Testing of Two Novel Semi-Implicit Particle-in-Cell Techniques

A thesis submitted in partial fulfillment of the

Requirements for the degree of

Master of Science in Engineering

By

Trenton J. Godar

B.S., Wright State University, 2012

2014

Wright State University

WRIGHT STATE UNIVERSITY
GRADUATE SCHOOL

May 30, 2014

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY Trenton J. Godar ENTITLED Testing of Two Novel Semi-Implicit Particle-In-Cell Techniques BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Master of Science in Engineering.

_____
James Menart, Ph.D.
Thesis Director

_____
George Huang, Ph.D.
Chair
Department of Mechanical and Materials Engineering

Committee on Final Examination

_____
James Menart, Ph.D.

_____
Zifeng Yang, Ph.D.

_____
Amir Farajian, Ph.D.

_____
Robert E. W. Fyffe Ph.D.
Vice President for Research and
Dean of the Graduate School

ABSTRACT

Godar, Trenton J., M.S. Egr College of Engineering and Computer Science, Wright State University, 2014. Testing of Two Novel Semi-Implicit Particle-In-Cell Techniques

PIC (Particle-in-cell) modeling is a computational technique which functions by advancing computer particles through a spatial grid consisting of cells, on which can be placed electric and magnetic fields. This method has proven useful for simulating a wide range of plasmas and excels at yielding accurate and detailed results such as particle number densities, particle energies, particle currents, and electric potentials. However, the detailed results of a PIC simulation come at a substantial cost of computational requirement and the algorithm can be susceptible to numerical instabilities. As processors become faster and contain more cores, the computational expense of PIC simulations is somewhat addressed, but this is not enough. Improvements must be made in the numerical algorithms as well. Unfortunately, a physical limit exists for how fast a silicon processor can operate, and increasing the number of processing cores increases the overhead of passing information between processors. Essentially, the solution for decreasing the computational time required by a PIC simulation is improving the solution algorithms and not through increasing the hardware capacity of the machine performing the simulation. In order to decrease the computational time and increase the stability of a PIC algorithm, it must be altered to circumvent the current limitations.

The goal of the work presented in this thesis is twofold. The first objective is to develop a three-dimensional PIC simulation code that can be used to study different numerical algorithms. This computer code focuses on the solution of the equation of motion for charged particles moving in an electromagnetic field (Newton-Lorentz equation), the solution of the electric potentials caused by boundary conditions and charged particles (Poisson's Equation), and the coupling of these two equations. The numerical solution of these two equations, their coupling, which is the primary cause of instabilities, and the severe computational requirements for PIC codes make writing this code a difficult task. Solving the Newton –Lorentz equation for large numbers of charged particles and Poisson's equation is complex. This is the focus of this newly developed computer code.

The second objective of the work presented in this thesis is to use the developed computer code to study two ideas for improving the numerical algorithm used in PIC codes. The two techniques investigated are: 1) implementing a fourth order electric field approximation in the equation of motion and 2) solving for the electric field, i.e. solving Poisson's equation, multiple times within a single time step. The first of these methods uses the electric fields of many cells that a charged particle may pass through in one time step. This is opposed to using only the cell of origin electric field for the particle's entire path during one time step. The idea here is to allow PIC codes to use larger time steps while remaing stable and avoiding numercal heating; thus reducing the overall computer time required. The second technique studied is utilizing multiple Poisson equation solves during a single time step. Typically, an explicit PIC model will solve the electric field only once during a time step; however, solving the field multiple times during the particle push allows particles to distribute themselves in a more electrically nuetral manner within a single time step. The idea here is to allow larger time steps to be used without obtaining unrealistic electric potentials due to an artificial degree of charge separation. This eliminates instabilities and numerical heating. Explicit PIC codes have limits on how large the numerical time step can be before the electric potentials blow up.

This work has shown that neither of these techniques, in their current state, are practical options to increase the time step of the PIC algorithm while maintaining the correct solution. However, stability improvements are observed which warrant further investigation into alternative implementations of these techniques. The current fourth order electric field technique seems to have little effect on the solution, but the multiple solves technique does show some improvement in stability over the explicit routine. At time steps where the explicit routine begins to oscillate and become unstable, the multiple solves routine remains stable. These techniques are not quite as developed as they could be, meaning some of the future work suggested in this report could lead to one or both of these techniques being successful in the future.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

NOMENCLATURE

A_x = Area normal to x direction

A_y = Area normal to y direction

A_z = Area normal to z direction

$\vec{B}$ = Magnetic field

$\vec{E}$ = Electric field

E_1, E_2, E_3, E_4, E_5 = Electric field values along a particle's projected path

$\vec{F}$ = Force

I_{sp} = Specific impulse

$N_{\frac{comp\ particles}{cell}}$ = Number of computer particles per cell

N_e = Total number of computer electrons

N_{proc} = Number of parallel processors dedicated to the simulation

S = Source term

T_e = Electron temperature

T_{ion} = Ion temperature

V = Volume

V_{total} = Volume of the entire computational domain

W_p = Particle Weighting

W_x= X position weighting factor

W_y= Y position weighting factor

W_z= Z position weighting factor

a_0, a_1, a_2, a_3, a_4 = Coefficients for fourth order electric field technique

$a_E, a_W, a_N, a_S, a_T, a_B, a_P$ = Coefficients used for solving Poisson's equation

a_p = Acceleration experienced by particle *p*

a_i = Sup-diagonal of TDMA matrix

b_i = Sub-diagonal of TDMA matrix

c_i = Diagonal of TDMA matrix

d_i = Source term of TDMA matrix

f(v) = Maxwell-Boltzmann probability function

$f_{parallel}$ = Fraction of the code that can be executed in parallel

i = X indicia

j = Y indicia

k = Z indicia

k_b = Boltzmann constant

m = Mass

n_e = Number density of electrons

n_{ion} = Number density of ions

q = Charge

$q_{i,j,k}$ = Charge accumulated in node i,j,k

t = Time

$\vec{v}$ = Velocity

ε_0 = Permittivity of free space

λ_D = Debye length

ω_{pe} = Plasma frequency

ρ = Charge density

Δt = Time step

Δx = Grid size in x direction

Δy = Grid size in y direction

Δz = Grid size in z direction

Φ = Electric potential

Φ_{fl} = Floating potential

Φ_{pe}= Plasma potential

Φ_w = Biased wall potential relative to plasma

Φ_{wall} = Constant wall potential

CHAPTER 1

INTRODUCTION

PIC (Particle-in-cell) modeling has successfully been used to theoretically predict particle motion of fluids, plasmas in particular, since the mid 1950's (Birdsall & Langdon, 1991). Although this computational method has proven to be a valuable tool, it continues to suffer from a number of drawbacks. Most notably, the computational requirement to perform PIC simulations is quite significant. The work presented here represents an attempt to alleviate a portion of this copious computational requirement via increasing the time step size that can be used in the algorithm. A very rough description of the PIC method is: inject particles, calculate electric fields, move particles, and repeat. The particle-in-cell method is named as such because it advances finite particles through a spatial area consisting of cells.

One of the major factors driving the computational expense of a PIC simulation is the size of the time step. The smaller the time step, the more realistic and stable the simulation; however, decreasing the time step increases the computational time required to obtain a steady state solution. Given only this information, the simple answer to decreasing computational time would be to increase the size of the time step. Increasing the time step not only decreases the accuracy of the solution, but stability and numerical heating also become issues. Numerical heating, in particular, mainly arises due to improperly transferring energy from the electric field to the particles (Ueda, Omura, Matsumoto, & Okuzawa, 1994), thus causing particles to become unrealistically fast, which leads to instability. The time scale of the plasma characteristics which need to be resolved also plays a role in determining the allowable size of the time step.

Another solution to reducing computational time that may come to mind is simply to improve the hardware of the machine running the simulation. While this can help to a degree, there are physical limits as to how much this can help. The speed of a silicon processor has both physical and practical limits regarding the gate size and heat dissipation within the chip. Even now processors are approaching their physical and practical limits as the manufacturing tolerances for next generation chips are on the order of two or three atoms (Intel, 2014). Reduction in computational time via parallel processing is also limited according to Amdahl's law which takes into account the additional overhead incurred by using multiple processors. Amdahl's law shows that even a code where 95% of the computation can be executed in parallel can at most be sped up by a factor of about 20. This maximum reduction factor decreases as the fraction of the code which cannot be executed in parallel increases. Amdahl's law is

$$Speedup = \frac{1}{\left(1 - f_{parallel}\right) + \frac{f_{parallel}}{N_{proc}}} \tag{1}$$

and is plotted for several cases in Figure 1.

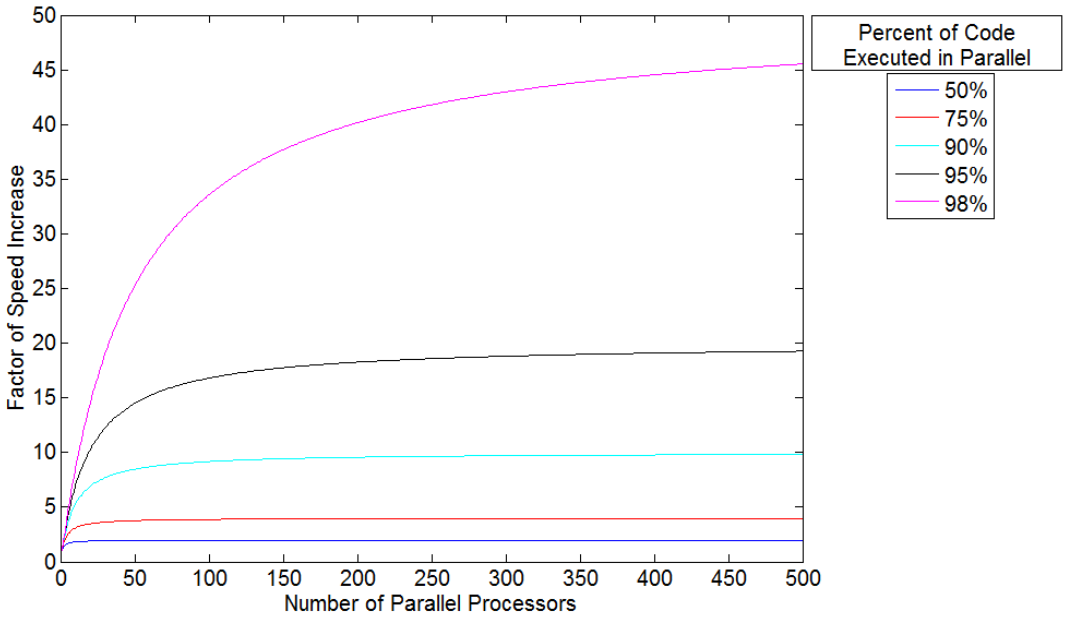

Figure 1. Amdahl's law for various percentages of parallel execution.

In equation (1), $f_{parallel}$ is the fraction of the code that is executed in parallel and N_{proc} is the number of parallel processors dedicated to the program. As is shown in Figure 1, even codes where nearly all of the code can be executed in parallel, a cap exists to how much speed-up can be achieved via parallel computing. This further reinforces the fact that algorithm improvement, not hardware improvement is required to tackle the computing time problem associated with PIC simulations.

1.1. Goals

The ultimate goal of this work is to reduce the computational time required to achieve a steady state solution of a given plasma problem using the PIC method. Two approaches to accomplish this goal are to decrease the time it takes to execute a single time step or to decrease the total number of time steps. The work presented here investigates the viability of two different techniques designed to decrease the total number of time steps in order to reduce overall computational time. With this approach, it is extremely likely that the time required to execute a single time step will increase, but if the time step size can be increased enough to overcome this additional computational requirement, then the overall computational time of the simulation will decrease; thus achieving our goal.

1.2. Motivation

Improving the speed of the PIC algorithm would be a very valuable tool for investigating any device or natural occurrence involving plasma. With plasma comprising 99.9% of the known matter in the universe and numerous plasma utilizing devices ranging from ion thrusters in space to plasmajets on the operating table, plasma can be found almost anywhere (Plasma Surgical, 2013) (Mullen, 1999). The ability to simulate natural plasma phenomenon in space may lead to discoveries about the past, present, and future of the cosmos and how it may affect us and our understanding of the universe. As for man-made devices, such as ion thrusters, the ability to simulate the device's performance is pivotal to determining if the device will work and how efficiently it will do so. If the simulation of a device can be completed in a reasonable amount of time, with a reasonable degree of accuracy, the device can be optimized and tested with little need for experimentation. The obvious benefit in this case is the reduced financial and physical risk associated with testing the device.

To illustrate the usefulness of the PIC simulation, ion thrusters and simulations of their operation will briefly be discussed. Ion thrusters offer several advantages over other propulsion types when it comes to space propulsion with no strict time limit on the mission. Most notably, an ion thruster offers the highest specific impulse (I_{sp}) of any propulsion technology that exists at this time. Specific impulse and the amount of fuel required to achieve a given total impulse are inversely proportional, so a high specific impulse means a smaller amount of fuel required. Ion thrusters have very practical applications, namely satellite station keeping and deep space missions where ion thrusters are mission enhancing or even mission enabling. Broadly speaking, ion thrusters operate by generating plasma within a discharge chamber then extracting and accelerating the ions using an electric field. Accelerating the ions out of the rear of the thruster then produces thrust in the opposite direction.

Presently, the explicit PIC algorithm has been successfully used to model the plasma within the discharge chamber of an ion thruster. Comparing the results produced by the X-Grafix Object Oriented Particle-in-Cell (XOOPIC) code, which uses the explicit PIC algorithm, and experimental results taken by Dan Herman (Herman, 2005) for NASA's Evolutionary Xenon Thruster (NEXT) highlights the validity and usefulness of the explicit PIC algorithm. These results are shown below Figures Figure 2, Figure 3, and Figure 4. These figures also highlight another useful aspect of the PIC code which is the ability to produce results in areas where experimental measurement is very difficult. Several detailed results of this ion thruster discharge chamber simulation which are difficult to measure experimentally, but can easily be found using a PIC simulation, include energies of each particle type, current magnitudes of each particle type, and current directions of each particle type. Bias et al. (2011) can be consulted to see the largest compilation of results for the plasma in an ion engine discharge chamber ever produced. These results were produced by a PIC algorithm.

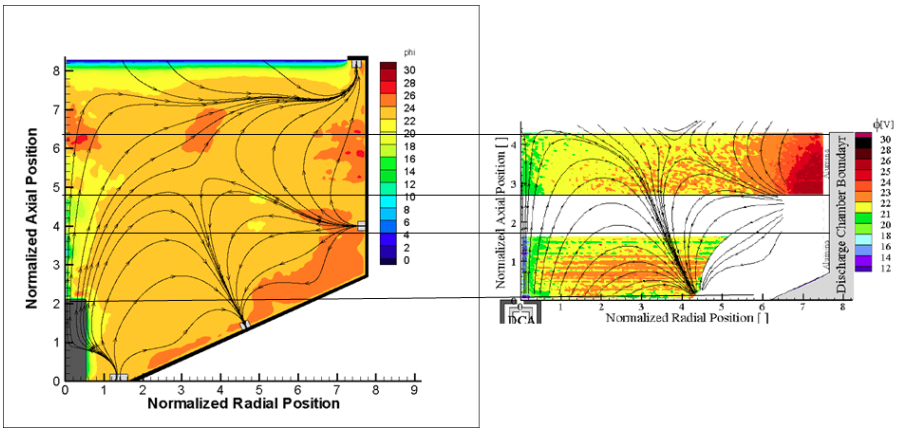

Figure 2. On the left are total electrical potentials produced by a PIC algorithm **(Mahalingam, Choi, Loverich, Stoltz, Jonell, & Menart, 2010)** and on the right are measured plasma potentials **(Herman, 2005)**. Note that the PIC electrical potentials are for the entire discharge chamber while the measured electrical potentials are for part of the discharge chamber.

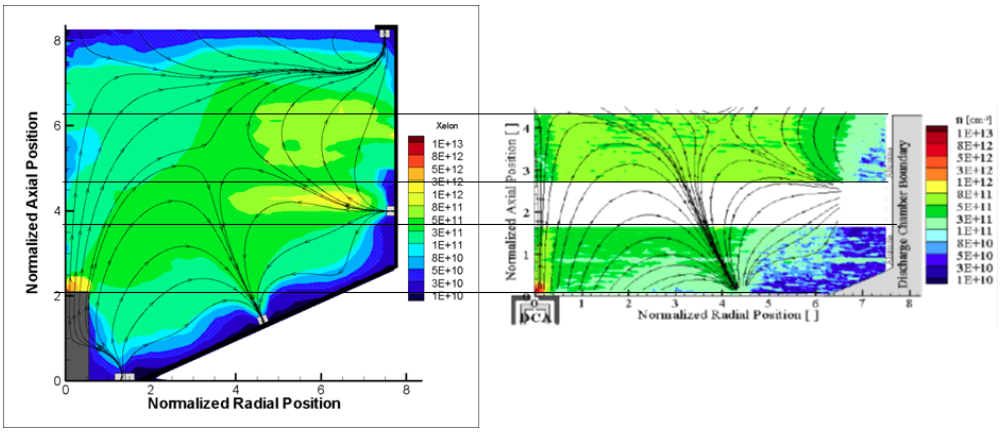

Figure 3. On the left are total ion densities produced by a PIC algorithm **(Mahalingam, Choi, Loverich, Stoltz, Jonell, & Menart, 2010)** and on the right are measured ion densities **(Herman, 2005)**. Note that the PIC ion densities are for the entire discharge chamber while the measured ion densities are for part of the discharge chamber.

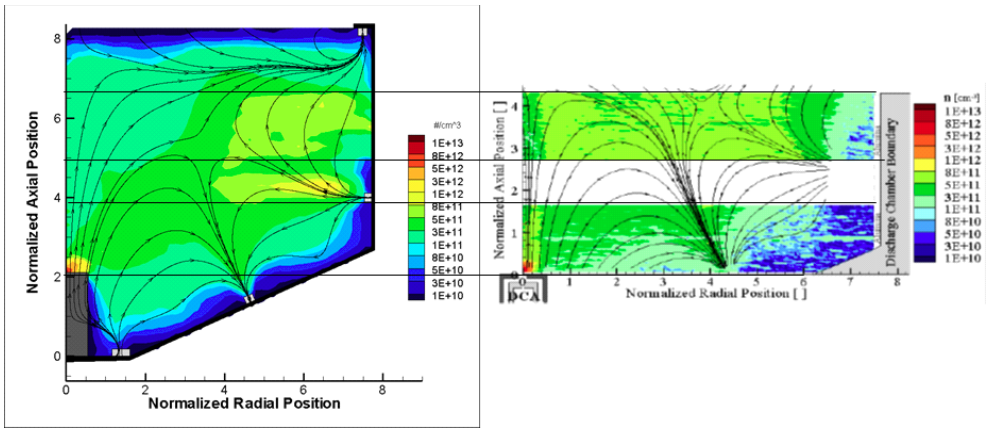

Figure 4. On the left are total electron densities produced by a PIC algorithm **(Mahalingam, Choi, Loverich, Stoltz, Jonell, & Menart, 2010)** and on the right are measured electron densities **(Herman, 2005)**. Note that the PIC electron densities are for the entire discharge chamber while the measured electron densities are for part of the discharge chamber

As shown in Figures Figure 2, Figure 3, and Figure 4, the explicit code is capable of yielding valid, detailed results at all spatial locations within the discharge chamber. The downside to obtaining the results shown above is the amount of processing power devoted to simulating a single case for a single configuration of the NEXT ion thruster; over 3 months on a supercomputer utilizing 64 cores. For example, if this computer time is rented from the Titan super computer at $0.05 / node hour, this single result would cost well over $7,000 (Oak Ridge National Laboratory, 2012). While $7,000 may sound like a lot of money, these same results would cost much more to obtain experimentally. If a thruster is not yet assembled, such would be the case with a new design or configuration, the process to go from design to experimental testing would take a tremendous amount of time and money. A juncture such as this is where the PIC simulation would be most beneficial. Using a PIC code, several different designs and operating conditions could be simulated with very little risk financially or physically as compared to an experimental setup. If a PIC code were able to produce valid, detailed results in a matter of days or weeks instead of months, the design and optimization process of ion thruster production could benefit tremendously. Although this section has focused primarily on ion thrusters, a similar process could be executed on any number of plasma utilizing devices.

1.3. Computational Methods of Plasma Modeling

Several methods exist for plasma modeling and are briefly presented here, but are not explored in great detail. The type of plasma parameters and the timescale with which features need to be resolved both play a role in determining the optimal modelling method. Low density plasma behaves more like a collection of finite particles as opposed to a dense plasma which may behave more like a fluid. If phenomena occurring on the order of nanoseconds or picoseconds (i.e. the plasma frequency timescale) need to be resolved, a fully kinetic code is desirable whereas the slower phenomena can be resolved using hybrid codes or fluid codes. In the spatial domain, if phenomena on the the order of the Debye length need to be resolved, a fully kinetic code should be used. Lastly if many of the plasma phenomena deviate from local thermodynamic equilibrium, kinetic codes are the more desireable option.

Codes used for modeling plasma fall into one of three broad categories, fluid descriptions, kinetic descriptions, or a hybrid of the two (Kim, Iza, Yang, Radmilovic-Radjenovic, & Lee, 2005). The fluid description solves a set of equations known as the magnetohydrodynamic (MHD) equations. This type of code does not track individual particles within the plasma, but rather treats the plasma as a fluid. Eliminating the need to track all the particles within the plasma significantly decreases the computational time required to reach a solution. In order to use MHD codes, several assumptions must be made about the plasma (velocity distributions, density distributions, etc.) which sacrifices the accuracy, spatial resolution, and temporal resolution as compared to a code using a kinetic description.

Codes using a kinetic description are generally either Vlasov codes, Fokker-plank codes, or particle codes. The PIC code used in this work falls under the particle code category (Birdsall & Langdon, 1991). Each of these kinetic descriptions branch off and become even more diverse when their methods of time integration and field solving are considered. Another type of kinetic particle code is a gridless code using finite particle sizes (Christlieb, Krasny, verboncoeur, Emhoff, & Boyd, 2006). In this type of code, finite particles are still used causing it to fall under the category of a particle code; however, the particles are not advanced through a grid. Instead, a Fourier transform is used to translate

between particle positions and the forces on other particles based on the positions of all the particles (Briguglio, Vlad, Martino, & Fogaccia, 2000).

Hybrid codes also exist which essentially replace part of a kinetic code with a fluid assumption in the interest of saving computational time. For example, a hybrid code may assume the distribution of electrons will behave as a massless fluid while tracking ions as particles. This allows the code to use larger time steps than a purely kinetic code, because the ions are generally several orders of magnitude slower than electrons. This type of hybrid code would also eliminate the need to track individual electrons which would further reduce computational time, since those particles would no longer need to be advanced. A brief history and description of hybrid codes can be found in Winske et al. (2003).

1.4. Particle-In-Cell (PIC) Modeling

The particle-in-cell algorithm falls under the category of a fully kinetic particle code. As far as the types of modelling available, PIC modelling falls on the extremes of both time scale and density. The PIC method is most useful for very fast phenomena which translates to very small time scales. As stated earlier, PIC modelling is also at the extreme for which type of plasma it is best suited for, which is low density. High density plasma is better modelled by fluid codes than low density plasma. A PIC simulation may be used to simulate high density plasma, however the computational times required cause it to become impractical very quickly.

1.4.1. Strengths

Since PIC modeling involves tracking each computer particle, of each species present, very high spatial resolution and temporal resolution is possible to achieve. In the limits of PIC model parameters for time step, grid size, and particle weighting, this method should give the correct solution because no fluid assumptions are made. In the limit of a PIC model, the time step and grid size would both be infinitesimally small while the number of computer particles would be exactly equal to the number of real particles.

In practice, some finite value must be given to the grid size, particle weight, and time step. The values of these numerical parameters strongly depend on the type of problem being solved and the physics which need to be resolved. Acceptable values for these numerical

parameters must be determined through a convergence study before applying this method to solving real problems. Most generally, the time step size, grid size, and particle weighting should all be as small as possible and only constrained by machine resources and computing time. If computing time is the primary target, then the time step, grid, and particle weighting should all be as large as possible with the simulation remaining stable and its solution converged.

1.4.2. Weaknesses

As previously discussed, the computing time required for a particle in cell code tends to be its biggest weakness. This downside can be somewhat alleviated using parallel processing; but, as stated earlier, this is not enough. PIC modeling is subject to several stability criteria which affect the size of the spatial grid, time step, and particle weighting. These stability criteria place practical limits on the performance of the PIC algorithm and are discussed in more detail in Chapter 2. Along with the large computational requirement, the PIC algorithm is vulnerable to instability and numerical heating from a number of parameters.

Numerical heating is a process by which the particles in the simulation are accelerated more than they physically should be, simply due to the finite nature of the PIC method. This artificial heating can be caused by almost any numerical parameter such as time step size, grid size, and even the number of particles in each cell (Ueda, Omura, Matsumoto, & Okuzawa, 1994). As will be shown later, many of the stability criteria which apply to the PIC method depend on the electron temperature. Thus if the electron temperature is artificially increased, the stability criteria may become violated leading to an unstable simulation. On a related note, the numerical parameters used for a PIC simulation are almost always related in some fashion which makes isolating the effect of a single parameter on the simulation very difficult to define. Thus, all the numerical parameters must be within acceptable means before the simulation begins. If even one of these parameters is off, the simulation may be susceptible to numerical heating and instabilities.

1.5. Literature Review

PIC modeling can largely be attributed to the work of John Dawson during the late 1950's and early 1960's (Birdsall & Langdon, 1991). During this time, PIC modeling was

quite impractical due to the lack of computing power available. Once computers became more capable, one dimensional electrostatic codes could be developed and practically applied. With the development of faster processors and super computers, two dimensional and three-dimensional codes have become a practical means to simulate plasma. As computers become more powerful and parallel computing becomes commonplace, PIC modelling becomes more feasible and practical. It needs to be restated that while computational power is a huge driving force in increasing the use of PIC modeling, enhancement in the numerical routines is still needed.

1.5.1. General PIC Modeling

The three-dimensional code developed in this work is largely based on the two dimensional work of Mahalingam (2007), which was developed with regards to ion engine discharge chamber modelling. Numerous types of PIC codes have been developed which model plasma in various situations such as discharge plasma, microwave plasma, space plasma, and nuclear plasma, along with several others (Tech-X Corporation, 2014). The reason for several different types of models, for different situations, is because the most efficient way to approach any code is to resolve as little physics as possible while still retaining and simulating the physics of interest. Regardless of the exact method, PIC simulations roughly follow the same cycle (Gibbons & Hewett, 1995):

1) use grid values to determine the force on each particle,
2) move the particles based on integrating the equations of motion,
3) use the particle's new locations to calculate the source terms for the electric field solver, and
4) use the source terms to determine the new grid quantities such as the electric field.

The differences between PIC simulations are how each of the four steps above is implemented. The first step can be performed by either using the quantity of the grid the particle of interest is in, or interpolating grid values based on some shape (linear, quadratic, etc.).

Step 2 shown above is the time integration of the equations of motion. When simulating charged particles, the differential equation which must be integrated over time is the Newton-

Lorentz force as shown in equation (2). The time integration method used in this work is known as the leapfrog method (Birdsall & Langdon, 1991); however, this method is not the only way to advance the particles to their new locations. Some other methods which can be used to advance the particles are the Euler's first order scheme, the biasing scheme (time centered and time decentered), the Runge-Kutta method, and several others (Tajima, 2004). Each of these methods has their own strengths and weakness which generally manifest as tradeoffs between stability, accuracy, and computational requirement.

Step 3 is essentially the relationship between the particle positions and the source terms. At this point, either a Fourier transform can be used to relate the particle positions to the electric field or the code can advance to step 4. The code developed in this work, as well as many other codes, uses a spatial weighting scheme to distribute each particle's charge to the surrounding grid points; however, the method by which this is done can vary. In observing how a single particle's charge is distributed to the grid, a zero-order scheme would attribute all of that particle's charge to the grid it currently resides in. Higher order schemes, such as the first order weighting scheme described in Chapter 2, attribute the particle's charge to several surrounding nodes (e.g. the first order scheme in three dimensions uses the 8 nearest nodes). How much of the particle's charge is actually allocated to the source term is also a source of variation among PIC codes. In an attempt to improve stability, some codes will inflate the permittivity of free space which effectively weakens the coupling between the particle positions and the source term used to calculate the grid quantities (Mahalingam, Choi, Loverich, Stoltz, Jonell, & Menart, 2010).

Once the source terms are calculated, step 4 can be executed which finds the grid quantities throughout the domain using a combination of source terms and boundary conditions. The method by which this is done is generally either a Fourier transform or by an iterative matrix solver (e.g. tri-diagonal matrix algorithm) aimed at solving Poisson's equation (equation (4)). Solving Poisson's equation actually solves for the electric potentials as opposed to the electric fields, so the electric fields are found by taking the gradient of the electric potentials.

Another step commonly found in PIC simulations, but not stated in the list above, is particle collisions. Collisions are commonly how plasmas are formed and therefore a subject

of interest in many plasma applications. Generally, if a PIC code considers particle collisions, it will use a Monte Carlo collision (MCC) model to simulate them. The MCC model takes into account the energy of each particle to determine its collision cross section and the number density of all particles capable of collisions. Using this information, the probability of a collision can be determined while random numbers are used to decide whether a collision takes place and what type of collision should be simulated (Vahedi & Surendra, 1995). The code developed in this work is a collisionless code.

1.5.2. Implicit and Semi-Implicit Routines

Several implicit and semi-implicit methods exist to improve the time step capability of the PIC algorithm. For a routine to be considered implicit or semi-implicit, some aspect of the future time or location or a combination of the two must be utilized while advancing the particles or updating the electric and magnetic fields. Many implicit methods seek to damp out high frequency phenomena while preserving the lower frequency physics of the plasma. High and low frequency in this case are relative to plasma parameters such as plasma frequency and gyrofrequency. Phenomena are considered high frequency if their time scales are on the order of these plasma parameters. Implicit methods in the most general sense fall under one of two categories, the implicit moment method or the direct implicit method.

The implicit moment method aims to use fluid assumptions in order to estimate spatial properties, such as the electric field and the magnetic field, at a future time. Essentially the goal of this method is to determine the future fields before the particles are advanced using equations of motion (Lapenta, 2008). Unlike the direct implicit method which will be described shortly, there is no need for an iterative solution to predict the future electric fields. As the name implies, the moment method uses a set of moment equations to predict the final position of the particles without actually applying the equations of motion to each particle. Approximating the future position of the particles provides an approximate value to the future source terms and thus the future fields can be determined and applied to the present particle positions. In order to make this approximation, current vectors and pressure tensors are calculated and used. Knowing the current vectors at all locations in space, Ampere's law can be applied to aid in prediction of the future current vectors and thus particle number densities. More detailed descriptions of the moment method can be found in

Lapenta, Brackbill, & Ricci (2006), Brackbill & Forslund (1982), and chapter 5 of *Numerical Techniques in Electromagnetics, Second Edition* (Sadiku, 2001).

The direct implicit method is a predictor-corrector type of method which requires an iterative solution to be fully implicit. As is done with the moment method, the goal of the direct implicit method is to predict the future source terms. However, the direct implicit method does not make any fluid approximations but rather uses the particles to determine the future source terms. Essentially, the particles are advanced to an intermediate time where the source terms are then adjusted. Once the source terms are known at this intermediate time, the particles are then advanced through the full time step using the fields calculated at the intermediate time step. This method involves pushing the particles twice during a single time step, which may seem like the time step should just be cut in half; however, the intermediate update of the source terms and electric fields can be estimated so as to remove the need for solving Poisson's equation over the entire domain. This process can be iterated upon to some convergence criteria so as to be truly fully implicit, but iterating this process once serves to sufficiently improve the stability of the algorithm (Gibbons & Hewett, 1995). If only one iteration of this method is used, the method should actually be considered semi-implicit rather than fully implicit.

Both the implicit moment method and the direct implicit method can be altered to be semi-implicit if past information is used in conjunction with the estimated future field values. Instead of using the predicted field to advance the particles in each of these implicit methods, a combination of the predicted field and the previous field may be used. Besides this, not many other types of semi-implicit routines exist for PIC modelling, especially on the particle level which is investigated in this work. Generally, the implicit and semi-implicit routines found in the literature strive to predict the future fields of the entire spatial domain. Though the goal of the implicit and semi-implicit methods is to increase the size of the time step while keeping the simulation stable, they are generally still restricted by the need for the particle to cross less than one cell in a single time step (see stability criteria in equations (58) to (60)) (Lapenta, 2010).

CHAPTER 2

PIC ALGORITHM

As described previously, the PIC algorithm uses a kinetic particle description to advance one computer particle at a time according to electromagnetic theory. This chapter describes the three-dimensional PIC routine used in this work. First, the governing kinetic equations for the PIC routine are described, next, the explicit PIC routine developed in this work is described in detail, and lastly the stability criteria which apply to the explicit PIC algorithm are defined. The PIC code is first written in explicit form and verified to ensure the code is working properly. This properly functioning explicit code will serve as a tool to test the viability of the two semi-implicit techniques investigated in this work.

2.1. Charged Particle Kinetics

In the code developed, the only species of particles present are charged particles. Thus, the forces exerted on the particles due to magnetic fields and electric fields are of the most importance. The kinetics for charged particles is largely described using the Newton-Lorentz equation and Poisson's equation. The newton-Lorentz equation is

$$m\frac{d\vec{v}}{dt} = \vec{F} = q(\vec{E} + \vec{v}\,X\,\overrightarrow{B}) \tag{2}$$

where m is the mass of the particle, $\vec{v}$ is the velocity, $\vec{F}$ is the force, $\vec{E}$ is the electric field, q is the particle's charge, t is time, and $\vec{B}$ is the magnetic field. The magnetic field in this work is not a point of interest since it can redirect the particle's velocity vector, but is unable to increase or decrease its velocity. Thus, the magnetic field strength is simply an input and is

considered static throughout the simulation. The electric field value is determined by the gradient of the electric potential, as

$$\vec{E} = -\vec{\nabla}\Phi \, . \tag{3}$$

In equation (3), Φ is the electric potential. The electric potential is found using Poisson's equation:

$$-\frac{\rho}{\varepsilon_0} = \nabla^2\Phi \, . \tag{4}$$

In equation (4), ρ is the charge density and ε_0 is the permittivity of free space, which is a constant. In many PIC codes, the value for ε_0 is artificially inflated to improve stability as previously mentioned in Chapter 1. Increasing this value effectively reduces how much the particles affect the electric potential. The code developed in this work does not inflate the permittivity, so that the full effect of the particles is present when electric potentials are calculated. Equation (4) must be solved as the simulation progresses in order to determine how the charge density and thus the electric potential and the electric fields change as a function of time. The charge density is found for each cell as

$$\rho_{i,j,k} = \frac{q_{i,j,k} W_p}{V_{i,j,k}} \tag{5}$$

where the subscript *i, j, k* denotes the node indices, W_p is the particle weighting, $V_{i,j,k}$ is the volume of cell *i, j, k*, and $q_{i,j,k}$ is the charge accumulated in node *i, j, k*. Equation (4) is solved numerically via an iterative method. In order to numerically solve Poisson's equation, it must first be rearranged into a suitable form. Expanding the right side of the equation into components yields

$$-\frac{\rho}{\varepsilon_0} = \frac{\partial}{\partial x}\left(\frac{\partial\Phi}{\partial x}\right) + \frac{\partial}{\partial y}\left(\frac{\partial\Phi}{\partial y}\right) + \frac{\partial}{\partial z}\left(\frac{\partial\Phi}{\partial z}\right) . \tag{6}$$

Assuming the center of the control volume is at location *x,y,z* and integrating both sides of this equation over the volume of a control volume with dimensions Δx in the *x* direction, Δy in the *y* direction, and Δz in the *z* direction, equation (6) becomes

$$\int_{x-\frac{\Delta x}{2}}^{x+\frac{\Delta x}{2}}\int_{y-\frac{\Delta y}{2}}^{y+\frac{\Delta y}{2}}\int_{z-\frac{\Delta z}{2}}^{z+\frac{\Delta z}{2}} -\frac{\rho}{\varepsilon_0}\, dxdydz = \int_{x-\frac{\Delta x}{2}}^{x+\frac{\Delta x}{2}}\int_{y-\frac{\Delta y}{2}}^{y+\frac{\Delta y}{2}}\int_{z-\frac{\Delta z}{2}}^{z+\frac{\Delta z}{2}} \left\{\frac{\partial}{\partial x}\left(\frac{\partial \Phi}{\partial x}\right)\hat{x} + \frac{\partial}{\partial y}\left(\frac{\partial \Phi}{\partial y}\right)\hat{y} + \frac{\partial}{\partial z}\left(\frac{\partial \Phi}{\partial z}\right)\hat{z}\right\} dxdydz \quad . \tag{7}$$

Each of the components on the right side of equation (7) must be integrated separately. Integrating both sides of equation (7) yields

$$-\frac{\rho \Delta x \Delta y \Delta z}{\varepsilon_0} = \frac{\partial \Phi}{\partial x}\Delta y \Delta z + \frac{\partial \Phi}{\partial y}\Delta x \Delta z + \frac{\partial \Phi}{\partial z}\Delta x \Delta y \, . \tag{8}$$

In equation (8), the $\Delta x \Delta y \Delta z$ term is simply the volume of the control volume. Each of the terms on the right side of the equation can be thought of as multiplying the electric potential gradient by the area of the control volume normal to the direction of the gradient. Substituting volume and area and including the indicies, equation (8) can be written as

$$-\frac{\rho_{i,j,k} \mathrm{V}_{i,j,k}}{\varepsilon_0} = \left(\frac{\partial \Phi}{\partial x}\right)_{i,j,k} A_{x,i,j,k} + \left(\frac{\partial \Phi}{\partial y}\right)_{i,j,k} A_{y,i,j,k} + \left(\frac{\partial \Phi}{\partial z}\right)_{i,j,k} A_{z,i,j,k} \, , \tag{9}$$

where A_x is the area normal to the *x*-direction, A_y is the area normal to the *y*-direction, and A_z is the area normal to the *z*-direction. Assigning the node at location *x,y,z* to the indices *i,j,k*, the partial derivate in each of the terms on the right hand side of the equation can be approximated by the following equations which use a center differencing method to approximate the derivative:

$$\left(\frac{\partial \Phi}{\partial x}\right)_{i,j,k} = \left(\frac{\Phi_{i,j,k} - \Phi_{i+1,j,k}}{2\Delta x}\right) + \left(\frac{\Phi_{i-1,j,k} - \Phi_{i,j,k}}{2\Delta x}\right), \tag{10}$$

$$\left(\frac{\partial \Phi}{\partial y}\right)_{i,j,k} = \left(\frac{\Phi_{i,j,k} - \Phi_{i,j+1,k}}{2\Delta y}\right) + \left(\frac{\Phi_{i,j-1,k} - \Phi_{i,j,k}}{2\Delta y}\right), \tag{11}$$

and

$$\left(\frac{\partial \Phi}{\partial z}\right)_{i,j,k} = \left(\frac{\Phi_{i,j,k} - \Phi_{i,j,k+1}}{2\Delta z}\right) + \left(\frac{\Phi_{i,j,k-1} - \Phi_{i,j,k}}{2\Delta z}\right). \tag{12}$$

Substituting equations (10), (11), and (12) into equation (9) and solving for $\Phi_{i,j,k}$ yields

$$\Phi_{i,j,k} * a_{P_{i,j,k,}} = a_{E_{i,j,k,}}\Phi_{i+1,j,k} + a_{W_{i,j,k,}} * \Phi_{i-1,j,k} + a_{N_{i,j,k,}} * \Phi_{i,j+1,k} + a_{S_{i,j,k,}} * \Phi_{i,j-1,k} + a_{T_{i,j,k,}} * \Phi_{i,j,k+1} + a_{B_{i,j,k,}} * \Phi_{i,j,k-1} + S_{i,j,k} \quad (13)$$

where

$$a_{E_{i,j,k,}} = \frac{A_x}{x_{i+1,j,k} - x_{i,j,k}}, \quad (14)$$

$$a_{W_{i,j,k,}} = \frac{A_x}{x_{i,j,k} - x_{i-1,j,k}}, \quad (15)$$

$$a_{S_{i,j,k,}} = \frac{A_y}{y_{i,j,k} - y_{i,j-1,k}}, \quad (16)$$

$$a_{N_{i,j,k,}} = \frac{A_y}{y_{i,j+1,k} - y_{i,j,k}}, \quad (17)$$

$$a_{B_{i,j,k,}} = \frac{A_z}{z_{i,j,k} - z_{i,j,k-1}}, \quad (18)$$

$$a_{T_{i,j,k,}} = \frac{A_z}{z_{i,j,k+1} - z_{i,j,k}}, \quad (19)$$

$$S_{i,j,k} = \frac{\rho_{i,j,k} \mathrm{V}_{i,j,k}}{\varepsilon_0}. \quad (20)$$

and $$a_{P_{i,j,k,}} = a_{E_{i,j,k,}} + a_{W_{i,j,k,}} + a_{S_{i,j,k,}} + a_{N_{i,j,k,}} + a_{B_{i,j,k,}} + a_{T_{i,j,k,}} \quad (21)$$

In equations (14) through (19), $a_{E_{i,j,k,}}$, $a_{W_{i,j,k,}}$, $a_{N_{i,j,k,}}$, $a_{S_{i,j,k,}}$, $a_{T_{i,j,k,}}$, $a_{B_{i,j,k,}}$, and $a_{P_{i,j,k,}}$ are the coefficients utilized to numerically solve Poisson's equation. Due to the uniform, structured nature of the mesh (detailed in section 2.2.1.2) many of these coefficients will be exactly equal because most of the node spacing and control volume sizes are equal. Only the nodes along the boundary of the computational domain and the nodes adjacent to the boundary nodes are different. For the nodes adjacent to the boundary nodes, the area and volume of the control volume remains the same as all the other control volumes, but the distance between nodes will be different. As for the nodes residing on the boundary, they

must be handeled specially in order to implement the boundary conditions. Boundary conditions and their relation to the coefficients are discussed in more detail in section 2.2.1.3.

2.2. Explicit Routine

The three-dimensional PIC code developed in this work is first implemented as a purely explicit PIC routine. This section describes the explicit version of the code in detail. The code is developed using C++ and is compiled and executed using the Microsoft Visual Studio 2010 Professional Integrated Development Environment (IDE). The C++ code outputs results from each simulation in the form of text files which are interpreted and plotted using Matlab R2013b.

The explicit PIC algorithm can basically be split into two different portions, a static portion and a dynamic portion. The static portion is relatively short compared to the dynamic portion of the code. The static portion mostly consists of setting up the magnetic field, creating the mesh, and generating coefficients to be used within the Poisson equation solver. It is not uncommon for PIC codes to assume a static electric field or potential profile within regions of large electric field magnitude; however, no assumptions are made about the electric field within the static portion of this code. The magnetic field is assumed to not change throughout the simulation. This is a good assumption because the current density within the plasma is not significant enough to influence the magnetic fields within the plasma.

The dynamic portion of the code includes the particle injection routine, particle push, electric field (Poisson) solver, and particle-wall interactions. The static portion and the dynamic portion of the explicit routine are described in greater detail in the following sections. A flow chart describing the overall code is shown in Figure 5. Each step within the dotted rectangle occurs within the time step loop, i.e. the dynamic portion of the code.

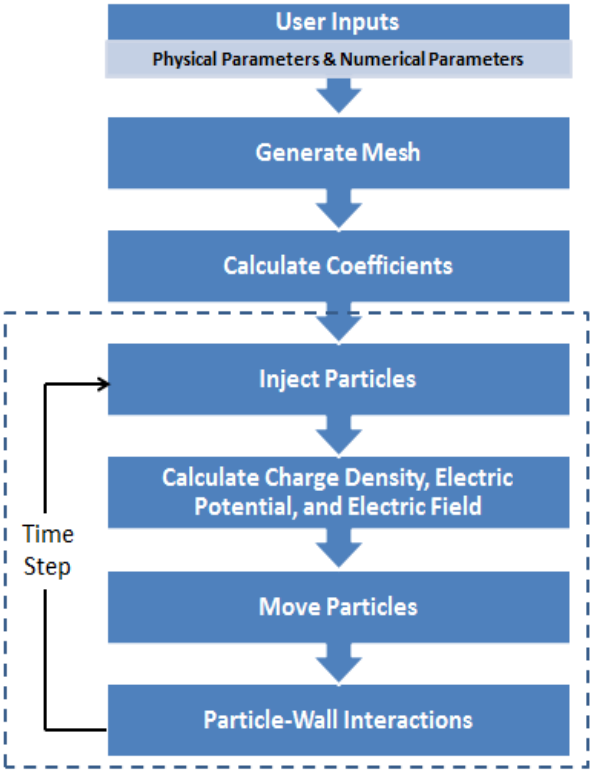

Figure 5. Flow chart of explicit PIC algorithm.

2.2.1. Static Portion

The Static portion of the explicit routine developed in this work mostly focuses on the magnetic fields, generating the mesh, and calculating the coefficients to be used for solving Poisson's equation. The boundary conditions are handled by carefully selecting the values for the coefficients of nodes along the boundary to be used when solving Poisson's equation.

2.2.1.1. User Inputs

In the user inputs section of the code, several numerical and physical parameters are defined, which essentially determines the type of problem to be solved. The user will specify physical parameters such as particle flow rate, particle mass, initial velocities, and boundary conditions. The numerical parameters are also defined by the user in this section which includes the time step, particle weighting, and grid size. The grid size is not directly defined, but rather the number of nodes in each direction is defined in conjunction with the domain size so that the mesh generation section can determine the size of each cell.

The type of plasma which will be studied during the simulation has a lot of bearing on the optimal numerical parameters to be used for the simulation. Unfortunately, the type of plasma present throughout the simulation is generally not known until after the simulation has completed or if a similar simulation has been done before. This being the case, a "base case" (i.e. a reference point) is necessary in order to hone in on the proper numerical parameters by testing them for convergence. A convergence study is performed to address this problem and is presented in Chapter 3.

2.2.1.2. Mesh

The mesh used in this work can be represented as a copious of rectangular prisms fitted together to form a larger rectangular prism known as the computational domain. A structured, uniform mesh is generated and used in this work. When creating the mesh, the number of nodes, and thus the number of control volumes, is specified by the user before beginning the simulation. One of the most important plasma parameters to take into account when defining the number of nodes in each direction is the Debye length, which is generally unknown until the simulation is complete. The Debye length becomes important because of one of the stability criteria concerning how course the mesh is allowed to be (see Section 2.3) largly depends on the Debye length. Particle velocity or thermal velocity also plays a role in determing the maximum allowable grid size as shown by the stability criteria in Section 2.3. Much like the Debye length, the steady state thermal velocity is generally unknown until the completion of the simulation or if a similar simulation has previously been completed. Because of structured nature of the mesh, the only shape this three-dimensional PIC code can simulate is a rectangular prism. An example of the control volumes and nodes which make up the mesh is shown in Figure 6. In this figure, five control volumes (seven nodes) are used in each direction and contained within a cubic domain which is one centimeter in length in each direction. Figure 6 may be somewhat difficult to decipher, so Figure 7 shows the arrangement of the nodes and control volumes as if viewed directly down the z axis, thus in the *x*-*y* plane.

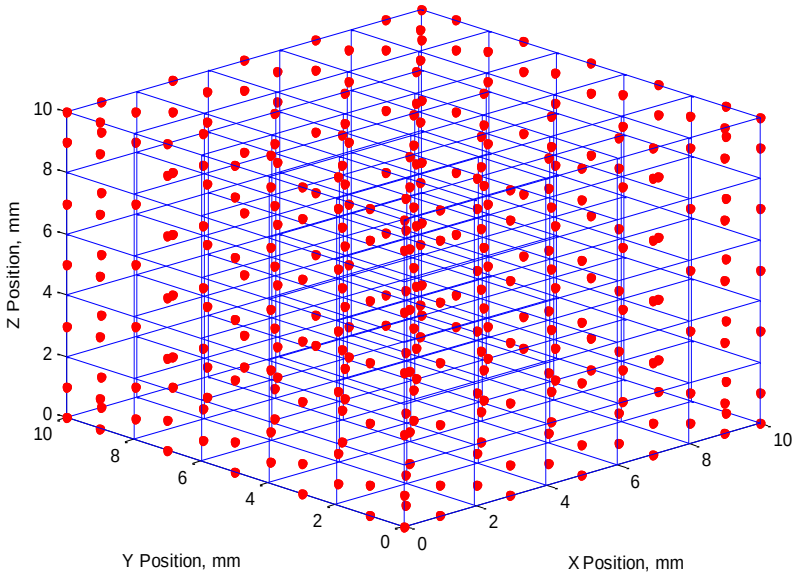

Figure 6. Three-dimensional visualization of control volumes (blue lines) and nodes (red dots).

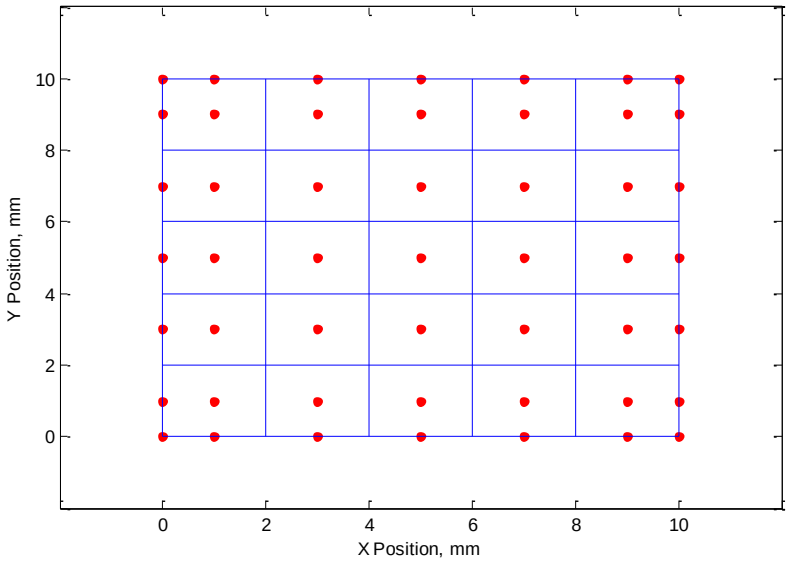

Figure 7. Two-dimensional representation of the control volumes (blue lines) and nodes (red dots).

Basically, each control volume outlined by the blue lines has one node (red dot) at the center of it. The control volumes along the edges of the computational domain have another node residing on the boundary of the domain. The nodes on the domain boundary are the method by which the boundary conditions are inserted into the Poisson solver, which is discussed in more detail in the next section. As a naming convention, a node's position relative to another node is referenced to a cardinal direction or "top" or "bottom". If a node is at the point *i,j,k*, the *i-1* position is referred to as west, the *i+1* position is east, the *j-1* position is south, the *j+1* position is north, the *k-1* position is below, and the *k+1* position is on top.

2.2.1.3. Boundary Conditions

Boundary conditions determine how the potential should behave and what should happen to the computer particles if they should impact the edge of the computational domain. The PIC code developed for this work is capable of simulating two types of boundary conditions: constant potential and dielectric.

The constant potential boundary condition is fairly straightforward. To satisfy this boundary, the nodes residing along the surface of the computational domain are held at a constant value throughout the simulation. When a particle impacts a constant potential boundary, the particle is absorbed by the wall; thus, it is removed from the simulation. Formally, Φ_{wall} is a real number defined at the beginning of the simulation and does not change with time. It is possible for the value of Φ_{wall} to change as a function of time or oscillate at a predetermined frequency to simulate a situation such as an RF oscillator; however, such a condition was not simulated in this work.

The dielectric boundary condition is slightly more complicated to simulate. For a dielectric boundary, the gradient of the potential must be zero such that the electric field magnitude approaches zero at the wall. Formally, if the dielectric wall is located at *x=0* the potential should behave as

$$\frac{d\Phi}{dx} = 0 \; at \; x = 0. \tag{22}$$

When a particle impacts a dielectric boundary, it is reflected back into the domain. If a particle is reflected, the impact is assumed to be perfectly elastic thus conserving the

particle's momentum. The dielectric walls in this simulation also assume no scattering effect so that the particle's angle of incidence is equal to its angle of exit.

In order to distinguish between a fixed potential boundary and a dielectric boundary, the coefficients used to solve Poisson's equation must be carefully assigned. The discretized version of Poisson's equation (equation (13)) shows that seven coefficients and a source term must be defined for each node within the domain. For a boundary held at a constant electric potential, setting the coefficients is very straightforward. For a node along a boundary held at constant potential, the coefficients become

$$a_P = 1\,, \tag{23}$$

$$S = \Phi_{wall}\,, \tag{24}$$

and

$$a_E = a_W = a_N = a_S = a_T = a_B = 0\,. \tag{25}$$

For a dielectric boundary, assigning values to coefficients becomes slightly more complicated. Since the gradient of the potential must be zero at the dielectric boundary, the node along the wall must have the same potential as the node just inside of it. In order to enforce this condition through the coefficients, the node along the wall must effectively receive all its information from the node just inside of it and have no source term (i.e. no charge density). For example, a dielectric boundary on the west wall (x=0 and i=0) will use the following coefficients for any point along that wall.

$$a_P = 1\,, \tag{26}$$

$$S = 0\,, \tag{27}$$

$$a_E = 1\,, \tag{28}$$

and

$$a_W = a_N = a_S = a_T = a_B = 0\,. \tag{29}$$

For nodes which lie on two intersecting dielectric boundaries, the coefficients will essentially split the information between the two neighboring nodes and average them. A

similar method is used when three walls intersect to form a corner. The corner node's information is just the average of the three nearest nodes. If the situation arises that a dielectric wall and a fixed voltage wall share a node along an edge of the domain, the node will be held at the constant potential value instead of using the previously described method to create a dielectric wall. An example of an edge where two dielectric walls meet, such as the southwest edge (*x=0, y=0*) would have coefficients defined as

$$a_P = 1\,, \tag{30}$$

$$S = 0\,, \tag{31}$$

$$a_E = 0.5\,, \tag{32}$$

$$a_N = 0.5\,, \tag{33}$$

and

$$a_W = a_S = a_T = a_B = 0\,. \tag{34}$$

2.2.2. Dynamic Portion

Once the parameters are defined, the mesh is initialized, and the coefficients are established, the dynamic portion of the code begins. The dynamic portion takes place within a single time step and is iterated some number of times as specified at the beginning of the simulation. This is the section inside the dotted rectangle in Figure 5. This portion of the code encompasses the particle injection, charge density calculation, electric potential calculation, electric field calculation, the particle push, and particle-wall interactions. Each of these steps is described in greater detail in the following sections.

2.2.2.1. Particle Injection

When a charged particle is injected into the simulation, both the velocity vector and the position must be defined. Positions are defined by a routine which initializes a set of particles, one electron and one ion, at a single location. Initializing a set of particles as opposed to randomly initializing each particle is done to better mimic an ionization event in which one electron and one ion are created from the same location. The placement of the set

of particles is random within the computational domain. However, particles are not injected at every space in the domain but rather a short distance away from the wall with the offset being 1 mm. This small offset from the walls is done for two reasons. Physically, the sheaths develop near walls held at a constant potential and the ionization rate within the sheath region of plasma is reduced as compared to the bulk of the plasma. Numerically, injecting particles right up to the wall can cause an unnatural loss rate of particles (typically ions) and numerical heating of other particles (typically electrons). The sheaths near the absorbing walls set up an electric field that drives ions toward the absorbing wall while driving electrons away from the absorbing wall. This electric field is the reason ions are lost too quickly and electrons are given too much energy if particle injection takes place in the sheath region. The actual value of the offset is chosen with regards to the approximate size of the sheath, thus the optimal offset will change as a function of the type of plasma being simulated.

The velocity of each computer particle is initialized by randomly assigning a velocity magnitude from a given range of velocities defined at the beginning of the simulation. The velocity vector is determined by generating a random unit vector. Most commonly in this work, the range of velocity magnitudes is a single number, thus every particle's initial speed is identical for a given species, but moves in a random direction.

In this work, a constant injection rate of particles is used. For each time step, a user defined number of computer particles are injected into the simulation using the method outlined above. It is up to the user to determine the proper number of computer particles to inject each time step, as well as the particle weighting in order to define a certain physical situation.

2.2.2.2. Charge Density

In order to determine how the particles within the simulation affect the electric fields, the charge density must be ascertained as the simulation progresses. Several methods exist for calculating the charge density within a fully kinetic PIC routine, but the main difference between the methods is the order of the approximation. Initially, the code developed in this work used a zero order weighting scheme which delegates all the charge of a single particle to whichever node is nearest. As was predicted by Birdsall and Langdon (1991), this method

of calculating charge density resulted in the electric potential containing large peaks. One way to think of zero order weighting is to imagine the particle as a hard cube which can only reside with its center at exactly a node position.

To resolve this issue, a first order weighting scheme was implemented. The first order weighting scheme uses linear weighting to distribute the charge of a single particle to the 8 nearest nodes based on the particle's relative position to each of those nodes. The following figure adopted and modified from Mahalingam (2007) illustrates this principle in two dimensions. The three-dimensional model used in the code is somewhat difficult to visualize, but it is essentially the two dimensional case shown in Figure 8 in addition to another dimension perpendicular to the plane shown in Figure 8.

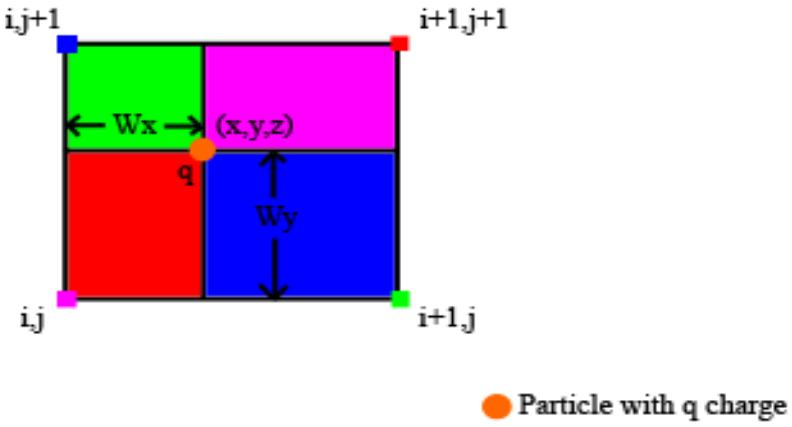

Figure 8. First order weighting scheme for charge density calculation.

With first order weighting, the particle can be thought of as a cubic cloud of charge which can reside anywhere within the computational domain. At any one time, the center of this cloud of charge will reside between eight nodes when using three dimensions. For the two dimensional example shown in Figure 8, the values W_x and W_y are used in determining how much of charge q is distributed among the four nodes shown. Similarly, W_x, W_y, and W_z are used to determine how much of charge q is distributed among the eight closest nodes. W_x, W_y, and W_z are defined as

$$W_x = x_p - x_{i,j,k} \, , \tag{35}$$

$$W_y = y_p - y_{i,j,k} \, , \tag{36}$$

and

$$W_z = z_p - z_{i,j,k} \, . \tag{37}$$

where the subscript *p* refers to the particle's location and the subscript *i,j,k* refers to the location of the nearest node which is south, west, and below the location of the particle. The equations for the amount of charge each node accumulates from a single charged particle are

$$\Delta q_{i,j,k} = q_p(1 - W_x)(1 - W_y)(1 - W_z) \, , \tag{38}$$

$$\Delta q_{i+1,j,k} = q_p W_x(1 - W_y)(1 - W_z) \, , \tag{39}$$

$$\Delta q_{i,j+1,k} = q_p W_y(1 - W_x)(1 - W_z) \, , \tag{40}$$

$$\Delta q_{i,j,k+1} = q_p W_z(1 - W_x)\big(1 - W_y\big), \tag{41}$$

$$\Delta q_{i+1,j+1,k} = q_p W_x W_y(1 - W_z) \, , \tag{42}$$

$$\Delta q_{i+1,j,k+1} = q_p W_x W_z(1 - W_y) \, , \tag{43}$$

$$\Delta q_{i,j+1,k+1} = q_p W_y W_z(1 - W_x) \, , \tag{44}$$

and

$$\Delta q_{i+1,j+1,k+1} = q_p W_x W_y W_z \, . \tag{45}$$

This process is repeated for each particle within the computational domain in order to determine the amount of charge assigned to each node. The change in charge accumulation for each node is summed to determine the charge density used in the source term calculation for that node. It is important to note that the *p* subscript on the charge indicates that the charge can be either positive or negative based on the type of particle.

Special care must be used when linearly weighting these particles near a wall. If a particle is between the wall and another node within the computational domain, the charge of that particular particle is partially allocated to the wall, regardless of what type of wall the particle is near (dielectric or constant voltage). This is done to prevent the charges near the walls from attributing more charge to one of its nearest nodes than would occur at any other point within the computational domain. After the charge density is found for the entire domain, the next step within the dynamic portion of the code is to determine the electric potential and electric fields.

2.2.2.3. Electric Field

Once the charge density has been calculated for each node location within the three-dimensional mesh, the electric fields can be determined. Before assigning electric field strength and direction to each control volume, the electric potential must be determined. The electric potential must be continuous and the electric potential at any one point is influenced by the electric potentials around that point as well as the charged particles near it (Griffiths, 1999). In the case of the code developed in this work, the point refers to a node within the mesh and the potentials near that point are the nearest nodes in every direction.

Using a structured mesh allows the potentials throughout the domain to be represented as a rectangular matrix. Given that the potentials can be stored as a matrix, one way to calculate the electric potential is to use the tri-diagonal matrix algorithm (TDMA) to solve the discretized Poisson equation (equation (13)). The TDMA solver is an iterative routine which must be executed along different directions. However, since the code developed in this work is three-dimensional, the TDMA solver sweeps in all three directions separately before checking for convergence. The convergence criteria used for this iterative routine is $1x10^{-6}$ and more details about the TDMA solver can be found in Appendix A.

Once the electric potentials are known at all points in space, the electric field is determined according to equation (3). This is done by using a linear average of the electric potential gradients in each direction. Since the mesh used in this work is uniform, the linear averaging method used to find the gradient of the potential is essentially the center differencing method. For each component separately, the electric field is defined as

$$E_{x,\ i,j,k} = \frac{\frac{\Phi_{i,j,k}-\Phi_{i+1,j,k}}{x_{i+1}-x_i}+\frac{\Phi_{i-1,j,k}-\Phi_{i,j,k}}{x_i-x_{i-1}}}{2}, \tag{46}$$

$$E_{y,\ i,j,k} = \frac{\frac{\Phi_{i,j,k}-\Phi_{i,j+1,k}}{y_{j+1}-y_j}+\frac{\Phi_{i,j-1,k}-\Phi_{i,j,k}}{y_j-y_{j-1}}}{2}, \tag{47}$$

and

$$E_{z,\ i,j,k} = \frac{\frac{\Phi_{i,j,k}-\Phi_{i,j,k+1}}{z_{k+1}-z_k}+\frac{\Phi_{i,j,k-1}-\Phi_{i,j,k}}{z_k-z_{k-1}}}{2}. \tag{48}$$

2.2.2.4. Particle Advance

The technique used in this work for pushing particles each time step is known as the leap frog scheme with a Boris advance. The leap frog scheme can be very briefly described as advancing the particles using half the electric field, adjusting the velocity vector due to the magnetic field (also called rotation), then finishing with the second half of the electric field push. The equations utilized for the leap frog scheme are given from the time centered finite difference equation version of equation (2) as

$$\frac{\vec{v}_p^{\,n+1/2}-\vec{v}_p^{\,n-1/2}}{\Delta t} = \frac{q}{m}\left(\vec{E}^n_{i,j,k} + \frac{\vec{v}_p^{\,n+\frac{1}{2}}+\vec{v}_p^{\,n-\frac{1}{2}}}{2}\times \vec{B}^n_{i,j,k}\right). \tag{49}$$

In order to implement the leap frog scheme, the following equations must be applied to each particle and solved in the following order:

$$\vec{v}_p^{\,n-1/2} = \vec{v}_p^{\,-} - \frac{q_p\vec{E}^n_{i,j,k}\Delta t}{2m_p}, \tag{50}$$

$$\vec{t}_p = \frac{q_p\vec{B}^n_{i,j,k}\Delta t}{2m_p}, \tag{51}$$

$$\vec{s}_p = \frac{2\vec{t}_p}{1+t_p^2}, \tag{52}$$

$$\vec{v}_p' = \vec{v}_p^- + \vec{v}_p^- \times \vec{t}_p \,, \tag{53}$$

$$\vec{v}_p^+ = \vec{v}_p^- + \vec{v}_p' \times \vec{s}_p \,, \tag{54}$$

$$\vec{v}_p^{n+1/2} = \vec{v}_p^+ + \frac{q_p \vec{E}_{i,j,k}^n \Delta t}{2m_p}, \tag{55}$$

and

$$\vec{x}_p^{n+1} = \vec{x}_p^n + \vec{v}_p^{n+1/2} \Delta t \,. \tag{56}$$

When beginning the leap frog routine, the only known quantities are the previous time step's velocity, electric field, time step size, mass of the particle, charge of the particle, and the magnetic field. Many of the variables introduced in the leap frog scheme serve only as intermediate steps on the way to finding the velocity vector used during the total particle advance, which is the last step of the leap frog routine. It should also be noted that in equation (50), the variable actually being solved for is $\vec{v}_p^-$ which corresponds to the first half of the electric field push.

Equations (50) through (56) can essentially be divided into four parts. First, half of the electric field is applied to find an intermediate velocity. Next, the magnetic field is applied which alters the trajectory of the particle, i.e. changing velocity vector but not magnitude. Third, the second half of the electric field is applied to the velocity which has been affected by the magnetic field. Lastly, the newly calculated velocity which takes into account both the electric field and the magnetic field is used to advance the particle to its new location.

2.3. Explicit Stability Criteria

The explicit PIC algorithm is subject to a number of stability criteria which limit its performance capabilities. Mostly the criteria define limits to the numerical parameters which include the maximum time step size, maximum grid spacing, and minimum number of particles per cell. The minimum number of particles per cell restriction affects the numerical parameter for particle weighting which is how many real particles are represented by a single computer particle. These maximum values typically depend on characteristics of the plasma

such as plasma frequency, Debye length, and electron velocity or electron temperature. The criteria which apply to the explicit code are

$$N_{\frac{comp\ particles}{cell}} \geq 20, \tag{57}$$

$$v_{p,x}\Delta t \leq \Delta x, \tag{58}$$

$$v_{p,y}\Delta t \leq \Delta y, \tag{59}$$

$$v_{p,z}\Delta t \leq \Delta z, \tag{60}$$

$$\omega_{pe}\Delta t \leq 2, \tag{61}$$

and

$$\frac{\lambda_D}{0.3} \leq \min(\Delta x, \Delta y, \Delta z). \tag{62}$$

In equations (58) through (62) λ_D is the Debye length, ω_{pe} is the plasma frequency, $v_{p,x}$, $v_{p,y}$, and $v_{p,z}$ are the particle velocities in each of the three Cartesian directions, and $\Delta x, \Delta y$ and Δz are the control volume sizes in each of the three Cartesian directions. The plasma frequency and the Debye length are defined as

$$\omega_{pe} = \sqrt{\frac{n_e q_e^2}{\varepsilon_0 m_e}} \tag{63}$$

and

$$\lambda_D = \sqrt{\frac{\varepsilon_0 k_b T_e}{n_e q_e^2}}. \tag{64}$$

In equations (63) and (64), n_e is the number density of electrons, T_e is the electron temperature, and k_b is the Boltzmann constant. Equation (57) requires that at least 20 computer particles are present in each cell for statistical purposes. Equations (58) through (60) essentially restrict the particle to moving no more than one cell in a single time step. Since the electric field and magnetic field within a single cell is valid only for that cell, if the particle moves more than one cell an incorrect value for the electric field will be utilized for

part of the particle advance. Equation (61) must be satisfied in order to resolve Langmuir wave propagation (Lapenta, 2006). The final stability criterion, shown in equation (62) dictates the grid size based on the Debye length of the electrons in the plasma. This stability criterion arises in order to capture shielding of the positive charges by the negative charges.

CHAPTER 3

EXPLICIT MODEL AND VERIFICATION

In order to evaluate the effectiveness of the semi-implicit techniques investigated in this work, the explicit version of the PIC algorithm must first be tested and verified. The previous chapter described the explicit version of the PIC code used for this work. Once a functioning version of the code has been produced (as shown in Appendix B), a base case needs to be established with which to compare the semi-implicit routines to the explicit routine. Before the base case can be established, a convergence study must be conducted which will determine acceptable physical and numerical parameters.

Determining proper parameters involves completing several simulations with the same physical situation but different numerical parameters. A numerical parameter is considered to be converged when the steady state solution is no longer a function of the numerical parameter. This condition should apply to all the numerical parameters so that the solution is not a function of any of them. The selection of the physical parameters (i.e. defining the problem to be solved) is based largely on time and resource constraints rather than trying to solve an engineering problem.

Once acceptable numerical and physical parameters have been identified, they are used to simulate a "base case" which will serve as a baseline comparison for each of the techniques to be investigated. The following section outlines the process for identifying the acceptable values of each numerical parameter by performing a convergence study.

3.1. Convergence Study

The PIC algorithm requires three numerical parameters to be defined before beginning the simulation: time step, particle weighting, and grid size. Stability guidelines exist for each of these three parameters and were outlined in the previous chapter. Since the optimal choice for many of these numerical parameters is determined by characteristics of the plasma, the acceptable parameters must be determined through a trial and error process.

The time step is the physical amount of time each iteration represents. The maximum allowable time step size was defined in the previous chapter (equations (58) to (61)) and depends on the plasma frequency, grid size, and particle velocity (particle temperature). Of the three numerical parameters presented here, the time step is of the most interest for this work since the goal is to increase its maximum allowable size.

The particle weighting defines how many physical particles one computer particle represents. The particle weighting mainly affects the number of particles per cell stability condition that was previously stated. The total number of computer particles greatly affects the overall computational time since most of the computational power is spent advancing the particles. Thus, using the fewest number of particles (i.e. the largest particle weighting) to adequately model the plasma is the most efficient particle weighting choice for the simulation.

Lastly, the grid size defines how large each control volume will be for the simulation. Finding the convergence criteria for grid size is somewhat difficult because it directly affects the stability criteria for both the time step and the particle weighting. A smaller grid size means more control volumes for a given computational domain which means the particle weighting must decrease in order to maintain the same number of computer particles per cell. The time step will also need to decrease with decreasing grid size in order to prevent particles from traversing more than one cell in a single time step.

For all the simulations run for the convergence study, the physical setup of the problem is the same; only the numerical parameters are altered in order to determine their acceptable values. The physical characteristics of the problem and boundary conditions are outlined in Figure 9, Table 1, and Table 2.

Table 1. Physical parameters for convergence studies.

Physical Parameter	Value
Ion injection rate	2.5×10^{14}, #/s
Electron injection rate	2.5×10^{14}, #/s
Computational domain volume (cube)	0.01 cm x 0.01 cm x 0.01 cm= 1×10^{-6} m^3
Initial electron velocity	100,000, m/s
Initial ion velocity	1,000, m/s
Electron mass	9.109×10^{-31}, kg
Ion mass (Hydrogen)	1.673×10^{-27}, kg
Permittivity of free space	8.854×10^{-12}, F/m
Number of time steps	10,000

Table 2. Boundary conditions for convergence studies.

Wall Location	Boundary Condition
West wall (x = 0 m)	Absorbing wall, 3V
East wall (x = 0.01 m)	Absorbing wall, 6V
South wall (y = 0 m)	Dielectric wall
North wall (y = 0.01 m)	Dielectric wall
Bottom wall (z = 0 m)	Dielectric wall
Top wall (z = 0.01 m)	Dielectric wall

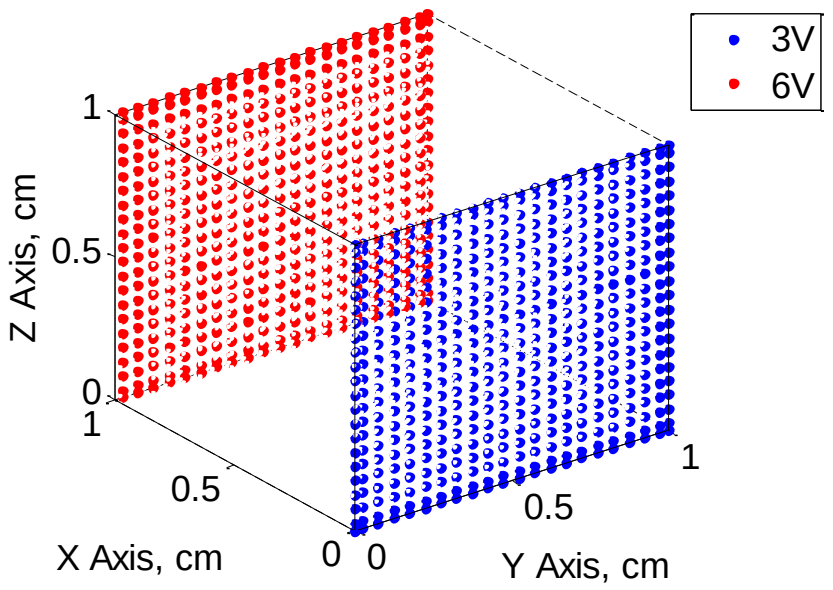

Figure 9. Boundary conditions for the convergence studies.

Once the physical problem has been defined, the acceptable time step size can be found by simulating the physical problem described above using different time steps. The steady state result used to compare the different time steps is the steady state electron number density. This plasma characteristic is used for comparison because it is generally more sensitive to the numerical parameters than other metrics such as the difference or percent difference between ions and electrons. Figure 10 shows the steady state electron number densities using different time steps. The steady state results are determined using an average of the last 5% of the simulation. The electron number density is an overall average density for the domain which is found by

$$Electron\ Number\ Density = \frac{N_e * W_p}{V_{total}} \tag{65}$$

where N_e is the total number of electrons in the computational domain and V_{total} is the total volume of the computational domain.

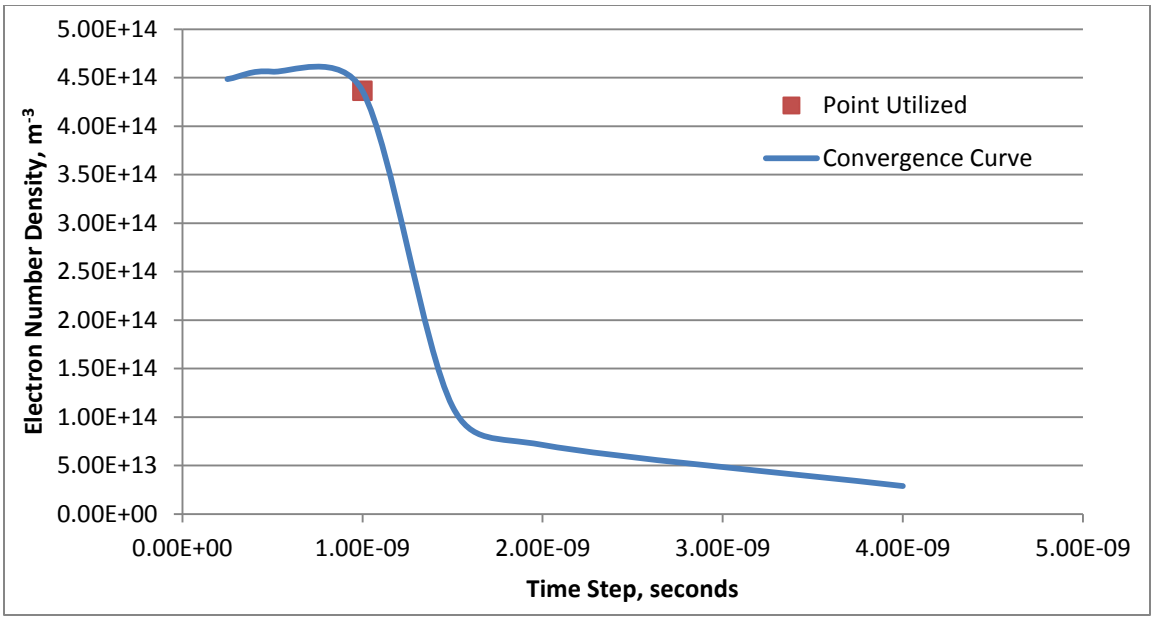

Figure 10. Time step convergence study.

Six simulations were completed to create the convergence curve shown in Figure 10. The time steps used were $2.5x10^{-10}$, $5x10^{-10}$, $1x10^{-9}$, $1.5x10^{-9}$, $2x10^{-9}$, and $4x10^{-9}$. The further to the left on the x axis in Figure 10, the smaller the time step and the more accurate the solution. Using a time step of $1x10^{-9}$ causes the electron number density to be converged within about 2.72% of the smallest time step tested. A very rapid change occurs in Figure 10 between the time steps of $1x10^{-9}$ and $1.5x10^{-9}$ where the steady state electron number density significantly decreases. This is likely due to the average electron velocity causing the electrons to traverse more than one cell (~2.48) per time step in the $1.5x10^{-9}$ time step simulation while the average electron velocity in the $1x10^{-9}$ time step simulation causes the particle to traverse less than one (~0.57) cell in a single time step. These values are obtained using a grid size of 0.0005 m. This significant change in number density is a manifestation of numerical heating which is caused by using too large of a time step.

Next, the grid size convergence study is conducted using the same physical setup as the time step convergence study. For this study, the time step is held constant at $1x10^{-9}$, but the particle weighting and thus the number of particles injected each time step changes in an effort to satisfy the stability criteria for the minimum number of computer particles per cell defined in equation (57). Since grid size is defined in the computer program by specifying the

number of nodes in each direction this is the number used to state grid size. For the grid study, 5 simulations are completed using the numerical parameters outlined in Table 3.

Table 3. Numerical parameters for grid size convergence study.

Control volumes in each direction	Grid size (control volume edge length), m	Particle weighting, physical particles / computer particle	Injection Rate, Computer Particles / Time Step
5	0.002	10000	25
10	0.001	4000	63
20	0.0005	500	500
40	0.00025	125	2000
80	0.000125	31	8000

Again using the electron number density as a metric, the results of these 5 runs are shown in Figure 11 as a function of the number of control volumes in each direction.

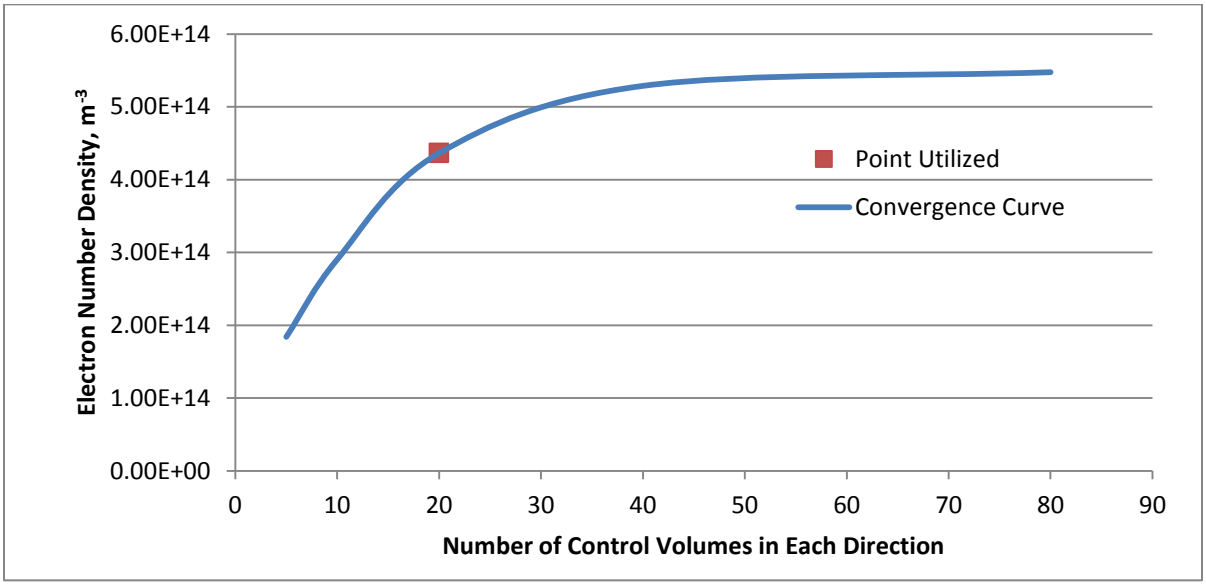

Figure 11. Grid size convergence study.

Using 20 control volumes in each direction, which corresponds to a grid size of one half millimeter, the result is converged within 20.4%. The 40 control volume simulation is converged within 3.5%. Ideally, the base case would use 40 control volumes or even 80

control volumes, but 20 control volumes is chosen in the interest of computational time. Doubling the number of control volumes in each direction increases the total number of control volumes by a factor of 8. Not only does this cause the field solver to take more computational time, but the number of computer particles must also increase by a factor of 8 in order to maintain a sufficient number of computer particles per cell. Doubling the number of control volumes in each direction should also warrant the maximum time step to decrease by a factor of 2 due to the stability criteria outlined in equations (58), (59), and (60). Essentially, doubling the number of control volumes in each direction does much more than double the computational time. Practically, halving the grid size increases the computational time requirement approximately by a factor of 18 if all the other numerical parameters are adjusted accordingly.

Thus far, we have chosen numerical parameters of 20 control volumes in each direction (0.0005m grid size) and a time step of $1x10^{-9}$ seconds. The last numerical parameter which must be found via a convergence study is the particle weighting, or more specifically, the number of computer particles per cell. The stability criterion shown in equation (57) indicates this number should be at least 20 computer particles per cell. However, this number is more of a general guideline and applies to stability, not accuracy. The stability criterion also applies to a two dimensional simulation, whereas the code developed in this work is three-dimensional, so the required number of computer particles per control volume may be different than the number of computer particles stated by equation (57). Table 4 summarizes the simulations completed for the particle weight convergence study while Figure 12 shows the steady state electron number density of each simulation versus the number of computer electrons per cell.

Table 4. Numerical parameters for particle weighting convergence study.

Particle Weighting, Real Particles / Computer Particle	Computer Electrons / Cell	Computer Ions / Cell
5000	5.9	6.5
2000	19.5	20.9
1000	46.9	49.6
500	109.1	114.5
250	222.2	233.3
125	452.3	474.4

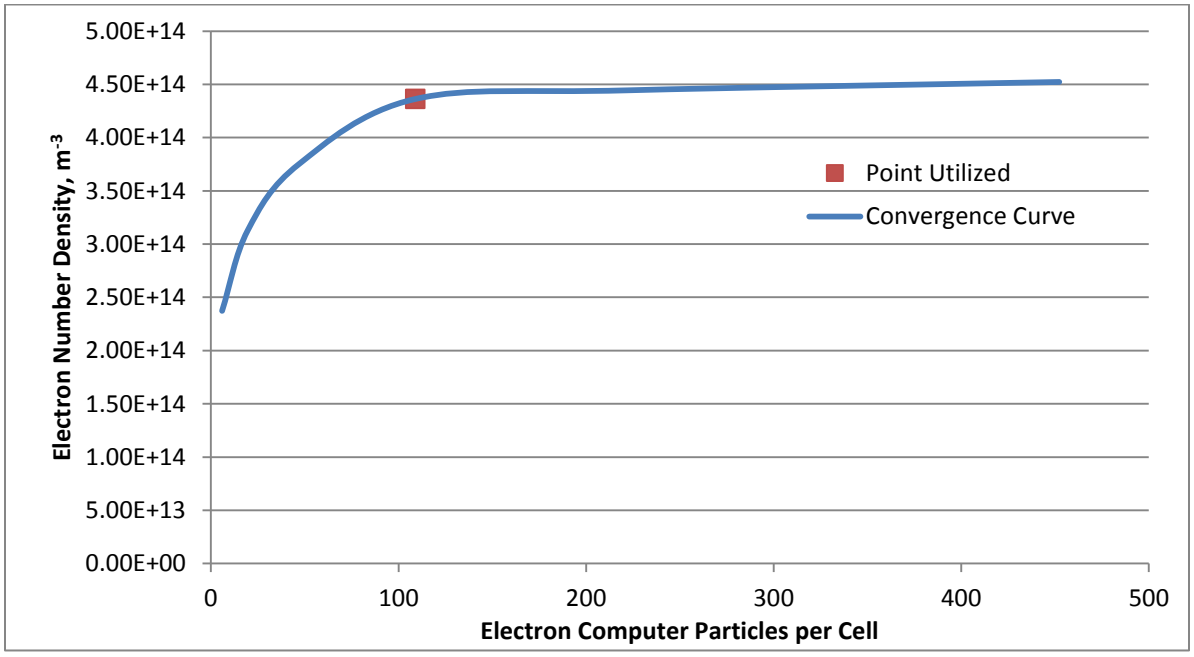

Figure 12. Steady state results from particle weighting convergence study.

Using the results from the particle weighting convergence study, we can see that the more computer particles per cell, the more converged the solution. In choosing the particle weighting of 500 real particles per computer particle, the steady state solution has an average of 109.1 computer electrons and 114.5 computer ions per cell. The steady state result using this weight is converged within about 3.5% of the smallest particle weight tested.

The base case will use the numerical parameters of: a time step of $1x10^{-9}$ seconds, a particle weighting of 500 real particles per computer particle, and 20 control volumes in each direction which corresponds to a grid spacing of 0.0005 meters. Ideally the number of control volumes in each direction would be higher but as previously stated, practical issues of computational time develop with a large number of control volumes, especially since the code is in three dimensions. Even though smaller control volumes would be desirable, the stability condition shown in equation (62) is near its limit of satisfaction.

3.2. Base Case

Before either of the novel semi-implicit techniques explained in Chapter 4 can be tested, the explicit version of the code must first be verified. In order to check the explicit code's results, a "base case" simulation is performed using the numerical parameters outlined in the previous section and the physical parameters shown in Table 5. The boundary conditions used for the base case are the same boundary conditions used for the convergence studies and are summarized in Table 6.

Table 5. Physical and numerical parameters for base case simulation.

Physical Parameter	Value
Ion injection rate	$2.5x10^{14}$ #/s
Electron injection rate	$2.5x10^{14}$ #/s
Particle weighting	500 real particles/computer particle
Time step	$1x10^{-9}$ s
Computational domain volume (cube)	$1x10^{-6}$ m^3 (1 cm^3)
Control volume edge length (Δx)	0.0005 m
Nodes in a single direction	22
Initial electron velocity	100,000 m/s
Initial ion velocity	1,000 m/s
Ion mass (hydrogen)	$1.673x10^{-27}$ kg
Electron mass	$9.109x10^{-31}$ kg
Permittivity of free space	$8.854\ x10^{-12}$ F/m
Total simulation time	$1x10^{-5}$ s therefore 10,000 time steps

Table 6. Boundary conditions for base case simulation.

Wall Location	Boundary Condition
West wall (x=0 m)	Absorbing wall, 3V
East Wall (x=0.01 m)	Absorbing wall, 6V
South Wall (y=0 m)	Dielectric wall
North Wall (y=0.01 m)	Dielectric wall
Bottom wall (z=0 m)	Dielectric wall
Top wall (z=0.01 m)	Dielectric wall

Using the values shown in Table 5 and Table 6, the explicit base case simulation yields the following results shown in Figure 13 through Figure 17. Using the steady state results from the base case, several plasma parameters can be determined including the particle number densities, particle temperatures, electrical potentials, average Debye length, and average plasma frequency. These steady state plasma parameters are displayed in Table 7.

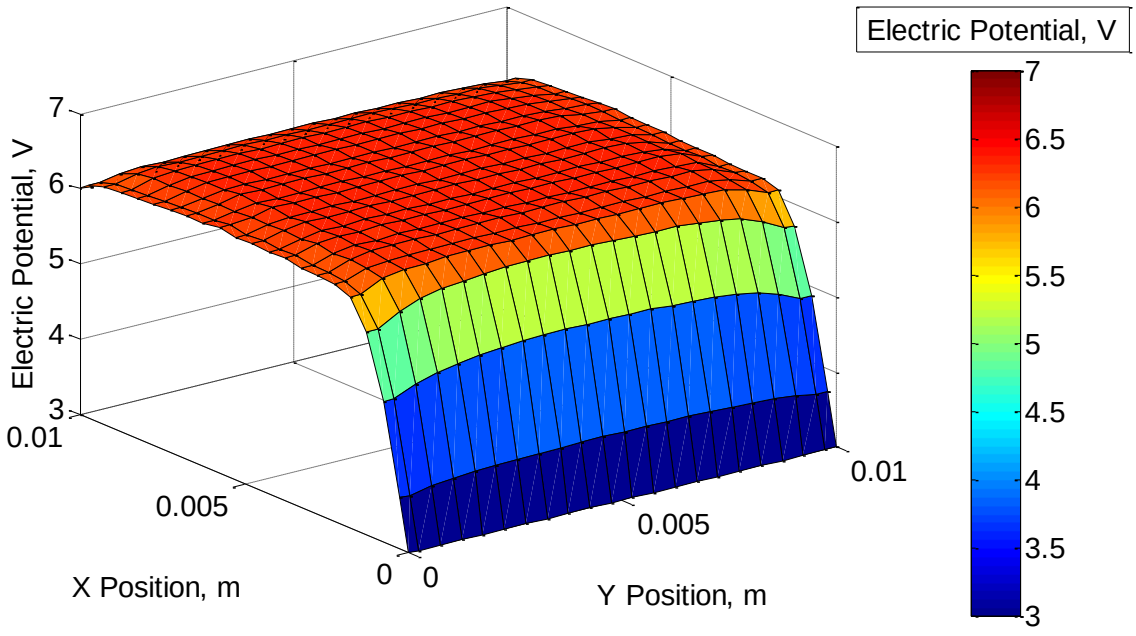

Figure 13. Electric potential contour plot at z = 0.00475 m for the explicit base case.

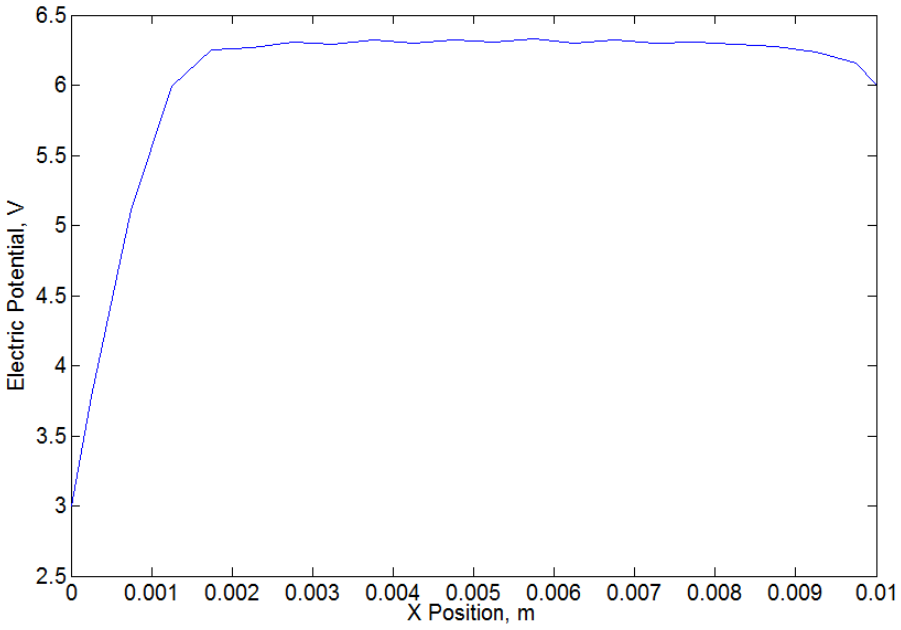

Figure 14. Electric potential profile in the x-direction at y = 0.00475 m and z = 0.00475 m for explicit base case.

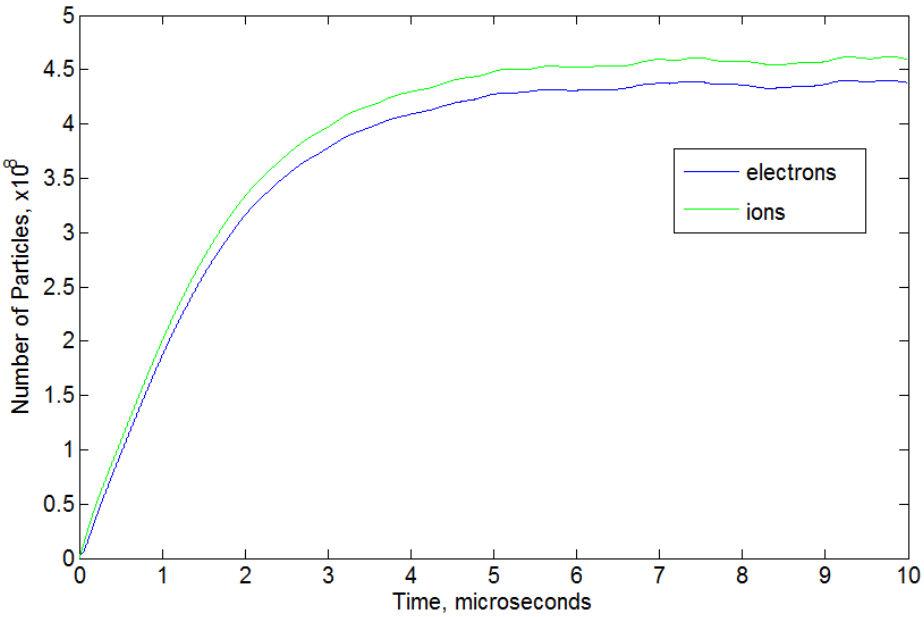

Figure 15. Total number of particles in the computational domain as a function of time for explicit base case.

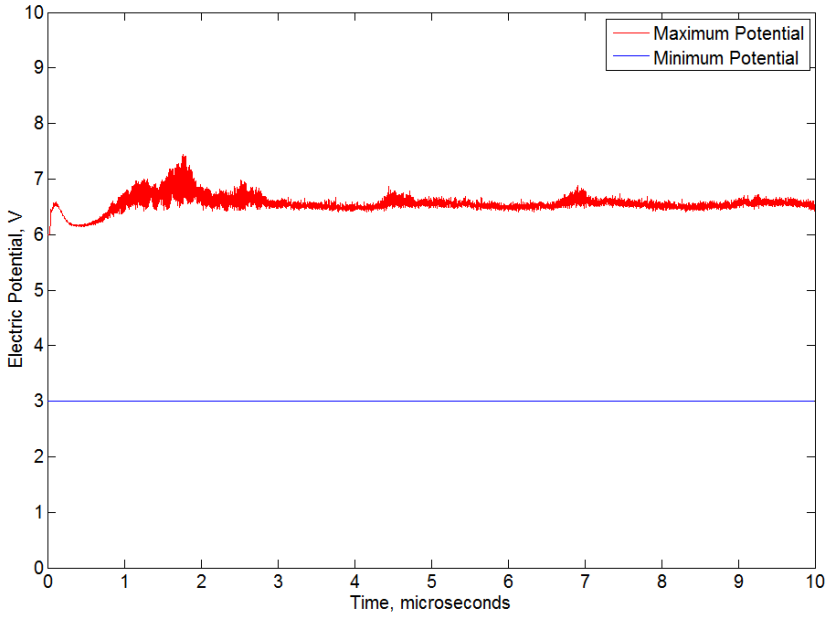

Figure 16. Maximum and minimum electric potential in the entire computational domain for explicit base case.

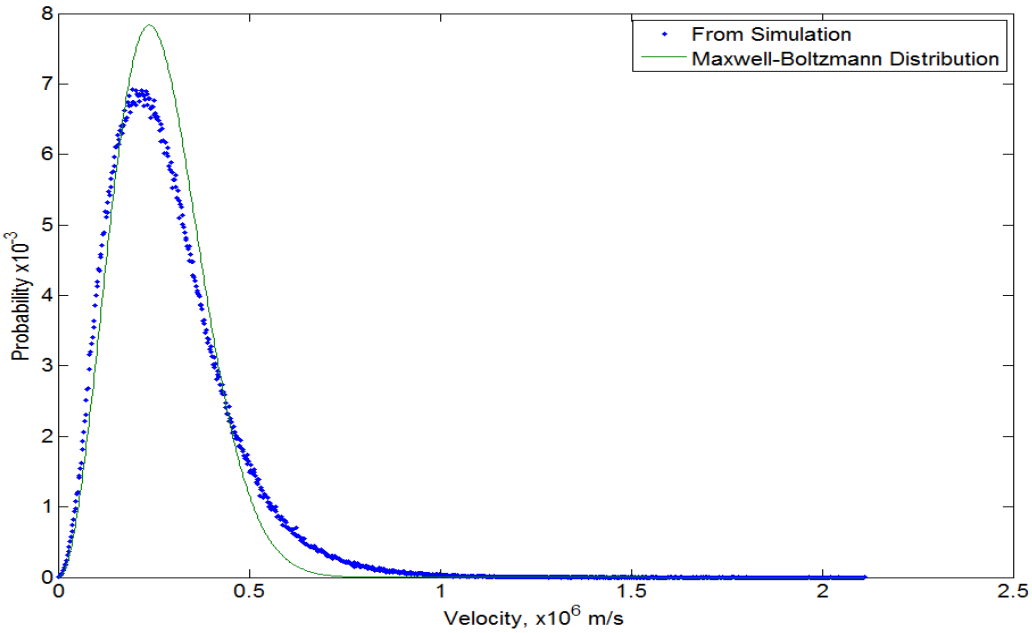

Figure 17. Electron Velocity distribution for explicit base case.

Table 7. Plasma parameters from the base case simulation.

Plasma Parameter	Value
Electron Number Density	4.393×10^{14} #/m^3
Ion Number Density	4.609×10^{14}) #/m^3
Electron Temperature	0.1539 eV
Ion Temperature	0.0339 eV
Plasma Potential	6.329 V
Debye length	0.1392 mm
Plasma frequency	1.183×10^{9} s^{-1}

The contour plot in Figure 13 shows the electric potentials of a plane across the center of the computational domain. The plane shown is an *x*-*y* plane at $z = 0.00475$ m which is as close to the center of the domain that is possible with an even number of control volumes at 0.0005 m each. The values shown in Figure 13 are averaged over the last 5 percent of the simulation. Figure 14 uses the values from Figure 13 and averages all the potentials in the *y* direction for a given *x* position, effectively giving a one dimensional plot of the electric potential as a function of *x*. Figure 15 shows the total number of physical particles for both ions and electrons within the computational domain as a function of time. Figure 16 shows the maximum and minimum electric potential throughout the entire domain as a function of time. The minimum potential is a constant 3 volts throughout the entire simulation because of the 3 volt boundary condition on the west wall. Figure 17 shows the velocity distribution of the electrons on the last time step of the simulation plotted with a Maxwell-Boltzmann velocity distribution at the same temperature. As is shown in Figure 17, the Maxwell-Boltzmann distribution and the distribution obtained from the simulation line up well, despite the fact that the particles are not injected using this distribution and collisions are not included in the model. The theoretical Maxwell-Boltzmann velocity distribution is (Goebel & Katz, 2008)

$$f(v) = \sqrt{\left(\frac{m_p}{2\pi k_b T_p}\right)^3} 4\pi v_p^2 e^{-\frac{m_p v_p^2}{2k_b T_p}} \tag{66}$$

3.3. Analytical Model Verification

In order to check the explicit code's results, the plasma potential is checked against a theoretical solution for the floating potential. The floating potential is determined iteratively using

$$1 - \frac{|q_e|\Phi_{fl}}{k_b T_{ion}} = \sqrt{\frac{m_i T_e}{m_e T_{ion}}} \frac{n_e}{n_{ion}} e^{\frac{|q_e|\Phi_{fl}}{k_b T_e}}. \tag{67}$$

In equation (67), Φ_{fl} is the floating potential relative to the wall potential, T_e is the electron temperature, n_{ion} is the ion number density, and T_{ion} is the ion temperature. Using the particle number density and temperature results from the steady state condition of the explicit base case, equation (67) was used to determine an estimate of the plasma potential for the base case simulation. In equation (67), the floating potential is the wall potential relative to the plasma, which is a negative number. The absolute value of Φ_{fl} is added to the 6 volt wall potential to obtain the floating potential to compare to the simulation output. Solving equation (67) yields a floating potential of 6.325 volts, while the simulation reports a plasma potential of 6.329 volts. The analytical result compares very nicely to the explicit PIC code's prediction of the plasma potential. The plasma potential reported by the simulation is found by taking the time-averaged potential of the last 5% of the run for every point in the computational domain and choosing the maximum value of these time averaged points.

Another check performed on the explicit PIC code developed for this work was to compare electric potential shapes between the explicit PIC code and those obtained by analytically solving for the Debye sheath defined as

$$\Phi(x) = \Phi_w e^{-x/\lambda_D}. \tag{68}$$

In equation (68), Φ_w is the biased wall potential relative to the plasma potential and *x* is the distance from the wall, note that this has a different meaning than *x* used in the PIC code. In order for equation (68) to be valid, the conditions for the potential and temperature

$$\left|\frac{q\Phi}{k_b T_e}\right| \ll 1 \tag{69}$$

and the velocity distribution function must satisfy equation (69). These conditions are not satisfied using the base case conditions due to the low electron temperature and the uniform initial velocities of the particles. To produce a case to verify the explicit PIC algorithm against equation (68), a higher electron temperature with a Maxwell-Boltzmann velocity profile, as shown in equation (66), must be used. Thus, for the results shown in Figure 18 the electrons are injected at a temperature of 5 eV with a Maxwell-Boltzmann distribution while the ions are injected at a low temperature of 0.05 eV, also with a Maxwell-Boltzmann distribution. As shown in Figure 18, comparisons between the explicit PIC results and those from equation (68) are excellent. This, along with the plasma potential comparison, verifies that the explicit PIC code used for this work was programmed correctly.

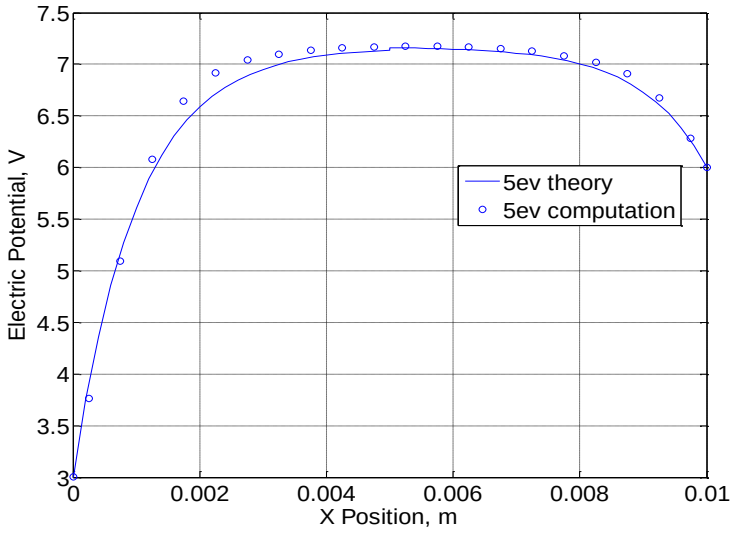

Figure 18. Analytical comparison to the explicit PIC code results.

3.4. Full Three-Dimensional Case

In order to display the three-dimensional capability of the code, a full three-dimensional simulation is performed. The physical parameters of the particle injection rate, particle masses, and initial velocities, as well as most of the numerical parameters, are kept the same as in the base case. The most notable changes to the physical parameters in the full

three-dimensional case are the boundary conditions which are summarized in Table 8 and Figure 19.

Table 8. Boundary conditions of full three-dimensional case.

Wall Location	Boundary Condition
West wall (x=0 m)	Absorbing wall, 2V
East Wall (x=0.01 m)	Absorbing wall, 6V
South Wall (y=0 m)	Absorbing wall, 0V
North Wall (y=0.01 m)	Dielectric wall
Bottom wall (z=0 m)	Absorbing wall, 4V
Top wall (z=0.01 m)	Dielectric wall

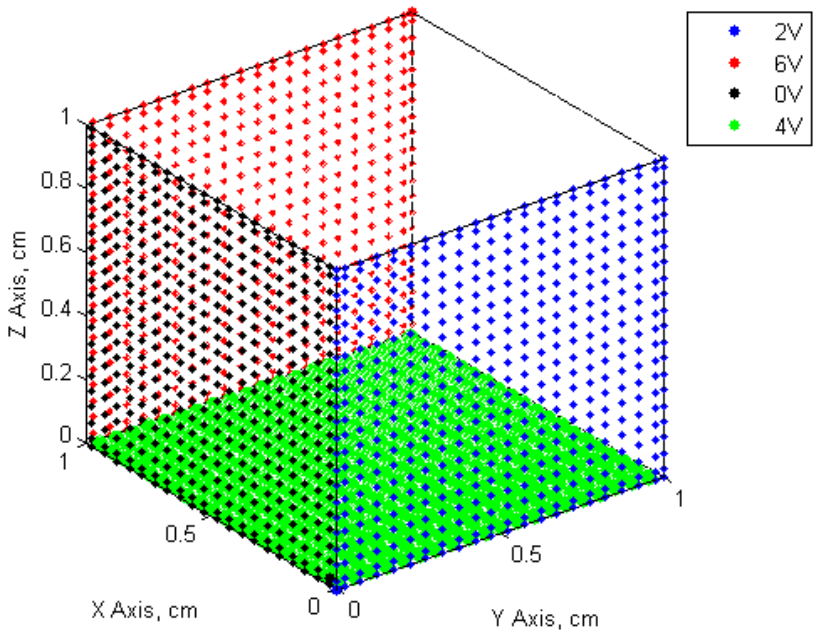

Figure 19. Boundary conditions of the full three-dimensional simulation. The legend identifies constant potential walls with colored points while surfaces without colored points are dielectric boundaries.

The only change made to the numerical parameters from the base case and used for the full three-dimensional case is the particle weighting. Since this case has twice the area for the particles to be absorbed, the steady state number density is lower than in the base case (slightly more than half). Thus, in order to maintain about 100 computer particles per cell, the particle weighting is reduced to 125 real particles per computer particle, which means the injection rate of computer particles is increased from 500 per time step to 2000 per time step. This change ensures that the same physical injection rate is used in the full three-dimensional simulation that was used in the base case. These physical parameters are outlined in Table 9, while the results are shown in Figures Figure 20 to Figure 25.

Table 9. Physical and numerical parameters for the full three-dimensional simulation.

Physical Parameter	Value
Ion injection rate	2.5×10^{14} #/s
Electron injection rate	2.5×10^{14} #/s
Particle weighting	125 real/macro
Time step	1×10^{-9} s
Computational domain volume (cube)	1×10^{-6} m^3
Control volume edge length (Δx)	0.0005 m
Nodes in a single direction	22
Initial electron velocity	100,000 m/s
Initial ion velocity	1,000 m/s
Ion mass (hydrogen)	1.673×10^{-27} kg
Electron mass	9.109×10^{-31} kg
Permittivity of free space	8.854×10^{-12} F/m
Total simulation time	1×10^{-5} s therefore 10,000 time steps

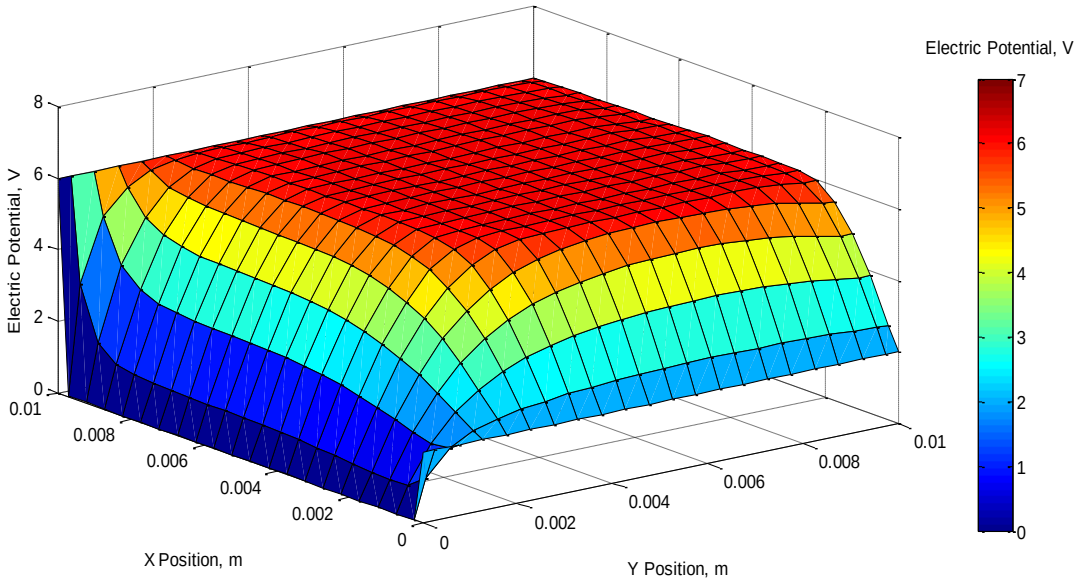

Figure 20. Electric potential contour of the x-y plane at z = 0.00475 m of the full three-dimensional simulation.

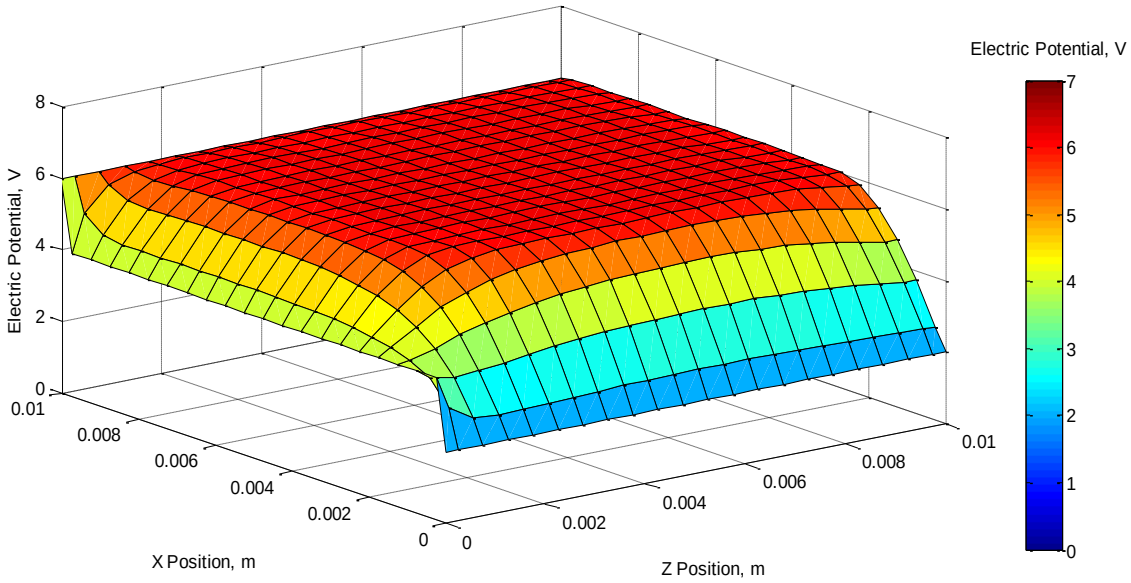

Figure 21. Electric potential contour of the x-z plane at y = 0.00475 m of the full three-dimensional simulation.

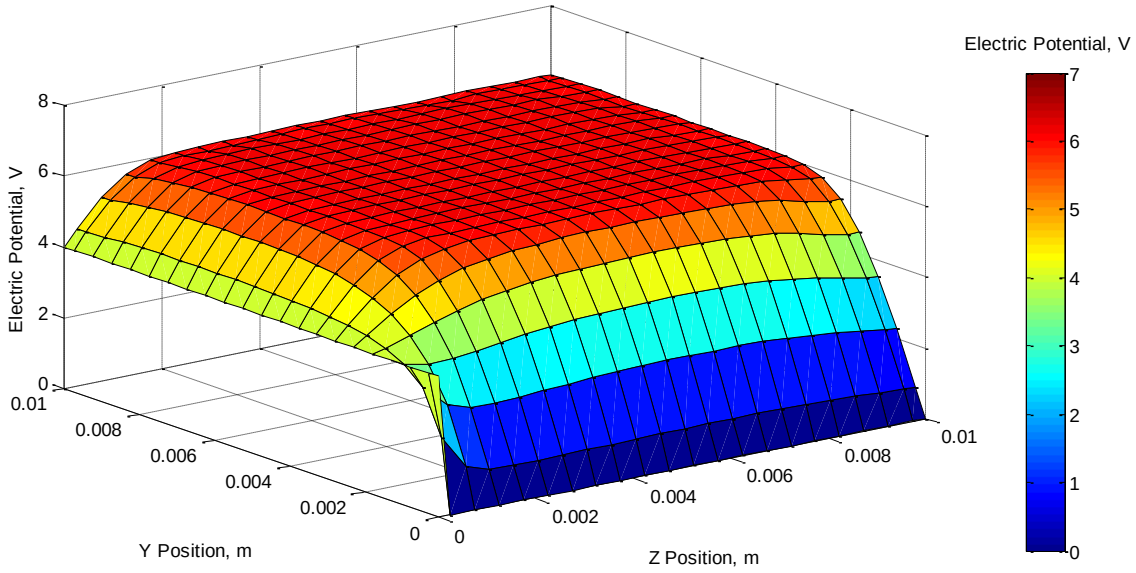

Figure 22. Electric potential contour in the y-z plane at x = 0.00475 m of the fully three-dimensional simulation.

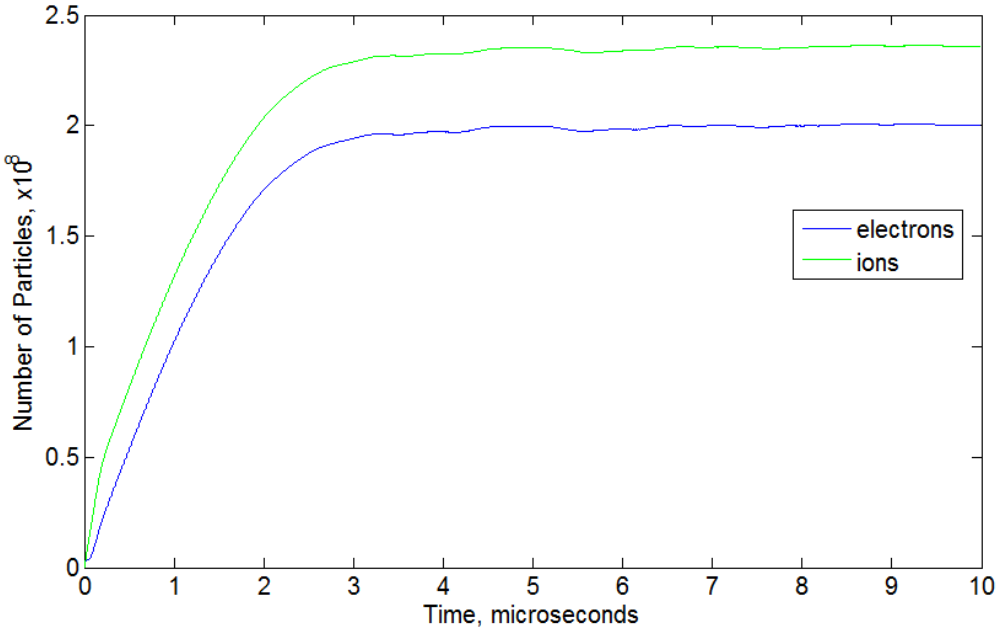

Figure 23. Total number of particles in full three-dimensional simulation.

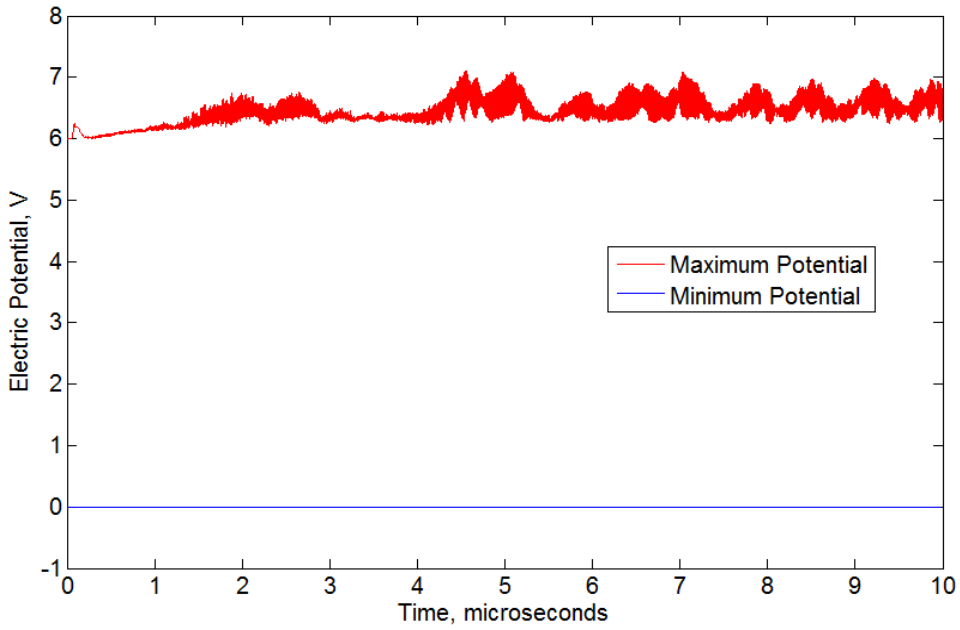

Figure 24. Minimum and maximum electric potential for the full three-dimensional simulation.

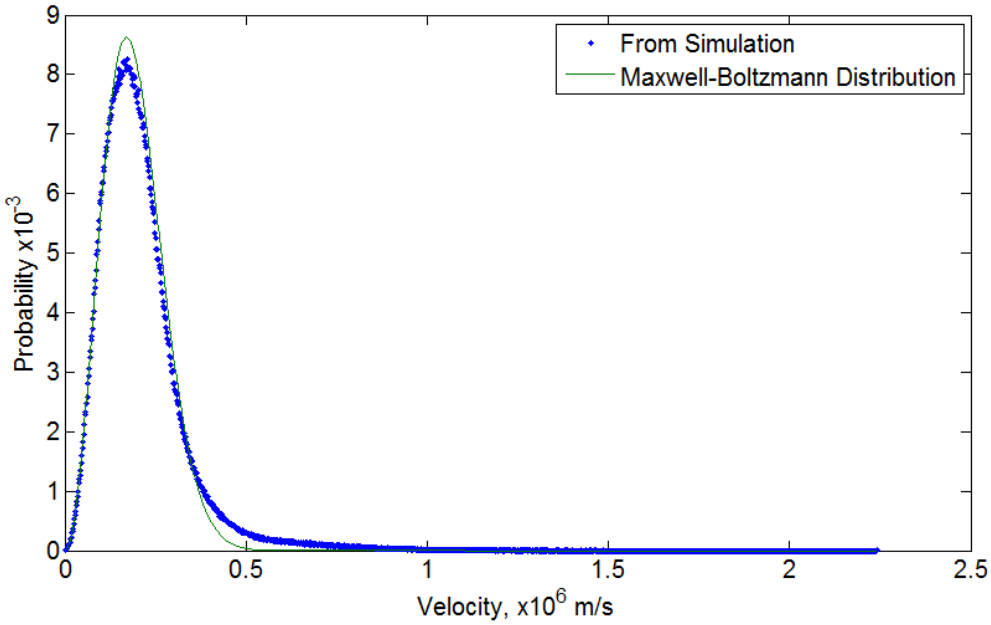

Figure 25. Electron velocity distribution for the full three-dimensional simulation.

Since the full three-dimensional simulation has different boundary conditions at every wall, except for the two dielectric walls, the electric potential contour is shown using three different planes. Each of the planes shown in Figure 20, Figure 21, and Figure 22 are taken at

the center of the perpendicular axis to those planes, much like the electric potential contour of the base case's *x-y* plane is shown at the center *z* location. As shown by the particle number plot, this system reaches steady state more quickly and at a much fewer particles in the computational domain than the base case. This is likely due to the increased area available for particles to escape the computational domain, i.e. more absorbing walls are present. The temperature of the electrons and ions is also somewhat reduced in this simulation compared to the base case, but differences such as these are to be expected since the boundary conditions are different. The steady state plasma parameters and some numerical parameters for the full three-dimensional simulation are shown in Table 10. Note that the values shown in Table 10 are average values throughout the entire computational domain. All of these values vary as a function of position, even after the simulation has achieved steady state. Most notably, the electron number density and ion number density changes as a function of position. The sheath region will have lower number densities and therefore a longer Debye length and lower frequency than the bulk plasma region.

Table 10. Plasma parameters and numerical parameters for full three-dimensional simulation.

Parameter	Value
Electron Number Density	2.002×10^{14} #/m^3
Ion Number Density	2.358×10^{14} #/m^3
Electron Temperature	0.08097 eV
Ion Temperature	0.04876 eV
Plasma Potential	6.174 V
Debye length	0.1495 mm
Plasma frequency	7.983×10^{8} s^{-1}
Computer electrons / cell	200.2
Computer ions / cell	235.8
Grids crossed / time step	0.4134

CHAPTER 4

NOVEL SEMI-IMPLICIT ROUTINES AND RESULTS

In this work, two techniques are formulated and implemented in an attempt to increase the maximum allowable time step while keeping the simulation stable. The two semi-implicit techniques investigated do not require any sort of iteration to numerically solve for any of the future values. Some of the fully implicit techniques explored in the literature review section require such a routine which can lead to a diverging solution if the fully implicit routine is not carefully implemented. The following sections describe the semi-implicit techniques of a fourth order electric field profile and multiple Poisson equation solves per time step.

These two techniques offer very different approaches to achieving the same goal, increase the code's stability when the time step size is increased. The fourth order electric field technique aims at allowing each particle to travel further than one cell per time step and still use an accurate representation of the electric field. Thus, this technique is applied at a particle level and is more of a spatially, semi-implicit technique as opposed to a temporally semi-implicit technique. The multiple Poisson solves per time step technique strives to allow the particles to distribute themselves in a more electrically neutral manner within a single time step to prevent instabilities and unrealistic charge separation. This technique is applied to the entire computational domain and is a temporal semi-implicit technique.

Since the overall goal of these semi-implicit techniques is to allow the size of the time step to be increased, each technique (explicit, fourth order electric field, and multiple Poisson solves) is performed using four different time step sizes. The smallest time step size is the same as the base case simulation which is $1x10^{-9}$ seconds. After the smallest time step is

investigated, larger time steps of $1.5x10^{-9}$, $2x10^{-9}$, and $4x10^{-9}$ are used and each of the semi-implicit techniques is compared to the explicit technique.

4.1. Fourth Order Electric Field

The first semi-implicit technique investigated is the fourth order electric field technique. This technique fits a fourth order polynomial to the electric field which allows each particle to be advanced via an electric field profile, as opposed to a single electric field value. Several problems occur when too large of a time step is used in a PIC simulation, and one of the more restrictive problems is related to the particle moving more than one control volume in a single time step. This is the problem we are trying to address with this fourth order electric field technique. The idea behind the fourth order technique is to allow the particle to use electric fields from all the control volumes it passes through as opposed to just the cell of origin. It was thought that this would make the simulation more stable when larger time steps are used.

If a particle is in a region of large electric field strength, such as a sheath, the particle can be accelerated using this large electric field for a longer time than it physically should. If the simulation allowed the particle to sense the electric field present in each control volume it passed through, this artificial acceleration known as numerical heating could be avoided. Problems also arise when a particle is moving relatively quickly within a region of small electric field, such as the bulk plasma region, but heading toward a region of large electric field, such as a retarding sheath. In this case, the large electric field that should slow down the particle and keep it in the computational domain may be missed entirely by the explicit numerical routine and the particle may be lost to an absorbing wall. Physically this particle may not have been lost because the large retarding electric fields would have reversed its direction of motion, but due to the finite nature of the simulation this retarding force may not be fully applied.

4.1.1. Routine

As previously shown in the particle advance section, the advance can be split into four parts, two of the parts being the electric field push. Since the code developed in this work is three-dimensional, each of the equations presented in this section are applied to the particle in each direction separately. Since each direction is done separately, the vector

accents over any vector quantity has been dropped in this section. An acceleration term is present in equations (50) and (55) which is found by integrating the acceleration due to the electric field over half of a time step as

$$a = \frac{F}{m_p} = \int_0^{\frac{\Delta t}{2}} \frac{q_p E_{i,j,k}^n t}{m_p} dt. \tag{70}$$

In the explicit version of the code where the electric field is constant throughout a time step, the acceleration term, i.e. the solution to equation (70) is

$$a = \frac{q_p E_{i,j,k}^n \Delta t}{2m_p}. \tag{71}$$

Explicitly, the electric field applied to a particle in cell *i,j,k* is $E_{i,j,k}^n$. Instead of just using the electric field for the cell in which the particle is located, the fourth order technique uses the electric field for 5 cells in the direction of the particle's current motion. Thus, a fourth order equation for the electric field is

$$E = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 . \tag{72}$$

The coefficients a_0, a_1, a_2, a_3, and a_4 in this equation must be determined and are

$$a_0 = E_1, \tag{73}$$

$$a_1 = -\frac{25E_1 - 48E_2 + 36E_3 - 16E_4 + 3E_5}{12\Delta x}, \tag{74}$$

$$a_2 = \frac{35E_1 - 104E_2 + 114E_3 - 56E_4 + 11E_4}{24(\Delta x)^2}, \tag{75}$$

$$a_3 = -\frac{5E_1 - 18E_2 + 24E_3 - 14E_4 + 3E_4}{12(\Delta x)^3}, \tag{76}$$

and

$$a_4 = \frac{E_1 - 4E_2 + 6E_3 - 4E_4 + E_4}{24(\Delta x)^4}. \tag{77}$$

These coefficients can be solved directly because the number of points used is one less than the order of the polynomial used to fit the points. This allows the polynomial developed in equation (72) to exactly fit every point thus eliminating the need to solve for the polynomial's coefficients using a least squares technique. It is desirable to avoid the least

squares technique because it involves finding the inverse of a matrix which is quite computationally expensive.

In the preceding equations, the variables E_1, E_2, E_3, E_4, and E_5 are the electric field values of the grid points along the particle's projected path. It is important to note that these equations only apply to a uniform, structured grid. If a non-uniform grid is used, then the Δx value between nodes will change depending on the node so the Δx terms could not be combined as is done currently for the coefficients. Instead, the Δx term appearing in the denominator of equations (74) to (77) would be a different term that may not necessarily be a common denominator.

Once the coefficients for the electric field are determined, the future position of the particle can be estimated based on the time step and the particle's velocity at the beginning of the time step,

$$x = v_p^{n-1/2} t \tag{78}$$

This must be done because the integration shown in equation (70) is performed with respect to time, not position. Thus, substituting the differential of equation (78) into equation (72) and integrating over half of a time step gives

$$a = \frac{q_p \Delta t}{2 m_p} \left(a_0 + \frac{a_1 v_p^{n-\frac{1}{2}} \Delta t}{4} + \frac{a_2 (v_p^{n-\frac{1}{2}})^2 \Delta t^2}{12} + \frac{a_3 (v_p^{n-\frac{1}{2}})^3 \Delta t^3}{32} + \frac{a_4 (v_p^{n-\frac{1}{2}})^4 \Delta t^4}{80} \right) \tag{79}$$

The terms in parenthesis, along with the Δt /2 term in front, represent the electric field integrated over half of a time step. Substituting this acceleration term into the leapfrog advance is ultimately how the fourth order electric field technique is implemented into the particle advance. The leapfrog integration method is still used for incorporating the magnetic field; only the way the electric field is applied is altered when comparing this technique to the explicit technique. The following section compares the results of the fourth order electric field technique to the purely explicit technique.

4.1.2. Results

Unfortunately, most of the results indicate that the fourth order electric field technique has little effect on the outcome of the simulation, regardless of the time step size. However, what effect the technique does have is generally in the correct direction of producing results closer to the converged solution than the explicit version of the code. Figure 26 shows the electric potential contour plot for the fourth order electric field technique when a time step of $1x10^{-9}$ seconds is used. Figure 27 compares the number of electrons in the computational domain from the explicit technique and the fourth order electric field technique, while Figure 28 compares the electric potential profiles of each technique. Again, these electric potential profiles are obtained by averaging the last 5% of the simulation.

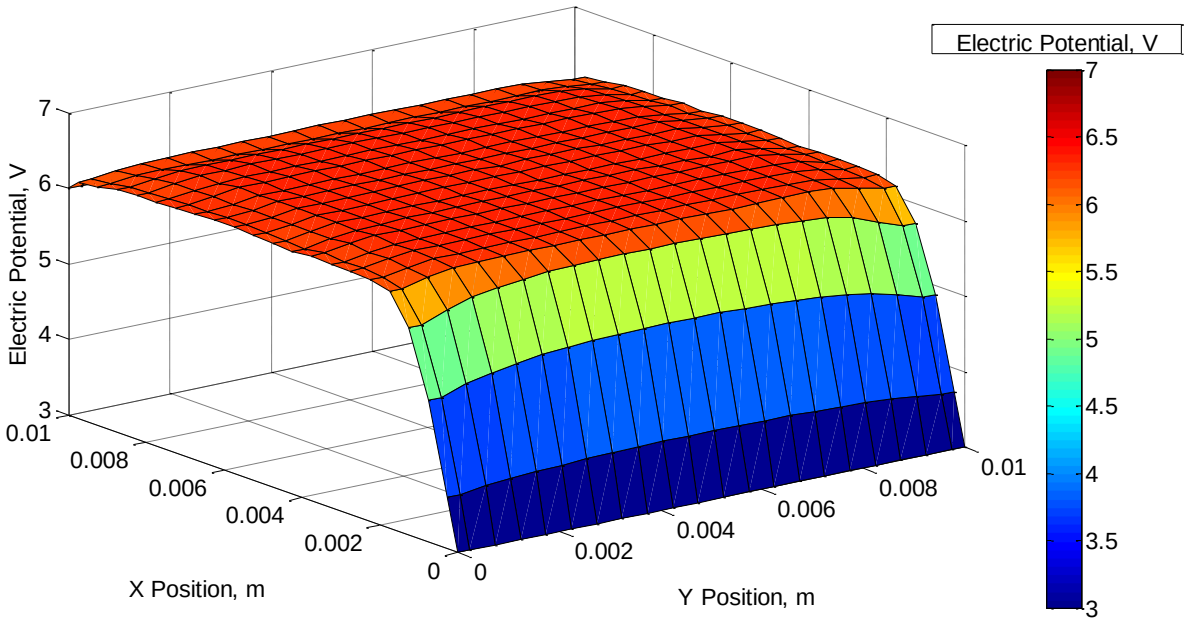

Figure 26. Electric potential contour using fourth order electric field technique, time step = $1x10^{-9}$ seconds.

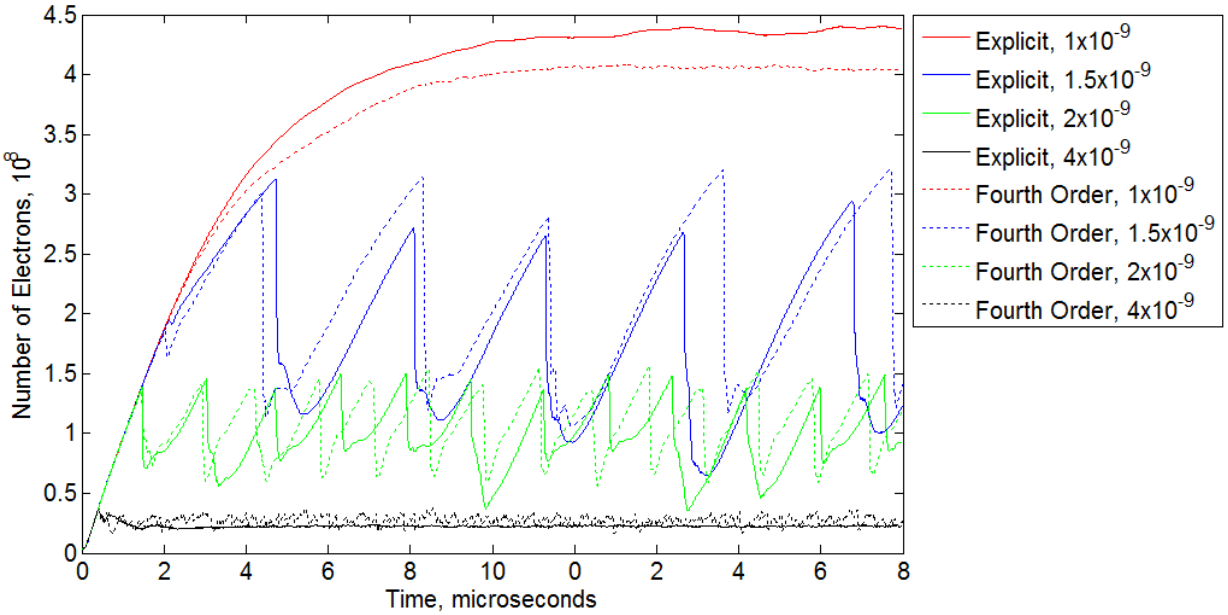

Figure 27. Total electron number comparison of explicit technique versus fourth order electric field technique.

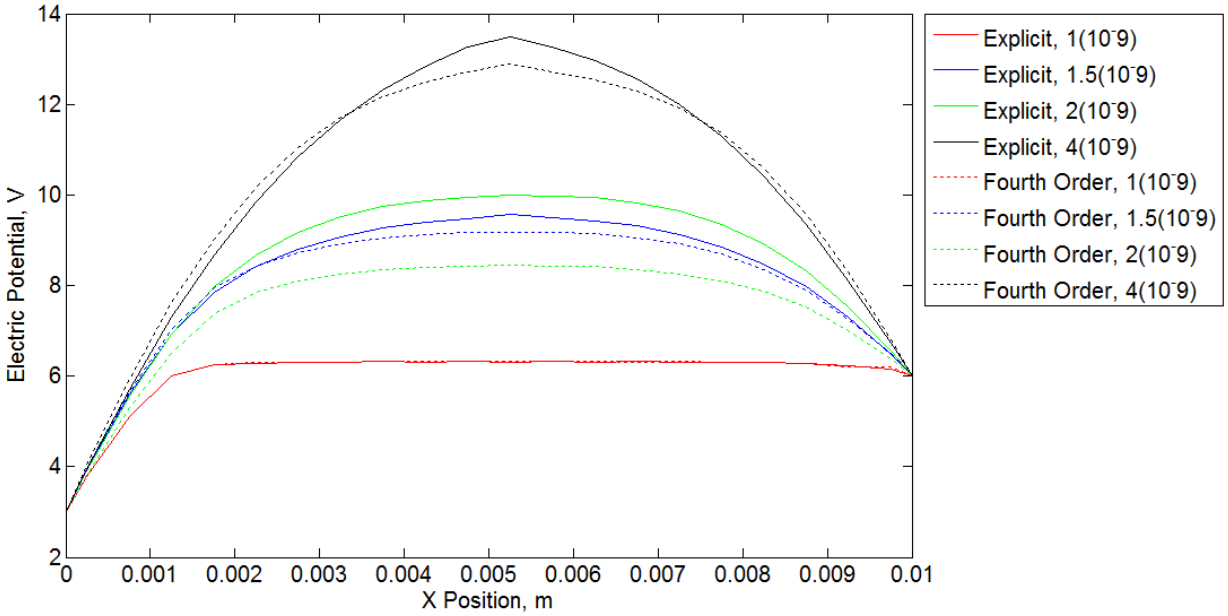

Figure 28. Electric potential profile comparison of explicit technique versus fourth order electric field technique.

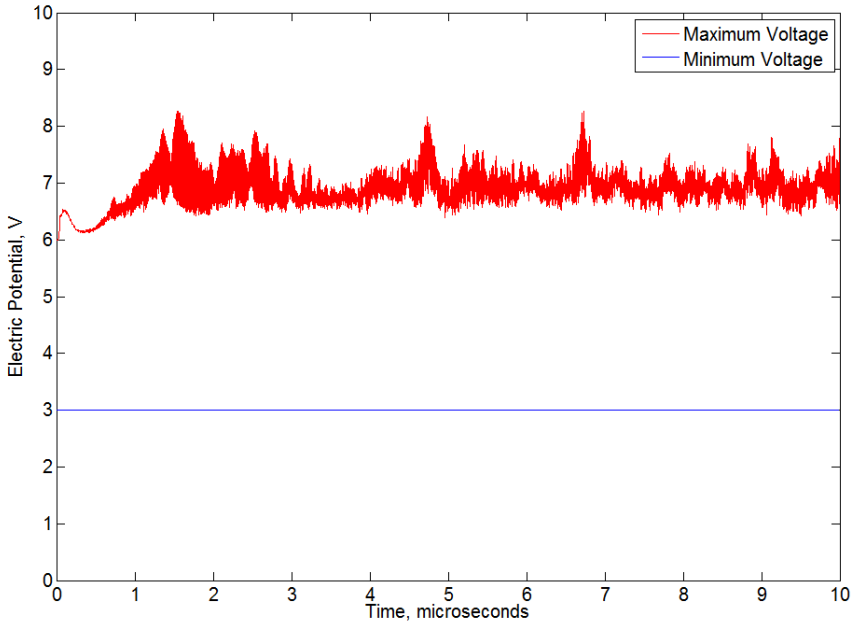

Figure 29. Minimum and maximum electric potential for the fourth order electric field technique, time step = $1x10^{-9}$ seconds.

Figure 26 compares nicely with the potential contour plot of the explicit case in Figure 13. Figure 27 shows that at time steps larger than $1x10^{-9}$, both techniques become unstable and show large oscillations of the number of electrons in the simulation. The oscillations are very similar between the two techniques, which indicate that the fourth order electric field technique did not quell the instabilities as hoped.

Figure 28 shows that the fourth order technique slightly improves the steady state electric potential profile at larger time steps. Using the smallest time step, the solid red line and the dotted red line coincide almost perfectly. The most noticeable difference between the two techniques occurs at a time step of $2x10^{-9}$ seconds. In Figure 28, the red lines are considered to be at the correct, converged solution, so achieving this result using a larger time step is desired. At the $2x10^{-9}$ time step, the fourth order technique is closer to the converged result than the explicit technique. Figure 29 shows that the maximum potential as a function of time has a value close to that of the explicit base case, but with slightly more noise (see Figure 16).

4.2. Multiple Poisson Solves

The second semi-implicit technique investigated in this work is the multiple Poisson solves technique which essentially updates the electric field multiple times in a single time step, in contrast to the explicit technique solving Poisson's equation only once per time step. The idea behind this technique is to allow the particles to distribute themselves in a more electrically neutral manner within a single time step. If the charged particles can distribute themselves more appropriately, this should prevent unnatural charge separation from occurring. Artificial charge separation causes charge buildup and/or charge deficiency throughout the plasma which means the quasi-neutrality that most plasmas exhibit in their bulk region is violated. This violation allows for spikes in electric potential, especially in a fully coupled code where the permittivity of free space is not inflated.

4.2.1. Routine

The routine for this technique is summarized in the flow chart shown in Figure 31. For this technique, no new equations are utilized only the order of the execution is changed. The flow chart shown below is much like Figure 5, except the particle advance section is split into a "partial ion advance" and a "partial electron advance" and loops are added where the multiple Poisson equation solves take place.

This technique can be implemented in one of two ways. The algorithm in Figure 30 shows the multiple solves occurring only during the electron advance while the algorithm in Figure 31 shows the multiple solves occurring during both the ion advance and the electron advance. Since the ions and electrons carry the same magnitude of charge and the ions are generally several orders of magnitude slower than the electrons, the electrons are capable of more drastically altering the electric field. Thus it may or may not be necessary or even helpful to solve Poisson's equation multiple times during the ion advance. Hereafter, the variation of the multiple Poisson solves technique taking place during only the electron advance will be referred to as "variation one" (Figure 30) while the multiple Poisson solves technique which takes place during both the electron advance and the ion advance will be referred to as "variation two" (Figure 31).

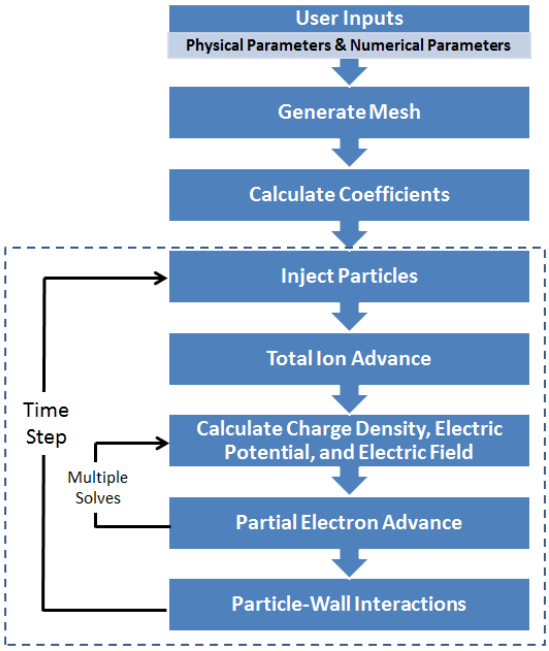

Figure 30. Flow chart for multiple Poisson solves technique during electron push only (variation one).

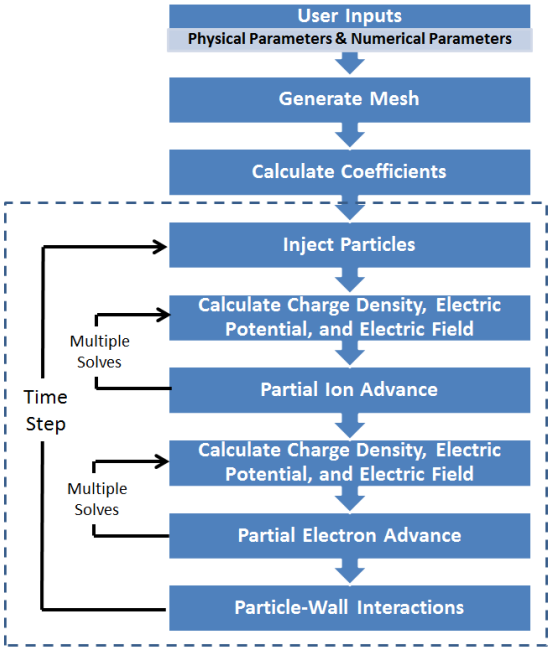

Figure 31. Flow chart for multiple Poisson solves technique during the ion push and electron push (variation two).

The optimal number of times the multiple solves routine should be called during a single time step likely depends on a combination of the particle weight and the number of computer particles present in the simulation. At this point, the optimal ratio of Poisson solves per time step to the number of particles (real or computer) is unknown. Thus for the purposes of evaluating the possible effectiveness of this technique, 20 Poisson solves per time step are used for variation two while 10 Poisson solves per time step are used for variation one.

4.2.2. Results

This section shows the results from the two variations of the multiple Poisson solves per time step technique and compares them with the explicit technique. Similar to the comparison between the fourth order electric field technique and the explicit technique, this comparison also uses time steps of $1x10^{-9}$, $1.5x10^{-9}$, $2x10^{-9}$, and $4x10^{-9}$. Figure 32 is the electric potential contour plot for variation one with a time step of $1x10^{-9}$ seconds. Similarly, Figure 34 is the electric potential contour plot for variation two using a time step of $1x10^{-9}$ seconds. Figure 36 compares the number of electrons between the three different techniques (explicit and the two variations of multiple Poisson solves) over the range of time steps previously stated. Lastly, Figure 37 compares the electric potential profile of all three techniques over the specified range of time steps.

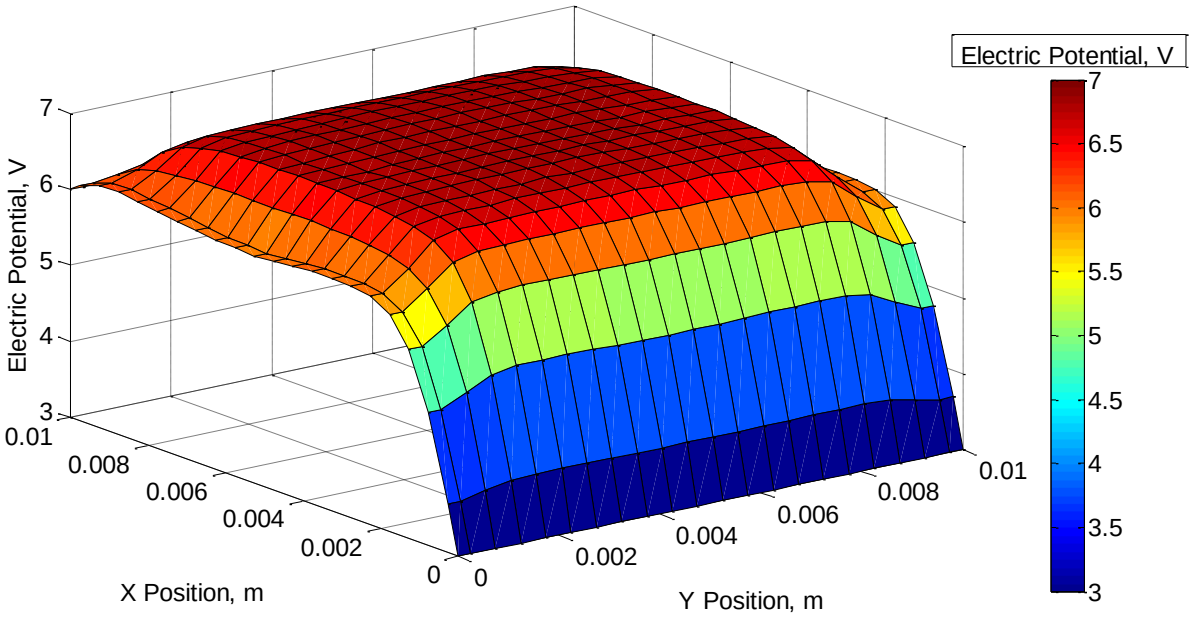

Figure 32. Electric potential contour of multiple Poisson solves per time step during electron advance only (variation one) time step = $1x10^{-9}$ seconds.

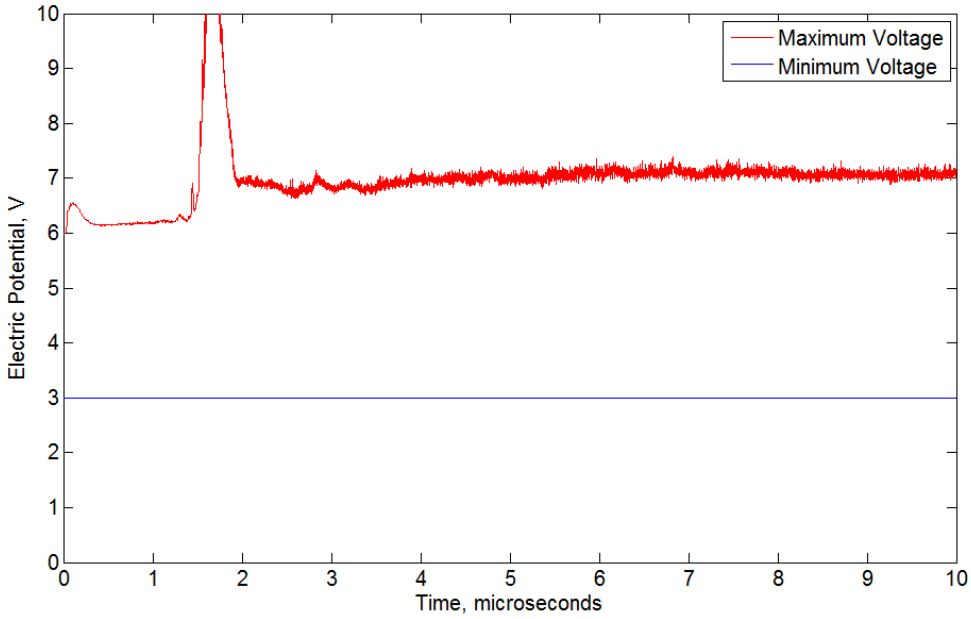

Figure 33. Minimum and maximum electric potential for multiple Poisson solves during the electron advance only (variation one) time step = $1x10^{-9}$ seconds**.**

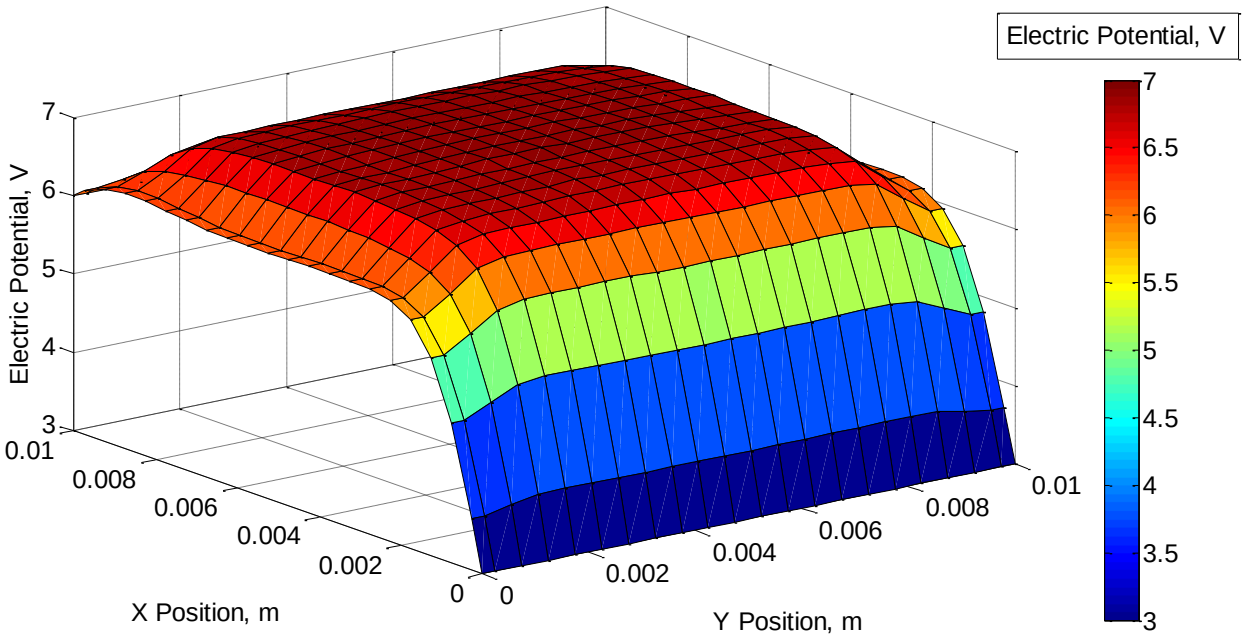

Figure 34. Electric potential contour for multiple Poisson solves per time step during electron and ion advance (variation two), time step = $1x10^{-9}$ seconds.

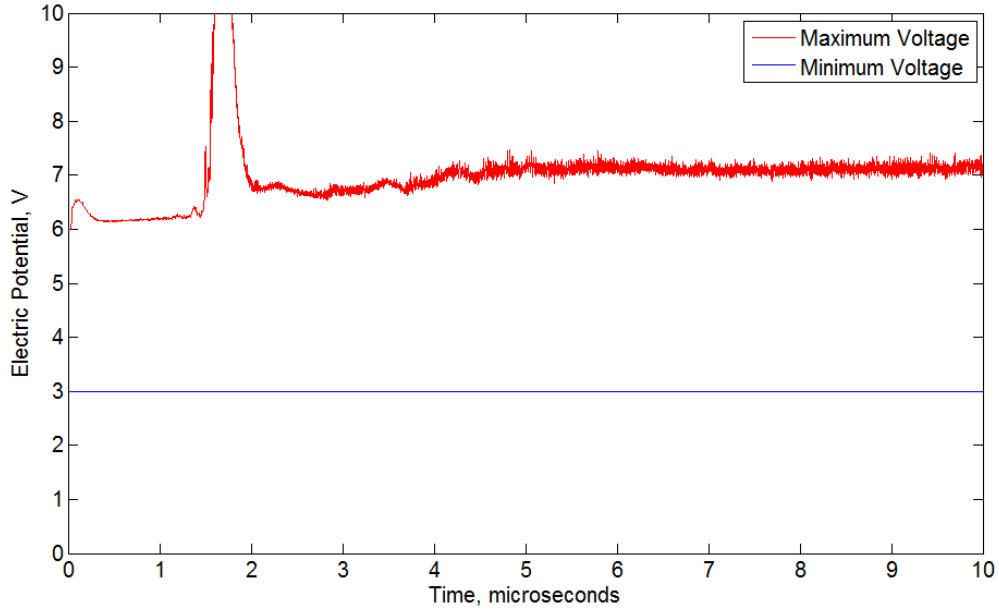

Figure 35. Minimum and maximum electric potential for multiple solves technique during the ion advance and electron advance (variation two), time step = $1x10^{-9}$ seconds.

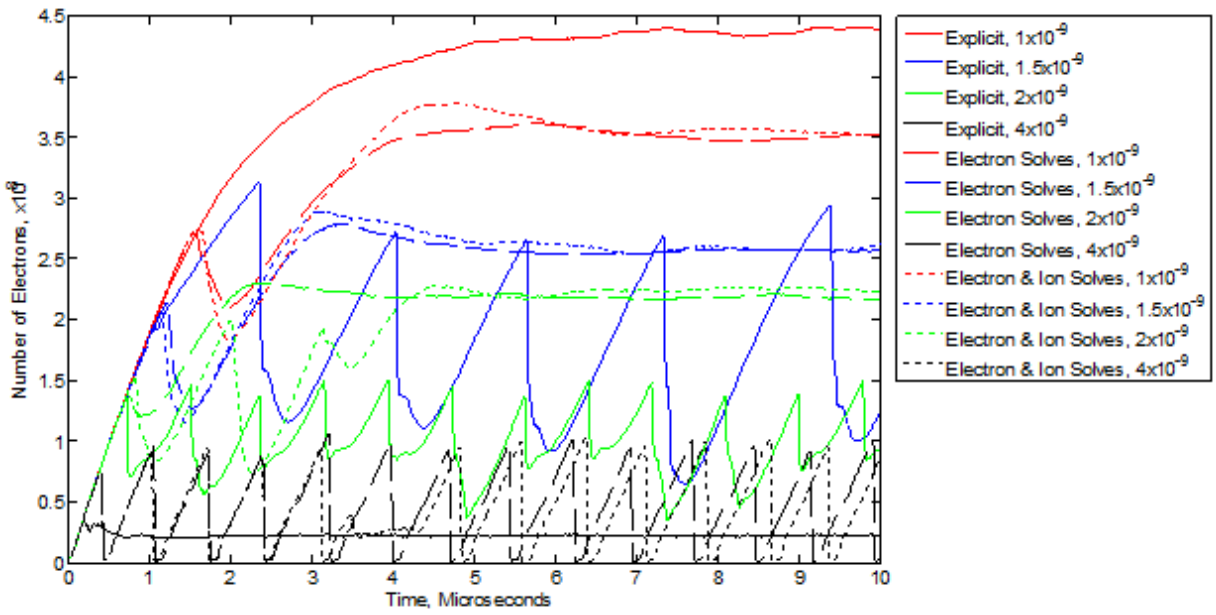

Figure 36. Comparison of number of electrons using multiple Poisson solves versus explicit solution technique. The time step sizes in the legend are in seconds.

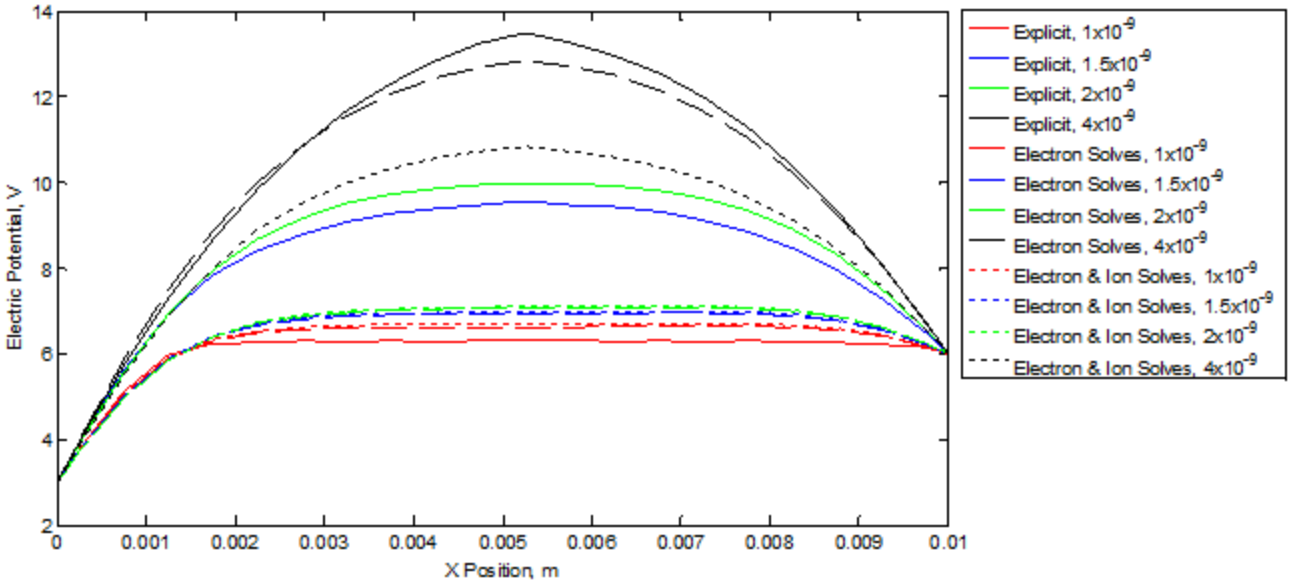

Figure 37. Comparison of electric potential profiles using multiple Poisson solves versus explicit solution technique. The time step sizes in the legend are in seconds.

As is shown in Figures Figure 32 through Figure 37 the differences between variation one and variation two are minimal and both add a measure of stability compared to

the explicit technique at larger time steps. Figure 36 clearly shows the added stability of the multiple Poisson solves technique at time steps of $1.5x10^{-9}$ and $2x10^{-9}$, the blue and green lines respectively. The explicit technique at these time steps shows large oscillations in the number of electrons indicating that the code is not stable. Although the multiple Poisson solves technique does not yield the same values at time steps of $1.5x10^{-9}$ and $2x10^{-9}$ seconds, the stability of the code is improved such that the electron number no longer oscillates. At a time step of $4x10^{-9}$ seconds the explicit total number of electrons curve in Figure 36 goes flat. The reader may take this as stable behavior, but this is not correct. This behavior is due to electrons exiting the small computational domain in one or two time steps. At a time step of $4x10^{-9}$ seconds the explicit technique has completely broken down and the results are wrong. The fact that the multiple Poisson solves routine starts to oscillate at this time step means that it has not reached the state of degradation that the explicit routine has at $4x10^{-9}$ seconds. At time steps larger than $4x10^{-9}$ seconds, the multiple Poisson solves routine will flat line as well.

The improved stability of the multiple Poisson solves routine also manifests itself in Figure 37 by showing the semi-implicit technique's results being much closer to the converged solution (solid red line) than the explicit technique's results for all times steps larger than $1x10^{-9}$ seconds tested. The minimum and maximum electric potential plots shown in Figure 33 and Figure 35 are almost identical and both are comparable to the explicit base case results shown in Figure 16. The spike in potential occurring at about 1.7 microseconds reaches a peak value of about 15 volts. These minimum and maximum electric potential plots are scaled in order to be directly comparable to the plot for the explicit base case. Thus the magnitude of this spike is not shown in the plots. While at a time step of $1x10^{-9}$ seconds, the multiple Poisson solves technique's and explicit technique's maximum and minimum electrical potential results do not look much different; however, using a time step of $2x10^{-9}$ seconds shows large differences. In comparing the explicit technique results in Figure 38 to the multiple Poisson solves results in Figure 39, the improvement in stability is quite apparent. Both of the figures use a time step of $2x10^{-9}$ seconds; however, the explicit case shows oscillations between -40 volts and 30 volts while variation one of the multiple Poisson solves technique shows very stable behavior with no oscillations.

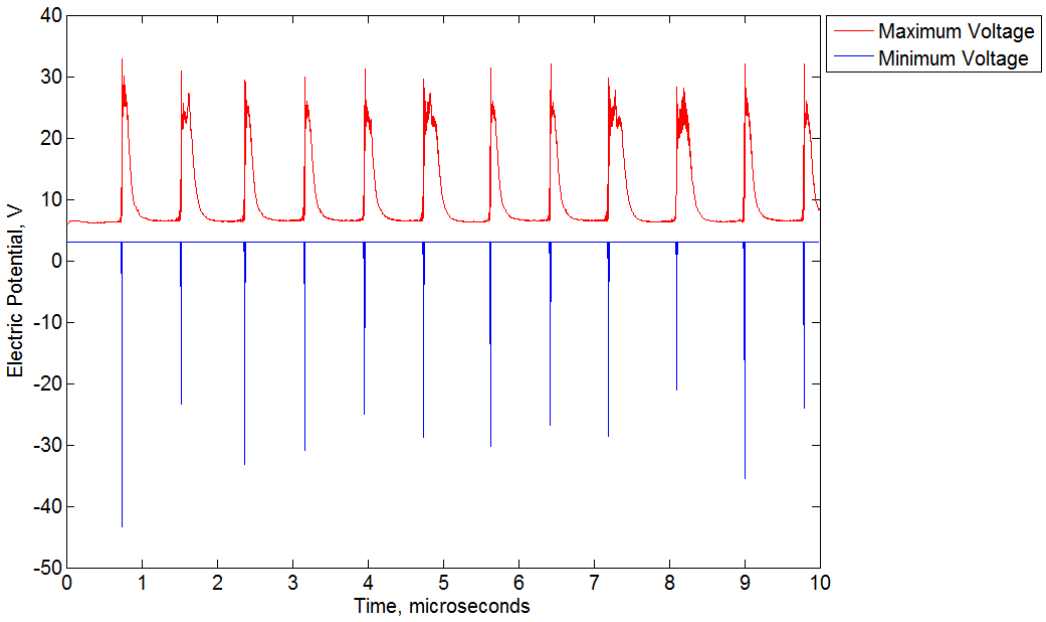

Figure 38. Minimum and maximum electric potential for explicit technique, time step = 2×10^{-9} seconds.

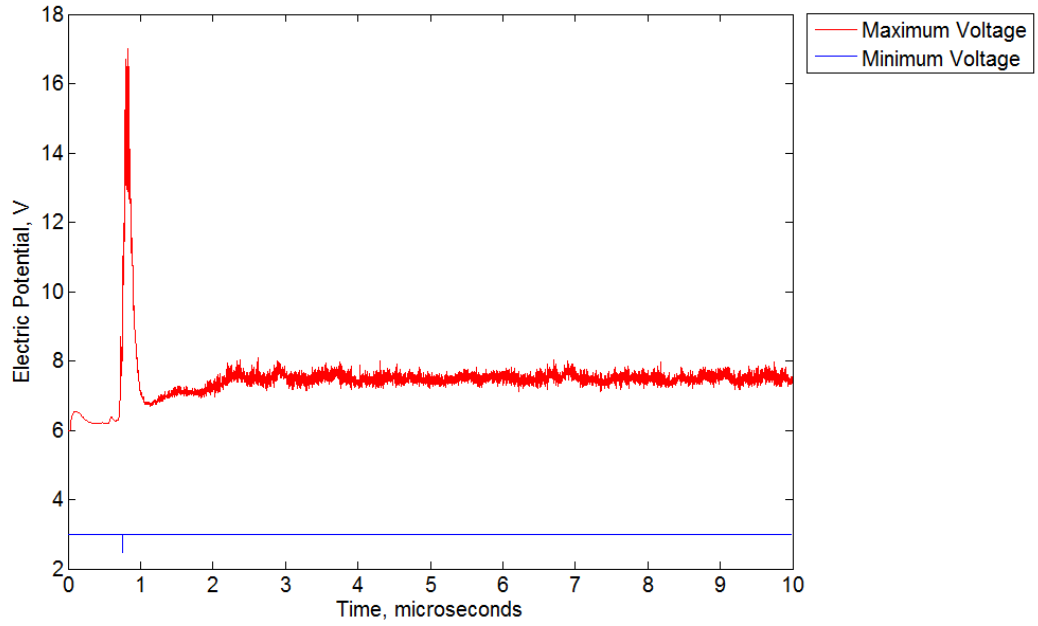

Figure 39. Minimum and maximum electric potential for multiple Poisson solves during the electron advance only (variation one), time step = 2×10^{-9} seconds.

CHAPTER 5

CONCLUSIONS

Neither of the techniques explored in this work should be considered fully developed or optimized. Keeping this in mind, some initial findings on the effectiveness of these two semi-implicit techniques can be presented with some certainty. Of the two semi-implicit techniques investigated in this work, version one of the multiple Poisson solves technique seems to show the most promise in terms of improving the stability of the PIC algorithm. Version two of the technique also provided similar, favorable results but at an increased amount of computational time. The fourth order electric field technique did not have much of an impact on the simulation.

Although the results from the multiple Poisson solves technique were not identical to the smallest time step of the explicit technique, the stability of the simulation improved at larger time steps, namely 1.5×10^{-9} seconds and 2×10^{-9} seconds. Thus, version one of the multiple Poisson solves technique in its present form is not the complete answer to addressing the computational time problem currently plaguing the explicit PIC algorithm, but it may prove to be a step in the right direction. Though each time step takes more time when Poisson's equation must be solved multiple times, the computational penalty can greatly benefit from parallel computing and a more efficient Poisson solving technique than the TDMA solver used in this work. Also in larger simulations, the particle push tends to be the largest contributor to computational time as opposed to the field solver. Thus in simulations with a large number of computer particles as is often the case in PIC modelling, this method will become more attractive.

One other aspect of the multiple Poisson solves technique to consider is the number of solves performed each time step. Though it has not been sufficiently investigated, it stands to reason that an optimal ratio of the number of computer particles to the number of field solves exists both in terms of accuracy and computational time. As previously stated, 10 solves per time step are used in version one, while 20 solves per step are used in version two, but the results are nearly identical. Given the additional solves during the ion advance have little to no bearing on the outcome of the simulation, the results of the two versions matching closely comes as no surprise, since the number of additional solves during the electron advance is the same in both versions. In this work the number of solves per time step remains constant regardless of the number of computer particles in the computational domain, which changes significantly as a function of time. Instead of keeping the number of solves each step constant, a better approach to implementing this technique may be to keep the ratio of the number of computer particles to the number of solves per time step constant or below a certain threshold.

The size of the domain is also a point of concern for the multiple Poisson solves technique. If the domain size is small enough such that the particles are able to escape the domain within a time step or two, even if their velocity is reasonable, the particles will immediately vacate the simulation regardless of the technique used. This behavior is shown by the explicit case using a time step of $4x10^{-9}$ in Figure 36. In this case the number of electrons for the explicit technique (solid black line) does not show oscillations, but rather shows a very "stable" flat line. At this time step, the electrons cover nearly half of the computational domain in a single time step using only their initial velocity. To address this and allow time for the multiple Poisson solves technique to have some sort of effect, the domain size should be increased to accommodate the electrons travelling so far. Doing this could show that the technique is capable of maintaining stability at even greater time steps than is shown in this work.

The fourth order electric field in its present form did little to affect the outcome of the simulation; however, some adjustments may be made to this technique in order to improve its performance. The number of points used for fitting the polynomial as well as the order of the polynomial are somewhat arbitrary at this point. Similar to the multiple Poisson solves

technique; the optimal value for the variables involved in the technique is likely a function of the numerical and physical parameters defined for the simulation. For the fourth order electric field, the optimal number of points and order of the polynomial likely depends on the grid size and the number of grids; e.g. if a finer mesh is used, it will likely require more grid points to fit to the polynomial in order for the technique to be effective.

Another approach to improving the fourth order electric field technique is to not fit the electric field at all, but rather the electric potentials. Using the first order weighting scheme as shown in Figure 8, the source terms (i.e. charge density) at each grid point are well known because a single particle is distributed among 8 nodes, whereas the electric field values at each grid point are linearly interpolated using only the two nearest nodes. Since the electric field is defined as the negative gradient of the potential, if a polynomial can be found to describe the potential then the derivative of that polynomial will yield the electric field over a range of space. The order of this polynomial could be chosen so that its derivative also describes how the electric field changes as a function of position over a region of the computational domain.

Neither of the techniques explored in this work are the complete answer to the computational time problem experienced by PIC modelling but further development of one or both of these techniques may help. Optimization may be the only step necessary to creating a viable multiple Poisson solves per time step technique. As for the fourth order electric field technique, at this time the technique does not affect the simulation to a large degree, but the core idea of using multiple grid points to affect the movement of a single particle may yet prove useful.

REFERENCES

Bias, B., Penkal, B., Jonell, M., Menart, J., & Mahalingam, S. (2011). Off Design Simulation Results of Several Operating Conditions of the NEXT Discharge Chamber. *47th AIAA/ASME/SAE/ASEE Joint Propulsion Conference & Exhibit*, (pp. 5660-1 - 5660-29). San Diego.

Birdsall, C. K., & Langdon, A. B. (1991). *Plasma Physics Via Computer Simulation.* New York : IOP Publishing Ltd.

Boyd, T., & Sanderson, J. (2003). *The Physics of Plasmas.* New York: Cambridge University Press.

Brackbill, J. U., & Forslund, D. W. (1982). An Implicit Method for Electromagnetic Plasma Simulation in Two Dimensions. *Journal of Computational Physics*, 271-308.

Briguglio, S., Vlad, G., Martino, B. D., & Fogaccia, G. (2000). Parallelization of Plasma Simulations Codes: Gridless Finite Size Particle Versus Particle in Cell Approach. *Future Generation Computer Systems*, 541-552.

Christlieb, A. J., Krasny, R., verboncoeur, J. P., Emhoff, J. W., & Boyd, I. D. (2006). Grid-Free Plasma Simulation Techniques . *IEEE Transactions of Plasma Science*, 149-165.

Davidson, L. (2005). *Numerical Methods for Turbulent Flow.* Gothenburg: Chalmers Institute of Technology.

Dinklage, A., Klinger, T., Marx, G., & Schweikhard, L. (2005). *Plasma Physics Confinement, Transport and Collective Effects.* New York: Springer-Verlag Berlin Heidelburg.

Gibbons, M. R., & Hewett, D. W. (1995). The Darwin Direct Implicit Particle-in-Cell (DADIPIC) Method for Simulation of Low Frequency Plasma Phenomena . *Journal of Computational Physics* , 231-247.

Goebel, D. M., & Katz, I. (2008). *Fundamentals of Electric Propulsion: Ion and Hall Thrusters.* Pasadena: Jet Propulsion laboratory.

Griffiths, D. J. (1999). *Introduction to Electrodynamics.* Upper Saddle River: Prentice-Hall, Inc. .

Herman, D. A. (2005). *The Use of Electrostatic Probes to Characterize the Discharge Plasma Structure and Identify Discharge Cathode Erosion Mechanisms in Ring-Cusp Ion Thrusters.* Ann Arbor: University of Michigan.

Hill, M. D., & Marty, M. R. (2007). *Amdhal's Law in the Multicore Era.* IBM.

Intel. (2014). *Intel 22nm Technology*. Retrieved May 11, 2014, from Intel: www.intel.com

Kawamura, E., Birdsall, C. K., & Vahedi, V. (2000). Physical and Numerical Method of Speeding Up Particle Codes and Paralleling as Applied to RF Discharges . *Plasma Sources Science and Technology* , 413-428.

Kim, H. C., Iza, F., Yang, S. S., Radmilovic-Radjenovic, M., & Lee, J. K. (2005). Particle and fluid simulations of low-temperature plasma discharges: benchmarks and kinetic effects. *Journal of Physics D: Applied Physics*, 283-301.

Lapenta, G. (2006). *Particle in Cell Method a brief description of the PIC method.*

Lapenta, G. (2008). The Algorithms of the Implicit Method. *ASP Conference Series* (pp. 1-7). Cornell University Library .

Lapenta, G. (2010). *Particle In Cell Methods With Application to Simulations in Space Weather.*

Lapenta, G., Brackbill, J. U., & Ricci, P. (2006). Kinetric Approach to Microscopic-Macroscopic Coupling in Space and laboratory Plasmas . *Physics of Plasmas*, 055904-1 - 055904-9.

Mahalingam, S. (2007). *Particle Based Plasma Simulation for an Ion Engine Discharge Chamber.* Dayton : Wright State University.

Mahalingam, S., Choi, Y., Loverich, J., Stoltz, P. H., Jonell, M., & Menart, J. A. (2010). Dynamic Electric Field Calculations Using a Fully Kinetic Ion Thruster Discharge Chamber Model. *46th AIAA/ASME/SAE/ASEE Joint Propulsion Conference & Exhibit .* Nashville.

Mullen, L. (1999, September 7). *Plasma, Plasma, Everywhere*. Retrieved April 25, 2014, from NASA Science: http://science1.nasa.gov/

Oak Ridge National Laboratory. (2012). *Introducing Titan | The Worlds #1 Open Science Super Computer*. Retrieved May 11, 2014, from Oak Ridge National Laboratory: https://www.olcf.ornl.gov/titan/

Plasma Surgical. (2013). *PLASMAJET*. Retrieved April 25, 2014, from Plasma Surgical: http://www.plasmasurgical.com/

Sadiku, M. N. (2001). *Numerical Techniques in Electromagnetics, Second Edition.* Boca Raton: CRC Press LLC, .

Tajima, T. (2004). *Computational Plasma Physics .* Boulder: Westview Press.

Tech-X Corporation. (2014, April 29). *Tech-X Corporation Home Page* . Retrieved May 2, 2014, from Tech-X Simulations Empowering your Innovations: www.txcorp.com

Ueda, H., Omura, Y., Matsumoto, H., & Okuzawa, T. (1994). A Study of the Numerical Heating in Electrostatic Particle Simulations. *Computer Physics Communications*, 249-259.

Vahedi, V., & Surendra, M. (1995). A Monte Carlo Collision Model for the Particle-in-Cell Method: Applications to Argon and Oxygen Discharges. *Computer Physics Communications*, 179-198.

Winske, D., Yin, L., Omidi, N., Karimabadi, H., & Quest, K. (2003). *Hybrid Simulation Codes: Past, Present and Future - A Tutorial.* Berlin: Springer-Verlag.

Appendix A

Tri-Diagonal Matrix Algorithm (TDMA)

In order to efficiently solve for the electric potentials across the entire computational domain, a tri-diagonal matrix algorithm (TDMA) solver is used. The TDMA solver is an iterative solver which essentially uses Gaussian elimination (Davidson, 2005). The work in Davidson is shown for a two dimensional simulation, but it is easily extended to three dimensions by adding two more adjacent terms and adjusting the inputs to the solver and source term accordingly. The TDMA used in this work is discussed in more detail in Appendix A. The equation actually being solved using the TDMA solver in this work is the discretized version of Poisson's equation (equation (13)). Starting with equation (13), it is re-written as

$$a_i \Phi_i = b_i \Phi_{i+1} + c_i \Phi_{i-1} + d_i \tag{80}$$

The subscript *n* denotes the node number in the direction of the current sweep. Assuming the current sweep is in the x direction (east and west) the subscript *n* is essentially *i* as it has been used thus far. For an x direction sweep, the values of *j* and *k* will remain the same throughout the sweep. The terms in equation (80) are defined as

$$a_i = aP_i \tag{81}$$

$$b_i = aE_i \tag{82}$$

$$c_i = aW_i \tag{83}$$

$$d_i = aN_i \Phi_{i,j+1,k} + aS_i \Phi_{i,j-1,k} + aT_i \Phi_{i,j,k+1} + aB_i \Phi_{i,j,k-1} + S_{i,j,k} \tag{84}$$

The goal is then to convert equation (80) to the form of equation (85)

$$\Phi_i = P_i \Phi_{i+1} + Q_i \tag{85}$$

This is done by first writing equation (80) in matrix form as shown in equation (86)

$$\begin{bmatrix} a_2 & b_2 & 0 & \dots & 0 \\ c_3 & a_3 & b_3 & 0 & \vdots \\ 0 & c_4 & a_4 & \ddots & 0 \\ \vdots & 0 & \ddots & \ddots & b_{N-1} \\ 0 & \dots & 0 & c_N & a_N \end{bmatrix} \begin{bmatrix} \Phi_2 \\ \Phi_3 \\ \Phi_4 \\ \vdots \\ \Phi_N \end{bmatrix} = \begin{bmatrix} d_2 + c_2 \Phi_1 \\ d_3 \\ d_4 \\ \vdots \\ d_N \end{bmatrix} \tag{86}$$

The goal now is to eliminate all the *c* terms from the matrix and force the diagonal terms of the first matrix to be 1. To do this, we define the following variables in equations (87) and (88).

$$P_i = \begin{cases} \dfrac{b_i}{a_i}, for\ i = 2 \\ \\ \dfrac{b_i}{a_i - c_i P_{i-1}}, for\ i > 2 \end{cases} \tag{87}$$

$$Q_n = \begin{cases} \dfrac{d_n + c_n \Phi_{n-1}}{a_n}, for\ n = 2 \\ \\ \dfrac{d_n + c_n Q_{n-1}}{a_n - c_n P_{n-1}}, for\ n > 2 \end{cases} \tag{88}$$

With these variables defined, we can rewrite equation (86) as the following.

$$\begin{bmatrix} 1 & P_2 & 0 & \dots & 0 \\ 0 & 1 & P_3 & 0 & \vdots \\ 0 & 0 & 1 & \ddots & 0 \\ \vdots & 0 & \ddots & \ddots & P_{N-1} \\ 0 & \dots & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \Phi_2 \\ \Phi_3 \\ \Phi_4 \\ \vdots \\ \Phi_N \end{bmatrix} = \begin{bmatrix} Q_2 \\ Q_3 \\ Q_4 \\ \vdots \\ Q_N \end{bmatrix} \tag{89}$$

Once all the values of equation (89) are determined, equation (85) can be used to solve for each electric potential at each node along the direction of the current sweep. The values obtained for the electric potential from one sweep are used during the next sweep in

that same direction. Essentially, the electric potentials are continuously updated throughout this process. Since the code developed in this work is three-dimensional, this process of sweeping and iteratively solving is executed in each direction. Convergence is checked for only after this process has been swept in all three directions. Convergence in this case applies globally, so all nodes within the computational domain must achieve convergence before the solution is considered to be correct.

Appendix B

Three-Dimensional C++ PIC Code

The three-dimensional particle-in-cell code used in this work is displayed in this section. The code was developed using Microsoft Visual Studio 2010 Professional. Three files are generated in the creation of this PIC code. The first is a "main.cpp" file which is the top level of the program and calls on function in the other two files. The other two files created are custom header files which essentially house all the functions called by the main program. These files are called "fields.h" and "push.h". Generally speaking, the "fields.h" file contains functions relating to the actual advancement of particles and their initialization. The "fields.h" file mostly contains functions relating the calculation of the electric potential, electric field, and other quantities involved in calculating these two. The following sections show each of the three files used for the code developed in this work.

B.1. Main.cpp

```
// main_v10 rpt main program

#include "stdafx.h"
#include <iostream>
#include <fstream>
#include <string>
#include "fields.h"
#include "push.h"
#include <math.h>
#include <time.h>
#include <vector>
#include <algorithm>
#define VECTORS 3   // constant for defining the number of vectors in several matrices
#define num_nodes_x 22   // number of nodes in the x direction
#define num_nodes_y 22   // number of nodes in the y direction
#define num_nodes_z 22   // number of nodes in the z direction
#define MAX_NODES 22     // maximum number of nodes in any direction
```

```
#define num_e 0          // number of electrons
#define num_Xe 0         // number of xenon ions
#define TOL 1e-6         // convergence tolerance for TDMA solver

using namespace std;

static float volt[num_nodes_x][num_nodes_y][num_nodes_z];
static float aE[num_nodes_x][num_nodes_y][num_nodes_z];
static float aW[num_nodes_x][num_nodes_y][num_nodes_z];
static float aS[num_nodes_x][num_nodes_y][num_nodes_z];
static float aN[num_nodes_x][num_nodes_y][num_nodes_z];
static float aT[num_nodes_x][num_nodes_y][num_nodes_z];
static float aB[num_nodes_x][num_nodes_y][num_nodes_z];
static float vol[num_nodes_x][num_nodes_y][num_nodes_z];
static float roe[num_nodes_x][num_nodes_y][num_nodes_z];
static float volt_old[num_nodes_x][num_nodes_y][num_nodes_z];
static float Ex[num_nodes_x][num_nodes_y][num_nodes_z];
static float Ey[num_nodes_x][num_nodes_y][num_nodes_z];
static float Ez[num_nodes_x][num_nodes_y][num_nodes_z];
static float Bx[num_nodes_x][num_nodes_y][num_nodes_z];
static float By[num_nodes_x][num_nodes_y][num_nodes_z];
static float Bz[num_nodes_x][num_nodes_y][num_nodes_z];
static float aP[num_nodes_x][num_nodes_y][num_nodes_z];
static float S[num_nodes_x][num_nodes_y][num_nodes_z];

static int e_countMain[num_nodes_x][num_nodes_y][num_nodes_z];
static int Xe_countMain[num_nodes_x][num_nodes_y][num_nodes_z];


int main(void)
{
// Inputs
int i,j,k;
const int time_steps = 10000; // Number of time steps
float DT = 1e-9;              // time step size (Seconds)
int flow_rate_e = 500;        // number of electrons introduced each time step
int flow_rate_Xe = 500;       // number of ions introduces each time step
int poisson_calls = 10;       // Number of Poisson calls
bool mid_solves = false;      // should multiple Poisson solves be used
bool ion_solve = false;       // should multiple solves be used in ion advance
int solve_step_e = 0;         // number of electrons moved per Poisson solve
int solve_step_Xe = 0;        // number of ions moved per Poisson solve
bool fourth_order = false;    // Should 4th order electric field be used

// Starting velocities
int V_max_e = 100000;         // maximum starting velocity for an electron (m/s)
int V_min_e = 100000;        // minimum starting velocity for an electron (m/s)
int V_max_Xe = 1000;          // maximum starting velocity for a Xenon ion (m/s)
int V_min_Xe = 1000;          // minimum starting velocity for a Xenon ion (m/s)

// boundary conditions
float x_pot_beg = 3.0;        // potential where x = 0 (west wall)
float x_pot_end = 6.0;        // potential where x = total x distance (east wall)
float y_pot_beg = 0.0;        // potential where y = 0 (south wall)
float y_pot_end = 0.0;        // potential where y = total y distance (north wall)
float z_pot_beg = 0.0;        // potential where z = 0 (bottom wall)
float z_pot_end = 0.0;        // potential where z = total z distance (top wall)
float pot_interior = 0.0;     // initial potential for interior nodes
```

```
bool x_diel_beg = false;      // true: dielectric where x=0 (west wall)
bool x_diel_end = false;      // true: dielectric where x=x_dist (east wall)
bool y_diel_beg = true;       // true: dielectric where y=0 (south wall)
bool y_diel_end = true;       // true: dielectric where y=y_dist (north wall)
bool z_diel_beg = true;       // true: dielectric where z=0 (bottom wall)
bool z_diel_end = true;       // true: dielectric where z=z_dist (top wall)

// physical parameters
float num_real = 500;         // Particle weight (real particles/computer particle)
const float M_E = 9.10938188e-31;          // mass of an electron (kg)
const float M_Xe = 1.67262178e-27;         // mass of a hydrogen ion (kg)
// const float M_Xe = 2.18012147e-25;      // mass of a Xenon ion (kg)
float Q_Xe = 1.60217646e-19;               // charge of a Xenon ion
float Q_E = -1.60217646e-19;               // charge of an electron (C)
const float ep_0 = 8.854187817620e-12;     // permittivity of free space (F/m)
const float x_dist = 1.0e-2;               // grid length in x direction (m)
const float y_dist = 1.0e-2;               // grid length in y direction (m)
const float z_dist = 1.0e-2;               // grid length in z direction (m)
const float x_start = 1.0e-3;              // injection offset, x direction (m)
const float y_start = 1.0e-3;              // injection offset, y direction (m)
const float z_start = 1.0e-3;              // injection offset, z direction (m)

// initalize velocity arrays
float* V_old_e[VECTORS+1]; // initialize v^(n-1/2) array for electrons
float* V_old_Xe[VECTORS+1];  // initialize v^(n-1/2) array for ions
float* V_new_Xe[VECTORS+1];  // initialize v^(n+1/2) array for ions
float* V_new_e[VECTORS+1];   // initialize v^(n+1/2) array for electrons

for (i=0; i<(VECTORS+1); i++)
{
       V_old_e[i] = new float [num_e];
       V_old_Xe[i] = new float [num_Xe];
       V_new_e[i] = new float [num_e];
       V_new_Xe[i] = new float [num_Xe];
}

srand(time(NULL));     // random number generator (time of day acts as seed)

// initalize positon arrays
float* pos_old_e[VECTORS+1];     // initialize x^n array for electrons
float* pos_new_e[VECTORS+1];     // initialize x^(n+1) array for electrons
float* pos_old_Xe[VECTORS+1];    // initialize x^n array for ions
float* pos_new_Xe[VECTORS+1];    // initialize x^(n+1) array for ions

for (i=0; i<(VECTORS+1); i++)
{
       pos_old_e[i] = new float [num_e];
       pos_old_Xe[i] = new float [num_Xe];
       pos_new_e[i] = new float [num_e];
       pos_new_Xe[i] = new float [num_Xe];
}

// grid setup
cout << "Initalizing Grid..." << "\t" ;
// declaring arrays and variables to be used in grid setup
// initalize arrays for CV face positions
float x_face_pos[num_nodes_x];
float y_face_pos[num_nodes_y];
```

```
float z_face_pos[num_nodes_z];

// initalize arrays for CV node (center) locations
float x_node_pos[num_nodes_x];
float y_node_pos[num_nodes_y];
float z_node_pos[num_nodes_z];

// initalize arrays for distance between CV NODES
float x_diff[num_nodes_x];
float y_diff[num_nodes_y];
float z_diff[num_nodes_z];

// initalize arrays to store cross sectinal areas
float A_x[num_nodes_y][num_nodes_z];
float A_y[num_nodes_z][num_nodes_x];
float A_z[num_nodes_x][num_nodes_y];

grid_setup(x_face_pos,y_face_pos,z_face_pos,x_node_pos,y_node_pos,z_node_pos,x_diff,y_
diff,z_diff,A_x,A_y,A_z,vol,x_dist,y_dist,z_dist);
cout << "done" << endl;

// initalize starting positions
cout << "initalizing Positions..." << "\t";
init_pos_v7(pos_old_e,pos_old_Xe,x_node_pos,y_node_pos,z_node_pos,x_start,y_start,z_st
art);
cout << "done" << endl;

// initalize starting velocities
cout << "initalizing velocities... " << "\t" ;
init_velocity_v3(V_old_e,V_max_e,V_min_e,V_old_Xe,V_max_Xe,V_min_Xe);
cout << "done" << endl;

// initialize charge denity (source term) and volt to zero
cout << "Finding coefficients..." << "\t" ;
for(i=0;i<num_nodes_x;i++)
{
	for(j=0;j<num_nodes_y;j++)
	{
		for(k=0;k<num_nodes_z;k++)
		{
			roe[i][j][k] = 0.0; //8.85e-7;
			volt[i][j][k] = 0.0;
		}
	}
}

// function to insert the boundary conditions into the grid
volt_bc(volt,x_pot_beg,x_pot_end,y_pot_beg,y_pot_end,z_pot_beg,z_pot_end,pot_interior)
;

// gets coefficients and calculates source term for the grid setup
poisson_solve_v2(vol,volt,x_diff,y_diff,z_diff,A_x,A_y,A_z,aE,aW,aN,aS,aT,aB,aP,S,roe,
ep_0,x_pot_beg,x_pot_end,y_pot_beg,y_pot_end,z_pot_beg,z_pot_end,x_diel_beg,
x_diel_end, y_diel_beg, y_diel_end, z_diel_beg, z_diel_end);
cout << "done" << endl;

// initalize inputs to the TDMA function (sweeping in x direction)
cout << "Performing first Poisson solve..." << "\t";
```

```
bool conv=false;
float vtempx[num_nodes_x];
float vtempy[num_nodes_y];
float vtempz[num_nodes_z];
float a[MAX_NODES];
float b[MAX_NODES];
float c[MAX_NODES];
float d[MAX_NODES];
float t1, t2, t3, t4;

for(i=0;i<num_nodes_x;i++)
{
	for(j=0;j<num_nodes_y;j++)
	{
		for(k=0;k<num_nodes_z;k++)
		{
			volt_old[i][j][k] = 0.0;
		}
	}
}

// sweeping in x direction TDMA
for(i=0;i<num_nodes_x;i++)
{
	for(j=0;j<num_nodes_y;j++)
	{
		for(k=0;k<num_nodes_z;k++)
		{
			volt_old[i][j][k] = volt[i][j][k];
		}
	}
}
while(!conv)
{
	// x sweep of tdma solver
	for(j=0;j<num_nodes_y;j++)
	{
		for(k=0;k<num_nodes_z;k++)
		{
			for(i=0;i<num_nodes_x;i++)
			{
				vtempx[i] = volt[i][j][k];
				a[i] = aP[i][j][k];
				b[i] = aE[i][j][k];
				c[i] = aW[i][j][k];

				if(j==(num_nodes_y-1))
				{
					t1 = 0.0;
				}
				else//if(j<(num_nodes_y-1))
				{
					t1 = aN[i][j][k]*volt[i][j+1][k];
				}
				if(j==0)
				{
					t2 = 0.0;
				}
```

```
                        else//if(j>0)
                        {
                                t2 = aS[i][j][k]*volt[i][j-1][k];
                        }
                        if(k==(num_nodes_z-1))
                        {
                                t3 = 0.0;
                        }
                        else//if(k<(num_nodes_z-1))
                        {
                                t3 = aT[i][j][k]*volt[i][j][k+1];
                        }
                        if(k==0)
                        {
                                t4 = 0.0;
                        }
                        else//if(k>0)
                        {
                                t4 = aB[i][j][k]*volt[i][j][k-1];
                        }
                        d[i] = t1+t2+t3+t4+S[i][j][k];
                }
                // send the newly defined a,b,c,d,n values to the TDMA solver
                tdma(a,b,c,d,vtempx,num_nodes_x);

                for(i=0;i<num_nodes_x;i++)
                {
                        volt[i][j][k] = vtempx[i];
                }
        }
} // ends x sweep of tdma solver

// y sweep of tdma
for(i=0;i<num_nodes_x;i++)
{
        for(k=0;k<num_nodes_z;k++)
        {
                for(j=0;j<num_nodes_y;j++)
                {
                        vtempy[j] = volt[i][j][k];
                        a[j] = aP[i][j][k];
                        b[j] = aN[i][j][k];
                        c[j] = aS[i][j][k];

                        if(i==(num_nodes_x-1))
                        {
                                t1 = 0.0;
                        }
                        else//if(i<(num_nodes_x-1))
                        {
                                t1 = aE[i][j][k]*volt[i+1][j][k];
                        }
                        if(i==0)
                        {
                                t2 = 0.0;
                        }
                        else//if(i>0)
                        {
```

```
                        t2 = aW[i][j][k]*volt[i-1][j][k];
                    }
                    if(k==(num_nodes_z-1))
                    {
                        t3 = 0.0;
                    }
                    else//if(k<(num_nodes_z-1))
                    {
                        t3 = aT[i][j][k]*volt[i][j][k+1];
                    }
                    if(k==0)
                    {
                        t4 = 0.0;
                    }
                    else//if(k>0)
                    {
                        t4 = aB[i][j][k]*volt[i][j][k-1];
                    }
                    d[j] = t1+t2+t3+t4+S[i][j][k];
                }
                // send the newly defined a,b,c,d,n values to the TDMA solver
                tdma(a,b,c,d,vtempy,num_nodes_y);

                for(j=0;j<num_nodes_y;j++)
                {
                    volt[i][j][k] = vtempy[j];
                }
        }
} // ends y sweep of tdma solver

// z sweep of tdma
for(j=0;j<num_nodes_y;j++)
{
        for(i=0;i<num_nodes_x;i++)
        {
                for(k=0;k<num_nodes_z;k++)
                {
                    vtempz[k] = volt[i][j][k];
                    a[k] = aP[i][j][k];
                    b[k] = aT[i][j][k];
                    c[k] = aB[i][j][k];
                        if(j==(num_nodes_y-1))
                    {
                        t1 = 0.0;
                    }
                    else//if(j<(num_nodes_y-1))
                    {
                        t1 = aN[i][j][k]*volt[i][j+1][k];
                    }
                    if(j==0)
                    {
                        t2 = 0.0;
                    }
                    else//if(j>0)
                    {
                        t2 = aS[i][j][k]*volt[i][j-1][k];
                    }
                    if(i==(num_nodes_x-1))
```

```
                                {
                                        t3 = 0.0;
                                }
                                else//if(i<(num_nodes_x-1))
                                {
                                        t3 = aE[i][j][k]*volt[i+1][j][k];
                                }
                                if(i==0)
                                {
                                        t4 = 0.0;
                                }
                                else//if(i>0)
                                {
                                        t4 = aW[i][j][k]*volt[i-1][j][k];
                                }
                                d[k] = t1+t2+t3+t4+S[i][j][k];
                        }
                        // send the newly defined a,b,c,d,n values to the TDMA solver
                        tdma(a,b,c,d,vtempz,num_nodes_z);

                        for(k=0;k<num_nodes_z;k++)
                        {
                                volt[i][j][k] = vtempz[k];
                        }
                }
        } // ends z sweep of tdma solver

        // convergence check
        conv = true;
        for(i=0;i<num_nodes_x;i++)
        {
                for(j=0;j<num_nodes_y;j++)
                {
                        for(k=0;k<num_nodes_z;k++)
                        {
                        if((abs(volt_old[i][j][k]-volt[i][j][k])/abs(volt[i][j][k]))>TOL)
                                {
                                        conv = false;
                                }
                        }
                }
        }

        if(!conv)
        {
                for(i=0;i<num_nodes_x;i++)
                {
                        for(j=0;j<num_nodes_y;j++)
                        {
                                for(k=0;k<num_nodes_z;k++)
                                {
                                        volt_old[i][j][k] = volt[i][j][k];
                                }
                        }
                }
        }
}
cout << "done" << endl;
```

```
// explicitly define the electric field on wall nodes to 0
cout << "Initalizing electric field..." << "\t" ;
e_field_edges(Ex,Ey,Ez);

// function to calculate electric field
e_field(Ex,Ey,Ez,volt,x_diff,y_diff,z_diff);
cout << "done" << endl;

// function to define the static magnetic field
cout << "Initalizing magnetic field..." << "\t";
b_fields(Bx,By,Bz);
cout << "done" << endl;

// Declare local variables to be used within the time loop
float E[VECTORS], B[VECTORS];
float V_minus[VECTORS], V_prime[VECTORS], V_plus[VECTORS], t[VECTORS], s[VECTORS];
int* indices_e[VECTORS];
int* indices_Xe[VECTORS];

for (i=0; i<VECTORS; i++)
{
       indices_e[i] = new int [num_e];
       indices_Xe[i] = new int [num_Xe];
}

string msg = "";
cout << "Notes: \n>";
getline(cin, msg);

// Create files to store results
cout << "Enter a reference name for this run: " << endl;
char fname_base[100];
cin >> fname_base;
char fname_out3[100];
char fname_out4[100];
char fname_out5[100];
char fname_out6[100];
char fname_out7[100];
char fname_out8[100];
char fname_out9[100];
char fname_out10[100];
char fname_out11[100];
char fname_out13[100];
char fname_out14[100];
char fname_out16[100];
char fname_out17[100];
char fname_out18[100];
char fname_out19[100];
char fname_out20[100];

strcpy(fname_out3,fname_base);
strcat(fname_out3,"_volt_xy.txt");
strcpy(fname_out4,fname_base);
strcat(fname_out4,"_run_parameters.txt");
strcpy(fname_out5,fname_base);
strcat(fname_out5,"_particle_num.txt");
strcpy(fname_out6,fname_base);
```

```
strcat(fname_out6,"_e_number_density.txt");
strcpy(fname_out10,fname_base);
strcat(fname_out10,"_xe_number_density.txt");
strcpy(fname_out7,fname_base);
strcat(fname_out7,"_Ex.txt");
strcpy(fname_out8,fname_base);
strcat(fname_out8,"_Ey.txt");
strcpy(fname_out9,fname_base);
strcat(fname_out9,"_Ez.txt");
strcpy(fname_out11,fname_base);
strcat(fname_out11,"_particle_diff.txt");
strcpy(fname_out13,fname_base);
strcat(fname_out13,"_volt-min-max.txt");
strcpy(fname_out14,fname_base);
strcat(fname_out14,"_max_velocity.txt");
strcpy(fname_out16,fname_base);
strcat(fname_out16,"average velocity.txt");
strcpy(fname_out17,fname_base);
strcat(fname_out17,"_electron_velocities.txt");
strcpy(fname_out18,fname_base);
strcat(fname_out18,"_ion_velocities.txt");
strcpy(fname_out19,fname_base);
strcat(fname_out19,"_volt_xz.txt");
strcpy(fname_out20,fname_base);
strcat(fname_out20,"volt_yz.txt");

ofstream outf3(fname_out3);
ofstream outf4(fname_out4);
ofstream outf5(fname_out5);
ofstream outf6(fname_out6);
ofstream outf7(fname_out7);
ofstream outf8(fname_out8);
ofstream outf9(fname_out9);
ofstream outf10(fname_out10);
ofstream outf11(fname_out11);
ofstream outf13(fname_out13);
ofstream outf14(fname_out14);
ofstream outf16(fname_out16);
ofstream outf17(fname_out17);
ofstream outf18(fname_out18);
ofstream outf19(fname_out19);
ofstream outf20(fname_out20);

outf13 << "step\tmaxV\ti\tj\tk\tminV\ti\tj\tk" << endl;
outf14 << "step\tion\telectron" << endl;
outf16 << "step\tion\telectron" << endl;

float e_max_vel, Xe_max_vel, e_avg_vel, Xe_avg_vel;

float comp_vol;
comp_vol = x_dist*y_dist*z_dist;
int num_cv;
num_cv = (num_nodes_x-2)*(num_nodes_y-2)*(num_nodes_z-2);

// writing parameters to the run parameter file:
time_t now = time(0);
char* dt = ctime(&now);
cout << "start time: " << dt << endl;
```

```
outf4 << "Start date and time: " << dt << endl;
outf4 << "notes: " << msg << endl;
outf4 << "time step size: " << DT << endl;
outf4 << "number of real particles per one computer particle: " << num_real << endl;
outf4 << "number of nodes (x,y,z): " << num_nodes_x << ", " << num_nodes_y << ", " <<
num_nodes_z << endl;
outf4 << "dimensions of computational space (x distance, y distance, z distance,
volume) [m]: " << x_dist << ", " << y_dist << ", " << z_dist << ", " << comp_vol <<
endl;
outf4 << "electron mass [kg]: " << M_E << endl;
outf4 << "ion mass [kg]: " << M_Xe << endl;
outf4 << "electron starting velocity range [m/s]: " << V_min_e << " to " << V_max_e <<
endl;
outf4 << "ion starting velocity range [m/s]: " << V_min_Xe << " to " << V_max_Xe <<
endl;
outf4 << "number of electrons introduced each time step (flow rate): " << flow_rate_e
<< endl;
outf4 << "number of ions introduced each time step (flow rate): " << flow_rate_Xe <<
endl;
outf4 << "Boundary conditions: " << endl;
outf4 << "Boundary\tDielectric\tVoltage" << endl;
outf4 << "East    \t" << x_diel_end << "\t" << x_pot_end << endl;
outf4 << "West    \t" << x_diel_beg << "\t" << x_pot_beg << endl;
outf4 << "North   \t" << y_diel_end << "\t" << y_pot_end << endl;
outf4 << "South   \t" << y_diel_beg << "\t" << y_pot_beg << endl;
outf4 << "Top     \t" << z_diel_end << "\t" << z_pot_end << endl;
outf4 << "Bottom  \t" << z_diel_beg << "\t" << z_pot_beg << endl;
if(mid_solves)
{
       outf4 << "Poisson solver called multiple times during timestep" << endl;
       outf4 << "Number of Poisson calls during electron time step:\t" <<
       poisson_calls << endl;
}
outf4 << "Fourth order electric field (0=no, 1=yes): " << fourth_order << endl;

i=0;
j=0;
k=0;
int r,p; //,check;
int mid_node_x = (num_nodes_x-1)/2;
int mid_node_y = (num_nodes_y-1)/2;
int mid_node_z = (num_nodes_z-1)/2;

// initalize a timer for the entire simulation
time_t start,end;
time(&start);

int x,y,z,o;
float vmag=0.0;
float maxV, minV;
int maxVpos[VECTORS], minVpos[VECTORS];
bool speedy = false;
int ind[VECTORS];

cout << "Beginning time step iterations" << endl;
cout << "Percent Complete:" << endl;
int one_per, percent, new_size_e, new_size_Xe, old_size_e, old_size_Xe,
num_absorbed_e, num_absorbed_Xe, l, reduce_inj_rate;
```

```
one_per = time_steps/100;
percent = 0;
reduce_inj_rate = 1;
new_size_Xe = num_Xe;
old_size_Xe = new_size_Xe;
new_size_e = num_e;
old_size_e = new_size_e;
int inner_count_e = 0;
int inner_count_Xe = 0;

outf3 << "time = 0" << endl;
outf6 << "time = 0" << endl;
outf10 << "time = 0" << endl;
outf11 << "time = 0" << endl;
for(i=0;i<num_nodes_x;i++)
{
	for(j=0;j<num_nodes_y;j++)
	{
		outf3 << volt[i][j][mid_node_z] << "\t" ;
		outf6 << 0 << "\t" ;
		outf10 << 0 << "\t" ;
		outf11 << 0 << "\t" ;
	}
	outf3 << endl;
	outf6 << endl;
	outf10 << endl;
	outf11 << endl;
}

outf19 << "time = 0" << endl;
for(i=0;i<num_nodes_x;i++)
{
	for(k=0;k<num_nodes_z;k++)
	{
		outf19 << volt[i][mid_node_y][k] << "\t" ;
	}
	outf19 << endl;
}
outf20 << "time = 0" << endl;
for(j=0;j<num_nodes_y;j++)
{
	for(k=0;k<num_nodes_z;k++)
	{
		outf20 << volt[mid_node_x][j][k] << "\t" ;
	}
	outf20 << endl;
}

////////////////// Begin time iterations ////////////////////////
for(int step=1 ; step<time_steps ; step++)
{
	if(step%one_per == 0)
	{
		percent++;
		cout << "\b\b" << percent;
	}
//// pre-leapfrog calculations of charge density, potential, and electric field ////
```

```
// determine which control volume each particle is currently in
indices_v2(pos_old_e,pos_old_Xe,indices_e,indices_Xe,x_dist,y_dist,z_dist,new_s
ize_e,new_size_Xe);

// calculate charge density for the current time step
calc_roe_v3(Q_Xe,roe,num_real,new_size_e,new_size_Xe,vol,pos_old_e,pos_old_Xe,x
_diff[2],y_diff[2],z_diff[2],x_node_pos,y_node_pos,z_node_pos);

// function to calculate source terms for during the time step
source_solve(vol,S,roe,ep_0,x_pot_beg,x_pot_end,y_pot_beg,y_pot_end,z_pot_beg,z
_pot_end,x_diel_beg, x_diel_end, y_diel_beg, y_diel_end, z_diel_beg,
z_diel_end);

// calculate electric potential for the current time step (TDMA)
conv = false;

while(!conv)
{
	// x sweep of tdma solver
	for(j=0;j<num_nodes_y;j++)
	{
		for(k=0;k<num_nodes_z;k++)
		{
			for(i=0;i<num_nodes_x;i++)
			{
				vtempx[i] = volt[i][j][k];
				a[i] = aP[i][j][k];
				b[i] = aE[i][j][k];
				c[i] = aW[i][j][k];

				if(j==(num_nodes_y-1))
				{
					t1 = 0.0;
				}
				else//if(j<(num_nodes_y-1))
				{
					t1 = aN[i][j][k]*volt[i][j+1][k];
				}
				if(j==0)
				{
					t2 = 0.0;
				}
				else//if(j>0)
				{
					t2 = aS[i][j][k]*volt[i][j-1][k];
				}
				if(k==(num_nodes_z-1))
				{
					t3 = 0.0;
				}
				else//if(k<(num_nodes_z-1))
				{
					t3 = aT[i][j][k]*volt[i][j][k+1];
				}
				if(k==0)
				{
					t4 = 0.0;
				}
```

```
                        else//if(k>0)
                        {
                                t4 = aB[i][j][k]*volt[i][j][k-1];
                        }
                        d[i] = t1+t2+t3+t4+S[i][j][k];
                }
        // send the newly defined a,b,c,d,n values to the TDMA solver
                tdma(a,b,c,d,vtempx,num_nodes_x);
                for(i=0;i<num_nodes_x;i++)
                {
                        volt[i][j][k] = vtempx[i];
                }
        }
} // ends x sweep of tdma solver
        // y sweep of tdma
for(i=0;i<num_nodes_x;i++)
{
        for(k=0;k<num_nodes_z;k++)
        {
                for(j=0;j<num_nodes_y;j++)
                {
                        vtempy[j] = volt[i][j][k];
                        a[j] = aP[i][j][k];
                        b[j] = aN[i][j][k];
                        c[j] = aS[i][j][k];

                        if(i==(num_nodes_x-1))
                        {
                                t1 = 0.0;
                        }
                        else//if(i<(num_nodes_x-1))
                        {
                                t1 = aE[i][j][k]*volt[i+1][j][k];
                        }
                        if(i==0)
                        {
                                t2 = 0.0;
                        }
                        else//if(i>0)
                        {
                                t2 = aW[i][j][k]*volt[i-1][j][k];
                        }
                        if(k==(num_nodes_z-1))
                        {
                                t3 = 0.0;
                        }
                        else//if(k<(num_nodes_z-1))
                        {
                                t3 = aT[i][j][k]*volt[i][j][k+1];
                        }
                        if(k==0)
                        {
                                t4 = 0.0;
                        }
                        else//if(k>0)
                        {
                                t4 = aB[i][j][k]*volt[i][j][k-1];
                        }
```

```
                        d[j] = t1+t2+t3+t4+S[i][j][k];
                }
        // send the newly defined a,b,c,d,n values to the TDMA solver
                tdma(a,b,c,d,vtempy,num_nodes_y);

                for(j=0;j<num_nodes_y;j++)
                {
                        volt[i][j][k] = vtempy[j];
                }
        }
} // ends y sweep of tdma solver

// z sweep of tdma
for(j=0;j<num_nodes_y;j++)
{
        for(i=0;i<num_nodes_x;i++)
        {
                for(k=0;k<num_nodes_z;k++)
                {
                        vtempz[k] = volt[i][j][k];
                        a[k] = aP[i][j][k];
                        b[k] = aT[i][j][k];
                        c[k] = aB[i][j][k];

                        if(j==(num_nodes_y-1))
                        {
                                t1 = 0.0;
                        }
                        else//if(j<(num_nodes_y-1))
                        {
                                t1 = aN[i][j][k]*volt[i][j+1][k];
                        }
                        if(j==0)
                        {
                                t2 = 0.0;
                        }
                        else//if(j>0)
                        {
                                t2 = aS[i][j][k]*volt[i][j-1][k];
                        }
                        if(i==(num_nodes_x-1))
                        {
                                t3 = 0.0;
                        }
                        else//if(i<(num_nodes_x-1))
                        {
                                t3 = aE[i][j][k]*volt[i+1][j][k];
                        }
                        if(i==0)
                        {
                                t4 = 0.0;
                        }
                        else//if(i>0)
                        {
                                t4 = aW[i][j][k]*volt[i-1][j][k];
                        }
                        d[k] = t1+t2+t3+t4+S[i][j][k];
                }
```

```
                // send the newly defined a,b,c,d,n values to the TDMA solver
                        tdma(a,b,c,d,vtempz,num_nodes_z);

                        for(k=0;k<num_nodes_z;k++)
                        {
                                volt[i][j][k] = vtempz[k];
                        }
                }
        } // ends z sweep of tdma solver

        // convergence check
        conv = true;
        for(i=0;i<num_nodes_x;i++)
        {
                for(j=0;j<num_nodes_y;j++)
                {
                        for(k=0;k<num_nodes_z;k++)
                        {
                                if((abs(volt_old[i][j][k]-
                        volt[i][j][k])/abs(volt[i][j][k]))>TOL)
                                {
                                        conv = false;
                                }
                        }
                }
        }

        if(!conv)
        {
                for(i=0;i<num_nodes_x;i++)
                {
                        for(j=0;j<num_nodes_y;j++)
                        {
                                for(k=0;k<num_nodes_z;k++)
                                {
                                        volt_old[i][j][k] = volt[i][j][k];
                                }
                        }
                }
        }
        }
// finding minimum and maximum potential in the domain
maxV = -1e6;
minV = 1e6;
for(i=0;i<VECTORS;i++)
{
        maxVpos[i] = 0;
        minVpos[i] = 0;
}
for(i=0;i<num_nodes_x;i++)
{
        for(j=0;j<num_nodes_y;j++)
        {
                for(k=0;k<num_nodes_z;k++)
                {
                        if(volt[i][j][k] > maxV)
                        {
                                maxV = volt[i][j][k];
```

```
                                maxVpos[0] = i;
                                maxVpos[1] = j;
                                maxVpos[2] = k;
                        }
                        if(volt[i][j][k] < minV)
                        {
                                minV = volt[i][j][k];
                                minVpos[0] = i;
                                minVpos[1] = j;
                                minVpos[2] = k;
                        }
                }
        }
        outf13 << step << "\t" << maxV << "\t" << maxVpos[0] << "\t" << maxVpos[1] <<
"\t" << maxVpos[2] << "\t" << minV << "\t" << minVpos[0] << "\t" << minVpos[1] << "\t"
<< minVpos[2] << endl;


        // calculate the electric field for the current time step
        e_field(Ex,Ey,Ez,volt,x_diff,y_diff,z_diff);

        for(i=0;i<num_nodes_x;i++)
        {
                for(j=0;j<num_nodes_y;j++)
                {
                        for(k=0;k<num_nodes_z;k++)
                        {
                                e_countMain[i][j][k] = 0;
                                Xe_countMain[i][j][k] = 0;
                        }
                }
        }

        // determine the number of electrons in each control volume
        for(o=0;o<new_size_e;o++)
        {
                x = indices_e[0][o];
                y = indices_e[1][o];
                z = indices_e[2][o];
                e_countMain[x][y][z]++;
        }

        // determine the number Xenon ions in each control volume
        for(o=0;o<new_size_Xe;o++)
        {
                x=indices_Xe[0][o];
                y=indices_Xe[1][o];
                z=indices_Xe[2][o];
                Xe_countMain[x][y][z]++;
        }

        outf3 << "time = " << step*DT << endl;
        outf6 << "time = " << step*DT << endl;
        outf10 << "time = " << step*DT << endl;
        outf7 << "time = " << step*DT << endl;
        outf8 << "time = " << step*DT << endl;
        outf9 << "time = " << step*DT << endl;
```

```
	outf11 << "time = " << step*DT << endl;
	for(i=0;i<num_nodes_x;i++)
	{
		for(j=0;j<num_nodes_y;j++)
		{
			outf3 << volt[i][j][mid_node_z] << "\t" ;
			outf7 << Ex[i][j][mid_node_z] << "\t" ;
			outf8 << Ey[i][j][mid_node_z] << "\t" ;
			outf9 << Ez[i][j][mid_node_z] << "\t" ;
			outf6 << e_countMain[i][j][mid_node_z] << "\t" ;
			outf10 << Xe_countMain[i][j][mid_node_z] << "\t" ;
			outf11 << Xe_countMain[i][j][mid_node_z]-
e_countMain[i][j][mid_node_z] << "\t" ;
		}
		outf3 << endl;
		outf7 << endl;
		outf8 << endl;
		outf9 << endl;
		outf6 << endl;
		outf10 << endl;
		outf11 << endl;
	}

	outf19 << "time = " << step*DT << endl;
	for(i=0;i<num_nodes_x;i++)
	{
		for(k=0;k<num_nodes_z;k++)
		{
			outf19 << volt[i][mid_node_y][k] << "\t" ;
		}
		outf19 << endl;
	}

	outf20 << "time = " << step*DT << endl;
	for(j=0;j<num_nodes_y;j++)
	{
		for(k=0;k<num_nodes_z;k++)
		{
			outf20 << volt[mid_node_x][j][k] << "\t" ;
		}
		outf20 << endl;
	}

	num_absorbed_e = 0;
	num_absorbed_Xe = 0;

	// write the number of electrons and ions to file for the current time step
	outf5 << new_size_e << "\t" << new_size_Xe << endl;

	e_max_vel = 0.0;
	Xe_max_vel = 0.0;
// end of pre-leapfrog calculations of charge density, potential, and electric field /

////////////////////////////// leapfrog advance for ions ////////////////////////////

	solve_step_Xe = (int)(new_size_Xe/(1+poisson_calls))+1;

	for(p=0;p<new_size_Xe;p++)
```

```
	{
		// multiple solves routine for ions
		if(p%solve_step_Xe == 0 && mid_solves && (p+solve_step_Xe)<new_size_Xe
&& ion_solve)
		{

		indices_v2(pos_old_e,pos_old_Xe,indices_e,indices_Xe,x_dist,y_dist,z_dis
		t,new_size_e,new_size_Xe);

			// calculate charge density for the current time step

	calc_roe_mid_push_v2(Q_Xe,roe,num_real,new_size_e,new_size_Xe,vol,pos_old_e,pos
_old_Xe,x_diff[2],y_diff[2],z_diff[2],x_node_pos,y_node_pos,z_node_pos);

		// function to calculate source terms for during the time step
	source_solve(vol,S,roe,ep_0,x_pot_beg,x_pot_end,y_pot_beg,y_pot_end,z_pot_beg,z
_pot_end,x_diel_beg, x_diel_end, y_diel_beg, y_diel_end, z_diel_beg, z_diel_end);

// calculate voltage for the current time step after partial electron move (TDMA)
			conv = false;
			while(!conv)
			{
				// x sweep of tdma solver
				for(j=0;j<num_nodes_y;j++)
				{
					for(k=0;k<num_nodes_z;k++)
					{
						for(i=0;i<num_nodes_x;i++)
						{
							vtempx[i] = volt[i][j][k];
							a[i] = aP[i][j][k];
							b[i] = aE[i][j][k];
							c[i] = aW[i][j][k];

							if(j==(num_nodes_y-1))
							{
								t1 = 0.0;
							}
							else//if(j<(num_nodes_y-1))
							{
								t1 =
aN[i][j][k]*volt[i][j+1][k];
							}
							if(j==0)
							{
								t2 = 0.0;
							}
							else//if(j>0)
							{
								t2 = aS[i][j][k]*volt[i][j-
1][k];
							}
							if(k==(num_nodes_z-1))
							{
								t3 = 0.0;
							}
							else//if(k<(num_nodes_z-1))
							{
```

```
                                                t3 =
aT[i][j][k]*volt[i][j][k+1];
                                        }
                                        if(k==0)
                                        {
                                                t4 = 0.0;
                                        }
                                        else//if(k>0)
                                        {
                                                t4 = aB[i][j][k]*volt[i][j][k-
1];
                                        }
                                        d[i] = t1+t2+t3+t4+S[i][j][k];
                                }
                        // send the newly defined a,b,c,d,n values to the TDMA solver
                                tdma(a,b,c,d,vtempx,num_nodes_x);

                                for(i=0;i<num_nodes_x;i++)
                                {
                                        volt[i][j][k] = vtempx[i];
                                }
                        }
                } // ends x sweep of tdma solver

                // y sweep of tdma
                for(i=0;i<num_nodes_x;i++)
                {
                        for(k=0;k<num_nodes_z;k++)
                        {
                                for(j=0;j<num_nodes_y;j++)
                                {
                                        vtempy[j] = volt[i][j][k];
                                        a[j] = aP[i][j][k];
                                        b[j] = aN[i][j][k];
                                        c[j] = aS[i][j][k];

                                        if(i==(num_nodes_x-1))
                                        {
                                                t1 = 0.0;
                                        }
                                        else//if(i<(num_nodes_x-1))
                                        {
                                                t1 =
aE[i][j][k]*volt[i+1][j][k];
                                        }
                                        if(i==0)
                                        {
                                                t2 = 0.0;
                                        }
                                        else//if(i>0)
                                        {
                                                t2 = aW[i][j][k]*volt[i-
1][j][k];
                                        }
                                        if(k==(num_nodes_z-1))
                                        {
                                                t3 = 0.0;
                                        }
```

```
                                        else//if(k<(num_nodes_z-1))
                                        {
                                                t3 =
aT[i][j][k]*volt[i][j][k+1];
                                        }
                                        if(k==0)
                                        {
                                                t4 = 0.0;
                                        }
                                        else//if(k>0)
                                        {
                                                t4 = aB[i][j][k]*volt[i][j][k-
1];
                                        }
                                        d[j] = t1+t2+t3+t4+S[i][j][k];
                                }
                        // send the newly defined a,b,c,d,n values to the TDMA solver
                                tdma(a,b,c,d,vtempy,num_nodes_y);
                                for(j=0;j<num_nodes_y;j++)
                                {
                                        volt[i][j][k] = vtempy[j];
                                }
                        }
                } // ends y sweep of tdma solver
                // z sweep of tdma
                for(j=0;j<num_nodes_y;j++)
                {
                        for(i=0;i<num_nodes_x;i++)
                        {
                                for(k=0;k<num_nodes_z;k++)
                                {
                                        vtempz[k] = volt[i][j][k];
                                        a[k] = aP[i][j][k];
                                        b[k] = aT[i][j][k];
                                        c[k] = aB[i][j][k];
                                        if(j==(num_nodes_y-1))
                                        {
                                                t1 = 0.0;
                                        }
                                        else//if(j<(num_nodes_y-1))
                                        {
                                                t1 =
aN[i][j][k]*volt[i][j+1][k];
                                        }
                                        if(j==0)
                                        {
                                                t2 = 0.0;
                                        }
                                        else//if(j>0)
                                        {
                                                t2 = aS[i][j][k]*volt[i][j-
1][k];
                                        }
                                        if(i==(num_nodes_x-1))
                                        {
                                                t3 = 0.0;
                                        }
                                        else//if(i<(num_nodes_x-1))
```

```
                                                {
                                                        t3 =
aE[i][j][k]*volt[i+1][j][k];
                                                }
                                                if(i==0)
                                                {
                                                        t4 = 0.0;
                                                }
                                                else//if(i>0)
                                                {
                                                        t4 = aW[i][j][k]*volt[i-
1][j][k];
                                                }
                                                d[k] = t1+t2+t3+t4+S[i][j][k];
                                        }
                        // send the newly defined a,b,c,d,n values to the TDMA solver
                                        tdma(a,b,c,d,vtempz,num_nodes_z);

                                        for(k=0;k<num_nodes_z;k++)
                                        {
                                                volt[i][j][k] = vtempz[k];
                                        }
                                }
                        } // ends z sweep of tdma solver

                        // convergence check
                        conv = true;
                        for(i=0;i<num_nodes_x;i++)
                        {
                                for(j=0;j<num_nodes_y;j++)
                                {
                                        for(k=0;k<num_nodes_z;k++)
                                        {
                                                if((abs(volt_old[i][j][k]-
volt[i][j][k])/abs(volt[i][j][k]))>TOL)
                                                {
                                                        conv = false;
                                                }
                                        }
                                }
                        }
                        if(!conv)
                        {
                                for(i=0;i<num_nodes_x;i++)
                                {
                                        for(j=0;j<num_nodes_y;j++)
                                        {
                                                for(k=0;k<num_nodes_z;k++)
                                                {
                                                        volt_old[i][j][k] =
volt[i][j][k];
                                                }
                                        }
                                }
                        }
                        e_field(Ex,Ey,Ez,volt,x_diff,y_diff,z_diff);
                }
```

```
		for(i=0;i<VECTORS;i++)
		{
			ind[i] = indices_Xe[i][p];
		}
		if(fourth_order)
		{
			E[0] = order_4_E_v4(V_old_Xe[0][p],DT,0,ind,Ex,x_diff[2]);
			E[1] = order_4_E_v4(V_old_Xe[1][p],DT,1,ind,Ey,y_diff[2]);
			E[2] = order_4_E_v4(V_old_Xe[2][p],DT,2,ind,Ez,z_diff[2]);
		}
		else
		{
			E[0] = Ex[ind[0]][ind[1]][ind[2]];
			E[1] = Ey[ind[0]][ind[1]][ind[2]];
			E[2] = Ez[ind[0]][ind[1]][ind[2]];
		}
		B[0] = Bx[ind[0]][ind[1]][ind[2]];
		B[1] = By[ind[0]][ind[1]][ind[2]];
		B[2] = Bz[ind[0]][ind[1]][ind[2]];

		// First electric field push
		if(V_old_Xe[3][p] > 0)
		{
			for(i=0; i<VECTORS; i++)
			{
				V_minus[i] = V_old_Xe[i][p];
			}
			V_old_Xe[3][p] = -1.0;
		}
		else
		{
			for(i=0; i<VECTORS; i++)
			{
			     V_minus[i] = calc_v_minus(Q_Xe,E[i],DT,M_Xe,V_old_Xe[i][p]);
			}
		}
		// Magnetic field push
		for(i=0; i<VECTORS; i++)
		{
			t[i] = calc_t(Q_Xe,B[i],DT,M_Xe);
			s[i] = calc_s(t[i]);
		}
		calc_v_prime(V_minus,t,V_prime);
		calc_v_plus(V_minus,V_prime,s,V_plus);

		// Second Electric field push
		for(i=0; i<VECTORS; i++)
			V_new_Xe[i][p] = calc_v(V_plus[i],Q_Xe,E[i],DT,M_Xe);

			vmag =
sqrt(pow(V_new_Xe[0][p],2)+pow(V_new_Xe[1][p],2)+pow(V_new_Xe[2][p],2));
		if(vmag > Xe_max_vel)
		{
			Xe_max_vel = vmag;
		}

		// total particle advance
```

```
              for(i=0; i<VECTORS; i++)
              {
                     pos_new_Xe[i][p] = calc_x(pos_old_Xe[i][p],V_new_Xe[i][p],DT);
                     pos_old_Xe[i][p] = pos_new_Xe[i][p];
              }

            // function to check particle position and reflect off walls if necessary
              if(x_diel_end || x_diel_beg || y_diel_end || y_diel_beg || z_diel_end ||
z_diel_beg)
              {
                     pos_new_Xe[0][p] =
check_reflect(pos_new_Xe[0][p],x_dist,0,p,V_new_Xe,x_diel_beg,x_diel_end);
                     pos_new_Xe[1][p] =
check_reflect(pos_new_Xe[1][p],y_dist,1,p,V_new_Xe,y_diel_beg,y_diel_end);
                     pos_new_Xe[2][p] =
check_reflect(pos_new_Xe[2][p],z_dist,2,p,V_new_Xe,z_diel_beg,z_diel_end);
              }

              // check particle-wall absorption
              if(pos_new_Xe[0][p] >= x_dist || pos_new_Xe[0][p] <= 0 ||
pos_new_Xe[1][p] >= y_dist || pos_new_Xe[1][p] <= 0 || pos_new_Xe[2][p] >= z_dist ||
pos_new_Xe[2][p] <= 0)
              {
                     pos_new_Xe[3][p] = 1;
                     pos_old_Xe[3][p] = 1;
                     num_absorbed_Xe++;
              }
              else
              {
                     pos_new_Xe[3][p] = 0;
                     pos_old_Xe[3][p] = 0;
              }
       }
//////////////////////////// end of leapfrog advance for ions ////////////////////////

//////////////////////////// Leap Frog advance for electrons ////////////////////////
       solve_step_e = (int)(new_size_e/(1+poisson_calls))+1;

       for(r=0;r<new_size_e;r++)
       {
              // Multiple Poisson solves routine for the electron push
              if(r%solve_step_e == 0 && mid_solves && (r+solve_step_e)<new_size_e)
              {
                     indices_v2(pos_old_e,pos_old_Xe,indices_e,indices_Xe,x_dist,y_dis
                     t,z_dist,new_size_e,new_size_Xe);

                     // calculate charge density for the current time step
                     calc_roe_mid_push_v2(Q_Xe,roe,num_real,new_size_e,new_size_Xe,vol
                     ,pos_old_e,pos_old_Xe,x_diff[2],y_diff[2],z_diff[2],x_node_pos,y_
                     node_pos,z_node_pos);

                     // function to calculate source terms for during the time step

                     source_solve(vol,S,roe,ep_0,x_pot_beg,x_pot_end,y_pot_beg,y_pot_e
                     nd,z_pot_beg,z_pot_end,x_diel_beg, x_diel_end, y_diel_beg,
                     y_diel_end, z_diel_beg, z_diel_end);

// calculate voltage for the current time step after partial electron move (TDMA)
```

```
                        conv = false;

                        while(!conv)
                        {
                                // x sweep of tdma solver
                                for(j=0;j<num_nodes_y;j++)
                                {
                                        for(k=0;k<num_nodes_z;k++)
                                        {
                                                for(i=0;i<num_nodes_x;i++)
                                                {
                                                        vtempx[i] = volt[i][j][k];
                                                        a[i] = aP[i][j][k];
                                                        b[i] = aE[i][j][k];
                                                        c[i] = aW[i][j][k];
                                                        if(j==(num_nodes_y-1))
                                                        {
                                                                t1 = 0.0;
                                                        }
                                                        else//if(j<(num_nodes_y-1))
                                                        {
                                                                t1 =
aN[i][j][k]*volt[i][j+1][k];
                                                        }
                                                        if(j==0)
                                                        {
                                                                t2 = 0.0;
                                                        }
                                                        else//if(j>0)
                                                        {
                                                                t2 = aS[i][j][k]*volt[i][j-
1][k];
                                                        }
                                                        if(k==(num_nodes_z-1))
                                                        {
                                                                t3 = 0.0;
                                                        }
                                                        else//if(k<(num_nodes_z-1))
                                                        {
                                                                t3 =
aT[i][j][k]*volt[i][j][k+1];
                                                        }
                                                        if(k==0)
                                                        {
                                                                t4 = 0.0;
                                                        }
                                                        else//if(k>0)
                                                        {
                                                                t4 = aB[i][j][k]*volt[i][j][k-
1];
                                                        }
                                                        d[i] = t1+t2+t3+t4+S[i][j][k];
                                                }
                        // send the newly defined a,b,c,d,n values to the TDMA solver
                                                tdma(a,b,c,d,vtempx,num_nodes_x);
                                                for(i=0;i<num_nodes_x;i++)
                                                {
                                                        volt[i][j][k] = vtempx[i];
```

```
                                        }
                                }
                        } // ends x sweep of tdma solver
                        // y sweep of tdma
                        for(i=0;i<num_nodes_x;i++)
                        {
                                for(k=0;k<num_nodes_z;k++)
                                {
                                        for(j=0;j<num_nodes_y;j++)
                                        {
                                                vtempy[j] = volt[i][j][k];
                                                a[j] = aP[i][j][k];
                                                b[j] = aN[i][j][k];
                                                c[j] = aS[i][j][k];
                                                if(i==(num_nodes_x-1))
                                                {
                                                        t1 = 0.0;
                                                }
                                                else//if(i<(num_nodes_x-1))
                                                {
                                                        t1 =
aE[i][j][k]*volt[i+1][j][k];
                                                }
                                                if(i==0)
                                                {
                                                        t2 = 0.0;
                                                }
                                                else//if(i>0)
                                                {
                                                        t2 = aW[i][j][k]*volt[i-
1][j][k];
                                                }
                                                if(k==(num_nodes_z-1))
                                                {
                                                        t3 = 0.0;
                                                }
                                                else//if(k<(num_nodes_z-1))
                                                {
                                                        t3 =
aT[i][j][k]*volt[i][j][k+1];
                                                }
                                                if(k==0)
                                                {
                                                        t4 = 0.0;
                                                }
                                                else//if(k>0)
                                                {
                                                        t4 = aB[i][j][k]*volt[i][j][k-
1];
                                                }
                                                d[j] = t1+t2+t3+t4+S[i][j][k];
                                        }
                        // send the newly defined a,b,c,d,n values to the TDMA solver
                                        tdma(a,b,c,d,vtempy,num_nodes_y);
                                        for(j=0;j<num_nodes_y;j++)
                                        {
                                                volt[i][j][k] = vtempy[j];
                                        }
```

```
                        }
                } // ends y sweep of tdma solver
                // z sweep of tdma
                for(j=0;j<num_nodes_y;j++)
                {
                        for(i=0;i<num_nodes_x;i++)
                        {
                                for(k=0;k<num_nodes_z;k++)
                                {
                                        vtempz[k] = volt[i][j][k];
                                        a[k] = aP[i][j][k];
                                        b[k] = aT[i][j][k];
                                        c[k] = aB[i][j][k];
                                        if(j==(num_nodes_y-1))
                                        {
                                                t1 = 0.0;
                                        }
                                        else//if(j<(num_nodes_y-1))
                                        {
                                                t1 =
aN[i][j][k]*volt[i][j+1][k];
                                        }
                                        if(j==0)
                                        {
                                                t2 = 0.0;
                                        }
                                        else//if(j>0)
                                        {
                                                t2 = aS[i][j][k]*volt[i][j-
1][k];
                                        }
                                        if(i==(num_nodes_x-1))
                                        {
                                                t3 = 0.0;
                                        }
                                        else//if(i<(num_nodes_x-1))
                                        {
                                                t3 =
aE[i][j][k]*volt[i+1][j][k];
                                        }
                                        if(i==0)
                                        {
                                                t4 = 0.0;
                                        }
                                        else//if(i>0)
                                        {
                                                t4 = aW[i][j][k]*volt[i-
1][j][k];
                                        }
                                                d[k] = t1+t2+t3+t4+S[i][j][k];
                                }
                        // send the newly defined a,b,c,d,n values to the TDMA solver
                                tdma(a,b,c,d,vtempz,num_nodes_z);
                                for(k=0;k<num_nodes_z;k++)
                                {
                                        volt[i][j][k] = vtempz[k];
                                }
                        }
```

```
                        } // ends z sweep of tdma solver
                        // convergence check
                        conv = true;
                        for(i=0;i<num_nodes_x;i++)
                        {
                                for(j=0;j<num_nodes_y;j++)
                                {
                                        for(k=0;k<num_nodes_z;k++)
                                        {
                                                if((abs(volt_old[i][j][k]-
volt[i][j][k])/abs(volt[i][j][k]))>TOL)
                                                {
                                                        conv = false;
                                                }
                                        }
                                }
                        }
                        if(!conv)
                        {
                                for(i=0;i<num_nodes_x;i++)
                                {
                                        for(j=0;j<num_nodes_y;j++)
                                        {
                                                for(k=0;k<num_nodes_z;k++)
                                                {
                                                        volt_old[i][j][k] =
volt[i][j][k];
                                                }
                                        }
                                }
                        }
                }
                e_field(Ex,Ey,Ez,volt,x_diff,y_diff,z_diff);
        }
        //end of multiple Poisson solves routine for the electron push
        for(i=0;i<VECTORS;i++)
        {
                ind[i] = indices_e[i][r];
        }
        if(fourth_order)
        {
                E[0] = order_4_E_v4(V_old_e[0][r],DT,0,ind,Ex,x_diff[2]);
                E[1] = order_4_E_v4(V_old_e[1][r],DT,1,ind,Ey,y_diff[2]);
                E[2] = order_4_E_v4(V_old_e[2][r],DT,2,ind,Ez,z_diff[2]);
        }
        else
        {
                E[0] = Ex[ind[0]][ind[1]][ind[2]];
                E[1] = Ey[ind[0]][ind[1]][ind[2]];
                E[2] = Ez[ind[0]][ind[1]][ind[2]];
        }
        B[0] = Bx[ind[0]][ind[1]][ind[2]];
        B[1] = By[ind[0]][ind[1]][ind[2]];
        B[2] = Bz[ind[0]][ind[1]][ind[2]];

        // First electric field push
        if(V_old_e[3][r] >0)
        {
```

```
                        for(i=0; i<VECTORS; i++)
                        {
                                V_minus[i] = V_old_e[i][r];
                        }
                        V_old_e[3][r] = -1.0;
                }
                else
                {
                        for(i=0; i<VECTORS; i++)
                        {
                                V_minus[i] = calc_v_minus(Q_E,E[i],DT,M_E,V_old_e[i][r]);
                        }
                }

                // Magnetic field push
                for(i=0; i<VECTORS; i++)
                {
                        t[i] = calc_t(Q_E,B[i],DT,M_E);
                        s[i] = calc_s(t[i]);
                }
                calc_v_prime(V_minus,t,V_prime);
                calc_v_plus(V_minus,V_prime,s,V_plus);

                // Second Electric field push
                for(i=0; i<VECTORS; i++)
                        V_new_e[i][r] = calc_v(V_plus[i],Q_E,E[i],DT,M_E);

                vmag =
sqrt(pow(V_new_e[0][r],2)+pow(V_new_e[1][r],2)+pow(V_new_e[2][r],2));
                if(vmag > e_max_vel)
                {
                        e_max_vel = vmag;
                }

                // total particle advance
                for(i=0; i<VECTORS; i++)
                {
                        pos_new_e[i][r] = calc_x(pos_old_e[i][r],V_new_e[i][r],DT);
                        pos_old_e[i][r] = pos_new_e[i][r];
                }

// function to check particle position and reflect if necessary
                if(x_diel_end || x_diel_beg || y_diel_end || y_diel_beg || z_diel_end ||
z_diel_beg)
                {
                        pos_new_e[0][r] =
check_reflect(pos_new_e[0][r],x_dist,0,r,V_new_e,x_diel_beg,x_diel_end);
                        pos_new_e[1][r] =
check_reflect(pos_new_e[1][r],y_dist,1,r,V_new_e,y_diel_beg,y_diel_end);
                        pos_new_e[2][r] =
check_reflect(pos_new_e[2][r],z_dist,2,r,V_new_e,z_diel_beg,z_diel_end);
                }

                // checking to flag particle if absorbed by wall
                if(pos_new_e[0][r] >= x_dist || pos_new_e[0][r] <= 0 || pos_new_e[1][r]
>= y_dist || pos_new_e[1][r] <= 0 || pos_new_e[2][r] >= z_dist || pos_new_e[2][r] <=
0)
                {
```

```
                        pos_new_e[3][r] = 1;
                        pos_old_e[3][r] = 1;
                        num_absorbed_e++;
                }
                else
                {
                        pos_new_e[3][r] = 0;
                        pos_old_e[3][r] = 0;
                }
        }
////////////////// End of leapfrog pushing technique for electrons ////////////////////

//////// update old arrays and new arrays and inject particles  ///////////////////////

        // record maximum velocity magnitudes
        outf14 << step << "\t" << Xe_max_vel << "\t" << e_max_vel << endl;

        // find and record average velocity magnitudes
        Xe_avg_vel = average_velocity(V_old_Xe,new_size_Xe);
        e_avg_vel = average_velocity(V_old_e,new_size_e);
        outf16 << step << "\t" << Xe_avg_vel << "\t" << e_avg_vel << endl;

        if(step<(time_steps-1))
        {
                old_size_e = new_size_e;
                old_size_Xe = new_size_Xe;
                new_size_e = old_size_e + flow_rate_e - num_absorbed_e;
                new_size_Xe = old_size_Xe + flow_rate_Xe - num_absorbed_Xe;

                // resize "old" arrays to "new_size"
                for(i=0;i<(VECTORS+1);i++)
                {
                        delete pos_old_e[i];
                        delete pos_old_Xe[i];
                }
                for(i=0;i<(VECTORS+1);i++)
                {
                        delete V_old_e[i];
                        delete V_old_Xe[i];
                }

                for(i=0;i<(VECTORS+1);i++)
                {
                        pos_old_e[i] = new float [new_size_e];
                        pos_old_Xe[i] = new float [new_size_Xe];
                }
                for(i=0;i<(VECTORS+1);i++)
                {
                        V_old_e[i] = new float [new_size_e];
                        V_old_Xe[i] = new float [new_size_Xe];
                }
                l=0;

// swap "new" arrays to old arrays, keeping only particles still in the domain
                for(r=0;r<old_size_e;r++)
                {
                        if(pos_new_e[3][r] == 0)
                        {
```

```
                        for(i=0;i<(VECTORS+1);i++)
                        {
                                V_old_e[i][l] = V_new_e[i][r];
                                pos_old_e[i][l] = pos_new_e[i][r];
                        }
                        pos_old_e[3][l] = 0;
                        V_old_e[3][l] = -1.0;
                        l++;
                }
        }
        l = 0;
        for(p=0;p<old_size_Xe;p++)
        {
                if(pos_new_Xe[3][p] == 0)
                {
                        for(i=0;i<(VECTORS+1);i++)
                        {
                                V_old_Xe[i][l] = V_new_Xe[i][p];
                                pos_old_Xe[i][l] = pos_new_Xe[i][p];
                        }
                        pos_old_Xe[3][l] = 0;
                        V_old_Xe[3][l] = -1.0;
                        l++;
                }
        }
        srand(rand());

        // initalize velocities of injected particles

    init_velocity_injected_v3(V_old_e,V_max_e,V_min_e,V_old_Xe,V_max_Xe,V_min_Xe,fl
ow_rate_e,flow_rate_Xe,new_size_e,new_size_Xe);

        // initalize positions of injected particles

    init_pos_injected_v5(pos_old_e,pos_old_Xe,x_node_pos,y_node_pos,z_node_pos,flow
_rate_e,flow_rate_Xe,new_size_e,new_size_Xe,x_start,y_start,z_start);

        // dynamically allocate space to "new" arrays and indices arrays
        for(i=0;i<(VECTORS+1);i++)
        {
                delete pos_new_e[i];
                delete pos_new_Xe[i];
                delete V_new_e[i];
                delete V_new_Xe[i];
        }
        for(i=0;i<VECTORS;i++)
        {
                delete indices_e[i];
                delete indices_Xe[i];
        }
        for(i=0;i<(VECTORS+1);i++)
        {
                pos_new_e[i] = new float [new_size_e];
                pos_new_Xe[i] = new float [new_size_Xe];
                V_new_e[i] = new float [new_size_e];
                V_new_Xe[i] = new float [new_size_Xe];
        }
        for(i=0;i<VECTORS;i++)
```

```
		{
			indices_e[i] = new int [new_size_e];
			indices_Xe[i] = new int [new_size_Xe];
		}
	}
} // End the time iterations
////////////////////////// end of time iterations //////////////////////

// Section to write all velocities to a file at the end of the simulation
outf17 << "Vx\tVy\tVz\t|V|" << endl;
outf18 << "Vx\tVy\tVz\t|V|" << endl;

for(r=0;r<new_size_e;r++)
{
	vmag = sqrt(pow(V_new_e[0][r],2)+pow(V_new_e[1][r],2)+pow(V_new_e[2][r],2));
	outf17 << V_new_e[0][r] << "\t" << V_new_e[1][r] << "\t" << V_new_e[2][r] <<
"\t" << vmag << endl;
}
for(r=0;r<new_size_Xe;r++)
{
	vmag = sqrt(pow(V_new_Xe[0][r],2)+pow(V_new_Xe[1][r],2)+pow(V_new_Xe[2][r],2));
	outf18 << V_new_Xe[0][r] << "\t" << V_new_Xe[1][r] << "\t" << V_new_Xe[2][r] <<
"\t" << vmag << endl;
}
time(&end);
outf3.close();
outf5.close();
outf6.close();
outf7.close();
outf8.close();
outf9.close();
outf10.close();
outf11.close();
outf13.close();
outf14.close();
outf16.close();
outf17.close();
outf18.close();
outf19.close();
outf20.close();
outf4 << "final number of electrons (computer particles, real particles): " <<
new_size_e << " , " << new_size_e*num_real <<  endl;
outf4 << "final number of ions (computer particles, real particles): " << new_size_Xe
<< " , " << new_size_Xe*num_real <<  endl;
outf4 << "final electron number density: " << new_size_e*num_real/comp_vol << " [m^-3]
and " << (float)new_size_e/num_cv << " computer particles per control volume" << endl;
outf4 << "final ion number density: " << new_size_Xe*num_real/comp_vol << " [m^-3] and
" << (float)new_size_Xe/num_cv << " computer particles per control volume" << endl;
outf4 << "number of time steps completed: " << time_steps << endl;
outf4 << "time to completion (seconds): " << difftime(end,start) << endl;
outf4.close();
cout << endl;
cout << "simulation complete" << endl;
cout << "\a \a \a" << endl; // 3 beeps to indicate completion
cout << "number of time steps: " << time_steps << endl;
cout << "total time elapsed (seconds): " << difftime(end,start) << endl;
system("pause");
return 0;
```

```
}
```

B.2. Fields.h

```
#ifndef FIELDS_H  // header guard to make sure fields.h has not already been included
#define FIELDS_H

#define VECTORS 3       // constant for defining the number of vectors in several
matrices
#define num_nodes_x 12  // number of nodes in the x direction
#define num_nodes_y 12  // number of nodes in the y direction
#define num_nodes_z 12  // number of nodes in the z direction
#define MAX_NODES 12    // maximum number of nodes in any direction
#define num_e 0   // number of electrons
#define num_Xe 0    // number of xenon ions

void grid_setup(float x_face_pos[],float y_face_pos[],float z_face_pos[],float
x_node_pos[],float y_node_pos[],float z_node_pos[],float x_diff[],float y_diff[],float
z_diff[],float A_x[][num_nodes_z],float A_y[][num_nodes_x],float
A_z[][num_nodes_y],float vol[num_nodes_x][num_nodes_y][num_nodes_z],float x_dist,float
y_dist,float z_dist)
{
       using namespace std;

       // calculate the spacing requred between each face (uniform spacing along a
given direction)
       float diff_face_x, diff_face_y, diff_face_z;
       diff_face_x = x_dist/(num_nodes_x-2);
       diff_face_y = y_dist/(num_nodes_y-2);
       diff_face_z = z_dist/(num_nodes_z-2);

       // initialize arrays for face positions so that the first position is at 0
       x_face_pos[0] = 0.0;
       y_face_pos[0] = 0.0;
       z_face_pos[0] = 0.0;

       // set the second face position to 0 as well
       x_face_pos[1] = 0.0;
       y_face_pos[1] = 0.0;
       z_face_pos[1] = 0.0;

       // calculate the remaining face locations in x
       int i, j, k, row=0;
       for(i = 2; i<num_nodes_x; i++)
       {
              x_face_pos[i] = x_face_pos[i-1]+diff_face_x;
       }

       // calculate the remaining face locations in y
       for(i = 2; i<num_nodes_y; i++)
       {
              y_face_pos[i] = y_face_pos[i-1]+diff_face_y;
```

```
	}

	// calculate the remaining face locations in z
	for(i = 2; i<num_nodes_z; i++)
	{
		z_face_pos[i] = z_face_pos[i-1]+diff_face_z;
	}

	// calculate the positions of each node in the x direction by averaging the two
faces it falls between
	// node[i] is the average position of face[i] and face[i+1]
	for( i= 0; i<num_nodes_x-1; i++)
	{
		x_node_pos[i] = (x_face_pos[i]+x_face_pos[i+1])/2.0;
	}
	x_node_pos[num_nodes_x-1] = x_face_pos[num_nodes_x-1];

	// calculate the positions of each node in the y direction by averaging the two
faces it falls between
	// node[i] is the average position of face[i] and face[i+1]
	for( i= 0; i<num_nodes_y-1; i++)
	{
		y_node_pos[i] = (y_face_pos[i]+y_face_pos[i+1])/2.0;
	}
	y_node_pos[num_nodes_y-1] = y_face_pos[num_nodes_y-1];

	// calculate the positions of each node in the z direction by averaging the two
faces it falls between
	// node[i] is the average position of face[i] and face[i+1]
	for( i= 0; i<num_nodes_z-1; i++)
	{
		z_node_pos[i] = (z_face_pos[i]+z_face_pos[i+1])/2.0;
	}
	z_node_pos[num_nodes_z-1] = z_face_pos[num_nodes_z-1];

	// calculate the distance between each CV node position in x (should be the
same for a uniform mesh except the ends )
	x_diff[0] = 0.0;
	for(i = 1; i<num_nodes_x; i++)
	{
		x_diff[i] = x_node_pos[i]-x_node_pos[i-1];
	}

	// calculate the distance between each CV node position in y (should be the
same for a uniform mesh except the ends )
	y_diff[0] = 0.0;
	for(i = 1; i<num_nodes_y; i++)
	{
		y_diff[i] = y_node_pos[i]-y_node_pos[i-1];
	}

	// calculate the distance between each CV node position in z (should be the
same for a uniform mesh except the ends )
	z_diff[0] = 0.0;
	for(i = 1; i<num_nodes_z; i++)
	{
		z_diff[i] = z_node_pos[i]-z_node_pos[i-1];
	}
```

```
	// calculate cross ectional areas in the x direction
	for(i = 0; i<num_nodes_y; i++)
	{
		for(j = 0; j<num_nodes_z; j++)
		{
			if(i == 0 || j == 0 || i == (num_nodes_y-1) || j == (num_nodes_z-
1)) // if y or z node falls on the edge of computational space, set cross sectional
area to 0
			{
				A_x[i][j] = 0.0;
			}
			else
			{
				A_x[i][j] = (y_face_pos[i+1]-
y_face_pos[i])*(z_face_pos[j+1]-z_face_pos[j]);
			}
		}
	}

	// calculate cross ectional areas in the y direction
	for(i = 0; i<num_nodes_z; i++)
	{
		for(j = 0; j<num_nodes_x; j++)
		{
			if(i == 0 || j == 0 || i == (num_nodes_z-1) || j == (num_nodes_x-
1)) // if z or x node falls on the edge of computational space, set cross sectional
area to 0
			{
				A_y[i][j] = 0.0;
			}
			else
			{
				A_y[i][j] = (z_face_pos[i+1]-
z_face_pos[i])*(x_face_pos[j+1]-x_face_pos[j]);
			}
		}
	}

	// calculate cross ectional areas in the z direction
	for(i = 0; i<num_nodes_x; i++)
	{
		for(j = 0; j<num_nodes_y; j++)
		{
			if(i == 0 || j == 0 || i == (num_nodes_x-1) || j == (num_nodes_y-
1)) // if y or z node falls on the edge of computational space, set cross sectional
area to 0
			{
				A_z[i][j] = 0.0;
			}
			else
			{
				A_z[i][j] = (x_face_pos[i+1]-
x_face_pos[i])*(y_face_pos[j+1]-y_face_pos[j]);
			}
		}
	}
```

```
	// calculate volume of each CV with respect to indices
	for(i=0; i<num_nodes_x; i++)
	{
		for(j=0; j<num_nodes_y; j++)
		{
			for(k=0; k<num_nodes_z; k++)
			{
				// first three columns are x,y,z node indices
				// if a node falls on the edge of computational space, set
volume equal to zero
				if(i == 0 || j == 0 || k == 0 || i == (num_nodes_x-1) || j
== (num_nodes_y-1) || k == num_nodes_z-1)
				{
					vol[i][j][k] = 0.0;
				}

				// if node falls within computational space, calculate
volume
				else
				{
					vol[i][j][k] = (x_face_pos[i+1]-
x_face_pos[i])*(y_face_pos[j+1]-y_face_pos[j])*(z_face_pos[k+1]-z_face_pos[k]);
				}
				row++;
			}
		}
	}
} // ends the function grid_setup

// function to solve for coefficients to be used in Poisson's equation
void poisson_solve_v2(float vol[num_nodes_x][num_nodes_y][num_nodes_z], float
volt[num_nodes_x][num_nodes_y][num_nodes_z], float x_diff[], float y_diff[], float
z_diff[], float A_x[][num_nodes_z], float A_y[][num_nodes_x], float
A_z[][num_nodes_y],float aE[num_nodes_x][num_nodes_y][num_nodes_z],float
aW[num_nodes_x][num_nodes_y][num_nodes_z],float
aN[num_nodes_x][num_nodes_y][num_nodes_z],float
aS[num_nodes_x][num_nodes_y][num_nodes_z],float
aT[num_nodes_x][num_nodes_y][num_nodes_z],float
aB[num_nodes_x][num_nodes_y][num_nodes_z],float
aP[num_nodes_x][num_nodes_y][num_nodes_z],float
b[num_nodes_x][num_nodes_y][num_nodes_z],float
roe[num_nodes_x][num_nodes_y][num_nodes_z],float ep_0,float x_pot_beg,float
x_pot_end,float y_pot_beg,float y_pot_end,float z_pot_beg,float z_pot_end,bool
x_diel_beg,bool x_diel_end,bool y_diel_beg,bool y_diel_end,bool z_diel_beg,bool
z_diel_end)
{
	int i,j,k;
	for(i=0;i<(num_nodes_x);i++)
	{
		for(j=0;j<(num_nodes_y);j++)
		{
			for(k=0;k<(num_nodes_z);k++)
			{
				if((i != 0) && (j != 0) && (k !=0) && (i!=(num_nodes_x-1))
&& (j!=(num_nodes_y-1)) && (k!=(num_nodes_z-1))) // interior node
				{
					aE[i][j][k] = A_x[j][k]/x_diff[i+1];
					aW[i][j][k] = A_x[j][k]/x_diff[i];
```

```
                                aN[i][j][k] = A_y[i][k]/y_diff[j+1];
                                aS[i][j][k] = A_y[i][k]/y_diff[j];
                                aT[i][j][k] = A_z[i][j]/z_diff[k+1];
                                aB[i][j][k] = A_z[i][j]/z_diff[k];
                                aP[i][j][k] =
aE[i][j][k]+aW[i][j][k]+aN[i][j][k]+aS[i][j][k]+aT[i][j][k]+aB[i][j][k];
                                b[i][j][k] = roe[i][j][k]*vol[i][j][k]/ep_0;
                        }
                        if(j==0) // south wall
                        {
                                aE[i][j][k] = 0;
                                aW[i][j][k] = 0;
                                aN[i][j][k] = 0;
                                aS[i][j][k] = 0;
                                aT[i][j][k] = 0;
                                aB[i][j][k] = 0;
                                aP[i][j][k] = 1;
                                b[i][j][k] = y_pot_beg;
                        }
                        if(j==num_nodes_y-1) // north wall
                        {
                                aE[i][j][k] = 0;
                                aW[i][j][k] = 0;
                                aN[i][j][k] = 0;
                                aS[i][j][k] = 0;
                                aT[i][j][k] = 0;
                                aB[i][j][k] = 0;
                                aP[i][j][k] = 1;
                                b[i][j][k] = y_pot_end;
                        }
                        if(k==0) // bottom wall
                        {
                                aE[i][j][k] = 0;
                                aW[i][j][k] = 0;
                                aN[i][j][k] = 0;
                                aS[i][j][k] = 0;
                                aT[i][j][k] = 0;
                                aB[i][j][k] = 0;
                                aP[i][j][k] = 1;
                                b[i][j][k] = z_pot_beg;
                        }if(k==num_nodes_z-1) // top wall
                        {
                                aE[i][j][k] = 0;
                                aW[i][j][k] = 0;
                                aN[i][j][k] = 0;
                                aS[i][j][k] = 0;
                                aT[i][j][k] = 0;
                                aB[i][j][k] = 0;
                                aP[i][j][k] = 1;
                                b[i][j][k] = z_pot_end;
                        }

                        if(i==0) // west wall
                        {
                                aE[i][j][k] = 0;
                                aW[i][j][k] = 0;
                                aN[i][j][k] = 0;
                                aS[i][j][k] = 0;
```

```
                        aT[i][j][k] = 0;
                        aB[i][j][k] = 0;
                        aP[i][j][k] = 1;
                        b[i][j][k] = x_pot_beg;
                }if(i==num_nodes_x-1) // east wall
                {
                        aE[i][j][k] = 0;
                        aW[i][j][k] = 0;
                        aN[i][j][k] = 0;
                        aS[i][j][k] = 0;
                        aT[i][j][k] = 0;
                        aB[i][j][k] = 0;
                        aP[i][j][k] = 1;
                        b[i][j][k] = x_pot_end;
                }
        }
    }
}// ends for loop of all positions

// dielectric boundary coefficients
// east and west dielectric boundaries
if(x_diel_end || x_diel_beg)
{
    for(j = 0; j<num_nodes_y ; j++)
    {
        for(k = 0; k<num_nodes_z ; k++)
        {
            if(x_diel_beg)
            {
                // west wall dielectric
                aE[0][j][k] = 1.0;
                b[0][j][k] = 0.0;
            }
            if(x_diel_end)
            {
                // east wall dielectric
                aW[num_nodes_x-1][j][k] = 1.0;
                b[num_nodes_x-1][j][k] = 0.0;
            }
        }
    }
}

// north and south dielectric boundaries
if(y_diel_beg || y_diel_end)
{
    for(i=0;i<num_nodes_x;i++)
    {
        for(k=0;k<num_nodes_z;k++)
        {
            if(y_diel_beg)
            {
                // south wall dielectric
                aN[i][0][k] = 1.0;
                b[i][0][k] = 0.0;
            }

            if(y_diel_end)
```

```
                        {
                                // north wall dielectric
                                aS[i][num_nodes_y-1][k] = 1.0;
                                b[i][num_nodes_y-1][k] = 0.0;
                        }
                }
        }
}

// top and bottom dielectric boundaries
if(z_diel_beg || z_diel_end)
{
        for(i=0;i<num_nodes_x;i++)
        {
                for(j=0;j<num_nodes_y;j++)
                {
                        // bottom wall dielectric
                        if(z_diel_beg)
                        {
                                aT[i][j][0] = 1.0;
                                b[i][j][0] = 0.0;
                        }

                        // top wall dielectric
                        if(z_diel_end)
                        {
                                aB[i][j][num_nodes_z-1] = 1.0;
                                b[i][j][num_nodes_z-1] = 0.0;
                        }
                }
        }
}

// section to properly assign coefficients to edge nodes (12 edges)
if(x_diel_beg && y_diel_beg) // 1
{
        for(k=1;k<(num_nodes_z-1);k++)
        {
                aE[0][0][k] = 0.5;
                aN[0][0][k] = 0.5;
                b[0][0][k] = 0.0;
        }
}
if(x_diel_beg && y_diel_end) // 2
{
        for(k=1;k<(num_nodes_z-1);k++)
        {
                aE[0][num_nodes_y-1][k] = 0.5;
                aS[0][num_nodes_y-1][k] = 0.5;
                b[0][num_nodes_y-1][k] = 0.0;
        }
}
if(x_diel_beg && z_diel_beg) // 3
{
        for(j=1;j<(num_nodes_y-1);j++)
        {
                aE[0][j][0] = 0.5;
                aT[0][j][0] = 0.5;
```

```
                b[0][j][0] = 0.0;
        }
}
if(x_diel_beg && z_diel_end) // 4
{
        for(j=1;j<(num_nodes_y-1);j++)
        {
                aE[0][j][num_nodes_z-1] = 0.5;
                aB[0][j][num_nodes_z-1] = 0.5;
                b[0][j][num_nodes_z-1] = 0.0;
        }
}
if(x_diel_end && y_diel_beg) // 5
{
        for(k=1;k<(num_nodes_z-1);k++)
        {
                aW[num_nodes_x-1][0][k] = 0.5;
                aN[num_nodes_x-1][0][k] = 0.5;
                b[num_nodes_x-1][0][k] = 0.0;
        }
}
if(x_diel_end && y_diel_end) // 6
{
        for(k=1;k<(num_nodes_z-1);k++)
        {
                aW[num_nodes_x-1][num_nodes_y-1][k] = 0.5;
                aS[num_nodes_x-1][num_nodes_y-1][k] = 0.5;
                b[num_nodes_x-1][num_nodes_y-1][k] = 0.0;
        }
}
if(x_diel_end && z_diel_beg) // 7
{
        for(j=1;j<(num_nodes_y-1);j++)
        {
                aW[num_nodes_x-1][j][0] = 0.5;
                aT[num_nodes_x-1][j][0] = 0.5;
                b[num_nodes_x-1][j][0] = 0.0;
        }
}
if(x_diel_end && z_diel_end) // 8
{
        for(j=1;j<(num_nodes_y-1);j++)
        {
                aW[num_nodes_x-1][j][num_nodes_z-1] = 0.5;
                aB[num_nodes_x-1][j][num_nodes_z-1] = 0.5;
                b[num_nodes_x-1][j][num_nodes_z-1] = 0.0;
        }
}
if(z_diel_end && y_diel_beg) // 9
{
        for(i=1;i<(num_nodes_x-1);i++)
        {
                aN[i][0][num_nodes_z-1] = 0.5;
                aB[i][0][num_nodes_z-1] = 0.5;
                b[i][0][num_nodes_z-1] = 0.0;
        }
}
if(z_diel_end && y_diel_end) // 10
```

```
{
	for(i=1;i<(num_nodes_x-1);i++)
	{
		aS[i][num_nodes_y-1][num_nodes_z-1] = 0.5;
		aB[i][num_nodes_y-1][num_nodes_z-1] = 0.5;
		b[i][num_nodes_y-1][num_nodes_z-1] = 0.0;
	}
}
if(z_diel_beg && y_diel_beg) // 11
{
	for(i=1;i<(num_nodes_x-1);i++)
	{
		aN[i][0][0] = 0.5;
		aT[i][0][0] = 0.5;
		b[i][0][0] = 0.0;
	}
}
if(z_diel_beg && y_diel_end) // 12
{
	for(i=1;i<(num_nodes_x-1);i++)
	{
		aS[i][num_nodes_y-1][0] = 0.5;
		aT[i][num_nodes_y-1][0] = 0.5;
		b[i][num_nodes_y-1][0] = 0.0;
	}
}

// section to properly assign coefficients to corner nodes (8 corners)
if(x_diel_beg && y_diel_beg && z_diel_beg) // 1
{
	aE[0][0][0] = 1.0/3.0;
	aN[0][0][0] = 1.0/3.0;
	aT[0][0][0] = 1.0/3.0;
	b[0][0][0] = 0.0;
}
if(x_diel_beg && y_diel_beg && z_diel_end) // 2
{
	aE[0][0][num_nodes_z-1] = 1.0/3.0;
	aN[0][0][num_nodes_z-1] = 1.0/3.0;
	aB[0][0][num_nodes_z-1] = 1.0/3.0;
	b[0][0][num_nodes_z-1] = 0.0;
}
if(x_diel_beg && y_diel_end && z_diel_beg) // 3
{
	aE[0][num_nodes_y-1][0] = 1.0/3.0;
	aS[0][num_nodes_y-1][0] = 1.0/3.0;
	aT[0][num_nodes_y-1][0] = 1.0/3.0;
	b[0][num_nodes_y-1][0] = 0.0;
}
if(x_diel_beg && y_diel_end && z_diel_end) // 4
{
	aE[0][num_nodes_y-1][num_nodes_z-1] = 1.0/3.0;
	aS[0][num_nodes_y-1][num_nodes_z-1] = 1.0/3.0;
	aB[0][num_nodes_y-1][num_nodes_z-1] = 1.0/3.0;
	b[0][num_nodes_y-1][num_nodes_z-1] = 0.0;
}
if(x_diel_end && y_diel_beg && z_diel_beg) // 5
{
```

```
		aW[num_nodes_x-1][0][0] = 1.0/3.0;
		aN[num_nodes_x-1][0][0] = 1.0/3.0;
		aT[num_nodes_x-1][0][0] = 1.0/3.0;
		b[num_nodes_x-1][0][0] = 0.0;
	}
	if(x_diel_end && y_diel_beg && z_diel_end) // 6
	{
		aW[num_nodes_x-1][0][num_nodes_z-1] = 1.0/3.0;
		aN[num_nodes_x-1][0][num_nodes_z-1] = 1.0/3.0;
		aB[num_nodes_x-1][0][num_nodes_z-1] = 1.0/3.0;
		b[num_nodes_x-1][0][num_nodes_z-1] = 0.0;
	}
	if(x_diel_end && y_diel_end && z_diel_beg) // 7
	{
		aW[num_nodes_x-1][num_nodes_y-1][0] = 1.0/3.0;
		aS[num_nodes_x-1][num_nodes_y-1][0] = 1.0/3.0;
		aT[num_nodes_x-1][num_nodes_y-1][0] = 1.0/3.0;
		b[num_nodes_x-1][num_nodes_y-1][0] = 0.0;
	}
	if(x_diel_end && y_diel_end && z_diel_end) // 8
	{
		aW[num_nodes_x-1][num_nodes_y-1][num_nodes_z-1] = 1.0/3.0;
		aS[num_nodes_x-1][num_nodes_y-1][num_nodes_z-1] = 1.0/3.0;
		aB[num_nodes_x-1][num_nodes_y-1][num_nodes_z-1] = 1.0/3.0;
		b[num_nodes_x-1][num_nodes_y-1][num_nodes_z-1] = 0.0;
	}
} // ends function to solve for coefficients

// function to update source terms
void source_solve(float vol[num_nodes_x][num_nodes_y][num_nodes_z],float
b[num_nodes_x][num_nodes_y][num_nodes_z],float
roe[num_nodes_x][num_nodes_y][num_nodes_z],float ep_0,float x_pot_beg,float
x_pot_end,float y_pot_beg,float y_pot_end,float z_pot_beg,float z_pot_end,bool
x_diel_beg,bool x_diel_end,bool y_diel_beg,bool y_diel_end,bool z_diel_beg,bool
z_diel_end)
{
	int i,j,k;
	for(i=0;i<(num_nodes_x);i++)
	{
		for(j=0;j<(num_nodes_y);j++)
		{
			for(k=0;k<(num_nodes_z);k++)
			{
				if((i != 0) && (j != 0) && (k !=0) && (i!=(num_nodes_x-1))
&& (j!=(num_nodes_y-1)) && (k!=(num_nodes_z-1))) // interior node
				{
					b[i][j][k] = roe[i][j][k]*vol[i][j][k]/ep_0;
				}
				if(j==0) // south wall
				{
					b[i][j][k] = y_pot_beg;
				}
				if(j==num_nodes_y-1) // north wall
				{
					b[i][j][k] = y_pot_end;
				}
				if(k==0) // bottom wall
				{
```

```
                        b[i][j][k] = z_pot_beg;
                    }
                    if(k==num_nodes_z-1) // top wall
                    {
                        b[i][j][k] = z_pot_end;
                    }
                    if(i==0) // west wall
                    {
                        b[i][j][k] = x_pot_beg;
                    }if(i==num_nodes_x-1) // east wall
                    {
                        b[i][j][k] = x_pot_end;
                    }
                }
        }
}// ends for loop of all positions

// dielectric boundary coefficients
// east and west dielectric boundaries
if(x_diel_beg || x_diel_end)
{
        for(j = 0; j<num_nodes_y ; j++)
        {
                for(k = 0; k<num_nodes_z ; k++)
                {
                        // west wall dielectric
                        if(x_diel_beg)
                        {
                                b[0][j][k] = 0.0;
                        }
                        // east wall dielectric
                        if(x_diel_end)
                        {
                                b[num_nodes_x-1][j][k] = 0.0;
                        }
                }
        }
}

// north and south dielectric boundaries
if(y_diel_beg || y_diel_end)
{
        for(i=0;i<num_nodes_x;i++)
        {
                for(k=0;k<num_nodes_z;k++)
                {
                        // south wall dielectric
                        if(y_diel_beg)
                        {
                                b[i][0][k] = 0.0;
                        }

                        // north wall dielectric
                        if(y_diel_end)
                        {
                                b[i][num_nodes_y-1][k] = 0.0;
                        }
                }
```

```
		}
	}

	// top and bottom dielectric boundaries
	if(z_diel_beg || z_diel_end)
	{
		for(i=0;i<num_nodes_x;i++)
		{
			for(j=0;j<num_nodes_y;j++)
			{
				// bottom wall dielectric
				if(z_diel_beg)
				{
					b[i][j][0] = 0.0;
				}

				// top wall dielectric
				if(z_diel_end)
				{
					b[i][j][num_nodes_z-1] = 0.0;
				}
			}
		}
	}
} // ends function to solve for the source term of poission's equation

// function to initalize potentials around a specified boundary. (x faces have
majority on edge nodes, while y faces have majority on remaining edges) volt matrix
boundary conditions
void volt_bc(float volt[num_nodes_x][num_nodes_y][num_nodes_z], float x_pot_beg, float
x_pot_end, float y_pot_beg, float y_pot_end, float z_pot_beg, float z_pot_end, float
pot_interior)
{
	int i,j,k;
	for(i=0;i<num_nodes_x;i++)
	{
		for(j=0;j<num_nodes_y;j++)
		{
			for(k=0;k<num_nodes_z;k++)
			{
				if(i==0)
				{
					volt[i][j][k] = x_pot_beg;
				}
				else if(i==(num_nodes_x-1))
				{
					volt[i][j][k] = x_pot_end;
				}
				else if(j==0 && (i != 0 || i != (num_nodes_x-1)))
				{
					volt[i][j][k] = y_pot_beg;
				}
				else if(j==(num_nodes_y-1) && (i != 0 || i !=
(num_nodes_x-1)))
				{
					volt[i][j][k] = y_pot_end;
				}
```

```
                        else if(k==0 && (i != 0 || i != (num_nodes_x-1)) && (j !=
0 || j != (num_nodes_y-1)))
                        {
                                volt[i][j][k] = z_pot_beg;
                        }
                        else if(k==(num_nodes_z-1) && (i != 0 || i !=
(num_nodes_x-1)) && (j != 0 || j != (num_nodes_y-1)))
                        {
                                volt[i][j][k] = z_pot_end;
                        }
                        else
                        {
                                volt[i][j][k] = pot_interior;
                        }
                }
        }
    }
} // ends function to initalize potentials along boundaries

// function to implement TDMA solver
void tdma(float a[],float b[],float c[],float d[],float temp[], int n)
{
    /*
    a - main diagonal (0 to n-1) [associated with i terms]
    b - sup diagonal (0 to n-2) [associated with i+1 terms]
    c - sub diagonal (1 to n-1) [associated with i-1 terms]
    d - right hand side, includes surrounding, perpendicular nodes and source term
(0 to n-1) [associated with i terms]
    volt - the answer (0 to n-1) [associated with i terms]
    n - number of equations to be solved i.e. the number of nodes in the desired
direction
    */
    int i;
    float p[MAX_NODES], q[MAX_NODES];
    p[0] = b[0]/a[0];
    q[0] = d[0]/a[0];

    // sweeping in some direction
    for(i=1;i<n;i++)
    {
        // find p and q to be used in this sweep of the TDMA
        p[i]=b[i]/(a[i]-c[i]*p[i-1]);
        q[i]=(d[i]+c[i]*q[i-1])/(a[i]-c[i]*p[i-1]);
    }

    for(i=(n-2);i>0;i--)
    {
        temp[i] = p[i]*temp[i+1]+q[i];
    }
}// ends tdma function

// function to initialize electric field array values to zero around the edges of the
computational space
void e_field_edges(float Ex[num_nodes_x][num_nodes_y][num_nodes_z],float
Ey[num_nodes_x][num_nodes_y][num_nodes_z],float
Ez[num_nodes_x][num_nodes_y][num_nodes_z])
{
    int i,j,k;
```

```
	for(j=0;j<num_nodes_y;j++)
	{
		for(k=0;k<num_nodes_z;k++)
		{
			Ex[0][j][k] = 0.0;
			Ex[num_nodes_x-1][j][k] = 0.0;
			Ey[0][j][k] = 0.0;
			Ey[num_nodes_x-1][j][k] = 0.0;
			Ez[0][j][k] = 0.0;
			Ez[num_nodes_x-1][j][k] = 0.0;

		}
	}
	for(i=0;i<num_nodes_x;i++)
	{
		for(k=0;k<num_nodes_z;k++)
		{
			Ex[i][0][k] = 0.0;
			Ex[i][num_nodes_y-1][k] = 0.0;
			Ey[i][0][k] = 0.0;
			Ey[i][num_nodes_y-1][k] = 0.0;
			Ez[i][0][k] = 0.0;
			Ez[i][num_nodes_y-1][k] = 0.0;
		}
	}

	for(i=0;i<num_nodes_x;i++)
	{
		for(j=0;j<num_nodes_y;j++)
		{
			Ex[i][j][0] = 0.0;
			Ex[i][j][num_nodes_z-1] = 0.0;
			Ey[i][j][0] = 0.0;
			Ey[i][j][num_nodes_z-1] = 0.0;
			Ez[i][j][0] = 0.0;
			Ez[i][j][num_nodes_z-1] = 0.0;
		}
	}
} // ends function to initalize electric field array values to zero around the edges
of the computational space


// function to calculate electric field for the computational grid
void e_field(float Ex[num_nodes_x][num_nodes_y][num_nodes_z],float
Ey[num_nodes_x][num_nodes_y][num_nodes_z],float
Ez[num_nodes_x][num_nodes_y][num_nodes_z],float
volt[num_nodes_x][num_nodes_y][num_nodes_z],float x_diff[],float y_diff[], float
z_diff[])
{
	int i,j,k; // initalize local counter variables

	for(i=1;i<(num_nodes_x-1);i++)
	{
		for(j=1;j<(num_nodes_y-1);j++)
		{
			for(k=1;k<(num_nodes_z-1);k++)
			{
```

```
                        Ex[i][j][k] = (((volt[i][j][k]-
volt[i+1][j][k])/(x_diff[i+1]))+((volt[i-1][j][k]-volt[i][j][k])/(x_diff[i])))/2.0;
                        Ey[i][j][k] = (((volt[i][j][k]-
volt[i][j+1][k])/(y_diff[j+1]))+((volt[i][j-1][k]-volt[i][j][k])/(y_diff[j])))/2.0;
                        Ez[i][j][k] = (((volt[i][j][k]-
volt[i][j][k+1])/(z_diff[k+1]))+((volt[i][j][k-1]-volt[i][j][k])/(z_diff[k])))/2.0;
                    }
                }
        }
} // ends function to calculate E field arrays for computational grid

// function to define static magnetic field
void b_fields(float Bx[num_nodes_x][num_nodes_y][num_nodes_z],float
By[num_nodes_x][num_nodes_y][num_nodes_z],float
Bz[num_nodes_x][num_nodes_y][num_nodes_z])
{
        int i,j,k;

        float uniform_field = 0.0;
        for(i=0;i<num_nodes_x;i++)
        {
                for(j=0;j<num_nodes_y;j++)
                {
                        for(k=0;k<num_nodes_z;k++)
                        {
                                Bx[i][j][k] = 0.0;
                                By[i][j][k] = 0.0;
                                Bz[i][j][k] = 0.0;
                        }
                }
        }
}// ends function to define static magnetic field

// function to algebraically determine the indices of each electron and ion
void indices_v2(float* pos_old_e[],float* pos_old_Xe[],int* indices_e[],int*
indices_Xe[], float x_dist, float y_dist, float z_dist,int new_size_e, int
new_size_Xe)
{
        int i,j,k,r,p;

        for(r=0;r<new_size_e;r++)
        {
                indices_e[0][r] = ceil((num_nodes_x-2)*(pos_old_e[0][r]/x_dist));
                indices_e[1][r] = ceil((num_nodes_y-2)*(pos_old_e[1][r]/y_dist));
                indices_e[2][r] = ceil((num_nodes_z-2)*(pos_old_e[2][r]/z_dist));
        }
        for(p=0;p<new_size_Xe;p++)
        {
                indices_Xe[0][p] = ceil((num_nodes_x-2)*(pos_old_Xe[0][p]/x_dist));
                indices_Xe[1][p] = ceil((num_nodes_y-2)*(pos_old_Xe[1][p]/y_dist));
                indices_Xe[2][p] = ceil((num_nodes_z-2)*(pos_old_Xe[2][p]/z_dist));
        }
}// ends function to algebraically determine the indices of each electron and Xenon
ion

// function to calculate charge density at each node using tri-linear interpolation
void calc_roe_v3(float Q_Xe,float roe[num_nodes_x][num_nodes_y][num_nodes_z],float
num_real,int new_size_e,int new_size_Xe,float
```

```
vol[num_nodes_x][num_nodes_y][num_nodes_z],float* pos_old_e[],float*
pos_old_Xe[],float dx, float dy,float dz,float x_node_pos[],float y_node_pos[],float
z_node_pos[])
{
	int i,j,k,r,t,x,y,z;
	float wx,wy,wz;
	float* charge[num_nodes_x][num_nodes_y];
	for(i=0;i<num_nodes_x;i++)
	{
		for(j=0;j<num_nodes_y;j++)
		{
			charge[i][j] = new float [num_nodes_z];
		}
	}
	for(i=0;i<num_nodes_x;i++)
	{
		for(j=0;j<num_nodes_y;j++)
		{
			for(k=0;k<num_nodes_z;k++)
			{
				charge[i][j][k] = 0.0;
			}
		}
	}
	for(r=0;r<new_size_e;r++)
	{
		// find the origin corner node for the current staggered grid
		i = (pos_old_e[0][r]+(dx/2.0))/dx;
		j = (pos_old_e[1][r]+(dy/2.0))/dy;
		k = (pos_old_e[2][r]+(dz/2.0))/dz;

		// find weighting factors
		wx = (pos_old_e[0][r]-x_node_pos[i])/(x_node_pos[i+1]-x_node_pos[i]);
		wy = (pos_old_e[1][r]-y_node_pos[j])/(y_node_pos[j+1]-y_node_pos[j]);
		wz = (pos_old_e[2][r]-z_node_pos[k])/(z_node_pos[k+1]-z_node_pos[k]);

		// Calculate the charge added to each nearest node (8 of them)
		charge[i][j][k] = charge[i][j][k] - Q_Xe*(1.0-wx)*(1.0-wy)*(1.0-
wz)*num_real;
		charge[i+1][j][k] = charge[i+1][j][k] - Q_Xe*(wx)*(1.0-wy)*(1.0-
wz)*num_real;
		charge[i][j+1][k] = charge[i][j+1][k] - Q_Xe*(1.0-wx)*(wy)*(1.0-
wz)*num_real;
		charge[i][j][k+1] = charge[i][j][k+1] - Q_Xe*(1.0-wx)*(1.0-
wy)*(wz)*num_real;
		charge[i+1][j+1][k] = charge[i+1][j+1][k] - Q_Xe*(wx)*(wy)*(1.0-
wz)*num_real;
		charge[i+1][j][k+1] = charge[i+1][j][k+1] - Q_Xe*(wx)*(1.0-
wy)*(wz)*num_real;
		charge[i][j+1][k+1] = charge[i][j+1][k+1] - Q_Xe*(1.0-
wx)*(wy)*(wz)*num_real;
		charge[i+1][j+1][k+1] = charge[i+1][j+1][k+1] -
Q_Xe*(wx)*(wy)*(wz)*num_real;
	}
	for(t=0;t<new_size_Xe;t++)
	{
		// find the origin corner node for the current staggered grid
		i = (pos_old_Xe[0][t]+(dx/2.0))/dx;
```

```
		j = (pos_old_Xe[1][t]+(dy/2.0))/dy;
		k = (pos_old_Xe[2][t]+(dz/2.0))/dz;

		// find weighting factors
		wx = (pos_old_Xe[0][t]-x_node_pos[i])/(x_node_pos[i+1]-x_node_pos[i]);
		wy = (pos_old_Xe[1][t]-y_node_pos[j])/(y_node_pos[j+1]-y_node_pos[j]);
		wz = (pos_old_Xe[2][t]-z_node_pos[k])/(z_node_pos[k+1]-z_node_pos[k]);

		// Calculate the charge added to each nearest node (8 of them)
		charge[i][j][k] = charge[i][j][k] + Q_Xe*(1.0-wx)*(1.0-wy)*(1.0-
wz)*num_real;
		charge[i+1][j][k] = charge[i+1][j][k] + Q_Xe*(wx)*(1.0-wy)*(1.0-
wz)*num_real;
		charge[i][j+1][k] = charge[i][j+1][k] + Q_Xe*(1.0-wx)*(wy)*(1.0-
wz)*num_real;
		charge[i][j][k+1] = charge[i][j][k+1] + Q_Xe*(1.0-wx)*(1.0-
wy)*(wz)*num_real;
		charge[i+1][j+1][k] = charge[i+1][j+1][k] + Q_Xe*(wx)*(wy)*(1.0-
wz)*num_real;
		charge[i+1][j][k+1] = charge[i+1][j][k+1] + Q_Xe*(wx)*(1.0-
wy)*(wz)*num_real;
		charge[i][j+1][k+1] = charge[i][j+1][k+1] + Q_Xe*(1.0-
wx)*(wy)*(wz)*num_real;
		charge[i+1][j+1][k+1] = charge[i+1][j+1][k+1] +
Q_Xe*(wx)*(wy)*(wz)*num_real;
	}
	for(i=1;i<(num_nodes_x-1);i++)
	{
		for(j=1;j<(num_nodes_y-1);j++)
		{
			for(k=1;k<(num_nodes_z-1);k++)
			{
				roe[i][j][k] = charge[i][j][k]/vol[i][j][k];
			}
		}
	}
} // ends function to calculate charge density (roe) at each node using tri-linear
interpolation (charge partially allocated to the wall)

// function to calculate charge density at each node using tri-linear interpolation
during the electron push multiple solves technique (charge is partially allocated to
the wall)
void calc_roe_mid_push_v2(float Q_Xe,float
roe[num_nodes_x][num_nodes_y][num_nodes_z],float num_real,int new_size_e,int
new_size_Xe,float vol[num_nodes_x][num_nodes_y][num_nodes_z],float* pos_old_e[],float*
pos_old_Xe[],float dx, float dy,float dz,float x_node_pos[],float y_node_pos[],float
z_node_pos[])
{
	int i,j,k,r,t,x,y,z;
	float wx,wy,wz;
	using namespace std;
	float* charge[num_nodes_x][num_nodes_y];
	for(i=0;i<num_nodes_x;i++)
	{
		for(j=0;j<num_nodes_y;j++)
		{
			charge[i][j] = new float [num_nodes_z];
		}
```

```
	}

	for(i=0;i<num_nodes_x;i++)
	{
		for(j=0;j<num_nodes_y;j++)
		{
			for(k=0;k<num_nodes_z;k++)
			{
				charge[i][j][k] = 0.0;
			}
		}
	}
	for(r=0;r<new_size_e;r++)
	{
		if(pos_old_e[0][r] < x_node_pos[num_nodes_x-1] && pos_old_e[0][r] > 0 &&
pos_old_e[1][r] < y_node_pos[num_nodes_y-1] && pos_old_e[1][r] > 0 && pos_old_e[2][r]
< z_node_pos[num_nodes_z-1] && pos_old_e[2][r] > 0)
		{
			// find the origin corner node for the current staggered grid
			i = (pos_old_e[0][r]+(dx/2.0))/dx;
			j = (pos_old_e[1][r]+(dy/2.0))/dy;
			k = (pos_old_e[2][r]+(dz/2.0))/dz;

			// find weighting factors
			wx = (pos_old_e[0][r]-x_node_pos[i])/(x_node_pos[i+1]-
x_node_pos[i]);
			wy = (pos_old_e[1][r]-y_node_pos[j])/(y_node_pos[j+1]-
y_node_pos[j]);
			wz = (pos_old_e[2][r]-z_node_pos[k])/(z_node_pos[k+1]-
z_node_pos[k]);

			// Calculate the charge added to each nearest node (8 of them)
			charge[i][j][k] = charge[i][j][k] - Q_Xe*(1.0-wx)*(1.0-wy)*(1.0-
wz)*num_real;
			charge[i+1][j][k] = charge[i+1][j][k] - Q_Xe*(wx)*(1.0-wy)*(1.0-
wz)*num_real;
			charge[i][j+1][k] = charge[i][j+1][k] - Q_Xe*(1.0-wx)*(wy)*(1.0-
wz)*num_real;
			charge[i][j][k+1] = charge[i][j][k+1] - Q_Xe*(1.0-wx)*(1.0-
wy)*(wz)*num_real;
			charge[i+1][j+1][k] = charge[i+1][j+1][k] - Q_Xe*(wx)*(wy)*(1.0-
wz)*num_real;
			charge[i+1][j][k+1] = charge[i+1][j][k+1] - Q_Xe*(wx)*(1.0-
wy)*(wz)*num_real;
			charge[i][j+1][k+1] = charge[i][j+1][k+1] - Q_Xe*(1.0-
wx)*(wy)*(wz)*num_real;
			charge[i+1][j+1][k+1] = charge[i+1][j+1][k+1] -
Q_Xe*(wx)*(wy)*(wz)*num_real;
		}
	}
	for(t=0;t<new_size_Xe;t++)
	{
		if(pos_old_Xe[0][t] < x_node_pos[num_nodes_x-1] && pos_old_Xe[0][t] > 0
&& pos_old_Xe[1][t] < y_node_pos[num_nodes_y-1] && pos_old_Xe[1][t] > 0 &&
pos_old_Xe[2][t] < z_node_pos[num_nodes_z-1] && pos_old_Xe[2][t] > 0)
		{
			// find the origin corner node for the current staggered grid
			i = (pos_old_Xe[0][t]+(dx/2.0))/dx;
```

```
                    j = (pos_old_Xe[1][t]+(dy/2.0))/dy;
                    k = (pos_old_Xe[2][t]+(dz/2.0))/dz;

                    // find weighting factors
                    wx = (pos_old_Xe[0][t]-x_node_pos[i])/(x_node_pos[i+1]-
x_node_pos[i]);
                    wy = (pos_old_Xe[1][t]-y_node_pos[j])/(y_node_pos[j+1]-
y_node_pos[j]);
                    wz = (pos_old_Xe[2][t]-z_node_pos[k])/(z_node_pos[k+1]-
z_node_pos[k]);

                    // Calculate the charge added to each nearest node (8 of them)
                    charge[i][j][k] = charge[i][j][k] + Q_Xe*(1.0-wx)*(1.0-wy)*(1.0-
wz)*num_real;
                    charge[i+1][j][k] = charge[i+1][j][k] + Q_Xe*(wx)*(1.0-wy)*(1.0-
wz)*num_real;
                    charge[i][j+1][k] = charge[i][j+1][k] + Q_Xe*(1.0-wx)*(wy)*(1.0-
wz)*num_real;
                    charge[i][j][k+1] = charge[i][j][k+1] + Q_Xe*(1.0-wx)*(1.0-
wy)*(wz)*num_real;
                    charge[i+1][j+1][k] = charge[i+1][j+1][k] + Q_Xe*(wx)*(wy)*(1.0-
wz)*num_real;
                    charge[i+1][j][k+1] = charge[i+1][j][k+1] + Q_Xe*(wx)*(1.0-
wy)*(wz)*num_real;
                    charge[i][j+1][k+1] = charge[i][j+1][k+1] + Q_Xe*(1.0-
wx)*(wy)*(wz)*num_real;
                    charge[i+1][j+1][k+1] = charge[i+1][j+1][k+1] +
Q_Xe*(wx)*(wy)*(wz)*num_real;
                }
        }
        for(i=1;i<(num_nodes_x-1);i++)
        {
                for(j=1;j<(num_nodes_y-1);j++)
                {
                        for(k=1;k<(num_nodes_z-1);k++)
                        {
                                roe[i][j][k] = charge[i][j][k]/vol[i][j][k];
                        }
                }
        }
        for(i=0;i<num_nodes_x;i++)
        {
                for(j=0;j<num_nodes_y;j++)
                {
                        delete charge[i][j];
                }
        }
} // ends function to calculate charge density (roe) at each node using tri-linear
interpolation (charge partially allocated to the wall)

// function to determine the fourth order electric field
float order_4_E_v4(float vel, float DT, int vector, int ind[], float
Evector[num_nodes_x][num_nodes_y][num_nodes_z], float dx)
{
        int I,N,J;
        const int num = 5; // number of coefficients needed for the 4th order
calculation
        long double E[num];
```

```
long double E_field = 0.0;
long double dvel = (long double) vel;
J = 0;
long double a0[num];

for(I=0;I<num;I++)
{
	a0[I] = 0.0;
}
if(vel>0) // velocity is in the positive direction
{
	N = ind[vector]+num;
	if(vector == 0) // velocity in the positive x direction
	{
		for(I=0;I<num;I++)
		{
			if((ind[0]+I)<(num_nodes_x-1))
			{
				E[I] = Evector[ind[0]+I][ind[1]][ind[2]];
			}
			else
			{
				J++;
				E[I] = -Evector[num_nodes_x-1-J][ind[1]][ind[2]];
			}
		}
	}
	else if(vector == 1) // velocity in the positive y direction
	{
		for(I=0;I<num;I++)
		{
			if((ind[1]+I)<(num_nodes_y-1))
			{
				E[I] = Evector[ind[0]][ind[1]+I][ind[2]];
			}
			else
			{
				J++;
				E[I] = -Evector[ind[0]][num_nodes_y-1-J][ind[2]];
			}
		}

	}
	else // vector == 2 so velocity in the positive z direction
	{
		for(I=0;I<num;I++)
		{
			if((ind[2]+I)<(num_nodes_z-1))
			{
				E[I] = Evector[ind[0]][ind[1]][ind[2]+I];
			}
			else
			{
				J++;
				E[I] = -Evector[ind[0]][ind[1]][num_nodes_z-1-J];
			}
		}
	}
```

```
}
else // velocity in the negative direction
{
	N = ind[vector]-4;
	if(vector == 0) // velocity in the negative x direction
	{
		for(I=0;I<num;I++)
		{
			if((ind[0]-I)>0)
			{
				E[I] = Evector[ind[0]-I][ind[1]][ind[2]];
			}
			else
			{
				J++;
				E[I] = -Evector[J][ind[1]][ind[2]];
			}
		}
	}

	else if(vector == 1) // velocity in the positive y direction
	{
		for(I=0;I<num;I++)
		{
			if((ind[1]-I)>0)
			{
				E[I] = Evector[ind[0]][ind[1]-I][ind[2]];
			}
			else
			{
				J++;
				E[I] = -Evector[ind[0]][J][ind[2]];
			}
		}

	}
	else // vector == 2 so velocity in the positive z direction
	{
		for(I=0;I<num;I++)
		{
			if((ind[2]-I)>0)
			{
				E[I] = Evector[ind[0]][ind[1]][ind[2]-I];
			}
			else
			{
				J++;
				E[I] = -Evector[ind[0]][ind[1]][J];
			}
		}
	}
}

a0[0] = E[0];
a0[1] = -1*((25*E[0]-48*E[1]+36*E[2]-16*E[3]+3*E[4])/(12*dx));
a0[2] = (35*E[0]-104*E[1]+114*E[2]-56*E[3]+11*E[4])/(24*dx*dx);
a0[3] = -1*((5*E[0]-18*E[1]+24*E[2]-14*E[3]+3*E[4])/(12*dx*dx*dx));
```

```
	a0[4] = (E[0]-4*E[1]+6*E[2]-4*E[3]+E[4])/(24*dx*dx*dx*dx);

	float X = abs(dvel*DT);
	E_field = a0[0] + (a0[1]*X/4) + (a0[2]*pow(X,2)/12) + (a0[3]*pow(X,3)/32) +
(a0[4]*pow(X,4)/80);
	return(E_field);

}// ends function to calculate the E field using a 4th order method

#endif
```

B.3. Push.h

```
#ifndef PUSH_H  // header guard to make sure push.h has not already been included
#define PUSH_H

#include <time.h>
#define VECTORS 3       // constant for defining the number of vectors in several
matrices
#define num_nodes_x 12  // number of nodes in the x direction
#define num_nodes_y 12  // number of nodes in the y direction
#define num_nodes_z 12  // number of nodes in the z direction
#define MAX_NODES 12    // maximum number of nodes in any direction (used for array
initalization)
#define num_e 0   // number of electrons
#define num_Xe 0    // number of xenon ions

// function to initalize velocity using 1 random number per component and set the
initial marker to 1
void init_velocity_v3(float* V_old_e[],int V_max_e,int V_min_e,float* V_old_Xe[],int
V_max_Xe,int V_min_Xe)
{
	int i, j;
	float range_e, range_Xe;
	float mag;
	float pi = 3.14159;
	range_e = V_max_e-V_min_e;
	range_Xe = V_max_Xe-V_min_Xe;
	float max = RAND_MAX;
	float num,num1,num2,num3;
	float Vx,Vy,Vz,f;

	// initalize random velocities for electrons
	for(i=0;i<num_e;i++)
	{
		num = rand();
		num1 = rand();
		num2 = rand();
		num3 = rand();
		mag = abs(V_min_e) + abs((num/max)*range_e) ;
		if((int)num2%2 ==0)
		{
			Vx = (num1/max)*mag/sqrt(3.0);
		}
```

```
        else
        {
                Vx = -(num/max)*mag/sqrt(3.0);
        }
        if((int)num3%2 == 0)
        {
                Vy = (num2/max)*mag/sqrt(3.0);
        }
        else
        {
                Vy = -(num2/max)*mag/sqrt(3.0);
        }
        if((int)num1%2 == 0)
        {
                Vz = (num3/max)*mag/sqrt(3.0);
        }
        else
        {
                Vz = -(num3/max)*mag/sqrt(3.0);
        }
        f = mag/(sqrt(pow(Vx,2)+pow(Vy,2)+pow(Vz,2)));
        V_old_e[0][i] = f*Vx;
        V_old_e[1][i] = f*Vy;
        V_old_e[2][i] = f*Vz;
        V_old_e[3][i] = 1.0;
}

// initalize random velocities for Xenon ions
for(j=0;j<num_Xe;j++)
{
        num = rand();
        num1 = rand();
        num2 = rand();
        num3 = rand();
        mag = abs(V_min_Xe) + abs((num/max)*range_Xe) ;
        if((int)num2%2 ==0)
        {
                Vx = (num1/max)*mag/sqrt(3.0);
        }
        else
        {
                Vx = -(num/max)*mag/sqrt(3.0);
        }
        if((int)num3%2 == 0)
        {
                Vy = (num2/max)*mag/sqrt(3.0);
        }
        else
        {
                Vy = -(num2/max)*mag/sqrt(3.0);
        }
        if((int)num1%2 == 0)
        {
                Vz = (num3/max)*mag/sqrt(3.0);
        }
        else
        {
                Vz = -(num3/max)*mag/sqrt(3.0);
```

```
                }
                f = mag/(sqrt(pow(Vx,2)+pow(Vy,2)+pow(Vz,2)));
                V_old_Xe[0][j] = f*Vx;
                V_old_Xe[1][j] = f*Vy;
                V_old_Xe[2][j] = f*Vz;
                V_old_Xe[3][j] = 1.0;
        }
} // ends the initalize velocity function

// function to initalize velocity for injected particles using 1 rand() per component
and estabilishing a first push marker
void init_velocity_injected_v3(float* V_old_e[],int V_max_e,int V_min_e,float*
V_old_Xe[],int V_max_Xe,int V_min_Xe,int flow_rate_e,int flow_rate_Xe,int
new_size_e,int new_size_Xe)
{
        int i, j;
        float range_e, range_Xe;
        float mag;
        float pi = 3.14159;
        range_e = V_max_e-V_min_e;
        range_Xe = V_max_Xe-V_min_Xe;
        float max = RAND_MAX;
        float num,num1,num2,num3;
        float Vx,Vy,Vz,f;

        // initalize random velocities for electrons
        for(i=0;i<flow_rate_e;i++)
        {
                num = rand();
                num1 = rand();
                num2 = rand();
                num3 = rand();
                mag = abs(V_min_e) + abs((num/max)*range_e) ;
                if((int)num2%2 ==0)
                {
                        Vx = (num1/max)*mag/sqrt(3.0);
                }
                else
                {
                        Vx = -(num/max)*mag/sqrt(3.0);
                }
                if((int)num3%2 == 0)
                {
                        Vy = (num2/max)*mag/sqrt(3.0);
                }
                else
                {
                        Vy = -(num2/max)*mag/sqrt(3.0);
                }
                if((int)num1%2 == 0)
                {
                        Vz = (num3/max)*mag/sqrt(3.0);
                }
                else
                {
                        Vz = -(num3/max)*mag/sqrt(3.0);
                }
                f = mag/(sqrt(pow(Vx,2)+pow(Vy,2)+pow(Vz,2)));
```

```
			V_old_e[0][new_size_e-flow_rate_e+i] = f*Vx;
			V_old_e[1][new_size_e-flow_rate_e+i] = f*Vy;
			V_old_e[2][new_size_e-flow_rate_e+i] = f*Vz;
			V_old_e[3][new_size_e-flow_rate_e+i] = 1.0;
		}

		// initalize random velocities for ions
		for(j=0;j<flow_rate_Xe;j++)
		{
			num = rand();
			num1 = rand();
			num2 = rand();
			num3 = rand();
			mag = abs(V_min_Xe) + abs((num/max)*range_Xe) ;
			if((int)num2%2 ==0)
			{
				Vx = (num1/max)*mag/sqrt(3.0);
			}
			else
			{
				Vx = -(num/max)*mag/sqrt(3.0);
			}
			if((int)num3%2 == 0)
			{
				Vy = (num2/max)*mag/sqrt(3.0);
			}
			else
			{
				Vy = -(num2/max)*mag/sqrt(3.0);
			}
			if((int)num1%2 == 0)
			{
				Vz = (num3/max)*mag/sqrt(3.0);
			}
			else
			{
				Vz = -(num3/max)*mag/sqrt(3.0);
			}
			f = mag/(sqrt(pow(Vx,2)+pow(Vy,2)+pow(Vz,2)));
			V_old_Xe[0][new_size_Xe-flow_rate_Xe+j] = f*Vx;
			V_old_Xe[1][new_size_Xe-flow_rate_Xe+j] = f*Vy;
			V_old_Xe[2][new_size_Xe-flow_rate_Xe+j] = f*Vz;
			V_old_Xe[3][new_size_Xe-flow_rate_Xe+j] = 1.0;
		}
} // ends the initalize velocity for injected particles function

// function to initalize starting positions assigning positive and negative particle
to the same starting positions and allowing for an amount of space between injection
and the domain boundary
void init_pos_v7(float* pos_old_e[],float* pos_old_Xe[],float x_node_pos[], float
y_node_pos[], float z_node_pos[],float x_start,float y_start,float z_start)
{
		int i,j;
		float x_dist = x_node_pos[num_nodes_x-1]-x_node_pos[0]-2.0*x_start;
		float y_dist = y_node_pos[num_nodes_y-1]-y_node_pos[0]-2.0*y_start;
		float z_dist = z_node_pos[num_nodes_z-1]-z_node_pos[0]-2.0*z_start;
		float max = RAND_MAX;
		float num,num2,num3;
```

```
	for(i=0;i<num_e;i++)
	{
		num = rand();
		num2 = rand();
		num3 = rand();
		pos_old_e[0][i] = (num/max)*x_dist+x_start;
		pos_old_e[1][i] = (num2/max)*y_dist+y_start;
		pos_old_e[2][i] = (num3/max)*z_dist+z_start;
		pos_old_e[3][i] = 0;
		pos_old_Xe[0][i] = pos_old_e[0][i];
		pos_old_Xe[1][i] = pos_old_e[1][i];
		pos_old_Xe[2][i] = pos_old_e[2][i];
		pos_old_Xe[3][i] = 0;
	}

}// ends function to initalize starting positions assigning positive and negative
particles to the same starting positions

// function to randomly initalize starting positions of injected particles by
assigning positive and negative particles to the same location
void init_pos_injected_v5(float* pos_old_e[],float* pos_old_Xe[],float x_node_pos[],
float y_node_pos[], float z_node_pos[],int flow_rate_e,int flow_rate_Xe,int
new_size_e,int new_size_Xe,float x_start,float y_start,float z_start)
{
	int i,j;
	float x_dist = x_node_pos[num_nodes_x-1]-x_node_pos[0]-2.0*x_start;
	float y_dist = y_node_pos[num_nodes_y-1]-y_node_pos[0]-2.0*y_start;
	float z_dist = z_node_pos[num_nodes_z-1]-z_node_pos[0]-2.0*z_start;
	float max = RAND_MAX;
	float num,num2,num3;

	for(i=0;i<flow_rate_e;i++)
	{
		num = rand();
		num2 = rand();
		num3 = rand();
		pos_old_e[0][new_size_e-flow_rate_e+i] = (num/max)*x_dist+x_start;
		pos_old_e[1][new_size_e-flow_rate_e+i] = (num2/max)*y_dist+y_start;
		pos_old_e[2][new_size_e-flow_rate_e+i] = (num3/max)*z_dist+z_start;
		pos_old_e[3][new_size_e-flow_rate_e+i] = 0;
		pos_old_Xe[0][new_size_Xe-flow_rate_Xe+i] = pos_old_e[0][new_size_e-
flow_rate_e+i];
		pos_old_Xe[1][new_size_Xe-flow_rate_Xe+i] = pos_old_e[1][new_size_e-
flow_rate_e+i];
		pos_old_Xe[2][new_size_Xe-flow_rate_Xe+i] = pos_old_e[2][new_size_e-
flow_rate_e+i];
		pos_old_Xe[3][new_size_Xe-flow_rate_Xe+i] = 0;
	}
}// ends function to randomly initalize starting positions of injected particle by
assigning positive and negative particle to the same location

// function to take a cross product of two vectors (a X b = ans)
void cross(float a[3], float b[3], float ans[3])
{
	ans[0] = (a[1]*b[2])-(a[2]*b[1]);
	ans[1] = (a[2]*b[0])-(a[0]*b[2]);
	ans[2] = (a[0]*b[1])-(a[1]*b[0]);
```

```
}

// function to calculate t (equation 3.24 of Sudhakar's Dissertation)
float calc_t(float Q_E, float B, float DT, float M_E)
{
	float t;
	t = (Q_E*B*DT)/(2.0*M_E);
	return(t);
}

// function to calculate s (equation 3.25 of Sudhakar's Dissertation)
float calc_s(float t)
{
    float s;
	s = (2.0*t)/(1.0+t*t);
	return(s);
}

// function to calculate V minus (equation 3.19 of Sudhakar's Dissertation)
float calc_v_minus(float Q_E, float E, float DT, float M_E, float v_minus_half)
{
	float v_minus;
	v_minus = v_minus_half + (Q_E*E*DT)/(2.0*M_E);
	return(v_minus);
}

// function to calculate V prime (equation 3.22 of Sudhakar's Dissertation)
void calc_v_prime(float v_minus[], float t[], float v_prime[])
{
	float ans[VECTORS];
	cross(v_minus,t,ans);

	for(int i=0; i<VECTORS; i++)
	{
		v_prime[i] = v_minus[i]+ans[i];
	}
}

// function to calculate V plus (equation 3.23 of Sudhakar's Dissertation)
void calc_v_plus(float v_minus[], float v_prime[], float s[], float v_plus[])
{
	float ans[VECTORS];
	cross(v_prime,s,ans);

	for(int i = 0; i<VECTORS; i++)
	{
		v_plus[i] = v_minus[i]+ans[i];
	}
}

// function to calculate velocity at half a time step in the future (equation 3.20 of
Sudhakar's Dissertation)
float calc_v(float v_plus, float Q_E, float E, float DT, float M_E)
{
    float v;
	v = v_plus + (Q_E*E*DT)/(2.0*M_E);
	return(v);
}
```

```
// function to calculate final particle position (equation 3.26 of Sudhakar's
Dissertation)
float calc_x(float x_previous,float vx_plus_half,float DT)
{
	float x;
	x = x_previous + vx_plus_half*DT;
	return(x);
}

// function to check particle position and reflect if necessary
float check_reflect(float pos, float dist,int vector,int particle,float* V_new[],bool
beg_ref,bool end_ref)
{
	if(beg_ref)
	{
	    if(pos<0.0)
	    {
		V_new[vector][particle] = -V_new[vector][particle];
		return(abs(pos));
	    }
	}

	if(end_ref)
	{
	    if(pos>dist)
	    {
		V_new[vector][particle] = -V_new[vector][particle];
		return(dist-(pos-dist));
	    }
	}
	return(pos);
} //ends function to check particle position and reflect if necessary

// function to calculate the average particle velocity of a single species
float average_velocity(float* Vel[], int n)
{
	int i,j;
	float avg,sum,mag;
	avg = 0.0;
	sum = 0.0;
	for(i=0;i<n;i++)
	{
		mag = sqrt(pow(Vel[0][i],2)+pow(Vel[1][i],2)+pow(Vel[2][i],2));
		sum = sum+mag;
	}
	avg = sum/n;
	return(avg);
}// ends function to calculate average paritcle velocity of a single species

#endif
```