

1994

Molecular dynamic simulation of Couette flow : a comparison to semi-analytic modeling

Gregory L. Kolte
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_theses

Recommended Citation

Kolte, Gregory L., "Molecular dynamic simulation of Couette flow : a comparison to semi-analytic modeling" (1994). *Master's Theses*. 925.

DOI: <https://doi.org/10.31979/etd.w5q4-e942>

https://scholarworks.sjsu.edu/etd_theses/925

This Thesis is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Theses by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

**MOLECULAR DYNAMIC SIMULATION OF COUETTE FLOW
A COMPARISON TO SEMI-ANALYTIC MODELING**

A Thesis

Presented to

The Faculty of the Department of Physics

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Gregory L. Kolte

December, 1994

UMI Number: 1361183

UMI Microform 1361183

Copyright 1995, by UMI Company. All rights reserved.

**This microform edition is protected against unauthorized
copying under Title 17, United States Code.**

UMI

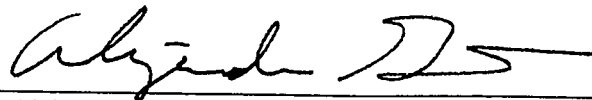
**300 North Zeeb Road
Ann Arbor, MI 48103**

© 1994

Gregory L. Kolte

ALL RIGHTS RESERVED

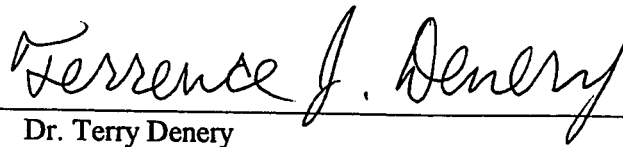
APPROVED FOR THE DEPARTMENT OF PHYSICS



Dr. Alejandro Garcia



Dr. Patrick Hamill



Dr. Terry Denery

APPROVED FOR THE UNIVERSITY



ABSTRACT

MOLECULAR DYNAMIC SIMULATION OF COUETTE FLOW A COMPARISON TO SEMI-ANALYTIC MODELING

by Gregory L. Kolte

A molecular dynamics (MD) simulation computer program was written to model general planar Couette flow. A comparison of MD simulation results to results produced by Denery's semi-analytical model was performed for Couette flow scenarios in the Knudsen number range 0.1 to 1.0. Because Denery's theory is for inverse-power law type potentials, MD simulation provides the most general means for testing it numerically. Comparison of predicted fluid properties including density, velocity, and temperature showed relatively close agreement between the Denery and MD models in the 0.1 Knudsen number regime. In the 1.0 Knudsen number regime, more significant differences between the two respective models were observed (particularly at the thermal boundaries). In a comparison between his model and Bird's Direct Simulation Monte Carlo (DSMC) model for hard spheres, Denery reported similar qualitative differences to those discovered in this study.

TABLE OF CONTENTS

	Page
INTRODUCTION	1
1 MOLECULAR DYNAMICS	4
1.1 INTRODUCTION	4
1.2 EQUATIONS OF MOTION.....	4
1.3 EFFECTIVE PAIR POTENTIALS.....	8
1.3.1 <i>Lennard-Jones Potential</i>	9
1.3.2 <i>Effective Potential For Denery Comparison</i>	10
1.3.3 <i>Cutoff Radius</i>	11
1.4 FLOW MODELS	12
1.4.1 <i>Planar Couette Flow</i>	13
1.4.2 <i>Planar Poiseuille Flow</i>	14
1.4.3 <i>Knudsen Number</i>	15
1.5 BOUNDARY CONDITIONS.....	15
1.5.1 <i>Periodic Boundaries</i>	16
1.5.2 <i>Thermal Boundaries</i>	17
2 THE MD CHANNEL COMPUTER MODEL	19
2.1 INTRODUCTION	19
2.2 ALGORITHM DESIGN AND IMPLEMENTATION	20
2.2.1 <i>Overview</i>	21
2.2.2 <i>Model Parameters</i>	23
2.2.3 <i>Initialization</i>	25
2.2.4 <i>Main Loop and Calculation of Net Force on all Particles</i>	30
2.2.5 <i>Update Particle Positions and Velocities</i>	39
2.2.6 <i>Impose Boundary Conditions</i>	40
2.2.7 <i>Calculate Conserved Quantities</i>	42
2.2.8 <i>Output Steady State Channel Averages</i>	43
2.3 MODEL VALIDATION USING L-J POTENTIAL.....	44
2.3.1 <i>Code Stabilization and Conservation</i>	45
2.3.2 <i>Comparison to the Morris Results</i>	49
3 DENERY RESEARCH	54
3.1 INTRODUCTION	54
3.2 THE DENERY MODEL	54
3.3 DENERY POTENTIAL AND PARAMETERS FOR THE MDC MODEL.....	61
3.4 COMPARISON OF MDC TO DENERY MODEL	64
SUMMARY	76

TABLE OF CONTENTS

	Page
APPENDIX A	A-1
APPENDIX B	B-1
APPENDIX C	C-1
APPENDIX D	D-1

Introduction

The principal motivation for this effort was inspired by the recent work of Dr. Terry Denery at Stanford University.¹ In Denery's work he developed a semi-analytical solution to the general Couette flow problem. By extending the previous work of Lees and Liu² to allow a general inverse power law form for the inter-particle potential, Denery developed an improved model for steady state Couette flows involving noble gases like argon. Although Denery made several approximations in developing his model, his results were shown to be quite good in Knudsen number regimes below 0.1. Denery's model is an attractive alternative to molecular dynamic (MD) type simulations because of the significant savings in the calculation time afforded by the method. A typical Denery numerical model of the type used in this effort required several minutes of computer CPU time to run in contrast to the many hours required by the MD simulations.

An MD simulation program was developed and used to test the Denery theory for Knudsen numbers of 0.1 and 1.0. For this situation, MD simulation provides the most general means by which to check Denery's theory because his model assumes an inverse power law type potential which cannot be solved analytically. Although MD simulation remains a premiere technique for calculating general fluid flows, a drawback to the MD

¹ Denery, T., Ph.D. Thesis, Stanford University, Oct. (1994).

² Liu, C., and Lees, L., "Kinetic Theory Description of Plane Compressible Couette Flow", *Advances in Applied Mechanics, Supplement 1, Rarefied Gas Dynamics*, Academic, (1961).

approach is the lengthy computation time required for systems of a few thousand particles to reach steady state flow. As an example, some of the MD simulations for this work required over ten of hours of CPU time on a SPARC10 workstation to complete the 100,000 time steps necessary for systems of 4096 particles to reach steady state configurations.

Given the computation time required by the respective models, analytical or semi-analytical models, such as Denery's, represent a significant computational improvement over MD simulations if valid results are produced. This research effort investigates the accuracy of the Denery model for Knudsen numbers beyond the principal range for which it was developed.

The presentation of our study is organized into three chapters. Chapter One is devoted to a general discussion of MD where we define and discuss the fundamental concepts and principles of MD. This provides the necessary framework for a detailed discussion of the development and validation of the MD Channel (MDC) computer simulation program which is presented in Chapter Two.[†] Chapter Three presents an overview of Denery's research and his semi-analytical model. A comparison of Denery and MDC model predictions in the 0.1 to 1.0 Knudsen number regime is provided as the last section of Chapter Three and is followed by a summary and general discussion of the results.

[†] Some of the sub-sections of section 2.2, which discuss the MDC algorithm implementation and design in rather close detail, can be skimmed or skipped by the reader who is not particularly interested in the algorithm details.

Four appendices, A-D, provide supplemental information. Appendix A contains information regarding the initialization of the MDC simulation program. Appendices B and C provide archives of all the comparison plots between the MDC model and the Morris and Denery models respectively. A complete computer listing of the MDC simulation code is provided in Appendix D.

1 Molecular Dynamics

1.1 Introduction

Molecular dynamics (MD) is the term given to the process of solving the classical or Newtonian equations of motion for a collection of molecules. Advances in computer technology made these types of calculations numerically feasible by the late 1950s. Computer simulations have made it possible to compute essentially exact results in MD problems in regimes where analytical methods provide only approximate solutions or perhaps no solution at all. Computer simulations provide a bridge between the microscopic description of the physical interaction between individual molecules and the macroscopic properties of substances (or materials) which can be measured experimentally. Additionally, computer simulations provide a practical means by which critical environments, where experimental measurement is impossible due to the severe environmental conditions such as high temperatures, extreme pressures, etc., can be studied.

1.2 Equations of Motion

Given a system of N atoms, the potential energy of the system can be represented as the sum of individual terms:

$$\mathcal{V}(\vec{r}) = \sum_{i=1}^N V_1(\vec{r}_i) + \sum_{i=1}^N \sum_{(j>i)}^N V_2(\vec{r}_i, \vec{r}_j) + \sum_{i=1}^N \sum_{(j>i)}^N \sum_{(k>j>i)}^N V_3(\vec{r}_i, \vec{r}_j, \vec{r}_k) + \dots$$

Equation 1-1

The first term represents the potential energy due to an external field, which might include the bounding walls. The details of this term are defined by the flow model of the system. The two different scenarios that were used for this study will be presented later in this chapter in the *Flow Models* section. The second term is the most important term and represents the pair potential between particles. In general this potential depends only on the separation $r_{ij} = |\vec{r}_i - \vec{r}_j|$ between particles. The third term represents the potential energy due to the interaction of triplets and is responsible for up to 10% of the lattice energy in argon at liquid densities.³ Contributions from fourth and higher order terms are negligible compared to the first three terms and therefore they are rarely computed in the MD model. Direct calculation of the third term, which would involve triple sums over all N particles, is computationally impractical in terms of the number of operations required. Unfortunately, the contribution from the third term is too significant to neglect. A practical solution to this circumstance is to define an *effective pair potential* which incorporates the second and all higher order effects into one second order term. In this way the effective pair potential, while involving only one sum over all particle pairs, approximates the combined contributions from all second and higher order terms. The

³ Allen, M. P., and Tildesley, D. J., *Computer Simulation of Liquids*, Clarendon Press, Oxford, (1987), Chap. 3.

effective pair potentials used for this effort will be presented later in this chapter in the section, *Effective Pair Potentials*.

Once we obtain a suitable expression for the effective pair potential of the form $V(r)$, where r is the radial distance between any two individual particles, we can calculate the force due to one particle acting on another from the expression:

$$\vec{F} = m\ddot{\vec{r}} = -\vec{\nabla} \cdot V(r) .$$

Equation 1-2

One can simulate fluid dynamics by integrating the equations of motion from this force expression. This is performed numerically by employing a finite difference scheme. The Verlet finite difference algorithm is often employed for this purpose.⁴ Several features that make the Verlet a popular choice include a compact, easy to program form, complete time reversibility, and excellent energy conserving properties. The method can be derived from the forward and backward Taylor expansions about $\vec{r}(t)$:

$$\vec{r}(t + \Delta t) = \vec{r}(t) + \Delta t \cdot \dot{\vec{r}}(t) + \frac{(\Delta t)^2}{2} \cdot \ddot{\vec{r}}(t) + \dots ,$$

Equation 1-3

and,

⁴ Verlet, L., "Computer 'experiments' on classical fluids, I. Thermodynamical properties of Lennard-Jones molecules", *Phys. Rev*, **165**, 201-14, (1967).

$$\vec{r}(t - \Delta t) = \vec{r}(t) - \Delta t \cdot \dot{\vec{r}}(t) + \frac{(\Delta t)^2}{2} \cdot \ddot{\vec{r}}(t) - \dots ,$$

Equation 1-4

respectively. Adding *Equation 1-3* and *Equation 1-4* together and rearranging the terms produces the Verlet scheme for calculating new particle positions given by:

$$\vec{r}(t + \Delta t) = 2 \cdot \vec{r}(t) - \vec{r}(t - \Delta t) + (\Delta t)^2 \cdot \ddot{\vec{r}}(t) + O[(\Delta t)^4].$$

Equation 1-5

The Verlet method for advancing particles, as given by *Equation 1-5*, has several interesting features. The truncation error for this method is on the order of $(\Delta t)^4$. It is also observed that explicit calculation of the velocity is not necessary to advance the particles for a given time step. Another feature of this scheme is time symmetry due to the presence of the $\vec{r}(t + \Delta t)$ term and the $\vec{r}(t - \Delta t)$ term. This time symmetry ensures that the method is completely reversible in time to within round-off error.

Although an explicit calculation of the particle velocities is not needed to calculate the trajectories of the particles on each time step, the particle velocities are necessary in the calculation of the total kinetic energy and therefore are required to calculate the total energy of the system. Calculation of the total system energy and the constraint that it remain constant during a computer simulation provides a way of making sure the numeric scheme is stable. A centered differences velocity scheme is derived by subtracting

Equation 1-4 from *Equation 1-3*, dividing by the result by two, and rearranging terms to get:

$$\vec{v}(t) = \dot{\vec{r}}(t) = \frac{\vec{r}(t + \Delta t) - \vec{r}(t - \Delta t)}{2 \cdot \Delta t} + O[(\Delta t)^2].$$

Equation 1-6

The truncation error for the centered differences velocity scheme is seen in *Equation 1-6* to be on the order of $(\Delta t)^2$. Other more accurate techniques for calculating the particle velocities are available but are not necessary since the intention is to use the velocity calculations solely for the purpose of monitoring the total energy of the system.

1.3 Effective Pair Potentials

As mentioned in the previous section, the second and higher order terms from *Equation 1-1* are usually represented in MD simulations by a single two-body term that attempts to reasonably approximate all of the many-body contributions to the total potential energy of the system. *Equation 1-7* shows the typical form of the potential in an MD simulation:

$$\mathcal{V}(\vec{r}) \approx \sum_{i=1}^N V_1(\vec{r}_i) + \sum_{i=1}^N \sum_{(j>i)}^N V_2^{\text{eff}}(\vec{r}_i, \vec{r}_j).$$

Equation 1-7

In *Equation 1-7* the expression $V_2^{\text{eff}}(\vec{r}_i, \vec{r}_j)$ is the *effective pair potential*. It is typically defined as some scalar potential function $V(r)$ that depends only on the radial separation r

between any two particles. The actual function $V(r)$ that is used in the MD simulation is determined by the physical properties of the system that is being modeled. For this research effort, two different scalar effective pair potential models were employed. Both of these models were *inverse power* models and each is described below.

1.3.1 Lennard-Jones Potential

One of the most common effective pair potentials used for computer simulations is the Lennard-Jones 6-12 scalar potential:

$$V_{LJ}(r) = 4\epsilon \left(\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^6 \right),$$

Equation 1-8

where r represents the scalar radial distance between any pairs $r_{ij} = |\vec{r}_i - \vec{r}_j|$.

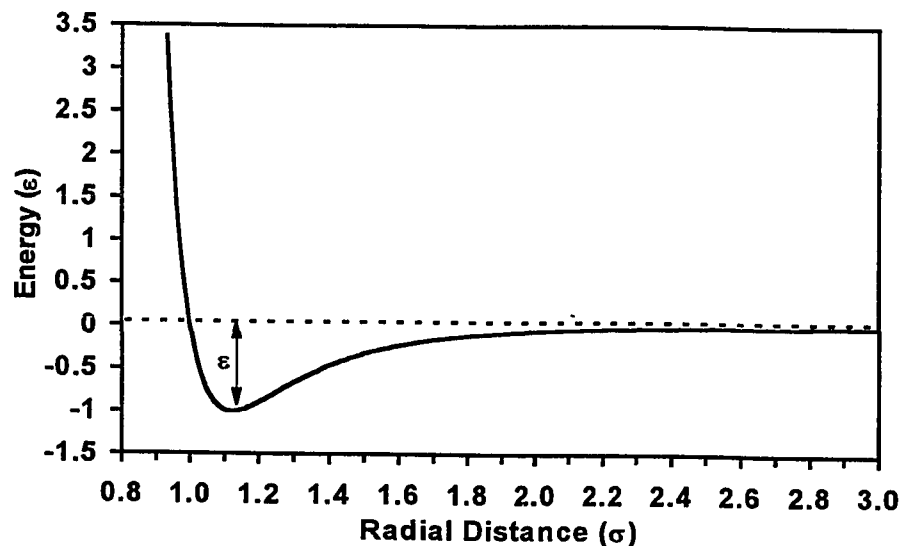


Figure 1-1: The Lennard-Jones Potential function.

A plot of this potential is shown in *Figure 1-1*. The Lennard-Jones potential function features a weak attractive tail that goes as $-(1/r^6)$ at larger radii and a strong repulsive force at radii less than σ . The weak attractive force present at radii greater than σ is indicative of an induced dipole force that is often encountered in neutral fluids while the short range repulsive force is a consequence of Pauli exclusion and the overlap between neighboring electron clouds. The parameter ε is observed to represent the depth of the potential well in *Figure 1-1*. It is found that the Lennard-Jones potential function adequately depicts the experimental properties of liquid argon when values of $\sigma = 3.405 \times 10^{-8}$ cm and $\varepsilon = (119.8 k)$ are used in the expression. In the above notation the energy ε is expressed as a temperature and $k = 1.38066 \times 10^{-16}$ ergs/K is the Boltzmann constant. The Lennard-Jones potential function was used in this research effort to validate and benchmark our simulation code against a similar code developed in a previous research effort by D. Morris.

1.3.2 Effective Potential For Denery Comparison

The expression for the effective pair potential employed for this work's comparison to the Denery model was chosen to have the following inverse power form:

$$V(r) = \frac{\beta}{r^9},$$

Equation 1-9

where $\beta = 1.76 \times 10^{-81} \text{ erg} \cdot \text{cm}^9$. Our choice of the *ninth* inverse power in the above potential function represents a special case of the *general* inverse power law potential function used by Denery. The derivation of this effective pair potential from parameters used by Denery will be presented in Chapter Three. It is observed that this potential function contains a single repulsive term and does not contain any attractive terms.

1.3.3 Cutoff Radius

As can be seen from the second term in *Equation 1-1*, N^2 calculations are required to calculate the net force acting on each particle in a system of N particles. This number of calculations is practical only if the number of particles in the system is small. As the number of particles increases, alternative methods to the *brute force* approach must be devised in order to reduce the number of calculations required for determining the net forces. One approach to addressing this problem is to define a *cutoff* radius for the potential function. The cutoff radius r_c is defined as the radius at which the inter-particle force acting between any particle pair is effectively zero. Examination of the plot of the Lennard-Jones potential function, which is presented in *Figure 1-1*, shows that as the separation between particles r increases, the slope of the potential function, and hence the inter-particle force, approaches zero. An appropriate value of r_c for the Lennard-Jones potential function is 3σ .

The number of calculations required to compute the net force contribution from particle pairs is greatly reduced by using a cutoff radius. Only those particle pairs within

the range defined by the cutoff radius are considered in the net force calculations. All calculations involving particle pairs outside of this range can be ignored. For example, the net force acting on a particle located at position A in *Figure 1-2* is calculated by considering *only* those particles located inside of the cutoff radius r_c at positions 1, 2, and 3 respectively. The MDC simulation program presented in the next chapter uses a cutoff radius to reduce computation time. The details of how the MDC model makes use of a cutoff radius will be presented in the next chapter.

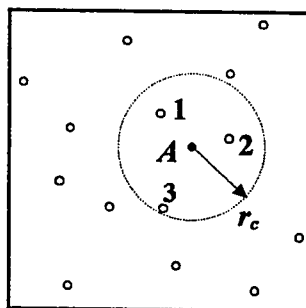


Figure 1-2: Particles located at 1, 2 and 3 are within cutoff radius of a particle located at A.

1.4 Flow Models

The previous section considered the coupling of individual particles with one another. This section considers external forces that influence the individual particle trajectories. The *flow model* defines all of the external forces and boundary conditions present in a MD simulation. Two different flow models were used in this research effort. The primary model used was the planar Couette flow model. The Couette flow model was used in the effort to validate the MDC simulation code as well as in the comparison

of the MDC model to the Denery model. The other flow model used was the planar Poiseuille flow model. The planar Poiseuille flow model was used exclusively in the validation phase of this research effort.

1.4.1 Planar Couette Flow

In planar Couette flow, a fluid is confined between two parallel walls that move in opposite parallel directions relative to one another. The walls are a fixed distance apart and may have different temperatures. *Figure 1-3* is a schematic representation of planar Couette flow.

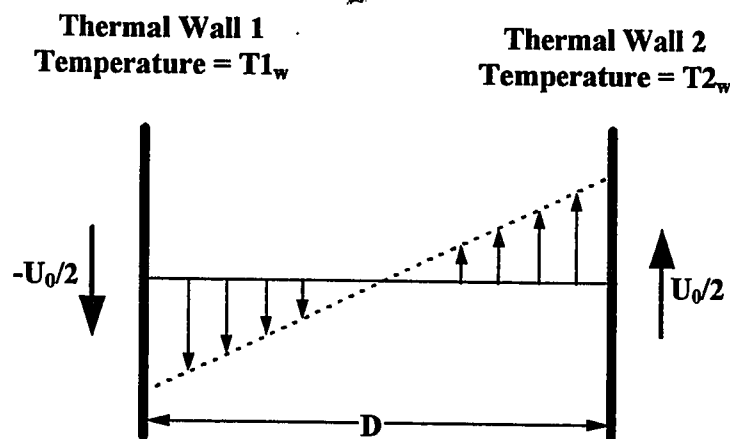


Figure 1-3: Planar Couette flow.

In this figure two thermal walls with temperatures T_{1_w} and T_{2_w} respectively are separated by a distance D . The left wall moves downward with constant velocity $U_0/2$ and the right wall moves upward with constant velocity $U_0/2$. The absolute difference in the relative wall speeds, U_0 , is often expressed as a dimensionless quantity, called *Mach number*, which is defined as the ratio of U_0 to the sound speed in the region between the thermal

walls. The dashed line inside the wall represents a typical steady state average particle velocity profile for cases where the thermal walls are at the same temperature and the relative wall speeds are given by low Mach numbers. Under these conditions, the steady state velocity profile is linear. Near the walls the average particle velocity approaches the respective wall velocity. The magnitude of the average particle velocity decreases with increasing distance from the respective wall until an equilibrium position is reached where the average particle velocity is zero. The location between the walls where the average particle velocity is zero is dependent on the relative temperatures of the thermal walls.

1.4.2 Planar Poiseuille Flow

In planar Poiseuille flow, a fluid is confined between two stationary parallel walls with temperatures $T1_w$ and $T2_w$ respectively. The fluid is subjected to an external potential that results in a force acting on the particles in the direction parallel to the walls. The planar Poiseuille model is illustrated in *Figure 1-4*.

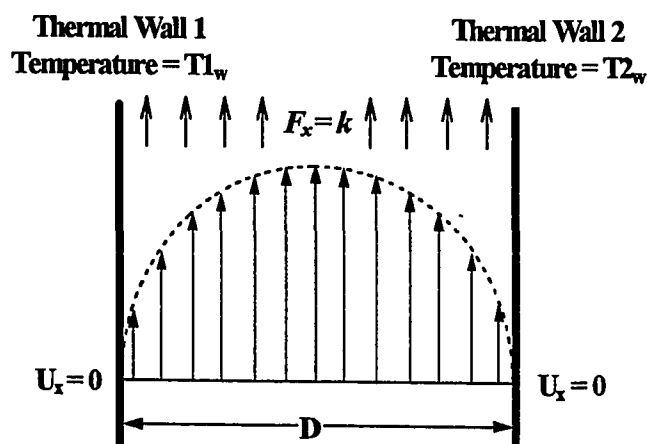


Figure 1-4: Planar Poiseuille flow.

The dashed arc between the walls in *Figure 1-4* is representative of the steady state average particle velocity in the fluid. Whereas the steady state velocity profile for planar Couette flow is linear, the steady state velocity profile for planar Poiseuille flow is quadratic. Near the walls the average particle velocity approaches the respective wall velocity, which in this case is zero. The average particle velocity increases with increasing distance from the respective wall until a maximum is reached somewhere between the two walls. The location of this maximum depends on the relative temperatures of the two stationary thermal walls.

1.4.3 Knudsen Number

The Knudsen number is a dimensionless quantity defined by:

$$K_n = \lambda/D,$$

Equation 1-10

where λ is the mean free path and D is some characteristic length of the system boundaries. In the case of the planar Couette and the planar Poiseuille flow models, this characteristic length is the distance D between the thermal walls.

1.5 Boundary Conditions

Two different types of boundary conditions are required to model planar Couette and planar Poiseuille flow. In one direction, a thermal boundary model is required. In the other two directions, a periodic boundary model is needed. A general description of each

of these models is a prerequisite to the detailed discussions of model implementation provided in the next chapter.

1.5.1 Periodic Boundaries

Three dimensional planar flow models require that the motion of the particles be uninfluenced by boundaries in two directions. For the planar flow models defined in this study, these two directions are chosen to be along the y and z axes. Although the physical system described by planar flow is effectively infinite in the y and z directions, practicality demands that the particles in a simulation be confined to a finite region of space in all directions. Periodic boundaries are the mechanism by which the properties of infinite boundaries can be simulated within a finite region. When a particle crosses a periodic boundary, like particle A crossing the z_{lower} boundary in *Figure 1-5*, it is treated as if it originated from the position of virtual particle A' which is a translation from the position of particle A in the z coordinate by an amount equal to the distance separating the z_{lower} and the z_{upper} boundaries.

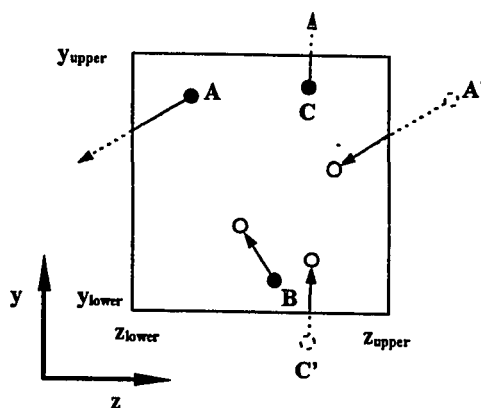


Figure 1-5: Illustration of periodic boundary conditions in two dimensions.

Particle C , in *Figure 1-5*, crosses the y_{upper} periodic boundary and is reentered into the system at the y_{lower} boundary in the proximity of particle B . When calculating the forces on particles near periodic boundaries, such as particles B and C in *Figure 1-5*, the position of particle C is considered to be at the position of virtual particle C' relative to particle B . In this example, the position of virtual particle C' is a translation from the position of particle C in the y coordinate by an amount equal to the distance separating the y_{lower} and the y_{upper} boundaries. This treatment prevents the possibility that particle C reenters the system at a position that is physically closer to particle B than would be normally encountered by the pair.

1.5.2 Thermal Boundaries

The situation at a thermal boundary is treated somewhat differently than that at a periodic boundary. When a particle crosses a thermal boundary during a given time step, it is returned to the system at its point of exit at the boundary, as shown in *Figure 1-6*, and given a new random velocity based on the *biased* Maxwell-Boltzmann distribution.⁵

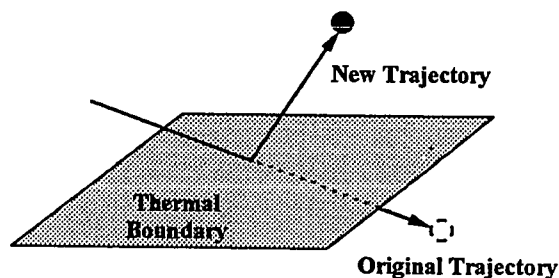


Figure 1-6: Particle crossing a thermal boundary.

⁵ Garcia, A., *Numerical Methods for Physics*, Prentice Hall, Englewood Cliffs, (1994), Chap. 10.

If the thermal boundary has a temperature T_w , the biased Maxwell-Boltzmann particle velocities for a particle of mass m leaving the thermal wall are given by:

$$v_x = \pm \sqrt{\frac{-2k T_w \ln(\mathcal{R}_1)}{m}},$$

Equation 1-11

$$v_y = \sqrt{\frac{-2k T_w \ln(\mathcal{R}_2)}{m}} \cdot \sin(\mathcal{R}_3),$$

Equation 1-12

and,

$$v_z = \sqrt{\frac{-2k T_w \ln(\mathcal{R}_2)}{m}} \cdot \cos(\mathcal{R}_3),$$

Equation 1-13

where \mathcal{R}_1 , \mathcal{R}_2 , and \mathcal{R}_3 are uniform random deviates in the range (0, 1] and k is the Boltzmann constant. The sign on the velocity component that is perpendicular to the thermal wall, which for this study is the x component, is set such that the velocity vector points into the fluid channel.

2 *The MD Channel Computer Model*

2.1 Introduction

A 3-D molecular dynamics simulation computer program was developed during the course of this research effort. This simulation program is henceforth referred to as the *MD Channel* model or *MDC* model. The MDC model calculates and outputs the steady state fluid properties including average particle density, velocity, and kinetic energy for spatial zones that collectively make up a *channel* region separating two thermal walls. Several objectives helped to shape the design of this program. The first was that the program design should meet the primary goal of the research effort, to model planar Couette flow for a dilute gas in the Knudsen number range 0.1 to 1.0. Second, the program should be flexible enough to model planar Poiseuille flow. The latter capability was needed to benchmark and validate the MDC code against MD results produced by Morris in a study where he examined *velocity slip* in dilute gases.⁶ The code should be flexible enough to allow either thermal or periodic boundaries. Modeling thermodynamic equilibrium using periodic boundaries in all directions provides an additional means by which the MDC model can be validated. The model should have the capability to terminate the simulation at any specified time step and save all of the information necessary to restart the simulation where it left off. This feature provides the capability to

⁶Morris, D., San Jose State University Physics TR92-2 (1992).

check interim results and produce a series of contiguous runs. Finally, the algorithms should be designed to execute as quickly as possible. With hundreds of thousands of time steps required for systems to reach steady state configurations, algorithm efficiency and speed are important considerations. The following section, *Algorithm Design and Implementation*, provides the details of how the above considerations were addressed in the design and development of the MDC simulation program.

The second half of this chapter, *Model Validation Using L-J Potential*, outlines the effort to validate the MDC model using the Lennard-Jones effective pair potential. In that section, steady state profiles of particle velocity, temperature, and density that were calculated using both periodic and planar flow models are presented. These profiles are compared to expected theoretical results as well as results obtained by Morris in his research effort.

2.2 Algorithm Design and Implementation

This section describes in detail the MDC simulation model. After presenting an overview of the code's basic structure and approach in performing MD simulation, each individual section of the model will be discussed with attention paid to algorithm philosophy and design.

2.2.1 Overview

The basic functionality of the MDC program is shown schematically in *Figure 2-1* and summarized in 9 steps:

- 1) Define the flow model parameters and other important model parameters including the number of particles in the system N and the total number of time steps in the calculation N_{max} .
- 2) Initialize the program, including the positions of N particles and their velocities.
- 3) Calculate the net force acting on each individual particle.
- 4) Use the net force to calculate new particle positions and their respective velocities over a time interval Δt .
- 5) Check new particle positions with respect to the boundaries and reset individual particle positions and velocities if required by the boundary conditions of the flow model.
- 6) Calculate and tabulate the diagnostic information including the total linear momenta in each direction, the total potential energy, and the total kinetic energy.
- 7) Save quantities of interest including number densities, linear momenta, and kinetic energies to an output file.
- 8) Check the total number of time steps, N_{step} , that have been completed in the simulation.
- 9) Stop the simulation, output the final configuration of the channel information, and exit the code if N_{step} (see *step 8*) is equal to or larger than N_{max} (see *step 1*), otherwise, increment N_{step} and go back to *step 3*.

Steps 1 - 7 collectively are the main functions of the MDC model and each of these warrants a separate and complete description.

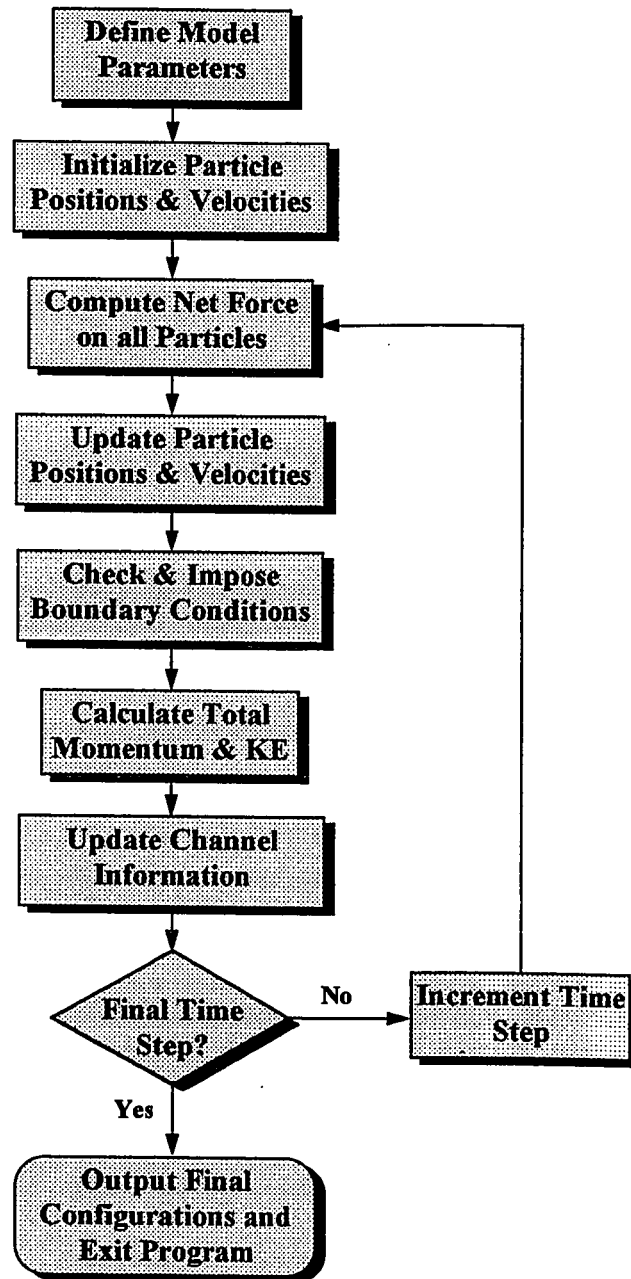


Figure 2-1: Flow chart of the MDC model.

2.2.2 Model Parameters

The model parameters within the MDC program define the flow model and the basic functionality of the computer program. Default values for all of the model parameters are set within the actual program. Several of these model parameters, such as those that define the number of cells in each direction and their respective dimensions, the number of bins for collecting statistical averages etc., are set internally using *#define* statements. If the user wishes to modify these parameters for a new model, the computer program must be recompiled in order to incorporate the changes. Recompilation is not very convenient for the user and therefore only those parameters that are *not* changed very often (e.g., the value of Boltzmann's constant) are defined in this manner. Most of the model parameters can be modified without recompilation via command line option switches. These switches and their associated parameter values are typed by the user on the command line when starting the simulation. *Table 2-1* lists the MDC model's sixteen available command line switch options. Most of the command line options require that additional parameters be entered immediately following the switch on the command line. The expected format of these parameters is listed next to the associated switch in *Table 2-1*.

The first task performed by the program following its initiation is to examine the command line and extract or *parse* the model parameter information from it. All model parameters set via command line options take precedence over default values. In other words, default values are used unless they are explicitly reset by command line options.

The default values for each of the command line options are listed in the last column of *Table 2-1*. The units in *Table 2-1* as well as those used by the MDC model and throughout this text are in CGS.

<i>Switch</i>	<i>Parameter</i>	<i>Description of Option</i>	<i>Default Value</i>
-dTime	<i>float #</i>	Time steps to use in simulation.	3.123×10^{-15} sec.
-nParts	<i>integer #</i>	Number of particles to use in simulation.	1 per cell.
-nIter	<i>integer #</i>	Number of iterations to run the simulation N_{max} .	1000
-periodic	---	Use periodic rather than thermal wall boundaries.	Thermal walls.
-readConfig	<i>filename</i>	Read initial particle configuration (positions & velocities) from file: <i>filename</i> .	Calculate initial configuration.
-tW1	<i>float #</i>	Temperature of thermal wall 1.	295 K.
-tW2	<i>float #</i>	Temperature of thermal wall 2.	295 K.
-tGas	<i>float #</i>	Initial average temperature of fluid.	295 K.
-vW1	<i>float #</i>	Velocity of thermal wall 1.	0 cm/sec.
-vW2	<i>float #</i>	Velocity of thermal wall 2.	0 cm/sec.
-accZ	<i>float #</i>	Acceleration due to external force acting in the direction parallel to the thermal walls.	0 cm/sec ² .
-printMod	<i>integer #</i>	Time step modulus for printing out diagnostic messages.	Every 20 time steps.
-diagFile	<i>filename</i>	Output diagnostic run-time messages to file: <i>filename</i> .	Output to file: max3Diag.out
-configFile	<i>filename</i>	Output final particle configuration (positions & velocities) to file: <i>filename</i> .	Output to file: max3Conf.out
-statFile	<i>filename</i>	Output statistical averages for density, particle velocity and kinetic energy to file: <i>filename</i> .	Output to file: max3Stat.out
-help	---	Prints out a line to the terminal screen showing the format for using the available command line switches.	---

Table 2-1: Command line switches, associated parameters, definitions, and default values.

2.2.3 Initialization

After the input parameters have been specified, the computer program needs to initialize several things prior to calculating the forces acting on particles and updating particle trajectories. The initialization process carries out two primary functions. The first is to calculate numerous quantities which are functions of the model parameters but otherwise remain constant during the calculation. The second is to initialize all of the particle positions and velocities prior to the start of the calculation.

There is a fundamental philosophy in performing numeric calculations which mandates that all invariant quantities used in a calculation within a loop should be defined or calculated *outside* of the loop. The obvious motivation for this is to eliminate unnecessary calculations and thereby minimize the amount of computer time needed to perform the calculation. The MDC model requires several layers of loops to perform its calculations. All quantities that are invariant within these nested loops are calculated prior to entering the loops during the initialization phase of the simulation. The largest group of variables in the MDC model that fall into the above category are several arrays of indices that are used in the determination of forward neighbor cells and inter-particle distances. A description of the forward neighbor index arrays and their initialization is provided in Appendix A. The details of how these arrays are used and a description of the forward neighbor cells in general will be provided in the next section on calculating the net forces acting on particles.

Following the initialization of static quantities, all of the initial particle positions and velocities must be determined. The MDC model provides two ways that this can be accomplished. The default method is to calculate these quantities based on model parameters. The other method for initializing the particle configuration is to read the initial particle positions and velocities into the program from a file.

In the default method, the initial particle positions are determined by setting up a uniformly spaced grid within the boundaries of the calculation such that there are *at least* as many grid points as particles. The grid points then define the initial particle positions. It is important that the particles be spaced as uniformly as possible at the start of the simulation. The numerical scheme for moving the particles during the simulation can become unstable if particles are placed too close to one another prior to reaching steady state conditions. This situation corresponds to inter-molecular distances that are physically too small (and thus energies too large) to be realistic in a steady state environment. Although spacing the particles uniformly provides the most stable initial configuration, it does not *assure* stability in the calculation. A time step that is too coarse for a given system configuration will result in model instability regardless of the initial particle configuration.

Given the number of cells in each direction N_x , N_y , and N_z and the total number of particles in the system N , the number of uniformly spaced grid points in each dimension n_x , n_y , and n_z are defined as follows:

$$n_x = \sqrt[3]{N \left(\frac{N_x^2}{N_y \cdot N_z} \right)} + 1,$$

Equation 2-1

$$n_y = n_x \cdot \left(\frac{N_y}{N_x} \right),$$

Equation 2-2

and,

$$n_z = \frac{N}{(n_x \cdot n_y)}.$$

Equation 2-3

Given the length of the system boundaries in each direction l_x , l_y , and l_z the particle grid spacing in each dimension $\Delta x = l_x/n_x$, $\Delta y = l_y/n_y$, and $\Delta z = l_z/n_z$ can be determined.

The particle velocities are determined using the initial gas temperature model parameter T_g . From the equipartition relationship:

$$\frac{1}{2}mv^2 = \frac{3}{2}kT,$$

Equation 2-4

where k is the Boltzmann constant, the magnitude of the initial particle velocities is determined by:

$$|v| = \sqrt{\frac{3kT_g}{m}}.$$

Equation 2-5

Calculation of the particle velocity directions requires the use of a uniform deviate generating function. The uniform deviate function employed by the MDC model is the function *RANI* obtained from the book *Numerical Recipes in C*.⁷ Given a function that generates uniform deviates in the range (0,1] symbolized by \mathcal{R} , a unit vector \mathbf{u} pointing in an arbitrary direction can be calculated. To calculate \mathbf{u} , weights for each direction w_x , w_y , and w_z and a corresponding normalization factor W need to be calculated by:

$$w_x = 0.5 - \mathcal{R},$$

$$w_y = 0.5 - \mathcal{R},$$

$$w_z = 0.5 - \mathcal{R},$$

and,

$$W = \sqrt{w_x^2 + w_y^2 + w_z^2}.$$

Equations 2-6

The unit vector \mathbf{u} is then determined by:

$$\bar{\mathbf{u}} = \frac{w_x}{W} \cdot \hat{\mathbf{x}} + \frac{w_y}{W} \cdot \hat{\mathbf{y}} + \frac{w_z}{W} \cdot \hat{\mathbf{z}}.$$

Equation 2-7

⁷ Press, W., Flannery, B., Teukolsky, S., and Vetterling, W., *Numerical Recipes in C, The Art of Scientific Computing*, Cambridge University Press, Cambridge, (1990), 210-211.

The particle velocities are then calculated by multiplying each unit vector \mathbf{u} by the velocity magnitude determined from *Equation 2-5*:

$$\vec{v} = |\mathbf{v}| \cdot \vec{u}.$$

Equation 2-8

The above prescription is really just a simple way to initialize the particle velocities such that the initial temperature of the system corresponds to T_g while at the same time locating the particles in a manner that will minimize the possibility that the inter-particle spacing is too close. These initial positions and velocities randomize rapidly as a result of inter-particle collisions once the simulation begins.

The initial particle configuration can also be read in from a file. *Table 2-1* from the previous section on model parameters indicates a command line option *readConfig* that is used for this purpose. Placing this option on the command line followed by the name of the file that contains the initial particle positions and velocities will *flag* the program to read the initial particle configuration from the file. The configuration file contains the number of particles in the system, each particle's x , y , and z positions as well as each particle's velocity components v_x , v_y , and v_z . When reading in the initial particle configuration from a file, the value for the number of particles N defined in the configuration file overrides any the default value or command line value set for this model parameter.

Recall from *Equation 1-5* that the Verlet method does not use the particle velocities directly for advancing the particle positions. Rather than use the velocities, the method uses the previous *two* particle positions to calculate the new particle positions. Therefore, the final step in setting up the initial particle configuration prior to advancing the particles is to use the initial particle position and velocity vectors to calculate the *previous* particle position vectors \mathbf{x}_{old} . This is easily performed using the time step model parameter Δt and the relationship:

$$\bar{\mathbf{x}}_{old} = \bar{\mathbf{x}} - \Delta t \cdot \bar{\mathbf{v}}.$$

Equation 2-9

2.2.4 Main Loop and Calculation of Net Force on all Particles

This section of the code is where the majority of the work in the simulation is performed and, although the procedure for calculating the net forces acting on individual particles is straightforward, the details of accomplishing this objective *efficiently* are somewhat more complicated. *Equation 1-2* and *Equation 1-7* suggest that the net force acting on a given particle at position \mathbf{r}_i is given by:

$$\bar{\mathbf{F}}(\bar{\mathbf{r}}_i) = \bar{\mathbf{F}}_{ext}(\bar{\mathbf{r}}_i) - \sum_{\substack{j=1 \\ j \neq i}}^N \bar{\mathbf{V}} \cdot \mathbf{V}(\mathbf{r}_{ij}),$$

Equation 2-10: Net force acting on a particle at position \mathbf{r}_i .

where N represents the total number of particles in the system, $F_{ext}(\mathbf{r}_i)$ is any external force acting on the particle at position \mathbf{r}_i , and $V(\mathbf{r}_{ij})$ is the effective pair potential between

particles at positions r_i and r_j respectively. As was previously discussed, it is computationally impractical to consider all particles when calculating the second term in *Equation 2-10*. The MDC program employs a cutoff radius r_c and effectively sets $V(r)$ equal to zero when r_{ij} is larger than r_c by ignoring the contribution of all particles outside of this range.

The basic approach taken by the MDC code to efficiently keep track of each particle and locate the particles within the cutoff radius r_c can be described as a *cell list* method.[†] In this method the total volume occupied by the particles and defined by the lengths L_x , L_y , and L_z is subdivided into cubic volume elements or *cells* with edges equal to the cutoff radius r_c . This results in N_x , N_y , and N_z cells in each dimension respectively with the total number of cells N_{cell} defined by the product ($N_x \times N_y \times N_z$). Individual cells are referenced by the notation $CELL_{i,j,k}$ where ($i = 1, 2, \dots, N_x$), ($j = 1, 2, \dots, N_y$), and ($k = 1, 2, \dots, N_z$) represent the cell indices in the x , y , and z directions respectively. With the space occupied by the system divided up in this manner, the cell spacing ensures that a particle in some $CELL_{i,j,k}$ can only interact with particles in the *same* cell or *neighboring* cells. All particles in non-adjacent cells are assured to be outside of the cutoff radius. This approach provides a mechanism by which atom pairs farther apart than r_c can be skipped without actually computing their separations. A cell $CELL_{i,j,k}$ is shown in *Figure 2-2* along with neighboring cells. Each cell has 26 neighbor cells in 3-D space. For example, in order to calculate the force acting on a particle P in the cell $CELL_{i,j,k}$, the

[†] These cells should not be confused with the grid used in initializing particle positions (see previous section).

program must consider all of the possible particle pairs within $CELL_{i,j,k}$ that include P , as well as all of the possible particle pairs within the 26 neighboring cells that include P . Thus, the task of determining the force acting on a given particle is limited to searching for particle pairs within 27 cells rather than all $N_x \times N_y \times N_z$ cells in the whole system. The forces acting on all particles in the entire system can be determined by using the above *nearest neighbor* approach and systematically applying it to every particle in the system.

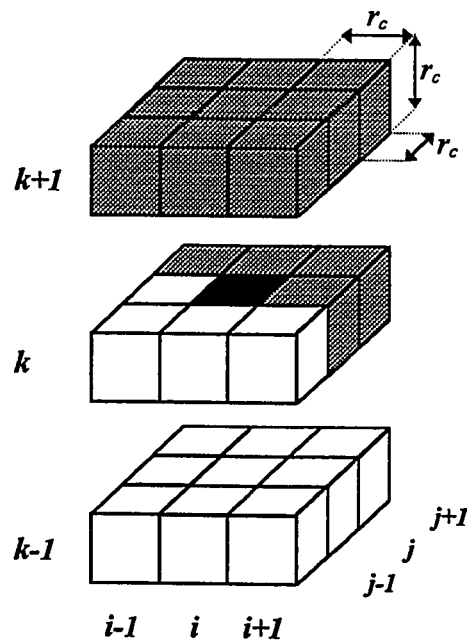


Figure 2-2: $CELL_{i,j,k}$ colored black and 26 neighboring cells. Forward neighbor cells to $CELL_{i,j,k}$ are gray and backward neighbor cells to $CELL_{i,j,k}$ are white. Note also that cell $CELL_{i,j,k}$ is a "forward neighbor" to each of the white cells.

A refinement of the above approach that will reduce computation is suggested by Newton's 3rd law. After calculating the force acting on particle P in a given cell due to particle Q , it is inefficient and unnecessary to repeat this calculation *again* when

calculating the force acting on particle Q due to particle P since these forces are equal and opposite. The key to avoiding this additional calculation is to calculate the inter-particle force acting between each particle pair only *once* and tally the result for *both* particles.

This approach suggests that the trick to developing an efficient algorithm is to find a way to reference all particle pairs from within the same or adjacent cells *once*. For particle pairs within the same cell this is accomplished with a single *nested* iteration scheme where the outer loop iterates over all particles in the cell and the inner loop iterates over all *other* particles in the cell not yet considered by the outer loop. This type of iteration scheme can be considered a *forward* scheme because the inner loop considers only those particles *ahead* of the current particle in the outer loop. This concept of *forward* iteration is extended to the case where the particle pairs are formed from adjacent cells. In this situation there are three nested loops. The outer loop iterates over all of the particles in a given cell $CELL_{i,j,k}$. The middle loop iterates over all of the *forward neighbor* cells relative to the cell $CELL_{i,j,k}$ with the inner-most loop iterating over all of the particles within the current *forward neighbor* cell referenced in the middle loop. The 13 forward neighbor cells relative to a cell $CELL_{i,j,k}$ are shaded gray in *Figure 2-2*. Also illustrated in *Figure 2-2* are the 13 backward neighbor cells relative to $CELL_{i,j,k}$ which are shaded white. Although it might appear that this scheme neglects the particle pairs formed between $CELL_{i,j,k}$ and the backward neighbor cells, in actuality these pairs are accounted for when $CELL_{i,j,k}$ is considered as a forward neighbor to other cells. One

way to verify that this scheme is sufficient to calculate all particle pair interactions, including those from backward neighbor cells, is to examine the spatial relationship between $CELL_{i,j,k}$ and each of its forward neighbors in *Figure 2-2* and then verify that $CELL_{i,j,k}$ is a forward neighbor to each of the white backward neighbor cells.

Referencing an arbitrary cell's forward neighbors is greatly simplified by using the IFN , JFN , and KFN matrices that are set up during initialization (see Appendix A). Examination of the indices of the forward neighbor cells in *Figure 2-2* relative to $CELL_{i,j,k}$ will verify that the following indexing scheme is sufficient for referencing the 13 forward neighbor cells of an arbitrary cell $CELL_{i,j,k}$. Taking ($n = 1, 13$):

$$CELL_{ifn(n,i),jfn(n,j),kfn(n,k)},$$

references all 13 forward neighbor cells for $CELL_{i,j,k}$. It should also be observed that this scheme provides the correct forward neighbor cell list even if $CELL_{i,j,k}$ is located on one or any of the system boundaries. This construct provides a powerful capability because computationally expensive conditional *if* statements are not necessary to properly handle special conditions near a boundary.

The $CELL_{i,j,k}$ that each particle occupies is determined from each particle's coordinates and the total number of particles contained in each cell is tabulated and put into the list $NPCELL_{i,j,k}$. This information is used to create two ordered lists that collectively identify the particles that are contained within each cell. The first list $IDPART$ contains particle ID numbers for each of N particles in the system. $IDPART$ is

assembled by stepping through each cell in the order of increasing cell rank and tabulating the particle IDs encountered. The cell rank of a given cell is determined by:

$$CELL_RANK_{i,j,k} = i + (j-1)*N_x + (k-1)*N_x*N_y,$$

Equation 2-11

where i , j , and k are the indices of the cell, and N_x , and N_y are the number of cells in the x and y directions respectively. The $CELL_RANK$ determines a unique integer index for each cell ranging value from 1 for $CELL_{1,1,1}$ to N_c for $CELL_{N_x,N_y,N_z}$. The offset into the $IDPART$ list, that locates the beginning of the list of particles contained within a given cell $CELL_{i,j,k}$, is calculated and maintained in the $IDOFF_{i,j,k}$ list. Collectively, the $NPCELL_{i,j,k}$, $IDPART$, and $IDOFF_{i,j,k}$ lists identify the particles contained within each cell.

The following 2-D example will illustrate the construction of the particle lists described above and how they are used to identify the individual particles within each cell. A system of 10 particles with particle IDs = 1, 2, ..., 10 are distributed in a 4×3 grid of cells in *Figure 2-3*.

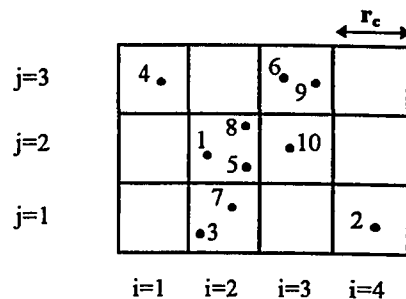


Figure 2-3: 2-D group of cells containing 10 particles.

The matrix $NPCELL_{i,j}$ is calculated by summing the particles in each cell. In this case:

$$NPCELL_{((i=1,4),j=1,3)} = (0, 2, 0, 1 ; 0, 3, 1, 0 ; 1, 0, 2, 0)$$

Equation 2-12

Using Equation 2-11 with $(i=1, 2, 3, 4 ; j=1, 2, 3 ; k=1)$, the cell rank $CELL_RANK_{ij}$ for each cell can be calculated. This results in a sequential ordering of cells that starts with $CELL_RANK_{1,1} = 1$ for $CELL_{1,1}$ in the lower left corner (see Figure 2-3) and moves from left to right, bottom to top, and ends with $CELL_RANK_{4,3} = 12$ for $CELL_{4,3}$ in the upper right corner. The list $IDPART = (3, 7, 2, 1, 5, 8, 10, 4, 6, 9)$ is obtained by moving through the cells sequentially by cell rank and collecting particle IDs. Finally, $IDOFF_{((i=1,4),j=1,3)} = (1, 1, 3, 3 ; 4, 4, 7, 8 ; 8, 9, 9, 10)$ lists the offset within the $IDPART$ list that corresponds to the first particle ID within the $CELL_{ij}$. Suppose that the particles within $CELL_{2,2}$ need to be identified. The list item $NPCELL_{2,2} = 3$ indicates that there are *three* particles within this cell. The list item $IDOFF_{2,2} = 4$ indicates that the desired three particle IDs will be the *fourth, fifth, and sixth* items in the $IDPART$ list which are particle IDs 1, 5, and 8 respectively. Note that if $NPCELL_{ij}$ is equal to zero (i.e., no particles in that cell) then there is no need to reference the other two lists.

Having set up the mechanisms from which to efficiently identify neighboring cells and the particles they contain, the calculation of the net force acting on each particle is straightforward. The main loop over time steps is entered first. Then, each of the particles is binned according to its current position and the appropriate lists for

referencing each particle within each cell are set up. The program is now prepared to calculate the net force acting on each particle. This process is initiated by looping over all cells. The net force on each particle in the current cell, due to other particles in the current cell, is calculated by looping over all particle pairs within the current cell. The net forces acting on all particles in the current cell, due to particles in forward neighbor cells, are calculated by looping over all particles in the current cell and then looping over all particles contained within each of the forward neighbor cells. The program then moves on to the next cell and repeats the loops over appropriate particle pairs. When each of the cells has been considered, the calculation of the net forces acting on each particle due to the existence of other particles is complete. If external forces are present this contribution is added to each particle at this time.

In order to calculate the force between a pair of particles, the separation of the particles must be calculated. This is usually quite straightforward. In all cases when the particles are within the same cell and in most cases when the particles are in adjacent cells, this calculation amounts to calculating the magnitude of the difference between the position vectors of each particle. The one exception to this is when one member of the particle pair is in a cell at a periodic boundary and the other member of the considered particle pair is in a neighboring cell at the periodic boundary on the *other* end of the system. In this situation, taking the difference between their respective position vectors gives the wrong result. Recall from the section on boundary conditions in the previous chapter that the particle position in the neighboring periodic boundary cell should be

translated by an amount equal to the dimension of the system in the appropriate direction prior to making the calculation. Because this radial distance calculation takes place inside of the MDC program's inner-most loop, conditionally checking for this situation is computationally expensive and therefore undesirable. The MDC program uses the $DXFN$, $DYFN$, and $DZFN$ arrays that were set up during the initialization phase of the program to avoid this conditional check (see Appendix A). Inside of the loop over particles in forward neighbor cells, the components of the inter-particle distances are calculated by:

$$\Delta x = x_2 - x_1 + dxfn_{n,i},$$

$$\Delta y = y_2 - y_1 + dyfn_{n,j},$$

and,

$$\Delta z = z_2 - z_1 + dzfn_{n,k},$$

where x_1, y_1, z_1 are the position coordinates of a particle in $CELL_{i,j,k}$, and x_2, y_2, z_2 are the position coordinates of a particle in the forward neighbor cell identified by the index n . Usually the additional $DXFN$, $DYFN$, or $DZFN$ matrix term is zero; however, if the two particles are on opposite ends of the system near a periodic boundary, the matrix term will be equal to $\pm L$ where L is the dimension of the system in the direction normal to the periodic boundary.

Before going on to the next section it should be mentioned that immediately following the calculation of the inter-particle separation, the separation distance r is compared to the cutoff radius to see if the force between the particle pair will be calculated. The Cartesian cell boundaries used in this model ensure that *all* particle pairs

within r_c of each other *will* be found within neighboring cells only but there is *no* assurance that *any* of the particle pairs from neighboring cells or even the same cell will be within r_c of each other. Therefore, this condition must be tested.

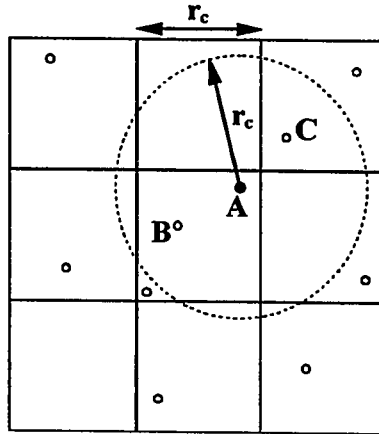


Figure 2-4: Particles B & C within cutoff radius of particle A.

This situation is depicted in *Figure 2-4* where particles at locations *B* and *C* are within the cutoff radius of a particle at location *A*. Other unlabeled particles within the same cell as *A* or in cells adjacent to the cell containing *A* are not within the cutoff radius.

2.2.5 Update Particle Positions and Velocities

Once the net force acting on each particle is determined, the corresponding acceleration for each particle is obtained from *Equation 1-2*, and the Verlet scheme, *Equation 1-5*, can be applied to obtain the new position of each particle. The new positions are then used in conjunction with the positions calculated two time steps back to produce the new particle velocities according to *Equation 1-6*. The final task

performed during this part of the calculation is to transfer the particle positions, calculated in the previous time step, to the memory locations used to hold the particle positions from two time steps back and, to transfer the new particle positions, just calculated, to the memory locations used to hold the positions from the previous time step. This operation saves the particle positions from the two most recent time steps and purges the values no longer needed for the trajectory calculations.

2.2.6 Impose Boundary Conditions

After the particle trajectories and velocities have been updated, the new particle positions must be checked to determine if they have crossed over any system boundaries. If a particle is determined to be outside of the system, its position (and possibly velocity) are reset according to the type of boundary crossed (see section on *Boundary Conditions* in Chapter One). Particles passing through periodic boundaries maintain their velocities but have their positions translated by the length of the system in the direction normal to the boundary. If a thermal boundary is crossed, then the particle's position is set to the location where it crossed the boundary and is given a new random velocity with a component that is normal to the boundary and directed into the channel.

The order in which the components of a particle's position are compared to the respective boundaries is important when both thermal and periodic boundaries are present in the flow model. The general rule is that the boundary conditions should be applied in the *same* chronological order in which the particle passed through the respective

boundaries. As an example consider the situation when a particle's new position is beyond a thermal boundary *and* beyond an adjacent periodic boundary having crossed the thermal boundary first. *Figure 2-5* illustrates the final particle position and direction of the particle after applying the thermal and periodic boundary conditions in opposite orders.

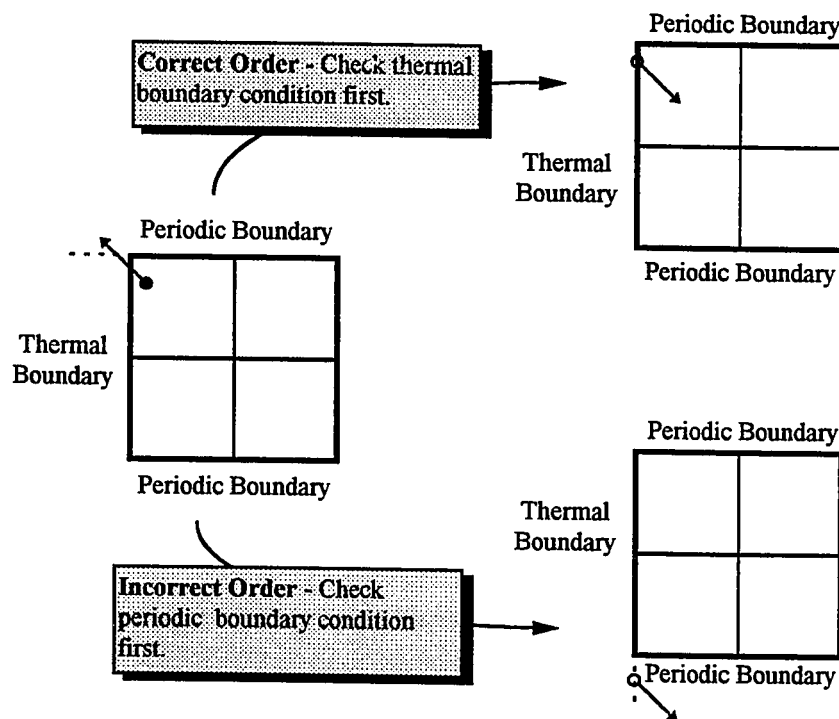


Figure 2-5: Sequential order of the application of boundary conditions.

It is clear from the example that in this case, the physically correct result is obtained by applying the thermal boundary conditions first. Because planar flow models do not have any adjacent thermal boundaries, the proper physical result is achieved by the model as long as the thermal boundary conditions are imposed *before* the periodic boundary conditions regardless of order in which the boundaries were crossed. This is a nice

circumstance because it simplifies the conditional checking required to impose the boundary conditions and thereby increases the run-time efficiency of the computer model.

2.2.7 Calculate Conserved Quantities

Throughout the simulation various conserved quantities are calculated for the purpose of providing diagnostic information regarding the integrity of the run. For instance, with periodic boundary conditions in all directions, it is required that the total energy and linear momenta of the system remain constant to within the truncation error of the Verlet scheme during the simulation. The total system energy is calculated by taking the sum of the total system potential energy and the total system kinetic energy:

$$\mathbf{E} = \sum_{i=1}^N \sum_{\substack{j=1 \\ j \neq i}}^N V(r_{ij}) + \frac{1}{2} m \sum_{i=1}^N |\vec{v}_i|^2,$$

Equation 2-13

where $V(r_{ij})$ is the effective pair potential, r_{ij} is the separation between particle pairs, m is the particle mass, and v_i is the particle velocity. Since the first term in *Equation 2-13* depends on all particle pairs that are within the cutoff radius of one another, this sum is calculated in the same inner-loop as the net force calculation. The second term in *Equation 2-13*, which calculates the total system kinetic energy, is calculated immediately following the calculation of the particle velocities. The total linear momentum of the system is also calculated at this time. These quantities are not necessarily printed out on every time step. Usually, time averaged values of these quantities are output to the terminal screen and a diagnostic output file at user specified intervals. The command line

switch *printMod* defined in *Table 2-1* controls the number of time steps between each diagnostic printout while the command line switch *diagFile* directs the output to a user specified file.

2.2.8 Output Steady State Channel Averages

Throughout the calculation *running* averages of the particle number density, momenta, and kinetic energy are computed and output to a disk file for later post-processing. The running averages are over both space and time. The spatial resolution of these averages is determined by the user via a *#define* parameter in the program. For this study, fifty spatial zones were used. Collectively these zones, which are parallel planar sub-volumes of equal width, span the entire channel volume between the thermal walls. The running averages are *not* updated on every time step because the particle configurations within each zone do not change that quickly. A more efficient strategy is to determine the average time that a particle spends within a spatial zone i.e., the time it takes a particle to travel the width of one spatial zone, and update the running averages at regular intervals equal to this time. The number of time steps between consecutive updates of the running averages, ΔN_{update} , is determined during initialization by using the relationship:

$$\Delta N_{update} = \frac{d}{\Delta t \cdot \langle v \rangle},$$

Equation 2-14

where d is the width of one spatial zone, Δt is the time step, and $\langle v \rangle$ is the magnitude of the average particle velocity. The running averages are written out to the disk file defined by the command line switch *statFile* at time intervals defined by the command line switch *printMod*.

2.3 Model Validation Using L-J Potential

The effort to validate the MDC model was separated into two phases. The first part of this effort was to check the basic physics at equilibrium in the MDC simulation and verify that the model conserved energy and linear momentum. The second phase of the validation was to check the simulation predictions for several non-equilibrium flow scenarios.

The MDC model was derived from an MD simulation code published by Morris in a research effort that studied slip lengths in dilute gases.⁶ Although the bookkeeping techniques, user options, and potential function were modified in the present work, the fundamental physics model employed by our MDC code is very similar to that employed by Morris in his research. Because of this similarity, it was possible to validate the implementation of these changes in the MDC code by using it to reproduce results reported by Morris. This verification phase required that the Lennard-Jones potential function be put into the MDC model since this was the potential function employed by Morris.

A list of the model parameters that were used for all models throughout the validation and verification phase of this effort are summarized below:

1. Cell widths the same in all directions and equal to 3σ ($\sigma = 3.405 \times 10^{-8}$ cm).
2. Ten cells in each direction (1000 total cells).
3. Mass equal to the argon atom mass (6.633×10^{-23} gm).
4. 1000 particles in the system.
5. Time step equal to 3.123×10^{-15} seconds.
6. 50 spatial zones for output of running averages.

The MDC program was compiled using the GNU GCC compiler and executed on SPARC 10 workstations for all of the runs presented for this effort.

2.3.1 Code Stabilization and Conservation

Prior to doing a direct comparison to the Morris results, a qualitative check of the basic physics model employed by the MDC code was performed. With periodic boundary conditions in all directions it was expected that a physically correct model would produce *constant* values for the total system energy and linear momenta throughout the calculation as well as produce spatially independent running averages of the particle number density, linear momenta, and kinetic energy. This verification was performed running the MDC model for 240,000 time steps and an initial gas temperature equal to 685 K.

Diagnostic outputs were requested every 1000 time steps in order to test the model's stability and verify the expected invariance in the total system energy and linear

momenta. Examination of the diagnostic output from the above run revealed a total system energy that was consistent with a lattice of 1000 particles and a temperature of 685 K. It was also observed that the total system energy, as well as each component of the total system linear momentum, remained statistically constant throughout the entire run.

Examination of the velocity components of the final particle configuration also indicated results consistent with expected steady state conditions for a periodic system. The average particle velocity components in each direction approached zero compared to the average magnitude of the particle velocity components in each zone. Spatial independence of the running average quantities was also observed. The steady state average number density in each zone was approximately equal to 20 which is consistent with a random distribution of 1000 particles in 50 uniform zones and, the steady state average temperature was also statistically identical in all zones and equal to the initial temperature of the system.

The integrity of the thermal model was also checked by replacing the periodic boundaries in the x direction with stationary thermal wall boundaries held constant at 457 K. Using the final particle configuration from the above model as the initial particle configuration for the new model, the MDC model was run for 240,000 time steps.

The steady state particle densities and temperature profiles from this run, which are shown in *Figure 2-6* and *Figure 2-7* respectively, are indicative of a valid model. The

number densities for each zone show a small statistical fluctuation about the expected average steady state value of 20. The expected statistical variation in this data s can be quantified with the following relationship:

$$s = \sqrt{\frac{\langle n_z \rangle}{N_s}},$$

Equation 2-15

where $\langle n_z \rangle$ is the expected number density in each zone and N_s is the number of statistical samples taken. For this run with $\langle n_z \rangle = 20$, and $N_s = 390$, the variation in the measured number density is: $s = 0.23$. Values of $3s$ away from the expected average number density are indicated by dashed lines in *Figure 2-6*; we expect 97% of the data to fall within the dashed lines.

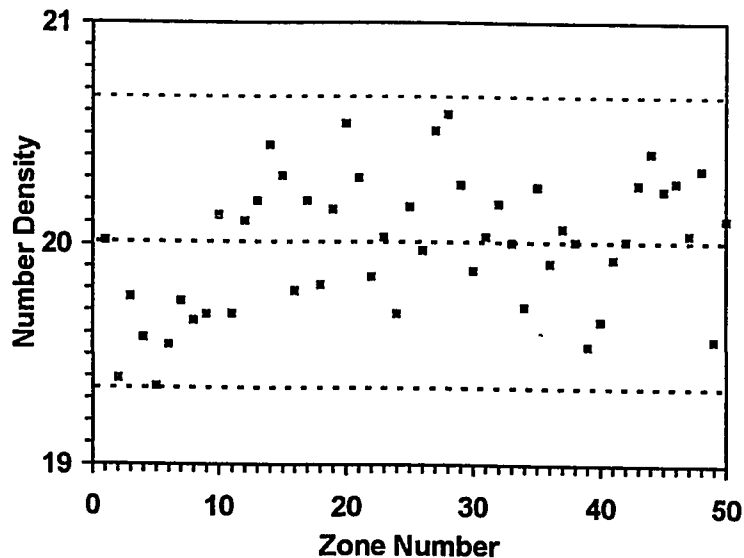


Figure 2-6: Steady State average number density for each zone. Expected and $3s$ values indicated by dashed lines.

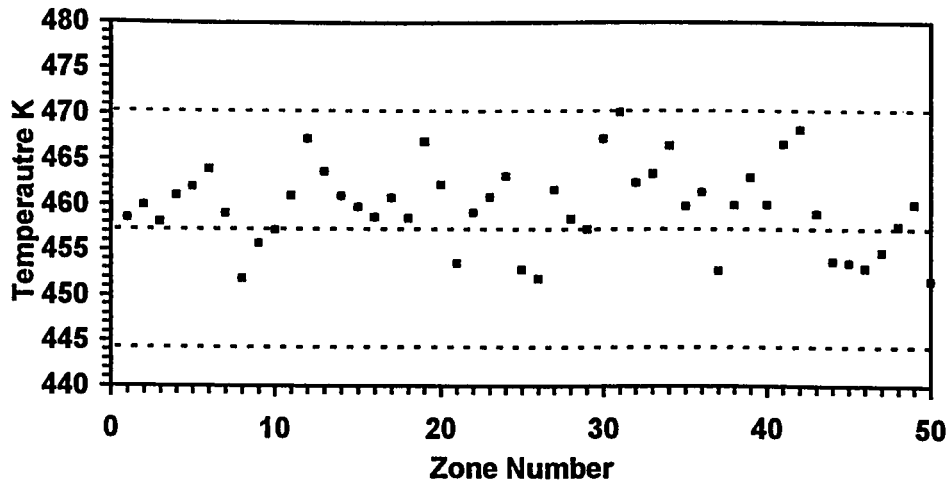


Figure 2-7: Steady state average zone temperatures calculated using equipartition relationship. Expected and 3s values indicated with dashed lines.

The average zone temperatures T are calculated using the equipartition relationship:

$$T = \frac{m}{3 \cdot k} \left(\langle v_x^2 + v_y^2 + v_z^2 \rangle - \langle v_x \rangle^2 - \langle v_y \rangle^2 - \langle v_z \rangle^2 \right),$$

Equation 2-16

where k is the Boltzmann constant, m is the particle mass, and v_x , v_y , and v_z are the average particle velocities in each direction respectively. Examination shows an expected small statistical fluctuation about the 457 K wall temperature. In this case the expected statistical variation, s , in the expected temperature T is given by:

$$s = \sqrt{\frac{T^2}{\frac{3}{2} \langle n_z \rangle N_s}},$$

Equation 2-17

where $\langle n_z \rangle$ is the average number density in each zone and N_s is the number of statistical samples taken.⁸ For $T = 457$ K, $\langle n_z \rangle = 20$, and $N_s = 390$ the variation in the measured temperature is: $s = 4.4$. Once again, 97% of the data is expected fall within the $3s$ limits.

2.3.2 Comparison to the Morris Results

Having verified that the MDC model produces expected steady state results for both periodic and static thermal boundaries, the second phase of the validation effort was to reproduce some of the results produced by Morris. Two cases run by Morris were selected for this purpose. These cases correspond to Couette and Poiseuille configurations for his run 6. Morris' description for setting up these runs was followed. The systems consisted of 1000 particles with a distance of 30σ between thermal walls of cross-sectional area of $900\sigma^2$. The initial particle configuration used for these models was the final configuration of the run outlined in the previous section with static thermal walls at 457 K. The thermal wall temperature for these runs was 457 K and therefore the initial particle configuration was representative of a system in thermal equilibrium with the wall boundaries.

Steady state statistics for the number density, linear momenta, and kinetic energy running averages were produced for both models by running each of the models for five consecutive runs of 100,000 time steps each. The final steady state particle configuration for any one run provided the initial particle configuration for the next consecutive run. In

⁸ Landau, L.D., and Lifshitz, E.M., *Fluid Mechanics*, Pergamon Press, Oxford, (1959).

this manner five output files containing steady state running averages were produced for each model. The five sets of data, each containing representative steady state predictions of the running averages across 50 zones, collectively provided information on the statistical errors associated with the MDC model predictions in each zone. These errors were determined through statistical analysis of the five data sets collectively.

The statistical analysis employed was based on the following three statistical relationships given n measurements of a quantity q :

$$\langle q \rangle = \frac{1}{n} \sum_{i=1}^n q_i ,$$

Equation 2-18

$$\sigma^2 = \frac{1}{n-1} \sum_{i=1}^n (q_i - \langle q \rangle)^2 ,$$

Equation 2-19

and,

$$\Delta = \frac{\sigma}{\sqrt{n}} .$$

Equation 2-20

Equation 2-18 is recognized as the mean value of a quantity q while *Equation 2-19* represents the *variance* of q about its mean value. *Equation 2-20* represents the statistical error associated with the value of $\langle q \rangle$. This statistical error is observed to approach zero as the number of samples n increases since the variance approaches a

constant as the number of observations increases. The magnitude of the statistical errors associated with the running averages in each zone is represented on all plots with *error bars*. A separate program was used as a post processor to evaluate the mean values and the statistical errors for each of the running averages in each zone by processing the data from all five runs for each model. The post-processing code uses the linear momenta to produce velocity profiles and the kinetic energy and velocity profiles to produce temperature profiles using *Equation 2-16*. A copy of this program is listed in *Appendix D* along with the listing of the MDC program.

For the Poiseuille runs, an external acceleration field of magnitude $2.27 \times 10^{15} \text{ cm/s}^2$ was applied and the thermal wall velocities were set to zero. The five Poiseuille runs were calculated, post-processed, and plots were produced. The steady state z velocity profile for the Poiseuille model is shown in *Figure 2-8*. Examination of the general shape of this velocity profile matches the quadratic description of Poiseuille z velocity flow discussed previously and illustrated in *Figure 1-4*. There are two velocity axes in *Figure 2-8* and they represent CGS units on the left and the scaled units used by Morris on the right respectively. The scaled units are produced by dividing the CGS units by $1.09 \times 10^5 \text{ cm/s}$. Comparison of these values to those published by Morris indicated that to within statistical uncertainty the MDC and the Morris models produced the same results. The MDC number density and temperature profiles, which are provided in *Appendix B*, also compared favorably with the results published by Morris.

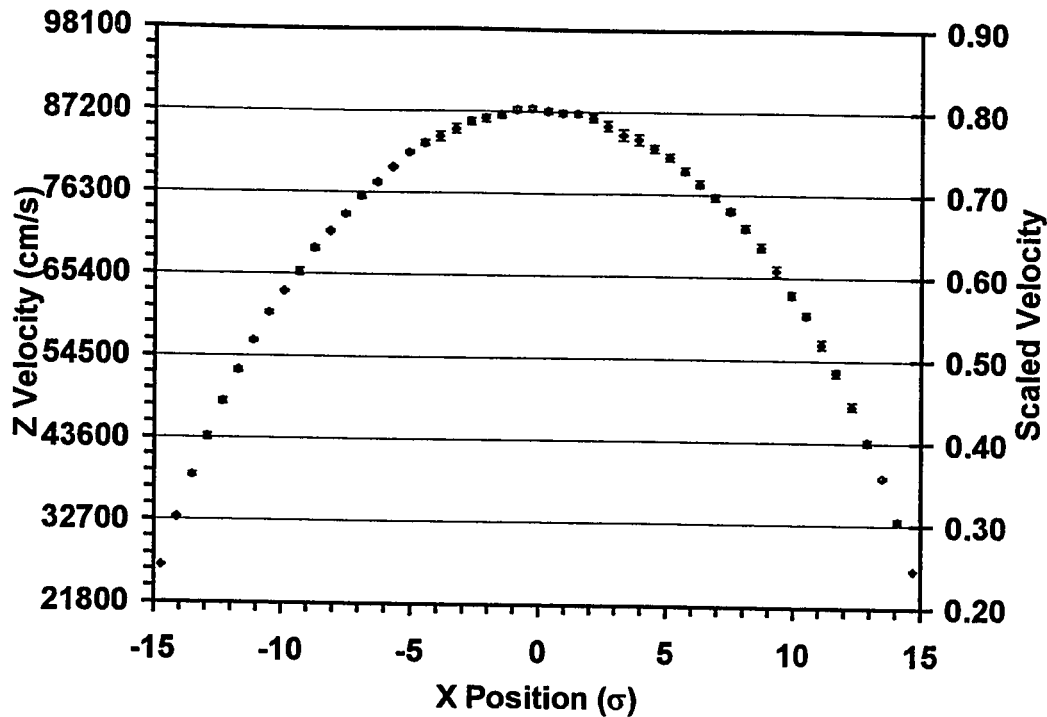


Figure 2-8: Poiseuille fluid velocity v_z vs. position. Scaled units are CGS units divided by 1.09×10^5 cm/s.

For the Couette runs the external acceleration field in the z direction was turned off and the left and right thermal walls were given velocities v_y equal to $\mp 4.36 \times 10^4$ cm/s respectively.[†] The five Couette runs were calculated, post-processed, and plots were produced. The steady state y velocity profile for the Couette model is shown in Figure 2-9. Examination of the general shape of this profile matches the linear description of the Couette velocity flow discussed previously and illustrated Figure 1-3. Once again, the data are plotted using both a CGS scale and a dimensionless scale. Comparison of these values to results published by Morris indicated that to within

[†] For argon at 273 K, these speeds correspond to \mp Mach 1.42 (see section 3.3).

statistical uncertainty the MDC model produced values consistent with his results. A comparison of the MDC number density and temperature profiles, which are provided in *Appendix B*, to the Morris results indicated favorable agreement between MDC and his model.

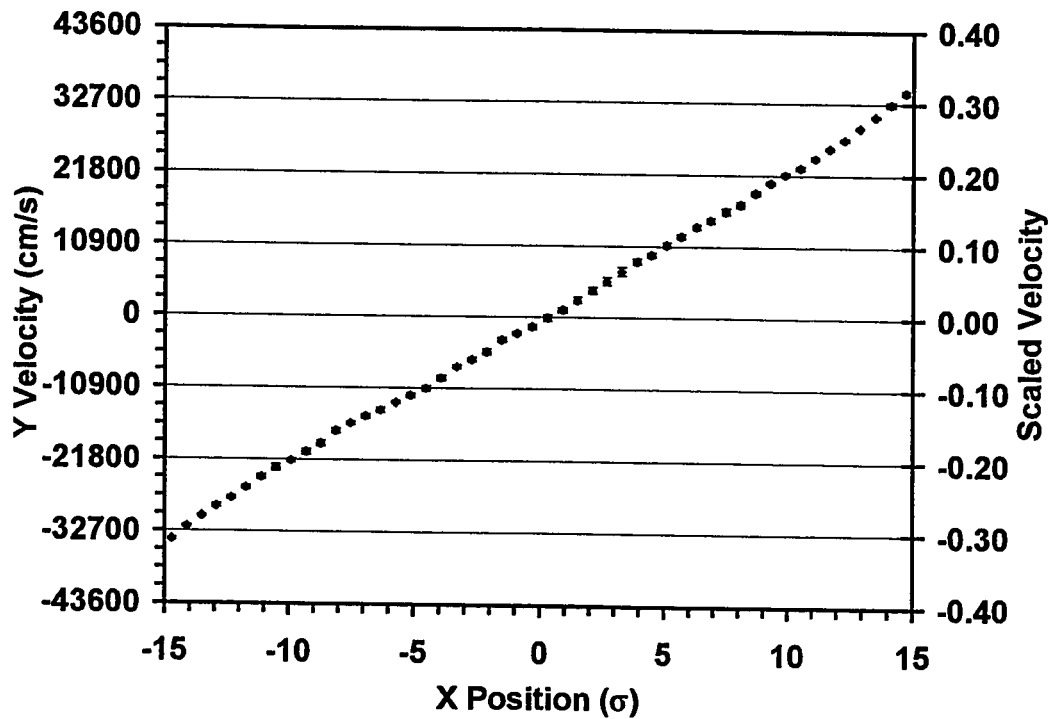


Figure 2-9: Couette fluid velocity v_y vs. position. Scaled units are CGS units divided by 1.09×10^5 cm/s.

The favorable comparisons of the MDC predictions to the results published by Morris provided strong evidence that the MDC model was producing valid results for argon in both the Poiseuille and Couette regimes. These results coupled with the thermodynamic equilibrium results presented in the previous section provide the necessary verification and validation of the MDC simulation model.

3 Denery Research

3.1 Introduction

This chapter presents the main results of this research effort which was to compare MD simulation of general Couette flow in the 0.1 to 1.0 Knudsen number regime to results produced by Denery's semi-analytical solution to this problem. The first section presents a brief description of the research and theory that provides the basis for Denery's model. That section is followed by a description of the model parameters used to define the run configurations for the MDC and Denery models. Finally, a comparison and discussion of the results produced by the two respective models is presented.

3.2 The Denery Model

The Denery model is an extension of the work performed by Lees and Liu in 1961 where they developed a general solution to Couette flow. Fundamental to the approach of Lees, Liu and Denery, was an assumption that the velocity distribution throughout the entire flow field was accurately described by a two sided Maxwellian velocity distribution. This distribution is formed by the superposition of two half Maxwellian distributions with independent temperatures and mean velocities as shown in *Figure 3-1*.

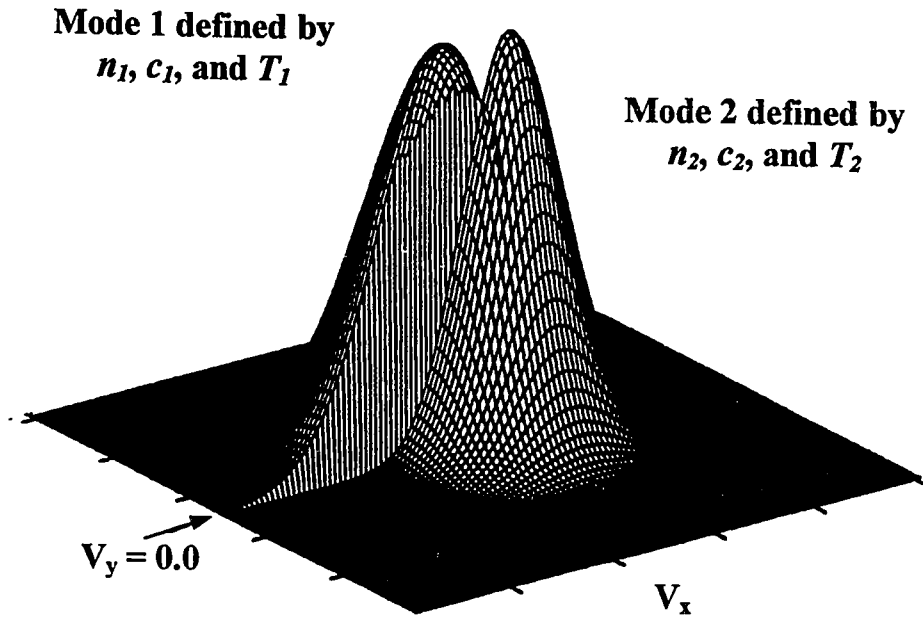


Figure 3-1: Two sided Maxwellian velocity distribution.

A discontinuity in the resulting function exists at the plane $v_y = 0$. The downward moving particles are characterized by a Maxwellian distribution defined by a temperature T_1 , mean downward velocity c_1 , and number density n_1 :

$$f_1(\vec{v}) = n_1 \left(\frac{m}{2\pi k T_1} \right)^{\frac{3}{2}} e^{-\frac{m}{2k T_1} [(v_x - c_1)^2 + v_y^2 + v_z^2]} \quad v_y < 0,$$

Equation 3-1

and the upward moving particles are characterized by a Maxwellian defined by a temperature T_2 , mean upward velocity c_2 , and number density n_2 .

$$f_2(\vec{v}) = n_2 \left(\frac{m}{2\pi k T_2} \right)^{\frac{3}{2}} e^{-\frac{m}{2\pi k T_2} [(v_x - c_2)^2 + v_y^2 + v_z^2]} \quad v_y > 0.$$

Equation 3-2

Using the Heavyside step function $h(x)$:

$$h(x) = 0 \quad \text{for } x < 0,$$

and,

$$h(x) = 1 \quad \text{for } x \geq 0,$$

the entire two sided Maxwellian can be represented by:

$$f(\vec{v}) = h(-v_y)f_1(\vec{v}) + h(v_y)f_2(\vec{v}).$$

Equation 3-3

The parameters n_1 , n_2 , c_1 , c_2 , T_1 , and T_2 , used by Lees and Liu to define their velocity distribution function, are determined by constraining them to satisfy six moments of the Boltzmann equation. Although this constitutes a *closed* form approach, a fundamental assumption is made when only *six* moment equations are used to constrain the velocity distribution function rather than the infinite number of constraints required for a full solution of the Boltzmann equation. The error introduced by this assumption is minimized by carefully selecting the six most important moments. This approximation coupled with the original assumption that the two-sided Maxwellian distribution is an appropriate representation of the flow field velocities constitute the fundamental factors governing the validity of the Lees and Liu solution.

The Boltzmann equation defines the evolution of the velocity distribution as a function of position, and in general, time. The moments of the Boltzmann equation, often called the Maxwell-Boltzmann moment equations, provide the important macroscopic properties that describe a flow including the density, fluid velocity, internal energy, stress, and heat conduction. The general Maxwell-Boltzmann moment equation for steady, one-dimensional flow is defined by:

$$\frac{\partial}{\partial y} \int_{-\infty}^{\infty} Q(\vec{v}) v_y f(\vec{v}) d\vec{v} = \Delta[Q],$$

Equation 3-4

where $Q(\vec{v})$ is any function of particle velocity and $\Delta[Q]$ is the moment of the Boltzmann collision integral corresponding to $Q(\vec{v})$. The moment $\Delta[Q]$ defines the change in the function $Q(\vec{v})$ produced by intermolecular collisions.

Lees and Liu sought the six most important moments of the Maxwell-Boltzmann equation to determine the six parameters of their assumed velocity distribution. The first four equations that they used consist of those expressions for the function $Q(\vec{v})$ which are conserved quantities in a collision between two molecules:

$$\begin{aligned} Q_1 &= m && \text{(molecular mass),} \\ Q_2 &= mv_x && \text{(x momentum),} \\ Q_3 &= mv_y && \text{(y momentum), and,} \\ Q_4 &= m \frac{v^2}{2} && \text{(energy).} \end{aligned}$$

Equations 3-5

Because of the collisional invariance for each of the quantities, the moment of the collision integral $\Delta[Q]$ corresponding to each is zero. The final two equations employed by Lees and Liu are the functions:

$$Q_5 = mv_x v_y,$$

Equation 3-6

and,

$$Q_6 = mv_y \frac{v^2}{2}.$$

Equation 3-7

With the mean velocity component normal to the thermal wall boundaries equal to zero, the moment of Q_5 , represented by $\langle Q_5 \rangle$, has the following relationship with shear stress τ_{xy} :

$$\langle Q_5 \rangle = -\frac{\tau_{xy}}{n},$$

Equation 3-8

and the moment of Q_6 represented by $\langle Q_6 \rangle$ has the following relationship with shear stress τ_{xy} and heat conduction q_y :

$$\langle Q_6 \rangle = \frac{1}{n}(q_y - \tau_{xy}c),$$

Equation 3-9

where,

$$c = \frac{n_1 c_1 + n_2 c_2}{n_1 + n_2},$$

is the magnitude of the center-of-mass velocity. Because this flow is dominated by both shear stress and heat conduction, Lees and Liu ensure that these two viscous phenomena are modeled correctly by including these last two constraints in their model.

Unlike the first four equations, the moments of the collision integral [Q5] and [Q6] are not zero and it is in the evaluation of these terms that the work of Denery diverges from that of Lees and Liu. In the completion of the Lees and Liu approach they assume a Maxwell-molecule intermolecular force potential of the form :

$$V(\mathbf{r}) = \frac{\beta}{r^4},$$

Equation 3-10

which allows them to find a closed form solution for $\Delta[Q_5]$ and $\Delta[Q_6]$. In contrast to this approach, Denery assumed a general inverse power form:

$$V(\mathbf{r}) = \frac{\beta}{r^\alpha},$$

Equation 3-11

where β is a constant and α is any power. In this case, a closed form solution for $\Delta[Q_5]$ and $\Delta[Q_6]$ cannot be found.

The general definition for the moment of the collision integral is given by:

$$\Delta[Q] = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \int_0^{4\pi} [Q(\vec{v}') - Q(\vec{v})] n^2 f(\vec{v}) f(\vec{u}) c \sigma d\Omega d\vec{v} d\vec{u}.$$

Equation 3-12

This integral is over all possible bimolecular collisions. The pre-collision velocities of the two respective particles is given by \bar{v} and \bar{u} . The post-collision velocity of the particle with previous velocity \bar{v} is given by \bar{v}' . Function f defines the velocity distributions of the two colliding particles. The collision cross section given by σ is, in general, a function of the relative speed c and the orientation of the respective particles. The vector parameter Ω , which is composed of two angles, describes the inter-particle orientation prior to the collision. The two velocities along with Ω produce an eight dimensional integral which is difficult and often impossible to solve in closed form.

By choosing the special case of the inverse fourth power, Lees and Liu were able to solve *Equation 3-12* in closed form for $\Delta[Q_5]$ and $\Delta[Q_6]$. Although the more general inverse power relationship chosen by Denery does not permit a *complete* closed form solution, he was able to reduce both equations for $\Delta[Q_5]$ and $\Delta[Q_6]$ from their original eight-dimensional form given by *Equation 3-12* down to a 2-dimensional form.

The dimensional reduction was performed by following Baganoff's approach. Baganoff performed integration over the orientation parameter Ω , and switched the velocity representation from the individual molecular velocities to the center of mass and the relative velocity of the molecular pair.⁹ Denery performed analytic integration over the center of mass velocity and reduced the order of the integrals for $\Delta[Q_5]$ and $\Delta[Q_6]$ down to 3-dimensions each respectively. He then removed one of the angular

⁹ Baganoff, D., "Maxwell's second- and third-order equations of transfer for non-Maxwellian gases", *Phys. Fluids*, A 4 (1), Jan. (1992), 141-147.

dependencies from each equation by representing the relative velocity in spherical coordinates and integrating over one of the angles. Finally, with the two equations reduced as far as analytic means would allow, the remaining integrations were performed numerically.

Having defined the six functions $Q(\vec{v})$ and calculated each of their respective moments of the collision integral $\Delta[Q]$, the remainder of the solution to the general inverse Couette problem is to finish solving the six equations suggested by *Equation 3-4* for the six unknowns $n_1, c_1, T_1, n_2, c_2,$ and T_2 . This procedure was performed numerically by Denery using an iterative Newton-Raphson technique. A comparison by Denery using a value of $\alpha = 4$ in his model produced almost identical results to those predicted by the Lees Liu closed form solution. In the third section of this chapter, results of the Denery model using a value of $\alpha = 9$ will be compared with simulation results from the MDC model.

3.3 Denery Potential and Parameters for the MDC Model

In order to prepare the MDC simulation program for comparison to the Denery model, several relationships between model parameters used by Denery need to be defined. Denery's work provides solutions to general Couette flow problems where the intermolecular force potential is of the form:

$$V(r) = \frac{\beta}{r^\alpha}$$

Equation 3-13

For the MDC comparison to the Denery model α is chosen to be 9 because it is generally accepted that this value most accurately approximates the inter-particle potential for noble gases.[†] Equation 3-13 is of the right form for use in the MDC model, however, it is necessary to evaluate the constant β before this equation can be used. The general Couette configuration to be modeled is illustrated in Figure 3-2.

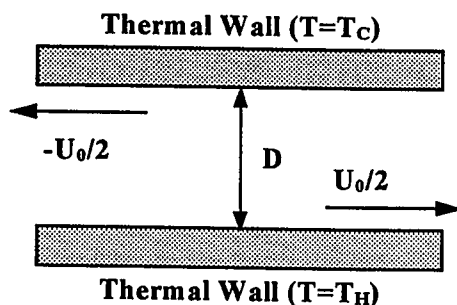


Figure 3-2: Illustration of general Couette configuration.

The value of β was selected to match as closely as possible the viscosity of argon.

From Hirschfelder, Curtis and Bird¹⁰, the viscosity of argon η is given by:

$$\eta = \frac{5}{8} \sqrt{\frac{km}{\pi}} \left(\frac{k}{\beta\alpha} \right)^{2/\alpha} \frac{T^{1/2+2/\alpha}}{\Gamma(4-2/\alpha)A^{(2)}(\alpha)},$$

Equation 3-14

[†] An even more accurate potential is obtained if an attractive term is included (e.g., Lennard-Jones).

¹⁰ Hirschfelder, J. O., Curtiss, C. F., and Bird, R. B., *Molecular Theory of Gases and Liquids*, John Wiley & Sons Inc., New York, (1954).

where, k is the Boltzmann constant, m is the mass, Γ is the gamma function, and $A^{(2)}(\alpha)$ is a special function. The value for β in Equation 3-14 can be calculated by using the following values: for argon at $T = 273$ K, $\eta = 2.099 \times 10^{-4}$ gm/cm \cdot s; for $\alpha = 9$, $\Gamma(4 - 2/9) = 4.5712$, and $A^{(2)}(9) = 0.327$; $m = 6.63 \times 10^{-23}$ gm, $k = 1.3806 \times 10^{-16}$ erg/K.

Using these values gives:

$$\beta = 1.759 \times 10^{-81} \text{ erg} \cdot \text{cm}^9.$$

Equation 3-15

The mean free path for this system must be defined in order to calculate the Knudsen number according to Equation 1-10. The mean free path λ is defined as:

$$\lambda = \frac{1}{3\sqrt{2}\pi A_2(\alpha)\Gamma(2 - 2/\alpha)} \frac{1}{n} \left(\frac{2kT}{\beta\alpha} \right)^{2/\alpha},$$

Equation 3-16

where n is the number density, Γ is the gamma function, and $A_2(\alpha)$ is a special function defined by Chapman and Cowling.¹¹ For $\alpha = 9$, $\Gamma(2 - 2/9) = 0.9275$, $A_2(9) = 0.382$, and $T = 273$ K the expression for λ in Equation 3-16 reduces to:

$$\lambda = \frac{1}{n} \cdot (1.39 \times 10^{14} \text{ cm}^{-2}).$$

Equation 3-17

Assuming a *cubic* system, the number density n is given by the relationship:

¹¹ Chapman, S., and Cowling, T. G., *The Mathematical Theory of Non-Uniform Gases*, Cambridge University Press, Cambridge, (1939).

$$n = N / D^3,$$

Equation 3-18

where N is the number of particles in the system and D is the distance between the thermal boundaries. Using *Equation 1-10*, *Equation 3-17*, and *Equation 3-18* an expression for the Knudsen number K_n in terms of the number of particles in the system N and the system dimension D is determined:

$$K_n = (1.39 \times 10^{14} \text{ cm}^{-2}) \frac{D^2}{N}.$$

Equation 3-19

Mach numbers are often used when defining the velocities of the thermal walls. The Mach number for a given velocity is a dimensionless quantity that is obtained by dividing a given velocity v by the sound speed. The sound speed in a dilute gas is given by:

$$c = \sqrt{\frac{\gamma k T}{m}},$$

Equation 3-20

where the constant γ is 5/3 for a monatomic gas. For argon at 273 K the sound speed c is: $c = 3.08 \times 10^4$ cm/s.

3.4 Comparison of MDC to Denery Model

The overall objective in comparing the MDC model results to the Denery model results is to examine the respective behaviors of each model in the larger Knudsen number

regimes between 0.1 and 1.0 under a variety of system configurations that feature different thermal wall temperatures and velocities. After consulting with Dr. Denery, eight cases were selected for the comparison of the MDC and Denery models. The model parameters for each of these cases are outlined in *Table 3-1*.

<i>Run</i>	T_H (K)	T_H/T_C	K_n	D (cm)	r_c (cm)	<i>System Cells</i>	$U_0/2$ (cm/s)	M_a
1	409.5	1.5	0.1	1.72×10^{-6}	1.075×10^{-7}	16x16x16	7700	0.5
2	409.5	1.5	1.0	5.43×10^{-6}	1.086×10^{-7}	50x50x50	7700	0.5
3	409.5	1.5	0.1	1.72×10^{-6}	1.075×10^{-7}	16x16x16	30800	2.0
4	409.5	1.5	1.0	5.43×10^{-6}	1.086×10^{-7}	50x50x50	30800	2.0
5	2730	10	0.1	1.72×10^{-6}	1.075×10^{-7}	16x16x16	7700	0.5
6	2730	10	1.0	5.43×10^{-6}	1.086×10^{-7}	50x50x50	7700	0.5
7	2730	10	0.1	1.72×10^{-6}	1.075×10^{-7}	16x16x16	30800	2.0
8	2730	10	1.0	5.43×10^{-6}	1.086×10^{-7}	50x50x50	30800	2.0

Table 3-1: Model parameters for comparison runs. 4096 particles were used for all runs.

Half of the cases are in the 0.1 Knudsen number regime and the other half are in the 1.0 Knudsen number regime. The number of particles was 4096 for all runs. Given the desired Knudsen number, K_n , and the total number of particles, N , *Equation 3-19* determines the distance, D , between the thermal walls. All of the runs contained one *cold* and one *hot* wall. The cold wall temperature $T_C = 273$ K was the same for all of the runs, and the hot wall temperature T_H was either 409.5 K or 2730 K. These temperatures produce wall temperature ratios, T_H/T_C equal to 1.5 and 10 respectively. The relative wall velocities U_0 were set according to two different Mach number regimes M_a equal to

0.5 and 2.0 which using $c = 3.08 \times 10^4$ cm/s corresponds to values of U_0 equal to 15,400 cm/s and 616,000 cm/s respectively.

The cell dimensions were determined for each case based on the system dimension D and choosing a reasonable value for the cutoff radius r_c . Recall that for the Lennard-Jones potential model, which is reasonable model for argon, the cutoff radius was 3σ which equals 1.02×10^{-7} cm. In an effort to maintain model features that reproduce reasonable results for argon, this cutoff radius was used as a general guideline for choosing the number of cells spanning the system dimension D . Selecting 16 and 50 cells across for the systems with D equal to 1.72×10^{-6} cm and 5.43×10^{-6} cm produces cell widths equal to 1.075×10^{-7} cm and 1.086×10^{-7} cm respectively. These cell dimensions correspond to reasonable values for the respective cutoff radii.

The model parameters in *Table 3-1* were used as inputs for MDC and Denery simulation runs. For each of the MDC simulation cases five contiguous runs of 100,000 time steps each were performed. The first of the five runs served to allow the particles to reach equilibrium with the respective system conditions. The subsequent four runs were used to generate statistics for the equilibrium values of the running average quantities across fifty equally spaced zones. For this comparison the running average quantities examined were the number of particles, the particle velocity parallel to the thermal walls, and the local temperature. Error bars indicating the statistical variation are plotted with the MDC predictions.

Comparison plots of the three running average quantities for all eight cases were produced. All twenty four plots can be found in *Appendix C*. Examination of the comparison plots indicates several interesting trends between the MDC and Denery model. The two models are in close agreement for the low temperature gradient cases with Knudsen number $K_n = 0.1$ (runs 1 and 3). The largest observed differences between the profiles in this regime are less than 5%. *Figure 3-3*, *Figure 3-4*, and *Figure 3-5* show comparison plots of the running average quantities for run 3. The Denery predictions are shown with a dashed line while the MDC predictions are indicated with diamond markers and error bars.

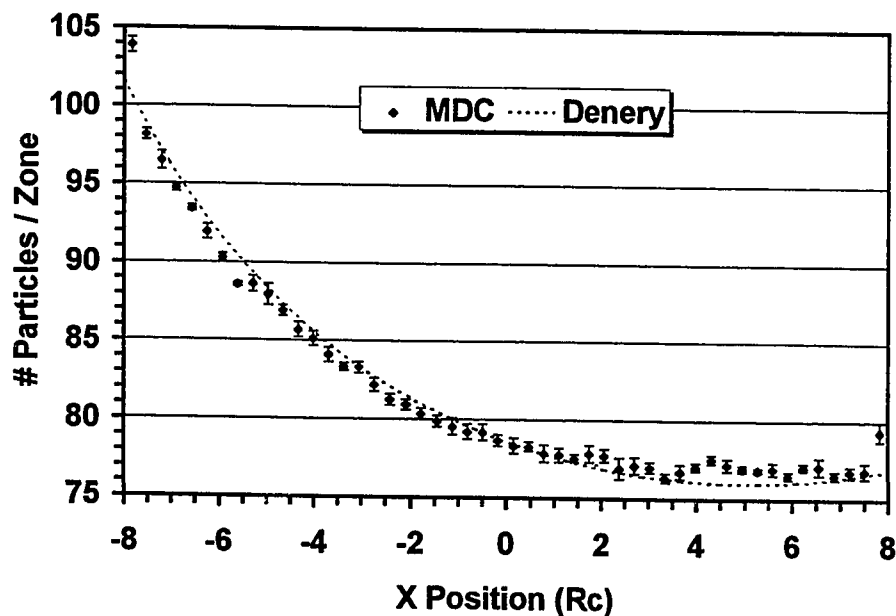


Figure 3-3: Average number of particles per zone as a function of position for run 3 where $T_W/T_C = 1.5$, $K_n = 0.1$, and $M_a = 2.0$.

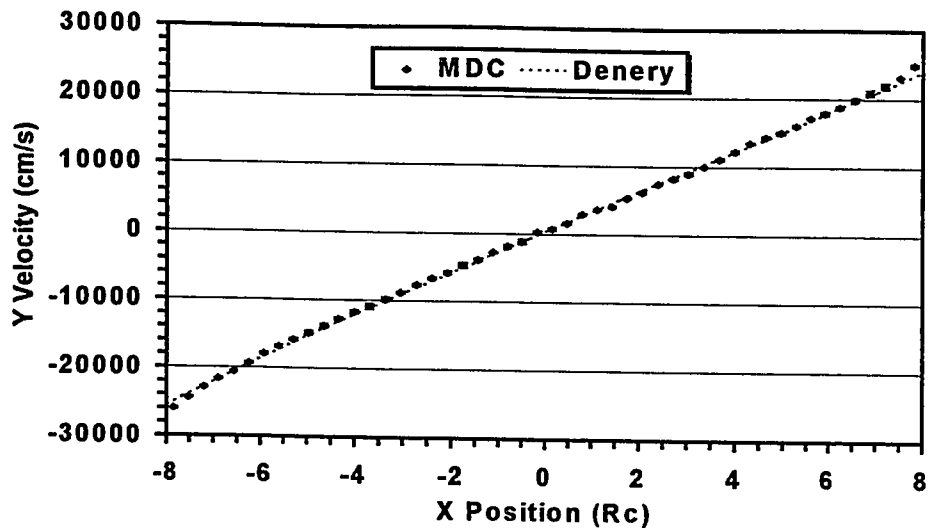


Figure 3-4: Average particle velocity (component parallel to thermal walls) as a function of position for run 3 where $T_W/T_C = 1.5$, $K_n = 0.1$, and $M_a = 2.0$.

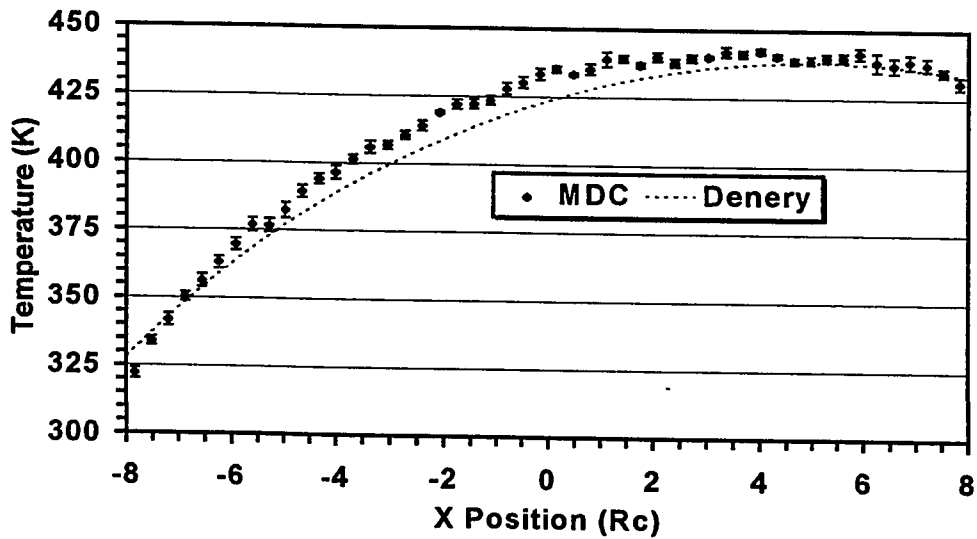


Figure 3-5: Average temperature as a function of position for run 3 where $T_W/T_C = 1.5$, $K_n = 0.1$, and $M_a = 2.0$.

The respective number density and temperature predictions of the MDC and Denery models are also in close agreement in the low temperature gradient cases where $K_n = 1.0$ (runs 2 and 4). The velocity profiles, however, which are presented in *Figure 3-6* and *Figure 3-7*, do not exhibit the same close agreement observed in the number density and temperature profiles. Here differences of up to 20% are observed near the thermal boundaries. In both of these cases the Denery velocity profiles indicate *more slip* near the boundaries than the MDC predictions.

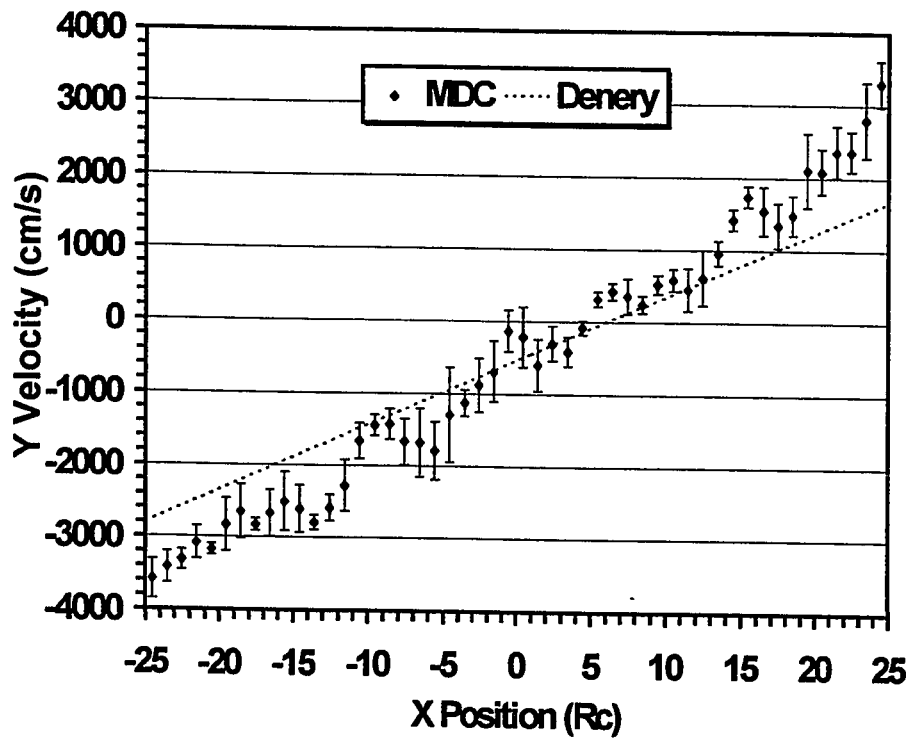


Figure 3-6: Average particle velocity (component parallel to thermal walls) as a function of position for run 2 where $T_H/T_C = 1.5$, $K_n = 1.0$, and $M_a = 0.5$.

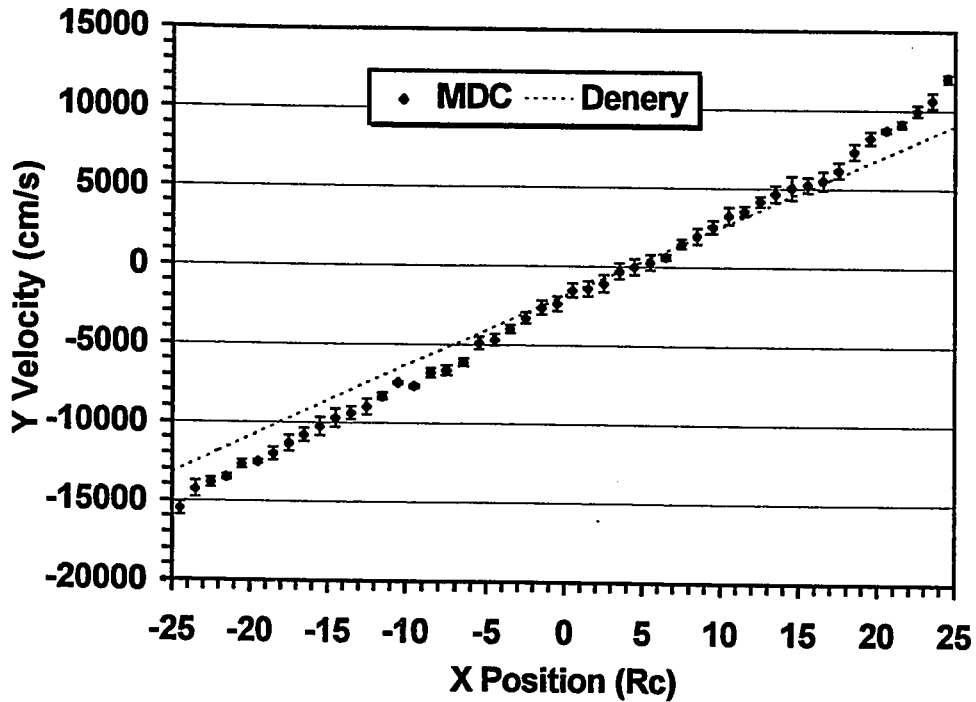


Figure 3-7: Average particle velocity (component parallel to thermal walls) as a function of position for run 4 where $T_H/T_C = 1.5$, $K_n = 1.0$, and $M_a = 2.0$.

Most of the differences between the two models are observed in the high temperature gradient cases (runs 5, 6, 7, and 8). Density differences (typically at the thermal boundaries) of up to 12% are observed for the high temperature gradient regime cases. The differences observed in the density profiles for runs 5 and 6, which are presented in *Figure 3-8* and *Figure 3-9* respectively, are representative of this trend.

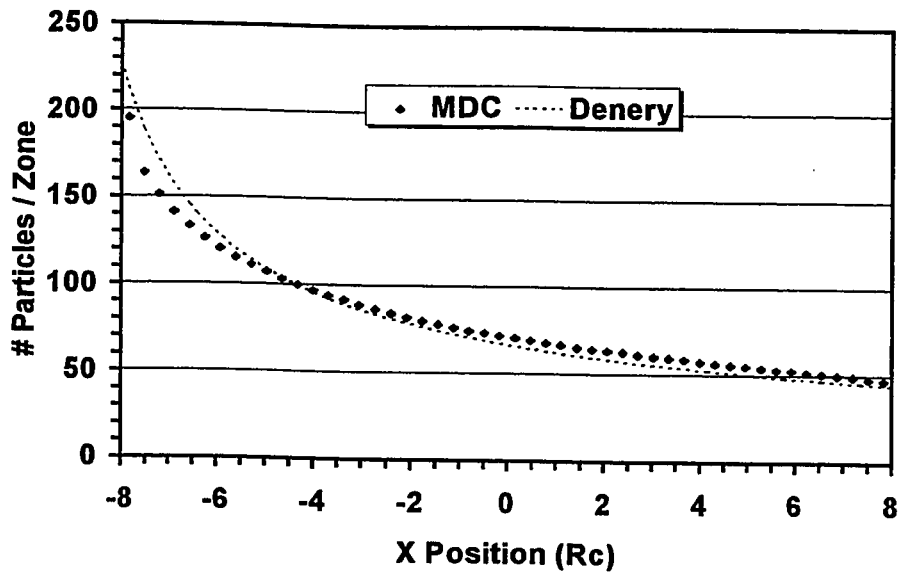


Figure 3-8: Average number of particles per zone as a function of position for run 5 where $T_H/T_C = 10.0$, $K_n = 0.1$, and $M_a = 0.5$.

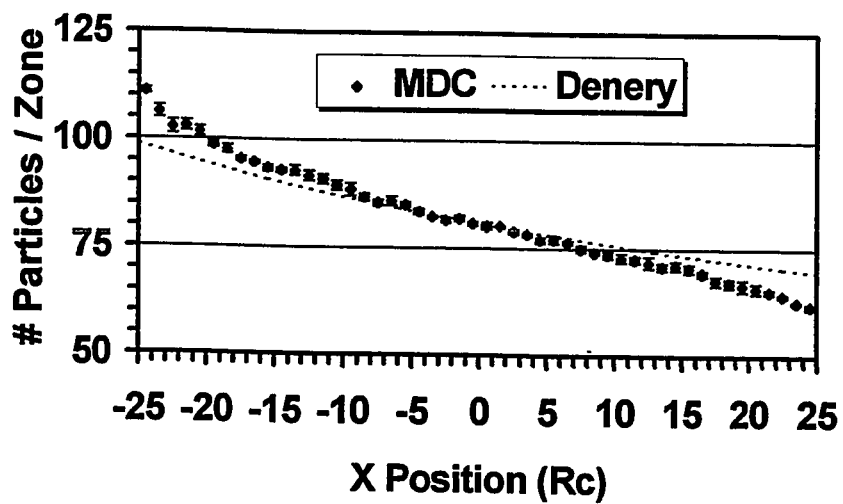


Figure 3-9: Average number of particles per zone as a function of position for run 6 where $T_H/T_C = 10.0$, $K_n = 1.0$, and $M_a = 0.5$.

The velocity profiles for $K_n = 1.0$ in the high temperature regime (runs 6 and 8) show the same trend as observed for the $K_n = 1.0$ cases in the low temperature gradient regime (runs 2 and 4 shown in *Figure 3-6* and *Figure 3-7*). In the high temperature gradient cases the Denery model predicts a lower velocity than MDC by up to 20% at the *cold* (left) thermal boundary and a lower velocity by up to 50% at the *hot* (right) boundary. This suggests that in the $K_n = 1.0$ regime, in general, there is a stronger coupling of momentum transfer between the thermal walls and the particles in the MDC model than in the Denery model (i.e., Denery's model consistently predicts a larger velocity slip at the walls). *Figure 3-10*, which shows the velocity profiles for run 8, provides further evidence of this trend.

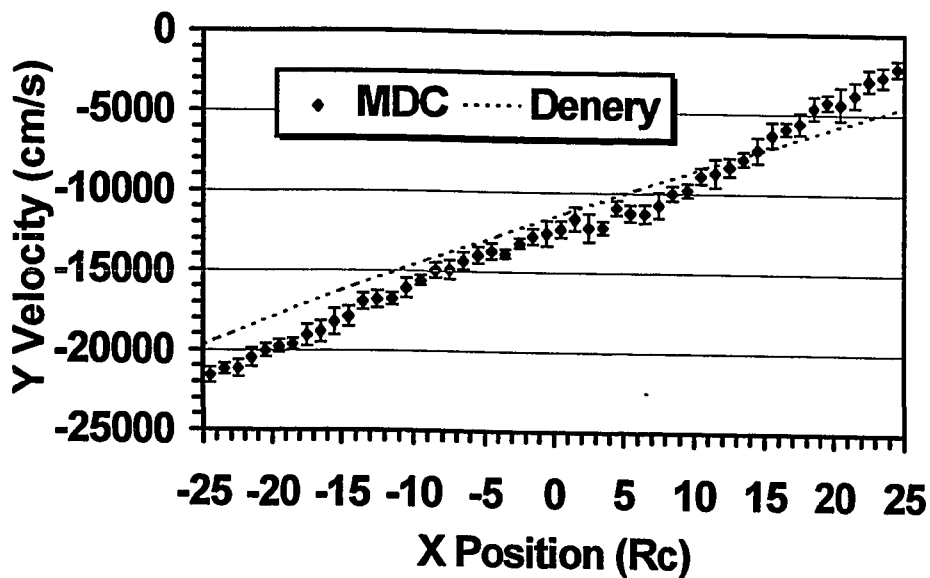


Figure 3-10: Average particle velocity (component parallel to thermal walls) as a function of position for run 8 where $T_H/T_C = 10.0$, $K_n = 1.0$, and $M_a = 2.0$.

It is interesting to observe the fluid velocity profiles near the hot (right) thermal wall in *Figure 3-10*. The average fluid velocity is actually downward in the zone closest to this wall despite the significant upward motion of the wall (+30,800 cm/s). This effect is explained by noting that, in this environment, the steady state number density is almost a factor of two higher near the cold wall than it is near the hot wall due to the steep temperature gradient created by the respective temperature differences between the two walls. The higher particle density near the cold wall allows a higher momentum transfer (i.e., a stronger coupling) between the cold wall and the fluid than is occurring between the hot wall and the fluid. As a result a *net downward* momentum is transferred to the fluid and creates the velocity effect observed in *Figure 3-10*.

The temperature profiles for the $K_n = 1.0$ (runs 2, 4, 6, and 8) regime show a trend consistent with the trend in the corresponding velocity profiles just mentioned. The Denery model predicts a lower temperature than MDC by up to 20% at the *cold* (left) thermal boundary and a lower temperature by up to 12% at the *hot* (right) boundary. The largest disagreements of this type are observed in the high temperature gradient regime (runs 6 and 8) and are shown in *Figure 3-11* and *Figure 3-12* respectively.

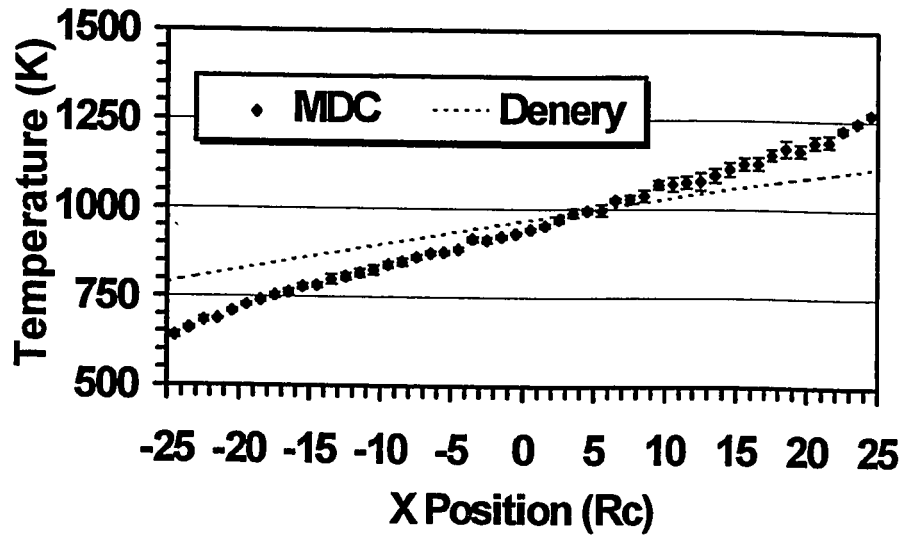


Figure 3-11: Average temperature as a function of position for run 6 where $T_H/T_C = 10.0$, $K_n = 1.0$, and $M_a = 0.5$.

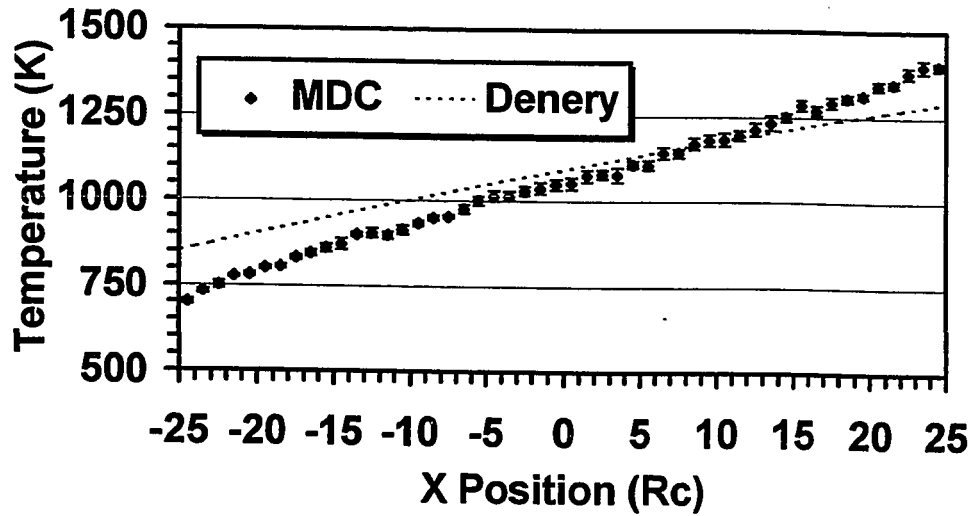


Figure 3-12: Average temperature as a function of position for run 8 where $T_H/T_C = 10.0$, $K_n = 1.0$, and $M_a = 2.0$.

The temperature profiles from run 5 given in *Figure 3-13* show a different behavior than what is observed for runs 6 and 8. In this case, the compared results are in close agreement near the thermal walls but differ by up to 10% in the interior regions of the channel.

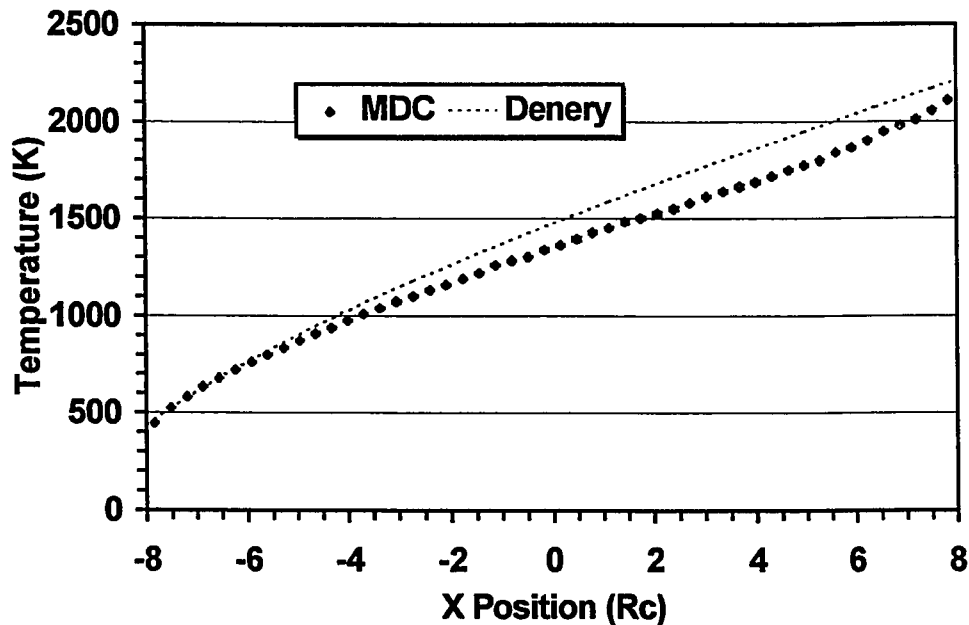


Figure 3-13: Average temperature as a function of position for run 5 where $T_H/T_C = 10.0$, $K_n = 0.1$, and $M_a = 0.5$.

All of the results obtained by the MDC simulations are in general agreement with numerical results obtained by Denery using a Monte Carlo simulation of hard sphere particles.

Summary

The MDC simulation computer program was written to model general planar Couette flow. After rigorous validation of the MDC model, a comparison of MDC results to results produced by Denery's semi-analytical model was performed for Couette flow scenarios in the Knudsen number range 0.1 to 1.0. Comparison of predicted fluid properties including number density, velocity, and temperature showed relatively close agreement between the Denery and MDC models in the 0.1 Knudsen number regime when the temperature gradient across the channel was small. Increasing the temperature gradient across the channel in this Knudsen number regime produced observable differences in the temperature and particle velocity profiles between the two respective models.

More significant differences between the two models were observed in simulations of the 1.0 Knudsen number regime. In this Knudsen number range, differences between the Denery theory and MD simulation were observed in both the small and large temperature gradient cases. A common trend in these cases was that the Denery theory consistently showed more *slip* in the fluid velocity profiles near the thermal boundaries than was observed in the MD predictions. Density and temperature profiles in this Knudsen number regime also showed consistent differences near the boundaries between the two respective models. These observations suggest that the two-sided Maxwellian velocity distribution function assumed by the Denery theory may not be an appropriate representation of channel fluid flow in Knudsen number ranges near 1.0.

Future work in this area might include the study of slip velocity and the dependency of slip velocity on temperature gradient. Another useful extension to this work is the study of Couette flow in concentric cylinders where curvature effects have been predicted to give anomalous results.

Appendix A

Forward Neighbor Arrays

Given N_x cells in the x direction the array IP ($i+$) is initialized as follows:

$$ip_n = n + 1, \quad \text{for } n = 1, 2, \dots, N_x - 1$$

and,

$$ip_n = 1, \quad \text{for } n = N_x \quad (\text{periodic boundaries}),$$

or,

$$ip_n = N_x + 1, \quad \text{for } n = N_x \quad (\text{thermal boundaries}).$$

Equations A-1

Similarly the array IM ($i-$) is defined by:

$$im_n = N_x \quad \text{for } n = 1, \quad (\text{periodic boundaries}),$$

or,

$$im_n = N_x + 1 \quad \text{for } n = 1, \quad (\text{thermal boundaries}),$$

and,

$$im_n = n - 1 \quad \text{for } n = 2, 3, \dots, N_x .$$

Equations A-2

Analogous arrays JP , JM , and KP are defined for N_y cells in the y and N_z cells in the z directions respectively. There are only periodic boundaries in these directions because the MDC model provides for thermal boundaries in one direction only. Also note that a KM type array is not needed in the definition of forward neighbor cells. After creating

the IP and IM arrays they are combined into a $(13 \times N_x)$ matrix IFN . The matrix IFN is used to define forward neighbor cells in the x direction and is given by:

$$\begin{aligned}
& (ifn_{1,n} = ip_n \quad \text{for } n = 1, 2, \dots, N_x), \\
& (ifn_{2,n} = im_n \quad \text{for } n = 1, 2, \dots, N_x), \\
& (ifn_{3,n} = n \quad \text{for } n = 1, 2, \dots, N_x), \\
& (ifn_{4,n} = ip_n \quad \text{for } n = 1, 2, \dots, N_x), \\
& (ifn_{5,n} = im_n \quad \text{for } n = 1, 2, \dots, N_x), \\
& (ifn_{6,n} = n \quad \text{for } n = 1, 2, \dots, N_x), \\
& (ifn_{7,n} = ip_n \quad \text{for } n = 1, 2, \dots, N_x), \\
& (ifn_{8,n} = im_n \quad \text{for } n = 1, 2, \dots, N_x), \\
& (ifn_{9,n} = n \quad \text{for } n = 1, 2, \dots, N_x), \\
& (ifn_{10,n} = ip_n \quad \text{for } n = 1, 2, \dots, N_x), \\
& (ifn_{11,n} = im_n \quad \text{for } n = 1, 2, \dots, N_x), \\
& (ifn_{12,n} = n \quad \text{for } n = 1, 2, \dots, N_x), \\
& (ifn_{13,n} = ip_n \quad \text{for } n = 1, 2, \dots, N_x).
\end{aligned}$$

Equations A-3: Forward neighbor index matrix $IFN_{m,n}$.

Analogously, the JP and JM arrays are combined into a $(13 \times N_y)$ matrix JFN . The matrix JFN is used to define forward neighbor cells in the y direction and is given by:

$$\begin{aligned}
& (jfn_{1,n} = n \quad \text{for } n = 1, 2, \dots, N_y), \\
& (jfn_{2,n} = jp_n \quad \text{for } n = 1, 2, \dots, N_y), \\
& (jfn_{3,n} = jp_n \quad \text{for } n = 1, 2, \dots, N_y), \\
& (jfn_{4,n} = jp_n \quad \text{for } n = 1, 2, \dots, N_y), \\
& (jfn_{5,n} = jm_n \quad \text{for } n = 1, 2, \dots, N_y), \\
& (jfn_{6,n} = jm_n \quad \text{for } n = 1, 2, \dots, N_y), \\
& (jfn_{7,n} = jm_n \quad \text{for } n = 1, 2, \dots, N_y),
\end{aligned}$$

$$\begin{aligned}
& (jfn_{8,n} = n && \text{for } n = 1, 2, \dots, N_y), \\
& (jfn_{9,n} = n && \text{for } n = 1, 2, \dots, N_y), \\
& (jfn_{10,n} = n && \text{for } n = 1, 2, \dots, N_y), \\
& (jfn_{11,n} = jp_n && \text{for } n = 1, 2, \dots, N_y), \\
& (jfn_{12,n} = jp_n && \text{for } n = 1, 2, \dots, N_y), \\
& (jfn_{13,n} = jp_n && \text{for } n = 1, 2, \dots, N_y).
\end{aligned}$$

Equations A-4: Forward neighbor index matrix $JFN_{m,n}$.

Note that the JFN matrix is composed somewhat differently than the IFN matrix. The KP array is used to build a $(13 \times N_z)$ matrix KFN . The matrix KFN is used to define forward neighbor cells in the z direction and is given by:

$$\begin{aligned}
& (kfn_{1,n} = n && \text{for } n = 1, 2, \dots, N_z), \\
& (kfn_{2,n} = n && \text{for } n = 1, 2, \dots, N_z), \\
& (kfn_{3,n} = n && \text{for } n = 1, 2, \dots, N_z), \\
& (kfn_{4,n} = n && \text{for } n = 1, 2, \dots, N_z), \\
& (kfn_{5,n} = kp_n && \text{for } n = 1, 2, \dots, N_z), \\
& (kfn_{6,n} = kp_n && \text{for } n = 1, 2, \dots, N_z), \\
& (kfn_{7,n} = kp_n && \text{for } n = 1, 2, \dots, N_z), \\
& (kfn_{8,n} = kp_n && \text{for } n = 1, 2, \dots, N_z), \\
& (kfn_{9,n} = kp_n && \text{for } n = 1, 2, \dots, N_z), \\
& (kfn_{10,n} = kp_n && \text{for } n = 1, 2, \dots, N_z), \\
& (kfn_{11,n} = kp_n && \text{for } n = 1, 2, \dots, N_z), \\
& (kfn_{12,n} = kp_n && \text{for } n = 1, 2, \dots, N_z), \\
& (kfn_{13,n} = kp_n && \text{for } n = 1, 2, \dots, N_z).
\end{aligned}$$

Equations A-5: Forward neighbor index matrix $KFN_{m,n}$.

In addition to the index arrays used to define forward neighbor cells, arrays to aid in the determination of inter-particle distances at a periodic boundaries are also initialized before starting the main calculation. Given a length in the x direction l_x subdivided into N_x cells the array DXP ($dx+$) is initialized as follows:

$$dxp_n = 0, \quad \text{for } n = 1, 2, \dots, N_x-1$$

and,

$$dxp_n = l_x, \quad \text{for } n = N_x \quad (\text{periodic boundaries}),$$

or,

$$dxp_n = 0, \quad \text{for } n = N_x \quad (\text{thermal boundaries}).$$

Equation A-6

Similarly the array DXM ($dx-$) is defined by:

$$dxm_n = -l_x \quad \text{for } n = 1, \quad (\text{periodic boundaries}),$$

or,

$$dxm_n = 0 \quad \text{for } n = 1, \quad (\text{thermal boundaries}),$$

and,

$$dxm_n = 0 \quad \text{for } n = 2, 3, \dots, N_x.$$

Equations A-7

Analogous arrays DYP , DYM , and DZP are defined for a length l_y in the y direction divided into N_y cells and a length l_z in the z direction divided into N_z cells respectively. Although these arrays are basically all zeros with the exception of the first or last entry, they will prove to be useful in calculating inter-particle distances at the boundaries of the

system. Just as the IP, IM, JP, JM, and KP arrays were combined into two dimensional forward neighbor cell arrays the DXP, DXM, DYP, DYM, and DZP arrays are also combined into forward neighbor cell arrays. The matrix $(13 \times N_x)$ matrix $DXFN$ is defined by:

$$\begin{aligned}
 (dxfn_{1,n} = dxp_n & \quad \text{for } n = 1, 2, \dots, N_x), \\
 (dxfn_{2,n} = dxm_n & \quad \text{for } n = 1, 2, \dots, N_x), \\
 (dxfn_{3,n} = 0 & \quad \text{for } n = 1, 2, \dots, N_x), \\
 (dxfn_{4,n} = dxp_n & \quad \text{for } n = 1, 2, \dots, N_x), \\
 (dxfn_{5,n} = dxm_n & \quad \text{for } n = 1, 2, \dots, N_x), \\
 (dxfn_{6,n} = 0 & \quad \text{for } n = 1, 2, \dots, N_x), \\
 (dxfn_{7,n} = dxp_n & \quad \text{for } n = 1, 2, \dots, N_x), \\
 (dxfn_{8,n} = dxm_n & \quad \text{for } n = 1, 2, \dots, N_x), \\
 (dxfn_{9,n} = 0 & \quad \text{for } n = 1, 2, \dots, N_x), \\
 (dxfn_{10,n} = dxp_n & \quad \text{for } n = 1, 2, \dots, N_x), \\
 (dxfn_{11,n} = dxm_n & \quad \text{for } n = 1, 2, \dots, N_x), \\
 (dxfn_{12,n} = 0 & \quad \text{for } n = 1, 2, \dots, N_x), \\
 (dxfn_{13,n} = dxp_n & \quad \text{for } n = 1, 2, \dots, N_x).
 \end{aligned}$$

Equations A-8: Forward neighbor cell delta X matrix $DXFN_{m,n}$.

Note the similarity between the construction of *Equations A-3* and the construction of *Equations A-8*. The matrix $(13 \times N_y)$ matrix $DYFN$ is defined by:

$$\begin{aligned}
 (dyfn_{1,n} = 0 & \quad \text{for } n = 1, 2, \dots, N_y), \\
 (dyfn_{2,n} = dyp_n & \quad \text{for } n = 1, 2, \dots, N_y), \\
 (dyfn_{3,n} = dyp_n & \quad \text{for } n = 1, 2, \dots, N_y), \\
 (dyfn_{4,n} = dyp_n & \quad \text{for } n = 1, 2, \dots, N_y),
 \end{aligned}$$

$$\begin{aligned}
& (dyfn_{5,n} = dym_n \quad \text{for } n = 1, 2, \dots, N_y), \\
& (dyfn_{6,n} = dym_n \quad \text{for } n = 1, 2, \dots, N_y), \\
& (dyfn_{7,n} = dym_n \quad \text{for } n = 1, 2, \dots, N_y), \\
& (dyfn_{8,n} = 0 \quad \text{for } n = 1, 2, \dots, N_y), \\
& (dyfn_{9,n} = 0 \quad \text{for } n = 1, 2, \dots, N_y), \\
& (dyfn_{10,n} = 0 \quad \text{for } n = 1, 2, \dots, N_y), \\
& (dyfn_{11,n} = dyp_n \quad \text{for } n = 1, 2, \dots, N_y), \\
& (dyfn_{12,n} = dyp_n \quad \text{for } n = 1, 2, \dots, N_y), \\
& (dyfn_{13,n} = dyp_n \quad \text{for } n = 1, 2, \dots, N_y).
\end{aligned}$$

Equations A-9: Forward neighbor cell delta Y matrix DYFN_{m,n}.

Again, note the similarity between the construction of *Equations A-4* and the construction of *Equations A-9*. Finally, the matrix $(13 \times N_z)$ matrix *DZFN* is defined by:

$$\begin{aligned}
& (dzfn_{1,n} = 0 \quad \text{for } n = 1, 2, \dots, N_z), \\
& (dzfn_{2,n} = 0 \quad \text{for } n = 1, 2, \dots, N_z), \\
& (dzfn_{3,n} = 0 \quad \text{for } n = 1, 2, \dots, N_z), \\
& (dzfn_{4,n} = 0 \quad \text{for } n = 1, 2, \dots, N_z), \\
& (dzfn_{5,n} = dzp_n \quad \text{for } n = 1, 2, \dots, N_z), \\
& (dzfn_{6,n} = dzp_n \quad \text{for } n = 1, 2, \dots, N_z), \\
& (dzfn_{7,n} = dzp_n \quad \text{for } n = 1, 2, \dots, N_z), \\
& (dzfn_{8,n} = dzp_n \quad \text{for } n = 1, 2, \dots, N_z), \\
& (dzfn_{9,n} = dzp_n \quad \text{for } n = 1, 2, \dots, N_z), \\
& (dzfn_{10,n} = dzp_n \quad \text{for } n = 1, 2, \dots, N_z), \\
& (dzfn_{11,n} = dzp_n \quad \text{for } n = 1, 2, \dots, N_z), \\
& (dzfn_{12,n} = dzp_n \quad \text{for } n = 1, 2, \dots, N_z),
\end{aligned}$$

$$(dzfn_{13,n} = dzp_n \quad \text{for } n = 1, 2, \dots, N_z).$$

Equations A-10: Forward neighbor delta Z matrix DZFN_{m,n}.

Comparison of these equations to *Equations A-5* will show a strong resemblance.

Appendix B

Lennard-Jones Plots

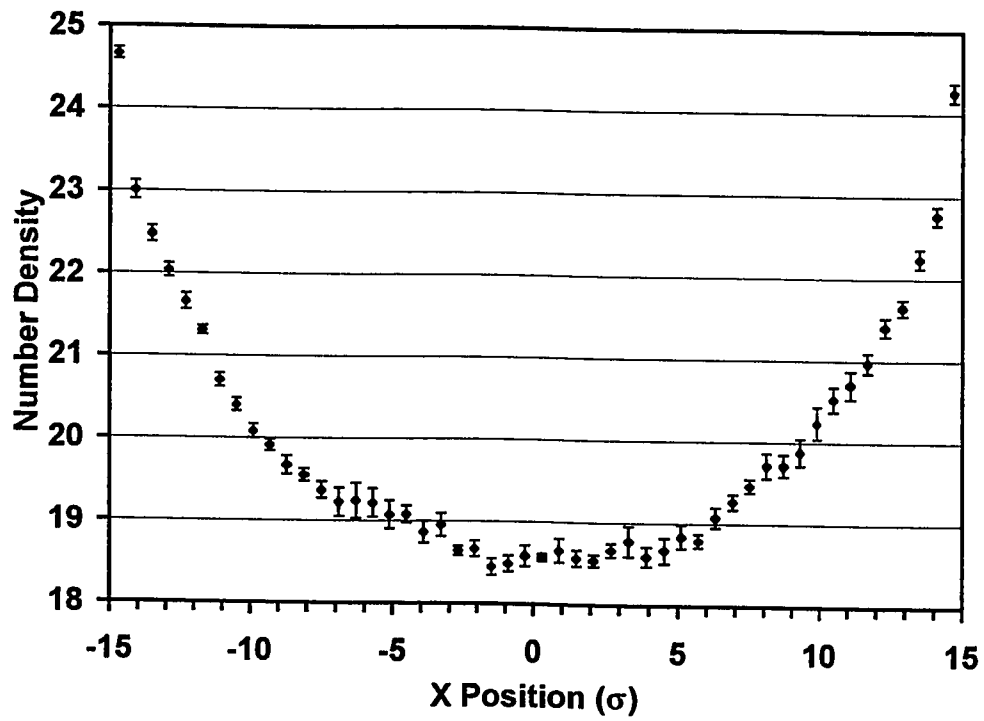


Figure B-1: Number density vs. position for Poiseuille comparison to Morris' run 6.

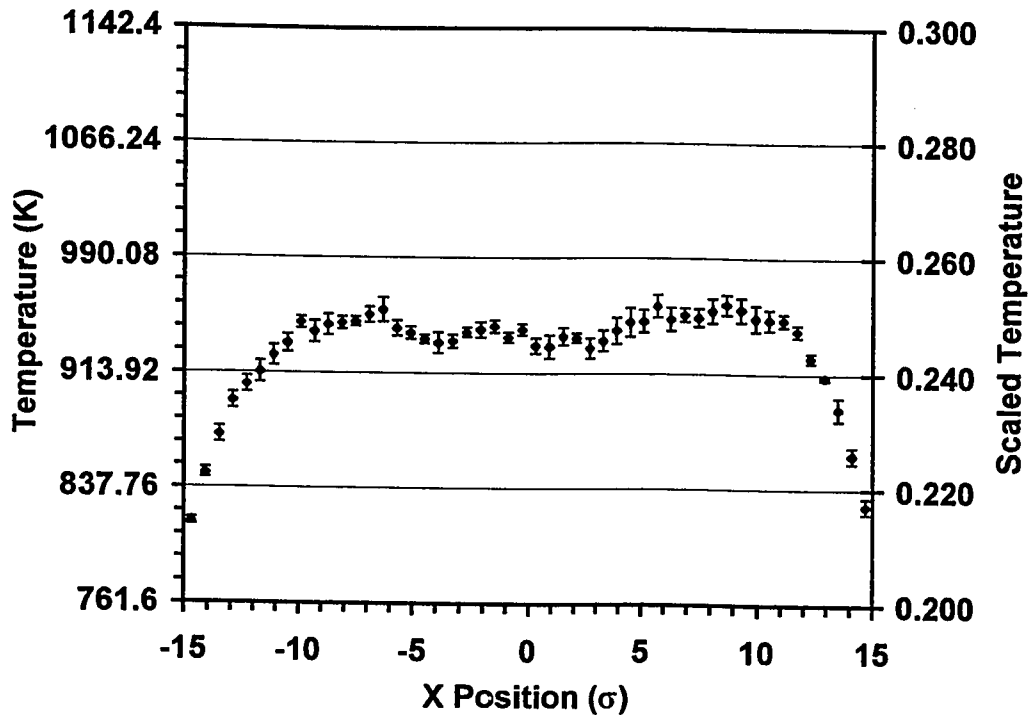


Figure B-2: Temperature vs. position for Poiseuille comparison to Morris' run 6. Scaled units are CGS divided by 3808K.

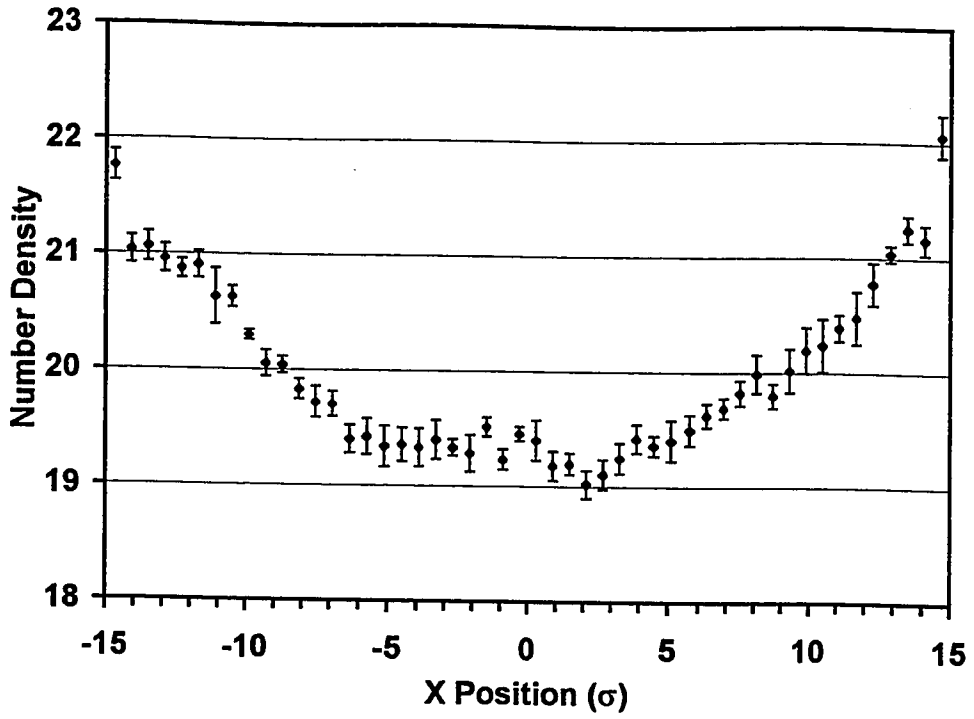


Figure B-3: Number density vs. position for Couette comparison to Morris' run 6.

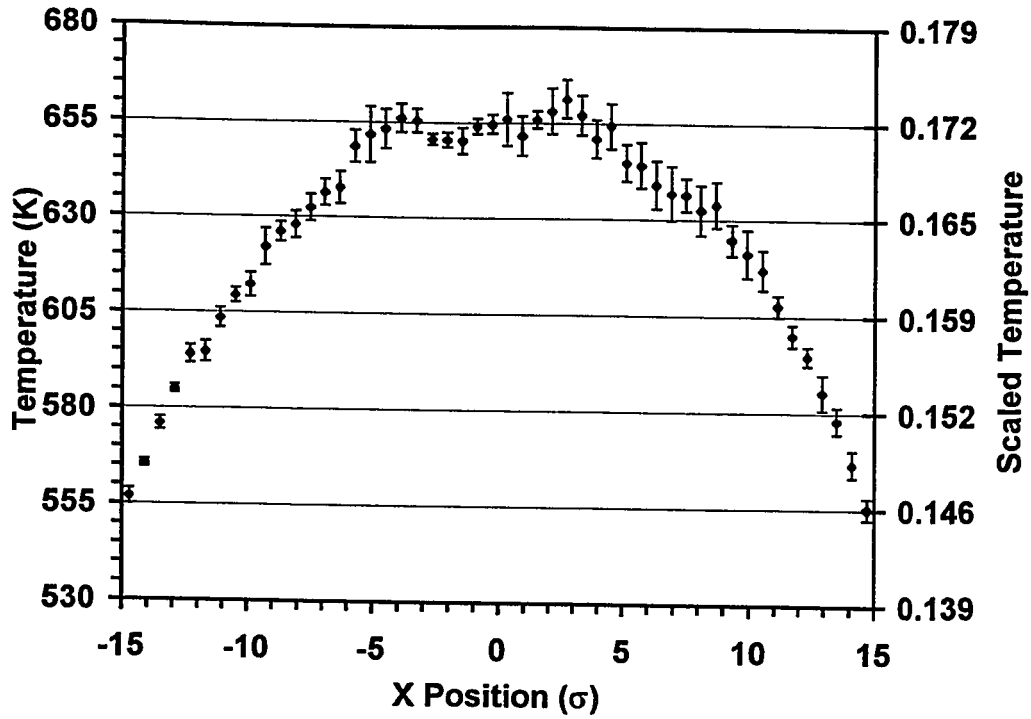


Figure B-4: Temperature vs. position for Couette comparison to Morris' run 6. Scaled units are CGS divided by 3808K.

Appendix C

Denery vs. MDC Simulation Plots

Profiles for Run 1:

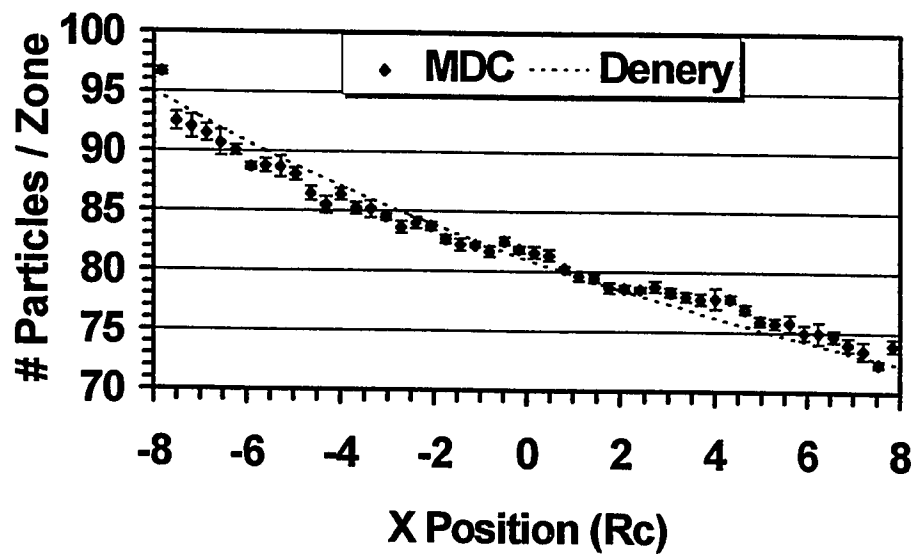


Figure C-1: Average number of particles per zone as a function of position for run 1 where $T_H/T_C = 1.5$, $K_n = 0.1$, and $M_a = 0.5$.

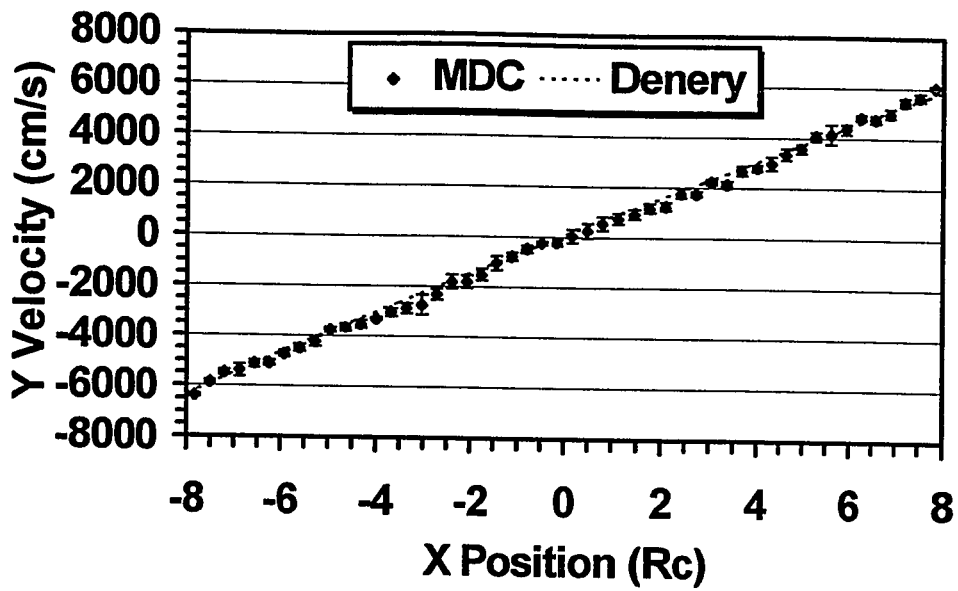


Figure C-2: Average particle velocity (component parallel to thermal walls) as a function of position for run 1 where $T_H/T_C = 1.5$, $K_n = 0.1$, and $Ma = 0.5$.

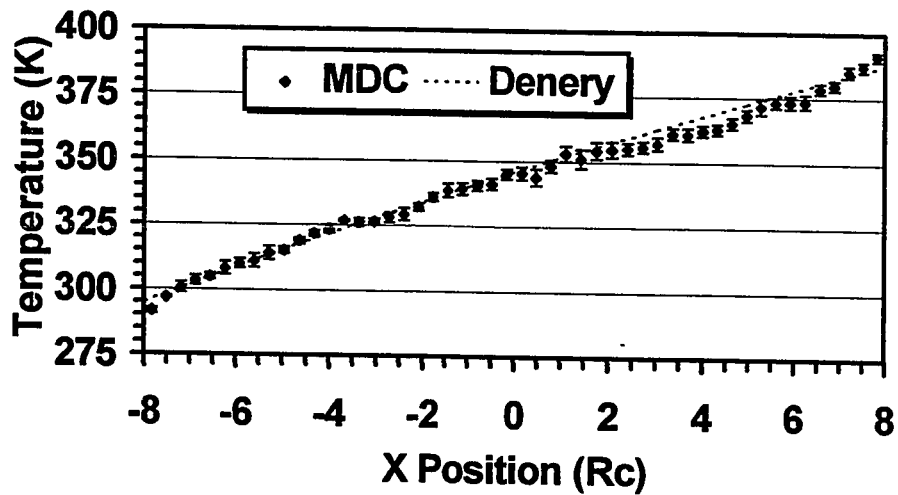


Figure C-3: Average temperature as a function of position for run 1 where $T_H/T_C = 1.5$, $K_n = 0.1$, and $Ma = 0.5$.

Profiles for Run 2:

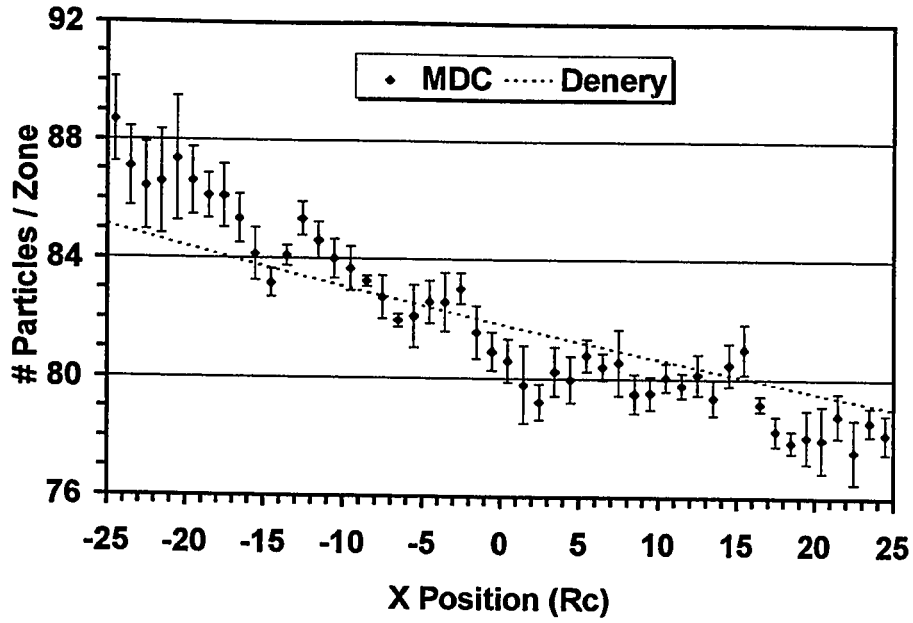


Figure C-4: Average number of particles per zone as a function of position for run 2 where $T_H/T_C = 1.5$, $K_n = 1.0$, and $M_a = 0.5$.

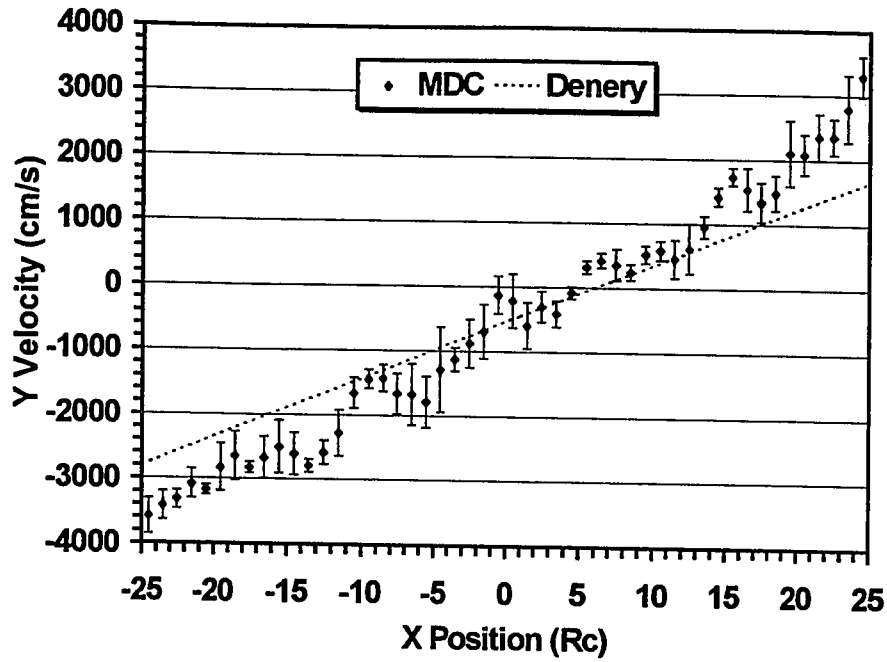


Figure C-5: Average particle velocity (component parallel to thermal walls) as a function of position for run 2 where $T_H/T_C = 1.5$, $K_n = 1.0$, and $Ma = 0.5$.

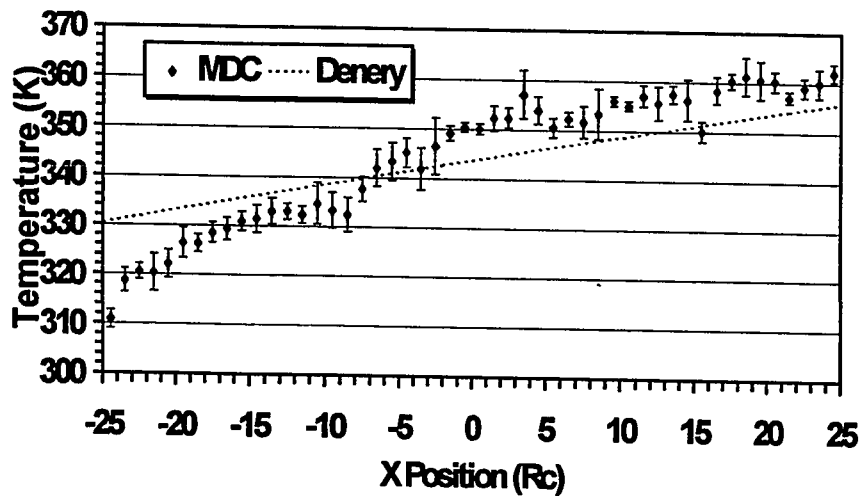


Figure C-6: Average temperature as a function of position for run 2 where $T_H/T_C = 1.5$, $K_n = 1.0$, and $Ma = 0.5$.

Profiles for Run 3:

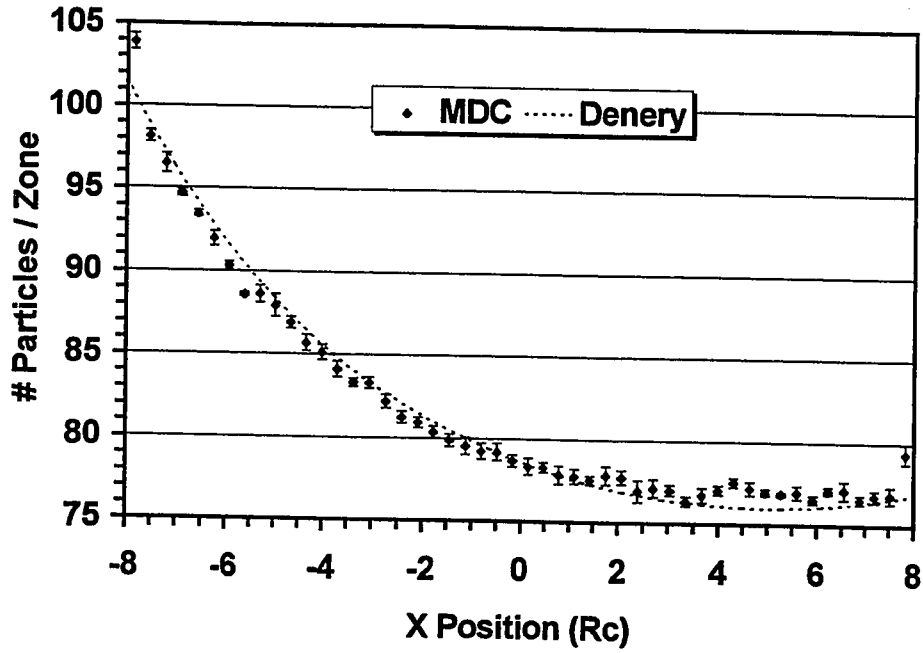


Figure C-7: Average number of particles per zone as a function of position for run 3 where $T_H/T_C = 1.5$, $K_n = 0.1$, and $M_a = 2.0$.

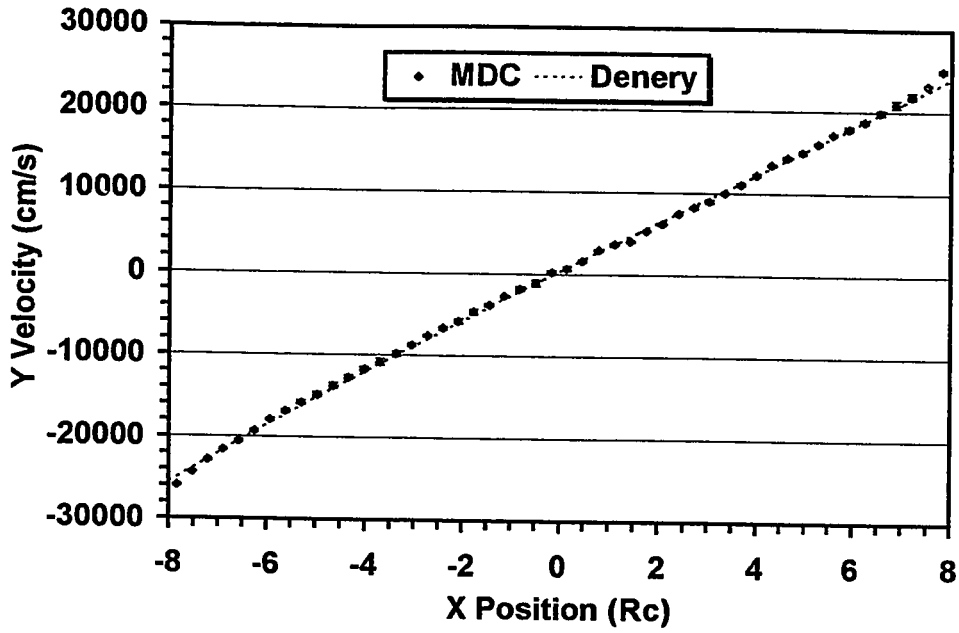


Figure C-8: Average particle velocity (component parallel to thermal walls) as a function of position for run 3 where $T_H/T_C = 1.5$, $K_n = 0.1$, and $M_a = 2.0$.

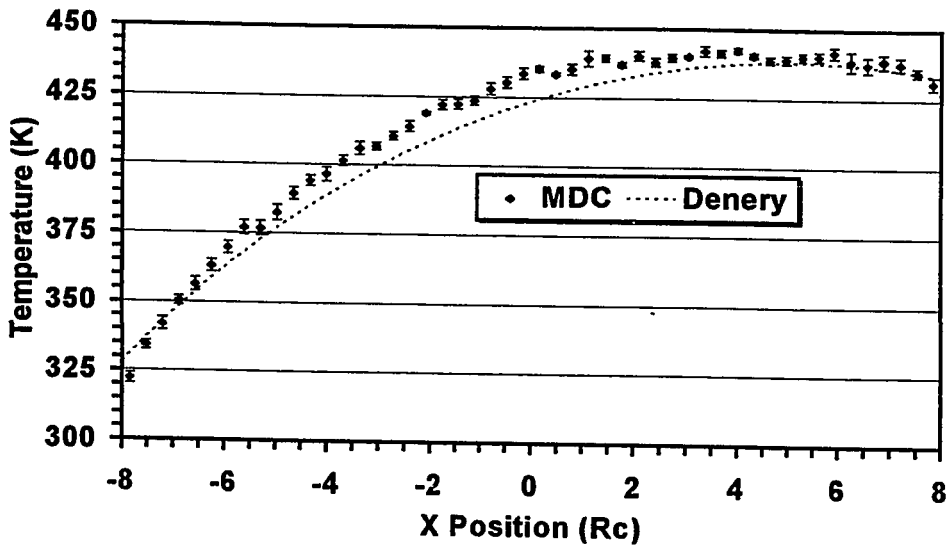


Figure C-9: Average temperature as a function of position for run 3 where $T_H/T_C = 1.5$, $K_n = 0.1$, and $M_a = 2.0$.

Profiles for Run 4:

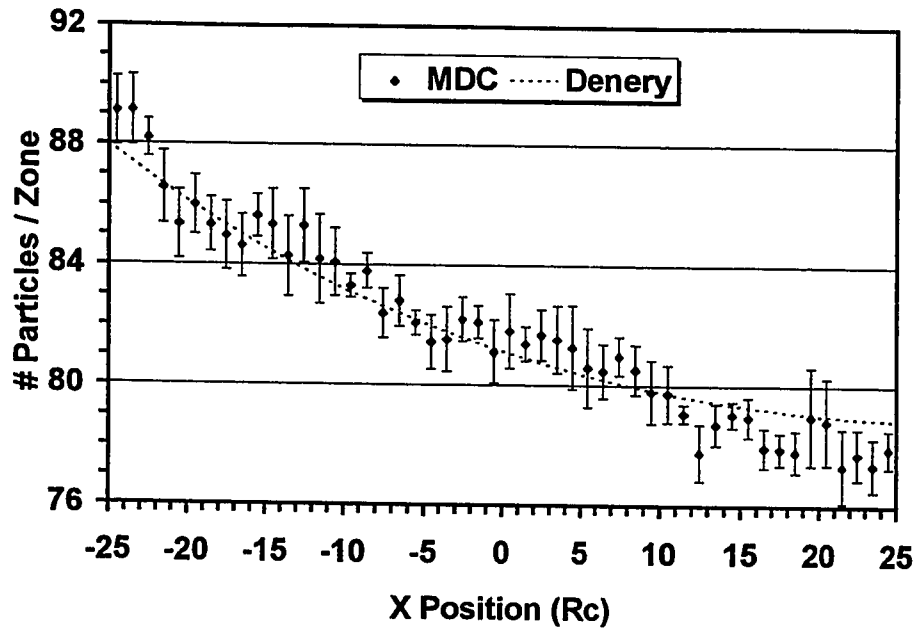


Figure C-10: Average number of particles per zone as a function of position for run 4 where $T_W/T_C = 1.5$, $K_n = 1.0$, and $M_a = 2.0$.

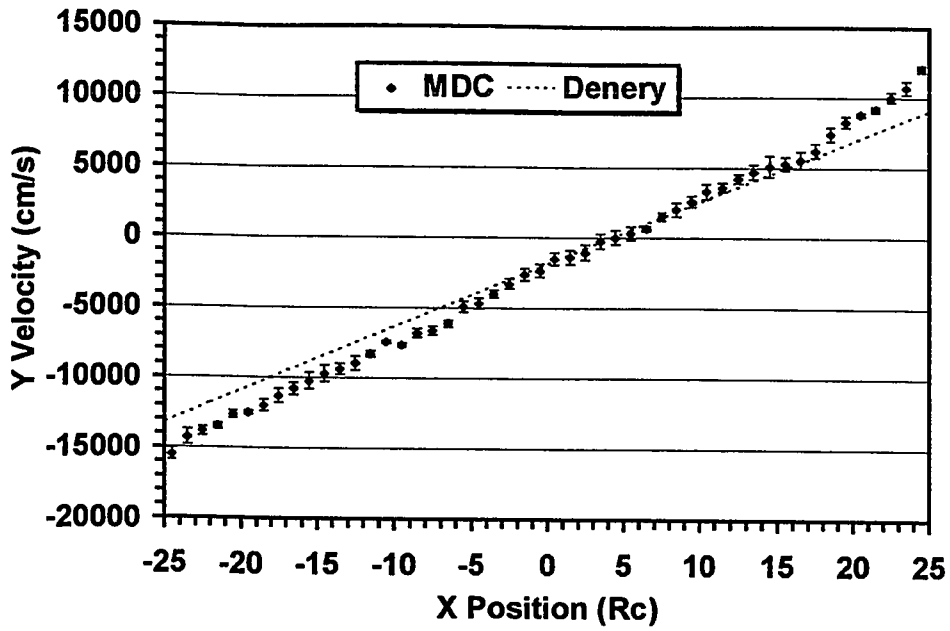


Figure C-11: Average particle velocity (component parallel to thermal walls) as a function of position for run 4 where $T_H/T_C = 1.5$, $K_n = 1.0$, and $M_a = 2.0$.

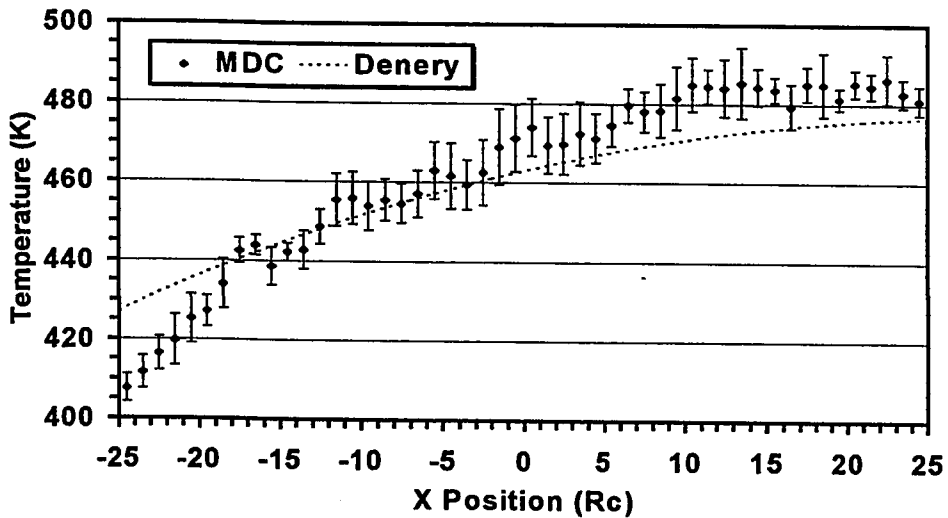


Figure C-12: Average temperature as a function of position for run 4 where $T_H/T_C = 1.5$, $K_n = 1.0$, and $M_a = 2.0$.

Profiles for Run 5:

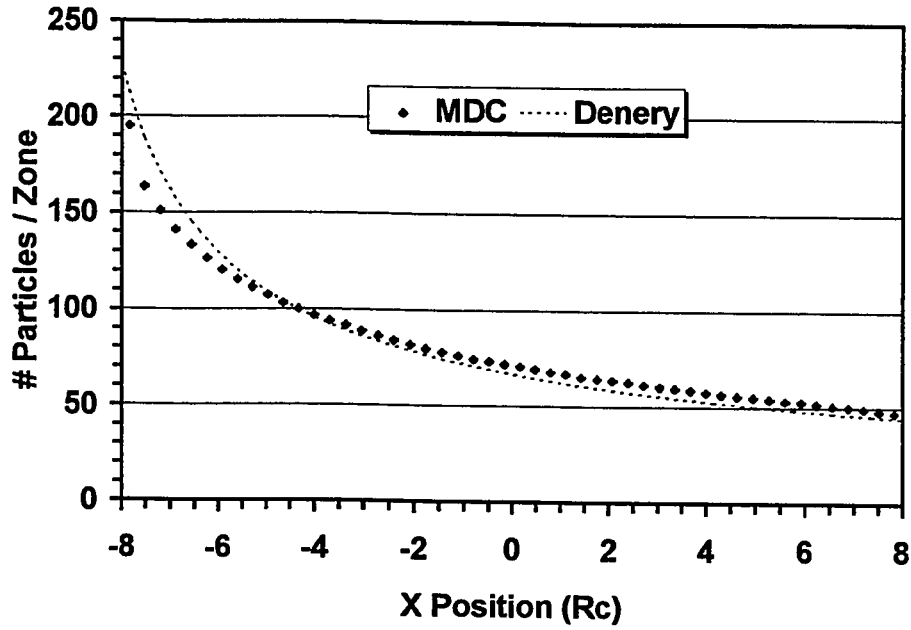


Figure C-13: Average number of particles per zone as a function of position for run 5 where $T_H/T_C = 10.0$, $K_n = 0.1$, and $M_a = 0.5$.

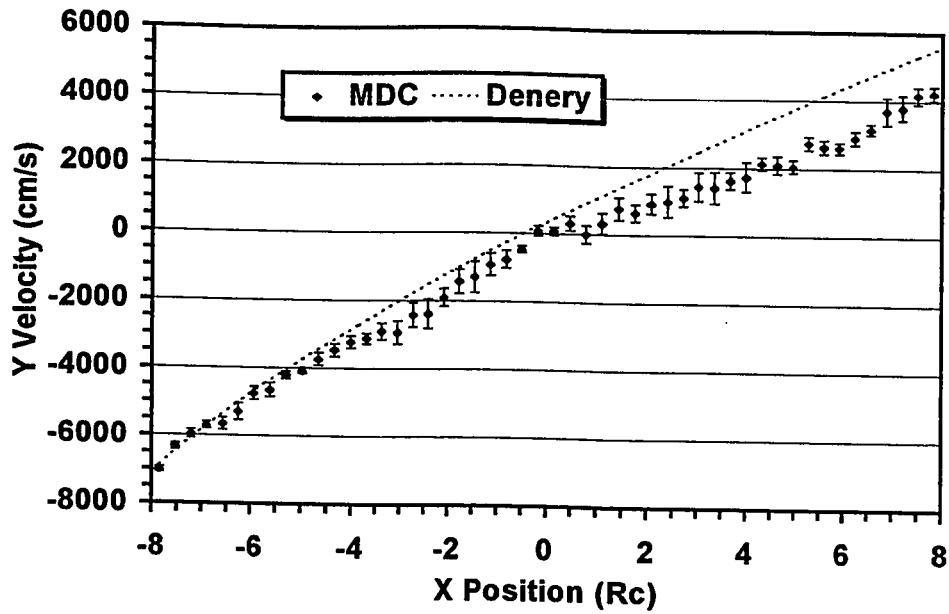


Figure C-14: Average particle velocity (component parallel to thermal walls) as a function of position for run 5 where $T_H/T_C = 10.0$, $K_n = 0.1$, and $Ma = 0.5$.

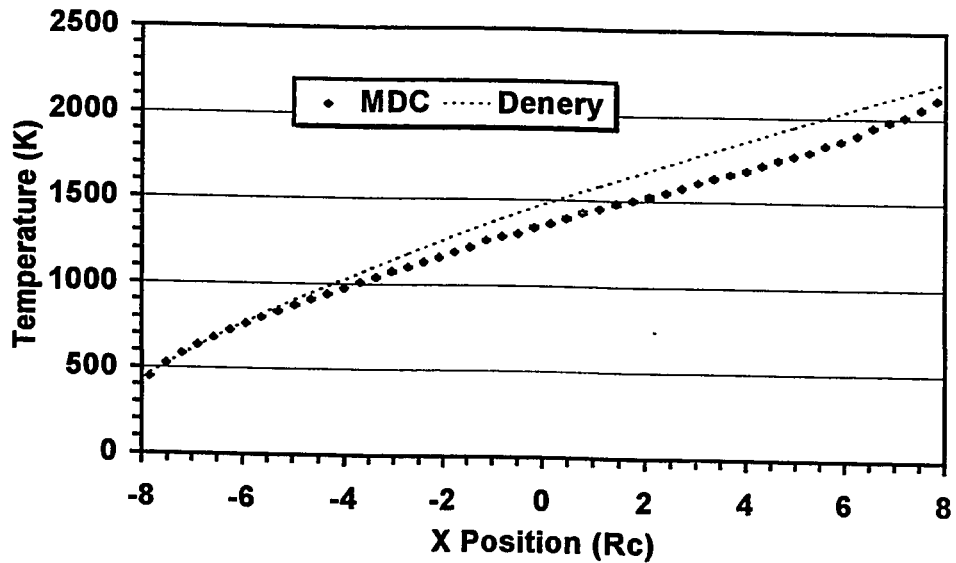


Figure C-15: Average temperature as a function of position for run 5 where $T_H/T_C = 10.0$, $K_n = 0.1$, and $Ma = 0.5$.

Profiles for Run 6:

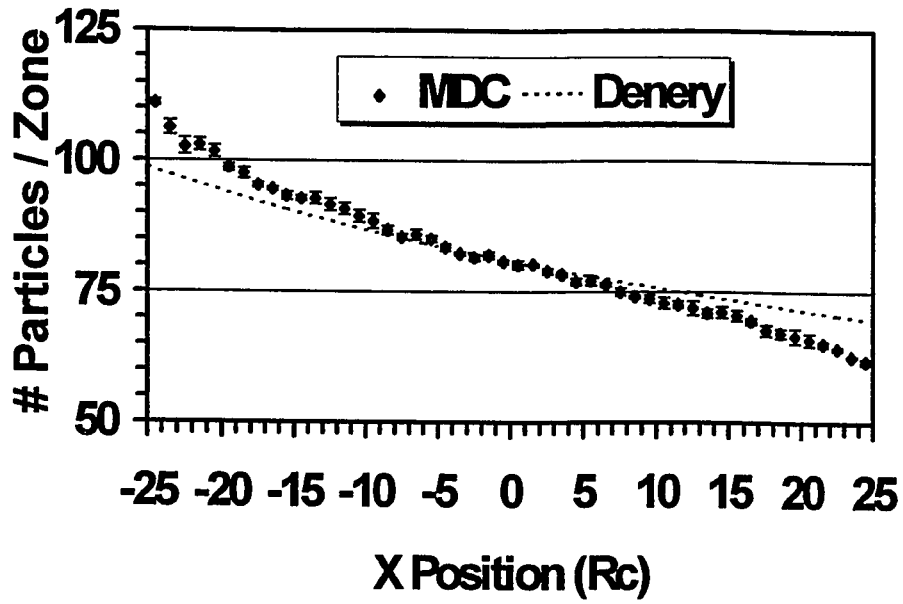


Figure C-16: Average number of particles per zone as a function of position for run 6 where $T_H/T_C = 10.0$, $K_n = 1.0$, and $M_a = 0.5$.

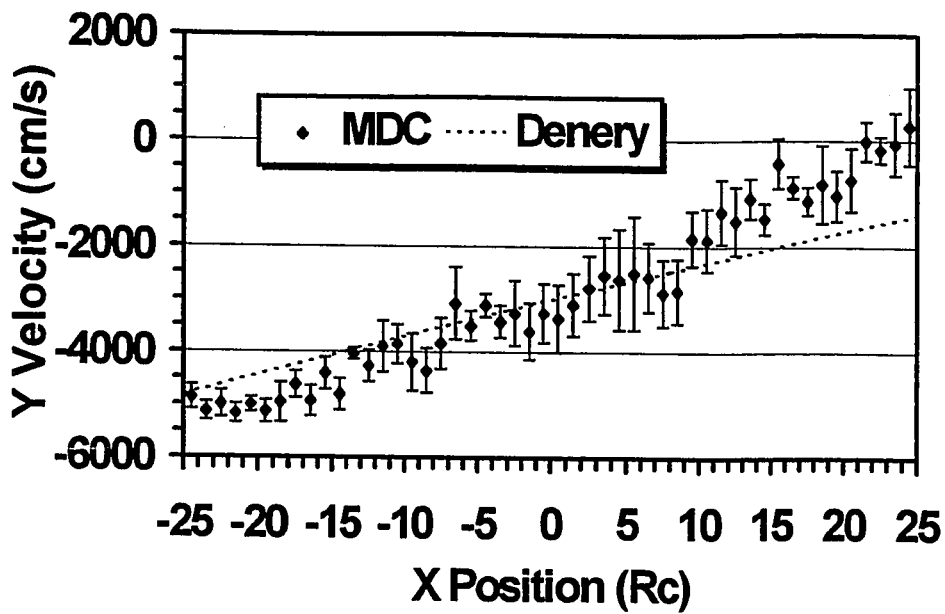


Figure C-17: Average particle velocity (component parallel to thermal walls) as a function of position for run 6 where $T_H/T_C = 10.0$, $K_n = 1.0$, and $M_a = 0.5$.

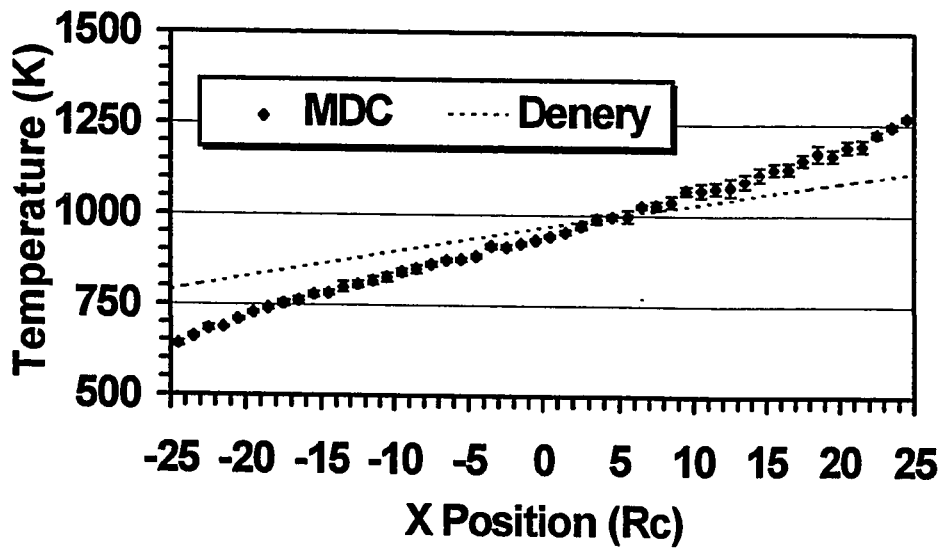


Figure C-18: Average temperature as a function of position for run 6 where $T_H/T_C = 10.0$, $K_n = 1.0$, and $M_a = 0.5$.

Profiles for Run 7:

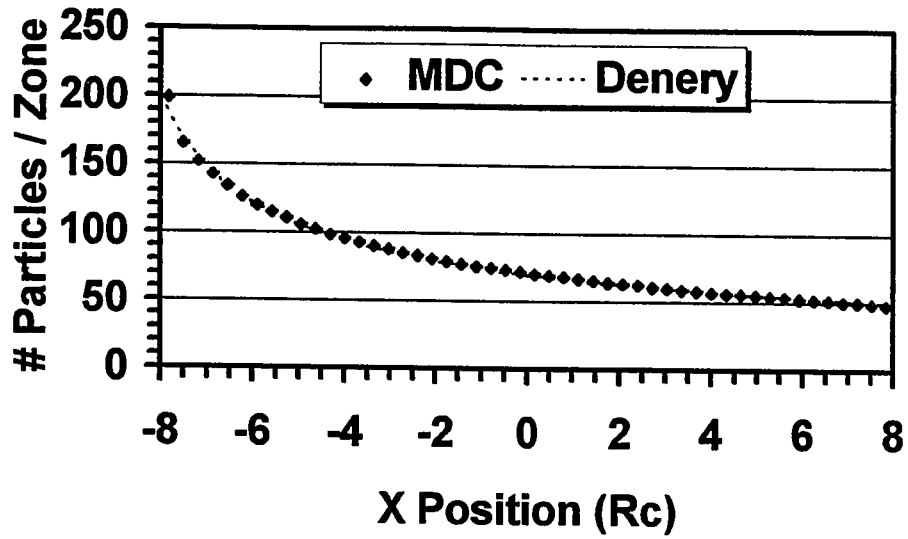


Figure C-19: Average number of particles per zone as a function of position for run 7 where $T_H/T_C = 10.0$, $K_n = 0.1$, and $M_a = 2.0$.

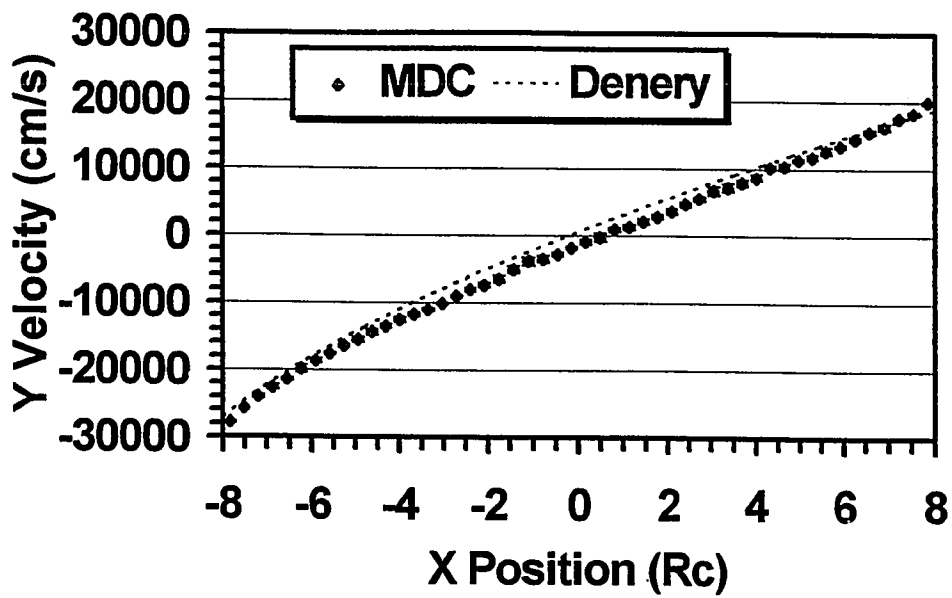


Figure C-20: Average particle velocity (component parallel to thermal walls) as a function of position for run 7 where $T_H/T_C = 10.0$, $K_n = 0.1$, and $M_a = 2.0$.

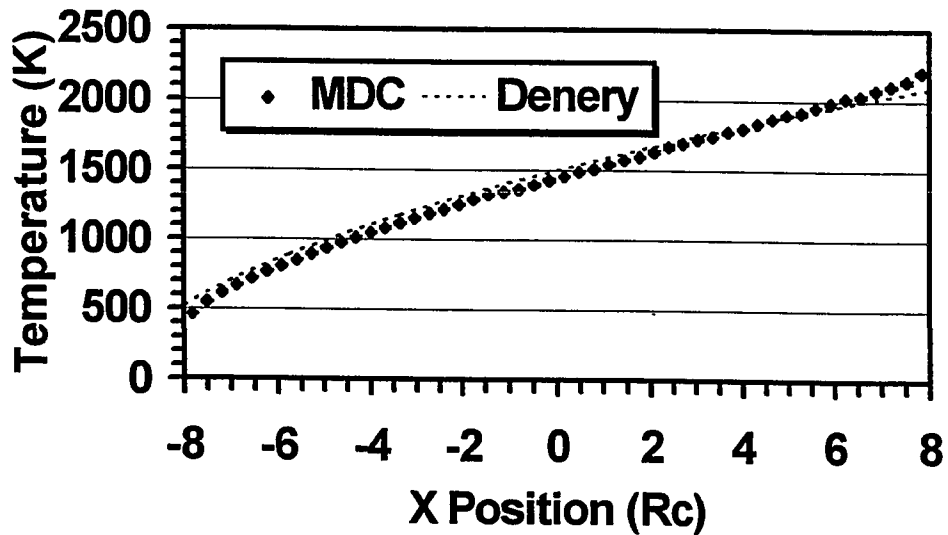


Figure C-21: Average temperature as a function of position for run 7 where $T_H/T_C = 10.0$, $K_n = 0.1$, and $M_a = 2.0$.

Profiles for Run 8:

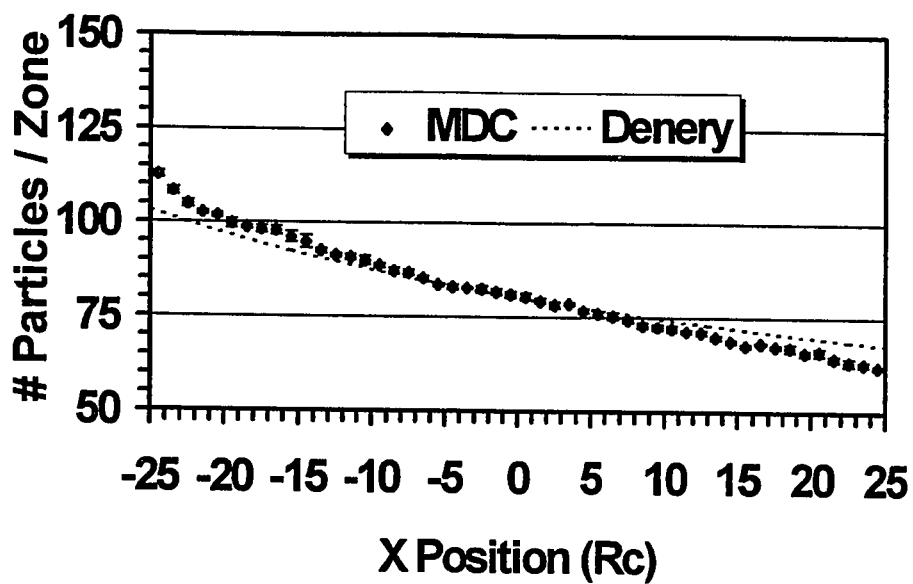


Figure C-22: Average number of particles per zone as a function of position for run 8 where $T_H/T_C = 10.0$, $K_n = 1.0$, and $M_a = 2.0$.

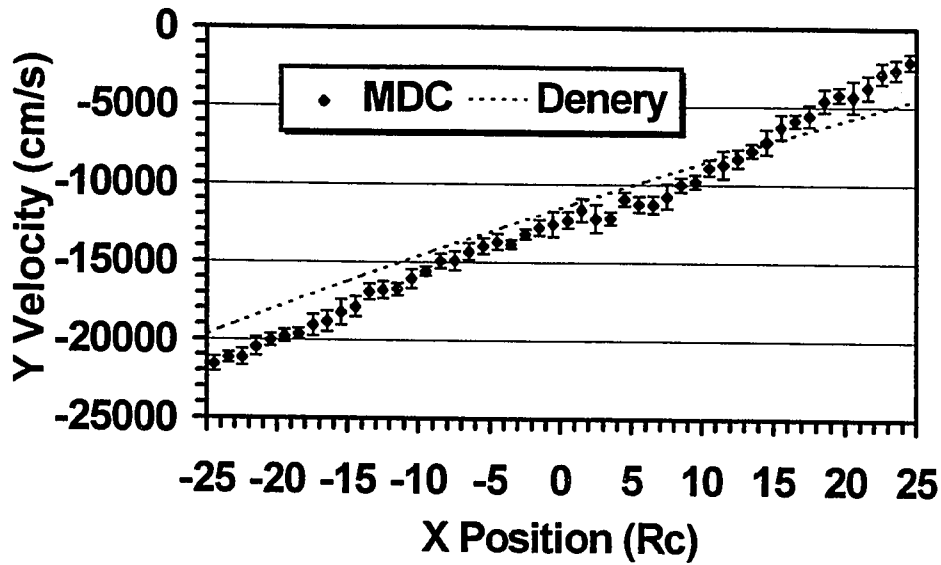


Figure C-23: Average particle velocity (component parallel to thermal walls) as a function of position for run 8 where $T_H/T_C = 10.0$, $K_n = 1.0$, and $M_a = 2.0$.

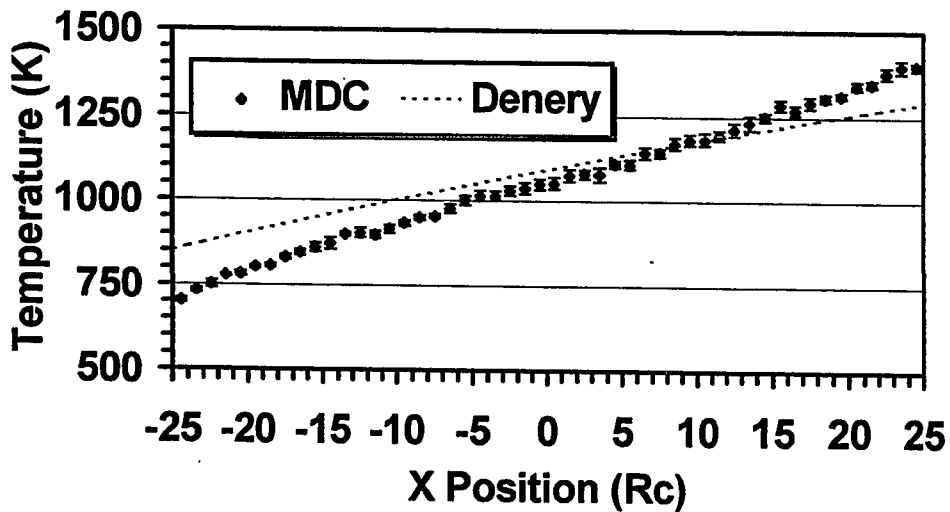


Figure C-24: Average temperature as a function of position for run 8 where $T_H/T_C = 10.0$, $K_n = 1.0$, and $M_a = 2.0$.

Appendix D

MDC Computer Source Code Listing

```
/* Include file for mdc.c program */

#define SIGMA 3.405e-08 /* cm */
#define MASS 6.633e-23 /* argon atom mass (gm) */
#define BOLTZ 1.381e-16 /* Boltzman constant (erg/K) */
#define EPSILON 1.654e-14 /* erg */
#define CUTOFF 1.075e-07 /* cm */

#define DXCELL CUTOFF
#define DYCELL CUTOFF
#define DZCELL CUTOFF

#define MKPART 4096
#define MKBIN 50
#define NCELLX 16
#define NCELLY 16
#define NCELLZ 16
#define NCELLS (NCELLX*NCELLY*NCELLZ)

#define SW_TIME_STEP 1
#define SW_NUM_PARTS 2
#define SW_NUM_ITER 3
#define SW_PERIODIC 4
#define SW_READ_CONF 5
#define SW_OUT_CONF 6
#define SW_OUT_DIAG 7
#define SW_OUT_STAT 8
#define SW_TEMP_W1 9
#define SW_TEMP_W2 10
#define SW_TEMP_GAS 11
#define SW_VEL_W1 12
#define SW_VEL_W2 13
#define SW_ACC_Z 14
#define SW_PRINT_MOD 15
#define SW_HELP 16

#ifndef PI
#define PI 3.141592654
#endif

#define TRUE 1
#define FALSE 0

typedef struct {
    int binX;
    int binY;
    int binZ;
} Bin3d;

extern int main ( int argc, char *argv[] );
extern void setIndicies ( float lX, float lY, float lZ );
extern void outputRunParams ( FILE *filePtr );
extern float ran1 ( long *idum );
extern void bzero ( char *ptr, int len );
extern void parseCommand ( int argc, char *argv[] );
extern void tallyStats ( void );
```

```

/* Molecular Dynamics Channel Program mdc.c */

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include "max3d.h"
#include "getswtch.h"

/* Define Globals */

switch_t switches[] = {
  {SW_TIME_STEP, "-dTime", 2, 1, "f", "#"},
  {SW_NUM_PARTS, "-nParts", 2, 1, "i", "#"},
  {SW_NUM_ITER, "-nIter", 2, 1, "i", "#"},
  {SW_PERIODIC, "-periodic", 2, 0, (char *)NULL, (char *)NULL},
  {SW_READ_CONF, "-readConfig", 2, 1, "n", "<filename>"},
  {SW_TEMP_W1, "-tW1", 3, 1, "f", "#"},
  {SW_TEMP_W2, "-tW2", 3, 1, "f", "#"},
  {SW_TEMP_GAS, "-tGas", 2, 1, "f", "#"},
  {SW_VEL_W1, "-vW1", 3, 1, "f", "#"},
  {SW_VEL_W2, "-vW2", 3, 1, "f", "#"},
  {SW_ACC_Z, "-accZ", 2, 1, "f", "#"},
  {SW_PRINT_MOD, "-printMod", 2, 1, "i", "#"},
  {SW_OUT_DIAG, "-diagFile", 2, 1, "n", "<filename>"},
  {SW_OUT_CONF, "-configFile", 2, 1, "n", "<filename>"},
  {SW_OUT_STAT, "-statFile", 2, 1, "n", "<filename>"},
  {SW_HELP, "-help", 2, 0, (char *)NULL, (char *)NULL},
};

int nswitches = sizeof(switches)/sizeof(switch_t);

double posX[MXPART], posY[MXPART], posZ[MXPART];
double oldPosX[MXPART], oldPosY[MXPART], oldPosZ[MXPART];
double velX[MXPART], velY[MXPART], velZ[MXPART];
double avgMVelX[MKBIN], avgMVelY[MKBIN], avgMVelZ[MKBIN];
double avgKE[MKBIN], avgNum[MKBIN];
double binWidth;
float dxm[NCELLX], dyp[NCELLY], dym[NCELLY], dyp[NCELLY];
float dzp[NCELLZ];
float dxfn[13][NCELLX], dyfn[13][NCELLY], dzfn[13][NCELLZ];
float fNPFPM, fNPFPM2, fPM;
int im[NCELLX], ip[NCELLX], jm[NCELLY], jp[NCELLY];
int kp[NCELLZ], cellCounts[NCELLX+1][NCELLY][NCELLZ];
int ifn[13][NCELLX], jfn[13][NCELLY], kfn[13][NCELLZ];
int statCount;
FILE *fpConfigIn, *fpConfigOut, *fpStatOut, *fpDiagOut;

/* Default input parameters */
float dt = 3.123e-15; /* sec */
float velWX1 = 0.; /* (cm)/(sec) */
float velWX2 = 0.; /* (cm)/(sec) */
float extAccZ = 0.; /* (cm)/(sec2) */
float radCut = 3*SIGMA; /* cm */
int numParts = NCELLS;
int numIter = 1000;
int printMod = 20;
int periodic = FALSE;
float tempGas = 295.; /* K */
float tempWX1 = 295.; /* K */
float tempWX2 = 295.; /* K */
char *inConfigFileName = (char *)NULL;
char *outDiagFileName = "max3Diag.out";
char *outConfigFileName = "max3Conf.out";
char *outStatFileName = "max3Stat.out";

int main ( argc, argv )

```

```

int  argc;
char *argv[];
{

    static double accX[MXPART], accY[MXPART], accZ[MXPART];

    static int    partIdOffset[NCELLX][NCELLY][NCELLZ], partId[MXPART];
    static int    tStep, iX, iY, iZ, sumParts, velGas;

    static float  lX, lY, lZ;
    static double wx, wy, wz, wNorm;
    static double dx, dy, dz, r2, r2i, r4i, r6i, r8i, r11i, aor, pE, kE;
    static double pX, pY, pZ, axTmp, ayTmp, azTmp;
    static double newPosX, newPosY, newPosZ, magV;
    static double sumKE=0., sumPE=0., sumPX=0., sumPY=0., sumPZ=0.;
    static double c1V, c2V, sigma6, c1A, c2A;
    static double dt2, twoDt, radCut2, avgTemp, statVel, statTime;
    static double mpvX1, mpvX2, phi, vPar11, xRatio;

    int          ic, jc, kc, np, npOff, npc, npcOff, nc1, nc2, n1, n2, nCellClr;
    int          npn, npnOff, nf, nn, statMod;
    static long  iseed = 1;

    Bin3d partBinLoc[MXPART];

    /* Parse command line looking for overrides to input parameters */
    parseCommand ( argc, argv );

    /* Open diagnostic output file */
    if ((fpDiagOut = fopen(outDiagFileName, "w")) == (FILE *)NULL) {
        fprintf(stderr,
            "Error opening diagnostic output file: max3d.out\n");
        exit(1);
    }

    /* Calculate dimensions of volume */
    lX = (float) NCELLX*DXCELL;
    lY = (float) NCELLY*DYCELL;
    lZ = (float) NCELLZ*DZCELL;

    /* Calculate Statistics Constants */
    binWidth = lX/( ( double ) MXBIN );
    statCount = 0;

    /* Initialize indices */
    setIndices(lX, lY, lZ);

    /* Calculate time constants */
    twoDt = 2.*dt;
    dt2 = dt*dt;

    /* Calculate square of the cutoff radius */
    radCut2 = radCut*radCut;

    /* Calculate most probable velocities at thermal boundaries */
    if ( periodic == FALSE ) {
        mpvX1 = sqrt( 2*BOLTZ*tempWX1/MASS );
        mpvX2 = sqrt( 2*BOLTZ*tempWX2/MASS );
    }

    /* Calculate potential energy coefficients */
    sigma6 = pow((double)SIGMA, (double)6.);
    c1V = 1.76e-81;
    /*c2V = 4*EPSILON*sigma6;*/

    /* Calculate acceleration coefficients */
    c1A = 9.*c1V/MASS;
    /*c2A = 6.*c2V/MASS;*/

```

```

/* Initialize particle positions */
if ( inConfigFileName == (char *) NULL ) {
/* Create initial particle configuration using gas temperature
to calculate magnitude random velocity */
velGas = sqrt(3.*BOLTZ*tempGas/MASS);
for (ic=0; ic<NCELLX; ic++)
for (jc=0; jc<NCELLY; jc++)
for (kc=0; kc<NCELLZ; kc++) {
np = ic + jc*NCELLX + kc*NCELLX*NCELLY;
posX[np] = (ic+0.5)*DXCELL;
posY[np] = (jc+0.5)*DYCELL;
posZ[np] = (kc+0.5)*DZCELL;
/* Pick arbitrary direction */
wX = 0.5 - ran1(&iseed);
wY = 0.5 - ran1(&iseed);
wZ = 0.5 - ran1(&iseed);
/* Calculate Norm */
wNorm = sqrt(wX*wX + wY*wY + wZ*wZ);
/* Calculate velocity weight */
wX /= wNorm;
wY /= wNorm;
wZ /= wNorm;
oldPosX[np] = posX[np] + wX*velGas*dt;
oldPosY[np] = posY[np] + wY*velGas*dt;
oldPosZ[np] = posZ[np] + wZ*velGas*dt;
}
}
else {
/* Read in initial particle configuration */
if ((fpConfigIn = fopen(inConfigFileName, "r")) == (FILE *)NULL) {
fprintf(stderr,
"Error opening initial configuration file: %s\n",
inConfigFileName);
exit(2);
}
fscanf(fpConfigIn, "%d\n\n", &numParts);
for (np=0; np<numParts; np++) {
fscanf ( fpConfigIn, "%lg %lg %lg %lg %lg %lg\n",
&posX[np], &posY[np], &posZ[np],
&velX[np], &velY[np], &velZ[np] );

/* Generate previous positions from velocities */
oldPosX[np] = posX[np] - velX[np]*dt;
oldPosY[np] = posY[np] - velY[np]*dt;
oldPosZ[np] = posZ[np] - velZ[np]*dt;
}
fclose ( fpConfigIn );
}

/* Calculate averaging constants */
fPM = (float) printMod;
fNPFPM = (float) numParts * fPM;
fNPFPM2 = fNPFPM*fNPFPM;

/* Calculate statistical modulation */
statVel = 0.;
for ( np=0; np<numParts; np++ ) {
statVel += sqrt( velX[np]*velX[np] + velY[np]*velY[np] +
velZ[np]*velZ[np] );
}
statVel /= (double)np;
statTime = binWidth/statVel;
statMod = (int)( statTime/dt );

/* Zero out Statistical arrays */
bzero( (char *)avgMVelX, (int)( MXBIN*sizeof(double) ) );
bzero( (char *)avgMVelY, (int)( MXBIN*sizeof(double) ) );

```

```

bzero( (char *)avgVelZ, (int)( MXBIN*sizeof(double) ) );
bzero( (char *)avgKE, (int)( MXBIN*sizeof(double) ) );
bzero( (char *)avgNum, (int)( MXBIN*sizeof(double) ) );

/* Print out run parameters */
outputRunParams ( stderr );
outputRunParams ( fpDiagOut );

/** Calculate size of Cell Count array */
nCellClr = ( NCELLX + 1 )*( NCELLY )*( NCELLZ );

/** Main Loop over Time */
for (tStep=1; tStep<=numIter; tStep++) {

    /* Initialize potential energy */
    pE = 0.;

    /* Initialize cellCounts to zero */
    bzero ( (char *) cellCounts, (int) ( nCellClr*sizeof(int) ) );

    /* Bin particles into cells */
    for ( np=0; np<numParts; np++ ) {
        iX = (int) ( posX[np]/DXCELL );
        /* Perform check for particle on right thermal boundary */
        if ( iX >= NCELLX )
            iX = NCELLX-1;
        iY = (int) ( posY[np]/DYCELL );
        iZ = (int) ( posZ[np]/DZCELL );
        cellCounts[iX][iY][iZ] ++;

        /* Save cell locations for each particle */
        partBinLoc[np].binX = iX;
        partBinLoc[np].binY = iY;
        partBinLoc[np].binZ = iZ;
    }

    /* Tabulate offset for each cell into
    particle id array -> partIdOffsets */
    sumParts = 0;
    for (kc=0; kc<NCELLZ; kc++)
        for (jc=0; jc<NCELLY; jc++)
            for (ic=0; ic<NCELLX; ic++) {
                partIdOffset[ic][jc][kc] = sumParts;
                sumParts += cellCounts[ic][jc][kc];
            }

    /* Reset cellCounts to zero */
    bzero ( (char *) cellCounts, (int) ( nCellClr*sizeof(int) ) );

    /* Calculate particle Ids within each cell and place
    Ids in sequential order in partical Id array */
    for (np=0; np<numParts; np++) {
        iX = partBinLoc[np].binX;
        iY = partBinLoc[np].binY;
        iZ = partBinLoc[np].binZ;
        cellCounts[iX][iY][iZ] ++;

        /* Calculate offset into particle Id array */
        npOff = partIdOffset[iX][iY][iZ] + cellCounts[iX][iY][iZ] - 1;

        /* Assign particle Id */
        partId[npOff] = np;
    }

    /* Calculate force acting on each particle due
    to neighboring particles */

```

```

/* (Force) Loop over all cells (ic,jc,kc) */
for (kc=0; kc<NCELLZ; kc++)
  for(jc=0; jc<NCELLY; jc++)
    for (ic=0; ic<NCELLX; ic++) {

      /*fprintf(stderr, "\nCurrent Cell: %d\n ",
        (1+ic+jc*NCELLX+kc*NCELLX*NCELLY));*/

      /* Offset into particle id array for current cell */
      npcOff = partIdOffset[ic][jc][kc];

      /* Number of particles in current cell biased by offset */
      npc = npcOff + cellCounts[ic][jc][kc];

      /* Loop over all particles within current cell */
      for (nc1= npcOff; nc1<npc; nc1++) {

        /* Current particle Id-Index */
        n1 = partId[nc1];

        /* Tabulate forces on current particle due to
          the presence of other particles within this cell */

        /* Loop over all other particles within current cell */
        for(nc2=nc1+1; nc2<npc; nc2++) {

          /* Id-Index of "other" particle */
          n2 = partId[nc2];

          /* Calculate (distance)**2 between both particles */
          dx = posX[n2] - posX[n1];
          dy = posY[n2] - posY[n1];
          dz = posZ[n2] - posZ[n1];
          r2 = dx*dx + dy*dy + dz*dz;

          /* Process only those particles that are
            within the cutoff radius of each other */
          if (r2 < radCut2) {
            /*r2i = 1./r2;
            r4i = r2i*r2i;
            r6i = r2i*r4i;
            r8i = r4i*r4i;*/
            r11i = pow( r2, (double) (-5.5) );
            aor = r11i*c1A;

            /* Acceleration of n1 due to presence of n2 */
            axTmp = aor*dx; accX[n1] -= axTmp;
            ayTmp = aor*dy; accY[n1] -= ayTmp;
            azTmp = aor*dz; accZ[n1] -= azTmp;

            /* Equal and opposite acceleration of n2 due
              to presence of n1 */
            accX[n2] += axTmp;
            accY[n2] += ayTmp;
            accZ[n2] += azTmp;

            /* Tabulate potential energy */
            pE += r11i*r2*c1V;

          } /* Particle pairs within cut-off radius */

        } /* Loop over other particles within current cell */

      } /* Tabulate forces on current particle due to the presence
        of other particles within neighboring cells */

```



```

/* Loop over current cell's 13 "forward-neighbor" cells */
for (nf=0; nf<13; nf++) {

    /* Offset into particle id array for
       current neighbor cell */
    npnOff = partIdOffset [ifn[nf][ic]]
                [jfn[nf][jc]]
                [kfn[nf][kc]];

    /* Number of particles in current neighbor cell
       biased by offset */
    npn = npnOff + cellCounts [ifn[nf][ic]]
                [jfn[nf][jc]]
                [kfn[nf][kc]];

    /* Loop over all particles within current neighbor cell */
    for (nn=npnOff; nn<npn; nn++) {

        /* Neighbor cell particle Id-Index */
        n2 = partId[nn];

        /* Calculate (distance)**2 between both particles */

        dx = posX[n2] - posX[n1] + dxfn[nf][ic];
        dy = posY[n2] - posY[n1] + dyfn[nf][jc];
        dz = posZ[n2] - posZ[n1] + dzfn[nf][kc];
        r2 = dx*dx + dy*dy + dz*dz;

        /* Process only those particles that are
           within the cutoff radius of each other */
        if ( r2 < radCut2 ) {
            /*r2i = 1./r2;
            r4i = r2i*r2i;
            r6i = r2i*r4i;
            r8i = r4i*r4i;*/
            r11i = pow( r2, (double) (-5.5) );
            aor = r11i*c1A;

            /* Acceleration of n1 due to presence of n2*/
            axTmp = aor*dx; accX[n1] -= axTmp;
            ayTmp = aor*dy; accY[n1] -= ayTmp;
            azTmp = aor*dz; accZ[n1] -= azTmp;

            /* Equal and opposite acceleration of n2 due
               to presence of n1 */
            accX[n2] += axTmp;
            accY[n2] += ayTmp;
            accZ[n2] += azTmp;

            /* Tabulate potential energy */
            pE += r11i*r2*c1V;

        } /* Particle within cutoff radius */

    } /* Loop over all particles within neighbor cell */
} /* Loop over all "forward-neighbors" of current cell */

} /* Loop over all particles within current cell */

} /* Force Loop over all cells */

/* Initialize total kinetic energy and total momentum */
kE = pX = pY = pZ = 0;

/* Loop over all particles */
for (np=0; np<numParts; np++) {

```

```

/* Calculate new positions using Verlet method */
newPosX = 2.*posX[np] - oldPosX[np] + dt2*accX[np];
newPosY = 2.*posY[np] - oldPosY[np] + dt2*accY[np];
newPosZ = 2.*posZ[np] - oldPosZ[np] + dt2*( accZ[np] + extAccZ );

/* Calculate new velocities for Total Momentum
and Kinetic Energy monitoring */
velX[np] = (newPosX - oldPosX[np])/twoDt;
velY[np] = (newPosY - oldPosY[np])/twoDt;
velZ[np] = (newPosZ - oldPosZ[np])/twoDt;

/* Transfer updated positions */
oldPosX[np] = posX[np]; posX[np] = newPosX;
oldPosY[np] = posY[np]; posY[np] = newPosY;
oldPosZ[np] = posZ[np]; posZ[np] = newPosZ;

/* Check for crossing of cell boundaries */

/* Thermal Boundary Conditions*/
if ( posX[np] < 0 ) {
  if ( periodic == TRUE ) {
    posX[np] += 1X;
    oldPosX[np] += 1X;
  }
  else { /* Thermal Boundaries */
    /* Set new velocities */
    phi = ran1( &iseed ) * 2 * PI;
    vPar11 = mpvX1 * sqrt( -log ( 1. - ran1( &iseed ) ) );
    velX[np] = mpvX1 * sqrt( -log ( 1. - ran1( &iseed ) ) );
    velY[np] = vPar11 * sin(phi) + velWX1;
    velZ[np] = vPar11 * cos(phi);
    /* Set position of particle back to the boundary */
    xRatio = ( -oldPosX[np] ) / ( posX[np] - oldPosX[np] );
    posX[np] = 0.;
    posY[np] = oldPosY[np] + ( posY[np] - oldPosY[np] ) * xRatio;
    posZ[np] = oldPosZ[np] + ( posZ[np] - oldPosZ[np] ) * xRatio;
    /* Set "old" loc. to be consistent with new loc. & vel. */
    oldPosX[np] = posX[np] - velX[np] * dt;
    oldPosY[np] = posY[np] - velY[np] * dt;
    oldPosZ[np] = posZ[np] - velZ[np] * dt;
  }
}

if ( posX[np] > 1X ) {
  if ( periodic == TRUE ) { /* Periodic Boundaries */
    posX[np] -= 1X;
    oldPosX[np] -= 1X;
  }
  else { /* Thermal Boundaries */
    /* Set new velocities */
    phi = ran1( &iseed ) * 2 * PI;
    vPar11 = mpvX2 * sqrt( -log ( 1. - ran1( &iseed ) ) );
    velX[np] = - mpvX2 * sqrt( -log ( 1. - ran1( &iseed ) ) );
    velY[np] = vPar11 * sin(phi) + velWX2;
    velZ[np] = vPar11 * cos(phi);
    /* Set position of particle back to the boundary */
    xRatio = ( 1X - oldPosX[np] ) / ( posX[np] - oldPosX[np] );
    posX[np] = 1X;
    posY[np] = oldPosY[np] + ( posY[np] - oldPosY[np] ) * xRatio;
    posZ[np] = oldPosZ[np] + ( posZ[np] - oldPosZ[np] ) * xRatio;
    /* Set "old" loc. to be consistent with new loc. & vel. */
    oldPosX[np] = posX[np] - velX[np] * dt;
    oldPosY[np] = posY[np] - velY[np] * dt;
    oldPosZ[np] = posZ[np] - velZ[np] * dt;
  }
}
}

```

```

/* Periodic Boundaries */
if ( posY[np] < 0 ) {
    posY[np] += 1Y;
    oldPosY[np] += 1Y;
}

if ( posY[np] >= 1Y ) {
    posY[np] -= 1Y;
    oldPosY[np] -= 1Y;
}

if ( posZ[np] < 0 ) {
    posZ[np] += 1Z;
    oldPosZ[np] += 1Z;
}

if ( posZ[np] >= 1Z ) {
    posZ[np] -= 1Z;
    oldPosZ[np] -= 1Z;
}

/* Calculate square of magnitude of velocity */
magV = velX[np]*velX[np] + velY[np]*velY[np] +
        velZ[np]*velZ[np];

/* Tally total kinetic energy */
kE += magV;

/* Tally total momentum */
pX += velX[np];
pY += velY[np];
pZ += velZ[np];

/* Zero out particle accelerations */
accX[np] = 0.;
accY[np] = 0.;
accZ[np] = 0.;

} /* Loop over all particles & update positions */

sumKE += kE;
sumPE += pE;
sumPX += pX;
sumPY += pY;
sumPZ += pZ;

/* Print out diagnostic quantities at specified intervals */
if ( ( tstep % printMod ) == 0 ) {

    /* Calculate time averaged quantities */
    avgTemp = ( MASS / ( 3.*BOLTZ ) )*( sumKE/fNPFPM -
        sumPX*sumPX/fNPFPM2 - sumPY*sumPY/fNPFPM2 -
        sumPZ*sumPZ/fNPFPM2 );

    sumKE *= ( 0.5*MASS/fPM );
    sumPE /= fPM;
    sumPX *= MASS/fPM;
    sumPY *= MASS/fPM;
    sumPZ *= MASS/fPM;

    fprintf ( stderr, "%d <KE>=%g <PE>= %g <E>=%g <Temp>=%g\n",
        tStep, sumKE, sumPE, sumKE+sumPE, avgTemp );
    fprintf ( fpDiagOut,
        "%d <KE>=%g <PE>= %g <E>=%g <Px>=%g <Py>=%g <Pz>=%g\n",
        tStep, sumKE, sumPE, sumKE+sumPE, sumPX, sumPY, sumPZ );

    /* Save particle configuration */
    fpConfigOut = fopen ( outConfigFileName, "w" );

```

```

fprintf ( fpConfigOut, "%d\n\n", numParts );
for ( np=0; np<numParts; np++ )
    fprintf ( fpConfigOut, "%g %g %g %g %g %g\n",
            posX[np], posY[np], posZ[np],
            velX[np], velY[np], velZ[np] );
fclose ( fpConfigOut );

/* Reinitialize time average sums */
sumKE = sumPE = sumPX = sumPY = sumPZ = 0.;

/* Save statistical information */
if ( statCount > 0 ) {
    fpStatOut = fopen ( outStatFileName, "w" );
    fprintf ( fpStatOut, "%d\n\n", MXBIN );
    for ( ic=0; ic<MXBIN; ic++ )
        fprintf ( fpStatOut, "%g %g %g %g %g\n",
                ( avgNum[ ic ]/statCount ),
                ( MASS*avgMVelX[ ic ]/statCount ),
                ( MASS*avgMVelY[ ic ]/statCount ),
                ( MASS*avgMVelZ[ ic ]/statCount ),
                ( 0.5*MASS*avgKE[ ic ]/statCount ) );

    fclose ( fpStatOut );
}

}

/* Tally[B Statistics */
if ( ( tStep % statMod ) == 0 ) {
    tallyStats();
}

} /* Time Increment Loop */

/* Last chance to save statistics information */
tallyStats();

if ( statCount > 0 ) {
    fpStatOut = fopen ( outStatFileName, "w" );
    fprintf ( fpStatOut, "%d\n\n", MXBIN );
    for ( ic=0; ic<MXBIN; ic++ )
        fprintf ( fpStatOut, "%g %g %g %g %g\n",
                ( avgNum[ ic ]/statCount ),
                ( MASS*avgMVelX[ ic ]/statCount ),
                ( MASS*avgMVelY[ ic ]/statCount ),
                ( MASS*avgMVelZ[ ic ]/statCount ),
                ( 0.5*MASS*avgKE[ ic ]/statCount ) );

    fclose ( fpStatOut );
}

/* Close diagnostic output file */
fclose ( fpDiagOut );

return 0;
}

void tallyStats ( )
{
    double    vXBIN[ MXBIN ], vYBIN[ MXBIN ], vZBIN[ MXBIN ], nBIN[ MXBIN ];
    double    kEBIN[ MXBIN ];
    int       i, binI;

    /* Increment counter */
    statCount ++;

    /* Initialize local sums */

```

```

for ( i=0; i<MXBIN; i++ )
    vXBin[ i ] = vYBin[ i ] = vZBin[ i ] = kEBin[ i ] = nBin[ i ] = 0;

/* Loop over particles */
for ( i=0; i<numParts; i++ ) {

    /* Calculate bin index */
    binI = posX[ i ]/binWidth;
    if ( binI == MXBIN )
        binI = MXBIN - 1;

    /* Calculate local sums */
    nBin[ binI ] ++;
    vXBin[ binI ] += velX[ i ];
    vYBin[ binI ] += velY[ i ];
    vZBin[ binI ] += velZ[ i ];
    kEBin[ binI ] += ( velX[ i ]*velX[ i ] + velY[ i ]*velY[ i ] +
                      velZ[ i ]*velZ[ i ] );
}

/* Calculate averages for each bin */
for ( i=0; i<MXBIN; i++ ) {
    avgNum[ i ] += nBin[ i ];
    avgMVelX[ i ] += ( vXBin[ i ]/nBin[ i ] );
    avgMVelY[ i ] += ( vYBin[ i ]/nBin[ i ] );
    avgMVelZ[ i ] += ( vZBin[ i ]/nBin[ i ] );
    avgKE[ i ] += ( kEBin[ i ]/nBin[ i ] );
}

return;
}

void parseCommand ( argc, argv )
int argc;
char *argv[];
{
    int gsw, curSwitch;

    /* Set program name */
    program_name = argv[ 0 ];

    while ((gsw = get_switch(argc, argv, switches, nswitches, &curSwitch)) !=
           GET_SWITCH_DONE)
        switch (gsw) {
            case SW_TIME_STEP:
                dt = atof(argv[curSwitch+1]);
                break;
            case SW_NUM_PARTS:
                numParts = atoi(argv[curSwitch+1]);
                break;
            case SW_NUM_ITER:
                numIter = atoi(argv[curSwitch+1]);
                break;
            case SW_PERIODIC:
                periodic = TRUE;
                break;
            case SW_READ_CONF:
                inConfigFileName = argv[ curSwitch+1 ];
                break;
            case SW_OUT_DIAG:
                outDiagFileName = argv[ curSwitch+1 ];
                break;
            case SW_OUT_CONF:
                outConfigFileName = argv[ curSwitch+1 ];
                break;
            case SW_OUT_STAT:
                outStatFileName = argv[ curSwitch+1 ];
                break;
        }
}

```

```

    case SW_TEMP_W1:
        tempWX1 = atof ( argv[ curSwitch+1 ] );
        break;
    case SW_TEMP_W2:
        tempWX2 = atof ( argv[ curSwitch+1 ] );
        break;
    case SW_TEMP_GAS:
        tempGas = atof ( argv[ curSwitch+1 ] );
        break;
    case SW_VEL_W1:
        velWX1 = atof ( argv[ curSwitch+1 ] );
        break;
    case SW_VEL_W2:
        velWX2 = atof ( argv[ curSwitch+1 ] );
        break;
    case SW_ACC_Z:
        extAccZ = atof ( argv[ curSwitch+1 ] );
        break;
    case SW_PRINT_MOD:
        printMod = atoi ( argv[ curSwitch+1 ] );
        break;
    case SW_HELP:
        usage( switches, nswitches );
        break;
    case NOT_A_SWITCH:
        usage( switches, nswitches );
        break;
    default:
        fprintf( stderr, "Internal error -- did not catch switch %d\n", gsw );
        exit( 1 );
    } /* End of switch on gsw */
}

void outputRunParams ( filePtr )
FILE *filePtr;
{
    fprintf ( filePtr, "Parameters for current Max3D run:\n\n" );
    fprintf ( filePtr, " Number of particles      : %d\n", numParts );
    fprintf ( filePtr, " Time step                : %g (sec)\n", dt );
    fprintf ( filePtr, " Number of iterations     : %d\n", numIter );
    fprintf ( filePtr, " Maximum interaction radius: %g (cm)\n", radCut );
    fprintf ( filePtr, " External Z Acceleration  : %g (cm/sec2)\n", extAccZ );
    if ( inConfigFileName )
        fprintf ( filePtr, " Read input configuration from file: %s\n",
                inConfigFileName );
    else
        fprintf ( filePtr, " Create initial configuration using T gas: %g (K)\n",
                tempGas );

    if ( outConfigFileName )
        fprintf ( filePtr,
                " Write final particle configuration to file: %s\n",
                outConfigFileName );
    else
        fprintf( filePtr, " Suppress output of final particle configuration.\n" );

    if ( outStatFileName )
        fprintf ( filePtr,
                " Write particle statistics to file: %s\n",
                outStatFileName );
    else
        fprintf( filePtr, " Suppress output of particle statistics.\n" );

    fprintf ( filePtr, "\nCell cavity dimensions:\n\n" );
    fprintf ( filePtr, " Number of cells in X direction: %d\n", NCELLX );
    fprintf ( filePtr, " Cell width in X direction   : %g (cm)\n", DXCELL );
    fprintf ( filePtr, " Number of cells in Y direction: %d\n", NCELLY );
    fprintf ( filePtr, " Cell width in Y direction   : %g (cm)\n", DYCELL );
}

```

```

fprintf ( filePtr, " Number of cells in Z direction: %d\n", NCELLZ );
fprintf ( filePtr, " Cell width in Z direction : %g (cm)\n", DZCELL );
fprintf ( filePtr, "\nAverage particle density: %g (Parts/cm3)\n",
(float)numParts/(NCELLX*DXCELL*NCELLY*DYCELL*NCELLZ*DZCELL) );
fprintf ( filePtr, "\nBoundary Conditions:\n\n" );
if ( periodic == TRUE )
    fprintf ( filePtr, " Periodic X boundaries selected.\n" );
else {
    fprintf ( filePtr, " Thermal walls at X boundaries selected:\n" );
    fprintf ( filePtr, " Velocity of Wall 1 : %g (cm/sec)\n", velWX1 );
    fprintf ( filePtr, " Temperature of Wall 1: %g (K)\n", tempWX1 );
    fprintf ( filePtr, " Velocity of wall 2 : %g (cm/sec)\n", velWX2 );
    fprintf ( filePtr, " Temperature of Wall 2: %g (K)\n\n", tempWX2 );
}
}

void setIndicies(IX, IY, IZ)
float IX;
float IY;
float IZ;
{
    int ic, jc, kc;

    /* Calculate indicies for indirect addressing of cells */
    if ( periodic == TRUE ) {
        im[0] = NCELLX-1; im[NCELLX-1] = NCELLX-2;
        ip[0] = 1; ip[NCELLX-1] = 0;
        dsm[0] = -IX; dsm[NCELLX-1] = 0.;
        dxp[0] = 0.; dxp[NCELLX-1] = IX;
    }
    else {
        im[0] = NCELLX; im[NCELLX-1] = NCELLX-2;
        ip[0] = 1; ip[NCELLX-1] = NCELLX;
        dsm[0] = 0.; dsm[NCELLX-1] = 0.;
        dxp[0] = 0.; dxp[NCELLX-1] = 0.;
    }

    for (ic=1; ic<NCELLX-1; ic++) {
        im[ic] = ic-1;
        ip[ic] = ic+1;
        dsm[ic] = 0.;
        dxp[ic] = 0.;
    }

    jm[0] = NCELLY-1; jm[NCELLY-1] = NCELLY-2;
    jp[0] = 1; jp[NCELLY-1] = 0;
    dym[0] = -IY; dym[NCELLY-1] = 0.;
    dyp[0] = 0.; dyp[NCELLY-1] = IY;

    for (jc=1; jc<NCELLY-1; jc++) {
        jm[jc] = jc-1;
        jp[jc] = jc+1;
        dym[jc] = 0.;
        dyp[jc] = 0.;
    }

    kp[NCELLZ-1] = 0;
    dzp[0] = 0.; dzp[NCELLZ-1] = IZ;

    for (kc=0; kc<NCELLZ-1; kc++) {
        kp[kc] = kc+1;
        dzp[kc] = 0.;
    }

    /* Use the above arrays to construct "forward-neighbor"
    cell-index-matricies and cell-delta-matricies */
}

```

```

for (ic=0; ic<NCELLX; ic++) {
  ifn [0] [ic] = ip[ic];
  ifn [1] [ic] = im[ic];
  ifn [2] [ic] = ic;
  ifn [3] [ic] = ip[ic];
  ifn [4] [ic] = im[ic];
  ifn [5] [ic] = ic;
  ifn [6] [ic] = ip[ic];
  ifn [7] [ic] = im[ic];
  ifn [8] [ic] = ic;
  ifn [9] [ic] = ip[ic];
  ifn [10][ic] = im[ic];
  ifn [11][ic] = ic;
  ifn [12][ic] = ip[ic];
  dxfn[0] [ic] = dxp[ic];
  dxfn[1] [ic] = dxm[ic];
  dxfn[2] [ic] = 0;
  dxfn[3] [ic] = dxp[ic];
  dxfn[4] [ic] = dxm[ic];
  dxfn[5] [ic] = 0;
  dxfn[6] [ic] = dxp[ic];
  dxfn[7] [ic] = dxm[ic];
  dxfn[8] [ic] = 0;
  dxfn[9] [ic] = dxp[ic];
  dxfn[10][ic] = dxm[ic];
  dxfn[11][ic] = 0;
  dxfn[12][ic] = dxp[ic];
}

for (jc=0; jc<NCELLY; jc++) {
  jfn [0] [jc] = jc;
  jfn [1] [jc] = jp[jc];
  jfn [2] [jc] = jp[jc];
  jfn [3] [jc] = jp[jc];
  jfn [4] [jc] = jm[jc];
  jfn [5] [jc] = jm[jc];
  jfn [6] [jc] = jm[jc];
  jfn [7] [jc] = jc;
  jfn [8] [jc] = jc;
  jfn [9] [jc] = jc;
  jfn [10][jc] = jp[jc];
  jfn [11][jc] = jp[jc];
  jfn [12][jc] = jp[jc];
  dyfn[0] [jc] = 0;
  dyfn[1] [jc] = dyp[jc];
  dyfn[2] [jc] = dyp[jc];
  dyfn[3] [jc] = dyp[jc];
  dyfn[4] [jc] = dym[jc];
  dyfn[5] [jc] = dym[jc];
  dyfn[6] [jc] = dym[jc];
  dyfn[7] [jc] = 0;
  dyfn[8] [jc] = 0;
  dyfn[9] [jc] = 0;
  dyfn[10][jc] = dyp[jc];
  dyfn[11][jc] = dyp[jc];
  dyfn[12][jc] = dyp[jc];
}

for (kc=0; kc<NCELLZ; kc++) {
  kfn [0] [kc] = kc;
  kfn [1] [kc] = kc;
  kfn [2] [kc] = kc;
  kfn [3] [kc] = kc;
  kfn [4] [kc] = kp[kc];
  kfn [5] [kc] = kp[kc];
  kfn [6] [kc] = kp[kc];
  kfn [7] [kc] = kp[kc];
  kfn [8] [kc] = kp[kc];
}

```



```

    kfn [9] [kc] = kp[kc];
    kfn [10][kc] = kp[kc];
    kfn [11][kc] = kp[kc];
    kfn [12][kc] = kp[kc];
    dzfn[0] [kc] = 0;
    dzfn[1] [kc] = 0;
    dzfn[2] [kc] = 0;
    dzfn[3] [kc] = 0;
    dzfn[4] [kc] = dzp[kc];
    dzfn[5] [kc] = dzp[kc];
    dzfn[6] [kc] = dzp[kc];
    dzfn[7] [kc] = dzp[kc];
    dzfn[8] [kc] = dzp[kc];
    dzfn[9] [kc] = dzp[kc];
    dzfn[10][kc] = dzp[kc];
    dzfn[11][kc] = dzp[kc];
    dzfn[12][kc] = dzp[kc];
}

}

#define IA 16807
#define IM 2147483647
#define AM (1.0/IM)
#define IQ 127773
#define IR 2836
#define NTAB 32
#define NDIV (1+(IM-1)/NTAB)
#define EPS 1.2e-7
#define RNMK (1.0-EPS)

float ran1(idum)
    long *idum;
{
    int j;
    long k;
    static long iy=0;
    static long iv[NTAB];
    float temp;

    if (*idum <= 0 || !iy) {
        if (-(*idum) < 1) *idum=1;
        else *idum = -(*idum);
        for (j=NTAB+7;j>=0;j--) {
            k=(*idum)/IQ;
            *idum=IA*( *idum-k*IQ)-IR*k;
            if (*idum < 0) *idum += IM;
            if (j < NTAB) iv[j] = *idum;
        }
        iy=iv[0];
    }
    k=(*idum)/IQ;
    *idum=IA*( *idum-k*IQ)-IR*k;
    if (*idum < 0) *idum += IM;
    j=iy/NDIV;
    iy=iv[j];
    iv[j] = *idum;
    if ((temp=AM*iy) > RNMK) return RNMK;
    else return temp;
}

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

```

```
#undef RMX
```

```
/* (C) Copr. 1986-92 Numerical Recipes Software 221.&#x26;V&#x26;V. */
```

Statistics Program Listing

```
/* Include file for Statistics Program stats.c */

#define      MASS      6.633e-23      /* argon atom mass (gm) */
#define      BOLTZ     1.381e-16     /* Boltzman constant (erg/K) */

#define      TRUE      1
#define      FALSE     0

extern int   main( int argc, char *argv[] );
extern double temperature( double KE, double pX, double pY, double pZ );
extern void  **callocate( int row, int col, unsigned long itemSize );

/* Statistics Program stats.c */

#include      <stdio.h>
#include      <math.h>
#include      <stdlib.h>
#include      <malloc.h>
#include      "stats.h"

int main( argc, argv )
  int  argc;
  char *argv[];
{
  double **avgND, **avgKE, **avgMVelX, **avgMVelY, **avgMVelZ;
  double *sumTemp, *sumTemp2, *sumNum, *sumNum2, *sumVelX, *sumVelX2;
  double *sumVelY, *sumVelY2, *sumVelZ, *sumVelZ2;
  double *meanND, *meanVelX, *meanVelY, *meanVelZ, *meanTemp;
  double *errND, *errVelX, *errVelY, *errVelZ, *errTemp;
  double temp, U0, Tc, Rc;
  int    statFileCount, numBins, i, j;
  FILE   *fpStatNames, *fpStatFileIn, *fpStatOut;

  /* Default input parameters */
  char  lineBuffer[128], *statNamesFile="statNames";
  char  *statOutFile="Stats.out";
  char  **inStatFile;

  /* Open Stat Name File */
  if ( ( fpStatNames = fopen(statNamesFile, "r") ) == (FILE *)NULL ) {
    fprintf(stderr,
            "Error opening Stat Names File: statNames\n");
    exit(1);
  }

  /* Read in number of Stat files */
  if ( fgets( lineBuffer, 128, fpStatNames ) == (char *)NULL ) {
    printf( "Error reading # of bins from statNames.");
    exit( 0 );
  }
  else
    sscanf( lineBuffer, "%d %lf %lf", &statFileCount, &U0, &Tc);

  /* Output parameters */
  printf( "\n Number of Stat Files: %d \n\n", statFileCount );
  printf( " Wall Velocity Differential: %f (cm/sec)\n", (float) U0 );
  printf( " Cold Wall Temperature: %f (K)\n\n", (float) Tc );
}
```

```

/* Allocate space for Stat File names */
inStatFile = (char **)calloc( statFileCount, 128, sizeof( char ) );

/* Loop through names of Stat Files */
for ( i=0; i<statFileCount; i++ ) {
    if ( fgets( lineBuffer, 128, fpStatNames ) == (char *)NULL ) {
        printf( "Error reading name of Stat File" );
        exit( 0 );
    }
    else
        sscanf( lineBuffer, "%s", inStatFile[i] );

/* Open Stat File */
if ( ( fpStatFileIn = fopen( inStatFile[i], "r" ) ) ==
    (FILE *)NULL ) {
    printf( "Error opening Stat File: %s\n", (char *)inStatFile[i] );
    exit( 0 );
}

/* Read in number of Bins */
if ( fgets( lineBuffer, 128, fpStatFileIn ) == (char *)NULL ) {
    printf( "Error reading # of bins from %s.\n", inStatFile[i] );
    exit( 0 );
}

    sscanf( lineBuffer, "%d", &numBins );

/* Output # of bins */
printf( " Processing %d bins from file %s ... \n", numBins,
        inStatFile[i] );

if ( fgets( lineBuffer, 128, fpStatFileIn ) == (char *)NULL ) {
    printf( "Error skipping line in file %s.\n", inStatFile[i] );
    exit( 0 );
}

/* Allocate space on first pass */
if ( i == 0 ) {
    avgND      = (double **)
        calloc( statFileCount, numBins, sizeof( double ) );
    avgKE      = (double **)
        calloc( statFileCount, numBins, sizeof( double ) );
    avgMVelX   = (double **)
        calloc( statFileCount, numBins, sizeof( double ) );
    avgMVelY   = (double **)
        calloc( statFileCount, numBins, sizeof( double ) );
    avgMVelZ   = (double **)
        calloc( statFileCount, numBins, sizeof( double ) );
}

/* Read in the goodies */
for ( j=0; j<numBins; j++ ) {
    if ( fgets( lineBuffer, 128, fpStatFileIn ) == (char *)NULL ) {
        printf( "Error reading record from file %s.\n", inStatFile[i] );
        exit( 0 );
    }

        sscanf( lineBuffer, "%lg %lg %lg %lg %lg", &avgND[i][j],
            &avgMVelX[i][j], &avgMVelY[i][j], &avgMVelZ[i][j],
            &avgKE[i][j] );
}
}

fclose( fpStatNames );
fclose( fpStatFileIn );

```

```

/* Initialize statistics arrays */
sumNum = (double *)calloc( numBins, sizeof( double ) );
sumNum2 = (double *)calloc( numBins, sizeof( double ) );
meanND = (double *)calloc( numBins, sizeof( double ) );
errND = (double *)calloc( numBins, sizeof( double ) );
sumVelX = (double *)calloc( numBins, sizeof( double ) );
sumVelX2 = (double *)calloc( numBins, sizeof( double ) );
meanVelX = (double *)calloc( numBins, sizeof( double ) );
errVelX = (double *)calloc( numBins, sizeof( double ) );
sumVelY = (double *)calloc( numBins, sizeof( double ) );
sumVelY2 = (double *)calloc( numBins, sizeof( double ) );
meanVelY = (double *)calloc( numBins, sizeof( double ) );
errVelY = (double *)calloc( numBins, sizeof( double ) );
sumVelZ = (double *)calloc( numBins, sizeof( double ) );
sumVelZ2 = (double *)calloc( numBins, sizeof( double ) );
meanVelZ = (double *)calloc( numBins, sizeof( double ) );
errVelZ = (double *)calloc( numBins, sizeof( double ) );
sumTemp = (double *)calloc( numBins, sizeof( double ) );
sumTemp2 = (double *)calloc( numBins, sizeof( double ) );
meanTemp = (double *)calloc( numBins, sizeof( double ) );
errTemp = (double *)calloc( numBins, sizeof( double ) );

/* Tally statistics */
for ( i=0; i<statFileCount; i++ )
  for ( j=0; j<numBins; j++ ) {
    sumNum[j] += avgND[i][j];
    sumNum2[j] += avgND[i][j]*avgND[i][j];
    sumVelX[j] += avgMVelX[i][j]/MASS;
    sumVelX2[j] += ( avgMVelX[i][j]/MASS )*( avgMVelX[i][j]/MASS );
    sumVelY[j] += avgMVelY[i][j]/MASS;
    sumVelY2[j] += ( avgMVelY[i][j]/MASS )*( avgMVelY[i][j]/MASS );
    sumVelZ[j] += avgMVelZ[i][j]/MASS;
    sumVelZ2[j] += ( avgMVelZ[i][j]/MASS )*( avgMVelZ[i][j]/MASS );
    temp = temperature( avgKE[i][j], avgMVelX[i][j], avgMVelY[i][j],
                       avgMVelZ[i][j] );
    sumTemp[j] += temp;
    sumTemp2[j] += temp*temp;
  }

/* Calculate mean values and error bars */
for ( i=0; i<numBins; i++ ) {
  meanND[i] = sumNum[i]/statFileCount;
  meanVelX[i] = sumVelX[i]/statFileCount;
  meanVelY[i] = sumVelY[i]/statFileCount;
  meanVelZ[i] = sumVelZ[i]/statFileCount;
  meanTemp[i] = sumTemp[i]/statFileCount;
  /* Statistical errors only if more than 1 run */
  if ( statFileCount > 1 ) {
    errND[i] = sqrt( ( sumNum2[i]/statFileCount -
                     meanND[i]*meanND[i] )/(statFileCount-1) );
    errVelX[i] = sqrt( ( sumVelX2[i]/statFileCount -
                       meanVelX[i]*meanVelX[i] )/(statFileCount-1) );
    errVelY[i] = sqrt( ( sumVelY2[i]/statFileCount -
                       meanVelY[i]*meanVelY[i] )/(statFileCount-1) );
    errVelZ[i] = sqrt( ( sumVelZ2[i]/statFileCount -
                       meanVelZ[i]*meanVelZ[i] )/(statFileCount-1) );
    errTemp[i] = sqrt( ( sumTemp2[i]/statFileCount -
                       meanTemp[i]*meanTemp[i] )/(statFileCount-1) );
  }
}

/* Save statistical information */
if ( statFileCount > 0 ) {
  /* Calculate normalization for number density */
  Rc = 0.;
  for ( i=0; i<numBins; i++ ) {
    Rc = Rc + meanND[i]*meanTemp[i];
  }
}

```

```

    Rc /= (numBins*Tc);
    printf( "\n Cold Wall Particle Density: %f\n", (float) Rc );
    fpStatOut = fopen ( statOutFile, "w" );

    /* Output scaling information */
    fprintf( fpStatOut, "#Sets: %d #Bins: %d Rc: %g U0: %g Tc: %g\n",
             statFileCount, numBins, Rc, U0, Tc );

    /* Output scaled numbers */
    for ( i=0; i<numBins; i++ )
        /* Output normalized quantities */
        fprintf ( fpStatOut, "%g %g %g %g %g %g %g %g %g\n",
                 meanND[i]/Rc, errND[i]/Rc,
                 meanVelX[i]/U0, errVelX[i]/U0,
                 meanVelY[i]/U0, errVelY[i]/U0,
                 meanVelZ[i]/U0, errVelZ[i]/U0,
                 meanTemp[i]/Tc, errTemp[i]/Tc );

    fclose ( fpStatOut );
}
printf( "\n Completed processing.\n" );
return 0;
}

double
temperature( KE, pX, pY, pZ )
double      KE;
double      pX;
double      pY;
double      pZ;
{
    double      temp;
    temp = (2*KE)/(3*BOLTZ) - ( pX*pX + pY*pY + pZ*pZ)/(3*BOLTZ*MASS) );
    return temp;
}

void
**calloc( row, col, itemSize )
int      row;
int      col;
unsigned long      itemSize;
{
    char **pointerHead, *dataHead;
    unsigned long      pointerSize, dataSize;
    int i;

    /* Check for item of type double and make sure that if there are an odd
       number of rows (pointers) to add an extra to ensure that the data starts
       on an even 8 byte boundary.  GCC demands this. */

    if ( itemSize == sizeof ( double ) )
        row += row % 2;

    pointerSize = (unsigned long)( row*sizeof( long ) );
    dataSize = (unsigned long)( row*col*itemSize );

    if ( ( pointerHead = (char **)calloc( pointerSize + dataSize,
                                         sizeof( char ) ) ) != (char **)NULL ) {
        dataHead = (char *) ( pointerHead + row );
        for ( i=0; i<row; i++ )
            pointerHead[i] = dataHead + i*col*itemSize;
    }
    else
        printf( "Error allocating space in calloc routine" );

    return ( (void **)pointerHead );
}

```