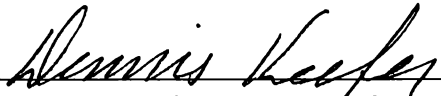


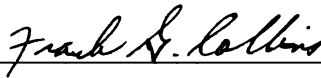
To the Graduate Council:

I am submitting herewith a thesis written by Thomas Edward Markusic entitled "Particle Simulation of a Langmuir Probe in Quiescent and Flowing Plasmas." I have examined the final copy of this thesis for form and content and recommend it be accepted in partial fulfillment of the requirement for the degree of Master of Science, with a major in Aerospace Engineering.



Dr. Dennis R. Keefer, Major Professor

We have read this thesis
and recommend its acceptance.





Accepted for the Council:



Associate Vice Chancellor
and Dean of The Graduate School

**Particle Simulation of a Langmuir Probe
in Quiescent and Flowing Plasmas**

A Thesis
Presented for the
Master of Science
Degree
The University of Tennessee, Knoxville

Thomas Edward Markusic
December 1996

DEDICATION

This thesis is dedicated to my parents

Mr. Stephen M. Markusic

and

Mrs. Marilyn E. Markusic

whom I love very much.

ACKNOWLEDGMENTS

The completion of this Master's thesis owes not only to my own effort, but to the invaluable support of two other people. Christa E. Markusic, my wife, endured many nights alone and, throughout the process, took care of me at home and inspired me with her love. Dr. Dennis R. Keefer, my major professor, provided the topic for this thesis. Further, he helped me work through technical problems that surfaced along the way — drawing on his seemingly intuitive knowledge of the subject.

Thank you, Christa and Dennis.

This material in this thesis is based upon work supported by the National Science Foundation under grant No. CTS-9512489. The computations were primarily carried out on computers purchased with this NSF Grant. The faculty at the Center for Laser Applications, especially Dr. Christian Parigger, were most helpful in making sure that the computers were in working order. Thank You.

Additionally, I would like to thank The University of Tennessee Space Institute and NASA, whose financial contributions made my studies possible.

ABSTRACT

A three dimensional electrostatic Particle-In-Cell (PIC) code has been developed to simulate a Langmuir probe in both quiescent and flowing plasmas. The code was written to model the use of the Langmuir probe in plasma regimes for which no closed-form analytical solutions exist; this is the case for a probe in an ion beam, such as the plume of an ion thruster.

Langmuir probes are used to determine local plasma properties, such as electron temperature, by careful dissection of the probe's Voltage-Current (V-I) characteristic. To interpret experimental data from a Langmuir probe, one must separate ion from electron current. This process is well documented for quiescent plasmas; however, no systematic techniques are available for interpreting data obtained using an electric probe in an ion beam. Ad hoc estimates of probe ion current in beam plasmas may lead to order of magnitude errors in the calculation of electron temperature. The PIC code described in this thesis was written to elucidate the beam-probe interaction and provide systematic techniques for legitimately interpreting experimental data.

Elements of electric probe and PIC theory in general are discussed; in particular, plasma sheath theory and methods used in fluxing particles across boundaries are described in detail. Code results (i.e. Current-Voltage characteristics) are presented for a low density, quiescent plasma and a neutralized ion beam. Code and theoretical probe traces for an infinite cylindrical probe in a quiescent plasma are shown to be in agreement. Also, code results for a plasma beam are compared with experimental data from the UTSI three grid ion thruster.

Contents

1	Introduction	1
2	Plasma Sheath and Electric Probe Theory	4
2.1	Sheath Theory	5
2.1.1	Bohm Sheath Theory	10
2.2	Electric Probe Theory	14
2.2.1	The Classical Probe Theory of Langmuir and Mott-Smith . . .	17
2.2.2	Other Descriptions of Quiescent Probe Theory	22
2.2.3	Probe Theory for a Flowing Plasma	23
3	The Particle-In-Cell Technique	27
3.1	What Is PIC ?	27
3.2	The Computational Cycle	30
3.2.1	Charge Weighting	31
3.2.2	Field Solver	35
3.2.3	Force Weighting	37
3.2.4	Integration of the Equations of Motion	38
3.3	Boundary and Initial Conditions	40
4	PROBEPIC	44
4.1	Computational Domain Layout	45

4.2	PROBEPIC Code Details	46
4.2.1	General Remarks (<code>probepic.h</code>)	49
4.2.2	Random Number Generation (<code>random_number.c</code>)	50
4.2.3	Initial Conditions (<code>parameter.c</code>)	51
4.2.4	Grid Generation (<code>grid.c</code>)	51
4.2.5	Maxwellian Velocity Generator (<code>maxwellian.c</code>)	52
4.2.6	Initial Particle loading (<code>initialize.c</code>)	54
4.2.7	Fluxing Particles (<code>inject_part.c</code> , <code>inject_beam.c</code>)	55
4.2.8	Charge Weighting (<code>q_weight.c</code>)	59
4.2.9	Field Solver (<code>e_field.c</code>)	60
4.2.10	Force Weighting (<code>f_weight.c</code>)	69
4.2.11	Moving Particles (<code>mover.c</code>)	71
4.2.12	Checking Boundaries (<code>boundary.c</code>)	72
4.2.13	Output (<code>output_data.c</code> , <code>graphics.c</code>)	74
5	Simulation Results	75
5.1	Probe in a Quiescent Plasma	75
5.2	Effect of Probe Aspect Ratio	77
5.3	Probe in a Flowing Plasma	79
6	Conclusion	83
6.1	Suggestions for the Improvement of PROBEPIC	83
6.2	Suggestions for the Future Implementation of PROBEPIC	86
	Bibliography	88

Appendices	92
A PROBEPIC Source Code	93
A.1 boundary.c	93
A.2 boundary_beam.c	95
A.3 charge_weight.c	97
A.4 force_weight.c	98
A.5 graphics.c	100
A.6 grid.c	107
A.7 initialize.c	108
A.8 initialize_beam.c	109
A.9 inject_beam.c	110
A.10 inject_part.c	112
A.11 locate.c	115
A.12 make_LU.c	116
A.13 make_velocity_table.c	123
A.14 mover.c	125
A.15 output_data.c	126
A.16 parameter.c	127
A.17 probepic.c	130
A.18 probepic.h	133
A.19 random_number.c	136
A.20 reset_grid.c	142
B Sample PROBEPIC Output	144
Vita	145

List of Tables

4.1	PROBEPIC sub-program description.	48
5.1	Quiescent plasma simulation plasma conditions.	76
5.2	Data from analysis of Keefer and Semak.	80
5.3	Conditions used in PROBEPIC and in the theoretical curve for the plasma beam.	80

List of Figures

2.1	Schematic of the potential distribution between two constant potential planes for different intervening media.	6
2.2	The variation of potential near a wall; ϕ is the negative of the potential.	11
2.3	Typical voltage-current (VI) characteristic for a Langmuir probe.	15
2.4	Langmuir theory VI characteristics for several temperatures.	19
2.5	Natural logarithm of electron current vs. potential for several temperatures.	21
2.6	Comparison of Langmuir and Laframboise results in a rarefied plasma.	23
2.7	Illustration of the end effect. The figure shows the variation of ion current with angle of attack. The dashed line is theoretical, from Langmuir. The solid line is experimental data, from Hester and Sonin (reproduced with permission of Sonin).	25
3.1	Schematic of PIC objects.	28
3.2	A typical cycle (one time step) in a PIC simulation.	30
3.3	Particle charge weighting. a) NGP weighting. b) First-order linear particle weighting	32
3.4	Electric Field generated by the weighted charge of a single particle as a function of the particle's position within the cell.	34
4.1	Schematic of PROBEPIC computational domain layout.	46

4.2	Snap-shot of PROBEPIC simulation.	47
4.3	Maxwellian distribution of peculiar speeds W for electrons at $T=2.0[\text{eV}]$, $f(W)$, and the normalized integral of $f(W)$, $F(W)$	53
4.4	Illustrative flux volume.	56
4.5	Schematic of field boundary conditions in PROBEPIC.	61
4.6	Two dimensional interior computational mesh-point and Gauss' law volume.	62
4.7	Gauss' law volume for $(r = 0)$ axis in front of the probe	65
4.8	Sketch of the band matrix form; non-zero elements are indicated by black diagonal lines.	67
4.9	Contour plot of regions of constant electrostatic potential for two dif- ferent conditions. The top result is for charge-free space. The bottom result is for a plasma filled computational domain. In both cases the probe is biased at $-2.0[\text{V}]$ relative to the plasma potential.	70
5.1	Comparison between PROBEPIC and Langmuir theory results.	77
5.2	Illustration of the effect of probe aspect ratio on agreement with infinite probe Langmuir theory.	78
5.3	Comparison between PIC, Langmuir theory, and experimental results.	81

Chapter 1

Introduction

It is estimated that more than 99.9 percent of the matter in the known universe is in the plasma state. In our generation, man has first attempted to extend his domain beyond the earth; therefore, it is only fitting and natural that man should strive to understand and exploit the unusual properties of the plasma state of matter.

This increasing interest has led to the development of many methods, or *diagnostic techniques*, to measure the composition and thermodynamic properties of plasmas. Among others, these include: microwave interferometry, electron, ion, and neutron beams, and electrostatic probes. The subject of this thesis is electrostatic probes or, more specifically, Langmuir probes (named in honor of Irving Langmuir, who developed the original theory and experimental methods for their use in the mid-twenties). The Langmuir probe has an important advantage over many other plasma diagnostic techniques: it allows one to obtain *local* as opposed to *average* (or line-integrated) plasma properties.

Langmuir probes have a broad range of applicability — from glow discharges to fusion plasmas. They find use in both laboratory and industrial environments. The simplicity of the Langmuir probe experimental setup makes it an attractive diagnostic for the experimentalist. In plasma processing control, they may be used to give an indication that a plasma processing device is producing the same plasma characteristics

as on a prior occasion.

Since their inception, many theoretical studies have been conducted to understand the behavior of Langmuir probes. I. Langmuir and H. Mott-Smith pioneered both the experimental and theoretical interpretation of probe data. More advanced treatments resulted from a better understanding of plasma sheaths; the work of Bohm [4] in the late forties is particularly noteworthy for its elucidation of sheath phenomena. In an effort to include the proper potential distribution in the sheath, Allen [1] derived expressions for current collection which included the effect of electron potential barriers, while ions were assumed to be immobile, or “cold.” Bernstein and Rabinowitz [2] expanded upon the work of Allen by also allowing for monoenergetic ions. Laframboise [5] completed the picture by incorporating both thermal electrons and ions.

All of the theoretical work described above assumes current collection in a collisionless, quiescent plasma by an *infinite* cylindrical probe. The purpose of this thesis is to introduce a particle-in-cell (PIC) code, PROBEPIC. PROBEPIC extends the work of previous treatments by simulating the behavior of *finite* length probes. In its present form, PROBEPIC may be used to conduct Langmuir probe experiments in both quiescent and flowing plasmas. We are compelled to use the term “experiments” because the PIC method, which uses computational particles to represent real electrons and ions, gets as close to reality as we can expect to on a computer. We are not simply numerically integrating a set of differential equations; the PIC method introduces experimental realities such as statistical deviations. PROBEPIC enables us to conduct parametric studies on and interpret experimental data from Langmuir probes. For example, the effect of probe dimensions or specific plasma conditions may be explored. Also, the results of the “hand” analysis of experimental data may be verified by running PROBEPIC with similar experimental conditions. In this thesis we perform such tasks. The effect of the probe aspect ratio on experimental results

is quantified, perhaps for the first time. The results of experimental data from an ion thruster plume are scrutinized.

While the results presented in this thesis are interesting, the knowledge attained in the development of PROBEPIC is equally important, as unique/novel techniques were developed. Consequently, this treatment presents PROBEPIC in the broad context of relevant theory, and then focuses on specific techniques to implement this theory in the computational environment. Chapter 2 describes analytical techniques that have previously been developed to grapple with the difficult task of interpreting experimental Langmuir probe data. The analytical techniques serve as limiting cases by which to evaluate the general validity of PROBEPIC output. Chapter 3 describes the various facets of the PIC technique in general, and establishes a theoretical framework for the algorithms used in PROBEPIC. Chapter 4 presents a thorough treatment of the theoretical and computational details specific to PROBEPIC. This chapter is the most important since it describes techniques that cannot be found elsewhere. Chapter 5 details the results of the application of PROBEPIC to several problems and summarizes the present study — suggesting future applications of PROBEPIC.

Chapter 2

Plasma Sheath and Electric Probe Theory

One of the most outstanding characteristics of a plasma is its ability to maintain internal charge neutrality. Near boundaries, which would otherwise disturb this quasi-neutrality, the plasma redistributes its constituent particles in such a manner as to shield the bulk of the plasma from the perturbation. This is accomplished through a thin plasma layer called a *sheath* in which ion and electron densities can differ. Thus, large electric fields can be sustained to counter those fields imposed by the perturbing object.

If we can develop accurate theoretical models for this shielding effect, then we may insert objects, or *electric probes*, into plasmas and infer bulk plasma thermodynamic properties from the careful analysis of charge collection for various applied probe potentials. In short, valid interpretation of experimental probe data hinges on a thorough understanding of the mechanisms at play in a plasma sheath.

Unfortunately, the customary equations that govern the motion of plasmas change character drastically in the vicinity of the boundaries. Consequently, the theory of probes is extremely complicated. For this reason the literature on probes is extensive, even attracting the interest of twentieth-century scientific giants such as Irving

Langmuir and David Bohm. The methodology used in the references is mathematically intensive. In the following sections we condense and filter previous research to suit the needs of the present treatment — looking qualitatively at results that will be directly applicable to the interpretation of PROBEPIC output found in chapter 5. The interested reader may find more detailed treatments in the references listed in the bibliography.

2.1 Sheath Theory

Bulk plasma tends to be quasi-neutral, even in systems which are finite and have boundaries. The plasma isolates itself from the boundaries through a non-neutral plasma sheath. The structure of sheaths is very much dependent on the particular problem geometry and the thermodynamic state of the plasma. Solutions almost always depend on simplifying assumptions; these assumptions are problem dependent. Therefore, in this treatment we focus on theory relevant to electric probes: *current collection* and *sheath boundary conditions* in steady state, uniform, isotropic, collisionless, single ion species, and unmagnetized plasmas.

Let us first consider some general, qualitative sheath properties. Figure 2.1 will be used to illustrate the structure of planar sheaths under various boundary conditions. First, consider boundaries A and B as infinite conducting planes whose potentials are ϕ_A and ϕ_B respectively. In the absence of an intervening plasma, the potential simply increases linearly as shown in the bottom curve. As a second example, let the boundary A be a grounded infinite conducting plane, and boundary B a point far away inside the (quasi-neutral) bulk plasma. The potential is modified significantly with the addition of a quiescent plasma. If the ions and electrons are in thermal equilibrium,

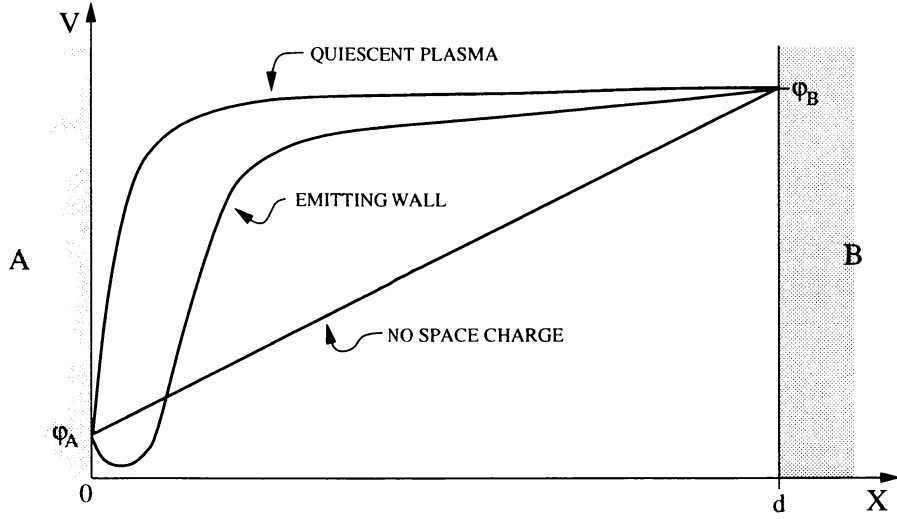


Figure 2.1: Schematic of the potential distribution between two constant potential planes for different intervening media.

then

$$\frac{1}{2}m_i\bar{v}_i^2 = \frac{1}{2}m_e\bar{v}_e^2, \quad (2.1)$$

$$\Rightarrow \frac{\bar{v}_e}{\bar{v}_i} = \sqrt{\frac{m_i}{m_e}}. \quad (2.2)$$

Equation (2.2) reveals that the mean velocity of the electrons in the plasma is much greater than that of the ions, because $m_i \gg m_e$. Consequently, unchecked, the electron flux to boundary *A* would be much greater than that of the ions, and the neutrality of the bulk plasma would soon be broken. The plasma avoids this by charging positive with respect to the boundary — setting up an electron repelling sheath to balance electron and ion losses. This is the origin of a finite *plasma potential*, ϕ_B , which is illustrated by the top curve in Figure 2.1. As a final illustrative example, again consider boundaries *A* and *B* as infinite conducting planes whose potentials are ϕ_A and ϕ_B respectively (the following derivation follows Chen [19] closely). Let $\phi_A = 0$ for simplicity. Further, consider boundary *A* as a surface which emits electrons (of mass m and charge $-e$) with zero velocity, and boundary *B* as a perfectly absorbing

surface at potential ϕ_B . In this conservative system the Hamiltonian is constant:

$$H = T + V = 0 , \quad (2.3)$$

$$\Rightarrow \frac{1}{2}mv^2 = eV(x) , \quad (2.4)$$

and therefore the instantaneous velocity of an electron is

$$v(x) = \left(\frac{2eV(x)}{m} \right)^{\frac{1}{2}} \quad (2.5)$$

If the current density is defined by

$$j(x) = n(x) v(x) , \quad (2.6)$$

then the number density is , using (2.5),

$$n(x) = j(x) \left[\frac{2eV(x)}{m} \right]^{-\frac{1}{2}} \quad (2.7)$$

Poisson's equation for the electrostatic potential may then be written (in Gaussian units)

$$\begin{aligned} \frac{d^2V}{dx^2} &= -4\pi\rho(x) \\ &= 4\pi en(x) \\ &= 4\pi ej(x) \left[\frac{2eV(x)}{m} \right]^{-\frac{1}{2}} . \end{aligned} \quad (2.8)$$

Multiplying both sides by dV/dx and integrating from $x = 0$, we find

$$\begin{aligned} \frac{1}{2} \left(\frac{dV}{dx} \right)^2 &= 4\pi ej(x) \int_0^V \left[\frac{2eV(x)}{m} \right]^{-\frac{1}{2}} dV \\ &= 4\pi j(x) (2me)^{\frac{1}{2}} V^{\frac{1}{2}} + \left(\frac{dV}{dx} \right)_0 . \end{aligned} \quad (2.9)$$

Since we have electrons with zero velocity at $x = 0$, the electric field must also vanish, that is,

$$\left(\frac{dV}{dx} \right)_0 = 0 . \quad (2.10)$$

Thus, rearranging (2.9) we have

$$V^{-\frac{1}{4}}dV = (8\pi j(x))^{\frac{1}{2}}(2me)^{\frac{1}{4}}dx . \quad (2.11)$$

Integrating (2.11) from $x = 0$ to $x = d$ we find

$$\frac{4}{3}V_B^{\frac{3}{4}} = (8\pi j(d))^{\frac{1}{2}}(2me)^{\frac{1}{4}}d \quad (2.12)$$

or

$$j(d) = \left(\frac{2}{me}\right)^{\frac{1}{2}} \frac{V_B^{\frac{3}{2}}}{9\pi d^2} . \quad (2.13)$$

Equation (2.13) is the Child-Langmuir $\frac{3}{2}$ -power law, which describes *space-charge-limited current flow* between two planes separated by a distance d with a potential V_B between them. A more realistic case would allow the electrons to enter with finite velocities. This has the effect of initially depressing the potential below zero, or the building up of a field to oppose the emission of electrons. The potential associated with this situation is represented by the middle curve in Figure 2.1. Langmuir [3] derived an expression for space-charge-limited current for particles which enter with a thermal, or Maxwellian distribution:

$$j(d) = \left(\frac{2}{me}\right)^{\frac{1}{2}} \frac{1}{9\pi} \frac{(V_B - V_m)^{\frac{3}{2}}}{(d - x_m)^2} \left(1 + \frac{2.66}{\sqrt{\eta}}\right) , \quad (2.14)$$

where V_m is the potential minimum, x_m is the associated position of the potential minimum, and $\eta = eV_B/kT$. Although we formulated this derivation for electrons, the formulas are also valid for ions in an attractive potential if appropriate masses and temperatures are used.

The derivation above has a more specific interpretation — it is a first order approximation for the current collected by a conductor immersed in a plasma. Boundary A is analogous to the outer edge of the sheath through which particles from the bulk plasma cross due to thermal motion, and boundary B is analogous to a current collecting conductor. This analogy, however, has its limitations. First, a collisionless planar

sheath is assumed. For a perfectly absorbing infinite plane conductor, all particles move *toward* the conductor; no particles move *away* from the conductor, implying that half of the distribution function is missing. In an experimental plasma, the plasma compensates for this by allowing the applied potential to extend far into the bulk plasma, thus blurring the concept of a sheath altogether. Second, the model assumes all charge that crosses the sheath edge is collected at the opposite boundary as current. In practice, however, particles are free to execute orbital trajectories which would not necessarily result in collection. In any event, it has been shown experimentally that the Child-Langmuir model gives reasonable results in the thin sheath limit. More will be said about these issues in Section 2.2 in the discussion of electric probes.

We have considered how current might be collected through a plasma sheath. To complete the picture we should define the relative dimensions of sheaths; the following derivation closely follows that of Chen [19]. Consider a plane conductor at $x = 0$ and potential V_0 immersed in a plasma of dimensions R and undisturbed number density n_0 . Poisson's equation for the one-dimensional problem is:

$$\frac{d^2V}{dx^2} = -4\pi e(n_i - n_e) . \quad (2.15)$$

If we introduce the dimensionless variables

$$\eta = -\frac{ev}{kT_e}, \quad \nu_i = \frac{n_i}{n_0}, \quad \nu_e = \frac{n_e}{n_0}, \quad \xi = \frac{x}{R} , \quad (2.16)$$

(2.15) becomes

$$\frac{h^2}{R^2} \frac{d^2\eta}{d\xi^2} = \nu_i(\eta) - \nu_e(\eta) , \quad (2.17)$$

where,

$$h := \left(\frac{kT_e}{4\pi n_0 e^2} \right)^{\frac{1}{2}} \quad (2.18)$$

For simplicity, assume that the ions are relatively immobile as compared to the electrons, that is, they are uniformly distributed. Assume that the electrons are in a

Boltzmann distribution:

$$n_e = n_0 e^{-\eta} . \quad (2.19)$$

Poisson's equation then takes the form

$$\frac{d^2 \eta}{d(x/h)^2} = 1 - e^{-\eta} . \quad (2.20)$$

If η is small, we can expand the exponential in a Taylor series:

$$e^{-\eta} \approx 1 - \eta + \dots , \quad (2.21)$$

which results in the Poisson equation taking the form

$$\frac{d^2 \eta}{d(x/h)^2} = \eta . \quad (2.22)$$

Finally, the solution of (2.22) yields

$$V = V_0 e^{-(x/h)} . \quad (2.23)$$

Equation (2.23) shows that an externally applied potential of magnitude V_0 is effectively shielded (i.e. reduced to $1/e$ of its initial value) within a distance of order h . The length h is called the *Debye shielding length*.

In this section we have shown that current collection by electric probes can be represented, at least to first-order, by space-charge-limited current flow through sheaths whose characteristic dimension is the Debye length. A more detailed treatment of the sheath has been given by Bohm [4], which includes stability boundary conditions.

2.1.1 Bohm Sheath Theory

The formulas derived above for space-charge-limited current assume that the electric field at the sheath edge is exactly zero. However, in reality the shielding is not quite perfect, and a small portion of the potential drop between electrode and plasma may

penetrate beyond the sheath edge. While at first glance it may seem pedantic to worry about small fields that extend beyond the sheath edge; however, it turns out that the extent to which the potential extends into the bulk plasma may determine the overall stability of the sheath.

Bohm proposed that Debye shielding be divided into three phenomenologically different regions, as illustrated in Figure 2.2: the plasma region, transition region, and sheath region. Within the plasma charge neutrality prevails, with a very gradual increase in potential in the direction of the electrode. The sheath region is characterized by a large potential gradient, and negligible electron density. In between, the transition region bridges the small and large field regions of the plasma and sheath respectively. In other words, there is no precise point where one can say the sheath ends and the plasma begins. The combined transition and adjacent plasma regions are often referred to as the *presheath*.

We will now derive a condition which explains why plasma fields cannot be neglected — why the plasma/sheath transition may not be abrupt.

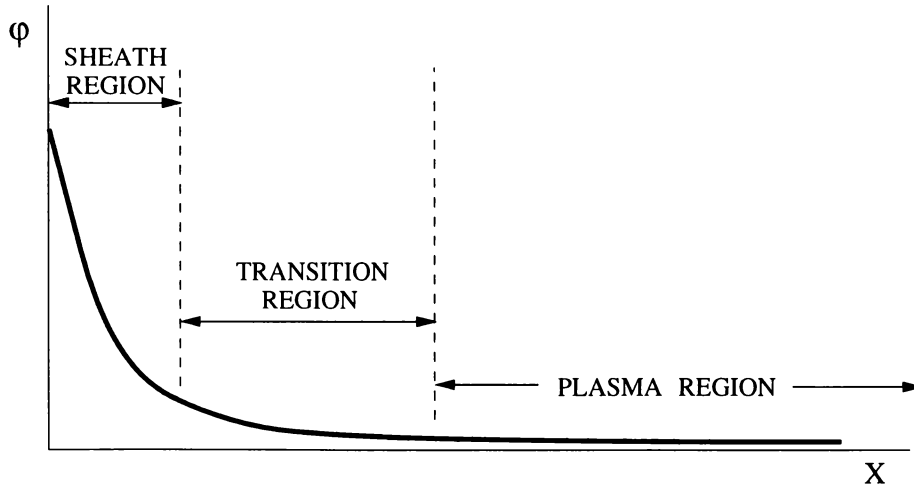


Figure 2.2: The variation of potential near a wall; ϕ is the negative of the potential.

Boundary Conditions for Sheath

In this simplified model we consider a one-dimensional sheath as illustrated in Figure 2.2. Ions are assumed to have been accelerated through the presheath up to the sheath edge through the plasma potential V_0 , that is, the role of the plasma fields are taken into account by assuming that they provide ions at the sheath edge with some mean energy equal to eV_0 .

Following Bohm [4], we first write down Poisson's equation for the potential inside the sheath region,

$$\nabla^2 V = 4\pi e(n_+ - n_e) , \quad (2.24)$$

where n_+ is the ion density and n_e is the electron density. We assume that the electrons are in a Boltzmann distribution:

$$n_e = n_0 \exp\left(-\frac{e(V - V_0)}{kT_e}\right) . \quad (2.25)$$

Since the ions have kinetic energy eV , their velocity is

$$v_+ = \sqrt{\frac{2eV}{m_+}} . \quad (2.26)$$

From continuity, the current density ($J = nv$) must be the same at the edge of the sheath and at all internal points; therefore,

$$J_+ = n_+ \sqrt{\frac{2eV}{m_+}} = n_0 \sqrt{\frac{2eV_0}{m_+}} \quad (2.27)$$

$$\Rightarrow n_+ = n_0 \sqrt{\frac{V_0}{V}} . \quad (2.28)$$

Reassembling (2.24) gives

$$\frac{d^2 V}{dx^2} = 4\pi n_0 e \left[\sqrt{\frac{V_0}{V}} - \exp\left(-\frac{e(V - V_0)}{kT_e}\right) \right] . \quad (2.29)$$

Multiplying (2.29) by dV/dx and integrating we find,

$$\frac{1}{2} \left(\frac{dV}{dx} \right)^2 = 4\pi n_0 e \left[\sqrt{V_0 V} + \frac{kT_e}{e} \exp \left(-\frac{e(V - V_0)}{kT_e} \right) \right] + C . \quad (2.30)$$

The constant is C found by assuming that the electric field vanishes as $V \rightarrow V_0$, that is, $dV/dx = 0$ when $V = V_0$. Implementing this condition and rearranging terms we find,

$$\left(\frac{dV}{dx} \right)^2 = 8\pi n_0 e \left\{ 2V_0 \left(\sqrt{\frac{V}{V_0}} - 1 \right) + \frac{kT_e}{e} \left[\exp \left(-\frac{e(V - V_0)}{kT_e} \right) - 1 \right] \right\} . \quad (2.31)$$

As V approaches V_0 , we are justified in expanding the right-hand side of (2.31) in a power series in ΔV (where $\Delta V = V - V_0$), in which case we find

$$\frac{1}{8\pi n_0 e} \left(\frac{dV}{dx} \right)^2 \approx \frac{1}{2} \left(\frac{e}{kT_e} - \frac{1}{2V_0} \right) (\Delta V)^2 . \quad (2.32)$$

Because $(dV/dx)^2 \geq 0$, *real* solutions are only possible if

$$\frac{e}{kT_e} \geq \frac{1}{2V_0} , \quad (2.33)$$

or,

$$eV_0 \geq \frac{kT_e}{2} . \quad (2.34)$$

Equation (2.34) tells us that a stable sheath is possible only when ions reach the sheath with a kinetic energy at least half the thermal energy of the electrons. In many experimental plasmas, the bulk plasma ion temperature may be significantly lower than the electron temperature. Equation (2.34) tells us that the sheath can be stabilized only if some mechanism is at work to bring the ion energy up to half the electron energy upon entry into the sheath. This leads us to conclude that the presheath is a real and necessary region in the Debye shielding process. It should be mentioned that for ion energies less than half the thermal energy of the electrons, the sheath Equation (2.29) predicts oscillatory solutions. This is what we mean by stable

vs. unstable sheaths. We should, however, remember that this derivation is based on the assumption of a planar sheath, and recall the comments from Section (2.1) about the validity of this assumption.

We will now turn our attention to how the sheath theories developed in this section are implemented to provide a basis for the theory of charge collection by laboratory electric probes.

2.2 Electric Probe Theory

Electrostatic probes are experimentally very simple devices. They generally consist of a metallic electrode (partially insulated and partially exposed) connected to an external circuit outside of the plasma. The external circuit usually consists of a DC power supply (which is used to bias the probe positive and negative relative to the plasma), and an ammeter or oscilloscope to measure the probe current. In this thesis we will be concerned only with the cylindrical Langmuir probe, which is basically a wire with part of the insulation stripped back.

Let us first get a qualitative idea of how Langmuir probes work. Figure 2.3 illustrates a typical, albeit somewhat idealized, experimental voltage-current (VI) characteristic. Total current to the probe is plotted as a function of probe bias potential. Electron current to the probe is taken as positive by convention, ion current, negative. The potential ϕ_s is the local zero, or plasma potential. For potentials greater than ϕ_s the probe attracts electrons; probe potentials less than ϕ_s repel electrons. Ions, having opposite charge, are repelled when $\phi > \phi_s$, and attracted when $\phi < \phi_s$. Let us consider the various regions of the the VI characteristic in more detail.

The point labeled C in Figure 2.3 corresponds to $\phi = \phi_s$, the plasma potential. There is no probe sheath at this potential; therefore, charged particles freely migrate

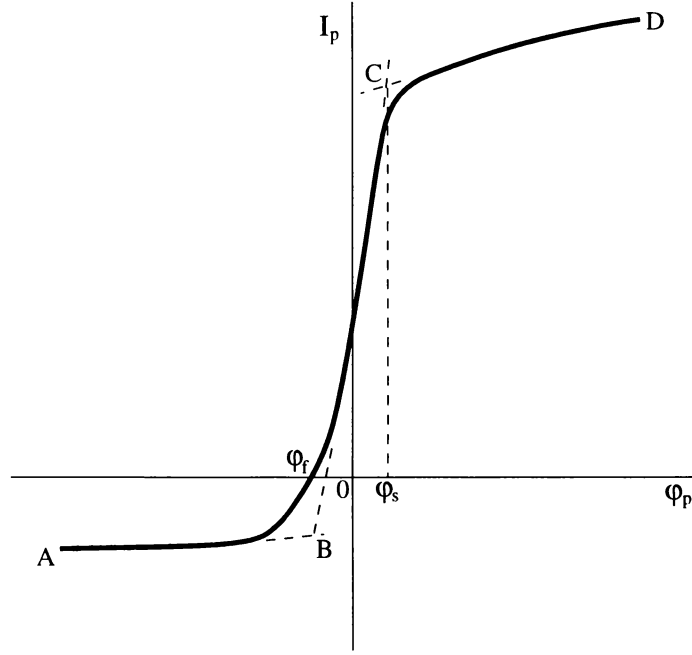


Figure 2.3: Typical voltage-current (VI) characteristic for a Langmuir probe.

to the probe because of their thermal velocities. As shown earlier in section 2.1, the electron thermal velocity is much greater than that of the ions; consequently, at ϕ_s , what is collected by the probe is predominantly electron current, as illustrated in the figure.

Between C and D the probe is biased positive relative to ϕ_s , collecting an increasing number of electrons and a diminishing number of ions. The exact amount of increased electron current for an incremental increase in probe potential is highly dependent on the state (e. g. number density, temperature, etc.) of the bulk plasma, that is, the effectiveness of the Debye shielding. In the thin sheath case, the VI characteristic has a pronounced knee at point C. Incremental increases in probe potential lead to little or no increase in probe current. For this reason, the potential and current associated with point C are often referred to as the electron *saturation* potential and current.

As the probe potential is made negative relative to ϕ_s , it begins to repel electrons

and attract ions. The electron current drops off very rapidly. As will be shown later, if the electrons are in a Maxwellian distribution the electron current will decrease exponentially in the region between points A and C. Between points B and C, a potential ϕ_f exists at which the total collected current is zero. This point is called the *floating potential*; at this point the field is sufficient to repel all electrons except a flux equal to the ion flux.

If the probe potential is made negative relative to ϕ_f we reach ion saturation at point B. The discussion above about electron saturation at point C applies analogously for ion saturation. It should be mentioned, though, that in plasmas which contain low energy ions the sheath formation for the ions may be quite different than for the electrons, since a large ion attracting presheath region may be necessary to satisfy the Bohm condition.

No exact analytical theory exists to determine the VI characteristic for arbitrary plasma conditions. The main difficulty is that the problem requires a self-consistent solution of the equations of motion and the Poisson equation. When one formulates the relevant equations, a set of simultaneous non-linear differential equations results for which no closed-form analytical solution is available. Simplifying assumptions may be introduced to make the problem more tractable; however, these assumptions restrict the solution's validity to a narrow set of plasma conditions.

Approximate solutions are available for two limiting cases: very thin sheaths, and very thick sheaths. For thin sheaths (i. e. $h \ll r_p$, where r_p is the probe radius) space-charge-limited current collection is assumed, where it is assumed that any particles crossing the sheath are collected. The resulting theoretical VI characteristic has very sharp knees, as pictured at points B and C in Figure 2.3. On the other hand, for very thick sheaths (i. e. $h \gg r_p$) *orbital motion limited* (OML) current collection is assumed. The OML current is the current collected by the probe when none of

the undisturbed particles (at infinity) capable of reaching the probe on the basis of energy considerations is excluded from doing so by intervening barriers of effective potentials. In other words, we neglect the influence of the sheath altogether, and simply compute particle orbits using the space-charge-free electric field of the the probe. Those orbits that intercept the probe are counted as current. In this case, the resulting VI characteristic does not have sharp knees at points B and C; rather, there is a very subtle (and often almost indiscernible) change from positive to negative curvature.

Since the OML theory is most relevant to the analysis of the results computed using PROBEPIC, we will focus on theory which describes the thick sheath limit.

2.2.1 The Classical Probe Theory of Langmuir and Mott-Smith

I. Langmuir and H.M. Mott-Smith conducted the first electric probe measurements, and consequently, developed the first probe theory [3] to interpret their data. In this section we describe their results for infinite cylindrical probes.

The Langmuir/Mott-Smith theory describes orbital motion limited current collection. Again, this approximation is valid in the thick sheath limit, where the Debye length is much larger than the probe radius. In this limit not all of the particles entering the sheath will strike the probe because of the possibility of orbital trajectories. Only those particles with the appropriate impact parameter, energy, and angular momentum will be collected. The details of the derivation are quite long, and can be found in the original paper; therefore, we will go directly to the result. For ions with a Maxwellian velocity distribution function, with the possibility of collection from r_p to infinity, the ion current collected by a cylindrical probe as a function of probe potential is given by:

$$I_i = A_p N_\infty Z_i e \sqrt{\frac{kT_i}{2\pi m_i}} j_i, \quad (2.35)$$

where,

$A_p :=$ probe area,

$N_\infty :=$ neutral plasma density,

$Z_i :=$ ion degree of ionization,

$e :=$ electron charge,

$k :=$ Boltzmann constant,

$T_i :=$ ion temperature,

$m_i :=$ mass of ion

and,

$$j_i = \frac{2}{\sqrt{\pi}} \left\{ \sqrt{-\chi_p} + \frac{\sqrt{\pi}}{2} e^{-\chi_p} [1 - \operatorname{erf}(\sqrt{-\chi_p})] \right\} , \text{ for } \chi_p < 0 , \quad (2.36)$$

$$j_i = e^{-\chi_p} , \text{ for } \chi_p > 0 , \quad (2.37)$$

where,

$$\chi_p = \frac{Z_i e \phi_p}{k T_i} . \quad (2.38)$$

ϕ_p is the probe potential measured relative to the plasma potential. Equations (2.37) and (2.38) are valid for electrons if the appropriate mass and charge sign are substituted into all formulas.

A theoretical VI characteristic is assembled by evaluating (2.37) and (2.38) for both the electrons and ions, and adding the individual contributions to yield the total current. Figure 2.4 illustrates theoretical Langmuir results for several plasma temperatures. In the plot, the electrons and ions were assumed to be in thermal equilibrium; all parameters other than temperature were held constant. The plot is typical of a quiescent rarefied plasma. There is no distinct knee to indicate the location of the plasma potential (in the illustrated curves the plasma potential was set to 0[V]). Increasing plasma temperature has the effect of shifting the characteristic up and flattening the electron repelling region. This intuitively makes sense; higher

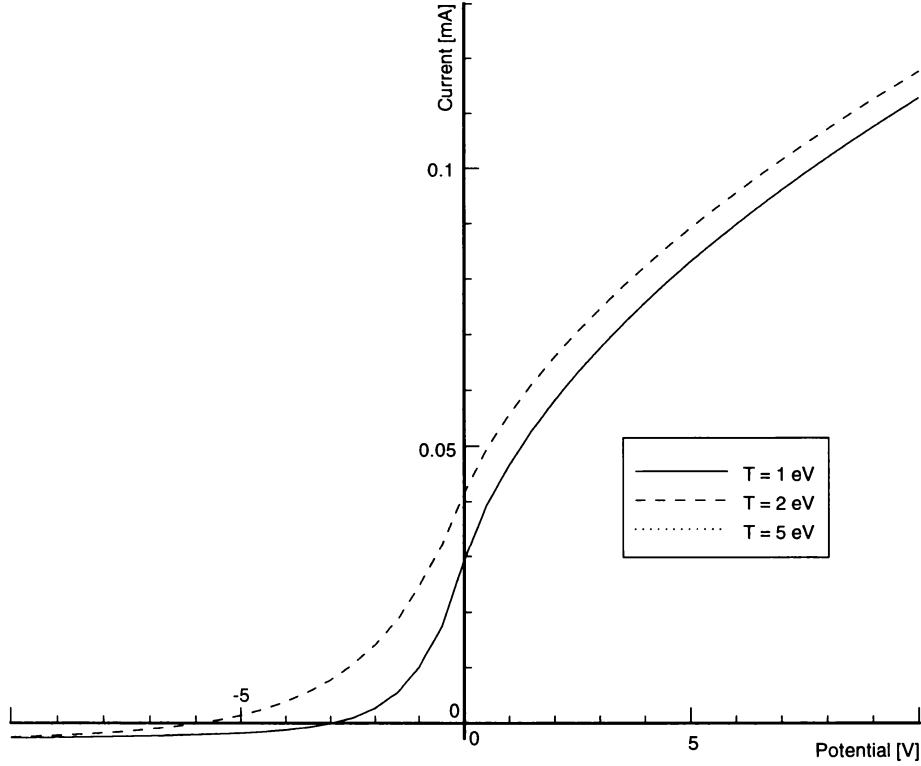


Figure 2.4: Langmuir theory VI characteristics for several temperatures.

temperatures are associated with higher energy electrons, which will have a greater ability to penetrate the potential barrier and sustain a dominant electron current to the probe.

The value of the Langmuir theory is that it enables us to determine the thermodynamic properties of experimental plasmas from their VI characteristic. Let us pretend that the data in Figure 2.4 is experimental data, that is, assume we know nothing about the temperature, number density, etc. associated with this pseudo-experimental data. Equation (2.35) has the general form (for electrons),

$$I_e = \beta j_e , \quad (2.39)$$

or, in the electron repelling region (using (2.37) and (2.38)),

$$I_e = \beta e^{\frac{e\phi_p}{kT_e}} . \quad (2.40)$$

Taking the natural logarithm of both sides of (2.40) we find

$$\ln(I_e) = \ln(\beta e^{\frac{e\phi_p}{kT_e}}) . \quad (2.41)$$

$$\Rightarrow \ln(I_e) = \frac{e}{kT_e} \phi_p + \ln(\beta) . \quad (2.42)$$

Equation (2.41) has the form of a straight line

$$y = \alpha x + b , \quad (2.43)$$

where,

$$y = \ln(I_e)$$

$$\alpha = \frac{e}{kT_e}$$

$$b = \ln(\beta) .$$

Notice that the only unknown value in α is the electron temperature, T_e . Thus, if we plot $\ln(I_e)$ as a function of ϕ_p , and pick off the value of the slope in the electron retarding region, we can immediately determine the temperature of the electrons. The natural logarithm of electron current for the same conditions used to produce Figure 2.4 is shown in Figure 2.5. Consider the lower curve, the solid line. From the figure we can estimate the slope in the electron repelling region,

$$\alpha = \frac{\Delta \ln(I_e)}{\Delta \phi_p} \approx \frac{10}{10} = 1 . \quad (2.44)$$

Thus, the estimated electron temperature is

$$T_e \approx \frac{e}{\alpha k} = 11594.2[k] = 1.0[eV] , \quad (2.45)$$

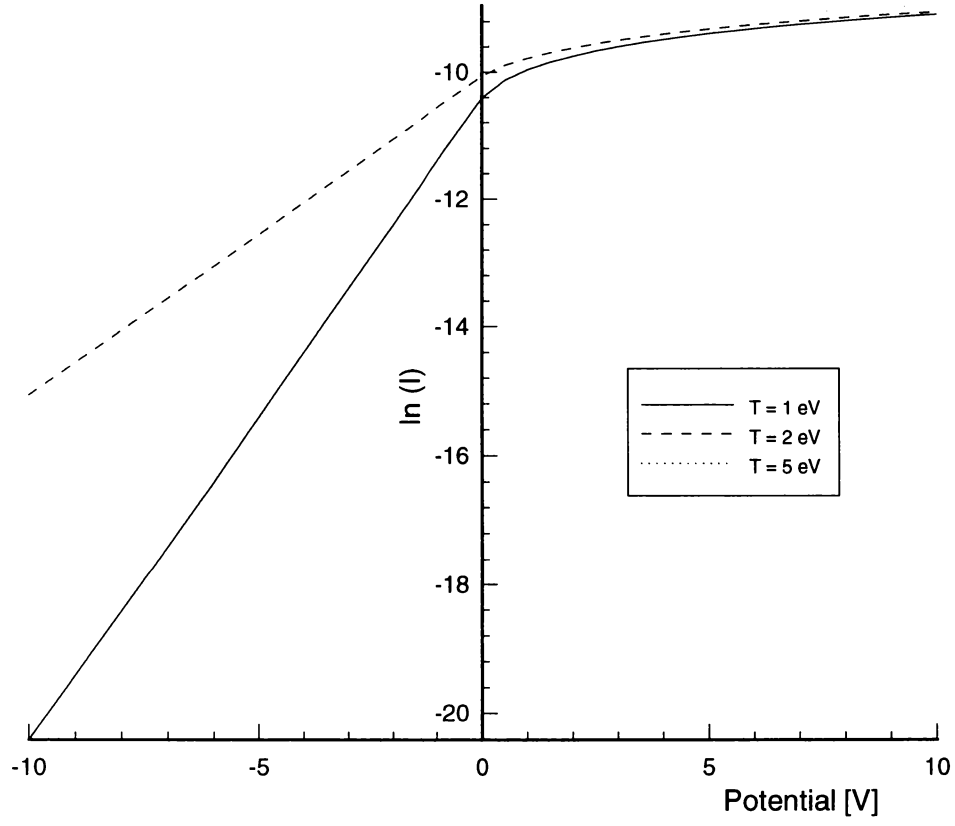


Figure 2.5: Natural logarithm of electron current vs. potential for several temperatures.

which corresponds directly to the actual temperature used to construct the characteristic initially.

The logarithmic plot also helps to identify the plasma potential. In Figure 2.5 the knee at 0 [V] is much more apparent than in Figure 2.4. If we know the plasma potential, then we can immediately identify the saturation current. At saturation, particles are unimpeded from moving toward the probe and therefore the collected charge is simply the thermal flux,

$$I = Ae \frac{n\bar{c}}{4}, \quad (2.46)$$

where,

$$\bar{c} = \sqrt{\frac{8kT}{\pi m}}. \quad (2.47)$$

Since we have already determined the plasma temperature, we can use the experimental value of saturation current and equation (2.44) to determine the number density.

In this section we have shown how the Langmuir and Mott-Smith theory may be used to determine the thermodynamic properties of a plasma in the OML limit. The theory provides important results even for plasma regimes which do not correspond to the OML limit; it provides an upper bound for the current collected by a probe under collisionless conditions. This is because potential barriers which result from sheath formation can only reduce the number of charged particles which are able to reach the probe.

2.2.2 Other Descriptions of Quiescent Probe Theory

It was mentioned earlier that no exact analytical theories exist for Langmuir probes; however, numerical solutions have been developed to solve the governing equations. The most widely accepted of these is that of Laframboise [5]. The techniques developed by Laframboise are sufficient for establishing the VI characteristics of cylindrical probes over essentially the entire range of conditions in which collisions can be neglected.

Laframboise developed a strategy to solve the collisionless Boltzmann equation. The assumption of a Maxwellian distribution for the attracted as well as repelled species results in a nonlinear system of integral equations. These equations must be solved numerically by an iterative procedure. The details of the procedure are quite complicated and may be found in the original paper.

Many experiments have shown the validity of the Laframboise formulation. Fig-

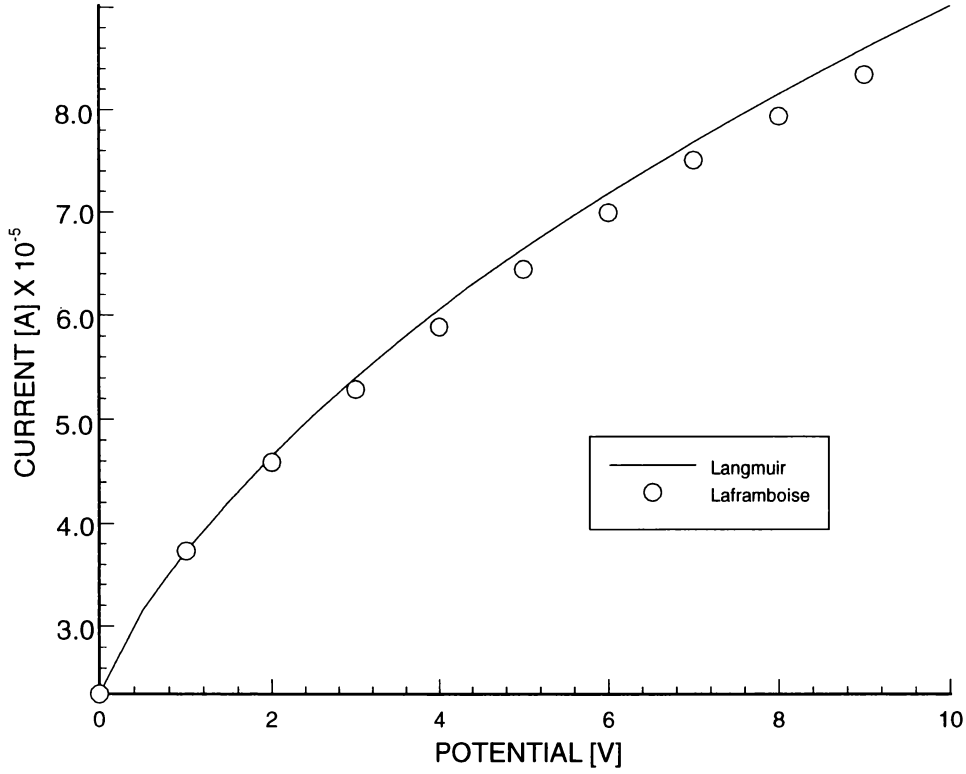


Figure 2.6: Comparison of Langmuir and Laframboise results in a rarefied plasma.

Figure 2.6 shows theoretical predictions of the Langmuir and Laframboise theory for identical plasma conditions. In general, the Laframboise formulation predicts lower current collection than predicted by the OML theory of Langmuir. The plasma represented in the figure is of very low density ($N = 8.0 \times 10^{-10} [\text{cm}^{-3}]$); plasmas at higher densities will show even greater deviation. This, of course, is expected since the previously unaccounted for sheath potential barriers are fully modeled in the Laframboise theory.

2.2.3 Probe Theory for a Flowing Plasma

We may wish to make Langmuir probe measurements in flowing plasmas (such as in the plume of an ion thruster), where the ions have a mean velocity much greater than

their thermal velocity. It has been found that electron collection will be essentially the same as in quiescent plasmas; however, ion collection mechanisms may be quite different.

If the directed energy of an ion beam is much greater than the thermal energy, it may be impossible for an ion sheath to be established. The situation is worst when the probe is aligned perfectly with the direction of the flow. Appealing to the Bohm stability criteria, which places a fundamental condition on the *normal* component of ion velocity, a probe aligned with the flow will see ions with predominantly tangential velocities, and consequently will be unable to form an ion sheath. As a result, the ion collection will be influenced most by geometric factors, that is, the exposed cross-sectional area of the probe normal to the flow direction. Therefore, we expect to collect a minimum amount of ion current when the probe is aligned with the flow (where the exposed cross-sectional area is essentially the probe tip cross-sectional area), and a maximum amount of ion current when the probe is perpendicular to the flow. Also, since the inertia of the directed ions may be quite large, the collected ion current may be independent of voltage.

The postulates put forth in the previous paragraph have been verified experimentally; however, one anomalous and initially overlooked effect has been found. When the probe is aligned with the flow, a phenomena called the “end effect” may come into play. The end effect leads to the counter-intuitive result that the ion current collection will be maximum when the probe is aligned with the direction of the flow. Experimental data illustrating this effect is shown in Figure 2.7, from Sonin [6]. The effect is found in plasma regimes in which the OML theory is applicable. In the OML regime, the effect of the sheath becomes small; that is, the probe potential may reach far into the bulk plasma and attract ions from many Debye lengths away. When the ions move in trajectories tangential to the probe surface (i. e. , when the probe is aligned with

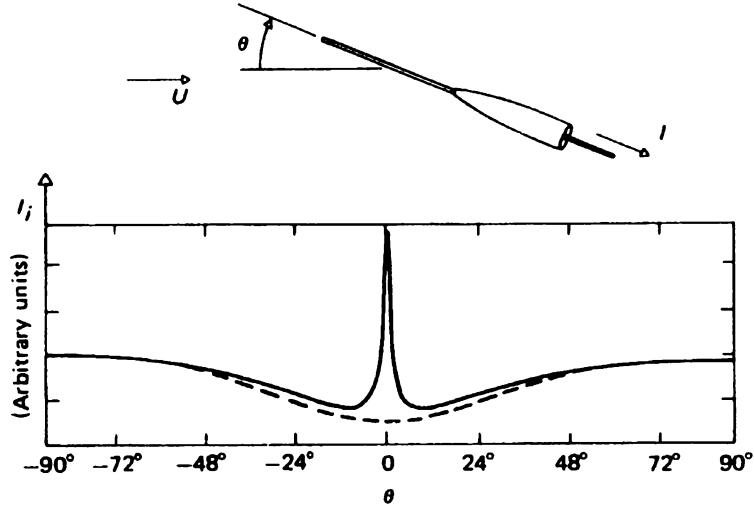


Figure 2.7: Illustration of the end effect. The figure shows the variation of ion current with angle of attack. The dashed line is theoretical, from Langmuir. The solid line is experimental data, from Hester and Sonin (reproduced with permission of Sonin).

the flow), they will spend the greatest amount of time (as compared to other probe orientations) under the influence of the attractive potential. Consequently, the probe may collect the greatest amount of current at zero angle of incidence. In this model, high aspect ratio (l_p/r_p) probes should be most vulnerable to the end effect. Perhaps the end effect should be more appropriately called the “thin probe effect”.

To avoid complications associated with the end effect, it may be experimentally preferable to orient the probe transverse to the flow. Theoretical prediction of the ion current is then straightforward, based simply on the side-view cross sectional area of the probe; however, interpretation of the electron current may be complicated because the upstream electron sheath may be obliterated by the high velocity ion flux. Therefore, calculating electron temperatures from the dissection of the electron-retarding portion of the VI characteristic must include a velocity correction. It is the author’s opinion that the introduction of such “fudge factors” belies the complexity of

the situation and should probably be avoided altogether.

This chapter has merely touched the surface of the theory that has been developed for the interpretation of Langmuir probe data. The vast amount of treatments available is evidence of the complexity of the problem (the papers by Chen [19] and Chung [21] give excellent summaries of the most important work), and justifies the present work — a direct particle simulation. What has been presented here is merely that information necessary to make statements about the validity of the PROBEPIC results. In the following two chapters the theory behind and development of the PROBEPIC code is described in detail.

Chapter 3

The Particle-In-Cell Technique

This chapter introduces the fundamentals of the particle-in-cell (PIC) technique. The basic strategy of the technique is outlined on an elementary level to enable the reader who is unfamiliar with PIC to move on to the latter chapters without having to consult other references. The various elements of the method will be clarified by specific examples. These examples will elucidate PIC techniques in general as well as establish a theoretical framework for the algorithms used in PROBEPIC, as presented in Chapter 4. The interested reader may consult the standard references [7, 8] for greater detail.

3.1 What Is PIC ?

PIC is a computational method used primarily for the simulation of plasma phenomena. The acronym PIC describes the major principles embodied in the technique. *Particle*: PIC uses many discrete particles to simulate the collective behavior of a plasma; in this sense it attempts to simulate reality as closely as possible. *Cell*: a spatial grid on which the electromagnetic fields are computed is superimposed on the computational domain; particles in the cells formed by the grid transfer part of their “identity” to the nodes that bound the cell. This eliminates the need to explicitly

compute the interaction of a given particle with every other particle. Instead the electromagnetic field is determined by using equivalent charge and current densities at the nodes only, which drastically reduces the complexity of the field calculation (there are generally many more particles than grid-points). Figure 3.1 is a schematic of the objects present in any PIC code. The particles and boundaries represent real physical entities, while the grid is a purely mathematical object.

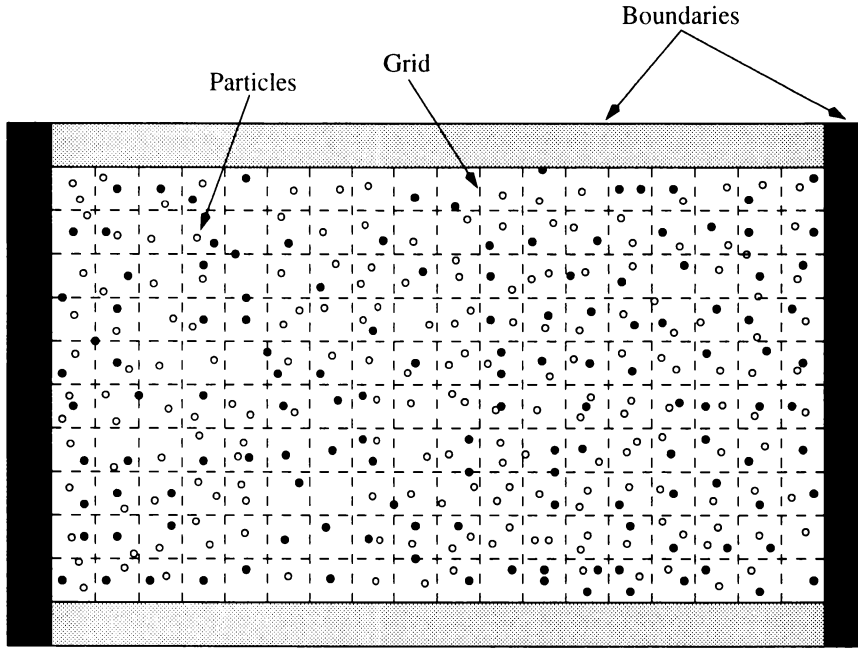


Figure 3.1: Schematic of PIC objects.

The first and most obvious question one might ask is, “How can we expect to simulate plasmas particle-by-particle, which even at low densities would require billions of particles.” The answer is *superparticles*. Superparticles are computational particles which represent many real particles. For example, one superparticle might represent a million or more real electrons. In this way PIC becomes feasible on today’s computers.

The plasma regimes for which PIC is applicable are limited primarily by number

density. For number densities greater than approximately $10^{13}[\text{cm}^{-3}]$ the number of superparticles required to produce reasonable statistics becomes computationally prohibitive; hence, the extension of PIC into denser plasma regimes is contingent upon the development of faster computers.

The *time-step* determines how far superparticles will move under the influence of the electromagnetic field in one computational cycle. A complication often arises in determining a suitable time-step. The disparity in the relative masses of electrons and heavy particles creates a situation in which electrons will move much (perhaps thousands, or millions of times) further in a given time-step than the ions – we have two different time scales. As a result a lot of computational time is wasted resolving electron oscillations while waiting for the ions to catch up. If one is only interested in modeling the response of a slightly perturbed plasma and observing the resulting plasma oscillations the ions may be treated as “frozen”, that is, fixed. The time-step will then be determined by the desired resolution of the plasma frequency of the electrons. However, when the mass motion of the ions is important, such as in the modeling of ion acoustic waves or a Langmuir probe, schemes must be developed to deal with the difference in the time scales. One such scheme is to *sub-cycle* the electrons; the electrons are allowed to move many time-steps while the ions are held stationary; the ions then move a single time-step on their own time-scale. Sub-cycling may destroy information concerning electron oscillations, but if the goal is to simulate the macroscopic motion of the plasma, suitable results may be obtained. An even more exotic technique is to use a “hybrid” code in which the electrons are treated as a fluid (see, for example, Fife [9]) and computations are carried out on the ion time-scale. If we are interested in modeling a plasma which is both oscillating and flowing we are left with no alternative but to accept that the simulation will take a long time.

3.2 The Computational Cycle

A PIC program is composed of several modules that work together to move particles through phase space. Each of these modules are executed once in a computational time step for each particle species. These modules are: the charge weighter, the field solver, the force weighter, and the mover. The order of execution of these tasks for an electrostatic simulation is illustrated in Figure 3.2.

Let us describe, in a general sense, one computational cycle; a more complete description of each module will follow in sections 3.2.1-3.2.4. A PIC program begins by initializing particle positions and velocities. From these initial conditions the charge density at the grid-points is computed. The Poisson equation is then solved yielding the electric field at the grid-points. The field at the grid-points is then interpolated back to the particles (i. e. the electric field at each particle position is computed). The final step is to integrate the Lorentz force equation to determine the new position and velocity for each particle. The process then starts over for the next time-step.

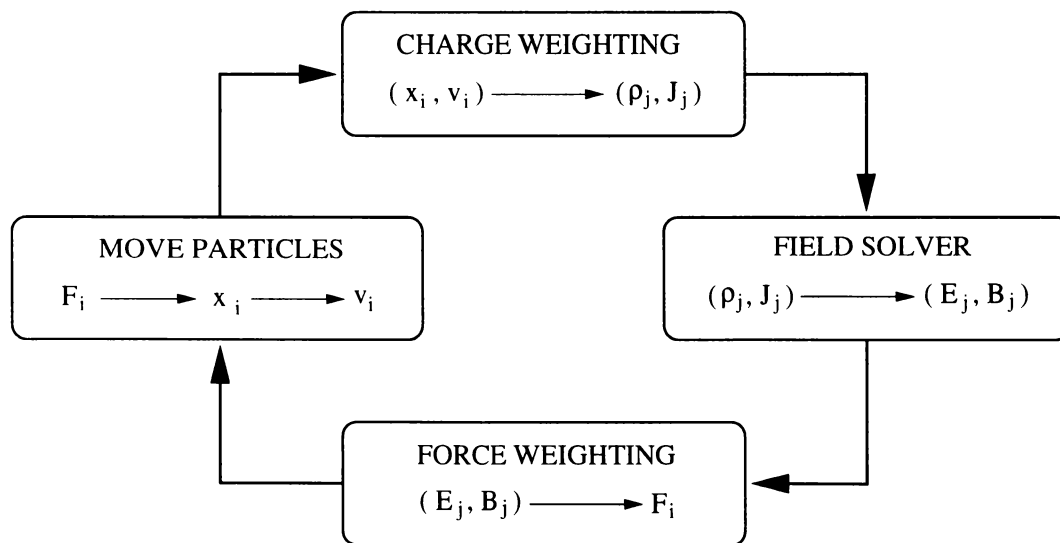


Figure 3.2: A typical cycle (one time step) in a PIC simulation.

The four steps described above are found in all PIC programs. Most programs will also include modules to handle boundary interactions and output diagnostics. Also, other modules may be included to handle interactions such as collisions. The beauty of the PIC technique is that new interactions may be almost seamlessly integrated into the program once the four core modules are operational; the problem does not require a complete theoretical reformulation. We will now consider the core PIC modules individually in greater detail. For simplicity we will consider only an electrostatic model; however, the principles discussed apply in general for a full electromagnetic model once the Maxwell equations have been decoupled under a suitable gauge transformation.

3.2.1 Charge Weighting

The use of a computational grid considerably simplifies the computation of the electric field. It is much more efficient to solve Poisson's equation on N_g grid-points rather than computing $N_p!$ Coulomb interactions between N_p particles (as, in general, $N_g \ll N_p$).

The task is then to develop methods for transferring attributes (e. g. charge) from the particles to the grid; this is the subject of *weighting*. The term weighting implies some sort of interpolation. Various interpolation schemes will now be explored.

Consider first, for simplicity, a system in which particles are constrained to move in one dimension. The simplest interpolation scheme is to assign the charge of a particle to the nearest grid-point (the so-called NGP method). Also referred to as zero-order weighting, the NGP method produces an effective charge density on the grid as pictured in Figure 3.3a). It is clear that as this particle, of width $\Delta X \equiv |X_j - X_{j-1}|$, moves along the x-axis it causes a discontinuous jump in the grid charge density. In turn, the spatial and temporal behavior of the electric field will be noisy. Consequently

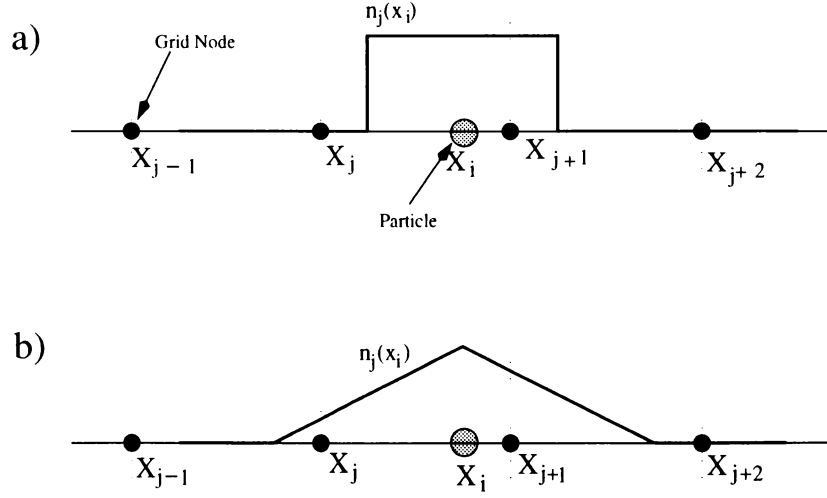


Figure 3.3: Particle charge weighting. a) NGP weighting. b) First-order linear particle weighting

the NGP weighting, while being computationally efficient, does not give satisfactory macroscopic results.

Thus, we seek an interpolation scheme which more uniformly distributes the particle's charge over space. First-order linear weighting allows the particle's charge to be distributed to not only one adjacent grid-point, but to its two nearest neighbors, as pictured in Figure 3.3b). The effective particle width then becomes $2\Delta X$, and the charge assigned to each grid-point is linearly related to the separation distance between the particle and the grid-point. For example, if the particle is at x_i , between grid-points at X_j and X_{j+1} , then,

$$q_j = q_i \left[\frac{\Delta X - (x_i - X_j)}{\Delta X} \right] = q_i \left[\frac{X_{j+1} - x_i}{\Delta X} \right] \quad (3.1)$$

$$q_{j+1} = q_i \left[\frac{x_i - X_j}{\Delta X} \right] . \quad (3.2)$$

We may choose to use even higher order weighting schemes to get a smoother charge distribution on the grid; higher order schemes might use quadratic or cubic splines. In general the effective shape of the particle is called the shape function $S(x)$, and the grid charge density may be written

$$\rho_j \equiv \rho(X_j) = \sum_i q_i S(X_j - x_i) , \quad (3.3)$$

for i charges in cells adjacent to X_j . Higher order interpolation is seldom used, as it becomes too computationally expensive.

Two properties of any shape function are desired: *charge conservation* and absence of a *self-force*. Charge conservation is satisfied if the sum of the weighted node charges is equal to the particle charge. For example, for the first-order weighting described above,

$$q_j + q_{j+1} = \frac{q_i}{\Delta X} [X_{j+1} - x_i + x_i - X_j] = q_i \frac{\Delta X}{\Delta X} = q_i . \quad (3.4)$$

A self-force occurs when the electric field, computed using the weighted node charges for a single particle, is nonzero at the particle's position (i. e. the particle exerts a force on itself). Again, for the first order weighting, the electric field at x_i due to the weighted node charges at X_j and X_{j+1} is:

$$E_i = q_i \left[\frac{q_j}{(x_i - X_j)^2} - \frac{q_{j+1}}{(X_{j+1} - x_i)^2} \right] \quad (3.5)$$

which, using equations (3.1) and (3.2), yields

$$E_i = q_i^2 \frac{(X_j - 2x_i + X_{j+1})(X_j^2 - X_j x_i + x_i^2 - X_j X_{j+1} + 1 - x_i X_{j+1} + X_{j+1}^2)}{(X_{j+1} - X_j)(x_i - X_j)^2 (X_{j+1} - x_i)^2} . \quad (3.6)$$

Thus, in general, first-order linear weighting produces a finite electric field, or self-force, at the particle position. Figure 3.4 shows the general behavior of (3.6). The

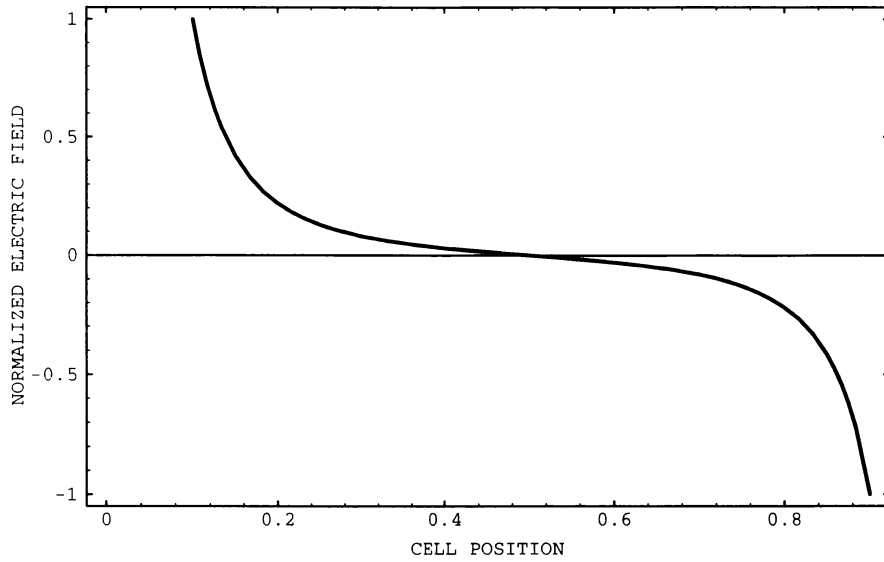


Figure 3.4: Electric Field generated by the weighted charge of a single particle as a function of the particle’s position within the cell.

self-force is small when the particle is near the center of a cell, and diverges for particle positions near the grid nodes. Further, as a particle traverses a cell it first experiences a repulsive force, but after it passes the cell center it experiences an attractive force; therefore, one might expect that the self-force effects will “wash out” provided the particle stays in one cell for two or more time-steps.

To further discuss the more philosophical issues of whether a weighting scheme is “better” if it conserves charge *or* produces no self-force is beyond the scope of this thesis. Of course we would hope to find a shape function that achieves both ends. Unfortunately, it is unclear whether or not such an object has been developed.

3.2.2 Field Solver

In the electrostatic model the Maxwell equations reduce to only one equation – the differential form of Coulomb’s law,

$$\vec{\nabla} \cdot \vec{\mathbf{E}}(\vec{\mathbf{x}}) = \frac{\rho(\vec{\mathbf{x}})}{\epsilon(\vec{\mathbf{x}})} , \quad (3.7)$$

where $\vec{\mathbf{E}}(\vec{\mathbf{x}})$ is the electric field intensity, $\rho(\vec{\mathbf{x}})$ is the charge density, and $\epsilon(\vec{\mathbf{x}})$ is the electric permittivity of the medium. Since $\vec{\mathbf{E}}(\vec{\mathbf{x}})$ is irrotational, $\vec{\mathbf{E}}(\vec{\mathbf{x}})$ may be written as the gradient of some scalar potential $\phi(\vec{\mathbf{x}})$:

$$\vec{\mathbf{E}}(\vec{\mathbf{x}}) = -\vec{\nabla}\phi(\vec{\mathbf{x}}). \quad (3.8)$$

Combining (3.7) and (3.8) gives

$$\vec{\nabla}^2\phi(\vec{\mathbf{x}}) = -\frac{\rho(\vec{\mathbf{x}})}{\epsilon(\vec{\mathbf{x}})} , \quad (3.9)$$

which is of the form of Poisson’s equation. The approach to determine the electrostatic field is to solve (3.9) for $\phi(\vec{\mathbf{x}})$ numerically with prescribed boundary conditions and charge density known at the grid nodes from previous application of the charge weighting algorithm. Once $\phi(\vec{\mathbf{x}})$ is known we may compute $\vec{\mathbf{E}}(\vec{\mathbf{x}})$ using (3.8).

The technique used to numerically solve Poisson’s equation is highly problem dependent. Factors that influence the choice of a particular solver include:

- Symmetries in the computational domain.
- Cartesian or curvilinear coordinates.
- Desired accuracy.
- Available computer memory.

If the computational domain is completely symmetrical, with periodic boundary conditions, fast Fourier transforms may be employed to produce a very fast field solver. In general the boundary conditions are not periodic and the differencing of (3.9) leads to very large, sparse matrices. The literature available on sparse matrix technology is voluminous (e. g. , see [10, 11]). The first decision one must make is the degree of accuracy desired. This leads one to choose an *exact solver*, which uses “brute force” to factor the matrix and give an exact (i. e. analytical) result (at least, to machine precision), or an *inexact solver*, which uses an approximate factorization to greatly reduce the number of required computations (and consequently gives a result that differs to some degree with the analytical result). The modeler must decide what accuracy/speed tradeoffs he is willing to make. A final, non-trivial, consideration is the amount of physical RAM available on the target machine. Some field solvers may require 50-100 MB of storage for the difference matrix alone. If the machine is not equipped with enough RAM it will utilize virtual memory and computations will be slowed.

Let us consider a simple example to clarify the points discussed above. Consider again a system in which the particles are constrained to move in one dimension in which case (3.8) and (3.9) may be written

$$E_x = -\frac{d\phi}{dx} , \quad (3.10)$$

and,

$$\frac{d^2\phi}{dx^2} = -\frac{\rho}{\epsilon} . \quad (3.11)$$

One approach is to use the finite difference forms of (3.10) and (3.11)

$$E_j = \frac{\phi_{j-1} - \phi_{j+1}}{2\Delta x} \quad (3.12)$$

$$\frac{\phi_{j-1} - 2\phi_j + \phi_{j+1}}{(\Delta x)^2} = -\frac{\rho}{\epsilon} , \quad (3.13)$$

where j is the running grid node index ($j=1,2, \dots, N$). Equation (3.13) can be written more compactly as

$$[\mathbf{A}]\vec{\phi} = -\frac{(\Delta x)^2}{\epsilon}\vec{\rho}, \quad (3.14)$$

where $[\mathbf{A}]$ is a $N \times N$ coefficient matrix, $\vec{\rho}$ is a N -dimensional column vector of source terms, and $\vec{\phi}$ is a column vector of N unknown potentials. To find $\vec{\phi}$ we operate on both sides of (3.14) with $[\mathbf{A}]^{-1}$, that is, invert $[\mathbf{A}]$. For the problem at hand, $[\mathbf{A}]$ is tridiagonal and, fortunately, many fast algorithms exist to invert tridiagonal matrices. Once the ϕ_j have been computed, (3.12) is evaluated to determine the electric field at the grid-points.

The field solver is generally called at every time-step; therefore, optimizing the field solver for each application is important if computational bottlenecks are to be avoided.

3.2.3 Force Weighting

Force weighting is the process of interpolating the known electrostatic field intensities at the grid nodes back to the particles, that is, it is the process of “telling” each particle what field it is seeing as a result of the field solver computation. But this is simply the inverse of the process used in the charge weighting algorithm to transfer information (charge) from the particles to the grid. In this case the information (field intensity) is transferred from the grid to the particles. Consequently, the same weighting algorithms that were discussed in section 4.2.1 may be applied directly to the force weighting problem. All of the conclusions regarding the suitability of a given weighting shape function remain unaltered.

In order to avoid (or at least reduce) self forces, it is desirable to use the same weighting shape function in both charge density and force calculations. So, for example, the compatible force weighting algorithm to the first-order linear charge weight-

ing would be (see (3.1) and (3.2)),

$$E(x_i) = \left[\frac{X_{j+1} - x_i}{\Delta X} \right] E_j + \left[\frac{x_i - X_j}{\Delta X} \right] E_{j+1} . \quad (3.15)$$

A “trick of the trade” may be pointed out here. As part of the charge weighting algorithm, the cell position will be computed (i. e. we determine which cell the particle is in). This information is needed again here. It therefore makes sense to *store* the position first in the charge weighting subroutine and simply retrieve it from memory here rather than recomputing it. The computational savings becomes pronounced in two and three dimensional codes which require large, computationally expensive subroutines to determine the cell location of a particle.

3.2.4 Integration of the Equations of Motion

Particles in an electrostatic PIC code move as a result of thermal excitation, Coulomb interactions with other particles, and the presence of electrodes biased with respect to the plasma potential. In the previous sections we discussed how the electrostatic force on each particle is computed. In this section we describe how that force changes the particles position in six-dimensional phase space, that is, how its position and velocity are updated during each time-step.

The Lorentz force equation in the electrostatic model reduces to

$$\vec{\mathbf{F}} = q\vec{\mathbf{E}} , \quad (3.16)$$

where $\vec{\mathbf{F}}$ is the electrostatic force on the particle, q is the superparticle charge, and $\vec{\mathbf{E}}$ is the electrostatic field intensity at the particle position (as determined in the force weighting algorithm). When combined with Newton’s second law and the definition of velocity, (3.16) yields two first order differential equations to be integrated separately for each particle

$$\frac{d\vec{\mathbf{v}}}{dt} = \frac{q}{m}\vec{\mathbf{E}} \quad (3.17)$$

$$\frac{d\vec{x}}{dt} = \vec{v} . \quad (3.18)$$

Two initial conditions are needed to solve these two first-order equations: the position and velocity at the end of the previous time-step.

Equations (3.17) and (3.18) can be solved to any degree of accuracy. That is, one can use a higher order scheme (such as Runge-Kutta [12]) or opt for a simpler first-order scheme, again trading accuracy for speed. The trend is to use a computationally efficient first-order scheme by replacing (3.17) and (3.18) by the finite-difference equations,

$$\frac{\vec{v}_{\text{new}} - \vec{v}_{\text{old}}}{\Delta t} = \frac{q}{m} \vec{E}_{\text{old}} , \quad (3.19)$$

$$\frac{\vec{x}_{\text{new}} - \vec{x}_{\text{old}}}{\Delta t} = \vec{v}_{\text{new}} , \quad (3.20)$$

or,

$$\vec{v}_{\text{new}} = \vec{v}_{\text{old}} + \left[\frac{q}{m} \vec{E}_{\text{old}} \right] \Delta t , \quad (3.21)$$

$$\vec{x}_{\text{new}} = \vec{x}_{\text{old}} + \vec{v}_{\text{new}} \Delta t , \quad (3.22)$$

where Δt is the time-step. This method has vanishing error as $\Delta t \searrow 0$.

The selection of an appropriate time-step, Δt , is problem dependent. For simulations which involve semi-infinite computational domains where only plasma oscillations are to be observed, it can be shown [7] that setting $\omega_o \Delta t \leq 0.3$ (where ω_o is the characteristic frequency, for example, the plasma frequency) results in small amplitude and phase error for some tens of cycles. In plasma devices (i.e. simulations which involve boundaries such as conductors) the situation is quite different. Electrodes in the computational domain may create field strengths on the order of 1000 [V/cm] or more which will rapidly accelerate particles. Consequently, the time-step must be chosen such that particles will not move distances greater than the dimensions of the physical boundaries. For example, in a Langmuir probe simulation we cannot allow a particle

near the probe to move from one side of the probe to the other (i. e. through the probe) without being collected. We must resolve its motion on a fine enough time scale to assure that several points in its trajectory will lie inside the probe so that boundary subroutines will be able to remove the particle and increment the current. The author has found that this restricts $\omega_o \Delta t$ to values at least two orders of magnitude smaller than is needed to resolve plasma oscillations alone.

3.3 Boundary and Initial Conditions

The previous section described the core PIC modules. These modules are developed independent of any particular problem. The physics of a specific simulation (e. g. a Langmuir probe simulation) must be introduced in the form of boundary conditions. If the physics of these conditions is incorrect, even if the core PIC modules function flawlessly, the ultimate results will be worthless. The careful process of preliminary design may be divided into three categories: general considerations, boundary conditions, and initialization conditions. Actual implementation of these concepts will be elaborated on further in chapter 5, where the PROBEPIC algorithms are developed.

General considerations at the beginning of the simulation design process include:

- Size of computational domain.
- Number of grid-points.
- Number of particles.
- Position of objects (conductors, insulators, etc.) relative to the boundaries.

The size of the computational domain must be large enough to encompass the interaction which is to be observed. For example, in the Langmuir probe simulation we are interested in simulating the formation of a plasma sheath around the probe;

therefore, the computational domain size must be chosen to be on the order of several Debye lengths to allow the relevant fields to be established (i. e. Debye shielding).

The number of grid-points is determined by the Debye length. Since perturbations of the plasma will result in electron oscillations on the order of a Debye length, the grid must contain at least two grid-points per Debye length for these oscillations to be resolved.

The number of particles in a simulation determines its statistical accuracy. The influence of the self-force which results from the charge weighting process can be made to “wash out” as progressively more particles are added. It is a “rule of thumb” that about ten or more particles should be introduced for each computational grid cell to give acceptable statistics. So, for example, a 10×10 computational grid would require 1000 or more particles. Also, the author has found that when dealing with systems which contain disparate particle masses (e. g. electrons and heavy ions), one should use a disproportionately large number of heavy particles in the simulation if their contribution to macroscopic results is to be accurately modeled. This is because the relative immobility of the heavy particles results in infrequent collisions with conducting surfaces; increasing the number of heavy super-particles increases the frequency of collisions, and consequently gives less noisy statistical results.

Attention must be paid to the positioning of objects within the computational domain in plasma device simulations. Objects should not be placed near boundaries which would result in non-physical effects. For example, if current collection on an electrode is being simulated, the electrode should not be placed near a computational boundary where particles are being introduced (fluxed across). Rather, it should be placed near the center of the computational domain, where particles have had a chance to thermalize, and the effects of the (non-physical) computational boundary have had time to relax.

Boundary conditions determine how the plasma interacts with its surroundings. Boundary conditions include:

- Particle fluxing across boundaries of the computational domain.
- Surface, or material interactions of the plasma with objects in the computational domain.
- Electromagnetic field boundary conditions.

If a PIC simulation models a small plasma region that is actually contained by a much larger plasma, the flux of particles across the computational boundary must be modeled. This requires the development of a statistical model for both the velocity and angular distribution of particles entering the region to be simulated. Since the plasma outside of the computational domain is usually assumed to be neutral, adequate charge must be fluxed into the computational domain to account for particles leaving through the boundaries.

In plasma device simulations one is usually interested in modeling how the plasma behaves as an active circuit element. The plasma interacts with the external circuit through contact with conducting surfaces within the computational domain. Thus, we must accurately model the physics of this interaction. For example, a particle collision with a conducting surface might contribute to the current in the external circuit, or perhaps cause secondary emission. Insulating surfaces might serve as catalytic surfaces for re-combination. In any event, we must be sure that surface interactions which occur in the simulation do not violate fundamental physical laws, such as matter/energy conservation, if the macroscopic results are to model the physical process accurately.

A final set of boundary conditions to be considered are those used in the solution of the Poisson equation for the electric field. These boundary conditions enforce, for

example, the absence of tangential electric fields at the surface of conductors, or define bias voltages for conducting surfaces.

Initializing a PIC code involves choosing an initial distribution function for each species, $f_\alpha(\vec{x}, \vec{v}, t = 0)$, including any initial perturbation, and distributing the particles throughout the computational domain to start the simulation.

Clearly the design work to correctly model the relevant boundary conditions is time well spent. In fact, it may be said that the computational results *are*, figuratively speaking, the boundary conditions; therefore, if hasty guesswork is involved in establishing the boundary conditions one may as well guess the results and not attempt a computer simulation at all.

Chapter 4

PROBEPIC

PROBEPIC is a 2D/3V (the electric field has two components while the particles are free to move in three dimensions) PIC code designed to simulate the behavior of a Langmuir probe in both quiescent and flowing plasmas. It is a “pure” PIC code in that it does not introduce any fluid modeling such as is found in a hybrid code. The theory presented in Chapter 4 will now be applied to an actual simulation. A few general remarks about the code will be made before the various algorithms are dissected in detail in the subsequent sections.

The programming philosophy was to develop a concise, intuitively accessible code with as little redundancy as possible. The C programming language was most compatible with these goals because: it allows for longer variable name lengths than FORTRAN, it allows for complicated data structures, and it gives the programmer control over dynamic memory allocation. PROBEPIC is made up of twenty-three independent subprograms, each of which perform an individual task (e. g. mover, injector, etc.). These functions were designed to be independent of any external variables. For example, the mover can move electrons, ions, or Π^+ mesons — by input of pointers to relevant field and dynamical variables, it performs the appropriate modifications to the pointer values. A lean executable file is not necessarily an indication of computational efficiency; a code can often be made to run faster by *storing* certain variables

that are used frequently rather than *recomputing* them over and over. For example, the computational time required to generate the LU decomposition of the difference matrix for the electric field solver takes about an hour on a Sun Sparc2000. This matrix remains unchanged and is used in every subsequent time step. Clearly, storing this matrix rather than recomputing it thousands of times makes good sense. The tradeoff is that the matrix occupies about sixty megabytes of RAM throughout the simulation. This size/speed tradeoff was exploited in many of the functions that make up PROBEPIC.

PROBEPIC requires two non-standard libraries to compile: LAPACK [22] (linear algebra routines used by the field solver), and VOGL [23] (graphical display routines). The latter is not necessary if graphical output is not desired (graphical output is generally suppressed except in special circumstances such as debugging or taking “snapshots”). PROBEPIC requires extensive computational resources. About 80-150MB of RAM is required. Approximately two data points can be generated per twenty-four hour period on a nominal 125 megaflop machine.

In the following sections the core PROBEPIC functions will be described in detail. In each case application of the theory developed in the beginning chapters and programming details will be described. A complete listing of the program source code may be found in Appendix B.

4.1 Computational Domain Layout

The computational domain in PROBEPIC consists of a cylindrical region which contains a cylindrical Langmuir probe, as shown in Figure 4.1. The probe itself consists of a cylindrical conducting wire partially covered by an alumina insulator. During a simulation the remainder of the computational domain is filled with approximately

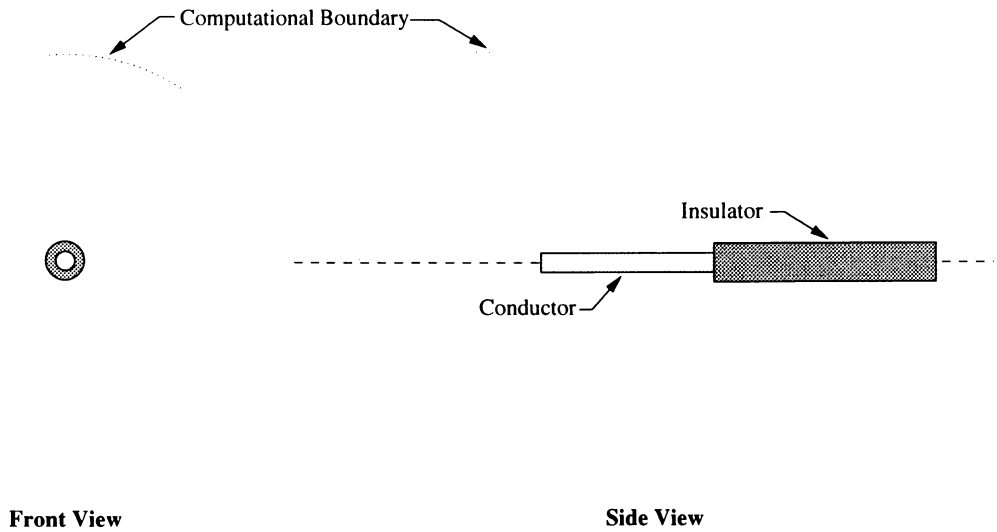


Figure 4.1: Schematic of PROBEPIC computational domain layout.

two-hundred thousand particles.

The dimensions of the computational boundary were chosen to provide ample room for a plasma sheath to form around the probe. Since these dimensions are not known a priori, they were determined as part of the initial design work. PROBEPIC was run several times with progressively larger computational domain sizes until the result (i.e. the current to the probe) stabilized.

A “snap-shot” of PROBEPIC in operation is shown in Figure 4.2. The red super-particles represent electrons, the green, ions.

4.2 PROBEPIC Code Details

All PROBEPIC sub-programs along with their function are listed in Table 4.1. In the following sub-sections both the theory and C implementation of the core PROBEPIC sub-programs will be presented side-by-side. This will enable future users of the code to understand what techniques were used in its development without having to dig through the source code. Where code is not explicitly shown, the reader may reference

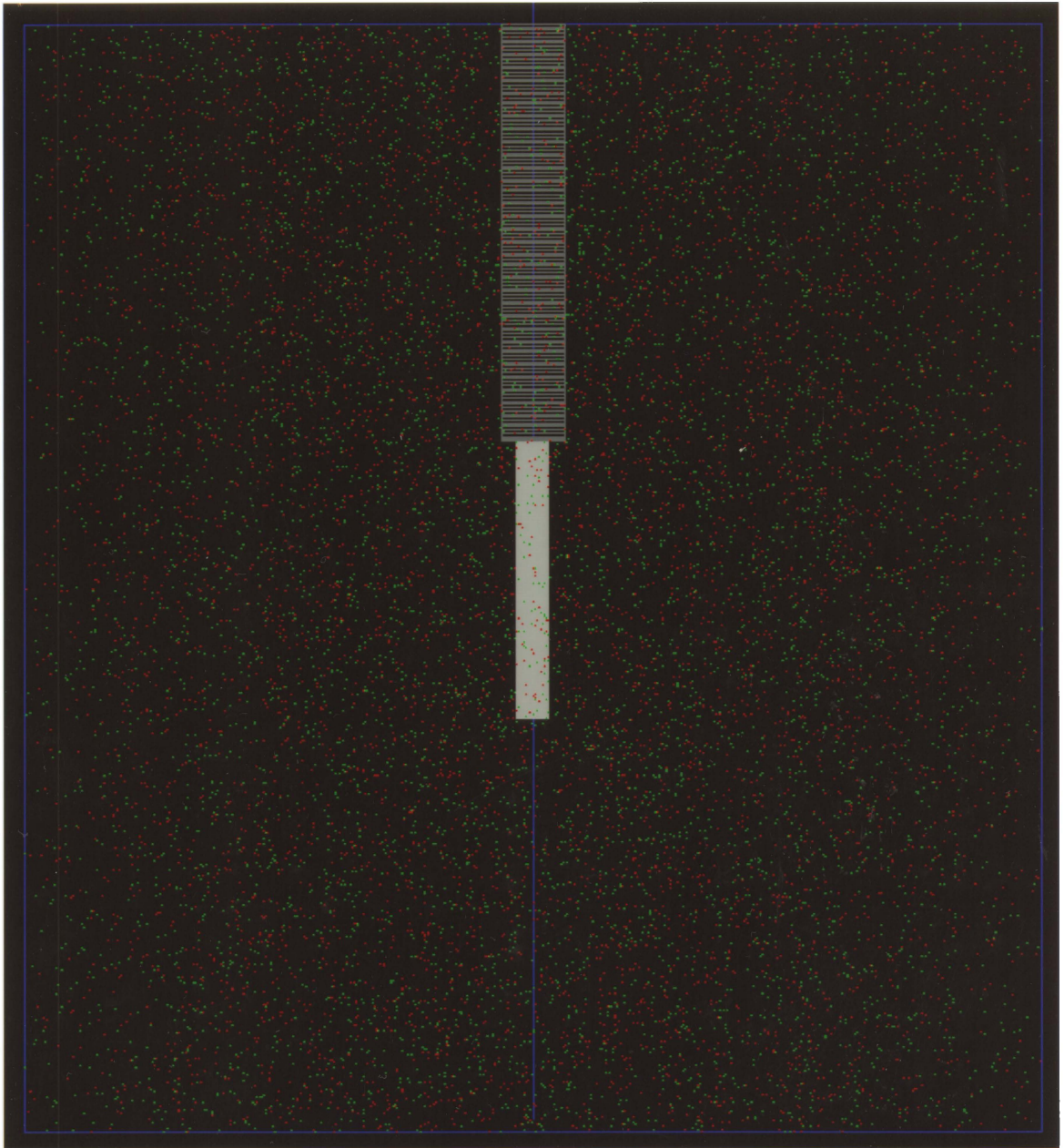


Figure 4.2: Snap-shot of PROBEPIC simulation.

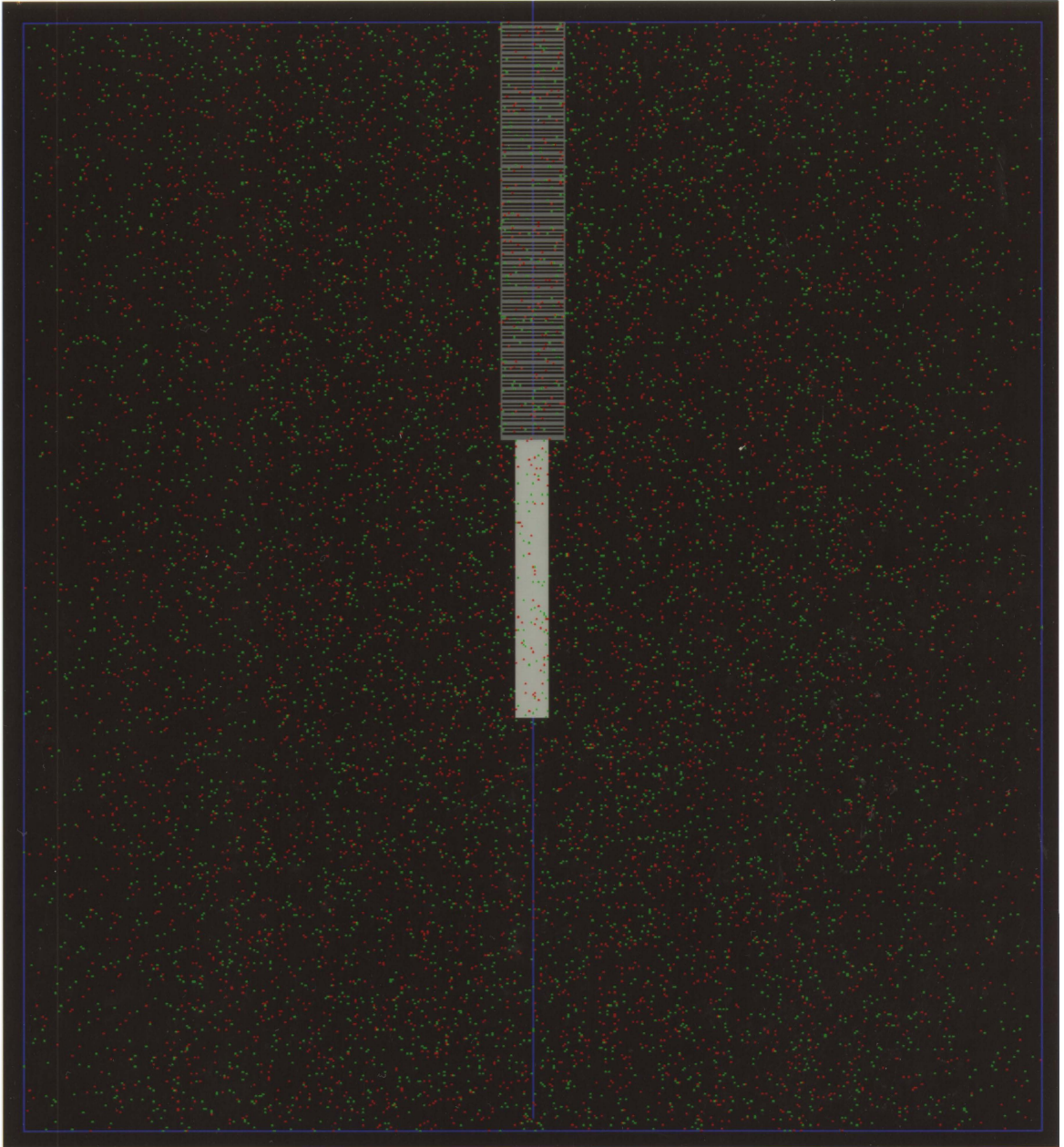


Figure 4.2: Snap-shot of PROBEPIC simulation.

Table 4.1: PROBEPIC sub-program description.

Sub-program	Function
boundary.c	Handles particle/boundary interactions for thermal particles.
boundary_beam.c	Handles particle/boundary interactions for beam particles.
charge_weight.c	The charge weighter.
field_solver.c	The electric field solver.
force_weight.c	The force weighter.
graphics.c	Graphical diagnostic output.
grid.c	Generates the computational grid.
initialize.c	Distributes the initial loading of thermal superparticles throughout the computational domain.
initialize_beam.c	Distributes the initial loading of beam superparticles throughout the computational domain.
inject_beam.c	Fluxes beam particles across boundaries into the computational domain.
inject_part.c	Fluxes thermal particles across boundaries into the computational domain.
locate.c	Determines what grid-cell a particle is in.
make_LU.c	Performs LU decomposition of difference matrix for subsequent use in field solver.
make_velocity_table.c	Creates arrays of velocities with for fluxing into the computational domain.
maxwell.c	Creates arrays of velocities with a Maxwellian distribution.
mover.c	The mover.
output_data.c	Generates PROBEPIC output file.
parameter.c	Determines some necessary parameters used in other sub-programs from initial plasma conditions.
probepic.c	PROBEPIC main program.
probepic.h	Header file for PROBEPIC.
random_number.c	Provides uniform random numbers for all PROBEPIC sub-programs.
reset_grid.c	Resets charge density, electric field, etc. at grid-points to zero at beginning of each time-step.

Appendix A to see how a particular expression was implemented in the source code (the subsections are labeled with the appropriate sub-programs for each task). These subsections will also serve a pedagogical role; the PIC methodology presented in Chapter 3 will be expanded upon, and in some instances, novel approaches to PIC problems which were developed for PROBEPIC will be debuted.

4.2.1 General Remarks (probepic.h)

A brief word about PROBEPIC's header file is in order because it includes some of the codes overall design philosophy. Structures were used in the program for clarity. For example, particles(electrons, ions, etc.) are of the type PARTICLEDEF — which has the associated properties of position, velocity, charge, etc. . The actual structure definition looks like:

```
typedef struct {
    FLOAT x,y,z,r,vx,vy,vz,Ex,Ey,q,m;
    unsigned long cellx,celly;
}PARTICLEDEF;
```

An array of say, fifty, electrons would be created by the declaration

```
PARTICLEDEF electron[50];
```

so that subsequent modification of, say, the x-component of velocity of the i^{th} electron has the form

```
electron[i].vx=...;
```

Large arrays were declared as pointers to allow their dimension to be set later using malloc(). This has the advantage of allowing large amounts of memory to be dynamically allocated or de-allocated during execution.

The physics used in the simulation may be characterized as classical and non-relativistic. The particles in the simulation, which are fermions, have high enough energies to safely assume that Fermi-Dirac statistics reduce to Maxwell-Boltzmann statistics. The maximum particle velocities are on the order of $0.01\ c$, so that relativistic effects may be ignored, although they can be readily included when required.

The PIC-specific features of PROBEPIC that are not typically found in other PIC simulations are: the method used in fluxing particles across boundaries, super-particle weighting that is dependent on the “birth-place” of the particle, and the use of an X-Y mover in cylindrical space. These unique approaches will be elaborated on in sections 4. 2. 7, and 4. 2. 11.

4.2.2 Random Number Generation (`random_number.c`)

The generation of pseudo-random numbers is an important element of any PIC or PIC'/Monte-Carlo simulation. The “quality” of the random numbers plays a major role in the statistical accuracy of the simulation. The quality of a random number generator is determined by its ability to generate (seemingly) uncorrelated numbers with a uniform deviate, with a period that won’t be approached in the duration of the simulation. The techniques used to generate random numbers are apparently somewhat of a black art. The fastest generators seem to exploit memory over-flow and other “scary” architecture based tricks. We say scary because the author has found that these fast random number generators may function properly on thirty-two bit systems, but produce unsatisfactory output on sixty-four bit machines. Portability was chosen over speed for PROBEPIC. A random number generator was down-loaded from the network and tested on multiple platforms to determine its quality and portability. The only documentation provided in the source code is:

```
/* portable lagged Fibonacci series uniform random number */
```

```
/* generator with "lags" -273 und -607: */
/* W.P. Petersen, IPS, ETH Zuerich, 19 Mar. 92 */
```

This algorithm, called `random_number.c` in this application, is particularly nice because it allows one to send it an array of any size, which upon return will be filled with random numbers of a uniform deviate. This falls in line with the overall PROBEPIC philosophy, where we store many values for later use rather than calculating them every time they are needed. When all of the random numbers in the random number array (`random_number[]` in PROBEPIC) are exhausted, the array is sent back to `random_number.c` with a new seed and reinitialized.

4.2.3 Initial Conditions (`parameter.c`)

The initial plasma parameters (such as temperature and number density) are set in `parameter.c`. From these user defined conditions, quantities such as plasma frequency and Debye length are calculated for use in later subroutines. `parameter.c` is called only once during the execution of PROBEPIC.

4.2.4 Grid Generation (`grid.c`)

The computational grid is a structure which associates the following properties to each grid-point: position, charge density, radial and axial electric field intensity, and electrostatic potential. The actual structure definition from `probepic.h` is:

```
typedef struct {
    double x,y,q_dens,Ey,Ex,phi;
}GRIDDEF;
```

PROBEPIC uses a non-uniform grid. The grid density is tightened in the vicinity of the probe to resolve the field within the sheath region. The grid dimensions (i. e. the

cell width and cell height) are set in `parameter.c`. `grid.c` uses these dimensions to set the x and y coordinates of the grid-points in the structure `GRIDDEF grid[ngp]`, where `ngp` is the total number of grid-points. The other properties (field quantities) are initialized to zero. `grid.c` is called only once during the execution of PROBEPIC.

4.2.5 Maxwellian Velocity Generator (`maxwellian.c`)

Particles in a quiescent plasma must have a Maxwellian distribution of speeds, as the Boltzmann H-theorem proves that the only possible equilibrium distribution is the Maxwellian distribution. Consequently, an algorithm is needed to generate random speeds, which after many calls, will reproduce a Maxwellian distribution. There is a general method for generating random numbers with any desired distribution; the steps shown below will illustrate that general method and give a result for the problem at hand.

For a Maxwellian distribution the probability of finding a particle, say an electron, with peculiar speed between W and $W + dW$ is

$$f(W) = \left(\frac{2}{\pi}\right)^{1/2} N \left(\frac{m}{kT}\right)^{3/2} W^2 e^{-(mW^2/2kT)} . \quad (4.1)$$

The normalized version of (4.1) is plotted as a function of W in Figure 4.3.

Now, we have available a random number generator which produces a uniform deviate (i. e. generates a uniform set of random numbers from 0.0 to 1.0), `random_number.c`. The question is, “how can we use `random_number.c` to generate a set of random speeds with a Maxwellian distribution?” The answer can be stated in mathematical probability transformation theory [12], or we can state in words a more intuitive, geometric prescription, referring to Figure 4.3:

Theorem 1 *The integral $F(W)$ is the area under the probability curve $f(W)$ to the*

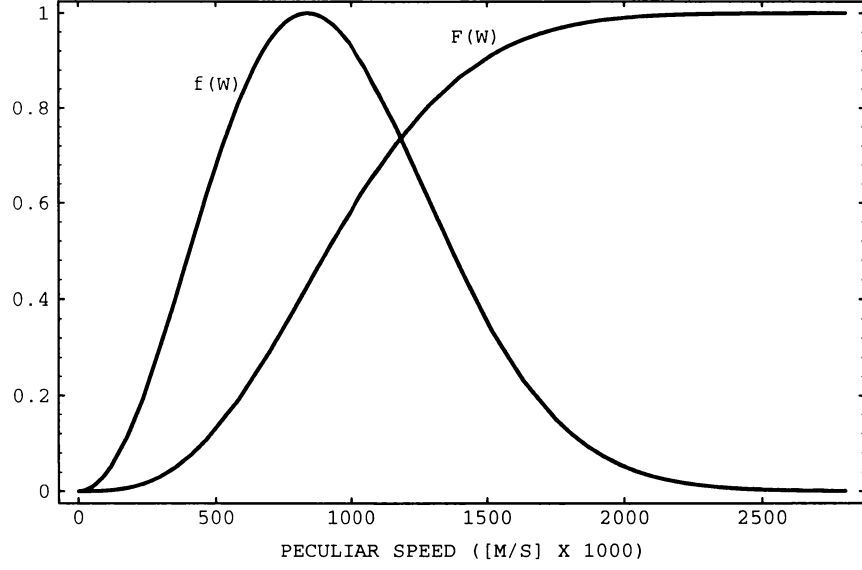


Figure 4.3: Maxwellian distribution of peculiar speeds W for electrons at $T=2.0[\text{eV}]$, $f(W)$, and the normalized integral of $f(W)$, $F(W)$.

left of W . To obtain random Maxwellian speed W , choose a uniform random number $y=F(W)$, then find the value W that has that fraction y of probability area to its left, and return the value W .

For the present problem, integrating (4.1) gives:

$$\begin{aligned}
 F(W) &= \int f(W) dW \\
 &= N \left(\frac{2}{\pi} \right)^{1/2} \left(\frac{m}{kT} \right)^{3/2} \\
 &\quad \left\{ \left(\frac{\pi}{2} \right)^{1/2} \left(\frac{kT}{m} \right)^{3/2} \text{Erf} \left[\left(\frac{m}{2kT} \right)^{1/2} W \right] - \frac{k T W}{m e^{(mW^2/2kT)}} \right\}
 \end{aligned} \tag{4.2}$$

Thus, to get W 's with a Maxwellian distribution we need to invert (4.2) for W , which is clearly a daunting task analytically. Instead, we can pick a uniform random number $F(W)$, and then use a numerical scheme such as the Newton-Raphson method to back out W . Judging from the complexity of (4.2) this numerical approach would be computationally expensive. To get around this PROBEPICT uses a “look-up” table. The

sub-program `maxwellian.c` initializes arrays (one for each species) of one hundred thousand elements into which speeds generated using (4.2) are put. Other PROBEPIC sub-programs simply look-up a speed at a random place in the table. This gives us the desired result of generating speeds with a Maxwellian distribution while keeping the computational time low.

4.2.6 Initial Particle loading (`initialize.c`)

The sub-program `initialize.c` loads a uniform distribution of computational particles to begin the simulation. The velocities of each species are set by randomly picking values from the velocity tables created in `maxwellian.c`. To make the initial number density uniform we have to account for volume's radial dependence, i. e. we have to distribute more particles toward the outside of the computational domain.

The probability of finding a particle between r and $r + dr$ is directly proportional the volume, that is,

$$p(r) dr = V(r) = 2 \pi r l dr . \quad (4.3)$$

Using logic similar to that presented in the previous section (taking $l = 1$ and normalizing with the outer radius of the computational domain $r = R$)

$$P(r) = \frac{\int p(r) dr}{\int_0^R p(r) dr} = \frac{r^2}{R^2} . \quad (4.4)$$

Thus, to determine the radial location of a particle in `initialize.c` we generate a uniform random number $\alpha = P(r)$ and use the inverted form of (4.4):

$$r = R (\alpha)^{1/2} . \quad (4.5)$$

The remainder of the superparticle parameters (mass, charge etc.) are also set for each particle. `initialize.c` is called only once during the execution of PROBEPIC.

4.2.7 Fluxing Particles (`inject_part.c`, `inject_beam.c`)

Particles diffuse into the computational domain from the outside plasma as a result of thermal or directed motion. In the quiescent plasma simulation the ions and electrons were treated as being in thermal equilibrium, while in the beam simulation the electrons were treated as thermalized and the ions were treated as having only directed energy. Determining the proper method for fluxing these particles into the computational domain proved to be the most difficult problem in the development of PROBEPIC. We need to determine: 1) How many particles enter the computational domain in a given time-step, and 2) The angular and speed distribution of these particles.

Let us first consider the problem of fluxing thermalized particles. The answer to the first question is straightforward. The flux of particles with a Maxwellian distribution function across a unit surface per unit time is

$$\Gamma = \frac{n \bar{c}}{4} , \quad (4.6)$$

where, Γ is the flux, n is the number density, and $\bar{c}$ is the mean speed. For a Maxwellian distribution,

$$\bar{c} = \sqrt{\frac{8 k T}{\pi m}} , \quad (4.7)$$

where k is the Boltzmann constant, T is the temperature, and m is the mass of the particle. Equation (4.6) gives us the total number of *real* particles crossing the surface of the computational domain. In PROBEPIC we divide those particles into considerably fewer super-particles — this ultimately determines the number of super-particles that will be in the computational domain once the simulation has reached steady state.

Next we consider the problem of determining the direction and speed in which a particle will cross the boundary. Consider an elemental area dA with unit normal $\hat{n}$.

Particles cross this area with a relative velocity between v and $v + dv$, polar angle relative to the normal between θ and $\theta + d\theta$, and azimuthal angle between ϕ and $\phi + d\phi$. In time dt all particles crossing dA with velocity v must have been within the volume of the prism shown in Figure 4.4. The volume of the prism is then

$$V = \vec{v} \cdot \hat{n} dA dt = v \cos \theta dA dt . \quad (4.8)$$

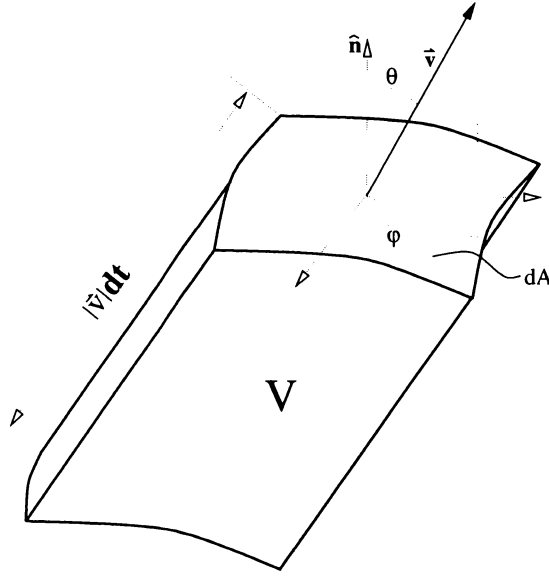


Figure 4.4: Illustrative flux volume.

The number of particles with the appropriate velocities and directions per unit volume is,

$$n_{v\theta\phi} = f d^3v , \quad (4.9)$$

where, d^3v is the incremental volume in velocity space. If the distribution function is expressed in spherical coordinates,

$$d^3v = v^2 \sin \theta d\theta d\phi dv , \quad (4.10)$$

and,

$$n_{v\theta\phi} = f v^2 \sin \theta d\theta d\phi dv . \quad (4.11)$$

If f is Maxwellian, then

$$f = n \left(\frac{m}{2\pi kT} \right)^{3/2} e^{-\left(\frac{mv^2}{2kT}\right)}, \quad (4.12)$$

where v is the magnitude of the velocity $\vec{v}$. Thus, the number of particles with the appropriate velocity crossing dA in time dt is

$$\begin{aligned} N_{v\theta\phi} &= n_{v\theta\phi} V \\ &= n \left(\frac{m}{2\pi kT} \right)^{3/2} v^3 e^{-\left(\frac{mv^2}{2kT}\right)} \sin \theta \cos \theta d\theta d\phi dv dA dt. \end{aligned} \quad (4.13)$$

This is the distribution of particles which must be put into the computational domain per unit area per unit time. The flux is then

$$d^3\Gamma_{v\theta\phi} = n \left(\frac{m}{2\pi kT} \right)^{3/2} v^3 e^{-\left(\frac{mv^2}{2kT}\right)} \sin \theta \cos \theta d\theta d\phi dv. \quad (4.14)$$

As a check we note that integrating 4.14 gives

$$\begin{aligned} \Gamma &= \int d^3\Gamma_{v\theta\phi} \\ &= \int_0^\infty \int_0^\pi \int_0^{2\pi} n \left(\frac{m}{2\pi kT} \right)^{3/2} v^3 e^{-\left(\frac{mv^2}{2kT}\right)} \sin \theta \cos \theta d\theta d\phi dv \\ &= \frac{n \bar{c}}{4}. \end{aligned} \quad (4.15)$$

We can now write down the probability of a particle entering the computational domain with speed v , polar angle θ , and azimuthal angle ϕ as the product of the individual probabilities given in 4.14,

$$P_\Gamma = P_v P_\theta P_\phi = \frac{d^3\Gamma_{v\theta\phi}}{\frac{n \bar{c}}{4}} \quad (4.16)$$

or,

$$P_v = \frac{4\pi}{\bar{c}} \left(\frac{m}{2\pi kT} \right)^{3/2} v^3 e^{-\left(\frac{mv^2}{2kT}\right)} \quad (4.17)$$

$$P_\theta = \frac{\sin \theta \cos \theta}{2} \quad (4.18)$$

$$P_\phi = \frac{1}{2\pi}. \quad (4.19)$$

We can now apply the formalism developed in section 4.2.5 to generate random injection speeds and angles for a particle. The sub-program `make_velocity_table.c` initializes an array of one-hundred thousand speeds for each species, using 4.17. This reduces the overall PROBEPIC computational time significantly, as `inject_part.c` can simply “look-up” values as it needs them rather than spending a great deal of time numerically inverting (4.17). All that remains is to determine *where* a particle is injected. The computational domain may be divided into three regions all having different characteristic areas: the back face (through which the probe and insulator pass), the sides, and the front face. The number of particles which flow across each of these surfaces in a given time-step is, of course, dependent on the surface area; consequently, in a given time-step, `inject_part.c` sets the super-particle weighting of each particle depending on where it is “born”.

The preceding discussion assumes we are simulating a quiescent plasma. In the case of a flowing plasma we still assume that the electrons are in a Maxwellian distribution, however, the ions are assumed to have a dominant directed velocity. Ions are injected through the front face only, with a constant velocity, and a random polar angle between $\pm 5^\circ$. The rationale for the random injection angle is that it takes the beam divergence, which is found in an ion thruster plume, into account. An ion thruster extracts and accelerates ions through hundreds of beamlet forming orifices. The beam emerging from these orifices expands downstream of the thruster; the random injection angle is a first order approximation to include this effect. To assure that sufficient ions are available to strike the probe, super-ions which are born with radial positions less than one-fifth of the total computational domain radius are further divided in several more super-particles. This increases the number of super-particles near the center of the computational domain, as desired. The sub-program `inject_part_beam.c` handles injection of the ion beam.

4.2.8 Charge Weighting (`q_weight.c`)

Cylindrical coordinates are used in PROBEPIC because of the cylindrical symmetry of the computational domain; therefore, we must develop charge weighting shape functions appropriate for cylindrical coordinates. Ruyten [13] has shown that traditional “area” weighting schemes do not conserve charge density in the radial direction. He proposed a new radial charge density conserving algorithm (henceforth referred to as Ruyten weighting), while maintaining linear weighting in the axial direction. This method was implemented in PROBEPIC. Ruyten’s results are given below without proof; the interested reader may consult his paper for greater detail.

If a particle is at (z_o, r_o) , in the cell bounded by grid-points at (z_i, r_j) , (z_i, r_{j+1}) , (z_{i+1}, r_{j+1}) , (z_{i+1}, r_j) , then the charge assigned to each of these grid-points is given by:

$$q_{i,j} = q_o \frac{(r_{j+1} - r_o)(z_{i+1} - z_o)}{2(r_{j+1}^2 - r_j^2)(z_{i+1} - z_i)}(2r_{j+1} + 3r_j - r_o) \quad (4.20)$$

$$q_{i,j+1} = q_o \frac{(r_o - r_j)(z_{i+1} - z_o)}{2(r_{j+1}^2 - r_j^2)(z_{i+1} - z_i)}(3r_{j+1} + 2r_j - r_o) \quad (4.21)$$

$$q_{i+1,j+1} = q_o \frac{(r_o - r_j)(z_o - z_i)}{2(r_{j+1}^2 - r_j^2)(z_{i+1} - z_i)}(3r_{j+1} + 2r_j - r_o) \quad (4.22)$$

$$q_{i+1,j} = q_o \frac{(r_{j+1} - r_o)(z_o - z_i)}{2(r_{j+1}^2 - r_j^2)(z_{i+1} - z_i)}(2r_{j+1} + 3r_j - r_o) . \quad (4.23)$$

The sub-program `q_weight.c` performs the charge weighting in PROBEPIC. It iterates through each particle, performing Ruyten weighting to yield the total charge density on the grid. Before applying equations (4.20) - (4.23), `q_weight.c` must determine what grid cell the particle is in. For this, it calls an external sub-program `locate.c`. Now, this cell location will also be needed in `force_weight.c`. For this reason, we opt to eliminate the redundant step, and append the cell information (i.e. store) to the particle for future use. For example, referring to the PARTICLEDEF

structure definition at the beginning of the chapter, cell information would be associated with the i^{th} electron by the statements,

```
electron[i].cellx = ...;
electron[i].celly = ...;
```

and subsequently recalled in `force_weight.c`.

4.2.9 Field Solver (`e_field.c`)

PROBEPIC uses a novel field solver. When dealing with curvilinear coordinates one almost always has to go to an exotic field solver — one that maps the coordinates into a rectangular space to solve Poisson’s equation, and then transforms the solution back to the curvilinear space (e. g. the ADI method). PROBEPIC makes direct use of Gauss’ law to eliminate the need for any transformation, yielding a simple, intuitive algorithm. The following discussion follows Peng [14] with minor changes.

Gauss’ law states that the surface integral of electric flux is equal to the total charge enclosed by the surface. Mathematically,

$$\int \int_S \vec{\mathbf{E}} \cdot d\vec{\mathbf{A}} = -\frac{1}{\epsilon_o} \int \int \int_V \rho dV , \quad (4.24)$$

or, if the surface is chosen to enclose one grid-point at (z_i, r_j) ,

$$\int \int_{dA_{i,j}} \vec{\mathbf{E}} \cdot d\vec{\mathbf{A}} = -\frac{1}{\epsilon_o} \int \int \int_{dV_{i,j}} \rho dV = Q_{j,k} , \quad (4.25)$$

where, $Q_{j,k}$ is the charge at (z_i, r_j) (ϵ_o has been absorbed into the definition of $Q_{j,k}$ for convenience), which results from the charge weighting algorithm.

All that remains is to discretize (4.25) subject to appropriate boundary conditions, and provide a method to solve the resulting system of equations. The electric field boundary conditions used on the various surfaces in PROBEPIC are illustrated in Figure 4.5.

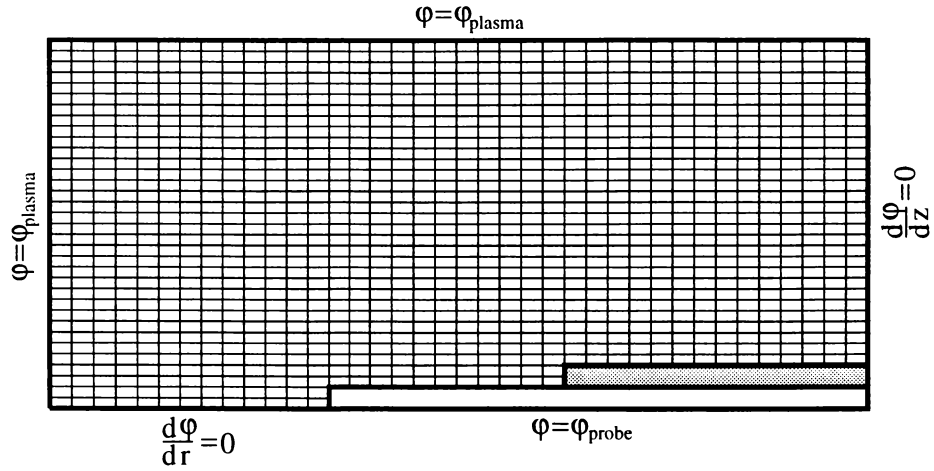


Figure 4.5: Schematic of field boundary conditions in PROBEPIC.

Discretization of Poisson's Equation

Because of the azimuthal symmetry in PROBEPIC we need only calculate fields in the $\hat{r}$ and $\hat{z}$ directions on the half plane shown in Figure 4.5. The computational mesh may be divided into two broad categories: interior mesh points and surface mesh points. Interior mesh points lie within the plasma, probe conductor, and probe insulator. Surface mesh points lie at the edges of the computational domain, and on the surfaces of the probe conductor and insulator. The surface boundary points can be further classified as surfaces of constant potential, or surfaces with special symmetry boundary conditions, such as vanishing tangential fields. Each of the grid regions will be described separately.

Case 1 : Interior Mesh-Points

Figure 4.6 illustrates a typical interior grid-point. The Gaussian surface is an annular ring, the cross-section of which is illustrated by dashed lines in the figure. We

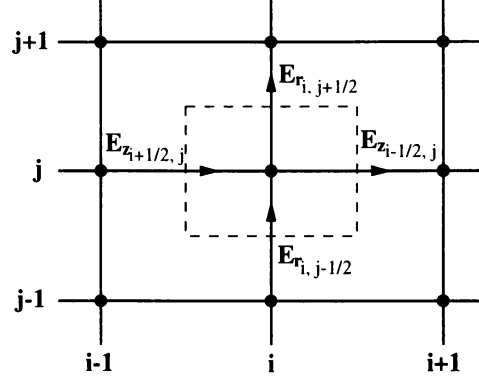


Figure 4.6: Two dimensional interior computational mesh-point and Gauss' law volume.

can write down Gauss' law for this Gaussian surface:

$$Q_{i,j} = \vec{E}_{z,i-1/2} \cdot \hat{a}_{z,i-1/2} dA_{z,i-1/2} + \vec{E}_{z,i+1/2} \cdot \hat{a}_{z,i+1/2} dA_{z,i+1/2} + \vec{E}_{r,j-1/2} \cdot \hat{a}_{r,j-1/2} dA_{r,j-1/2} + \vec{E}_{r,j+1/2} \cdot \hat{a}_{r,j+1/2} dA_{r,j+1/2} . \quad (4.26)$$

The surface area elements are given by

$$dA_{z,i-1/2} = dA_{z,i+1/2} = \pi(r_{j+1/2}^2 - r_{j-1/2}^2) \quad (4.27)$$

$$dA_{r,j-1/2} = 2\pi r_{j-1/2}(z_{i+1/2} - z_{i-1/2}) \quad (4.28)$$

$$dA_{r,j+1/2} = 2\pi r_{j+1/2}(z_{i+1/2} - z_{i-1/2}) , \quad (4.29)$$

where

$$r_{j+1/2} = \frac{r_{j+1} + r_j}{2} \quad (4.30)$$

$$r_{j-1/2} = \frac{r_{j-1} + r_j}{2} \quad (4.31)$$

$$z_{i+1/2} = \frac{z_{i+1} + z_i}{2} \quad (4.32)$$

$$z_{i-1/2} = \frac{z_{i-1} + z_i}{2} . \quad (4.33)$$

If we define

$$(\Delta r^2)_j = r_{j+1/2}^2 - r_{j-1/2}^2 \quad (4.34)$$

$$\Delta r_{j+1/2} = r_{j+1/2} - r_{j-1/2} \quad (4.35)$$

$$\Delta r_{j-1/2} = r_{j+1/2} - r_{j-1/2} \quad (4.36)$$

$$\Delta z_i = z_{i+1/2} - z_{i-1/2} \quad (4.37)$$

$$\Delta z_{i+1/2} = z_{i+1} - z_i \quad (4.38)$$

$$\Delta z_{i-1/2} = z_i - z_{i-1} , \quad (4.39)$$

and use first order differencing to express the electric fields in terms of the potentials,

$$E_{r,i,j+1/2} = -\frac{\phi_{i,j+1} - \phi_{i,j}}{\Delta r_{j+1/2}} \quad (4.40)$$

$$E_{r,i,j-1/2} = -\frac{\phi_{i,j} - \phi_{i,j-1}}{\Delta r_{j-1/2}} \quad (4.41)$$

$$E_{z,i+1/2,j} = -\frac{\phi_{i+1,j} - \phi_{i,j}}{\Delta z_{i+1/2}} \quad (4.42)$$

$$E_{z,i-1/2,j} = -\frac{\phi_{i,j} - \phi_{i-1,j}}{\Delta z_{i-1/2}} , \quad (4.43)$$

then Gauss' law, or the difference equation for interior mesh points, may be written in the simple form:

$$\begin{aligned} -Q_{i,j} &= a_{i,j+1}\phi_{i,j+1} + a_{i,j-1}\phi_{i,j-1} + a_{i+1,j}\phi_{i+1,j} \\ &+ a_{i-1,j}\phi_{i-1,j} + a_{i,j}\phi_{i,j} , \end{aligned} \quad (4.44)$$

where

$$a_{i,j+1} = 2\pi \frac{\Delta z_i r_{j+1/2}}{\Delta r_{j+1/2}} \quad (4.45)$$

$$a_{i,j-1} = 2\pi \frac{\Delta z_i r_{j-1/2}}{\Delta r_{j-1/2}} \quad (4.46)$$

$$a_{i+1,j} = \pi \frac{(\Delta r^2)_j}{\Delta z_{i+1/2}} \quad (4.47)$$

$$a_{i-1,j} = \pi \frac{(\Delta r^2)_j}{\Delta z_{i-1/2}} . \quad (4.48)$$

$$a_{i,j} = -(a_{i,j+1} + a_{i,j-1} + a_{i+1,j} + a_{i-1,j}) . \quad (4.49)$$

Another class of interior mesh-points are those that lie within the probe conductor and insulator. Points lying within the conductor can be treated simply with the definition:

$$\phi_{i,j} = \phi_{probe} . \quad (4.50)$$

Points inside the insulator may be treated with the same prescription as in (4.44). Of course, pedantically, we should use the proper insulator dielectric constant in the definition of $Q_{i,j}$; however, we are not interested in the fields within the insulator, so we need not treat these points any differently.

Case 2 : Boundary Mesh-Points

As shown in Figure 4.5 the PROBEPIC computational domain terminates on boundaries of either constant potential or vanishing tangential or normal field. The difference equations for each case will be treated separately.

Surfaces of constant potential include the the top and front of the computational domain, and the conductor section of the probe. Mesh-points on these surfaces are simply assigned the constant potential value; for example, points on the the probe conductor are given the potential $\phi = \phi_{probe}$, and those on the edge of the computational domain $\phi = \phi_{plasma}$, etc. . Thus no formal difference equations are required for these points.

Special restrictions are put on the electric field along the ($r = 0$) axis in front of the probe and along the ($z = z_{max}$) downstream boundary. At ($r = 0$) we require from symmetry considerations that the electric field has no radial component. Referring to Figure 4.7, Gauss' law for points on the ($r = 0$) axis gives:

$$Q_{i,0} = 2\pi\Delta z_i r_{1/2} E_{r,i,1/2} + \pi(\Delta r^2)_0 (E_{z,i+1/2,0} - E_{z,i-1/2,0}) , \quad (4.51)$$

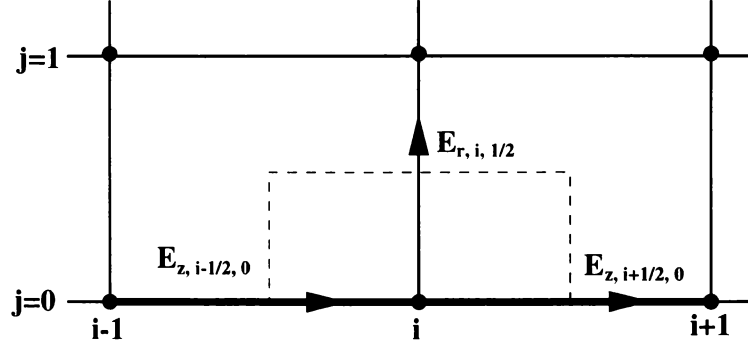


Figure 4.7: Gauss' law volume for $(r = 0)$ axis in front of the probe .

where

$$(\Delta r^2)_0 = r_{1/2}^2 - r_0^2 = r_{1/2}^2 . \quad (4.52)$$

Taking the $E_{r,i,1/2}$ term to be zero, and again using the first order expressions for the electric field, the difference equation along the $(r = 0)$ axis becomes:

$$-Q_{i,0} = a_{i,1}\phi_{i,1} + a_{i+1,0}\phi_{i+1,0} + a_{i-1,0}\phi_{i-1,0} + a_{i,0}\phi_{i,0} , \quad (4.53)$$

where,

$$a_{i,1} = 2\pi \frac{\Delta z_i r_{1/2}}{\Delta r_{1/2}} \quad (4.54)$$

$$a_{i+1,0} = \pi \frac{(\Delta r^2)_0}{\Delta z_{i+1/2}} \quad (4.55)$$

$$a_{i-1,0} = \pi \frac{(\Delta r^2)_0}{\Delta z_{i-1/2}} \quad (4.56)$$

$$a_{i,0} = -(a_{i,1} + a_{i+1,0} + a_{i-1,0}) . \quad (4.57)$$

At the downstream $(z = z_{max})$ boundary we require that the electric field have no axial component; this implies that the field has returned to what one expects from an infinite cylindrical conductor, or that the effect of the probe tip has become small, since it is far away. The Gaussian surface is the same as that used in deriving the $(r = 0)$ expressions, except rotated ninety degrees. We can therefore immediately

write down an analogous result for the difference equations at the downstream surface of the computational domain (for $i_{max} = n$):

$$-Q_{n,j} = a_{n,j+1}\phi_{n,j+1} + a_{n,j-1}\phi_{n,j-1} + a_{n-1,j}\phi_{n-1} + a_{n,j}\phi_{n,j} , \quad (4.58)$$

where

$$a_{n,j+1} = 2\pi \frac{\Delta z_n r_{j+1/2}}{\Delta r_{j+1/2}} \quad (4.59)$$

$$a_{n,j-1} = 2\pi \frac{\Delta z_n r_{j-1/2}}{\Delta r_{j-1/2}} \quad (4.60)$$

$$a_{n-1,j} = \pi \frac{(\Delta r^2)_j}{\Delta z_{n+1/2}} \quad (4.61)$$

$$a_{n,j} = -(a_{n,j+1} + a_{n,j-1} + a_{n-1,j}) . \quad (4.62)$$

Solution of the System of Equations

In the previous section we derived the difference equations for all mesh-points in the PROBEPIC computational domain. These difference equations define m (where m is the total number of mesh-points) simultaneous equations for the potential at each mesh-point. These equations may be written as a matrix equation

$$[\mathbf{A}]\vec{\phi} = -\vec{\mathbf{Q}} , \quad (4.63)$$

where $[\mathbf{A}]$ is the coefficient matrix of $a_{i,j}$ s, $\vec{\phi}$ is the vector of electrostatic potentials at each grid-point, and $\vec{\mathbf{Q}}$ is the source vector of weighted grid-point charges, $Q_{i,j}$.

When the coefficient matrix $[\mathbf{A}]$ is explicitly constructed using the prescriptions for the $a_{i,j}$ s given in the previous section, a square *band matrix* of order m results. The band matrix form is schematically illustrated in Figure 4.8. A band matrix consists of several diagonals surrounded by “null”, or zero elements. As a result it is wasteful of resources (and in fact is often impossible) to store the entire matrix, which is composed almost entirely of zeros. Also, solving the system (4.63) by traditional

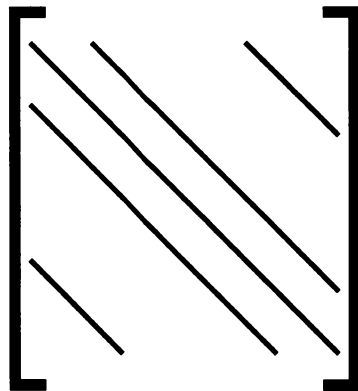


Figure 4.8: Sketch of the band matrix form; non-zero elements are indicated by black diagonal lines.

methods such as Gaussian elimination is out of the question for large band matrices (e. g. the difference matrix in PROBEPIC is approximately 14000×14000).

Therefore, we are led to algorithms specialized for the solution of band matrices. Many such algorithms are available, ranging from simple exact (i. e. returning analytical results) to exotic, inexact methods. While all methods apparently give satisfactory numerical results, the inexact methods are generally much faster. Unfortunately, the implementation of the more exotic techniques (e. g. the ICCG method) requires one to become knowledgeable of the underlying mathematical and computational “tricks” that make the method work — the time investment may become substantial. In this first version of PROBEPIC, we opt for a simpler, more robust (albeit slower) algorithm because it is easier to implement, and is more likely to give the expected results. Future versions of PROBEPIC may implement faster techniques, which can be tested for accuracy against the present results.

PROBEPIC uses standard subroutines from the LAPACK [22] linear algebra library, which applies the LU decomposition method to the solution of band matrices. In

this method the difference matrix $[\mathbf{A}]$ is decomposed into the product of two matrices,

$$[\mathbf{A}] = [\mathbf{L}][\mathbf{U}] , \quad (4.64)$$

where $[\mathbf{L}]$ is a lower triangular matrix (a matrix which has elements only on the diagonal and below), and $[\mathbf{U}]$ is an upper triangular matrix (a matrix which has elements only on the diagonal and above). Using this definition (4.63) may be written

$$[\mathbf{A}]\vec{\phi} = ([\mathbf{L}][\mathbf{U}])\vec{\phi} = [\mathbf{L}]([\mathbf{U}]\vec{\phi}) = -\vec{\mathbf{Q}} \quad (4.65)$$

Thus, the solution of (4.63) becomes a two step process. Setting $\vec{\alpha} = [\mathbf{U}]\vec{\phi}$, we first solve

$$[\mathbf{L}]\vec{\alpha} = -\vec{\mathbf{Q}} , \quad (4.66)$$

for the vector $\vec{\alpha}$ and use this result to solve

$$[\mathbf{U}]\vec{\phi} = -\vec{\alpha} , \quad (4.67)$$

yielding the desired vector $\vec{\phi}$ of electrostatic potentials at the grid-points. This method is quite efficient for two reasons. First, very fast algorithms for the solution of (4.66) and (4.67) exist. Second, since the difference matrix depends only on the geometry of the computational domain (which does not change in the course of a simulation), $[\mathbf{A}]$ does not change; consequently, the *LU* decomposition only needs to be performed once, provided we save the result of the original decomposition. This decomposition is performed once at the beginning of PROBEPIC's execution by the sub-program `make_LU.c`, which consists of a complicated set of loops to form the bands in a form usable by the LAPACK routine.

The formalism illustrated in this section was successfully implemented in PROBEPIC. Comparison of fields generated by PROBEPIC and “rough” analytical results are in complete agreement. We say “rough” analytical results because, of course, the analytical solution of the Poisson equation with mixed (Neumann and Dirichlet) boundary

conditions, as are found in PROBEPIIC, is impossible. However, we can easily derive an analytical expression for an infinite cylindrical conductor in charge-free space (i. e. we are now solving the Laplace equation). Far away from the probe tip, where the electric field is almost completely radial, PROBEPIIC gives potential distributions in complete agreement with this analytical result. A contour plot of electrostatic potentials generated by PROBEPIIC is shown in Figure 4.9. The top of the figure is the result for charge-free space, while the bottom is the steady state result for a plasma ($n = 1.0 \times 10^9 [\text{cm}^{-3}]$) filled computational domain. In both cases the probe was biased -2 [V] relative to the plasma potential. The bottom plot clearly shows that PROBEPIIC simulates the plasma, or Debye shielding — the bulk of the plasma is shielded from the probe potential by a thin plasma sheath.

4.2.10 Force Weighting (f_weight.c)

As mentioned in Chapter 3, it is advisable to use the same weighting scheme for both weighting the charge to the grid and weighting the field back to the particles. Therefore, in PROBEPIIC, Ruyten weighting is inverted to give the field at the particle position. In analogy to equations (4.20) - (4.23) the x and y components of the electric field for the i^{th} particle in terms of adjacent node field intensities are given by:

$$\begin{aligned}
E_{x,i} = & E_{x,i,j} \frac{(r_{j+1} - r_o)(z_{i+1} - z_o)}{2(r_{j+1}^2 - r_j^2)(z_{i+1} - z_i)} (2r_{j+1} + 3r_j - r_o) \\
& + E_{x,i,j+1} \frac{(r_o - r_j)(z_{i+1} - z_o)}{2(r_{j+1}^2 - r_j^2)(z_{i+1} - z_i)} (3r_{j+1} + 2r_j - r_o) \\
& + E_{x,i+1,j+1} \frac{(r_o - r_j)(z_o - z_i)}{2(r_{j+1}^2 - r_j^2)(z_{i+1} - z_i)} (3r_{j+1} + 2r_j - r_o) \\
& + E_{x,i+1,j} \frac{(r_{j+1} - r_o)(z_o - z_i)}{2(r_{j+1}^2 - r_j^2)(z_{i+1} - z_i)} (2r_{j+1} + 3r_j - r_o) ,
\end{aligned} \tag{4.68}$$

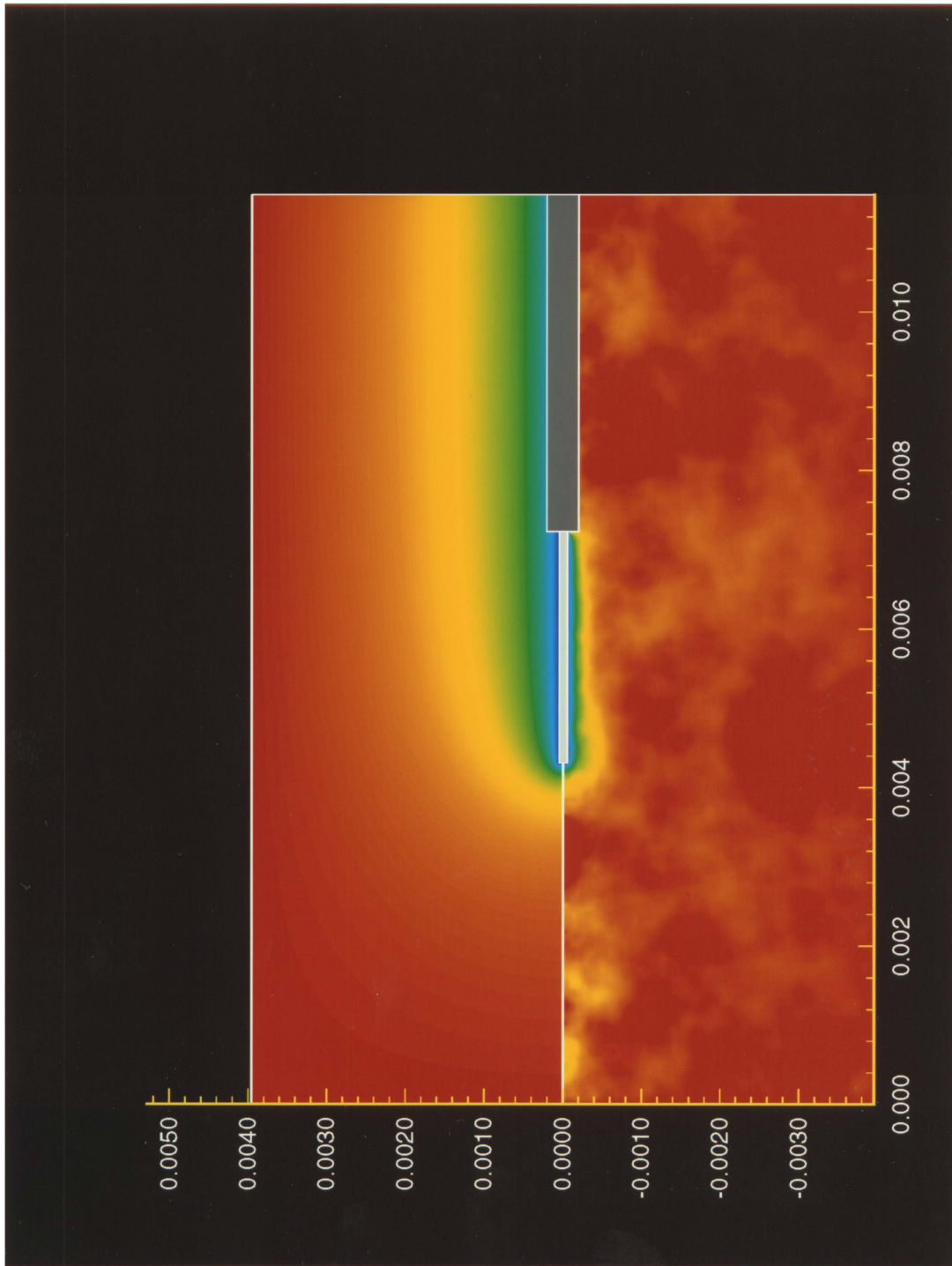


Figure 4.9: Contour plot of regions of constant electrostatic potential for two different conditions. The top result is for charge-free space. The bottom result is for a plasma filled computational domain. In both cases the probe is biased at $-2.0[\text{V}]$ relative to the plasma potential.

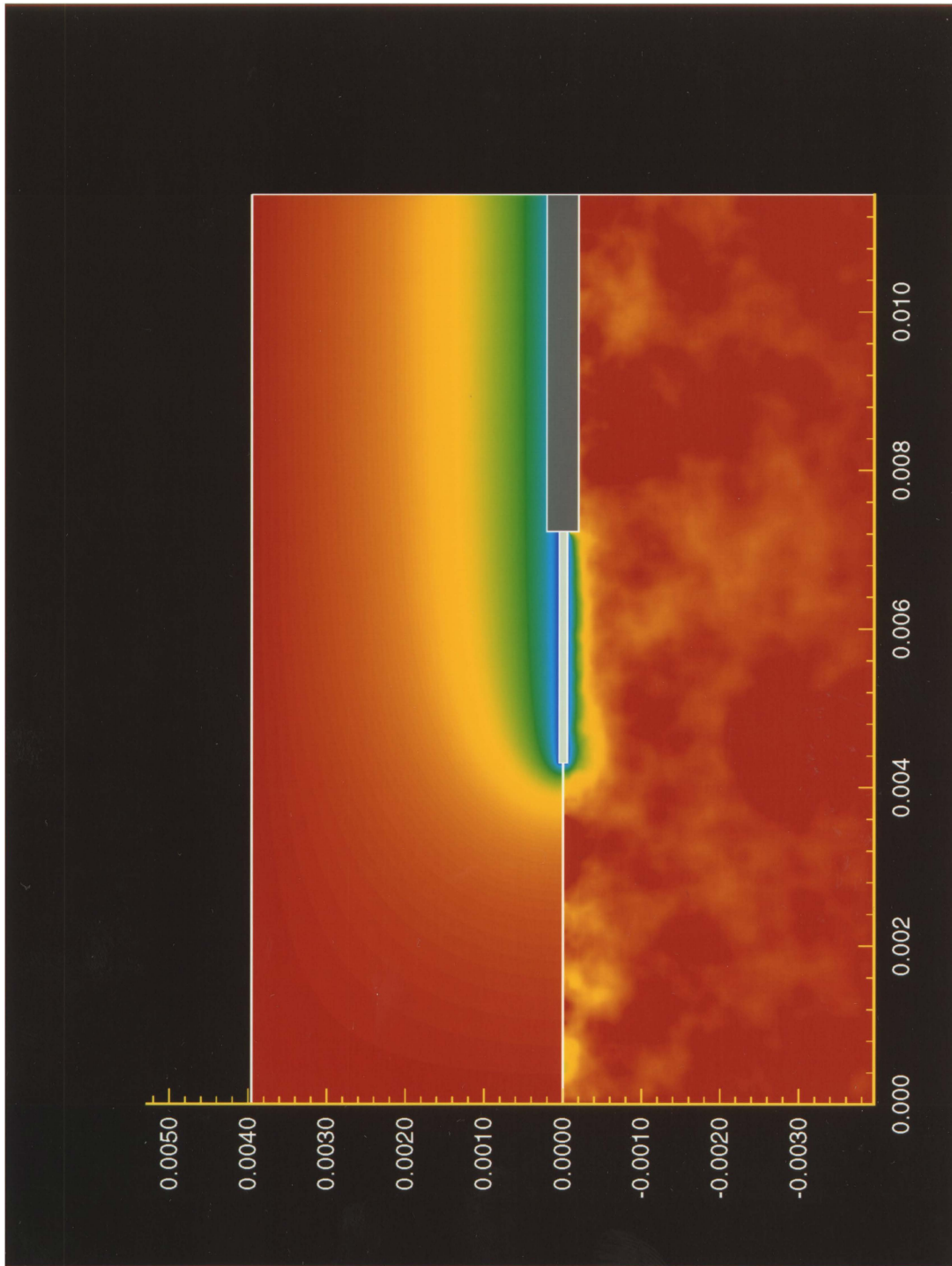


Figure 4.9: Contour plot of regions of constant electrostatic potential for two different conditions. The top result is for charge-free space. The bottom result is for a plasma filled computational domain. In both cases the probe is biased at $-2.0[\text{V}]$ relative to the plasma potential.

$$\begin{aligned}
E_{y,i} &= E_{y,i,j} \frac{(r_{j+1} - r_o)(z_{i+1} - z_o)}{2(r_{j+1}^2 - r_j^2)(z_{i+1} - z_i)} (2r_{j+1} + 3r_j - r_o) \\
&+ E_{y,i,j+1} \frac{(r_o - r_j)(z_{i+1} - z_o)}{2(r_{j+1}^2 - r_j^2)(z_{i+1} - z_i)} (3r_{j+1} + 2r_j - r_o) \\
&+ E_{y,i+1,j+1} \frac{(r_o - r_j)(z_o - z_i)}{2(r_{j+1}^2 - r_j^2)(z_{i+1} - z_i)} (3r_{j+1} + 2r_j - r_o) \\
&+ E_{y,i+1,j} \frac{(r_{j+1} - r_o)(z_o - z_i)}{2(r_{j+1}^2 - r_j^2)(z_{i+1} - z_i)} (2r_{j+1} + 3r_j - r_o) .
\end{aligned} \tag{4.69}$$

4.2.11 Moving Particles (mover.c)

The mover updates the position and velocity of each super-particle given its previous state and the electric field intensity at the particle position. The first order technique described in chapter 3 is implemented in PROBEPIC. In review,

$$\vec{\mathbf{v}}_{\text{new}} = \vec{\mathbf{v}}_{\text{old}} + \left[\frac{q}{m} \vec{\mathbf{E}}_{\text{old}} \right] \Delta t \tag{4.70}$$

$$\vec{\mathbf{x}}_{\text{new}} = \vec{\mathbf{x}}_{\text{old}} + \vec{\mathbf{v}}_{\text{new}} \Delta t . \tag{4.71}$$

The PROBEPIC computational domain has cylindrical symmetry; therefore it seems most logical to write the equations of motion in terms of set cylindrical basis vectors. Gopinath [24], however, shows that this poses a problem as the particle passes close to the origin. For example, if the incremental change in the polar angle θ is given by $\Delta\theta = (v_\theta \Delta t)/r$, then $\Delta\theta$ becomes very large when a particle passes near the origin, that is, $r \rightarrow 0$. The solution is to move the particles in cartesian coordinates and then transform the new particle position back into cylindrical coordinates (the field solver, charge weighting, etc. are carried out in cylindrical coordinates). This technique was implemented in PROBEPIC to avoid the singularity at $r = 0$.

The only as yet undefined quantity in (4.70) and (4.71) is Δt . It is through Δt that we can attempt to rectify the problem associated the disparate masses of the electrons and heavy particles, through *sub-cycling*. In essence, sub-cycling implies the use of multiple time-scales.

In PROBEPIIC we define separately an electron time-step (`dte`) and an ion time-step (`dti`). The electron time-step is much smaller than the ion time-step, since the mean velocity of electrons is much greater than ion mean velocity. Now, if different time-scales are used we must be sure to structure the program so that the particles are not moving out of phase relative to one another through (real) time. Thus, the electrons must be moved an integral number of times (equal the ratio of the ion and electron time steps) before the ions are moved once. Consider the following illustrative example. In the case of the ion beam the beam velocity is approximately equal to $4.0 \times 10^4 [\frac{m}{s}]$ while the mean thermal velocity of the electrons is on the order of $4.0 \times 10^5 [\frac{m}{s}]$. Therefore, it makes sense to use an ion time-step that is ten times greater than the electron time-step. To compensate for this time-step difference we must move the electrons ten times for each time we move the ions.

Both the ions and electrons use the same sub-program (`mover.c`) as a mover, whose prototype is;

```
void mover(PARTICLEDEF particle[numpart], long int num_part, FLOAT timestep);
```

We simply pass the appropriate `dt` to the variable `timestep`.

As a result of sub-cycling, we do not waste computational time moving the ions at every time-step. When the number of ion super-particles is large, the computational savings may be substantial.

4.2.12 Checking Boundaries (`boundary.c`)

The sub-program `boundary.c` handles interactions of thermalized particles with computational and physical boundaries. Beam particles are handled by `boundary_beam.c`. In either routine we must determine what happens to a particle when it encounters the outer edges of the computational domain, the probe conductor, or the probe insulator.

In `boundary.c` any particle that exits through the periphery of the computational

domain is removed from the simulation. If a particle strikes the conducting surface of the probe, it is removed from the simulation and the probe current variable is incremented by the particle's charge. If a particle strikes the probe insulator it is reflected specularly, i. e. it undergoes a perfectly elastic collision and remains in the simulation.

`boundary_beam.c` is identical to `boundary.c` except for the way it treats particle interactions with the outer edge of the computational domain. Recall that we can divide the faces of the computational domain into three classes: the front(upstream) face, the back(downstream) face, and the side faces. Since beam particles have small radial components of velocity, they are injected only through the front face and then reflected off of the side faces back into the computational domain. This gives the effect of fluxing particles through the side faces without having to do so explicitly. The particle interactions with the front and back faces and the probe are treated the same as in `boundary.c`.

One might ask why not reflect all particles from the edges back into the computational domain, thus obviating the need for particle injection altogether. In theory this would be great, but in practice anomalous effects such as “numerical heating” might occur. It is best to continually introduce fresh particles.

As mentioned above, the particle interaction with the probe insulator was treated simply as a reflection; clearly this does not embody full physical reality, where the particles might stick to the insulator, or undergo re-combination to neutralize an ion. These interactions were not integrated into PROBEPIC at this time because we were not as interested in the global effect of the the presence of the insulator as we are interested in modeling the particle kinetics, and the plasma sheath around the probe. In the present version of PROBEPIC the insulator serves as a buffer zone between the conducting section of the probe and the back face of the computational

domain boundary, which minimizes the effect of non-thermalized particles from the boundary striking the probe. This allows us to better match the assumptions made in the Langmuir-Mott-Smith probe model, and consequently produce data in closer agreement with that model.

4.2.13 Output (`output_data.c`, `graphics.c`)

PROBEPIC is capable of producing both graphical and text output. The graphical output is generally only useful in the debugging process, while the text output provides the simulation results.

PROBEPIC uses standard VOGL [23] libraries in its graphics output sub-program `graphics.c`. The user may view either a front or side view of the computational domain (i. e. configuration space) in *real-time* (as opposed to storing particle trajectories and viewing them at a later time), that is, we can observe the trajectories of the particles as they are injected, reflected from the insulator, absorbed by the conductor, etc. . Obviously this is useful in determining if the code is qualitatively behaving as we expect, and producing snap-shots like the one shown in Figure 4.2.

Ultimately we are interested in quantitative results. PROBEPIC uses the sub-program `output_data.c` to write an output file at specified times which contains the following information: total number of time-steps executed, probe potential, number of super-electrons in the computational domain, number of super-ions in the computational domain, average number density of electrons, average number density of ions, average total current to the probe, average electron current to the probe, and average ion current to the probe. A sample of raw data from a PROBEPIC simulation is given in Appendix B.

Chapter 5

Simulation Results

Three computational experiments were conducted to evaluate the usefulness of PROBEPIC; two were initially planned, while a third was necessitated by some peculiar data that resulted from the first experiment. The first computation simulated a Langmuir probe in a rarefied, quiescent plasma. The second was designed to quantify the effect of varying the probe aspect ratio. The final experiment simulated the behavior of a Langmuir probe in a flowing plasma. The final simulation was designed to test the validity of a data reduction procedure developed by Keefer and Semak [15] for ion thruster plumes.

In this chapter we will present the results of all simulations and make a critical assessment of their validity using available theory, which was derived in chapter 2.

5.1 Probe in a Quiescent Plasma

Simulations were conducted in a quiescent plasma. The plasma conditions were chosen to correspond to the OLM domain, so that theoretical results would be readily available. Thus, we were able to determine whether PROBEPIC was producing physically valid results.

Two simulations were carried out: one for a hydrogen plasma, and another for an

(artificial) ion to electron mass ratio of 100. The “light ion” experiment was carried out in order to verify that ion collection was being properly modeled; in real plasmas, heavy, immobile ions contribute very little current. The simulation conditions for the two experiments are given in Table 5.1.

The results of the simulation are pictured in Figure 5.1. For probe potentials less than the plasma potential, the PROBEPIC results agree quite well with the Langmuir theory; however, in the electron saturation region, the PROBEPIC results diverge from the expected theoretical result. At the time when the data was coming in, this divergence was particularly troubling because, as explained in Chapter 2, the Langmuir theory gives the *maximum* current that should be measured. If there were to be any disagreement between the PROBEPIC and theoretical results, PROBEPIC data should always fall *below* the upper OLM threshold.

After rigorously checking the validity of the numerical algorithms used in PROBEPIC, it was discovered that the disagreement has a real, physical origin. The Langmuir theory was developed for an infinite cylindrical probe. In PROBEPIC and laboratory experiments finite length probes are used. We will now discuss what the effect of truncating a probe has on its overall VI characteristic.

Table 5.1: Quiescent plasma simulation plasma conditions.

Hydrogen	
n	$1.0 \cdot 10^9 [cm^{-3}]$
T	$2.0[eV]$
Plasma Potential	$0.0[V]$
Probe Aspect Ratio	45.92
$m_e/m_i = 100$	
n	$1.0 \cdot 10^9 [cm^{-3}]$
T	$2.0[eV]$
Plasma Potential	$0.0[V]$
Probe Aspect Ratio	45.92

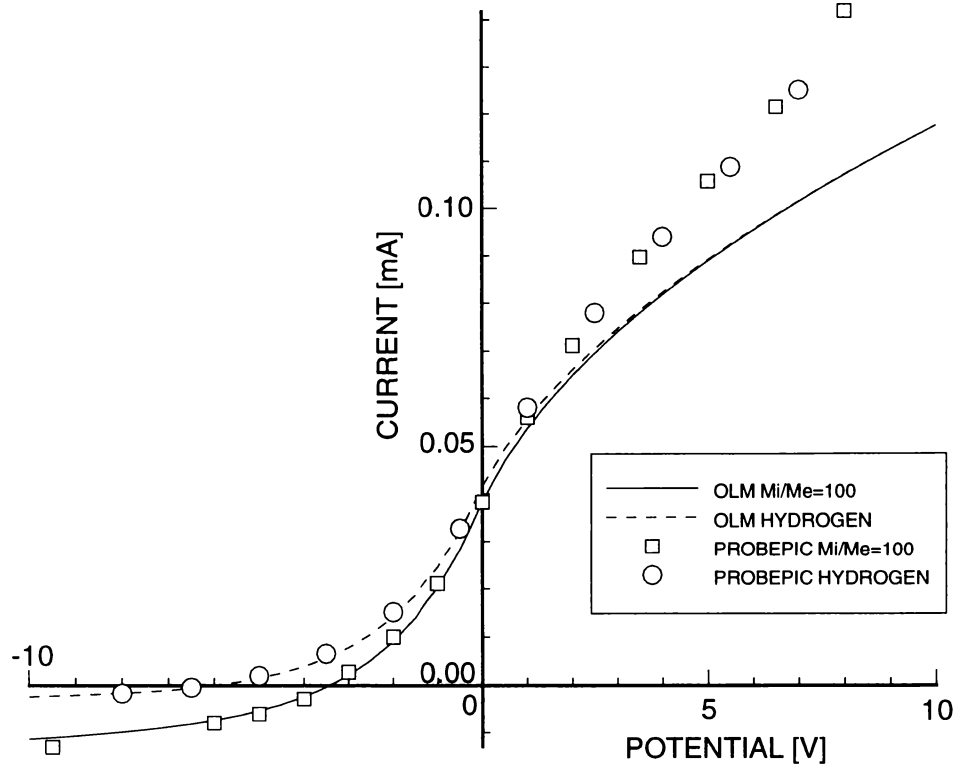


Figure 5.1: Comparison between PROBEPIC and Langmuir theory results.

5.2 Effect of Probe Aspect Ratio

Probes of finite length may produce significantly different VI characteristics than those predicted by infinite probe theory. This may be attributed the effect of the probe tip, which may produce overall electric field structures that differ significantly from infinite probes. It can be shown (see Jackson [18]) that sharp corners on conductors create intense electric fields in the region around the discontinuity. Thus, the sharp corner of the probe tip creates large electric fields that are not accounted for in the infinite probe model. The tip also creates axial fields (whereas only radial fields are included in the infinite probe model). These axial fields draw particles from in front of the probe to increase the overall current; as shown in Figure 5.2, current is then

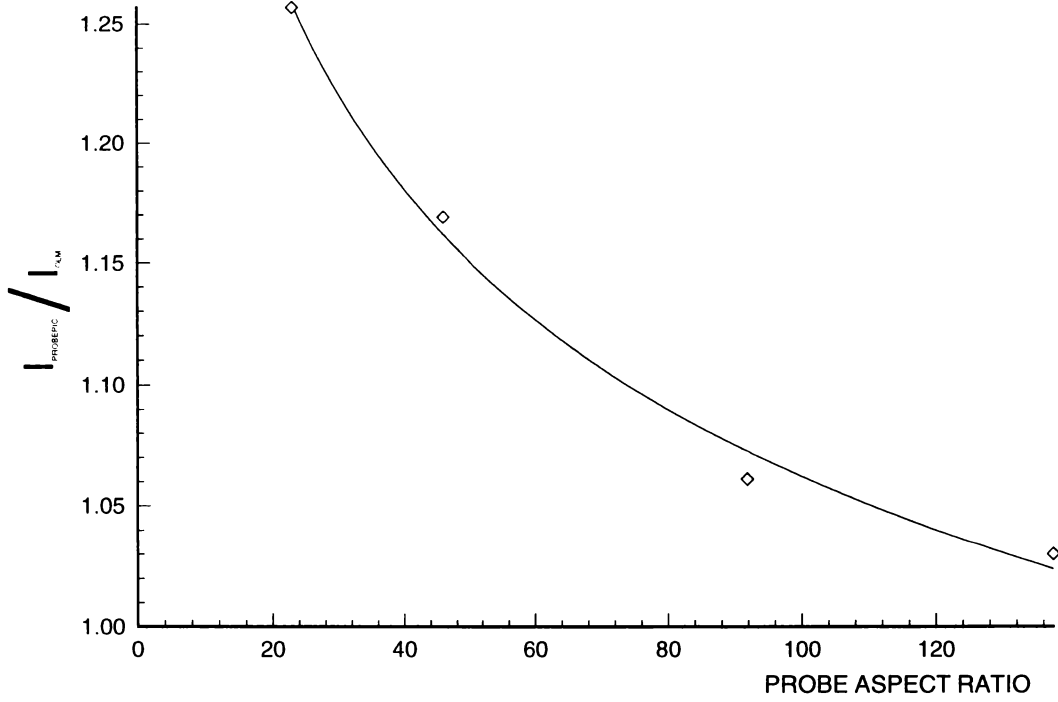


Figure 5.2: Illustration of the effect of probe aspect ratio on agreement with infinite probe Langmuir theory.

also a function of probe aspect ratio.

The relevant parameter in this discussion is the probe aspect ratio, γ ,

$$\gamma = \frac{l_p}{r_p} , \quad (5.1)$$

where, l_p is the probe length and r_p is the probe radius. As the probe aspect ratio becomes increasingly large, we expect the probe current to converge to the infinite probe result. Indeed, this effect was observed using PROBEPIIC, and is illustrated in Figure 5.2.

The points plotted were obtained using a hydrogen plasma at the operating conditions listed in Table 5.1, and a 5[V] probe potential; only the aspect ratio of the probe was varied.

In conclusion, the deviation of PROBEPIC results shown in Figure 5.1 may be attributed to the effect of finite probe length. As a result of the present study, as indicated in Figure 5.2, experimentalists should consider using probes with aspect ratios greater than 200 (at least for probes operating in the simulated plasma regime) if close agreement with theory is desired. A more comprehensive study of probes operating in different plasma regimes might be a valuable future application of PROBEPIC.

5.3 Probe in a Flowing Plasma

The final simulation was designed to test the validity of a data reduction procedure developed by Keefer and Semak [15] for ion thruster plumes. As discussed in chapter 2, the analysis of experimental data is quite difficult. To determine the electron temperature (assuming the distribution is Maxwellian), we must separate electron and ion current, and plot the natural logarithm of the electron current in the electron retarding region to obtain a straight line which may be correlated with electron temperature. The current separation process is complicated by the presence of the ion beam; in short, Keefer and Semak developed an iterative procedure in which the ion current was varied until a satisfactory linear electron current region was obtained, from which the electron temperature and number density could be extracted.

The author was given experimental data and proposed plasma conditions (as calculated from the new data reduction procedure). These values are shown in Table 5.2.

These values were input into PROBEPIC to generate a VI characteristic. Unfortunately, it became immediately apparent that something was wrong with the given data. For example, at electron saturation the electron current should be

$$I_{s,e} = \frac{n_e \bar{c}}{4} . \quad (5.2)$$

Table 5.2: Data from analysis of Keefer and Semak.

n_e	$3.1 \cdot 10^9 [cm^{-3}]$
T_e	$0.228 [eV]$
Plasma Potential	$0.96 [V]$
$I_{s,e}$	$4.7 \cdot 10^{-2} [mA]$
$I_{s,i}$	$4.1 \cdot 10^{-3} [mA]$
Beam Velocity	$\approx 4.0 \cdot 10^4 [\frac{m}{s}]$
l_p	$0.381 [cm]$
r_p	$0.0127 [cm]$
Probe Aspect Ratio	30.0

A quick hand calculation using the values given in Table 5.2 results in an electron saturation current of $I_{s,e} = 1.2 \times 10^{-1} [mA]$, which is a factor of three greater than the value arrived at in their data analysis.

A PROBEPIC simulation was conducted to determine if the electron temperature arrived at in the data reduction scheme was valid. A number density value was chosen that was more consistent with the experimental data ($n \approx 1.04 \times 10^9 [cm^{-3}]$). The complete list of parameters used in the PROBEPIC simulation is given in Table 5.3.

The results of this simulation are shown in Figure 5.3. The PROBEPIC, Langmuir theory, and experimental data agree quite well in the electron retarding region of the V-I characteristic. Above electron saturation, the Langmuir theory and the experimental

Table 5.3: Conditions used in PROBEPIC and in the theoretical curve for the plasma beam.

n	$1.04 \cdot 10^9 [cm^{-3}]$
T	$0.24 [eV]$
Plasma Potential	$0.96 [V]$
l_p	$0.381 [cm]$
r_p	$0.0127 [cm]$
Probe Aspect Ratio	30.0

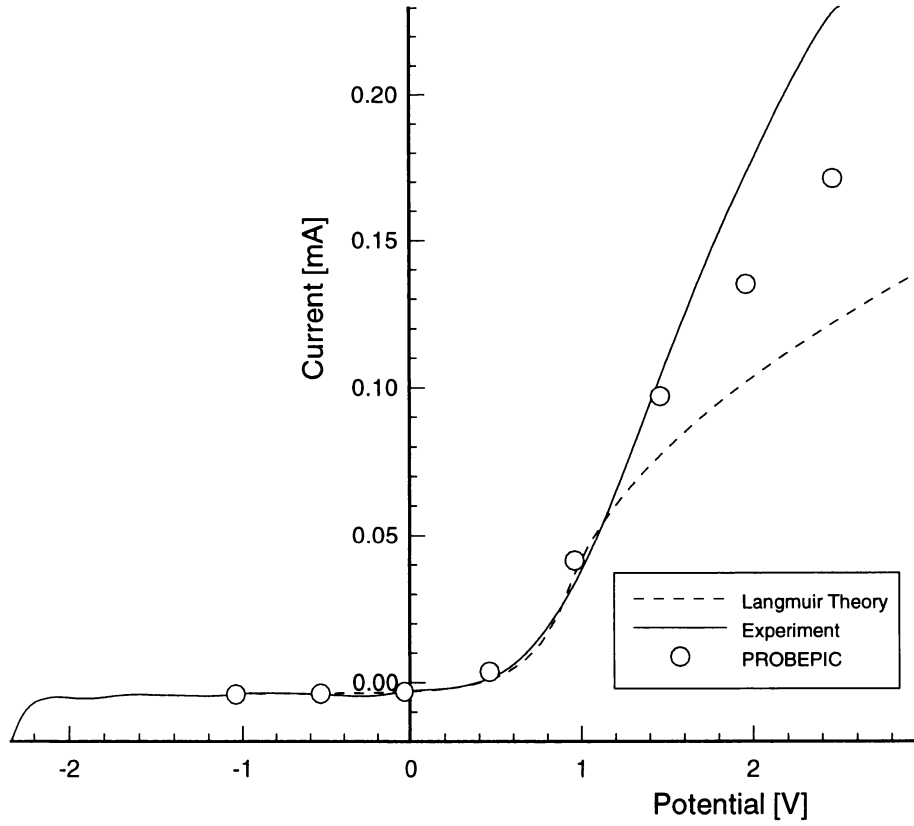


Figure 5.3: Comparison between PIC, Langmuir theory, and experimental results.

data disagree greatly, with the PROBEPICT result lying somewhere in between.

Since the electron temperature is calculated using the electron retarding region of the V-I characteristic, we may conclude that PROBEPICT verifies the temperature arrived at in the data reduction procedure ($T_e \approx 0.23$ [eV]). Above saturation, the strong deviation from the theoretical result is expected with the low aspect ratio probe used, as explained in the previous section. In this region, the PROBEPICT result is closer to the experimental data than the theoretical result. Further steps may be taken to bring the experimental and probepic results into closer agreement. By lowering the electron temperature, the PROBEPICT electron saturation current will be lowered and the current collection above saturation will be increased — bringing the PROBEPICT

result into closer agreement with the experimental data.

The question still remains as to why the data reduction procedure of Semak predicts an incorrect value of number density. The number density is calculated from the temperature. Since the Semak and PROBEPIC predicted temperatures agree, it is likely that Semak has an error in the formula he uses to calculate number density. It should be mentioned that Semak's predicted number density is off by a factor of approximately π , which may point to a simple omission of that factor in the calculation of the probe area.

Chapter 6

Conclusion

PROBEPIC, a Particle-In-Cell code, has been developed to simulate the behavior of finite length Langmuir probes. Results for the simulation of probes in quiescent plasmas and ion beams have been shown to be in agreement with both theory and experimental results. Also, the effect of probe aspect ratio on current collection has been qualitatively illustrated.

This thesis contributes to understanding Langmuir probes on two levels. First, it develops PIC specific techniques that are vital to producing a valid simulation. The proper technique for fluxing particles into the computational domain, which was developed in Chapter 4, proved to be the greatest theoretical challenge in the development of PROBEPIC; consequently, the details will probably be useful to future PIC practitioners. Second, PROBEPIC may provide a valuable tool for interpreting experimental Langmuir probe data.

6.1 Suggestions for the Improvement of PROBEPIC

The present version of PROBEPIC is in many ways a first-draft — there is room for improvement in both computational efficiency and more accurate modeling of the underlying physics. The purpose of this work was to verify accurate modeling of

the physical processes; to this end, widely used, robust algorithms were employed. While these methods may not be the most computationally efficient, they minimize the “surprises” that are often indigenous to using “tricks”. We now have a code that apparently models reality, at least to first order, quite well. Future work on PROBEPIC may now focus on improving the computational efficiency and integrating more realistic physical interactions.

To improve the computational efficiency the following areas should be investigated:

- Profiling software to find the fraction of time used by each subroutine.
- Applicability of the hybrid model.
- Effect of grid density on macroscopic results.
- Inexact field solvers.
- Vectorization of source code.

As discussed in the previous chapter, implementing a fluid model for the electrons would eliminate the need for two different time-scales. This may allow results to be obtained more quickly; however, the author is uncertain about the suitability of the hybrid method for this particular problem.

The grid density in PROBEPIC was chosen to resolve plasma oscillations; however, it is uncertain as to whether modeling these oscillations is necessary to obtain satisfactory results for the mass motion of the plasma. If the grid density can be relaxed, we will have fewer grid-points; this will cause the program to both occupy less RAM and to run faster.

Since we now have a field solver that gives accurate results, we can try to implement a faster, more exotic field solver, and use the present one to determine the accuracy of

the new one. A suggested technique to look into is the ICCG method, which apparently works quite well. The author is uncertain as to how much faster the field solver can be made since the present field solver uses libraries that have been professionally optimized for specific machines and are consequently very fast.

A final, more radical, computational optimization worth considering is a complete restructuring of the sub-programs to make them more amenable to vectorization. This would allow the code to take full advantage of the vector architecture found in most super-computers. PROBEPIC was run on a convex C4 super-computer at the AEDC High Performance Computer Center. The performance was unimpressive, in fact, slower than an SGI R8000 workstation. This can be attributed to the fact that PROBEPIC is composed almost entirely of scalar code. A completely vectorized version of PROBEPIC would probably run an order of magnitude faster on a super-computer.

A few aspects of the physics modeled in PROBEPIC should be considered further; more specifically,

- Injection angle of the ion beam.
- Interaction of particles with the insulator surface.

A major unknown variable in the ion beam simulation is the proper angular distribution for injected particles. The angular distribution used in PROBEPIC is simply an educated guess. A more accurate method would be to use the ion thruster code by Peng [14] to study the proper direction to inject particles.

As mentioned in the previous section, the physics of the interaction of particles with the probe insulator was not rigorously modeled. A more thorough treatment of this interaction would involve adding the proper physics(e. g. surface contact potential,

recombination, etc.) to `boundary.c`. Studies could then be made to quantify the effect of the insulator, if any, on the overall probe response.

6.2 Suggestions for Future Implementation

Aside from improvements that could be made to the code, further work should include modeling different experiments, and developing PROBEPIC as a diagnostic aid for evaluating data taken from actual experiments.

The possibilities for its future application are virtually limitless. The results of Section 5.2 indicate that a more comprehensive treatment of the effect of probe aspect ratio throughout the range of Langmuir probe applicability may be valuable. Also, more work needs to be done to refine the data analysis technique of Keefer and Semak. In an even broader context, PROBEPIC might be modified to simulate other electrostatic plasma devices, such as ion thrusters, or plasma semiconductor etching devices.

The simulation results for a quiescent plasma presented in Chapter 5 show that PROBEPIC works; but who cares if we can model experiments for which we already have good theoretical models? The full value of PROBEPIC will be realized only when it is used to evaluate and predict experimental results for which no satisfactory analytical model exists. To some degree this was the case with the probe in a plasma beam. For PROBEPIC to be used as a diagnostic tool a “front end” will have to be developed. This front end would serve as a bridge between the raw experimental data and PROBEPIC; it would provide initial guesses for plasma parameters needed by PROBEPIC by scrutinizing the experimental data. Further, it would establish and monitor convergence criteria to determine when PROBEPIC had iterated to the experimental result.

Many lessons have been gleaned in the development of PROBEPIC. It is the author's hope that the effort spent in including adequate detail in this thesis will allow it to serve as a starting point for future PIC practitioners at UTSI, thus incrementing the overall progress of the institution.

BIBLIOGRAPHY

Bibliography

- [1] Allen, J.E., Boyd, R.L.F., and Reynolds, P.(1957), *Proc. Phys. Soc. B.*, **70**, 297.
- [2] Bernstein, I.B. and Rabinowitz, I.(1959), *Phys. Fluids*, **2**, 112.
- [3] Langmuir, I. and Mott-Smith, H.M.. *Collected Works of Irving Langmuir*. Pergamon, 1961.
- [4] Bohm, D.. *Minimum Ionic Kinetic Energy for a Stable Sheath*. New York : McGraw-Hill, 1949.
- [5] Laframboise, J.G. and Parker, L.W.(1973), *Phys. Fluids*, **16**, 629.
- [6] Hester, S.D. and Sonin, A.A.(1969), *Phys. Fluids*, **13**, 1265.
- [7] C.K. Birdsall and A.B. Langdon. *Plasma Physics Via Computer Simulation*. New York : Adam Hilger, 1991.
- [8] R.W. Hockney and J.W. Eastwood. *Computer Simulation Using Particles*. Philadelphia : Institute of Physics Publishing, 1988.
- [9] Fife, J.M.. *Two-Dimensional Hybrid Particle-In-Cell Modeling of Hall Thrusters*. Master's Thesis: Massachusetts Institute of Technology, 1995.
- [10] Kershaw, D.S.(1978), *Journal of Computational Physics*, **26**, 43.

- [11] Meijerink, J.A. and Van Der Vorst, H.A.(1981), *Journal of Computational Physics*, **44**, 134.
- [12] Press, Teukolsky, Vetterling, Flannery. *Numerical Recipes in C*. New York : Cambridge University Press, 1994.
- [13] Wilhelmus M. Ruyten. *Density-Conserving Shape Factors for Particle Simulations in Cylindrical and Spherical Coordinates*. *Journal of Computational Physics*, **105**,1993.
- [14] Peng, X.. *Particle Simulation of Grid Erosion in an Ion Thruster*. Dissertation : The University of Tennessee, 1991.
- [15] D. Keefer and V.V. Semak. *Measurements of Radial and Axial Distributions of Ion Thruster Plasma Parameters Using a Langmuir Probe*. (unpublished).
- [16] Jackson, J.D.. *Classical Electrodynamics*. New York : John Wiley & Sons, 1975.
- [17] Hershkowitz, Noah. *How Langmuir Probes Work*. *Plasma Diagnostics : Discharge Parameters and Chemistry*, Vol.1,pgs.113-183, Academic Press, 1989.
- [18] Holt, E.H. and Haskell, R.E.. *Foundations of Plasma Dynamics*. New York : The Macmillan Company, 1965.
- [19] Chen, F.F.. *Electric Probes*. New York : Plasma Diagnostic Techniques, Academic Press, 1965.
- [20] Hershkowitz, Noah.. *How Does the Potential Get from A to B in a Plasma*. IEEE Review Paper, 1993.
- [21] Chung, Talbot, Touryan. *Electric Probes in Stationary and Flowing Plasmas*. New York : Springer - Verlag, 1975.

- [22] Oak Ridge National Laboratory. *Linear Algebra Package*.
- [23] Eric H. Echimna. *Very Ordinary Graphics Language*.
- [24] V.P. Gopinath, P. Mirrashidi, D. Cooperberg, V. Vahedi, J.Verboncoeur, and C.K. Birdsall. *XPDC2 - R0 A 2 Dimensional Electrostatic Code*.,University of California, Berkely(Unpublished).

APPENDIX

Appendix A

PROBEPIC Source Code

The source code for the PROBEPIC sub-programs are given below in the order listed in Table 4.1. A soft copy of the source code is included on the floppy affixed to the inner back cover.

A.1 boundary.c

```
#define EXTERN extern
#include "lprobe60.h"

/* This function sorts out which particles have crossed a boundary, i.e.
   the probe or extent of computational domain. Particles crossing the
   probe boundary are counted as current */

void boundary (PARTICLEDEF particle[numpart], long int *num_part, FLOAT q)
{
    long int i, i2, numtemp;
    FLOAT r_part, vzold, vyold;

    numtemp=*num_part; i2=-1;

    for (i=0; i < numtemp; i++){
        i2=i2+1;
        r_part=particle[i2].r;

        /* OUT OF COMPUTATIONAL DOMAIN */

        if (particle[i2].x > 0.0 || particle[i2].x < -length ||
            particle[i2].y > radius || particle[i2].y < -radius ||
            particle[i2].z > radius || particle[i2].z < -radius){
```

```

particle[i2].x=particle[*num_part-1].x;
particle[i2].y=particle[*num_part-1].y;
particle[i2].z=particle[*num_part-1].z;
particle[i2].r=particle[*num_part-1].r;
particle[i2].vx=particle[*num_part-1].vx;
particle[i2].vy=particle[*num_part-1].vy;
particle[i2].vz=particle[*num_part-1].vz;
particle[i2].q=particle[*num_part-1].q;
particle[i2].m=particle[*num_part-1].m;
i2=i2-1;*num_part = *num_part - 1;
}

/* CONTACT WITH PROBE */

else if (r_part <= r_probe && particle[i2].x >=-(l_probe+l_insul) &&
particle[i2].x <=-l_insul){
particle[i2].x=particle[*num_part-1].x;
particle[i2].y=particle[*num_part-1].y;
particle[i2].z=particle[*num_part-1].z;
particle[i2].r=particle[*num_part-1].r;
particle[i2].vx=particle[*num_part-1].vx;
particle[i2].vy=particle[*num_part-1].vy;
particle[i2].vz=particle[*num_part-1].vz;

particle[i2].m=particle[*num_part-1].m;

current=current + particle[i2].q;
if(q == q_elec ) curre=curre + particle[i2].q;
else if(q == q_ion) curri=curri + particle[i2].q;

particle[i2].q=particle[*num_part-1].q;
i2=i2-1;*num_part = *num_part - 1;
}

/* CONTACT WITH CYLINDRICAL SURFACE OF INSULATOR */

else if (r_part <= r_insul && particle[i2].x >=-(l_insul-cell_width)){
vzold=particle[i2].vz;
vyold=particle[i2].vy;
particle[i2].vz=(vzold*(-particle[i2].z*particle[i2].z +
particle[i2].y*particle[i2].y)
- 2.0*vyold*particle[i2].y*particle[i2].z)/(particle[i2].r*particle[i2].r) ;
particle[i2].vy=(vyold*(particle[i2].z*particle[i2].z -
particle[i2].y*particle[i2].y)
- 2.0*vzold*particle[i2].y*particle[i2].z)/(particle[i2].r*particle[i2].r) ;
}

```

```

/* CONTACT WITH FLAT FACE OF INSULATOR */

else if (r_part <= r_insul && particle[i2].x >= -(1_insul)){
    particle[i2].vx=-particle[i2].vx;
}
}
}

```

A.2 boundary_beam.c

```

#define EXTERN extern
#include "lprobe60.h"

/* This function sorts out which particles have crossed a boundary, i.e.
the probe or extent of computational domain. Particles crossing the
probe boundary are counted as current */

void boundary_beam (PARTICLEDEF particle[numpart], long int *num_part, FLOAT q)
{
    long int i, i2, numtemp;
    FLOAT r_part, vzold, vyold;

    numtemp=*num_part; i2=-1;

    for (i=0; i < numtemp; i++){
        i2=i2+1;
        r_part=particle[i2].r;

        /* OUT OF COMPUTATIONAL DOMAIN */

        if (particle[i2].x > 0.0 || particle[i2].x < -length){

            particle[i2].x=particle[*num_part-1].x;
            particle[i2].y=particle[*num_part-1].y;
            particle[i2].z=particle[*num_part-1].z;
            particle[i2].r=particle[*num_part-1].r;
            particle[i2].vx=particle[*num_part-1].vx;
            particle[i2].vy=particle[*num_part-1].vy;
            particle[i2].vz=particle[*num_part-1].vz;
            particle[i2].q=particle[*num_part-1].q;
            particle[i2].m=particle[*num_part-1].m;
            i2=i2-1; *num_part = *num_part - 1;
        }

        /* REFLECTION OFF BOUNDARY */

        else if (particle[i2].r >= radius){

```

```

    vzold=particle[i2].vz;
    vyold=particle[i2].vy;
    particle[i2].vz=(vzold*(-particle[i2].z*particle[i2].z +
        particle[i2].y*particle[i2].y)- 2.0*vyold*particle[i2].y*
        particle[i2].z)/(particle[i2].r*particle[i2].r) ;
    particle[i2].vy=(vyold*(particle[i2].z*particle[i2].z -
        particle[i2].y*particle[i2].y) - 2.0*vzold*particle[i2].y*
        particle[i2].z)/(particle[i2].r*particle[i2].r) ;
}

/* CONTACT WITH PROBE */

else if (r_part <= r_probe && particle[i2].x >=-(l_probe+l_insul) &&
        particle[i2].x <=-l_insul){
    particle[i2].x=particle[*num_part-1].x;
    particle[i2].y=particle[*num_part-1].y;
    particle[i2].z=particle[*num_part-1].z;
    particle[i2].r=particle[*num_part-1].r;
    particle[i2].vx=particle[*num_part-1].vx;
    particle[i2].vy=particle[*num_part-1].vy;
    particle[i2].vz=particle[*num_part-1].vz;

    particle[i2].m=particle[*num_part-1].m;

    current=current + particle[i2].q;
    if(q == q_elec ) curre=curre + particle[i2].q;
    else if(q == q_ion) curri=curri + particle[i2].q;

    particle[i2].q=particle[*num_part-1].q;
    i2=i2-1;*num_part = *num_part - 1;
}

/* CONTACT WITH CYLINDRICAL SURFACE OF INSULATOR */

else if (r_part <= r_insul && particle[i2].x >=-(l_insul-cell_width)){
    vzold=particle[i2].vz;
    vyold=particle[i2].vy;
    particle[i2].vz=(vzold*(-particle[i2].z*particle[i2].z +
        particle[i2].y*particle[i2].y)
        - 2.0*vyold*particle[i2].y*particle[i2].z)/(
        particle[i2].r*particle[i2].r) ;
    particle[i2].vy=(vyold*(particle[i2].z*particle[i2].z -
        particle[i2].y*particle[i2].y)
        - 2.0*vzold*particle[i2].y*particle[i2].z)/(
        particle[i2].r*particle[i2].r) ;
}

/* CONTACT WITH FLAT FACE OF INSULATOR */

```

```

        else if (r_part <= r_insul && particle[i2].x >= -(l_insul)){
            particle[i2].vx=-particle[i2].vx;
        }
    }
}

```

A.3 charge_weight.c

```

#define EXTERN extern
#include "lprobe60.h"

/* This function weights the charge to the array */

void qweight(GRIDDEF array[ngp1][ngp2],PARTICLEDEF particle[numpart],long int number)
{
    unsigned long positionx,positiony;
    long int count1;
    FLOAT dr,dr1,dr2,dz,dz1,dz2;

    for(count1=0;count1 < number;count1++){

        if (particle[count1].r <= radius){

            locatex(array,ngp2,particle[count1].x,&positionx);
            locatey(array,ngp1,particle[count1].r,&positiony);

            /*    SAVE CELL LOCATION FOR FORCE WEIGHT    */

            particle[count1].cellx=positionx;
            particle[count1].celly=positiony;

            dr=array[positiony+1][positionx].y*array[positiony+1][positionx].y
            - array[positiony][positionx].y
              *array[positiony][positionx].y;
            dr1=particle[count1].r - array[positiony][positionx].y;
            dr2=array[positiony+1][positionx].y - particle[count1].r;

            dz=array[positiony][positionx+1].x - array[positiony][positionx].x;
            dz1=particle[count1].x - array[positiony][positionx].x;
            dz2=array[positiony][positionx+1].x - particle[count1].x;

            array[positiony][positionx].q_dens=array[positiony][positionx].q_dens
            + particle[count1].q *
              dr2 * dz2
              * (2.0 * array[positiony+1][positionx].y + 3.0
            * array[positiony][positionx].y

```

```

        - particle[count1].r) / (2.0 * dr * dz);
    array[positiony][positionx+1].q_dens=array[positiony][positionx+1].q_dens
+ particle[count1].q *
        dr2 * dz1
        * (2.0 * array[positiony+1][positionx+1].y + 3.0
* array[positiony][positionx].y
        - particle[count1].r) / (2.0 * dr * dz);
    array[positiony+1][positionx].q_dens=array[positiony+1][positionx].q_dens
+ particle[count1].q *
        dr1 * dz2
        * (3.0 * array[positiony+1][positionx].y + 2.0
* array[positiony][positionx].y
        - particle[count1].r) / (2.0 * dr * dz);
    array[positiony+1][positionx+1].q_dens=array[positiony+1][positionx+1].q_dens
+ particle[count1].q *
        dr1 * dz1
        * (3.0 * array[positiony+1][positionx+1].y + 2.0
* array[positiony][positionx+1].y
        - particle[count1].r) / (2.0 * dr * dz);
    }
}
}

```

A.4 force_weight.c

```

#define EXTERN extern
#include "lprobe60.h"

/* This function weights the field from the array */

void fweight(GRIDDEF array[ngp1][ngp2],PARTICLEDEF particle[numpart],long int number)
{
    long int count1;
    FLOAT dr,dr1,dr2,dz,dz1,dz2;

    /*      CLEAN UP OLD ELECTRIC FIELD      */

    for (count1=0;count1 < number;count1++){
        particle[count1].Ey=particle[count1].Ex=0.0;
    }

    for(count1=0;count1 < number;count1++){

        if (particle[count1].r <= radius){

            dr=array[particle[count1].celly+1][particle[count1].cellx].y*

```

```

        array[particle[count1].celly+1][particle[count1].cellx].y -
        array[particle[count1].celly][particle[count1].cellx].y
        *array[particle[count1].celly][particle[count1].cellx].y;
    dr1=particle[count1].r - array[particle[count1].celly][particle[
count1].cellx].y;
    dr2=array[particle[count1].celly+1][particle[count1].cellx].y - p
article[count1].r;

    dz=array[particle[count1].celly][particle[count1].cellx+1].x -
        array[particle[count1].celly][particle[count1].cellx].x;
    dz1=particle[count1].x - array[particle[count1].celly][particle[c
ount1].cellx].x;
    dz2=array[particle[count1].celly][particle[count1].cellx+1].x - p
article[count1].x;

    particle[count1].Ey=array[particle[count1].celly][particle[count1
].cellx].Ey*dr2
    * dz2
                                * (2.0 * array[particle[count1].celly+1][particle
[count1].cellx].y
    + 3.0
                                * array[particle[count1].celly][particle[count1]
.cellx].y -
                                particle[count1].r) / (2.0 * dr * dz)
                                + array[particle[count1].celly][particle[count1]
.cellx+1].Ey*dr2
    * dz1
                                * (2.0 * array[particle[count1].celly+1][particl
e[count1].cellx+1].y
    + 3.0 *
                                array[particle[count1].celly][particle[count1].
cellx].y -
    particle[count1].r) /
                                (2.0 * dr * dz ) + array[particle[count1].celly
+1][
    particle[count1].cellx+1].Ey*
                                dr1 * dz1 * (3.0 * array[particle[count1].celly
+1][
    particle[count1].cellx+1].y +
                                2.0 * array[particle[count1].celly][particle[co
unt1].cellx+1].y -
                                particle[count1].r) / (2.0 * dr * dz) +
                                array[particle[count1].celly+1][particle[count1
].cellx].Ey*dr1
    * dz2 * (3.0 *
                                array[particle[count1].celly+1][particle[count1]
.cellx].y + 2.0 *
                                array[particle[count1].celly][particle[count1].ce
llx].y -
    particle[count1].r)

```



```

        / (2.0 * dr * dz);

        particle[count1].Ex=array[particle[count1].celly][particle[count1]
        .cellx].Ex*dr2
        * dz2
        * (2.0 * array[particle[count1].celly+1][particle[
        count1].cellx].y
        + 3.0
        * array[particle[count1].celly][particle[count1].c
        ellx].y -
        particle[count1].r) / (2.0 * dr * dz)
        + array[particle[count1].celly][particle[count1].c
        ellx+1].Ex*dr2
        * dz1
        * (2.0 * array[particle[count1].celly+1][particle[
        count1].cellx+1].y
        + 3.0 *
        array[particle[count1].celly][particle[count1].ce
        llx].y -
        particle[count1].r) /
        (2.0 * dr * dz ) + array[particle[count1].celly+1][
        particle[count1].cellx+1].Ex*
        dr1 * dz1 * (3.0 * array[particle[count1].celly+1][
        particle[count1].cellx+1].y +
        2.0 * array[particle[count1].celly][particle[count1].
        cellx+1].y -
        particle[count1].r) / (2.0 * dr * dz) +
        array[particle[count1].celly+1][particle[count1].cell
        x].Ex*
        dr1 * dz2 * (3.0 *
        array[particle[count1].celly+1][particle[count1].cell
        x].y + 2.0 *
        array[particle[count1].celly][particle[count1].cellx].y -
        particle[count1].r)
        / (2.0 * dr * dz);

    }

}

}

```

A.5 graphics.c

```

#define EXTERN extern
#include "lprobe60.h"

void graph(GRIDDEF array[ngp1][ngp2],PARTICLEDEF particle1[numpart],PARTICLE
DEF particle2[numpart])

```

```

{
static int count1,count2;
static float probepts[4][2],inspts[4][2];
static float xmouse, ymouse;
static PushButton SwitchButton, ExitButton,PrintButton;
backbuffer();
color(0);

/*if(t_count == 1){
    clear();
}*/

ClearMouseBuffer();

if (view == 1){
    color(0);
    clear();

    MakeButton(PRINTBOXx,PRINTBOXy, &PrintButton, "Print\0");
    if ( CheckButton(PrintButton) )
    {
        view = 3;
    }

    MakeButton(SWITCHBOXx, SWITCHBOXy, &SwitchButton, "Front\0");
    if ( CheckButton(SwitchButton) )
    {
        view = 0;
    }

    MakeButton(EXITBOXx, EXITBOXy, &ExitButton, "Exit\0");
    if ( CheckButton(ExitButton) ){
        vexit();
        exit ( 1 );
    }

    color(52);
    polyfill(0);
    rect(1.5*(array[0][0].x+0.5),-2.0*array[ngp_vert-1][0].y,
        1.5*(array[0][ngp_horiz-1].x+0.5),2.0*array[ngp_vert-1][0].y);
    move2(-0.95,0.0);
    draw2(1.0,0);

    color(50);
    /* for (count1=0;count1 < ngp_vert;count1++)
    {
        for (count2=0;count2 < ngp_horiz;count2++)
        {
            point2(1.5*(array[count1][count2].x+.5),2.0*array[count1][count2].y);
        }
    }*/
}

```

```

color(56);
probepts[0][0]=1.5*(.5-1_insul);          probepts[0][1]=2.0*r_probe;
probepts[1][0]=1.5*(.5-1_insul);          probepts[1][1]=-2.0*r_probe;
probepts[2][0]=1.5*(.5-1_insul-1_probe);  probepts[2][1]=-2.0*r_probe;
probepts[3][0]=1.5*(.5-1_insul-1_probe);  probepts[3][1]=2.0*r_probe;
inspts[0][0]=1.5*.5;inspts[0][1]=2.0*r_insul;
inspts[1][0]=1.5*.5;inspts[1][1]=-2.0*r_insul;
inspts[2][0]=1.5*(.5-1_insul);inspts[2][1]=-2.0*r_insul;
inspts[3][0]=1.5*(.5-1_insul);inspts[3][1]=2.0*r_insul;

if ( CheckButton(SwitchButton) )
{
    view = 0;
}

polyfill(1);
poly2(4,probepts);
color(50);
polyhatch(1); hatchang(90.0); hatchpitch(0.005);
poly2(4,inspts);

color(53);
for (count1=0;count1 < numelec;count1++)
{
    point2(1.5*(particle1[count1].x+.5),2.0*particle1[count1].y);
}
color(2);
for (count1=0;count1 < numion;count1++)
{
    point2(1.5*(particle2[count1].x+.5),2.0*particle2[count1].y);
}

if ( CheckButton(SwitchButton) )
{
    view = 0;
}
if ( CheckButton(ExitButton) ){
    vexit();
    exit ( 1 );
}

}

if (view == 0){
    clear();
    color(0);
    color(56);
    polyfill(1);

```

```

circle(0.0,0.0,2.5*r_probe);
polyhatch(1); hatchang(90.0); hatchpitch(0.005);
circle(0.0,0.0,2.5*r_insul);

MakeButton(PRINTBOXx,PRINTBOXy, &PrintButton, "Print\0");
if ( CheckButton(PrintButton) )
{
    view = 3;
}

MakeButton(SWITCHBOXx, SWITCHBOXy, &SwitchButton, "Side\0");
if ( CheckButton(SwitchButton) )
{
    view = 1;
}

MakeButton(EXITBOXx, EXITBOXy, &ExitButton, "Exit\0");
if ( CheckButton(ExitButton) ){
    vexit();
    exit ( 1 );
}
color(52);
polyfill(0);
circleprecision(64);
circle(0.0,0.0,2.5*radius);

color(53);
for (count1=0;count1 < numelec;count1++)
{
    point2(2.5*particle1[count1].z,2.5*particle1[count1].y);
}

if ( CheckButton(SwitchButton) )
{
    view = 1;
}
if ( CheckButton(ExitButton) ){
    vexit();
    exit ( 1 );
}

color(2);
for (count1=0;count1 < numion;count1++)
{
    point2(2.5*particle2[count1].z,2.5*particle2[count1].y);
}
}
swapbuffers();

if (view == 3)
{

```

```

vexit();
voutput("picture.ps");
vinit("cps");
color(0);
clear();
color(52);
polyfill(0);
rect(2.25*(array[0][ngp_horiz].x+.5),-3.5*array[ngp_vert][ngp_horiz].y,
      2.25*(array[0][0].x+0.42),3.5*array[ngp_vert][0].y);
move2(-0.95,0.0);
draw2(1.0,0);

color(50);
for (count1=0;count1 < ngp_vert+1;count1++)
{
    for (count2=0;count2 < ngp_horiz+1;count2++)
    {
        point2(2.25*(array[count1][count2].x+0.42),3.5*array[count1][count2].y);
    }
}
color(56);
probepts[0][0]=2.25*(0.42-1_insul);          probepts[0][1]=3.5*r_probe;
probepts[1][0]=2.25*(0.42-1_insul);          probepts[1][1]=-3.5*r_probe;
probepts[2][0]=2.25*(0.42-1_insul-1_probe); probepts[2][1]=-3.5*r_probe;
probepts[3][0]=2.25*(0.42-1_insul-1_probe); probepts[3][1]=3.5*r_probe;
inspts[0][0]=2.25*0.42;inspts[0][1]=3.5*r_probe;
inspts[1][0]=2.25*0.42;inspts[1][1]=-3.5*r_probe;
inspts[2][0]=2.25*(0.42-1_insul);inspts[2][1]=-3.5*r_probe;
inspts[3][0]=2.25*(0.42-1_insul);inspts[3][1]=3.5*r_probe;

if ( CheckButton(SwitchButton) )
{
    view = 0;
}

polyfill(1);
poly2(4,probepts);
color(50);
polyhatch(1); hatchang(90.0); hatchpitch(0.005);
poly2(4,inspts);

color(53);
for (count1=0;count1 < numelec;count1++)
{
    point2(2.25*(particle1[count1].x+0.42),3.5*particle1[count1].y);
}

vexit();exit(1);

```

```

    }

}

short CheckButton ( PushButton TomButton )

{
    static float xmouse, ymouse;
    if ( slocator( &xmouse, &ymouse ))
    {
        if ( (xmouse > TomButton.x1PositionUp ) && (xmouse < TomButton.x2P
ositionUp) &&
            (ymouse < TomButton.y1PositionUp ) && (ymouse > TomButton.y2P
ositionUp) )
        {
            DrawButtonDown(TomButton);
            return 1;
        }
    }
    return 0;
}

void ClearMouseBuffer()

{
    float xmouse, ymouse;

    slocator( &xmouse, &ymouse );
}

short MakeButton ( float x1, float y1, PushButton *TomButton, char ButtonText[20])
{
    TomButton->x1PositionUp = x1;
    TomButton->y1PositionUp = y1;
    TomButton->x2PositionUp = x1+xButtonOffset;
    TomButton->y2PositionUp = y1-yButtonOffset;
    strcpy(TomButton->ButtonText, ButtonText);
    TomButton->TextColor = 3;
    TomButton->aBorderColorUp = lightgray;
    TomButton->bBorderColorUp = gray;
    TomButton->cBorderColorUp = gray;
    TomButton->dBorderColorUp = lightgray;
    TomButton->aBorderColorDown = gray;
    TomButton->bBorderColorDown = lightgray;
    TomButton->cBorderColorDown = lightgray;
    TomButton->dBorderColorDown = gray;
    TomButton->FillColorUp = midblue;
    TomButton->FillColorDown = gray;
    DrawButton(*TomButton);
}

```

```

    return ( 0 );
}

void DrawButton(PushButton TomButton)
{
    float f , g;

    font ("/usr/people/tmarkusi/vogle/font/futura.m");

    color(TomButton.FillColorUp);
    polyfill(TomButton.FillColorUp);
    rect(TomButton.x1PositionUp, TomButton.y1PositionUp,
        TomButton.x2PositionUp, TomButton.y2PositionUp);

    color(TomButton.TextColor);
    f = fabs((TomButton.x1PositionUp - TomButton.x2PositionUp));
    g = fabs((TomButton.y2PositionUp - TomButton.y1PositionUp));
    boxtext(TomButton.x1PositionUp, TomButton.y2PositionUp, f, g,
        TomButton.ButtonText);

    move2(TomButton.x1PositionUp, TomButton.y1PositionUp);
    color(TomButton.aBorderColorUp);
    draw2(TomButton.x1PositionUp, TomButton.y2PositionUp);

    color(TomButton.bBorderColorUp);
    draw2(TomButton.x2PositionUp, TomButton.y2PositionUp);

    color(TomButton.cBorderColorUp);
    draw2(TomButton.x2PositionUp, TomButton.y1PositionUp);

    color(TomButton.dBorderColorUp);
    draw2(TomButton.x1PositionUp, TomButton.y1PositionUp);
}

void DrawButtonDown(PushButton TomButton)
{
    float f, g;

    color(TomButton.FillColorDown);
    polyfill(TomButton.FillColorDown);
    rect(TomButton.x1PositionUp, TomButton.y1PositionUp,
        TomButton.x2PositionUp, TomButton.y2PositionUp);

    color(TomButton.TextColor);
    f = fabs((TomButton.x1PositionUp - TomButton.x2PositionUp));
    g = fabs((TomButton.y2PositionUp - TomButton.y1PositionUp));
    boxtext(TomButton.x1PositionUp, TomButton.y2PositionUp, f, g,
        TomButton.ButtonText);
}

```

```

    move2(TomButton.x1PositionUp,TomButton.y1PositionUp);

    color(TomButton.aBorderColorDown);
    draw2(TomButton.x1PositionUp,TomButton.y2PositionUp);

    color(TomButton.bBorderColorDown);
    draw2(TomButton.x2PositionUp,TomButton.y2PositionUp);

    color(TomButton.cBorderColorDown);
    draw2(TomButton.x2PositionUp,TomButton.y1PositionUp);

    color(TomButton.dBorderColorDown);
    draw2(TomButton.x1PositionUp,TomButton.y1PositionUp);

}

```

A.6 grid.c

```

#define EXTERN extern
#include "lprobe60.h"

void gridgen(GRIDDEF array[ngp1][ngp2])
{
    long int count1,count2,cell1_count,cell6_count;
    FLOAT index1,index2;

    ngp_vert=ngp1;
    ngp_horiz=ngp2;
    cell1_count=15,cell6_count=74;

    index1=-1.0;
    for (count1=0;count1 < cell1_count;count1++){
        index1=index1+1.0;
        index2=-1.0;
        for (count2=0;count2 < ngp_horiz;count2++){
            index2=index2+1.0;
            array[count1][count2].x=-length+index2*cell_width;
            array[count1][count2].y=index1*cell_height1;
        }
    }
    index2=-1.0;
    for (count2=0;count2 < ngp_horiz;count2++){
        index2=index2+1.0;
        array[15][count2].x=-length+index2*cell_width;
        array[15][count2].y= array[14][count2].y+cell_height2;
    }
    index2=-1.0;
    for (count2=0;count2 < ngp_horiz;count2++){

```



```

        index2=index2+1.0;
        array[16][count2].x=-length+index2*cell_width;
        array[16][count2].y=array[15][count2].y+cell_height3;
    }
    index2=-1.0;
    for (count2=0;count2 < ngp_horiz;count2++){
        index2=index2+1.0;
        array[17][count2].x=-length+index2*cell_width;
        array[17][count2].y=array[16][count2].y+cell_height4;
    }
    index2=-1.0;
    for (count2=0;count2 < ngp_horiz;count2++){
        index2=index2+1.0;
        array[18][count2].x=-length+index2*cell_width;
        array[18][count2].y=array[17][count2].y+cell_height5;
    }
    index1=0.0;
    for (count1=19;count1 < cell6_count;count1++){
        index1=index1+1.0;
        index2=-1.0;
        for (count2=0;count2 < ngp_horiz;count2++){
            index2=index2+1.0;
            array[count1][count2].x=-length+index2*cell_width;
            array[count1][count2].y=array[18][count2].y+index1*cell_height6;
        }
    }

    /*      Clean-up grid for next time step      */
    for (count1=0;count1 < ngp_vert;count1++){
        for (count2=0;count2 < ngp_horiz;count2++){
            array[count1][count2].q_dens=array[count1][count2].Ex=
            array[count1][count2].Ey=0.0;
        }
    }

}

```

A.7 initialize.c

```

#define EXTERN extern
#include "lprobe60.h"

void initialize(PARTICLEDEF particle[numpart],long int *num_part,
               long int init_num,FLOAT super_init,FLOAT *v_table,FLOAT m,FLOAT q)
{
    long int i,rannum;
    FLOAT theta,phi,v,r;

```

```

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

    theta=pi*random_num[rancount];
    rancount ++;
    phi=2.0*pi*random_num[rancount];
    rancount ++;
    rannum=((long)100000.0*random_num[rancount]);
    rancount ++;
    v=v_table[rannum];
    rancount ++;
    r=radius * sqrt(random_num[rancount]);
    rancount ++;

    particle[*num_part].x=-length * random_num[rancount];
    rancount++;
    particle[*num_part].y=r*sin(theta);
    particle[*num_part].z=r*cos(theta);
    particle[*num_part].r=sqrt(particle[*num_part].y*particle
[*num_part].y
+ particle[*num_part].z * particle[*num_part].z);
    particle[*num_part].vx=v * sin(theta) *sin(phi);
    particle[*num_part].vy=v * cos(theta);
    particle[*num_part].vz=v * sin(theta) * cos(phi);
    particle[*num_part].Ex=0.0;
    particle[*num_part].Ey=0.0;
    particle[*num_part].q= q * super_init;
    particle[*num_part].m= m * super_init;
    (*num_part)++;

}

}

```

A.8 initialize_beam.c

```

#define EXTERN extern
#include "lprobe60.h"

void initialize_beam(PARTICLEDEF particle[numpart],long int *num_part,
    long int init_num,FLOAT super_init,FLOAT m,FLOAT q)
{
    long int i;
    FLOAT theta,v,r;

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

```

```

        r=radius * sqrt(random_num[rancount]);
rancount ++;
        theta=2.0*pi*random_num[rancount];
rancount ++;
        v=beam_vel;

        particle[*num_part].x=-length * random_num[rancount];
rancount++;
        particle[*num_part].y=r*sin(theta);
        particle[*num_part].z=r*cos(theta);
        particle[*num_part].r=sqrt(particle[*num_part].y*particle[*num
_part].y
        + particle[*num_part].z * particle[*num_part].z);
        particle[*num_part].vx=v;
        particle[*num_part].vy=0;
        particle[*num_part].vz=0;
        particle[*num_part].Ex=0.0;
        particle[*num_part].Ey=0.0;
        particle[*num_part].q= q * super_init;
        particle[*num_part].m= m * super_init;
        (*num_part)++;

        /*if(particle[*num_part].x > (-l_insul) && particle[*num_part].r
< (r_insul)){
            (*num_part)--;
            i--;
        }*/

    }

}

```

A.9 inject_beam.c

```

#define EXTERN extern
#include "lprobe60.h"
/* This function injects particles at each time step */

void inject_beam(PARTICLEDEF particle[numpart],long int *num_part,FLOAT
m,FLOAT q)
{
    long int i,i2;
    FLOAT theta,phi,v;

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

        v=beam_vel;

```

```

        theta=(pi / 36.0) * random_num[rancount]; /* Up to five degree
beam divergence */
        rancount ++;
        phi=2.0*pi*random_num[rancount];
        rancount ++;

        particle[*num_part].x=-length;
        particle[*num_part].y=radius * sqrt(random_num[rancount]);
        rancount ++;
        particle[*num_part].z=0;
        particle[*num_part].r=sqrt(particle[*num_part].y*particle[*num_part].y
+ particle[*num_part].z * particle[*num_part].z);
        particle[*num_part].vx=v * cos(theta);
        particle[*num_part].vy=v * sin(theta) * cos(phi);
        particle[*num_part].vz=v * sin(theta) * sin(phi);
        particle[*num_part].q= q * super_beam;
        particle[*num_part].m= m * super_beam;
        particle[*num_part].Ex=0.0;
        particle[*num_part].Ey=0.0;
        (*num_part)++;

        if (particle[*num_part-1].r < (0.2 * radius)){

                particle[*num_part-1].q= q * super_beam / ion_sub_inject;
                particle[*num_part-1].m= m * super_beam / ion_sub_inject;

                for(i2=0;i2 < (ion_sub_inject-1);i2++){

                        theta = -(pi / 36.0) + 2.0*(pi / 36.0) * random_num[rancount];
/* Up
to five degree beam divergence */
                        rancount ++;
/*
                        phi=2.0*pi*random_num[rancount];
                        rancount ++;*/

                        particle[*num_part].x=particle[*num_part-1].x;
                        particle[*num_part].y=particle[*num_part-1].y;
                        particle[*num_part].z=particle[*num_part-1].z;
                        particle[*num_part].r=particle[*num_part-1].r;
                        particle[*num_part].vx=v * cos(theta);
                        particle[*num_part].vy=v * sin(theta);
                        particle[*num_part].vz=0.0;
/*
                        particle[*num_part].vy=v * sin(theta) * cos(phi);
                        particle[*num_part].vz=v * sin(theta) * sin(phi); */
                        particle[*num_part].q= q * super_beam / ion_sub_inject;
                        particle[*num_part].m= m * super_beam / ion_sub_inject;
                        particle[*num_part].Ex=0.0;
                        particle[*num_part].Ey=0.0;
                        (*num_part)++;
                }
}

```

```

    }
}
}

```

A.10 inject_part.c

```

#define EXTERN extern
#include "lprobe60.h"
/* This function injects particles at each time step */

void inject(PARTICLEDEF particle[numpart],long int *num_part,FLOAT super_back,
FLOAT super_front,
            FLOAT super_side,long int superinj_back,long int superinj_front,
            long int superinj_side,FLOAT *v_table,FLOAT m,FLOAT q)
{
    long int i,rannum;
    FLOAT theta,phi,v;

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

        /*      FRONT      */
        theta=acos(sqrt(1 - random_num[rancount]));
        rancount ++;
        phi=2.0*pi*random_num[rancount];
        rancount ++;
        rannum=((long int)100000.0*random_num[rancount]);
        rancount ++;
        v=v_table[rannum];
        particle[*num_part].x=-length;
        particle[*num_part].y=height * random_num[rancount] - height / 2.0;
        rancount ++;
        particle[*num_part].z=width * random_num[rancount] - width / 2.0;
        rancount ++;
        particle[*num_part].r=sqrt(particle[*num_part].y*particle[*num_part]
        ].y
        + particle[*num_part].z * particle[*num_part].z);
        particle[*num_part].vx=v * cos(theta);
        particle[*num_part].vy=v*sin(theta)*cos(phi);
        particle[*num_part].vz=v*sin(theta)*sin(phi);
        particle[*num_part].q= q * super_front;
        particle[*num_part].m= m * super_front;
        particle[*num_part].Ex=0.0;
        particle[*num_part].Ey=0.0;
        (*num_part)++;
    }

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

```

```

        /*      BACK      */
        theta=acos(sqrt(1 - random_num[rancount]));
        rancount ++;
        phi=2.0*pi*random_num[rancount];
        rancount ++;
        rannum=((long int)100000.0*random_num[rancount]);
        rancount ++;
        v=v_table[rannum];
        do{
            particle[*num_part].x=0.0;
            particle[*num_part].y=height * random_num[rancount] - height
/ 2.0;
            rancount ++;
            particle[*num_part].z=width * random_num[rancount] - width
/ 2.0;
            rancount ++;
            particle[*num_part].r=sqrt(particle[*num_part].y*particle
[*num_part].y
+ particle[*num_part].z * particle[*num_part].z);
            particle[*num_part].vx=-v * cos(theta);
            particle[*num_part].vy=v*sin(theta)*cos(phi);
            particle[*num_part].vz=v*sin(theta)*sin(phi);
            particle[*num_part].Ex=0.0;
            particle[*num_part].Ey=0.0;
            particle[*num_part].q= q * super_back;
            particle[*num_part].m= m * super_back;
        }while( particle[*num_part].r <= r_insul);

        (*num_part)++;

    }

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

        /*      BOTTOM SIDE      */
        theta=acos(sqrt(1 - random_num[rancount]));
        rancount ++;
        phi=2.0*pi*random_num[rancount];
        rancount ++;
        rannum=((long)100000.0*random_num[rancount]);
        rancount ++;
        v=v_table[rannum];
        particle[*num_part].x=-length * random_num[rancount];
        rancount ++;
        particle[*num_part].y=-radius;
        particle[*num_part].z=width * random_num[rancount] - width /
2.0;
        rancount ++;
        particle[*num_part].r=sqrt(particle[*num_part].y*particle[*num

```

```

_part].y
+ particle[*num_part].z * particle[*num_part].z);
    particle[*num_part].vx=v * sin(theta) *sin(phi);
    particle[*num_part].vy=v * cos(theta);
    particle[*num_part].vz=v*sin(theta)*cos(phi);
    particle[*num_part].Ex=0.0;
    particle[*num_part].Ey=0.0;
    particle[*num_part].q= q * super_side;
    particle[*num_part].m= m * super_side;
    (*num_part)++;

    /*      TOP      Side      */
    theta=acos(sqrt(1 - random_num[rancount]));
    rancount ++;
    phi=2.0*pi*random_num[rancount];
    rancount ++;
    rannum=((long)100000.0*random_num[rancount]);
    rancount ++;
    v=v_table[rannum];
    particle[*num_part].x=-length * random_num[rancount];
    rancount ++;
    particle[*num_part].y=radius;
    particle[*num_part].z=width * random_num[rancount] - width / 2.0;
    rancount ++;
    particle[*num_part].r=sqrt(particle[*num_part].y*particle[*num_part].y
    + particle[*num_part].z * particle[*num_part].z);
    particle[*num_part].vx=v * sin(theta) * sin(phi);
    particle[*num_part].vy=-v * cos(theta);
    particle[*num_part].vz=v * sin(theta) * cos(phi);
    particle[*num_part].Ex=0.0;
    particle[*num_part].Ey=0.0;
    particle[*num_part].q= q * super_side;
    particle[*num_part].m= m * super_side;
    (*num_part)++;

    /*      FRONT SIDE      */
    theta=acos(sqrt(1 - random_num[rancount]));
    rancount ++;
    phi=2.0*pi*random_num[rancount];
    rancount ++;
    rannum=((long)100000.0*random_num[rancount]);
    rancount ++;
    v=v_table[rannum];
    particle[*num_part].x=-length * random_num[rancount];
    rancount ++;
    particle[*num_part].y=height * random_num[rancount] - height
    / 2.0;
    rancount ++;

```

```

        particle[*num_part].z=radius;
        particle[*num_part].r=sqrt(particle[*num_part].y*particle[*num
_part].y
+ particle[*num_part].z * particle[*num_part].z);
        particle[*num_part].vx=v * sin(theta) * cos(phi);
        particle[*num_part].vy=v * sin(theta) * sin(phi);
        particle[*num_part].vz=-v * cos(theta);
        particle[*num_part].Ex=0.0;
        particle[*num_part].Ey=0.0;
        particle[*num_part].q= q * super_side;
        particle[*num_part].m= m * super_side;
        (*num_part)++;

        /*    BACK SIDE    */
        theta=acos(sqrt(1 - random_num[rancount]));
        rancount ++;
        phi=2.0*pi*random_num[rancount];
        rancount ++;
        rannum=((long)100000.0*random_num[rancount]);
        rancount ++;
        v=v_table[rannum];
        particle[*num_part].x=-length * random_num[rancount];
        rancount ++;
        particle[*num_part].y=height * random_num[rancount] - heig
ht / 2.0;
        rancount ++;
        particle[*num_part].z=-radius;
        particle[*num_part].r=sqrt(particle[*num_part].y*particle
[*num_part].y
+ particle[*num_part].z * particle[*num_part].z);
        particle[*num_part].vx=v * sin(theta) * cos(phi);
        particle[*num_part].vy=v * sin(theta) * sin(phi);
        particle[*num_part].vz=v * cos(theta);
        particle[*num_part].Ex=0.0;
        particle[*num_part].Ey=0.0;
        particle[*num_part].q= q * super_side;
        particle[*num_part].m= m * super_side;
        (*num_part)++;

    }

}

```

A.11 locate.c

```
#define EXTERN extern
```



```

#include "lprobe60.h"

void locatex(GRIDDEF xx[ngp1][ngp2], unsigned long n, FLOAT x,
unsigned long *jj)
{
    unsigned long ju,jm,jl;
    int ascnd;

    jl=0;
    ju=n+1;
    ascnd=(xx[0][n-1].x > xx[0][1].x);
    while (ju-jl > 1) {
        jm=(ju+jl) >> 1;
        if (x > xx[0][jm].x == ascnd)
            jl=jm;
        else
            ju=jm;
    }
    *jj=jl;
}

void locatey(GRIDDEF xx[ngp1][ngp2], unsigned long n, FLOAT x,
unsigned long *jj)
{
    unsigned long ju,jm,jl;
    int ascnd;

    jl=0;
    ju=n+1;
    ascnd=(xx[n-1][0].y > xx[1][0].y);
    while (ju-jl > 1) {
        jm=(ju+jl) >> 1;
        if (x > xx[jm][0].y == ascnd)
            jl=jm;
        else
            ju=jm;
    }
    *jj=jl;
}

```

A.12 make_LU.c

```

#define EXTERN extern
#include "lprobe60.h"

/*      This function computes the LU decomposition of the
band matrix generated
      by differencing Poisson's Equation on the grid */
void make_LU(GRIDDEF array[ngp1][ngp2])
{

```

```

FILE *abddata,*abddata2;
int info;
long int i,j,k,jmax,kmax,count,counter1,counter2,counter3,
counter4,counter5;

abddata=fopen("abddata.dat","w+");
abddata2=fopen("abddata2.dat","w+");

/*  MAKE BANDS  */

jmax=ngp_vert;
kmax=ngp_horiz;
counter1=upper_width-1;
counter2=0;
counter3=-1;
counter4=0;
counter5=0;

/*  ****  */
/*      Initialize all non-relevant positions to zero  */
for(i=0;i < ngp;i++){
    band1[i]=band2[i]=band3[i]=band4[i]=band5[i]=0.0;
}

for (j=0;j < jmax;j++){
    for(k=0;k < kmax;k++){

        if(counter1<(ngp-1)){
            counter1++;
        }

        if(counter2<(ngp-1)){
            counter2++;
        }
        counter4++;counter5++;

        /*  y < r_probe  */
        if(j <= jprobe){
            if (j==0 && k==1){
                counter4=0;
            }
            if(j==1 && k==0){
                counter5=0;
            }
            if( k < kprobe){
                if(j==0){ /*
Lower boundary */
                    if(k==0) {
                        band1
[counter1]=0.0;

```

```

[counter2]=0.0;
[counter4]=0.0;
[counter5]=0.0;

    }
    else {
        band1
[counter1]=pi*cell_width;
        band2
[counter2]=pi*array[j+1][k].y
*array[j+1][k].y/(4.0*cell_width);
        band4
[counter4]=pi*array[j+1][k].y
*array[j+1][k].y/(4.0*cell_width);
        band5
[counter5]=0.0;
    }
}
else{ /*
Rest to top of probe */
    if(k==0) {
        band1
[counter1]=0.0;
        band2
[counter2]=0.0;
        band4
[counter4]=0.0;
        band5
[counter5]=0.0;
    }
    else{
        band1
[counter1]=2.0*pi*cell_width*
(array[j][k].y+(array[j+1][k].y-array[j][k].y)/2.0)/(
array[j+1][k].y-array[j][k].y);
        band2
[counter2]=pi*((array[j][k].y+
(array[j+1][k].y-array[j][k].y)/2.0)*(array[j][k].y+(
array[j+1][k].y-array[j][k].y)
/2.0)
        -(ar
ray[j][k].y-(array[j][k].y-
array[j-1][k].y)/2.0)*(array[j][k].y-(array[j][k].y-a
rray[j-1][k].y)/2.0))/cell_width;
        band4[
counter4]=pi*((array[j][k].y+
(array[j+1][k].y-array[j][k].y)/2.0)*(array[j][k].y+(a
rray[j+1][k].y-array[j][k].y)

```

```

/2.0)

                                                    -(ar
ray[j][k].y-(array[j][k].y-
array[j-1][k].y)/2.0)*(array[j][k].y-(array[j][k].y-arr
ay[j-1][k].y)/2.0))/cell_width;

                                                    band5[c
ounter5]=2.0*pi*
cell_width*(array[j][k].y-(array[j][k].y-array[j-1][k].
y)/2.0)/(array[j][k].y
-array[j-1][k].y);
                                                    }
                                                    }
                                                    }
                                                    else { /* on and inside p
robe */
                                                    band1[counter1]
=band2[counter2]=band4[counter4]
=band5[counter5]=0.0;
                                                    }

}
else if (j>jprobe && j<(jmax-1)){
    if((k > 0) && (k < (kmax-1))){
        band1[counter1]=2.0*pi*ce
ll_width*(array[j][k].y+(
array[j+1][k].y-array[j][k].y)/2.0)/(array[j+1][k].y-arr
ay[j][k].y);
        band2[counter2]=pi*((array
[j][k].y+(array[j+1][k].y
-array[j][k].y)/2.0)*(array[j][k].y+(array[j+1][k].y-arr
ay[j][k].y)/2.0) -(array[j]
[k].y-(array[j][k].y-array[j-1][k].y)/2.0)*(array[j][k].
y-(array[j][k].y-array[j-1]
[k].y)/2.0))/cell_width;
        band4[counter4]=pi*((array[
j][k].y+(array[j+1][k].y-
array[j][k].y)/2.0)*(array[j][k].y+(array[j+1][k].y-array
[j][k].y)/2.0) -(array[j]
[k].y-(array[j][k].y-array[j-1][k].y)/2.0)*(array[j][k].
y-(array[j][k].y-array[j-1]
[k].y)/2.0))/cell_width;
        band5[counter5]=2.0*pi*cel
l_width*(array[j][k].y-(
array[j][k].y-array[j-1][k].y)/2.0)/(array[j][k].y-array
[j-1][k].y);
    }

    else if(k==0){
        band1[counter1]=0.0;
        band2[counter2]=0.0;
    }
}

```

```

band4[counter4]=0.0;
band5[counter5]=0.0;

}
else {
band1[counter1]=2.0*pi*(
array[j][k].y+(
array[j+1][k].y-array[j][k].y)/2.0)*cell_width/(array[j+
1][k].y-array[j][k].y);
band2[counter2]=0.0;
band4[counter4]=pi*((arra
y[j][k].y+(
array[j+1][k].y-array[j][k].y)/2.0)*(array[j][k].y+(array[
j+1][k].y-array[j][k].y)/
2.0)
-(array[j][k].y-(array[j
][k].y-array[j-1][k].y)/
2.0)*(array[j][k].y-(array[j][k].y-array[j-1][k].y)/2.0))/
cell_width;
band5[counter5]=2.0*pi*( a
rray[j][ngp_horiz-1].y-(
array[j][k].y-array[j-1][k].y)/2.0)*cell_width/(array[j][
k].y-array[j-1][k].y);
}
}
else{
/*band1[counter1]=0.0;*/
band2[counter2]=0.0;
band4[counter4]=0.0;
band5[counter5]=0.0;
}
}
}
/*for(i=0;i<ngp;i++){
printf("%ld %e %e %e %e %e
\n",i,band1[i],band2[i
],band3[i],band4[i],band5[i]);
}*/

/* Diagonal */
for (j=0;j < jmax;j++){
/* y < r_probe */
if(j <= jprobe){
for(k=0;k < kmax;k++){
counter3++;
if(k >= 0 && (k < kprobe)){
if(j==0){
if(k==0){

```

```

band3[coun
ter3]=-1.0;
}
else{
band3[coun
ter3]=-pi*(cell_width+
(array[j+1][k].y-array[j][k].y)*(array[j+1][k].y-array[j][k
].y)/(2.0*cell_width));
}
}
else {
if(k==0){
band3[counte
r3]=-1.0;
}
else{
band3[counter
3]=-(2.0*pi*cell_
width*(array[j][k].y+(array[j+1][k].y-array[j][k].y)/2.0)/(arr
ay[j+1][k].y-array
[j][k].y) + pi*((array[j][k].y+(array[j+1][k].y-array[j][k].
y)/2.0)*(array[j][k].y
+(array[j+1][k].y-array[j][k].y)/2.0) -(array[j][k].y-(array[
j][k].y-array[j-1][k
].y)/2.0)*(array[j][k].y-(array[j][k].y-array[j-1][k].y)/2.0))
/cell_width + pi*((ar
ray[j][k].y+(array[j+1][k].y-array[j][k].y)/2.0)*(array[j][k].
y+(array[j+1][k].y-a
rray[j][k].y)/2.0) -(array[j][k].y-(array[j][k].y-array[j-1][
k].y)/2.0)*(array[j][
k].y-(array[j][k].y-array[j-1][k].y)/2.0))/cell_width + 2.0*pi
*cell_width*(array[j
][k].y-(array[j][k].y-array[j-1][k].y)/2.0)/(array[j][k].y-arr
ay[j-1][k].y));
}
}
}
else{
band3[counter3]=-1.0;
}
}
}
/* y > r_probe */
else if(j<(jmax-1)){
for(k=0;k < kmax;k++){
counter3++;

if(k > 0 && (k < (kmax-1)) && (j
<(jmax-1))){
band3[counter3]=-(2.0*pi*

```

```

cell_width*(array[j][k].y+
(array[j+1][k].y-array[j][k].y)/2.0)/(array[j+1][k].y-arr
ay[j][k].y) + pi*((array[j
][k].y+(array[j+1][k].y-array[j][k].y)/2.0)*(array[j][k].
y+(array[j+1][k].y-array[j
][k].y)/2.0) -(array[j][k].y-(array[j][k].y-array[j-1][k
].y)/2.0)*(array[j][k].y-(a
rray[j][k].y-array[j-1][k].y)/2.0))/cell_width + pi*((ar
ray[j][k].y+(array[j+1][k].
y-array[j][k].y)/2.0)*(array[j][k].y+(array[j+1][k].y-ar
ray[j][k].y)/2.0) -(array[j
][k].y-(array[j][k].y-array[j-1][k].y)/2.0)*(array[j][k]
.y-(array[j][k].y-array[j-1
][k].y)/2.0))/cell_width + 2.0*pi*cell_width*(array[j][k]
.y-(array[j][k].y-array[j-1
][k].y)/2.0)/(array[j][k].y-array[j-1][k].y));

        }
        else if(k==0){
            band3[counter3]=-1.0;
        }
        else{
            band3[counter3]=-(2.0*pi*
cell_width*(array[j][k].y+
(array[j+1][k].y-array[j][k].y)/2.0)/(array[j+1][k].y-arr
ay[j][k].y) + pi*((array[j
][k].y+(array[j+1][k].y-array[j][k].y)/2.0)*(array[j][k].
y+(array[j+1][k].y-array[j
][k].y)/2.0) -(array[j][k].y-(array[j][k].y-array[j-1][k]
.y)/2.0)*(array[j][k].y-(ar
ray[j][k].y-array[j-1][k].y)/2.0))/cell_width + 2.0*pi*ce
ll_width*(array[j][k].y-(ar
ray[j][k].y-array[j-1][k].y)/2.0)/(array[j][k].y-array[j-
1][k].y));
        }

    }
    else{
        for(k=0;k < kmax;k++){
            counter3++;
            band3[counter3]=-1.0;
        }
    }
}
/*for(i=0;i<ngp;i++){
    printf("%ld      %f      %f      %f      %f\n",i,band1[i],b
and2[i],band3[i],band4[i],band5[i]);
}

```

```

}*/
/*      Form band vector to be submitted to the band solver
*/
count=-1;
for(i=0;i<ngp;i++){
    for(j=1;j<=(2*lower_width+upper_width+1);j++){
        count++;
        if(j==lower_width+1)          abd
[count]=band1[i];
        else if(j==(lower_width+upper_width))  abd
[count]=band2[i];
        else if(j==(lower_width+upper_width+1)) abd
[count]=band3[i];
        else if(j==(lower_width+upper_width+2)) abd
[count]=band4[i];
        else if(j==(lower_width+upper_width+2+lower_
width-1))  abd[count]=
band5[i];
        else    abd[count]=0.0;

    }
}

printf("Going into LU factorization...\n");

/*      Call LAPACK subroutine to perform the LU factorization
*/
sgbtrf_(&m1,&n1,&lower_width,&upper_width,abd,&ldab,piv,&info);

if (info==0){
printf("LU decomposition successful...\n");
}
else{
printf("LU decomposition failed...\n");
exit(1);
}

fwrite(abd,sizeof(float),count,abddata);
fclose(abddata);

fwrite(piv,sizeof(int),ngp,abddata2);
fclose(abddata2);
}

```

A.13 make_velocity_table.c

```

#define EXTERN extern
#include "lprobe60.h"

```



```

void make_velocity_table(void)
{

long i,num_div;
FLOAT  accuracy,delta,deltay,v,p,p_exact;

num_div=100000;
deltay=1.0/((FLOAT)num_div);
accuracy=5.0e-6;
v=-1.0;


/*      Electrons      */
for(i=0;i < num_div-1;i++){

    p=deltay * ((FLOAT)i);

    do {

        v=v+1.0;

        /*p_exact=sqrt(2.0 / pi) * pow((m_elec/su
pere) / (boltz *
T * 11588.7),1.5)
        * (sqrt(pi / 2.0) * pow(boltz * T *115
88.7 / (m_elec/
supere),1.5)
        *erf(sqrt((m_elec/supere) / (2.0 * bolt
z * T * 11588.7
)) * v)
        - boltz * T * 11588.7 * v / ((m_elec/sup
ere) * exp((m_
elec/supere) * v * v
        / (2.0 * boltz * T * 11588.7)))));*/

        p_exact=boltz * pow((m_elec) / (boltz * T *
11588.7),1.5)
        * T * 11588.7
        * (-2.0 * boltz * T * 11588.7 + 2.0 * exp(
(m_elec) * v
* v
        / (2.0 * boltz * T * 11588.7)) * boltz * T
        * 11588.7 -
(m_elec) * v * v)
        /(2.0 * exp((m_elec) * v * v / (2.0 * boltz
* T * 11588.
7))) * (m_elec)
        * (m_elec) * sqrt(boltz * T * 11588.7 / (m_elec)));

        delta=fabs(p-p_exact);

    }while(delta >= accuracy);
}

```

```

        v_table_e[i]=v;
    }

    printf("Done creating electron velocity table...\n");

    /*      IONS      */
    v=-0.01;

    for(i=0;i< num_div - 1;i++){

        p=deltay * ((FLOAT)i);

        do {

            v=v + 0.01;

            /*p_exact=sqrt(2.0 / pi) * pow((m_ion ) / (boltz * T
* 11588.7),1.5)
            * (sqrt(pi / 2.0) * pow(boltz * T *11588.7 / (m_i
on ),1.5)
            *erf(sqrt((m_ion ) / (2.0 * boltz * T * 11588.7))
* v)
            - boltz * T * 11588.7 * v / ((m_ion ) * exp((m_ion
) * v * v
            / (2.0 * boltz * T * 11588.7)))));*/

            p_exact=boltz * pow((m_ion) / (boltz * T * 11588.7),
1.5) * T * 11588.7
            * (-2.0 * boltz * T * 11588.7 + 2.0 * exp((m_ion)
* v * v
            / (2.0 * boltz * T * 11588.7)) * boltz * T * 11588
.7 - (m_ion) * v * v)
            /(2.0 * exp((m_ion) * v * v / (2.0 * boltz * T * 1
1588.7)) * (m_ion)
            * (m_ion) * sqrt(boltz * T * 11588.7 / (m_ion)));

            delta=fabs(p-p_exact);

        }while(delta >= accuracy);
        v_table_i[i]=v;
    /*printf("%ld  %lf\n",i,v);*/
    }

    printf("Done creating ion velocity table...\n");
}

```

A.14 mover.c

```
#define EXTERN extern
```

```

#include "lprobe60.h"

/*   This function moves the particles   */

void mover(PARTICLEDEF particle[numpart],long int num_part,
FLOAT timestep)
{
long int i;
FLOAT accelx,accelr,theta;

for(i=0;i < num_part;i++){
    accelx=particle[i].q * particle[i].Ex / particle[i].m;
    accelr=particle[i].q * particle[i].Ey / particle[i].m;
    theta=atan2(particle[i].y,particle[i].z);
    particle[i].vx=particle[i].vx + accelx * timestep;
    particle[i].vy=particle[i].vy + accelr * timestep * sin(theta);
    particle[i].vz=particle[i].vz + accelr * timestep * cos(theta);
    particle[i].x=particle[i].x + particle[i].vx * timestep;
    particle[i].y=particle[i].y + particle[i].vy * timestep;
    particle[i].z=particle[i].z + particle[i].vz * timestep;
    particle[i].r=sqrt(particle[i].y*particle[i].y + particle[i].z*
particle[i].z);
}

}

```

A.15 output_data.c

```

#define EXTERN extern
#include "lprobe60.h"

void output_data(PARTICLEDEF particle1[numpart],PARTICLEDEF particle2
[numpart],long int numpart1,long int numpart2,FLOAT me,FLOAT mi)
{

    FILE *currdata;
    FLOAT nde,ndi;
    long int i2;

    currdata=fopen("current_q100.dat","a");

    avecurrent=current/(500.0*dti);avecurre=curre/(500.0*dti);avecurri=
curri/(500.0*dti);

    /*   COMPUTE NUMBER DENSITY   */

    nde=ndi=0.0;

```

```

    for(i2=0;i2 < numpart1;i2++){
        if(particle1[i2].r <= radius){
            nde=nde + particle1[i2].m / me;
        }
    }
    nde=nde / (pi*radius*radius*length);

    for(i2=0;i2 < numpart2;i2++){
        if(particle2[i2].r <= radius){
            ndi=ndi + particle2[i2].m / mi;
        }
    }
    ndi=ndi / (pi*radius*radius*length);

    if (t_count2 == 1){
        fprintf(currdata,"      V          t      ne      ni      nde      ndi
j          je          ji\n \n");
    }

    fprintf(currdata,"%f      %ld %ld %ld %e %e %e %e %e\n",V,
t_count2,numelec,numion,nde,ndi,avecurre,avecurre,avecurre);
    fclose(currdata);

    t_count=0;
    current=curri=curre=0.0;

}

```

A.16 parameter.c

```

#define EXTERN extern
#include "lprobe60.h"

/*This function sets initial plasma conditions and computes plasma
parameters to be used throughout the rest of the simulation */

void param()
{
    FLOAT avveli,avvele,gammai,gammae;

    FLOAT numiperstep_back,numiperstep_front,
        numiperstep_side,numeperstep_back,numeperstep_front,numeperste
p_side,fluxi_back,fluxi_front,
        fluxi_side,fluxe_back,fluxe_front,fluxe_side,afront,aback,aside;

    /*  NUMBER OF PARTICLES TO INJECT AT EACH TIME STEP      */

    superii_init=40000;
}

```

```

superii_back=2;
superii_front=2;
superii_side=4;
superei_init=40000;
superei_back=5;
superei_front=5;
superei_side=8;

/*      GEOMETRY      */

/*r_probe=1.25e-4;*/
r_probe=1.25e-4 - 3.0*(0.00003125);
/*r_insul=2.0*r_probe;*/
l_probe=0.00287;
l_insul=1.5*l_probe;
/*l_probe=0.00287 + 0.5 * 0.00287;
l_insul=1.0*l_probe;*/
radius=0.00395705;
length=4.0*l_probe;
height=width=2.0 * radius;
ioniter=70;
ioncount=0;
ion_move_count=100;
ecounter=0;
view=1;

T=2.0;
n=1.0e15;
m_ion=1.673e-27;
q_ion=1.6e-19;
m_elec=9.11e-31;
q_elec=-1.6e-19;
wpdt=0.009;
plasma_pot=0.0;

h=6.9*sqrt((T*11588.7)/(n*1.0e-6)) * 0.01;
cell_height1=0.00003125;
cell_height2=0.00003713;
cell_height3=0.00004301;
cell_height4=0.00004889;
cell_height5=0.00005477;
cell_height6=0.00006065;
cell_width=0.0000574;

r_insul=r_probe + cell_height1;

wp=2.0*pi*9000.0*sqrt(n*1.0e-6);
elec_iter=15;
dt=wpdt/wp;dte=dt;dti=15.0*dte;
t_count=0;

```

```

sigmae=sqrt((boltz*T*11588.7)/m_elec);
sigmai=sqrt((boltz*T*11588.7)/m_ion);

/* Boundary Fluxing Setup */

afront=4.0* radius * radius;aside=2.0 * radius * length;aback=
4.0 * radius * radius - pi*r_insul*r_insul;

avveli=sqrt((8.0*boltz*T*11589.7)/(pi*m_ion));
avvele=sqrt((8.0*boltz*T*11589.7)/(pi*m_elec));
gammai=n*avveli/4.0;
gammae=n*avvele/4.0;

fluxi_back=gammai*aback;
fluxe_back=gammae*aback;
fluxi_front=gammai*afront;
fluxe_front=gammae*afront;
fluxi_side=gammai*aside;
fluxe_side=gammae*aside;

numiperstep_back=fluxi_back*dti;
numeperstep_back=fluxe_back*dte;
numiperstep_front=fluxi_front*dti;
numeperstep_front=fluxe_front*dte;
numiperstep_side=fluxi_side*dti;
numeperstep_side=fluxe_side*dte;

superi_back=numiperstep_back/((FLOAT)superii_back);
supere_back=numeperstep_back/((FLOAT)superei_back);
superi_front=numiperstep_front/((FLOAT)superii_front);
supere_front=numeperstep_front/((FLOAT)superei_front);
superi_side=numiperstep_side/((FLOAT)superii_side);
supere_side=numeperstep_side/((FLOAT)superei_side);

/* Initial Loading Injection Setup */

VOLUME=length*height*width - (pi* r_probe * r_probe * l_probe) -
(pi* r_insul * r_insul * l_insul);
superi_init=n * ((pi* radius * radius * length)-
(pi* r_probe * r_probe * l_probe) - (pi* r_insul * r_insul
* l_insul))
/ ((FLOAT)superii_init);
supere_init=n * ((pi* radius * radius * length)-
(pi* r_probe * r_probe * l_probe) - (pi* r_insul * r_insul
* l_insul))
/ ((FLOAT)superei_init);
injsuperi=superii_init;
injsupere=superei_init;

```

```

/*      Parameteters needed for field solver      */
jprobe=((long int)(r_probe/cell_height1));
kprobe=((long int)((length-l_probe-l_insul)/cell_width));
}

```

A.17 probepic.c

```

#define EXTERN
#include "lprobe60.h"
#include<stdlib.h>

/*      This is the main program for a two dimensional PIC
simulation of a Langmuir probe.      */

void main(void)
{
/*FILE *efdata;*/
long int seed,i,i3,iter,i4,i5,i6;

/*  FIELD SOLVER GLOBAL VARIABLES  */
test1=0,test2=0,test3=0,ldab=3*ngp2+1,n1=ngp,m1=ngp,ldb=
ngp,upper_width=ngp2,lower_width=ngp2;
x=521288629,y=362436069,z=16163801,c=1,n2=1131199299;

electron=(PARTICLEDEF *)malloc(numpart * sizeof(PARTICLEDEF));
ion=(PARTICLEDEF *)malloc(numpart * sizeof(PARTICLEDEF));
abd=(FLOAT *)malloc((2*lower_width+upper_width+2)*ngp*size
of(FLOAT));
b=(FLOAT *)malloc(ngp*sizeof(FLOAT));
phi=(FLOAT *)malloc(ngp*sizeof(FLOAT));
v_table_e=(FLOAT *)malloc(100001*sizeof(FLOAT));
v_table_i=(FLOAT *)malloc(100001*sizeof(FLOAT));
random_num=(FLOAT *)malloc((NPTS+1)*sizeof(FLOAT));
piv=(int *)malloc(ngp*sizeof(int));

/*efdata=fopen("efdat30.dat","w+");*/

numelec=numion=t_count=t_count2=0;
V=3.5;

/*prefposition(20,5);
prefsize(900,700);
vinit("X11");
mapcolor(50,100,100,100);
mapcolor(56,175,175,175);
mapcolor(51,139,69,0);
mapcolor(52,20,20,200);

```

```

mapcolor(53,255,0,0);
mapcolor(55,238,201,0);*/

/*  INITIALIZE ARRAY OF UNIFORM RANDOM NUMBERS      */
seed=100;
random_number(seed);
rancount=0;

param();

gridgen(grid);

make_velocity_table();

initialize(electron,&numelec,superei_init,supere_init,
v_table_e,m_elec,q_elec);

seed=101;
random_number(seed);
rancount=0;

initialize(ion,&numion,superii_init,superi_init,v_table_i,
m_ion,q_ion);

printf("initial number electrons ==>%ld    initial number
ions ==>%ld\n",numelec,numion);

make_LU(grid);

printf("Entering main program...\n");

iter=12001;
for (i3 = 0;i3 < 18;i3++){
    V=V+1.5;
    if (i3 > 0)
        iter=3501;
    for (i = 0;i < iter;i++){

        if (rancount > (NPTS - 500)){
            seed=seed + 100;
            random_number(seed);
            rancount=0;
        }

        for(i6=0;i6 < 15;i6++){

            reset_grid(grid);

            qweight(grid,electron,numelec);

```



```

        qweight(grid,ion,numion);

        e_field(grid);

        fweight(grid,electron,numelec);

        inject(electron,&numelec,supere_back,supere_
front,supere_side,
                superei_back,superei_front,superei_
side,v_table_e,m_elec,q_elec);

        mover(electron,numelec,dte);

        boundary(electron,&numelec,q_elec);
    }

    fweight(grid,ion,numion);

    inject(ion,&numion,superi_back,superi_front,super
i_side,
            superii_back,superii_front,superii_
side,v_table_i,m_ion,q_ion);

    mover(ion,numion,dti);

    boundary(ion,&numion,q_ion);

    t_count++;
    t_count2++;
    if (t_count == 500 || t_count2==1){
        output_data(electron,ion,numelec,numion,
m_elec,m_ion);
    }
}
}
/*fclose(efdata);*/
/*vexit();*/
exit(1);

}

```

```

#undef IA
#undef IM
#undef AM
#undef IQ
#undef IR
#undef NTAB
#undef NDIV
#undef EPS

```

```
#undef RNMX
#undef NRANSI
```

A.18 probepic.h

```
#include <stdio.h>
#include <math.h>
/**include<sgidefs.h>*/

#define FLOAT float

#define xButtonOffset 0.15
#define yButtonOffset 0.08

#define SWITCHBOXx -0.07
#define SWITCHBOXy -.85
#define EXITBOXx 0.84
#define EXITBOXy SWITCHBOXy
#define PRINTBOXx .65
#define PRINTBOXy SWITCHBOXy

#define EPS0 8.85e-12
#define EPSA103 10.44e-12
#define pi 3.14159265359
#define ngp 14874
#define ngp1 74
#define ngp2 201
#define numpart 80000
#define boltz 1.38e-23
#define RAD 0.00378864
#define IA 16807
#define IM 2147483647
#define AM (1.0/IM)
#define IQ 127773
#define IR 2836
#define NTAB 32
#define NDIV (1+(IM-1)/NTAB)
#define EPS 1.2e-7
#define RNMX (1.0-EPS)
#define NRANSI
#define NR_END 1
#define FREE_ARG char*
#define NPTS 600005

/* Environment Variables */
#define black 0
#define white 15
#define gray 50
```

```

#define gray2 56
#define lightgray 56
#define darkorange 51
#define midblue 52
#define red 53
#define gold 55

/* Global Variables */

typedef struct {
    FLOAT x,y,q_dens
    ,Ey,Ex,phi;
}GRIDDEF;

typedef struct {
    FLOAT x,y,z,r,vx,vy,vz,Ex,Ey,q,m;
    unsigned long cellx,celly;
}PARTICLEDEF;

typedef unsigned long int unlong;

typedef struct
{
    FLOAT x1PositionUp;
    FLOAT x2PositionUp;
    FLOAT y1PositionUp;
    FLOAT y2PositionUp;
    char ButtonText[20];
    short TextColor;
    short aBorderColorUp;
    short bBorderColorUp;
    short cBorderColorUp;
    short dBorderColorUp;
    short aBorderColorDown;
    short bBorderColorDown;    short cBorderColorDown;
    short dBorderColorDown;
    short FillColorUp;
    short FillColorDown;
} PushButton;

/* GLOBAL VARIABLES */

/* PARTICLE GLOBAL VARIABLES */

EXTERN PARTICLEDEF *ion,*electron;

/* GRID/GEOMETRY GLOBAL VARIABLES */
EXTERN long int ngp_vert,ngp_horiz,jprobe,kprobe,jinsul,kinsul;
EXTERN FLOAT cell_height1,cell_height2,cell_height3,cell_height4,

```

```

cell_height5,cell_height6,cell_height,cell_width,r_probe,r_insul,
l_probe,l_insul,radius,length;
EXTERN GRIDDEF grid[ngp1][ngp2];

/*      FIELD SOLVER GLOBAL VARIABLES      */
EXTERN int test1,test2,test3,ldab,*piv,n1,m1,ldb,upper_width,lower_
width,info,nrhs;
EXTERN long int jprobe,kprobe,kinsul;
EXTERN FLOAT band1[ngp],band2[ngp],band3[ngp],band4[ngp],band5[ngp],
*abd,*b,*phi;

/*      PLASMA PARAMETER GLOBAL VARIABLES      */
EXTERN FLOAT T,n,m_ion,q_ion,m_elec,q_elec,m_part,plasma_pot;

/*      INJECTION SETUP GLOBAL VARIABLES      */
EXTERN long int injsuperi,injsupere,superii_init,superii_back,superii_
_front,superii_side,superei_init,
                superei_back,superei_front,superei_side,rancount,elec
_iter;
EXTERN FLOAT supere,superi,sigma,sigmae,sigmai,wpdt,h,wp,dt,superi_in
it,superi_back,superi_front,superi_side,supere_init,supere_back,supere
_front,supere_side,*v_table_e,*v_table_i,height,width,*random_num,dti,dte;

/*      COUNTING GLOBAL VARIABLES      */
EXTERN long int tom,numelec,numion,t_count,t_count2,num,ioncount,ioniter
,ion_move_count,ecounter;

/*      GRAPHICS GLOBAL VARIABLES      */
EXTERN long int view,page;

/*      BOUNDARY GLOBAL VARIABLES      */
EXTERN FLOAT V,current,VOLUME,avecurent,curri,curre,avecurri,avecurre
,ins_charge;

/*      TESTING VARIABLES      */
EXTERN long int t1,t2;
EXTERN FLOAT *vxdata;

EXTERN ulong x,y,z,c,n2;

/*      Prototypes      */
void random_number(long int seed);
ulong msmith_ran();
void ReadInBigArrays( void );
void gridgen(GRIDDEF array[ngp1][ngp2]);
void param();
void initialize2(PARTICLEDEF particle[numpart],PARTICLEDEF particle2[n
umpart]);
void initialize(PARTICLEDEF particle[numpart],long int *num_part,long
int init_num,FLOAT super_init,FLOAT *v_table,FLOAT m,FLOAT q);

```

```

void make_velocity_table(void);
void injecte(PARTICLEDEF particle[numpart]);
void injecti(PARTICLEDEF particle[numpart]);
void inject(PARTICLEDEF particle[numpart],long int *num_part,FLOAT super_back,FLOAT super_front,
           FLOAT super_side,long int superinj_back,long int superinj_front,
           long int superinj_side,FLOAT *v_table,FLOAT m,FLOAT q);
void qweight(GRIDDEF array[ngp1][ngp2],PARTICLEDEF particle[numpart],long int number);
void make_LU(GRIDDEF array[ngp1][ngp2]);
void e_field(GRIDDEF array[ngp1][ngp2]);
void fweight(GRIDDEF array[ngp1][ngp2],PARTICLEDEF particle[numpart],long int number);
void boundary (PARTICLEDEF particle[numpart],long int *num_part,FLOAT q);
/*void mover(PARTICLEDEF array[numpart],long int num_part);*/
void mover(PARTICLEDEF array[numpart],long int num_part,FLOAT timestep);
void graph(GRIDDEF array[ngp1][ngp2],PARTICLEDEF particle1[numpart],PARTICLEDEF particle2[numpart]);
FLOAT gasdev(long int *idum);
FLOAT ran1(long int *idum);
void output_data(PARTICLEDEF particle1[numpart],PARTICLEDEF particle2[numpart],long int numpart1,long int numpart2,FLOAT me,FLOAT mi);
void locatex(GRIDDEF xx[ngp1][ngp2], unsigned long int n, FLOAT x, unsigned long int *jj);
void locatey(GRIDDEF xx[ngp1][ngp2], unsigned long int n, FLOAT x, unsigned long int *jj);
void reset_grid(GRIDDEF array[ngp1][ngp2]);
void recombination (PARTICLEDEF heavy_part[numpart],long int *num_heavy,PARTICLEDEF elec[numpart],long int *num_part);

/*FLOAT *vector(long int nl, long int nh);
void nrerror(char error_text[]);
void free_vector(FLOAT *v, long int nl, long int nh); */

short CheckButton ( PushButton TomButton );
short MakeButton ( FLOAT x1, FLOAT y1, PushButton *TomButton, char ButtonText[20] );
void ClearMouseBuffer();
void DrawButton(PushButton TomButton);
void DrawButtonDown(PushButton TomButton);
void main(void);

```

A.19 random_number.c

```

#define EXTERN extern
#include "lprobe60.h"

/* Common Block Declarations */

```

```

struct klotz0_1_ {
    float buff[607];
    long int ptr;
};

#define klotz0_1 (*(struct klotz0_1_ *) &klotz0_)
#define min(a,b) (a<b)?a:b

struct klotz1_1_ {
    FLOAT xbuff[1024];
    long int first, xptr;
};

#define klotz1_1 (*(struct klotz1_1_ *) &klotz1_)

/* Initialized data */

struct {
    long int fill_1[1214];
    long int e_2;
} klotz0_ = { {0}, 0 };

struct {
    FLOAT fill_1[1024];
    long int e_2[2];
    FLOAT e_3;
} klotz1_ = { {0}, 0, 0, 0. };

void random_number(long int seed)
{
    /*long int tom;*/
    extern long int fischet_(), zufalli_(), normalt_(), zufallt_();

    zufalli_(seed);
    zufall_(NPTS,random_num);

    /*for(tom=0;tom<100;tom++) prlong intf("random_number[%d]=%f\n",tom,ra
ndom_num[tom]);*/
}

int zufall_(n, a)
long int n;
FLOAT *a;
{
    long int buffsz = 607;

```

```

    long int left, aptr, bptr, aptr0, i, k, q;
    FLOAT t;
    long int nn, vl, qq, k273, k607, kptr;

/* portable lagged Fibonacci series uniform random number */
/* generator with "lags" -273 und -607: */
/* W.P. Petersen, IPS, ETH Zuerich, 19 Mar. 92 */

    aptr = 0;
    nn = n;

L1:

    if (nn <= 0) {
        return 0;
    }

/* factor nn = q*607 + r */

    q = (nn - 1) / 607;
    left = buffsz - klotz0_1.ptr;

    if (q <= 1) {

/* only one or fewer full segments */

        if (nn < left) {
            kptr = klotz0_1.ptr;
            for (i = 0; i < nn; ++i) {
                a[i + aptr] = klotz0_1.buff[kptr + i];
            }
            klotz0_1.ptr += nn;
            return 0;
        } else {
            kptr = klotz0_1.ptr;
/*pragma _CRI ivdep*/
            for (i = 0; i < left; ++i) {
                a[i + aptr] = klotz0_1.buff[kptr + i];
            }
            klotz0_1.ptr = 0;
            aptr += left;
            nn -= left;
/* buff -> buff case */
            vl = 273;
            k273 = 334;
            k607 = 0;
            for (k = 0; k < 3; ++k) {
/*pragma _CRI ivdep*/
                for (i = 0; i < vl; ++i) {

```

```

        t = klotz0_1.buff[k273+i]+klotz0_1.buff[k607+i];
        klotz0_1.buff[k607+i] = t - (FLOAT) ((long int) t);
    }
    k607 += v1;
    k273 += v1;
    v1 = 167;
    if (k == 0) {
        k273 = 0;
    }
}
goto L1;
}
} else {

/* more than 1 full segment */

    kptr = klotz0_1.ptr;
/*pragma _CRI ivdep*/
    for (i = 0; i < left; ++i) {
        a[i + aptr] = klotz0_1.buff[kptr + i];
    }
    nn -= left;
    klotz0_1.ptr = 0;
    aptr += left;

/* buff -> a(aptr0) */

    v1 = 273;
    k273 = 334;
    k607 = 0;
    for (k = 0; k < 3; ++k) {
        if (k == 0) {
/*pragma _CRI ivdep*/
            for (i = 0; i < v1; ++i) {
                t = klotz0_1.buff[k273+i]+klotz0_1.buff[k607+i];
                a[aptr + i] = t - (FLOAT) ((long int) t);
            }
            k273 = aptr;
            k607 += v1;
            aptr += v1;
            v1 = 167;
        } else {
/*pragma _CRI ivdep*/
            for (i = 0; i < v1; ++i) {
                t = a[k273 + i] + klotz0_1.buff[k607 + i];
                a[aptr + i] = t - (FLOAT) ((long int) t);
            }
            k607 += v1;
            k273 += v1;
            aptr += v1;
        }
    }
}

```



```

    }
    nn += -607;

/* a(aptr-607) -> a(aptr) for last of the q-1 segments */

    aptr0 = aptr - 607;
    v1 = 607;

    for (qq = 0; qq < q-2; ++qq) {
        k273 = aptr0 + 334;
/*pragma _CRI ivdep*/
        for (i = 0; i < v1; ++i) {
            t = a[k273 + i] + a[aptr0 + i];
            a[aptr + i] = t - (FLOAT) ((long int) t);
        }
        nn += -607;
        aptr += v1;
        aptr0 += v1;
    }

/* a(aptr0) -> buff, last segment before residual */

    v1 = 273;
    k273 = aptr0 + 334;
    k607 = aptr0;
    bptr = 0;
    for (k = 0; k < 3; ++k) {
        if (k == 0) {
/*pragma _CRI ivdep*/
            for (i = 0; i < v1; ++i) {
                t = a[k273 + i] + a[k607 + i];
                klotz0_1.buff[bptr + i] = t - (FLOAT) ((long int) t);
            }
            k273 = 0;
            k607 += v1;
            bptr += v1;
            v1 = 167;
        } else {
/*pragma _CRI ivdep*/
            for (i = 0; i < v1; ++i) {
                t = klotz0_1.buff[k273 + i] + a[k607 + i];
                klotz0_1.buff[bptr + i] = t - (FLOAT) ((long int) t);
            }
            k607 += v1;
            k273 += v1;
            bptr += v1;
        }
    }
    goto L1;
}
} /* zufall_ */

```

```

long int zufalli_(seed)
long int seed;
{
    /* Initialized data */

    long int kl = 9373;
    long int ij = 1802;

    /* Local variables */
    long int i, j, k, l, m;
    FLOAT s, t;
    long int ii, jj;

    /* generates initial seed buffer by linear congruential */
    /* method. Taken from Marsaglia, FSU report FSU-SCRI-87-50 */
    /* variable seed should be 0 < seed <31328 */

    if (seed != 0) {
        ij = seed;
    }

    i = ij / 177 % 177 + 2;
    j = ij % 177 + 2;
    k = kl / 169 % 178 + 1;
    l = kl % 169;
    for (ii = 0; ii < 607; ++ii) {
        s = 0.;
        t = .5;
        for (jj = 1; jj <= 24; ++jj) {
            m = i * j % 179 * k % 179;
            i = j;
            j = k;
            k = m;
            l = (l * 53 + 1) % 169;
            if (l * m % 64 >= 32) {
                s += t;
            }
            t *= (FLOAT).5;
        }
        klotz0_1.buff[ii] = s;
    }
    return 0;
} /* zufalli_ */

long int zufallsv_(svblk)
FLOAT *svblk;

```

```

{
    long int i;

    /* saves common blocks klotz0, containing seeds and */
    /* polong inter to position in seed block. IMPORTANT: svblk must be */
    /* dimensioned at least 608 in driver. The entire contents */
    /* of klotz0 (polong inter in buff, and buff) must be saved. */

    /* Function Body */
    svblk[0] = (FLOAT) klotz0_1.ptr;
    /*pragma _CRI ivdep*/
    for (i = 0; i < 607; ++i) {
        svblk[i + 1] = klotz0_1.buff[i];
    }

    return 0;
} /* zufallsv_ */

long int zufallrs_(svblk)
FLOAT *svblk;
{
    long int i;

    /* restores common block klotz0, containing seeds and pointer */
    /* to position in seed block. IMPORTANT: svblk must be */
    /* dimensioned at least 608 in driver. The entire contents */
    /* of klotz0 must be restored. */

    klotz0_1.ptr = (long int) svblk[0];
    /*pragma _CRI ivdep*/
    for (i = 0; i < 607; ++i) {
        klotz0_1.buff[i] = svblk[i + 1];
    }

    return 0;
} /* zufallrs_ */

```

A.20 reset_grid.c

```

#define EXTERN extern
#include "lprobe60.h"

void reset_grid(GRIDDEF array[ngp1][ngp2])
{
    long int count1, count2;

```

```
/*      CLEAN UP GRID      */

for (count1=0;count1 < ngp_vert;count1++){
  for (count2=0;count2 < ngp_horiz;count2++){
    array[count1][count2].q_dens=0.0;
  }
}
}
```

Appendix B

Sample PROBEPIC Output

v	t	ne	ni	nde	ndi	j	je	ji
-2.000000	1	50122	89975	1.034675e+15	1.035351e+15	4.434605e-08	-6.485609e-07	6.929070e-07
-2.000000	1001	60742	89679	1.027824e+15	1.036154e+15	-1.253903e-06	-1.586498e-06	3.325954e-07
-2.000000	2001	60595	89475	1.026945e+15	1.035040e+15	9.146372e-07	0.000000e+00	9.146372e-07
-2.000000	3001	60929	89042	1.028149e+15	1.035303e+15	1.448702e-06	0.000000e+00	1.448702e-06
-2.000000	4001	60881	88407	1.027870e+15	1.037666e+15	2.413954e-06	-5.488053e-08	2.468835e-06
-2.000000	5001	60934	87555	1.031526e+15	1.038886e+15	3.422525e-06	-5.488053e-08	3.477406e-06
-2.000000	6001	60983	86618	1.032126e+15	1.039155e+15	3.741312e-06	0.000000e+00	3.741312e-06
-2.000000	7001	60821	85669	1.029668e+15	1.037647e+15	3.729057e-06	-5.488053e-08	3.783939e-06
-2.000000	8001	60802	84940	1.026566e+15	1.036567e+15	3.920064e-06	-5.488053e-08	3.974945e-06
-2.000000	9001	60837	85168	1.028196e+15	1.036306e+15	3.682870e-06	0.000000e+00	3.682870e-06
-2.000000	10001	60846	85056	1.026780e+15	1.036053e+15	4.335474e-06	0.000000e+00	4.335474e-06
-1.500000	11001	60838	85051	1.028687e+15	1.036256e+15	3.622842e-06	-3.292832e-07	3.952126e-06
-1.500000	12001	61063	85174	1.031903e+15	1.035799e+15	3.703354e-06	-3.172273e-07	4.020582e-06
-1.500000	13001	61030	85522	1.028492e+15	1.036193e+15	3.303270e-06	-3.841637e-07	3.687434e-06
-1.500000	14001	61028	85535	1.030389e+15	1.036176e+15	3.751090e-06	-1.097611e-07	3.860853e-06
-1.500000	15001	61119	85329	1.031474e+15	1.036185e+15	3.792049e-06	-1.646416e-07	3.956690e-06
-1.500000	16001	61110	85497	1.030035e+15	1.036208e+15	3.650458e-06	-2.195221e-07	3.869980e-06
-1.500000	17001	61030	85328	1.030514e+15	1.036523e+15	3.901344e-06	-2.744027e-07	4.175746e-06
-1.000000	18001	61264	85323	1.033299e+15	1.036517e+15	1.924923e-06	-1.799002e-06	3.723944e-06
-1.000000	19001	61057	85441	1.033367e+15	1.036220e+15	2.753990e-06	-1.152491e-06	3.906489e-06

VITA

Thomas E. Markusic was born and raised in rural northeastern Ohio. He spent his early years in Mantua, enjoying the benefits of country living – exploring nature, harassing various farm animals, and appreciating the freedom from worry that comes with separation from the bustle.

Tom was educated public secondary schools. In high school, he was active in wrestling and football. At the encouragement of his girlfriend, Christa J. English (now Christa E. Markusic, his wife), Tom entered engineering school, obtaining a B.S. in Aeronautical and Astronautical Engineering from The Ohio State University in 1993. After receiving a NASA Space Grant Fellowship, he enrolled in the Master's program at The University of Tennessee Space Institute. In the fall of 1996 he completed the degree requirements for Master's degrees in both Physics and Aerospace Engineering. Also, in March of this year, his first child was born, Elena Maria Markusic.

Early in 1996 Tom was awarded an Air Force Palace Knight Fellowship. He intends to use this fellowship to pursue a Ph.D. in Mechanical Engineering at Stanford University, specializing in electric propulsion.