = kwargs.get('title', None)
    x_title = kwargs.get('x_title', None)
    y_title = kwargs.get('y_title', None)
    y2 = kwargs.get('y2', None)
    y3 = kwargs.get('y3', None)
    err1 = kwargs.get('err1', None)
    err2 = kwargs.get('err2', None)
    legend = kwargs.get('legend', None)
    xlims = kwargs.get('xlims', None)
    ylims = kwargs.get('ylims', None)

    #   Create the figure that will hold our plot and choose its size
    fig = plt.figure(figsize=(20.0,20.0))

    #   Change the size of the number labels (ticks)
    ax = fig.add_subplot(111)
    for tick in ax.xaxis.get_major_ticks():
        tick.label.set_fontsize(40)
    for tick in ax.yaxis.get_major_ticks():
        tick.label.set_fontsize(40)

    text = ax.yaxis.get_offset_text()
    text.set_size(40)

    #   Change plot limits
    if xlims is not None:
        ax.set_xlim(xlims)
    if ylims is not None:
        ax.set_ylim(ylims)

    #   Create scatter plot
    plt.scatter(x_vals, y_vals, color = 'red', edgecolors = 'black', s = size)
    if y2 is not None:
        plt.plot(x_vals,y2, '-', color = 'blue', linewidth = 5)
    if y3 is not None:
        plt.plot(x_vals,y3, '-', color = 'green', linewidth = 10)
    if err1 is not None:
        plt.errorbar(x_vals,y_vals,yerr=err1,ls="none",color="black")
    if err2 is not None:
        plt.errorbar(x_vals,y2,yerr=err2,ls="none",color="black")


    #   Add titles/Label axes/Legend
    plt.title(title,fontsize = 40)
    plt.ylabel(y_title,fontsize = 40)
    plt.xlabel(x_title,fontsize = 40)
    ax.ticklabel_format(style='sci',axis='y',scilimits=(-3,3))

    if legend is not None:
        plt.legend(legend, loc='best',fontsize = 30)

    #   Make plot
    plt.show()

#   As above, this reads in values and turns it into an array of floats
vx_in = open("C:\\Users\\dil_e\\Dropbox\\Physics\\Research\\MD_Sim\\md_sim.c\\D
num_vals = int(vx_in.readline())
data_str_vx = vx_in.read().split('␣')
data_vx_orig = list(map(float,data_str_vx))
data_vx = data_vx_orig[int(len(data_vx_orig)/10):len(data_vx_orig)]

#Convert data_vx from debye lengths per second to mm/s
data_vx = np.multiply(data_vx,250E-6)

#   Open file and read the values
pos_in = open("C:\\Users\\dil_e\\Dropbox\\Physics\\Research\\MD_Sim\\md_sim.c\\
B_field = float(pos_in.readline())
q_actual = float(pos_in.readline())
T = float(pos_in.readline())
size = float(pos_in.readline())
time_step = float(pos_in.readline())
num_vals = int(pos_in.readline())


#   Left_bount = lb and right bound = rb. These can be manipulated to include
# or exclude outlier data
lb = 0
rb = len(data_vx) - 2
```

```

#   Only using data within our bounds, the average and standard deviation are
#calculated. This is then printed
data_vx_avg = np.average(data_vx[lb:rb])
data_vx_stdev = np.std(data_vx[lb:rb])
print("\n\n\n\nAverage:␣%f\nCalculated␣Standard␣Deviation␣(Data):␣%f\nCalculate
Temp_calc = (data_vx_stdev)**2*mass/kb

#INFILE = open("D:\\Dropbox\\Physics\\Research\\MD_Sim\\md_sim.c\\values.txt")
INFILE = open("C:\\Users\\dil_e\\Dropbox\\Physics\\Research\\MD_Sim\\md_sim.c\\
num_parts = int(INFILE.readline())
num_steps = int(INFILE.readline())
plot_width = int(INFILE.readline())
plot_height = float(INFILE.readline())
INFILE.close()
num_steps = num_steps - plot_start
t_total = num_steps*time_step

#This is a bit clunky but np.loadtxt is great for reading in arrays and it resi
data = np.loadtxt("C:\\Users\\dil_e\\Dropbox\\Physics\\Research\\MD_Sim\\md_sim
interval = 1

#Convert data_vx from debye lengths to m
data = np.multiply(data,250E-6)


i = 0

```

```
#   Exclude the last element due to size mismatch. This is in case smoothing
# is used and this has one less element
val_mod = val_range[:-1]

#   Array full_hist_data is goign to be our histogram array at each timestep so
# have a row for each element and a column for each y position (hist data)
hist_data = np.zeros((int(num_steps/interval) - 2, len(val_mod)))
hist_datax = np.zeros((int(num_steps/interval) - 2, len(val_mod)))

#   This will be our array of all gradients at each timestep. this can be used
# to find our charge at each timestep
grad_array = np.zeros(int(num_steps/interval)-2)


#   This odd looking function uses python's numpy "slicing" feature. This is
# saying data[1::2] is basically starting at element 1 (the second element)
# we would like to use every 2nd item. data[0::3] would start at element 0 and
# use every third element etc.
data_y = data[1::2,:]
data_x = data[0::2,:]

#   The range of this function starts at the end of our first summing interval
# and continues until the total sim time minus the time interval
for x in range(plot_start,num_steps+plot_start-3*int(interval)):

    #Loop through the array until reaching x == time interval
    if x%int(interval) == 0:

        # This says that between x and x + interval make a histogram of all y
        # positional data.
        hist_data[i,:], bin_data = np.histogram(data_y[x:x+int(interval),:], bin
        hist_datax[i,:],bin_datax = np.histogram(data_x[x:x+int(interval),:],bi
        # This picks the current-timestep element from our histogram array
        # that will be used for our linear data fit
        hist_data_i = hist_data[i]

        # this is our linear fit for the current timestep, i
        if plot_flag == 0:
            polyfit_i = np.polyfit(val_mod,hist_data_i,1)
            grad_array[i] = polyfit_i[0]
        if plot_flag == 1:
            expfit_i = scipy.optimize.curve_fit(func_exp,val_mod,hist_data_i)
            grad_array[i] = expfit_i[0][1]
        i+=1

def heatmap2d(arr: np.ndarray):
    fig, ax = plt.subplots(figsize = (20,20))
    for tick in ax.xaxis.get_major_ticks():
        tick.label.set_fontsize(40)
    for tick in ax.yaxis.get_major_ticks():
        tick.label.set_fontsize(40)
    im = ax.imshow(arr.T, cmap='plasma',origin='lower', aspect='auto')
    cbar = fig.colorbar(im)
    cbar.ax.tick_params(labelsize=10)

    ax.ticklabel_format(style='sci',axis='y',scilimits=(-3,3))
    plt.title("Particle Density vs Time",fontsize = 40)
    plt.ylabel(r"Particle y position (Debye Lengths)",fontsize = 40)
    plt.xlabel("Timestep",fontsize = 40)
    plt.show()

def colorbarfmt(x, pos):
    a, b = '{:.2e}'.format(x).split('e')
    b = int(b)
    return r'${} \times 10^{{{}}}$'.format(a, b)

def colormesh2d(arr: np.ndarray):
    fig, ax = plt.subplots(figsize = (20,20))
    for tick in ax.xaxis.get_major_ticks():
        tick.label.set_fontsize(40)
    for tick in ax.yaxis.get_major_ticks():
        tick.label.set_fontsize(40)
    plot_range = np.arange(interval*time_step,t_total - 3*interval*time_step,ti
    im = ax.pcolormesh(plot_range,val_mod,arr[:int(len(plot_range))].T, cmap='p
    cbar = fig.colorbar(im)
    cbar.ax.tick_params(labelsize=40)
    cbar.set_label(r"Particle density", size=40)
    #plt.title("Particle Density over time",fontsize = 40)
    plt.ylabel(r"Particle y position [m]",fontsize = 40)
    plt.xlabel("Time (s)",fontsize = 40)

    ax.ticklabel_format(style='sci',axis='y',scilimits=(-3,3))
    #plt.xlim(0,15)
    plt.show()
```

```
#heatmap2d(hist_data)

colormesh2d(np.multiply(hist_data ,1000))
colormesh2d(np.multiply(hist_datax ,1000))
ux = np.average(data_vx)
q_calc = kb*Temp_calc*(grad_array)/(B_field*ux)/(-1.6E-19)


time_arr = np.arange(0,len(q_calc))


sum_grad2 = np.polyfit(val_mod[int(len(val_mod)/2):],hist_data.mean(axis=0)[int
#=====================================================
#     Functions to fit to both linear and exponential
#=====================================================
sum_grad = np.polyfit(val_mod,hist_data.mean(axis=0),1)
y_linear = sum_grad[0]*val_mod + sum_grad[1]
if plot_flag == 0:
    fit = sum_grad
    y_plot = sum_grad[0]*val_mod + sum_grad[1]
    y_plot2 = sum_grad2[0]*val_mod[int(len(val_mod)/2):] + sum_grad2[1]
    plot_legend = [r'Equation␣of␣Fit:␣$%.3f␣y␣+␣%.3f␣$' % (sum_grad[0],sum_grad
    fit_calc = sum_grad[0]
if plot_flag == 1:
    fit = scipy.optimize.curve_fit(func_exp,val_mod,hist_data.mean(axis=0))
    y_exp = func_exp(val_mod,fit[0][0],fit[0][1],fit[0][2])
    #q_fit = scipy.optimize.curve_fit(func_exp,time_arr[:-1]*2*interval*0.005,-
    y_plot = y_exp
```

```python
    y_plot2 = sum_grad2[0]*val_mod[int(len(val_mod)/2):] + sum_grad2[1]
    fit_calc = fit[0][1]
    plot_legend = [r'Number of particles at each y position',r'Equation of Fit:



#=====================================================
#    Plot of density vs position
#=====================================================
plot_func(val_mod,hist_data.mean(axis=0),
          title="N(y)",x_title="y-position",y_title=r"Number of particles",
          y2 = y_plot,
          #err1 = np.sqrt(hist_data.mean(axis=0)),
          legend = plot_legend)

residuals = hist_data.mean(axis=0) - y_plot
"""
residual_fit = scipy.optimize.curve_fit(func_residuals,val_mod,residuals)
y_residual = func_residuals(val_mod, residual_fit[0][0], residual_fit[0][1], re
"""
#=====================================================
#    Plot of density residuals
#=====================================================
plot_func(val_mod,residuals,
          title="Residual Fit",x_title="y-position [m]",y_title=r"$N_{fit}-N_{d
          #y2 = y_residual)

time_arr = np.arange(0,len(q_calc))
```

```

#========================================================
#     Plot of charge value over time
#========================================================
plot_func(time_arr[:-1]*2*interval*0.005,-q_calc[:-1],
          title="Calculated Charge Value over time",x_title="Time (s)",y_title=



#image_folder = 'D:\\Dropbox\\Physics\\Research\\MD_Sim\\md_sim.c\\plots'
image_folder = 'C:\\Users\\dil_e\\Dropbox\\Physics\\Research\\MD_Sim\\md_sim.c\
q1 = kb*Temp_calc*(-fit_calc)/(B_field*ux)/(-1.6E-19)
#print("\n\n\n\nN-fit: %f y + %f" % (polyfit_i[0], polyfit_i[1]))
print("q_avg = %.5f" % np.average(-q_calc))
print("q_final = %.5f" % q1)
print("q_actual = %.5f" % (q_actual))
#ffmpeg -f image2 -r 10 -i plot_%05d.png -vcodec mpeg4 -y movie.mp4
gc.collect()

q2 = round(q1,6)
list_data = [filename, q_actual, num_parts, B_field, T, size, time_step, num_va

with open("C:\\Users\\dil_e\\Dropbox\\Physics\\Research\\MD_Sim\\md_sim.c\\Data
    for line in list_data:
        f_data.write(str(line))
        f_data.write('\n')
```

B.2 High coupling analytical code

```
# -*- coding: utf-8 -*-
"""
Nearest Neighbor Calcs

@author: dil_e
"""

import matplotlib as mpl
import matplotlib.cm as cm
import matplotlib.pyplot as plt
import numpy as np
import math
from scipy.spatial import Voronoi, voronoi_plot_2d


plot_size = 40.0
font_size = 20

e0 = 8.85E-12
lambdaD = 250E-6

kb = 1.381E-23
mass = 3.105E-14

box_width_y = 0.1
lb1 = 1
rb1 = 5
```

```
#This creates an array that will hold our histogram boxes
val_range = np.arange(lb1,rb1,box_width_y)
val_mod = val_range[:-1]

# This rejects large outliers (such a those in corners or on edge without 6
# nearest neighbors which form a hexagon)
def reject_outliers(data, m = 4.):
    d = np.abs(data - np.median(data))
    mdev = np.median(d)
    s = d/mdev if mdev else 0.
    return data[s<m]

# Same as above but for more complicated data sets(aka those with multiple
# dimensions)
def remove_outlier_2D(data, m = 4):
    d = np.abs(data[:,2] - np.median(data[:,2]))
    mdev = np.median(d)
    s = d/mdev if mdev else 0.
    return data[:][s<m]

def floor_round(x):
    return int(math.floor(x / 100.0) * 100)

def ceil_round(x):
    return int(math.ceil(x / 100.0) * 100)

def hist_plotter(hist_array):
    upper_bound = ceil_round(max(hist_array))
    lower_bound = floor_round(min(hist_array))
    dif = upper_bound - lower_bound
    spacing = int(dif/100)
    bins_def = np.arange(lower_bound,upper_bound, spacing)
    plt.hist(hist_array,bins = bins_def)

#==========================================================================
#   plot_func is a simple plotter function to generalize all of our plots
#==========================================================================
def plot_func(x_vals, y_vals,title, x_title, y_title, size = 100,fontsize = 40,
    y2 = kwargs.get('y2', None)
    y3 = kwargs.get('y3', None)
    err1 = kwargs.get('err1', None)
    err2 = kwargs.get('err2', None)
    legend = kwargs.get('legend', None)
    xlims = kwargs.get('xlim', None)
    ylims = kwargs.get('ylim', None)

    #   Create the figure that will hold our plot and choose its size
    fig = plt.figure(figsize=(20.0,20.0))

    #   Change the size of the number labels (ticks)
    ax = fig.add_subplot(111)
    for tick in ax.xaxis.get_major_ticks():
        tick.label.set_fontsize(40)
    for tick in ax.yaxis.get_major_ticks():
        tick.label.set_fontsize(40)

    text = ax.yaxis.get_offset_text()
    text.set_size(40)

    #   Create scatter plot
    plt.scatter(x_vals, y_vals, color = 'red', edgecolors = 'black', s = size)
    if y2 is not None:
        plt.plot(x_vals,y2, '-', color = 'blue', linewidth = 10)
    if y3 is not None:
        plt.plot(x_vals,y3, '-', color = 'green', linewidth = 10)
    if err1 is not None:
        plt.errorbar(x_vals,y_vals,yerr=err1,ls="none",color="black")
    if err2 is not None:
        plt.errorbar(x_vals,y2,yerr=err2,ls="none",color="black")

    #   Change plot limits
    if xlims is not None:
        ax.set_xlim(xlims)
    if ylims is not None:
        ax.set_ylim(ylims)

    #   Add titles/Label axes/Legend
    plt.title(title,fontsize = 40)
    plt.ylabel(y_title,fontsize = 40)
    plt.xlabel(x_title,fontsize = 40)
    ax.ticklabel_format(style='sci',axis='y',scilimits=(-3,3))

    if legend is not None:
        plt.legend(legend, loc='best',fontsize = 30)

    #    Make plot
    plt.show()

#   This function creates a voronoi plot for our
def crystal_voronoi(d2D, qj, q_act):
    # generate Voronoi tessellation
    vor = Voronoi(d2D)

    q = np.abs(qj/np.sqrt(2))
    numerator = q_act-q
    denominator = q_act
    qdiff = np.divide(numerator,denominator)
    qdiff_reject = reject_outliers(qdiff)

    # find min/max values for normalization
    minima = min(qdiff_reject)
    maxima = max(qdiff_reject)*2

    # normalize chosen colormap
    norm = mpl.colors.Normalize(vmin=minima, vmax=maxima, clip=True)
    mapper = cm.ScalarMappable(norm=norm, cmap=cm.Blues_r)

    plt.rcParams['figure.figsize'] = [25,20]
    # plot Voronoi diagram, and fill finite regions with color mapped from spee
    voronoi_plot_2d(vor, show_points=True, show_vertices=False, s=1)
    for r in range(len(vor.point_region)):
        region = vor.regions[vor.point_region[r]]
```

```python
            if not -1 in region:
                polygon = [vor.vertices[i] for i in region]
                plt.fill(*zip(*polygon), color=mapper.to_rgba(qdiff[r]))
    plt.show()

# Split array into a 2D array of particle positions for analysis
def splitter(array,timestep):
    xvals = array[timestep,:]
    yvals = array[timestep+1,:]
    xyarray = [xvals,yvals]
    array2D = np.zeros([len(xyarray[0]),2])
    for i in range(0,len(xyarray[0])):
        array2D[i] = [xyarray[0][i],xyarray[1][i]]
    return array2D

# Calculate the Euclidean distance between two vectors
def euclidean_distance(row1,row2):
    separation = (row1 - row2)**2
    distance = 0.0
    distance = np.sqrt(separation[0]+separation[1])
    return distance

# Locate the most similar neighbors
def get_neighbors(train, test_row, num_neighbors):
        distances = list()
        for train_row in train:
                dist = euclidean_distance(test_row, train_row)
                distances.append((train_row, dist))
        distances.sort(key=lambda tup: tup[1])
        neighbors = list()
        for i in range(num_neighbors):
                neighbors.append(distances[i][0])
        return neighbors

# Plot nearest neighbors to the chosen particle
def plot_neighbors(dataArr, nArr, timestep):
    fig = plt.figure(figsize=(plot_size,plot_size))
    ax = fig.add_subplot(111)
    for tick in ax.xaxis.get_major_ticks():
        tick.label.set_fontsize(font_size*2)
    for tick in ax.yaxis.get_major_ticks():
        tick.label.set_fontsize(font_size*2)
    ax.scatter(dataArr[timestep,:],dataArr[timestep+1,:],color = 'gray', edgecc
    ax.scatter(nArr[0][0],nArr[0][1],color = 'blue', edgecolors='black',s=320)
    for i in range(1,len(nArr)):
        ax.scatter(nArr[i][0],nArr[i][1],color = 'red', edgecolors='black',s=32
    ax.set_xlim(0.0,float(plot_width*0.25))
    ax.set_ylim(0.0,float(plot_height*0.1))
    ax.set_aspect(1.0)
    ax.set_xlabel(r'x␣position␣[mm]', fontsize = font_size*2)
    ax.set_ylabel(r'y␣position␣[mm]', fontsize = font_size*2)
    ax.set_title(r'Sample␣simulation␣frame', fontsize = font_size*2)


#   This was originally to calculate bond order, not particlularly useful here
# it turns out (not actually hexagonal crystals)
```

```
def bond_order(nArr):
    #Sort by y value to evaluate heights
    column_index = 1
    neighbors_np = np.stack(nArr,axis=0)
    sortArr = neighbors_np[neighbors_np[:,column_index].argsort()]

    #Sort to specify each particle by location and convert to meters
    up1 = sortArr[-1]/10**3
    up2 = sortArr[-2]/10**3
    mid1 = sortArr[-3]/10**3
    mid2 = sortArr[-4]/10**3
    mid3 = sortArr[-5]/10**3
    down1 = sortArr[-6]/10**3
    down2 = sortArr[-7]/10**3

    #Specify test particle
    mid = neighbors_np[0]/10**3

    xup1 = np.abs(up1[0] - mid[0])
    yup1 = np.abs(up1[1] - mid[1])
    xup2 = np.abs(up2[0] - mid[0])
    yup2 = np.abs(up2[1] - mid[1])
    xmid1 = np.abs(mid2[0] - mid[0])
    ymid1 = np.abs(mid2[1] - mid[1])
    xmid2 = np.abs(mid3[0] - mid[0])
    ymid2 = np.abs(mid3[1] - mid[1])
    xdown1 = np.abs(down1[0] - mid[0])
    ydown1 = np.abs(down1[1] - mid[1])
    xdown2 = np.abs(down2[0] - mid[0])
    ydown2 = np.abs(down2[1] - mid[1])

    theta = np.zeros(6)
    theta[0] = np.arctan(yup1/xup1)
    theta[1] = np.arctan(yup2/xup2)
    theta[2] = np.arctan(ymid1/xmid1)
    theta[3] = np.arctan(ymid2/xmid2)
    theta[4] = np.arctan(ydown1/xdown1)
    theta[5] = np.arctan(ydown2/xdown2)

    psi_i = np.zeros(6)
    psi6 = 0
    for i in range(len(theta)):
        psi_i[i] = 1/len(psi_i)*(np.exp(6j*theta[i]))
        psi6 = psi6 + psi_i[i]
    return psi6



# This function sorts each particle by height and subsequently calculates the
# charge value based on the nearest neighbors
def q_calculator(nArr,Bz, vavg, flag):
    e0 = 8.85E-12
    lambdaD = 250E-6
    fext = 3E-15

    #Sort by y value to evaluate heights
    column_index = 1
    neighbors_np = np.stack(nArr, axis=0)
    sortArr = neighbors_np[neighbors_np[:, column_index].argsort()]

    #Sort to specify each particle by location and convert to meters
    up1 = sortArr[-1]/10**3
    up2 = sortArr[-2]/10**3
    mid1 = sortArr[-3]/10**3
    mid2 = sortArr[-4]/10**3
    mid3 = sortArr[-5]/10**3
    down1 = sortArr[-6]/10**3
    down2 = sortArr[-7]/10**3

    #Specify test particle
    mid = neighbors_np[0]/10**3

    xup1 = np.abs(up1[0] - mid[0])
    yup1 = np.abs(up1[1] - mid[1])
    xup2 = np.abs(up2[0] - mid[0])
    yup2 = np.abs(up2[1] - mid[1])
    xup_avg = (xup1+xup2)/2
    yup_avg = (yup1+yup2)/2
    xmid1 = np.abs(mid1[0] - mid[0])
    ymid1 = np.abs(mid1[1] - mid[1])
    xmid2 = np.abs(mid3[0] - mid[0])
    ymid2 = np.abs(mid3[1] - mid[1])
    xdown1 = np.abs(down1[0] - mid[0])
    ydown1 = np.abs(down1[1] - mid[1])
```

```python
    xdown2 = np.abs(down2[0] - mid[0])
    ydown2 = np.abs(down2[1] - mid[1])
    xdown_avg = (xdown1+xdown2)/2
    ydown_avg = (ydown1+ydown2)/2
    rup = np.sqrt(xup_avg**2 + yup_avg**2)
    rdown = np.sqrt(xdown_avg**2 + ydown_avg**2)


    ru1 = euclidean_distance(mid2, up1)
    ru2 = euclidean_distance(mid2, up2)
    rm1 = euclidean_distance(mid2, mid1)
    rm2 = euclidean_distance(mid2, mid3)
    rd1 = euclidean_distance(mid2, down1)
    rd2 = euclidean_distance(mid2, down2)

    f1x =  (ru1 +lambdaD)*yup1*np.exp(-ru1/lambdaD)/(lambdaD*rdown*rdown*rdown)
    vx_avg = vavg/10**3
    denom1 = 2*(rdown +lambdaD)*ydown_avg*np.exp(-rdown/lambdaD)/(lambdaD*rdown
    denom2 = 2*(rup +lambdaD)*yup_avg*np.exp(-rup/lambdaD)/(lambdaD*rup*rup*rup


    if(flag == 0):
        #   This first is with driving force and lorentz force compression/sedi
        q = (4*np.pi*e0*vx_avg*Bz)/(denom1 - denom2)
    elif(flag == 1):
        #   This first is charge for external y compression/sedimentation
        # force with no driving force
        q = 2*np.sqrt(4*np.pi*e0*fext/np.abs(denom1-denom2))
    q_e = np.abs(q)/1.61E-19
    return q_e


"""-----------------------------------------------------------------------------
# This calculates the height required to reach our cutoff for coupling and only
# returns values below that height.
# Coupling coefficient = inter-particle potential energy/thermal energy
def coupling_reject(data,q_act,T_act, coupling_cutoff = 1.):
    coupling_coef_const = (q_act**2/(4*np.pi*e0))/(kb*T_act)

    #Sort by y value to evaluate heights (column index 1 = y position)
    column_index = 1
    sorted_data = data[data[:,column_index].argsort()]
    r_cutoff = np.sqrt(coupling_coef_const/coupling_cutoff)
    x_cutoff = sorted_data[len(sorted_data),0] - sorted_data[len(sorted_data),0
    y_cutoff =
    valid = [x for x in sorted_data if x < ]
    return valid
-----------------------------------------------------------------------------"""
def neighbors_dist(data):
    distData = np.zeros(len(neighbors)-1)
    for i in range(0,len(data)-1):
        distData[i] = euclidean_distance(data[0], data[i])
        i+=1
    spacing = np.average(distData)
    return spacing

```

```
#   colormesh2d is what creates our density plots , not currently being used her
def colormesh2d(arr: np.ndarray ,lb ,rb ):
    fig , ax = plt.subplots(figsize = (20,20))
    for tick in ax.xaxis.get_major_ticks():
        tick.label.set_fontsize(40)
    for tick in ax.yaxis.get_major_ticks():
        tick.label.set_fontsize(40)
    plot_range = np.arange(lb ,rb)
    im = ax.pcolormesh(plot_range ,val_mod ,arr[:int(len(plot_range))].T, cmap='p
    cbar = fig.colorbar(im)
    cbar.ax.tick_params(labelsize=40)
    cbar.set_label(r"Particle␣Counts", size=40)
    plt.title("Particle␣Density␣vs␣Time",fontsize = 40)
    plt.ylabel(r"Particle␣y␣position␣[mm]",fontsize = 40)
    plt.xlabel("Time␣(s)",fontsize = 40)
    plt.show()

def example_plot(dataArr , nArr , timestep , Bz, vavg , flag ):
    neighbors_np = np.stack(nArr ,axis=0)
    xmin = min(neighbors_np[:,0]) - 0.5
    xmax = max(neighbors_np[:,0]) + 0.5
    ymin = min(neighbors_np[:,1]) - 0.5
    ymax = max(neighbors_np[:,1]) + 0.5

    fig = plt.figure(figsize=(plot_size*.25,plot_size*.25))
    ax = fig.add_subplot(111)
    for tick in ax.xaxis.get_major_ticks():
        tick.label.set_fontsize(font_size)
    for tick in ax.yaxis.get_major_ticks():
        tick.label.set_fontsize(font_size)
    ax.scatter(dataArr[timestep,:],dataArr[timestep+1,:],color = 'gray', edgecc
    ax.scatter(nArr[0][0],nArr[0][1],color = 'blue', edgecolors='black',s=400)
    for i in range(1,len(nArr)):
        ax.scatter(nArr[i][0],nArr[i][1],color = 'red', edgecolors='black',s=40
    ax.set_xlim(xmin,xmax)
    ax.set_ylim(ymin,ymax)
    ax.set_aspect(1.0)
    ax.set_xlabel(r'x position [mm]', fontsize = font_size)
    ax.set_ylabel(r'y position [mm]', fontsize = font_size)
    ax.set_title(r'Sample Nearest Neighbors', fontsize = font_size)

    e0 = 8.85E-12
    lambdaD = 250E-6

    #Sort by y value to evaluate heights
    column_index = 1
    neighbors_np = np.stack(nArr,axis=0)
    sortArr = neighbors_np[neighbors_np[:,column_index].argsort()]

    #Sort to specify each particle by location and convert to meters
    up1 = sortArr[-1]/10**3
    up2 = sortArr[-2]/10**3
    mid1 = sortArr[-3]/10**3
    mid2 = sortArr[-4]/10**3
    mid3 = sortArr[-5]/10**3
    down1 = sortArr[-6]/10**3
    down2 = sortArr[-7]/10**3

    ru1 = euclidean_distance(mid2, up1)
    ru2 = euclidean_distance(mid2, up2)
    rm1 = euclidean_distance(mid2, mid1)
    rm2 = euclidean_distance(mid2, mid3)
    rd1 = euclidean_distance(mid2, down1)
    rd2 = euclidean_distance(mid2, down2)

    q = q_calculator(nArr,Bz,vavg,flag)
    q_act = q_actual
    diff = np.abs(q - q_act)/np.average([q,q_act])*100

    table_data = {"Center": [round(mid2[0]*10**3,5), round(mid2[1]*10**3,5), '-
                  1: [round(up1[0]*10**3,5), round(up1[1]*10**3,5), round(ru1*1
                  2: [round(up2[0]*10**3,5), round(up2[1]*10**3,5), round(ru2*1
                  3: [round(mid1[0]*10**3,5), round(mid1[1]*10**3,5), round(rm1
                  4: [round(mid3[0]*10**3,5), round(mid3[1]*10**3,5), round(rm2
                  5: [round(down1[0]*10**3,5), round(down1[1]*10**3,5), round(r
                  6: [round(down2[0]*10**3,5), round(down2[1]*10**3,5), round(r

    print ("|{:<12} | {:<12} | {:<12} | {:<12}".format('Particle','x-pos','y-po
    print("---------------------------------------------------")
    for k, v in table_data.items():
        xpos, ypos, r = v
        print ("|{:<12} | {:<12} | {:<12} | {:<12}".format(k, xpos, ypos, r))
    print("---------------------------------------------------")
    print("Q Calculated = %.5f" % q)
    print("Q Actual = %.5f" % q_act)
    print("Percent Difference = %.5f%%" % diff)


filename = input("Please Enter Filename: ")
flag_type = int(input("What type of data is this? (0 for lorentz , 1 for externa
#   As above, this reads in values and turns it into an array of floats
vx_in = open("C:\\Users\\dil_e\\Dropbox\\Physics\\Research\\MD_Sim\\md_sim.c\\D
num_vals = int(vx_in.readline())
data_str_vx = vx_in.read().split(' ')
data_vx_orig = list(map(float ,data_str_vx))
data_vx = data_vx_orig[int(len(data_vx_orig)/10):len(data_vx_orig)-1]

#Data is in debye lengths, this function converts to mm
data_vx = np.multiply(data_vx ,0.25)

#   Left_bount = lb and right bound = rb. These can be manipulated to include
#or exclude outlier data
lb = 0
rb = len(data_vx) - 2

#   Only using data within our bounds, the average and standard deviation are
#calculated. This is then printed
data_vx_avg = np.average(data_vx[lb:rb])

#   Open file and read the values
pos_in = open("C:\\Users\\dil_e\\Dropbox\\Physics\\Research\\MD_Sim\\md_sim.c\\
B = float(pos_in.readline())
```

```
q_actual = float(pos_in.readline())
T = float(pos_in.readline())
size = float(pos_in.readline())
time_step = float(pos_in.readline())
num_vals = int(pos_in.readline())

#INFILE = open("D:\\Dropbox\\Physics\\Research\\MD_Sim\\md_sim.c\\values.txt")
INFILE = open("C:\\Users\\dil_e\\Dropbox\\Physics\\Research\\MD_Sim\\md_sim.c\\
num_parts = int(INFILE.readline())
num_steps = int(INFILE.readline())
plot_width = int(INFILE.readline())
plot_height = float(INFILE.readline())

data = np.loadtxt("C:\\Users\\dil_e\\Dropbox\\Physics\\Research\\MD_Sim\\md_sim
data = np.multiply(data,0.25)

step = int(num_steps-11)
j = 0
lb = int(num_steps - 21)
rb = int(num_steps-11)
step_range = range(lb, rb, 2)
q_avg_i = np.zeros(len(step_range))
q_avg_all_i = np.zeros(len(step_range))
val_range = np.arange(5000,11000, 250)
q_hist = np.zeros(len(val_range)-1)
qtot = np.empty(0)
for step in step_range:
    data2D = splitter(data,step)
    #part_num = input("Pick a particle number (Total %d): " % len(data2D))
    qi = np.zeros(len(data[0]))
    psi6 = np.zeros(len(qi))
    space_i = np.zeros(len(data[0]))
    for i in range(0,len(data2D) - 1):
        neighbors = get_neighbors(data2D, data2D[i],7)
        qi[i] = q_calculator(neighbors, B, data_vx_avg, flag_type)
        space_i[i] = neighbors_dist(neighbors)
        #psi6[i] = bond_order(neighbors)
    qi_reject = reject_outliers(np.abs(qi), m=1)
    #neighbors = get_neighbors(data2D, data2D[40],7)
    #plot_neighbors(data, neighbors, step)
    #q_e = q_calculator(neighbors, B, data_vx_avg, flag_type)
    qtot = np.append(qtot,qi_reject)
    #crystal_voronoi(data2D,qi,q_actual/1.61E-19)
    q_avg_i[j] = np.average(np.abs(qi_reject))
    j+=1


q_avg = np.average(q_avg_i)
print("Calculated␣charge␣value:␣%.4f\n" % q_avg)
print("Actual␣charge␣value:␣%.4f\n" % (float(q_actual)))


# Sample case
neighbors = get_neighbors(data2D, data2D[60],7)
print(neighbors[1:])
plot_neighbors(data, neighbors, step)
```

```
q_e = q_calculator(neighbors, B, data_vx_avg, flag_type)
example_plot(data, neighbors, step, B, data_vx_avg, flag_type)

particle_i = np.zeros(len(data[0]))
space_i = np.zeros(len(data[0]))
q_calc_i = np.zeros(len(data[0]))
for i in range(0,len(data2D)-1):
    #   This function gets the chosen particle coordinates (particle 0) as well
    # as the 6 nearest neighbors. NOTE: 6 nearest neighbors has a few issues
    # for particles along the edge.
    neighbors = get_neighbors(data2D, data2D[i],7)

    #   chosen particle is the
    chosen_particle = neighbors[0]

    #   particle_i is an array made up of particle y-coordinates that will corr
    #to spacing and calculated charge (i.e. particle_i[10] will have correspond
    #spacing space_i[10] and q_calc_i[10])
    particle_i[i] = chosen_particle[1]

    #   space_i is n array which holds the average spacing (inter-particle dist
    # of the nearest neighbors to particle i
    space_i[i] = neighbors_dist(neighbors)

    #   Using whichever calculation is necessary, q_calc_i is the calculated
    # charge of particle i
    q_calc_i[i] = q_calculator(neighbors, B, data_vx_avg,flag_type)

```

```

#   This is a function which calculates coupling coefficient for each particle
# based on the input charge and spacing that was calculated
coupling_i_actual = np.divide((np.power(q_actual*1.61E-19,2)/(4*np.pi*e0*kb*T)*


#   This is a function which calculates coupling coefficient for each particle
# based on the input charge and spacing that was calculated
coupling_i = np.divide((np.power(q_calc_i[:-1]*1.61E-19,2)/(4*np.pi*e0*kb*T)*np

#   combined simply creates an array with four columns, particle_i, space_i, ch
# coupling_i, {NOTE: the final value tends to be blank so i remove it with
# the [:-1] which includes all values but the last one}
combined = np.vstack((particle_i[:-1],space_i[:-1],q_calc_i[:-1],coupling_i)).T

#   this function removes outliers from the combined array by the calculated ch
# The purpose of this is to remove those particles near the edge as they will
# have less than 6 valid nearest neighbors and therefore will not be usable
combined_rejected = remove_outlier_2D(combined,1)

#   This function does the same calculation but without the edge particles
coupling_i_rejected = np.divide((np.power(combined_rejected[:,2],2)/(4*np.pi*e0

#------------------------------------------------------------------------
# Plots:
#   1) spacing vs y position
#   2) coupling coefficient vs y position
#   3) Coupling coefficient vs charge (% difference) (NOTE: not done yet)
```

```
#-----------------------------------------------------------------------
plot_func(particle_i[:-1],space_i[:-1],
          "Spacing vs y position","Particle y-position [mm]",
          r"spacing (mm)", xlims = [lb1,rb1])

plot_func(combined_rejected[:-1,0],combined_rejected[:-1,1],
          "Spacing vs y position (filtered)","Particle y-position [mm]",
          r"spacing (mm)", xlims = [lb1,rb1])

#NOTE: Removed as some outliers are too large and throw off scale
#plot_func(particle_i[:-1],coupling_i,
#           "Coupling parameter vs y position","Particle y position [mm]",
#           r"Coupling Parameter", xlims = [lb1,rb1])

plot_func(combined_rejected[:,0],coupling_i_rejected,
         "Coupling parameter vs position [mm]","Particle y-position [mm]",
         r"Coupling Parameter")

plot_func(combined_rejected[:,2],coupling_i_rejected,
         "Coupling parameter vs Q_calculated [e]","Q_calculated [e]",
         r"Coupling Parameter")

#plot_func(combined_rejected[:,0],combined_rejected[:,2],
#           "Particle Charge vs y position","Particle y position [mm]",
#           r"Particle Charge", xlims = [lb1,rb1])

#plot_func(step_range, q_avg_i,
#           "Particle Charge vs y position","Particle y position [mm]",
#               r"Particle Charge", xlims = [lb1,rb1])

#hist_plotter(qtot)
```

Appendix C

Experimental Diagrams

In this section are included many diagrams which have been developed for the experiment proposed in chapter 5 of this dissertation.

C.1 Circuit Development

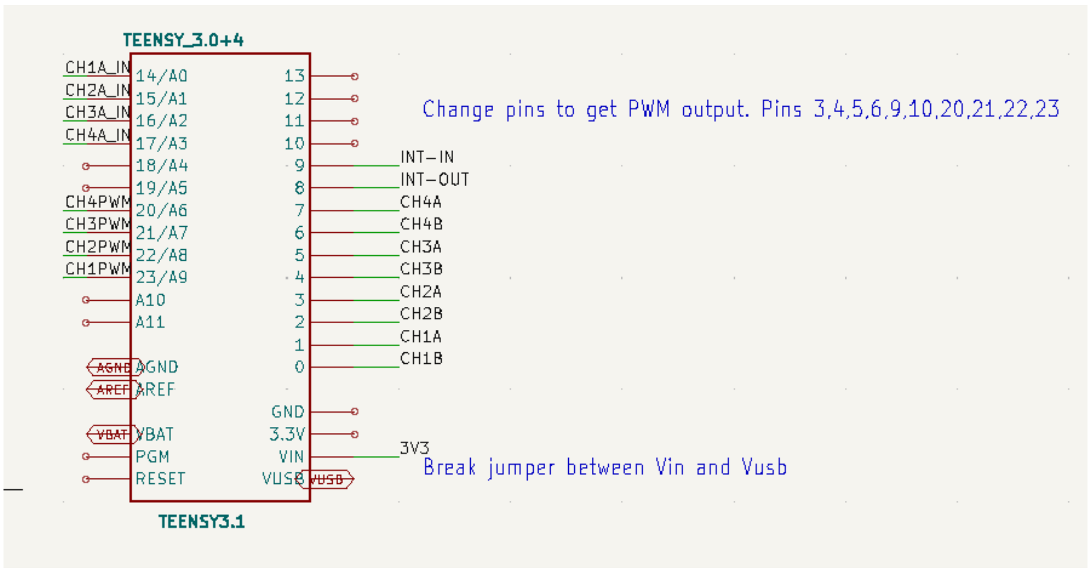


Figure C.1: Circuit Diagram for Teensy/Arduino Board

C.2 Arduino Code

```
const int syncPin = 52;
const int clkPin = 50;
```

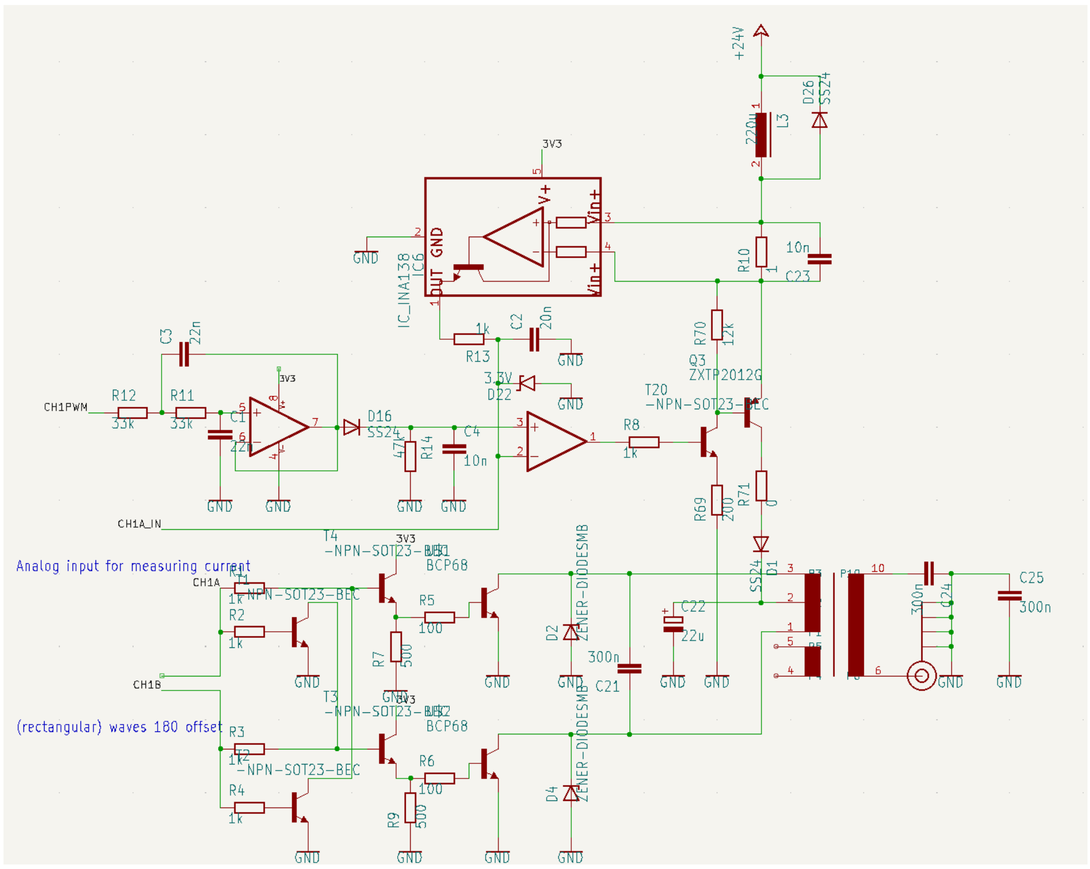


Figure C.2: Circuit diagram for high voltage generator

```
const int dPin0 = 34;
const int dPin1 = 36;
const int dPin2 = 38;
const int dPin3 = 40;
const int dPin4 = 42;
const int dPin5 = 44;
const int dPin6 = 46;
const int dPin7 = 47;

int da0 = 1023;
char dArray1[10];

//Note: Be sure to write on low, because it reads on high
```

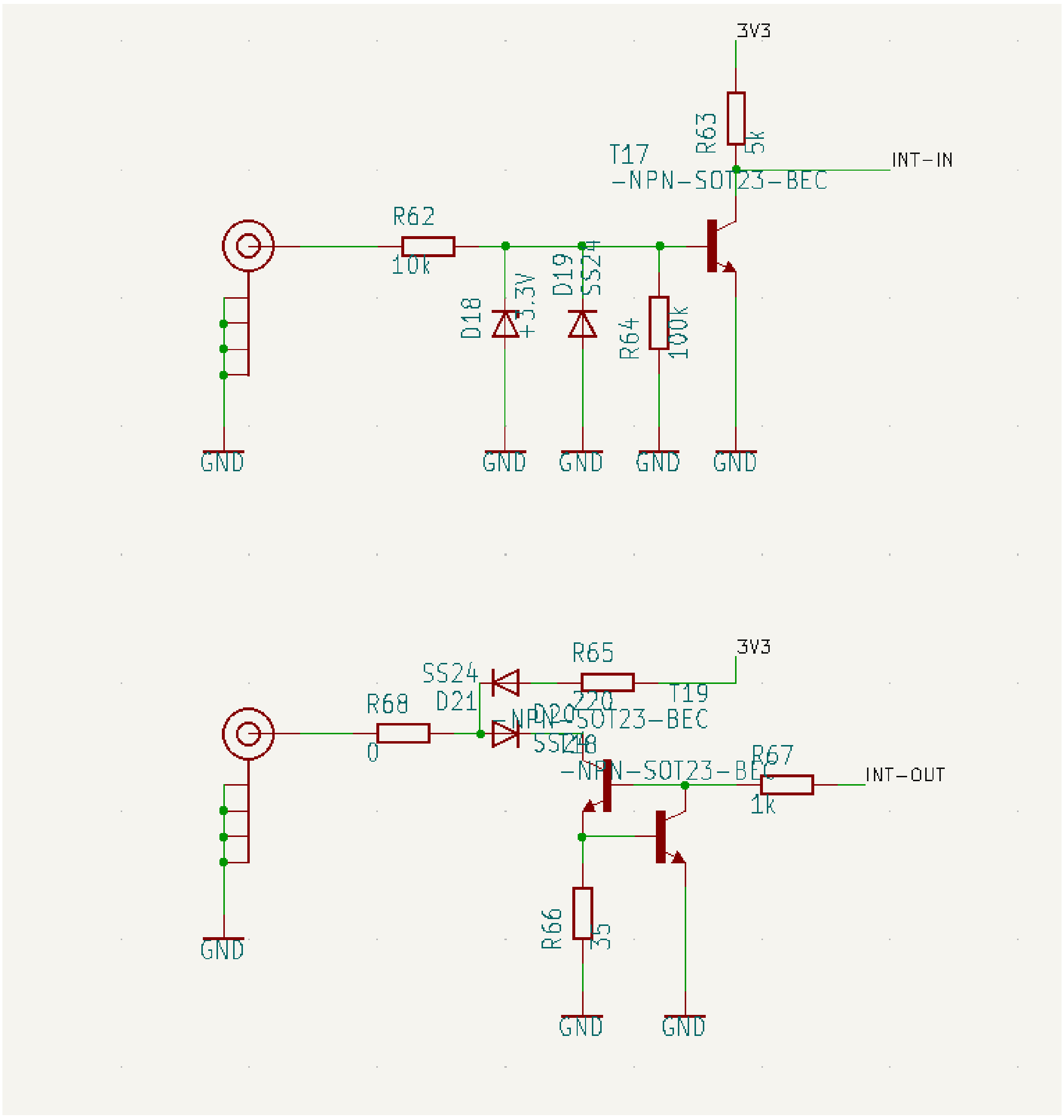


Figure C.3: Circuit diagram for IO Control

```
//For test, 2.5V = 50% = 512/1024 (10 bit max = 1024). 512 = 01000000000

void setup() {
  // put your setup code here, to run once:
  pinMode(syncPin, OUTPUT);
  pinMode(clkPin, OUTPUT);

  pinMode(dPin0, OUTPUT);
```

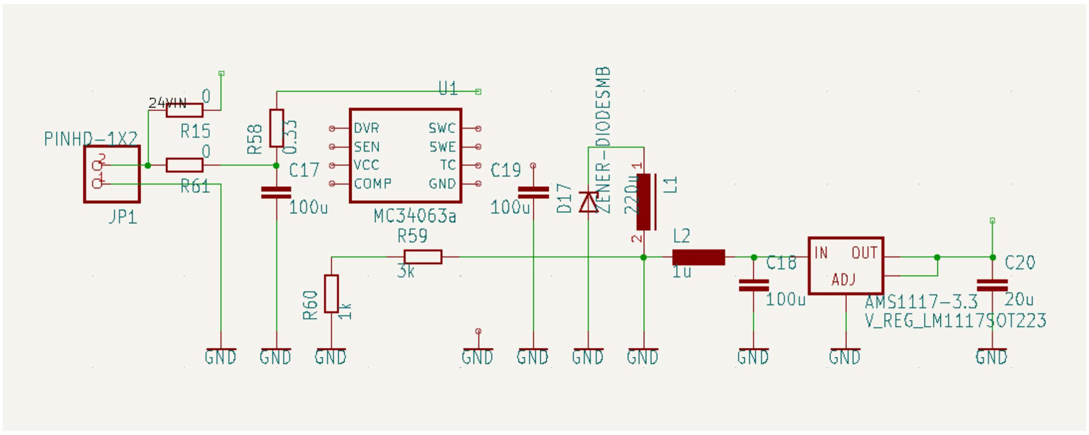


Figure C.4: Circuit diagram for step down converter

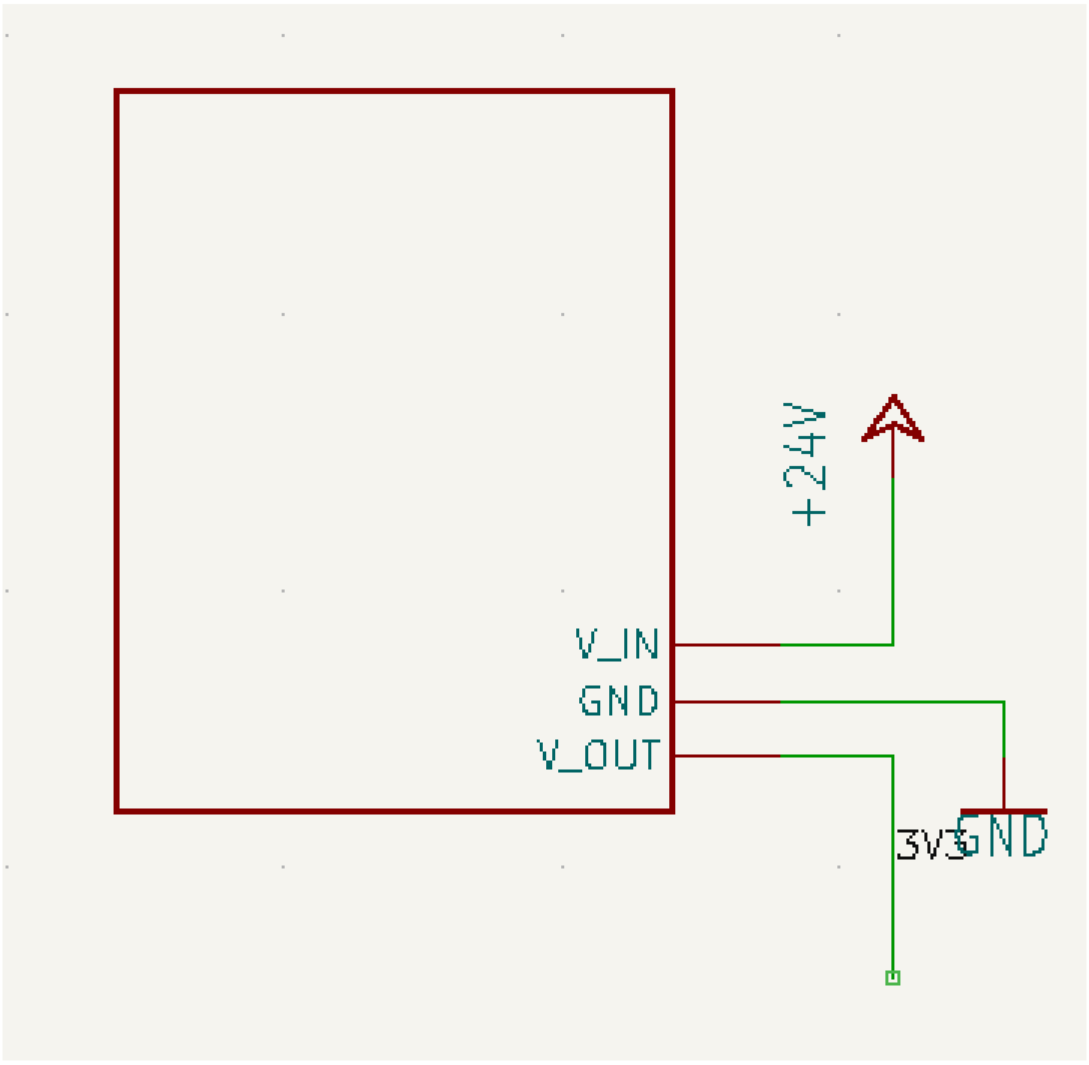


Figure C.5: Circuit diagram for step up converter

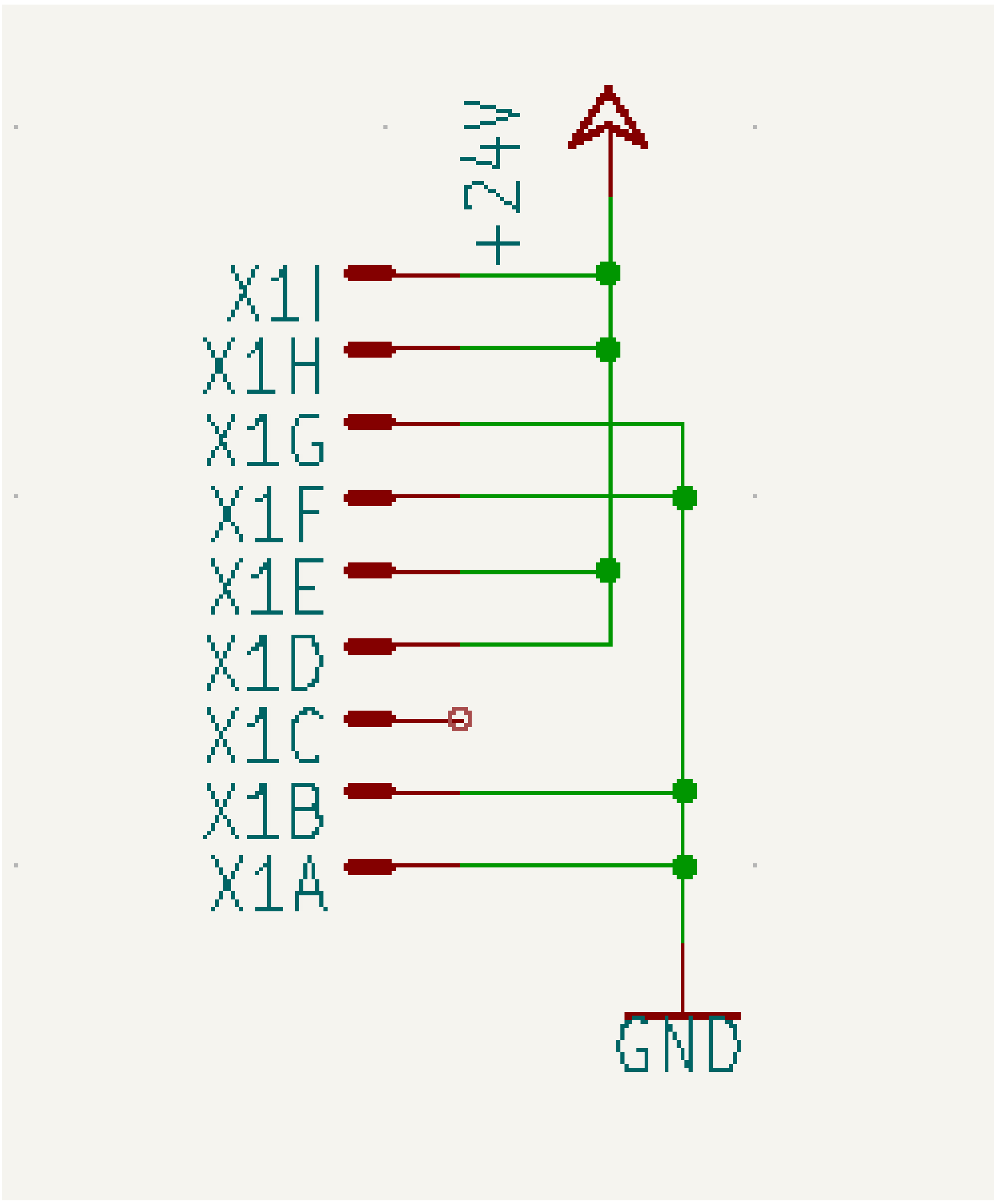
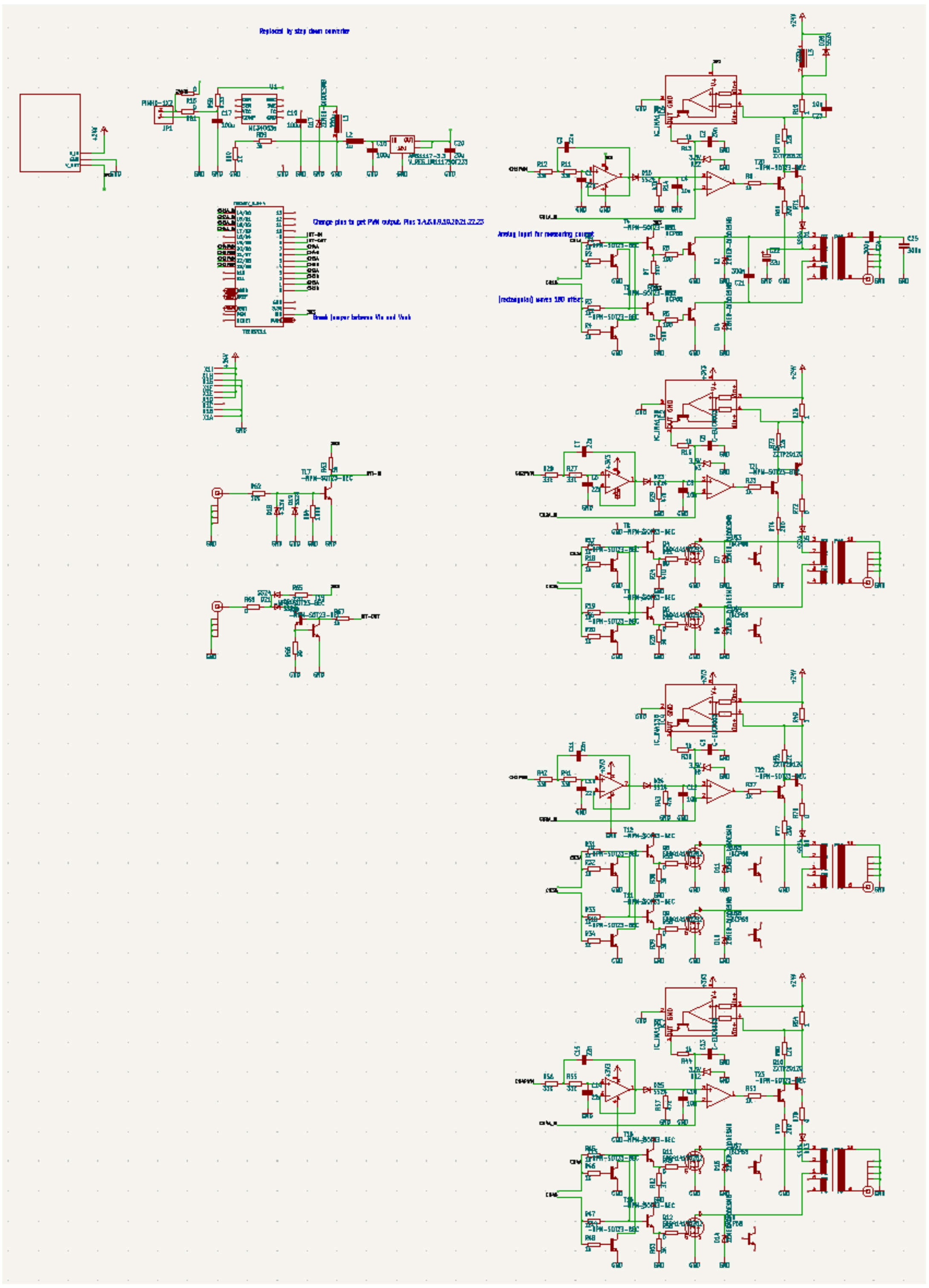


Figure C.6: Circuit diagram for digital output

```
    pinMode(dPin1, OUTPUT);
    pinMode(dPin2, OUTPUT);
    pinMode(dPin3, OUTPUT);
    pinMode(dPin4, OUTPUT);
    pinMode(dPin5, OUTPUT);
    pinMode(dPin6, OUTPUT);
```



Replaced by step down converter
Change pins to get PWM output. Pins 3,4,5,6,9,10,20,21,22,23
Analog input for measuring current
(rectangular) waves 180 offset
Break jumper between Vin and Vusb

Figure C.7: Circuit diagram for 4 channel electrode controller. This includes all previous components with some replicated for multiple channels

```
    pinMode(dPin7, OUTPUT);

```

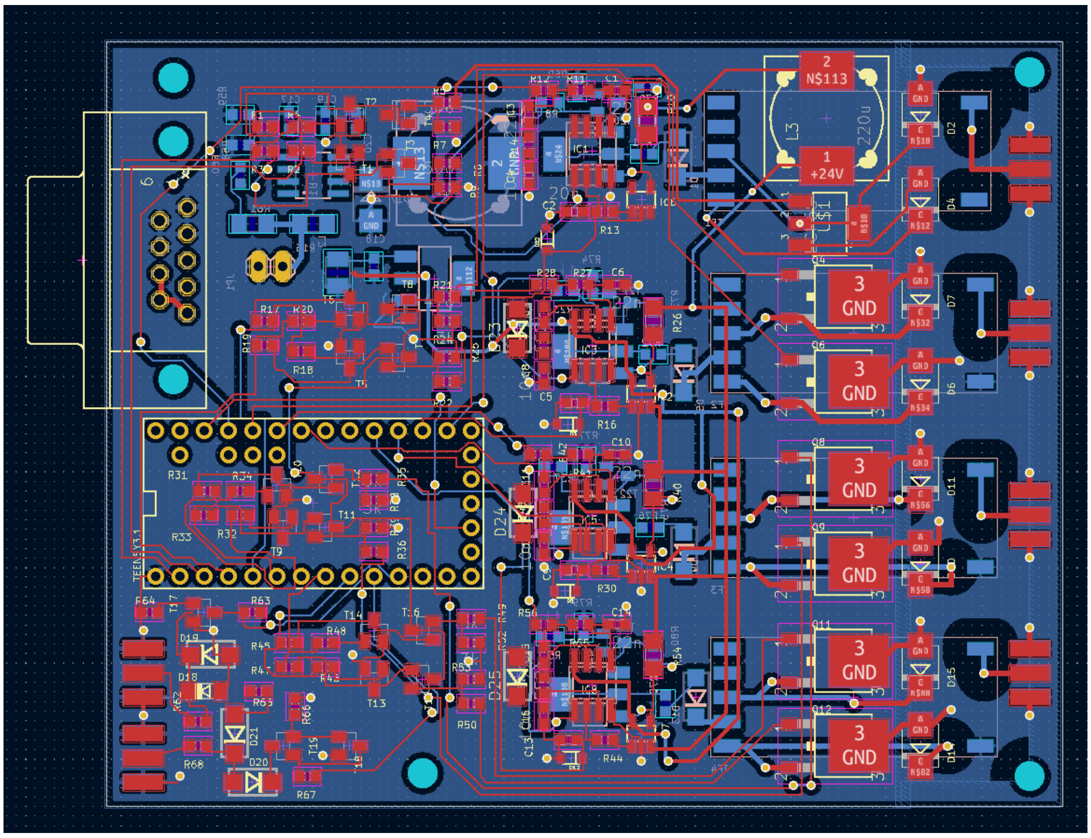


Figure C.8: Circuit board created for 4 channel control of experimental system

```
    digitalWrite(syncPin, HIGH);
    digitalWrite(clkPin, HIGH);

    digitalWrite(dPin0, HIGH);
    digitalWrite(dPin1, HIGH);
    digitalWrite(dPin2, HIGH);
    digitalWrite(dPin3, HIGH);
    digitalWrite(dPin4, HIGH);
    digitalWrite(dPin5, HIGH);
    digitalWrite(dPin6, HIGH);
    digitalWrite(dPin7, HIGH);

    Serial.begin(9600);
```

```
  bitConvert(da0);
}

void bitConvert(int temp)
{
  for(int i = 0; i < 10; i++)
  {
    dArray1[10-i] = temp & 1;
    Serial.println(dArray1[10 - i],BIN);
    temp >>= 1;
  }
  Serial.println();
}
void writeToPins(int bit_count)
{
  if ((dArray1[bit_count] & 0x001) == 0) digitalWrite(dPin0, LOW); else digital
//  if (((da1 >> bit_count) & 0x001) == 0) digitalWrite(dPin1, LOW); else digit
//  if (((da2 >> bit_count) & 0x001) == 0) digitalWrite(dPin2, LOW); else digit
//  if (((da3 >> bit_count) & 0x001) == 0) digitalWrite(dPin3, LOW); else digit
//  if (((da4 >> bit_count) & 0x001) == 0) digitalWrite(dPin4, LOW); else digit
//  if (((da5 >> bit_count) & 0x001) == 0) digitalWrite(dPin5, LOW); else digit
//  if (((da6 >> bit_count) & 0x001) == 0) digitalWrite(dPin6, LOW); else digit
//  if (((da7 >> bit_count) & 0x001) == 0) digitalWrite(dPin7, LOW); else digit
}

//Need to:
//          1)Implement writeToPins
//          2)Read in values from serial after each loop
//            3)Convert values from serial to binary
//            4)Convert arrays to be properly sent to D/A
void loop() {

  digitalWrite(clkPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(clkPin, LOW);
  delayMicroseconds(10);

  int bit_count = 0;
  digitalWrite(syncPin, LOW);

  //First two bits are don't cares
  digitalWrite(clkPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(clkPin, LOW);
  delayMicroseconds(10);
  digitalWrite(clkPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(clkPin, LOW);
  delayMicroseconds(10);

  //Next two bits are power down control, 0,0 for normal operation
  digitalWrite(clkPin, HIGH);
  digitalWrite(dPin0, LOW);
  digitalWrite(dPin1, LOW);
  digitalWrite(dPin2, LOW);
  digitalWrite(dPin3, LOW);
```

```
    digitalWrite(dPin4, LOW);
    digitalWrite(dPin5, LOW);
    digitalWrite(dPin6, LOW);
    digitalWrite(dPin7, LOW);
    delayMicroseconds(10);
    digitalWrite(clkPin, LOW);
    delayMicroseconds(10);

    digitalWrite(clkPin, HIGH);
    delayMicroseconds(10);
    digitalWrite(clkPin, LOW);
    delayMicroseconds(10);

    //Next 10 pins are the data pins
    digitalWrite(clkPin, HIGH);
    bit_count++;
    writeToPins(bit_count);
    delayMicroseconds(10);
    digitalWrite(clkPin, LOW);
    delayMicroseconds(10);

    digitalWrite(clkPin, HIGH);
    bit_count++;
    writeToPins(bit_count);
    delayMicroseconds(10);
    digitalWrite(clkPin, LOW);
    delayMicroseconds(10);

    digitalWrite(clkPin, HIGH);
    bit_count++;
    writeToPins(bit_count);
    delayMicroseconds(10);
    digitalWrite(clkPin, LOW);
    delayMicroseconds(10);

    digitalWrite(clkPin, HIGH);
    bit_count++;
    writeToPins(bit_count);
    delayMicroseconds(10);
    digitalWrite(clkPin, LOW);
    delayMicroseconds(10);

    digitalWrite(clkPin, HIGH);
    bit_count++;
    writeToPins(bit_count);
    delayMicroseconds(10);
    digitalWrite(clkPin, LOW);
    delayMicroseconds(10);

    digitalWrite(clkPin, HIGH);
    bit_count++;
    writeToPins(bit_count);
    delayMicroseconds(10);
    digitalWrite(clkPin, LOW);
    delayMicroseconds(10);

    digitalWrite(clkPin, HIGH);
    bit_count++;
    writeToPins(bit_count);
    delayMicroseconds(10);
    digitalWrite(clkPin, LOW);
    delayMicroseconds(10);

    digitalWrite(clkPin, HIGH);
    bit_count++;
    writeToPins(bit_count);
    delayMicroseconds(10);
    digitalWrite(clkPin, LOW);
    delayMicroseconds(10);

    digitalWrite(clkPin, HIGH);
    bit_count++;
    writeToPins(bit_count);
    delayMicroseconds(10);
    digitalWrite(clkPin, LOW);
    delayMicroseconds(10);

    digitalWrite(clkPin, HIGH);
    bit_count++;
    writeToPins(bit_count);
    delayMicroseconds(10);
    digitalWrite(clkPin, LOW);
    delayMicroseconds(10);

```

```

  //Last two bits are dont cares
  digitalWrite(clkPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(clkPin, LOW);
  delayMicroseconds(10);
  digitalWrite(clkPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(clkPin, LOW);
  digitalWrite(syncPin, HIGH);
  delayMicroseconds(10);

}
```