A STUDY OF THE PLASMA ATMOSPHERE AROUND A MAGNETIZED NEUTRON STAR

A thesis presented to the faculty of
San Francisco State University
In partial fulfillment of
The Requirements for
The Degree

Master of Science
In
Physics

by

Manav Singh

San Francisco, California

December 2015

CERTIFICATION OF APPROVAL

I certify that I have read *A STUDY OF THE PLASMA ATMOSPHERE AROUND A MAGNETIZED NEUTRON STAR* by Manav Singh and that in my opinion this work meets the criteria for approving a thesis submitted in partial fulfillment of the requirements for the degree: Master of Science in Physics at San Francisco State University.

Dr. Susan Lea
Professor of Physics

Dr. Ron Marzke
Professor of Physics

Dr. Joseph Barranco
Associate Professor of Physics

A STUDY OF THE PLASMA ATMOSPHERE AROUND A MAGNETIZED NEUTRON STAR

Manav Singh
San Francisco State University
2015

In this thesis I use a Lagrangian fluid code to study the plasma atmosphere around an accreting, magnetized neutron star in a high mass x-ray binary. An old version of the code, initially written in FORTRAN, is rewritten in Python using modern algorithms and updated numerical techniques. The code is validated against known analytical solutions of various physical processes relevant to the system. It is then used to numerically time-evolve the system in question in order to study the behavior of a generated shock wave. I explain why I believe the code was unable to achieve this overall goal and elaborate on where future work should be focused in order to resolve the issue.

I certify that the Abstract is a correct representation of the content of this thesis.

Chair, Thesis Committee Date

ACKNOWLEDGMENTS

I'd like to thank Dr. Susan Lea for her endless support, both moral and academic, in not only this project but also in my career as a master's student. I would be no where without her guidance as an advisor, professor, and mentor. I can only hope to one day have a resolve and passion for physics as resolute as her.

Besides my advisor I'd also like to thank the other members of my committee, Dr. Joe Barranco and Dr. Ron Marzke, for their encouragement and optimism and also for presenting me with difficult challenges to overcome during my defense, just as I had hoped.

I thank my peers and colleagues for many thought-provoking discussions. They've consistently challenged me to answer the tough questions and I would have it no other way.

Finally I thank my parents and my sister for their unwavering faith in my ability to succeed. They've made me who I am and I would be nothing without them.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Background

It would be wonderful to say that I was the first person to work on this project, but I would be no where without the work of my predecessors. The work presented in this thesis is a continuation of two earlier projects by Gary Linford (Linford, 1985) and David Stratton (Stratton, 1989). Each had their own version of the code, though they differed in goals and their use of the code. All three of us worked with a set of papers written by Susan Lea and Jonathan Arons as our starting point (see Arons and Lea, 1976a,b).

1.1 Description of the System

In a high mass x-ray binary, a blue giant and a neutron star for example, stellar wind from the donor star is captured by the accretor. The goal of this thesis is to study the atmosphere of the neutron star, the accretor. As plasma accretes into the neutron star's atmosphere, the magnetic field generated by the star's magnetic moment begins to "fight" the gravitational force acting on the incoming plasma. The magnetic field pushes the plasma outward and gravity pulls the plasma inward. Both forces have a position dependent strength, so there is a position where they will be equal and opposite. This location is known as the magnetopause, the radius at which the inward force of gravity on the plasma matches the neutron star's outward magnetic force on the plasma.

The magnetopause effectively acts as a barrier to the plasma that stops the inflow and slowly the plasma density at this barrier will begin to increase. A shock wave will form and propagate outward. What is unique about this system is that as Arons and Lea describe (Arons and Lea, 1976a), there is a critical density above which mass will begin to leak through the magnetopause. This leakage will continue until the density at the magnetopause falls below the critical density.

Solving this system as a fully 3-dimensional problem would be quite the challenge. A crucial assumption allows us to reduce the problem to a single dimension: the accretion process is essentially spherically symmetric. The closer our region of study is to the neutron star, the more valid this assumption becomes. This has been shown by Hunt (Hunt, 1971). We must also be careful of the fact that the magnetic field lines around the neutron star are not spherically symmetric. However, work done by Arons and Lea (Arons and Lea, 1976a) shows that the asymmetric nature of the magnetic field lines is important only around the poles of the neutron star that make up a small fraction of the solid angle around the star. Taking these works into consideration we see that a spherically symmetric system can model the physics of this problem well.

This is a general description of the system, a more detailed explanation of the various processes involved is provided in chapter 2. My predecessors and I have all studied slightly different aspects of this system as I will explain in the next few sections.

Some parameters of the system are outlined in Table 1.1.

Parameter	**Variable Representation**	**Value**
Neutron Star Mass	M_{ns}	10^{33} g
Neutron Star Radius	R_{ns}	10^{6} cm
Neutron Star Magnetic Moment	μ	10^{30} Gauss cm^{3}
Accretion Disk Temperature	$RAtemp$	10^{4} K
Accretion Disk Radius	$AccRad$	2.5×10^{10} cm
Adiabatic Index	γ	$\frac{5}{3}$

Table 1.1: System Parameters

1.2 History of the Code

The original version of the BURST code was written in 1985 by Gary Linford (Linford, 1985) as part of his Master's thesis. Gary was focused primarily on x-ray bursts generated by our system. More specifically, Gary was studying the infall of mass on to the neutron star. His version of the code was written in FORTRAN.

Gary's original code has been through several modifications as it changed hands over the years. Michael Rinaldi, John Winchester, Susan Lea, and David Stratton have changed and improved the code over the years. I greatly appreciate their efforts in providing useful comments that sped up my learning process.

1.3 The Phoenix Code

The last person to work on this code before it fell into my hands was David Stratton (Stratton, 1989). He worked on this code in 1989, over 25 years before me. His version of the code has been a valuable resource in my work, but it did require a few crucial changes. I've made two significant modifications to the code that warrant mention.

Every earlier version of the code before can be considered an edit of the original code. The original code was written in FORTRAN, a language that has aged significantly over the years. I started my work by rewriting the code in Python, a much more modern language with ample documentation for advanced use. The rewriting process was invaluable in helping me understand how the code worked.

The second change I made involved a modification in the way the physical equations underwent time-evolution. The old code evolved the system through a half-time step and then a full time step. I was able to show that that process was an unnecessary complication. My version of the code now evolves the equations one full time step at a time.

I've made many other small changes to the code, but these two are the biggest modifications. I've heavily commented the various sections and subroutines so that future users will have no trouble understanding their purpose. The code is now much more modern, preparing it for use in future work.

Chapter 2

The Physics and The Code

To make the code more readable and easier to debug it is broken up into a small set of isolated functions. A flowchart of the code's structure is presented in Figure 2.1. Each function plays a crucial role in the code's operation.

The setup routine is told what type of analysis is about to be run and enables or disables the appropriate physical factors such as cooling, gravity, accretion, or zone combination. It also assigns a location to the lowest interface in the atmosphere and runs the appropriate initialization routine.

The initialization routine initializes the atmosphere with the conditions relevant to the analysis that is about to be run. The initialization parameters for each analysis type are outlined in Chapter 3.

As outlined in Section 2.3, the time step routine calculates the time step based on the state of the system. This routine is also responsible for combining zones when necessary. An explanation of zone combination is provided in Section 2.6.

Section 2.1 describes the process by which the iteration routine computes the physical state of the system. This routine is also responsible for calculating mass accretion (Section 2.4) and mass infall (Section 2.5).

Data output is a critical part of the code, however it must be handled with as much care as any other aspect of the code. Our data output rate must be high enough that we don't miss any of the system's behavior. However, a high output rate results in large output files which are much slower to analyze. Conversely a low output rate makes data analysis very quick, but we could miss out on crucial behavior. The output rate is chosen based on the type of analysis being performed. The Blast Wave Test (Section 3.1) and The Full Run (Section 3.4) require a high output rate whereas the Rest Test (Section 3.2) and the Cooling Test (Section 3.3) allow for a low output rate.

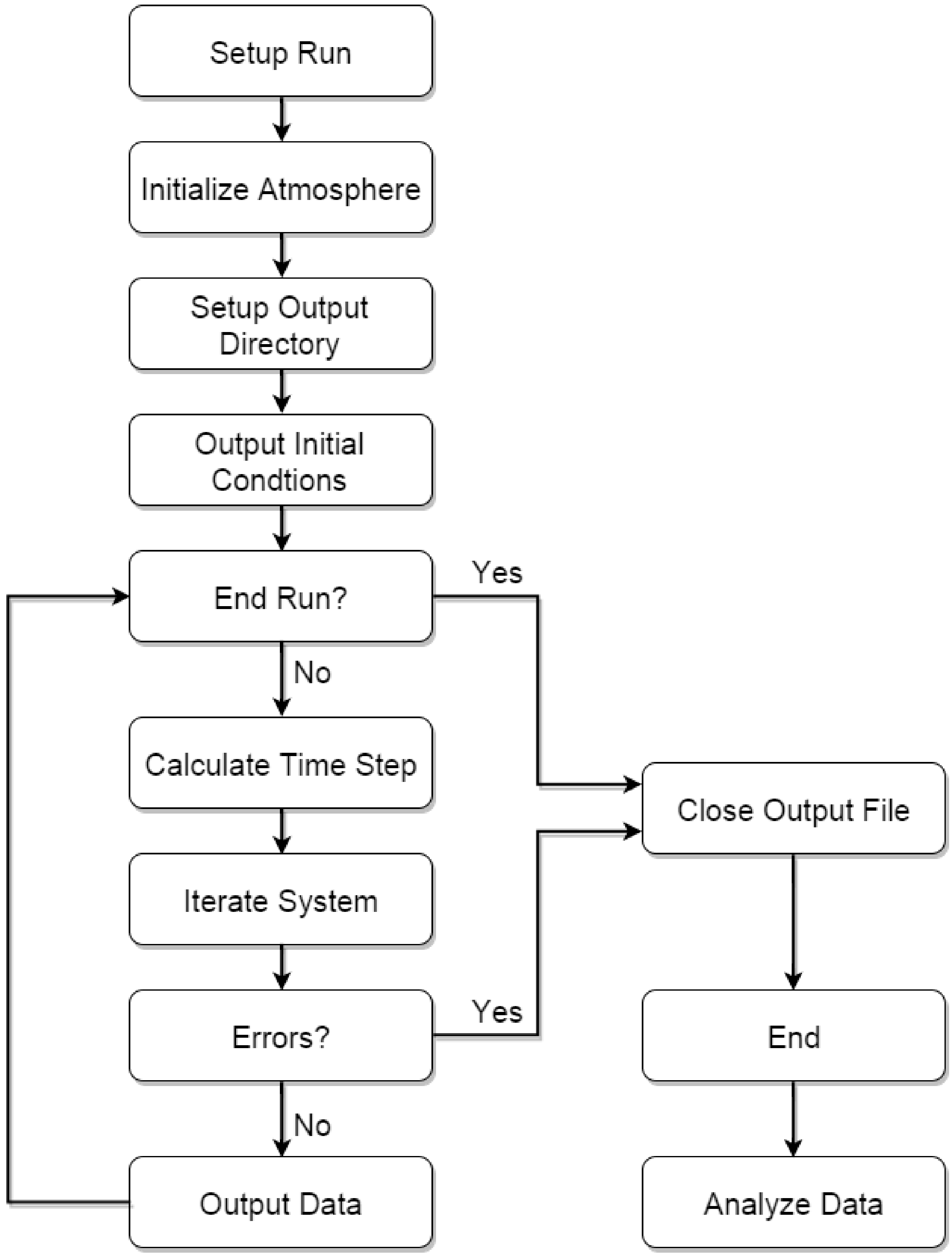

Figure 2.1: Phoenix Code Flowchart

2.1 The Fluid Equations

This Lagrangian fluid code breaks up the plasma into discrete pieces called zones. Each zone is separated by a zone interface. It must be noted that the interfaces are simply computational tools used to isolate zones and do not reflect the physical presence of any actual boundary. In this model the interfaces move and the plasma does not cross interfaces. The flow of the plasma is governed by a few fundamental differential equations relating the state variables of the system. The interfaces themselves are only described by their position, velocity, and inertia. The plasma zones are described by their density, specific energy, artificial viscosity, pressure, and mass.

The primary equations that describe the system are as follows,

$$\frac{\partial U}{\partial t} = 4\pi R^2 \frac{\delta(P+Q)}{\delta(Inertia)} - \frac{GM_{ns}}{R^2} \tag{2.1}$$

$$\frac{\partial E}{\partial t} = -(P+Q)\frac{\partial V}{\partial t} - E_0 V T^{1/2} n^2 \tag{2.2}$$

$$P = (\gamma - 1)\frac{E}{V} \tag{2.3}$$

$$Q = Q_0 \rho (\delta U)^2 \tag{2.4}$$

Where U is the interface velocity, R is the interface position, P is the zone pressure, Q is the artificial zone viscosity, $Inertia$ is the inertia of an interface, ρ is the zone plasma density, Q_0 is the artificial viscosity constant, and γ is the adiabatic index.

Since the plasma is essentially a monatomic gas, γ is 5/3.

In the following sections I will describe why these equations are needed, how they are implemented, and any supplemental equations needed to solve these.

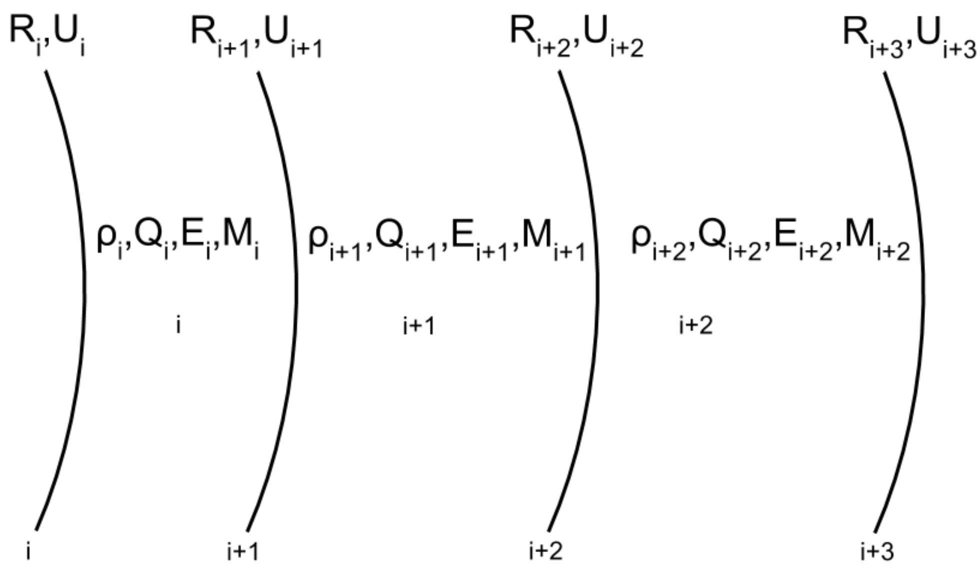

Figure 2.2: Interface and Zone Description

Much of the literature labels interfaces with half-integers and zones with integers, this increases complexity when writing pseudo-code and we will not be using this convention. We assign integer indices to all variables and simply keep in mind that only velocities and positions refer to interfaces, while all other variables are properties of the zones (The interfaces do have an inertia but this variable is labeled "interfaceInertia" to avoid confusion with the zone masses). The convention to keep in mind is that zones have the same index as the interface bounding them from below. Figure 2.2 illustrates this convention.

2.1.1 Physics at the Interfaces

The movement of any interface is determined by the forces acting upon it. Every interface is influenced by gas pressure from the zones around it and by gravity from the neutron star. Though the interfaces do not have any physical mass, they do have inertia due to the mass of the surrounding zones. This inertia allows gravity to influence the interfaces.

For the interior zones (Besides the topmost and bottommost zones) we can use the gas pressure in each zone to determine the force that each zone exerts on an interface,

$$F_i = \big[(P_{i-1} + Q_{i-1}) - (P_i + Q_i)\big] \times A_i$$

Where P_i and Q_i are the pressure and artificial viscosity of the ith zone and the area of the interface is defined as,

$$A_i = 4\pi R_i^2$$

From the position and inertia of the interface we can also determine the force due to gravity,

$$F_i = -\frac{GM_{ns}Inertia_i}{R_i^2}$$

Where M_{ns} is the mass of the neutron star and G is Newton's Gravitational Constant. From the net force on the interfaces we can determine their change in velocity over a time step,

$$\delta U_i = F_i \times \frac{\delta t}{Inertia_i}$$

Finally we can compute the total change in velocity for an interface,

$$\delta U_i = \left(4\pi R_i^2\big[(P_{i-1} + Q_{i-1}) - (P_i + Q_i)\big] - \frac{GM_{ns}Inertia_i}{R_i^2}\right)\frac{\delta t}{Inertia_i} \tag{2.5}$$

Since velocity is the time derivative of position we can use a first order finite forward difference to find the change in interface position,

$$U_i = \frac{\partial R_i}{\partial t}$$
$$U_i^n = \frac{R_i^{n+1} - R_i^n}{\delta t}$$

With this we can now compute δR_i,

$$\delta R_i = U_i \delta t \tag{2.6}$$

For the topmost and bottommost interfaces we must take additional care. The top interface has no zone above it, but of course there is still plasma present. We expect no artificial viscosity in the top zone and gas pressure above and below the top interface is assumed to be approximately equal. Therefore only gravity contributes to the change in velocity of the top zone,

$$\delta U_{top} = -\frac{GM_{ns}\delta t}{R_{top}^2}$$
$$\delta R_{top} = U_{top}\delta t$$

U_{top} has been updated with the change δU_{top}.

The bottom interface is also affected by magnetic pressure from the neutron star's magnetic field. The strength of this field at the equator is,

$$|\vec{B_{ns}}| = \frac{\mu}{R^3}$$

μ is the star's magnetic moment. This magnetic field directly translates to a magnetic pressure on the bottom interface of the plasma,

$$P_B = \frac{B_{ns}^2}{8\pi} = \frac{\mu^2}{8\pi R_{bottom}^6}$$

This pressure creates an additional force on the bottom interface of the plasma. The net force on the bottom interface becomes,

$$F_{bottom} = \left[\frac{\mu^2}{8\pi R^6} - (P_{bottom} + Q_{bottom})\right] \times A_{bottom}$$

The net change in velocity of the bottom interface over a time step then becomes,

$$\delta U_{bottom} = 4\pi R_{bottom}^2 \left(\frac{\mu^2}{8\pi R_{bottom}^6} - (P_{bottom} + Q_{bottom})\right) \frac{\delta t}{Inertia_{bottom}}$$

For the bottommost interface we neglect the force of gravity. Since every interface above is influenced by gravity, the gas pressure generated will account for the gravitational influence.

The physical conditions within the zones depend only on the position and velocity of the bounding interfaces, with these established we can move on.

2.1.2 Physics in the Zones

The physical conditions within the zones depend only on the positions and velocities of the bounding interfaces. The equations that describe the physics within the zones are as follows, Density and Specific Volume:

$$\rho = \frac{3M}{4\pi(\delta R^3)} \tag{2.7}$$

$$V = \frac{1}{\rho} = \frac{1}{\eta m_p} \tag{2.8}$$

Specific Energy:

$$\frac{\partial E}{\partial t} = -(P+Q)\frac{\partial V}{\partial t} - E_0 N_a \eta \sqrt{T} \tag{2.9}$$

Temperature and Pressure:

$$T = \frac{E}{3N_a K_b} \tag{2.10}$$

$$P = (\gamma - 1)\frac{E}{V} \tag{2.11}$$

Artificial Viscosity:

$$Q = \begin{cases} Q_0 \rho \delta U^2 & \text{If compression} \\ 0 & \text{If no compression} \end{cases} \tag{2.12}$$

In these expressions N_a is Avogadro's Number, K_b is Boltzmann's Constant, η is the number of protons, m_p is the proton mass, γ is the adiabatic index, E_0 is the Bremsstrahlung cooling constant, and Q_0 is the artificial viscosity constant.

The artificial viscosity equation is of particular importance to a numerical simulation of fluids. The thickness of a shock wave can often be much less than that of a zone. This causes fluctuations in the system just before and after shock. Artificial viscosity helps smooth out these fluctuations so that shock front is modeled correctly. Of course, this viscosity is only needed near a shock. Zone compression indicates the need for viscosity. The value of constant Q_0 is determined in the Blast Wave Test (see section 3.1).

2.2 Discretizing the Equations

We are using a Lagrangian fluid code to evolve this system. The interfaces are free to move and the mass within a zone is constant (Accretion and infall may effect the mass in the innermost and outermost zones only). In order to apply our equations to the grid we must first write them in discrete form. We will make use of first forward differentiation to do this. All variables will be assigned a superscript to signify a time step (zone mass does not receive a superscript since it does not change in time) and a subscript to signify a zone (An example of this is done in section 2.1.1).

For reference, the first order numerical derivative is defined as follows:

$$\frac{\partial f}{\partial t} = \frac{f_i^{n+1} - f_i^n}{\delta t}$$

We apply this to our equations to obtain discrete forms.

Density and Specific Volume:

$$\rho_i^{n+1} = \frac{3M_i}{4\pi\left({R_{i+1}^{n+1}}^3 - {R_i^{n+1}}^3\right)} \tag{2.13}$$

$$V_i^{n+1} = \frac{1}{\rho_i^{n+1}} \tag{2.14}$$

$$\eta_i^{n+1} = \frac{\rho_i^{n+1}}{m_p} \tag{2.15}$$

Specific Energy:

$$E_i^{n+1} = E_i^n - (P_i^n + Q_i^n)(V_i^{n+1} - V_i^n) - E_0 N_a \eta_i^n \sqrt{T_i^n}\delta t \tag{2.16}$$

Temperature and Pressure:

$$T_i^{n+1} = \frac{E_i^{n+1}}{3N_a K_b} \tag{2.17}$$

$$P_i^{n+1} = (\gamma - 1)\frac{E_i^{n+1}}{V_i^{n+1}} \tag{2.18}$$

Artificial Viscosity:

$$Q_i^{n+1} = \begin{cases} Q_0\rho\left(U_i^{n+1} - U_{i+1}^{n+1}\right)^2 & \text{If compression} \\ 0 & \text{If no compression} \end{cases} \tag{2.19}$$

Note that in equation 2.16 the ion number density η is defined as,

$$\eta_i^n = \rho_i^n \frac{N_a}{m_{hydrogen}}$$

Where $m_{hydrogen}$ is the molecular mass of hydrogen, taken to be 1.0 g/mol.

With the equations discretized, the code is ready to evolve the system.

2.3 Calculating The Time Step

To maximize efficiency the code uses a dynamic time step. This time step is based on the velocity of the interfaces, fast moving zones lead to small time steps. The time step is determined such that it is always small enough that no two interfaces will crossover one another, but large enough that we are not wasting cycles computing negligible changes in the system.

The time step routine loops through the interfaces and finds the interface that is most likely to cross over adjacent interface.

We begin by moving to a reference frame in which the interface is at rest and determine the width of the adjacent zone.

$$U_{rel} = |U_i - U_{i+1}|$$

$$\delta R = R_{i+1} - R_i$$

The fastest that information can travel from one interface to the next is,

$$U_{max} = U_{rel} + C_i$$

Where C_i is the sound speed of the zone above interface i. Using these two pieces of information we can determine the minimum time it would take for information to travel from interface i to interface $i + 1$.

$$t_{i,min} = \frac{\delta R}{U_{max}}$$

$t_{i,min}$ is then saved for every interface. The time step is chosen to be 1/10th of the smallest $t_{i,min}$.

2.4 Accretion

Plasma from the companion star is constantly entering our system and creating pressure on the plasma already present, it is crucial to take this accretion into consideration.

We start with the standard equation for Bondi accretion (Bondi, 1952),

$$\dot{M} = 4\pi R^2 \rho U \tag{2.20}$$

At the top of the system this plasma is in free fall toward the star,

$$\dot{M} = 4\pi R^2 \rho \sqrt{\frac{GM_{ns}}{R}} = 4\pi R^{3/2} \rho \sqrt{GM_{ns}} \tag{2.21}$$

From this we compute the density of the accreting plasma.

$$\rho_{freefall} = \frac{\dot{M}}{4\pi R^{3/2}\sqrt{GM_{ns}}}$$

The top interface will fall inward every time step. When it has fallen far enough to fit a new zone (the width of which we determine using equation 2.21 and fixed values of $\dot{M}$ and ρ.),we add a new zone above the top interface. This new zone is given free fall density and a mass equal to the mass of the initial zones. The new zone has no artificial viscosity and its temperature is the same as the plasma temperature at the accretion radius. The new interface, which is now the topmost interface, is assigned free fall velocity.

The system iteration routine (see Figure 2.1) handles accretion based on whether or not there is sufficient space to add a new zone at the top of the atmosphere.

2.5 Mass Infall

As derived in (Arons and Lea, 1976b), there is a critical density above which plasma will begin to leak through the magnetopause. The critical density is,

$$\rho_{crit} = \frac{1.3445\mu^2}{4\pi G M_{ns} R_0^5} \tag{2.22}$$

Where R_0 is the position of the magnetopause. We will use the term "gate" to refer to the physical magnetic barrier by which the magnetopause stops the flow of plasma. The gate opens when the density at the magnetopause (in other words, the density in the lowest zone) exceeds the critical density, at which point the plasma in the lowest zone is pulled down to the surface of the neutron star by gravity (the plasma would still flow along the magnetic field lines).

While the gate is open plasma will leave the lowest zone every time step. Gravity becomes the dominant force on the plasma once the gate opens. From the volume of plasma that crosses the magnetopause in one time step and the density of the falling plasma we can determine the mass that is dropped.

The falling plasma decreases the density near the magnetopause. Eventually the density will fall below the critical density and the gate will close.

This process of mass infall decreases the pressure behind the shock (by "behind the shock" I am referring to the space that is radially lower than the shock). This will cause the shock to move inward until the density at the magnetopause falls below the critical density and the mass infall stops, which we predict will result in the outward motion of the shock again. We predict some type of episodic (not necessarily periodic) motion of the shock.

2.6 Zone Combination

When the gate is open, the mass in the bottom zone decreases. This leads to a decrease in density and consequently pressure in the bottom zone. Ultimately the bottom zone will become very narrow and the time step routine will always choose this zone to set the time step. The required time step becomes smaller and smaller. To deal with this issue we combine the bottom two zones when the time step satisfies,

$$\delta t \sim \frac{\delta R_{bottom}}{C_{bottom}} \tag{2.23}$$

Where δR_{bottom} is the width of the bottom zone and C_{bottom} is the sound speed of the bottom zone.

The zone combination routine is a subroutine of the time step calculation routine. The routine begins by computing the time step as described in section 2.3. It then checks the time step against the zone combination condition, equation 2.23. If zones need to be combined, the routine combines the zones by removing the common interface and conserving energy and momentum. The mass of the bottom two zones is summed into a single zone and the density is determined by the position of the two new bounding interfaces. Energy and artificial viscosity are calculated by a mass-weighted average of the two zones. Temperature and pressure are then calculated based on the new energy and density.

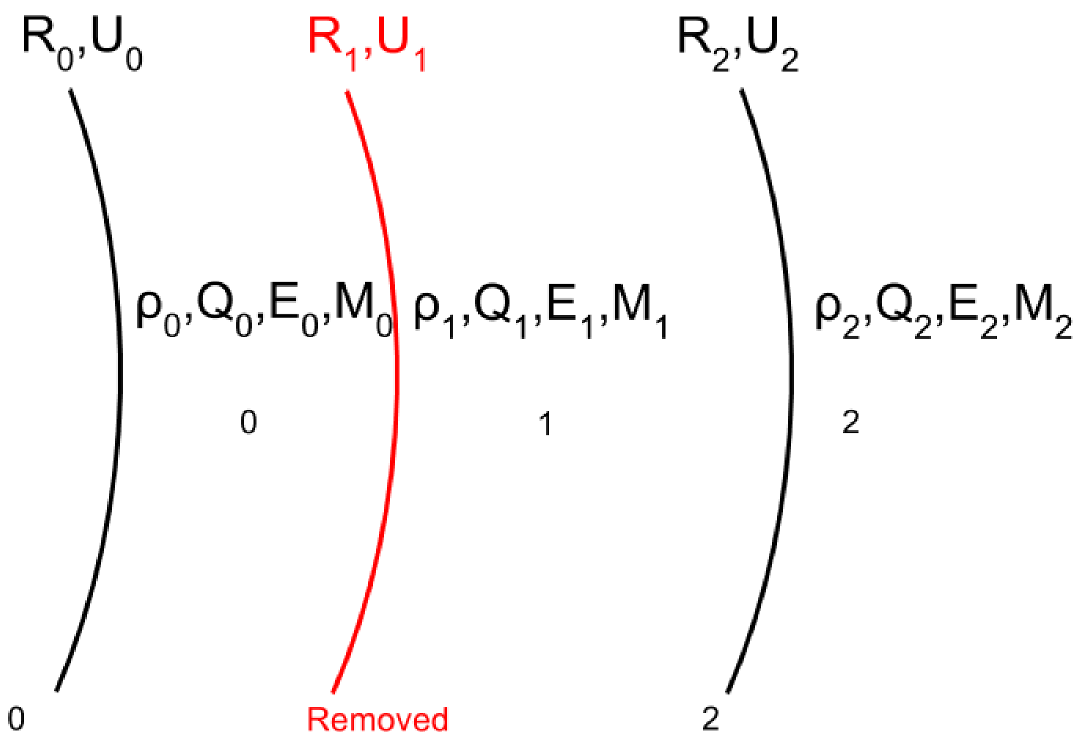

Figure 2.3: Zone Combination
(Interface 0 is the bottom interface)

As shown in figure 2.3, interface 1 is removed. The plasma in zone 0 and zone 1 is now in the same zone. Interfaces 0 and 2 are not moved, but they do receive some change in energy and momentum. Once the plasma state variables have been updated, all zones and interfaces are renumbered appropriately.

Chapter 3

Testing The Code

To ensure the code produces trustworthy results we must test each of the crucial functions against theoretical predictions. Three tests were chosen to verify the code's validity, a rest case test, a test to verify the properties of any shock waves, and a test to verify the effects of Bremsstrahlung cooling. Further testing to verify the accretion and zone combination subroutines is done during analysis of the full run.

3.1 Blast Wave Test

As the plasma falls inward it will encounter the magnetopause and start to build up and compress. Eventually the system will form a shock wave that propagates outward. The goal of the blast wave test is to determine the value of the viscosity constant, Q_0, which gives us shock jump conditions that match theoretical predictions. The initialization parameters of the blast wave test are outlined in Table 3.1

Parameter	**Value**	**Note**
$\dot{M}$	0	No Accretion
G	0	No Gravity
E_0	0	No Cooling
U_i^0	0	No Initial Velocity
Blast Zone Temp	$10^9 K$	-
Atmosphere Temp	$10^1 K$	-
ρ_i^0	$10^5 g/cm^3$	Constant Density
Q_0	$0.5 - 2.0$	To be Varied

Table 3.1: Blast Wave Test Initialization Parameters

We measure the properties of the shock in terms of the shock jump conditions, the change in state variables across the shock wave. The theoretical predictions for these conditions are derived in section 2.4 of Dr. Susan Lea's *Fluids In Astrophysics* notes. In particular we wish to verify equation (47), the jump condition for density, in these notes. All of the jump conditions are related and verifying any one of them would be sufficient. I've shown here the results for the density jump (All the conditions should still be verified). The theoretical density jump condition is,

$$\tilde{\rho} = \frac{\gamma + 1}{(\gamma - 1) + \frac{2}{M^2}} \tag{3.1}$$

Where M is the Mach number of the shock, gamma is the adiabatic index, and $\tilde{\rho}$ is the density jump. The Blast Wave Test uses a very large value of M so equation 3.1 reduces to,

$$\tilde{\rho} = 4.0 \tag{3.2}$$

As per table 3.1 the Blast Wave Test is run with gravity, cooling, and accretion turned off. We must therefore artificially generate the shock. This is done by initializing the system with a few of the lowest zones at a very large temperature relative to the rest of the atmosphere.

The results in Figure 3.1 were generated with $Q_0 = 1.0$. With a higher value of Q_0 the density jump is weaker than needed and a lower value of Q_0 results in a density jump that is too large. The time scale of this run is very short because a large mach number results in a shock that moves very quickly to the top of the atmosphere. The data indicates that with time the density jump quickly approaches the predicted value and we can conclude the Blast Wave Test is successful.

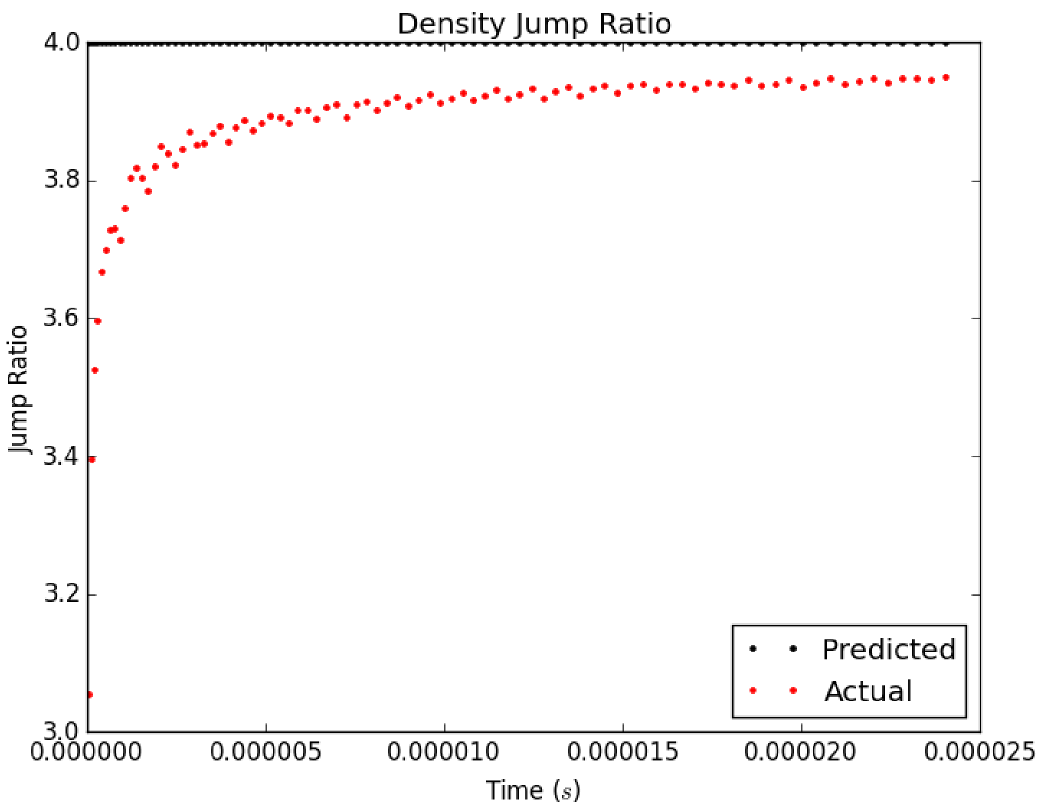

Figure 3.1: Blast Wave Test: Density Jump over Time

3.2 Rest Test

The rest case test will verify that the plasma environment obeys Newton's first law. The parameters for this test are listed in Table 3.2. The atmosphere is initialized with the plasma at a uniform temperature and density close to what we expect in the full run. With gravity and cooling turned off there is nothing to give the plasma any

Parameter	**Value**	**Note**
$\dot{M}$	0	No Accretion
G	0	No Gravity
E_0	0	No Cooling
U_i^0	0	No Initial Velocity
T_i^0	$10^5 K$	Constant Temperature
ρ_i^0	$10^{-14} g/cm^3$	Constant Density
Q_0	1.0	Standard Viscosity

Table 3.2: Rest Test Initialization Parameters

substantial velocity. The system is expected to approximately remain at the initial temperature, density, and velocity.

Figures 3.2 and 3.3 clearly illustrate the lack of change in the temperature and density of the plasma over time. Figure 3.4 shows some change in velocity but we must keep in mind the scale of these changes. In a full run the typical velocities we expect are close to free fall velocities, for $R \sim R_{AccretionRadius}$ the free fall velocity is approximately 5×10^7 cm/s. It is also crucial to keep in mind that the full run is expected to run for around 600 seconds whereas this test was run for 6000 seconds. Although velocity fluctuations begin to occur around 100 seconds, these oscillations are 16 orders of magnitude weaker than the typical velocities we expect to see in a full run. We can conclude that these velocity fluctuations are negligible and thus the Rest Test is successful.

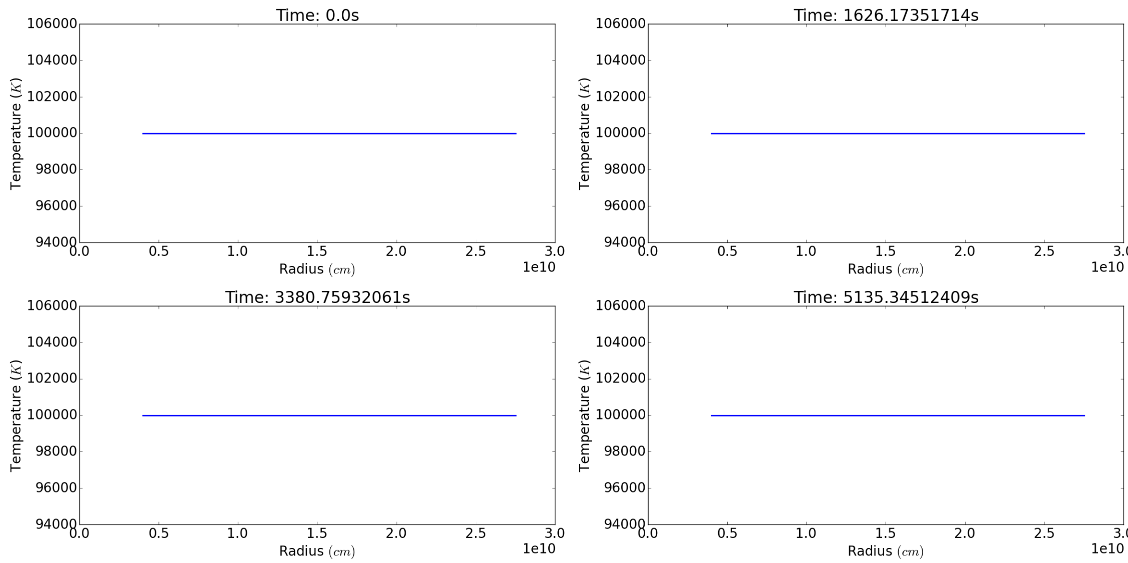

Figure 3.2: Rest Test: Atmospheric Temperature over Time

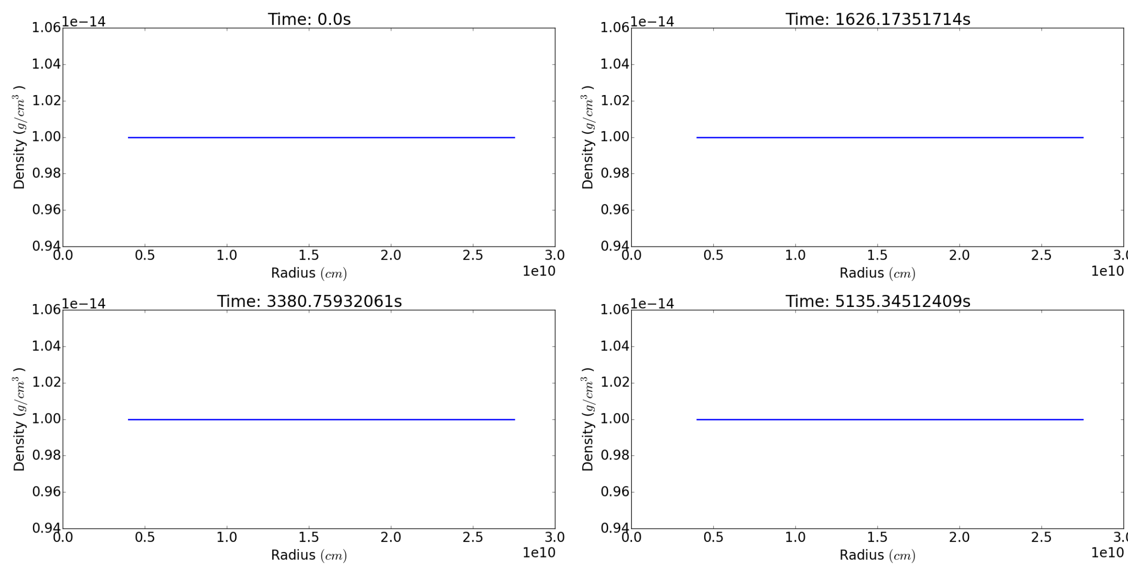

Figure 3.3: Rest Test: Atmospheric Density over Time

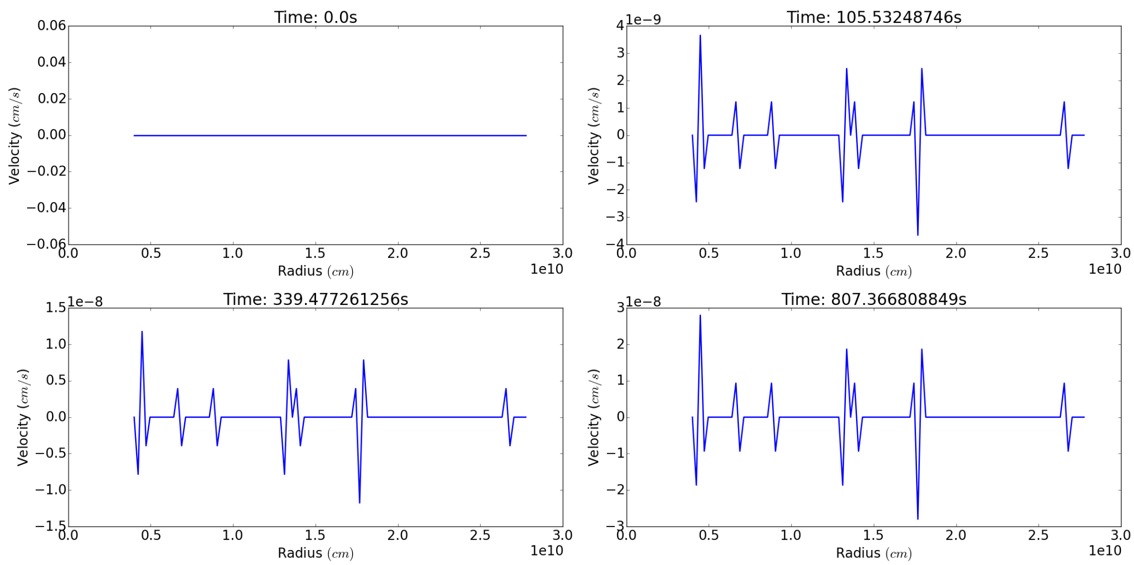

Figure 3.4: Rest Test: Plasma Velocity over Time
(Notice the times in this figure are different from those in earlier figures.)

3.3 Cooling Test

Once we have verified that the system forms the correct shock wave and is also stable without perturbations we must verify the cooling properties of the system. The system is cooled by Bremsstrahlung radiation, the derivation and properties of this cooling is outlined in Dr. Susan Lea's *Bremsstrahlung* notes. We wish to verify that our system cools at a rate,

$$P = 1.4 \times 10^{-27} \eta^2 \sqrt{T} \frac{\mathrm{erg}}{\mathrm{cm^3 s}} \tag{3.3}$$

Where P is the energy dissipated by cooling per unit time, η is ion density, and T is the temperature. The initialization parameters of the cooling test are outlined in Table 3.3.

Parameter	**Value**	**Note**
$\dot{M}$	0	No Accretion
G	0	No Gravity
E_0	1.4×10^{-27}	Standard Cooling
U_i^0	0	No Initial Velocity
T_i^0	$10^8 K$	Constant Temperature
ρ_i^0	$10^{-10} g/cm^3$	Constant Density
Q_0	1.0	Standard Viscosity

Table 3.3: Cooling Test Initialization Parameters

With constant density we can derive the temperature at any given time,

$$\frac{\mathrm{d}T}{\mathrm{d}t} = -CT^{1/2}$$
$$T(t) = \frac{1}{4}(-Ct + B)^2$$

Let T_0 be the initial temperature and the cooling time, τ,be defined as,

$$\tau = \frac{T_0}{\frac{\mathrm{d}T}{\mathrm{d}t}\big|_{T=T_0}}$$

We find our temperature as function of time and initial temperature to be,

$$T(t) = T_0\Big(1 - \frac{t}{2\tau}\Big)^2 \tag{3.4}$$

Figure 3.5 shows the results of the cooling test. The actual temperature curve very closely matches the expected cooling curve given by equation 3.4 and with this we can conclude that the Cooling Test is successful.

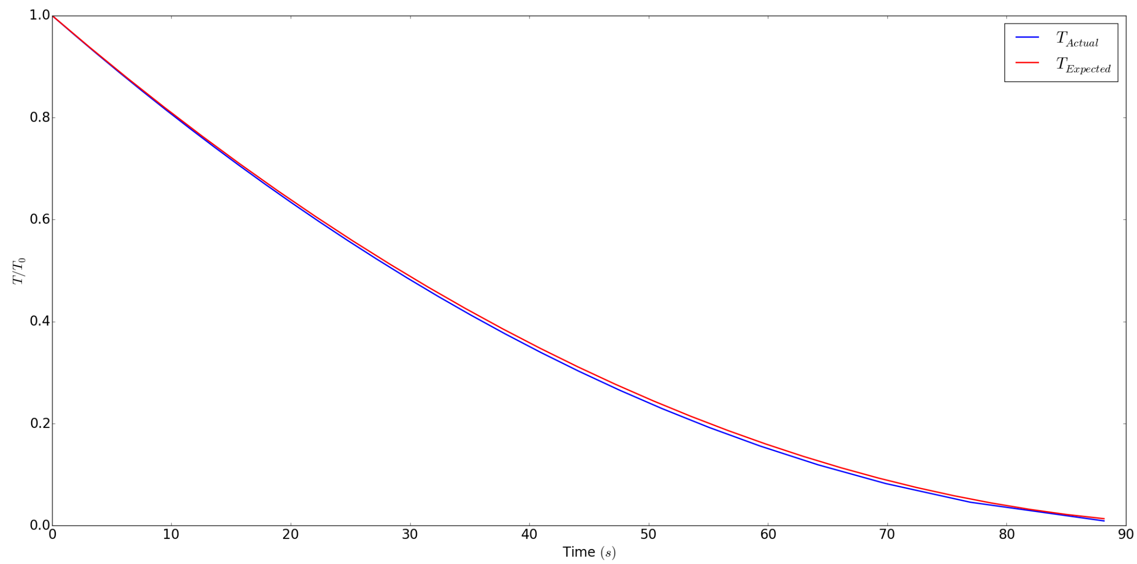

Figure 3.5: Cooling Test: Atmospheric Temperature over Time

3.4 The Full Run

The full run of the test is what the code was written to do. With this run we hope to study the behavior of the shock wave, specifically whether there is an episodic nature to its motion. This run makes use of all the physical processes involved; gravity, cooling, mass infall, accretion, and zone combination. The initialization parameters of the full run are outlined in Table 3.4.

The system is initialized with equal mass in each zone to simulate a steady accretion rate. The velocity of each zone is determined by standard free fall velocity based on radial position. The density is initialized using equation 2.21 with free fall velocity and the position of the interface bounding the zone from below. The results

Parameter	**Value**	**Note**
$\dot{M}$	10^{13} g/s	Standard Accretion
G	6.67×10^{-8}	Standard Gravity
E_0	1.4×10^{-27}	Standard Cooling
U_i^0	$-\sqrt{\frac{GM_{ns}}{R_i}}$	Free Fall Velocity
T_i^0	10^4 K	Even Temperature Distribution
ρ_i^0	$\frac{\dot{M}}{4\pi R_i^2 U_i}$	Free Fall Density
Q_0	1.0	Standard Viscosity
M_i	10^{13} g	Constant Mass Per Zone

Table 3.4: Full Run Initialization Parameters

of the full run are analyzed in Chapter 4.

Chapter 4

Analysis of Results

We expected the full run to illustrate some episodic motion of the shock that the system generates. Unfortunately, a critical bug in the code prevented the project from getting this far. This source of this bug was narrowed to the accretion subroutine. Figure 4.1 illustrates the erratic density that the accretion process generates.

The density plot follows the expected behavior below 1.75×10^{10} cm, a shock wave forms and moves outward. The shocks and spikes beyond 1.75×10^{10} cm are a cause for concern. There is no reason for the density to be so erratic near the top of the atmosphere, this is where the effects of the shock wave are least important. Beyond the expected shock wave (which is around 0.75×10^{10} cm), the density plot should be quite smooth all the way to the top of the atmosphere.

The theoretical work absolutely does not predict any such behavior. I feel confident saying that there is no physical reason behind this. This error is purely computational and is due to an improper method used to simulate the accretion process.

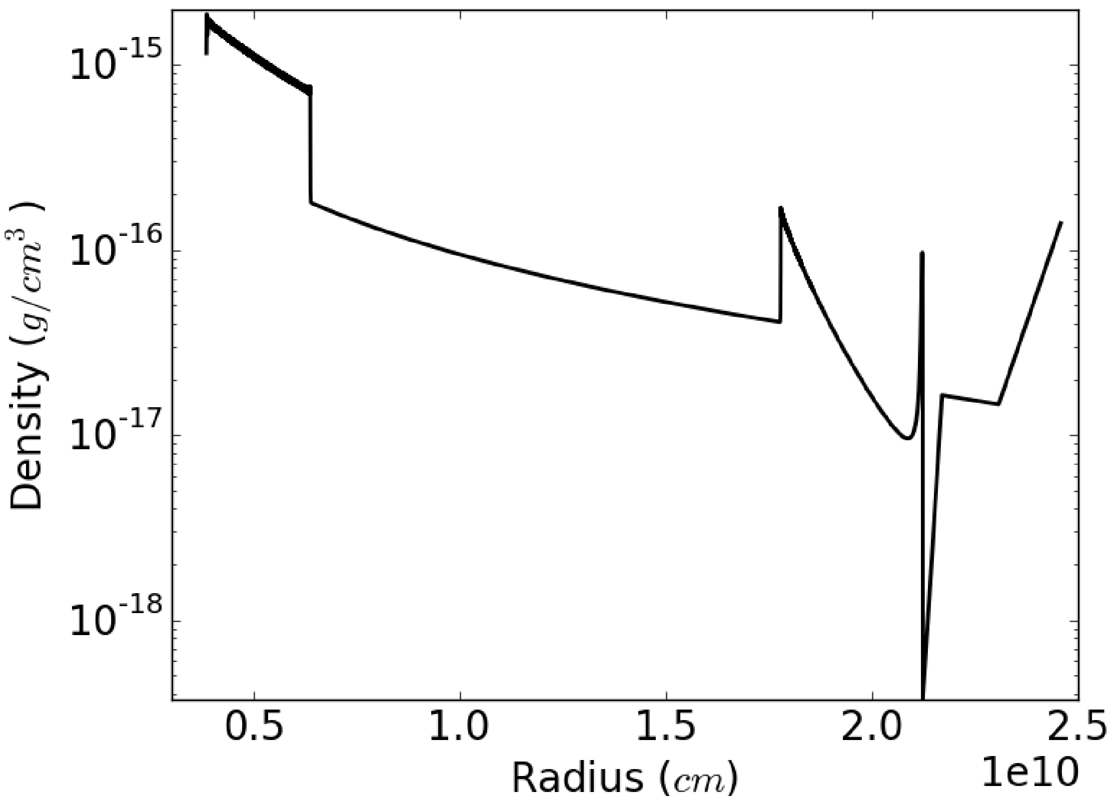

Figure 4.1: Full Run: Erratic Densities

This problem could have been avoided, or at least detected much earlier, if I had run a proper accretion test. The lack of a test that would verify the validity of the accretion routine is ultimately what led to this issue.

Conclusion

The goal of this project was to write a code that was a capable of accurately evolving the plasma atmosphere around a magnetized neutron star and learn more about the behavior of the shock wave that this system generates. Though this goal was not met, the project is not without value. A robust, modern, and efficient Lagrangian fluid code was written to study various properties of a magneto-hydrodynamic system. The code's capacity to properly generate and maintain shock waves (Section 3.1, Blast Wave Test), minimize round-off errors and instability growth, (Section 3.2, Rest Test), and accurately apply the effects of Bremsstrahlung Cooling (Section 3.3, Cooling Test) were all validated.

For future work I would do more rigorous testing of the accretion process. Accurate application of the effects of accretion bring this code much closer to being able to solve the initial problem. I urge my successors to study my comments and read my notes in the hopes of finding my mistakes. Run the code as much as possible and analyze every aspect of the data. As Dr. Lea always said, "When you run into something you can't figure out, get more data!".

Bibliography

Arons, J. and Lea, S. M. (1976a). Accretion onto magnetized neutron stars: normal mode analysis of the interchange instability at the magnetopause. *Astrophysical Journal*, 210:792–804.

Arons, J. and Lea, S. M. (1976b). Accretion onto magnetized neutron stars: structure and interchange instability of a model magnetosphere. *Astrophysical Journal*, 207:914–936.

Bondi, H. (1952). Studies of spherically symmetric accretion. *Monthly Notices of the Royal Astronomical Society*, 112:195–201.

Hunt, R. (1971). A fluid dynamical study of the accretion process. *Monthly Notices of the Royal Astronomical Society*, 154:141–165.

Linford, G. A. (1985). A study of hydrodynamic events in accretion phenomena. Master's thesis, San Francisco State University.

Stratton, D. M. (1989). Accretion onto magnetized neutron stars: Numerical simulation of plasma flow across the magnetopause. Master's thesis, San Francisco State University.

Appendices

Appendix A

Source Code

```
# -*- coding: utf-8 -*-
import os
import csv
from sys import exit
import numpy as np
from matplotlib import pyplot as plt
from matplotlib import animation as ani
plt.rcParams.update({'font.size':20})
plt.rcParams['animation.ffmpeg_path'] = 'C:\\Users\\msing_000\\Documents\\Grad\\Research\\Python_Code\\ffmpeg\\bin\\ffmpeg'
np.set_printoptions(threshold='nan')
np.set_printoptions(linewidth='nan')

Na = 6.022141E23        #Avogadro's Number
K = 1.38065E-16         #Boltzmann Constant
gamma = 5./3.           #Adiabatic Constant
pi = np.pi              #Define pi
Q0 = 1.0                #V.N.R. Artificial Viscosity Constant
G = 6.67E-8             #Universal Gravitational Constant
E0 = 1.4E-27            #Bremsstrahlung Cooling Source Constant, expected 1.E-27
Mns = 1.E33             #Neutron Star Mass
Rns = 1.E6              #Radius of the neutron star
```

```
Mdot = 1.E13            #Accretion Rate, originally 1E14
mu = 1.E30              #Magnetic Moment at surface of Neutron
    Star
x = 1.0                 #Luminosity Constant, xE34 ergs/s is the
    Luminosity. Between 0.1 and 100
AccRad = 2.5E10         #Accretion Radius
RAtemp = 1.E4           #Initial temperature of plasma at
    accretion radius
ztop = 299              #Subscript of top zone in atmosphere
zbot = 0                #Subscript of bottom zone in atmosphere
zones = [i for i in range(zbot, ztop)]

radius = np.zeros([ztop+1,3])
plasmaVelocity = np.zeros([ztop+1,3])
specificVolume = np.zeros([ztop,3])
internalEnergy = np.zeros([ztop,3])
pressure = np.zeros([ztop,3])
density = np.zeros([ztop,3])
temperature = np.zeros([ztop,3])
artificialViscosity = np.zeros([ztop,3])
interfaceInertia = np.zeros(ztop+1)
mass = np.zeros(ztop)

dtp2 = .01  #dtp2 is timestep
dtm2 = .01  #dtm2 is previous timestep

spos = 0    #Position of the shock, initialize at 0 in runner
accel = 0.0 #Stores the acceleration of the magnetopause

nGate = 0   #The number of loops the gate is open
gateOpen = 0
stepE = 0.0 #Luminosity in a given time step

Eflux = np.zeros(ztop+1)
Eflow = np.zeros(ztop+1)
```

```
TA = []
width = 1.*(AccRad-1.E9)/(ztop+1) #Initial zone width.
atmosphereTop = AccRad

runType = 0 #0: Full run, 1: plasma at rest test, 2: Blast wave test. Assigned in setup()
zoneAppended = 0 #Flag signifying whether a new zone was appended this loop

uMag = 0.0 #Velocity of the magnetopause
massAdded = 0.0
massDropped = 0.0
#We consider the magnetopause and the bottom interface of the plasma to be two separate objects
#These objects are generally at the same location except when the gate is open.

##############################
# Status #
# Time-steps need to be very small, don't let a zone move farther than 1/200th of
# the distance to the next zone. Less than that doesn't change much.
##############################

def dynamicAtmosphere(U, R, V, E, P, Q, T, rho, DM, interfaceInertia, spos, nGate, Eflux, Eflow, loop):
    global stepE
    global gateOpen
    global uMag
    global massDropped
    #Accrete mass
    if(runType == 0):
        U,R,P,V,T,rho,Q,E,DM,interfaceInertia = AccreteMass(U,R,P,V,T,rho,Q,E,DM,interfaceInertia)
```

```

    for i in range(zbot+1, ztop): #Excludes first and last zones
        Area = 4.*pi*R[i,0]**2
        dVelocity = Area*(-(P[i,0]+Q[i,0])+(P[i-1,0]+Q[i-1,0]))*dtp2/interfaceInertia[i]-(G*Mns*dtp2/R[i,0]**2)
        if (runType == 3): #If Cooling Test
            dVelocity = 0
        U[i,2] = U[i,0] + dVelocity

    if (runType != 2 and runType != 5): #If not Blast Wave and not Accretion Test
        #bPressure = ((2.75*mu/R[zbot,0]**3)**2)/(8.*pi) #Magnetic Pressure
        bPressure = ((mu/R[zbot,0]**3)**2)/(8.*pi) #Magnetic Pressure
        Area = 4.*pi*R[zbot,0]**2
        bcrMag = Area*(-(P[zbot,0]+Q[zbot,0])+bPressure)*dtp2/interfaceInertia[zbot]
        uMag = uMag + bcrMag
        U[zbot,2] = uMag
    elif(runType == 5):
        Area = 4.*pi*R[zbot,0]**2
        U[zbot,2] = U[zbot,0] - Area*(P[zbot,0]+Q[zbot,0])*dtp2/interfaceInertia[zbot]
        U[zbot,2] = U[zbot,2] - (G*Mns*dtp2/R[zbot,0]**2)
    else:
        U[zbot,0] = 0

    if (runType != 0 and runType != 5): #If not full run or Accretion Test
        #U[ztop,2] = (4*pi*R[ztop,0]**2)*(P[ztop-1,0]+Q[ztop-1,0])*dtp2/interfaceInertia[ztop] #Open the top
        U[ztop,2] = 0.0
```

```
    else:
        #U[ztop,2] = -np.sqrt(G*Mns/R[ztop,0])
        U[ztop,2] = U[ztop,0] - (G*Mns*dtp2/R[zbot,0]**2)
    #Compute position from velocity
    for i in range(zbot, ztop+1): #Runs over all zones
        R[i,2] = R[i,0] + U[i,0]*dtp2
        if(i != 0 and R[i,2] <= R[i-1,2]): exit("Interface "+str(i-1)+" is radially farther than interface "+str(i))

    #Compute density from zone volume
    for i in range(zbot, ztop):
        rho[i,2] = DM[i]/((4*pi/3.)*(R[i+1,2]**3-R[i,2]**3))

#####Check for stability at magnetopause#####

    if (runType == 0): #If full run
        #Compute max velocity of B-field in the plasma
        temporaryVar = np.sqrt(2.0*G*Mns/R[zbot,0])
        temporaryVar2 = 1.0 - P[zbot,0]/(rho[zbot,0]*temporaryVar**2)
        if (temporaryVar2 > 0.0):
            vBmax = 0.5*temporaryVar*np.sqrt(temporaryVar2*(R[spos,0] - R[zbot,0])/R[zbot,0])
        else: vBmax = 0.0
        if (vBmax > np.sqrt(gamma*P[zbot,0]*V[zbot,0])): vBmax = np.sqrt(gamma*P[zbot,0]*V[zbot,0])

    maxRho = (1.3445*mu**2)/(4.*pi*G*Mns*R[zbot,2]**5)
    #maxRho = 0.0
    #Check if the gate is open. If the gate is open, plasma can leak through the magnetopause
    if(rho[zbot,2] > maxRho and runType == 0):
        #Open the gate
        Area = 4.*pi*R[zbot,0]**2
```

```
    dUplasma = -Area*(P[zbot,0]+Q[zbot,0])*dtp2/interfaceInertia[zbot]-G*Mns*dtp2/R[zbot,0]**2
    if (not gateOpen): #Gate was not open in the last timestep
        firstInstanceGate = 1 #This is the first timestep with the gate open
        gateOpen = 1
    else: firstInstanceGate = 0 #Gate was open last timestep
    uPlasma = U[zbot,0] + dUplasma

    rPlasma = R[zbot,0] + uPlasma*dtp2

    if (R[zbot,2] < rPlasma):
        rPlasma = R[zbot,2]

    #Determining the amount of plasma that fell through the magnetopause
    dRatio = 1.0 #Ratio of density below magnetopause to above magnetopause
    dropVol = 4.0*pi/3.0 * (R[zbot,2]**3 - rPlasma**3) #Volume of plasma that falls through
    keepVol = 4.0*pi/3.0 * (R[zbot+1,2]**3 - R[zbot,2]**3) #Volume of plasma that remains
    if (dropVol > 0.0):
        rho[zbot,2] = DM[zbot]/(keepVol + dRatio*dropVol)
        #Drop plasma below R[zbot,2]
        massBelow = rho[zbot,2]*dRatio*dropVol #Mass of plasma that fell through
        if (massBelow < 0.0): exit("Mass below is negative for loop #",loop)
    else: massBelow = 0.0

    if (massBelow > 0.0):
        massDropped = massDropped + massBelow
```

```
            nGate = nGate+1
            stepE = stepE + (G*Mns*massBelow/Rns)/dtp2
            if(massBelow < DM[zbot]): #If there is enough mass in the bottom zone.
                DM[zbot] = DM[zbot]-massBelow
                rho[zbot,2] = DM[zbot]/((4*pi/3.)*(R[zbot+1,2]**3-R[zbot,2]**3))

                #The bottom zone has fallen below the magnetopause
                #Some of the plasma will now leave the bottom zone
                #That plasma will carry away some momentum and energy
                #We need to solve for the new momemntum and energy of the system
                #We are going from R[zbot+1,0] and rPlasma as our lowest zones to
                #rPlasma, R[zbot,0], and R[zbot+1,0] as our lowest zones
                tempMass = np.array([massBelow,DM[zbot],DM[zbot+1]])
                tempVel = np.zeros(2)
                tempInertia = np.zeros(3)
                Inertia(tempMass,tempInertia,0,1,len(tempMass),ztop)
                momentum = interfaceInertia[zbot]*uPlasma + (interfaceInertia[zbot+1]-tempInertia[2])*U[zbot+1,0]
                kineticEnergy = interfaceInertia[zbot]*uPlasma**2 + (interfaceInertia[zbot+1]-tempInertia[2])*U[zbot+1,0]**2
                Conserve(tempInertia[0],tempVel[0],tempInertia[1],tempVel[1],momentum,kineticEnergy,uPlasma)
```

```
                #tempVel[0] should be close to uPlasma
                Inertia(DM, interfaceInertia, zbot, zbot, zbot+3, ztop)
                if(uMag-tempVel[1] > np.sqrt(gamma*P[zbot,0]*V[zbot,0])):
                    uMag = np.sqrt(gamma*P[zbot,0]*V[zbot,0])+tempVel[1]#Interface cannot move faster than the speed limit.
                U[zbot,0] = tempVel[1]
            else: exit("Not enough mass in bottom zone")
        #end if for massBelow>0
    elif(gateOpen):
        #The gate is now closed but was open last time step
        lastInstanceGate = 1 #The gate was open last time step
        U[zbot,2] = U[zbot,0] + bcrMag
        R[zbot,2] = R[zbot,0] + U[zbot,0]*dtp2
        rho[zbot,2] = DM[zbot]/((4*pi/3.0) * (R[zbot+1,2]**3-R[zbot,2]**3))
        gateOpen = 0
    else: lastInstanceGate = 0
##############
    for i in range(zbot, ztop):
        V[i,2] = 1.0/rho[i,2]
        E[i,2] = E[i,0] - (Q[i,0]+P[i,0])*(V[i,2]-V[i,0])
        if(E[i,0] < (Q[i,0]+P[i,0])*(V[i,2]-V[i,0])): exit("Energy going negative in zone " + str(i))
        E[i,2] = E[i,2] - E0*np.sqrt(T[i,0])*(rho[i,0]*Na)**2 * dtp2*V[i,0] #Cooling Term
            #Recall E is specific energy, energy per unit mass
            #Cooling term needs the density to be in particles/cm^3 (assume hydrogen)
            #Taking molar mass of hydrogen to be 1.0 g/mol
        if (runType == 3): #If cooling test
```

```
            E[i,2] = E[i,0] - E0*np.sqrt(T[i,0])*(rho[i,0]*Na)**2 * dtp2*V[i,0] #Cooling Term
        T[i,2] = E[i,2]/(3.*Na*K)
        P[i,2] = (gamma-1)*E[i,2]*rho[i,2]
        if (T[i,2] <= 0): exit("T is nonsense")
        if (V[i,2] < V[i,0] and U[i,2]-U[i+1,2] > 0): Q[i,2] = Q0*rho[i,2]*(U[i,2]-U[i+1,2])**2
        else: Q[i,2] = 0.0 #Only non-zero when there is compression.
        if (Q[i,2] < 0): Q[i,2] = 0.0
        stabilityParam = Q0*(U[i,2]-U[i+1,2])*(dtp2/(R[i+1,2]-R[i,2]))
        if (stabilityParam > 0.5): exit("Diffusion Instability: " + str(stabilityParam))

    #Update Time Subscripts
    for i in range(zbot, ztop):
        #Current becomes old
        rho[i,1] = rho[i,0]
        R[i,1] = R[i,0]
        P[i,1] = P[i,0]
        V[i,1] = V[i,0]
        E[i,1] = E[i,0]
        T[i,1] = T[i,0]
        U[i,1] = U[i,0]
        Q[i,1] = Q[i,0]
        #New becomes current
        R[i,0] = R[i,2]
        rho[i,0] = rho[i,2]
        P[i,0] = P[i,2]
        V[i,0] = V[i,2]
        E[i,0] = E[i,2]
        T[i,0] = T[i,2]
        U[i,0] = U[i,2]
        Q[i,0] = Q[i,2]
```

```
    R[ztop,1] = R[ztop,0]
    R[ztop,0] = R[ztop,2]
    U[ztop,1] = U[ztop,0]
    U[ztop,0] = U[ztop,2]

    return U,R,P,V,T,rho,Q,E,DM,interfaceInertia
  #End Dynamic Atmosphere

def initTestCase4(U,R,P,V,T,rho,Q,E,DM,DM2):
    #Rest Test
    global G
    global E0
    global width
    G = 0
    E0 = 0
    width = 1.*(2.0E10-R[0,0])/(ztop+1)

    #Radius, Temperature
    for i in range(zbot+1, ztop+1):
        R[i,0] = R[i-1,0]+width
    #Density and Specific Volume
    rho[0,0] = 1.E-14
    V[0,0] = 1./rho[0,0]
    for i in range(zbot+1, ztop):
        rho[i,0] = rho[0,0]
        V[i,0] = V[0,0]
    #Viscosity, Energy, Pressure, Mass, Temperature, Velocity
    for i in range(zbot, ztop):
        T[i] = 1.E5
        Q[i] = 0.0
        E[i,0] = 3.*Na*K*T[i,0]
        P[i,0] = (gamma-1)*(E[i,0]*rho[i,0])
        DM[i] = rho[0,0]*((4*pi/3.)*(R[i+1,0]**3-R[i,0]**3))
        U[i,0] = 0
    #Inertia
```

```
    Inertia(DM, DM2, zbot, zbot, ztop, ztop)

def initTestCase3(U,R,P,V,T,rho,Q,E,DM,DM2):
    #CoolingTest
    global G, width
    G = 0
    width = 1.*(2.0E10-R[0,0])/(ztop+1)
    #Radius, Temperature
    for i in range(zbot+1, ztop+1):
        R[i,0] = R[i-1,0]+width
    #Density and Specific Volume
    rho[0,0] = 1.E-10
    V[0,0] = 1./rho[0,0]
    for i in range(zbot+1, ztop):
        rho[i,0] = rho[0,0]
        V[i,0] = V[0,0]
    #Viscosity, Energy, Pressure, Mass, Temperature, Velocity
    for i in range(zbot, ztop):
        T[i] = 1.E8
        Q[i] = 0.0
        E[i,0] = 3.*Na*K*T[i,0]
        P[i,0] = (gamma-1)*(E[i,0]*rho[i,0])
        DM[i] = rho[0,0]*((4*pi/3.)*(R[i+1,0]**3-R[i,0]**3))
        U[i,0] = 0
    #Inertia
    Inertia(DM, DM2, zbot, zbot, ztop, ztop)

def initTestCase2(U,R,P,V,T,rho,Q,E,DM,DM2, cooling = 0):
    #Blast Wave Test
    global G
    global E0
    G = 0
    if (cooling == 0): E0 = 0

    #Radius, Temperature
```

```
    for i in range(zbot+1, ztop+1):
        R[i,0] = R[i-1,0]+width
    #Density and Specific Volume
    rho[0,0] = 1.E3
    V[0,0] = 1./rho[0,0]
    for i in range(zbot+1, ztop):
        rho[i,0] = rho[0,0]
        V[i,0] = V[0,0]
    #Viscosity, Energy, Pressure, Mass, Temperature, Velocity
    for i in range(zbot, ztop):
        T[i] = 1.E1
        Q[i] = 0.0
        E[i,0] = 3.*Na*K*T[i,0]
        P[i,0] = (gamma-1)*(E[i,0]*rho[i,0])
        DM[i] = rho[0,0]*((4*pi/3.)*(R[i+1,0]**3-R[i,0]**3))
        U[i,0] = 0
    T[0:2,0] = 1.E9
    E[0:2,0] = 3.*Na*K*T[0,0]
    P[0:2,0] = (gamma-1)*(E[0,0]*rho[0,0])
    #Inertia
    Inertia(DM, DM2, zbot, zbot, ztop, ztop)

def initTestCase1(U,R,P,V,T,rho,Q,E,DM,DM2, cooling = 1):
    global E0
    global atmosphereTop
    global width
    if (cooling == 0): E0 = 0
    width = 1.*(2.0E10-R[0,0])/(ztop)

    #Radius, Temperature
    for i in range(zbot+1, ztop+1):
        R[i,0] = R[i-1,0]+width
    atmosphereTop = R[ztop,0]
    #Temperature, Velocity
    for i in range(zbot, ztop):
```

```
        T[i,0] = RAtemp
        U[i,0] = -np.sqrt(G*Mns/R[i,0])
    U[ztop,0] = -np.sqrt(G*Mns/R[ztop,0])
    #Density, Specific Volume, Viscosity, Energy, Pressure, Mass
    for i in range(zbot, ztop):
        rho[i,0] = Mdot/(4.*pi*.25*(R[i,0]+R[i+1,0])**2 * .5*abs(U[i,0])+U[i+1,0])#Needs Radii and Vels
        V[i,0] = 1.0/rho[i,0] #Needs Densities
        Q[i] = 0.0
        E[i,0] = 3.*Na*K*T[i,0]
        P[i,0] = (gamma-1)*(E[i,0]*rho[i,0])
        DM[i] = rho[i,0]*((4*pi/3.)*(R[i+1,0]**3-R[i,0]**3))
    #Inertia
    Inertia(DM, DM2, zbot, zbot, ztop, ztop)

def initTestCase0(U,R,P,V,T,rho,Q,E,DM,DM2):
    #Constant Mass, free fall density
    global atmosphereTop, width #Width defined as space between ztop and ztop-1
    initMass = 1.E13
    #Radii
    A = np.sqrt(G*Mns)*3.*initMass/Mdot
    #R[0,0] is set by setup routine.
    for i in range(zbot,ztop):
        R[i+1,0] = (A*R[i,0]**(1.5) + R[i,0]**3)**(1./3)
        if (R[i+1,0] > AccRad): exit("Too many zones after zone #" + str(i+1))
        U[i+1,0] = -np.sqrt(G*Mns/R[i+1,0])
    U[zbot,0] = -np.sqrt(G*Mns/R[zbot,0])
    width = radius[ztop,0] - radius[ztop-1,0]
    atmosphereTop = radius[ztop,0]
    for i in range(zbot, ztop):
        T[i,0] = RAtemp
        rho[i,0] = Mdot/(4.*pi*R[i,0]**2 * abs(U[i,0]))
```

```
            V[i,0] = 1.0/rho[i,0] #Needs Densities
            Q[i] = 0.0
            E[i,0] = 3.*Na*K*T[i,0]
            P[i,0] = (gamma-1)*(E[i,0]*rho[i,0])
            DM[i] = initMass
    #Inertia
    Inertia(DM, DM2, zbot, zbot, ztop, ztop)

def initTestCase5(U,R,P,V,T,rho,Q,E,DM,DM2):
    #Accretion Test, Constant initial Mass, freefall Density
    global atmosphereTop, width #Width defined as space between ztop and ztop-1
    global mu, E0
    mu = 0
    E0 = 0

    initMass = 1.E13
    #Radii
    A = np.sqrt(G*Mns)*3.*initMass/Mdot
    #R[0,0] is set by setup routine.
    for i in range(zbot,ztop):
        R[i+1,0] = (A*R[i,0]**(1.5) + R[i,0]**3)**(1./3)
        U[i+1,0] = -np.sqrt(G*Mns/R[i+1,0])
    U[zbot,0] = -np.sqrt(G*Mns/R[zbot,0])
    width = radius[ztop,0] - radius[ztop-1,0]
    atmosphereTop = radius[ztop,0]
    for i in range(zbot, ztop):
        T[i,0] = RAtemp
        rho[i,0] = Mdot/(4.*pi*R[i,0]**2 * abs(U[i,0]))
        V[i,0] = 1.0/rho[i,0] #Needs Densities
        Q[i] = 0.0
        E[i,0] = 3.*Na*K*T[i,0]
        P[i,0] = (gamma-1)*(E[i,0]*rho[i,0])
        DM[i] = initMass
    #Inertia
```

```
    Inertia(DM, DM2, zbot, zbot, ztop, ztop)

def AccreteMass2(U,R,P,V,T,rho,Q,E,DM,DM2):
    global ztop
    global zoneAppended
    global massAdded
    if (atmosphereTop - R[ztop,0] > 1.0*width): #If there is space to fit a new zone
        newR = R[ztop,0] + 1.0*width
        R = np.append(R,np.ones([1,3])*newR,0)
        ztop = ztop+1
        UFreefall = G*Mns*dtp2/R[ztop,0]**2
        U = np.append(U,np.ones([1,3])*UFreefall,0)
        rhoFreefall = Mdot/(4.*pi*R[ztop,0]**2*U[ztop,0])
        rho = np.append(rho,np.ones([1,3])*rhoFreefall,0)
        newV = 1./rho[ztop-1,0]
        V = np.append(V,np.ones([1,3])*newV,0)
        Q = np.append(Q,np.zeros([1,3]),0)
        T = np.append(T,np.ones([1,3])*RAtemp,0)
        newE = 3.*Na*K*RAtemp
        E = np.append(E,np.ones([1,3])*newE,0)
        newP = (gamma-1)*(E[ztop-1,0]*rho[ztop-1,0])
        P = np.append(P,np.ones([1,3])*newP,0)
        newM = rho[ztop-1,0]*(4./3 *pi*(R[ztop,0]**3-R[ztop-1,0]**3))
        DM = np.append(DM,newM)
        DM2 = np.append(DM2,1.0) #Values corrected with Inertia routine.
        zoneAppended = 1
        massAdded = massAdded + newM
    else:
        zoneAppended = 0

    Inertia(DM, DM2, zbot, zbot, ztop, ztop) #Needed for velocity
```

```
    #Conserve energy and momentum, regardless of new zone or not.
    momentum = DM2[ztop]*U[ztop,0]+DM2[ztop-1]*U[ztop-1,0]
    KE = DM2[ztop]*U[ztop,0]**2+DM2[ztop-1]*U[ztop-1,0]**2
    Conserve(DM2[ztop-1],U[ztop-1,0],DM2[ztop],U[ztop,0],momentum,KE,U[ztop-1,0])

    if (abs(U[zbot-1,0]) > np.sqrt(gamma*P[ztop-2,0]*V[ztop-2,0])):
        U[ztop-1,0] = -np.sqrt(gamma*P[ztop-2,0]*V[ztop-2,0])

    if (abs(U[zbot,0]) > np.sqrt(gamma*P[ztop-1,0]*V[ztop-1,0])):
        U[ztop,0] = -np.sqrt(gamma*P[ztop-1,0]*V[ztop-1,0])

    return U,R,P,V,T,rho,Q,E,DM,DM2

    Inertia(DM, DM2, zbot, zbot, ztop, ztop) #Needed for velocity
    #Conserve energy and momentum, regardless of new zone or not.
    momentum = DM2[ztop]*U[ztop,0]+DM2[ztop-1]*U[ztop-1,0]
    KE = DM2[ztop]*U[ztop,0]**2+DM2[ztop-1]*U[ztop-1,0]**2
    Conserve(DM2[ztop-1],U[ztop-1,0],DM2[ztop],U[ztop,0],momentum,KE,U[ztop-1,0])

    if (abs(U[zbot-1,0]) > np.sqrt(gamma*P[ztop-2,0]*V[ztop-2,0])):
        U[ztop-1,0] = -np.sqrt(gamma*P[ztop-2,0]*V[ztop-2,0])

    if (abs(U[zbot,0]) > np.sqrt(gamma*P[ztop-1,0]*V[ztop-1,0])):
        U[ztop,0] = -np.sqrt(gamma*P[ztop-1,0]*V[ztop-1,0])

    return U,R,P,V,T,rho,Q,E,DM,DM2
```

```

def Inertia(DM,interfaceInertia,zbot,zmag,top,ztop):
    for i in range(zbot+1,top):
        interfaceInertia[i] = 0.5*(DM[i-1] + DM[i])
    if (zmag >= zbot): interfaceInertia[zbot] = DM[zbot]*.5
    else: interfaceInertia[zbot] = .5*(DM[zbot-1]+DM[zbot])
    if (top == ztop): interfaceInertia[top] = .5*(DM[ztop-1])

def Conserve(A, x, B, y, C, D, x0):
    #Finds a solution to Ax + By = C
    #and Ax^2 + By^2 = D
    #nearest to x0
    root = A*B*(D*(A+B)-C**2)
    if (root < 0): exit("root is negative in Conserve")
    root = np.sqrt(root)
    x = (A*C - root)/(A*(A+B))
    x1 = (A*C + root)/(A*(A+B))
    if (abs(x0-x1) < abs(x0-x)): x = x1
    y = (C-A*x)/B

def Tyme3(U,R,P,V,T,rho,Q,E,DM,interfaceInertia,loop, t1, t2):
    #Calculates timestep based on velocities of zone interfaces
    #If bottom zone is too small, combines bottom two zones.
    global ztop

    abu = np.zeros(ztop+1)
    zt = np.zeros(ztop+1)
    mintim = np.zeros(ztop)
    soundSpeed = np.zeros(ztop)

    for j in range(zbot, ztop):
        abu[j] = np.abs(U[j,0] - U[j+1,0])
        zt[j] = R[j+1,0] - R[j,0]
```

```
            soundSpeed[j] = np.sqrt(gamma*P[j,0]*V[j,0])
            mintim[j] = zt[j]/(abu[j]+soundSpeed[j] + np.sqrt(E[j,0]))

        magDt = (R[zbot+1,0]-R[zbot,0])/np.sqrt(gamma*P[0,0]*V[0,0])
        if (np.all(magDt<mintim[1:])):
        #if (np.all(mintim[0]<mintim[1:])):
            #Combine bottom zones
            oldMass = DM[zbot]
            oldInertia = interfaceInertia[zbot]
            DM[zbot] = DM[zbot+1]+oldMass
            Inertia(DM, interfaceInertia, zbot, zbot, zbot+3, ztop)

            DM[zbot+1] = 0.0
            rho[zbot+1,0] = DM[zbot]/((4.*pi/3)*(R[zbot+2,0]**3-R[zbot,0]**3))
            Q[zbot+1,0] = (Q[zbot,0]*V[zbot,0]+Q[zbot+1,0]*V[zbot+1,0])*rho[zbot+1,0]
            #Note V[zbot+1,0] has not been updated yet, we want the old value above
            #But rho has been updated, and we want the new value.
            V[zbot+1,0] = 1./rho[zbot+1,0]
            E[zbot+1,0] = (E[zbot,0]*DM[zbot]+E[zbot+1,0]*oldMass)/DM[zbot]
            T[zbot+1,0] = E[zbot+1,0]/(3.*Na*K)
            P[zbot+1,0] = (gamma-1)*E[zbot+1,0]*rho[zbot+1,0]

            #Delete empty zone and second interface
            R = np.delete(R,1,0)
            U = np.delete(U,1,0)
            V = np.delete(V,1,0)
            P = np.delete(P,1,0)
            T = np.delete(T,1,0)
```

```
            rho = np.delete(rho,1,0)
            E = np.delete(E,1,0)
            DM = np.delete(DM,1,0)
            interfaceInertia = np.delete(interfaceInertia,1,0)
            Q = np.delete(Q,1,0)
            ztop = ztop-1

            #Now that bottom zones have been combined, call Tyme again.
            U,R,P,V,T,rho,Q,E,DM,interfaceInertia,t1,t2 = Tyme3(U,R,P,V,T,rho,Q,E,DM,interfaceInertia,loop,t1,t2)
            return U,R,P,V,T,rho,Q,E,DM,interfaceInertia,t1, t2

        mintm = np.amin(mintim)
        if (loop <= 10):
            t1 = .001*mintm
            t2 = t1
        else:
            t1 = t2
            t2 = .05*mintm
            #We don't want the fastest interface moving more than this much of the distance to the next interface.

        return U,R,P,V,T,rho,Q,E,DM,interfaceInertia,t1, t2

def setup(case = 0,cooling = 1):
        global Mdot
        global runType
        if (case==2):
            print "Setup for Test Case 2 (Blast Wave Test)."
            Mdot = 0
            Rmatch = 0
            radius[0,0] = Rmatch
            runType = 2
            initTestCase2(plasmaVelocity,radius,pressure,
```

```
            specificVolume ,temperature ,density ,
            artificialViscosity ,internalEnergy ,mass ,
            interfaceInertia , cooling)

    elif(case==0):
        print "Setup for Full Run."
        radius[0,0] = 4.E9/x
        runType = 0
        #initTestCase1(plasmaVelocity ,radius ,pressure ,
            specificVolume ,temperature ,density ,
            artificialViscosity ,internalEnergy ,mass ,
            interfaceInertia , cooling)
        initTestCase0(plasmaVelocity ,radius ,pressure ,
            specificVolume ,temperature ,density ,
            artificialViscosity ,internalEnergy ,mass ,
            interfaceInertia)


    elif(case==3):
        print "Setup for Cooling Test (Test Case 3)"
        radius[0,0] = 4.E9/x
        Mdot = 0
        runType = 3
        initTestCase3(plasmaVelocity ,radius ,pressure ,
            specificVolume ,temperature ,density ,
            artificialViscosity ,internalEnergy ,mass ,
            interfaceInertia)

    elif(case==4):
        print "Setup for Rest Test"
        radius[0,0] = 4.E9/x
        Mdot = 0
        runType = 4
        initTestCase4(plasmaVelocity ,radius ,pressure ,
            specificVolume ,temperature ,density ,
```

```
            artificialViscosity,internalEnergy,mass,
            interfaceInertia)

    elif(case==5):
        print "Setup for Accretion Test"
        radius[0,0] = 10.E9/x
        runType = 5
        initTestCase5(plasmaVelocity,radius,pressure,
            specificVolume,temperature,density,
            artificialViscosity,internalEnergy,mass,
            interfaceInertia)

def runSystem(runName = "test", runTime = 0, maxLoops = 0,
    shockCheck = 1, shockStop = 1, outputFreq = 100, minTime =
    0):
    global dtp2, dtm2
    global spos
    global TA
    global stepE
    global plasmaVelocity,radius,pressure,specificVolume,
        temperature,density
    global artificialViscosity,internalEnergy,mass,
        interfaceInertia
    spos = 0
    loop = 0
    output = 0  #Flag indicating whether or not to write data
        to file
    TA = [0.0]

    if (runName != "test"):
        if (os.path.exists(runName)):
            exit("Folder of name " + runName + " already
                exists!")
        else: #Setup file and write initial conditions
            os.mkdir(runName)
```

```
            output = 1
            data = open(runName+"/output.txt", 'w')
            data.write(str(ztop) + "\n")
            data.write(printState(loop, TA[-1], spos, radius, plasmaVelocity, density, temperature, internalEnergy, pressure, artificialViscosity, mass, specificVolume))

    while TA[-1] < runTime:
        if (loop % outputFreq == 0 or TA[-1] <= minTime):
            print "\n"
            print "Loop =", loop
            print "Simulation Time =", TA[-1], "seconds"
            print "Length of Arrays =", ztop
            print "Lowest zone is at", radius[0,0]
        if ztop > 4999: exit("Arrays are getting too large")

        plasmaVelocity,radius,pressure,specificVolume,temperature,density,artificialViscosity,internalEnergy,mass,interfaceInertia,dtm2,dtp2 = Tyme3(plasmaVelocity,radius,pressure,specificVolume,temperature,density,artificialViscosity,internalEnergy,mass,interfaceInertia,loop, dtm2, dtp2)
        if np.isnan(dtp2): exit("dtp2 is nan")
        if (dtp2<=0): exit("dtp2 is nonsensible")
        if (loop % outputFreq == 0 or TA[-1] <= minTime):
            print "dtp2 is",dtp2
            maxRho = (1.3445*mu**2)/(4.*pi*G*Mns*radius[zbot,0]**5)
            print "Density ratio is", density[zbot,0]/maxRho

        if (gateOpen and (loop % outputFreq == 0 or TA[-1] <= minTime)): print "The gate is open."

```

```
        plasmaVelocity,radius,pressure,specificVolume,
            temperature,density,artificialViscosity,
            internalEnergy,mass,interfaceInertia =
            dynamicAtmosphere(plasmaVelocity,radius,
            specificVolume,internalEnergy,pressure,
            artificialViscosity,temperature,density,mass,
            interfaceInertia, spos, nGate, Eflux, Eflow,loop)

        if shockCheck:  #Find position of the shock wave
            size = int(ztop/10)
            kbot = spos - int(size/2)
            if (kbot < 0): kbot = 0
            if (kbot+size >= ztop): kbot = ztop-size
            Qsub = np.zeros(size)
            for i in range(0, len(Qsub)):
                Qsub[i] = artificialViscosity[i+kbot,0]
            spos = np.argmax(Qsub) + kbot

        if (loop % outputFreq == 0 or TA[-1] <= minTime):
            print "Shock is at zone",spos
            print "Luminosity is", stepE

        #Update time and write data
        TA.append(TA[-1] + dtp2)
        if (output and (loop % outputFreq == 0 or TA[-1] <=
            minTime)):
            data.write(printState(loop, TA[-1], spos, radius,
                plasmaVelocity, density, temperature,
                internalEnergy, pressure, artificialViscosity,
                mass, specificVolume))

        #Reset stepE
        stepE = 0.0

        #Increment loop
```

```python
        loop = loop + 1

        if (radius[spos,0] >= .9*atmosphereTop and shockStop):
            print "Shock is too close to top of atmosphere"
            break

        if (radius[zbot,0] <= Rns):
            print "Bottom zone hit surface of neutron star."
            break

    print "\nComplete"
    if (output):
        data.write(printState(loop, TA[-1], spos, radius, plasmaVelocity, density, temperature, internalEnergy, pressure, artificialViscosity, mass, specificVolume))
        data.close()

def printState(loop, time, spos, R, U, rho, T, E, P, Q, M, V):
    state = "*****" + "\n " #Flag to denote start of new timestep.
    state += str(loop) + ' ' + str(time) + ' ' + str(spos) + " \n "
    state += np.array_str(R[:,0],max_line_width = 10000000).strip("[]") + " \n "
    state += np.array_str(U[:,0],max_line_width = 10000000).strip("[]") + " \n "
    state += np.array_str(rho[:,0],max_line_width = 10000000).strip("[]") + " \n "
    state += np.array_str(T[:,0],max_line_width = 10000000).strip("[]") + " \n "
    state += np.array_str(E[:,0],max_line_width = 10000000).strip("[]") + " \n "
```

```
    state += np.array_str(P[:,0],max_line_width = 10000000).strip("[]") + " \n "
    state += np.array_str(Q[:,0],max_line_width = 10000000).strip("[]") + " \n "
    state += np.array_str(mass,max_line_width = 10000000).strip("[]") + " \n "
    state += np.array_str(V[:,0],max_line_width = 10000000).strip("[]") + " \n "
    state += str(stepE) + " \n "
    return state

def readFile(fileName = "output.txt"):
    #Returns ztop, total number of loops, 2 1D arrays and 9 2D arrays.
    #Rows are constant time, Cols are constant zone.
    R = []
    U = []
    rho = []
    T = []
    E = []
    P = []
    Q = []
    mass = []
    V = []
    loops = []
    times = []
    spos = []
    L = []
    ztop = 99
    counter = -1
    with open(fileName, 'r') as data:
        reader = csv.reader(data, delimiter = ' ')
        for line in reader:
            line = filter(None,line)
            if (len(line) == 0):
```

```
                print "EOF"
                break
            if (counter == -1):
                ztop = int(line[0])   #First line should
                    contain 1 element
                counter = 0
                continue
            elif (line[0] == "*****"):
                counter = 0
                continue
            else:
                if (counter == 0):
                    loops.append(float(line[0]))
                    times.append(float(line[1]))
                    spos.append(float(line[2]))
                    counter += 1
                    continue
                elif (counter == 1):
                    R.append(np.array(map(float,line)))
                    counter += 1
                    continue
                elif (counter == 2):
                    U.append(np.array(map(float,line)))
                    counter += 1
                    continue
                elif (counter == 3):
                    rho.append(np.array(map(float,line)))
                    counter += 1
                    continue
                elif (counter == 4):
                    T.append(np.array(map(float,line)))
                    counter += 1
                    continue
                elif (counter == 5):
                    E.append(np.array(map(float,line)))
```

```
                        counter += 1
                        continue
                    elif (counter == 6):
                        P.append(np.array(map(float,line)))
                        counter += 1
                        continue
                    elif (counter == 7):
                        Q.append(np.array(map(float,line)))
                        counter += 1
                        continue
                    elif (counter == 8):
                        mass.append(np.array(map(float,line)))
                        counter += 1
                        continue
                    elif (counter == 9):
                        V.append(np.array(map(float,line)))
                        counter += 1
                        continue
                    elif (counter ==10):
                        L.append(map(float,line))
                        counter += 1
                    else: exit("Error: counter > 10")
    return ztop, loops, times, spos, np.array(R), np.array(U)
        , np.array(rho), np.array(T), np.array(E), np.array(P)
        , np.array(Q), np.array(mass), np.array(V), np.array(L
        )

def animatePlot(x, dataSet, times, frameTime, xlabel = '',
    ylabel = '', title = '', save=0, name = "animation.mp4",
    rate = 60, log = 0):
    #For when arrays don't vary in size
    fig = plt.figure()
    plt.xlim(np.min(x),np.max(x))
    plt.ylim(np.min(dataSet),np.max(dataSet))
    plt.xlabel(xlabel)
```

```
    plt.ylabel(ylabel)
    plt.title(title)
    if log:
        line, = plt.semilogy([],[],'k',lw=2)
    else: line, = plt.plot([],[], 'k', lw = 2)
    time = plt.figtext(.15, .85, '')

    def init():
        line.set_data([], [])
        return line,
    def animate(i, dataSet, line, time, times):
        time.set_text("Time = " + str(times[i]) + " s")
        line.set_data(x, dataSet[i,:])
        return line,

    anim = ani.FuncAnimation(fig, animate, frames=len(dataSet), interval = frameTime, fargs = (dataSet, line, time, times), repeat = True, init_func = init, blit = False, repeat_delay = 1000)
    plt.draw()

    if (save):
        writer = ani.FFMpegWriter()
        anim.save(name, writer = writer, fps=rate, extra_args=['-vcodec', 'libx264'])

def animatePlot2(x, y, times, xlims, ylims, frameTime, flag=1, xlabel = '', ylabel = '', title = '', save=0, name = "animation.mp4"):
    #For when arrays vary in size
    fig = plt.figure()
    plt.xlim(xlims)
    plt.ylim(ylims)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
```

```
    plt.title(title)
    line,=plt.plot([],[],'k',lw=2.0)
    time = plt.figtext(.15,.85,"")

    def init():
        line.set_data([],[])
        return line,
    def animate(i,y,line,time,times):
        time.set_text("Time = " + str(times[i]) + " s")
        if flag: line.set_data(x[i], y[i])
        else: line.set_data(x[i][:-1], y[i])
        return line,

    anim = ani.FuncAnimation(fig,animate,frames=len(times)-1,interval=frameTime,fargs=(y, line, time, times),repeat=1,init_func = init, blit = 0,repeat_delay = 1000)
    plt.draw()

    if (save):
        writer = ani.FFMpegWriter()
        anim.save(name, writer = writer, extra_args=['-vcodec', 'libx264'])

def sposSpeed(spos, time, radii, P, V, U, postOffset = 5, order = 1):
    sposVel = np.zeros(len(spos))
    if (order == 1):
        for i in range(1, len(spos)-1):
            sposVel[i] = (-radii[i-1,spos[i-1]] + radii[i+1,spos[i+1]])/(time[i+1]-time[i-1])
        sposVel[0] = (-radii[0,spos[0]]+radii[1,spos[1]])/(time[1]-time[0])
        sposVel[-1] = (radii[-1,spos[-1]]-radii[-2,spos[-2]])/(time[-1]-time[-2])

```

```
    machNumber = [(U[i,spos[i]]-sposVel[i])/np.sqrt(gamma*P[i,spos[i]+postOffset]*V[i,spos[i]+postOffset]) for i in range(0, len(sposVel))]
    return np.array(sposVel), np.array(machNumber)

def jumpConditions(spos, sposVel, times, M, rho, U, T, P, preOffset=0, postOffset=5, plot = 0):
    #Predicted Jump conditions from Dr. Lea's Astr Notes
    theoRhoJump = (gamma+1)/(gamma-1 + 2./(M**2))
    theoVelJump = 1./theoRhoJump
    theoTJump = (5./16)*M**2 #This is for large M, see Lea Fluids notes.
    theoPJump = (1./(gamma+1))*(2*gamma*M**2 - (gamma-1))

    #Actual Jumps from data
    actualRhoJump = np.array([np.max(rho[i,:]/rho[i,spos[i]+postOffset]) for i in range(0,len(times))])
    actualVelJump = np.array([(sposVel[i]-np.max(U[i,:]))/(sposVel[i]-U[i,spos[i]+postOffset]) for i in range(0,len(times))])
    actualPJump = np.array([np.max(P[i,:]/P[i,spos[i]+postOffset]) for i in range(0,len(times))])
    actualTJump = []
    for i in range(0,len(times)):
        minTValue = T[0,0] #The Inital Temp of the Blast Zone
        for j in range(0,int(spos[i]-2*Q0)):
            if (np.min(T[i,j]/T[0,-2]) < minTValue):
                minTValue = np.min(T[i,j]/T[0,-2])
        actualTJump.append(minTValue)
    actualTJump=np.array(actualTJump)

    #4 2-D arrays. First index of each array determines predicted vs. actual.
    #Second index determines which time step.
    rhoJump = np.array([theoRhoJump,actualRhoJump])
```

```
    velJump = np.array([theoVelJump,actualVelJump])
    TJump = np.array([theoTJump,actualTJump])
    PJump = np.array([theoPJump,actualPJump])

    if plot:
        plt.figure(1)
        plt.plot(times, rhoJump[0], 'k.', times, rhoJump[1],
            'r.')
        plt.xlabel(r"Time ($s$)")
        plt.ylabel("Jump Ratio")
        plt.title("Density Jump Ratio")
        plt.legend(["Predicted","Actual"], loc=4)

        plt.figure(2)
        plt.plot(times, velJump[0], 'k.', times, velJump[1],
            'r.')
        plt.xlabel(r"Time ($s$)")
        plt.ylabel("Jump Ratio")
        plt.title("Velocity Jump Ratio")
        plt.legend(["Predicted","Actual"], loc=1)

        plt.figure(3)
        plt.semilogy(times, TJump[0], 'k.', times, TJump[1],
            'r.')
        plt.xlabel(r"Time ($s$)")
        plt.ylabel("Jump Ratio")
        plt.title("Temperature Jump Ratio")
        plt.legend(["Predicted ","Actual"], loc=1)

        plt.figure(4)
        plt.semilogy(times, PJump[0], 'k.', times, PJump[1],
            'r.')
        plt.xlabel(r"Time ($s$)")
        plt.ylabel("Jump Ratio")
        plt.title("Pressure Jump Ratio")
```

```
            plt.legend(["Predicted","Actual"], loc=1)

    return rhoJump, velJump, TJump, PJump

def multiplot(R, y, times, t1, t2, t3, t4, yaxis = '', title='', flag = 1):
    f,axes = plt.subplots(2,2)
    ((ax1,ax2),(ax3,ax4)) = axes
    if flag:
        ax1.plot(R[t1][:-1],y[t1], lw=2)
        ax1.set_title("Time: "+str(times[t1])+"s")
        ax2.plot(R[t2][:-1],y[t2], lw=2)
        ax2.set_title("Time: "+str(times[t2])+"s")
        ax3.plot(R[t3][:-1],y[t3], lw=2)
        ax3.set_title("Time: "+str(times[t3])+"s")
        ax4.plot(R[t4][:-1],y[t4], lw=2)
        ax4.set_title("Time: "+str(times[t4])+"s")
    else:
        ax1.plot(R[t1][:],y[t1], lw=2)
        ax1.set_title("Time: "+str(times[t1])+"s")
        ax2.plot(R[t2][:],y[t2], lw=2)
        ax2.set_title("Time: "+str(times[t2])+"s")
        ax3.plot(R[t3][:],y[t3], lw=2)
        ax3.set_title("Time: "+str(times[t3])+"s")
        ax4.plot(R[t4][:],y[t4], lw=2)
        ax4.set_title("Time: "+str(times[t4])+"s")

    #Consider generalizing this to any [square] number of plots in the multiplot
    #
    #It doesn't even have to be a square number of plots...

    for axis in axes:
        for element in axis:
            element.set_xlabel(r"Radius $(cm)$")
```

```
            element.set_ylabel(yaxis)
            element.xaxis.label.set_fontsize(20)
            element.yaxis.label.set_fontsize(20)

    f.tight_layout()
```