TRACE ELEMENT ANALYSIS IN A NUMERICAL SIMULATION OF AN INDUCTIVELY COUPLED PLASMA MASS SPECTROMETER

by

Andrew Joseph Sampson

A senior thesis submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Bachelor of Science

Department of Physics and Astronomy

Brigham Young University

August 2007

BRIGHAM YOUNG UNIVERSITY

DEPARTMENT APPROVAL

of a senior thesis submitted by

Andrew Joseph Sampson

This thesis has been reviewed by the research advisor, research coordinator, and department chair and has been found to be satisfactory.

Date	Ross Spencer, Advisor
Date	Eric Hintz, Research Coordinator
Date	Ross Spencer, Chair

ABSTRACT

TRACE ELEMENT ANALYSIS IN A NUMERICAL SIMULATION OF AN INDUCTIVELY COUPLED PLASMA MASS SPECTROMETER

Andrew Joseph Sampson

Department of Physics and Astronomy

Bachelor of Science

FENIX is a program designed to simulate an inductively coupled plasma mass spectrometer using a particle-in-cell method call Direct Simulation Monte Carlo. FENIX specifically focuses on the supersonic expansion region in the inductively coupled plasma mass spectrometer. I have been working on the introduction of an analyte with the end result of comparing computational results with experimental results. Initial testing has shown similar behavior to the main element within the inductively coupled plasma mass spectrometer, but has also revealed areas that need to be re-evaluated.

ACKNOWLEDGMENTS

I would like to first acknowledge my beautiful wife whose never-ending support is always felt and appreciated. I would also like to thank my advisor Ross Spencer for his bottomless reservoirs of knowledge, enthusiasm, and kindness that are always available upon request. Appreciation is also given to the members of the FENIX research group: Adam Payne, William Somers, Charlie Woods, Jamie Palmer, and Jaron Krogel for all their input and expertise.

Financial support from the United States Department of Energy is gratefully acknowledged.

Contents

List of Figures

Chapter 1

Introduction

1.1 Inductively Coupled Plasma Mass Spectrometry

Mass spectrometry is used to identify unknown molecules from the charge-to-mass ratio of ions made from the molecules. In inductively coupled plasma (ICP) mass spectrometry, a hot plasma (about 1 eV) is used to break a molecule into ions that are processed by a mass spectrometer. A device that employs this process is called an ICP mass spectrometer (ICP-MS).

A schematic of an ICP-MS is found in Fig. 1.1. In an ICP-MS, a nonreactive gas, usually argon, flows into a chamber at atmospheric pressure. A nebulizer puts thousands of tiny droplets of sample within the chamber. This argon/sample mixture is then directed into a cylindrical area surrounded by a load coil, a large inductor with an oscillating electric current. Through the application of Faraday's law, a powerful electric field is produced that ionizes the argon. This forces a phase change in the argon from gas to a plasma. (The use of an inductor to couple energy with the plasma gives rise to the name "inductively coupled plasma.") During argon ionization and

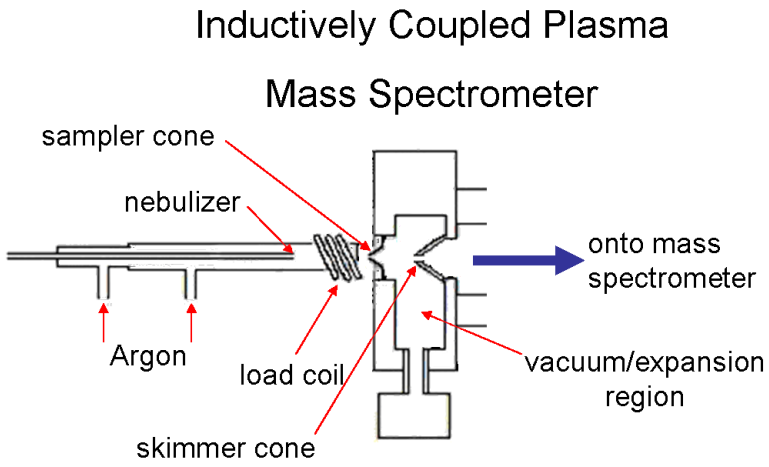

Figure 1.1 Schematic of an inductively coupled plasma mass spectrometer

afterwards, the sample experiences collisional heating and ionization from the hot argon. The flow is then incident upon what is called the sampler cone of the ICP-MS (see Fig. 1.1). At the tip of the sampler cone is a hole of diameter 1 mm that opens into a vacuum region typically held at 1 torr. This pressure gradient between 760 torr and 1 torr causes the mixture to undergo a supersonic expansion through the nozzle of the cone into the vacuum/expansion region. A second cone called the skimmer cone is 6.3 mm downstream from the sampler and skims off a collimated beam of ions that travels onward to the mass spectrometer for analysis.

Because of the method used to introduce the sample via a nebulizer, the ICP-MS is mainly used to identify trace elements/analytes in a solution. It has use in urine testing, water sampling for metal toxins and other environmental analysis, cancer diagnosis [1], and many other applications as well.

1.2 Uncertainty in the Field

There are many areas of uncertainty in the field of ICP mass spectrometry. The effect of the matrix in the region between the load coil and the sampler cone is one of these areas of uncertainty. (The matrix is everything in the sampler other than the analyte and it will be discussed with more detail below.) Matrix effects were studied as early as 1988 with a paper published by Douglas and French [2]. They concluded, from a very simple model, that the matrix effects would be negligible and not affect the ion sampling distribution. This has been refuted by a number of authors, however [3]. To understand this phenomenon, a more thorough introduction to how an ICP-MS is used will be needed.

Many times in ICP mass spectrometry it is desirable not only to identify the unknown analyte, but also to find its concentration within the sample. To do this, the user will premix a few aqueous solutions with known concentrations of the analyte and analyze them with the ICP-MS. The counts received from the mass spectrometer directly correspond to the known concentrations. Together they produce a concentration calibration curve for the analyte within the original sample. Interpolation is used to close the gaps between the data for the known concentration samples. The original sample with unknown concentration is then analyzed with the ICP-MS and compared with the calibration curve to find the concentration of the analyte from the original sample.

As said before, the matrix is everything in the sample other then the analyte. For example, if an analysis of blood is needed to determine whether lead is present, the blood is the matrix. On the other hand when a calibration curve for lead is produced, the aqueous base to the solutions of known lead concentrations is the matrix. Matrix effects are observed when different counts from the mass spectrometer are received

from the same concentrations of analyte when different matrices are used in the ICP-MS [4]. For example, when a sample of blood is analyzed with the ICP-MS, it returns a different count than a premixed solution of the same concentration. One way to minimize matrix effects is to create the calibration curve from a solution with a matrix similar to the one in the sample in question [5].

Another area of uncertainty discovered in experiments here at Brigham Young University is that the analyte ion density drops more quickly approaching the nozzle than does the neutral argon density [6]. This has proven difficult to explore because of the difficulty of studying individual ion interactions with argon particles.

1.3 FENIX with the Introduction of Trace Elements

To further understand the difficulties faced in ICP mass spectrometry, our group at BYU is working with Paul Farnsworth in the chemistry department to discover the inner workings of an ICP-MS. We have written a code called FENIX that simulates the supersonic expansion through the sampler cone. The Farnsworth group has been doing experimental research using the fluorescence of different analytes to measure the analytes density through the sampler cone of the ICP-MS. Using FENIX, our goal is to help them understand their data by comparing results from our simulation to their experimental results. Because argon has such a high excitation energy it is hard to use fluorescence to view its density; therefore, Dr. Farnsworth has extremely limited data on the argon physics. Because of the scarcity of argon data, it is necessary to compare analyte data instead. This means FENIX must have a trace element included in its simulations. I have been working on adding trace elements to FENIX with the ultimate goal of comparing results with those obtained experimentally. There are

many ways to introduce a trace element and only one way will be discussed in this paper. Results from testing have been encouraging, and at the same time they have revealed computational techniques that need to be improved.

Chapter 2

Computational Set-up

2.1 Direct Simulation Monte Carlo

Direct simulation Monte Carlo (DSMC) is a computational technique developed by Graeme Bird [7] in the late 1960s to simulate fluid dynamics. DSMC keeps track of each simulation particle's position and velocity; it does not provide solutions to the fluid equations. DSMC models representative particles in real positions with real velocities, thus the term "direct simulation" is used in the title of the algorithm. "Monte Carlo" refers to the continual use of a random number generator in selecting particles for collision and calculating their velocities after collision.

Although DSMC uses a direct method for simulating a fluid, there are a few approximations that it utilizes. The main approximation is forced on us because of limited computer memory. Garcia [8] points out that if you were to simulate every single particle of ambient air in $1\mu m^3$, there would be 27×10^6 particles. That roughly requires about 1.3 gigabytes of memory. That's feasible, but that's also only $1\mu m^3$. The simulation region for introducing trace elements is roughly 50 mm^3, and there are 10^9 μm^3 in 1 mm^3. The memory required for the storage of particle

information is about 6×10^{10} gigabytes. To solve this problem, Bird introduced the idea of representative particles. One representative particle actually simulates N_{ef} real particles. This approximation is made under the assumption that a group of particles will flow similarly to one particle in the same vicinity. Making this approximation makes it possible to directly simulate a fluid with current computers.

The most important element of the simulation involves collisions between representative particles. The collider, the algorithm that processes collisions, is designed to simulate the effects of real particle collisions during the course of colliding representative particles with each other. First it picks two representative particles at random that are relatively close to each other. The collider then picks them as a collision pair if they pass a test on their relative velocity, ensuring that the collision rate is proportional to relative velocity. The person running DSMC will code the collider to use the type of potential desired between the colliding particles. The most basic potential is the hard sphere scattering potential which treats particles like billiard balls and collides them elastically. More complicated scattering potentials can be used if desired. After the collision is processed, the direction of the two colliding particles is randomly chosen, and the collider then picks two new particles to repeat the process.

For the collider to correctly simulate the effects of real particle collisions, three things must happen. First the simulation region must be divided into collision cells; an array keeps track of the simulation particles in each cell. The collider then uses the information contained in this array to process collisions between particles in the same collision cell. To best simulate real collisions, the collision cells need to be smaller in size then a mean-free path and the time step `tau` must less than the average collision time. Second, the accuracy of the simulation increases with the number of representative particles in a cell. The more representative particles, the better the statistics. For the statistics to be adequate, there should be about 20-30

simulation particles per collision cell. The third constraint in the collider is that the two colliding representative particles must have the same N_{ef} for the statistics to be modeled correctly. Remember that DSMC makes the approximation that several particles will flow the same as one particle in the same area. The collider employs this approximation by basically processing a collision between two *real* particles and then applies the result to a pair of *representative* particles. This means that with every *representative* collision, the collider processes N_{ef} *real* collisions with N_{ef} *real* collision pairs. When adding a trace element in DSMC, two species are needed with different densities. This causes a different N_{ef} to be defined for each species' representative particles for the desired number of simulation particles in a cell to be reached. The simulation would then have two different N_{ef}'s, N_{ef}^1 and N_{ef}^2, for the different species. When processing a collision, the difference $|N_{ef}^1 - N_{ef}^2|$ corresponds to *real* particles that were computationally collided with no collision partners. This is why when processing a collision between two representative particles in the traditional way, they must have the same N_{ef}. It poses the main problem when adding a trace element in FENIX and will be discussed in a later section. If any of the three constraints discussed in this section are violated, it would be statistically incorrect and would give invalid results in a simulation.

A more exhaustive explanation of DSMC can be found in Refs. [7] and [8].

2.2 Navier Stokes vs. DSMC

It could be asked why DSMC should be used when the fluid equations are generally more simple to work with. The fluid equations are more readily used to solve fluid dynamics, but they have limitations. The Navier Stokes equation for a compressible

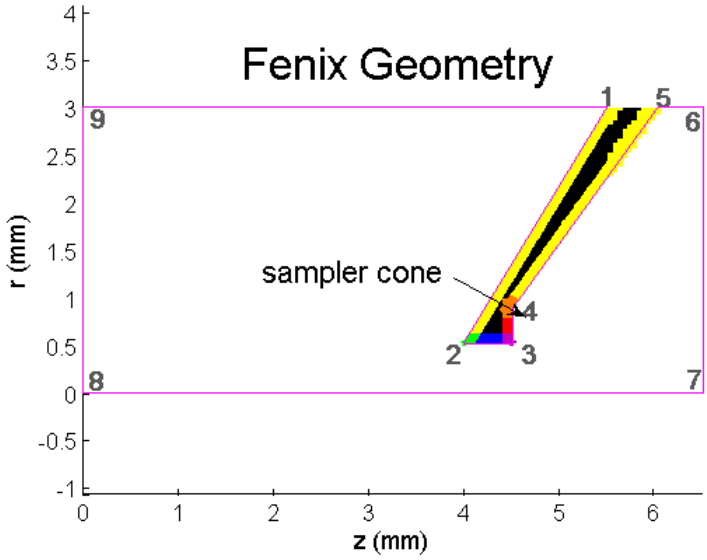

Figure 2.1 FENIX geometry for introducing trace elements.

fluid is:

$$\rho\left[\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v}\cdot\nabla\mathbf{v}\right] = -\nabla \mathrm{P} + \mu\left[\nabla^2\mathbf{v} + \frac{1}{3}\nabla(\nabla\cdot\mathbf{v})\right] \tag{2.1}$$

where $\mathbf{v}$ is the velocity vector, P is the pressure, ρ is the density, and μ is the viscosity. The reason we cannot use (2.1) is because the Knudsen number is too large in the expansion region of the ICP-MS. The Knudsen number is the ratio of the local particle mean-free-path to the length scale of the simulation. If the Knudsen number is close to or greater than about 0.1, then the assumptions made in fluid mechanics are no longer a good approximation and a statistical approach should be used instead. In ICP, the Knudsen number is too large in the expansion region to use fluid mechanics, so we use DSMC.

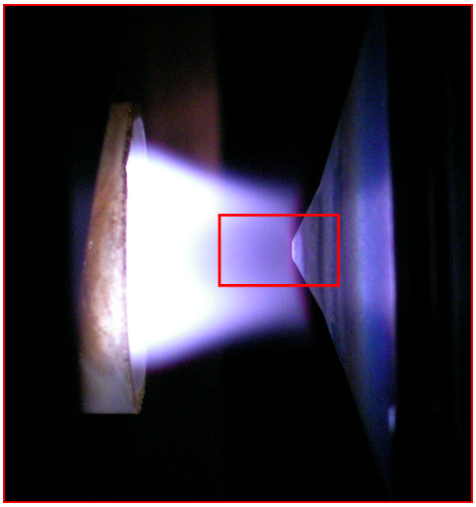

Figure 2.2 A picture of the torch and sampling cone in an ICP-MS. The red box is drawn around the region simulated in FENIX.

2.3 FENIX

FENIX is a particle-in-cell simulation that employs DSMC to simulate the ICP-MS, and it has been a work-in-progress within our research group for the past four years. Fig. 2.1 shows the basic geometry for FENIX. To simplify many geometric calculations in the simulation, FENIX has been built with cylindrical symmetry. This allows the program to approach a three dimensional problem in only two dimensions. Fig. 2.2 is an actual photograph taken of our simulation region within the ICP-MS.

The different colored polygons in Fig. 2.1 represent different masked areas. A mask is a region marked for a particular action. The masks in FENIX are for processing reflections with the metal sampler cone. When a particle crosses the metal boundary into a mask, the mask will tell FENIX how to act on the particles and what type of reflection to process. The masks are utilized as a simple organization tool.

FENIX runs on a time loop that continually repeats until the program finishes. There are four main actions that take place during with loop with a number of supporting routines. First FENIX simply moves the particles using the basic kinematic

equation:

$$\mathbf{x} = \mathbf{x}_0 + \mathbf{v}\tau \tag{2.2}$$

where $\mathbf{x}$ is the new position in three-space, $\mathbf{x}_0$ is the old position in three-space, $\mathbf{v}$ is the velocity in three-space, and τ is the time step. Next the collision cell each particle is in is determined. Then metal boundary conditions are processed on the sampler cone and particles reflect based on their mask. In other words, if a particle has moved into a masked region inside the sampler cone, FENIX will know that particle crossed and metal boundary and reflects it accordingly. finally, collisions are processed between representative particles cell-by-cell. FENIX entails much more then these four steps, but these four are the heart of the algorithm.

2.4 Limitations on FENIX

Although FENIX is a powerful tool for simulating fluid dynamics, FENIX is bounded by three main limitations: memory, axial cell volume, and the consequences of using representative particles. FENIX is limited by random access memory (RAM) because of the great volume that is needed to simulate the introduction of trace elements. Even though representative particles have been introduced, it does not free up enough RAM because of FENIX's large geometry. An easy solution to this would be to use a smaller geometry for trace elements in FENIX. A smaller geometry would allow the use of a smaller number of representative particles. Unfortunately, this would also require an understanding of the trace-element-boundary conditions outside the simulation region upstream from the sampler cone. This is not known; therefore, we must stay with the larger geometry for the time being and find a way to get around the RAM limitation.

Another limitation on FENIX is the axial cell volume. The collision cells are defined

by a two-dimensional, mesh grid that overlays Fig. 2.1. Since this is in cylindrical symmetry, as the mesh is rotated about the z-axis, each square in the grid traces out a toroid with a square cross-section. These toroids are the three-dimensional collision cells. At the top of the simulation, these cells have a large volume. Since the density is practically the same throughout the simulation upstream from the sampler cone, these high volume cells have a large number of particles compared with the cells on the radial axis of symmetry (axial cells) that have a mush smaller volume. As stated before, DSMC requires about 20-30 particles per cell for statistics to work correctly. Currently in a simulation of 5 million particles, there are about two simulation particles per axial cell, and it uses about 3 GB of RAM. To get the minimum of 20 particles per cell on axis, roughly multiply the number of particles by ten giving 50 million simulation particles. This simulation requires about 14 GB of RAM. The largest supercomputer available to our group has a maximum of 8 GB of RAM per node. FENIX currently can only run on 1 node giving it the limit of 8 GB. It is possible to add particles until the 8 GB limit has been reached, but this also requires more time to run the program to completion. The geometry for trace element introduction has been loaded to the maximum RAM and takes roughly a month to run to completion. That also gives about 15 particles per cell along the axis.

The final limitation in FENIX is the consequence of using representative particles. As explained in Sec. 2.1, a collision between two representative particles with two different N_{ef}'s is inaccurate. This restriction can arise when introducing a trace element because the trace element has a much smaller particle density than that of the main element. Because of this difference, the trace N_{ef} is naturally much smaller than the main N_{ef} to keep the ideal greater than 20 particles-per-cell requirement for statistics. Their different N_{ef}'s create an unphysical result when a collision is processed between the two different species. This has been a major roadblock in

the development of trace elemental analysis in FENIX, but a possible solution will be presented in the next section.

2.5 Computational Approach to Trace Element Analysis

As mentioned above the idea of representative particles has made it difficult to add a trace element in FENIX. Cross-colliding these two simulation particles would be disastrous for the statistics in DSMC. To get around this problem, a simple approximation is made using a fundamental property of trace elements. Because of its low density a trace element should not noticeably affect the main element. We may exploit this principle when processing a cross-collision. In effect, as in a normal collision, two real particles are taken from the cross-colliding-representative particles: one from the main element, and another from the trace element. A collision is processed between the two real particles, and the effects of the collision are applied to the rest of the trace-representative particle. The main element on the other hand is kept the same as it was before the collision: all effects from the cross-collision are annulled, and it was as if the collision never took place for the main element. Basically, the cross-collision is processed as if the trace-representative particle collided with a main-representative particle of the same N_{ef}. This is statistically accurate for the trace element. On the other hand, the main element acts as if it was never collided, thus not destroying the statistical outcome. It is here that the constraint that a trace element should not noticeably affect the main element is enforced: the main element's properties never change through cross-collisions. With this approximation, FENIX is able to simulate trace elements.

Chapter 3

Refinement of Numerical Techniques

3.1 Trace Element Flow Properties

Three main checks were done to test the cross-collision algorithm discussed in Sec. 2.5. These tests were: barium diffusion through the argon, a flow comparison between barium and argon, and a temperature comparison between the barium and argon. The first two checks are considered in this section, while the latter will be the subject of the next section.

First, the diffusion of barium in argon will be considered. When the barium is introduced, it is done with a radial-Maxwellian density profile with a radial full-width at half-maximum of 1.5 mm at a location 5 mm upstream from the sampler cone (this corresponds to $z = 0$ mm in the geometry). As this barium approaches the nozzle, the barium should diffuse out causing the Maxwellian density distribution to widen radially. Fig. 3.1 shows the density of barium in steady state in the radially symmetric FENIX geometry. The diffusion is small but discernible in this picture. The barium

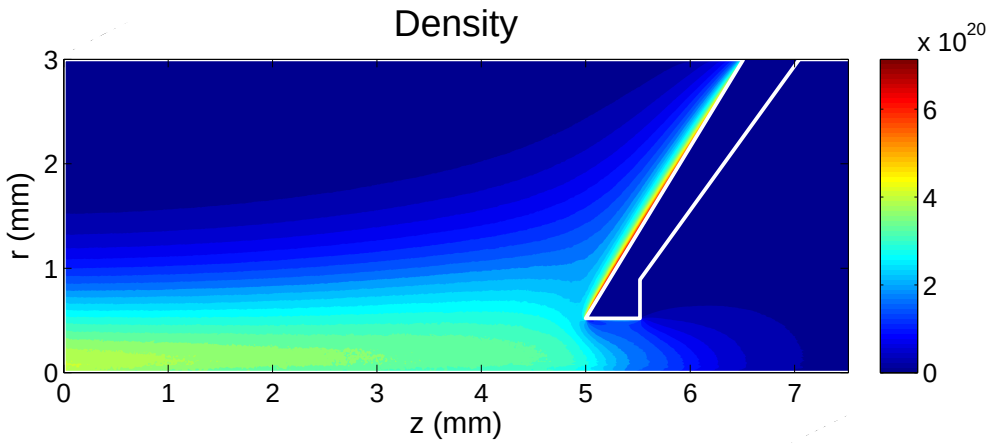

Figure 3.1 A contour plot of the barium density in FENIX's radially symmetric geometry.

diffusion is better seen in Fig. 3.2 which shows a graph of four different radial scans of the barium density taken at four different z positions. The first scan is an average radial density at the interval z =[0,1] mm, the second at [1,2] mm, the third at [2,3] mm, and the fourth from [3,4] mm. This picture shows the diffusion of barium through the argon as it approaches the sampler cone. Something else noticeable in Fig. 3.2 is a consistent dip in barium density on axis. As of yet, we do not understand this effect.

Next, a comparison is needed between the flow characteristics of barium and argon. Fig. 3.3 is a plot of both argon and barium streamlines in FENIX. It is observed that although the streamlines start the at the same radii, the barium streamlines do not overlay the argon streamlines exactly. The barium does follow the argon, especially where there is high velocity through the nozzle, but its streamlines spread more than the argon streamlines. This correlates with the barium diffusion. Because the barium diffuses outward it has a flow pattern that will spread more radially than the argon.

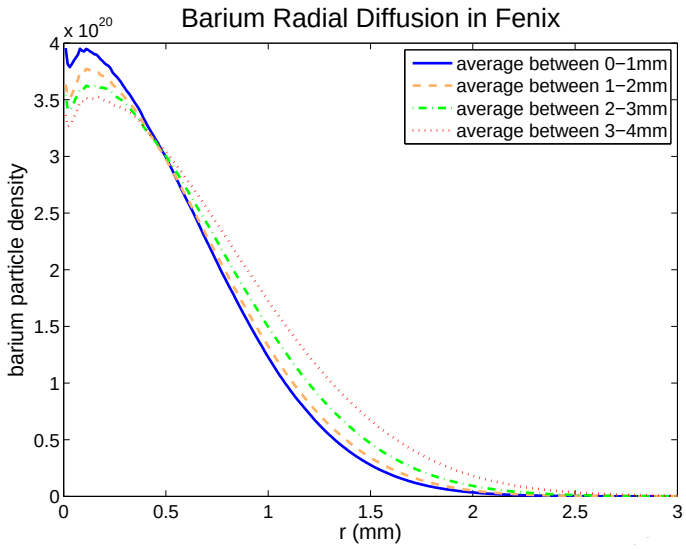

Figure 3.2 This plot shows four radial scans of barium density averaged over a 1 mm each in z. The changing of the Maxwellian distribution shows the diffusion of barium within the argon.

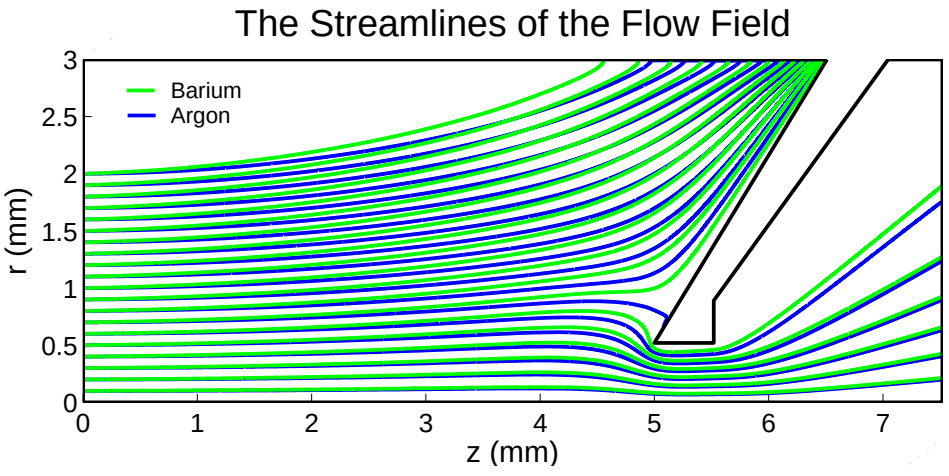

Figure 3.3 Streamlines of both barium and argon in FENIX.

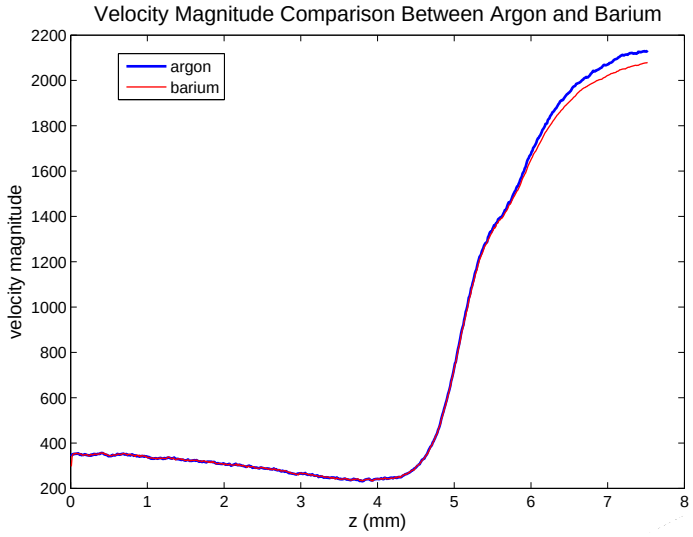

Figure 3.4 The velocity magnitude of argon vs. barium along the axis of symmetry in FENIX.

Another result is the agreement between the barium and argon velocity magnitude down the axis of the simulation. Fig. 3.4 shows the velocity magnitude of both argon and barium along the central axis in FENIX. They match closely until about z =5.52e-3 mm which happens to be the end of the sampler cone nozzle. Following the nozzle, the barium velocity is a little slower than the argon. This is actually a physical result. Past the sampler cone, the mean-free-path increases causing the barium and argon to be no longer tightly coupled. They feel the same pressure force and since barium has a higher mass, it experiences less acceleration.

3.2 Barium and Argon Temperature Comparison

Since barium is introduced with a temperature of 3000 K and is in thermal contact with 4900 K argon, the two should come to thermal equilibrium. Because argon is not affected by the cross-collider (Sec. 2.5), the barium should be 4900 K after thermal equilibrium. Figs. 3.5 and 3.6 show the temperature comparison for both a radial scan

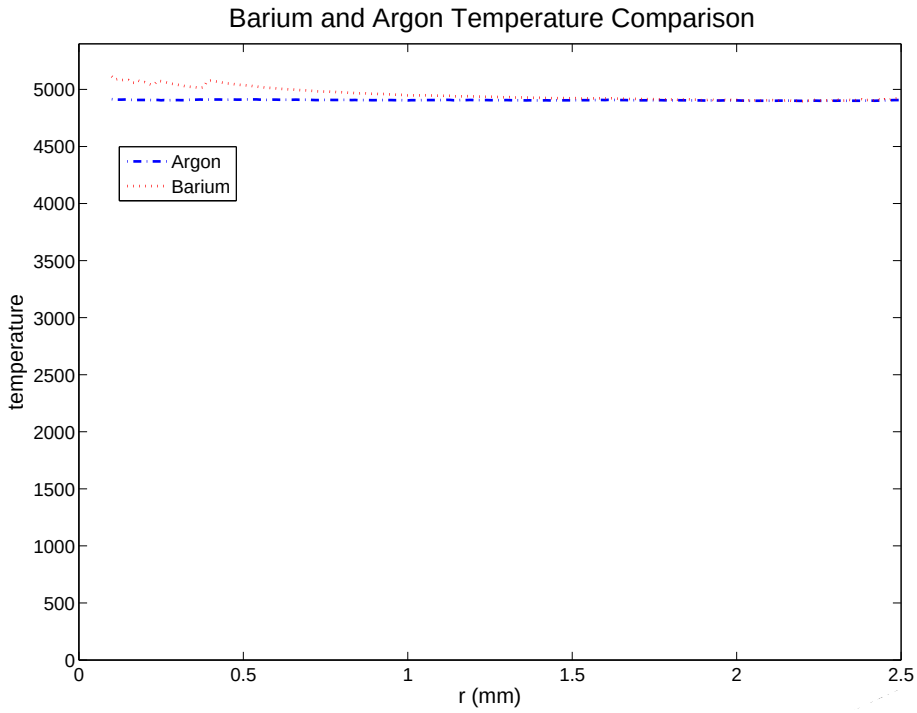

Figure 3.5 Radial temperature comparison of barium with argon averaged over 2-4 mm in z upstream from the sampler cone.

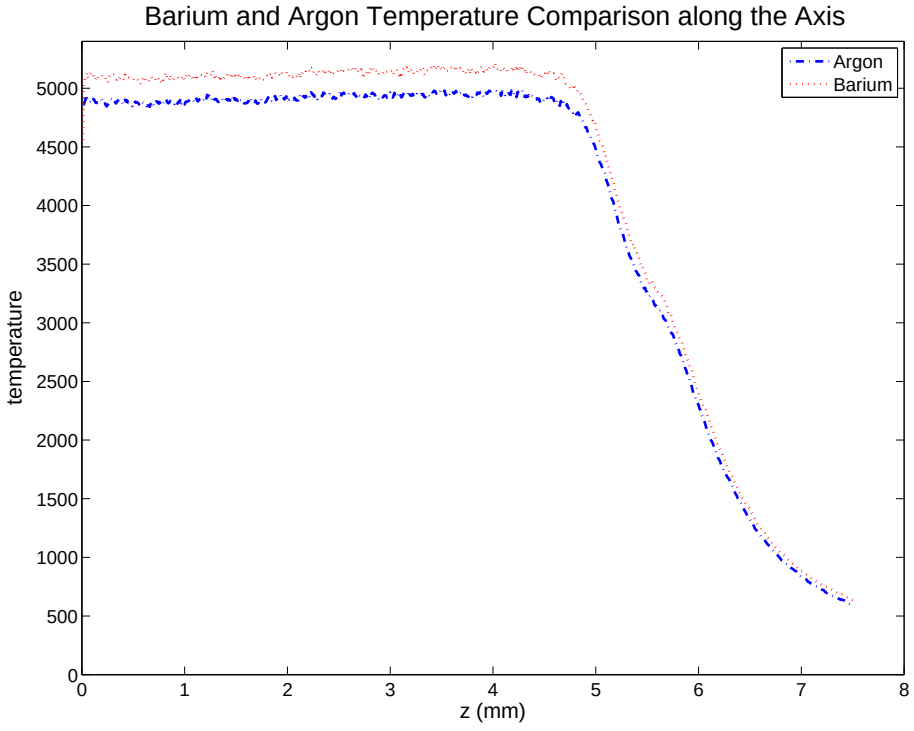

Figure 3.6 Axial temperature comparison of barium with argon.

and an axial scan. On the radial scan, the barium has a much higher temperature on axis than the argon. As the radius increases, the barium temperature converges to the argon temperature. This is a completely unexpected result, and is physically incorrect. The raised temperature of barium above that of the argon is also evident in the centerline temperature in Fig. 3.6. It is a good sign, however, to see that barium and argon have the same behavior down the axis. This shows the collider is trying to bring about thermal equilibrium but fails as demonstrated by both Fig . 3.5 and 3.6. This difference in temperature demonstrates a flaw in FENIX that will be the subject of the next section.

3.3 The Temperature Solution

In section 2.4, the 20-30 particles per cell requirement was emphasized. DSMC is a code that thrives on enough particles to make statistics work; if there are not enough particles, something will suffer. In the case of FENIX with trace elements the barium temperature on axis is higher than the argon. As a test to see if this really was the issue, a matlab m-file (contained in the appendix called `fenix9coll.m`) was written that simulates one collision cell in FENIX. Collisions take place between an adjustable amount of both barium and argon particles. In the FENIX run that produced Figs. 3.5 and 3.6, there was an average of 8 argon particles and 20 barium particles in the axial cells. Fig. 3.7 was produced by setting up `fenix9coll.m` with 8 argon and 20 barium particles and time averaged the velocities of the barium particles. The time-averaged velocity distribution was then fitted to a Maxwellian. To recreate the effect in FENIX, the argon used was at 4900 K. The time-averaged temperature of the barium was 5079 K (the fit returned 5049 K, off by .5% because it forces a Maxwellian fit to a distribution that might not be 100% Maxwellian). Fig. 3.7 reveals the same behavior

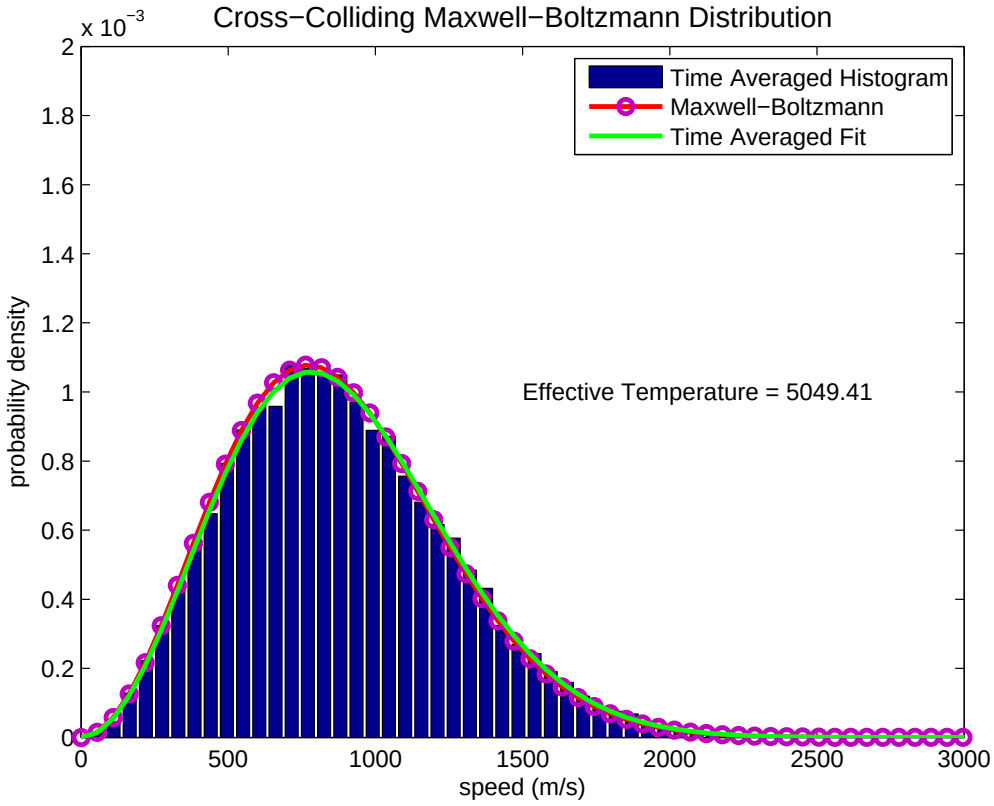

Figure 3.7 The red curve represents the Maxwell-Boltzmann distribution the barium should have. The green curve shows a least-squares fit to the histogram of the barium speed distribution. The effective temperature is a value obtained from the fit. The calculated temperature from the barium data itself is 5078.8 K.

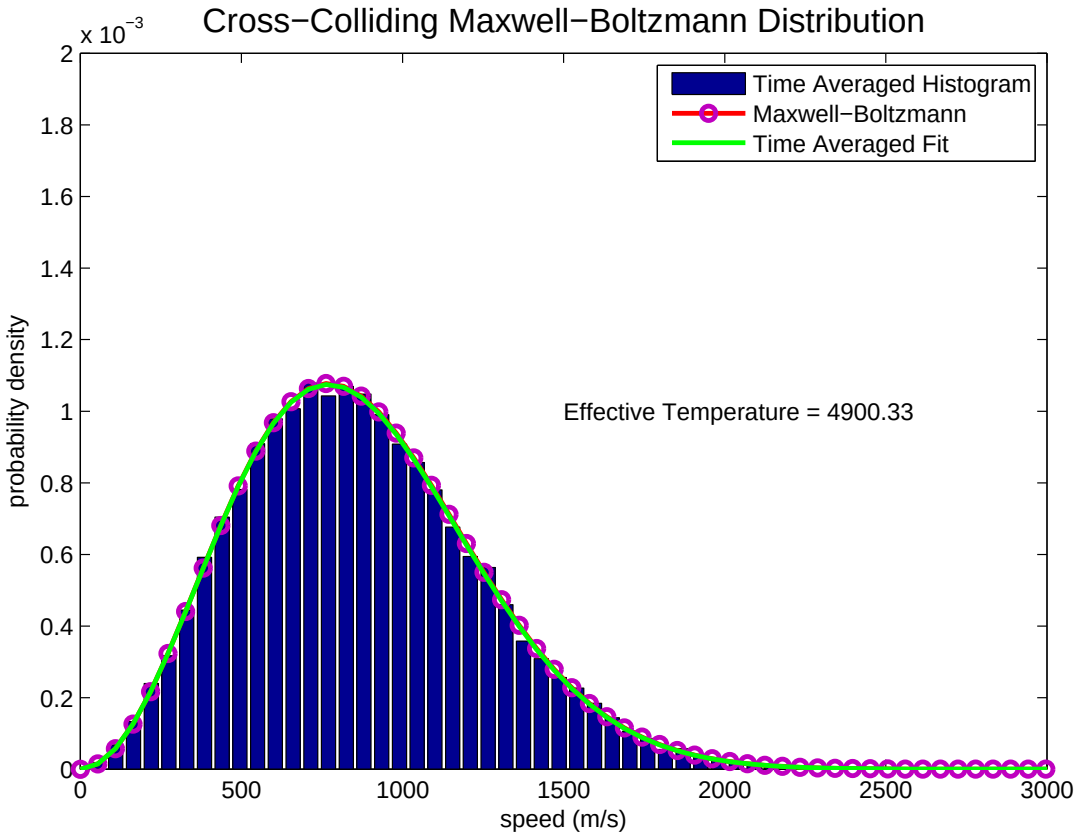

Figure 3.8 This is much better fit then Fig. 3.7. This is using 100 argon particles instead of 10.

as FENIX on axis. After increasing the number of argon particles from 8 to 100, the new distribution is shown in Fig. 3.8. It is clear that with more argon particles in the cell, the barium temperature converges to the argon temperature. In the outer radial cells, there are on average about 100 argon simulation particles. This explains why the barium temperature does not converge with the argon on axis, where there are only an average of 8 argon particles, but converges in the outer radial cells. The simple solution to this problem would be to add more argon particles, but this would also lead back to the memory issues explained in Sec. 2.4. FENIX is already at the memory limits and can access no more.

To correct these unphysical results without having to add more particles we are developing a method that redefines how the argon gas is represented in FENIX. This method would not require the storage of argon simulation particles, freeing up memory for other usage. FENIX has been simulating argon long enough for us to be confident in its results. For each collision cell in the simulation we know the argon density, temperature, and flow velocity. With this information stored for every collision cell, it is possible to create an argon particle on the fly as a collision candidate for barium. The argon density information will determine how many collision candidates are chosen. For each barium candidate, an argon particle will be created, using the standard Box-Muller algorithm for reproducing a Maxwellian distribution. The fundamental equations:

$$\begin{aligned} v_\perp &= v_{th}\sqrt{-2.0\ln \Re_i} \\ \theta &= 2\pi \Re_j \\ v_x &= v_\perp \cos\theta \\ v_y &= v_\perp \sin\theta \end{aligned} \tag{3.1}$$

are used in the Box-Muller algorithm, where $\Re_i$ and $\Re_j$ are random numbers on the unit interval. After the new argon particle's velocity has been produced in this way, the local flow velocity is added to it. This procedure thus produces a local random-thermalized-argon particle to collide with the barium collision candidate. The collider would then function as it normally does and process the collision. Instead of trying to sample a full distribution from 8-100 argon particles, this method samples from a local Maxwellian allowing collisions to be statistically sound and not require extensive memory. Fig. 3.9 is again a time averaged plot of the barium distribution after colliding with argon particles that were created on the spot from a Maxwellian distribution at 5400 K. The temperature of the barium converged to 5401 K, an error of $\sim 0.002\%$. Treating the argon this way will give us an opportunity to simulate the

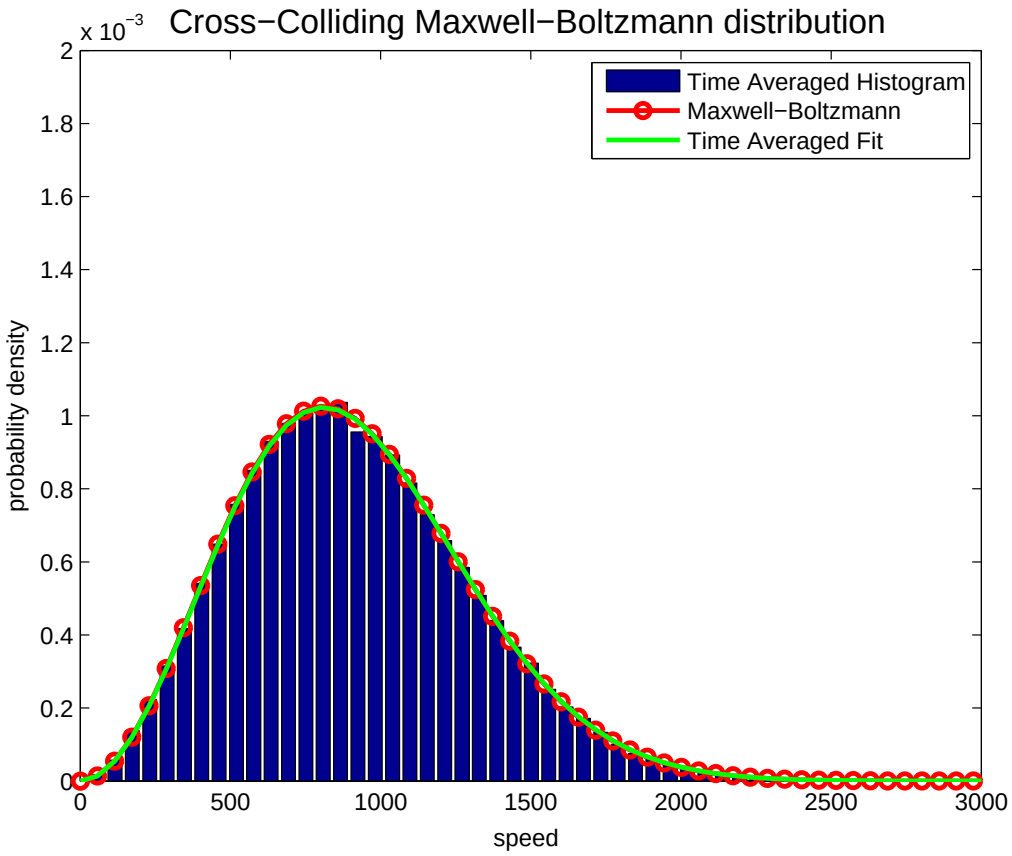

Figure 3.9 This picture uses the new method for colliding barium with argon by creating a collision partner on the fly. Temperature of barium in this distribution is 5401.05 K while the argon distribution was at 5400 K.

trace element in FENIX and bring us closer to comparing results with experimental data.

3.4 Conclusion

We have discovered that the method discussed here for the introduction of an analyte in FENIX will work statistically with the hard-sphere model. On the other hand, the way argon is treated must change to allow barium statistics to function properly. These changes will be explored in the near future.

Bibliography

[1] S. Boulyga, V. Loreti, J. Bettmer, and K. Heumann, "Application for SEC-ICP-MS for comparative analyses of metal-containing species in cancerous and heathly human thyroid samples," Analytical and Bioanalytical Chemistry **380,** 198–203 (2004).

[2] D. J. Douglas and J. B. French, "Gas Dynamics of the Inductively Coupled Plasma Mass Spectrometry," Journal of Analytical Atomic Spectrometry **3,** 743–747 (1988).

[3] S. H. Hobbs and J. W. Olesik, "The influence of incompletely disolved droplets and vaporizing particles on chemical matrix effects in inductively coupled plasma spectrometry: time-gated optical emission and laser-induced fluorescence measurements," Spectochimica Acta Part B **52,** 353–367 (1997).

[4] A. C. Lazar and P. B. Farnsworth, "Matrix Effect Studies in the Inductively Coupled Plasma with Monodisperse Droplets. Part I: The Influence of Matrix on the Verticle Analyte Emission Profile," Applied Spectroscopy **53,** 457–469 (1999).

[5] S. D. Tanner, "Characterization of Ionization and Matrix Suppression in Inductively Coupled 'Cold' Plasma Mass Spectrometry," Journal of Analytical Atomic Spectrometry **10,** 905–921 (1995).

[6] A. A. Mills, J. H. Macedone, and P. B. Farnsworth, "High resolution imaging of barium ions and atoms near the sampling cone of an inductively coupled plasma mass spectrometer," Spectrochimica Acta Part B **61,** 1039–1049 (2006).

[7] G. A. Bird, *Molecular Gas Dynamics and the Direct Simulation of Gas Flows* (Oxford University Press, New York, NY, 1994).

[8] A. L. Garcia, *Numerical Methods for Physics*, 2 ed. (Prentice Hall, Upper Saddle River, NJ, 2000).

Appendix A

Fenix8.f Segments of Code

A.1 crosscollider.f

```
c BEGIN CROSSCOLLIDER5

      !This collider will work off the idea that Trace elements do not affect
      ! the main element. With this idea, the main element will act on the
      ! Trace element, but the Trace element will not act on the main element.
      ! In this case, only the Trace element will have changing values from the
      ! collision, but NOT THE MAIN ELEMENT. ITS VALUES WILL NOT CHANGE IN
      ! THIS COLLIDER, ONLY THE TRACE!!!

      subroutine crosscollider5(xpf,ypf,zpf,vxpf,vypf,vzpf,psabsminf,
     &     xptf,yptf,zptf,vxptf,vyptf,vzpTf,psabsmintf,
     &     sigmaX,vrmXf,nincellmaxf,nc,collfracXf,
     &     cellistTf,cellistf,nincellmaxTf,nmsim,ntrsim,
     &     ncbegcoll,ncendcoll)

      use constants
      use geometry
      use timestepvars
      use toggles
      use timings
      use randomnumbers
      use specievars
```

```
      implicit none
c Argument declarations
      integer nmsim,ntrsim,nincellmaxf,nincellmaxTf
      integer nc,cellistf(nincellmaxf,nc),cellistTf(nincellmaxTf,nc)
      real(8) xpf(1:nmsim),ypf(1:nmsim),zpf(1:nmsim)
      real(8) vxpf(1:nmsim),vypf(1:nmsim),vzpf(1:nmsim)
      real(8) xptf(1:ntrsim),yptf(1:ntrsim),zpTf(1:ntrsim)
      real(8) vxptf(1:ntrsim),vyptf(1:ntrsim),vzpTf(1:ntrsim)
      real(8) psabsminf,psabsmintf,collfracXf(1:nc),vrmXf(1:nc)
      real(8) sigmaX

      !local variables
      integer i,j,k,n,one
      integer ncoll,ncbegcoll,ncendcoll,nincellm,nincellt
      integer Ncand,k1,k2
      real(8) tmp,tempmul,dvx,dvy,dvz,vrel2,vrel
      real(8) phi,cp,sp,ct,st,fa,ft,div
      real(8) vperp,stcp12,stsp12,stsp3,dvfac
      real(8) r1,r2
      real(8) vcm(3),vr(3)
      real(8) vzavg,count
      real ran90


c KLUGE
      vzavg=0.
      count=0.

      if(verbose.eq.1) write(*,*)'  Crosscollider'

      !Restart the counter
      ncoll=0

      !set some variables that will be used for the collider
      div=1/(mar+mtrac)
      fa=mar*div
      ft=mtrac*div
```

```
      do i=ncbegcoll,ncendcoll



c control the random upward growth of vrmf
         vrmXf(i)=0.999*vrmXf(i)

         nincellm=cellistf(1,i)
         nincellt=cellistTf(1,i)

         !collide cells that meet certain conditions
         if(nincellm.ge.1.and.nincellt.ge.1.and.mask(i).ne.mcdelete.and.
     &        Vc(i).ne.0.d0)then

            !Compute the number of collision candidates
            tmp=nincellt*nincellm*psabsminf*sigmaX*sqrt(vrmXf(i))*tau/
     &           (Vc(i))+ collfracXf(i)

            Ncand=int(tmp)
            collfracXf(i)=tmp-Ncand

            !Top of candidate loop
            do n=1,Ncand
               !Randomly select 2 particles in cell: one argon and one Trace
               !Argon
               k1=nincellm*random(randcnt)+1.d0
                 randcnt=randcnt-1
                 if(randcnt==0)randcnt=randmax

               !Trace
               k2=nincellt*random(randcnt)+1
               k2=ran90()*nincellt+1
                 randcnt=randcnt-1
                 if(randcnt==0)randcnt=randmax

               !Convert k1 and k2 into particle indices
               k1=cellistf(k1+1,i)
               k2=cellistTf(k2+1,i)

               !Compute the squared relative veloicty
```

```
dvx=vxpf(k1)-vxpTf(k2)
dvy=vypf(k1)-vypTf(k2)
dvz=vzpf(k1)-vzpTf(k2)

vrel2=dvx**2+dvy**2+dvz**2
!update the maximum relative velocity squared in the this cell
vrmXf(i)=max(vrmXf(i),vrel2)



!Compute a collision if the relative velocity is favorable
! This is acceptance-rejection
r1=random(randcnt)
  randcnt=randcnt-1
  if(randcnt==0)randcnt=randmax

!top of vrelative acceptance rejection if
if(vrel2.gt.vrmXf(i)*real(r1**2))then

   !Compute the center of mass velocity vector
   vcm(1)=fa*vxpf(k1)+ft*vxpTf(k2)
   vcm(2)=fa*vypf(k1)+ft*vypTf(k2)
   vcm(3)=fa*vzpf(k1)+ft*vzpTf(k2)

   !Compute the relative velocity
   vrel=sqrt(vrel2)

   !get the random scattering angle phi, then cos(phi) and sin(phi)
   phi=twopi*real(random(randcnt))
     randcnt=randcnt-1
     if(randcnt==0)randcnt=randmax

   cp=cos(phi)
   sp=sin(phi)

   !get the random cos(theta)and sin(theta)
   ct=2.d0*real(random(randcnt))-1.d0
     randcnt=randcnt-1
     if(randcnt==0)randcnt=randmax
   st=sqrt(1.d0-ct**2)
```

```
                !various factors for computing the relative velocity vector
                !This is subtle - we are using the fact that all particles are
                ! at y=0

                vperp=sqrt(dvx**2+dvy**2)
                stcp12=st*cp*vrel
                stsp12=st*sp*dvz
                stsp3=st*sp
                dvfac=1.d0/vperp

                !New relative velocity vector
                vr(1)=ct*dvx + dvfac*(dvy*stcp12+dvx*stsp12)
                vr(2)=ct*dvy + dvfac*(dvy*stsp12-dvx*stcp12)
                vr(3)=ct*dvz-stsp3*vperp

!*********************************
c     kluge
c                 vr(1)=vrel*cp*st
c                 vr(2)=vrel*sp*st
c                 vr(3)=vrel*ct

!*********************************

                !This is where we enact the trace particle collider. ONLY THE
                ! TRACE ELEMENT IS UPDATED, NOT THE MAIN ELEMENT!!!

                vxpTf(k2)=vcm(1)-fa*vr(1)
                vypTf(k2)=vcm(2)-fa*vr(2)
                vzpTf(k2)=vcm(3)-fa*vr(3)

                vzavg=vzavg+vzpTf(k2)
                count=count+1.




c     kluge: write velocities to file to look at distributions
```

```
c                 write(382,340)vxpTf(k2),vypTf(k2),vzpTf(k2)
 340              format(3(1pe12.4))

                  !count this collision in the back half diagnostic
                  if(zpTf(k2).lt.2e-3)then
                     ncoll=ncoll+1
                  end if

               endif
               !end of acceptance rejection

            end do
            ! end of candidate loop

         end if
         ! bottom of "legal cell" loop

      end do
      ! bottom of loop over cells to process collisions

      !Print the back half collision count diagnostic
      write(*,*)'      # collisions in cross collider:',ncoll

      return
      end
c END CROSSCOLLIDER5
```

Appendix B

Matlab m-files

B.1 fenix9coll

```
%This will be a one cell simulation to test the new collider for
%fenix9. Specifically it will test if the statistics will work for
%calling box-muller to create a single particle candidate or if it
%needs to create a whole distribution of particles for the barium to
%collide with. It also will test whether the small number of argon
%particles in a cell contributes to the horrible temperature
%difference in the simulation.

clear all;clc;close all;
format long e

%first define what type of simulation you want to run.
  %massargon=1: loading the cell with argon particles
  %massargon=2: using box-muller once to create one argon particle for
  % a collision candidate
massargon=0;

%define the temperature for box-muller and density for determining the
% number of collision candidates.
atemp=5400;
adens=1.129e24;
bdens=adens/1000;
```

```
mar=40*1.660538e-27;
kb=1.381e-23;
mbar=2.28e-25;
% mbar=mar;
vthar=sqrt(kb*atemp/mar);
vthtrac=sqrt(kb*atemp/mbar);
vrmX=(3.0*vthar)^2;
sigmaX=3.0e-19;
collfrac=0;



%define the timestep
tau=1.3e-9;

%set some variables that will be used in the collider
div=1/(mar+mbar);
fa=mar*div;
ft=mbar*div;
vcm=zeros(3,1);
vr=zeros(3,1);




%top of loop
num=1;
narsim=100;
nsteps=10000;
equali=10;
ntracsim=20;

temp=zeros(num,1);

for j=1:num



%open the file for writing
fid=fopen('particledata.dat','w');
  %write the header file
  fprintf(fid,'%16.8E %16.8E %16.8E %16.8E %16.8E %16.8E %16.8E\n',...
```

```
  vthtrac,sigmaX,1.4e-3,mbar,atemp,0.0,1.0);

%if it is desired to load the cell with argon particles for the
% collider then define how many you would like to load
narsim=narsim;
fprintf('narsim = %g\n',narsim);

%then define the array that will keep track of the argon information
avels=zeros(3,narsim);
bvels=zeros(3,ntracsim);



%set the collider counter
ncoll=0;
vx=0;
vy=0;
vz=0;
vx2=0;
vy2=0;
vz2=0;
nn=0;

%duplicate counters
dup=zeros(narsim);
duptot=0;
ncollisions=0;

%start the main loop
display('starting the loop')
for nt=1:nsteps;
%  fprintf('nt: %g\n',nt)

vrmX=vrmX*.999;
%start the collider on the particles
if(massargon==1)
for k=1:narsim
vtmp=vthar*sqrt(-2.0*log(1.0-.9999*rand(1,1)));
theta=2.0*pi*.9999*rand(1,1);
avels(1,k)=vtmp*cos(theta);
```

```
avels(2,k)=vtmp*sin(theta);

vtmp=vthar*sqrt(-2.0*log(1.0-.9999*rand(1,1)));
theta=2.0*pi*.9999*rand(1,1);
avels(3,k)=vtmp*cos(theta);

end
end

%compute the number of collision candidates
tmp=ntracsim*adens*sigmaX*sqrt(vrmX)*tau+collfrac;
Ncand=floor(tmp);
collfrac=tmp-Ncand;



%top of the candidate loop
for n=1:Ncand

%pick an argon particle from the cell if massargon=1
if(massargon==1)
k1=floor(narsim*.9999*rand(1,1)+1.0);
vxa=avels(1,k1);
vya=avels(2,k1);
vza=avels(3,k1);
else
%if you want to pick it on the fly, then use box-muller
vtmp=vthar*sqrt(-2.0*log(1.0-.9999*rand(1,1)));
theta=2.0*pi*.9999*rand(1,1);
vxa=vtmp*cos(theta);
vya=vtmp*sin(theta);

vtmp=vthar*sqrt(-2.0*log(1.0-.9999*rand(1,1)));
theta=2.0*pi*.9999*rand(1,1);
vza=vtmp*cos(theta);



end

%pick the barium particle
```

```
k2=floor(ntracsim*.9999*rand(1,1)+1.0);
vxb=bvels(1,k2);
vyb=bvels(2,k2);
vzb=bvels(3,k2);

%compute the squared relative velocity
dvx=vxa-vxb;
dvy=vya-vyb;
dvz=vza-vzb;
vrel2=dvx^2+dvy^2+dvz^2;

%update the maximum relative velocity
vrmX2=max(vrel2,vrmX);
vrmX=vrmX2;

%Acceptance/rejection
if(vrel2>vrmX*.9999^2*rand(1,1)^2)

%make the duplicate count
if(massargon==1)
dup(k1)=dup(k1)+1;
ncollisions=ncollisions+1;
end

%compute the center of mass velocity
vcm(1)=fa*vxa+ft*vxb;
vcm(2)=fa*vya+ft*vyb;
vcm(3)=fa*vza+ft*vzb;

%compute their relative velocity
vrel=sqrt(vrel2);

%get the random scattering angle: phi
phi=2.0*pi*.9999*rand(1,1);
sp=sin(phi);
cp=cos(phi);

%get the random cos(theta), and then sin(theta)
ct=2.0*.9999*rand(1,1)-1.0;
```

```
st=sqrt(1.0-ct^2);



vr(1)=vrel*cp*st;
vr(2)=vrel*sp*st;
vr(3)=vrel*ct;



%update only the trace element

bvels(1,k2)=vcm(1)-fa*vr(1);
bvels(2,k2)=vcm(2)-fa*vr(2);
bvels(3,k2)=vcm(3)-fa*vr(3);



end

end



bvx2=mean(bvels(1,:).^2);
bvy2=mean(bvels(2,:).^2);
bvz2=mean(bvels(3,:).^2);
T(nt)=mbar/kb/3*(bvx2+bvy2+bvz2);
%fprintf(' T = %g \n',T)

if(nt>=equali)



%start sums for the temperature keeper
for i=1:ntracsim
vx=vx+bvels(1,i);
vy=vy+bvels(2,i);
vz=vz+bvels(3,i);
vx2=vx2+bvels(1,i)^2;
vy2=vy2+bvels(2,i)^2;
vz2=vz2+bvels(3,i)^2;
ncoll=ncoll+1;
```

```
%write the velocities to file if past the equilibrium mark
fprintf(fid,'%16.8E %16.8E %16.8E %16.8E %16.8E %16.8E %16.8E\n',...
bvels(1,i),bvels(2,i),bvels(3,i),0.0,0.0,0.0,0.0);

end

if(massargon==1)
%make the duplicate count
duptot=duptot+sum(nonzeros(dup)-1);
dup(:)=0;
end

end



end

fprintf('   number of collisions: %g\n',ncollisions)
fprintf('   number of duplicates: %g\n',duptot)
fprintf('   collision frac: %g\n',duptot/ncollisions)
display(' ')
fprintf('   number of entries: %g\n',ncoll);
vx=vx/ncoll;
vy=vy/ncoll;
vz=vz/ncoll;
vx2=vx2/ncoll;
vy2=vy2/ncoll;
vz2=vz2/ncoll;
temp(j)=mbar/3/kb*(vx2+vy2+vz2-vx^2-vy^2-vz^2);
fprintf('   barium temperature: %g\n',temp(j))
display(' ')

end

% plot(temp)
% vx=0;
% vy=0;
% vz=0;
% vx2=0;
```

```
% vy2=0;
% vz2=0;
% for k=1:narsim
%  vx=vx+avels(1,k);
%  vy=vy+avels(2,k);
%  vz=vz+avels(3,k);
%  vx2=vx2+avels(1,k)^2;
%  vy2=vy2+avels(2,k)^2;
%  vz2=vz2+avels(3,k)^2;
% end
% vx=vx/narsim;
% vy=vy/narsim;
% vz=vz/narsim;
% vx2=vx2/narsim;
% vy2=vy2/narsim;
% vz2=vz2/narsim;
% temp=mar/3/kb*(vx2+vy2+vz2-vx^2-vy^2-vz^2);
% fprintf('argon temperature: %g\n',temp)

display('all done')
% fclose(fid);
```

B.2 maxwell

This is a the maxwellian fitter.

```
close all;

% This is going to be my maxwell solver
loadnow=input('Do you want to load the datafiles? (1-yes and 0 -no): ')
if loadnow==1;
clear all
load particledata.dat
end
format long e
```

```
%initializations of different variables
conts=particledata(1,:);
%variables for barium loading
sigma=conts(2);
tracwidth=conts(3);
particleskip=conts(7);
%variables for the desired maxwellian
vth=conts(1);
m=conts(4);
T=conts(5);
kb=1.381e-23;
% T=3000;
% vth=sqrt((kb*T)/m);
% m=6.68e-26;
% vth=1057;
vmax=10*vth;
bins=100;
dv=vmax/bins;
v=0:dv:vmax;

%This is the real distribution of my particles
% mxwl=2e-9*v.^2.*exp(-2e-6*v.^2/2);
mxwl=(m/(2*pi*kb*T))^(3/2)*4*pi*v.^2.*exp(-m*v.^2/(2*kb*T));

%plot(v,mxwl,'r-','linewidth',2);

[a,b]=size(particledata);
vxp=particledata(2:a,1);
vyp=particledata(2:a,2);
vzp=particledata(2:a,3);
xp=particledata(2:a,4);
zp=particledata(2:a,6);
ntracsim2=particledata(2:a,7);
ntracsim=nonzeros(ntracsim2);
%set it to the number of particles in the collision loop
ntracsim=input('enter the number of entries: ');
fprintf('you have %g blocks to go through\n',length(ntracsim));
k=0;
nn=0;
```

```
epsilon=m/(kb*T)
A1=(m/(2*pi*kb*T))^(3/2)*4*pi
averagefreq=1;
count=0;
Vhtot=zeros(1,bins);
Vxtot=zeros(1,bins);
Vytot=zeros(1,bins);
Vztot=zeros(1,bins);
Patot=v.*0;

%Hard code the first guess
% A1=2e-9;
% epsilon=2e-6;
vxm=0;
vym=0;
vzm=0;
for i=1:length(ntracsim);
Vxhist=zeros(ntracsim(i),1);
Vyhist=zeros(ntracsim(i),1);
Vzhist=zeros(ntracsim(i),1);
i
count=count+1;
    for q=1:ntracsim(i);
        nn=nn+1;
k=k+1;
Vxhist(k)=vxp(nn);
Vyhist(k)=vyp(nn);
Vzhist(k)=vzp(nn);
end
Vhist=sqrt(Vxhist.^2+Vyhist.^2+Vzhist.^2);
%Compute means for diagnostics
%  fprintf('<x> = %g\n',mean(Vxhist))
%  fprintf('<y> = %g\n',mean(Vyhist))
%  fprintf('<z> = %g\n',mean(Vzhist))
vxm=vxm+mean(Vxhist);
vym=vym+mean(Vyhist);
vzm=vzm+mean(Vzhist);
```

```
%set up the edges needed for histograms
if(i==1)
de=(vmax)/(bins-1);
edges=0:de:vmax;
edges2=-vmax:2*de:vmax;
end

%Create the histograms
Vx=hist(Vxhist,edges2);
Vy=hist(Vyhist,edges2);
Vz=hist(Vzhist,edges2);
Vh=hist(Vhist,edges);
%normalize Vh,Vx,Vy,Vz
A=trapz(edges,Vh);
Vh=Vh./A;
A=trapz(edges2,Vx);
Vx=Vx./A;
A=trapz(edges2,Vy);
Vy=Vy./A;
A=trapz(edges2,Vz);
Vz=Vz./A;

%this does the approximation on individual time steps
%  x=zeros(2,1);
%  x(1)=A1;
%  x(2)=epsilon;
%  [epsilon,A1]=newtn(edges,x,Vh);
%  Pa=A1*v.^2.*exp(-epsilon.*v.^2/2);

%this is the time average
Vhtot=Vhtot+Vh;
Vxtot=Vxtot+Vx;
Vytot=Vytot+Vy;
Vztot=Vztot+Vz;
%  Patot=Patot+Pa;

if count==averagefreq
Vhav=Vhtot./averagefreq;
Vxhav=Vxtot./averagefreq;
```

```
Vyhav=Vytot./averagefreq;
Vzhav=Vztot./averagefreq;
%  Paav=Patot./averagefreq;

%this does the approximation in side the averaging
x=zeros(2,1);
x(1)=A1;
x(2)=epsilon;
[epsilon,A1]=newtn(edges,x,Vhav);
Paav=A1*v.^2.*exp(-epsilon.*v.^2/2);

%calculate the effective temperature
Teff=m/(2*pi*kb)*(A1/(4*pi))^(-2/3);

%compute the averages
vxm=vxm/averagefreq;
vym=vym/averagefreq;
vzm=vzm/averagefreq;

%CREATE THE SUBPLOTS
% Create the speed plot
%  subplot(2,2,1)
bar(edges,Vhav);
axis([0 3000 0 2e-3])
hold on
plot(v,mxwl,'r-','linewidth',2);
plot(v,Paav,'g-','linewidth',2);

%print the effective temperature on the figure
s=sprintf('Effective Temperature = %g',Teff);
text(1500,1e-3,s);

t=sprintf('Cross-Colliding Maxwell-Boltzmann distribution, Timestep %g',(i)*particleskip);
title(t);
legend('Time Averaged Histogram','Maxwell-Boltzmann','Time Averaged Fit')
xlabel('Speed')
ylabel('Probability Density')
hold off
```

```
%  %create the vx plot
%  subplot(2,2,2)
%  bar(edges2,Vxhav);
%  s=sprintf('Averaged Histogram of Vxp, <vxp> = %g',vxm);
%  title(s)
%  axis([-4500 4000 0 7e-4])
%
%  %create the vy plot
%  subplot(2,2,3)
%  bar(edges2,Vyhav);
%  s=sprintf('Averaged Histogram of Vyp, <vyp> = %g',vym);
%  title(s)
%  axis([-4500 4000 0 7e-4])
%
%  %create the vz plot
%  subplot(2,2,4)
%  bar(edges2,Vzhav);
%  s=sprintf('Averaged Histogram of Vzp, <vzp> = %g',vzm);
%  title(s)
%  axis([-4500 4000 0 7e-4])

count=0;
Vhtot(:)=0;
Vxtot(:)=0;
Vytot(:)=0;
Vztot(:)=0;
Patot(:)=0;
% pause
end
% break
    clear Vxhist Vyhist Vzhist Vhist
k=0;
% break
end
```

This is the Newton solver that employs Newton's method needed for MAXWELL.

```
% This is going to be my maxwell solver for a maxwellian distribution
function [epsilon,A]=newtn(v,guess,Vh)
```

```
n=length(v);
x=guess;
format long e
y=f(x,n,v,Vh);
while (abs(y(1))>1e-5 || abs(y(2))>1e-5);
x1=x-Df(x,n,v,Vh)\f(x,n,v,Vh);
x=x1;
y=f(x,n,v,Vh);
end
epsilon=x(2);
A=x(1);
return

%these are my functions that solve the derivative and the normal
function out=f(x,n,v,Vh)
sum1=0;
sum2=0;
out=zeros(2,1);
for i=1:n;
sum1=sum1+x(1)*v(i)^4*exp(-x(2)*v(i)^2)-Vh(i)*v(i)^2*exp(-x(2)*v(i)^2/2);
sum2=sum2+Vh(i)*x(1)*v(i)^4*exp(-x(2)*v(i)^2/2)-x(1)^2*v(i)^6*exp(-x(2)*v(i)^2);
end
out(1)=2*sum1;
out(2)=sum2;
return

function out=Df(x,n,v,Vh)
sum1=0;
sum2=0;
sum3=0;
sum4=0;
out=zeros(2,2);
for i=1:n;
sum1=sum1+2*v(i)^4*exp(-x(2)*v(i)^2);
sum2=sum2+Vh(i)*v(i)^4*exp(-x(2)*v(i)^2/2)-2*x(1)*v(i)^6*exp(-x(2)*v(i)^2);
sum3=sum3+Vh(i)*v(i)^4*exp(-x(2)*v(i)^2/2)-2*x(1)*v(i)^6*exp(-x(2)*v(i)^2);
sum4=sum4+x(1)^2*v(i)^8*exp(-x(2)*v(i)^2)-.5*Vh(i)*x(1)*v(i)^6*exp(-x(2)*v(i)^2/2);
end
out(1,1)=sum1;
```

```
out(1,2)=sum2;
out(2,1)=sum3;
out(2,2)=sum4;
return
```

Index