Simulation of MPD Flows Using a Flux-Limited Numerical Method for the MHD Equations

Kameshwaran Sankaran

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF MASTER OF SCIENCE IN ENGINEERING

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
MECHANICAL AND AEROSPACE ENGINEERING

January, 2001

Simulation of MPD Flows Using a Flux-Limited Numerical Method for the MHD Equations

Prepared by:

Kameshwaran Sankaran

Approved by:

Acknowledgments

My experience, so far, as a graduate researcher, can be best described by the words of Sherlock Holmes: *"They say that genius is an infinite capacity for taking pains. It's a very bad definition, but it does apply to detective work."* Yet, in many ways it was also a very rewarding experience, and many people have played a role in helping it to be so.

I chose to attend graduate school in Princeton largely due to a tradition of excellence in electric propulsion research, built by Prof. Robert Jahn. After I came here, Prof. Jahn taught me the fundamentals of the electric propulsion, and was always available to talk to me about "the big picture" of what research should be.

Prof. Luigi Martinelli taught me the fundamentals of numerical methods and helped convince me that a new route to simulating plasma flows has to be taken.

Prof. Steve Jardin taught me everything I know about computational plasma physics. He was around whenever I hit a wall (which I did frequently) and patiently guided me through the rough times.

The most import role was that of my advisor Prof. Edgar Choueiri. I owe him a lot of thanks for giving me the opportunity to work on this problem, and persisting with me. He was as demanding in the lab as he was in the squash courts, and on both fronts I've only gotten better as a result of interacting with him.

However, the best part of my grad school experience was working with the fellow graduate students. Invariably, they were my companions through the long hours in E-Quad. My discussions with them not only helped me pass the General Exam, but also contributed to much of my learning during this period. And of course, playing softball and squash with them was helpful in maintaining some sanity.

Before this starts to resemble a speech at the Academy Awards, I'll wrap it up with this: a friend told me a long time ago, *"When you get a Bachelor's degree, you'll feel like you know everything; Then you go on for a Master's degree and at the end of which you'll feel like you know nothing; Then you go on for a Doctorate and ..."*. I'll complete the rest of the prophecy after I get a chance to verify the last part.

This dissertation carries the designation 3074-T in the records of the Department of Mechanical and Aerospace Engineering.

Abstract

A new numerical scheme to accurately compute plasma flows of interest to propulsion was developed and validated first against standard test problems. The scheme treats the flow and the field in a self-consistent manner, and conserves mass, momentum, magnetic flux and energy. The characteristics-splitting scheme, which was developed from concepts used for the solution of Euler equations, was applied to solve the ideal MHD equations. The ability of this scheme to capture discontinuities monotonically was demonstrated. Flux-limited anti-diffusion was used to improve spatial accuracy away from discontinuities. Further improvements to the physical model, such as the inclusion of relevant classical transport properties, a real equation of state, multi-level equilibrium ionization models, anomalous transport, and multi-temperature effects, that are essential for the realistic simulation magnetoplasmadynamic flows, are implemented without affecting the underlying scheme. The solver, including the improved physical model, is then used to simulate plasma flow in a real magnetoplasmadynamic thruster configuration. The results of the simulation were found to to be in general agreement with experimental observations. They illustrate the need for using a real equation of state, instead of the ideal one, to enhance the realism and stability of the simulation. A multi-level equilibrium ionization model was found to give satisfactory results for species densities. The predicted value of thrust was found to be in excellent agreement with analytical predictions, though the predicted efficiency was significantly off, due to the lack of an electrode drop model.

Nomenclature

a	Sonic speed
$\mathbf{A}$	Jacobian of the hyperbolic system
$B_{r,\theta,z}$	Radial, azimuthal and axial magnetic induction
$\mathbf{B}$	Magnetic induction vector
$\bar{\bar{\mathcal{B}}}_M$	Maxwell stress tensor
$C_{A,F,S}$	Alfven, fast magnetosonic and slow magnetosonic velocity
$\mathbf{Dr}, \mathbf{Dz}$	Numerical dissipation in the radial and axial flux directions
e	Charge of an electron
E	Electric field strength
E'	Effective electric field, as seen by the plasma
$\mathcal{E}$	Energy density of the plasma
$\mathcal{E}_g$	Gasdynamic energy density
$\mathcal{E}_{e,i}$	Internal energy density of electrons and heavy species
$\mathcal{F}_{conv}$	Convective flux tensor
$\mathcal{F}_{diff}$	Dissipative flux tensor
$\mathbf{Hr}, \mathbf{Hz}$	Radial and axial flux across the cell face
$\mathbf{j}$	Current density
j, k	Cell indices
J_{max}	Total current
k_B	Boltzmann's constant
$\bar{\bar{k}}_{th}$	Thermal conductivity tensor
$\mathbf{Lr}, \mathbf{Lz}$	Flux limiters in the radial and axial directions
m_e	Mass of an electron
$\dot{m}$	Propellant mass flow rate
M_{tot}	Initial spacecraft mass
M_{prop}	Propellant mass
M_i	Atomic mass of propellant
$\dot{n}_e$	Net ionization/recombination rate
n_e	Electron number density
$\mathbf{n}$	Unit normal
p	Gasdynamic pressure
$p_{e,i}$	Electron and heavy species pressure
$\bar{\bar{p}}$	Isotropic pressure tensor
P	Total pressure (magnetic pressure + gasdynamic pressure)
$\mathbf{q}$	Energy source/sinks
$r_{a,c}$	Anode and cathode radius
$\mathbf{R}, \mathbf{R}^{-1}$	Matrix of eigenvectors, and its inverse

R_m	Magnetic Reynolds' number
T	Thrust
T_e	Electron temperature
T_h	Heavy species temperature
$\mathbf{u}$	Velocity vector
$\mathbf{U}$	Vector of conserved variables, ρ, $\rho\mathbf{u}$, $\mathbf{B}$, and $\mathcal{E}$
$\mathbf{U}^n_{J,K}$	Value of $\mathbf{U}$ at time $n \cdot \Delta t$ at the point $(J \cdot \Delta r, K \cdot \Delta z)$
u_{de}	Electron drift velocity
u_e	Exhaust velocity
V_D	E $\times$ B drift velocity
v_{ti}	Ion thermal velocity
Z_{eff}	Average ionization fraction at a spatial location
γ	Ratio of specific heats
$\Delta r, \Delta z$	Cell-size in the radial and axial direction
$\Delta\mathbf{V}$	Characteristic increment in spacecraft velocity
ϵ_o	permitivity of free space
η_0	Classical resistivity
η_{th}	Thrust efficiency
$\bar{\bar{\eta}}$	Anisotropic resistivity tensor
λ	Eigenvalue
λ_D	Debye length
λ_{mfp}	Mean free path between collisions
$\mathbf{\Lambda}$	Diagonal matrix of eigenvalues
μ_o	Magnetic permeability of free space
μ_{vis}	Coefficient of viscous dissipation
ν_{ei}	Energy averaged momentum transfer collision frequency between electrons and ions
ν_{AN}	Anomalous collision frequency
ρ	Average density of the fluid
ρ_e	Unbalanced charge density
τ_{res}	Residence time scale
τ_{equi}	Time scale for energy equilibration between electrons and ions
$\bar{\bar{\tau}}_{vis}$	Viscous stress tensor
ω_{pe}	Electron plasma frequency
$\Omega_{e,i}$	Electron, ion Hall parameter (ratio of gyro and collisional frequncies)
Ω_j	Control volume in a mesh (cell)
$\mid \Omega_j \mid$	Volume of the cell Ω_j

Contents

Chapter 1

INTRODUCTION

The Earth is the cradle of the mind, but we cannot live forever in a cradle.

Konstantin E. Tsiolkovsky

1.1 Role of Plasma Propulsion

Arguably the greatest technological achievement of the twentieth century was NASA's Apollo mission that successfully put men on the surface of the moon and brought them safely back to earth. However, only about 2% of the total mass of about 2750 metric tonnes of the rocket was useful payload. The bulk of the remaining mass was filled by the 960 thousand gallons of fuel needed for propulsion. The reason for the large propellant fraction is clear from the *rocket equation,* derived by Tsiolkovsky[1] in 1903. In the absence of external forces, the ratio of mass of propellant to the total mass of the rocket is given by the relation,

$$\frac{m_{prop}}{m_{tot}} = 1 - e^{-\Delta V/u_e} \ , \tag{1.1}$$

where u_e is the exhaust velocity of the propellant and ΔV is the characteristic velocity increment imparted to the rocket.

Since the exhaust velocities of the chemical rockets used in the *Saturn V* were small (the F-1 engines used in the first stage had exhaust velocities ranging from 2600 to 3000 m/s from sea level to high altitude, and the J-2 engines used in the second and third stage had an exhaust velocity of 4200 m/s at high altitude) compared to the ΔV requirements for the mission, the mass of propellant required was enormous. Despite the advances in combustion research, the highest exhaust velocity of a functional chemical propulsion system, 3600 to 4500 m/s from sea level to high altitude (of the Space Shuttle Main Engine), is still inadequate for most deep-space missions of interest[2]. For missions beyond the moon, the ΔV requirements indicate that chemical propulsion is not a practical solution, and a more viable alternative has to be used.

Functionally, the inability of chemical propulsion systems to achieve higher exhaust velocities is due to the limitation in the maximum tolerable temperature in the combustion chamber, to avoid excessive heat transfer to the walls. Fundamentally, there is also an intrinsic limitation on the maximum energy that is available from the chemical reactions.

Both these limitations can be overcome by the use of electric propulsion (cf. Fig.(1.1)), a working definition of which is found in ref.[2],

> *The acceleration of gases for propulsion by electrical heating and/or by electric and magnetic body forces.*

Two distinct means to harness electrical power to accelerate propellants can be identified:

1. Heating the propellant locally, such that average temperatures are higher than those that can be tolerated by the walls,
2. Acceleration of the propellant by the application of body forces.

The first method can be explained by considering that the electrical power deposited per unit volume of the plasma is,

$$\mathbf{j} \cdot \mathbf{E} = \left\{ \eta j^2 \right\} + \left\{ (\mathbf{j} \times \mathbf{B}) \cdot \mathbf{u} \right\} \ . \tag{1.2}$$

By maximizing the first term, the Ohmic heating, the electrical power can be used to increase the enthalpy of the propellant in a localized fashion, thus avoiding excessive temperatures near the walls. This allows for the average chamber temperature to be higher than those attainable in chemical propulsion systems. The enthalpy can be recovered and converted into directed kinetic energy using a nozzle, as in a chemical rocket. This is the acceleration mechanism in *electrothermal thrusters* such as *arcjets, resistojets* and *microwave-heated thrusters*.

The second acceleration method relies on bypassing thermal expansion altogether, via application of direct body forces. This can be achieved by forces exerted by electrical and magnetic fields on an ionized gas:

$$\mathbf{f}_{ext} = \left\{ \rho_e \mathbf{E} \right\} + \left\{ \mathbf{j} \times \mathbf{B} \right\} \ . \tag{1.3}$$

From eqn.(1.3) two distinct means of application of body forces can be identified. The first term is the body force due to an external electric field. This is the acceleration mechanism in *electrostatic thrusters* such as *ion thrusters* and *field emission thrusters*.

For a highly conducting, quasineutral working fluid, the first term is small (to be shown in §2.1 , eqn.(2.4)), compared to the second, the electromagnetic body force. This is the driving force in *electromagnetic thrusters* such as *magnetoplasmadynamic thrusters*, and *pulsed plasma thrusters*. The energy expended in this process is given by the second term in eqn.(1.2).

It is important to note that, although the various means for using electrical power to accelerate gases has been explained in a conceptual fashion, the discovery of these methods was often empirical. For instance, magnetoplasmadynamic acceleration was discovered when an arcjet was operated under conditions of very low mass flow rates and high currents[3], at which the second term of eqn.(1.2) dominated the first.

The characteristics of some electric propulsion systems (electrostatic thrusters, electromagnetic thrusters, resistojets, and arcjets) are compared to that of other prevalent propulsion systems in Fig.(1.1). Though the advantage of electric propulsion, the ability to provide higher exhaust velocities, is immediately noticeable, so are its two major limitations.

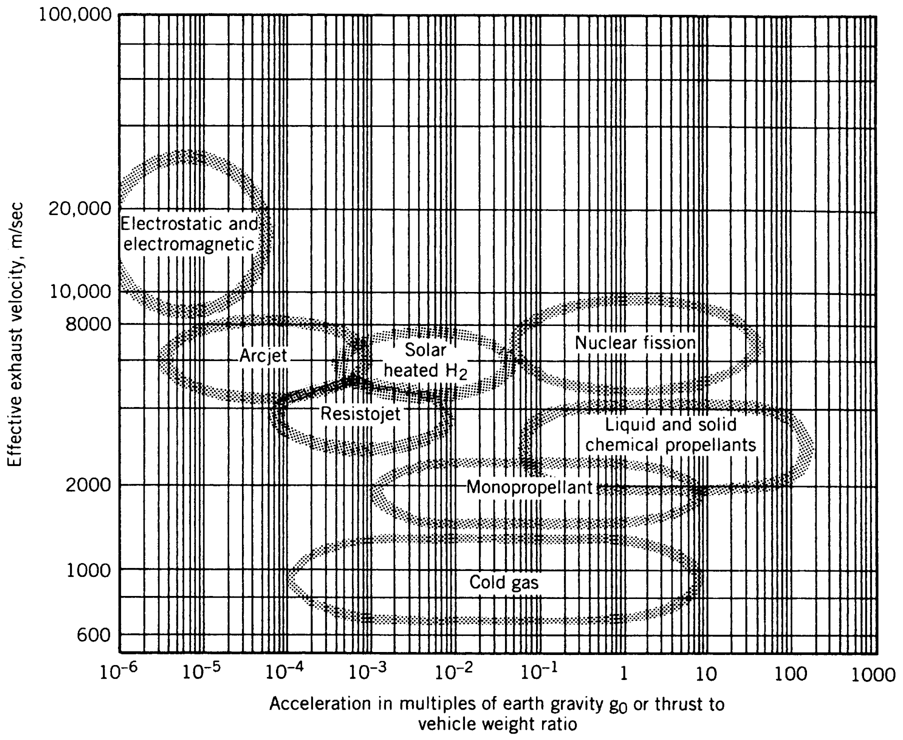


Figure 1.1: Realm of electric propulsion among other space propulsion systems (from ref.[4]).

First, in order to provide the obligatory electrical power, an on board power supply has to be carried along. Therefore, the problem now reduces to that of minimizing the combined mass of the power supply and the mass of the propellant, instead of merely the latter. As seen in eqn.(1.1), the required propellant mass *decreases* with increasing exhaust velocity. However, for a system that provides a constant thrust, at a constant efficiency, the mass of the powerplant *increases* with increasing exhaust velocity[2]. Thus, the exhaust velocity should be at an optimum value that minimizes the combined mass of propellant and power supply.

Second, all electric propulsion systems are intrinsically thrust limited (Fig.1.1), because, it is not practically feasible to supply electric power of the same order as that is available from chemical/nuclear reactions. This is obvious from observing that the kinetic power in the exhaust of the *Saturn V* rocket (about 120 GW) is more than 50 times greater than the maximum electrical power generation capacity of the Hoover dam. In the absence of such colossal electrical power supplies, electric propulsion systems are not suitable for overcoming steep gravitational potential wells, such as launching from earth's surface. However, they are well suited for missions where the instantaneous thrust requirements are small, but the total impulse ($\int_{t_0}^{t_f} T dt$) requirements are large.

Due to the above mentioned power supply penalty, the choice of a propulsion system should be mission specific. Over their periods of development, each of the varieties of electric propulsion systems have spawned their own array of technical specialties and subspecialties, and their own cadres of proponents and detractors[5]. In a serious assessment, many of these technologies have carved out their own niche, and have validly qualified for specific applications, to various degrees of suitability (cf. Fig.(1.1)).

Magnetoplasmadynamic thrusters (MPDT) and pulsed-plasma thrusters (PPT) use a quasineutral plasma as a working fluid and therefore, unlike ion thrusters, are not constrained by space-charge limitations. Moreover, since MPDTs and PPTs rely on the interaction of the collision dominated, rather than the Hall, current with either applied or self-induced magnetic field, to produce thrust, they can operate at much larger mass flow

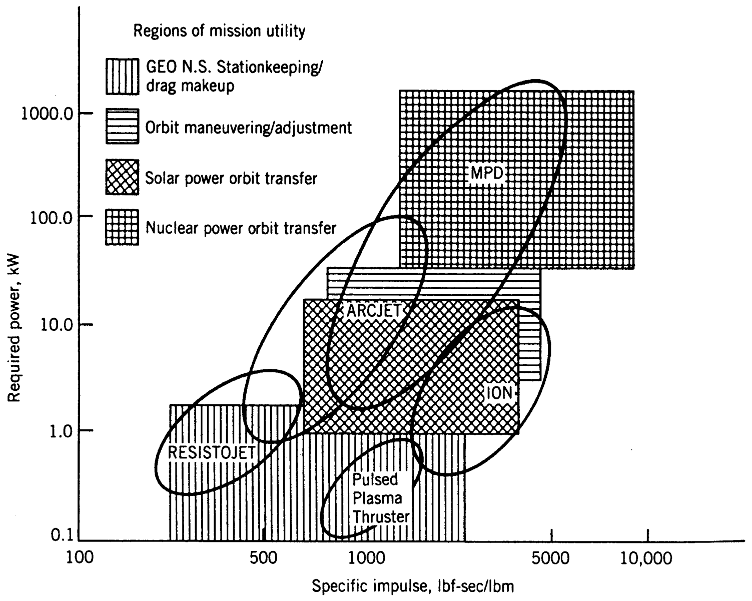


Figure 1.2: Realm of electromagnetic accelerators among other electric propulsion systems (from ref.[4]).

rates and densities than Hall thrusters.

The MPDT has the unique capability, among all developed electric thrusters, of processing megawatt power levels in a simple, small and robust device, producing thrust densities as high as 10^5 N/m^2. These features render it an attractive option for high energy deep-space missions requiring higher thrust levels than other EP systems[5], such as piloted and cargo missions to Mars and other outer planets, as well as for near term missions such as transfer from LEO to GEO[6].

1.2 Motivation for This Thesis

Among the various types of thrusters mentioned earlier, this thesis focuses only on magnetoplasmadynamic thrusters, even though the methods developed in this work can be extended to study pulsed plasma thrusters. All the subsequent discussions will be on the gas-fed, self-field MPDT. Further information on the PPT can be obtained from ref.[2] or from more recent works such as ref.[7].

Despite over three decades of research and development, involving experimental tests of many permutations of geometries and operating conditions, and space flights[8], currently no MPDTs are used on operational spacecraft.

This is because standard gas-fed MPDTs have unacceptably low efficiencies at low power levels that are available today, their solid cathodes have intolerably high erosion rates, and exhibit performance limiting oscillations at operation above a certain value of $J^2/\dot{m}$.

Recent work in Russia[9] and Princeton has demonstrated that using alkali metals such as lithium as the propellant, and a multi-channel hollow cathode, could alleviate these two pitfalls. First, since lithium has a low ionization potential, the fraction of input power expended in ionizing the propellant is reduced. Second, lithium coated electrodes have a lower work function than uncoated electrodes[9, 10]. This reduces the electrode erosion rates significantly, thus improving their lifetime. This version of the MPDT, called the Lithium Lorentz force accelerator (Li-LFA)[9] has demonstrated essentially erosion-free operation for over 500 hours of steady thrusting at 12.5 N, 4000 sec I_sp, and 48% efficiency at 200 kW.

For high energy, deep-space missions, these thrusters would operate at power levels of several megawatts. However, at these conditions, both ground and space testing of these highly promising devices present formidable technological and economic challenges. Indeed, there are no experimental facilities in the U.S. that are capable of long-term MW operation of steady-state MPDTs[5]. This implores the need for a rational alternative to empirical testing, as a way to predict performance.

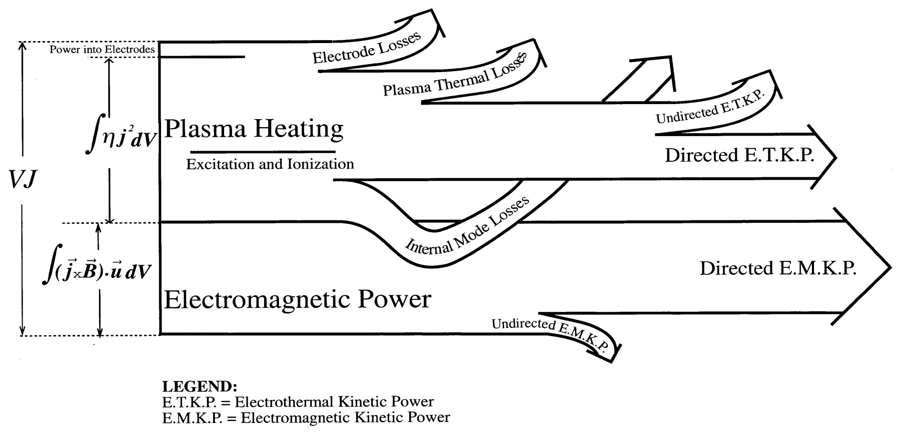


Figure 1.3: Expenditure of input power in an electromagnetic accelerator (not to scale, from ref.[11]).

Unfortunately the simple explanation for the acceleration mechanism described earlier, in §, belies the complexity that underlies the electromagnetic acceleration process, which embodies interlocking aspects of compressible gasdynamics, ionized gas physics, electromagnetic field theory, particle electrodynamics[2] and plasma-surface interactions[10]. The resulting complexity makes the realistic description of the flow analytically intractable. This inability is the foremost hindrance to understanding the details of processes by which the electrical energy is partitioned among various energy sinks, including acceleration.

As shown in Fig.(1.2), the electrical power deposited into the plasma can be expended into many sinks, only two of which, directed electromagnetic kinetic power and directed electrothermal kinetic power, are useful for propulsion. Understanding and quantifying these disparate processes is essential to improving the efficiency of these devices. Since it is difficult to do so using an empirical or analytical approach alone, numerical simulations are valuable tools in plasma thruster research.

Besides their importance in the understanding of electrical energy conversion process, numerical simulations can play a significant role in guiding research issues such as thermal modeling, near-cathode plasma characterization and electrode thermal management to understand erosion processes, propellant selection, active turbulence control, near-field plume model as a source for far-field plume simulations that study the role of plasma interactions and contamination of spacecraft, and other contentious issues in overall design optimization.

The objective of this work is to develop numerical schemes, and adapt existing ones, for the solution of the MHD equations, and integrate associated physical models to simulate plasma flows in self-field MPDTs. Through the years, there have been several notable attempts to develop multi-dimensional numerical models for MPDT flows. Some of them will be summarized below.

Kimura $et\ al.$ [12], and Fujiwara $et\ al.$[13], started developing single-temperature, 2-D models on simple geometries, and have continued to make improvements to their models. Currently, the efforts of Fujiwara[14] $et\ al.$ are directed at studying critical phenomena in magnetoplasmadynamic thrusters, using multi-temperature models.

Caldo and Choueiri[15] developed a two-temperature model to study the effects of anomalous transport, described in ref.[16], on MPD flows. The steady state form of Faraday's law and time-dependent form of the flow equations were solved using a multigrid, multi-stage time iteration scheme, on a gird that was customized for Princeton's Full-Scale Benchmark Thruster (FSBT).

The effort by LaPointe[17] was aimed at simulating plasma flows in existing thrusters, specifically Univ.Stuttgart's ZT-1 thruster and Princeton's Half-Scaled Flared Anode Thruster (HSFAT). An attempt to understand geometric scaling issues was also made in this work.

Martinez-Sanchez[18],[19] $et\ al.$ have developed two-temperature axisymmetric numerical models to study various aspects of the flow. The model of Niewood includes a non-equilibrium ionization model developed by Sheppard[20], and accounts for effects due to the presence of neutrals, such as ion-neutral slip.

Turchi[21] $et\ al.$ use MACH2, an unsteady MHD solver developed for high power plasma gun simulation, to model PPTs and MPDTs in many geometries. MACH3[22], the next generation of MACH2, is also used to simulate possible 3-D effects in specific situations.

The most persistent effort so far has been that of Sleziona[23],[24] $et\ al.$ at University of Stuttgart, who have been developing numerical models for MPDTs since 1981. Detailed models for many transport processes and multiple levels of ionization have been incorporated into their governing equations, which are solved on unstructured adaptive grids. The major objective of this work is to predict the overall performance of the ZT-1 and the HAT series of thrusters. This code is also used as a tool to study the role of gradient-driven instabilities in MPD flows.

Though these works have made significant progress in simulating MPD flows, they have notable shortcomings. In general, three shortcomings of existing numerical models can be identified.

1. Some of the existing codes exhibit numerical instabilities at high current levels. MPD thrusters perform better at higher currents, until a value at which voltage oscillations occur, and many of the important research questions tend to also occur at higher current levels. Consequently, the inability of a simulation to work reliably at those situations undermines its value.

   A probable explanation for these instabilities is the failure to solve the magnetic field evolution self-consistently with the flow. For highly resistive flows, the time scale for resistive diffusion of the magnetic field is orders of magnitude smaller than that of convection. However, in MPD flows it is common to have resistivities of

$\mathcal{O}(10^{-4})$ Ohm.m. In such situations, these time scales are not very far off, and there is a strong coupling between the flow and the magnetic field. The corresponding magnetic Reynolds' numbers indicate that both convective and resistive diffusion of the magnetic field are important. Moreover, the Alfvèn and fluid time scales are not very disparate. Therefore, the full set of equations describing the flow field and magnetic field evolution have to be computed self-consistently.

An important feature of the MHD formalism is the multitude of waves it permits to exist. The nonlinear coupling of these waves play an important role in determining physical phenomena and in computing the solution, as explained in ref.[25]. Solving Maxwell's equations consistently with compressible gasdynamics equations naturally produces waves physically associated with the problem, such as Alfvèn and magnetosonic waves, as eigenvalues. Such a formulation is thus suitable for handling MHD waves and shocks.

2. Some of the earlier efforts[15, 26] have experienced problems conserving mass, momentum, and energy. A conservative formulation of the governing equations ensures that these quantities are indeed conserved. Such a formulation also facilitates the application of boundary conditions, since the fluxes are the only quantities to be specified at the boundaries. From the perspective of numerical solution, it can be shown that conservative formulation is necessary for accurately capturing discontinuities.

3. As also noted in ref.[19], none of the existing models (with the exception of recent work ref.[24]) take advantage of the developments in the techniques for numerical solution of Euler and Navier-Stokes equations.

Each of the problems mentioned above can be overcome, respectively, by adapting the following approach:

1. Treat the flow and magnetic field equations in a self-consistent manner,

2. Formulate the governing equations in a conservative form,

3. Use characteristics-splitting techniques satisfying Rankine-Hugoniot relations, combined with anti-diffusion to increase accuracy. These techniques can capture shocks and other strong gradients in a non-oscillatory manner, and can have good spatial accuracy in smooth regions of the flow.

1.3 Organization of This Thesis

The physical models that describe the evolution of the flow and the field are developed in chapter 2. A concise review of existing analytical models is made, and the need for a detailed multi-dimensional, time-dependent numerical simulation is demonstrated. Subsequently, the MHD equations, which form the core of the current model, are briefly reviewed. Models for classical and anomalous transport coefficients, effects of thermal nonequilibrium between electrons and ions, a real equation of state, and a multi-level equilibrium ionization model are also developed in chapter 2.

The techniques that are used to obtain a numerical solution of the governing equations are discussed in chapter 3. First, the relevant fundamental concepts are summarized. Following that, the new characteristics-splitting scheme developed for the solution of the ideal MHD equations is described. Finally, the validation of this scheme, by solving standard test problems with known analytical solutions, is also described in chapter 3.

The physical models, developed in chapter 2, are incorporated into the numerical scheme, developed in chapter 3, and are used to simulate plasma flows in a real MPDT configuration in chapter 4. First, a description of the thruster configuration is made, followed by the appropriate boundary conditions required to obtain a solution. The profiles of many relevant physical properties, obtained from the converged numerical solution, are then compared with experimental data in chapter 4.

A summary of this work and the recommendations for the future of this work are done in chapter 5.

Chapter 2

PLASMA FLOW MODELING

No knowledge can be certain if it is not based upon mathematics, or upon some other knowledge which is itself based upon the mathematical sciences.
Leonardo da Vinci.

The physical laws governing the flow of the plasma and the evolution of the magnetic field in MPDTs are discussed in this chapter. First, the case for a continuum treatment of the plasma is made. Then, a brief review of existing analytical models for MPDT flows is made and the need for a comprehensive multi-dimensional time-dependent model is demonstrated. Subsequently such a model, the set of MHD equations, is discussed along with appropriate expressions for classical and anomalous transport, a real equation of state, a multi-level equilibrium ionization model, and the effects of thermal nonequilibrium.

2.1 Continuum Model of the Plasma

The mechanism of plasma acceleration in an electromagnetic thruster can be described in terms of the mean trajectories of the current-carrying electrons, as done in ref.[2]. While attempting to follow the applied electric field, the electrons are turned in the stream direction by the magnetic field with a velocity,

$$\mathbf{V}_D = \frac{\mathbf{E} \times \mathbf{B}}{B^2} \ . \tag{2.1}$$

The motion of the electrons sets up microscopic polarization fields that accelerate the ions. The transfer of streamwise momentum from electrons to the bulk of the plasma can also be accomplished through collisions. It is important to observe that, in either process, the working fluid remains quasineutral, as there is no macroscopic charge separation.

In the availability of limitless computing power, the perfect simulation would solve for the trajectory of every particle in three-dimensional space, subject to fundamental physical laws such as Maxwell's equations and Newton's laws. However, for conditions of interest to plasma thrusters, with electron and ion densities of $\mathcal{O}\left(10^{21}\right)/\mathrm{m}^3$, bulk velocities $\mathcal{O}(10^4)$ m/s, temperatures of ~ 2 eV, thermodynamic pressures of $\mathcal{O}(10^{-1})$ Torr and magnetic pressures of $\mathcal{O}(10^1)$ Torr, particle simulations are not presently practical. The most useful approach to understanding the nature of electromagnetic acceleration is that of magneto-hydrodynamics, in which the ionized gas is treated as a continuum fluid, whose physical properties are described by a set of bulk parameters, and whose dynamical behavior is represented by a set of conservation relations[2].

For the conditions described above, the characteristic length scale of charge separation, the *Debye length*, is much smaller than the dimensions of the device,

$$\frac{\lambda_D}{L} \sim \mathcal{O}\left(10^{-5}\right) \ . \tag{2.2}$$

The characteristic time scale for the electrons to adjust to the charge separation (inverse of *electron plasma frequency*) is much smaller than the residence time scale,

$$\frac{\tau_{res}}{1/\omega_{pe}} \sim \mathcal{O}\left(10^{5}\right) \ . \tag{2.3}$$

Moreover, the ratio of electrostatic force (due to any charge imbalance) to inertial forces,

$$\frac{\rho_e E}{\rho \mathbf{U} \cdot \nabla \mathbf{U}} \sim \mathcal{O}\left(10^{-9}\right) \ , \tag{2.4}$$

is negligible. For the reasons described above, the assumption of quasineutrality is justified.

The continuum treatment is sensible because, the mean free path between collisions is found to be much smaller than the characteristic length scale of the device, as illustrated by their ratio,

$$\frac{L}{\lambda_{mfp}} \sim \mathcal{O}\left(10^3\right) \ . \tag{2.5}$$

In self-field accelerators, the flow of the plasma is perpendicular to the magnetic field. In these situations, the magnetic field further bolsters the continuum approximation by playing the role of collisions in maintaining Maxwellian distributions and in providing the 'localizing influence' that is the essential ingredient of the fluid theory[27].

Now that the continuum treatment has been justified, the acceleration mechanism (in a self-field device) can now be described as follows: the applied electric field, **E**, induces a current of density **j**, which induces a magnetic field **B**. The interaction of the current and the magnetic field produces a distributed body force density, $\mathbf{f_B} = \mathbf{j} \times \mathbf{B}$, that accelerates the flow.

2.2 Simplified Analytical Models

An analytical model, based on the continuum description, to predict the *electromagnetic* component of thrust was developed by Maecker[28], and later expounded by Jahn[2]. The essence of this approach is to calculate the unbalanced magnetic pressure acting on the thruster, from the surface integrals of the magnetic stress tensor (to be explained in §2.3.2). The result of the analysis is that for a total operating current of J, the thrust is given by,

$$T = \frac{\mu_o}{4\pi}\left(\ln\frac{r_a}{r_c} + A\right) J^2, \tag{2.6}$$

where r_a and r_c are the radii of the anode and the cathode respectively, and A is a dimensionless constant between 0 and 1. Note that this formula needs no information on field distribution patterns inside the thruster, or the propellant type, or even the mass flow rate. Yet, the Maecker's formula (eqn.(2.6)) generally predicts the thrust with acceptable accuracy over a wide range of operating conditions.

However, as pointed out in ref.[29], the deviations from Maecker's law are significant at currents below a critical value. The model of Choueiri[29] accounts for variations in current distribution patterns, and propellant types. However, this model is semi-empirical in nature since it requires some experimental data for current distribution pattern on the electrodes, and the pressure distribution on the backplate. Nevertheless, it provides excellent predictions of thrust and is very useful in the understanding of thrust scaling trends. Specifically, it was shown that the non-dimensional parameter

$$\xi \equiv \frac{\frac{\mu_o J^2}{4\pi \dot{m}}\left(\ln \frac{r_a}{r_c} + A\right)}{\sqrt{2\phi_i/M}}\ , \tag{2.7}$$

where,

A is a dimensionless constant between 0 and 1,

$\dot{m}$ is the mass flow rate,

ϕ_i is the first ionization potential of the propellant,

M is the atomic weight of the propellant,

plays an important role in many scaling relations. It has been shown that nominal operation is achieved at $\xi \simeq 1$.

By and large, these analytical models have been successful in accurately predicting the electromagnetic thrust. This is because a good estimate of the magnetic pressure acting on the boundaries can be made from the total current, without detailed knowledge of field distributions within the channel.

However, as mentioned in §1.2, prediction of efficiency is a far more challenging endeavor. Some of the earlier efforts to understand the energetics were made by DiCapua[30], Villani[31] and King[32] at Princeton. Armed with experimental information on relevant parameters, they constructed simple analytical models to estimate the expenditure of energy.

DiCapua investigated the energy loss mechanisms in a parallel plate accelerator, using a simple 1-D model. A magnetic boundary layer model, in which the convection and diffusion of the magnetic field was estimated, was used to explain some of the features of the discharge region. The analysis using momentum and energy balance indicated that over currents ranging from 10 kA to 100 kA, with argon mass flow rates such that $J^2/\dot{m} = 37.0$ kA^2/g/s, up to 85% of the input power appeared in the exhaust. However, only $\sim 20\%$ was in the form of directed kinetic energy of the flow, with the remainder going into ionization and raising the enthalpy. It was concluded that the Ohmic heating term was always greater than the energy expended in accelerating the working fluid (cf. ref.1.2).

Villani's efforts were focused on the understanding effect of current distribution on the efficiency of coaxial thrusters. From simple order-of-magnitude analysis, it was determined that predicting the total power consumed by the thruster reduces to the problem of estimating the volume integral of the Ohmic heating term. Observing that, in the channel, the variations in j^2 far exceed the corresponding variations in the values of resistivity, it is demonstrated that Ohmic heating can be minimized with a curl-free current distribution[31].

Using 1-D models, King attempted to relate the terminal characteristics, such as voltage, specific impulse, and thrust efficiency, to the total current, mass flow rate and geometry of self-field coaxial MPDTs. It was determined that decreasing the anode radius and increasing the electrode length precluded the occurrence of large Hall parameters, and thus delayed the onset of voltage oscillations. By relating the thermodynamics of energy addition to the plasmadynamic quantities, this model predicted an upper bound of $\sim 70\%$ for the thrust efficiency, at large magnetic Reynolds' numbers.

Though these analytical investigations have reasoned out what the important issues are, they provide no assistance in prescribing precise guidelines for design. For instance, they do not attempt to *predict* the current distribution patterns, and thus Ohmic heating terms, for a given geometry and operating condition. This is because the task of assessing the energy deposition into various modes of the plasma in an accelerator requires detailed knowledge of the electromagnetic fields, the gasdynamic flow fields, and the thermodynamic state of the plasma. The difficulty of the task is further magnified by the complex non-equilibrium nature and the non-ideal equation-of-state of the working fluid[32]. Therefore, any investigation neglecting these complexities is reduced to a "black box" or a terminal analysis of the accelerator[30].

To overcome these limitations, it is apparent that a comprehensive time-dependent, multi-dimensional treatment of the plasma flow is required. One such model, with appropriate descriptions of transport and non-equilibrium effects, will be described in the next section.

2.3 Multidimensional Time-Dependent Model

The governing equations of conservation of mass, momentum, energy and magnetic flux will be derived in this section. Much of the discussion in this section is based on refs.[33, 27].

2.3.1 Conservation of Mass

In a single fluid formulation, the global continuity equation for the plasma can be written as,

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0 \, , \tag{2.8}$$

where ρ is the average density of all the species. Note that there are no source/sink terms because the average density is not affected by ionization/recombination reactions.

However, the electrons are created and destroyed in ionization/recombination reactions, and they obey a different continuity relation,

$$\frac{\partial \rho_e}{\partial t} + \nabla \cdot (\rho_e \mathbf{u}_e) = m_e \dot{n}_e , \tag{2.9}$$

where ρ_e is the density of the electron fluid, $\dot{n}_e$ is the net ionization/recombination rate, and the electron velocity is $\mathbf{u}_e = \mathbf{u} - \mathbf{j}/en_e$.

The adapted models for ionization and recombination will be described in §2.3.7.

2.3.2 Conservation of Momentum

The relation for conservation of momentum of the plasma is analogous to the Navier-Stokes momentum equation, with external body force acting on the fluid. This can be written as,

$$\frac{\partial \rho \mathbf{u}}{\partial t} + \nabla \cdot (\rho \mathbf{u}\mathbf{u} + \bar{\bar{p}}) = \nabla \cdot \bar{\bar{\tau}}_{vis} + \mathbf{f}_{ext} \, .$$

For the case under consideration, the external force is the Lorentz force, $\mathbf{f}_{ext} = \mathbf{j} \times \mathbf{B}$, per unit volume of the plasma.

Using the vector identities eqns. (A.1), (A.2), and (A.3), along with the definition of the Maxwell stress tensor,

$$\bar{\bar{\mathcal{B}}}_M = \frac{1}{\mu_o} \left[\mathbf{B}\mathbf{B} - \frac{B^2}{2} \bar{\bar{\mathcal{I}}} \right] \, , \tag{2.10}$$

the conservation of momentum is written as:

$$\frac{\partial \rho \mathbf{u}}{\partial t} + \nabla \cdot \left(\rho \mathbf{u}\mathbf{u} + \bar{\bar{p}} - \bar{\bar{\mathcal{B}}}_M \right) = \nabla \cdot \bar{\bar{\tau}}_{vis} + \mathbf{B} \left(\nabla \cdot \mathbf{B} \right) \ . \tag{2.11}$$

In a self-field MPDT, the inertial term, ρu^2, is typically $\mathcal{O}\left(10^4\right)$ Pa, the thermodynamic pressure is typically $\mathcal{O}\left(10^3\right)$ Pa, and the magnetic pressure is typically $\mathcal{O}\left(10^4\right)$ Pa.

The importance of viscous effects in plasma thruster flows is still an open question, although it is known to depend on the overall geometry. Viscosity tends to complicate the discharge physics by reducing the back EMF, which alters the current distribution in the channel. This, in turn, redistributes the local dissipation, which affects the local temperature, which changes the coefficient of viscosity. Wolff[34] attributed the unexpectedly low values of thrust from geometries with long electrodes to the detrimental effects of viscous drag. Although DiCapua[30] and Villani[31] have shown from order-of-magnitude analysis that viscous dissipation is probably a second order effect, they have never been empirically quantified in plasma thrusters. Heimerdinger[35] examined the strong variation of the coefficient of viscosity with temperature and concluded that it is important in certain regimes, though the effect on the overall characteristics are small. For now, viscous effects can be included in the general formulation of the physical model, though they will be ignored during the applications of this model. Recall from §2.2 that the use of Maxwell stress tensor permitted the distributed body force density, $\mathbf{j} \times \mathbf{B}$, to be written as a surface pressure term.

Although the Maxwell's equations prescribe that $\nabla \cdot \mathbf{B} = 0$, the terms involving this quantity are maintained in the present treatment for a numerical reason that will be explained in §2.3.9.

All the preceding discussion had an implicit assumption that all the heavy species, ions of various stages of ionization and neutrals, can be treated as a single fluid. However, it is important to realize that the neutrals are not affected by the electromagnetic forces, but rather by collisions with ions. Under extreme conditions of very low densities and very high magnetic fields, ion-neutral collisions may become sufficiently rare that the ions traverse the accelerator channel or achieve cycloidal drift of their own, without interacting with the neutrals. This condition is referred to as *ion slip*; if severe, as explained in ref.[2], ion slip could lead to an uncoupling of the electromagnetic processes from the gasdynamics thus resulting in an inefficient thruster. However, for the conditions of interest here, both the ionization fraction and the collision frequencies are high enough to warrant neglecting ion slip. Further discussion on this will be made in §2.3.3.

2.3.3 Faraday's Law

The relevant equation to determine the magnetic field evolution is Faraday's law:

$$\frac{\partial \mathbf{B}}{\partial t} = -\nabla \times \mathbf{E} \; . \tag{2.12}$$

In a collisionless plasma, the particles will be frozen to the magnetic field lines, and the induced EMF will cancel out the electric field. However, in the presence of collisions, the particles slip away from the field lines, and the electric field in the reference frame of the plasma is finite.

The resulting classical resistivity can be estimated from the electron momentum equation,

$$\rho_e \left[\frac{\partial \mathbf{u}_e}{\partial t} + (\mathbf{u}_e \cdot \nabla)\, \mathbf{u}_e \right] + \nabla p_e = -en \left\{ \mathbf{E} + (\mathbf{u}_e \times \mathbf{B}) \right\} + \sum_s m_e n \nu_{es} (\mathbf{u}_s - \mathbf{u}_e) \; , \tag{2.13}$$

where ν_{es} is the collision frequency between electrons with species *s*. Collisions among electrons do not contribute to this because the momentum of any interacting pair of electrons is conserved and thus the total electric current carried by the pair is preserved (cf. ref.[36]). For the case in consideration, with effective ionization fraction $Z_{eff} \simeq 1$, the electron-ion collisions are far more important than electron-neutral collisions.

Using the relation $\mathbf{u}_e = \mathbf{u} - (\mathbf{j}/en_e)$, and ignoring the electron inertial terms, the resulting relation, *Ohm's law,* takes the form:

$$\mathbf{E} + (\mathbf{u} \times \mathbf{B}) = \eta_o j + \frac{(\mathbf{j} \times \mathbf{B}) - \nabla p_e}{en_e} \ , \tag{2.14}$$

where the resistivity is,

$$\eta_o = \frac{m_e \sum_s \nu_{es}}{n_e e^2} \ . \tag{2.15}$$

The energy-weighted average of the momentum transfer collision frequency between the electrons and ions is given by (cf. refs.[33, 37]),

$$\nu_{es} = n_s Q_{es} \sqrt{\frac{8k_B T_e}{\pi m_e}} \ ,$$

$$Q_{es} = \frac{\pi}{4}\left(\frac{Z_s e^2}{4\pi\epsilon_o k_B T_e}\right)^2 \ln\left(1 + \frac{144\pi^2 \left(\epsilon_o k_B T_e\right)^3}{n_e e^6 Z_{eff}^2 \left(Z_{eff}+1\right)}\right) \ .$$

The typical value of resistivity for plasma under consideration is, $\mathcal{O}\,(10^{-3} - 10^{-4})$Ohm.m. Under these circumstances the magnetic Reynolds' number, $Rm = \mu_o uL/\eta$, which is an estimate of the relative importance of back EMF to resistive voltage drop, is $\mathcal{O}\,(1)$. This implies that an ideal MHD treatment is not acceptable for these situations.

The Hall effect, which induces current conduction normal to the magnetic field and the applied electric field, can be ignored if the gyration time scale is much smaller than mean time between collisions. However, for electrons, the primary current carriers, this ratio, the *electron Hall parameter*, is,

$$\Omega_e = \frac{\omega_{c,e}}{\nu_{es}} \sim \mathcal{O}\,(1) \ . \tag{2.16}$$

Therefore, the Hall effect must be included in the Ohm's law.

As mentioned in §2.3.2, there could be an ion slip contribution to the current. This can be estimated to be,

$$\mathbf{j}_\mathbf{I} = (1-\alpha)^2 \frac{\Omega_e \Omega_i}{B^2} \left\{(\mathbf{j} \times \mathbf{B}) \times \mathbf{B}\right\} \ , \tag{2.17}$$

where Ω_e and Ω_i are the electron and ion Hall parameters respectively, $\alpha = n_i/(n_A + n_i)$ is the fractional degree of ionization ($0 \leq \alpha \leq 1$). In the ionization model adapted later, the ionization fraction, $\alpha \simeq 1$. For the cases considered here, $\Omega_e \sim \mathcal{O}(1)$. The ratio of ion to electron Hall parameters,

$$\frac{\Omega_i}{\Omega_e} \simeq \sqrt{\frac{m_e}{M_i}} \sim \mathcal{O}\left(10^{-2}\right) \ . \tag{2.18}$$

Therefore, the ion slip effect is small and can be neglected in the present discussion.

The Hall effect causes the resistivity to be a tensor with off-diagonal components, as shown in the appendix §A.2. So, Faraday's law can be written as,

$$\frac{\partial \mathbf{B}}{\partial t} = -\left[\nabla \times \{(\bar{\bar{\eta}} \cdot \mathbf{j}) - (\mathbf{u} \times \mathbf{B})\}\right] \ . \tag{2.19}$$

The convective diffusion can be written, using eqns. (A.4) and (A.2) from the appendix, as the divergence of a tensor. Maintaining the same formulation, the resistive diffusion can also be written as the divergence of a tensor, using some manipulations shown in §A.2 in the appendix. Thus, the relation for the evolution of magnetic field takes the form,

$$\frac{\partial \mathbf{B}}{\partial t} + \nabla \cdot (\mathbf{uB} - \mathbf{Bu}) = \nabla \cdot \bar{\bar{E}}_{res} + \mathbf{u}\left(\nabla \cdot \mathbf{B}\right) \ . \tag{2.20}$$

2.3.4 Conservation of Energy

The Navier-Stokes relation for the conservation of gasdynamic energy density,

$$\mathcal{E}_g = \frac{p}{\gamma - 1} + \frac{1}{2}\rho \mathbf{u} \cdot \mathbf{u} \ , \tag{2.21}$$

can be written as,

$$\frac{\partial \mathcal{E}_g}{\partial t} + \nabla \cdot \left[\left(\mathcal{E}_g + p\right)\mathbf{u}\right] = \nabla \cdot \left(\bar{\bar{\tau}}_{vis} \cdot \mathbf{u}\right) + \nabla \cdot \left(\bar{\bar{k}}_{th} \cdot \nabla T\right) + \dot{q} \ , \tag{2.22}$$

where,

$\frac{\partial \mathcal{E}_g}{\partial t}$	= rate of change of the energy density,
$\left(\frac{\rho u^2}{2}\right)\mathbf{u}$	= convective flux of kinetic energy,
γ	= ratio of specific heats,
$\frac{\gamma p}{\gamma-1}\mathbf{u}$	= convective flux of internal energy,
$\bar{\bar{\tau}}_{vis}\cdot\mathbf{u}$	= viscous heat transfer,
$\bar{\bar{k}}_{th}\cdot\nabla T$	= thermal conduction,
$\dot{q}$	= external energy source/sink.

For a typical MPDT plasma, the gasdynamic energy density is $\mathcal{O}\left(10^4\right)$ J/m^3. This is partitioned between the kinetic energy and the internal energy, and the ratio is a function of Mach number squared (kinetic energy/internal energy $\sim v^2/T \sim (v/a)^2 \sim M^2$).

For the reasons explained in §2.10, the viscous dissipation of energy can be assumed to be negligible. Globally, the thermal conduction can be shown to be small compared to the dominant terms. However, in the presence of strong thermal gradients, and in regions where convective heat transfer is small, thermal conduction can be significant and therefore has to be included in the model.

The external source is clearly the electrical power per unit volume, $\dot{q} = \mathbf{j}\cdot\mathbf{E}$. Using Faraday's law, Ampère's law without the displacement current, and the vector identity:

$$\begin{aligned}\nabla\cdot(\mathbf{E}\times\mathbf{B}) &= \mathbf{B}\cdot(\nabla\times\mathbf{E}) - \mathbf{E}\cdot(\nabla\times\mathbf{B}) \\ &= -\frac{\partial}{\partial t}\left(B^2/2\right) - \mu_o\left(\mathbf{j}\cdot\mathbf{E}\right)\ ,\end{aligned}$$

the power input to the plasma can be written as:

$$\mathbf{j}\cdot\mathbf{E} = -\left[\frac{\partial}{\partial t}\left(B^2/2\mu_o\right) + \nabla\cdot\frac{(\mathbf{E}\times\mathbf{B})}{\mu_o}\right]. \qquad (2.23)$$

The first term can be identified as the rate of change of energy density in the magnetic field, and the second term is the Poynting flux of electromagnetic energy. Defining the magnetogasdynamic total energy density (gasdynamic energy density + energy density in magnetic field) $\mathcal{E} = \mathcal{E}_g + \frac{B^2}{2\mu_o}$, the conservation of total energy density of the plasma can be written as:

$$\frac{\partial \mathcal{E}}{\partial t} + \nabla \cdot \left[\left(\mathcal{E} + p \right) \mathbf{u} - \bar{\bar{\mathcal{B}}}_M \cdot \mathbf{u} \right] = \nabla \cdot \left[\frac{-\mathbf{E}' \times \mathbf{B}}{\mu_o} + \bar{\bar{k}}_{th} \cdot \nabla T \right] + \left(\frac{\mathbf{B}}{\mu_o} \cdot \mathbf{u} \right) \left(\nabla \cdot \mathbf{B} \right) \ , \quad (2.24)$$

where the contribution of the back EMF and resistive drop to the electric field (in the Poynting flux term) have been separated to emphasize the energy expended in acceleration (back EMF) and in heating (resistive drop).

Under some physical conditions, when the magnetic pressure is several orders of magnitude larger than thermodynamic pressure, the conservation form of the energy equation may not be suitable. In these cases, since p is calculated from subtraction of one large number ($B^2/2\mu_o$) from another ($\mathcal{E}$), the associated numerical errors could be large. However, for the conditions that are of interest to plasma propulsion, the magnetic pressure is seldom two to three orders of magnitude greater than thermodynamic pressure. Thus the conservation form of the energy equation is, in general, numerically suitable here.

In order to treat the fluid as if it were in thermal equilibrium, the characteristic residence time scale should be much larger than the time scale for energy equilibration between electrons and ions. This ratio is,

$$\frac{\tau_{res}}{\tau_{equi}} \sim \mathcal{O}\left(10\right) \ . \quad (2.25)$$

Since this condition is not strongly satisfied, it may be important to treat the electrons and ions as separate fluids with separate temperatures. In fact, there is sufficient experimental evidence[38, 39] that this is indeed the case, but the disparity is less than an order of magnitude.

This suggests that the temperature of the individual species can be obtained by subtracting the energy of other components from the total energy. In order to do so, some rearrangements are necessary. The definition of the fluid energy density, eqn.(2.21), has to be split into the internal energy density and the kinetic energy density:

$$\mathcal{E}_g = \mathcal{E}_{int} + \mathcal{E}_{KE} \ . \tag{2.26}$$

With this definition, the conservation relation for the internal energy takes the form:

$$\frac{\partial \mathcal{E}_{int}}{\partial t} + \nabla \cdot [\mathcal{E}_{int}\mathbf{u}] + p\nabla \cdot \mathbf{u} = \eta j^2 + \nabla \cdot \left(\bar{\bar{k}}_{th} \cdot \nabla T\right) \ . \tag{2.27}$$

The internal energy of the fluid can be further split into those pertaining to electrons and ions,

$$\mathcal{E}_{int} = \mathcal{E}_i + \mathcal{E}_e \ . \tag{2.28}$$

The thermal conductivity in the total energy equation is the sum of the contributions from both electrons and ions,

$$k_{th}\nabla T = (k_e \nabla T_e) + (k_i \nabla T_h) \ . \tag{2.29}$$

With these assumptions, the conservation relations for the internal energy of electrons can be written as,

$$\frac{\partial \mathcal{E}_e}{\partial t} + \nabla \cdot [\mathcal{E}_e\mathbf{u}] + p_e\nabla \cdot \mathbf{u} = \eta j^2 - \Delta\dot{\mathcal{E}}_{ie} + \nabla \cdot \left(\bar{\bar{k}}_{th,e} \cdot \nabla T_e\right) \ , \tag{2.30}$$

and that of ions as,

$$\frac{\partial \mathcal{E}_i}{\partial t} + \nabla \cdot [\mathcal{E}_i\mathbf{u}] + p_i\nabla \cdot \mathbf{u} = \Delta\dot{\mathcal{E}}_{ie} + \nabla \cdot \left(\bar{\bar{k}}_{th,i} \cdot \nabla T_h\right) \ . \tag{2.31}$$

In deriving eqns.(2.30) and (2.31) it was also assumed that Ohmic heating primarily affects the electrons. Note that the energy expended in acceleration, $(\mathbf{j} \times \mathbf{B}) \cdot \mathbf{u}$, does not appear in eqns.(2.27), (2.30) and (2.31) becuase they are relations for the *internal* energy only. The acceleration energy would appear if the kinetic energy were also included in the definition of energy density.

It can be shown[33] that,

$$\frac{k_e}{k_i} \simeq \sqrt{\frac{M_i}{m_e}} \sim \mathcal{O}\left(10^2\right) \ . \tag{2.32}$$

Since the temperatures of electrons and ions are not very disparate, one could make the assumption that thermal conduction of the ions is negligible compared to that of the electrons. However, there may be some regions, such as stagnation points, where thermal conduction may be an important dissipation mechanism for the ions. Therefore it is advisable to retain this term.

In eqns.(2.30) and (2.31), the rate of exchange of energy between the electrons and the ions, through collisions, can be estimated as[27],

$$\Delta\dot{\mathcal{E}}_{ie} = \frac{3\rho_e \nu_{ei}}{M_i} k_B \left(T_e - T_h\right) \ . \tag{2.33}$$

Energy losses due to radiation are important in many types of plasmas. However, earlier work by Boyle[38], Villani[31], and Bruckner[39] suggests that the relative magnitude of this sink is not significant for the MPDT plasma. Consequently, they will be ignored in the current model. If deemed necessary, they can be added in without affecting the basic framework of the model, as explained by LeVeque *et al.*[40].

2.3.5 Equation of State

For a system with *N* structureless molecules in thermal equilibrium at a temperature *T*, moving freely in a volume *V*, the pressure is given by the ideal gas law. However, real molecules possess energy in modes other than the translational. In these situations, the relationship between pressure, density and temperature is of the form,

$$p = Nk_B T \frac{\partial \ln Q}{\partial V} \ . \tag{2.34}$$

Ignoring nuclear contributions, the total partition function, *Q* can be written as,

$$Q = Q_{rot} Q_{vib} Q_{tr} Q_{el} \ , \tag{2.35}$$

where Q_{rot} is the contribution of rotational energy levels, Q_{vib} that of the vibrational energy levels, and Q_{el} that of the electronic energy levels.

The estimation of the translation partition function is relatively straightforward (ref.[41]), and is found to be,

$$Q_{tr} = V\left(\frac{2\pi M_i k_B T}{h^2}\right)^{3/2} . \qquad (2.36)$$

However, the estimation of the internal partition functions is not as easy. Fortunately, most of the propellants of interest to plasma propulsion are monatomic in nature, and therefore, the rotational and vibrational contributions are absent. Thus the problem of finding the equation of state of a real gas reduces to the problem of estimating the electronic excitation partition function Q_{el}. These partition functions, for many elements of interest, can be found in references such as ref.[42]. Based on this work, Choueiri[43] derived expressions to obtain the temperature from pressure and density. As shown in Fig.(2.1), it is clear that at temperatures at or above 10^4K, the deviations from the ideal gas model are significant. This relation for the equation of state will be used in the current model, and further details

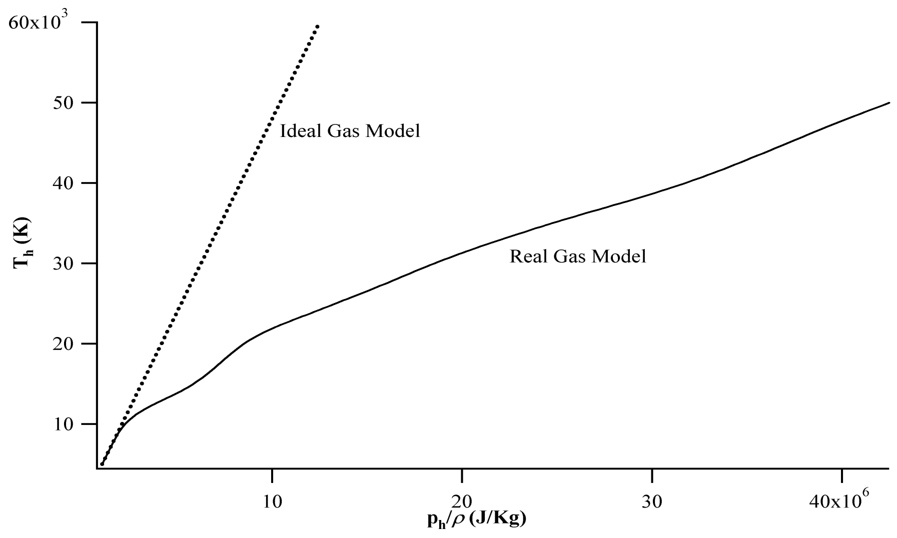


Figure 2.1: Deviation from ideal gas behavior for Argon

can be found in the appendix.

As energy is deposited into the internal modes, the ratio of specific heats also changes. Once again, this can be calculated from the data of partition functions. As seen in Fig.(2.2), the deviation from the ideal value of $5/3$ is severe at temperatures above 10^4K, which is consistent with Fig.(2.1).

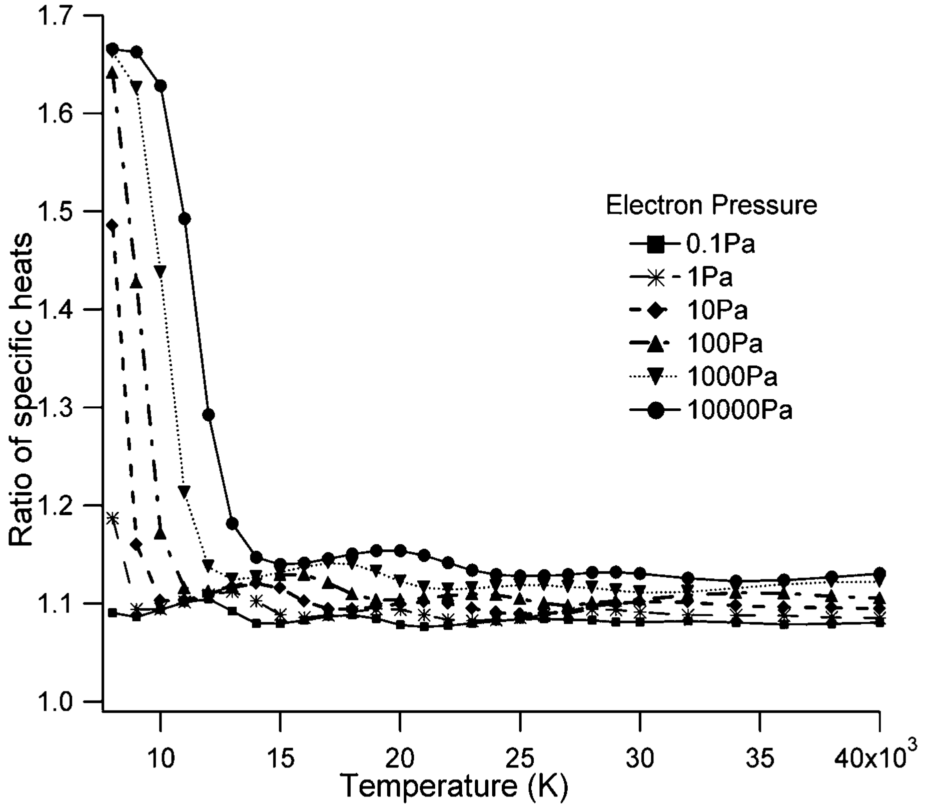


Figure 2.2: Variation of the ratio of specific heats for argon (calculated from data in ref.[42])

2.3.6 Anomalous Transport

It is known that the current can drive microinstabilities in the thruster plasma which may, through induced microturbulence, substantially increase dissipation and adversely impact the efficiency. The presence of microinstabilities in such accelerator plasmas has been established experimentally in the plasma of the MPDT at both low and high power levels[44], [45].

Choueiri[16] has developed a model to estimate the resulting anomalous transport and heating in terms of macroscopic parameters. Under this formulation, apart from the classical collision frequency of the particles, there exists additional momentum and energy transferring collisions between particles and waves. The resulting anomalous collision frequency is important whenever the ratio of electron drift velocity to ion thermal velocity,

$$\frac{u_{de}}{v_{ti}} = \frac{j}{en_e}\sqrt{\frac{M_i}{2k_B T_h}} \geq 1.5 \ . \tag{2.37}$$

Above this threshold, the ratio of anomalous collision frequency to classical collision frequency was found to depend on the classical electron Hall parameter, Ω_e, and the ratio of ion to electron temperatures, T_h/T_e. Polynomials giving these relations were derived in ref.[16] to be,

$$\begin{aligned} \tfrac{\nu_{e,AN}}{\nu_{e,cl}} = \ & \{0.192 + 3.33 \times 10^{-2}\Omega_e + 0.212\Omega_e^2 - 8.27 \times 10^{-5}\Omega_e^3\} \\ & + \tfrac{T_h}{T_e}\{1.23 \times 10^{-3} - 1.58 \times 10^{-2}\Omega_e - 7.89 \times 10^{-3}\Omega_e^3\} \ , \end{aligned}$$

and are shown in Fig.(2.3).

As a result, the effective resistivity of the plasma is now,

$$\eta_{eff} = \frac{m_e\left(\nu_{e,cl} + \nu_{e,AN}\right)}{e^2 n_e} \ . \tag{2.38}$$

2.3.7 Ionization Processes

It is imperative that, within the acceleration region, a significant fraction of the working fluid remains in a state of ionization, as the free charges are responsible for carrying current, and thereby establishing the electromagnetic fields required for acceleration. The reaction rates for ionization reactions must account for transitions from the ground states, as well as

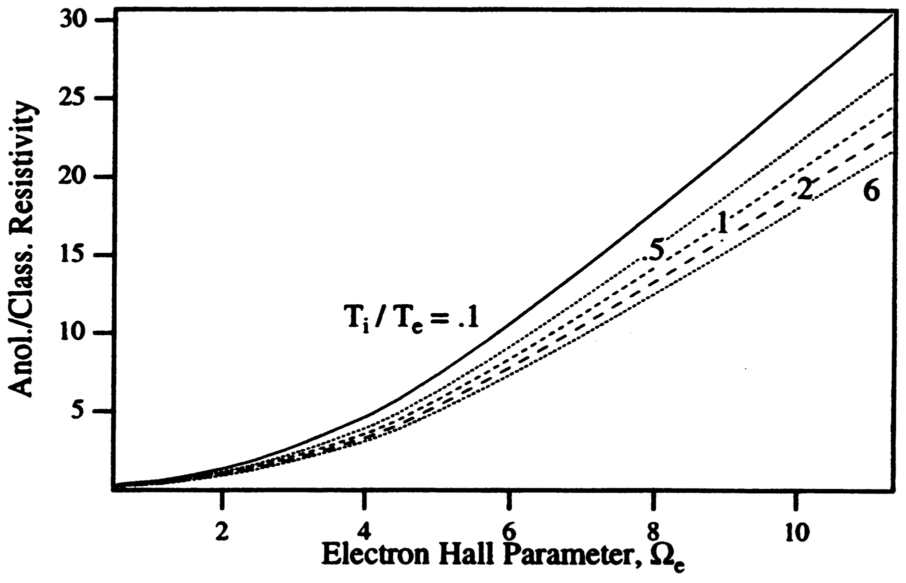


Figure 2.3: Ratio of anomalous to classical resistivity in argon plasmas (from ref.[16])

those from excited states. The work at the University of Stuttgart[46] has indicated that, for the conditions of interest to MPD plasmas, the solution of flow fields using the seemingly restrictive assumption of equilibrium ionization model may sufficiently be close to reality.

In equilibrium, irrespective of the manner in which the species are created, the densities of the electrons, n_e, ions, n_i, and the neutrals , n_o, are related by the Saha[47] equation,

$$\frac{n_i n_e}{n_{i-1}} = \frac{2\,(2\pi m_e k_B T)^{3/2}}{h^3} \frac{\sum_l g_l^i \; e^{-\epsilon_l^i/k_B T}}{\sum_l g_l^{i-1} \; e^{-\epsilon_l^{i-1}/k_B T}} = K_i \; , \tag{2.39}$$

where ϵ_l^i is the l^{th} energy level of the species of ionization level i, and g_l^i is the corresponding statistical weight.

Similar expressions can be written for higher ionization levels, with the energy levels scaled to a common ground. The propellant considered in this work is argon, and its first, second and third ionization potentials are 15.755 eV, 27.63 eV, and 40.90 eV respectively. The relevant energy levels of argon atom and its ions, and their statistical weights are given in Table (2.1).

Even in the presence of thermal nonequilibrium between electrons and ions, a modified Saha equation can be applicable. As shown in refs.[48] and [49], in such situations, due to the high mobility of the electrons, the temperature in eqn.(2.39) can be replaced with the temperature of the electron fluid, and the resulting modified Saha equation is an accurate model.

Table 2.1: Energy levels and statistical weights in argon and argon ions (obtained from refs. [50], [51] and [52])

Ar I		Ar II		Ar III		Ar IV	
E_l^0 (eV)	g_l^0	E_l^+ (eV)	g_l^+	E_l^{++} (eV)	g_l^{++}	E_l^{+++} (eV)	g_l^{+++}
0.000	1	0.059	6	0.111	29	0.000	4
11.577	8	13.476	2	1.737	6	3.478	16
11.802	4	16.420	20	4.124	2	14.671	24
13.096	24	16.702	12	14.214	6	31.133	24
13.319	12	17.177	6	17.856	1	35.568	40
14.019	48	17.688	28	17.964	10		
14.242	24	18.016	6	19.460	14		
14.509	24	18.300	12	20.066	1		
14.690	12	18.438	10	20.222	8		

For a model with *N* levels of ionization, the electron number density can be obtained by finding the single positive root of the polynomial (from Heiermann[37]),

$$n_e^{N+1} + \sum_{l=1}^{N} \left[n_e^{N-l} \left(n_e - l n_o \right) \prod_{m=1}^{l} K_m \right] = 0 \, , \tag{2.40}$$

where n_o is the total number density of all nuclei, and the equilibrium constant, K_m is from eqn.(2.39).

2.3.8 Summary of Governing Equations

The nucleus of this model is the set of single fluid MHD equations. The corresponding conservation laws, given by eqns. (2.8), (2.10), (2.12) and (2.24), can be summarized in the vector form:

$$\frac{\partial}{\partial t}\begin{bmatrix} \rho \\ \rho\mathbf{u} \\ \mathbf{B} \\ \mathcal{E} \end{bmatrix} + \nabla\cdot\begin{bmatrix} \rho\mathbf{u} \\ \rho\mathbf{u}\mathbf{u} + \bar{\bar{p}} - \bar{\bar{\mathcal{B}}}_M \\ \mathbf{u}\mathbf{B} - \mathbf{B}\mathbf{u} \\ (\mathcal{E}+p)\,\mathbf{u} - \bar{\bar{\mathcal{B}}}_M\cdot\mathbf{u} \end{bmatrix} = \nabla\cdot\begin{bmatrix} \mathbf{0} \\ \bar{\bar{\tau}}_{vis} \\ \bar{\bar{E}}_{res} \\ \mathbf{q} \end{bmatrix} . \tag{2.41}$$

Under this framework, ancillary relations such as the energy equation for the individual species, which do not fit into the conservation form, are solved separately, without affecting the underlying solver. Notice that the equations strictly contain a $\nabla\cdot\mathbf{B}$ term, that is not included in the conservation form. This will be discussed in the subsequent section.

2.3.9 Zero Divergence Constraint

Though it is physically true that $\nabla\cdot\mathbf{B}\equiv 0$, it is often not true numerically, as there may be truncation errors. The treatment of the terms are important, since they could be a cause of numerical instabilities, as explained in ref.[53]. The technique used here, based on the work of Powell[54], has a modified eigensystem (cf. appendix §B.1) that accounts for any possible errors in $\nabla\cdot\mathbf{B}$. The resulting scheme satisfies the relation,

$$\frac{\partial}{\partial t}(\nabla\cdot\mathbf{B}) + \nabla\cdot[(\nabla\cdot\mathbf{B})\,\mathbf{u}] = 0 \; . \tag{2.42}$$

In other words, the numerical scheme used in this work ensures that any artificial source of $\nabla\cdot\mathbf{B}$ is convected out of the domain.

In the case of a self-field accelerator in a coaxial geometry, the magnetic field is purely azimuthal. If the assumption of axisymmetry is made, then

$$\nabla \cdot \mathbf{B} = \frac{1}{r}\frac{\partial B_\theta}{\partial \theta} = 0 \ . \tag{2.43}$$

Therefore, in this work, the divergence of the magnetic field is always zero throughout the domain.

2.4 Summary

In summary, the physical models and the corresponding governing equations that describe MPDT flows were discussed in this chapter. The set of MHD equations, comprising the conservation relations for mass, momentum, energy and magnetic flux were described. Further improvements to the MHD model, such as the effects of thermal nonequilibrium, anomalous transport effects, along with relations for classical transport, a real equation of state and a multi-level equilibrium ionization model, were also developed in this chapter.

The techniques for obtaining a numerical solution of these governing equations will be discussed in the following chapter.

Chapter 3

NUMERICAL SOLUTION

Where the mind is without fear, and the head is held high ...

Rabindranath Tagore

Despite several earlier efforts in the front of MPDT flow simulation, as mentioned in §1.2, there remains a need for accurate and robust numerical schemes for this purpose. A new numerical method that has the potential to overcome the problems described in §1.2, will be described in this chapter. First, some fundamental concepts, pertaining to the guiding principles to be used in this thesis, will be reviewed. Then, the characteristics-splitting technique for the solution of the convection equations will be developed and validated. That will be followed by a brief discussion of the well known techniques for the solution of the diffusion equations.

3.1 Guiding Principles

In light of the discussion in §1.2, it is imperative that the numerical scheme developed in this work to strictly adhere to the guidelines listed below:

1. Treat the flow and magnetic field equations in a self-consistent manner,

2. Use a conservative formulation of the problem,
3. Use non-oscillatory discontinuity capturing techniques that satisfy Rankine-Hugoniot relations, combined with techniques to limit numerical diffusion and improve spatial accuracy.

The mathematical foundation for the numerical solver used in this work will be described in this chapter.

3.2 Mesh System

The fundamental aspect of a numerical solution obtained through differencing schemes, unlike that of an analytical solution, is that it is defined on a discrete domain. A rigorous treatment of this issue can be found in ref.[55], and only the information directly relevant to the present application will be discussed here. The true domain, $\mathcal{D}$, is divided into small control volumes, Ω_j, whose centers are given by position vectors $\mathbf{c}_j$. Within each of these control volumes, the solution $\mathbf{U}(x,t)$ is approximated by a constant $\mathbf{U}_j\left(t\right)$, which should be considered as an approximation of the mean value of **U** over the cell Ω_i rather than the value of **U** at point $\mathbf{c}_j$,

$$\mathbf{U}_j\left(t\right) \cong \frac{1}{|\Omega_i|}\int_{\Omega_i}\mathbf{U}\left(\mathbf{x},t\right)d^3\mathbf{x}, \tag{3.1}$$

where $|\Omega_i|$ is the volume of Ω_i.

Given an initial distribution $\mathbf{U}(x,0)$, and using the definition in eqn.(3.1), the time rate of change of **U** inside the control volume can be calculated from the sum of the fluxes through its boundaries,

$$\frac{\partial\mathbf{U}_j\left(t\right)}{\partial t} = -\frac{1}{|\Omega_i|}\sum\int_{\Gamma_{ij}}\mathcal{F}\cdot\mathbf{n}_i dA, \tag{3.2}$$

where Γ_{ij} is the boundary between cells Ω_i and Ω_j.

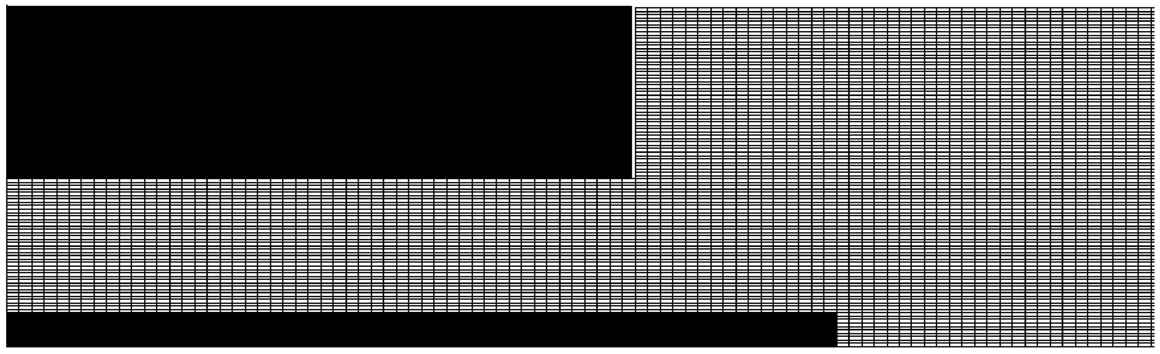


Figure 3.1: Uniform, orthogonal structured grid.

The adjoining issue is the detailed description of the control volumes, their identities, shapes, sizes, and locations. Mesh generation is an entire field of study in itself, and is beyond the scope of this work to get into the intricacies of that trade. Only a brief review that is relevant to the problem at hand will be made.

Generally, cylindrical coordinates are preferable for the study of plasma thrusters. Three distinct techniques of dissecting the domain into control volumes can be identified:

1. Structured, concentric cylindrical shells, separated by lines of constant $\hat{r}$, $\hat{\theta}$ and $\hat{z}$. Combined with the axisymmetric assumption, the control volumes are simply rectangles in the $r-z$ plane (cf. Fig.(3.1)).

2. Structured shells, separated by lines of a constant stream functions, η, ξ that fit the true boundaries as close as possible. Combined with the axisymmetric assumption, the control volumes are quadrilaterals in the $r-z$ plane (cf. Fig.(3.2)).

3. Unstructured irregular tetrahedrons that fit the true boundary exactly. Combined with the axisymmetric assumption, the control volumes are triangles in the $r-z$ plane (cf. Fig.(3.3)).

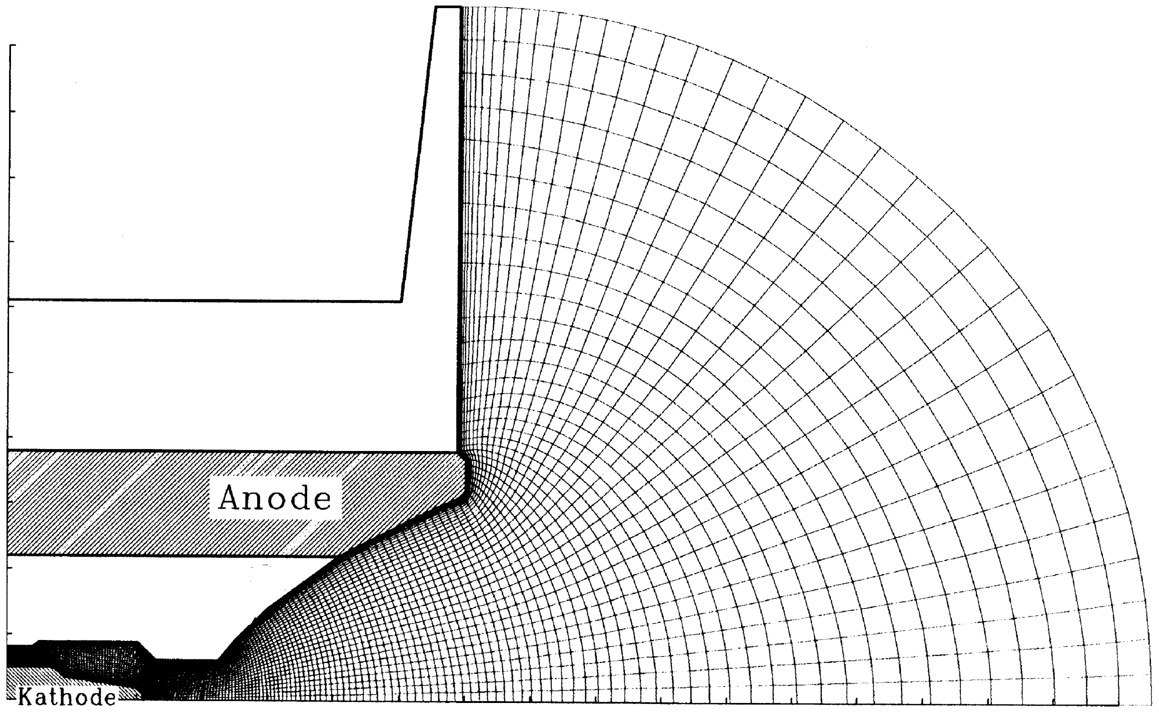


Figure 3.2: Non-uniform, structured grid (obtained from ref.[26]).

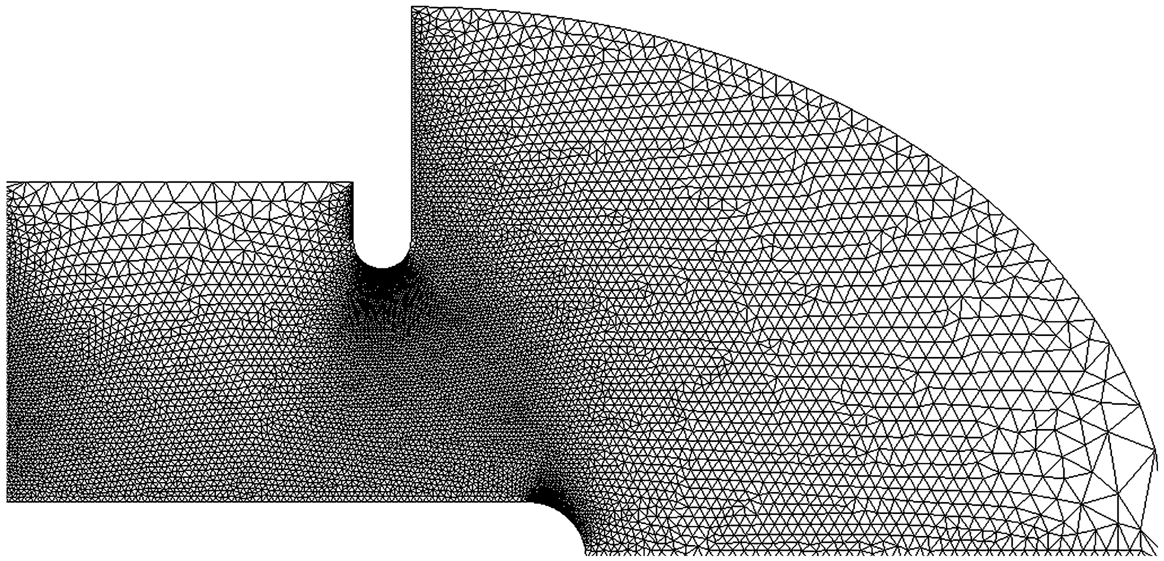


Figure 3.3: Non-uniform, unstructured grid.

Grids of type 3 may seem very attractive because of their adaptability to complex geometries, and have been popular (cf. refs.[23, 24]). However there are some disadvantages that may not be immediately apparent. Unstructured grids are computationally expensive and there are problems in extending higher order accurate schemes to them. Since the precise control of geometry may not be as critical to the design of plasma thrusters as it is to, say aircraft design, the use of unstructured grids may not be as worthwhile.

This work currently uses grids of type 1. Though they are computationally inexpensive and easy to implement, they has been found to be too restrictive on the types of real thrusters that can be simulated. Therefore, work is underway, beyond what is reported in this thesis, to implement this solver on body-fitted grids of type 2. It is important to note that the numerical schemes, that will be developed in this chapter, will not require any fundamental changes to be applied on grids of type 2.

The variables to be computed, given by **U** in eqn.(3.1), can be stored either in the vertices of the cells, or in the center of the cells (further discussion can be found in ref.[56]). In the former, the variables will coincide with the boundary, and they will be specified as boundary conditions. In the latter, the faces of the cells will be aligned with the walls, and the fluxes of these variables will be specified as boundary conditions. While solving the conservative formulation, it is preferable to choose the cell-centered scheme since specifying the fluxes is more compatible with the governing equations.

3.3 Consistency, Stability, and Convergence

The true mathematical form of the systems of conservation laws are integral relations, while the partial differential forms are actually a special case, in which smoothness of the solution is assumed. To use the differential form in the presence of a discontinuity in the solution, one has to introduce the concept of weak form of the differential equations (cf.

LeVeque[40]). Often, there are more than one possible weak solutions, of which, of course, only one is physical. If the numerical scheme converges to a solution, the question to be answered is, “Is it the correct solution?”.

3.3.1 Lax-Wendroff Theorem

The above question is answered by the *Lax-Wendroff theorem*, that states that (cf. [40]):

> *If the numerical approximation computed with a consistent and conservative method converges, then the converged solution is the correct solution of the conservation law*.

Notice that though the Lax-Wendroff theorem is a powerful result, it does not help to determine *if* a scheme is convergent, but merely assures that the converged solution is the correct one.

3.3.2 Criteria for Stability and Convergence

For a linear numerical scheme there are a few techniques available to analyze stability. The most popular of these was developed by John von Neumann[57] during the Manhattan project and is commonly referred to as the von Neumann stability analysis (VNSA). It involves discrete Fourier transforming the solution and inspecting the growth of waves in the frequency space. This approach is intuitively obvious because, like many physical instabilities, numerical instabilities are often a result of unbounded growth of certain oscillations. Using this technique, it is possible to decide on physically allowable combinations of grid spaces and time steps, that can result in a stable numerical scheme.

Given a consistent and stable linear numerical scheme, the goal is to ensure that, as the discrete grid is refined to better emulate the continuous domain of the true solution, the discrete solution approaches the true solution. In other words, the solution should converge. These three concepts, consistency, stability and convergence, are related by the *Lax Equivalence Theorem* which assures that (cf. [40]):

> *For a properly posed initial value problem, with a consistent linear numerical scheme, stability is the necessary and sufficient condition for convergence.*

This is indeed a useful result, since the VNSA can determine the stability of a linear method, and consistency can be easily verified, it is straightforward to determine if the numerical solution will converge to the true solution.

However, nonlinear convergence and stability are not simple extensions of their analog in the linear system. The Lax equivalence theorem is strictly applicable only to linear numerical schemes. Though some nonlinear equations can be linearized to obtained approximate answers, it is not of much help for truly nonlinear set of equations. For problems, such as the set of conservation laws seen in eqn.(2.41), requiring nonlinear operators, the need for establishing convergence still exists. Compounding the difficulty is the fact that VNSA is of practical use only for linear equations. In the complex set of highly nonlinear equations, such as MHD equations and Euler equations of compressible gas dynamics, a discrete Fourier transform would yield different frequencies. However, without the principle of linear superposition, any sort of VNSA becomes intractable, as explained by Laney[58].

Over the years, research in numerical techniques has led to the development of techniques to determine the stability and convergence of nonlinear systems of equations. The most useful among these are discussed below.

Total Variation Diminishing Schemes

One method of analyzing stability stems from the classical concept of total variation in real and functional analysis (cf. ref.[59]). Harten[60] adapted this to the sorts of functions seen in computational gasdynamics. The *total variation* of a function **U** can be defined as:

$$TV\left(\mathbf{U}\right) = \int_{-\infty}^{\infty} \left| \frac{d\mathbf{U}}{dx} \right| dx. \tag{3.3}$$

For the discrete situation, 3.3 can be written as:

$$TV\left(\mathbf{U}^n\right) = \sum_j \left| \mathbf{U}^n_{j+1} - \mathbf{U}^n_j \right|. \tag{3.4}$$

It can be proven (cf. ref.[58]) that the solutions to the conservation equations must be *total variation diminishing (TVD)* in the sense:

$$TV\left(\mathbf{U}^{n+1}\right) \leq TV\left(\mathbf{U}^n\right), \tag{3.5}$$

for all time levels. Since oscillations add to the total variation, the TVD condition cannot be satisfied by unbounded growth of oscillations. Thus, the TVD condition can be used as a stability check for nonlinear equations.

Local Extremum Diminishing Schemes

Though the TVD principle is a popular check for convergence, it has its limitations. Since it imposes a condition only on the global variations, a scheme satisfying TVD condition eqn.(3.5) could, theoretically, allow spurious local oscillations (cf. ref.[58]). A more practical limitation of the TVD condition is that its extension to multidimensional problems does not provide a satisfactory measure of oscillations. To overcome both these shortcomings, Jameson[61] has developed the concept of *local extremum diminishing (LED)* schemes, which is summarized below.

Any time dependent conservation law, such as eqn.(2.41), can be written in a general form:

$$\frac{d\mathbf{U}_j}{dt} = \sum_{j \neq k} C_{j,k} \left(\mathbf{U}_k - \mathbf{U}_j \right). \tag{3.6}$$

If the numerical scheme has a compact stencil, in which the value at the point is directly dependent only on its nearest neighbors, and if the coefficients are all non-negative, then:

$$C_{j,k} = \begin{cases} \geq 0; k = j \pm 1, \\ = 0; \text{else} \end{cases} \tag{3.7}$$

If $\mathbf{U}_j$ is a local maximum, then, $(\mathbf{U}_k - \mathbf{U}_j) \leq 0$, causing $\frac{d\mathbf{U}_j}{dt} \leq 0$. Conversely, if $\mathbf{U}_j$ is a local minimum, then, $(\mathbf{U}_k - \mathbf{U}_j) \geq 0$, causing $\frac{d\mathbf{U}_j}{dt} \geq 0$. These schemes are, therefore, aptly called *local extremum diminishing* (LED). Apart from ensuring that there are no local oscillations, the schemes built on this condition can be easily extended to multi dimensions. It can be shown that TVD is actually a 1-D special case of the LED concept.

It can be shown[61] that schemes built on obtaining information from the upwind part of a characteristic do satisfy positivity constraints (eqn.(3.7)) and are thus stable. This concept is used in the developing the numerical scheme used in this work.

3.4 Conservation Form

From a numerical solution perspective, it can be shown that the conservative formulation is necessary to accurately capture discontinuities. This can be seen even in a simple equation such as the Burger's equation (cf. ref.[40]):

$$\frac{\partial u}{\partial t} + \frac{\partial \left(u^2/2 \right)}{\partial x} = 0 \tag{3.8}$$

If eqn.(3.8) is solved using non-conservative schemes such as Lax-Friedrichs,

$$u_j^{n+1} = \frac{u_{j+1}^n + u_{j-1}^n}{2} - \Delta t \left[\frac{u_{j+1}^n + u_{j-1}^n}{2} \frac{u_{j+1}^n - u_{j-1}^n}{2\Delta x} \right], \tag{3.9}$$

or backward differencing,

$$u_j^{n+1} = u_j^n - \Delta t \left[u_j^n \frac{u_j^n - u_{j-1}^n}{\Delta x} \right], \tag{3.10}$$

it can be verified that both eqns. and give incorrect solutions of eqn.3.8.
However, a conservative numerical formulation of eqn.(3.8),

$$u_j^{n+1} = u_j^n - \frac{\Delta t}{2\Delta x} \left[(u_j^n)^2 - (u_{j-1}^n)^2 \right], \tag{3.11}$$

will give the correct answer. For further information on this issue, refer to LeVeque[40].

Most of the discussion on the numerical techniques has emphasized the hyperbolic nature of the convective part of the problem. This is because the goal of this work is to simulate problems in propulsion, consequently computing the flow is the most important part. Also, the convective problem is the harder one to solve numerically. The dissipative part of the problem, which is responsible for adding a parabolic nature to the governing equations, is relatively well understood. However, as explained in section 3.1, there is strong coupling between the hyperbolic and the parabolic part of the problem.

This coupling raises important issues in spatial as well as temporal discretization. The issue regarding the time scales is discussed in §3.5.2. In the case of spatial discretization, the issue is that schemes that are good for parabolic equations are not suited for hyperbolic equations. Consider a simple scalar diffusion equation,

$$\frac{\partial u}{\partial t} = \mu \frac{\partial^2 u}{\partial x^2}. \tag{3.12}$$

A standard scheme to solve this equation is the explicit central differencing scheme:

$$u_j^{n+1} = u_j^n + \Delta t \left[\frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{\Delta x^2} \right]. \tag{3.13}$$

However, it can be easily verified that for a simple scalar convection equation,

$$\frac{\partial u}{\partial t} + V \frac{\partial u}{\partial x} = 0, \tag{3.14}$$

the FTCS scheme, eqn.(3.13), does not satisfy the positivity condition (eqn.(3.7)) and even fails the VNSA.

On the other hand, a good scheme for solving the scalar convection equation is the explicit backward differencing scheme:

$$u_j^{n+1} = u_j^n - \Delta t \cdot V \left[\frac{u_j^n - u_{j-1}^n}{\Delta x} \right]. \tag{3.15}$$

Obviously, this scheme will not work for the diffusion equation because the numerical domain of dependence does not contain the physical domain of dependence (recall the CFL condition).

Therefore, it is clear that discretization of the convective and dissipative parts of the problem must be treated separately.

3.5 Hyperbolic (Convection) Equations

3.5.1 Spatial Discretization

The numerical solution to the set of hyperbolic equations (eqn.2.41 without the diffusion terms) is based on techniques that are extensively used in computational fluid dynamics. Based on the pioneering work of Godunov[62], [63], the principles underlying the design of non-oscillatory discretization schemes for compressible flows have been well established. There are two important issues in the design of discretization schemes:

- Estimating the numerical flux through cell boundaries, accounting for waves (discussed in appendix §B.1) traveling at different speeds, and possibly in different directions.

- Obtaining non-oscillatory solutions and capturing discontinuities with sufficient accuracy.

The scheme used in this thesis is developed in terms of local extremum diminishing (LED) principle of Jameson[61]. The method can be explained using eqn.(2.41) in one spatial dimension,

$$\frac{d\mathbf{U}_j}{dt} + \frac{\mathbf{Hz}_{j+1/2} - \mathbf{Hz}_{j-1/2}}{\Delta z} = 0, \tag{3.16}$$

where,$\mathbf{U}$ is the vector of conserved variables, and $\mathbf{Hz}$ is the approximation of flux in the $\hat{z}$ direction.

The true flux, obtained from eqn.(2.41), in the $\hat{z}$ direction can be split as,

$$\mathbf{Fz}(\mathbf{U}) = \mathbf{Fz}(\mathbf{U})^{+} + \mathbf{Fz}(\mathbf{U})^{-}, \tag{3.17}$$

where the eigenvalues of $d\mathbf{Fz}^{+}/d\mathbf{U}$ are all non-negative, and the eigenvalues of $d\mathbf{Fz}^{-}/d\mathbf{U}$ are all non-positive. Then, the approximation of flux is estimated as,

$$\mathbf{Hz}_{j+\frac{1}{2}} = \mathbf{Fz}_{j}^{+} + \mathbf{Fz}_{j+1}^{-}.$$

Using eqn.(3.17), this can be rewritten as,

$$\mathbf{Hz}_{j+\frac{1}{2}} = \frac{1}{2}\left(\mathbf{Fz}_j + \mathbf{Fz}_{j+1}\right) - \mathbf{Dz}_{j+\frac{1}{2}},$$

where

$$\mathbf{Dz}_{j+\frac{1}{2}} = \frac{1}{2}\left[\left\{\mathbf{Fz}_{j+1}^{+} - \mathbf{Fz}_{j}^{+}\right\} - \left\{\mathbf{Fz}_{j+1}^{-} - \mathbf{Fz}_{j}^{-}\right\}\right]. \tag{3.18}$$

There still remains a question of how $\mathbf{Fz}^{+}$ and $\mathbf{Fz}^{-}$ can be evaluated. This evaluation is possible if there is a matrix $\mathbf{A}$, such that

$$\Delta\mathbf{Fz}_{j+1/2} = \mathbf{A}\cdot\left(\Delta\mathbf{U}_{j+1/2}\right)\ , \tag{3.19}$$

where $\Delta\mathbf{Fz}_{j+1/2} = \mathbf{Fz}_{j+1} - \mathbf{Fz}_j$, and $\Delta\mathbf{U}_{j+1/2} = \mathbf{U}_{j+1} - \mathbf{U}_j$. Note that, in the case the points $j+1$ and j are on opposite sides of a discontinuity, eqn.(3.19) indicates that this scheme satisfies the Rankine-Hugoniot jump conditions exactly.

Since the ideal MHD equations are hyperbolic, they have real characteristics. Therefore, the characteristic directions or characteristic manifolds have important physical meaning, since all information propagates along them (as explained in refs. [64, 65]). Moreover, the eigenvectors of the Jacobian, **A**, are orthogonal and can be normalized. Therefore, the Jacobian can be diagonalized as:

$$\mathbf{A} \equiv \mathbf{R}\mathbf{\Lambda}\mathbf{R}^{-\mathbf{1}}, \tag{3.20}$$

where $\mathbf{R}$ contains the right eigenvectors of **A** as its columns, and $\mathbf{R}^{-1}$ contains the left eigenvectors of **A** as its rows. $\mathbf{\Lambda}$ is the diagonal matrix of eigenvalues of **A**. Since $\mathbf{\Lambda}$ can be easily split into,

$$\mathbf{\Lambda} = \mathbf{\Lambda}^{+} + \mathbf{\Lambda}^{-},$$

using eqn.(3.20), **A** can be split as,

$$\mathbf{A}^{\pm} \equiv \mathbf{R}\mathbf{\Lambda}^{\pm}\mathbf{R}^{-1}\ . \tag{3.21}$$

Thus if there exists an **A** such that eqn.(3.19) is true, then,

$$\Delta\mathbf{Fz}^{\pm}_{j+1/2} = \mathbf{A}^{\pm} \cdot \left(\Delta\mathbf{U}_{j+1/2}\right)\ . \tag{3.22}$$

Defining $|\mathbf{A}| = \mathbf{A}^{+} - \mathbf{A}^{-}$, eqn.(3.18) can be written as,

$$\mathbf{Dz}_{j+\frac{1}{2}} = \frac{1}{2}\,|\mathbf{A}| \cdot \Delta\mathbf{U}_{j+\frac{1}{2}}\ . \tag{3.23}$$

For the Euler equations, the matrix **A** was derived by Roe[66, 67]. However, this is not necessarily applicable to MHD equations. There have been efforts by Cargo[68] to derive such matrices for MHD equations. The literature[69] suggests that various forms of averaged matrices work satisfactorily.

From Godunov's theorem, it is evident that the scheme can only be first-order accurate, if it is to capture discontinuities. However, away from the discontinuities, the spatial accuracy of the scheme can be improved by including flux-limited anti-diffusion, **Lz**, described by Jameson[61]:

$$\mathbf{Dz}_{j+\frac{1}{2}} = \frac{1}{2}\left|\mathbf{A}\right|\left[\Delta\mathbf{U}_{j+\frac{1}{2}} - \mathbf{Lz}\left(\Delta\mathbf{U}_{j+\frac{3}{2}}, \Delta\mathbf{U}_{j-\frac{1}{2}}\right)\right]. \tag{3.24}$$

Essentially, this reduces numerical diffusion where it is not required.

Similar equations can be written for the corresponding terms in the $\hat{r}$ direction.

An alternative to characteristics-splitting for solving conservation form of the equations is to use artificial viscosity (scalar diffusion). In this formalism, the equivalent expression for eqn.(3.23) is,

$$\mathbf{Dz}_{j+\frac{1}{2}} = \frac{1}{2}\left|\lambda\right|_{max}\Delta\mathbf{U}_{j+\frac{1}{2}}. \tag{3.25}$$

Because of its low computational cost, scalar diffusion schemes such as eqn.(3.25) have been successfully adapted for industrial applications such as aircraft design. However, since these schemes tend to artificially smooth out the solution [61], eqn.(3.25) was only used in this work for comparison with eqn.(3.23).

3.5.2 Temporal Discretization

Unlike in fluid mechanics, the equations of MHD allow many different types of waves to exist. Even though physically the flow velocity is the sought quantity of most interest to propulsion, numerically the velocity of the fastest wave is what determines the time-step constraints. In plasmas of propulsion interest, the fluid velocity is $\mathcal{O}(10^4)$ m/s . For a quasineutral plasma with charge density of $\mathcal{O}(10^{21})/\mathrm{m}^3$ and thermodynamic pressures of $\mathcal{O}(10^{-1})$ Torr and magnetic pressure of $\mathcal{O}(10^1)$ Torr, the fast magnetosonic wave speed is typically of the same order of magnitude as the flow velocity. This indicates that an explicit time marching scheme is suitable. From the CFL criterion, the time step for such a problem would be $\mathcal{O}(10^{-8} - 10^{-9})$ s.

A multi-stage scheme can be chosen to march forward in time. Writing eqn.(2.41) as,

$$\frac{d\mathbf{U}}{dt} + \mathcal{F}\left(\mathbf{U}\right) = 0 \,, \tag{3.26}$$

where $\mathcal{F}\left(\mathbf{U}\right)$ represents the sum of all the fluxes, the multi-stage scheme can be written as:

$$\begin{aligned} \mathbf{U}^1 &= \mathbf{U}^n - \alpha_1 \Delta t \, \mathcal{F}\left(\mathbf{U}^n\right) \,, \\ \mathbf{U}^2 &= \mathbf{U}^n - \alpha_2 \Delta t \, \mathcal{F}\left(\mathbf{U}^1\right) \,, \\ \mathbf{U}^3 &= \mathbf{U}^n - \alpha_3 \Delta t \, \mathcal{F}\left(\mathbf{U}^2\right) \,, \\ \mathbf{U}^4 &= \mathbf{U}^n - \alpha_4 \Delta t \, \mathcal{F}\left(\mathbf{U}^3\right) \,, \\ \mathbf{U}^{n+1} &= \mathbf{U}^4 \,. \end{aligned} \tag{3.27}$$

The coefficients used in this work are $\alpha_1 = 0.1084, \alpha_2 = 0.2602, \alpha_3 = 0.5052, \alpha_4 = 1.0$, and were obtained from ref.[70].

3.5.3 Verification

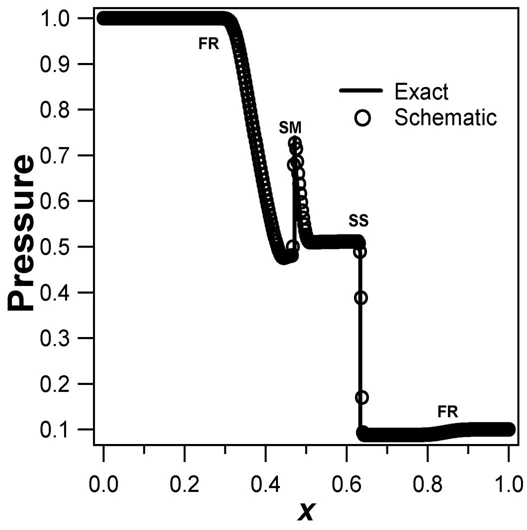
Unsteady Case

Riemann problem

The test problem chosen to validate this scheme was of the classical Riemann problem type, which consists of a single jump discontinuity in an otherwise smooth initial conditions. In 1-D the problem is:

$$\mathbf{U}\left(x,0\right) = \begin{cases} \mathbf{U}_L & \text{if } x < \frac{L}{2} \\ \mathbf{U}_R & \text{if } x \geq \frac{L}{2} \end{cases} \,. \tag{3.28}$$

The Riemann problem was chosen because it is one of the very few that have an analytical solution. This problem provides an excellent illustration of the wave nature of the equations. The solution to the Riemann problem is useful to verify the capturing of both smooth waves (characteristics) as well as non-smooth waves (shocks).



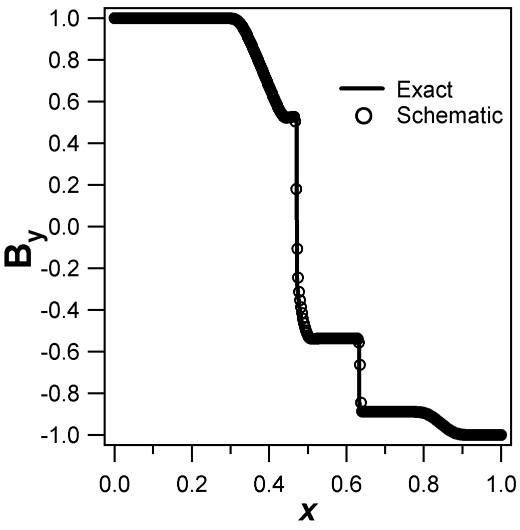


Figure 3.4: Comparison of calculated profiles of pressure and magnetic field, with exact solution

The initial states used were very similar to the Sod's problem[71] for Euler equations. They were:

$$\text{Left}:\begin{cases}\rho &= 1.0\\ V_x &= 0.0\\ V_y &= 0.0\\ V_z &= 0.0\\ B_x &= \frac{3}{4}\\ B_y &= 1.0\\ B_z &= 0.0\\ p &= 1.0\end{cases}\quad \text{Right}:\begin{cases}\rho &= \frac{1}{8}\\ V_x &= 0.0\\ V_y &= 0.0\\ V_z &= 0.0\\ B_x &= \frac{3}{4}\\ B_y &= -1.0\\ B_z &= 0.0\\ p &= \frac{1}{10}\end{cases}. \tag{3.29}$$

Some sample results are shown in Fig.(3.5.3).

In these figures, the fast rarefaction (FR) wave can be seen on the far right and the far left, as it is the fastest of the waves present in the problem. The slow shock (SS) and the compound wave (SM) have speeds less than that of the FR wave.

Steady-State Case

Taylor state

In order to simulate steady-state MHD flows, this solver can be used to solve the unsteady equations and marched to steady state. An important question is whether the solution remains in that steady state. To answer this question, a test problem was chosen, whose equilibrium solution is known analytically. This equilibrium solution is given as the initial condition for the solver. After marching several hundreds or thousands of time steps, a check is performed if the variables have changed from the initial conditions.

The test problem chosen for this simulation was the Taylor State configuration[72]. Under certain conditions, described in ref.[72], when a bounded plasma is allowed to evolve, it will move quickly and dissipate energy before coming to rest. This stable equilibrium configuration can be analytically found using the minimum energy principle, and is of the form:

$$\nabla \times \mathbf{B} = \lambda \mathbf{B}, \tag{3.30}$$

where λ is an eigenvalue.

Since the current is parallel to the magnetic field, the $\mathbf{j} \times \mathbf{B}$ body force is identically zero. Furthermore, if there are no thermodynamic pressure gradients, the plasma is in a state of force-free equilibrium. For an axisymmetric geometry, the resulting magnetic field profile is:

$$B_\theta = B_0 J_1(\lambda r);\ B_z = B_0 J_0(\lambda r), \tag{3.31}$$

where B_o is a constant amplitude, J_0 and J_1 are Bessel functions of the first kind, of orders 0 and 1 respectively.

For a Cartesian grid of dimensions $L_x \times L_z$, with symmetry along the $\hat{y}$ direction, the magnetic field distribution satisfying eqn.(3.30) is:

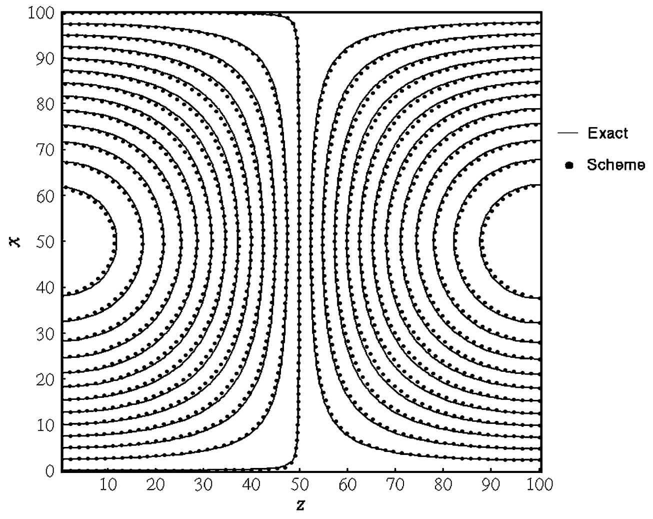


Figure 3.5: Magnetic field in the Taylor state configuration

$$\begin{aligned}
B_x &= -\tfrac{B_0}{\sqrt{2}}\ \sin\left(\tfrac{m\pi x}{L_x}\right)\cos\left(\tfrac{n\pi z}{L_z}\right),\\
B_y &= B_0\ \sin\left(\tfrac{m\pi x}{L_x}\right)\sin\left(\tfrac{n\pi z}{L_z}\right),\\
B_z &= \tfrac{B_0}{\sqrt{2}}\ \cos\left(\tfrac{m\pi x}{L_x}\right)\sin\left(\tfrac{n\pi z}{L_z}\right),
\end{aligned} \tag{3.32}$$

where m and n are eigenvalues.

With these initial conditions, the code was run for 10000 time steps on a 100×100 grid. At the end, the solution had deviated from equilibrium by less than 0.5%. The results from the code for B_x given in eqn.(3.32) are compared with the exact solution in Fig.(3.5).

3.6 Parabolic (Diffusion) Equations

3.6.1 Spatial Discretization

Numerical methods for parabolic equations are relatively commonplace. The parabolic terms can be written as,

$$\mathbf{S}_{dis} = \nabla \cdot \begin{bmatrix} \mathbf{0} \\ \bar{\bar{\tau}}_{vis} \\ \bar{\bar{E}}_{res} \\ \mathbf{q} \end{bmatrix} ,$$

where

$$\nabla \cdot \bar{\bar{\mathcal{T}}}_{vis} = \nabla \cdot \left[\mu_{vis} \nabla \mathbf{u}\right] , \tag{3.33}$$

represents the loss of momentum due to viscous forces,

$$\nabla \cdot \bar{\bar{E}}_{res} = -\nabla \times \left[\frac{\bar{\bar{\eta}} \cdot (\nabla \times \mathbf{B})}{\mu_o}\right] , \tag{3.34}$$

represents the resistive diffusion of the magnetic flux, including the Hall effect, and,

$$\nabla \cdot \mathbf{q} = \nabla \cdot \left[\{\bar{\bar{\tau}}_{vis} \cdot \mathbf{u}\} - \left\{\frac{\mathbf{E}' \times \mathbf{B}}{\mu_o}\right\} + \left\{\bar{\bar{k}}_{th} \cdot \nabla T\right\}\right] , \tag{3.35}$$

represents the energy sources/sinks due to viscous heating, Ohmic heating, and thermal conduction respectively.

Thus, the equations dictate that the numerical scheme should be second-order accurate in space. In the framework used here, this implies that the first derivatives of variables are to be known across cell faces. Therefore, a simple central-differencing scheme will be sufficient for this problem.

3.6.2 Temporal Discretization

Physical dissipation brings in different characteristic time scales into the problem. They are:

Viscous diffusion:	= $\rho\Delta r^2/\mu_{visc}$	$\sim 10^{-9}$ s
Magnetic diffusion:	= $\mu_o\Delta r^2/\eta$	$\sim 10^{-10} - 10^{-11}$ s
Heat conduction:	= $n_e k_B \Delta r^2/\kappa_{th}$	$\sim 10^{-9} - 10^{-11}$ s.

Since these time scales could reach the extremities of the ranges mentioned, the choice of the time stepping scheme can be made on a case-by-case basis. If these were vastly different, that would call for an implicit treatment of time stepping. If they are not vastly different, and an explicit *fractional time-stepping* scheme can be chosen.

Fractional Time Stepping

In the situations when the time scales are less than two orders of magnitude apart, it may not be worthwhile to choose an implicit scheme, however a standard explicit or multi-stage time-stepping scheme would be prohibitively expensive because it would require evaluating the convective fluxes at the time scales of dissipative fluxes. In order to find an optimum, a fractional time-stepping scheme can be chosen. In this method, the equation,

$$\frac{\partial \mathbf{U}}{\partial t} + \nabla \cdot \mathcal{F}_{\mathit{conv}} = \nabla \cdot \mathcal{F}_{\mathit{diff}}, \tag{3.36}$$

is marched forward at the dissipative time scale, Δt_d, dictated by the dissipative fluxes, $\mathcal{F}_{\mathit{diff}}$. However the convective fluxes, $\mathcal{F}_{\mathit{conv}}$, are evaluated only after *N* dissipative time steps. The number *N* is chosen such that the effective convective time step, $\Delta t_c = N \cdot \Delta t_d$, is still smaller then the convective time step dictated by the CFL condition. By reducing the time consuming operation of evaluation of the convective fluxes, the effective speed of the computation increases significantly. This is the current choice of temporal discretization currently used in this work.

Implicit Treatment

Depending upon the particular case being simulated, the convective and dissipative time scales could be far more disparate. In this situation, the reasonable choice would be to use an implicit time-stepping scheme. In such a situation, some standard method, such as Alternating Direction Implicit (ADI) (cf. ref. [64]) can be used to solve the diffusion equations, with the convective fluxes assumed to be a constant during that time scale. However, if this solver is to be implemented on a parallel computer, an ADI scheme will interfere will the effectiveness of the parallel computation process. Therefore, the pros and cons of this approach have to be investigated.

3.7 Summary

The fundamental concepts of stability and convergence of a numerical solution were reviewed in light of the techniques used in this work. The concept of local extremum diminishing (LED) schemes was introduced, and a new characteristics-splitting scheme, with flux-limited anti-diffusion to improve spatial accuracy was developed for the solution of the ideal MHD equations. This scheme was validated against unsteady (Riemann problem) and force-free equilibrium (Taylor state) test cases, and has demonstrated the ability the capture discontinuities monotonically, and good spatial accuracy in smooth regions of the solution. In conjunction with standard techniques for the numerical solution of parabolic equations, these schemes will be applied to simulate plasma flows in a real MPDT configuration in the following chapter.

Chapter 4

MPD THRUSTER SIMULATIONS

"Has anything escaped me? I trust that there is nothing of consequence which I have overlooked?"

Dr.Watson to Sherlock Holmes
The Hound of the Baskervilles

In this chapter, the numerical techniques described in chapter 3 will be applied to solve the governing equations of the physical model developed in chapter 2, to simulate plasma flows in a gas-fed, self-field MPDT with a constant area channel. The geometry of the thruster, and the relevant initial and boundary conditions for obtaining a solution are described in this chapter. The resulting profiles for many plasma parameters, obtained from the calculations, are compared with experimental data for the given operating conditions, but on a somewhat different geometry.

4.1 Application of Governing Equations

The conservation form of the MHD equations, given in eqn.(2.41), describe the evolution of eight variables, namely the total density (ρ), three components of momentum ($\rho u, \rho v, \rho w$), three components of magnetic field (B_r, B_θ, B_z), and the total energy ($\mathcal{E}$). However, in a self-field MPDT the magnetic field is purely azimuthal. Moreover, due to the assumption of axisymmetry, azimuthal momentum can be neglected. Though the solver described in §3.1 - §3.6 is for the generalized set of eight equations, it can be reduced for a system of five equations: the total density (ρ), two components of momentum ($\rho u, \rho w$), one component of magnetic field (B_θ), and the total energy ($\mathcal{E}$).

Expanding the vector-tensor form of eqn.(2.41) in cylindrical coordinates, using the identities eqns. (A.5), (A.6), and (A.7), along with the assumptions stated above, the MHD equations are:

$$\frac{\partial}{\partial t}\begin{bmatrix} \rho \\ \rho u \\ \rho w \\ B_\theta \\ \mathcal{E} \end{bmatrix} = \frac{\partial}{\partial r}\begin{bmatrix} -\rho u \\ -\left(\rho u^2 + p + \frac{B^2}{2\mu_o}\right) \\ -\rho u w \\ E'_z - uB_\theta \\ q_r - u\left(\mathcal{E} + p + \frac{B^2}{2\mu_o}\right) \end{bmatrix} + \frac{\partial}{\partial z}\begin{bmatrix} -\rho w \\ -\rho u w \\ -\left(\rho w^2 + p + \frac{B^2}{2\mu_o}\right) \\ -\left(E'_r + wB_\theta\right) \\ q_z - w\left(\mathcal{E} + p + \frac{B^2}{2\mu_o}\right) \end{bmatrix} + \mathbf{S}_r \quad (4.1)$$

with

$$\mathbf{S}_r = \frac{1}{r}\begin{bmatrix} -\rho u \\ -\left(\rho u^2 + \frac{B^2}{\mu_o}\right) \\ -\rho u w \\ 0 \\ q_r - u\left(\mathcal{E} + p + \frac{B^2}{2\mu_o}\right) \end{bmatrix} \qquad \begin{aligned} q_r &= \frac{E'_z B_\theta}{\mu_o} + \left(k_{th,e}\frac{\partial T_e}{\partial r}\right) + \left(k_{th,i}\frac{\partial T_h}{\partial r}\right) \\ q_z &= \frac{-E'_r B_\theta}{\mu_o} + \left(k_{th,e}\frac{\partial T_e}{\partial z}\right) + \left(k_{th,i}\frac{\partial T_h}{\partial z}\right) \end{aligned}$$

These equations are solved, in combination with species energy equations (eqn.(2.30) and/or eqn.(2.31)), at every time level, throughout a specified domain (to be described in §4.2), for a given set of initial and boundary conditions, to be described in §4.4 and §4.3 respectively.

4.2 Geometry

The geometry chosen for this simulation was one of the series of constant area coaxial thrusters, shown in Fig.(4.1), used by Villani[31]. The primary reason for the choice of this thruster was that its simple geometry was more amenable to the structured orthogonal grid generation routine that is currently being used. The computational domain is shown in Fig.(4.2). Due to the assumption of axisymmetry, only one half of the cross section, starting from the centerline (surface #1), had to be included in the simulation.

In this particular case, the cathode and the anode radii were 0.95 cm and 5.10 cm respectively. The cathode and the anode lengths were 26.4 cm and 20.0 cm respectively. Because of the inadequacy of the grid generation routine, the hemispherical tip of the cathode was represented as a flat tip (surface #8) in the simulation. For the same reason, the anode in the simulation has a sharp corner (interface of surfaces #4 and #5), instead of the rounded corner of the actual anode.

The location of propellant injection ports in the real thruster are shown in Fig.(4.1). Due to the complexity of the physical processes near the inlet region in MPDTs (to be discussed in §), the simulation has an ionized propellant entering the domain uniformly (surface #6).

The real thruster would operate either in the unbounded vacuum of space or in a bounded vacuum of a test chamber. However, the computational domain has to be restricted to something much smaller in size (surfaces #2 and #3). In this case, it is truncated at one cathode length downstream of the tip of the cathode (52.8 cm). Issues in this truncation are discussed in §4.3.

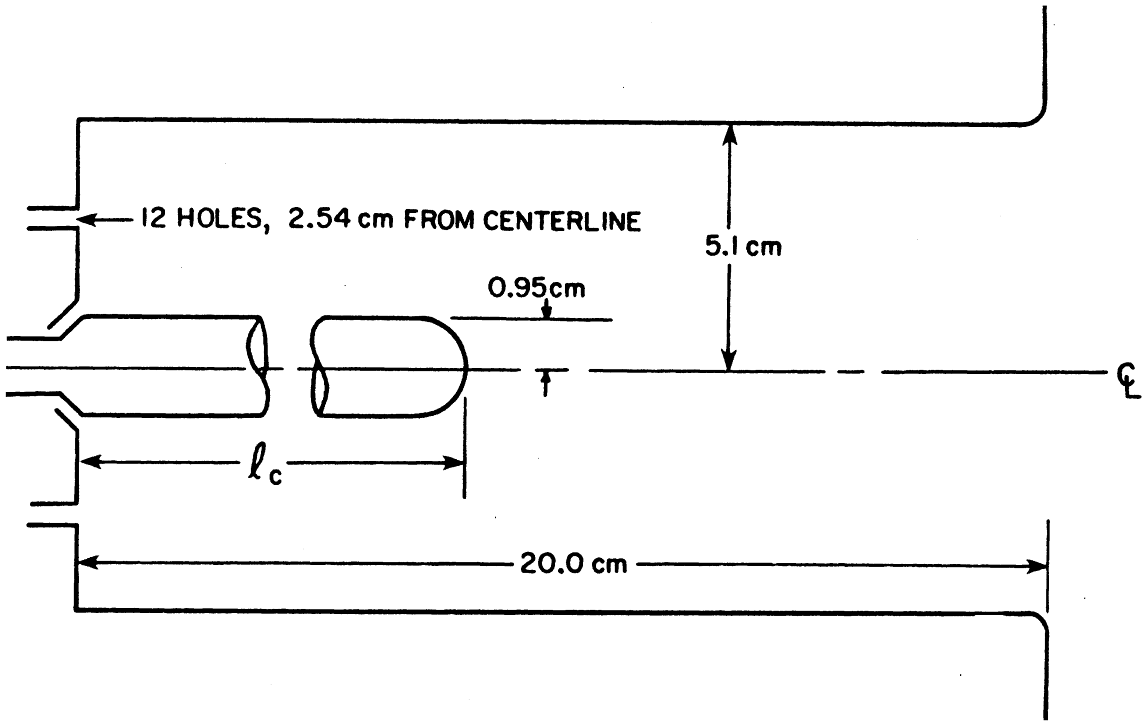


Figure 4.1: Geometry of the thruster chosen for simulation (l_c = 26.4 cm)

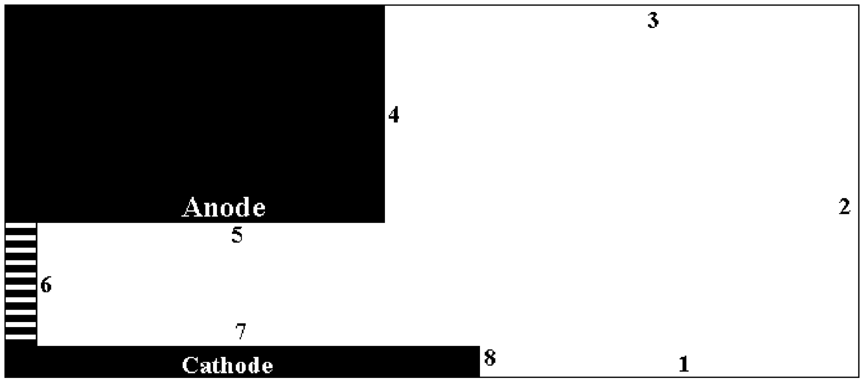


Figure 4.2: Schematic of the computational domain (not to scale)

4.3 Boundary Conditions

The set of governing equations (4.1) describe the evolution of many types of drastically different plasma flows. It is the role of the boundary conditions to distinguish one problem from another. For the particular MPDT simulation at hand, there are eight boundaries in the computational domain (cf. Fig.(4.2)), and they are of various types. This section will discuss the estimation of the convective and dissipative terms at each type of boundary.

4.3.1 Flow Properties

Freestream

The computational domain is assumed to be large enough such that there are no normal gradients in any of the flow properties at the free stream boundaries (surfaces #2 and #3).

Solid Walls

In reality, a sheath is formed at the interface of the plasma and the solid boundaries (surfaces #4, #5, #7, and #8). However, the sheath is a non-quasineutral region, and the fluid theory is not applicable there. Moreover, the size of the sheath region is typically of the order of a few Debye lengths, making it too small to be resolved by conventional grids. Therefore sheaths are currently ignored in this simulation. Discussion on inclusion of this effect will be made in §5.

At solid boundaries, all convective fluxes into the wall, given in eqn.(2.41), are zero. If viscous effects were included, they would require that the tangential component of velocity to be zero at the wall.

However, diffusive fluxes are nonzero. To calculate thermal conduction into a wall, the equations require that either the temperature of the wall, or the net heat flux to the wall itself, be specified. Ideally, the simulation would self-consistently and continuously compute the heat transfer from the plasma to the wall, then the heat transfer within the metallic conductor, then calculate the temperature distribution of the wall, and from that information recompute the heat transfer from the plasma to the wall. Clearly, these calculations are computationally expensive, and may be impractical. However, there isn't sufficient data to make good estimates of either of those quantities. Unlike the Li-LFA (described in §1.2, and in ref.[9]), high-power, inert gas, self-field MPDTs are generally operated in a quasi-steady mode, in which the pulse length are ≤ 2ms. The heat capacity of the plasma is typically not high enough to raise the temperature of the electrodes significantly in that duration, and there have been no known reliable measurements of heat transfer. In reality, there may be a thermal boundary layer in which the temperature varies from a few hundred Kelvin to more than an eV in a very small distance. Such sharp gradients may not be resolvable in simulations. Therefore a judicious estimate for heat transfer or wall temperature has to be made. The results in §4.7 were obtained with the temperature of the wall fixed at 2500 K.

Centerline

At the axis of symmetry (surface #1), there are no radial convective fluxes. Moreover, there are no radial gradients. Therefore, there is no thermal conduction across the centerline.

Inlet

At the inlet (surface #6), a specified mass flow rate of the propellant enters at a specified temperature at sonic conditions. In reality, the propellant is injected as neutral gas at room temperature, and it gets almost fully ionized within a few millimeters from the inlet[51]. However, it is believed[73] that this ionization process cannot be modeled by fluid theory. Therefore, in this model, the inlet temperature is chosen to be high enough such that the propellant is sufficiently ionized. Effectively, the backplate of the numerical model is not the true backplate, but a region few millimeters downstream of it.

4.3.2 Field Properties

Freestream

The computational domain is chosen to be large enough such that all the current is enclosed within the domain. Thus, from Ampère's law, the magnetic field at the free stream boundaries (surfaces #2 and #3) is zero. Note that if the domain is too small making this assumption unreasonable, the simulation will yield unphysical results. For instance, the gasdynamic pressure is obtained by subtracting the contributions of magnetic field energy and kinetic energy from the total energy (refer to §2.3.4). If the magnetic field is sharply reduced to zero at the freestream boundaries, then there will be a corresponding unphysical increase in the gasdynamic pressure. Therefore, the domain has to be large enough to make this assumption reasonable.

Solid Walls

At all other boundaries, the magnetic field is computed purely from Faraday's law. Using Stokes' theorem it can be written as,

$$\int_A \frac{\partial \mathbf{B}}{\partial t} \cdot d\mathbf{A} = -\oint_C \mathbf{E} \cdot d\mathbf{l}\,. \tag{4.2}$$

In the cell-centered scheme used in this work, eqn.(4.2) implies that the evolution of the magnetic flux is specified by the contour integral of electric field around the cell. Therefore, the only information required is the electric field drop along the boundaries.

From classical electromagnetic theory[74], the jump in the magnetic field, $\mathbf{H}_2 - \mathbf{H}_1$, across an interface between two media has to satisfy the relation,

$$\hat{\mathbf{n}} \times (\mathbf{H}_2 - \mathbf{H}_1) = \mathbf{J}_s\,, \tag{4.3}$$

where $\mathbf{J}_s$ is the surface current per unit length. Due to the no mass flux condition, the potential drop at a wall (surfaces #4, #5, #7, and #8) is entirely resistive, and is given by,

$$\mathbf{E}_w = \eta_w \mathbf{j}_w\,. \tag{4.4}$$

At conducting boundaries (surfaces #4, #5, #7, and #8), all the current entering the discharge flows at the surface, at least in the transient case. Therefore, even though resistivity, η_w, for most conductors is very small compared to the plasma resistivity,

$$\begin{aligned} \eta_{\text{plasma}} &\sim \mathcal{O}\left(10^{-3} - 10^{-4}\right) \text{ Ohm.m} , \\ \eta_{\text{copper}} &= 1.7 \times 10^{-8} \text{ Ohm.m} , \\ \eta_{\text{tungsten}} &= 5.6 \times 10^{-8} \text{ Ohm.m} , \end{aligned} \tag{4.5}$$

the surface electric field is significant, due to the large current density in the transient case. In a true steady state, after the magnetic field has diffused into the conductor, the surface potential drop decreases to zero.

At insulated boundaries, the magnetic field diffuses into the wall instantaneously. Therefore, the jump in the the magnetic field, and subsequently the surface current, is zero.

Centerline

At the axis of symmetry (surface #1), the inductive component of the electric field is zero because there is no flow across it. The resistive component can be related to the magnetic field from the point next to $r = 0$, through a simple Taylor series expansion,

$$E'_z|_{r=0} = \eta j_z|_{r=0} = \eta \frac{4\ B_\theta|_{\Delta r/2}}{\mu_o \Delta r} \ . \tag{4.6}$$

Inlet

At the backplate, which also serves as the inlet (surface #6), the total voltage drop is set as the boundary condition. Emulating a true constant current circuit, this applied voltage is adjusted every time step to maintain the specified amount of current to flow in the channel.

4.4 Initial Conditions

The governing equations (4.1) also require that the initial spatial distribution of the quantities be prescribed. The code is typically started with the entire domain filled with a back-ground pressure of 10^{-4} Torr at a temperature of 300 K.

Then, the inlet boundary conditions, corresponding to a fully ionized plasma entering at a specified mass flow rate, are imposed. After this plasma has filled the thrust chamber, the voltage at the backplate is made finite, introducing the effects of current and the magnetic field into the problem. For the calculations shown in §4.7, the current increased from 0 to 15 kA in $\sim 5\mu$s, and this rate is controlled by the adjustments to backplate voltage every time step.

Until this moment, the ionization is frozen to be Z=1, and the ratio of specific heats is fixed to be $\gamma = 5/3$. After the current has permeated throughout the thrust chamber, the effects of multi-level equilibrium ionization and non-ideal equation of state are slowly introduced. For instance, let γ^* be the calculated value of γ at a time level $n+1$, and γ^n be the old value at time level n. Then, the value of γ used at time level $n+1$ is:

$$\gamma^{n+1} = \{\alpha\gamma^*\} + \{(1-\alpha)\,\gamma^n\} \ , \tag{4.7}$$

where α is a relaxation parameter between 0 and 1. A similar method is used for introducing ionization effects. This sort of "relaxation" is required to make the transition from an unphysical initial condition to a more realistic scenario.

4.5 Schematic of Calculations

For a given set of initial and boundary conditions, the governing equations are solved using the techniques described in §3.1- §3.6. The entire process of simulation is illustrated in a flow chart in §4.5. The actual code, written in C/C++, is given in appendix §C.

The input, consisting of the mass flow rate, the total current, the geometry, the grid sizes, and the total time are handled by the program "InputOutput.CPP" (§C.4). The grid itself is generated by the program "Grid.CPP" (§C.5). The initial values for all the rel-

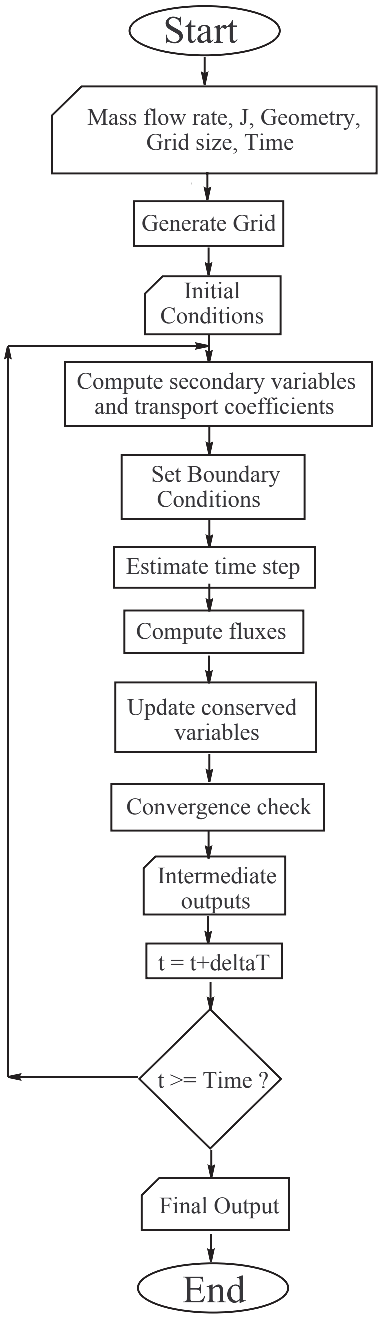


Figure 4.3: Schematic of the simulation process

evant variables are prescribed by the program “InitialConditions.CPP” (§C.6). The required secondary variables, and transport properties are computed in the program “RaCalculate.CPP” (§C.17). The boundary conditions are handled by “BoundaryConditions.CPP” (§C.7). The time steps is set by “SetTimeStep.CPP” (§C.9). The convective fluxes are computed in “EvalConv.CPP” (§C.10). The diffusive fluxes are computed in “EvalDiss.CPP” (§C.14. The conservative variables are updated for the new time level in “TimeMarch.CPP” (§C.15). At the same time, the species energies are computed in “Energy.CPP” (§C.16). At every time level, the convergence rates are calculated in the program “ReCalculate.CPP” (§C.17). The outputs are handled by “InputOutput.CPP”(§C.4).

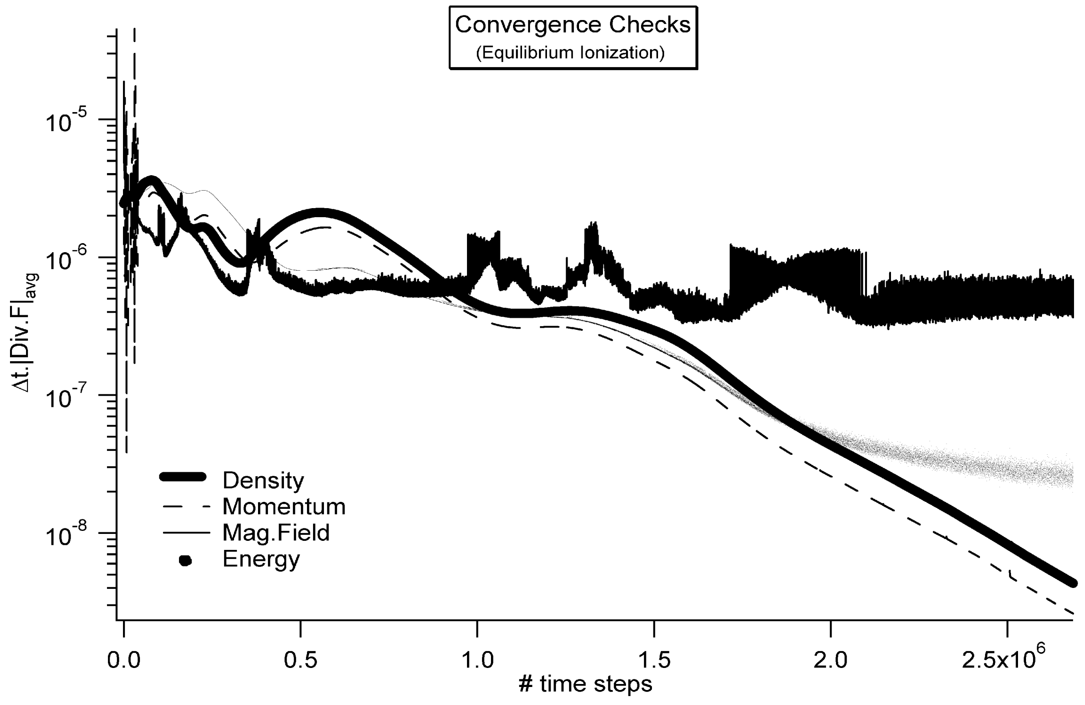
4.6 Convergence and Stability Checks

In order to verify convergence, there are two types of diagnostics in the code. In the first diagnostic, the change of conserved variables between every time level is calculated. In each case the largest *change* in the domain, and the average *change* over the entire domain are stored. Monitoring the maximum change helps check for stability. Monitoring the average change helps check for convergence. Since the entire set of equations is of the form,

$$\frac{d\mathbf{U}}{dt} + \nabla \cdot \mathcal{F} = 0 \; , \tag{4.8}$$

monitoring $\Delta t \cdot \nabla \cdot \mathcal{F}$ is the appropriate measure of change in **U**. The time history of the average change throughout the domain is shown in Fig.(4.4).

In the second diagnostic, the maximum *value* in the domain, and the average *value* over the domain of the conserved variables are stored. As before, monitoring the maximum value helps check for stability, and monitoring the average value helps check for convergence. The time history of the average value throughout the domain is shown in Fig.(4.4). From these plots, it was observed that convergence is reached only after $\sim 2.0 \times 10^6$ time



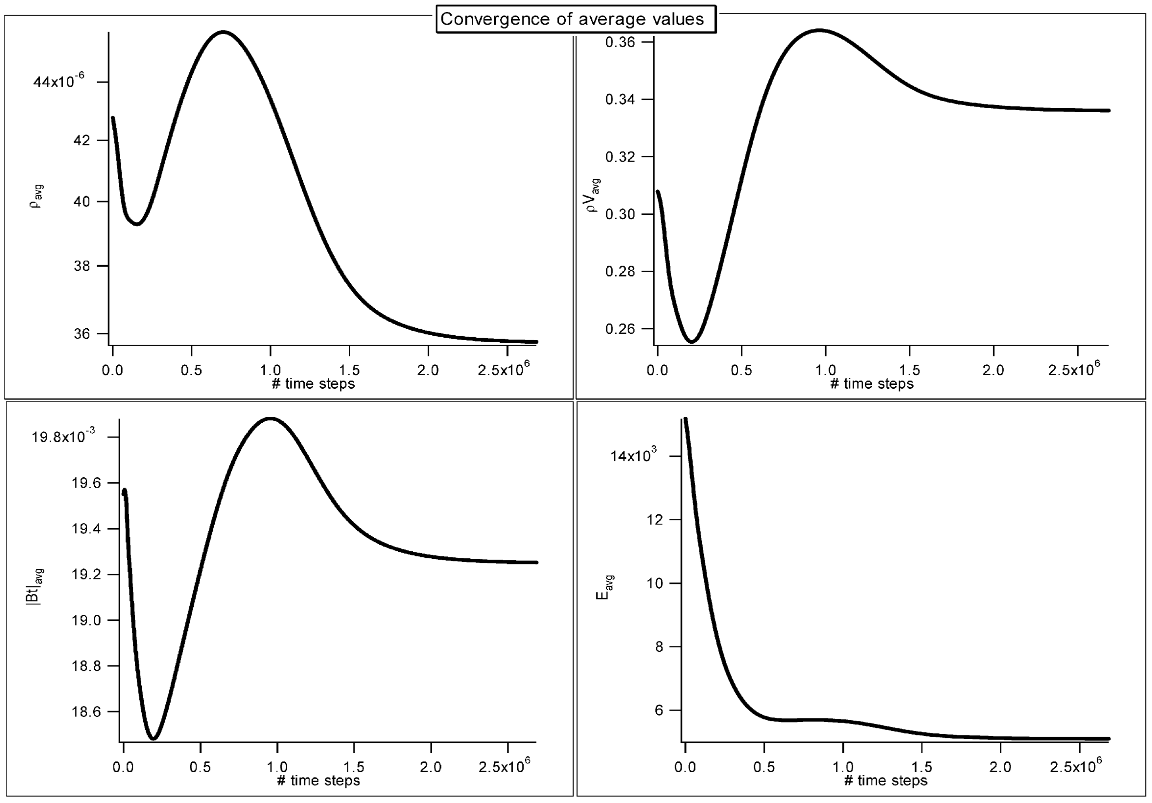


Figure 4.4: A. Convergence rates for the conserved variables; B. Domain averaged values of i) Density, ii) Axial momentum, iii) Magnetic field, and iv) Total energy at each time step

steps, which corresponds to $\sim 200\mu$s of physical time. The converged values of relevant variables are shown in §4.7.

4.7 Results

The results shown in this section are for the geometry in §4.2, with argon flowing in at 6.0 g/s and a discharge current of 15.0 kA. These conditions correspond to nominal operating conditions for the MPDT, since, from eqn.(2.7), $\xi \simeq 1.0$. There exists sufficient experimental data[31, 75] at these conditions, but for somewhat different geometries. The profiles of various relevant quantities are shown in this section.

4.7.1 Density

The electron number densities within the chamber range from $\simeq 1 \times 10^{21}/m^3$ near the anode region, to $\simeq 3 \times 10^{21}/m^3$ near the cathode (see Fig.(4.5)). This increase may be attributed to the radial pumping force, $j_z B_\theta$, which pushes the plasma away from the anode, towards the cathode. This trend has been observed in experiments and in previous simulations[19].

4.7.2 Ionization Levels

The effective ionization fraction is calculated to be,

$$Z_{eff} = \frac{n_e}{\sum\limits_{i=0}^{i=N} n_i} \, , \tag{4.9}$$

where n_e is the electron number density, and n_i is the density of the i^{th} ionized species. The resulting distribution is shown in Fig.(4.6). The presence of Ar-III and a small amount of Ar-IV in the plume is in agreement with experimental observations (cf. ref.[39]) for these

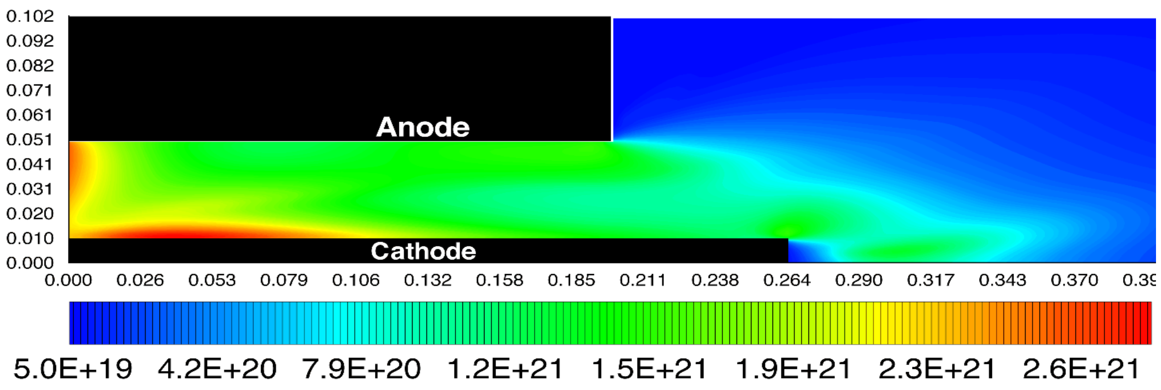


Figure 4.5: Distribution of electron number densities (in #/m^3)

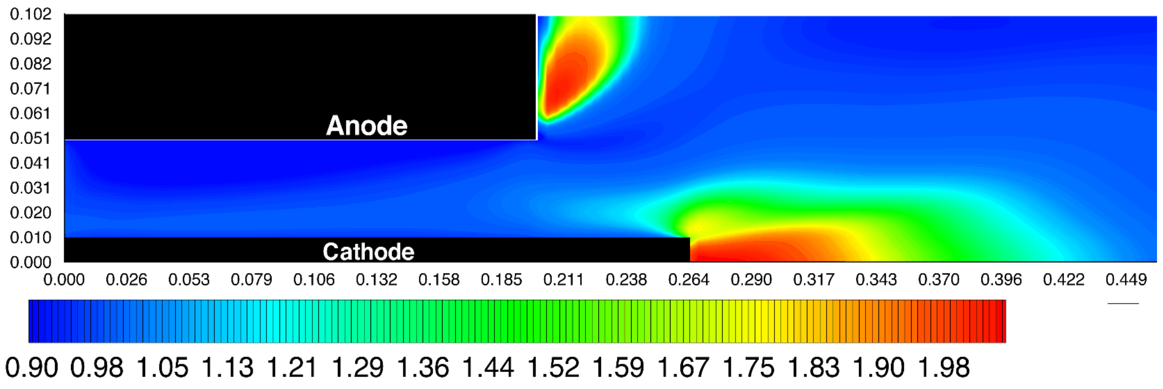


Figure 4.6: Distribution of ionization levels

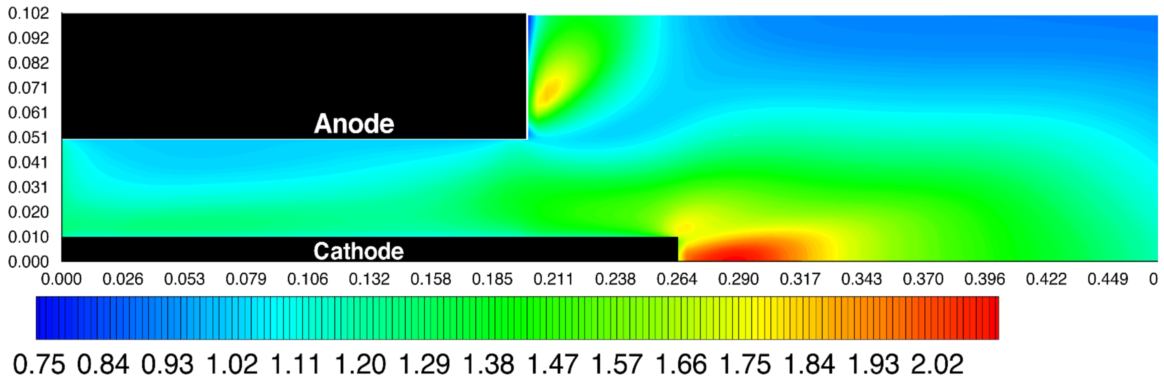


Figure 4.7: Distribution of electron temperatures (in eV)

operating conditions. From Fig.(4.6), it can be seen that the effective ionization fraction in the chamber is $Z_{eff} \simeq 1.0$. Since the model with frozen ionization ($Z_{eff} \equiv 1.0$) has been known to converge faster, one could use the single-level ionization model instead of the multi-level Saha model described in §2.3.7, to obtain rough estimates inside the thrust chamber.

4.7.3 Electron Temperatures

The distribution of electron temperatures (in eV) is shown in Fig.(4.7). Within the thrust chamber, T_e varies from about 1.0 to 1.5 eV. The lower values near the anode are probably due to the lower value of ηj^2 at higher *r*, and large heat transfer to the walls. Nevertheless, these numbers are in general agreement with measurement[75] at these operating conditions.

The hot spot at the tip of the anode, where T_e is about 2.0 eV, is probably due to the strong current attachment in that region, causing augmented values of Ohmic heating (ηj^2). The hot spot at the tip of the cathode may be due to stagnation, when the kinetic energy of the electron fluid is reduced and appears as thermal energy.

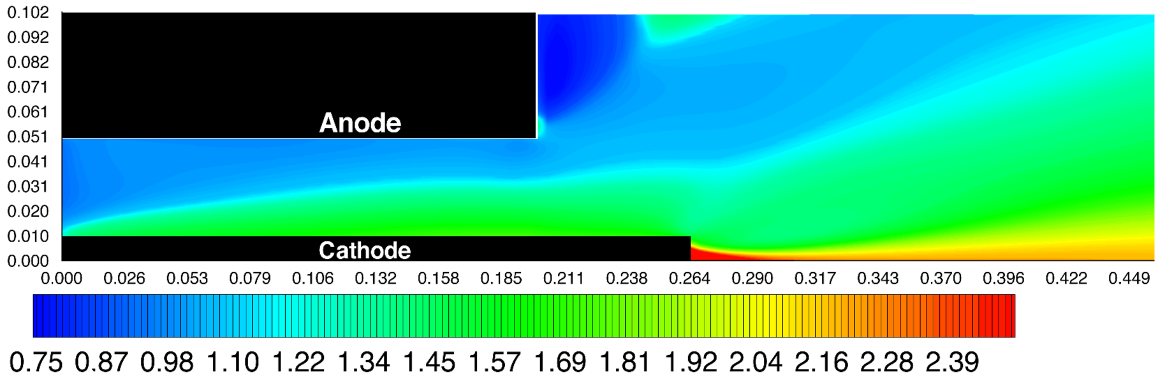


Figure 4.8: Distribution of ion temperatures (in eV)

4.7.4 Ion Temperatures

The distribution of ion temperatures (in eV) is shown in Fig.(4.8). Within the thrust chamber, T_h varies from about 1.0 to 1.5 eV. The hottest region of ion temperatures occurs at the axis of symmetry. At the center of the cathode tip (*r*=0), the ion temperature exceeds 4.0 eV. A partial explanation for this region of high temperature may once again be related to stagnation. Another possible explanation could be that the axisymmetric assumption causes thermal conduction at the centerline to go to zero. If there are symmetry breaking oscillations in reality, then there would be thermal conduction that would reduce the temperature in that region[76]. Nevertheless, these numbers are in general agreement with measurements[75] at these operating conditions.

4.7.5 Velocities

The distribution of axial velocities in the domain are shown in Fig.(4.9). At the anode plane, the axial velocity ranges from 8.0 km/s to 15.0 km/s. The maximum velocity increases to a maximum of 17.0 km/s slightly further downstream. As expected, the velocity increases with decreasing radius, because of the similar trend in $\mathbf{j} \times \mathbf{B}$. As mentioned in §2.2, the

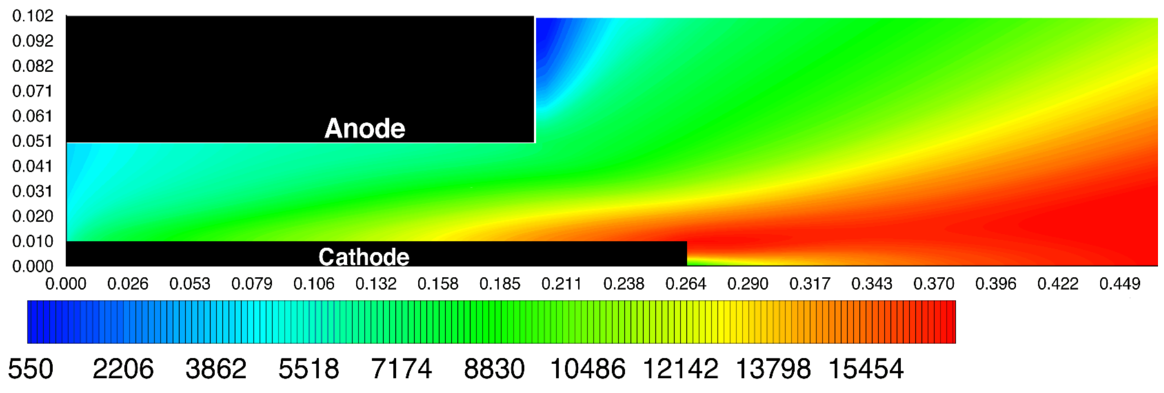


Figure 4.9: Distribution of axial velocities (in m/s)

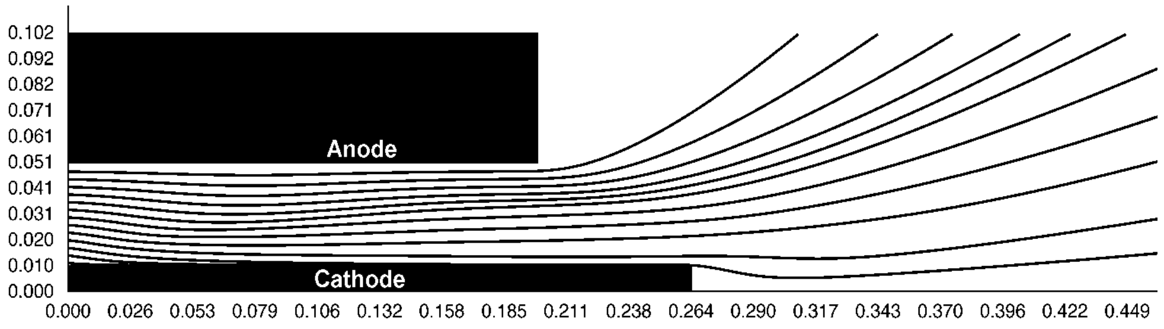


Figure 4.10: Velocity stream lines in the flow

thrust, consequently exhaust velocities, should be roughly independent of the geometry. As expected, these numbers from these simulations compare well with measurements of a different geometry in ref.[75].

To look at the effect of radial velocity on the flow, the velocity streamlines (where the velocity vector is a tangent at every point) are shown throughout the domain in Fig.(4.10).

4.7.6 Magnetic Field & Enclosed Current

The spatial distribution of magnetic field strength in the domain is shown in Fig.(4.11), as a fraction of the maximum value. The maximum value attained, 0.26 T, is at the intersection

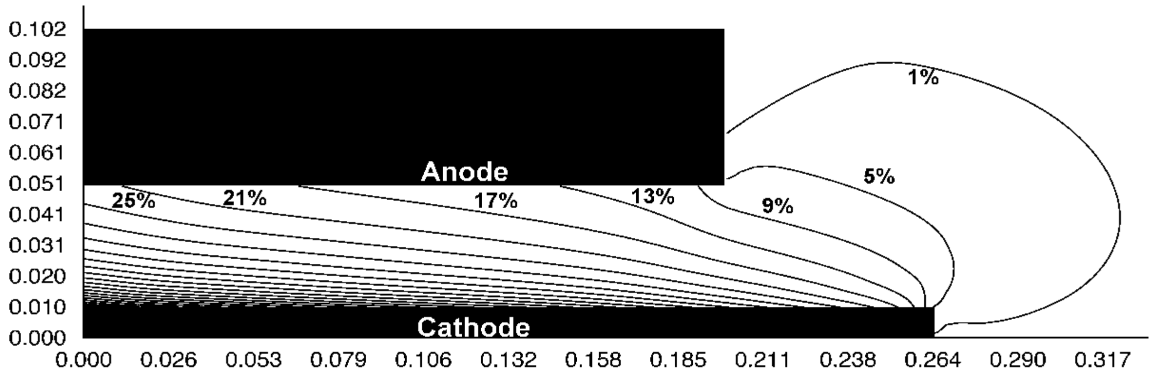


Figure 4.11: Distribution of magnetic field (as % of maximum)

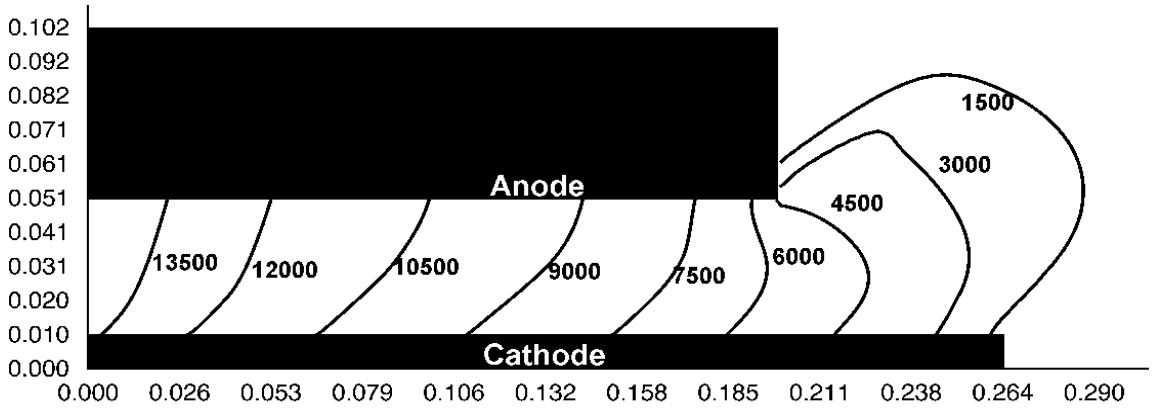


Figure 4.12: Enclosed current contours (1500 A between lines)

of the backplate and the cathode. Generally, B_θ varies as $1/r$ with radius and decreases linearly with axial distance. This is strictly true for a uniform current distribution in the channel. However, the current and magnetic field propagate downstream via convection and diffusion, and their distributions are no longer uniform.

The enclosed current is calculated as,

$$J_{encl} = \frac{2\pi r B_\theta}{\mu_o} \, . \tag{4.10}$$

The contours of enclosed current are shown in Fig.(4.12). Clearly, the current attachment is strongest near the backplate, and at the exit. This pattern is generally observed in many

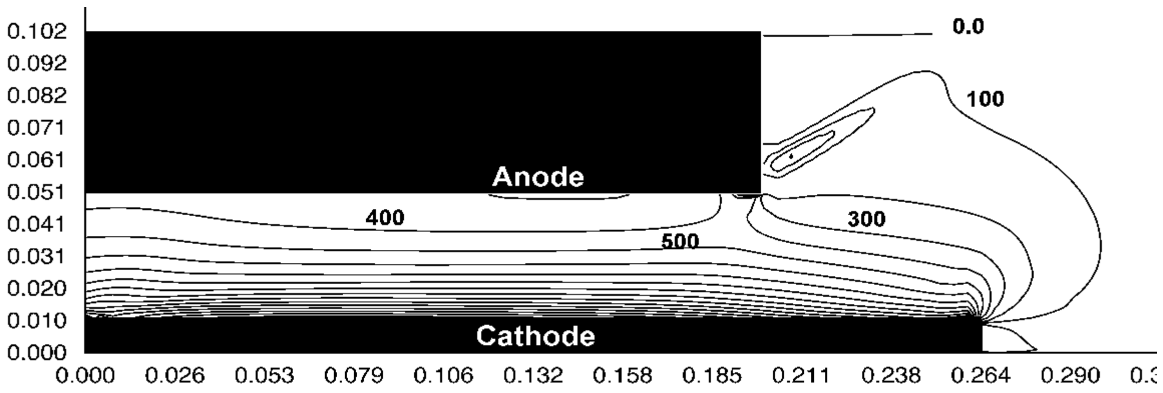


Figure 4.13: Radial electric field contours (in Volts/m)

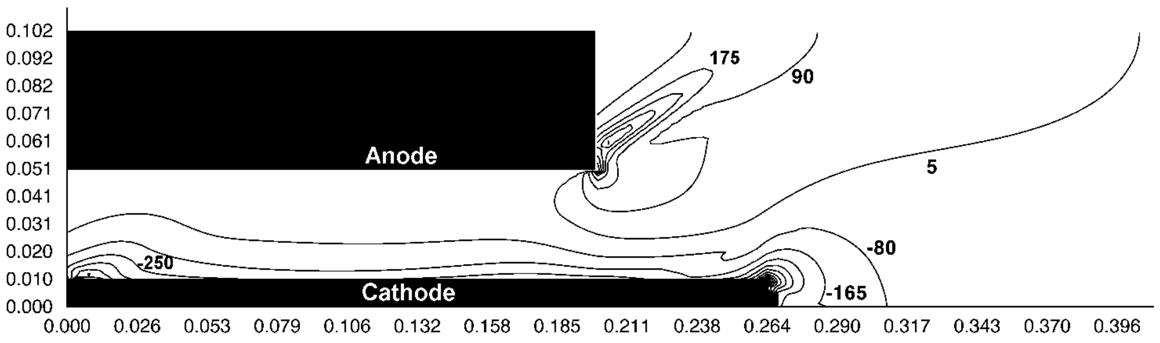


Figure 4.14: Axial electric field contours (in Volts/m)

MPDT geometries.

4.7.7 Electric Field & Potential

By definition, the relationship between a static electric field and its potential is,

$$\mathbf{E}\left(r,z\right)=-\nabla\phi\left(r,z\right)\ . \tag{4.11}$$

So, the potential difference between any two points can be calculated as,

$$\begin{aligned}\phi\left(r_2\right)-\phi\left(r_1\right)&=-\int_{r_1}^{r_2}\mathbf{E}\cdot d\mathbf{r}\ ,\\ \phi\left(z_2\right)-\phi\left(z_1\right)&=-\int_{z_1}^{z_2}\mathbf{E}\cdot d\mathbf{z}\ .\end{aligned} \tag{4.12}$$

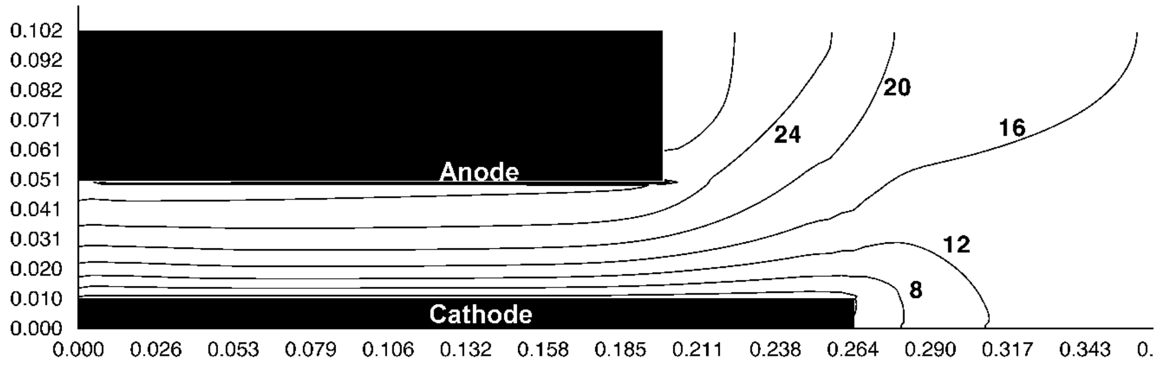


Figure 4.15: Potential contours (in Volts)

In this calculation, the cathode was set at a reference potential of 0. The potential at every other point in the domain was computed using eqn.(4.12). It is important to note that the predicted values of voltage do not include electrode drops, and are therefore cannot be compared to the measured value across the electrodes. Nevertheless, they do serve the purpose of quantifying the plasma part of the voltage drop. This simulation predicts a voltage drop across the plasma of 30.83 Volts. The true voltage drop is 56 Volts [31], and the difference can be attributed to the 25 Volts of anode drop that was measured.

4.7.8 Hall Parameter & Anomalous Transport

The distribution of the electron Hall parameter in the domain is shown in Fig.(4.16). Within the channel, it varies from 0.3 to 1.2. Since the magnetic field and species temperatures increase with decreasing radius, so does the Hall parameter. Outside the channel, the Hall parameter is higher at the electrode tips than in any other region. Once again, referring to Fig.(4.11) and Fig.(4.7), the magnetic field is higher (cyclotron frequency is higher) and temperatures are higher (collision frequencies are lower) in those regions.

It maybe worthwhile to note that these values of the electron Hall parameter are substantially lower than those observed in other MPDT configurations, such as the FSBT men-

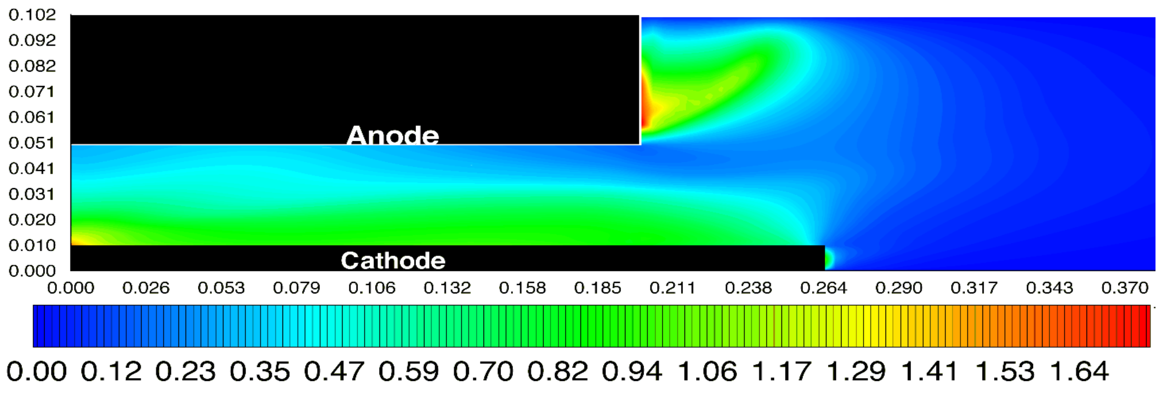


Figure 4.16: Distribution of the Hall parameter

tioned in §2.2. This is because, unlike in many MPDT configurations (e.g. ref.[75]) in which the anode is only a "lip" at the exit, the anode here is along the entire length of the channel, leading to a more diffuse current attachment pattern. Moreover, the propellant injection in this simulation is more uniform than in reality. This precludes regions of low densities and collisionality in the thrust chamber.

As seen in Fig.(2.3), the ratio of anomalous resistivity to classical resistivity (eqn.2.38) is a strong function of the electron Hall parameter, beyond a cutoff, $u_{de}/v_{ti} \geq 1.5$. The ratio of anomalous resistivity to classical resistivity for this case is shown in Fig.(4.17). As expected for this value of $J^2/\dot{m}$, the Hall parameter is not very large in this particular case, and therefore the overall effect of anomalous transport is limited (cf. Fig.(2.3)).

4.7.9 Thrust & Efficiency

By definition, the thrust is computed using the following relation,

$$T = \int_A u_z \left(\rho \mathbf{u} \cdot d\mathbf{A}\right) \ , \tag{4.13}$$

where the integral is performed over all the boundaries in Fig.(4.2).

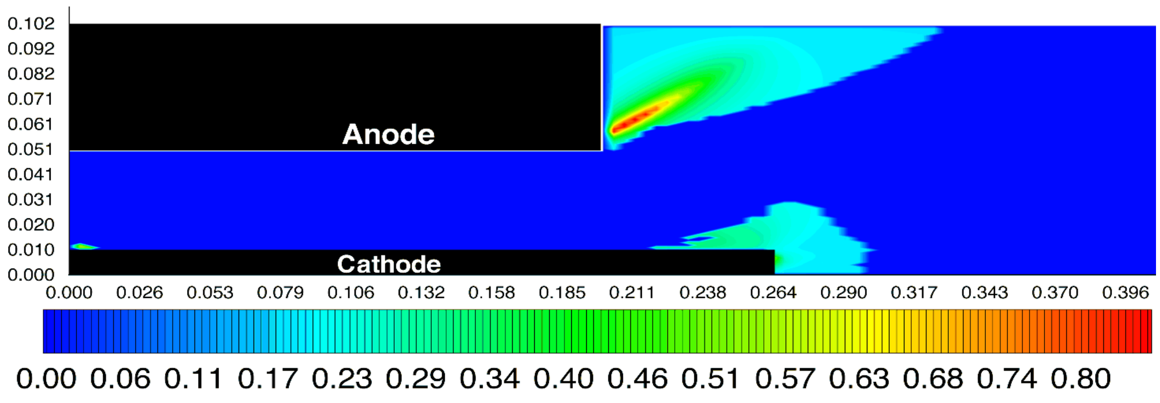


Figure 4.17: Ratio of anomalous to classical resistivity

As discussed in §2.2, the analytical expression for thrust is,

$$T = \frac{\mu_o}{4\pi}\left(\ln\frac{r_a}{r_c} + A\right) J^2, \tag{4.14}$$

where r_a and r_c are the radii of the anode and the cathode respectively. For this particular case, the appropriate value for the current attachment parameter, A, is 0.15 (refer to Villani[31]). Notice that A is much smaller than $\ln\frac{r_a}{r_c}$ (which is 1.68).

Using eqn.(4.13), the code predicts a thrust of 42.9 N. This compares well with the analytically calculated value is 41.2 N.

By definition, the thrust efficiency is,

$$\eta_{th} = \frac{\frac{T^2}{2\dot{m}}}{VJ}, \tag{4.15}$$

where *V* is the total voltage drop, 30.83 Volts, and *J* is the total current, 15.0 kA. This yields a thrust efficiency of 33.2%. Clearly, this value is higher than the real thrust efficiency because voltage drops due to electrode falls are neglected. As mentioned earlier, the true voltage drop is 56 Volts, resulting in a thrust efficiency of 18.3%.

4.8 Summary

Due to the limited capabilities of the grid generation routine used in this work, a simple constant area geometry was chosen for simulation. The drawback of this choice was the dearth of data available for this geometry. In fact, the only available data were the current and potential contours.

The measured and calculated current contours are compared in Fig.(4.18). It is evident that in the simulation the current has not propagated as far downstream as in the experiment. A possible explanation is that the values of resistivity in the channel in the simulation, are higher than in reality. Though the computed values of electron temperatures are generally in the correct range, resistivity is a strong function of temperature ($\sim T_e^{-3/2}$), and without experimental data on the T_e distribution, it is impossible to say if the resistivity values are correct. Increased values of T_e, which will yield reduced values of η will allow for the current lines to propagate further downstream. This may be related to the unresolved issue of temperature boundary conditions, mentioned in §4.3.1.

A direct comparison between measured and calculated potential contours cannot be made, because the calculations do not include electrode drops. Since the experimental measurements indicate that the anode drop is 25 V, the calculated potential contours were shifted by 25 V. Though this value is not a constant along the anode, it nevertheless allows comparison of calculations with measurements. These contours are shown in Fig.(4.19). It can be seen that the region in space that corresponds to the 30 V contour in the experiment, roughly corresponds to the 33 V contour in the simulation. Similarly, the region in space that corresponds to the 40 V contour in experiment, roughly corresponds to the 45 V contour in calculations. Again, it is important to bear in mind that the calculated contours were shifted by an estimate for anode drop. The unadjusted calculated contours are in Fig.(4.15).

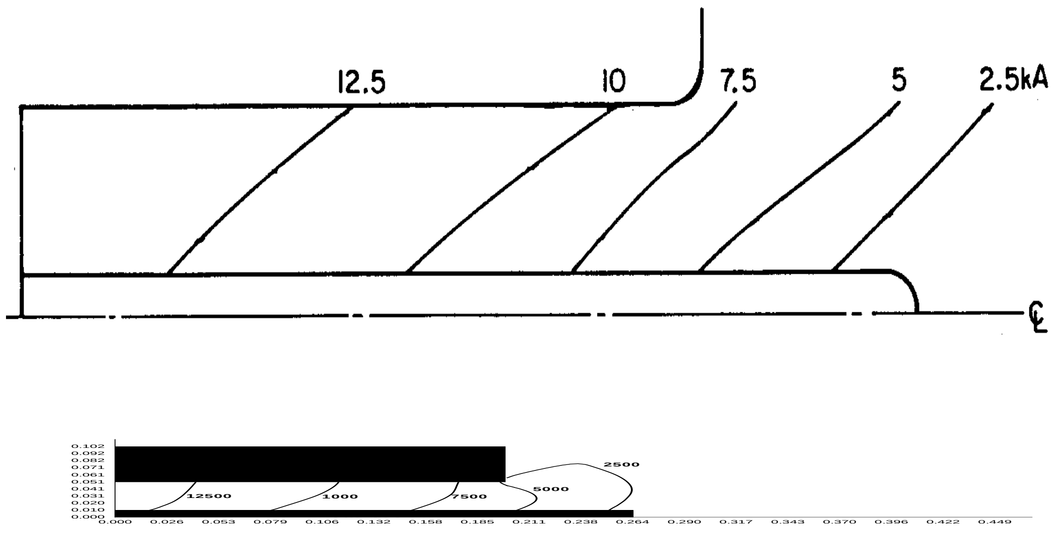


Figure 4.18: Measured (top) and calculated (bottom) current contours

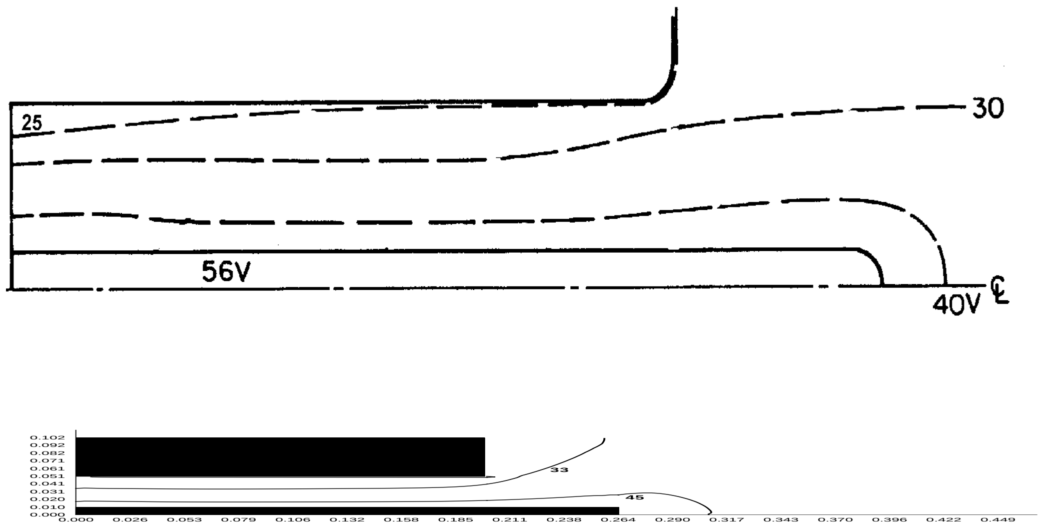


Figure 4.19: Measured (top) and calculated (bottom) potential contours

As far as the other variables are concerned, the only comment that can be made is that their values and distributions are within a reasonable range of values observed in experiments on similar geometries for these operating conditions.

Anomalous transport was found to have no major effect on the solution at these conditions. This is to be expected, since at this modest value of $J^2/\dot{m}$ the electron Hall parameters are not very high for this geometry (cf. Fig.(2.3)).

Though other researchers[24] have found the ideal equation of state suitable to simulate MPDTs at lower current levels, for the chosen operating condition, $J = 15.0$kA, and $\dot{m} = 6.0$g/s, there were insufficient energy sinks with the ideal equation of state, and the simulation did not converge. A real equation of state was crucial to obtaining a realistic and stable solution.

This chapter illustrated the application of the numerical techniques developed in this thesis to solve the governing equations of the physical model relevant to the simulation of plasma flows in a gas-fed, self-field MPDT functioning at nominal operating conditions. The calculated profiles for many relevant plasma parameters, and the calculated contours of current and electric potential, were compared with experimental data for the given operating conditions, but on somewhat different MPDT geometry. The calculated value of thrust was found to be in excellent agreement with the Maecker's law (eqn.(2.6)).

Chapter 5

CONCLUDING REMARKS

. . . I have promises to keep,
And miles to go before I sleep,
And miles to go before I sleep.
Robert Frost

5.1 What are the contributions of this thesis?

The objective of this thesis was to develop a numerical simulation model for MPD flows, that incorporates the state-of-the-art in numerical methods and physical models.

For this purpose, a new numerical scheme to accurately compute plasma flows of interest to propulsion was developed and validated against standard test problems. The scheme treats the flow and the field in a self-consistent manner, and conserves mass, momentum, magnetic flux and energy. The characteristics-splitting scheme, which was developed from concepts used for the solution of Euler equations, was used to solve the ideal MHD equations. The ability of this scheme to capture discontinuities monotonically was demonstrated. Flux-limited anti-diffusion was used to improve spatial accuracy away from discontinuities.

On top on the ideal MHD model, relevant diffusive effects, namely resistivity with Hall effect and gradient drifts, and electron and ion thermal conduction were included. Furthermore, effects of thermal non-equilibrium between electrons and ions were incorporated. Anomalous transport effects, which account for momentum transfer between waves and particles in finite beta MPD flows, were included in the transport models. A real equation of state model, to account for energy deposition into internal modes, was included and was found to have significant effect on producing realistic and stable solutions. A multi-stage equilibrium ionization model was used to obtain ionization levels and species densities.

This solver was then used to simulate a constant area coaxial MPDT, at nominal operating conditions corresponding to nominal operating conditions ($\xi \simeq 1.0$). The resulting profiles of densities, velocities, species temperatures, along with current and potential contours, when compared with existing data on MPDTs operating at this condition, but on somewhat different geometry, were found to be realistic. The value of thrust predicted by the simulation was in excellent agreement with predictions from analytical models.

With that, the goal stated earlier has been attained. With the inclusion of anomalous transport effects and a real equation of state, the capabilities of the physical model exceed that of some of the most persistent efforts at other institutions to simulate MPD flows. There is no documentation of any existing code that was validated against standard MHD test cases. Furthermore, unlike any other existing MPDT flow simulation codes, this one uses modern numerical methods for the solution of the governing equations.

Though this solver has demonstrated the ability to simulate MPD flows, it still requires development in both physical modeling and numerical capabilities. These are outlined in the subsequent section.

5.2 What remains to be done?

5.2.1 Computational Methods

In order for this code to be a more practical tool for MPDT research, it requires improvements in the computational methods. First and foremost is the need to accelerate convergence, while maintaining temporal accuracy. Currently, with the explicit time-stepping scheme that is being used (though a sub-stepping scheme is sometimes applicable) it takes somewhere between many days to two weeks (on a Pentium-II 450MHz) for the solution to converge. Clearly, this hampers productivity and precludes the ability to test various cases. Therefore, the immediate next step in this work is to accelerate convergence by modifying the code to work efficiently on parallel computers. Though the time-step limiting diffusive fluxes can be treated implicitly, this will interfere will the effectiveness of the parallel computation process. The pros and cons of this approach have to be investigated.

Perhaps the biggest limitation faced so far by the code is its structured orthogonal grid generation routine. This has precluded further validation of the code with existing experimental data, because of the limitations on the geometries that can be allowed by the simulation. Therefore, it is of utmost importance to develop a better grid generation routine. However, it is important to note that the numerical schemes developed in this thesis will not require any fundamental changes to be applied on grids of type 2 (Fig.(3.2)). Structured, curvilinear, body-fitted grids (Fig.(3.2)) are probably the best choice, and they will allow the code to simulate almost all known MPDT geometries.

After the implementation of this grid generation routine, the code will be used to simulate the Princeton Configuration 'A' thruster [75]. Since there is a plethora of data available on this geometry, it will serve as a good validation for the code.

5.2.2 Physical Models

The list of possible improvements to the physical model can be extensive. Among these, a few are discussed in this section, as they are believed to be most significant to obtaining realistic and reliable simulations.

In order to bolster its reliability, this code has to be used to simulate MPDTs operating at higher values of $J^2/\dot{m}$, for various geometries. In the case shown in §4.7, the effect of anomalous transport was limited because of the low values of electron Hall parameter. At higher values of $J^2/\dot{m}$, the effects of anomalous transport will be more pronounced.

Some aspects of the physical models have been neglected so far, because their effects were assumed to be insignificant. First among these is viscosity. Given good physical models for the coefficient of viscosity, this effect can be added into the code without much difficulty. Another effect, which can be also added without any modifications, is ion-neutral slip. It was argued earlier, in §2.3.2 and §2.3.3, that these two effects were small. However, Niewood[19] attributes the high observed ion temperatures to viscous heating, and to ion-neutral slip. Considering the ease with which these effects can be introduced, they will be incorporated soon into the code to test these propositions.

While most of the boundary conditions, such as the no flux conditions, and electromagnetic jump conditions across interfaces, are self-explanatory, intuitively obvious, and are commonly used, others are not. As discussed in §4.3.1, the temperature boundary condition on the walls is not prescribed self consistently. The effect of changing the wall temperature, and/or modifying the heat flux condition has to be investigated thoroughly. Also, as discussed in §4.3.2, the size of the computational domain dictates whether the $B_\theta = 0$ condition at the freestream boundary is indeed realistic. It may be worthwhile to investigate the boundary condition on the centerline. As discussed in §4.7, in reality there may be some symmetry breaking oscillations at the centerline. Due to the axisymmetry assumption used here, they are not accounted for. It is clear from the simulations that the species temperatures are high near this region. There may be a correlation between these two effects, which has to be investigated. As briefly mentioned in §4.3.1, there are two other issues that remain to be explored in the application of boundary conditions: sheaths, and inlet ionization.

In reality, there exists a non-quasineutral sheath at the interface of a plasma and a solid boundary. For the plasma conditions of interest, the typical dimension of this quasineutral region, in which a large fraction of the voltage drop occurs, is $\sim \mathcal{O}(10^{-5})$m. Clearly, this cannot be self-consistently computed because: i) the physical processes in this region are beyond the regime of MHD theory, and ii) the dimensions are too small to be captured by grids. Therefore, the only way to include this effect is to superimpose a sheath model as a boundary condition. Though an attempt was made by Boie *et al.*[23], it was later discontinued. Ideally, this model will incorporate the effect of thermionic emission, and possible effects of magnetic field, on a high-voltage sheath in an unsteady plasma.

As mentioned earlier, the propellant to a MPDT is injected as neutral gas at room temperature, and it gets almost fully ionized within a few millimeters from the inlet[51]. Choueiri and Okuda[73] have performed further investigations on anomalous ionization in MPDTs, and have shown that the most likely explanation is due to non-Maxwellian distribution of electron energies due to the effect of microinstabilities. It is believed that this ionization process cannot be modeled by fluid theory. This work currently employs a multi-stage equilibrium model. However, there is sufficient evidence[51, 20] that the equilibrium model is too simplistic. Niewood[19] used a finite-rate ionization model developed by Sheppard[20], and Caldo[15] has used another finite-rate ionization model. However, these finite-rate models do not predict higher levels of ionization, which have been observed experimentally [39, 51]. As mentioned earlier, it is important to allow for the presence of

Ar-III and other highly ionized argon species in the model. Thus, it is clear that there is still a lot of room to develop a good model to describe ionization processes in MPDTs, and implement them in this code.

The MPDT research at Princeton is focussed on the Lithium Lorentz Force Accelerator (Li-LFA), a variant of the conventional MPDT. In order to model this thruster, a real equation of state model for lithium has to be developed and implemented. Due to the lack of megawatt level testing facilities, many MPDTs, including Li-LFA, are sometimes operated with an applied magnetic field, to increase their thrust efficiency. Though the solver was originally developed with all components of magnetic field, it was later trimmed down to have only the azimuthal component of the magnetic field, to simulate self-field MPDTs in coaxial geometry. Therefore, for simulating applied-field MPDTs the code has to be modified to handle the effects of applied magnetic fields.

Since this code solves the governing equations in a time-dependent manner, in principle it can be extended to the simulation of pulsed plasma thrusters (PPT) as well. Without delving into the physical processes in PPTs, it is clear that the most important additions to the code for this purpose should be a finite-rate ionization model, and a model for circuit equation.

It is possible that this code could also play a role in guiding MPDT research issues such as thermal modeling, near-cathode plasma characterization and electrode thermal management to understand erosion processes, propellant selection, active turbulence control, near-field plume model as a source for far-field plume simulations that study the role of plasma interactions and contamination of spacecraft, and other contentious issues in overall design optimization.

In conclusion, it is important to note that the numerical solution techniques and many of the physical models discussed in this thesis can be applied to simulate plasma flows in other devices such as railguns[77], plasma flow switch ciruit breakers [78], Z-pinch devices for fusion[79] and the Variable Specific Impulse Magnetoplasma Rocket (VASIMR) [80, 81].

Bibliography

[1] K.E. Tsiolkovsky. Exploration of the Universe with Reaction Machines. *The Science Review*, 5, 1903. English translation from the website of *Tsiolkovsky State Museum of the History of Cosmonautics*.

[2] R.G. Jahn. *Physics of Electric Propulsion*. McGraw-Hill, 1968.

[3] A. C. Ducati, G. M. Giannini, and E. Muehlberger. Experimental results in high-specific impulse thermo-ionic acceleration. *AIAA J.*, 2(8):1452–1454, 1964.

[4] G. P. Sutton. *Rocket Propulsion Elements*. John Wiley, 1992.

[5] R. G. Jahn and E. Y. Choueiri. Electric propulsion. *Academic Press Encyclopedia of Physical Science & Technology*, 2000.

[6] E. Y. Choueiri, A. J. Kelly, and R. G. Jahn. Mass savings domain of plasma propulsion for LEO to GEO transfer. *J. Spacecraft & Rockets*, 1993.

[7] J. K. Ziemer. *Performance Scaling of Gas-Fed Pulsed Plasma Thrusters*. PhD thesis, Princeton U., 2000.

[8] K. Toki and K. Kuriki. On-orbit demonstration of a pulsed self-field magnetoplasma-dynamic thruster system. *J. Prop. Power*, 16(5):880–886, 2000.

[9] V. Kim, V. Tikhonov, and S. Semenikhin. Fourth quarterly (final) report to NASA-JPL: 100-150 kw lithium thruster research. *Technical Report NASW-4851*, 1997.

[10] J. E. Polk. *Mechanisms of Cathode Erosion in Plasma Thrusters*. PhD thesis, Princeton U., 1995.

[11] E. Y. Choueiri. *Advanced Problems in Plasma Propulsion*. Princeton U. Lecture Notes, 1998.

[12] K. Toki I. Kimura and M. Tanaka. Current distribution on the electrodes of MPD arcjets. *AIAA J.*, 20(7):889, 1982.

[13] T.Ao and T. Fujiwara. Numerical and experimental study of an MPD thruster. *IEPC-84-08*, 1984.

[14] T. Miyasaka and T. Fujiwara. Numerical prediction of onset phenomenon in a 2-dimensional axisymmetric MPD thruster. *AIAA-99-2432*, 1999.

[15] G. Caldo. Numerical simulation of MPD thruster flows with anomalous transport. Master's thesis, Princeton University, 1994.

[16] E.Y. Choueiri. Anomalous resistivity and heating in current-driven plasma thrusters. *Phys. Plasmas*, 6(5):2290, 1999.

[17] M. LaPointe. Numerical simulation of geometric scale effects in cylindrical self-field MPD thrusters. *NASA-CR-189224*, 1992.

[18] J.M.G. Chanty and M. Martinez-Sanchez. Two-dimensional numerical simulation of MPD flows. *AIAA-87-1090*, 1987.

[19] E. H. Niewood. *An Explanation for Anode Voltage Drops in an MPD*. PhD thesis, MIT, 1993.

[20] E. J. Sheppard. *Ionization Nonequilibrium and Ignition in Self-Field Magnetoplasmadynamic Thrusters*. PhD thesis, MIT, 1992.

[21] P.J. Turchi, P.G. Mikellides, K.W. Hohman, R.J. Leiweke, I.G. Mikellides, C.S. Schmahl, N.F. Roderick, and R.E. Peterkin Jr. Progress in modeling plasma thrusters and related plasma flows. *IEPC-95-159*, 1995.

[22] D.C. Lilekis and R.E. Peterkin Jr. Effects of azimuthal injection asymmetry of MPD thruster performance using the MACH3 code. *IEPC-95-2677*, 1995.

[23] C. Boie, M. Auweter-Kurtz, H.J. Kaeppeler, and P.C. Sleziona. Application of adaptive numerical schemes for MPD thruster simulation. *IEPC-97-115*, 1997.

[24] J. Heiermann, M. Auweter-Kurtz, J. J. Kaeppeler, A. Eberle, U. Iben, and P. C. Sleziona. Recent improvements of numerical methods for the simulation of MPD thruster flow on adaptive meshes. *IEPC-99-169*, 1999.

[25] R. S. Myong. *Theoretical and Computational Investigation of Nonlinear Waves in Magnetohydrodynamics*. PhD thesis, U. Michigan, 1996.

[26] M. Auweter-Kurtz et al. Numerical modeling of the flow discharge in MPD thrusters. *J. Propulsion & Power*, 1989.

[27] R. J. Goldston and P. H. Rutherford. *Introduction to Plasma Physics*. Institute of Physics Publishing, 1995.

[28] H. Maecker. Plasma jets in arcs in a process of self-induced magnetic compression. *Z. Phys.*, 141(1):198–216, 1955.

[29] E.Y. Choueiri. The scaling of thrust in self-field MPD thrusters. *J. Prop. Power*, 14(5):744–753, 1998.

[30] M. S. DiCapua. *Energy Deposition in Parallel-Plate Plasma Accelerators*. PhD thesis, Princeton U., 1971.

[31] D. D. Villani. *Energy Loss Mechanisms in a Magnetoplasmadynamic Arcjet*. PhD thesis, Princeton U., 1982.

[32] D.Q. King. *Magnetoplasmadynamic Channel Flow for Design of Coaxial MPD Thrusters*. PhD thesis, Princeton U., 1981.

[33] M. Mitchner and C.H. Kruger. *Partially Ionized Gases*. Willy-Interscience, 1973.

[34] M. J. Wolff. A High Performance Magnetoplasmadynamic Thruster. Master's thesis, Princeton University, 1984.

[35] D. J. Heimerdinger. *Fluid Mechanics in a Magnetoplasmadynamic Thruster*. PhD thesis, MIT, 1988.

[36] Yu. P. Raizer. *Gas Discharge Physics*. Springer, 1997.

[37] J. Heiermann, M.Auweter-Kurtz, and P.C. Sleziona. Adaptive computation of the current-carrying plasma in an MPD rocket thruster. *Time-Dependent Magnetohydrodynamics: Anayltical, Numerical, and Application Aspects*, 1998.

[38] M. J. Boyle. *Acceleration Processes in the Quasi-Steady Magnetoplasmadynamic Discharge*. PhD thesis, Princeton U., 1974.

[39] A. P. Bruckner. *Spectroscopic Studies of the Exhaust Plume of a Quasi-Steady MPD Accelerator*. PhD thesis, Princeton U., 1972.

[40] R. J. LeVeque, D. Mihalas, E. A. Dorfi, and E. Müller. *Computational Methods for Astrophysical Fluid Flow*. Springer, 1998.

[41] W. G. Vincenti and C.H. Kruger. *Introduction to Physical Gas Dynamics*. Kruger Publishing, 1965.

[42] W. M. Sparks and D. Fischel. *Partition Functions and Equations of State in Plasmas*. NASA SP-3066, 1971.

[43] E. Y. Choueiri. A Scaling Strategy for the Preliminary Design of MPD Thrusters. Master's thesis, Syracuse University, 1983.

[44] E.Y. Choueiri, A. J. Kelly, and R.G. Jahn. Current-driven plasma acceleration versus current-driven energy dissipation: Part ii: Electromagnetic wave stability theory and experiments. *IEPC-91-100*, 1991.

[45] D.L. Tilley, E.Y. Choueiri, A. J. Kelly, and R.G. Jahn. Microinstabilities in a 10-kilowatt self-field magnetoplasmadynamic thruster. *J.Prop.Power*, 12(2):381, 1996.

[46] H. O. Schrade, P. C. Sleziona, T. Wegmann, and H. L. Kurtz. Basic processes of plasma propulsion: Final report. *AFOSR: 91-0118*, 1991.

[47] M. N. Saha. Ionization in the solar chromosphere. *Phil. Mag.*, 40:472, 1920.

[48] J. L. Kerrebrock and M. A. Hoffman. Nonequilibrium ionization due to electron heating. *AIAA J.*, 2:1072, 1964.

[49] S. Suckewer. Excitation and ionization of atoms and ions in a non-thermal plasma. II: Ionization equilibrium. *J. Phys. B*, 3:390–398, 1970.

[50] J. Vlcek. A collisional-radiative model applicable to argon discharges over a wide range of conditions: Formulation and basic data. *J. Phys.D*, 22:623, 1988.

[51] T. M. Randolph. Measurement if Ionization Levels in the Interelectrode Region of an MPD Thruster. Master's thesis, Princeton University, 1994.

[52] V. Kaufmann and W. Whaling. Improved wavelengths and energy levels of doubly-ionized argon (ar iii). *J. Res. Natl. Inst. Stand. Technol.*, 101:691, 1996.

[53] J.U. Brackbill and D.C. Barnes. The effect of nonzero $\nabla \cdot \mathrm{B}$ on the numerical solution of the magnetohydrodynamic equations. *J. Comp. Phys*, 35:426, 1980.

[54] K.G. Powell. An approximate Riemann solver for magnetohydrodynamics (that works in more than one dimension). *NASA ICASE Report 94-24*, 1994.

[55] E. Godlewski and P. A. Raviart. *Numerical Approximation of Hyperbolic Systems of Conservation Laws*. Springer, 1996.

[56] L. Martinelli. *Calculations of Viscous Flows With a Multigrid Method*. PhD thesis, Princeton U., 1987.

[57] J. von Neumann. A method for the numerical calculation of hydrodynamic shocks. *J. Appl. Phys.*, 21:232–237, 1950.

[58] C. B. Laney. *Computational Gasdynamics*. Cambrigde, 1998.

[59] H. L. Royden. *Real Analysis*. MacMillan, 1968.

[60] A. Harten. High resolution schemes for hyperbolic conservation laws. *J. Comp. Phys*, 49:357–393, 1983.

[61] A. Jameson. Analysis and design of numerical schemes for gas dynamics, 1: Artificial diffusion, upwind biasing, limiters and their effect on accuracy and multigrid convergence. *Comp.Fluid.Dyn*, 1995.

[62] S.K. Godunov. Finite difference method for numerical computation of discontinous solution of the equations of fluid dynamics. *Matematicheskii Sbornik*, 47:15–21, 1959.

[63] S.K. Godunov. Symmetric form of the equations of magnetohydrodynamics. *Numerical Methods for Mechanics of Continuum Medium*, 1972.

[64] S. C. Jardin. *Computational Methods in Plasma Physics*. Princeton U. Lecture Notes, 1998.

[65] Jeffrey and Taniuti. *Nonlinear Wave Propagation With Applications to Physics and Magnetohydrodynamics*. Academic Press, 1964.

[66] P. Roe. Approximate Riemann solvers, parameter vectors, and difference schemes. *J. Comp. Phys*, 43:357, 1981.

[67] P. Roe. Characteristics-based schemes for the Euler equations. *Annual Review of Fluid Mechanics*, 18:337, 1986.

[68] P. Cargo and G. Gallice. Roe matrices for ideal MHD and systematic construction of Roe matrices for systems of conservation laws. *J. Comp. Phys*, 136:446, 1997.

[69] N. Aslan. Two-dimensional solutions of MHD equations with an adapted Roe method. *Int. J. Num. Meth. in Fluids*, 23(11):1211, 1996.

[70] B. van Leer, C. H. Tai, and K. G. Powell. Design of optimally smoothing multi-stage schemes for the euler equations. *AIAA-89-1933*, 1989.

[71] G.A Sod. A survey of finite-difference methods for systems of nonlinear conservation laws. *J. Comp. Phys*, 27:1, 1978.

[72] J.B. Taylor. Relaxation of toroidal plasma and generation of reverse magnetic fields. *Physical Review Letters*, 1974.

[73] E.Y. Choueiri and H. Okuda. Anomalous ionization in MPD thrusters. *IEPC-93-067*, 1993.

[74] J.D. Jackson. *Classical Electrodynamics*. Wiley, 1975.

[75] E. Y. Choueiri, A. J. Kelly, and R. G. Jahn. Pulsed electromagnetic acceleration: JPL contract no.954997. *MAE Report*, 1692.23, 1986.

[76] R. G. Jahn, S. C. Jardin, and E. Y. Choueiri. Perosnal communication. *Princeton University*, 2000.

[77] M. H. Frese. The internal structure and dynamics of the railgun plasma armature between infinitely wide ablating rails. *Los Alamos Natl. Lab. Technical Report: LA-SUB-93-178*, 1993.

[78] J. Buff, M. H. Frese, A. J. Giancola, R. E. Peterkin, and N. F. Roderick. Simulations of a plasma flow switch. *IEEE Transactions on Plasma Science*, PS-15:766–771, 1987.

[79] U. Shumlak. A near-term, z-pinch fusion space thruster. *AIAA-2000-3368*, 2000.

[80] F. R. Chang Diaz, J. P. Squire, R. D. Bengston, B. N. Breizman, F. W. Baity, and M. D. Carter. The physics and engineering of the VASIMR engine. *AIAA-00-3756*, 2000.

[81] J. L. Cambier. Perosnal communication. *MSE Technology Applications, Inc.*, 2000.

[82] J.D. Huba. *NRL Plasma Formulary*. Naval Research Laboratory, 1994.

[83] K.G. Powell et al. An upwind scheme for magnetohydrodynamics. *AIAA-95-1704*, 1995.

[84] K.G. Powell et al. A solution-adaptive upwind scheme for ideal magnetohydrodynamics. *J. Comp. Phys*, 154:284–309, 1999.

Appendix A

VECTOR OPERATIONS

A.1 IDENTITIES

The vector identities used in this chapter are available from the NRL Plasma Formulary[82]. In this notation, **A, B, U** are vectors, $\bar{\bar{\mathcal{I}}}$ is the unit dyad and $\bar{\bar{\mathcal{T}}}$ is a tensor.

$$(\nabla \times \mathbf{B}) \times \mathbf{B} = (\nabla \mathbf{B}) \cdot \mathbf{B} - (\mathbf{B} \cdot \nabla) \mathbf{B} \tag{A.1}$$

$$\nabla \cdot (\mathbf{A}\mathbf{B}) = (\nabla \cdot \mathbf{A}) \mathbf{B} + (\mathbf{A} \cdot \nabla) \mathbf{B} \tag{A.2}$$

$$(\nabla \mathbf{B}) \cdot \mathbf{B} = \nabla \cdot \left[\frac{\mathbf{B} \cdot \mathbf{B}}{2} \bar{\bar{\mathcal{I}}} \right] \tag{A.3}$$

$$\nabla \times [\mathbf{u} \times \mathbf{B}] = [\mathbf{u} (\nabla \cdot \mathbf{B}) + (\mathbf{B} \cdot \nabla) \mathbf{u} - \mathbf{B} (\nabla \cdot \mathbf{u}) - (\mathbf{u} \cdot \nabla) \mathbf{B}] \tag{A.4}$$

$$\nabla \cdot \mathbf{A} = \frac{\partial A_r}{\partial r} + \frac{A_r}{r} + \frac{1}{r} \frac{\partial A_\theta}{\partial \theta} + \frac{\partial A_z}{\partial z} \tag{A.5}$$

$$\nabla \cdot \bar{\bar{\mathcal{T}}} = \left[\begin{array}{l} \frac{1}{r}\frac{\partial}{\partial r}\left(rT_{rr}\right) + \frac{\partial T_{zr}}{\partial z} - \frac{T_{\theta\theta}}{r} \\ \frac{1}{r}\frac{\partial}{\partial r}\left(rT_{r\theta}\right) + \frac{\partial T_{z\theta}}{\partial z} + \frac{T_{\theta r}}{r} \\ \frac{1}{r}\frac{\partial}{\partial r}\left(rT_{rz}\right) + \frac{\partial T_{zz}}{\partial z} \end{array} \right] \tag{A.6}$$

$$\nabla \mathbf{A} = \left[\begin{array}{ccc} \frac{\partial A_r}{\partial r} & \frac{\partial A_\theta}{\partial r} & \frac{\partial A_z}{\partial r} \\ \frac{1}{r}\left(\frac{\partial A_r}{\partial \theta} - A_\theta\right) & \frac{1}{r}\left(\frac{\partial A_\theta}{\partial \theta} + A_r\right) & \frac{1}{r}\frac{\partial A_z}{\partial \theta} \\ \frac{\partial A_r}{\partial z} & \frac{\partial A_\theta}{\partial z} & \frac{\partial A_z}{\partial z} \end{array} \right] \tag{A.7}$$

A.2 MANIPULATIONS

Using the definition,

$$\bar{\bar{j}} = \frac{1}{\mu_o}\left[\nabla \mathbf{B} - \nabla \mathbf{B}^\dagger\right] \ ,$$

and the Ohm's law (without ∇p_e drift) written in the form,

$$\mathbf{E}' = \eta_o \mathbf{j} + \frac{\mathbf{j} \times \mathbf{B}}{e n_e} = \bar{\bar{\eta}} \cdot \mathbf{j} \ , \tag{A.8}$$

the resistive diffusion of the magnetic field can be written as the divergence of the tensor,

$$\bar{\bar{E}}_{res} = \left[\bar{\bar{\eta}} \cdot \bar{\bar{j}}\right] - \left[\bar{\bar{\eta}} \cdot \bar{\bar{j}}\right]^\dagger - \left[\eta_o \bar{\bar{j}}\right] \ .$$

In other words,

$$\nabla \times \mathbf{E}' = -\nabla \cdot \bar{\bar{E}}_{res} \ . \tag{A.9}$$

Appendix B

EIGENSYSTEM

B.1 Eigensystem of MHD

Alfvèn speeds: $C_{A;r,\theta,z} = \frac{B_{r,\theta,z}}{\sqrt{\mu_o \rho}}$,

Sonic speed: $a = \sqrt{\frac{\gamma p}{\rho}}$

Normalization coefficients (based on the work in refs.[83], [84])

$$\beta_{r;\theta,z} = \frac{C_{A;\theta,z}}{\sqrt{C_{A;\theta}^2 + C_{A;z}^2}} \quad \alpha_{r;f,s} = \sqrt{\pm \frac{a^2 C_{S,F;r}^2}{C_{F;r}^2 - C_{S;r}^2}}$$
$$\beta_{z;r,\theta} = \frac{C_{A;r,\theta}}{\sqrt{C_{A;r}^2 + C_{A;\theta}^2}} \quad \alpha_{z;f,s} = \sqrt{\pm \frac{a^2 C_{S,F;z}^2}{C_{F;z}^2 - C_{S;z}^2}}$$

Fast and slow magnetosonic waves:

$$C_{F,S;\,r}^2 = \frac{1}{2}\left[(\frac{\mathbf{B}\cdot\mathbf{B}}{\mu_o \rho} + a^2) \pm \sqrt{(\frac{\mathbf{B}\cdot\mathbf{B}}{\mu_o \rho} + a^2)^2 - (4a^2 C_{A;\,r}^2)} \right]$$
$$C_{F,S;\,z}^2 = \frac{1}{2}\left[(\frac{\mathbf{B}\cdot\mathbf{B}}{\mu_o \rho} + a^2) \pm \sqrt{(\frac{\mathbf{B}\cdot\mathbf{B}}{\mu_o \rho} + a^2)^2 - (4a^2 C_{A;\,z}^2)} \right]$$

The Jacobian of transformation between primitive and conservation variables:

$$
\frac{d\mathbf{U}}{d\mathbf{W}} = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & \\
u & \rho & 0 & 0 & 0 & 0 & 0 & 0 \\
v & 0 & \rho & 0 & 0 & 0 & 0 & 0 \\
w & 0 & 0 & \rho & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
\frac{\mathbf{u}\cdot\mathbf{u}}{2} & \rho u & \rho v & \rho w & \frac{B_x}{\mu_o} & \frac{B_y}{\mu_o} & \frac{B_z}{\mu_o} & \frac{1}{\gamma-1}
\end{bmatrix}
$$

B.1.1 $\hat{\mathbf{r}}$ Direction

Eigenvalues

(in non-decreasing order):

$$[u - C_{F;r},\ u - C_{A;r},\ u - C_{S;r},\ u,\ u,\ u + C_{S;r},\ u + C_{A;r},\ u + C_{F;r}] \tag{B.1}$$

Ortho-normalized eigenvectors

$$
\begin{aligned}
L1_r &= \left[0, \frac{-\alpha_{r;f}C_{F;r}}{2a^2}, \frac{\alpha_{r;s}C_{S;r}\beta_{r;\theta}Sgn[B_r]}{2a^2}, \frac{\alpha_{r;s}C_{S;r}\beta_{r;z}Sgn[B_r]}{2a^2}, 0, \frac{\alpha_{r;s}\beta_{r;\theta}}{2a\sqrt{\mu_o\rho}}, \frac{\alpha_{r;s}\beta_{r;z}}{2a\sqrt{\mu_o\rho}}, \frac{\alpha_{r;f}}{2\rho a^2}\right] \\
L2_r &= \left[0, 0, -\frac{\beta_{r;z}}{\sqrt{2}}, \frac{\beta_{r;\theta}}{\sqrt{2}}, 0, -\frac{\beta_{r;z}}{\sqrt{2\mu_o\rho}}, \frac{\beta_{r;\theta}}{\sqrt{2\mu_o\rho}}, 0\right] \\
L3_r &= \left[0, \frac{-\alpha_{r;s}C_{S;r}}{2a^2}, \frac{-\alpha_{r;f}C_{F;r}\beta_{r;\theta}Sgn[B_r]}{2a^2}, \frac{-\alpha_{r;f}C_{F;r}\beta_{r;z}Sgn[B_r]}{2a^2}, 0, \frac{-\alpha_{r;f}\beta_{r;\theta}}{2a\sqrt{\mu_o\rho}}, \frac{-\alpha_{r;f}\beta_{r;z}}{2a\sqrt{\mu_o\rho}}, \frac{\alpha_{r;f}}{2\rho a^2}\right] \\
L4_r &= \left[1, 0, 0, 0, 0, 0, 0, \frac{-1}{a^2}\right] \\
L5_r &= [0, 0, 0, 0, 1, 0, 0, 0] \\
L6_r &= \left[0, \frac{\alpha_{r;s}C_{S;r}}{2a^2}, \frac{\alpha_{r;f}C_{F;r}\beta_{r;\theta}Sgn[B_r]}{2a^2}, \frac{\alpha_{r;f}C_{F;r}\beta_{r;z}Sgn[B_r]}{2a^2}, 0, \frac{-\alpha_{r;f}\beta_{r;\theta}}{2a\sqrt{\mu_o\rho}}, \frac{-\alpha_{r;f}\beta_{r;z}}{2a\sqrt{\mu_o\rho}}, \frac{\alpha_{r;f}}{2\rho a^2}\right] \\
L7_r &= \left[0, 0, -\frac{\beta_{r;z}}{\sqrt{2}}, \frac{\beta_{r;\theta}}{\sqrt{2}}, 0, \frac{\beta_{r;z}}{\sqrt{2\mu_o\rho}}, -\frac{\beta_{r;\theta}}{\sqrt{2\mu_o\rho}}, 0\right] \\
L8_r &= \left[0, \frac{\alpha_{r;f}C_{F;r}}{2a^2}, \frac{-\alpha_{r;s}C_{S;r}\beta_{r;\theta}Sgn[B_r]}{2a^2}, \frac{-\alpha_{r;s}C_{S;r}\beta_{r;z}Sgn[B_r]}{2a^2}, 0, \frac{\alpha_{r;s}\beta_{r;\theta}}{2a\sqrt{\mu_o\rho}}, \frac{\alpha_{r;s}\beta_{r;z}}{2a\sqrt{\mu_o\rho}}, \frac{\alpha_{r;f}}{2\rho a^2}\right]
\end{aligned}
$$

$$\begin{aligned}
R1_r &= \Big[\rho\alpha_{r;f}, -\alpha_{r;f}C_{F;r}, \alpha_{r;s}C_{S;r}\beta_{r;\theta}Sgn\left[B_r\right], \alpha_{r;s}C_{S;r}\beta_{r;z}Sgn\left[B_r\right], 0, \alpha_{r;s}a\beta_{r;\theta}\sqrt{\mu_o\rho}, \\
&\quad \alpha_{r;s}a\beta_{r;z}\sqrt{\mu_o\rho}, \rho a^2\alpha_{r;f}\Big] \\
R2_r &= \Big[0, 0, \tfrac{-\beta_{r;z}}{\sqrt{2}}, \tfrac{\beta_{r;\theta}}{\sqrt{2}}, 0, \tfrac{-\beta_{r;z}\sqrt{\mu_o\rho}}{\sqrt{2}}, \tfrac{\beta_{r;\theta}\sqrt{\mu_o\rho}}{\sqrt{2}}, 0\Big] \\
R3_r &= \Big[\rho\alpha_{r;s}, -\alpha_{r;s}C_{S;r}, -\alpha_{r;f}C_{F;r}\beta_{r;\theta}Sgn[B_r], -\alpha_{r;f}C_{F;r}\beta_{r;z}Sgn[B_r], 0, -\alpha_{r;f}a\beta_{r;\theta}\sqrt{\mu_o\rho}, \\
&\quad -\alpha_{r;f}a\beta_{r;z}\sqrt{\mu_o\rho}, \rho a^2\alpha_{r;f}\Big] \\
R4_r &= [1, 0, 0, 0, 0, 0, 0, 0] \\
R5_r &= [0, 0, 0, 0, 1, 0, 0, 0] \\
R6_r &= \Big[\rho\alpha_{r;s}, \alpha_{r;s}C_{S;r}, \alpha_{r;f}C_{F;r}\beta_{r;\theta}Sgn[B_r], \alpha_{r;f}C_{F;r}\beta_{r;z}Sgn[B_r], 0, -\alpha_{r;f}a\beta_{r;\theta}\sqrt{\mu_o\rho}, \\
&\quad -\alpha_{r;f}a\beta_{r;z}\sqrt{\mu_o\rho}, \rho a^2\alpha_{r;f}\Big] \\
R7_r &= \Big[0, 0, \tfrac{-\beta_{r;z}}{\sqrt{2}}, \tfrac{\beta_{r;\theta}}{\sqrt{2}}, 0, \tfrac{\beta_{r;z}\sqrt{\mu_o\rho}}{\sqrt{2}}, \tfrac{-\beta_{r;\theta}\sqrt{\mu_o\rho}}{\sqrt{2}}, 0\Big] \\
R8_r &= \Big[\rho\alpha_{r;f}, \alpha_{r;f}C_{F;r}, -\alpha_{r;s}C_{S;r}\beta_{r;\theta}Sgn[B_r], -\alpha_{r;s}C_{S;r}\beta_{r;z}Sgn[B_r], 0, \alpha_{r;s}a\beta_{r;\theta}\sqrt{\mu_o\rho}, \\
&\quad \alpha_{r;s}a\beta_{r;z}\sqrt{\mu_o\rho}, \rho a^2\alpha_{r;f}\Big]
\end{aligned}$$

B.1.2 $\hat{\mathbf{z}}$ Direction

Eigenvalues

(in non-decreasing order)

$$\left[\; w - C_{F;z},\; w - C_{A;z},\; w - C_{S;z},\; w,\; w,\; w + C_{S;z},\; w + C_{A;z},\; w + C_{F;z} \;\right] \qquad \text{(B.2)}$$

Ortho-normalized eigenvectors

$$
\begin{aligned}
L1_z &= \left[0, \tfrac{\alpha_{z;s} C_{S;z} \beta z;r Sgn[B_z]}{2a^2}, \tfrac{\alpha_{z;s} C_{S;z} \beta_{z;\theta} Sgn[B_z]}{2a^2}, \tfrac{-\alpha_{z;f} C_{F;z}}{2a^2}, \tfrac{\alpha_{z;s}\beta_{z;r}}{2a\sqrt{\mu_o\rho}}, \tfrac{\alpha_{z;s}\beta z;\theta}{2a\sqrt{\mu_o\rho}}, 0, \tfrac{\alpha_{z;f}}{2\rho a^2}\right] \\
L2_z &= \left[0, \tfrac{-\beta_{z;\theta}}{\sqrt{2}}, \tfrac{\beta_{z;r}}{\sqrt{2}}, 0, \tfrac{-\beta_{z;\theta}}{\sqrt{2\mu_o\rho}}, \tfrac{\beta_{z;r}}{\sqrt{2\mu_o\rho}}, 0, 0\right] \\
L3_z &= \left[0, \tfrac{-\alpha_{z;f} C_{F;z} \beta z;r Sgn[B_z]}{2a^2}, \tfrac{-\alpha_{z;f} C_{F;z} \beta_{z;\theta} Sgn[B_z]}{2a^2}, \tfrac{-\alpha_{z;s} C_{S;z}}{2a^2}, \tfrac{-\alpha_{z;f}\beta_{z;r}}{2a\sqrt{\mu_o\rho}}, \tfrac{-\alpha_{z;f}\beta z;\theta}{2a\sqrt{\mu_o\rho}}, 0, \tfrac{\alpha_{z;s}}{2\rho a^2}\right] \\
L4_z &= [0, 0, 0, 0, 0, 0, 1, 0] \\
L5_z &= \left[1, 0, 0, 0, 0, 0, 0, \tfrac{-1}{a^2}\right] \\
L6_z &= \left[0, \tfrac{\alpha_{z;f} C_{F;z} \beta z;r Sgn[B_z]}{2a^2}, \tfrac{\alpha_{z;f} C_{F;z} \beta_{z;\theta} Sgn[B_z]}{2a^2}, \tfrac{\alpha_{z;s} C_{S;z}}{2a^2}, \tfrac{-\alpha_{z;f}\beta_{z;r}}{2a\sqrt{\mu_o\rho}}, \tfrac{-\alpha_{z;f}\beta z;\theta}{2a\sqrt{\mu_o\rho}}, 0, \tfrac{\alpha_{z;s}}{2\rho a^2}\right] \\
L7_z &= \left[0, \tfrac{-\beta_{z;\theta}}{\sqrt{2}}, \tfrac{\beta_{z;r}}{\sqrt{2}}, 0, \tfrac{\beta_{z;\theta}}{\sqrt{2\mu_o\rho}}, \tfrac{-\beta_{z;r}}{\sqrt{2\mu_o\rho}}, 0, 0\right] \\
L8_z &= \left[0, \tfrac{-\alpha_{z;s} C_{S;z} \beta z;r Sgn[B_z]}{2a^2}, \tfrac{-\alpha_{z;s} C_{S;z} \beta_{z;\theta} Sgn[B_z]}{2a^2}, \tfrac{\alpha_{z;f} C_{F;z}}{2a^2}, \tfrac{\alpha_{z;s}\beta_{z;r}}{2a\sqrt{\mu_o\rho}}, \tfrac{\alpha_{z;s}\beta z;\theta}{2a\sqrt{\mu_o\rho}}, 0, \tfrac{\alpha_{z;f}}{2\rho a^2}\right]
\end{aligned}
$$

$$
\begin{aligned}
R1_z &= \Big[\rho\alpha_{z;f}, \alpha_{z;s} C_{S;z}\beta_{z;r} Sgn[B_z], \alpha_{z;s} C_{S;z}\beta_{z;\theta} Sgn[B_z], -\alpha_{z;f} C_{F;z}, \alpha_{z;s} a\beta_{z;r}\sqrt{\mu_o\rho}, \\
&\quad \alpha_{z;s} a\beta_{z;\theta}\sqrt{\mu_o\rho}, 0, \rho a^2\alpha_{z;f}\Big] \\
R2_z &= \left[0, \tfrac{-\beta_{z;\theta}}{\sqrt{2}}, \tfrac{\beta_{z;r}}{\sqrt{2}}, 0, \tfrac{-\beta_{z;\theta}\sqrt{\mu_o\rho}}{\sqrt{2}}, \tfrac{\beta_{z;r}\sqrt{\mu_o\rho}}{\sqrt{2}}, 0, 0\right] \\
R3_z &= \Big[\rho\alpha_{z;s}, -\alpha_{z;f} C_{F;z}\beta_{z;r} Sgn[B_z], -\alpha_{z;f} C_{F;z}\beta_{z;\theta} Sgn[B_z], -\alpha_{z;s} C_{S;z}, -\alpha_{z;f} a\beta_{z;r}\sqrt{\mu_o\rho}, \\
&\quad -\alpha_{z;f} a\beta_{z;\theta}\sqrt{\mu_o\rho}, 0, \rho a^2\alpha_{z;s}\Big] \\
R4_z &= [0, 0, 0, 0, 0, 0, 1, 0] \\
R5_z &= [1, 0, 0, 0, 0, 0, 0, 0] \\
R6_z &= \Big[\rho\alpha_{z;s}, \alpha_{z;f} C_{F;z}\beta_{z;r} Sgn[B_z], \alpha_{z;f} C_{F;z}\beta_{z;\theta} Sgn[B_z], \alpha_{z;s} C_{S;z}, -\alpha_{z;f} a\beta_{z;r}\sqrt{\mu_o\rho}, \\
&\quad -\alpha_{z;f} a\beta_{z;\theta}\sqrt{\mu_o\rho}, 0, \rho a^2\alpha_{z;s}\Big] \\
R7_z &= \left[0, \tfrac{-\beta_{z;\theta}}{\sqrt{2}}, \tfrac{\beta_{z;r}}{\sqrt{2}}, 0, \tfrac{\beta_{z;\theta}\sqrt{\mu_o\rho}}{\sqrt{2}}, \tfrac{-\beta_{z;r}\sqrt{\mu_o\rho}}{\sqrt{2}}, 0, 0\right] \\
R8_z &= \Big[\rho\alpha_{z;f}, -\alpha_{z;s} C_{S;z}\beta_{z;r} Sgn[B_z], -\alpha_{z;s} C_{S;z}\beta_{z;\theta} Sgn[B_z], \alpha_{z;f} C_{F;z}, \alpha_{z;s} a\beta_{z;r}\sqrt{\mu_o\rho}, \\
&\quad \alpha_{z;s} a\beta_{z;\theta}\sqrt{\mu_o\rho}, 0, \rho a^2\alpha_{z;f}\Big]
\end{aligned}
$$

Appendix C

COMPUTER PROGRAM

C.1 Main

```
//-----------------------------------------------------------
//Thruster-Axi.CPP; Written by Kameshwaran Sankaran
//-----------------------------------------------------------

#include "Thruster-Axi.h"

#include "InputOutput.cpp"
#include "Grid.cpp"
#include "Memory.cpp"
#include "InitialConditions.cpp"
#include "Solve.cpp"
#include "SetTimeStep.cpp"
#include "BoundaryConditions.cpp"
#include "EvalConv.cpp"
#include "EvalDiss.cpp"
#include "TimeMarch.cpp"
#include "ReCalculate.cpp"


void main()
{
    GetInput();
    MemAllocate();
    SetInitial();
    Solve();
}
```

C.2 Declarations

```
//--------------------------------------------------------
//Thruster-Axi.h; Written by Kameshwaran Sankaran
//--------------------------------------------------------

#include <iostream.h>
#include <iomanip.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <complex>

FILE *ConvergeDiff, *ConvergeVal;
FILE *Converge;

const double PI=3.14159265359;
const double Root2=1.4142136;
//Acceleration due to gravity at Earth, in m/s^2
const double go=9.81;
//Permeability of free space
const double Mu=1.256637061436e-6;
//Permittivity of free space
const double Eps0=8.854187817e-12;
//Ideal ratio of specific heats for Argon
const double gamConst=1.666666666;
//Boltzmann’s constant in Joules/Kelvin
const double kBoltz=1.380658e-23;
//Gas constant for Argon in J/Kg.K
const double Rargon= 208.1333519883;
//Mass of Argon ion in Kg
const double mAr=6.63352599096e-26;
//Mass of election in Kg
const double mEl=9.1093897e-31;
//Charage of electron in Coulombs
const double q=1.60217733e-19;
//1 electron volt in K
//Resistivity of tungsten in Ohm.m
const double eV=11604.44751705;
const double ResTungst=5.6e-8;
//Resistivity of copper in Ohm.m
const double ResCopper=1.7e-8;
//Thermal conductivity of tungsten in W/m.K
const double ThermTungst=174.0;
//Thermal conductivity of copper in W/m.K
const double ThermCopper=401.0;

int Ra, Rc, Za, Zc, numR=0, numZ=0, numT=0,
    row, col, i=0, j=0, k=0, l=0, m=0, J=0, K=0, numSteps=0, SubStep;
int MaxStep=10;
//These are variables for the equation of state.
int    In1, In2, In3, In4, In5, In2A, In2B, In3A, In3B, In4A, In4B;
int Zone, Count;
int ExtraMemory =0;

double AnodeLength,CathodeLength,Ranode,Rcathode,MassFlowRate,Jmax;
double RAD, Z, deltaR, deltaZ, TotTime, deltaT=1.0e-8,
       deltaTconv, deltaTtherm, deltaTresist, Lmax;
double t = 0.0;
double eps = 1.0e-15;//A small number, used to estimate cut offs.
double Jback=0.0, Vappl = 10.0;//Initial guesses.
//Guess for the temperature of the electrodes, in K.
double ElecTemp = 3.0e3;
double K0, K1, K2;//These are coefficients in the equation of state.
double RhoIn, VzIn, peIn, phIn, TeIn, ThIn;
double Tinlet=0.0, Tupstream=0.0, Texit=0.0, Thrust=0.0,
       MaeckerT=0.0, Isp;

int Rmin[4], Rmax[4], Zmin[4], Zmax[4];    //Vertices of the zone.

//These are scalars at every point.
double    **Vr, **Vz, **Cf, **SpecRadR, **SpecRadZ,
        **VSq, **aSq, **CmSq;
double    **Bt, **BSq;
double    **Er, **Ez, **ErH, **EzH, **EFr, **EFz, **jr, **jz,
          **jDens, **Jencl, **Potential;
double    **n, **Rho, **p, **E, **P;
double    **Res, **EIcollFreq, **ElecGyro, **ElecHall,
          **CoulombLog, **kTherm, **ElCondR, **ElCondZ,
          **kIon, **IonCondR, **IonCondZ, **ThermCondR,
          **ThermCondZ;
double    **Vde, **Vti, **AnomFreqEl, **TotCollFreq;
double    **PR, **gam, **gamOld;


//These are vectors at every point.
double **u[5], **ConvFlux[5],
       **Hr[5], **Fr[5], **Dr[5], **SourceConv[5], **SourceR[5], **Lr[5], **Sr[5],
       **Hz[5], **Fz[5], **Dz[5], **Lz[5], **Sz[5],
       **DissFlux[5], **DissipR[5], **SourceDiss[5], **DissipZ[5];

//These are from "EvalD()".
double **RdelU[5], **ZdelU[5];

//Variables for the two-temperature model.
double **EintEl, **EintH, **ne, **neOld, **ni, **nii, **niii, **nA,
       **pe, **ph, **Te, **Th, **Zeff, **Qei, **Qeii, **Qeiii,
       **nuei, **nueii, **nueiii, **CGvosQ, **Ce, **FrEn, **FzEn,
       **DrEn, **DzEn, **SrEn, **ElecComp, **Ohmic, **Exchange,
       **EnFluxR, **EnFluxZ, **EnSourceR;

//Convergence checks
double    **uOld[5];
double    u0Dmax=0.0, u1Dmax=0.0, u2Dmax=0.0, u3Dmax=0.0, u4Dmax=0.0,
        u0Davg=0.0, u1Davg=0.0, u2Davg=0.0, u3Davg=0.0, u4Davg=0.0,
        u0DMaxR, u0DMaxZ, u1DMaxR, u1DMaxZ, u2DMaxR, u2DMaxZ,
        u3DMaxR, u3DMaxZ, u4DMaxR, u4DMaxZ, u0Max=0.0, u1Max=0.0,
        u2Max=0.0, u3Max=0.0, u4Max=0.0, u0Avg=0.0, u1Avg=0.0,
        u2Avg=0.0, u3Avg=0.0, u4Avg=0.0, u0MaxR, u0MaxZ, u1MaxR,
        u1MaxZ, u2MaxR, u2MaxZ, u3MaxR, u3MaxZ, u4MaxR, u4MaxZ;

double    **KEold, **KE, **Etherm, **EthOld;
double    DensD=0.0, KED=0.0, BtD=0.0, EtotD=0.0, EthD=0.0,
        DensDavg=0.0, KEDavg=0.0, BtDavg=0.0,EtotDavg=0.0,
        EthDavg=0.0, DensMaxR, DensMaxZ, KEmaxR, KEmaxZ,
        BtMaxR, BtMaxZ, EtotMaxR, EtotMaxZ, EthMaxR, EthMaxZ;
/*
//These are required by CharSplit().
double **rho, **ur, **uz, **bt, **hOld, **BtAvg;
double **alfF, **alfS, **betaT, **ZalfF, **ZalfS, **ZbetaT;
double **L1, **L2, **L3, **L4, **L5, **L1m, **L2m, **L3m, **L4m,
       **L5m, **L1p, **L2p, **L3p, **L4p, **L5p, **ZL1, **ZL2,
       **ZL3, **ZL4, **ZL5, **ZL1m, **ZL2m, **ZL3m, **ZL4m,
       **ZL5m, **ZL1p, **ZL2p, **ZL3p, **ZL4p, **ZL5p;

//These are vectors are every point.
double **R1[5], **R2[5], **R3[5], **R4[5], **R5[5], **Left1[5],
       **Left2[5], **Left3[5], **Left4[5], **Left5[5],
       **R1z[5], **R2z[5], **R3z[5], **R4z[5], **R5z[5], **Left1z[5],
       **Left2z[5], **Left3z[5], **Left4z[5], **Left5z[5];

//These are matrices at every point.
double **Lplus[5][5], **Lminus[5][5], **LplusZ[5][5], **LminusZ[5][5];
double **Lambda[5][5], **LambdaZ[5][5], **AbsLambda[5][5],
       **AbsLambdaZ[5][5];
double **Aplus[5][5], **Aminus[5][5], **Bplus[5][5], **Bminus[5][5];
double **Rp[5][5], **Lp[5][5], **RpZ[5][5], **LpZ[5][5];
double **R[5][5], **Rinv[5][5], **RZ[5][5], **RinvZ[5][5];
double **M[5][5], **Minv[5][5], **AbsA[5][5], **AbsB[5][5];
double **temp[5][5];
*/
//FUNCTION DECLARATIONS...
double EvalSign(double x);

void ConvBound();
void DissBound();
void EquationOfState();
void EvalConv();
void EvalDiss();
void EvalEnergy();
void EvalF();
void EvalGamma();
void EvalLim();
void EvalNumDiss();
void EvalParam();
void EvalS();
void EvalTimeStep();
void EvalVar();
void GetInput();
void Grid();
void MemAllocate();
void ReCalculate();
void Saha();
void SetInitial();
void Solve();
void TerminalChars();
void TimeMarch();
void Transport();
void WriteFile();

//Memory functions
double **AllocMatrix(int RowBegin,int RowEnd,int ColBegin,int ColEnd);
/*
void ArtVisc();
void CharSplit();
void AllocMemoryEXTRA();
void EvalLambdaR(int Zone);
void EvalLambdaZ(int Zone);
void EvalVectorsR(int Zone);
void EvalVectorsZ(int Zone);
void EvalA(int Zone);
void AverageR(int Zone);
void AverageZ(int Zone);
void EvalM(int Zone);
void EvalMinv(int Zone);
void MulMatrix(double **MatA[5][5], double **MatB[5][5], double **MatC[5][5], int Zone);
void AddMatrix(double **MatA[5][5], double **MatB[5][5], double **MatC[5][5], int Zone);
void SubMatrix(double **MatA[5][5], double **MatB[5][5], double **MatC[5][5], int Zone);
void MatTimesVec(double **MatA[5][5], double **VecB[5],
                 double **VecC[5], int Zone);
void AddVector(double **VecA[5], double **VecB[5],
               double **VecC[5], int Zone);
```

```
void SubVector(double **VecA[5], double **VecB[5],
               double **VecC[5], int Zone);
*/
```

C.3 Memory Allocations

```
//-------------------------------------------------------
//Memory.CPP; Written by Kameshwaran Sankaran
//-------------------------------------------------------
void MemAllocate()
{
//Allocating the scalars.
//Conservation variables
    Rho =AllocMatrix(0, numR+1, 0, numZ+1);
    Vr    =AllocMatrix(0, numR+1, 0, numZ+1);
    Vz    =AllocMatrix(0, numR+1, 0, numZ+1);
    VSq =AllocMatrix(0, numR+1, 0, numZ+1);
    Bt    =AllocMatrix(0, numR+1, 0, numZ+1);
    BSq =AllocMatrix(0, numR+1, 0, numZ+1);
    E    =AllocMatrix(0, numR+1, 0, numZ+1);

//Primitive variables
    n    =AllocMatrix(0, numR+1, 0, numZ+1);
    p    =AllocMatrix(0, numR+1, 0, numZ+1);
    P    =AllocMatrix(0, numR+1, 0, numZ+1);
    Er    =AllocMatrix(0, numR+1, 0, numZ+1);
    Ez    =AllocMatrix(0, numR+1, 0, numZ+1);
    ErH    =AllocMatrix(0, numR+1, 0, numZ+1);
    EzH    =AllocMatrix(0, numR+1, 0, numZ+1);
    EFr    =AllocMatrix(0, numR+1, 0, numZ+1);
    EFz    =AllocMatrix(0, numR+1, 0, numZ+1);
    jr    =AllocMatrix(0, numR+1, 0, numZ+1);
    jz    =AllocMatrix(0, numR+1, 0, numZ+1);
    jDens    =AllocMatrix(0, numR+1, 0, numZ+1);
    Jencl    =AllocMatrix(0, numR+1, 0, numZ+1);
    Potential=AllocMatrix(0, numR+1, 0, numZ+1);

//Variables from the two-temperature model.
    EintEl    =AllocMatrix(0, numR+1, 0, numZ+1);
    EintH    =AllocMatrix(0, numR+1, 0, numZ+1);
    ne    =AllocMatrix(0, numR+1, 0, numZ+1);
    neOld    =AllocMatrix(0, numR+1, 0, numZ+1);
    ni    =AllocMatrix(0, numR+1, 0, numZ+1);
    nii    =AllocMatrix(0, numR+1, 0, numZ+1);
    niii=AllocMatrix(0, numR+1, 0, numZ+1);
    nA    =AllocMatrix(0, numR+1, 0, numZ+1);
    pe    =AllocMatrix(0, numR+1, 0, numZ+1);
    ph    =AllocMatrix(0, numR+1, 0, numZ+1);
    Te    =AllocMatrix(0, numR+1, 0, numZ+1);
    Th    =AllocMatrix(0, numR+1, 0, numZ+1);
    FrEn    =AllocMatrix(0, numR+1, 0, numZ+1);
    FzEn    =AllocMatrix(0, numR+1, 0, numZ+1);
    DrEn    =AllocMatrix(0, numR+1, 0, numZ+1);
    DzEn    =AllocMatrix(0, numR+1, 0, numZ+1);
    SrEn    =AllocMatrix(0, numR+1, 0, numZ+1);
    ElecComp=AllocMatrix(0, numR+1, 0, numZ+1);
    Ohmic    =AllocMatrix(0, numR+1, 0, numZ+1);
    Exchange=AllocMatrix(0, numR+1, 0, numZ+1);
    EnFluxR    =AllocMatrix(0, numR+1, 0, numZ+1);
    EnFluxZ    =AllocMatrix(0, numR+1, 0, numZ+1);
    EnSourceR    =AllocMatrix(0, numR+1, 0, numZ+1);

//Dissipation variables
    Res            =AllocMatrix(0, numR+1, 0, numZ+1);
    EIcollFreq    =AllocMatrix(0, numR+1, 0, numZ+1);
    AnomFreqEl    =AllocMatrix(0, numR+1, 0, numZ+1);
    TotCollFreq    =AllocMatrix(0, numR+1, 0, numZ+1);
    ElecGyro    =AllocMatrix(0, numR+1, 0, numZ+1);
    ElecHall    =AllocMatrix(0, numR+1, 0, numZ+1);
    CoulombLog    =AllocMatrix(0, numR+1, 0, numZ+1);
    kTherm        =AllocMatrix(0, numR+1, 0, numZ+1);
    kIon        =AllocMatrix(0, numR+1, 0, numZ+1);
    IonCondR    =AllocMatrix(0, numR+1, 0, numZ+1);
    IonCondZ    =AllocMatrix(0, numR+1, 0, numZ+1);
    ElCondR        =AllocMatrix(0, numR+1, 0, numZ+1);
    ElCondZ        =AllocMatrix(0, numR+1, 0, numZ+1);
    ThermCondR    =AllocMatrix(0, numR+1, 0, numZ+1);
    ThermCondZ    =AllocMatrix(0, numR+1, 0, numZ+1);
//Transport
    Zeff    = AllocMatrix(0, numR+1, 0, numZ+1);
    Qei        = AllocMatrix(0, numR+1, 0, numZ+1);
    Qeii    = AllocMatrix(0, numR+1, 0, numZ+1);
    Qeiii    = AllocMatrix(0, numR+1, 0, numZ+1);
    nuei    = AllocMatrix(0, numR+1, 0, numZ+1);
    nueii    = AllocMatrix(0, numR+1, 0, numZ+1);
    nueiii    = AllocMatrix(0, numR+1, 0, numZ+1);
    CGvosQ    = AllocMatrix(0, numR+1, 0, numZ+1);
    Ce        = AllocMatrix(0, numR+1, 0, numZ+1);
//Speeds
    aSq    =AllocMatrix(0, numR+1, 0, numZ+1);
    CmSq=AllocMatrix(0, numR+1, 0, numZ+1);
    Cf    =AllocMatrix(0, numR+1, 0, numZ+1);
    SpecRadR    =AllocMatrix(0, numR+1, 0, numZ+1);
    SpecRadZ    =AllocMatrix(0, numR+1, 0, numZ+1);

//For anomalous transport
    Vde    =AllocMatrix(0, numR+1, 0, numZ+1);
    Vti    =AllocMatrix(0, numR+1, 0, numZ+1);
    AnomFreqEl    =AllocMatrix(0, numR+1, 0, numZ+1);

//For the equation of state
    PR    =AllocMatrix(0, numR+1, 0, numZ+1);
    gam    =AllocMatrix(0, numR+1, 0, numZ+1);
    gamOld    =AllocMatrix(0, numR+1, 0, numZ+1);

//For convergence checks
    KEold    = AllocMatrix(0, numR+1, 0, numZ+1);
    KE        = AllocMatrix(0, numR+1, 0, numZ+1);
    Etherm    = AllocMatrix(0, numR+1, 0, numZ+1);
    EthOld    = AllocMatrix(0, numR+1, 0, numZ+1);

//Allocating the vectors.
    for (i=0; i<5; i++)
    {
        u[i]        =AllocMatrix(0, numR+1, 0, numZ+1);
        uOld[i]        =AllocMatrix(0, numR+1, 0, numZ+1);
        ConvFlux[i]    =AllocMatrix(0, numR+1, 0, numZ+1);
        Dr[i]        =AllocMatrix(0, numR+1, 0, numZ+1);
        Fr[i]        =AllocMatrix(0, numR+1, 0, numZ+1);
        Hr[i]        =AllocMatrix(0, numR+1, 0, numZ+1);
        SourceR[i]    =AllocMatrix(0, numR+1, 0, numZ+1);
        SourceConv[i]    =AllocMatrix(0, numR+1, 0, numZ+1);
        Lr[i]        =AllocMatrix(0, numR+1, 0, numZ+1);
        Sr[i]        =AllocMatrix(0, numR+1, 0, numZ+1);
        Dz[i]        =AllocMatrix(0, numR+1, 0, numZ+1);
        Fz[i]        =AllocMatrix(0, numR+1, 0, numZ+1);
        Hz[i]        =AllocMatrix(0, numR+1, 0, numZ+1);
        Lz[i]        =AllocMatrix(0, numR+1, 0, numZ+1);
        Sz[i]        =AllocMatrix(0, numR+1, 0, numZ+1);
        RdelU[i]    = AllocMatrix(0, numR+1, 0, numZ+1);
        ZdelU[i]    = AllocMatrix(0, numR+1, 0, numZ+1);
        DissFlux[i]        =AllocMatrix(0, numR+1, 0, numZ+1);
        DissipR[i]    =AllocMatrix(0, numR+1, 0, numZ+1);
        SourceDiss[i]    =AllocMatrix(0, numR+1, 0, numZ+1);
        DissipZ[i]    =AllocMatrix(0, numR+1, 0, numZ+1);
    }
}
//---------------------------------------
double **AllocMatrix(int RowBegin,int RowEnd,int ColBegin,int ColEnd)
{
    int i;
    double **Mat;

//Creating the blank matrix
//    Mat= new double*[ColEnd-ColBegin+1];
//    Mat -= ColBegin;
    Mat= new double*[RowEnd-RowBegin+1];
    Mat -= RowBegin;

    for(i=RowBegin;i<=RowEnd;i++)
    {
//        Mat[i]= new double[RowEnd-RowBegin+1];
//        Mat[i] -= RowBegin;
        Mat[i]= new double[ColEnd-ColBegin+1];
        Mat[i] -= ColBegin;
    }

//Initializing
    for(k=RowBegin;k<=RowEnd;k++)
    {
        for(l=ColBegin;l<=ColEnd;l++)
        {
            Mat[k][l]=0.0;
        }
    }

    return Mat;
}
//---------------------------------------
```

C.4 Input/Output

```
//-----------------------------------------------------------
//InputOutput.CPP; Written by Kameshwaran Sankaran
//-----------------------------------------------------------
#include <fstream.h>

void GetInput()
{

//Accepting input from file...
    ifstream in("INPUT.DAT");
    in >> MassFlowRate >> Jmax >> RAD >> Ranode >> Rcathode
       >> Z >> AnodeLength >> CathodeLength >> numR >> numZ
       >> TotTime ;

    Grid();

}
//-------------------------------------

void WriteFile()
{
//Temporary output
    if(numSteps%1000 == 0)
    {
        ofstream OutTemp("OUT-Temp.DAT");
        OutTemp<<" \"z\", \"r\", \"ne\", \"n1\", \"n2\", \"n3\",
                   \"nA\", \"Vr\", \"Vz\", \"Bt\", \"Jencl\",
                   \"p\", \"Te\",\"Th\", \"gam\", \"Pot\" "
               <<"\n";

        for(Zone = 1; Zone <= 3; Zone++)
        {
            for (J=Rmin[Zone]+1; J<=Rmax[Zone]; J++)
            {
                for(K=Zmin[Zone]+1; K<=Zmax[Zone]; K++)
                {
                    OutTemp<<(K-1)*deltaZ<<"\t"<<setw(7)
                           <<(J-1)*deltaR<<"\t"<<setw(7)
                           <<ne[J][K]<<"\t"<<setw(7<<ni[J][K]
                           <<"\t"<<setw(7) <<nii[J][K]<<"\t"<<setw(7)
                           <<niii[J][K]<<"\t"<<setw(7)<<nA[J][K]
                           <<"\t"<<setw(7)<<Vr[J][K]<<"\t"<<setw(7)
                           <<Vz[J][K]<<"\t"<<setw(7)<<Bt[J][K]
                           <<"\t"<<setw(7<<Jencl[J][K]<<"\t"<<setw(7)
                           <<p[J][K]<<"\t"<<setw(7)<<Te[J][K]<<"\t"
                           <<setw(7)<<Th[J][K]<<"\t"<<setw(7)
                           <<gam[J][K]<<"\t"<<setw(7)
                           <<Potential[J][K]<<endl;
                }
             }
        }
    }
//Writing the output.
//The output file is formatted to be an input file for TECPLOT.
//Note that even though the variable are stored at cell centers,
//the output values for (z,r) correspond to the lowest indexed
//corner of the cell. So, variables at point (J,K), which is
//(j-0.5, k-0.5), are identified by (j,k).
//For more information, read TECPLOT User’s Manual.
    if (t>=TotTime)
    {
        cerr<<"Completed "<<numSteps<<" time steps in t =
              "<<t<<" seconds; deltaT = "<<deltaT<<endl;
        ofstream OutFinal("OUTPUT.DAT");

        for(Zone = 1; Zone <= 3; Zone++)
        {
            OutFinal<<"VARIABLES = "<<" \"z\", \"r\", \"ne\", \"n1\",
                                        \"n2\", \"n3\", \"nA\", \"Vr\",
                                        \"Vz\", \"Bt\", \"Jencl\",
                                        \"p\", \"Te\", \"Th\", \"gam\",
                                        \"Pot\" " <<"\n"
                    <<"ZONE I="<<Zmax[Zone]-Zmin[Zone]+1<<", J="
                    <<Rmax[Zone]-Rmin[Zone]<<", F=POINT"<<"\n";
            for (J=Rmin[Zone]+1; J<=Rmax[Zone]; J++)
            {
                for(K=Zmin[Zone]+1; K<=Zmax[Zone]+1; K++)
                {
                    OutFinal<<(K-1)*deltaZ<<"\t"<<setw(7)
                            <<(J-1)*deltaR<<"\t"<<setw(7)<<ne[J][K]
                            <<"\t"<<setw(7)<<ni[J][K]<<"\t"<<setw(7)
                            <<nii[J][K]<<"\t"<<setw(7)<<niii[J][K]
                            <<"\t"<<setw(7)<<nA[J][K]<<"\t"<<setw(7)
                            <<Vr[J][K]<<"\t"<<setw(7)<<Vz[J][K]
                            <<"\t"<<setw(7)<<Bt[J][K]<<"\t"<<setw(7)
                            <<Jencl[J][K]<<"\t"<<setw(7)<<p[J][K]
                            <<"\t"<<setw(7)<<Te[J][K]<<"\t"<<setw(7)
                            <<Th[J][K]<<"\t"<<setw(7)<<gam[J][K]<<"\t"
                            <<setw(7)<<Potential[J][K]<<endl;
                 }
            }
        }

        TerminalChars();
        ofstream OutTerm("THRUST.DAT");
        OutTerm<<"Isp = "<<Isp<<" s"<<endl<<"Tupstream = "<<Tupstream
               <<" N"<<endl<<"Texit = "<<Texit<<" N"<<endl
               <<"Tinlet = "<<-Tinlet<<" N"<<endl<<"Thrust = "
               <<Thrust<<" N"<<endl<<"Maecker Thrust = "<<MaeckerT
               <<" N "<<endl;

//All the variables are written to a data file, so that they can
//be reloaded.
        ofstream OutNe("NeOut.DAT");
        ofstream OutNo("NoOut.DAT");
        ofstream OutTe("TeOut.DAT");
        ofstream Outh("PhOut.DAT");
        ofstream OutPh("PhOut.DAT");
        ofstream OutTh("ThOut.DAT");
        ofstream OutGam("GamOut.DAT");
        ofstream OutVr("VrOut.DAT");
        ofstream OutVz("VzOut.DAT");
        ofstream OutBt("BtOut.DAT");

//The variables are stored at cell centers. So, ’J’ refers to ’J-0.5’
//and ’K’ refers to ’K-0.5’.
        for(Zone = 1; Zone <= 3; Zone++)
        {
            for(J=Rmin[Zone]+1; J<=Rmax[Zone]; J++)
            {
                for(K=Zmin[Zone]+1; K<=Zmax[Zone]; K++)
                {
                    OutNe<<ne[J][K]<<endl;
                    OutNo<<n[J][K]<<endl;
                    OutTe<<Te[J][K]<<endl;
                    OutPh<<ph[J][K]<<endl;
                    OutTh<<Th[J][K]<<endl;
                    OutGam<<gam[J][K]<<endl;
                    OutVr<<Vr[J][K]<<endl;
                    OutVz<<Vz[J][K]<<endl;
                    OutBt<<Bt[J][K]<<endl;
                }
            }
        }

        ofstream OutVolt("Voltage.DAT");
        OutVolt<<Vappl;
    }//end of "TotTime" if...the else follows.
    else
    {
        if ( (t>=9*TotTime/10) &&
             (t<=((9*TotTime/10)+(MaxStep*deltaT))) )
        {
            cerr<<"Completed "<<numSteps<<" time steps in t = "<<t
                <<" seconds; deltaT = "<<deltaT<<endl;
            ofstream Out90("90PERCENT.DAT");

            for(Zone = 1; Zone <= 3; Zone++)
            {
                Out90<<"VARIABLES = "<<" \"z\", \"r\", \"ne\", \"n1\",
                       \"n2\", \"n3\", \"nA\", \"Vr\", \"Vz\", \"Bt\",
                       \"Jencl\", \"p\", \"Te\", \"Th\" "<<"\n"
                    <<"ZONE I="<<Zmax[Zone]-Zmin[Zone]+1<<", J="
                    <<Rmax[Zone]-Rmin[Zone]<<", F=POINT"<<"\n";

                for (J=Rmin[Zone]+1; J<=Rmax[Zone]; J++)
                {
                    for(K=Zmin[Zone]+1; K<=Zmax[Zone]+1; K++)
                    {
                        Out90<<(K-1)*deltaZ<<"\t"<<setw(7)
                             <<(J-1)*deltaR<<"\t"<<setw(7)<<ne[J][K]
                             <<"\t"<<setw(7)<<ni[J][K]<<"\t"<<setw(7)
                             <<nii[J][K]<<"\t"<<setw(7)<<niii[J][K]
                             <<"\t"<<setw(7)<<nA[J][K]<<"\t"<<setw(7)
                             <<Vr[J][K]<<"\t"<<setw(7)<<Vz[J][K]<<"\t"
                             <<setw(7)<<Bt[J][K]<<"\t"<<setw(7)
                             <<Jencl[J][K]<<"\t"<<setw(7)<<p[J][K]
                             <<"\t"<<setw(7)<<Te[J][K]<<"\t"<<setw(7)
                             <<Th[J][K]<<endl;
                    }
                }
            }

        }//end of "90PERCENT" if...the else follows.
        else
        {
            if ( (t>=8*TotTime/10) &&
                 (t<=((8*TotTime/10)+(MaxStep*deltaT))) )
            {
                cerr<<"Completed "<<numSteps<<" time steps in t = "
                    <<t<<" seconds; deltaT = "<<deltaT<<endl;
                ofstream Out80("80PERCENT.DAT");

                for(Zone = 1; Zone <= 3; Zone++)
                {
                    Out80<<"VARIABLES = "<<" \"z\", \"r\", \"ne\",
                           \"n1\", \"n2\", \"n3\", \"nA\", \"Vr\",
                           \"Vz\", \"Bt\", \"Jencl\", \"p\", \"Te\",
                           \"Th\" "<<"\n"
                             <<"ZONE I="<<Zmax[Zone]-Zmin[Zone]+1
                        <<", J="<<Rmax[Zone]-Rmin[Zone]
                        <<", F=POINT"<<"\n";
                for (J=Rmin[Zone]+1; J<=Rmax[Zone]; J++)
                {
                    for(K=Zmin[Zone]+1; K<=Zmax[Zone]+1; K++)
                    {
                        Out80<<(K-1)*deltaZ<<"\t"<<setw(7)
                             <<(J-1)*deltaR<<"\t"<<setw(7)
                             <<ne[J][K]<<"\t"<<setw(7)<<ni[J][K]
                             <<"\t"<<setw(7)<<nii[J][K]<<"\t"
                             <<setw(7)<<niii[J][K]<<"\t"<<setw(7)
                             <<nA[J][K]<<"\t"<<setw(7)<<Vr[J][K]
                             <<"\t"<<setw(7)<<Vz[J][K]<<"\t"
                             <<setw(7)<<Bt[J][K]<<"\t"<<setw(7)
                             <<Jencl[J][K]<<"\t"<<setw(7)
                             <<p[J][K]<<"\t"<<setw(7)<<Te[J][K]
                             <<"\t"<<setw(7)<<Th[J][K]<<endl;
                    }
                }
            }
        }//end of "80PERCENT" if...the else follows.
        else
        {

            if ( (t>=7*TotTime/10) &&
                 (t<=((7*TotTime/10)+(MaxStep*deltaT))) )
            {
                cerr<<"Completed "<<numSteps<<" time steps in t = "
                    <<t<<" seconds; deltaT = "<<deltaT<<endl;
                ofstream Out70("70PERCENT.DAT");

                for(Zone = 1; Zone <= 3; Zone++)
                {
                    Out70<<"VARIABLES = "<<" \"z\", \"r\", \"ne\",
                          \"n1\", \"n2\", \"n3\", \"nA\", \"Vr\",
                          \"Vz\", \"Bt\", \"Jencl\", \"p\", \"Te\",
                          \"Th\" "<<"\n"
                         <<"ZONE I="<<Zmax[Zone]-Zmin[Zone]+1<<",
                           J="<<Rmax[Zone]-Rmin[Zone]<<", F=POINT"
                         <<"\n";
                    for (J=Rmin[Zone]+1; J<=Rmax[Zone]; J++)
                    {
                        for(K=Zmin[Zone]+1; K<=Zmax[Zone]+1; K++)
                        {
                            Out70<<(K-1)*deltaZ<<"\t"<<setw(7)
                                 <<(J-1)*deltaR<<"\t"<<setw(7)
                                 <<ne[J][K]<<"\t"<<setw(7)
                                 <<ni[J][K]<<"\t"<<setw(7)
                                 <<nii[J][K]<<"\t"<<setw(7)
                                 <<niii[J][K]<<"\t"<<setw(7)
                                 <<nA[J][K]<<"\t"<<setw(7)
                                 <<Vr[J][K]<<"\t"<<setw(7)
                                 <<Vz[J][K]<<"\t"<<setw(7)
                                 <<Bt[J][K]<<"\t"<<setw(7)
                                 <<Jencl[J][K<<"\t"<<setw(7)
                                 <<p[J][K]<<"\t"<<setw(7)
                                 <<Te[J][K]<<"\t"<<setw(7)
                                 <<Th[J][K]<<endl;
                        }
                    }
                }
            }//end of "70PERCENT" if...the else follows.
//The tab allignments are intentionally moved off.
    else
```

```
{
    if( (t>=6*TotTime/10) &&
        (t<=((6*TotTime/10)+(MaxStep*deltaT))) )
    {
        cerr<<"Completed "<<numSteps<<" time steps in t = "
            <<t<<" seconds; deltaT = "<<deltaT<<endl;
        ofstream Out60("60PERCENT.DAT");

        for(Zone = 1; Zone <= 3; Zone++)
        {
            Out60<<"VARIABLES = "<<" \"z\", \"r\", \"ne\",
                  \"n1\", \"n2\", \"n3\", \"nA\", \"Vr\", \"Vz\",
                  \"Bt\", \"Jencl\", \"p\", \"Te\", \"Th\" "
                  <<"\n"<<"ZONE I="<<Zmax[Zone]-Zmin[Zone]+1
                  <<", J="<<Rmax[Zone]-Rmin[Zone]
                  <<", F=POINT"<<"\n";
            for (J=Rmin[Zone]+1; J<=Rmax[Zone]; J++)
            {
                for(K=Zmin[Zone]+1; K<=Zmax[Zone]+1; K++)
                {
                    Out60<<(K-1)*deltaZ<<"\t"<<setw(7)
                         <<(J-1)*deltaR<<"\t"<<setw(7)<<ne[J][K]
                         <<"\t"<<setw(7)<<ni[J][K]<<"\t"<<setw(7)
                         <<nii[J][K]<<"\t"<<setw(7)<<niii[J][K]
                         <<"\t"<<setw(7)<<nA[J][K]<<"\t"<<setw(7)
                         <<Vr[J][K]<<"\t"<<setw(7)<<Vz[J][K]
                         <<"\t"<<setw(7)<<Bt[J][K]<<"\t"<<setw(7)
                         <<Jencl[J][K]<<"\t"<<setw(7)<<p[J][K]
                         <<"\t"<<setw(7)<<Te[J][K]<<"\t"<<setw(7)
                         <<Th[J][K]<<endl;
                 }
             }
         }
     }//end of "60PERCENT" if...the else follows.
     else
     {
         if( (t>=5*TotTime/10) &&
              (t<=((5*TotTime/10)+(MaxStep*deltaT))) )
         {
             cerr<<"Completed "<<numSteps<<" time steps in t = "
                 <<t<<" seconds; deltaT = "<<deltaT<<endl;
             ofstream Out50("50PERCENT.DAT");

             for(Zone = 1; Zone <= 3; Zone++)
             {
                 Out50<<"VARIABLES = "<<" \"z\", \"r\", \"ne\",
                       \"n1\", \"n2\", \"n3\", \"nA\", \"Vr\",
                       \"Vz\", \"Bt\", \"Jencl\", \"p\", \"Te\",
                       \"Th\" "<<"\n"<<"ZONE I="<<Zmax[Zone]-
                       Zmin[Zone]+1<<", J="<<Rmax[Zone]-Rmin[Zone]
                      <<", F=POINT"<<"\n";
                 for (J=Rmin[Zone]+1; J<=Rmax[Zone]; J++)
                 {
                     for(K=Zmin[Zone]+1; K<=Zmax[Zone]+1; K++)
                     {
                         Out50<<(K-1)*deltaZ<<"\t"<<setw(7)
                              <<(J-1)*deltaR<<"\t"<<setw(7)
                              <<ne[J][K]<<"\t"<<setw(7)<<ni[J][K]
                              <<"\t"<<setw(7)<<nii[J][K]<<"\t"
                              <<setw(7)<<niii[J][K]<<"\t"<<setw(7)
                              <<nA[J][K]<<"\t"<<setw(7)<<Vr[J][K]
                              <<"\t"<<setw(7)<<Vz[J][K]<<"\t"
                              <<setw(7)<<Bt[J][K]<<"\t"<<setw(7)
                              <<Jencl[J][K]<<"\t"<<setw(7)
```

```
                                 <<p[J][K]<<"\t"<<setw(7)<<Te[J][K]
                                 <<"\t"<<setw(7)<<Th[J][K]<<endl;
                          }
                     }
                 }
              }//end of "50PERCENT" if...the else follows.
              else
              {
                  if( (t>=4*TotTime/10) &&
                      (t<=((4*TotTime/10)+(MaxStep*deltaT))) )
                  {
                      cerr<<"Completed "<<numSteps<<" time steps in
                             t = "<<t<<" seconds; deltaT = "<<deltaT
                          <<endl;
                      ofstream Out40("40PERCENT.DAT");

                      for(Zone = 1; Zone <= 3; Zone++)
                      {
                          Out40<<"VARIABLES = "<<" \"z\", \"r\",
                                \"ne\",\"n1\", \"n2\", \"n3\",
                                \"nA\", \"Vr\", \"Vz\", \"Bt\",
                                \"Jencl\", \"p\", \"Te\", \"Th\"
                                "<<"\n"<<"ZONE I="<<Zmax[Zone]-
                                Zmin[Zone]+1<<", J="<<Rmax[Zone]-
                                Rmin[Zone]<<", F=POINT"<<"\n";
                           for (J=Rmin[Zone]+1; J<=Rmax[Zone]; J++)
                           {
                               for(K=Zmin[Zone]+1;K<=Zmax[Zone]+1;K++)
                               {
                                    Out40<<(K-1)*deltaZ<<"\t"<<setw(7)
                                         <<(J-1)*deltaR<<"\t"<<setw(7)
                                         <<ne[J][K]<<"\t"<<setw(7)
                                         <<ni[J][K]<<"\t"<<setw(7)
                                         <<nii[J][K]<<"\t"<<setw(7)
                                         <<niii[J][K]<<"\t"<<setw(7)
                                         <<nA[J][K]<<"\t"<<setw(7)
                                         <<Vr[J][K]<<"\t"<<setw(7)
                                         <<Vz[J][K]<<"\t"<<setw(7)
                                         <<Bt[J][K]<<"\t"<<setw(7)
                                         <<Jencl[J][K]<<"\t"<<setw(7)
                                         <<p[J][K]<<"\t"<<setw(7)
                                         <<Te[J][K]<<"\t"<<setw(7)
                                         <<Th[J][K]<<endl;
                                 }
                             }
                         }
                     }//end of "40PERCENT" if...the else follows.
//The tab allignments are intentionally offset.
    else
    {
        if( (t>=3*TotTime/10) &&
            (t<=((3*TotTime/10)+(MaxStep*deltaT))) )
        {
            cerr<<"Completed "<<numSteps<<" time steps in t = "
                <<t<<" seconds; deltaT = "<<deltaT<<endl;
            ofstream Out30("30PERCENT.DAT");

            for(Zone = 1; Zone <= 3; Zone++)
            {
                Out30<<"VARIABLES = "<<" \"z\", \"r\", \"ne\", \"n1\",
                      \"n2\", \"n3\", \"nA\", \"Vr\", \"Vz\", \"Bt\",
                      \"Jencl\", \"p\", \"Te\", \"Th\" "<<"\n
                     <<"ZONE I="<<Zmax[Zone]-Zmin[Zone]+1<<", J="
                     <<Rmax[Zone]-Rmin[Zone]<<", F=POINT"<<"\n";
                for (J=Rmin[Zone]+1; J<=Rmax[Zone]; J++)
                {
                    for(K=Zmin[Zone]+1; K<=Zmax[Zone]+1; K++)
                    {
                        Out30<<(K-1)*deltaZ<<"\t"<<setw(7)
                             <<(J-1)*deltaR<<"\t"<<setw(7)<<ne[J][K]
                             <<"\t"<<setw(7)<<ni[J][K]<<"\t"<<setw(7)
                             <<nii[J][K]<<"\t"<<setw(7)<<niii[J][K]
                             <<"\t"<<setw(7)<<nA[J][K]<<"\t"<<setw(7)
                             <<Vr[J][K]<<"\t"<<setw(7)<<Vz[J][K]<<"\t"
                             <<setw(7)<<Bt[J][K]<<"\t"<<setw(7)
                             <<Jencl[J][K]<<"\t"<<setw(7)<<p[J][K]
                             <<"\t"<<setw(7)<<Te[J][K]<<"\t"<<setw(7)
                             <<Th[J][K]<<endl;
                     }
                 }
             }
         }//end of "30PERCENT" if...the else follows.
         else
         {
             if( (t>=2*TotTime/10) &&
                 (t<=((2*TotTime/10)+(MaxStep*deltaT))) )
             {
                 cerr<<"Completed "<<numSteps<<" time steps in t = "
                     <<t<<" seconds; deltaT = "<<deltaT<<endl;
                 ofstream Out20("20PERCENT.DAT");

                 for(Zone = 1; Zone <= 3; Zone++)
                 {
                     Out20<<"VARIABLES = "<<" \"z\", \"r\", \"ne\",
                           \"n1\", \"n2\", \"n3\", \"nA\", \"Vr\",
                           \"Vz\", \"Bt\", \"Jencl\", \"p\", \"Te\",
                           \"Th\" "<<"\n"<<"ZONE I="<<Zmax[Zone]-
                           Zmin[Zone]+1<<", J="<<Rmax[Zone]-
                           Rmin[Zone]<<", F=POINT"<<"\n";
                     for (J=Rmin[Zone]+1; J<=Rmax[Zone]; J++)
                     {
                         for(K=Zmin[Zone]+1; K<=Zmax[Zone]+1; K++)
                         {
                             Out20<<(K-1)*deltaZ<<"\t"<<setw(7)
                                  <<(J-1)*deltaR<<"\t"<<setw(7)
                                  <<ne[J][K]<<"\t"<<setw(7)<<ni[J][K]
                                  <<"\t"<<setw(7)<<nii[J][K]<<"\t"
                                  <<setw(7)<<niii[J][K]<<"\t"<<setw(7)
                                  <<nA[J][K]<<"\t"<<setw(7)<<Vr[J][K]
                                  <<"\t"<<setw(7)<<Vz[J][K]<<"\t"
                                  <<setw(7)<<Bt[J][K]<<"\t"<<setw(7)
                                  <<Jencl[J][K]<<"\t"<<setw(7)<<p[J][K]
                                  <<"\t"<<setw(7)<<Te[J][K]<<"\t"
                                  <<setw(7)<<Th[J][K]<<endl;
                         }
                     }
                 }
             }//end of "20PERCENT" if...the else follows.
//The tab allignments are intentionally offset.
    else
    {
        if( (t>=TotTime/10) &&
            (t<=((TotTime/10)+(MaxStep*deltaT))) )
        {
            cerr<<"Completed "<<numSteps<<" time steps in t = "<<t
                <<" seconds; deltaT = "<<deltaT<<endl;
            ofstream Out10("10PERCENT.DAT");
            for(Zone = 1; Zone <= 3; Zone++)
            {
                Out10<<"VARIABLES = "<<" \"z\", \"r\", \"ne\", \"n1\",
                      \"n2\", \"n3\", \"nA\", \"Vr\", \"Vz\", \"Bt\",
                      \"Jencl\", \"p\", \"Te\", \"Th\" "<<"\n"
                     <<"ZONE I="<<Zmax[Zone]-Zmin[Zone]+1<<", J="
                     <<Rmax[Zone]-Rmin[Zone]<<", F=POINT"<<"\n";

                for (J=Rmin[Zone]+1; J<=Rmax[Zone]; J++)
                {
                    for(K=Zmin[Zone]+1; K<=Zmax[Zone]+1; K++)
                    {
                        Out10<<(K-1)*deltaZ<<"\t"<<setw(7)
                             <<(J-1)*deltaR<<"\t"<<setw(7)<<ne[J][K]
                             <<"\t"<<setw(7)<<ni[J][K]<<"\t"<<setw(7)
                             <<nii[J][K]<<"\t"<<setw(7)<<niii[J][K]
                             <<"\t"<<setw(7)<<nA[J][K]<<"\t"<<setw(7)
                             <<Vr[J][K]<<"\t"<<setw(7)<<Vz[J][K]<<"\t"
                             <<setw(7)<<Bt[J][K]<<"\t"<<setw(7)
                             <<Jencl[J][K]<<"\t"<<setw(7)<<p[J][K]
                             <<"\t"<<setw(7)<<Te[J][K]<<"\t"<<setw(7)
                             <<Th[J][K]<<endl;
                     }
                 }
             }
         }//end of 10 percent loop.
         }//end of 20 percent loop.
         }//end of 30 percent loop.
         }//end of 40 percent loop.
         }//end of 50 percent loop.
         }//end of 60 percent loop.
         }//end of 70 percent loop.
         }//end of 80 percent loop.
         }//end of 90 percent loop.
    }//end of total-time loop.
}
//---------------------------------------
```

C.5 Grid

```
//-----------------------------------------------------------
//Grid.CPP; Written by Kameshwaran Sankaran
//-----------------------------------------------------------
void Grid()
{
/*
In the cell-centered scheme, there is one face more than the center.
In the channel:
    Faces:    j = Rcathode, ... , Ra
            k =    0, ... , EL
    Centers:J = Rcathode+1, ... , Ra
            K = 1, ... , EL

In the plume:
    Faces:    j = 0, ... , numR
            k =    EL, ..., numZ
    Centers:J = 1, ... , numR
            K = EL+1, ... , numZ
*/
    deltaR = RAD/double(numR);
    deltaZ = Z/double(numZ);
//These 0.5s are added for the following reason:
//to get a integer value of, for e.g, 3 from 2.99999999 and 3.00000001,
//using int() would give 2 and 3 respectively.
//To get 3 from both, add 0.5 to them and then int() them.
    Rc    = int((Rcathode/deltaR)+0.5);
    Ra    = int((Ranode/deltaR)+0.5);

    Za    = int((AnodeLength/deltaZ)+0.5);
    Zc    = int((CathodeLength/deltaZ)+0.5);

//Corners of ZONE1
    Rmin[1] = Rc;
    Rmax[1] = Ra;
    Zmin[1] = 0;
    if(Zc <= Za)
        Zmax[1] = Zc;
    else
        Zmax[1] = Za;

//Corners of ZONE2
    if(Zc <= Za)
        Rmin[2] = 0;
    else
        Rmin[2] = Rc;

    if(Zc <= Za)
        Rmax[2] = Ra;
    else
        Rmax[2] = numR;

    if(Zc <= Za)
        Zmin[2] = Zc;
    else
        Zmin[2] = Za;

    if(Zc <= Za)
        Zmax[2] = Za;
    else
        Zmax[2] = Zc;

//Corners of ZONE3
```

```
    Rmin[3] = 0;
    Rmax[3] = numR;
    if(Zc <= Za)
        Zmin[3] = Za;
    else
        Zmin[3] = Zc;
    Zmax[3] = numZ;
}

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

C.6 Initial Conditions

```
//-----------------------------------------------------------
//InitialConditions.CPP; Written by Kameshwaran Sankaran
//-----------------------------------------------------------
void SetInitial()
{
/*
//This is in case of starting after purely fluid flow.
        ifstream InRho("RhoInitial.DAT");
        ifstream InPress("PressInitial.DAT");
        ifstream InVr("VrInitial.DAT");
        ifstream InVz("VzInitial.DAT");

        for(Zone = 1; Zone <= 3; Zone++)
        {
            for(J=Rmin[Zone]+1; J<=Rmax[Zone]; J++)
            {
                for(K=Zmin[Zone]+1; K<=Zmax[Zone]; K++)
                {
                    InRho>>Rho[J][K];
                    InPress>>p[J][K];
                    InVr>>Vr[J][K];
                    InVz>>Vz[J][K];
                    gam[J][K]= 5.0/3.0;
                    pe[J][K] = 0.5*p[J][K];
                    ph[J][K] = pe[J][K];
                    Th[J][K] = ph[J][K]/((Rho[J][K]/mAr)*kBoltz);
                    Te[J][K] = Th[J][K];
                }
            }
        }
*/

//This is in case of restarting after some period of MHD flow.
        ifstream LoadNo("NoOut.DAT");
        ifstream LoadNe("NeOut.DAT");
        ifstream LoadTe("TeOut.DAT");
        ifstream LoadTh("ThOut.DAT");
        ifstream LoadPh("PhOut.DAT");
        ifstream LoadGam("GamOut.DAT");
        ifstream LoadVr("VrOut.DAT");
        ifstream LoadVz("VzOut.DAT");
        ifstream LoadBt("BtOut.DAT");

        for(Zone = 1; Zone <= 3; Zone++)
        {
            for(J=Rmin[Zone]+1; J<=Rmax[Zone]; J++)
            {
                for(K=Zmin[Zone]+1; K<=Zmax[Zone]; K++)
                {
                    LoadNo>>n[J][K];
                    LoadNe>>ne[J][K];
                    LoadTe>>Te[J][K];
                    LoadTh>>Th[J][K];
                    LoadPh>>ph[J][K];
                    LoadGam>>gam[J][K];
                    LoadVr>>Vr[J][K];
                    LoadVz>>Vz[J][K];
                    LoadBt>>Bt[J][K];
                    Rho[J][K] = n[J][K]*mAr;
                    pe[J][K]= ne[J][K]*kBoltz*Te[J][K];

                    VSq[J][K]    = (Vr[J][K]*Vr[J][K])
```

```
                              +(Vz[J][K]*Vz[J][K]);
                BSq[J][K]    = Bt[J][K]*Bt[J][K];
                p[J][K]        = pe[J][K] + ph[J][K];
                KE[J][K]    = 0.5*Rho[J][K]*VSq[J][K];
                Etherm[J][K]= p[J][K]/(gam[J][K]-1);
                E[J][K]        = Etherm[J][K]+KE[J][K]
                                 +(0.5*BSq[J][K]/Mu);
                E[J][K]        = (p[J][K]/(gam[J][K]-1))
                                +(0.5*Rho[J][K]*VSq[J][K])
                                +(0.5*BSq[J][K]/Mu);
            }
        }
    }

    ifstream LoadVolt("Voltage.DAT");
    LoadVolt>>Vappl;

    Saha();

    for(Zone = 1; Zone <= 3; Zone++)
    {
        for(J=Rmin[Zone]+1; J<=Rmax[Zone]; J++)
        {
            for(K=Zmin[Zone]+1; K<=Zmax[Zone]; K++)
            {
                u[0][J][K] = Rho[J][K];
                u[1][J][K] = Rho[J][K]*Vr[J][K];
                u[2][J][K] = Rho[J][K]*Vz[J][K];
                u[3][J][K] = Bt[J][K];
                u[4][J][K] = E[J][K];
            }
        }
    }

    ConvBound();
    EvalParam();
    DissBound();

    for(J=Rc+1; J<=Ra; J++)
    {
        aSq[J][0]    = gamConst*p[J][0]/Rho[J][0];
        CmSq[J][0]    = aSq[J][0]+(Bt[J][0]*Bt[J][0]
                       /(Mu*Rho[J][0]));
        Cf[J][0]    = sqrt(CmSq[J][0]);
//Now, Cf = Cfz. So Cfz is no longer computed
//Largest eigenvalue
        SpecRadZ[J][0]    = fabs(Vz[J][0]+Cf[J][0]);
    }

    RhoIn= Rho[Rc+25][1];
    VzIn = Vz[Rc+25][1];
    peIn = pe[Rc+25][1];
    TeIn = Te[Rc+25][1];
    phIn = ph[Rc+25][1];
    ThIn = Th[Rc+25][1];

  ConvergeDiff= fopen("ConvergeDiff.DAT","wb");
  fprintf(ConvergeDiff, "%s \t %s \t  %s \t %s \t  %s \t %s \t %s \t
                       %s \t %s \t %s \t  %s \t %s \t %s \n",
                "numSteps", "R", "Z", "u0Dmax", "R", "Z", "u1Dmax",
                "R", "Z", "u2Dmax","R", "Z", "u3Dmax", "R", "Z",
                "u4Dmax","u0Davg","u1Davg","u2Davg","u3Davg",
                "u4Davg");
  ConvergeVal = fopen("ConvergeVal.DAT","wb");
```

```
    fprintf(ConvergeVal, "%s \t %s \t  %s \t %s \t  %s \t %s \t %s \t
                         %s \t %s \t %s \t  %s \t %s \t %s \n",
                    "numSteps", "R", "Z", "u0Max", "R", "Z", "u1Max",
                    "R", "Z", "u2Max","R", "Z", "u3Max", "R", "Z",
                    "u4Max","u0Avg","u1Avg","u2Avg","u3Avg","u4Avg");

    Converge    = fopen("Converge.DAT","wb");
    fprintf(Converge, "%s \t %s \t  %s \t %s \t  %s \t %s \t %s \t
                       %s \t %s \t %s \t  %s \t %s \t %s \n",
                    "numSteps", "R", "Z", "DensDmax", "R", "Z",
                    "KEDmax", "R", "Z", "BtDmax", "R", "Z",
                    "EtotDmax", "R", "Z", "EthDmax", "DensDavg",
                    "KEDavg","BtDavg", "EtotDavg", "EthDavg");

}//End of function
//-------------------------------------
```

C.7 Boundary Conditions

```
//-----------------------------------------------------------
//BoundaryConditions.CPP; Written by Kameshwaran Sankaran
//-----------------------------------------------------------
void ConvBound()
{
//These are the boundary conditions for convective fluxes

//Backplate conditions. These variables are defined at
//an imaginary point K=0. Fluid enters the domain from
//a porous backplate at sonic conditions.
    for(J=Rc+1; J<=Ra; J++)
    {
//There is no real need to compute the variables here,
//but it makes the code for specifying the fluxes neater.
        Te[J][0] = 15000.0;
        Th[J][0] = 15000.0;
        Vz[J][0] = sqrt(gamConst*kBoltz*(Te[J][0]+Th[J][0])/mAr);
        VSq[J][0]= Vz[J][0]*Vz[J][0];
        Rho[J][0]= MassFlowRate/(Vz[J][0]*PI*(pow(Ranode,2.0)
                                     -pow(Rcathode,2.0)));
        n[J][0]  = Rho[J][0]/mAr;
        ne[J][0] = 0.999*n[J][0];
        pe[J][0] = ne[J][0]*kBoltz*Te[J][0];
        ph[J][0] = n[J][0]*kBoltz*Th[J][0];
//For the given Th & pe, this the value of gamma is 1.13.
        gam[J][0]= 1.13;
        p[J][0]  = pe[J][0]+ph[J][0];
//The field diffuses instantaneously inside an insulator.
        Bt[J][0] = Bt[J][1];
        BSq[J][0]= Bt[J][0]*Bt[J][0];

//Now, the boundary condition for the fluxes. The point k=0
//corresponds to the physical backplate boundary.
        Hz[0][J][0]    = Rho[J][0]*Vz[J][0];

        Hz[1][J][0] = 0.0;

        Hz[2][J][0] = (Rho[J][0]*VSq[J][0])+p[J][0]
                      +(0.5*BSq[J][0]/Mu);

//The inductive drop is also included in the dissipative part,
//becuase it is evaluated more frequently.
        Hz[3][J][0] = 0.0;

        E[J][0]    = (p[J][0]/(gam[J][0]-1))
                     +(0.5*Rho[J][0]*VSq[J][0])
                     +(0.5*BSq[J][0]/Mu);
        Hz[4][J][0] = Vz[J][0]*(E[J][0]+p[J][0]
                              +(0.5*BSq[J][0]/Mu));
    }

//Anode inner.
    for(K=1; K<=Za; K++)
    {
        Hr[0][Ra][K] = 0.0;
//      Hr[1][Ra][K] = P[Ra][K];
        Hr[1][Ra][K] = Hr[1][Ra-1][K]
                      -(deltaR*SourceConv[1][Ra-1][K]);
        Hr[2][Ra][K] = 0.0;
        Hr[3][Ra][K] = 0.0;
        Hr[4][Ra][K] = 0.0;
    }
```

```
//Anode tip.
    for(J=Ra+1; J<=numR; J++)
    {
        Hz[0][J][Za] = 0.0;
        Hz[1][J][Za] = 0.0;
        Hz[2][J][Za] = P[J][Za+1];
        Hz[3][J][Za] = 0.0;
        Hz[4][J][Za] = 0.0;

        EnFluxZ[J][Za] = 0.0;
    }

//Upper freestream.
    for(K=Za+1; K<=numZ; K++)
    {
        Rho[numR+1][K]    = Rho[numR][K];
        n[numR+1][K]    = n[numR][K];
        ne[numR+1][K]    = ne[numR][K];
        Vr[numR+1][K]    = Vr[numR][K];
        Vz[numR+1][K]    = Vz[numR][K];
        Bt[numR+1][K]    = 0.0;
        E[numR+1][K]    = E[numR][K];

        VSq[numR+1][K]    = (Vr[numR+1][K]*Vr[numR+1][K])
                           +(Vz[numR+1][K]*Vz[numR+1][K]);
        BSq[numR+1][K]    = 0.0;
        gam[numR+1][K]    = gam[numR][K];
        p[numR+1][K]    = p[numR][K];
        P[numR+1][K]    = p[numR+1][K];
        EintEl[numR+1][K]= EintEl[numR][K];
        pe[numR+1][K]    = pe[numR][K];
        Te[numR+1][K]    = Te[numR][K];
        EintH[numR+1][K]= EintH[numR][K];
        ph[numR+1][K]    = ph[numR][K];
        Th[numR+1][K]    = Th[numR][K];

        Fr[0][numR+1][K] = Rho[numR+1][K]*Vr[numR+1][K];
        Fr[1][numR+1][K] = (Rho[numR+1][K]*Vr[numR+1][K]
                              *Vr[numR+1][K])+P[numR+1][K];
        Fr[2][numR+1][K] = Rho[numR+1][K]*Vr[numR+1][K]
                                         *Vz[numR+1][K];
        Fr[3][numR+1][K] = Bt[numR+1][K]*Vr[numR+1][K];
        Fr[4][numR+1][K] = Vr[numR+1][K]*(E[numR+1][K]
                                         +P[numR+1][K]);
    }

//Exit.
    for(J=numR; J>=1; J--)
    {
        Rho[J][numZ+1]    = Rho[J][numZ];
        n[J][numZ+1]    = n[J][numZ];
        ne[J][numZ+1]    = ne[J][numZ];
        Vr[J][numZ+1]    = Vr[J][numZ];
        Vz[J][numZ+1]    = Vz[J][numZ];
        Bt[J][numZ+1]    = 0.0;
        Jencl[J][numZ+1]= 0.0;
        E[J][numZ+1]    = E[J][numZ];
        gam[J][numZ+1]    = gam[J][numZ];
        EintEl[J][numZ+1]=EintEl[J][numZ];
        pe[J][numZ+1]    = pe[J][numZ];
        Te[J][numZ+1]    = Te[J][numZ];
        EintH[J][numZ+1]= EintH[J][numZ];
        Th[J][numZ+1]    = Th[J][numZ];
        ph[J][numZ+1]    = ph[J][numZ];
        Potential[J][numZ+1]= Potential[J][numZ];

        VSq[J][numZ+1]    = VSq[J][numZ];
        BSq[J][numZ+1]    = 0.0;
        p[J][numZ+1]    = p[J][numZ];
        P[J][numZ+1]    = p[J][numZ+1];

        Fz[0][J][numZ+1] = Rho[J][numZ+1]*Vz[J][numZ+1];
        Fz[1][J][numZ+1] = Rho[J][numZ+1]*Vr[J][numZ+1]
                                         *Vz[J][numZ+1];
        Fz[2][J][numZ+1] = (Rho[J][numZ+1]*Vz[J][numZ+1]
                            *Vz[J][numZ+1])+P[J][numZ+1];
        Fz[3][J][numZ+1] = Bt[J][numZ+1]*Vz[J][numZ+1];
        Fz[4][J][numZ+1] = Vz[J][numZ+1]*(E[J][numZ+1]
                                         +P[J][numZ+1]);
    }

//Centerline.
    for(K=numZ; K>=Zc+1; K--)
    {
        Hr[0][0][K] = 0.0;
        Hr[1][0][K] = P[1][K];
        Hr[2][0][K] = 0.0;
        Hr[3][0][K] = 0.0;
        Hr[4][0][K] = 0.0;

        EnFluxR[0][K] = 0.0;

        SourceConv[0][0][K] = 0.0;
        SourceConv[1][0][K] = 0.0;
        SourceConv[2][0][K] = 0.0;
        SourceConv[3][0][K] = 0.0;
        SourceConv[4][0][K] = 0.0;
    }

//Cathode tip.
    for(J=1; J<=Rc; J++)
    {
        Hz[0][J][Zc] = 0.0;
        Hz[1][J][Zc] = 0.0;
        Hz[2][J][Zc] = P[J][Zc+1];
        Hz[3][J][Zc] = 0.0;
        Hz[4][J][Zc] = 0.0;
    }

//Cathode outer.
    for(K=Zc; K>=1; K--)
    {
        Hr[0][Rc][K] = 0.0;
        Hr[1][Rc][K] = Hr[1][Rc+1][K];
//      Hr[1][Rc][K] = P[Rc+1][K];
        Hr[2][Rc][K] = 0.0;
        Hr[3][Rc][K] = 0.0;
        Hr[4][Rc][K] = 0.0;

        EnFluxR[Rc][K] = 0.0;

        SourceConv[0][Rc][K] = 0.0;
        SourceConv[1][Rc][K] = 0.0;
        SourceConv[2][Rc][K] = 0.0;
        SourceConv[3][Rc][K] = 0.0;
        SourceConv[4][Rc][K] = 0.0;
    }
```

```
}//End of function

//-------------------------------------
void DissBound()
{
//Backplate
    if(fabs(Jback) < Jmax)
        Vappl += 0.0005;
    else
        Vappl -= 0.0005;

    for(J=Rc+1; J<=Ra; J++)
    {
        DissipZ[0][J][0] = 0.0;
        DissipZ[1][J][0] = 0.0;
        DissipZ[2][J][0] = 0.0;
//Total (inductive+resistive) voltage drop at the backplate.
        DissipZ[3][J][0] = Vappl/(log(Ranode/Rcathode)
                               *((J-0.5)*deltaR));
        DissipZ[4][J][0] = -((-Vappl/(log(Ranode/Rcathode)
                                   *((J-0.5)*deltaR)) )
                          -(Vz[J][0]*Bt[J][1]))*Bt[J][1]/Mu;

//Boundary conditions for the species energy equations.
        Te[J][0]    = 15000.0;
        Th[J][0]    = 15000.0;
        Vz[J][0]    = sqrt(gamConst*kBoltz*(Te[J][0]+Th[J][0])/mAr);
        VSq[J][0]    = Vz[J][0]*Vz[J][0];
        Rho[J][0]    = MassFlowRate/(Vz[J][0]*PI*(pow(Ranode,2.0)
                                             -pow(Rcathode,2.0)));
        n[J][0]    = Rho[J][0]/mAr;
        ne[J][0]    = 0.99*n[J][0];
        pe[J][0]    = ne[J][0]*kBoltz*Te[J][0];
        ph[J][0]    = n[J][0]*kBoltz*Th[J][0];
        p[J][0]    = pe[J][0]+ph[J][0];
//The field diffuses instantaneously inside an insulator.
        Bt[J][0]    = Bt[J][1];
        BSq[J][0]    = Bt[J][0]*Bt[J][0];

        EintEl[J][0]    = pe[J][0]/(gamConst-1);
        EnFluxZ[J][0]    = Vz[J][0]*(EintEl[J][0]+pe[J][0]);
    }

//Anode inner.
    for(K=0; K<=Za; K++)
    {
        DissipR[0][Ra][K]    = 0.0;
        DissipR[1][Ra][K]    = 0.0;
        DissipR[2][Ra][K]    = 0.0;

//Bt[Ra+1][K]=0.
        DissipR[3][Ra][K]    = ResTungst*((-Bt[Ra][K]/deltaR)
                                +(Bt[Ra][K]/((Ra-0.5)*deltaR)))/Mu;

        ElecTemp = 2500.0;
        ElCondR[Ra][K]    = kTherm[Ra][K]*(ElecTemp-Te[Ra][K])/deltaR;
        IonCondR[Ra][K]   = kIon[Ra][K]*(ElecTemp-Th[Ra][K])/deltaR;
        ThermCondR[Ra][K] = ElCondR[Ra][K] + IonCondR[Ra][K];

//Bt[Ra+1][K]=0.
        DissipR[4][Ra][K] = (DissipR[3][Ra][K]*0.5*Bt[Ra][K]/Mu)
                           + ThermCondR[Ra][K];

        EnFluxR[Ra][K]    = 0.0;
    }

//Anode tip.
    for(J=Ra+1; J<=numR; J++)
    {
        DissipZ[0][J][Za]= 0.0;
        DissipZ[1][J][Za]= 0.0;
        DissipZ[2][J][Za]= 0.0;

//Bt[J][Za] = 0.
        DissipZ[3][J][Za]= ResTungst*(Bt[J][Za+1]/deltaZ)/Mu;

        ElecTemp = 2500.0;
        ElCondZ[J][Za]   = kTherm[J][Za+1]*(Te[J][Za+1]-ElecTemp)
                                             /deltaZ;
        IonCondZ[J][Za]  = kIon[J][Za+1]*(Th[J][Za+1]-ElecTemp)
                                             /deltaZ;
        ThermCondZ[J][Za]= ElCondZ[J][Za] + IonCondZ[J][Za];

//Bt[J][Za] = 0.
        DissipZ[4][J][Za]= (DissipZ[3][J][Za]*0.5*Bt[J][Za+1]/Mu)
                           +ThermCondZ[J][Za];

        EnFluxZ[J][Za]   = 0.0;
    }

//Upper freestream.
    for(K=Za+1; K<=numZ; K++)
    {
        Rho[numR+1][K] = Rho[numR][K];
        n[numR+1][K]   = n[numR][K];
        ne[numR+1][K]  = ne[numR][K];
        Vr[numR+1][K   = Vr[numR][K];
        Vz[numR+1][K]  = Vz[numR][K];
        Bt[numR+1][K]  = 0.0;
        E[numR+1][K]   = E[numR][K];
        VSq[numR+1][K] = (Vr[numR+1][K]*Vr[numR+1][K])
                        +(Vz[numR+1][K]*Vz[numR+1][K]);
        BSq[numR+1][K] = 0.0;
        p[numR+1][K]   = p[numR][K];
        EintEl[numR+1][K]= EintEl[numR][K];
        pe[numR+1][K]  = pe[numR][K];
        Te[numR+1][K]  = Te[numR][K];
        ph[numR+1][K]  = ph[numR][K];
        Th[numR+1][K]  = Th[numR][K];

        kTherm[numR+1][K]= kTherm[numR][K];
        kIon[numR+1][K]= kIon[numR][K];
        Res[numR+1][K] = Res[numR][K];

        EintEl[numR+1][K]= EintEl[numR][K];
        FrEn[numR+1][K]= Vr[numR+1][K]*(EintEl[numR+1][K]
                                       +pe[numR+1][K]);
    }

//Exit.
    for(J=numR; J>=1; J--)
    {
        Rho[J][numZ+1]    = Rho[J][numZ];
        n[J][numZ+1]    = n[J][numZ];
        ne[J][numZ+1]    = ne[J][numZ];
        Vr[J][numZ+1]    = Vr[J][numZ];
        Vz[J][numZ+1]    = Vz[J][numZ];
        Bt[J][numZ+1]    = 0.0;
//This is for the sake of the output file.
        Jencl[J][numZ+1]= 0.0;
        E[J][numZ+1]    = E[J][numZ];
        gam[J][numZ+1]    = gam[J][numZ];
        EintEl[J][numZ+1]=EintEl[J][numZ];
        pe[J][numZ+1]    = pe[J][numZ];
        Te[J][numZ+1]    = Te[J][numZ];
        EintH[J][numZ+1]= EintH[J][numZ];
        Th[J][numZ+1]    = Th[J][numZ];
        ph[J][numZ+1]    = ph[J][numZ];
        Potential[J][numZ+1]= Potential[J][numZ];

        VSq[J][numZ+1]    = VSq[J][numZ];
        BSq[J][numZ+1]    = 0.0;
        p[J][numZ+1]    = p[J][numZ];
        P[J][numZ+1]    = p[J][numZ+1];

        E[J][numZ+1]    = E[J][numZ];
        gam[J][numZ+1]    = gam[J][numZ];
        EintEl[J][numZ+1]=EintEl[J][numZ];
        pe[J][numZ+1]    = pe[J][numZ];
        Te[J][numZ+1]    = Te[J][numZ];
        EintH[J][numZ+1]= EintH[J][numZ];
        Th[J][numZ+1]    = Th[J][numZ];
        ph[J][numZ+1]    = ph[J][numZ];
        Potential[J][numZ+1]= Potential[J-1][numZ]
                             +((0.5*(DissipZ[3][J][numZ]
                                    +DissipZ[3][J][K-1]))-
                            (Vz[J][numZ]*Bt[J][numZ]));

        EIcollFreq[J][numZ+1]= EIcollFreq[J][numZ];
        kTherm[J][numZ+1]= kTherm[J][numZ];
        kIon[J][numZ+1]    = kIon[J][numZ];
        Res[J][numZ+1]    = Res[J][numZ];

        FzEn[J][numZ+1]  = Vz[J][numZ+1]*(EintEl[J][numZ+1]
                                         +pe[J][numZ+1]);
    }

//Centerline.
    for(K=numZ; K>=Zc+1; K--)
    {
        DissipR[0][0][K]    = 0.0;
        DissipR[1][0][K]    = 0.0;
        DissipR[2][0][K]    = 0.0;

//Symmetry implies Bt[1][K] = -Bt[-1][K].
        if(fabs(Bt[1][K]) > eps)
            DissipR[3][0][K]= Res[1][K]*(4*Bt[1][K]/deltaR)/Mu;
        else
            DissipR[3][0][K]= 0.0;

        DissipR[4][0][K]    = 0.0;

        SourceDiss[0][0][K]    = 0.0;
        SourceDiss[1][0][K]    = 0.0;
        SourceDiss[2][0][K]    = 0.0;
        SourceDiss[3][0][K]    = 0.0;
        SourceDiss[4][0][K]    = 0.0;

        EnSourceR[0][K] = 0.0;
    }

//Cathode tip.
```

```
    for(J=1; J<=Rc; J++)
    {
        DissipZ[0][J][Zc] = 0.0;
        DissipZ[1][J][Zc] = 0.0;
        DissipZ[2][J][Zc] = 0.0;

//Bt[J][Zc] = 0.
        DissipZ[3][J][Zc] = ResTungst*(Bt[J][Zc+1]/deltaZ)/Mu;

        ElecTemp = 2500.0;
        ElCondZ[J][Zc] = kTherm[J][Zc+1]*(Te[J][Zc+1]-ElecTemp)
                                          /deltaZ;
        IonCondZ[J][Zc] = kIon[J][Zc+1]*(Th[J][Zc+1]-2500)
                                          /deltaZ;
        ThermCondZ[J][Zc] = ElCondZ[J][Zc] + IonCondZ[J][Zc];

//Bt[J][Zc] = 0.
        DissipZ[4][J][Zc] = (DissipZ[3][J][Zc]*0.5*Bt[J][Zc+1]/Mu)
                            +ThermCondZ[J][Zc];

        EnFluxZ[J][Zc] = 0.0;
    }

//Cathode outer.
    for(K=Zc; K>=0; K--)
    {
        DissipR[0][Rc][K] = 0.0;
        DissipR[1][Rc][K] = 0.0;
        DissipR[2][Rc][K] = 0.0;

//Bt[Rc][K] = 0.
        DissipR[3][Rc][K] = ResTungst*(Bt[Rc+1][K]/deltaR)/Mu;

        ElecTemp = 2500.0;
        ElCondR[Rc][K] = kTherm[Rc+1][K]*(Te[Rc+1][K]-ElecTemp)/deltaR;
        IonCondR[Rc][K]= kIon[Rc+1][K]*(Th[Rc+1][K]-5000)/deltaR;
        ThermCondR[Rc][K] = ElCondR[Rc][K] + IonCondR[Rc][K];

//Bt[Rc][K] = 0.
        DissipR[4][Rc][K] = (DissipR[3][Rc][K]*0.5*Bt[Rc+1][K]/Mu)
                            +ThermCondR[Rc][K];

        SourceDiss[0][Rc][K]    = 0.0;
        SourceDiss[1][Rc][K]    = 0.0;
        SourceDiss[2][Rc][K]    = 0.0;
        SourceDiss[3][Rc][K]    = 0.0;
        SourceDiss[4][Rc][K]    = DissipR[4][Rc][K]/((Rc+1)*deltaR);

        EnSourceR[Rc][K] = 0.0;
    }
}
//-------------------------------------
```

C.8 Solve

```
//------------------------------------------------------------
//Solve.CPP; Written by Kameshwaran Sankaran
//------------------------------------------------------------
void Solve()
{
    if(t >= TotTime)
//This is only to deal with the case TotTime <= 0.0.
    {
        ReCalculate();
        WriteFile();
    }
    while(t < TotTime)
    {
//This is updated only at convective time scales.
        EvalTimeStep();

//Convective fluxes are calculated only at convective time scales.
        EvalConv();

//Calculates "EvalDiss()", updates "u[]",  and increments time.
        TimeMarch();

//These variables are computed only at convective time scales.
        EvalParam();

//Verifies if it is time to write to a file, and then does so.
        WriteFile();
    }//finished computing for "TotTime".

//Closing the files for verifying convergence.
    fclose(ConvergeDiff);
    fclose(ConvergeVal);
    fclose(Converge);
}
//----------------------------------------
```

C.9 Time Steps

```
//---------------------------------------------------------
//SetTimeStep.CPP; Written by Kameshwaran Sankaran
//---------------------------------------------------------
void EvalTimeStep()
{

//The time step is chosen to satisfy the CFL criterion.
//For the hyperbolic part, deltaT = C * (GridSize/Max.wave speed)
    Lmax = SpecRadZ[Rc+25][0];//Initial guess.
//SpecRadR and SpecRadZ are at centers. So, the face values are
//obtained by averaging them.
    for(Zone = 1; Zone <= 3; Zone++)
    {
        for(j=J=Rmin[Zone]+1; j<=Rmax[Zone]-1; j++,J++)
        {
            for(K=Zmin[Zone]+1; K<=Zmax[Zone]; K++)
            {
                if (0.5*(SpecRadR[J][K]+SpecRadR[J+1][K]) > Lmax)
                {
                Lmax = 0.5*(SpecRadR[J][K]+SpecRadR[J+1][K]);
                }
            }
        }
        for(J=Rmin[Zone]+1; J<=Rmax[Zone]; J++)
        {
            for(k=K=Zmin[Zone+1]; k<=Zmax[Zone]; k++,K++)
            {
                if (0.5*(SpecRadZ[J][K]+SpecRadZ[J][K+1]) > Lmax)
                {
                    Lmax = 0.5*(SpecRadZ[J][K]+SpecRadZ[J][K+1]);
                }
            }
        }
    }//end for all zones.

    deltaTconv    = 0.6*deltaR/Lmax;
    deltaT    = deltaTconv;

//For thermal conduction, VNSA dictates that
//deltaT <= 0.25*n[J][K]*kBoltz*deltaR*deltaR/kTherm[J][K].

    deltaTtherm = 5.0e-9;//Initial guess.

    for(Zone = 1; Zone <= 3; Zone++)
    {
        for (J=Rmin[Zone]+1; J<=Rmax[Zone]; J++)
        {
            for(K=Zmin[Zone]+1; K<=Zmax[Zone]; K++)
            {
                if(0.2*0.25*n[J][K]*kBoltz*deltaR*deltaR/kTherm[J][K]
                      < deltaTtherm)
                   deltaTtherm = 0.2*0.25*n[J][K]*kBoltz*deltaR*deltaR
                              /kTherm[J][K];
            }
        }
    }
    if (deltaTtherm < deltaT)
    {
        deltaT = deltaTtherm;
    }

//For resistive diffusion, VNSA dictates that
//deltaT <= 0.25*Mu*deltaR*deltaR/Res[J][K].

    deltaTresist = 5.0e-10;//Initial guess.

    for(Zone = 1; Zone <= 3; Zone++)
    {
        for (J=Rmin[Zone]+1; J<=Rmax[Zone]; J++)
        {
            for(K=Zmin[Zone]+1; K<=Zmax[Zone]; K++)
            {
                if(0.2*0.25*Mu*deltaR*deltaR/Res[J][K] < deltaTresist)
                    deltaTresist =0.2*0.25*Mu*deltaR*deltaR/Res[J][K];
            }
        }
    }
    if (deltaTresist < deltaT)
    {
        deltaT = deltaTresist;
    }

//This is if substepping is used.
//    if(deltaTtherm > deltaTresist)
//        MaxStep = int((deltaTconv/deltaTtherm)/10);
//    else
//        MaxStep = int((deltaTconv/deltaTresist)/10);

//    if(MaxStep < 1)
        MaxStep = 1;
//    if(MaxStep > 5)
//        MaxStep = 5;
}
//-------------------------------------
```

C.10 Convective Fluxes

```
//-----------------------------------------------------------
//EvalConv.CPP; Written by Kameshwaran Sankaran
//-----------------------------------------------------------
void EvalConv()
{
//IMPORTANT:
//Refer to notes about indexing scheme for the fluxes.

//This is the cell center value of convective fluxes. The
//cell face values are given by Hr and Hz.
    EvalF();
//This is the cell face value of numerical dissipation.
    EvalNumDiss();

//While computing SourceConv, remember:
//The 'r' is not from the variable, but from the upper flux
//surface, and the term is from the lower flux surface.
//From the implementation in EvalF, SourceR evaluates the
//term at 'J' = 'j-0.5'. The value at 'j' is needed.

    for( i=0; i<5; i++)
    {
        for(Zone = 1; Zone <= 3; Zone++)
        {
            for(j=J=Rmin[Zone]+1; j<=Rmax[Zone]; j++,J++)
            {
                for (K=Zmin[Zone]+1; K<=Zmax[Zone]; K++)
                {
                    Hr[i][j][K] = (0.5*(Fr[i][J+1][K]+Fr[i][J][K]))
                                - Dr[i][j][K];
                    SourceConv[i][j][K] = (0.5*(SourceR[i][J+1][K]
                               +SourceR[i][J][K]))/((j+1)*deltaR);
                }
            }
            for(J=Rmin[Zone]+1; J<=Rmax[Zone]; J++)
            {
                for (k=K=Zmin[Zone]+1; k<=Zmax[Zone]; k++,K++)
                {
                    Hz[i][J][k] = (0.5*(Fz[i][J][K+1]+Fz[i][J][K]))
                                 - Dz[i][J][k];
                }
            }
        }
    }

    ConvBound();//Calculates the convective terms at the boundaries.


    for(i=0; i<5; i++)
    {
        for(Zone = 1; Zone <= 3; Zone++)
        {
            for (j=J=Rmin[Zone]+1; J<=Rmax[Zone]; j++,J++)
            {
                for(k=K=Zmin[Zone]+1; K<=Zmax[Zone]; k++,K++)
                {
                    ConvFlux[i][J][K] = ((Hr[i][j][K]-Hr[i][j-1][K])
                                        /deltaR)+SourceConv[i][j-1][K]
                                      + ((Hz[i][J][k]-Hz[i][J][k-1])
                                        /deltaZ);
                }
            }
        }
    }
}

//---------------------------------------

void EvalF()
{
//The variables are stored at cell centers. So, ’J’ refers to
//’j-0.5’ and ’K’ refers to ’k-0.5’.
    for(Zone = 1; Zone <= 3; Zone++)
    {
        for (J=Rmin[Zone]+1; J<=Rmax[Zone]; J++)
        {
            for (K=Zmin[Zone]+1; K<=Zmax[Zone]; K++)
            {
//Fluxes in the "r" direction
                Fr[0][J][K] = Rho[J][K]*Vr[J][K];
                Fr[1][J][K] = (Rho[J][K]*Vr[J][K]*Vr[J][K])
                           +p[J][K]+(0.5*Bt[J][K]*Bt[J][K]/Mu);
                Fr[2][J][K] = Rho[J][K]*Vr[J][K]*Vz[J][K];
                Fr[3][J][K] = Bt[J][K]*Vr[J][K];
                Fr[4][J][K] = Vr[J][K]*(E[J][K]+P[J][K]);

//Source terms due to the 1/r dependence.
                SourceR[0][J][K] = Rho[J][K]*Vr[J][K];
                SourceR[1][J][K] = (Rho[J][K]*Vr[J][K]*Vr[J][K])
                                 +(Bt[J][K]*Bt[J][K]/Mu);
                SourceR[2][J][K] = Rho[J][K]*Vr[J][K]*Vz[J][K];
                SourceR[3][J][K] = 0.0;
                SourceR[4][J][K] = Vr[J][K]*(E[J][K]+P[J][K]);

//Fluxes in the "z" direction
                Fz[0][J][K] = Rho[J][K]*Vz[J][K];
                Fz[1][J][K] = Rho[J][K]*Vr[J][K]*Vz[J][K];
                Fz[2][J][K] = (Rho[J][K]*Vz[J][K]*Vz[J][K])
                            +p[J][K]+(0.5*Bt[J][K]*Bt[J][K]/Mu);
                Fz[3][J][K] = Bt[J][K]*Vz[J][K];
                Fz[4][J][K] = Vz[J][K]*(E[J][K]+P[J][K]);
            }
        }
    }
}

//---------------------------------------
void EvalNumDiss()
{
//Numerical dissipation.

    EvalLim();

//The numerical dissipation through boundaries should be zero.
//Thus, the values at boundaries are overwritten
//by 0 in BoundaryConditions.

    ExtraMemory++;//First time it is called, it is =1. Then it is >1.

    if(ExtraMemory ==1)
        AllocMemoryEXTRA();

    for(i=0; i<5;i++)
    {
        for(Zone=1; Zone<=3; Zone++)
        {
            for (j=Rmin[Zone]+1; j<=Rmax[Zone]-1; j++)
            {
                for (k=Zmin[Zone]+1; k<=Zmax[Zone]; k++)
                {
                    RdelU[i][j][K]    = u[i][J+1][K] - u[i][J][K];
                }
            }
            for (j=Rmin[Zone]+1; j<=Rmax[Zone]; j++)
            {
                for (k=Zmin[Zone]+1; k<=Zmax[Zone]; k++)
                {
                    ZdelU[i][J][k]    = u[i][J][K+1] - u[i][J][K];
                }
            }
        }
    }//end of ’i’ loop.

//CHAMBER
    EvalA(1);

    MatTimesVec(AbsA, RdelU, Dr, 1);
    MatTimesVec(AbsB, ZdelU, Dz, 1);

//PLUME
    EvalA(2);

    MatTimesVec(AbsA, RdelU, Dr, 2);
    MatTimesVec(AbsB, ZdelU, Dz, 2);

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

    for(i=0; i<5;i++)
    {
        for(Zone = 1; Zone <= 3; Zone++)
        {
            for(j=J=Rmin[Zone]+1; j<=Rmax[Zone]-1; j++,J++)
            {
                for (K=Zmin[Zone]+1; K<=Zmax[Zone]; K++)
                {
                    RdelU[i][j][K] = u[i][J+1][K] - u[i][J][K];
                    Dr[i][j][K]=0.5*0.5*(SpecRadR[J][K]
                                  +SpecRadR[J+1][K])
                            *((RdelU[i][j][K])-Lr[i][j][K]);
                }
            }
            for(J=Rmin[Zone]+1; J<=Rmax[Zone]; J++)
            {
                for (k=K=Zmin[Zone]+1; k<=Zmax[Zone]; k++,K++)
                {
                    ZdelU[i][J][k]= u[i][J][K+1] - u[i][J][K];
                    Dz[i][J][k]=0.5*0.5*(SpecRadZ[J][K]
                                  +SpecRadZ[J][K+1])
                            *((ZdelU[i][J][k])-Lz[i][J][k]);
                }
            }
        }
    }
****************/

}
//-------------------------------------

void EvalLim()
{
```

```
//This stupid compiler doesn’t have min(), max() functions.

    EvalS();

//Alpha-bee
    double a, b, aN, bN, d, q;

    q=0.3;

    for (i=0; i<5; i++)
    {
        for(Zone = 1; Zone <= 3; Zone++)
        {
            for (j=Rmin[Zone]+1; j<=Rmax[Zone]-2; j++)
            {
                for(k=Zmin[Zone]+1; k<=Zmax[Zone]-2; k++)
                {
//"r" direction
                    a = q*fabs(u[i][j+2][k] - u[i][j+1][k]);
                    b = fabs(u[i][j][k] - u[i][j-1][k]);

                    if (a<b)
                        aN = a;
                    else
                        aN = b;

                    a = fabs(u[i][j+2][k] - u[i][j+1][k]);
                    b = q*fabs(u[i][j][k] - u[i][j-1][k]);

                    if (a<b)
                        bN = a;
                    else
                        bN = b;

                    if (aN > bN)
                        d = aN;
                    else
                        d = bN;

                    Lr[i][j][k] = Sr[i][j][k]*d;
//"z" direction
                    a = q*fabs(u[i][j][k+2] - u[i][j][k+1]);
                    b = fabs(u[i][j][k] - u[i][j][k-1]);

                    if (a<b)
                        aN = a;
                    else
                        aN = b;

                    a = fabs(u[i][j][k+2] - u[i][j][k+1]);
                    b = q*fabs(u[i][j][k] - u[i][j][k-1]);

                    if (a<b)
                        bN = a;
                    else
                        bN = b;

                    if (aN > bN)
                        d = aN;
                    else
                        d = bN;

                    Lz[i][j][k] = Sz[i][j][k]*d;
                }
            }
        }//end for all zones.
    }//end for all "i".
}

//-------------------------------------

void EvalS()
{
    double t1, t2;

    for( i=0; i<5; i++)
    {
        for(Zone = 1; Zone <= 3; Zone++)
        {
            for( j=Rmin[Zone]+1; j<=Rmax[Zone]-2; j++)
            {
                for(k=Zmin[Zone]+1; k<=Zmax[Zone]-2; k++)
                {
//"r" direction
                    t1 = EvalSign(u[i][j+2][k] - u[i][j+1][k]);
                    t2 = EvalSign(u[i][j][k] - u[i][j-1][k]);
                    Sr[i][j][k] = 0.5*(t1+t2);
//"z" direction
                    t1 = EvalSign(u[i][j][k+2] - u[i][j][k+1]);
                    t2 = EvalSign(u[i][j][k] - u[i][j][k-1]);
                    Sz[i][j][k] = 0.5*(t1+t2);
                }
            }
        }
    }
}

//-------------------------------------
double EvalSign(double x)
{
    double sign;

    if (x == 0)
        sign = 0.0;
    else
         sign = x/fabs(x);

    return sign;
}

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

C.11 Eigensystem

```
//----------------------------------------------------------
//Eigensystem.CPP; Written by Kameshwaran Sankaran
//----------------------------------------------------------
void EvalLambdaR(int Zone)
{
    for (j=Rmin[Zone]+1; j<=Rmax[Zone]; j++)
    {
    for (k=Zmin[Zone]+1; k<=Zmax[Zone]; k++)
    {
//These are the eigenvalues of the Jacobian
        L1[j][k] = Vr[j][k]-Cf[j][k];
        L2[j][k] = Vr[j][k];
        L3[j][k] = Vr[j][k];
        L4[j][k] = Vr[j][k];
        L5[j][k] = Vr[j][k]+Cf[j][k];

//Splitting into positive values...
        L1p[j][k] = (L1[j][k] + fabs(L1[j][k]))/2;
        L2p[j][k] = (L2[j][k] + fabs(L2[j][k]))/2;
        L3p[j][k] = (L3[j][k] + fabs(L3[j][k]))/2;
        L4p[j][k] = (L4[j][k] + fabs(L4[j][k]))/2;
        L5p[j][k] = (L5[j][k] + fabs(L5[j][k]))/2;

//Splitting into negative values...
        L1m[j][k] = (L1[j][k] - fabs(L1[j][k]))/2;
        L2m[j][k] = (L2[j][k] - fabs(L2[j][k]))/2;
        L3m[j][k] = (L3[j][k] - fabs(L3[j][k]))/2;
        L4m[j][k] = (L4[j][k] - fabs(L4[j][k]))/2;
        L5m[j][k] = (L5[j][k] - fabs(L5[j][k]))/2;

//Creating the diagonal matrix with positive eigenvalues...
        for (row =0; row<5; row++)
        {
            for (col=0; col<5; col++)
            {
                Lplus[row][col][j][k] = 0.0;
            }
        }

        Lplus[0][0][j][k] = L1p[j][k];
        Lplus[1][1][j][k] = L2p[j][k];
        Lplus[2][2][j][k] = L3p[j][k];
        Lplus[3][3][j][k] = L4p[j][k];
        Lplus[4][4][j][k] = L5p[j][k];

//Creating the diagonal matrix with negative eigenvalues...
        for (row =0; row<5; row++)
        {
            for (col =0; col<5; col++)
            {
                Lminus[row][col][j][k] = 0.0;
            }
        }

        Lminus[0][0][j][k] = L1m[j][k];
        Lminus[1][1][j][k] = L2m[j][k];
        Lminus[2][2][j][k] = L3m[j][k];
        Lminus[3][3][j][k] = L4m[j][k];
        Lminus[4][4][j][k] = L5m[j][k];

    }//end of "k" loop
    }//end of "j" loop
```

```
//The final matrix is created by adding the positive and negative.
//NOTE: This is not used.
    AddMatrix(Lplus, Lminus, Lambda, Zone);
    SubMatrix(Lplus, Lminus, AbsLambda, Zone);

}

//-------------------------------------
void EvalLambdaZ(int Zone)
{
    for (j=Rmin[Zone]+1; j<=Rmax[Zone]; j++)
    {
    for (k=Zmin[Zone]+1; k<=Zmax[Zone]; k++)
    {
//These are the eigenvalues of the Jacobian
        ZL1[j][k] = Vz[j][k]-Cfz[j][k];
        ZL2[j][k] = Vz[j][k];
        ZL3[j][k] = Vz[j][k];
        ZL4[j][k] = Vz[j][k];
        ZL5[j][k] = Vz[j][k]+Cfz[j][k];

//Splitting into positive values...
        ZL1p[j][k] = (ZL1[j][k] + fabs(ZL1[j][k]))/2;
        ZL2p[j][k] = (ZL2[j][k] + fabs(ZL2[j][k]))/2;
        ZL3p[j][k] = (ZL3[j][k] + fabs(ZL3[j][k]))/2;
        ZL4p[j][k] = (ZL4[j][k] + fabs(ZL4[j][k]))/2;
        ZL5p[j][k] = (ZL5[j][k] + fabs(ZL5[j][k]))/2;

//Splitting into negative values...
        ZL1m[j][k] = (ZL1[j][k] - fabs(ZL1[j][k]))/2;
        ZL2m[j][k] = (ZL2[j][k] - fabs(ZL2[j][k]))/2;
        ZL3m[j][k] = (ZL3[j][k] - fabs(ZL3[j][k]))/2;
        ZL4m[j][k] = (ZL4[j][k] - fabs(ZL4[j][k]))/2;
        ZL5m[j][k] = (ZL5[j][k] - fabs(ZL5[j][k]))/2;


//Creating the diagonal matrix with positive eigenvalues...
        for (row =0; row<5; row++)
        {
            for (col=0; col<5; col++)
            {
                LplusZ[row][col][j][k] = 0.0;
            }
        }

        LplusZ[0][0][j][k] = ZL1p[j][k];
        LplusZ[1][1][j][k] = ZL2p[j][k];
        LplusZ[2][2][j][k] = ZL3p[j][k];
        LplusZ[3][3][j][k] = ZL4p[j][k];
        LplusZ[4][4][j][k] = ZL5p[j][k];

//Creating the diagonal matrix with negative eigenvalues...
        for (row =0; row<5; row++)
        {
            for (col =0; col<5; col++)
            {
                LminusZ[row][col][j][k] = 0.0;
            }
        }

        LminusZ[0][0][j][k] = ZL1m[j][k];
        LminusZ[1][1][j][k] = ZL2m[j][k];
        LminusZ[2][2][j][k] = ZL3m[j][k];
        LminusZ[3][3][j][k] = ZL4m[j][k];
        LminusZ[4][4][j][k] = ZL5m[j][k];

    }//end of "k" loop
    }//end of "j" loop

//The final matrix is created by adding the positive and negative.
//NOTE: This is not used.
    AddMatrix(LplusZ, LminusZ, LambdaZ, Zone);
    SubMatrix(LplusZ, LminusZ, AbsLambdaZ, Zone);

}

//---------------------------------------

void EvalVectorsR(int Zone)
{
    for (j=Rmin[Zone]+1; j<=Rmax[Zone]; j++)
    {
    for (k=Zmin[Zone]+1; k<=Zmax[Zone]; k++)
    {
//Definitions:
        alfF[j][k] = sqrt(aSq[j][k]/(Cf[j][k]*Cf[j][k]));
        alfS[j][k] = sqrt(((Cf[j][k]*Cf[j][k]) - aSq[j][k])
                          /(Cf[j][k]*Cf[j][k]));
        betaT[j][k] = Bt[j][k]/sqrt(Bt[j][k]*Bt[j][k]);

//These are the right eigenvectors, R1 through R5.

        R1[0][j][k] = Rho[j][k]*alfF[j][k];
        R1[1][j][k] = -alfF[j][k]*Cf[j][k];
        R1[2][j][k] = 0.0;
        R1[3][j][k] = alfS[j][k]*sqrt(Mu*Rho[j][k]*aSq[j][k])
                       *betaT[j][k];
        R1[4][j][k] = alfF[j][k]*gam*p[j][k];
//------------------
        R2[0][j][k] = 0.0;
        R2[1][j][k] = 0.0;
        R2[2][j][k] = betaT[j][k];
        R2[3][j][k] = 0.0;
        R2[4][j][k] = 0.0;
//------------------
        R3[0][j][k] = 1.0;
        R3[1][j][k] = 0.0;
        R3[2][j][k] = 0.0;
        R3[3][j][k] = 0.0;
        R3[4][j][k] = 0.0;
//------------------
        R4[0][j][k] = Root2*Rho[j][k]*alfS[j][k];
        R4[1][j][k] = 0.0;
        R4[2][j][k] = 0.0;
        R4[3][j][k] = -Root2*alfF[j][k]*betaT[j][k]
                      *sqrt(Mu*Rho[j][k]*aSq[j][k]);
        R4[4][j][k] = Root2*alfS[j][k]*gam*p[j][k];
//------------------
        R5[0][j][k] = Rho[j][k]*alfF[j][k];
        R5[1][j][k] = alfF[j][k]*Cf[j][k];
        R5[2][j][k] = 0.0;
        R5[3][j][k] = alfS[j][k]*betaT[j][k]
                     *sqrt(Mu*Rho[j][k]*aSq[j][k]);
        R5[4][j][k] = alfF[j][k]*gam*p[j][k];
//------------------

//Creating the matrix of right eigenvectors.
```

```
        for (row=0; row<5; row++)
        {
            Rp[row][0][j][k] = R1[row][j][k];
            Rp[row][1][j][k] = R2[row][j][k];
            Rp[row][2][j][k] = R3[row][j][k];
            Rp[row][3][j][k] = R4[row][j][k];
            Rp[row][4][j][k] = R5[row][j][k];
        }

//------------------

//Creating the inverse of the matrix.

        Left1[0][j][k] = 0.0;
        Left1[1][j][k] = -alfF[j][k]*Cf[j][k]/(2*aSq[j][k]);
        Left1[2][j][k] = 0.0;
        Left1[3][j][k] = alfS[j][k]*betaT[j][k]
                        /(2*sqrt(Mu*Rho[j][k]*aSq[j][k]));
        Left1[4][j][k] = alfF[j][k]/(2*Rho[j][k]*aSq[j][k]);
//------------------
        Left2[0][j][k] = 0.0;
        Left2[1][j][k] = 0.0;
        Left2[2][j][k] = betaT[j][k]/Root2;
        Left2[3][j][k] = 0.0;
        Left2[4][j][k] = 0.0;
//------------------
        Left3[0][j][k] = 1.0;
        Left3[1][j][k] = 0.0;
        Left3[2][j][k] = 0.0;
        Left3[3][j][k] = 0.0;
        Left3[4][j][k] = -1/aSq[j][k];
//------------------
        Left4[0][j][k] = 0.0;
        Left4[1][j][k] = 0.0;
        Left4[2][j][k] = 0.0;
        Left4[3][j][k] = -alfF[j][k]*betaT[j][k]
                         /sqrt(2*Mu*Rho[j][k]*aSq[j][k]);
        Left4[4][j][k] = alfS[j][k]/(Root2*Rho[j][k]*aSq[j][k]);
//------------------
        Left5[0][j][k] = 0.0;
        Left5[1][j][k] = alfF[j][k]*Cf[j][k]/(2*aSq[j][k]);
        Left5[2][j][k] = 0.0;
        Left5[3][j][k] = alfS[j][k]*betaT[j][k]
                         /(2*sqrt(Mu*Rho[j][k]*aSq[j][k]));
        Left5[4][j][k] = alfF[j][k]/(2*Rho[j][k]*aSq[j][k]);
//------------------
//Creating the matrix of left eigenvectors.
        for (col=0; col<5; col++)
        {
            Lp[0][col][j][k] = Left1[col][j][k];
            Lp[1][col][j][k] = Left2[col][j][k];
            Lp[2][col][j][k] = Left3[col][j][k];
            Lp[3][col][j][k] = Left4[col][j][k];
            Lp[4][col][j][k] = Left5[col][j][k];
        }

    }//end of "k" loop
    }//end of "j" loop
//------------------
//Obtaining the eigenvectors for the conservative form:
    MulMatrix(M, Rp, R, Zone);
    MulMatrix(Lp, Minv, Rinv, Zone);

}
```

```
//---------------------------------------
void EvalVectorsZ(int Zone)
{
    for (j=Rmin[Zone]+1; j<=Rmax[Zone]; j++)
    {
    for (k=Zmin[Zone]+1; k<=Zmax[Zone]; k++)
    {
//Definitions:
        ZalfF[j][k] = sqrt(aSq[j][k]/(Cfz[j][k]*Cfz[j][k]));
        ZalfS[j][k] = sqrt(((Cfz[j][k]*Cfz[j][k]) - aSq[j][k])
                             /(Cfz[j][k]*Cfz[j][k]));
        ZbetaT[j][k] = Bt[j][k]/sqrt(Bt[j][k]*Bt[j][k]);

//These are the right eigenvectors, R1z through R5z.

        R1z[0][j][k] = Rho[j][k]*ZalfF[j][k];
        R1z[1][j][k] = 0.0;
        R1z[2][j][k] = -Cfz[j][k]*ZalfF[j][k];
        R1z[3][j][k] = ZalfS[j][k]*ZbetaT[j][k]
                       *sqrt(Mu*Rho[j][k]*aSq[j][k]);
        R1z[4][j][k] = Rho[j][k]*aSq[j][k]*ZalfF[j][k];
//--------------------

        R2z[0][j][k] = 0.0;
        R2z[1][j][k] = -ZbetaT[j][k]/Root2;
        R2z[2][j][k] = 0.0;
        R2z[3][j][k] = 0.0;
        R2z[4][j][k] = 0.0;
//--------------------

        R3z[0][j][k] = 1.0;
        R3z[1][j][k] = 0.0;
        R3z[2][j][k] = 0.0;
        R3z[3][j][k] = 0.0;
        R3z[4][j][k] = 0.0;
//--------------------

        R4z[0][j][k] = Root2*Rho[j][k]*ZalfS[j][k];
        R4z[1][j][k] = 0.0;
        R4z[2][j][k] = 0.0;
        R4z[3][j][k] = -Root2*ZalfF[j][k]*ZbetaT[j][k]
                       *sqrt(Mu*Rho[j][k]*aSq[j][k]);
        R4z[4][j][k] = Root2*Rho[j][k]*aSq[j][k]*ZalfS[j][k];
//--------------------

        R5z[0][j][k] = Rho[j][k]*ZalfF[j][k];
        R5z[1][j][k] = 0.0;
        R5z[2][j][k] = Cfz[j][k]*ZalfF[j][k];
        R5z[3][j][k] = ZalfS[j][k]*ZbetaT[j][k]
                       *sqrt(Mu*Rho[j][k]*aSq[j][k]);
        R5z[4][j][k] = Rho[j][k]*aSq[j][k]*ZalfF[j][k];

//--------------------
//Creating the matrix of right eigenvectors.
        for (row=0; row<5; row++)
        {
            RpZ[row][0][j][k] = R1z[row][j][k];
            RpZ[row][1][j][k] = R2z[row][j][k];
            RpZ[row][2][j][k] = R3z[row][j][k];
            RpZ[row][3][j][k] = R4z[row][j][k];
            RpZ[row][4][j][k] = R5z[row][j][k];
        }
//--------------------
//Creating the inverse of the matrix.

        Left1z[0][j][k] = 0.0;
        Left1z[1][j][k] = 0.0;
        Left1z[2][j][k] = -Cfz[j][k]*ZalfF[j][k]/(2*aSq[j][k]);
        Left1z[3][j][k] = ZalfS[j][k]*ZbetaT[j][k]
                          /(2*sqrt(Mu*Rho[j][k]*aSq[j][k]));
        Left1z[4][j][k] = ZalfF[j][k]/(2*Rho[j][k]*aSq[j][k]);
//------------------
        Left2z[0][j][k] = 0.0;
        Left2z[1][j][k] = -ZbetaT[j][k]/Root2;
        Left2z[2][j][k] = 0.0;
        Left2z[3][j][k] = 0.0;
        Left2z[4][j][k] = 0.0;
//------------------
        Left3z[0][j][k] = 1.0;
        Left3z[1][j][k] = 0.0;
        Left3z[2][j][k] = 0.0;
        Left3z[3][j][k] = 0.0;
        Left3z[4][j][k] = -1/aSq[j][k];
//------------------
        Left4z[0][j][k] = 0.0;
        Left4z[1][j][k] = 0.0;
        Left4z[2][j][k] = 0.0;
        Left4z[3][j][k] = -ZalfF[j][k]*ZbetaT[j][k]
                      /(Root2*sqrt(Mu*Rho[j][k]*aSq[j][k]));
        Left4z[4][j][k] = ZalfS[j][k]/(Root2*Rho[j][k]*aSq[j][k]);
//------------------
        Left5z[0][j][k] = 0.0;
        Left5z[1][j][k] = 0.0;
        Left5z[2][j][k] = Cfz[j][k]*ZalfF[j][k]/(2*aSq[j][k]);
        Left5z[3][j][k] = ZalfS[j][k]*ZbetaT[j][k]
                      /(2*sqrt(Mu*Rho[j][k]*aSq[j][k]));
        Left5z[4][j][k] = ZalfF[j][k]/(2*Rho[j][k]*aSq[j][k]);
//------------------
//Creating the matrix of left eigenvectors.
        for (col=0; col<5; col++)
        {
            LpZ[0][col][j][k] = Left1z[col][j][k];
            LpZ[1][col][j][k] = Left2z[col][j][k];
            LpZ[2][col][j][k] = Left3z[col][j][k];
            LpZ[3][col][j][k] = Left4z[col][j][k];
            LpZ[4][col][j][k] = Left5z[col][j][k];
        }
    }//end of "k" loop
    }//end of "j" loop
//------------------
//Obtaining the eigenvectors for the conservative form:
    MulMatrix(M, RpZ, RZ, Zone);
    MulMatrix(LpZ, Minv, RinvZ, Zone);
}

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

C.12 Jacobian

```
void EvalA(int Zone)
{
//---------------------------------------------------------
//Jacobian.CPP; Written by Kameshwaran Sankaran
//---------------------------------------------------------
/**************************************************
        Performing calculations in the "r" direction
**************************************************/
//"AverageR" calculates the values of the variables at j+(1/2).
//Since they are stored in the same array, the old values
//are over-written.
    AverageR(Zone);


//M= dU/dW, where U=conservative variables, W=primitive variables
    EvalM(Zone);
    EvalMinv(Zone);

//Calculating the diagonal matrix of eigenvalues, Plus & Minus
    EvalLambdaR(Zone);
//Calculating the riht eigenvalue matrix and its inverse
    EvalVectorsR(Zone);

    MulMatrix(R, Lplus, temp, Zone);
//Aplus has non-negative eigenvalues
    MulMatrix(temp, Rinv, Aplus, Zone);

    MulMatrix(R, Lminus, temp, Zone);
//Aminus has non-positive eigenvalues
    MulMatrix(temp, Rinv, Aminus, Zone);

     SubMatrix(Aplus, Aminus, AbsA, Zone);

//As mentioned in the beginning, the values at node-points
//were over-written by values at half-points. "ReCalculate"
//resets it back to node-points. This also calls the function
//"EvalVar", thus resetting the speeds to the node-points.
    ReCalculate();


/***********************************************************
        Performing calculations in the "z" direction
***********************************************************/

//"AverageZ" calculates the values of the variables at k+(1/2).
//Since they are stored in the same array, the old values
//are over-written.
    AverageZ(Zone);

//M= dU/dW, where U=conservative variables, W=primitive variables
    EvalM(Zone);
    EvalMinv(Zone);

//Calculating the diagonal matrix of eigenvalues, Plus & Minus
    EvalLambdaZ(Zone);
//Calculating the right eigenvalue matrix and its inverse
    EvalVectorsZ(Zone);

    MulMatrix(RZ, LplusZ, temp, Zone);
//Aplus has non-negative eigenvalues
    MulMatrix(temp, RinvZ, Bplus, Zone);
    MulMatrix(RZ, LminusZ, temp, Zone);
//Aminus has non-positive eigenvalues
    MulMatrix(temp, RinvZ, Bminus, Zone);

     SubMatrix(Bplus, Bminus, AbsB, Zone);

//As mentioned in the beginning, the values at node-points
//were over-written by values at half-points. "ReCalculate"
//resets it back to node-points. This also calls the function
//"EvalVar", thus resetting the speeds to the node-points.
    ReCalculate();
}
//-------------------------------------
void AverageR(int Zone)
{
//Setting temporary variables to the value at node-points
    for (j=Rmin[Zone]+1; j<=Rmax[Zone]; j++)
    {
        for (k=Zmin[Zone]+1; k<=Zmax[Zone]; k++)
        {
            rho[j][k]    = Rho[j][k];
            ur[j][k]    = Vr[j][k];
            uz[j][k]    = Vz[j][k];
            hOld[j][k]    = h[j][k];
            bt[j][k]    = Bt[j][k];
        }
    }
//This is Roe averaging. So far, the values of the variables are at
//node-points.Once this averaging is done, the values are at j+(1/2).
    for (j=Rmin[Zone]+1; j<=Rmax[Zone]; j++)
    {
        for (k=Zmin[Zone]+1; k<=Zmax[Zone]; k++)
        {
            Rho[j][k] = pow(rho[j+1][k]*rho[j][k],0.5);

            Vr[j][k] = ((pow(rho[j+1][k],0.5)*ur[j+1][k])
                        +(pow(rho[j][k],0.5)*ur[j][k]))/
                       (pow(rho[j+1][k],0.5)+pow(rho[j][k],0.5));

            Vz[j][k] = ((pow(rho[j+1][k],0.5)*uz[j+1][k])
                        +(pow(rho[j][k],0.5)*uz[j][k]))/
                       (pow(rho[j+1][k],0.5)+pow(rho[j][k],0.5));

            h[j][k] = ((hOld[j+1][k]*pow(rho[j+1][k], 0.5))
                       +(hOld[j][k]*pow(rho[j][k], 0.5)))/
                       (pow(rho[j+1][k],0.5)+pow(rho[j][k],0.5));

            BtAvg[j][k] = ((bt[j+1][k]*pow(rho[j][k], 0.5))
                           +(bt[j][k]*pow(rho[j+1][k], 0.5)))/2;

            p[j][k] = ((gam-1)/gam)*Rho[j][k]*( h[j][k]
                  -0.5*((Vr[j][k]*Vr[j][k])+(Vz[j][k]+Vz[j][k]))
                    - (BtAvg[j][k]*BtAvg[j][k]/(Mu*rho[j][k])) );
        }//end of "k"
    }//end of "j"
//The function "EvalVectors" needs values of speeds at j+(1/2).
//So, call "EvalVar"
    EvalVar(Zone);

//To restore the values at node-points, "ReCalculate"
//function is called by "EvalA".

}
//-------------------------------------
```

```
void AverageZ(int Zone)
{
//Setting temporary variables to the value at node-points
    for (j=Rmin[Zone]+1; j<=Rmax[Zone]; j++)
    {
        for (k=Zmin[Zone]+1; k<=Zmax[Zone]; k++)
        {
            rho[j][k]    = Rho[j][k];
            ur[j][k]    = Vr[j][k];
            uz[j][k]    = Vz[j][k];
            hOld[j][k]    = h[j][k];
            bt[j][k]    = Bt[j][k];
        }
    }
//This is Roe averaging. So far, the values of the variables are at
//node-points.Once this averaging is done, the values are at k+(1/2).
    for (j=Rmin[Zone]+1; j<=Rmax[Zone]; j++)
    {
        for (k=Zmin[Zone]+1; k<=Zmax[Zone]; k++)
        {
            Rho[j][k] = pow(rho[j][k+1]*rho[j][k],0.5);

            Vr[j][k] = ((pow(rho[j][k+1],0.5)*ur[j][k+1])
                       +(pow(rho[j][k],0.5)*ur[j][k]))/
                      (pow(rho[j][k+1],0.5)+pow(rho[j][k],0.5));

            Vz[j][k] = ((pow(rho[j][k+1],0.5)*uz[j][k+1])
                       +(pow(rho[j][k],0.5)*uz[j][k]))/
                      (pow(rho[j][k+1],0.5)+pow(rho[j][k],0.5));

            h[j][k] = ((hOld[j][k+1]*pow(rho[j][k+1], 0.5))
                      +(hOld[j][k]*pow(rho[j][k], 0.5)))/
                     (pow(rho[j][k+1],0.5)+pow(rho[j][k],0.5));

            BtAvg[j][k] = ((bt[j][k+1]*pow(rho[j][k], 0.5))
                          +(bt[j][k]*pow(rho[j][k+1], 0.5)))/2;

            p[j][k] = ((gam-1)/gam)*Rho[j][k]*( h[j][k]
                    -0.5*((Vr[j][k]*Vr[j][k])+(Vz[j][k]+Vz[j][k]))
                   - (BtAvg[j][k]*BtAvg[j][k]/(Mu*rho[j][k])) );
        }//end of "k"
    }//end of "j"
//The function "EvalVectors" needs values of speeds at k+(1/2).
//So, call "EvalVar"
    EvalVar(Zone);

//To restore the values at node-points, "ReCalculate" function
//is called by "EvalA".

}
//---------------------------------------

void EvalM(int Zone)
{
//"M" is the Jacobian = dU/dW, i.e., the transformation matrix between
//primitive and conservation variables.
    for (j=Rmin[Zone]+1; j<=Rmax[Zone]; j++)
    {
    for (k=Zmin[Zone]+1; k<=Zmax[Zone]; k++)
    {
        M[0][0][j][k] = 1.0;
        M[0][1][j][k] = 0.0;
        M[0][2][j][k] = 0.0;
        M[0][3][j][k] = 0.0;
        M[0][4][j][k] = 0.0;

        M[1][0][j][k] = Vr[j][k];
        M[1][1][j][k] = Rho[j][k];
        M[1][2][j][k] = 0.0;
        M[1][3][j][k] = 0.0;
        M[1][4][j][k] = 0.0;

        M[2][0][j][k] = Vz[j][k];
        M[2][1][j][k] = 0.0;
        M[2][2][j][k] = Rho[j][k];
        M[2][3][j][k] = 0.0;
        M[2][4][j][k] = 0.0;

        M[3][0][j][k] = 0.0;
        M[3][1][j][k] = 0.0;
        M[3][2][j][k] = 0.0;
        M[3][3][j][k] = 1;
        M[3][4][j][k] = 0.0;

        M[4][0][j][k] = 0.5*((Vr[j][k]*Vr[j][k])+(Vz[j][k]*Vz[j][k]));
        M[4][1][j][k] = Rho[j][k]*Vr[j][k];
        M[4][2][j][k] = Rho[j][k]*Vz[j][k];
        M[4][3][j][k] = Bt[j][k]/Mu;
        M[4][4][j][k] = 1/(gam-1);

    }//end of k
    }//end of j
}
//---------------------------------------
void EvalMinv(int Zone)
{
//"Minv" is the inverse of "M".
    for (j=Rmin[Zone]+1; j<=Rmax[Zone]; j++)
    {
    for (k=Zmin[Zone]+1; k<=Zmax[Zone]; k++)
    {
        Minv[0][0][j][k] = 1.0;
        Minv[0][1][j][k] = 0.0;
        Minv[0][2][j][k] = 0.0;
        Minv[0][3][j][k] = 0.0;
        Minv[0][4][j][k] = 0.0;

        Minv[1][0][j][k] = -Vr[j][k]/Rho[j][k];
        Minv[1][1][j][k] = 1/Rho[j][k];
        Minv[1][2][j][k] = 0.0;
        Minv[1][3][j][k] = 0.0;
        Minv[1][4][j][k] = 0.0;

        Minv[2][0][j][k] = -Vz[j][k]/Rho[j][k];
        Minv[2][1][j][k] = 0.0;
        Minv[2][2][j][k] = 1/Rho[j][k];
        Minv[2][3][j][k] = 0.0;
        Minv[2][4][j][k] = 0.0;

        Minv[3][0][j][k] = 0.0;
        Minv[3][1][j][k] = 0.0;
        Minv[3][2][j][k] = 0.0;
        Minv[3][3][j][k] = 1.0;
        Minv[3][4][j][k] = 0.0;

        Minv[4][0][j][k] = (gam-1)*0.5*( (Vr[j][k]*Vr[j][k])
                                        +(Vz[j][k]*Vz[j][k]) );
        Minv[4][1][j][k] = -(gam-1)*Vr[j][k];
        Minv[4][2][j][k] = -(gam-1)*Vz[j][k];
        Minv[4][3][j][k] = -(gam-1)*Bt[j][k]/Mu;
        Minv[4][4][j][k] = gam-1;
    }//end of k
    }//end of j
}

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

C.13 Linear Algebra

```
//-----------------------------------------------------------
//LinearAlgebra.CPP; Written by Kameshwaran Sankaran
//-----------------------------------------------------------
//These are supplementary matrix and vector algebra functions

//----------------------------------------

void MulMatrix(double **MatA[5][5], double **MatB[5][5],
               double **MatC[5][5], int Zone)
{
    for (j=Rmin[Zone]+1; j<=Rmax[Zone]; j++)
    {
        for (k=Zmin[Zone]+1; k<=Zmax[Zone]; k++)
        {
            for(row=0; row<5; row++)
            {
                for(col=0; col<5; col++)
                {
                    MatC[row][col][j][k] = 0.0;
                    for(l=0; l<5; l++)
                    {
                        MatC[row][col][j][k] += MatA[row][l][j][k]*
                                              MatB[l][col][j][k];
                    }
                }
            }
        }
    }
}
//----------------------------------------

void AddMatrix(double **MatA[5][5], double **MatB[5][5],
               double **MatC[5][5], int Zone)
{
    for (j=Rmin[Zone]+1; j<=Rmax[Zone]; j++)
    {
        for (k=Zmin[Zone]+1; k<=Zmax[Zone]; k++)
        {
            for(row=0; row<5; row++)
            {
                for(col=0; col<5; col++)
                {
                    MatC[row][col][j][k] = MatA[row][col][j][k]
                                          +MatB[row][col][j][k];
                }
            }
        }
    }
}

//----------------------------------------
void SubMatrix(double **MatA[5][5], double **MatB[5][5],
               double **MatC[5][5], int Zone)
{
    for (j=Rmin[Zone]+1; j<=Rmax[Zone]; j++)
    {
        for (k=Zmin[Zone]+1; k<=Zmax[Zone]; k++)
        {
            for(row=0; row<5; row++)
            {
                for(col=0; col<5; col++)
                {
                    MatC[row][col][j][k] = MatA[row][col][j][k]
                                          -MatB[row][col][j][k];
                }
            }
        }
    }
}
//-------------------------------------
void MatTimesVec(double **MatA[5][5], double **VecB[5],
                 double **VecC[5], int Zone)
{
    for (j=Rmin[Zone]+1; j<=Rmax[Zone]; j++)
    {
        for (k=Zmin[Zone]+1; k<=Zmax[Zone]; k++)
        {
            for(row=0; row<5; row++)
            {
                VecC[row][j][k] = 0.0;
                for(col=0; col<5; col++)
                {
                    VecC[row][j][k] += MatA[row][col][j][k]
                                      *VecB[col][j][k];
                }
            }
        }
    }
}

//-------------------------------------
void AddVector(double **VecA[5], double **VecB[5],
               double **VecC[5], int Zone)
{
    for (j=Rmin[Zone]+1; j<=Rmax[Zone]; j++)
    {
        for (k=Zmin[Zone]+1; k<=Zmax[Zone]; k++)
        {
            for(row=0; row<5; row++)
            {
                VecC[row][j][k] = VecA[row][j][k]+VecB[row][j][k];
            }
        }
    }
}
//-------------------------------------
void SubVector(double **VecA[5], double **VecB[5],
               double **VecC[5], int Zone)
{
    for (j=Rmin[Zone]+1; j<=Rmax[Zone]; j++)
    {
        for (k=Zmin[Zone]+1; k<=Zmax[Zone]; k++)
        {
            for(row=0; row<5; row++)
            {
                VecC[row][j][k] = VecA[row][j][k]-VecB[row][j][k];
            }
        }
    }
}

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

C.14 Dissipative Fluxes

```
//-----------------------------------------------------------
//EvalDiss.CPP; Written by Kameshwaran Sankaran
//-----------------------------------------------------------
void EvalDiss()
{
//Physical dissipation
//CHAMBER
    for(Zone = 1; Zone <= 3; Zone++)
    {
        for(j=J=Rmin[Zone]+1; j<=Rmax[Zone]; j++,J++)
        {
            for (K=Zmin[Zone]+1; K<=Zmax[Zone]; K++)
            {
//There is no need to waste time computing in places without magnetic field.
                if(BSq[J][K] > eps)
                {
                    jz[j][K] = (((J+0.5)*Bt[J+1][K])
                               -((J-0.5)*Bt[J][K]))
                              /(Mu*(J+0.5)*deltaR);
                    Ez[j][K] = 0.5*(Res[J][K]+Res[J+1][K])*jz[j][K];
                    EzH[j][K]= (0.5*(jr[J][K]+jr[J][K-1]))
                              *(0.5*((Bt[J][K]/n[J][K])
                                    +(Bt[J+1][K]/n[J+1][K]))/q);
                    if((Zone==1)&&(K==1))
                        EzH[j][K] = 0.0;
                    if(Zone==2)
                    {
                        if(Zc <= Za)
                        {
                            if((j<=Rc)&&(K==Zc+1))
                                EzH[j][K] = 0.0;
                        }
                        else
                        {
                            if( ((j>Ra)&&(K==Za+1)) || (j==numR) )
                                EzH[j][K] = 0.0;
                        }
                    }
                    if(Zone==3)
                    {
                        if(Zc <= Za)
                        {
                            if( ((j>Ra)&&(K==Za+1)) || (j==numR) )
                                EzH[j][K] = 0.0;
                        }
                        else
                        {
                            if( ((j<=Rc)&&(K==Zc+1)) || (j==numR) )
                                EzH[j][K] = 0.0;
                        }
                    }
                    DissipR[3][j][K]= Ez[j][K] + EzH[j][K];

                    ElCondR[j][K]= 0.5*(kTherm[J][K]+kTherm[J+1][K])
                                  *(Te[J+1][K]-Te[J][K])/deltaR;
                    IonCondR[j][K]= 0.5*(kIon[J][K]+kIon[J+1][K])
                                  *(Th[J+1][K]-Th[J][K])/deltaR;
                    ThermCondR[j][K]= ElCondR[j][K] + IonCondR[j][K];

                    DissipR[4][j][K]= (DissipR[3][j][K]*
                                   0.5*(Bt[J][K]+Bt[J+1][K])/Mu)
                                  + ThermCondR[j][K];
                    SourceDiss[4][j][K] = DissipR[4][j][K]
                                         /((j+1)*deltaR);
                }//end of ’if’.
            }//end of ’K’.
        }//end of ’j,J’.


        for(J=Rmin[Zone]+1; J<=Rmax[Zone]; J++)
        {
            for (k=K=Zmin[Zone]; k<=Zmax[Zone]-1; k++,K++)
            {
                if(BSq[J][K] > eps)
                {
                    jr[J][k] = -(Bt[J][K+1]-Bt[J][K])/(Mu*deltaZ);
                    Er[J][k] = -0.5*(Res[J][K]+Res[J][K+1])*jr[J][k];
                    ErH[J][k]= -(0.5*(jz[J][K]+jz[J-1][K]))
                               *(0.5*((Bt[J][K+1]/n[J][K+1])
                               +(Bt[J][K]/n[J][K]))/q);

                    if((Zone==1)&&((J==Rc+1)||(J==Ra)))
                        ErH[J][k] = 0.0;
                    if(Zone==2)
                    {
                        if(Zc<=Za)
                        {
                            if((J==1)||(J==Ra))
                                ErH[J][k] = 0.0;
                        }
                        else
                        {
                            if((J==Rc+1)||(J==numR))
                                ErH[J][k] = 0.0;
                        }
                    }
                    if(Zone==3)
                    {
//No Hall effect on the boundary cells.
                        if((J==1) || (J==numR) || (k == numZ))
                            ErH[J][k]    = 0.0;
                    }

                    DissipZ[3][J][k]= Er[J][k] + ErH[J][k];

                    ElCondZ[J][k] = 0.5*(kTherm[J][K]+kTherm[J][K+1])
                                       *(Te[J][K+1]-Te[J][K])/deltaZ;
                    IonCondZ[J][k] = 0.5*(kIon[J][K]+kIon[J][K+1])
                                        *(Th[J][K+1]-Th[J][K])/deltaZ;
                    ThermCondZ[J][k] = ElCondZ[J][k] + IonCondZ[J][k];

                    DissipZ[4][J][k]= (DissipZ[3][J][k]
                                      * 0.5*(Bt[J][K]+Bt[J][K+1])/Mu)
                                     + ThermCondZ[J][k];
                }
            }//end of ’k,K’.
        }//end of ’J’.
    }//end for all zones.

    DissBound();//Computes the dissipative terms at the boundaries.
}
//-------------------------------------
```

C.15 March in Time

```
//-----------------------------------------------------------
//TimeMarch.CPP; Written by Kameshwaran Sankaran
//-----------------------------------------------------------
void TimeMarch()
{
//The variables are stored at cell centers. So, ’J’ refers
//to ’j-0.5’ and ’K’ refers to ’k-0.5’.
//For fluxes, ’j’ and ’k’ refer to correct values.

    for(SubStep = 1; SubStep <= MaxStep; SubStep++)
    {
        EvalDiss();//Computes physical dissipation.

        for(i=0; i<5; i++)
        {
            for(Zone = 1; Zone <= 3; Zone++)
            {
                for (j=J=Rmin[Zone]+1; J<=Rmax[Zone]; j++,J++)
                {
                    for(k=K=Zmin[Zone]+1; K<=Zmax[Zone]; k++,K++)
                    {
                        uOld[i][J][K] = u[i][J][K];
                        u[i][J][K]+=deltaT*(((DissipR[i][j][k]
                                              -DissipR[i][j-1][k])
                                             /deltaR)
                                            +SourceDiss[i][j-1][k]
                                            +((DissipZ[i][j][k]
                                              -DissipZ[i][j][k-1])
                                              /deltaZ)
                                           - ConvFlux[i][J][K] );
                    }
                }
            }
        }
//The primary variables are computed from the conservation variables.
        ReCalculate();

//This accounts for the thermal nonequilibrium between
//electrons and ions.
        EvalEnergy();

        Saha();

//This calculates the ion temperature.
        EquationOfState();

        EvalGamma();

        t += deltaT;

        numSteps++;
    }//end of "SubStep" loop.
}
//---------------------------------------
```

C.16 Species Energy

```
//---------------------------------------------------------
//EvalEnergy.CPP; Written by Kameshwaran Sankaran
//---------------------------------------------------------
void EvalEnergy()
{
//Equivalent to "EvalF".
    for(Zone = 1; Zone <= 3; Zone++)
    {
        for (J=Rmin[Zone]+1; J<=Rmax[Zone]; J++)
        {
            for (K=Zmin[Zone]+1; K<=Zmax[Zone]; K++)
            {
                FrEn[J][K] = Vr[J][K]*(EintEl[J][K]+pe[J][K]);
                SrEn[J][K] = FrEn[J][K];

                FzEn[J][K] = Vz[J][K]*(EintEl[J][K]+pe[J][K]);
            }
        }
    }

//Equivalent to "EvalNumDiss".
    for(Zone = 1; Zone <= 3; Zone++)
    {
        for(j=J=Rmin[Zone]+1; j<=Rmax[Zone]-1; j++,J++)
        {
            for (K=Zmin[Zone]+1; K<=Zmax[Zone]; K++)
            {
                DrEn[j][K]=0.5*0.5*(SpecRadR[J][K]+SpecRadR[J+1][K])
                                  *(EintEl[J+1][K]-EintEl[J][K]);
            }
        }
        for(J=Rmin[Zone]+1; J<=Rmax[Zone]; J++)
        {
            for (k=K=Zmin[Zone]+1; k<=Zmax[Zone]; k++,K++)
            {
                DzEn[J][k]=0.5*0.5*(SpecRadZ[J][K]+SpecRadZ[J][K+1])
                                  *(EintEl[J][K+1]-EintEl[J][K]);
            }
        }
    }

//Equivalent to "EvalConv".
    for(Zone = 1; Zone <= 3; Zone++)
    {
        for(j=J=Rmin[Zone]+1; j<=Rmax[Zone]; j++,J++)
        {
            for (K=Zmin[Zone]+1; K<=Zmax[Zone]; K++)
            {
                EnFluxR[j][K] = (0.5*(FrEn[J+1][K]+FrEn[J][K]))
                                 - DrEn[j][K];
                EnSourceR[j][K] = (0.5*(SrEn[J+1][K]+SrEn[J][K]))
                                  /((j+1)*deltaR);
            }
        }
        for(J=Rmin[Zone]+1; J<=Rmax[Zone]; J++)
        {
            for (k=K=Zmin[Zone]+1; k<=Zmax[Zone]; k++,K++)
            {
                EnFluxZ[J][k] = (0.5*(FzEn[J][K+1]+FzEn[J][K]))
                                 - DzEn[J][k];
            }
        }
    }

//Equivalent to "TimeMarch".
    for(Zone = 1; Zone <= 3; Zone++)
    {
        for (j=J=Rmin[Zone]+1; J<=Rmax[Zone]; j++,J++)
        {
            for(k=K=Zmin[Zone]+1; K<=Zmax[Zone]; k++,K++)
            {
//First term is the compressional energy,
//the second term is the Ohmic heating, the third term is the
//e-i energy exchange, the fourth term is the convective flux,
//the fifth term is the thermal conduction.

                if(J==Rmax[Zone])
                {
                    pe[J+1][K] = pe[J][K];
                    jz[j][K] = 0.5*Bt[J][K]/Mu;
                }
                if(J==Rmin[Zone]+1)
                {
                    pe[J-1][K] = pe[J][K];
                    jz[j-1][K] = 0.5*Bt[J][K]/Mu;
                }
                if((K==Zmax[Zone])&&(Zone==3))
                    pe[J][K+1] = pe[J][K];
                if(K==Zmin[Zone]+1)
                {
                    if(Zone==1)
                    {
                        pe[J][K-1] = peIn;
                        jr[J][k-1] = 0.0;
                    }
                    if((Zone==2)&&(J>=Ra))
                    {
                        jr[J][k-1] = 0.5*Bt[J][K]/Mu;
                        pe[J][K-1] = pe[J][K];
                    }
                    if((Zone==3)&&(J<=Rc))
                    {
                        jr[J][k-1] = 0.5*Bt[J][K]/Mu;
                        pe[J][K-1] = pe[J][K];
                    }
                }

                ElecComp[J][K] = (Vr[J][K]*(0.5*(pe[J+1][K]
                                                -pe[J-1][K])
                                                   /deltaR))
                                  +(Vz[J][K]*(0.5*(pe[J][K+1]
                                                  -pe[J][K-1])
                                                     /deltaZ));

                Ohmic[J][K] = (0.5*((-jr[J][k]*Er[J][k])
                                    +(-jr[J][k-1]*Er[J][k-1])))
                                 +(0.5*((-jz[j][K]*Ez[j][K])
                                    +(-jz[j-1][K]*Ez[j-1][K])));

//Tab allignments are intentionally offset.
    Exchange[J][K] = 3*EIcollFreq[J][K]*n[J][K]*mEl
                  *kBoltz*(Te[J][K]-Th[J][K])/mAr;

    EintEl[J][K] += deltaT*(ElecComp[J][K]+Ohmic[J][K]-Exchange[J][K]
                       -(((EnFluxR[j][K]-EnFluxR[j-1][K])/deltaR)
                       +EnSourceR[j-1][K]+((EnFluxZ[J][k]-
                                           EnFluxZ[J][k-1])/deltaZ))
                           +(((ElCondR[j][K]-ElCondR[j-1][K])/deltaR)
                           +((ElCondZ[J][k]-ElCondZ[J][k-1])/deltaZ)));

//  pe[J][K]    = (gamConst-1)*EintEl[J][K];
//  ph[J][K]    = p[J][K] - pe[J][K];

    Te[J][K]    = (gamConst-1)*EintEl[J][K]/(ne[J][K]*kBoltz);
    }
    }
    }
}
//-------------------------------------
```

C.17 Calculate Variables

```
//--------------------------------------------------------
//ReCalculate.CPP; Written by Kameshwaran Sankaran
//--------------------------------------------------------
void ReCalculate()
{
//Variables for convergence checks.
    u0Dmax    = 0.0;
    u1Dmax    = 0.0;
    u2Dmax    = 0.0;
    u3Dmax    = 0.0;
    u4Dmax    = 0.0;

    u0Davg    = 0.0;
    u1Davg    = 0.0;
    u2Davg    = 0.0;
    u3Davg    = 0.0;
    u4Davg    = 0.0;

    DensD    = 0.0;
    KED        = 0.0;
    BtD        = 0.0;
    EtotD    = 0.0;
    EthD    = 0.0;

    DensDavg= 0.0;
    KEDavg    = 0.0;
    BtDavg    = 0.0;
    EtotDavg= 0.0;
    EthDavg    = 0.0;

    u0Max    = 0.0;
    u1Max    = 0.0;
    u2Max    = 0.0;
    u3Max    = 0.0;
    u4Max    = 0.0;

    u0Avg    = 0.0;
    u1Avg    = 0.0;
    u2Avg    = 0.0;
    u3Avg    = 0.0;
    u4Avg    = 0.0;

    Count    = 0;

//The variables are stored at cell centers.
//So, ’J’ refers to ’j-0.5’ and ’K’ refers to ’k-0.5’.

    for(Zone = 1; Zone <= 3; Zone++)
    {
        for( J=Rmin[Zone]+1; J<=Rmax[Zone]; J++)
        {
            for (K=Zmin[Zone]+1; K<=Zmax[Zone]; K++)
            {
//Storing some old variables
                KEold[J][K] = KE[J][K];
                EthOld[J][K]= Etherm[J][K];

//Recalculating new variables
                Rho[J][K] = u[0][J][K];
                Vr[J][K]  = u[1][J][K]/u[0][J][K];
                Vz[J][K]  = u[2][J][K]/u[0][J][K];
                Bt[J][K]  = u[3][J][K];
                E[J][K]   = u[4][J][K];

//Other necessary variables
                VSq[J][K] = (Vr[J][K]*Vr[J][K])+(Vz[J][K]*Vz[J][K]);
                BSq[J][K] = Bt[J][K]*Bt[J][K];
                KE[J][K]  = 0.5*Rho[J][K]*VSq[J][K];
                Etherm[J][K]= E[J][K]-KE[J][K]-(0.5*BSq[J][K]/Mu);
//              p[J][K]   = (gam[J][K]-1)*Etherm[J][K];
                p[J][K]   = (gam[J][K]-1)*(E[J][K]-(0.5*Rho[J][K]
                                  *VSq[J][K])-(0.5*BSq[J][K]/Mu));
                n[J][K]   = Rho[J][K]/mAr;
                Jencl[J][K]= 2*PI*(J-0.5)*deltaR*Bt[J][K]/Mu;
                Potential[J][K] = Potential[J-1][K]
                                 +((0.5*(DissipZ[3][J][K]
                                        +DissipZ[3][J][K-1]))-
                                  (Vz[J][K]*Bt[J][K]));
            }
        }
    }
//Convergence checks
    for(Zone = 1; Zone <= 3; Zone++)
    {
        for( J=Rmin[Zone]+1; J<=Rmax[Zone]-1; J++)
        {
            for (K=Zmin[Zone]+1; K<=Zmax[Zone]-1; K++)
            {
                if(fabs(u[0][J][K]-uOld[0][J][K])/u[0][J][K]>u0Dmax)
                {
                    u0Dmax  = fabs(u[0][J][K]-uOld[0][J][K])
                                     /u[0][J][K];
                    u0DMaxR = (J-1)*deltaR;
                    u0DMaxZ = (K-1)*deltaZ;
                }
                if(fabs(u[1][J][K]-uOld[1][J][K])/u[1][J][K]>u1Dmax)
                {
                    u1Dmax  = fabs(u[1][J][K]-uOld[1][J][K])
                                     /u[1][J][K];
                    u1DMaxR = (J-1)*deltaR;
                    u1DMaxZ = (K-1)*deltaZ;
                }
                if(fabs(u[2][J][K]-uOld[2][J][K])/u[2][J][K]>u2Dmax)
                {
                    u2Dmax  = fabs(u[2][J][K]-uOld[2][J][K])
                                     /u[2][J][K];
                    u2DMaxR = (J-1)*deltaR;
                    u2DMaxZ = (K-1)*deltaZ;
                }
                if(fabs((u[3][J][K]-uOld[3][J][K])/u[3][J][K])>u3Dmax)
                {
                    u3Dmax  = fabs((u[3][J][K]-uOld[3][J][K])
                                        /u[3][J][K]);
                    u3DMaxR = (J-1)*deltaR;
                    u3DMaxZ = (K-1)*deltaZ;
                }
                if(fabs(u[4][J][K]-uOld[4][J][K])/u[4][J][K]>u4Dmax)
                {
                    u4Dmax  = fabs(u[4][J][K]-uOld[4][J][K])
                                       /u[4][J][K];
                    u4DMaxR = (J-1)*deltaR;
                    u4DMaxZ = (K-1)*deltaZ;
                }

                u0Davg  += fabs(u[0][J][K]-uOld[0][J][K])/u[0][J][K];
                u1Davg  += fabs(u[1][J][K]-uOld[1][J][K])/u[1][J][K];
                u2Davg  += fabs(u[2][J][K]-uOld[2][J][K])/u[2][J][K];
                u3Davg  += fabs((u[3][J][K]-uOld[3][J][K])/u[3][J][K]);
                u4Davg  += fabs(u[4][J][K]-uOld[4][J][K])/u[4][J][K];

                if(u[0][J][K] > u0Max)
                {
                    u0Max    = u[0][J][K];
                    u0MaxR    = (J-1)*deltaR;
                    u0MaxZ    = (K-1)*deltaZ;
                }
                if(u[1][J][K] > u1Max)
                {
                    u1Max    = u[1][J][K];
                    u1MaxR    = (J-1)*deltaR;
                    u1MaxZ    = (K-1)*deltaZ;
                }
                if(u[2][J][K] > u2Max)
                {
                    u2Max    = u[2][J][K];
                    u2MaxR    = (J-1)*deltaR;
                    u2MaxZ    = (K-1)*deltaZ;
                }
                if(fabs(u[3][J][K]) > fabs(u3Max))//Bt < 0.
                {
                    u3Max    = fabs(u[3][J][K]);
                    u3MaxR    = (J-1)*deltaR;
                    u3MaxZ    = (K-1)*deltaZ;
                }
                if(u[4][J][K] > u4Max)
                {
                    u4Max    = u[4][J][K];
                    u4MaxR    = (J-1)*deltaR;
                    u4MaxZ    = (K-1)*deltaZ;
                }

                u0Avg    += u[0][J][K];
                u1Avg    += u[1][J][K];
                u2Avg    += u[2][J][K];
                u3Avg    += fabs(u[3][J][K]);
                u4Avg    += u[4][J][K];

//Old convergence checks
                if(fabs(Rho[J][K]-uOld[0][J][K])/Rho[J][K] > DensD)
                {
                    DensD   =fabs(Rho[J][K]-uOld[0][J][K])/Rho[J][K];
                    DensMaxR=(J-1)*deltaR;
                    DensMaxZ=(K-1)*deltaZ;
                }
                if(fabs(KE[J][K]-KEold[J][K])/KE[J][K] > KED)
                {
                    KED     =fabs(KE[J][K]-KEold[J][K])/KE[J][K];
                    KEmaxR  =(J-1)*deltaR;
                    KEmaxZ  =(K-1)*deltaZ;
                }
                if(fabs((u[3][J][K]-uOld[3][J][K])/u[3][J][K])>BtD)
                {
                    BtD     =fabs((u[3][J][K]-uOld[3][J][K])
                                         /u[3][J][K]);
                    BtMaxR  =(J-1)*deltaR;
                    BtMaxZ  =(K-1)*deltaZ;
                }
                if(fabs(E[J][K]-uOld[4][J][K])/E[J][K] > EtotD)
                {
                    EtotD   =fabs(E[J][K]-uOld[4][J][K])/E[J][K];
                    EtotMaxR=(J-1)*deltaR;
                    EtotMaxZ=(K-1)*deltaZ;
                }
                if(fabs(Etherm[J][K]-EthOld[J][K])/Etherm[J][K]>EthD)
                {
                    EthD    =fabs(Etherm[J][K]-EthOld[J][K])
                                       /Etherm[J][K];
                    EthMaxR =(J-1)*deltaR;
                    EthMaxZ =(K-1)*deltaZ;
                }

                DensDavg+= fabs(Rho[J][K]-uOld[0][J][K])/Rho[J][K];
                KEDavg  += fabs(KE[J][K]-KEold[J][K])/KE[J][K];
                BtDavg  += fabs((u[3][J][K]-uOld[3][J][K])
                                      /u[3][J][K]);
                EtotDavg+= fabs(E[J][K]-uOld[4][J][K])/E[J][K];
                EthDavg += fabs(Etherm[J][K]-EthOld[J][K])
                                     /Etherm[J][K];

                Count++;
            }
        }
    }

//Convergence checks
    u0Davg    /= Count;
    u1Davg    /= Count;
    u2Davg    /= Count;
    u3Davg    /= Count;
    u4Davg    /= Count;

    u0Avg    /= Count;
    u1Avg    /= Count;
    u2Avg    /= Count;
    u3Avg    /= Count;
    u4Avg    /= Count;

//Old convergence checks
    DensDavg    /= Count;
    KEDavg        /= Count;
    BtDavg        /= Count;
    EtotDavg    /= Count;
    EthDavg        /= Count;

    if(numSteps%50 == 0)
    {
        fprintf(ConvergeDiff, "%i \t %f \t %f \t %e \t %f \t %f \t
                              %e \t %f \t %f \t %e \t %f \t %f \t
                              %e \t %f \t %f \t %e \t %e \t %e \t
                              %e \t %e \t %e \n",
                numSteps, u0DMaxR, u0DMaxZ, u0Dmax, u1MaxR, u1MaxZ,
                u1Dmax, u2DMaxR, u2DMaxZ, u2Dmax, u3DMaxR, u3DMaxZ,
                u3Dmax, u4DMaxR, u4DMaxZ, u4Dmax, u0Davg, u1Davg,
                u2Davg, u3Davg, u4Davg);
        fprintf(ConvergeVal, "%i \t %f \t %f \t %e \t %f \t %f \t
                             %e \t %f \t %f \t %e \t %f \t %f \t
                             %e \t %f \t %f \t %e \t %e \t %e \t
                             %e \t %e \t %e \n",
                numSteps, u0MaxR, u0MaxZ, u0Max, u1MaxR, u1MaxZ,
                u1Max, u2MaxR, u2MaxZ, u2Max, u3MaxR, u3MaxZ,
                u3Max, u4MaxR, u4MaxZ, u4Max, u0Avg, u1Avg,
                u2Avg, u3Avg, u4Avg);
        fprintf(Converge, "%i \t %f \t %f \t %e \t %f \t %f \t %e \t
                          %f \t %f \t %e \t %f \t %f \t %e \t %f \t
```

```
                           %f \t %e \t %e \t %e \t %e \t %e \t %e \n",
                numSteps, DensMaxR, DensMaxZ, DensD, KEmaxR,
                KEmaxZ, KED, BtMaxR, BtMaxZ, BtD, EtotMaxR,
                EtotMaxZ, EtotD, EthMaxR, EthMaxZ, EthD,
                DensDavg, KEDavg, BtDavg, EtotDavg, EthDavg);
    }


    for (J=Rc+1; J<=Ra; J++)
    {
        Jback += Jencl[J][1];
    }

//This is the total current flowing in the channel.
    Jback /= Ra-Rc;
}

//-------------------------------------
void EvalParam()//This is called every convective time scale.
{
    for(Zone = 1; Zone <= 3; Zone++)
    {
        for (j=J=Rmin[Zone]+2; J<=Rmax[Zone]-1; j++,J++)
        {
            for(k=K=Zmin[Zone]+2; K<=Zmax[Zone]-1; k++,K++)
            {
                jDens[J][K] = sqrt(pow(0.5*(jr[J][k]
                                      +jr[J][k-1]),2.0)+
                                   pow(0.5*(jz[j][K]
                                      +jz[j-1][K]),2.0));
            }
        }

        for (J=Rmin[Zone]+1; J<=Rmax[Zone]; J++)
        {
            for (K=Zmin[Zone]+1; K<=Zmax[Zone]; K++)
            {
                P[J][K]   = p[J][K]+(0.5*BSq[J][K]/Mu);
                aSq[J][K] = gam[J][K]*p[J][K]/Rho[J][K];
                CmSq[J][K]= aSq[J][K]+(BSq[J][K]/(Mu*Rho[J][K]));
                Cf[J][K]  = sqrt(CmSq[J][K]);//Cf = Cfz.
//Largest eigenvalue
                if (Vr[J][K]<0)
                    SpecRadR[J][K] = fabs(Vr[J][K]-Cf[J][K]);
                else
                    SpecRadR[J][K] = fabs(Vr[J][K]+Cf[J][K]);

                if (Vz[J][K]<0)
                    SpecRadZ[J][K] = fabs(Vz[J][K]-Cf[J][K]);
                else
                    SpecRadZ[J][K] = fabs(Vz[J][K]+Cf[J][K]);

                CoulombLog[J][K]   = log(sqrt(Eps0*kBoltz*Te[J][K]
                                            /(q*q*n[J][K]))/
                                      (q*q/(12*PI*Eps0*kBoltz
                                               *Te[J][K])));
                EIcollFreq[J][K]   = 3.63312216e-06*n[J][K]
                                        *CoulombLog[J][K]
                                      *pow(Te[J][K],-1.5);
                if(EIcollFreq[J][K] < 1.0e9)
                    EIcollFreq[J][K]= 1.0e9;
                ElecGyro[J][K]        = q*sqrt(BSq[J][K])/mEl;
                ElecHall[J][K]        = ElecGyro[J][K]
                                       /EIcollFreq[J][K];
                Vde[J][K]            = jDens[J][K]/(q*n[J][K]);
                Vti[J][K]            = sqrt(2*kBoltz*Th[J][K]/mAr);
                if(Vde[J][K]/Vti[J][K] > 1.5)
                {
                    AnomFreqEl[J][K]
                             = EIcollFreq[J][K]*(((0.192)
                              +(3.33e-2*ElecHall[J][K])
                              +(0.212*pow(ElecHall[J][K],2.0))
                              +(-8.27e-5*pow(ElecHall[J][K],3.0)))
                              +((Th[J][K]/Te[J][K])*((1.23e-3)
                              +(-1.58e-2*ElecHall[J][K])
                              +(-7.89e-3*pow(ElecHall[J][K],2.0)))));
                }
                else
                {
                    AnomFreqEl[J][K]= 0.0;
                }
                TotCollFreq[J][K]=EIcollFreq[J][K] + AnomFreqEl[J][K];
                Res[J][K]        =mEl*TotCollFreq[J][K]/(n[J][K]*q*q);

//This is so that the near vacuum regions
//do not set time step constraint.
                if(Res[J][K] > 7.5e-4)
                    Res[J][K] = 7.5e-4;
                kTherm[J][K]    = 3.20*kBoltz*kBoltz*n[J][K]*Te[J][K]/
                                        (mEl*EIcollFreq[J][K]);
                kIon[J][K]        = 2.84586e-13*pow(Th[J][K],2.5)
                              /log(1.239e7*sqrt(pow(Th[J][K],3.0)
                                                    /n[J][K]));
                if(kTherm[J][K] > 20.0)
                    kTherm[J][K]    = 20.0;
            }
        }
    }//end for all zones.

//At the inlet, the value of SpecRadZ is needed.
    for (J=Rc+1; J<=Ra; J++)
    {
        aSq[J][0]        = gam[J][K]*p[J][0]/Rho[J][0];
        CmSq[J][0]        = aSq[J][0]+(BSq[J][0]/(Mu*Rho[J][0]));
        Cf[J][0]        = sqrt(CmSq[J][0]);
        SpecRadZ[J][0]    = Vz[J][0]+Cf[J][0];
    }
}
//---------------------------------------
void EquationOfState()
{
//Calculates the temperature based on the ratio of pressure to density.
//The coefficients are obtained by a fit to the data
//from a table of partition functions.
    for(Zone = 1; Zone <= 3; Zone++)
    {
        for (J=Rmin[Zone]+1; J<=Rmax[Zone]; J++)
        {
            for (K=Zmin[Zone]+1; K<=Zmax[Zone]+1; K++)
            {
                PR[J][K]    = ph[J][K]/Rho[J][K];

//The model used above is only good for temperatures above ~5000 K.
                if(PR[J][K] <= 2994000)
                    In1 = 1;
                else
                    In1 = 0;
                if(PR[J][K] > 2994000)
                    In2B = 1;
                else
                    In2B = 0;

                if(PR[J][K] <= 5507000)
                    In2A = 1;
                else
                    In2A = 0;

                In2 = In2A * In2B;

                if(PR[J][K] > 5507000)
                    In3B =1;
                else
                    In3B = 0;

                if(PR[J][K] <= 9827000)
                    In3A = 1;
                else
                    In3A = 0;

                In3 = In3A * In3B;

                if(PR[J][K] > 9827000)
                    In4B = 1;
                else
                    In4B = 0;

                if(PR[J][K] <= 18290000)
                    In4A = 1;
                else
                    In4A = 0;

                In4 = In4A * In4B;

                if(PR[J][K] > 18290000)
                    In5 = 1;
                else
                    In5 = 0;

                K0 = (In2*7935) + (In4*12460) + (In5*14820);
                K1 = (In1*0.00599) + (In2*0.00119) + (In3*0.00317)
                    + (In4*0.00094) + (In5*0.000811);
                K2 = (-7.18e-10 * In1) + (-9.79e-11 * In3);

                if(PR[J][K] < 1.0e6)
                {
                    Th[J][K] = ph[J][K]/((Rho[J][K]/mAr)*kBoltz);
                    gam[J][K]= 5.0/3.0;
                }
                else
                {
                    Th[J][K] = ( ((K2*PR[J][K])+K1)*PR[J][K] ) + K0;
                }
            }
        }
    }//end for all zones.
}
//-------------------------------------
void TerminalChars()
{
    Tinlet = RhoIn*VzIn*VzIn*PI*(pow(Ranode,2.0)-pow(Rcathode,2.0));
    for(K= Za+1; K<=numZ; K++)
        Tupstream += 2*PI*RAD*Rho[numR][K]*Vr[numR][K]*Vz[numR][K]
                      *deltaZ;

    for(J=1; J<= numR; J++)
        Texit += 2*PI*Rho[J][numZ]*Vz[J][numZ]*Vz[J][numZ]
                      *(J-0.5)*deltaR*deltaR;

    Thrust = Texit+Tupstream-Tinlet;

//The coefficient 0.15 was obtained from Villani.
    MaeckerT = 1.0e-7*(log(Ranode/Rcathode) + 0.15)*Jmax*Jmax;

    Isp = (Thrust/MassFlowRate)/go;
}
//-------------------------------------
void EvalGamma()
{
    for(Zone = 1; Zone <= 3; Zone++)
    {
        for (J=Rmin[Zone]+1; J<=Rmax[Zone]; J++)
        {
            for (K=Zmin[Zone]+1; K<=Zmax[Zone]; K++)
            {
                gamOld[J][K] = gam[J][K];

                if(Th[J][K] < 8000.0)
                {
                    gam[J][K] = 5.0/3.0;
                }
                else
                {
                    if(Th[J][K] < 13000.0)
                    {
                        gam[J][K] = 1.11217+(0.529956*exp(-pow
                           ((Th[J][K]-8050.61)/1318.85,2.0)));
                    }
                    else
                    {
                        if(Th[J][K] < 40000.0)
                        {
                            gam[J][K] = 1.1054+(0.0252666*exp(
                        -pow((Th[J][K]-15142.8)/2394.06,2.0)));
                        }
//Above 40000K, gamma is almost a const ˜ 1.1.
                        else
                        {
                            gam[J][K] = 1.10;
                        }
                    }
                }

//Relaxation for gamma.
                gam[J][K] = (0.99999*gamOld[J][K])+(0.00001*gam[J][K]);

//                EintH[J][K] = ph[J][K]/(gam[J][K]-1);
//                E[J][K] = EintEl[J][K]+EintH[J][K]+(0.5*Rho[J][K]
                                   *VSq[J][K])+(0.5*BSq[J][K]/Mu);
//                u[4][J][K] = E[J][K];
            }//end of ’k’
        }//end of ’j’
    }//end for all zones.
}
//-------------------------------------
```

```
void Saha()
{
//From the given Te & n, it computes ne, ni, nii, niii and nA.
    for(Zone = 1; Zone <= 3; Zone++)
    {
        for (J=Rmin[Zone]+1; J<=Rmax[Zone]; J++)
        {
            for (K=Zmin[Zone]+1; K<=Zmax[Zone]; K++)
            {
//Normalizing
                double h=exp(1.5*log(Te[J][K]))*2.41500819e21/n[J][K];

//Local variables
                double K1 = 11.   * h * exp(-1.82892573e5/Te[J][K]);
                double K2 =  3.72 * h * exp(-3.20294100e5/Te[J][K]);
                double K3 =  1.6  * h * exp(-4.74638720e5/Te[J][K]);
                double K4 =  2.1  * h * exp(-6.92810064e5/Te[J][K]);
                double K5 =  1.04 * h * exp(-8.72685373e5/Te[J][K]);
                double K6 =  0.48 * h * exp(-1.05836311e6/Te[J][K]);

                double K12 = K1*K2;
                double K123 = K12*K3;
                double K1234 = K123*K4;
                double K12345 = K1234*K5;
                double K123456 = K12345*K6;

                double a[8];

                a[7] = 1.;
                a[6] = K1;
                a[5] = K12 - K1;
                a[4] = K123 - 2.*K12;
                a[3] = K1234 - 3.*K123;
                a[2] = K12345 - 4.*K1234;
                a[1] = K123456 - 5.*K12345;
                a[0] = -6.*K123456;

                double neN=6.; //Normalized initial guess for ne=6*n
                double ne_old;
                do
                {
                    double poly=a[7];
                    double dp=0.;
                    for(int i=6;i>=0;i--)
                    {
                        dp=dp*neN+poly;
                        poly=poly*neN+a[i];
                    }
                    ne_old=neN;
                    neN = neN - poly/dp; // Newton-Raphson
                }
                while(fabs(neN-ne_old)/neN > 1.e-6);

                double ne6=neN*neN*neN;
                ne6=ne6*ne6;

                nA[J][K]  =n[J][K]*ne6/(K123456+neN*(K12345+neN
                    *(K1234+neN*(K123+neN*(K12+neN*(neN+K1))))));
                ni[J][K]  =nA[J][K]*K1/neN;
                nii[J][K] =ni[J][K]*K2/neN;
                niii[J][K]=nii[J][K]*K3/neN;

//Ensuring that there is at least 1 particle of every species.
                nA[J][K]    = (nA[J][K]<1.) ? 1. : nA[J][K];
                ni[J][K]    = (ni[J][K]<1.) ? 1. : ni[J][K];
                nii[J][K]    = (nii[J][K]<1.) ? 1. : nii[J][K];
                niii[J][K]    = (niii[J][K]<1.) ? 1. : niii[J][K];

//Saving the old value before updating ne.
                neOld[J][K] = ne[J][K];

                ne[J][K] = neN*n[J][K]; //De-normalizing.
                if(ne[J][K]/n[J][K] < 0.05)
                    ne[J][K]= 0.05*n[J][K];

//Relaxation for ne.
                ne[J][K]    = (0.999*neOld[J][K])+(0.001*ne[J][K]);

                pe[J][K]    = ne[J][K]*kBoltz*Te[J][K];
//Freeze everything else
//                EintEl[J][K]= pe[J][K]/(gamConst-1);

                ph[J][K]    = n[J][K]*kBoltz*Th[J][K];
//              EintH[J][K]    = ph[J][K]/(gam[J][K]-1);
//              E[J][K]        = EintEl[J][K] + EintH[J][K]
//              + (0.5*BSq[J][K]/Mu)+ (0.5*Rho[J][K]*VSq[J][K]);
//              u[4][J][K]    = E[J][K];

                ph[J][K]    = p[J][K] - pe[J][K];

//              EintH[J][K]    = E[J][K] - ((0.5*BSq[J][K]/Mu)
//                     + (0.5*Rho[J][K]*VSq[J][K])) - EintEl[J][K];
//                ph[J][K]    = (gam[J][K]-1)*EintH[J][K];

            }//end of ’K’
        }//end of ’J’
    }//end for all Zones.
}
//---------------------------------------
void Transport()
{
//Obtained from Joerg Heiermann.

    Zeff[J][K] = ne[J][K]/n[J][K];
    Ce[J][K]    = sqrt(2*kBoltz*Te[J][K]/mEl);

//Computes collision cross sections of electrons with
//each of the heavy species
    CGvosQ[J][K]= 2.19320509e-10 * log(1. + 1.53478873e14
                              *Te[J][K]*Te[J][K]*Te[J][K]/
               (ne[J][K]*Zeff[J][K]*Zeff[J][K]*(Zeff[J][K]+1.)));
//Computes the collision frequencies of electrons with
//each of the heavy species
//Neutrals to be included later: Qea = 4.0e-20.
    Qei[J][K]    = CGvosQ[J][K]/(Te[J][K]*Te[J][K]);
    Qeii[J][K]    = CGvosQ[J][K]*2*2/(Te[J][K]*Te[J][K]);
    Qeiii[J][K]    = CGvosQ[J][K]*3*3/(Te[J][K]*Te[J][K]);

    nuei[J][K]    = ni[J][J]*Qei[J][K]*Ce[J][K];
    nueii[J][K]    = nii[J][J]*Qeii[J][K]*Ce[J][K];
    nueiii[J][K]= niii[J][J]*Qeiii[J][K]*Ce[J][K];

//Computes resistivity
    double Sum=0.;
    Sum += nuei[J][K]+nueii[J][K]+nueiii[J][K];
    Res[J][K] = Sum/2.11355633e-8*ne[J][K];
}
//---------------------------------------
```