

AN ABSTRACT OF THE THESIS OF

Conner Myers for the degree of Master of Science in Nuclear Engineering presented on June 3, 2020

Title: The Development of a Hybrid Particle-in-Cell Simulation Code in C++ for the Modeling of the SPT-70 Hall Thruster.

Abstract approved:

Brian G. Woods

In order to support the growing satellite propulsion industry, a hybrid Particle-In-Cell (PIC) simulation with fluid treatment of electrons and neutral particles was developed to model the SPT-70 Hall thruster. The 2D, C++ code assumes azimuthal symmetry and a uniform magnetic field in the radial direction with the simulation domain limited to the thruster cavity. Ions are treated as discrete superparticles while electrons are assumed to distribute instantaneously along magnetic field lines according to the Boltzmann relationship and diffuse across magnetic field lines through classical electron diffusion. The results from simulations with various operational parameters of the SPT-70 are compared to experimental data. Simulated thrust underestimates experimental thrust but approximately models electron, ion, and neutral particle transport within the thruster cavity.

©Copyright by Conner Myers
June 3, 2020
Creative Commons License

The Development of a Hybrid Particle-in-Cell Simulation Code in C++ for the
Modeling of the SPT-70 Hall Thruster

by
Conner Myers

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented June 3, 2020
Commencement June 2020

Master of Science thesis of Conner Myers presented on June 3, 2020

APPROVED:

Major Professor, representing Nuclear Engineering

Head of the School of Nuclear Science and Engineering

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Conner Myers, Author

ACKNOWLEDGEMENTS

I would like to express my sincere appreciation to my advisor, Dr. Brian Woods, who skillfully guided me through my research while still allowing my work to be my own. He introduced me to the field of plasma physics and provided an opportunity to pursue my educational goals that I would not have found anywhere else.

I would also like to thank my committee members, Dr. Andrew Klein, Dr. Wade Marcum, and Dr. Donghua Xu for making this possible by being responsive and accommodating, even in the midst of a global pandemic.

Lastly, I would like to thank the Oregon State University foundation for generously supporting my academic and research endeavors.

TABLE OF CONTENTS

	<u>Page</u>
1 Introduction.....	1
1.1 Background	1
1.2 Purpose	5
1.3 Importance	6
1.4 Assumptions	10
1.5 Limitations	11
2 Literature Review.....	13
2.1 Hall Thrusters	13
2.2 Development of Plasma Simulations	14
2.3 Particle-In-Cell Simulations	15
2.3.1 Superparticles	15
2.3.2 Modeling Particle Collisions	16
2.3.3 Accuracy and Stability Conditions	16
2.4 Other Plasma Models	17
2.5 Hybrid Particle-in-Cell Simulations	18
2.5.1 Hybrid-PIC Hall Thruster Simulations	20
3 Methods	23
3.1 The Challenge of Kinetic Simulations	23
3.2 Particle-in-Cell Simulation Structure	25
3.2.1 Simulation Overview	25
3.2.2 Initialization	26

TABLE OF CONTENTS (Continued)

	<u>Page</u>
3.2.3 Loop Functions	28
3.2.4 Simulation Output	35
3.2.4 Error Analysis	36
4 Results	39
4.1 System Setup	39
4.2 Simulation Results	40
4.2.1 Simulation Results with Standard Operation Parameters	41
4.2.2 Simulation Results with Varied Operation Parameters	43
4.3 Effects of Superparticle Size on Simulation Accuracy	53
5 Conclusion	55
5.1 Summary	55
5.2 Future Work	57
Bibliography	59
Appendix	62

LIST OF FIGURES

<u>Figure</u>	<u>Page</u>
1.1 Cross Sectional View of a Hall Thruster.....	2
1.2 ExB Drift for Ions and Electrons.....	3
3.1 Algorithm flowchart for the hybrid-PIC simulation.....	25
3.2 A sketch of the contribution of the ion's charge to each grid point in 1D.....	29
3.3 A sketch of the contribution of the ion's charge to each grid point in 2D.....	29
4.1 Thrust as a function of operating voltage and neutral mass injection rate for experimental and simulation results.....	43
4.2 Specific Impulse as a function of operating voltage and neutral mass injection rate for experimental and simulation results.....	44
4.3 Simulated centerline electric potential as a function of axial distance from the cathode for an operating voltage of 300 V.....	45
4.4 Simulated centerline electric potential as a function of axial distance from the cathode for an operating voltage of 250 V.....	45
4.5 Simulated centerline electric potential as a function of axial distance from the cathode for an operating voltage of 200 V.....	46
4.6 Simulated centerline neutral particle density as a function of axial distance from the cathode for an operating voltage of 300 V.....	47
4.7 Simulated centerline neutral particle density as a function of axial distance from the cathode for an operating voltage of 250 V.....	47
4.8 Simulated centerline neutral particle density as a function of axial distance from the cathode for an operating voltage of 200 V.....	48
4.9 Centerline electron densities for experimental and simulation results for an operating voltage of 300 V.....	49
4.10 Centerline electron densities for experimental and simulation results for an operating voltage of 250 V.....	49

LIST OF FIGURES (Continued)

<u>Figure</u>	<u>Page</u>
4.11 Centerline electron densities for experimental and simulation results for an operating voltage of 200 V.....	50
4.12 Simulated centerline electron drift velocity as a function of axial distance from the cathode for standard operating conditions	52

LIST OF TABLES

<u>Table</u>	<u>Page</u>
4.1 Simulation results for all standard and varied operation parameters.....	40
4.2 Experimental and numerical results for the SPT-70 Hall thruster with standard operating parameters.....	41
4.3 Simulation results for varying superparticle size with standard operating parameters.....	53

DEDICATION

I would like to dedicate this work to my partner Catherine for inspiring me and supporting me through many long nights studying. I know that I would not have found success without you by my side and I hope that I will be able to support you in kind for all the challenges and adventures that lie ahead.

1 Introduction

1.1 Background

Since the origins of space exploration, chemical propulsion has been the primary means of transporting cargo and people into space. To this day, chemical propulsion remains the only option to launch a payload from the Earth's surface into orbit. There are some missions, however, where the use of chemical propulsion has proven much less effective than alternative propulsion systems. Electric propulsion systems rely on electricity to provide energy to the propellant as opposed to releasing internal chemical energy through combustion and provide significant advantages in terms of the amount of thrust that can be utilized from a given mass of propellant [1]. For this reason, electric propulsion systems are frequently used for satellite station keeping and navigation, as less propellant is needed in order to provide thrust for the duration of the satellite's lifetime.

Hall thrusters are relatively simple and inexpensive electric thrusters that use a static axial electric field and radial magnetic field to expel plasma propellant from the thruster. The propellant usually consists of heavy noble gases such as xenon or krypton, but other gases or even metals can be used [2]. The propellant is injected in the back of the thruster cavity at the anode, as seen in figure 1.1. The gas is then ionized through collisions with electrons and the resulting plasma is used to provide thrust. The thruster cavity is typically annular in shape with the anode at the back end of the cavity and the cathode existing as either a metal ring on the walls at the exit of the cavity or, as in figure 1.1, simply as a plume of electrons.

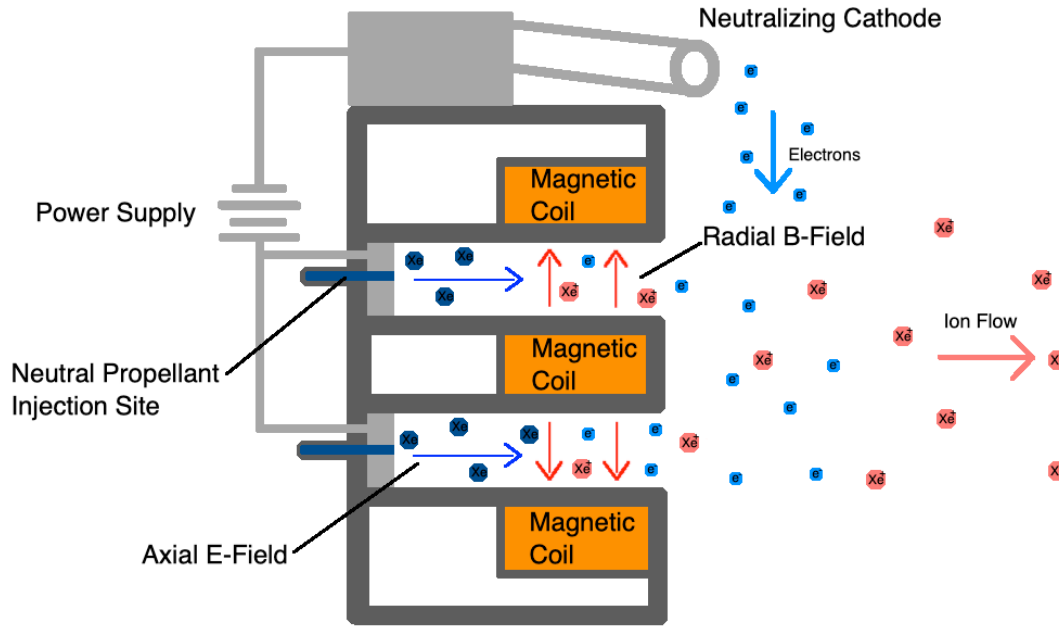


Figure 1.1. Cross Sectional View of a Hall Thruster.

The electric field in a Hall thruster serves to accelerate ions from the propellant to provide the thrust, while the magnetic field serves to confine the electrons to movement in the azimuthal direction through $\mathbf{E} \times \mathbf{B}$ drift. To understand this process, consider the Lorentz force:

$$\mathbf{F} = q(\mathbf{E} + \mathbf{v} \times \mathbf{B}) \quad (1)$$

In equation 1, \mathbf{F} is the force on the particle, q and \mathbf{v} are the charge and velocity of the particle respectively, and \mathbf{E} and \mathbf{B} are the electric and magnetic fields. Under a static electric and magnetic field, a stationary ion would be accelerated in the direction of the electric field and would initially be unaffected by the magnetic field as the particle's velocity is zero. Now imagine that the magnetic field is perpendicular to the electric field. As the particle is accelerated by the electric field, its velocity increases and it experiences force with a component perpendicular to both \mathbf{v} and \mathbf{B} due to the

cross-product term. The $\mathbf{v} \times \mathbf{B}$ term does not change the magnitude of the velocity vector, but rather rotates it in the plane orthogonal to the magnetic field. Without an electric field, a particle moving in a magnetic field would follow a stationary circular path. In the presence of an electric field, the particle travels faster when on the “downhill” side of the electric field compared to the “uphill” side, so the particle follows a windy path depicted in figure 1.2 below. The time-averaged velocity of the particle is perpendicular to both the electric and magnetic field. A negative ion or electron follows the same trajectory but with a rotation in the reverse direction due to the negative charge q in the Lorentz force. However, the drift velocity is in the same direction as a particle with a positive charge.

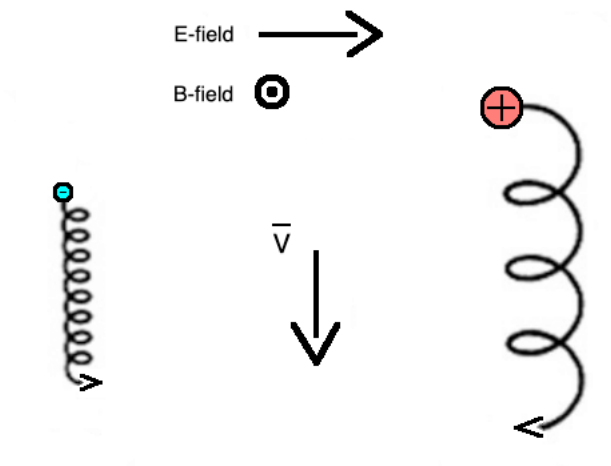


Figure 1.2. $\mathbf{E} \times \mathbf{B}$ Drift for Ions and Electrons.

With an axial electric field and radial magnetic field, an electron in the cavity of a Hall thruster would experience $\mathbf{E} \times \mathbf{B}$ drift in the azimuthal direction and be confined to travel in a ring around the Hall thruster. Ideally, the electrons would be perfectly confined as long as the thruster maintained the applied fields, but in reality, the electrons gradually diffuse out of the thruster due to collisions and other factors [3].

Electrons lost to diffusion are replaced by electrons liberated from neutral propellant and from an electron gun that also serves to neutralize the plasma plume traveling away from the thruster.

In the cavity of a Hall thruster, the ions are not contained by $\mathbf{E} \times \mathbf{B}$ drift due to the relatively larger mass of ions compared to electrons. For a particle experiencing $\mathbf{E} \times \mathbf{B}$ drift, the radius of the orbit about its guiding center is given by [4]:

$$r_g = \frac{mv_{\perp}}{|q|\mathbf{B}} \quad (2)$$

In equation 2, r_g is the radius of the orbit about the guiding center, m is the mass of the particle, and v_{\perp} is the component of velocity perpendicular to the magnetic field. Therefore, the radius of the circular motion, or Larmor radius, is directly proportional to particle mass. For a Hall thruster using Xenon propellant, a singly charged xenon ion has a mass that is approximately 240,000 times larger than an electron. However, the electrons inside a Hall thruster typically have velocities that are two orders of magnitude greater than the ions in a Hall thruster, which relatively increases the Larmor radius of electrons [1]. For a typical Hall thruster, the Larmor radius of electrons is several millimeters while the Larmor radius of ions is in the range of tens of meters. This greatly exceeds the length of Hall thrusters, typically around 10 centimeters, and so the magnetic field has a negligible effect on the ions in the cavity of the thruster. The Larmor radius for lighter propellants such as Krypton is smaller but still significantly exceeds the length of the Hall thruster cavity. The electrons in a Hall thruster are said to be “magnetically confined” while the ions are not.

Because the electrons are confined to motion in the azimuthal direction, voltages of a few hundred volts are all that is required for operation of a Hall thruster.

This makes the design and development of Hall thrusters relatively simple compared to other electric propulsion systems like the gridded ion thruster, frequently used on deep space missions, which require tens of kilovolts for operation [1]. Additionally, Hall thrusters are electrostatic devices; The only moving component is the neutral propellant that is constantly injected into the thruster during operation. As a result, Hall thrusters are cost effective propulsion systems in use on numerous small and medium satellites.

1.2 Purpose

The purpose of this work is to create a simulation code to model the plasma inside of a Hall thruster and extract data about the Hall thruster's performance. The simulation will be a two-dimensional, axisymmetric code with radial and axial domains. The SPT-70 Hall thruster will be modeled, and the results will be compared to other simulations and experimental data from the SPT-70 thruster. Parameters such as time-averaged thrust, electron and neutral density profile, and potential profile will be examined as well as the effect of superparticle size on simulation accuracy.

While the scope of this project is limited when compared to existing commercial and open-source codes, the process of developing a particle-in-cell code is laid out to provide a roadmap to others wishing to develop their own code, modify an existing code, or simply to better understand the mechanics of an existing code to use it more accurately and effectively. The applications of particle-in-cell codes extend to many applications beyond Hall thrusters, including other types of plasma thrusters, particle beams and accelerators, and fusion devices. This code aims to serve as an example of an intermediate level code which incorporates phenomena such as

fluid treatment of electrons and neutral particles and magnetic confinement of electrons while leaving out more advanced topics such as varied cell geometry and transient behavior. Those interested in using particle-in-cell codes may use this project as a reference or as a starting point to build other projects that simulate a variety of plasma phenomena.

1.3 Importance

In recent months, SpaceX has made significant strides in developing and launching communication satellites for its Starlink program. The program intends to launch thousands of communication satellites in inclined orbits to cover the Earth in a "constellation," so that any Starlink antenna on Earth within range of a Starlink satellite would be able establish a low latency broadband internet connection [5]. This would provide high speed internet to customers in remote locations, as well as a low latency connection between networks on different continents. SpaceX has plans to expand its constellation quickly, having successfully deployed 182 of its satellites over two missions by November 2019 [6]. In 2020, the company aims to deploy 1,000 satellites with the goal of eventually operating 12,000 satellites in low earth orbit by 2027 and has filed paperwork for an additional 30,000 satellites [6].

Aside from SpaceX, many other companies are working towards developing a constellation of communication satellites. OneWeb, founded in 2012, has plans to deploy 650 satellites in low earth orbit by 2021 and is expecting to expand the constellation by 1,972 satellites in the following decade [7]. Additionally, Amazon unveiled plans to deploy a constellation of 3,236 satellites in April 2019 [8]. According to Pixelytics Earth Observation Ltd., in November of 2019 there were

1957 active satellites and 3030 inactive satellites orbiting earth [9]. If SpaceX, OneWeb, and Amazon reach full deployment according to current FCC filings, an additional 17,000 satellites would be put into orbit. Furthermore, if SpaceX is granted approval for the 30,000 additional communication satellites, this would vastly increase the number of objects orbiting Earth.

In light of the massive scale of these projects, financial constraints will force companies that are developing constellations to minimize the cost and weight of each satellite. It is likely that Hall Thrusters will be used for propulsion on the vast majority of these satellites, considering their history of use and simple design which would ease production. Aside from constellation satellites, the decreasing cost of satellite launches has led to an increase in the number of small and mini satellites launched by small companies, universities, and numerous countries. Many of these satellites lack propulsion systems, but for any extended missions a scalable and efficient system is needed. Hall thrusters serve both of these requirements well and thrusters for microsatellites, themselves weighing as little as ten kilograms, are currently in development [10].

Total production of Hall thrusters will need to grow dramatically in order to keep up with demand if full scale satellite constellations are to become a reality. To date, only a few hundred Hall thrusters are on active satellites in space, yet if full constellation deployment is achieved tens of thousands of Hall thrusters would have to be produced [1]. Advancements made in optimizing and refining of computational models of Hall thrusters would benefit manufacturers of satellites, as computational modeling is a crucial step in the manufacturing of Hall thrusters. Programs modeling

the plasma inside Hall thrusters require significant computational resources in order to produce an accurate model so the development of accurate simplifications would save costs in decreasing total run time. Additionally, any work towards more efficient and reliable Hall thrusters would reduce the risk of thruster failure, decreasing the risk of satellite collisions resulting in space junk [11].

Simulation software serves as a critical tool to aid in the design and development of Hall thrusters. Research in the field of plasma simulation with applications to electric propulsion can increase the accessibility and efficiency of accurate simulation software. Hall thrusters are particularly challenging to model because of the different behavior of ions and electrons in the thruster cavity. In applications where the electrons and ions are coupled, the plasma can often be treated as a single conducting fluid which follows the equations laid out by the field of magnetohydrodynamics. These simulations can be solved using existing CFD software with the addition of magnetohydrodynamics software packages [12]. In Hall thrusters, however, the paths of large groups of particles must be tracked in order to capture the macroscopic behavior of the plasma, therefore significant computational resources are often needed to run full three-dimensional simulations of Hall thrusters with high accuracy. Plasma simulation codes that track the motion of groups of particles kinetically are called Particle-in-Cell (PIC) codes, due to the discretization of the spatial domain into small units or cells.

Several techniques are employed to increase the speed of the simulations while preserving accuracy. The first approach often taken is to model the problem in two dimensions while assuming symmetry of the collapsed dimension. In Hall

thrusters, azimuthal symmetry can be assumed after modeling electron motion in the remaining directions with diffusion coefficients [13]. Electrons in Hall thrusters are confined to move primarily in the azimuthal direction due to the $\mathbf{E} \times \mathbf{B}$ drift inside the thruster cavity, with limited non-gyro rotational motion in the radial and axial directions due to particle collisions and other factors. Eliminating the azimuthal direction in simulation, modeling the remaining electron mobility with a diffusion model significantly reduces the computational resources needed.

Another common approach includes applying a fluid model for the electrons in the plasma while maintaining a kinetic, particle-based approach for the ions and neutral particles. Because of the discrepancy of the charge to mass ratios for electrons and ions in plasmas used in Hall thrusters, electron behavior occurs in time scales orders of magnitude smaller than ions [13]. Eliminating the need to track microscopic electron behavior significantly decreases program runtime and allows for considerably larger or longer simulations to be carried out. Fluid modeling of electrons in Hall thrusters is viable because of the strong coupling between the electrons and the magnetic fields [1]. The electrons' motion is largely dictated by the magnetic and electric fields with individual particle behavior only playing a minor role- one that can be well modeled with the addition of an electron diffusion term in the equations describing electron motion. PIC codes that utilize fluid treatment of electrons are called Hybrid-PIC simulations.

Hybrid-PIC codes and codes utilizing two-dimensional analysis have been used to model Hall thrusters in research and industry. These codes are commercially available for licensing and a hybrid-PIC code specific for Hall thruster modeling

called HPHall was developed by the Jet Propulsion Laboratory and is available for free to contractors working with NASA [14]. However, open source PIC codes modeling Hall thrusters are scarce and implementing a hybrid-PIC code without commercial software essentially requires constructing the code from scratch. An open source 2D hybrid-PIC code would be valuable to companies or institutions looking to investigate new thruster designs without having to license expensive software and spending considerable computational resources. Additionally, while the simplifications made in building the software in this thesis cause the results to be less accurate than commercial software, the program can fully execute in several minutes on modern personal computers. More accurate simulations generally take hours to run and often require immense computational resources. Academic and industrial researchers may find this code valuable in running preliminary simulations of Hall thrusters in early stages of research and development. Alternatively, developers might find that the fast subroutines within the simulation are able to reduce runtime when executed within another program.

1.4 Assumptions

The models used in the program rely on the physics describing particle behavior to be correct. Ions are described by the kinetic theory of plasmas while electrons are described by the fluid theory of plasmas, which are both assumed to accurately model plasma behavior. Additionally, it is assumed that the computers used to simulate the plasmas do so as accurately as dictated by the numerical model methods- no internal errors are generated when running the program.

The geometry of the system is limited to two dimensions and the system is assumed to be symmetric about the azimuthal axis. Phenomenon arising from variations about the azimuthal axis such as oscillations are assumed to not have a significant impact on electron transport beyond what is contained in the electron diffusion approximation.

Given that the domain of the simulation ends at the thruster channel, ions may exit the thruster before accelerating to a potential of zero volts. To get around this problem, ions leaving the domain will be accelerated in the most recent direction of travel to their final velocity which is then used in thrust calculation. It is assumed that this acceleration captures the future acceleration of the particle despite omitting plasma interactions in the plume outside the exit of the Hall thruster.

1.5 Limitations

This project will focus on the development of a hybrid-PIC Hall thruster simulation code in C++. The plasma domain will be represented in axial and radial dimensions and azimuthal symmetry will be assumed. Only the region of plasma within the chamber will be simulated as outside of the thruster cavity the magnetic field lines become curved which complicate the task of simulating electron transport. Electron motion will be assumed to be instantaneous in the radial direction and will be approximated through diffusion in the axial direction. Neutral particle motion will be approximated with a simplified diffusion model described in chapter 3.

While the code can be adapted to simulate a variety of Hall thrusters, only the SPT-70 Hall thruster will be modeled in this project. The results from this simulation will be compared to other simulations of the SPT-70 thruster along with experimental

data, but further analysis about the accuracy of the simulation in varied systems will be left for future work.

A copy of the simulation code will be provided in the appendix. The code will include some basic annotations, but full descriptions of the processes used in the code will not be included.

2 Literature Review

2.1 Hall Thrusters

Various forms of electric propulsion systems have been envisioned for over a century, with the first concepts being conceived independently by famous rocket scientists Robert Goddard in 1906 and Konstantin Tsiolkovsky in 1911 [1]. Russia preceded the United States in launching electric propulsion systems including Hall thrusters, which were first tested in the 1960s using cesium and mercury as propellants. Hall thrusters were tested by the US and Russia through the 1980s. The first Hall thrusters launched for propulsion outside of testing flew in 1971 on Russian satellites. As of 2009, only 238 Hall thrusters had been flown on 48 different spacecraft [1].

A common mission that historically employed the use of Hall thrusters is the deployment and station keeping of geostationary satellites. Because of the relative simplicity of Hall thrusters, low power requirements, and high specific impulse, Hall thrusters are reliable and effective means to raise orbits to the desired profile and then maintain an orbit for an extended period of time. In one study evaluating a 6 kW class Hall thruster, the authors demonstrated that two to four 6 kW Hall thrusters were sufficient to raise the orbit of a 3 to 10 ton satellite from a geostationary transfer orbit, a highly elliptical orbit with the apogee at the geostationary orbit, to a geostationary orbit over a 4 to 6 month period [15].

Hall thrusters with a range of power and size have been investigated and employed on missions. A 200 W thruster was developed in 2017 with oblique shaped

channel that saw an increase in thrust, specific impulse, propellant utilization, and efficiency of 20% [16]. This development along with other advancements in smaller Hall thrusters will have implications for missions with satellites with masses in the range of one ton, such as the communication constellation satellites expected to be launched in large numbers this decade. At the ultimate ends of the range of Hall thruster sizes, thrusters as large as 50 kW have been tested for use in deep space missions and a thruster as small as 35 grams with a power of 3 W has been designed for deorbiting cubesats [17,18].

2.2 Development of Plasma Simulations

Computers have been used for the simulation of plasma phenomena since nearly the dawn of the computer age, with simple one-dimensional simulations being carried out in the 1950s [13,19]. Due to the limited computational resources available in early computing systems, there was little practical use in plasma simulations aside from systems involving few particles. By the early-1960s, simulations of one-dimensional plasma phenomena were used to study applications such as beam-plasma interactions and radio frequency heating of a plasma in between two parallel plates [20,21]. At Stanford University in 1964, the invention of fast accurate 2D Poisson solvers made the generalization of plasma codes to two dimensions possible [22]. These 2D simulations later became known as “Particle-in-Cell” methods, as the system was discretized spatially into cells from where the fields generated by particles were calculated. In recent decades, 3D simulations have been used to model some plasma geometries, but they usually require supercomputers or networks with massive parallel computing ability [23].

2.3 Particle-in-Cell Simulations

Particle-in-cell (PIC) methods allowed for the efficient simulation of weakly coupled plasmas where long range forces dominate interactions, since short range effects are not included in PIC calculations [13]. In strongly coupled plasmas and fluids, short range forces contribute significantly to plasma behavior and alternate simulation methods are required. Through careful choice of parameters used depending on the application, PIC simulations have been used for electrostatic and electromagnetic plasmas to capture details such as phase-space distribution functions, particle transport, and nonlinear behavior [24].

2.3.1 Superparticles

An essential aspect of PIC simulations is the use of superparticles, which are simulated particles representing anywhere from thousands to billions to high orders of ions or electrons at a single point. The resulting particle has the same charge to mass ratio of an individual particle and therefore experience the same acceleration and trajectories as its component particle [25]. Due to the fewer number of particles per Debye length, however, an increased quantity of noise is introduced to the system [26]. Given that physical systems may contain fractions of a mole of particles, the use of large superparticles is often necessary and must be weighed against accuracy and computational resources.

2.3.2 Modeling Particle Collisions

Another component function of many PIC codes is the implementation of collisions between particles in plasmas. For simulations with a large number of particles, tracking the probability of collision for each pair of particles at each timestep would introduce a large computational burden. One approach to simplify collision calculations is to only calculate the collision probabilities between particles in each cell in the simulation, termed the “Binary Collision Model” [27]. This method provides a significant reduction in the total calculations required for each time step and makes running large scale simulations computationally feasible. In simulations where collisions between multiple different plasma species must be calculated, such as ion-neutral, electron-neutral, and inelastic collisions between neutrals, the separate treatment of each class of collision further encumbers the simulation. Implementation of collisions in these simulations is usually done by the Direct Simulation Monte Carlo method, where the collision probability for each particle is found independently based on information such as particle collision frequency [19,28].

2.3.3 Accuracy and Stability Conditions

The necessary spatial and temporal discretization conditions to converse accuracy and stability for a given plasma simulation are discussed at length in [13]. In particular, the conditions relate the grid size and time step to the Debye length- the distance over which the electric potential from a charge in a plasma will be reduced by a factor of e , and the plasma frequency- the resonant frequency of electron oscillations in a plasma or metal [4]. To prevent nonphysical behavior of the simulated plasma, the spatial grid size should be less than 3.4 Debye lengths of the

plasma and the time step should be less than 2 times the inverse plasma frequency [13,29]:

$$\Delta x < 3.4\lambda_D \quad (3)$$

$$\Delta t < 2\omega_{pe}^{-1} \quad (4)$$

Outside of these conditions, the effects of the discretization will introduce inconsistencies in particle treatment that can compound and destroy the accuracy of the system. The application of accuracy and stability conditions are seen in [30,31].

2.4 Other Plasma Models

Apart from Particle-in-Cell simulations, a number of different models have been applied to simulate plasma systems. Prior to the invention of computing systems, the field of plasma physics was done through analytical methods. Instead of calculating the position and velocity of all the particles in a system at every point in time, one can look at the distribution of particles in the position-velocity phase space, denoted $f(\mathbf{x}, \mathbf{v}, t)$ [4]. Working with the distribution function of a system to extract information simplifies the calculations at the expense of losing information about individual particles. From the distribution function, the Vlasov equation, also called the collisionless Boltzmann equation, can be used to derive the fluid equations describing a plasma [32]:

$$\frac{df(\mathbf{x}, \mathbf{v}, t)}{dt} = 0 \quad (5)$$

Where d/dt is the total derivative. The Vlasov equation can be multiplied by \mathbf{v}^i where $i=0,1,2$, and integrated to produce the zeroth, first, and second order “moments” of the Boltzmann equation which describe mass, momentum, and energy conservation respectively [4]. Together with Maxwell’s equations, one arrives at a

complete set of differential equations that describe the fluid behavior, with a set of fluid equations for each species described in the plasma. This model is referred to as the two-fluid model and is used in simulations of plasmas where electrons and ions have distinctly differing behavior, such as in plasma sheaths near conducting walls [33]. In plasmas that are strongly collisional, particle distributions can be assumed to be Maxwellian and the two-fluid model can be collapsed to a single conducting fluid [4]. The magnetohydrodynamic (MHD) description of plasmas are often applied to systems with high densities, such as fusion plasmas and aerospace plasmas [34,35].

Fluid modeling of plasmas is a rich field and combinations of MHD, two fluid, and hybrid fluid-kinetic simulations are commonly used for different plasma phenomena. The treatment of electrons as a fluid is used in this thesis and is described in detail in the later chapters. However, alternative modeling methods will not be discussed further.

2.5 Hybrid Particle-in-Cell Simulations

Hybrid simulations utilize fluid treatment of one or more species, typically electrons, while utilizing kinetic treatment for other species such as ions. Because of the large disparity between the charge-to-mass ratios of electrons and ions, electron motion occurs over timescales that are up to three orders of magnitude smaller than ions [13]. In order to satisfy the accuracy and stability condition for time in equation 4 above, a PIC simulation would be limited in discretization of time by the oscillating frequency of electrons. Alternatively, treating the electrons as a fluid allows for the ion motion to dictate the minimum time discretization, allowing for simulations to run for approximately 10^3 times longer using the same total number of time steps.

In unmagnetized plasmas, plasmas without an external magnetic field applied, the electron motion is only constrained by electric forces. In plasmas without applied external electric fields, such as in the plume of a plasma thruster on a spacecraft, the forces on the electrons are solely due to the electric potential within the plasma, which is in turn generated from the net charge density. As electrons have negligible mass compared to ions, they move practically instantaneously to neutralize any large-scale, net charge accumulation, leading to what is called quasi-neutrality of the plasma [1,4,36]. In the case of no external fields, the modeling of the electron fluid becomes relatively simple, depending only on the electric potential and electron temperature through the Boltzmann relation [37]. With an added electric field, the potential from that field must also be included when finding the electric potential of points within a plasma.

Applying a magnetic field to a plasma further complicates the motion of the electron fluid. In general, motion of electrons is confined to directions parallel to magnetic field lines due to the various mechanics arising from the Lorentz force on charged particles and the small mass of the electron [4]. Depending on the type of phenomenon observed and the associated magnetic field, various mechanisms are used to model the behavior of the electron fluid. Typically, many investigations into electron fluid behavior of plasmas center on the transport of electrons across magnetic field lines as mechanisms for this type of transport are not well understood [1]. An important application subject to the consequences of anomalous electron transport is in Hall thrusters, which utilize magnetic fields to trap electrons, where the loss of

electrons can negatively impact the efficiency and reduce the lifetime of the thruster through degradation of surfaces via electron impact [38].

2.5.1 Hybrid-PIC Hall Thruster Simulations

In principle, most concepts for plasma propulsion systems rely on the same mechanism to generate thrust: accelerate ions using electric fields to generate thrust. However, Hall thrusters differ through their use of magnetic fields to confine electrons and simplify their design. Gridded ion thrusters operate similar to Hall thrusters but without a magnetic field, leading to electrons collide with the positively charged anode and complete the circuit with an electron current coming from the cathode [1]. As a result, gridded ion thrusters require voltages of tens of kilovolts to operate, requiring a high voltage power system on a spacecraft that complicates design considerations. However, the simulation of such systems is relatively simpler as electrons can be modeling using the Boltzmann relation as Jia et al. did in [39]. Hall thrusters are the opposite in that they are significantly simpler to construct but require more extensive modeling considerations.

Hybrid-PIC simulations all use the same formulations for the majority of system mechanics, differing mostly in the modeling of phenomena that are not well understood such as electron transport across magnetic field lines. The discussion here will focus on the modeling of electron transport across field lines as this is the most consequential to the development of the model in this thesis. The most common electron diffusion model used is the classical model, contributing electron mobility to collisions primarily with neutrals, but also sometimes including electron-ion, electron-wall, and electron-electron collisions [40]:

$$v_{z\,drift} = -\mu_{\perp} E_z - \frac{D_{\perp}}{n_e} \frac{\partial n_e}{\partial z} \quad (6)$$

Where

$$\mu_{\perp} = \frac{e}{m_e v_{en} [1 + (\frac{\omega_{ce}}{v_{en}})^2]} \quad (7)$$

$$D_{\perp} = \frac{kT_e}{e} \mu_{\perp} \quad (8)$$

Here, $v_{z\,drift}$ is the electron drift velocity in the z direction, μ_{\perp} is the electron mobility, D_{\perp} is the diffusion coefficient for electrons crossing magnetic field lines, e is the elementary charge, m_e is the electron mass, n_e is the electron density, v_{en} is the electron-neutral collision frequency, and ω_{ce} is the electron cyclotron frequency.

Shashkov et al. in [41] also includes electron-ion collisions in the classical mobility model while Hara et al. in [42] included electron-neutral and electron-wall collisions.

Classical electron diffusion deviates from experimental results yet works as a close approximation for many applications. Some authors will further augment the model by including Bohm diffusion, derived from a random-walk model of electron transport within the plasma. The diffusion coefficient from Bohm diffusion is:

$$D_{Bohm} = \frac{1}{16} \frac{k_B T}{e B} \quad (9)$$

Where e is the electron charge, T is temperature, k_B is the Boltzmann constant, and B is the magnetic field. From this Fife in [43] arrives at the expression for classical-Bohm diffusion given by equation 8:

$$m\mu_{\perp} = \frac{m\mu_e}{\beta_e^2} + K_B \frac{1}{16B} \quad (10)$$

Where $m\mu_e$ is the classical mobility, β_e is the electron Hall parameter given by $\frac{\omega_{ce}}{\nu_{en}}$, and K_B is an adjustable parameter to weight the contribution of Bohm diffusion. In the SPT-70 Hall thruster modeled by Fife a K_B value of 0.15 gave theoretical values of electron diffusion that were in agreement with experimental results. Cao et al. in [44] also used this approach while using a K_B value of 0.25 to model the SPT-100 Hall thruster.

Other approaches to the problem of electron diffusion modeling include using purely empirical data and avoiding fluid treatment of electrons altogether. Sommer et al. in [45] utilized this approach, achieving strong agreement between experimental and theoretical data. This method potentially obscures more complicated and anomalous behavior of electron transport that might be useful to include in some applications. Other authors such as Cao et al. in [46] and Coche and Garrigues in [47] use a kinetic simulation of electrons, forgoing the hybrid-PIC method for a pure PIC simulation. A full PIC model has the advantages of accurately modeling electron behavior at the expense of significantly decreasing the maximum time step allowed for each iteration, dramatically increasing the computational cost of a simulation. While all approaches have their respective strengths and weaknesses, the classical electron diffusion model will be used in this thesis.

3. Methods

3.1 The Challenge of Kinetic Simulations

Plasma simulations utilize various tools to simplify the complicated interactions within a region of plasma. The main difference between the interactions between particles in a typical fluid and the particles in the plasma is the distance over which interactions can occur. In a gas, for example, particles essentially only interact through collisions, either with each other or a surface in contact with the gas [48]. Macroscopic behaviors of gases then arise from largely collisions and other short-range interactions. In a plasma, the plasma species carry charges that can interact over larger ranges within the bulk of the plasma. Recall the Debye length mentioned in earlier chapters, which we will define here by the relationship [4]:

$$\lambda_D = \sqrt{\frac{\epsilon_0 k_B T_e}{N_0 q_e^2}} \quad (11)$$

Where λ_D is the Debye length, ϵ_0 is the vacuum permittivity, k_B is the Boltzmann constant, T_e is the electron temperature, N_0 is the density of both the ions and electrons assuming a quasineutral plasma, and q_e is the charge of an electron. The Debye length is the distance over which the electric field from a charge in the plasma will decrease by a factor of e . From here it can be estimated that the number of particles that can be affected by a single charge fall within a sphere with a radius corresponding to the Debye length [4]:

$$N_D = N_0 \frac{4\pi\lambda_D^3}{3} \quad (12)$$

If we apply this relationship to plasmas found in a Hall thruster, where electron temperatures frequently fall into the range of 5 eV and charged particle densities are around 10^{17} particles per meter cubed, we find the Debye length is about 52.3 microns [1]. Plugging this into equation 12 we find that a particle in a Hall thruster interacts with 60,000 other particles in the same type. Since ions and electrons can also interact, each particle is effectively in contact with 120,000 other particles, meaning that a purely kinetic simulation would have to compute 120,000 interactions for each particle at each time step.

To avoid having to simulate the numerous interactions between particles in a plasma, simplifications can be made by discretizing the domain into cells. From there, the charge density due to the particles in each cell can be calculated. Using Poisson's equation describing the relationship between charge density and electric potential, the electric field can be calculated at each grid point which is then used to accelerate particles. In this way, particles are interacting not with one another but with the special domain surrounding it, only indirectly interacting with particles around it. This feature of the simulation is where the name "particle-in-cell" originates.

3.2 Particle-in-Cell Simulation Structure

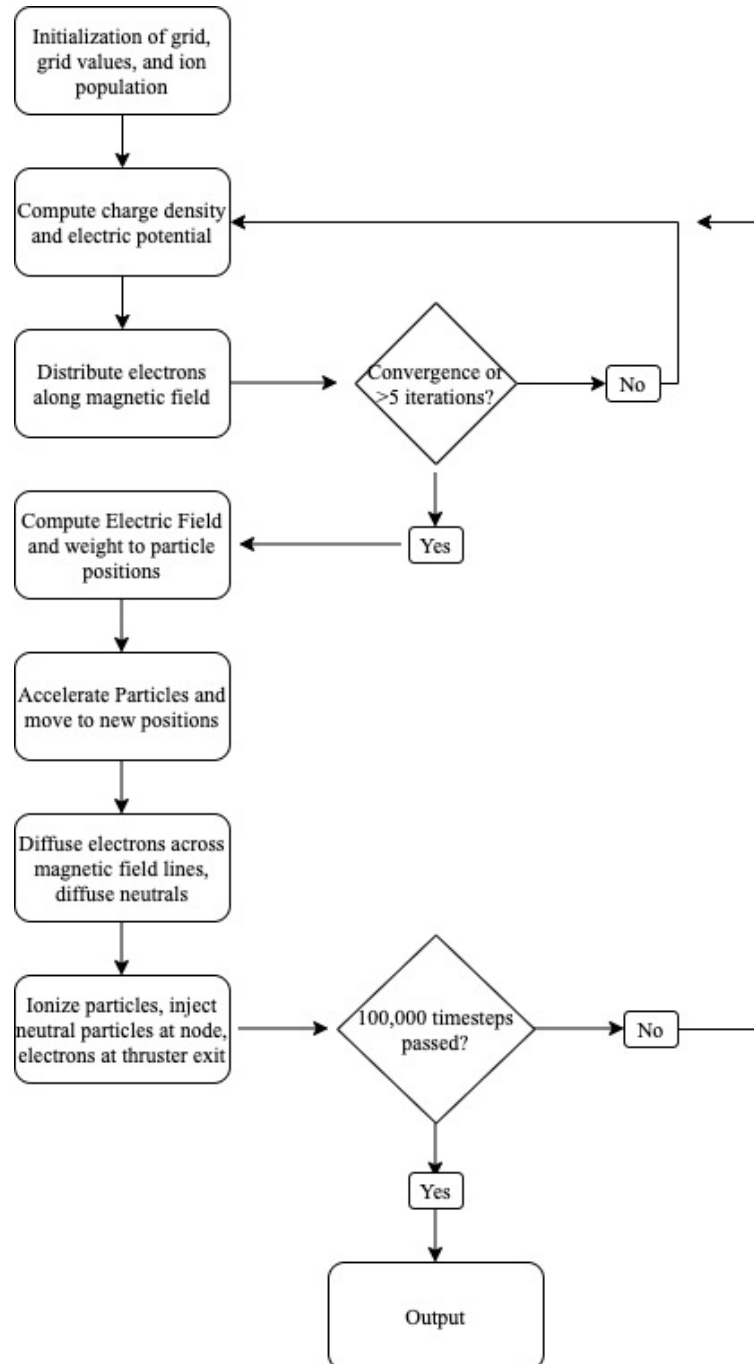


Figure 3.1. Algorithm flowchart for the hybrid-PIC simulation

3.2.1 Simulation Overview

The simulation code for this thesis is written in C++. The structure of the simulation code is shown in figure 3.1 above. For the sake of comparing results to

existing data, the SPT-70 Hall thruster was chosen as an initial model to design the code. The special domain is divided into radial and axial segments of 1 mm each. The SPT-70 Hall thruster channel has an inner radius of 20 mm, outer radius of 35 mm, and an axial length of 29 mm [43]. As a result, the domain contains 16 by 30 grid points with cells in between. A time step of 10 ns was initially chosen to ensure that even the fastest particles during the simulation would travel less than one grid length per time step.

3.2.2 Initialization

As the SPT-70 is a commercially available Hall thruster, some detailed data is unavailable such as the plasma density. Therefore, data from the Stanford Hall thruster which has a similar power level was used for initial plasma values. An initial ion density of $1.7 \times 10^{17} \text{ m}^{-3}$ and an initial neutral gas density of $3.2 \times 10^{19} \text{ m}^{-3}$ at the gas inlet and $1.6 \times 10^{19} \text{ m}^{-3}$ at the thruster exit were used [40]. For the first program runs, 9 ion superparticles were placed in each cell. Since the sides of each cell measure 1 mm, each superparticle represents 1.89×10^7 ions using the previous initial ion density. Since electrons and neutrals are not modeled as discrete superparticles, the values of each at every grid point can be stored as a continuous value. The plasma is assumed to be quasi-neutral and so the initial electron density is the same as the initial ion density.

For each ion, radial and axial position and velocity are stored in an array with a length corresponding to the maximum number of particles that the simulation can handle. Because C++ is a compiled programming language, the size of each array must be predetermined at runtime unless the use of dynamic arrays is employed,

significantly increasing the complexity of the code and running the risk of crashing without a carefully crafted algorithm. Initially, a maximum number of ions was set to 5000 times the initial number of superparticles in each cell to give room for a large increase in ions over the initial density.

At initialization, a loop places 9 ion superparticles in each cell with a random r - and z - position within the cell. The initial radial velocity is set to zero while the initial axial velocity is calculated based on its axial position. A linear relationship is applied where particles with $z = 0$ at the inlet of the thruster have a velocity of zero and particles with $z = 29$ mm at the exit have a velocity of 17 km/s, which is around the measured exhaust velocity of similar Hall thrusters [40]. Velocity is interpolated based on each superparticle's z - position. To avoid unused particles in the array contributing to calculations within the domain, particles not initialized to the grid are given a position of $(-1, -1)$.

Initial electron densities are uniform across the grid according to the quasi-neutral approximation of the plasma. However, since electron values are stored at grid points, electron densities are slightly smaller than the number of ions per grid since there are $(n-1)$ by $(m-1)$ cells within an n by m grid. Neutral particle densities are also stored at each grid point.

Arrays are also initialized to store information about the fields within the domain. Charge density, electric potential, and electric field are all initialized as independent arrays. Charge density is initially set to zero since values for charge density are found before it is used in any calculations. The SPT-70 operates at 300 V and so the voltage at the left edge of the domain, also where the gas inlet is, was set to

300 V. A linear relationship is applied that drops the potential by 10 V for every axial grid point towards the thruster exit, leaving the grid points at the right edge of the domain at 10 V. This is a valid initial estimate since an electrode gun outside of the thruster serves as the cathode, meaning that the point of zero electric potential could easily lie outside of the domain [43]. Initially there is no radial variation in potential introduced into the problem. Initial electric field is found by using a finite difference method to take a derivative of the electric potential [13]:

$$E_{z,i} = -\frac{\phi_{i+1,j} - \phi_{i-1,j}}{2\Delta z} \quad (13)$$

And at boundaries:

$$E_{z,i} = -\frac{\phi_{i,j} - \phi_{i-1,j}}{\Delta z} \quad (14)$$

$$E_{z,i} = -\frac{\phi_{i+1,j} - \phi_{i,j}}{\Delta z} \quad (15)$$

3.2.3 Loop Functions

At the start of each loop the first step is to compute the charge density at the grid points using the position of each ion superparticle. The scheme used to distribute the charges from each ion to the surrounding grid points is called a 1st order scatter operation [13]. It is a linear operation that weights the contribution of charge to each cell based on the distance of the particle from each grid point. A sketch of this operation in one dimension is shown in figure 3.2. In two dimensions, the scatter operation acts in both dimensions at the same time. The weight dictating the contribution of charge to each grid point then corresponds to an area with sides composed of the length from the particle to the grid point in each dimension. A sketch of this is shown in figure 3.3.

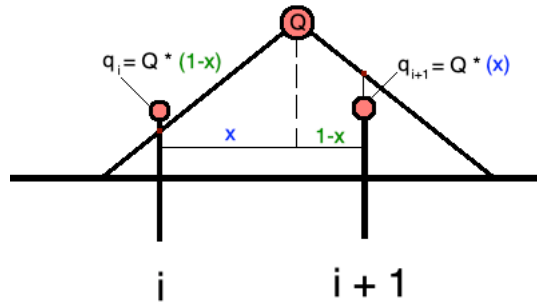


Figure 3.2. A sketch of the contribution of the ion's charge to each grid point in 1D.

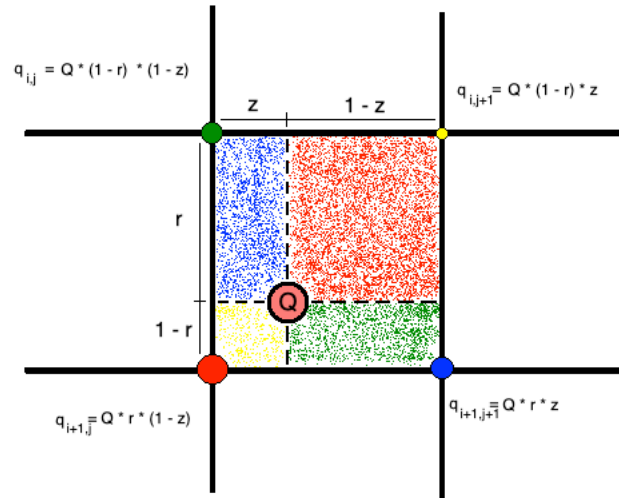


Figure 3.3 A sketch of the contribution of the ion's charge to each grid point in 2D.

Once the charge density at each grid point is calculated, electric potential at each point can then be found using Poisson's equation. Poisson's equation, reduced from Maxwell's equations, relates the total charge in a region of space to the Laplacian of electric potential through [49]:

$$\nabla^2 \phi = -\frac{\rho}{\epsilon_0} \quad (16)$$

Where ϕ is electric potential, ρ is the charge density, and ϵ_0 is the vacuum permittivity. Using the finite difference approximation for derivatives and recalling

that our system uses radial coordinates, we arrive at the discretized version of Poisson's equation [13]:

$$\frac{\phi_{i,j+1} - 2\phi_{i,j} + \phi_{i,j-1}}{(\Delta r)^2} + \frac{1}{r_{i,j}} \frac{\phi_{i,j+1} - \phi_{i,j-1}}{2\Delta r} + \frac{\phi_{i+1,j} - 2\phi_{i,j} + \phi_{i-1,j}}{(\Delta z)^2} = -\frac{\rho}{\epsilon_0} \quad (17)$$

This can be rewritten in terms of $\phi_{i,j}$ to yield:

$$\phi_{i,j} = \left[\frac{\phi_{i,j+1} + \phi_{i,j-1}}{(\Delta r)^2} + \frac{1}{r_{i,j}} \frac{\phi_{i,j+1} - \phi_{i,j-1}}{2\Delta r} + \frac{\phi_{i+1,j} + \phi_{i-1,j}}{(\Delta z)^2} + \frac{\rho}{\epsilon_0} \right] * \left[\frac{2}{\Delta z} + \frac{2}{\Delta r} \right]^{-1} \quad (18)$$

Using a Dirichlet boundary condition at the inlet of the thruster which is fixed at 300 V, and Neumann boundary conditions for the other boundaries, a solution to this equation can be found numerically. Recall that a Dirichlet boundary condition is fixed at a given value, while for a Neumann boundary condition the derivative of the function is zero, or in our case, $\frac{d\phi}{dx} = 0$ [50]. From here any Poisson solver can be used to find the solution. In this project, an iterative Jacobi solver method is used.

An additional step is taken in hybrid simulations, where the electrons are not treated as superparticles like ions. In practice this means solving for the distribution of electrons and electric potential simultaneously. Since electron motion occurs on a time scale that is much shorter than the time scale of the simulation, the electrons appear to instantaneously redistribute according to the potential. The electrons carry charge themselves, however, and potential must then be found with the electrons in place. This process must be iterated over until an acceptable convergence is found and the electron density and potential reach stable values.

In Hall thrusters, electrons are magnetically confined in the axial direction but more or less free in the radial direction. Therefore, electron diffusion across magnetic field lines is slow and takes place in between time steps in the simulation. In the case

of motion along magnetic field lines, the Boltzmann approximation can be applied to simplify calculations. The Boltzmann relationship comes from thermodynamics and relates the number of particles in each state to the energy level of each state [48]. In the case of electrons in an electric potential, the Boltzmann relationship becomes:

$$n_e = n_{e0} e^{q(\phi - \phi_0)/kT_e} \quad (19)$$

Where n_e is the electron number density, n_{e0} is a reference electron density, q is the electron charge, ϕ is the electric potential, ϕ_0 is a reference potential, and kT_e is the electron temperature. Prior to computing the potential and electron distribution in each column along the magnetic field lines, the average potential is found as used as the reference potential, and the total number of electrons in the column is determined and used to normalize the Boltzmann relationship. Since electrons are assumed to be isothermal, a value of $kT_e = 5 \text{ eV}$ is used in all calculations, closely approximating real electron temperatures in [40,43].

Once the electric potential and electron distributions are found, the electric fields at each grid point can be calculated using equations 11, 12, and 13 with equivalent equations in the radial direction. The next step is to move the particles. A method called the leapfrog method is frequently used in PIC codes due to its simplicity and numerical stability [13]. It involves moving particle velocity and positions in half time steps sequentially, with velocity change coming from the Lorentz force:

$$v^{t+0.5} = v^{t-0.5} + \frac{qE\Delta t}{m} \quad (20)$$

$$x^{t+1} = x^t + v^{t+0.5}\Delta t \quad (21)$$

The velocity and position steps are varied since it allows the average velocity between t and $t+1$ to be applied towards the change in position over the time step. To implement the leapfrog method, the velocity needs to be traced backwards half a time step for the first step. Since the initial velocities in the system modeled here are significantly higher than the velocity added in half a time step, ignoring the velocity backtracking introduces a negligible difference in the initial conditions, which essentially serve as an initial guess to the steady state of the system.

After the ion particles are moved to their new positions, electron diffusion across magnetic field lines is determined using the classical diffusion described in equations 6-8. The value for electron cyclotron frequency ω_{ce} can be determined for our system globally since a uniform magnetic field of 0.02 T exists in the SPT-70 thruster chamber [43]. The equation for ω_{ce} is provided in [4]:

$$\omega_{ce} = \frac{|q|B}{m_e} \quad (22)$$

Plugging in a value of 0.02 T, we find $\omega_{ce} = 3.52 * 10^9 \text{ Hz}$. Ahedo, Martinez-Cerezo, and Martinez-Sanchez provide an expression for the electron-neutral collision frequency in [51]:

$$\nu_{en} = n_n \sigma_{en} c_e \quad (23)$$

Where n_n is the neutral particle density, σ_{en} is the collision cross section which the authors report is $2.7 * 10^{-19} \text{ m}^2$ for neutral Xenon and eV range electrons, and $c_e =$

$\sqrt{8T_e/\pi m_e}$ is the mean electron thermal velocity. Since we assume isothermal

electrons with a temperature of 5 eV, c_e will be a constant value.

After electrons are diffused across magnetic field lines, then neutral particles are allowed to flow across grid lines towards the exit of the thruster. Velocity of the neutral particles was assumed to be a constant value given by the specific impulse of a cold gas thruster. The specific impulse of a xenon cold gas thruster has been measured to be approximately 28 s, which corresponds to an exhaust velocity of 274.4 meters per second [52]. While the speed of the gas is assumed to be constant, the direction was assumed to have some radial component as the gas flows through the Hall thruster chamber. In Dettleff and Grabe, particle flux as a function of direction was found through experiment and numerical simulation [52]. Using the angular flux distribution in the publication, approximately 2.12% of particles will have an angle of either positive or negative 45 degrees with respect to the axial axis. At an angle of 45 degrees, half of the velocity will be in the radial direction meaning that neutral particles will translate one cell towards the exit of the thruster and one cell in the positive or negative z- direction. Particle distributions at angles greater than 45 degrees were insignificant.

Following the diffusion of electrons and neutral particles, the ionization of neutral particles is calculated. This is essentially a particle injection function, where ions and electrons are produced as neutral particles are ionized. For the sake of simplicity, we assume that recombination is negligible in the hall thruster channel, occurring mostly in the plasma plume outside of the domain of the modeled system [1]. Assuming that every collision ionizes a neutral particle by freeing an electron, the of change of ions, electrons, and neutral particles due to ionizations is given by:

$$\frac{dn_{ion}}{dt} = \nu_{en} n_{electron} \quad (24)$$

$$\frac{dn_{electron}}{dt} = \nu_{en} n_{electron} \quad (25)$$

$$\frac{dn_{neutral}}{dt} = -\nu_{en} n_{electron} \quad (26)$$

Since the electron neutral collision frequency term depends on the neutral particle density, the total amount of collisions at each point depends on both electron and neutral particle densities as expected.

Note that the ionization step takes place after diffusion has taken place. The reason for this is not because of the factors that go into ionization but because of the products that come out. Ideally, ionization would occur with the calculation of radial distribution concurrently, but since the electron distribution is coupled with the Poisson solver, this would require further coupling ionization which would introduce a large computational burden. Since the ionization depends on the recently diffused electrons, a discrepancy will arise in that the electron concentration will be artificially high, however the electrons will be redistributed in the potential solver prior to their use in any calculations. As for the ions, the introduction of particles one cell to the right will only result in the loss of acceleration of one grid cell, an acceptable error for this calculation. Introduced ions are born with randomized positions in the cell and with velocities calculated using the same method as in the initialization.

The last step in the loop iteration is the introduction of neutral particles at the thruster inlet and electrons at the thruster exit from the cathode. Neutral particles are injected according to the mass flow rate, which for the SPT-70 Hall thruster is 2.34

mg/s [43]. The introduction of electrons is dependent on the current for the Hall thruster, again for the SPT-70 Hall thruster is 2.2 Amps.

3.2.4 Simulation Output

At the end of the series of loops conducted for the simulation, either for a present number of iterations or until convergence is reached, data can be extracted to provide information about the Hall thruster. In this simulation, the loop was run for 100,000 iterations to allow for convergence to a steady state without prohibiting variations that arise between time steps. To extract useful information from the simulation, an average value of the last 100 iterations is taken, smoothing out the effects of any natural oscillations.

The values of interest to both validate the simulation code and to estimate values for Hall thruster performance are time-averaged thrust, electric potential profile, and electron density. The calculation of thrust is the primary purpose of the simulation and can be found by summing up that axial component of ion velocities for particles that leave the system. We assume that the net of velocities in the radial direction is zero and do not sum over radial velocities. Often the potential at the exit of the thruster is nonzero, meaning that the ions will continue to accelerate until they reach an electric potential of zero. To account for this, ions are artificially accelerated in their direction of travel the moment they leave the thruster with any additional radial velocity being counted towards thrust.

The electric potential profile is important in that it provides a picture of how the ions are accelerated within the channel. The shape of the potential inside of a Hall thruster along with the electron density also allows for a comparison with

experimental Hall thruster data. The discretization of the spatial and temporal domains will frequently eliminate phenomenon seen in physical systems as well as create nonphysical phenomenon. An analysis of the final ion distributions will not be included in this work but electron and neutral particle densities at the centerline of the thruster will be provided.

3.2.5 Error Analysis

The estimation of numerical error in particle-in-cell simulations is difficult due to the numerous simplifications that are made in modeling the problem. Cartwright and Radtke highlight this in [53]: “Numerical error estimation for PIC plasma simulations is challenging due to multiple discretization parameters” including “grid, time step, and macroparticle weight” and stochastic noise arising from these factors. Individually, error arising from each factor can be determined by testing varying values for each parameter. One common factor used for error estimation in fluid simulations is the spatial discretization. A procedure for estimated error using spatial discretization size is laid out in [54]. This involves starting with some value for average grid size used in the simulation and then testing courses and finer meshes to simulate the same system. From there, finding the difference between solutions for some value such as flux will eventually show convergence to some value. This value can then be used to estimate the error introduced by the spatial discretization used in the simulation.

This method cannot be used in particle-in-cell methods, however. Applying a finer mesh to PIC simulations can actually introduce additional error from a variety of factors. One of the largest issues is that the charge from superparticles is distributed

only to the grid points of the cell containing the superparticle. As the mesh size shrinks, fewer superparticles will be in each cell and the resulting electric field around each cell originates from fewer superparticles. In the extreme case, where many cells are empty and the rest containing only one superparticle, the electric field that moves the superparticles comes only from itself, creating a nonphysical self-interaction that can introduce enormous error [13].

Varying sizes of superparticles and time discretization can be used as a tool to estimate error. In the system simulated in this project, the time discretization is already chosen to ensure that superparticles travel less than one grid per time step except in outlying cases. Varying the time scale will likely have an effect on the fastest superparticles in the simulation and on some transient behavior such as oscillations. However, in small scale test runs, reducing the timestep by a factor of ten led to no significant change in the results at a much larger computational cost. Additionally, transient plasma behavior is not included directly in any output of the simulation and the error introduced by the time scale size is acceptable for the scope of this project.

The parameter that will be used to estimate error in this project will be superparticle size. The smaller the superparticles, the closer the simulation becomes to a kinetic simulation, which is essentially an extreme case where the superparticle mass is equal to a regular particle. Smaller particle size will result in a smoother distribution of charge within each cell, resulting in a system that more closely models a charge distribution in a real plasma. The smaller error then feeds into the calculation of potential, electric field, and eventually the force on each particle, resulting in a

more accurate simulation. The program run time increases significantly, however, as superparticle size is reduced due to the number of times loop functions need to be called for every superparticle.

Several different values of superparticle size will be used to determine its relationship with simulation accuracy. A large decrease in superparticle size will not be feasible for this study as the simulation would take too long to run, but incremental decreases in size will be examined. The results will be presented in the next chapter as well as a discussion on the benefit of using smaller superparticle size compared to the increase in computational cost.

4 Results

4.1 System Setup

The simulation was run on a 2017 macbook pro with the 10.15.3 version of macOS Catalina. The computer contained a 2.3 GHz Dual-Core Intel i5 processor and 8 GB of 2133 MHz LPDDR3 RAM. The code, written in C++, was executed using the GNU Compiler Collection (GCC) compiler. The Integrated Development Environment (IDE) used was the Eclipse C++ IDE and the simulations were run through the Eclipse interface. When using the standard operating parameters for the SPT-70 Hall thruster the simulation consistently executed around 4 minutes. Variations of the standard input parameters led to no noticeable change in execution time. Decreasing the superparticle size, leading to a larger number of total particles, did lead to an increase in execution time but still never exceeded an execution time of 9 minutes.

For each simulation a total of 100,000 timesteps was used with each step corresponding to 10 ns for a total simulation time of 1 ms. In all cases the thrust reached a convergence of 10^{-5} within 50,000 timesteps. The initial superparticle density was set to 9 superparticles per cell corresponding to 1.89×10^7 particles for each superparticle. For analysis against experimental data, 9 simulations were run varying the anode voltage between 300 V, 250 V, and 200V with neutral mass injection varied between $2.34 \mu\text{g/s}$, $1.76 \mu\text{g/s}$, and $2.93 \mu\text{g/s}$. To test the effect of superparticle size on simulation accuracy, 4 additional simulations were run with

anode voltage at 300 V, a neutral mass injection of $2.34 \mu\text{g/s}$, and initial superparticle densities of 3, 6, 12, and 15 per cell.

4.2 Simulation Results

Simulation	Anode Voltage	Injection Rate	Superparticle Size (particles per superparticle)	Thrust	Specific Impulse (I_{sp})
SOP-1	300 V	$2.34 \mu\text{g/s}$	1.89×10^7	33.2 mN	1446 s
VOP-1	300 V	$1.76 \mu\text{g/s}$	1.89×10^7	20.2 mN	1172 s
VOP-2	300 V	$2.93 \mu\text{g/s}$	1.89×10^7	46.1 mN	1603 s
VOP-3	250 V	$2.34 \mu\text{g/s}$	1.89×10^7	29.5 mN	1285 s
VOP-4	250 V	$1.76 \mu\text{g/s}$	1.89×10^7	19.1 mN	1107 s
VOP-5	250 V	$2.93 \mu\text{g/s}$	1.89×10^7	39.8 mN	1384 s
VOP-6	200 V	$2.34 \mu\text{g/s}$	1.89×10^7	24.7 mN	1074 s
VOP-7	200 V	$1.76 \mu\text{g/s}$	1.89×10^7	17.0 mN	985 s
VOP-8	200 V	$2.93 \mu\text{g/s}$	1.89×10^7	32.5 mN	1132 s
SOP-2	300 V	$2.34 \mu\text{g/s}$	5.67×10^7	24.8 mN	1080 s
SOP-3	300 V	$2.34 \mu\text{g/s}$	2.83×10^7	29.1 mN	1268 s
SOP-4	300 V	$2.34 \mu\text{g/s}$	1.42×10^7	34.7 mN	1512 s
SOP-5	300 V	$2.34 \mu\text{g/s}$	1.13×10^7	34.9 mN	1520 s

Table 4.1. Simulation results for all standard and varied operation parameters.

The results from all of the simulation runs are summarized in table 4.1.

Simulations are named according to whether the inputs were standard operating parameters (SOP) or varied operating parameters (VOP). The inputs are anode

voltage, injection rate, and superparticle size while the outputs are thrust and specific impulse.

4.2.1 Simulation Results with Standard Operating Parameters

	Simulation	Experiment [43]	CASE 1 [43]	CASE 2 [43]
Thrust	33.2 mN	37.8 mN	34.2 mN	41.4 mN
Specific Impulse (I_{sp})	1446 s	1644 s	1489 s	1803 s

Table 4.2. Experimental and numerical results for the SPT-70 Hall thruster with standard operating parameters. Experimental values from [43].

Table 4.2 displays the results of SOP-1, experimental results, and results from other simulations of the SPT-70 Hall thruster during standard operating conditions. While error associated with experimental measurements are not provided, the thrust stand used has an associated error of 2% [55]. In [43], the author presents an advanced numerical model for the SPT-70 Hall thruster and provides experimental and numerical results for standard operating conditions of the thruster, in addition to experimental results for varied operating parameters. Being published in 1998, the two numerical simulations conducted took approximately 8 hours to execute, therefore simulations were not run for nonstandard operating conditions.

The simulated thrust is found to be 12.2% lower than the experimental result in [43]. Recall that the specific impulse (I_{sp}) is a measure of the amount of thrust generated per mass of propellant, given by the equation [1]:

$$I_{sp} = \frac{F_{Thrust}}{\dot{m}_{propellant} g_{earth}} \quad (27)$$

Where $\dot{m}_{propellant}$ is the mass flow rate of the propellant and g_{earth} is the acceleration due to gravity at the earth's surface, approximately 9.81 m/s². The

gravitational acceleration term is used as a standard to reduce I_{sp} to units of seconds so that different types of propellant systems can be easily compared. Since I_{sp} is directly proportional to thrust, the calculated I_{sp} for the simulation is proportionally lower than the experimental results.

There are many factors that can contribute to the discrepancy in thrust, including limitations of the size of the simulated domain, discretization of the space and time domains, approximating ions as superparticles, and errors introduced from numerically solving the system. The effect of superparticle size on simulation accuracy is examined later in the chapter. The assumptions made in designing the program such as isothermal electrons and azimuthal symmetry also contribute to deviations between modeled and real plasma behavior. A combination of all of these factors likely contribute to the error in the simulation leading to an underestimation of thrust.

A large portion of the missing thrust in the simulation results is due to the limited size of the domain. The region that is modeled is limited to inside of the thruster cavity where the magnetic field lines are parallel. This simplifies the modeling of electron behavior since the cells line up with the magnetic field lines, allowing for the differentiating of electron behavior in the radial and axial directions. Outside of the thruster cavity, the magnetic field lines take on a curved shape, requiring more advanced cell geometries and field solving methods.

In the SPT-70 Hall thruster, electrons are injected outside of the thruster cavity through a neutralizing cathode. Electrons then diffuse in through the cavity, ionizing neutral particles along the way, until reaching the anode. At the site of the

electron injection, the large number of electrons create a relatively low electric potential compared to the surrounding regions outside of the thruster. The electron injection system is designed to deposit electrons near the middle of the ion beam leaving the thruster which serves to focus or tighten the ion beam by pulling ions on the edge of the beam towards the center. Since the domain is limited to the thruster cavity in the simulation, the effects of beam tightening are not observed. Beam tightening has a significant impact on thrust since the axial velocity of particles is proportional to the cosine of the angle from the z-axis. The deflection of ion velocity towards the z-axis by a few degrees can result in a few percentage points increase in thrust.

4.2.2 Simulation Results with Varied Operating Parameters

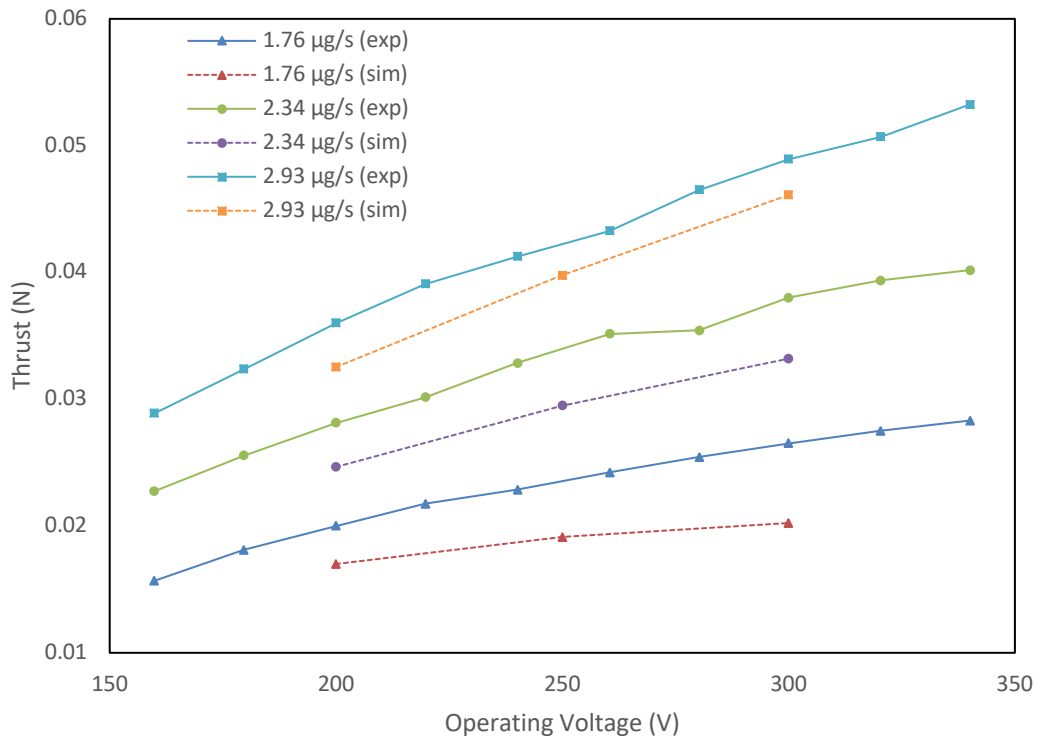


Figure 4.1. Thrust as a function of operating voltage and neutral mass injection rate for experimental and simulation results. Experimental values from [43].

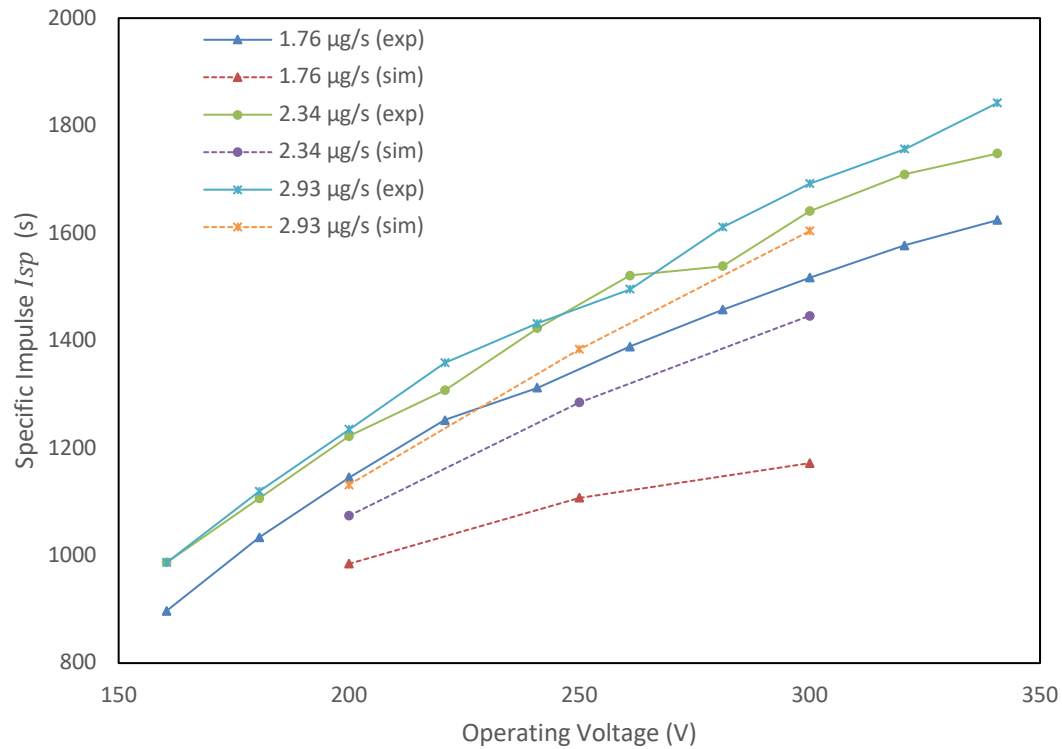


Figure 4.2. Specific Impulse as a function of operating voltage and neutral mass injection rate for experimental and simulation results. Experimental values from [43].

Experimental data was recovered from Fife in [43] using the macOS version of webplot digitizer, a program that allows the extraction of digital values from printed plots. This data was then compared to simulation results for the various input parameters used. Figure 4.1 shows the relationship between applied voltage, neutral mass injection rate, and thrust for experimental (exp) and simulation (sim) results. When comparing the results from the simulation to the measured values, the thrust is underestimated for all operating voltages and neutral mass injection rates. The simulations show a similar sensitivity to voltage as in experimental results, indicating that the discrepancy factor is not highly dependent on voltage or neutral mass injection rate. Figure 4.2 displays specific impulse as a function of operating voltage and neutral mass injection and shows a similar trend with simulated results.

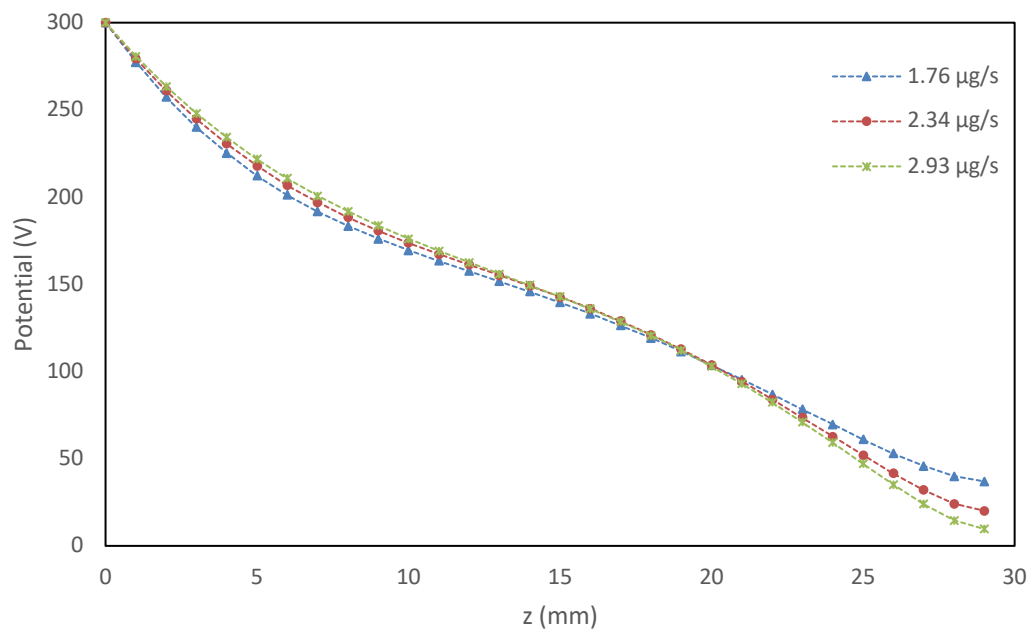


Figure 4.3. Simulated centerline electric potential as a function of axial distance from the anode for an operating voltage of 300 V.

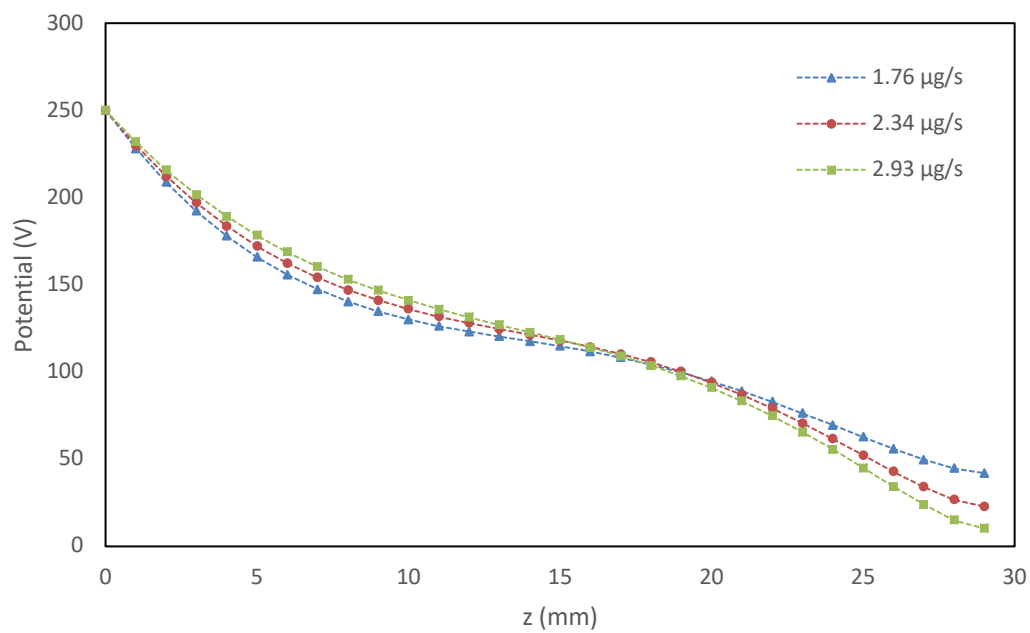


Figure 4.4. Simulated centerline electric potential as a function of axial distance from the anode for an operating voltage of 250 V.

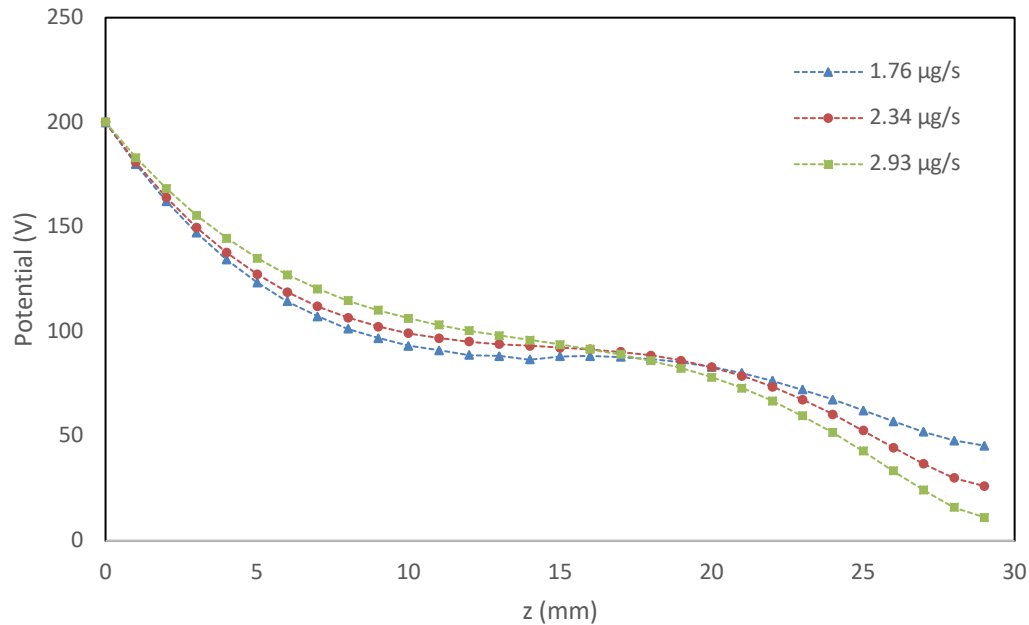


Figure 4.5. Simulated centerline electric potential as a function of axial distance from the anode for an operating voltage of 200 V.

Figures 4.3, 4.4, and 4.5 show the electric potential at the centerline with a radius of 27.5 mm, or 7.5mm from the inside and outside radial edges of the thruster cavity. Experimental values are not available for comparison as introducing an electrode will cause a Debye sheath to form around the electrode which changes the local potential [43]. If no charged particles were in the thruster cavity, the slope of the electric potential would be linear from the anode to the cathode. The deviations from a linear slope then arise from concentrations of charge in the cavity, typically electrons, since the ions rapidly accelerate out of the thruster without being magnetically confined. The electron motion, on the other hand, is coupled to the electric potential through the classical electron diffusion model described in equation 6 and nonlinear interactions such as ionization of neutral particles which introduces more electrons into the system. At lower operating voltages, the deviations from a linear slope become more pronounced for two reasons. First, the electric potential has

a shallower gradient when starting at a lower value, so the variation in potential arising from charge density is relatively larger. Second, electron densities are higher for lower operating voltages, leading to higher concentrations of charge that impact the local potential.

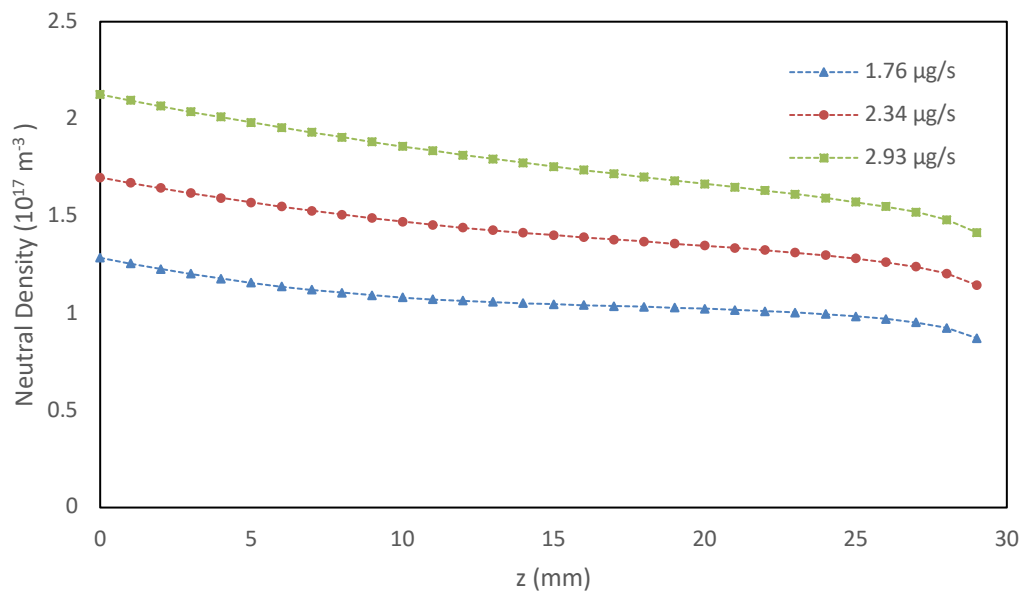


Figure 4.6. Simulated centerline neutral particle density as a function of axial distance from the anode for an operating voltage of 300 V.

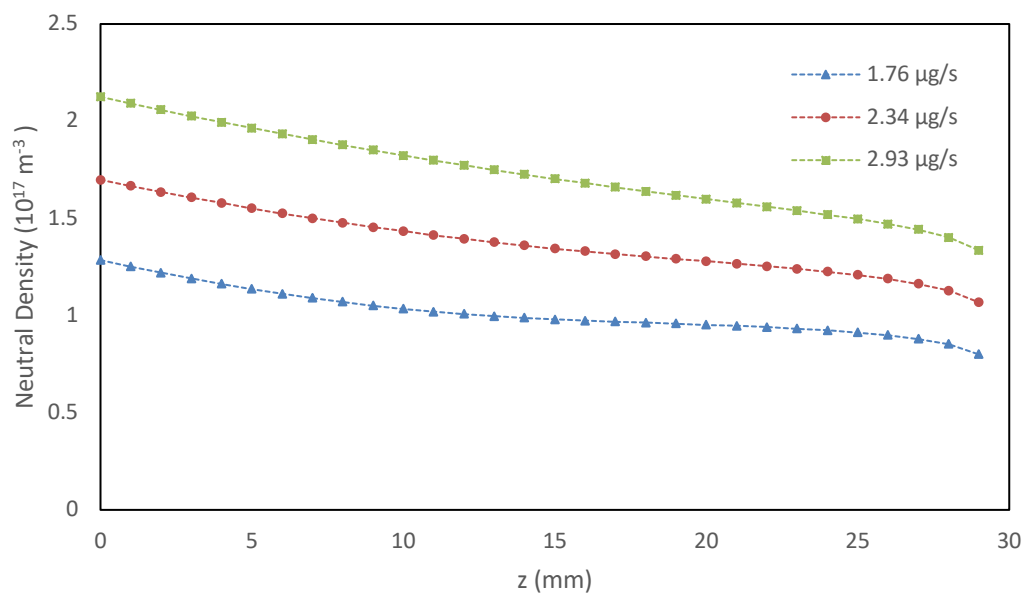


Figure 4.7. Simulated centerline neutral particle density as a function of axial distance from the anode for an operating voltage of 250 V.

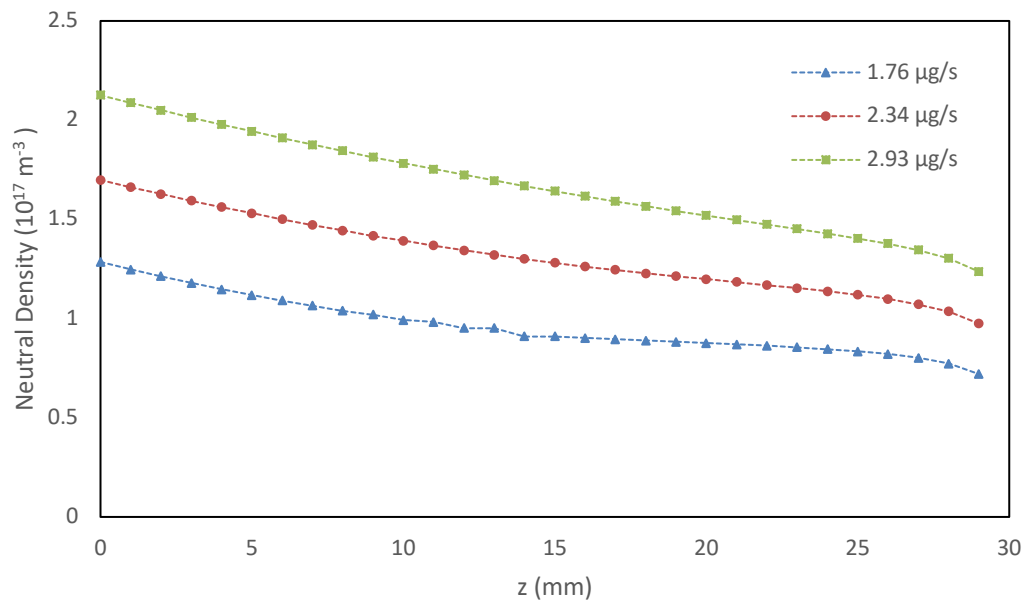


Figure 4.8. Simulated centerline neutral particle density as a function of axial distance from the anode for an operating voltage of 200 V.

Figures 4.6, 4.7, and 4.8 show the neutral particle density at the thruster centerline for the axial distance from the anode at operating voltages of 300 V, 250 V, and 200 V. Experimental values of neutral particle densities are not provided, however since neutral particles are only lost through ionization and diffusion, general plasma characteristics can be inferred from neutral particle densities. Any neutral particle not diffusing in the axial direction is ionized, so the slope corresponds to the local ionization rate.

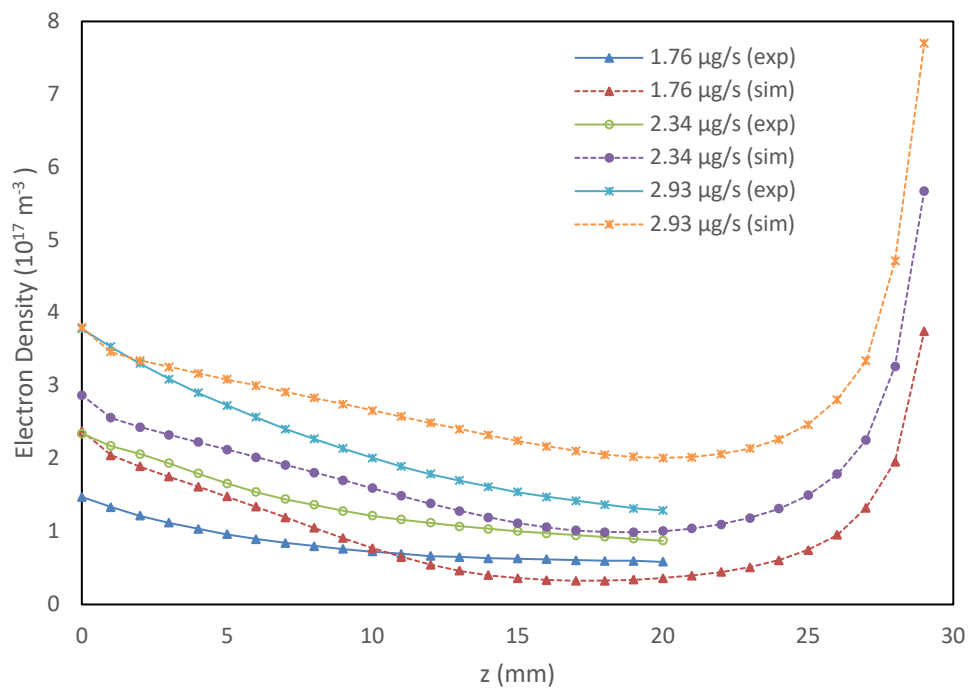


Figure 4.9. Centerline electron densities for experimental and simulation results for an operating voltage of 300 V. Experimental values from [43].

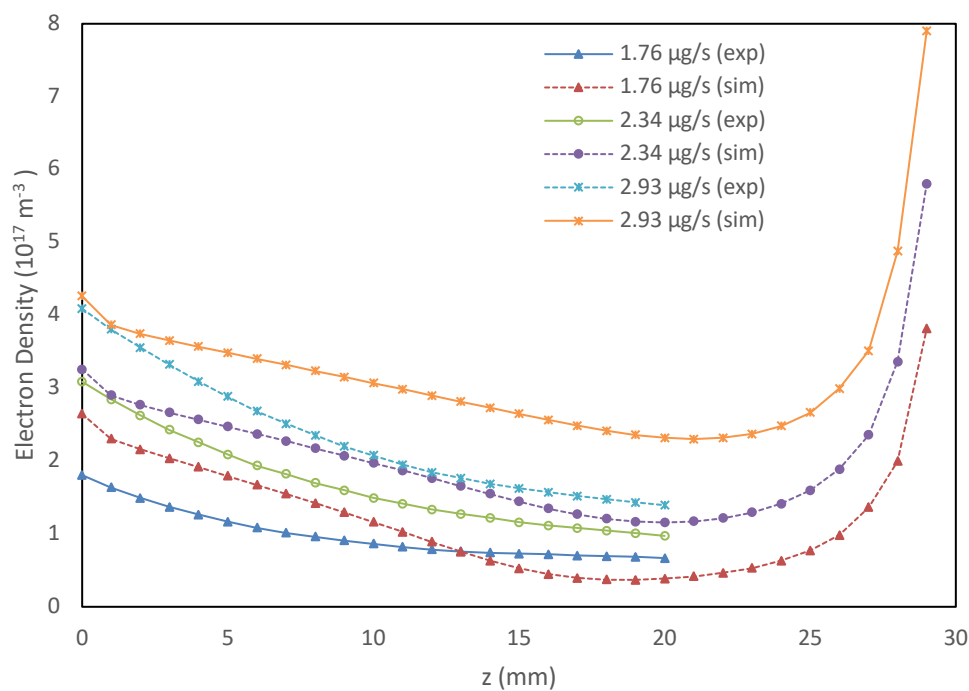


Figure 4.10. Centerline electron densities for experimental and simulation results for an operating voltage of 250 V. Experimental values from [43].

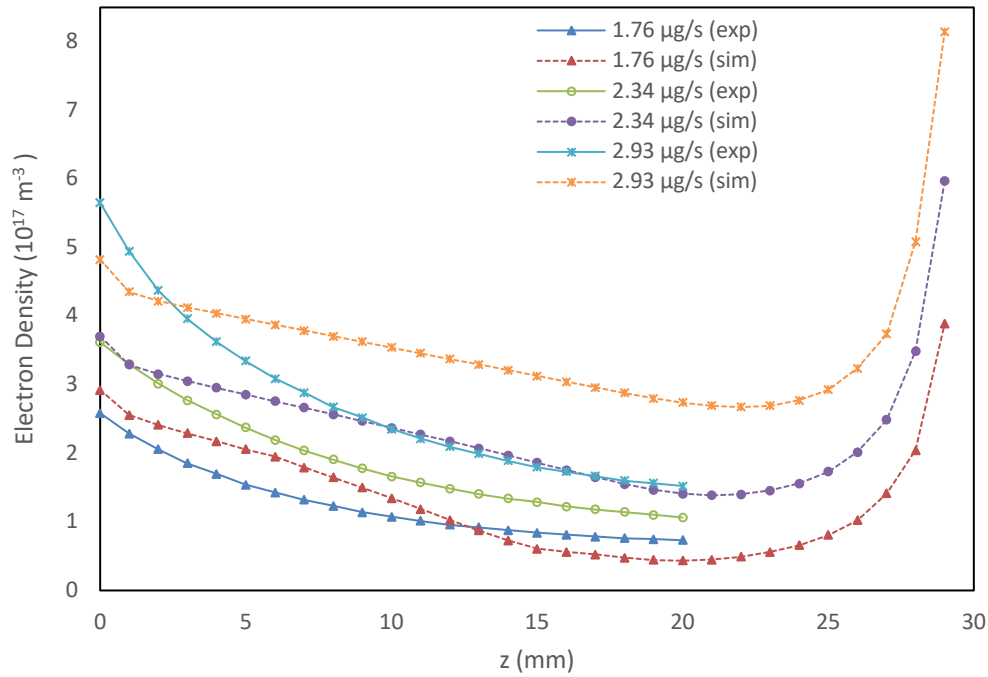


Figure 4.11. Centerline electron densities for experimental and simulation results for an operating voltage of 200 V. Experimental values from [43].

Figures 4.9, 4.10, and 4.11 show the electron density at the thruster centerline for the axial distance from the anode for operating voltages of 300 V, 250 V, and 200 V for experimental (exp) and simulation (sim) results. Error associated with experimental measurements of electron density are not provided in [43]. Experimental values are only provided to a distance of 20 mm from the anode. The general shape of the electron density in both simulation and experiment has the electron density increasing as the electrons flow towards the anode. As electrons diffuse through the chamber, they ionize neutral particles, freeing new electrons in the process. These electrons can then themselves go on to ionize more neutrals, creating a positive feedback loop. This is countered by electron mobility which increases with increasing electron density, although not immediately evident.

From equations 6-8, electron mobility and diffusion at first seems inversely related to electron-neutral collision frequency. However, upon closer inspection the $\left(\frac{\omega_{ce}}{\nu_{en}}\right)^2$ term contributes significantly more. In section 3, from equation 22 it was derived that a Hall thruster with a 0.02 T magnetic field would have a cyclotron frequency $\omega_{ce} = 3.52 * 10^9 \text{ Hz}$. Using equation 23, plugging in an electron temperature of 5 eV and a neutral particle density of $1.5 * 10^{17} \text{ m}^{-3}$, $\nu_{en} = 6.06 * 10^4 \text{ Hz}$. Therefore, $\left(\frac{\omega_{ce}}{\nu_{en}}\right)^2$ is much greater than 1 and electron mobility and diffusion become approximately proportional to ν_{en} as opposed to being inversely proportional. As the electron density increases, mobility increases alongside electron production to an equilibrium value.

While experimental data is not available for comparison near the thruster exit, the high electron density can be explained in a similar manner. Looking at equation 6 describing electron drift velocity, the diffusion term depends on the derivative of electron density with respect to the z-axis. Near the anode, the mobility term is large due to higher rates of electron-neutral collision with the diffusion term negatively contributing to electron motion due to the increasing electron density towards the anode. At the thruster exit, the diffusion term contributes significantly to electron motion due to the large electron density gradient while the mobility term is smaller due to the shallower electric potential in this region.

The electron drift velocity in the direction of the anode at the channel centerline is plotted in figure 4.12 for SOP-1. Equations 6-8 together with equation 23 were used along with the previously specified values for ω_{ce} , σ_{en} , and c_e . The derivative of electron density and electric potential, used to find the electric field, was

found using the finite difference approximation. Notably, the drift velocity has a maximum near the anode and a minimum near the thruster exit. Values of drift velocity at the edges of the domain are likely inaccurate due to errors of the finite difference approximation at boundaries.

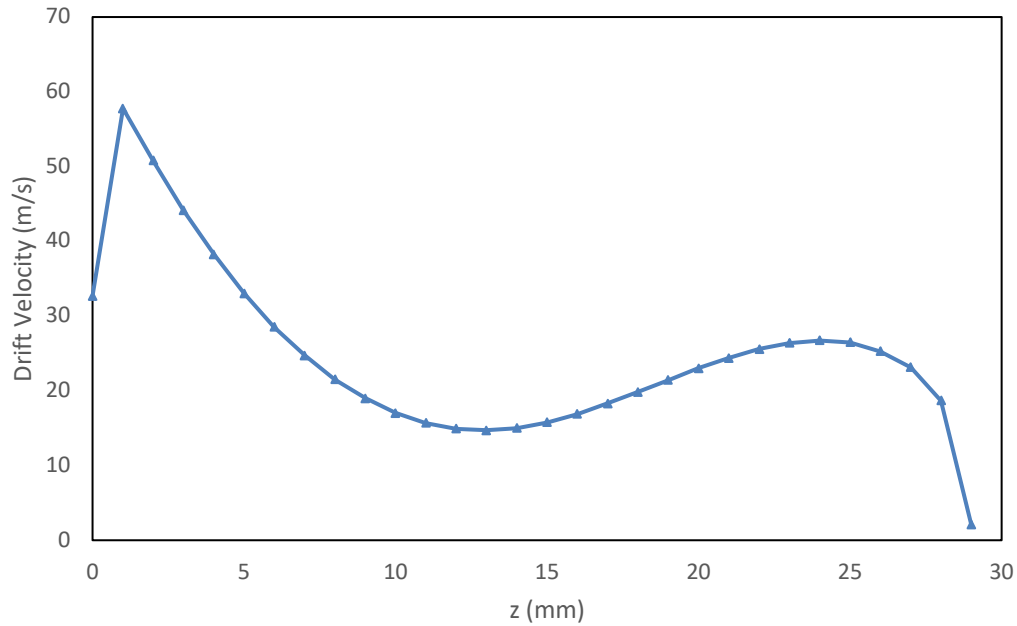


Figure 4.12. Simulated centerline electron drift velocity as a function of axial distance from the anode for standard operating conditions.

The assumptions made in the development of the simulation contributed to discrepancies between simulated and experimental electron densities. Notably, the electron densities in simulated results are generally higher while still following the overall shape of the measured electron density profiles. In chapter 2, it was mentioned that the classical diffusion model was used in the simulation, while other authors used different diffusion models or combinations of several. Many other models include higher order terms which increase electron diffusion, leading to a lower electron population as electrons diffuse out of the system at a faster rate. Similarly, while interactions in the azimuthal direction were ignored due to the assumed azimuthal

symmetry, other authors have constructed two dimensional ϕ -z simulations of Hall thrusters to investigate the impact of azimuthal oscillations on electron transport [40]. Electron oscillations in the azimuthal were shown to increase electron diffusion across magnetic field lines, also leading to a lower electron population. Additionally, without the assumption of a constant electron temperature, many constants such as the neutral atom cross section would vary and quantities such as electron-neutron collision frequency would no longer behave linearly. This can introduce factors that increase or decrease electron diffusion in different regions, giving the electron density profile a different slope. While all of these likely had some impact on the simulation accuracy, they will not be examined further in this discussion.

4.3 Effects of Superparticle Size on Simulation Accuracy

Initial superparticle density	3 per cell	6 per cell	9 per cell	12 per cell	15 per cell
Superparticle size (particles per super)	5.67×10^7	2.83×10^7	1.89×10^7	1.42×10^7	1.13×10^7
Thrust	24.8 mN	29.1 mN	33.2 mN	34.7 mN	34.9 mN
Specific Impulse (I_{sp})	1080 s	1268 s	1446 s	1512 s	1520 s

Table 4.3. Simulation results for varying superparticle size with standard operating parameters.

As mentioned in chapter 3, determining the accuracy of a particle-in-cell simulation is difficult due to the number of factors that contribute to error within the simulation. The structure of the code is not well suited for variable cell size and modifications are needed before such simulations can be executed. Superparticle size was chosen as the factor to be varied in order to test the validity of the code. The base

superparticle size used in the simulation was 1.89×10^7 particles per superparticle, corresponding to 9 superparticles placed in each cell during initialization. Four additional simulations were run with an operating voltage of 300 V and a neutral mass injection rate of $2.34 \mu\text{g/s}$ with superparticle sizes of 5.67×10^7 , 5.67×10^7 , 1.42×10^7 , and 1.13×10^7 particles per superparticle, corresponding to 3, 6, 12, and 15 superparticles per cell during initialization respectively. The results for the simulations and for the base simulation are provided in table 4.3.

As seen in table 4.3, at a large superparticle size the simulation deviates from physical results significantly. This is largely due to nonphysical factors that arise when the number of particles in a given cell is low. If a cell is occupied by only one superparticle, then the electric field that acts on that particle originates from its own charge, which is not possible due to Newton's third law of motion. At smaller superparticle sizes, the calculated thrust more closely matches the experimental thrust of 37.8 mN provided in table 4.2. The thrust converges around 35 mN for 12 and 15 superparticles per cell during initialization. Above 15 superparticles per cell during initialization, the program crashes after several loop iterations and the source of this issue has not been identified.

Ideally, the superparticle size would be selected after determining the ideal size to balance accuracy with computational cost. The simulation was not created with greatly varied superparticle size in mind, and it is not well equipped to handle significantly smaller superparticle sizes than the size used for the base simulation. The base superparticle size of 1.89×10^7 particles per superparticle appears to exist in the range of sizes for which the simulation most closely matches physical values.

5 Conclusion

5.1 Summary

In this project a 2D axisymmetric code with azimuthal symmetry was developed and used to model the SPT-70 Hall thruster. Discrete superparticle treatment of ions was used while neutrals and electrons were treated as continuous fluid distributions. Neutral particles were assumed to diffuse with a uniform velocity and normal angular distribution of that of a xenon cold gas thruster, while electrons were assumed to distribute instantaneously along magnetic field lines according to the Boltzmann relationship. Electron diffusion across magnetic field lines was modeled through classical electron diffusion. The electric potential was found by solving Poisson's equation using an iterative Jacobi solver. The simulation was initialized with an estimation of steady-state values and then continued to loop for 100,000 timesteps at which point the properties of the thruster plasma converged to stable values.

The simulation determines steady-state characteristics of the thruster including thrust, centerline electric potential, centerline neutral particle density, and centerline electron density. From the output of the simulation, specific impulse was determined. Simulation data was compared to experimental data for the SPT-70 Hall thruster for operating voltages of 300 V, 250 V, and 200 V and for neutral mass injection rates of $1.76 \mu\text{g/s}$, $2.34 \mu\text{g/s}$, and $2.93 \mu\text{g/s}$. Thrust and specific impulse for simulated and experimental results were compared for the various operational parameters and comparisons between electric potential and particle densities were investigated.

The simulation code developed approximately modeled the electron transport within the thruster cavity as well as ion acceleration and neutral particle diffusion. Thrust was consistently lower than the experimental value, indicating the simulation is missing a mechanism outside of the system domain or beyond the scope of the model that leads to an increase in thrust. The sensitivity of the simulation to superparticle size was examined, and smaller superparticle sizes lead to a convergence of thrust around 35 mN.

The simulation offers advantages in computational efficiency due to the nature of the programming language. Since C++, built as an extension of C code, is a compiled programming language as opposed to an interpreted one, code can be executed more efficiently than higher level codes. In a PIC simulation, several kilobytes of data are processed and manipulated for each time step, resulting in potentially several billions of calculations being carried out in order to complete a simulation. However, calculations within a PIC code typically consist of simple arithmetic operations, so only basic functions are needed to execute the code. In this simulation, the program stores global arrays for electric potential, electric field, charge density, population densities, and ions as pointer variables, further increasing the speed at which code can be processed at the expense of error handling. Pointer variables are references to addresses in computer memory and allow a program to bypass traditional methods of referencing computer memory which can be inefficient when dealing with large quantities of data. Implementing pointers results in the program being executed in a highly optimized way for simple PIC simulations. The

framework of the code can be implemented into other Hall thruster codes or related simulations for a potential decrease in execution time.

5.2 Future Work

The main priority in further developing the simulation code will be to implement a more complete model of electron transport within a Hall thruster. Electron behavior is difficult to model in Hall thrusters due to the different mechanics of electron motion in the radial, axial, and azimuthal directions. It will be difficult to improve the accuracy of the simulation without taking into account more complicated electron interactions. Since the simulation runs relatively quickly, the computational cost of adding complexity to the model would be insignificant relative to similar simulations.

To make the simulation more applicable to various systems, the ability to handle varied system geometry can be introduced to the program. All of the underlying principles of PIC simulations are still valid when working with cells of different geometry such as triangles, trapezoids, and parallelograms. In its current state, the simulation only works when the magnetic field is uniform and parallel to the cell columns. This doesn't properly model the true magnetic field within a Hall thruster and restricts the domain of the simulation to the cavity where the magnetic field lines are approximately straight. Variable cell sizes would also provide value in allowing investigations into how cell size impacts the accuracy of the simulation.

Segments of the program developed for this project can be applied to similar thruster types such as the gridded ion thruster which operates like a Hall thruster without a magnetic field. If possible, the generalization of this program to simulate a

multitude of electric propulsion systems would provide a powerful open source tool for groups without access to commercial software. Additionally, a generalized simulation program could serve as a platform for the development of more detailed or specialized PIC simulations, saving future developers time in constructing PIC simulations.

Bibliography

- [1] D. M. Goebel and I. Katz, *Fundamentals of Electric Propulsion: Ion and Hall Thrusters* (Jet Propulsion Laboratory, Pasadena, CA, 2008).
- [2] J. J. Szabo, in *Light Metal Propellant Hall Thrusters* (University of Michigan, Ann Arbor, MI, 2009).
- [3] M. S. McDonald, Ph.D dissertation, University of Michigan, 2012.
- [4] U. S. Inan and M. Gołkowski, *Principles of Plasma Physics for Engineers and Scientists* (Cambridge University Press, Cambridge, 2011).
- [5] L. Grush, *SpaceX just launched two of its space internet satellites*, The Verge (2018).
- [6] C. Henry, *SpaceX becomes operator of world's largest commercial satellite constellation with Starlink Launch*, SpaceNews.com (2020).
- [7] D. Mohny, *OneWeb secures \$1.25 billion for global satellite broadband network*, Space IT Bridge (2019).
- [8] J. Porter, *Amazon will launch thousands of satellites to provide internet around the world*, The Verge (2019).
- [9] A. Lavender, *How many satellites orbiting the Earth in 2019?*, Pixalytics Ltd (2019).
- [10] J. W. Dankanich, *Small Satellite Propulsion*, AstroRecon (2015).
- [11] P. O'Dowd and F. Paris, *Possible Satellite Collision Above Pittsburgh A Reminder Of The Risk Of 'Space Junk', Here & Now* (2020).
- [12] A. Vakhrushev, *Numerical modelling of the MHD flow in continuous casting mold by two CFD platforms ANSYS Fluent and OpenFOAM*, (2018).
- [13] C. K. Birdsall and A. B. Langdon, *Plasma Physics via Computer Simulation* (McGraw Hill, New York, 1985).
- [14] R. R. Hofer, I. Katz, I. G. Mikellides, and M. Gamero-Castano, *Hybrid-PIC Computer Simulation of the Plasma and Erosion Processes in Hall Thrusters* (Pasadena, CA, 2010).
- [15] I. Funaki, S. Cho, T. Sano, T. Fukatsu, Y. Tashiro, T. Shiiki, Y. Nakamura, H. Watanabe, K. Kubota, Y. Matsunaga, and K. Fuchigami, *Acta Astronautica* 170, 163 (2020).
- [16] Y. Ding, W. Peng, H. Sun, Y. Xu, L. Wei, H. Li, M. Zeng, F. Wang, and D. Yu, *Physics of Plasmas* 24, 023507 (2017).
- [17] D. Manzella, R. Jankovsky, and R. Hofer, 38th AIAA/ASME/SAE/ASEE Joint Propulsion Conference & Exhibit (2002).
- [18] P. Lascombes and J.-L. Maria, *The Smallest Hall Effect Thruster Designed for Cubesats*.
- [19] C. Birdsall, *IEEE Transactions on Plasma Science* 19, 65 (1991).
- [20] I. J. Morey and R. W. Boswell, *Physics of Fluids B: Plasma Physics* 1, 1502 (1989).
- [21] R. W. Boswell and I. J. Morey, *Applied Physics Letters* 52, 21 (1988).
- [22] A. B. Langdon, *IEEE Transactions on Plasma Science* 42, 1317 (2014).
- [23] A. Revel, S. Mochalsky, I. M. Montellano, D. Wunderlich, U. Fantz, and T. Minea, *Journal of Applied Physics* 122, 103302 (2017).
- [24] J. M. Dawson, *Reviews of Modern Physics* 55, 403 (1983).

- [25] J. A. Byers, Scientific and Technical Aerospace Reports 8, (1969).
- [26] M. Melzani, R. Walder, D. Folini, and C. Winisdoerffer, International Journal of Modern Physics: Conference Series 28, 1460194 (2014).
- [27] T. Takizuka and H. Abe, Journal of Computational Physics 25, 205 (1977).
- [28] V. Vahedi and M. Surendra, Computer Physics Communications 87, 179 (1995).
- [29] K. Ghooos, W. Dekeyser, G. Samaey, P. Börner, D. Reiter, and M. Baelmans, Contributions to Plasma Physics 56, 616 (2016).
- [30] F. Ebadpour and A. Navid, International Journal of Applied Physics and Mathematics 187 (2012).
- [31] A. Abedalmuhdi, B. E. Wells, and K.-I. Nishikawa, 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM) (2017).
- [32] C. Cheng and G. Knorr, Journal of Computational Physics 22, 330 (1976).
- [33] M. Hong and G. A. Emmert, Journal of Applied Physics 78, 6967 (1995).
- [34] Nishikawa Kyōji and M. Wakatani, Plasma Physics: Basic Theory with Fusion Applications (Springer, Berlin, 2000).
- [35] S. O. Macheret, M. N. Shneider, and R. B. Miles, AIAA Journal 40, 74 (2002).
- [36] N. A. Gatsonis and X. Yin, Journal of Propulsion and Power 17, 945 (2001).
- [37] J. Y. Bang and C. W. Chung, Physics of Plasmas 16, 093502 (2009).
- [38] E. Sommer, M. K. Scharfe, N. Gascon, M. A. Cappelli, and E. Fernandez, IEEE Transactions on Plasma Science 35, 1379 (2007).
- [39] Y. Jia, J. Chen, N. Guo, X. Sun, C. Wu, and T. Zhang, Plasma Science and Technology 20, 105502 (2018).
- [40] C. M. Lam, E. Fernandez, and M. A. Cappelli, IEEE Transactions on Plasma Science 43, 86 (2015).
- [41] A. Shashkov, A. Lovtsov, and D. Tomilin, Physics of Plasmas 24, 043501 (2017).
- [42] K. Hara, I. D. Boyd, and V. I. Kolobov, Physics of Plasmas 19, 113508 (2012).
- [43] J. M. Fife, Ph.D Dissertation, Massachusetts Institute of Technology, 1998.
- [44] X. Cao, G. Hang, H. Liu, Y. Meng, X. Luo, and D. Yu, Plasma Science and Technology 19, 105501 (2017).
- [45] E. Sommer, M. K. Scharfe, N. Gascon, M. A. Cappelli, and E. Fernandez, IEEE Transactions on Plasma Science 35, 1379 (2007).
- [46] X.-F. Cao, H. Liu, W.-J. Jiang, Z.-X. Ning, R. Li, and D.-R. Yu, Chinese Physics B 27, 085204 (2018).
- [47] P. Coche and L. Garrigues, Physics of Plasmas 21, 023503 (2014).
- [48] D. V. Schroeder, An Introduction to Thermal Physics (Addison Wesley Longman, San Francisco, CA, 2005).
- [49] D. J. Griffiths, Introduction to Electrodynamics (Pearson, Boston, 2014).
- [50] S. J. Farlow, Partial Differential Equations for Scientists and Engineers (Dover Publications, Inc., New York, 2016).
- [51] E. Ahedo, Martínez-Cerezo P., and Martínez-Sánchez M., Physics of Plasmas 8, 3058 (2001).
- [52] G. Dettleff and M. Grabe, Basics of Plume Impingement Analysis for Small Chemical and Cold Gas Thrusters (Göttingen, 2011).

- [53] K. L. Cartwright and G. A. Radtke, Numerical Uncertainty Estimation for Stochastic Particle-in-Cell Simulations Applied to Verification and Validation (Albuquerque, NM, 2015).
- [54] Journal of Fluids Engineering 130, 078001 (2008).
- [55] T. W. Haag, in *Thrust Stand for High Power Electric Propulsion Devices*, Proceedings of the 25th Joint Propulsion Conference, Cleveland, (1989).

Appendix

```
//=====
=====
// Name      : hybridSimCode.cpp
// Author     : Conner Myers
// Version    : 1.0
// Copyright  : Creative Commons License
// Description : Hybrid-PIC Simulation for SPT-70 Hall
// Thruster
//=====
=====

#include <iostream>
using namespace std;
#include <random>
#include <math.h>

// Size of spatial grid - In this problem each node is
// separated by dr, dz
const int rNodes = 16; //16
const int zNodes = 30; //30
const double dr = 1;
const double dz = 1;
const double rin = 20;
const double dt = 1; // 10 ns

// Number of superparticles placed in each cell during
// initialization
const int numParticlesPerCell = 9;
double particlesPerSuper = (1.7 * pow(10.0, 8.0)) /
numParticlesPerCell;

const int maxNumberIons = 5000 * numParticlesPerCell;

// Calibration factors
const double neutralInjectionCal = 1;
const double phiCal = 1;

// Number of time steps to be iterated over (10 ns each)
const int timeSteps = 100000;

// INITIALIZATION
```

```

// *****
// *****
// *****
// *****

// Declare rho (charge density) array and pointer array for
rho
double rho[zNodes][rNodes];
double *ptrRho[zNodes][rNodes];

// Initializes ptrRho array, pointing to values for rho. Sets
initial values of rho to zero
void initPtrRho() {
    for (int i = 0; i < zNodes; i++) {
        for (int j = 0; j < rNodes; j++) {
            ptrRho[i][j] = &rho[i][j];
            *ptrRho[i][j] = 0;
        }
    }
    cout << "initPtrRho successful" << endl;
}

//Declare phi (electric potential) array and pointer array for
phi
double phi[zNodes][rNodes];
double *ptrPhi[zNodes][rNodes];

// Initializes ptrPhi array, pointing to values for phi. Sets
initial values of phi to 300-(10*i) Volts
void initPtrPhi() {
    for (int i = 0; i < zNodes; i++) {
        for (int j = 0; j < rNodes; j++) {
            ptrPhi[i][j] = &phi[i][j];
            *ptrPhi[i][j] = (300 - 10 * i) * phiCal;
            // Note: 29 cells, 300V at left end, 10 V at
right end
        }
    }
    cout << "initPtrPhi successful" << endl;
}

//Declare eField (electric field - this time a vector with 2
values) array and pointer array for eField
double eField[zNodes][rNodes][2];

```

```

double *ptrEField[zNodes][rNodes][2];

// Initializes ptrEField array, pointing to values for eField.
Doesn't calculate initial values
void initPtrEField() {
    for (int i = 0; i < zNodes; i++) {
        for (int j = 0; j < rNodes; j++) {
            for (int k = 0; k < 2; k++) {
                ptrEField[i][j][k] = &eField[i][j][k];
            }
        }
    }
    cout << "initPtrEField successful" << endl;
}

// Initializes EField by differencing electric potential. Use
if conditions for boundary cases
void setInitEField() {
    for (int i = 0; i < zNodes; i++) {
        for (int j = 0; j < rNodes; j++) {
            *ptrEField[i][j][1] = 0;
            if (i == 0) {
                *ptrEField[i][j][0] = -(*ptrPhi[i +
1][j] - *ptrPhi[i][j])
                                / (0.001);
            } else if (i == (zNodes - 1)) {
                *ptrEField[i][j][0] = -(*ptrPhi[i][j] -
*ptrPhi[i - 1][j])
                                / (0.001);
            } else {
                *ptrEField[i][j][0] = -(*ptrPhi[i +
1][j] - *ptrPhi[i - 1][j])
                                / (0.002);
            }
        }
    }
    cout << "setInitEField successful" << endl;
}

// Declare ion superparticle array and pointers to ions
double ionArray[maxNumberIons][4];
double *ptrIonArray[maxNumberIons][4];

```

```

// Initializes ptrIonArray, pointing to values for ions, sets
position values to -1.-1 (out of bounds)
void initPtrIonArray() {
    for (int i = 0; i < maxNumberIons; i++) {
        for (int j = 0; j < 4; j++) {
            ptrIonArray[i][j] = &ionArray[i][j];
        }
        *ptrIonArray[i][0] = -1.0;
        *ptrIonArray[i][1] = -1.0;
    }
    cout << "initPtrIonArray successful" << endl;
}

// Global seed variable for random number generator in
setInitIons()
int seed = 36;

// Sets initial values for initial ion superparticles (random
RZ pos within cell, Z- vel, 0 R- vel)
// Initial Z- vel models linear acceleration from 0 to 17 km/s
at exhaust
void setInitIons() {
    int ionParticle = 0;
    default_random_engine e(seed);
    uniform_real_distribution<float> n(0, 1);
    for (int i = 0; i < zNodes - 1; i++) {
        for (int j = 0; j < rNodes - 1; j++) {
            for (int k = 0; k < numParticlesPerCell; k++)
            {
                *ptrIonArray[ionParticle][0] = n(e) + i;
                *ptrIonArray[ionParticle][1] = n(e) + j;
                *ptrIonArray[ionParticle][2] = (0.17 /
29.0)
                *
                (*ptrIonArray[ionParticle][0]);
                *ptrIonArray[ionParticle][3] = 0;
                ionParticle += 1;
            }
        }
    }
    seed += 1;
    cout << "setInitIons successful" << endl;
}

```

```

// Declare electron density points at the center of each cell
(-> 1 less point than grid pts)
double electrons[zNodes][rNodes];
double *ptrElectrons[zNodes][rNodes];

// Sets initial density for electrons, corresponding to 1 *
xenon superparticles (charge of Xe)
void initPtrElectrons() {
    for (int i = 0; i < zNodes; i++) {
        for (int j = 0; j < rNodes; j++) {
            ptrElectrons[i][j] = &electrons[i][j];
            *ptrElectrons[i][j] = ((zNodes - 1) * (rNodes
- 1)
                                * numParticlesPerCell) / (zNodes *
rNodes);
        }
    }
    cout << "initPtrElectrons successful" << endl;
}

// Declare neutral particle densities for each cell (-> 1 less
point than grid pts, like electrons)
double neutrals[zNodes][rNodes];
double *ptrNeutrals[zNodes][rNodes];

// Sets initial density for neutrals, using initial data
(density neutrals ~= 188 * density ions)
void initPtrNeutrals() {
    for (int i = 0; i < zNodes; i++) {
        for (int j = 0; j < rNodes; j++) {
            ptrNeutrals[i][j] = &neutrals[i][j];
            *ptrNeutrals[i][j] = 188
                                * (1 - (static_cast<double>(i) / (2
* zNodes - 2)))
                                * numParticlesPerCell *
neutralInjectionCal;
        }
    }
    cout << "initPtrNeutrals successful" << endl;
}

// Set variable and pointer to store partial ion values
double remainderIons[zNodes - 1][rNodes - 1];
double *ptrRemainderIons[zNodes - 1][rNodes - 1];

```

```

// Sets remainderIons values to zero and initializes pointer
variable
void initPtrRemainderIons() {
    for (int i = 0; i < zNodes - 1; i++) {
        for (int j = 0; j < rNodes - 1; j++) {
            ptrRemainderIons[i][j] = &remainderIons[i][j];
            *ptrRemainderIons[i][j] = 0;
        }
    }
    cout << "initPtrRemainderIons successful" << endl;
}

// Declare variable to store thrust at each step
double thrust;
double *ptrThrust = &thrust;
double outThrust;
double *ptrOutThrust = &outThrust;

// LOOP FUNCTIONS
// *****
// *****
// *****
// *****

// Computes Rho for each node using linear (1st order) scatter
operation. Only works for
// particles that are within bounds
void computeRho() {
    // First clear ptrRho
    for (int i = 0; i < zNodes; i++) {
        for (int j = 0; j < rNodes; j++) {
            *ptrRho[i][j] = 0;
        }
    }
    // Distribute ion charges to grid points
    for (int i = 0; i < maxNumberIons; i++) {
        if ((*ptrIonArray[i][0] >= 0) &&
            (*ptrIonArray[i][1] >= 0)
            && (*ptrIonArray[i][0] <= 29) &&
            (*ptrIonArray[i][1] <= 15)) {
            int zBasePos = (int) *ptrIonArray[i][0];
            int rBasePos = (int) *ptrIonArray[i][1];

```

```

        double zh = *ptrIonArray[i][0] - (double)
zBasePos;
        double rh = *ptrIonArray[i][1] - (double)
rBasePos;
        *ptrRho[zBasePos][rBasePos] += (1 - zh) * (1 -
rh);
        *ptrRho[zBasePos][rBasePos + 1] += (1 - zh) *
(rh);
        *ptrRho[zBasePos + 1][rBasePos] += (zh) * (1 -
rh);
        *ptrRho[zBasePos + 1][rBasePos + 1] += (zh) *
(rh);
    }
}

```

```

// Declare Epsilon0 for use in computePhiIons (Calculated in
base units of superparticles)
// eps0inv depends on the initial number of particles in each
cell
// since the initial ion density is independent of
superparticle size
double eps0inv = 30.773 / (1.8125 * numParticlesPerCell);

void computePhi() {
    // INITIALIZATION: DECLARE TEMPORARY VALUES FROM POINTERS
    // *****
    // Declare temporary electron points and potential for
calculations;
    double mobileElectrons[zNodes][rNodes];
    double phiValue[zNodes][rNodes];
    double newPhiValue[zNodes][rNodes];
    for (int i = 0; i < zNodes; i++) {
        for (int j = 0; j < rNodes; j++) {
            mobileElectrons[i][j] = *ptrElectrons[i][j];
            phiValue[i][j] = *ptrPhi[i][j];
        }
    }
    // *****
    // FIRST STEP: DISTRIBUTE ELECTRONS USING THE BOLTZMANN
RELATIONSHIP
    // *****
    // Declare normalization factor and total electrons for
each column

```

```

double normFactorElectrons[zNodes];
double totalColumnElectrons[zNodes];
double averagePhiValue[zNodes];
// Compute averagePhiValue here, used below
for (int i = 0; i < zNodes; i++) {
    averagePhiValue[i] = 0;
    for (int j = 0; j < rNodes; j++) {
        averagePhiValue[i] += phiValue[i][j];
    }
    averagePhiValue[i] = (1) * averagePhiValue[i] /
(float) zNodes;
}
// Store the exponential of potential in an array, used
for Boltzmann distribution of electrons (assume T=5eV)
double expPhi[zNodes][rNodes];
for (int i = 0; i < zNodes; i++) {
    for (int j = 0; j < rNodes; j++) {
        expPhi[i][j] = exp((phiValue[i][j] -
averagePhiValue[i]) / 5);
    }
}
// Store values for total electrons and total exponential
terms in each column (clearing at the start of each sum)
for (int i = 0; i < zNodes; i++) {
    normFactorElectrons[i] = 0;
    totalColumnElectrons[i] = 0;
    for (int j = 0; j < rNodes; j++) {
        normFactorElectrons[i] += expPhi[i][j];
        totalColumnElectrons[i] +=
mobileElectrons[i][j];
    }
}
// Proportionally redistribute electrons based on the
exponential of potential at each point (normalized)
for (int i = 0; i < zNodes; i++) {
    for (int j = 0; j < rNodes; j++) {
        mobileElectrons[i][j] = (1)
            * (totalColumnElectrons[i] *
expPhi[i][j])
            / normFactorElectrons[i];
    }
}
// *****

```



```

// SECOND STEP: CALCULATE NEW POTENTIAL AT EACH GRID
POINT USING NEW ELECTRON POSITIONS
// *****
// Declare and find values for electric potential due to
charge density
double phiFromRho[zNodes][rNodes];
for (int i = 0; i < zNodes; i++) {
    for (int j = 0; j < rNodes; j++) {
        phiFromRho[i][j] = (*ptrRho[i][j] -
mobileElectrons[i][j])
                                * eps0inv;
    }
}
// First apply Dirichlet boundary condition at inlet
(redundant step just in case)
for (int j = 0; j < rNodes; j++) {
    newPhiValue[0][j] = 300 * phiCal;
}
// Find Phi for internal grid points second
for (int i = 1; i < zNodes - 1; i++) {
    for (int j = 1; j < rNodes - 1; j++) {
        newPhiValue[i][j] = (phiFromRho[i][j]
                                + (phiValue[i][j + 1] +
phiValue[i][j - 1]) / (dr * dr)
                                + (phiValue[i][j + 1] -
phiValue[i][j - 1])
                                                / (2 * dr * rin * (dr +
j))
                                + (phiValue[i - 1][j] + phiValue[i
+ 1][j]) / (dz * dz))
                                / ((2 / (dr * dr)) + (2 / (dz *
dz)));
    }
}
// Then apply the Neumann boundary condition at the top
(dphi/dr = 0)
for (int i = 1; i < zNodes - 1; i++) {
    newPhiValue[i][0] = (phiFromRho[i][0] + (2 *
phiValue[i][1]) / (dr * dr)
                                + (phiValue[i - 1][0] + phiValue[i +
1][0]) / (dz * dz))
                                / ((2 / (dr * dr)) + (2 / (dz * dz)));
}

```

```

    // Next apply the Neumann boundary condition at the
    bottom ( $d\phi/dr = 0$ )
    for (int i = 1; i < zNodes - 1; i++) {
        newPhiValue[i][rNodes - 1] = (phiFromRho[i][rNodes
- 1]
            + (2 * phiValue[i][rNodes - 2]) / (dr *
dr)
            + (phiValue[i - 1][rNodes - 1] +
phiValue[i + 1][rNodes - 1])
                / (dz * dz)) / ((2 / (dr *
dr)) + (2 / (dz * dz)));
    }
    // After that, apply the Neumann boundary condition at
    the exit ( $d\phi/dz = 0$ )
    for (int j = 1; j < rNodes - 1; j++) {
        newPhiValue[zNodes - 1][j] = (phiFromRho[zNodes -
1][j]
            + (phiValue[zNodes - 1][j + 1] +
phiValue[zNodes - 1][j - 1])
                / (dr * dr)
            + (phiValue[zNodes - 1][j + 1] -
phiValue[zNodes - 1][j - 1])
                / (2 * dr * rin * (dr + j))
            + (2 * phiValue[zNodes - 2][j]) / (dz *
dz))
            / ((2 / (dr * dr)) + (2 / (dz * dz)));
    }
    // Lastly, the upper and lower right corners of the
    domain are subject to double Neumann boundary conditions
    newPhiValue[zNodes - 1][0] = (phiFromRho[zNodes - 1][0]
        + (2 * phiValue[zNodes - 1][1]) / (dr * dr)
        + (2 * phiValue[zNodes - 2][0]) / (dz * dz))
        / ((2 / (dr * dr)) + (2 / (dz * dz)));
    newPhiValue[zNodes - 1][rNodes - 1] = (phiFromRho[zNodes
- 1][rNodes - 1]
        + (2 * phiValue[zNodes - 1][rNodes - 2]) / (dr
* dr)
        + (2 * phiValue[zNodes - 2][rNodes - 1]) / (dz
* dz))
        / ((2 / (dr * dr)) + (2 / (dz * dz)));
    // Then, set phiValue to newPhiValue for the next
    iteration
    for (int i = 0; i < zNodes; i++) {
        for (int j = 0; j < rNodes; j++) {

```

```

        phiValue[i][j] = newPhiValue[i][j];
    }
}
// *****
// FINAL STEP: STORE FOUND VALUES OF PHI AND ELECTRONS IN
// POINTER VARIABLES
// *****
for (int i = 0; i < zNodes; i++) {
    for (int j = 0; j < rNodes; j++) {
        *ptrElectrons[i][j] = mobileElectrons[i][j];
        *ptrPhi[i][j] = newPhiValue[i][j];
    }
}

}

// Compute EField from the derivative of Phi (using finite
// difference)
void computeEField() {
    // First find the z component of the EField
    for (int i = 0; i < zNodes; i++) {
        for (int j = 0; j < rNodes; j++) {
            *ptrEField[i][j][1] = 0;
            if (i == 0) {
                *ptrEField[i][j][0] = -(*ptrPhi[i +
1][j] - *ptrPhi[i][j])
                    / (0.001);
            } else if (i == (zNodes - 1)) {
                *ptrEField[i][j][0] = -(*ptrPhi[i][j] -
*ptrPhi[i - 1][j])
                    / (0.001);
            } else {
                *ptrEField[i][j][0] = -(*ptrPhi[i +
1][j] - *ptrPhi[i - 1][j])
                    / (0.002);
            }
        }
    }
    // Then the r component of the EField
    for (int i = 0; i < zNodes; i++) {
        for (int j = 0; j < rNodes; j++) {
            if (j == 0) {
                *ptrEField[i][j][1] = -(*ptrPhi[i][j +
1] - *ptrPhi[i][j])
                    / (0.001);
            }
        }
    }
}

```

```

        } else if (j == rNodes - 1) {
            *ptrEField[i][j][1] = -(*ptrPhi[i][j] -
*ptrPhi[i][j - 1])
                                / (0.001);
        } else {
            *ptrEField[i][j][1] = -(*ptrPhi[i][j +
1] - *ptrPhi[i][j - 1])
                                / (0.002);
        }
    }
}
}

```

```

void moveParticles() {
    double avgPhiAtExit;
    double totalPhiAtExit = 0;
    *ptrThrust = 0;
    for (int j = 0; j < rNodes; j++) {
        totalPhiAtExit += *ptrPhi[zNodes - 1][j];
    }
    avgPhiAtExit = totalPhiAtExit / rNodes;
    for (int i = 0; i < maxNumberIons; i++) {
        if (*ptrIonArray[i][0] != -1.0 &&
*ptrIonArray[i][1] != -1.0) {
            // First interpolate Efield at each particle
            double eFieldAtIon[2] = { 0, 0 };
            int zBasePos = (int) *ptrIonArray[i][0];
            int rBasePos = (int) *ptrIonArray[i][1];
            double zh = *ptrIonArray[i][0] - (double)
zBasePos;
            double rh = *ptrIonArray[i][1] - (double)
rBasePos;
            for (int k = 0; k < 2; k++) {
                eFieldAtIon[k] =
*ptrEField[zBasePos][rBasePos][k] * (1 - zh)
                                * (1 - rh)
                                +
*ptrEField[zBasePos][rBasePos + 1][k] * (1 - zh)
                                * (rh)
                                + *ptrEField[zBasePos +
1][rBasePos][k] * (zh)
                                * (1 - rh)
                                + *ptrEField[zBasePos +
1][rBasePos + 1][k] * (zh)

```

```

        * (rh);
    }
    // Accelerate Velocity through  $v = v\_0 +$ 
     $(q/m) * E * dt$ , move position with updated velocity
    // With E in V/m (equivalently N/C), time = 10
    ns ( $=dt$ ),  $q/m = 7.346 * 10^{-8}$ 
    for (int k = 0; k < 2; k++) {
        *ptrIonArray[i][k + 2] += eFieldAtIon[k]
* 7.346
        * pow(10.0, -8.0) * dt;
        *ptrIonArray[i][k] += *ptrIonArray[i][k
+ 2] * dt;
    }
    // Then check to see if the particle is out of
    bounds, either reflect or absorb particles
    // Recall that we store unused particles at (-
    1,-1), so dont count those
    if (*ptrIonArray[i][0] < 0) {
        *ptrIonArray[i][0] = -1 *
(*ptrIonArray[i][0]);
        *ptrIonArray[i][2] = -1 *
(*ptrIonArray[i][2]);
    } else if (*ptrIonArray[i][1] > (double)
(rNodes - 1)) {
        *ptrIonArray[i][1] -=
(*ptrIonArray[i][1]
        - (double) (rNodes - 1));
        *ptrIonArray[i][3] = -1 *
(*ptrIonArray[i][3]);
    } else if (*ptrIonArray[i][1] < 0) {
        *ptrIonArray[i][1] = -1 *
(*ptrIonArray[i][1]);
        *ptrIonArray[i][3] = -1 *
(*ptrIonArray[i][3]);
    } else if (*ptrIonArray[i][0] > (double)
(zNodes - 1)) {
        // If ions leave the domain at the exit,
        accelerate the particles in current direction
        double twoKEoverM = 2 * avgPhiAtExit /
(1.223 * pow(10.0, 8));
        // Weight acceleration in z-dir by
 $v\_z/v\_total$ ;  $\Delta v = \sqrt{2\Delta KE/m}$ ;  $\Delta v$  given in terms of c,  $*10^5$ 
        *ptrIonArray[i][2] +=
(*ptrIonArray[i][2]

```

```

                                / (*ptrIonArray[i][3] +
*ptrIonArray[i][2]))
                                * sqrt(twoKEoverM) * 3 *
pow(10.0, 3.0);
                                *ptrThrust += *ptrIonArray[i][2] *
(particlesPerSuper)
                                * (2.18 * pow(10.0, -12.0));
                                // After tallying the particle into the
thrust, remove from domain for later use
                                *ptrIonArray[i][0] = -1.0;
                                *ptrIonArray[i][1] = -1.0;
                                *ptrIonArray[i][2] = 0;
                                *ptrIonArray[i][3] = 0;
                                // Finally, check if reflected particles
are out of bounds
                                if ((*ptrIonArray[i][0] < 0) ||
(*ptrIonArray[i][0] > 30)
                                || (*ptrIonArray[i][1] < 0)
                                || (*ptrIonArray[i][1] > 16))
{
                                *ptrIonArray[i][0] = -1.0;
                                *ptrIonArray[i][1] = -1.0;
                                *ptrIonArray[i][2] = 0;
                                *ptrIonArray[i][3] = 0;
                                }
                                }
                                }
                                }
                                }

// Constants used in the diffuse electrons function, based on
0.02 T B-field and 5 eV electrons
double omegaCE = 3.52 * pow(10.0, 9.0);

// Diffuse Electrons using drift diffusion approximation
void diffuseElectrons() {
    // Declare arrays used in computations
    double dneAxial[zNodes][rNodes];
    double nuEN[zNodes][rNodes];
    double omegaDnu2[zNodes][rNodes];
    double diffusion[zNodes][rNodes];
    double mobility[zNodes][rNodes];
    double driftVel[zNodes][rNodes];
    double initialElectrons[zNodes][rNodes];

```

```

    double finalElectrons[zNodes][rNodes];
    double deltaElectrons[zNodes][rNodes];
    // Compute the derivative of neutral particle density at
each point
    for (int i = 1; i < zNodes - 1; i++) {
        for (int j = 0; j < rNodes; j++) {
            dneAxial[i][j] = (particlesPerSuper)
                * (*ptrElectrons[i + 1][j] -
*ptrElectrons[i - 1][j])
                / (2 * dz * pow(10.0, -3.0));
        }
    }
    for (int j = 0; j < rNodes; j++) {
        dneAxial[0][j] = (particlesPerSuper)
            * (*ptrElectrons[1][j] -
*ptrElectrons[0][j])
            / (dz * pow(10.0, -3.0));
        dneAxial[zNodes - 1][j] = (particlesPerSuper)
            * (*ptrElectrons[zNodes - 1][j] -
*ptrElectrons[zNodes - 2][j])
            / (dz * pow(10.0, -3.0));
    }
    // Compute electron neutral collision frequency at each
points and (omegaCE/NuEN)^2
    // Multiply number of neutral particles (not
superparticles) by NuEN constant in mm^-3 units
    for (int i = 0; i < zNodes; i++) {
        for (int j = 0; j < rNodes; j++) {
            nuEN[i][j] = (particlesPerSuper) *
(*ptrNeutrals[i][j]) * 4.039
            * pow(10.0, -4.0);
            omegaDNu2[i][j] = (omegaCE / nuEN[i][j]) *
(omegaCE / nuEN[i][j]);
        }
    }
    // Compute Diffusion coefficient and electron mobility at
each point
    for (int i = 0; i < zNodes; i++) {
        for (int j = 0; j < rNodes; j++) {
            mobility[i][j] = (1.759 * pow(10.0, 11))
                * (1 / (nuEN[i][j] * (1 +
omegaDNu2[i][j]))));
            diffusion[i][j] = (8.794 * pow(10.0, 11))

```

```

        * (1 / (nuEN[i][j] * (1 +
omegaDNu2[i][j]))));
    }
}
// Calculate the drift velocity of electrons at each
point
for (int i = 0; i < zNodes; i++) {
    for (int j = 0; j < rNodes; j++) {
        driftVel[i][j] = -(ptrEField[i][j][0] *
mobility[i][j]) * (15.0)
        - (diffusion[i][j] *
dneAxial[i][j]) * (15.0)
        / (ptrElectrons[i][j] *
particlesPerSuper);
    }
}
// Set temporary variables for electrons used in
calculating electron diffusion
for (int i = 0; i < zNodes; i++) {
    for (int j = 0; j < rNodes; j++) {
        initialElectrons[i][j] = ptrElectrons[i][j];
        finalElectrons[i][j] = ptrElectrons[i][j];
    }
}
// Use drift velocity to move electrons everywhere except
thruster exit and anode
for (int i = 1; i < zNodes - 1; i++) {
    for (int j = 0; j < rNodes; j++) {
        if (driftVel[i][j] <= 0.0) {
            deltaElectrons[i][j] =
initialElectrons[i][j] * -driftVel[i][j]
            * pow(10.0, -5.0);
            if (deltaElectrons[i][j] >
initialElectrons[i][j]) {
                finalElectrons[i - 1][j] +=
initialElectrons[i][j];
                finalElectrons[i][j] -=
initialElectrons[i][j];
            } else {
                finalElectrons[i - 1][j] +=
deltaElectrons[i][j];
                finalElectrons[i][j] -=
deltaElectrons[i][j];
            }
        }
    }
}

```



```

        } else {
            deltaElectrons[i][j] =
initialElectrons[i][j] * driftVel[i][j]
                        * pow(10.0, -5.0);
            if (deltaElectrons[i][j] >
initialElectrons[i][j]) {
                finalElectrons[i + 1][j] +=
initialElectrons[i][j];
                finalElectrons[i][j] -=
initialElectrons[i][j];
            } else {
                finalElectrons[i + 1][j] +=
deltaElectrons[i][j];
                finalElectrons[i][j] -=
deltaElectrons[i][j];
            }
        }
    }
    // At the Anode, diffusing electrons complete the circuit
and are removed from the system
    for (int j = 0; j < rNodes; j++) {
        if (driftVel[0][j] <= 0.0) {
            deltaElectrons[0][j] = initialElectrons[0][j]
* -driftVel[0][j]
                        * pow(10.0, -5.0);
            if (deltaElectrons[0][j] >
initialElectrons[0][j]) {
                finalElectrons[0][j] -=
initialElectrons[0][j];
            } else {
                finalElectrons[0][j] -=
deltaElectrons[0][j];
            }
        } else {
            deltaElectrons[0][j] = initialElectrons[0][j]
* driftVel[0][j]
                        * pow(10.0, -5.0);
            if (deltaElectrons[0][j] >
initialElectrons[0][j]) {
                finalElectrons[1][j] +=
initialElectrons[0][j];
                finalElectrons[0][j] -=
initialElectrons[0][j];
            }
        }
    }

```

```

        } else {
            finalElectrons[1][j] +=
deltaElectrons[0][j];
            finalElectrons[0][j] -=
deltaElectrons[0][j];
        }
    }
    // At thruster exit, electrons are introduced through
hollow cathode supplying electrons
    for (int j = 0; j < rNodes; j++) {
        // First diffuse electrons just like before
        if (driftVel[zNodes - 1][j] <= 0.0) {
            deltaElectrons[zNodes - 1][j] =
initialElectrons[zNodes - 1][j]
                * -driftVel[zNodes - 1][j] *
pow(10.0, -5.0);
            if (deltaElectrons[zNodes - 1][j]
                > initialElectrons[zNodes - 1][j])
{
                finalElectrons[zNodes - 2][j] +=
                    initialElectrons[zNodes -
1][j];
                finalElectrons[zNodes - 1][j] -=
                    initialElectrons[zNodes -
1][j];
            } else {
                finalElectrons[zNodes - 2][j] +=
deltaElectrons[zNodes - 1][j];
                finalElectrons[zNodes - 1][j] -=
deltaElectrons[zNodes - 1][j];
            }
        } else {
            deltaElectrons[zNodes - 1][j] =
initialElectrons[zNodes - 1][j]
                * driftVel[zNodes - 1][j] *
pow(10.0, -5.0);
            if (deltaElectrons[zNodes - 1][j]
                > initialElectrons[zNodes - 1][j])
{
                finalElectrons[zNodes - 1][j] -=
                    initialElectrons[zNodes -
1][j];
            } else {

```

```

        finalElectrons[zNodes - 1][j] -=
deltaElectrons[zNodes - 1][j];
    }
}
// Then introduce new electrons from 2.2 Amp
current
// Remember to normalize the current based on
simulation area / total area
    finalElectrons[zNodes - 1][j] += 0.0933
        * (pow(neutralInjectionCal, 2.5))
        * (double) numParticlesPerCell;
}
for (int i = 0; i < zNodes; i++) {
    for (int j = 0; j < rNodes; j++) {
        finalElectrons[i][j] *= 0.9
            + 0.1 * exp(-(double) (i) / 29) /
0.63212;

    }
}
// Finally, set the global values for electrons to
computed final electrons
for (int i = 0; i < zNodes; i++) {
    for (int j = 0; j < rNodes; j++) {
        *ptrElectrons[i][j] = finalElectrons[i][j];
    }
}
}

// Neutral Drift Vel, 274.4 m/s -> 0.002744 mm/(10ns)
double neutralDriftVel = 0.002774;

// Diffuse neutrals
void diffuseNeutrals() {
    // Declare variables used in computing neutral diffusion
    double initialNeutrals[zNodes][rNodes];
    double finalNeutrals[zNodes][rNodes];
    // Initialize temporary variables
    for (int i = 0; i < zNodes; i++) {
        for (int j = 0; j < rNodes; j++) {
            initialNeutrals[i][j] = *ptrNeutrals[i][j];
            finalNeutrals[i][j] = *ptrNeutrals[i][j];
        }
    }
}

```

```

// Calculate diffusion for internal points
for (int i = 1; i < zNodes - 1; i++) {
    for (int j = 1; j < rNodes - 1; j++) {
        // 97.88% of drift velocity is straight ahead
        finalNeutrals[i + 1][j] +=
initialNeutrals[i][j] * neutralDriftVel
        * 0.9788;
        finalNeutrals[i + 1][j + 1] +=
initialNeutrals[i][j]
        * neutralDriftVel * 0.0106;
        finalNeutrals[i + 1][j - 1] +=
initialNeutrals[i][j]
        * neutralDriftVel * 0.0106;
        finalNeutrals[i][j] -= initialNeutrals[i][j] *
neutralDriftVel;
    }
}
// At top and bottom, neutral particles reflect of
thruster walls
for (int i = 1; i < zNodes - 1; i++) {
    finalNeutrals[i + 1][0] += initialNeutrals[i][0] *
neutralDriftVel
        * 0.9894;
    finalNeutrals[i + 1][1] += initialNeutrals[i][0] *
neutralDriftVel
        * 0.0106;
    finalNeutrals[i + 1][rNodes - 1] +=
initialNeutrals[i][rNodes - 1]
        * neutralDriftVel * 0.9894;
    finalNeutrals[i + 1][rNodes - 2] +=
initialNeutrals[i][rNodes - 1]
        * neutralDriftVel * 0.0106;
    finalNeutrals[i][0] -= initialNeutrals[i][0] *
neutralDriftVel;
    finalNeutrals[i][rNodes - 1] -=
initialNeutrals[i][rNodes - 1]
        * neutralDriftVel;
}
// At the thruster exist neutral particles leave the
domain
for (int j = 0; j < rNodes; j++) {
    finalNeutrals[zNodes - 1][j] -=
initialNeutrals[zNodes - 1][j]
        * neutralDriftVel;
}

```

```

    }
    // At inlet, particles enter domain at drift velocity
    // Assume initial density is correct, corresponds to
    addition of ~3.09 neutral superparticles added each step
    for (int j = 1; j < rNodes - 1; j++) {
        finalNeutrals[1][j] += initialNeutrals[0][j] *
neutralDriftVel * 0.9788;
        finalNeutrals[1][j + 1] += initialNeutrals[0][j] *
neutralDriftVel
            * 0.0106;
        finalNeutrals[1][j - 1] += initialNeutrals[0][j] *
neutralDriftVel
            * 0.0106;
        finalNeutrals[0][j] = 1128 * neutralInjectionCal;
    }
    finalNeutrals[1][0] += initialNeutrals[0][0] *
neutralDriftVel * 0.9894;
    finalNeutrals[1][1] += initialNeutrals[0][0] *
neutralDriftVel * 0.0106;
    finalNeutrals[1][rNodes - 1] += initialNeutrals[0][rNodes
- 1]
        * neutralDriftVel * 0.9894;
    finalNeutrals[1][rNodes - 2] += initialNeutrals[0][rNodes
- 1]
        * neutralDriftVel * 0.0106;
    finalNeutrals[0][0] = 1128 * neutralInjectionCal;
    finalNeutrals[0][rNodes - 1] = 1128 *
neutralInjectionCal;
    // Set global neutral values to computed neutral values
    for (int i = 0; i < zNodes; i++) {
        for (int j = 0; j < rNodes; j++) {
            *ptrNeutrals[i][j] = finalNeutrals[i][j];
            if (*ptrNeutrals[i][j] < 0) {
                *ptrNeutrals[i][j] = 0;
            }
        }
    }
}

int ionExceededCount = 0;

// findNextValue finds the next available value in the Ion
Array
int findNextValue(int startingValue) {

```

```

// Declare variables used
int ionCounter = startingValue;
bool counterFlag = 0;
bool loopingFlag = 0;
bool skipInjectFlag = 0;
// Loop over the values in the Ion Array, returning to 0
once if last value is filled. Return available index
while (counterFlag == 0) {
    if (*ptrIonArray[ionCounter][0] <= -0.95
        && *ptrIonArray[ionCounter][1] <= -0.95)
{
        counterFlag = 1;
    } else {
        ionCounter += 1;
    }
    if (ionCounter >= maxNumberIons) {
        if (loopingFlag == 1) {
            counterFlag = 1;
            ionExceededCount += 1;
            skipInjectFlag = 1;
        }
        ionCounter = 0;
        loopingFlag = 1;
    }
}
if (skipInjectFlag == 1) {
    return -1;
    cout << "Index of -1 returned" << endl;
} else {
    return ionCounter;
}
}

// Initialize an ion array counter to speed up findNextValue
function
int ionArrayCounter = (zNodes - 1) * (rNodes - 1) *
(numParticlesPerCell) - 1;

// Ionize Particles using electron and neutral densities,
adjust values and inject ion particles
void ionizeParticles() {
    // Declare temporary variables
    double nuEN[zNodes][rNodes];
    double deltaElectrons[zNodes][rNodes];

```

```

double deltaIons[zNodes][rNodes];
double deltaNeutrals[zNodes][rNodes];
double deltaAtIons[zNodes - 1][rNodes - 1];
double injectIons[zNodes - 1][rNodes - 1];
*ptrOutThrust = 0;
// Set DeltaAtIons to zero
for (int i = 0; i < zNodes - 1; i++) {
    for (int j = 0; j < rNodes - 1; j++) {
        deltaAtIons[i][j] = 0;
    }
}
// Compute nuEN at each point, concurrently finding
deltaElectrons and deltaNeutrals
for (int i = 0; i < zNodes; i++) {
    for (int j = 0; j < rNodes; j++) {
        nuEN[i][j] = (particlesPerSuper) *
(*ptrNeutrals[i][j]) * 4.039
        * pow(10.0, -6.0) * 0.75 /
pow(neutralInjectionCal, 2.0);
        // delta variables should use units of
superparticles
        deltaElectrons[i][j] = nuEN[i][j] *
(*ptrElectrons[i][j])
        / (particlesPerSuper);
        deltaIons[i][j] = deltaElectrons[i][j];
        deltaNeutrals[i][j] = -deltaElectrons[i][j];
        double twoKEoverM = 3 * *ptrPhi[i][j] / (1.223
* pow(10.0, 8));
        if (twoKEoverM < 0) {
            twoKEoverM = abs(twoKEoverM);
        }
        double inj = neutralInjectionCal * phiCal;
        if (neutralInjectionCal > 1) {
            inj = sqrt(neutralInjectionCal) *
phiCal;
        }
        *ptrOutThrust += deltaIons[i][j] *
sqrt(twoKEoverM)
        * inj * pow(10.0, 5.0)
        * (particlesPerSuper /
numParticlesPerCell)
        * (3.093 * pow(10.0, -11.0));
    }
}

```

```

// Change Neutral and Electron values
for (int i = 0; i < zNodes; i++) {
    for (int j = 0; j < rNodes; j++) {
        *ptrElectrons[i][j] += deltaElectrons[i][j];
        *ptrNeutrals[i][j] += deltaNeutrals[i][j];
        if (*ptrElectrons[i][j] < 0) {
            *ptrElectrons[i][j] = 0;
        }
        if (*ptrNeutrals[i][j] < 0) {
            *ptrNeutrals[i][j] = 0;
        }
    }
}

// Compute delta ions at the center of each grid point,
including any remaining particle ions
// from previous ionization steps
for (int i = 1; i < zNodes - 1; i++) {
    for (int j = 1; j < rNodes - 1; j++) {
        deltaAtIons[i - 1][j - 1] += 0.25 *
deltaIons[i][j];
        deltaAtIons[i - 1][j] += 0.25 *
deltaIons[i][j];
        deltaAtIons[i][j - 1] += 0.25 *
deltaIons[i][j];
        deltaAtIons[i][j] += 0.25 * deltaIons[i][j];
    }
}

// At top and bottom, only distribute ions to 2 grid
center points
for (int i = 1; i < zNodes - 2; i++) {
    deltaAtIons[i - 1][0] += 0.5 * deltaIons[i][0];
    deltaAtIons[i][0] += 0.5 * deltaIons[i][0];
    deltaAtIons[i - 1][rNodes - 2] += 0.5 *
deltaIons[i][rNodes - 1];
    deltaAtIons[i][rNodes - 2] += 0.5 *
deltaIons[i][rNodes - 1];
}

// At left and right ends, only distribute ions to 2 grid
center points
for (int j = 1; j < rNodes - 2; j++) {
    deltaAtIons[0][j - 1] += 0.5 * deltaIons[0][j];
    deltaAtIons[0][j] += 0.5 * deltaIons[0][j];
    deltaAtIons[zNodes - 2][j - 1] += 0.5 *
deltaIons[zNodes - 1][j];
}

```



```

        deltaAtIons[zNodes - 2][j] += 0.5 *
deltaIons[zNodes - 1][j];
    }
    // Finally, at the corners the ions are distributed to
their only neighboring grid center
    deltaAtIons[0][0] += deltaIons[0][0];
    deltaAtIons[zNodes - 2][0] += deltaIons[zNodes - 1][0];
    deltaAtIons[0][rNodes - 2] += deltaIons[0][rNodes - 1];
    deltaAtIons[zNodes - 2][rNodes - 2] += deltaIons[zNodes -
1][rNodes - 1];
    // Now add back any remainder ions from previous
ionization steps, calculate ions to inject and remainder
    for (int i = 0; i < zNodes - 1; i++) {
        for (int j = 0; j < rNodes - 1; j++) {
            deltaAtIons[i][j] += *ptrRemainderIons[i][j];
            injectIons[i][j] =
static_cast<int>(deltaIons[i][j]);
            *ptrRemainderIons[i][j] = deltaAtIons[i][j]
-
static_cast<double>(injectIons[i][j]);
        }
    }
    // Next inject ions in each cell using calculated
injections
    // Index variable to call findNextValue function, call
randomizer object to generate random numbers
    int index;
    bool breakFlag = 0;
    default_random_engine e(seed);
    uniform_real_distribution<float> n(0, 1);
    for (int i = 0; i < zNodes - 1; i++) {
        if (breakFlag == 1) {
            break;
            cout << "Ionize Particles Break" << endl;
        }
        for (int j = 0; j < rNodes - 1; j++) {
            if (breakFlag == 1) {
                break;
            }
            for (int k = 0; k < injectIons[i][j]; k++) {
                if (breakFlag == 1) {
                    break;
                }
            }
            index = findNextValue(ionArrayCounter);

```

```

        if (index < 0) {
            breakFlag = 1;
            break;
            cout << "Index < 0 for i,j,k: " <<
i << ", " << j << ", "
                                << k << endl;
        }
        *ptrIonArray[index][0] = n(e) + i;
        *ptrIonArray[index][1] = n(e) + j;
        *ptrIonArray[index][2] =
neutralDriftVel;
        *ptrIonArray[index][3] = 0;
        ionArrayCounter += 1;
    }
}
seed += 1;
}

double avgPhiGlobal = 140 * phiCal;

void normalizePhi() {
    // Ensures constant global potential in the system,
    ignores potential at anode which is fixed
    double totalPhi = 0;
    double avgPhi = 0;
    double deltaPhi = 0;
    for (int i = 1; i < zNodes; i++) {
        for (int j = 0; j < rNodes; j++) {
            totalPhi += *ptrPhi[i][j];
        }
    }
    avgPhi = totalPhi / ((zNodes - 1) * (rNodes));
    deltaPhi = (avgPhiGlobal - avgPhi) / 2;
    for (int i = 1; i < zNodes; i++) {
        for (int j = 0; j < rNodes; j++) {
            *ptrPhi[i][j] += deltaPhi;
        }
    }
}

int main() {

    // Initialization function calls

```

```

initPtrRho();
initPtrPhi();
initPtrEField();
setInitEField();
initPtrIonArray();
setInitIons();
initPtrElectrons();
initPtrNeutrals();
initPtrRemainderIons();

double timeAvgThrust = 0;

for (int k = 0; k < timeSteps; k++) {
    for (int t = 0; t < 5; t++) {
        computePhi();
        normalizePhi();
    }
    computeEField();
    moveParticles();
    diffuseElectrons();
    diffuseNeutrals();
    ionizeParticles();
    cout << "ptrOutThrust for t = " << k + 1 << ": " <<
*ptrOutThrust
        << endl;
    if (k > timeSteps - 101) {
        timeAvgThrust += *ptrOutThrust;
    }
}

cout << "Time Averaged Thrust: " << timeAvgThrust / 100
<< endl;

for (int i = 0; i < zNodes; i++) {
    cout << "ptrElectrons at (" << i << ",7.5): "
        << (*ptrElectrons[i][8] +
*ptrElectrons[i][7]) / 2 << endl;
}
for (int i = 0; i < zNodes; i++) {
    cout << "ptrNeutrals at (" << i << ",7.5): "
        << (*ptrNeutrals[i][8] +
*ptrNeutrals[i][7]) / 2 << endl;
}
for (int i = 0; i < zNodes; i++) {

```

```
        cout << "ptrPhi at (" << i << ",7.5): "  
              << (*ptrPhi[i][8] + *ptrPhi[i][7]) / 2  
<< endl;  
    }  
  
    // NOTES: *****  
    return 0;  
}
```