1997

Particle simulations of microchannel flows with realistic boundaries

Troy Edward Wenski
San Jose State University

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

Recommended Citation

Wenski, Troy Edward, "Particle simulations of microchannel flows with realistic boundaries" (1997). *Master's Theses*. 1480.

DOI: <https://doi.org/10.31979/etd.cj8z-hhd3>

https://scholarworks.sjsu.edu/etd_theses/1480

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

PARTICLE SIMULATIONS OF
MICROCHANNEL FLOWS WITH
REALISTIC BOUNDARIES

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

Troy Edward Wenski

May, 1997

UMI Number: 1384729

UMI Microform 1384729
Copyright 1997, 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

© 1997

Troy Edward Wenski

ALL RIGHTS RESERVED

APPROVED FOR BY THE DEPARTMENT OF PHYSICS

Dr. Alejandro Garcia

Dr. Patrick Hamill

4-10-97

Dr. David Bogy

APPROVED FOR THE UNIVERSITY

ABSTRACT

PARTICLE SIMULATIONS OF MICROCHANNEL FLOWS WITH REALISTIC BOUNDARIES

by Troy E. Wenski

In modern disk drives, the read/write head is aerodynamically designed to fly close ($\approx 25 - 50$ nm) to the moving platter; the determination of the air flow in the narrow gap between the head and platter is a crucial design calculation in this industry. Since the channel spacing is smaller than the mean free path in air (64 nm), the flow is strongly influenced by the nature of the gas-surface interactions. Using experimental data measured by C. Rettner of IBM Almaden, scattering routines can simulate the interactions of relevant materials used in disk drive design and production. The routines are implemented in a Direct Simulation Monte Carlo code using a parallel-processing program architecture running on a Cray T3D. The simulation results show good agreement with the Fukui–Kaneko generalized lubrication equation.

A c k n o w l e d g m e n t s

The work of this thesis has been supported by a number of people. I wish to acknowledge a few individuals who have supported me and contributed to this thesis.

First of all, I thank my thesis advisor, Alex Garcia, who has guided me along the path of this research with patience and wisdom. I am grateful for being given the opportunity to explore the concepts presented here and have gained enormously through the course of this work. In addition, I thank Terry Olson for his relentless search for a particle-surface scattering model and for his friendship along the way. I also thank Charlie Rettner for sharing details of his surface experiments and for his interaction with us. I owe thanks to Erich Gordon for his efforts in producing graphics found in this thesis.

I am indebted to my parents and family for their financial support, guidance, and education throughout the years of my life. They have made this thesis and many other things possible for me.

The majority of this work was completed under the support of the National Science Foundation. Without their funding this project would not have been possible.

Table of Contents

	<i>Page</i>
Acknowledgments.....	v
List of Tables.....	viii
List of Figures.....	ix
1 Introduction	1
2 Kinetic Gas Theory	6
1.1 Boundary Scattering	15
1.2 Basic Theory	6
1.3 Maxwell–Boltzmann Distribution	8
1.3.1 Specular and Thermal Collisions.....	15
1.3.2 Maxwell Accommodation	18
1.3.3 Cercignani–Lampis Scattering Model.....	18
1.3.4 Experimental Measurement of Surface Parameters	28
1.4 Flux Boundary.....	30
1.4.1 Velocity Distribution across a Boundary.....	32
1.4.2 Acceptance–Rejection.....	34
1.4.3 Poisson Distribution.....	35
2 Analytical Formulation	38
2.1 Model of Fluid Motion	38
2.2 Conservation of Mass	39
2.3 Conservation of Momentum	41
2.4 Navier–Stokes Equations of Motion.....	44
2.5 Reynolds Equation.....	45
2.5.1 Continuum Flow	49
2.5.2 First-Order Slip Flow Correction.....	51
2.6 Generalized Lubrication Equation.....	56
3 Parallel Processing	58
3.1 Introduction	58
3.2 Parallel Virtual Machine	58
4 Direct Simulation Monte Carlo Method.....	63
4.1 Direct Simulation Monte Carlo	63

4.2	DSMC Program Description.....	71
5	Simulation Results.....	80
5.1	Simulation Setup	80
5.2	Validation of the Program Output.....	82
5.2.1	Verification at Equilibrium with No Flow	82
5.2.2	Verification of Scattering Routine	86
5.3	Simulation Results for Moving Platter.....	88
5.3.1	Full Accommodation Thermal Boundary	89
5.3.2	Glass Platter.....	93
5.3.3	IBM Disk Platter.....	96
5.3.4	Pt(111) Surface Lubricated with z-dol.....	99
6	Summary and Conclusions	103
	Bibliography	106
Appendix A	PVM example.....	108
A.1	PSDOT Description	108
A.2	PSDOT Source Code.....	111
A.3	SDOT Source Code	115
Appendix B	Simulation Source Code.....	116

List of Tables

<i>Table</i>		<i>Page</i>
Table 2.1	Scattering characteristics of N ₂ from z-dol/Pt(111) at 273 K. The thermal accommodation coefficient, α_n , and the accommodation for parallel momentum, σ_p , are defined by (2-23) and (2-24).	29
Table 2.2	Results of linear fit of experimental scattering data using (2-31) for each surface material.	30
Table 3.1	Summary of “thin film assumptions” used to reduce the Navier-Stokes equations of motion.	48
Table 5.1	List outlining the basic steps of the DSMC algorithm in <i>slider.f</i> . Subroutine calls are shown in parentheses.	73

List of Figures

<i>Figure</i>		<i>Page</i>
Figure 1.1	Schematic of a typical disk drive. The transducer is fixed to the slider which is suspended above the spinning disk by the load beam.	2
Figure 1.2	An exaggerated and simplified illustration of a disk drive air bearing. The channel spacing is less than the mean free path of molecules in air.	3
Figure 2.1	Maxwell-Boltzmann velocity distribution function for air at $T = 273$ K. The dotted line marks the average speed, $\langle v \rangle$	11
Figure 2.2	Path of a hard sphere particle as it collides with other stationary particles.	13
Figure 2.3	Plot of biased distribution for the normal component of velocity for air at STP.	17
Figure 2.4	Geometrical representation of the C-L model for the normal velocity component u . The circle represents a 2-D Gaussian function centered on the u_1 axis.	25
Figure 2.5	C-L model illustrated for the case of tangential velocity components v and w	26
Figure 2.6	Histogram of 10,000 scattered particles. The data series 'x' shows the predicted frequency using the C-L tangential distribution function. The tangential component of incident velocity is 343 m/s and σ_p is set at 0.9.	27
Figure 2.7	Histogram of normal velocity from scattering routine. Data series 'x' shows predicted frequency using the C-L normal distribution function. The normal component of incident velocity is -198 m/s and σ_n is set at 0.9.	28
Figure 2.8	Slider geometry shown with thermal reservoirs at the faces and upper and lower rigid boundaries.	31
Figure 2.9	Plane AB represents the reservoir interface with the system. Only particles within the cylinder can cross the surface dS within the time dt	31
Figure 2.10	Illustration of acceptance-rejection method. Random value (trial) is accepted if the trial point lies below the curve in the shaded region.	35
Figure 3.1	Shaded block represents a test mass moving through space with time. The capital letters label fixed regions in space.	39

List of Figures (cont'd)

<i>Figure</i>		<i>Page</i>
Figure 3.2	The cube represents a volume element. Each face contains two shear stresses and one normal stress.	43
Figure 3.3	The basic slider geometry with the tilt greatly exaggerated. Lower surface moves left to right with a constant velocity U	46
Figure 3.4	Illustration of a simple system with horizontal walls. A typical particle path is shown as it collides in the gas, hits the surface, and is scattered off in a new direction.	52
Figure 3.5	Fluid flow as a function of height in the channel for parallel surfaces. Flow speed is a linear function of height for this geometry.	53
Figure 4.1	Slider system divided by 3 worker processors. The arrows show information paths as particles cross these worker boundaries. The master processor is used for data I/O.	62
Figure 5.1	Schematic of the collision cells used by the DSMC program. Particles are free to have collisions, scatter off surfaces, and exit through the faces.	65
Figure 5.2	Schematic of the collision process. Note that v_{cm} and v_r are conserved. The direction of v_r' is randomly assigned.	69
Figure 6.1	Plot shows 1-Dimensional pressure along the channel length. Pressure is uniform with deviation about 1 atm as expected.	83
Figure 6.2	Plot shows 1-Dimensional temperature down the length of the channel. Temperature is uniform with variation about 273 K as expected.	83
Figure 6.3	Plot of consecutive instantaneous pressure measurements in a vertical slice of the system. The measurements fluctuate about the mean. Only 100 samples are shown here.	85
Figure 6.4	The plot shows the velocity distribution for the normal component of velocity. Incident and reflected velocity distributions are in good agreement. ('o'=incident distribution, 'x'=scattered distribution).	87
Figure 6.5	Incident and reflected velocity distributions for the parallel component of velocity are in good agreement.	88

List of Figures (cont'd)

<i>Figure</i>		<i>Page</i>
Figure 6.6	Slider bearing pressure profile for $Kn_0=0.956$, $\Lambda=40.05$, $h_0=50$ nm, and a fully accommodating platter. Data series include continuum solution, first-order slip correction, Fukui-Kaneko solution, and DSMC data.....	90
Figure 6.7	Slider bearing pressure profile for $Kn_0=1.912$, $\Lambda=160.2$, $h_0=25$ nm, and a fully accommodating platter. Data series include continuum solution, first-order slip correction, Fukui-Kaneko solution, and DSMC data.....	91
Figure 6.8	Slider bearing pressure profile for $Kn_0=4.780$, $\Lambda=1001$, $h_0=10$ nm, and a fully accommodating platter. Data series include continuum solution, first-order slip correction, Fukui-Kaneko solution, and DSMC data.....	92
Figure 6.9	Slider bearing pressure profile for $h_0=50$ nm using the Cercignani-Lampis scattering routine and accommodation parameters for glass.	94
Figure 6.10	Slider bearing pressure profile for $h_0=25$ nm using the Cercignani-Lampis scattering routine and accommodation parameters for glass.	95
Figure 6.11	Slider bearing pressure profile for $h_0=10$ nm using the Cercignani-Lampis scattering routine and accommodation parameters for glass.	96
Figure 6.12	Slider bearing pressure profile for $h_0=50$ nm using the Cercignani-Lampis scattering routine and accommodation parameters for an IBM disk platter.....	97
Figure 6.13	Slider bearing pressure profile for $h_0=25$ nm using the Cercignani-Lampis scattering routine and accommodation parameters for an IBM disk platter.....	98
Figure 6.14	Slider bearing pressure profile for $h_0=10$ nm using the Cercignani-Lampis scattering routine and accommodation parameters for an IBM disk platter.....	99
Figure 6.15	Slider bearing pressure profile for $h_0=50$ nm using the Cercignani-Lampis scattering routine and accommodation parameters for a surface lubricated with z-dol.	100
Figure 6.16	Slider bearing pressure profile for $h_0=25$ nm using the Cercignani-Lampis scattering routine and accommodation parameters for a surface lubricated with z-dol.	101

List of Figures (cont'd)

<i>Figure</i>		<i>Page</i>
Figure 6.17	Slider bearing pressure profile for $h_0=10$ nm using the Cercignani-Lampis scattering routine and accommodation parameters for a surface lubricated with z-dol.	102

Chapter 1

INTRODUCTION

A modern fixed-disk drive found in a typical personal computer consists of one or more metal disks stacked concentrically on a rotating spindle. The disk substrate is covered with a thin layer of magnetic material which serves as a medium for storing and reading data. A small transducer, or read/write head, navigates along the disk by the placement of thin circular concentric tracks on the disk. Data is stored and read from these tracks on the disk. The device resembles a scaled-down version of a phonograph in that a transducer is suspended by a long arm above a spinning disk. In contrast to a phonograph, there is no contact between the read/write head and the magnetically encoded disk. However, a thin protective/lubricating layer lies above the magnetic layer and protects against occasional contact with the read/write head.

Figure 1.1 Schematic of a typical disk drive. The transducer is fixed to the slider which is suspended above the spinning disk by the load beam.

The read/write transducer is fixed to an aerodynamic wedge, called a slider, that is suspended very close to the disk surface by an arm which can navigate among the data tracks. Figure 1.1 shows a typical disk drive slider and arm where the slider bearing has two runners that resemble skis. The flying height between the slider and disk platter is precisely controlled by the lubricating air film that flows through the channel between the head and the platter. The arrangement of a slider, air film, and moving disk surface is called a slider bearing or an air bearing (see Figure 1.2). The very narrow region between the slider surface and disk surface is called a

microchannel. Modern air bearing designs are more complicated than the simple wedge considered here. The basic ideas are the same, however.

Figure 1.2 An exaggerated and simplified illustration of a disk drive air bearing. The channel spacing is less than the mean free path of molecules in air.

The slider floats on the pressurized air in the channel which is an effect of the geometry of the slider. Air is dragged through the channel by the spinning disk. At the channel entrance the slider height from the disk is the greatest while the least height is typically located near the channel exit. As air flows through the channel, the gas compresses and becomes pressurized due to the tapering of the channel. Thus, the increased pressure in the channel provides lift to the slider surface.

The prescribed flying height of the slider bearing is very important for correct operation of the disk drive. The optimal flying height depends on the magnetic properties of the recording surface, the desired recording density, and the minimum allowable error rate. Disk heads that flew at 200 nm a few years ago now fly at 25 nm resulting in much greater data storage capacity. The method of calculating the

flying height is extremely critical for the realization of future super-capacity disk drives.

Lubrication theory has provided an analytical formulation of the pressure profile in the channel region. The resulting nonlinear differential equations are called Reynolds-type lubrication equations. The fundamental assumption of the Reynolds equation is that the mean free path of a lubricating molecule, approximately 50 nm for air, is small compared to the spacing between the surfaces. This assumption is violated by the small bearing spacing and the Reynolds formulation must be adjusted to account for interactions at the particle level rather than continuum level. Molecular Gas Lubrication (MGL) theory, formulated by Fukui and Kaneko, resembles the Reynolds equation but uses flow rates obtained from approximate solutions of the Boltzmann equation.

The focus of this work is to test the validity of the Reynolds-type equations at extremely small flying heights. For comparison, the well-founded Direct Simulation Monte Carlo (DSMC) method has been adapted to the slider problem. DSMC is a numerical simulation method that models gas dynamics at the particle level. In addition, realistic boundary conditions have been incorporated into the simulation by using actual surface scattering data to model particle-surface interactions. The methods will be compared for various bearing heights and for various disk platter materials that are relevant to disk drives. The results of the comparison show that the method developed by Fukui and Kaneko agrees well with the DSMC results for all

channel heights and platter materials while simple Reynolds equation approximations do not.

Chapter 2

KINETIC GAS THEORY

2.1 Basic Theory

The focus of this research is to study the flow of a gas through a microchannel. If this gas is dilute then the analysis can be greatly simplified since two-body interactions can be assumed. Complicated interactions involving more than two molecules are less likely since the particles are, on average, spaced far apart in a dilute gas. The criterion for a dilute gas is that the average distance between particles is large compared to the effective diameter, d_{eff} . In the case for air, the effective diameter is an average value of the molecular diameters of air constituents. Consider the gas to occupy a volume V that contains N particles. This criterion for dilute gas may be written as,

$$d_{eff} \ll \sqrt[3]{V / N} \quad (2-1)$$

For air at STP, the mass density is 1.29 kg/m^3 and $\sqrt[3]{V / N}$ is $33.4 \text{ \AA}$. The effective diameter is $4.19 \text{ \AA}$ and satisfies (2-1) by a small but acceptable margin. The dilute gas approximation for air is valid at gas pressures up to 100 atm. Since the temperature and pressure of air in the microchannel stay near STP for test conditions (less than 2

atm for tested slider configurations), the assumption for a dilute gas will always be valid in the simulations.

One way to model the dilute gas is to assume that the interaction force for a pair of particles is only a function of their relative separation

$$\mathbf{F}_{ij} = \mathbf{F}(\mathbf{r}_i - \mathbf{r}_j) = -\mathbf{F}_{ji} \quad (2-2)$$

where $\mathbf{F}_{ij}$ is the force on particle i due to particle j and $\mathbf{r}_i - \mathbf{r}_j$ gives their relative separation. Once the force is determined, either numerically or theoretically, the entire dynamics of the system is given by the equations of motion

$$\frac{d^2}{dt^2} \mathbf{r}_i = \frac{1}{m} \sum_{\substack{j=1 \\ j \neq i}}^N \mathbf{F}_{ij} \quad (2-3)$$

where m is the mass of a particle. In principle, the future state of this system can be found by evaluating the system of ODEs given the initial state of the system. This method, referred to as molecular dynamics, has been successful in computing microscopic properties of fluids [2].

Molecular dynamics is limited to microscopic systems because (2-3) is computationally expensive since the number of particle pairs is large and goes as N^2 as the system volume increases. The microchannel system studied here has a volume

of roughly 0.9 cubic microns. At STP this system will contain roughly 2×10^7 molecules.

Since the number of particles is very large even for extremely small systems, statistical mechanics and kinetic theory are used to create a much more feasible method of numerically modeling this system. This method is called Direct Simulation Monte Carlo (DSMC). The method does not attempt to predict the instantaneous state of an individual particle. Instead, DSMC takes advantage of the enormous number of particles and randomly selects collisions using known statistical properties from kinetic theory. Before the DSMC method can be formulated for a dilute gas, some properties of the statistical distributions that describe the gas must be known. The gas in a DSMC simulation is not Maxwell–Boltzmann distributed unless the gas is at equilibrium. However, many properties from equilibrium statistical mechanics (e.g., temperature) are useful in formulating the method.

2.2 Maxwell–Boltzmann Distribution

An classical ideal gas can be defined as a system of weakly interacting particles in the classical regime. A free particle is one that may be confined to a box but has no restrictions or external forces acting on it. The classical regime resides in the limit of low particle concentrations where the average number of atoms in each quantum state is much less than unity.

Consider the Fermi–Dirac (FD) and Bose–Einstein (BE) distribution functions [11] given by

$$P(E) = \frac{1}{e^{(E-\mu)/kT} \pm 1} \quad (2-4)$$

where the plus sign is for the FD distribution, the minus sign for the BE distribution and k is the Boltzmann constant. The quantity E is the energy of a particular state and μ is the chemical potential and $P(E)$ is the probability for energy E . The classical regime requires that $P(E)$ be much smaller than unity for all possible states. This requirement leads to the inequality

$$e^{(E-\mu)/kT} \gg 1 \quad (2-5)$$

which allows the ± 1 term in the denominator of (2-4) to be neglected. When (2-5) is satisfied, both the FD and BE distributions become

$$P(E) \cong e^{(\mu-E)/kT} = Ae^{-E/kT} \quad (2-6)$$

where $A = e^{\mu/kT}$. This result is called the classical distribution function and will be used to describe an ideal gas.

For a particle in an ideal gas, it is assumed that its energy is purely kinetic,

$$E(\vec{r}, \vec{v}) = \frac{1}{2} m |\vec{v}|^2 \quad (2-7)$$

This assumption is valid since interparticle forces are short-ranged and their potential energy may be neglected. In a dilute gas, the particle spacing is large enough so that particles spend much of their time traveling with constant velocity while occasionally interacting with single particles in binary collisions.

Returning to the classical distribution function, the probability that a particle has a velocity between $\mathbf{v}$ and $\mathbf{v} + d\mathbf{v}$ is independent of position. This independence of $\mathbf{r}$ means that the particle is equally likely to be found anywhere inside the volume V . By making use of the isotropy of the distribution, the probability density may now be expressed as

$$P(\mathbf{v})d\mathbf{v} = A v^2 e^{-\frac{1}{2}mv^2/kT} dv \sin(\theta) d\theta d\phi \quad (2-8)$$

The constant A can be determined so as to normalize the distribution; the integral of probability over all possible states must equal unity. Imposing this condition on A and integrating the angular parts gives

$$P(v)dv = 4\pi \left(\frac{m}{2\pi kT} \right)^{3/2} v^2 e^{-\frac{1}{2}mv^2/kT} dv \quad (2-9)$$

where $P(v)dv$ is the probability that a particle's speed is between v and $v + dv$. This distribution is called the Maxwell-Boltzmann distribution and is shown in Figure 2.1.

Figure 2.1 Maxwell-Boltzmann velocity distribution function for air at $T = 273$ K. The dotted line marks the average speed, $\langle v \rangle$.

The MB distribution can be used to compute various useful quantities that characterize the gas. For example, the average particle speed for a given gas temperature may be found by integrating the distribution over all possible speeds by

$$\langle v \rangle = \int_0^{\infty} v P(v) dv = \frac{2}{\pi} \sqrt{\frac{2kT}{m}} \quad (2-10)$$

and the root mean square particle speed is given by

$$\sqrt{\langle v^2 \rangle} = \sqrt{\int_0^{\infty} v^2 P(v) dv} = \sqrt{\frac{3kT}{m}} \quad (2-11)$$

Using (2-11), the average kinetic energy may be defined as

$$\langle K \rangle = \left\langle \frac{1}{2} m v^2 \right\rangle = \frac{3}{2} kT \quad (2-12)$$

The peak of the velocity distribution, the most probable speed, can be found by differentiating and maximizing the distribution function,

$$\begin{aligned} \frac{d}{dv} P(v) \Big|_{v=v_{mp}} &= 0 \\ v_{mp} &= \sqrt{\frac{2kT}{m}} \end{aligned} \quad (2-13)$$

where v_{mp} is the velocity of maximum probability.

2.3 Mean Free Path and Collision Frequency

Consideration of particle interactions (i.e., collisions) is required in the formulation of the DSMC method. There are two primary quantities necessary for the formulation of the method: the mean free path of a particle in a dilute gas and the frequency of collisions.

To begin, consider an ideal gas as a cloud of tiny billiard balls of diameter d . These particles collide elastically whenever their separation equals this diameter. The mean free path, λ , is the average distance traveled by a particle between collisions. The average number of collisions per particle per unit time is called the collision frequency and is denoted by f . These two quantities are related to the average particle speed $\langle v \rangle$ by

$$\lambda = \frac{\langle v \rangle}{f} \quad (2-14)$$

To determine the collision frequency, consider the volume swept out by a particle's path as it collides with other particles one at a time. This volume can be approximated as a cylindrical tube centered on the path of the particle. The cylinder will have a radius of d since this is the separation at contact for hard spheres. Figure 2.2 illustrates the jagged path of a particle and the tube swept out by its path.

Figure 2.2 Path of a hard sphere particle as it collides with other stationary particles

Consider now a single particle moving for a time t_0 with average relative velocity $\langle v_r \rangle$ as in Figure 2.2. Despite the fact that the tube has sharp bends, its volume is approximated as a single cylinder,

$$V_{tube} = \pi d^2 \ell = \pi d^2 \langle v_r \rangle t_0 \quad (2-15)$$

where ℓ is the total length of the path of the particle. The collision frequency is then equal to the number of particles contained within the cylinder divided by the time t_0 .

Since the gas is assumed homogenous, the number of particles in the tube is

$$M_{tube} = \frac{N}{V} \pi d^2 \langle v_r \rangle t_0 \quad (2-16)$$

where N is the number of gas molecules found in a volume V . The collision frequency is then given by

$$f = \frac{\langle M_{tube} \rangle}{t_0} = \frac{N}{V} \pi d^2 \langle v_r \rangle \quad (2-17)$$

The result of (2-17) assumes that only one particle moves while all others are held fixed. In a real gas all particles move. To correct for this, $\langle v_r \rangle$ should be distinguished from $\langle v \rangle$ of (2-10) and identified as the average relative speed between particles

$$\begin{aligned} \langle v_r \rangle &= \langle |\mathbf{v}_1 - \mathbf{v}_2| \rangle = \iint d\mathbf{v}_1 d\mathbf{v}_2 |\mathbf{v}_1 - \mathbf{v}_2| P(\mathbf{v}_1) P(\mathbf{v}_2) \\ &= \frac{4}{\sqrt{\pi}} \sqrt{\frac{kT}{m}} \end{aligned} \quad (2-18)$$

where $P(\mathbf{v})$ is given by (2-8). After substituting (2-17) and (2-18) into (2-14), the mean free path is

$$\lambda = \frac{V}{N\pi d^2 \langle v_r \rangle} = \frac{V}{\sqrt{2} N\pi d^2} \quad (2-19)$$

For air at STP, f is roughly 9×10^9 collisions per particle per second and λ is approximately 48 nm.

2.4 Boundary Scattering

In addition to interparticle collisions, there are also boundary interactions (e.g., gas-surface scattering) that must be considered. In the system studied here, the surfaces consist of upper and lower rigid walls. Boundary conditions on the left and right sides of the system are discussed in the next section. The lower wall moves from left to right with a constant velocity while the upper wall is stationary but also tilted at a small angle. The simulation treats these walls as geometric planes. Particles are free to interact with these boundaries and a complete treatment of this interaction would involve quantum mechanics. Instead of using a complicated model for these interactions, simple but adequate mechanical approximations to these process are used here.

2.4.1 Specular and Thermal Collisions

Two of the simplest models for surface interactions include specular and thermal collisions. The first type of interaction, specular, may be described as the way a billiard ball reflects off a bumper. Since the mass of the wall is much greater than the mass of a single particle and if the collision between the wall and particle is purely

elastic, the particle merely reflects off the wall. Only the component of velocity normal to the wall is reversed in sign after the collision. The two tangential components of velocity are completely conserved. A specular reflection can occur when the velocity of the particle is nearly parallel to the surface of the wall. For the case of a glancing impact, specular reflection is more likely.

If the particle happens to be in contact with the wall long enough to exchange energy, the wall can fully thermalize the particle. Consequently, particles that interact with the wall in this way have their energy modified such that, on average, the distribution of post-interaction velocities matches the equilibrium distribution of a gas at the temperature of the wall. In contrast to specular collisions, the velocity resulting from a thermal collision has no correlation to the velocity prior to the collision. The two components of velocity parallel to the surface, u and w , can be drawn from the Maxwell-Boltzmann distribution (2-9) using Gaussian functions. Only velocities directed away from the wall are allowed for the normal component of velocity. The normal velocity, v , is drawn from the biased Maxwellian distribution. Each component is distributed according to the probability densities,

$$\begin{aligned}
 P(v) &= \pm 2\beta^2 v e^{-\beta^2 v^2} \\
 P(u) &= \frac{\beta}{\sqrt{\pi}} e^{-\beta^2 u^2} \\
 P(w) &= \frac{\beta}{\sqrt{\pi}} e^{-\beta^2 w^2}
 \end{aligned}
 \tag{2-20}$$

where $\beta = \sqrt{\frac{m}{2kT_w}}$ and T_w is the temperature of the wall. The theory for these

boundary scattering distributions are presented in section 2.5.

The plus sign for $P(v)$ in (2-20) corresponds to a surface normal in the +y direction. This normal component velocity distribution is biased toward faster moving particles due to the term v located outside the exponential term. Thus, it is more likely for faster moving particles to reflect off a wall since they are more likely to reach the wall (see the discussion below of reciprocity). Figure 2.3 is a plot of the biased Maxwellian distribution.

Figure 2.3 Plot of biased distribution for the normal component of velocity for air at STP.

2.4.2 Maxwell Accommodation

Another model for gas-surface interactions, Maxwell accommodation, assumes that each surface collision is either specular or thermal. A probability for thermal collisions, α , may be set and the probability for specular collisions is then $1 - \alpha$.

The value for α will vary for different materials. A light particle (e.g., hydrogen) scattered from an extremely smooth surface may have an accommodation of 0.7. A surface with a high accommodation, a value of 0.9 or higher, may be a rough surface where most particles scatter thermally.

2.4.3 Cercignani–Lampis Scattering Model

To date, most DSMC simulations of rarefield gas flows have assumed diffuse thermal reflection or have adopted the Maxwell accommodation method for reflection of gas particles from a solid surface. Molecular beam studies have shown that neither of these two models, which were discussed in the previous section, are realistic (see next section). These experiments show that reflected molecules, on the average, retain some momentum parallel to the surface. This is not possible with the two models presented so far. If the collision is specular then all parallel momentum is conserved, but if the collision is diffuse then no parallel momentum is conserved on the average.

What is needed is a model that will satisfy detail balance (reciprocity), produce physically reasonable velocity and energy reemission distributions, and

provide a continuous spectrum of scattering interactions from specular reflection to diffuse thermal reflection. Cercignani and Lampis [4] have produced a model, the C-L model, that meets these requirements.

The general requirement for equilibrium between a gas and a solid surface at the molecular level is that the interaction should satisfy the reciprocity condition. It is a condition that relates the probability for a pair of incident and reflected velocities with the probability of the inverse action. Reciprocity demands that, at equilibrium, the distributions of velocities and energies must be conserved. Specifically, a gas at thermodynamic equilibrium with the surface must preserve a Maxwell-Boltzmann distribution. Both the diffuse and specular scattering models, in addition to other scattering models, satisfy this condition for a gas in equilibrium with a surface. The reciprocity condition can be formulated mathematically by considering a velocity $\mathbf{c}_i$ incident on a surface and a velocity $\mathbf{c}_r$ reflected from the surface. The vector $\hat{\mathbf{n}}$ is the unit normal pointing away from the surface. At equilibrium, the probability for velocity $\mathbf{c}_i$ is proportional to $e^{-E_i/kT}$ where $E_i = \frac{1}{2}m|\mathbf{c}_i|^2$ using (2-9). The net rate at which particles hit the surface with $\mathbf{c}_i$ is proportional to $-\mathbf{c}_i \cdot \hat{\mathbf{n}} e^{-E_i/kT}$. The reciprocity condition can now be written for any distribution function P as

$$\mathbf{c}_r \cdot \hat{\mathbf{n}} P(-\mathbf{c}_r, -\mathbf{c}_i) e^{-E_r/kT} = -\mathbf{c}_i \cdot \hat{\mathbf{n}} P(\mathbf{c}_i, \mathbf{c}_r) e^{-E_i/kT} \quad (2-21)$$

This can be qualitatively expressed as

$$\left(\begin{array}{c} \text{Net rate of } \mathbf{c}_r \\ \text{hitting surface} \end{array} \right) \cdot \left(\begin{array}{c} \text{Pr obability that } \mathbf{c}_r \\ \text{becomes } \mathbf{c}_i \text{ after collision} \end{array} \right) = \left(\begin{array}{c} \text{Net rate of } \mathbf{c}_i \\ \text{hitting surface} \end{array} \right) \cdot \left(\begin{array}{c} \text{Probability that } \mathbf{c}_i \\ \text{becomes } \mathbf{c}_r \text{ after collision} \end{array} \right) \quad (2-22)$$

As an example, consider a pair of velocities $\mathbf{c}_i$ and $\mathbf{c}_r$ such that molecules with velocity $\mathbf{c}_i$ are much more common than with velocity $\mathbf{c}_r$. Reciprocity, by (2-21), then requires that the scattering probability $P(\mathbf{c}_i, \mathbf{c}_r)$ be much less than $P(-\mathbf{c}_r, -\mathbf{c}_i)$ since $e^{-E_r/kT}$ is much less than $e^{-E_i/kT}$ (i.e., since $P(\mathbf{c}_r) < P(\mathbf{c}_i)$). Moreover, the scattering distribution function should scatter a large number of particles to a velocity of $\mathbf{c}_i$ since $\mathbf{c}_i$ is common (has high probability) in the gas. The scattering function is conserving the large fraction of molecules with velocity $\mathbf{c}_i$. On the other hand, fewer particles will scatter off the surface with a velocity of $\mathbf{c}_r$ since it is less common in the gas. Hence, the relatively small fraction of particles with velocity $\mathbf{c}_r$ is being conserved, on average.

Only those gas-surface scattering models that satisfy the reciprocity condition can be used in DSMC simulations. Without reciprocity the second law of thermodynamics is violated (a gas and surface initially at the same temperature will not remain at the same temperature), which is clearly unacceptable. Interestingly, curve-fits of experimentally measured scattering distributions often violate reciprocity [17]. While such curve-fits are good for describing the experimental data, they should

be used indirectly to extract parameters, such as accommodation coefficients, for use in distributions that do satisfy reciprocity.

Away from thermodynamic equilibrium the velocity distributions of the incident and reflected molecules are allowed to have different temperatures. Let T_i be the temperature of gas that is in equilibrium with particles incident on the surface and let T_r be the temperature of a gas of reflected particles. Also, T_r may differ from the temperature of the surface T_w . The energy (or thermal) accommodation coefficient is defined by

$$\alpha = \frac{\langle E_i \rangle - \langle E_r \rangle}{\langle E_i \rangle - \langle E_w \rangle} \quad (2-23)$$

where $\langle E_i \rangle$ and $\langle E_r \rangle$ are respectively the incident and reflected average kinetic energy. $\langle E_w \rangle$ is the expectation value of kinetic energy for a fully thermalized gas leaving the wall with $T_r = T_w$. The thermal accommodation coefficient is an indication of the extent to which the reflected molecules have their temperature adjusted toward that of the surface. For the two previous models, an α of zero corresponds to no accommodation which is true of a specular reflection. An accommodation value of unity corresponds to complete thermal accommodation or diffuse reflection.

It is possible to formulate accommodation parameters associated with a particular component of velocity. Normal energy accommodation, α_n , is expressed using (2-23) where the average energies are associated with the normal components of velocity.

Similarly, an accommodation value for momentum parallel to the surface may be defined as

$$\sigma_p = \frac{\langle p_i \rangle - \langle p_r \rangle}{\langle p_i \rangle} \quad (2-24)$$

where $\langle p_i \rangle$ and $\langle p_r \rangle$ are the incident and reflected average momenta. The momentum quantities are defined relative to the reference frame of the surface. The relation between parallel momentum accommodation and parallel energy accommodation is obtained using a careful consideration of average incoming velocity and outgoing velocity parallel to the surface,

$$1 - \alpha_p = (1 - \sigma_p)^2 \quad (2-25)$$

Using (2-24) and (2-25) the average value for a tangential component of velocity is

$$\langle v_r \rangle = (1 - \alpha_p)^{1/2} \langle v_i \rangle \quad (2-26)$$

The C-L model assumes that there is no coupling between the normal and tangential components of the velocity during the reflection process. The distribution

for the normal component of velocity, u , presented by Cercignani and Lampis is given by

$$P(u_i, u_r) = \left(\frac{2u_r}{\alpha_n} \right) I_0 \left\{ 2(1-\alpha_n)^{1/2} \frac{u_i u_r}{\alpha_n} \right\} \exp \left[-\frac{u_r^2 + (1-\alpha_n)u_i^2}{\alpha_n} \right] \quad (2-27)$$

where u_i and u_r are the incident and reflected velocities normalized to the most probable molecular speed at the surface temperature and I_0 is given by the modified Bessel function. This distribution is normalized and satisfies the reciprocity condition.

The C-L scattering probability distribution for a tangential velocity component is

$$P(v_i, v_r) = \frac{1}{\sqrt{\pi\sigma_p(2-\sigma_p)}} \exp \left\{ -\frac{[v_r - (1-\sigma_p)v_i]^2}{\sigma_p(2-\sigma_p)} \right\} \quad (2-28)$$

For isotropic surfaces, the two tangential components of velocity, v and w , must behave identically since no direction in the plane of the surface is preferred over any other. These velocities must be exactly similar in form and obey the same distribution function. The probability density for the magnitude of tangential velocity $(v^2 + w^2)^{1/2}$ is also given by the normal velocity distribution function (2-27) using appropriate changes of variable. Therefore, one probability density function can then be used to describe the reflected state for the velocity components u , v , and w .

The distribution function in (2-27) is not easily inverted because its integral is very complicated. However, it can be expressed as a two-dimensional Gaussian distribution centered off the origin,

$$P(r, \theta) = \frac{1}{\pi\alpha} e^{-r^2/\alpha} \quad (2-29)$$

where r is the distance from the center of the distribution and α is the normal or tangential accommodation [14]. Note that all values of θ are equally probable as required by an isotropic surface.

Consider first using the distribution for the normal component of velocity represented in Figure 2.4. The state of the reflected particle is located by the coordinates (u_1, u_2) which are two independent random Gaussian distributed numbers. The incident velocity, u_i , and the center of the two-dimensional distribution, u^* , are located on the u_l axis. The location of u^* relative to u_i obeys an expression exactly similar in form to equation (2-26). The magnitude of the reflected velocity is expressed as

$$u_r = (u_1^2 + u_2^2)^{1/2} \quad (2-30)$$

and will also satisfy the distribution function given by (2-27).

Figure 2.4 Geometrical representation of the C-L model for the normal velocity component u . The circle represents a 2-D Gaussian function centered on the u_1 axis.

This method works similarly for the tangential velocities v and w . Now the Gaussian is located at the center (v^*, w^*) given by (2-26) using α_p for accommodation. The reflected velocities, v_r and w_r , are located separately by two Gaussian random numbers and the center of the distribution as shown in Figure 2.5.

Figure 2.5 C-L model illustrated for the case of tangential velocity components v and w .

The schemes illustrated in Figure 2.4 and Figure 2.5 should be verified in that the C-L distribution functions given by (2-27) and (2-28) are satisfied. A routine has been constructed that draws from the C-L distribution using the schemes described in the figures. The scattering routine receives the same incident velocity at each call and a scattering distribution can be constructed. The results indicate that the routine agrees with the C-L distribution functions using the same incident velocity (see Figure 2.6 and Figure 2.7). The incident particle is an air molecule having a kinetic energy of 0.035 eV. The surface temperature is set at 273 K.

Figure 2.6 Histogram of 10,000 scattered particles. The data series 'x' shows the predicted frequency using the C-L tangential distribution function. The tangential component of incident velocity is 343 m/s and σ_p is set at 0.9.

Figure 2.7 Histogram of normal velocity from scattering routine. Data series 'x' shows predicted frequency using the C-L normal distribution function. The normal component of incident velocity is -198 m/s and α_n is set at 0.9.

2.4.4 Experimental Measurement of Surface Parameters

Surface accommodation parameters have been experimentally measured by analysis of the velocity and angular distributions of scattered molecular beams [16]. The molecular beam consists of a collimated burst of N_2 molecules with very narrow and highly controlled energy distribution. The beam is aimed at a scattering surface with an incident angle θ_i . A detector, placed at various scattering angles θ_f , senses the scattered burst of particles. The time between the beam pulse and the detector

response is called the time of flight, TOF. By correlating all of the TOF distributions for a given incident angle, accommodation parameters can be found.

Various surfaces of relevance to the disk-drive air bearing were studied by Rettner using the beam scattering technique. These surfaces include an actual 3.5-in disk, a Pt(111) surface lubricated with a commercial perfluoro polyether called z-dol, and sections of a glass disk used in fly-height testing. Results for z-dol/Pt(111) are shown in Table 2.1. The other tested surfaces show similar results. At incident energies characteristic of a gas at room temperature, all surfaces were essentially fully accommodating.

Table 2.1 Scattering characteristics of N_2 from z-dol/Pt(111) at 273 K. The thermal accommodation coefficient, α_n , and the accommodation for parallel momentum, σ_p , are defined by (2-23) and (2-24).

E_i (eV)	θ_i (deg.)	α_n	σ_p
0.171	30	0.86 ± 0.11	0.82 ± 0.08
0.171	45	0.83 ± 0.07	0.83 ± 0.06
0.171	60	0.78 ± 0.06	0.82 ± 0.06
0.086	30	0.94 ± 0.03	0.94 ± 0.04
0.086	45	0.94 ± 0.03	0.94 ± 0.03
0.086	60	0.95 ± 0.03	0.94 ± 0.03
0.076	30	0.97 ± 0.03	0.96 ± 0.04
0.076	45	0.96 ± 0.02	0.96 ± 0.03
0.076	60	0.95 ± 0.02	0.94 ± 0.03

The principal use of scattering data for this research is to extract accommodation parameters for use in the C-L scattering model. The accommodation results show

that both coefficients decrease with increasing collision energy. Only the incident energy dependence of the accommodation is considered since the C-L model has no angular dependence. A simple linear fit of the scattering results for each surface has been performed using

$$\begin{aligned}\alpha_n &= 1 - m_n E_i \\ \sigma_p &= 1 - m_p E_i\end{aligned}\tag{2-31}$$

where m_n and m_p are the linear fit parameters for the thermal and momentum accommodations and E_i is the incident energy of the particle. Results of the data fit have been presented in Table 2.2.

Table 2.2 Results of linear fit of experimental scattering data using (2-31) for each surface material.

Surface Material	m_n (eV) ⁻¹	m_p (eV) ⁻¹
z-dol/Pt(111)	0.709397	0.796853
Glass	0.992389	0.651207
Disk	1.096200	0.557888

2.5 Flux Boundary

In addition to the rigid thermal boundaries discussed in section 2.4, there are also flux boundaries to consider. The flux boundaries of the slider system consist of open faces at each end where particles are allowed to move in and out freely. The regions outside of these boundaries are considered to be fluid reservoirs of constant mass density ρ_o , average fluid velocity (u_o , v_o , w_o), and temperature T_o (see Figure 2.8).

Figure 2.8 Slider geometry shown with thermal reservoirs at the faces and upper and lower rigid boundaries.

If the flow of the system is, for instance, left to right, then the reservoir on the left will inject many more particles than its counterpart on the right. The question remains to find out how many particles will cross a face within a time interval and what is their velocity distribution.

Figure 2.9 Plane AB represents the reservoir interface with the system. Only particles within the cylinder can cross the surface dS within the time dt

2.5.1 Velocity Distribution across a Boundary

Begin by considering the reservoir on the left hand side of the system as shown in Figure 2.9. The wall AB forms the boundary between the reservoir and the system. Only particles moving to the right will cross plane AB . Let the velocity of a particle in the reference frame of the plane be $\mathbf{c} = (u, v, w)$ where u , v , and w are the $\hat{\mathbf{i}}$, $\hat{\mathbf{j}}$, and $\hat{\mathbf{k}}$ components of velocity, respectively. Define the function f such that

$$f(\mathbf{c}) d\mathbf{c} dV = n \quad (2-32)$$

where n is the number of particles with velocity between $\mathbf{c}$ and $\mathbf{c} + d\mathbf{c}$ within the volume element dV . The velocity of a particle in the frame of the fluid is $\mathbf{C} = \mathbf{c} - \mathbf{c}_0$ where $\mathbf{c}_0$ is the average fluid velocity. Now, the velocity of a particle in the frame of the plane AB can be rewritten as

$$\mathbf{C} + u_0 \hat{\mathbf{i}} = u \hat{\mathbf{i}} + (v - v_0) \hat{\mathbf{j}} + (w - w_0) \hat{\mathbf{k}} \quad (2-33)$$

It is desired to find how many particles cross dS in a time dt . Any particles with velocity between $\mathbf{C}$ and $\mathbf{C} + d\mathbf{C}$ would cross dS if they were within the volume $dV = u dt dS$. Thus, the number of particles within dV and having a velocity within this range is given by

$$f(\mathbf{C}) d\mathbf{C} dV = f(\mathbf{C}) d\mathbf{C} u dt dS \quad (2-34)$$

Since the reservoir fluid is an ideal gas, the distribution of particle velocities is Maxwellian. The velocity distribution of particles crossing the plane AB is then given by

$$\begin{aligned}
 P(\mathbf{C}) &= \frac{1}{\alpha} u e^{-\beta^2 C^2} \\
 &= \left[\frac{1}{\alpha_x} u e^{-\beta^2 (u-u_0)^2} \right] \left[\frac{1}{\alpha_y} e^{-\beta^2 (v-v_0)^2} \right] \left[\frac{1}{\alpha_z} e^{-\beta^2 (w-w_0)^2} \right] \\
 &= P_x(u) P_y(v) P_z(w)
 \end{aligned} \tag{2-35}$$

where $u \in [0, \infty]$; $v, w \in [-\infty, \infty]$; $\beta = \sqrt{m/2kT_0}$ and m is the mass of particles.

Setting each probability component to unity and integrating over velocity the normalization constants are

$$\begin{aligned}
 \alpha_x &= \frac{1}{2\beta^2} e^{-\beta^2 u_0^2} + \frac{\sqrt{\pi}}{2\beta} u_0 (1 + \operatorname{erf}(\beta u_0)) \\
 \alpha_y &= \alpha_z = \sqrt{\pi}/\beta
 \end{aligned} \tag{2-36}$$

where $\operatorname{erf}(x)$ is the error function.

These distributions may also be used to model a rigid thermal boundary discussed in the previous section. The distributions expressed by (2-20) are arrived at by setting u_0 , v_0 , and w_0 equal to zero in (2-35) and (2-36). The result is the distribution of velocities coming off a rigid thermal wall. A rigid thermal wall and a thermal reservoir are statistically equivalent when the flow is removed from the reservoir.

It is important to note that particles behave identically in the direction parallel to the plane of the interface. Velocities in the y and z direction are both Gaussian distributed. The direction normal to the plane is non-Gaussian. The results of (2-35) and (2-36) may be used for any orientation using normal and tangential coordinates. They may also be used for non-stationary rigid walls. If the wall is not stationary, then at least one component of $\mathbf{c}_o$ must be non-zero.

2.5.2 Acceptance-Rejection

Generating the y and z components of velocity from (2-35) is easy since these Gaussian distributions are easily inverted. Unfortunately, the x component of velocity is not easy to find since it is difficult to invert the distribution for u . Instead, acceptance-rejection can be used to generate random values for u . The most probable x -component of velocity, u_m , can be found by differentiating (2-35) with respect to u and setting it equal to zero. The result for the most probable velocity is

$$u_m = \frac{1}{2} \left(u_0 + \sqrt{u_0^2 + 2v_{th}^2} \right) \quad (2-37)$$

where $v_{th} = 1/\beta$ is the most probable thermal velocity. Since it is necessary to pick a velocity from a finite range, a reasonable cut-off velocity is defined by

$$u_{top} = u_m + \min(6v_{th} + u_m) \quad (2-38)$$

It has been found that the integral of $P_x(u)$ over the cut-off region is negligibly small.

Figure 2.10 Illustration of acceptance–rejection method. Random value (trial) is accepted if the trial point lies below the curve in the shaded region.

Figure 2.10 illustrates the acceptance–rejection process for u . If the random point, (P_{trial}, u_{trial}) , is below the curve defined by $P_x(u)$, then the trial value is accepted. Begin by selecting a trial u value as $u_{trial} = u_{top} R_1$ and a random height as $P_{trial} = P_m R_2$ where, R_1 and R_2 are uniformly distributed random numbers in $[0, 1)$.

The trial value is accepted if

$$P_x(u_{trial}) > P_{trial} \quad (2-39)$$

If the trial value is rejected, the procedure is repeated.

2.5.3 Poisson Distribution

The average number of particles crossing the plane AB during a time Δt remains to be found. The distribution function $f(C)$ defined by (2-32) is Maxwellian and may be written as

$$f(C) = \frac{1}{\gamma} e^{-\beta^2 C^2} \quad (2-40)$$

The normalization of $f(C)$ is given by

$$\gamma = \frac{\pi^{1/2}}{\beta^3 n} \quad (2-41)$$

where n is the number of particles in the reservoir per unit volume. Using (2-35) the average number of particles, $\langle N_p \rangle$, crossing the plane AB is

$$\begin{aligned} \langle N_p \rangle &= \int_{t_1}^{t_2} dt \int_0^{L_y L_z} dS \int_0^\infty du \int_{-\infty}^\infty dv \int_{-\infty}^\infty dw \frac{\beta^3}{\pi^{1/2}} n u e^{-\beta^2 C^2} \\ &= \Delta t L_y L_z n \frac{1}{2\beta\sqrt{\pi}} \left\{ e^{-\beta^2 u_0^2} + \beta\sqrt{\pi} u_0 [1 + \text{erf}(\beta u_0)] \right\} \end{aligned} \quad (2-42)$$

where $L_y L_z$ is the cross-section of plane AB and n is the number density.

It is now needed to determine the distribution function to which the average number belongs. That is, what is the probability that N_p particles cross the plane AB in a time interval Δt ? Consider a particle crossing the plane to be an event. These events are occurring randomly in time. Each event can be represented by a point on the time axis where the points are randomly distributed. Suppose that now the time axis is broken up into non-overlapping intervals all of which are mutually independent. Since there is a discrete number of events in each interval, the desired distribution function will be restricted to discrete probabilities rather than continuous

distributions. The probability that N events occur during an interval is given by the Poisson distribution.

There are many physical experiments and statistical observations that fit the above description. A few examples include radioactive particle detection, incoming calls at a telephone exchange, and even the distribution of flying-bomb hits on London during World War II [6]. All of these observations fit the Poisson distribution remarkably well.

The Poisson distribution may then be used to randomly generate numbers of particles that will cross the plane. The distribution should be peaked at the average number of particles crossing the plane, given by (2-42), and is written as

$$P(N_p) = \frac{\langle N_p \rangle^{N_p} e^{-\langle N_p \rangle}}{N_p!} \quad (2-43)$$

Efficient algorithms for obtaining Poisson distributed random numbers and for evaluating the error function are discussed in Numerical Recipes [15].

Chapter 3

ANALYTICAL FORMULATION

3.1 Model of Fluid Motion

In this section we develop the partial differential equations that model the flow of a fluid in a region of space. Fluid motion can be represented mathematically using a series of various approximations of the actual flow situation. The resulting model is then limited by these approximations and must be used only in proper applications. In order to establish the model, the fluid is considered to be a continuous medium. Later, the model will be adjusted to account for individual molecules in a rarefield gas.

3.2 Conservation of Mass

Figure 3.1 Shaded block represents a test mass moving through space with time. The capital letters label fixed regions in space

Consider a mass of fluid occupying a region of space, also called the control volume, at time t represented as region A in Figure 3.1. At any time the mass in region A is denoted by $m_A(t)$. At a later time $t + \Delta t$, the fluid once contained by region A has now moved to region D . The mass contained in region A at this new time is given by

$$m_A(t + \Delta t) = m_A(t) + m_C(t + \Delta t) - m_B(t + \Delta t) \quad (3-1)$$

Rearranging and dividing by Δt ,

$$\frac{m_A(t + \Delta t) - m_A(t)}{\Delta t} = \frac{m_C(t + \Delta t) - m_B(t + \Delta t)}{\Delta t} \quad (3-2)$$

Taking the limit as Δt goes to 0, the left side of (3-2) becomes

$$\lim_{\Delta t \rightarrow 0} \frac{m_A(t + \Delta t) - m_A(t)}{\Delta t} = \frac{\partial}{\partial t}(m)_{C.V.} = \frac{\partial}{\partial t} \int_{C.V.} \rho V \quad (3-3)$$

where ρ is the mass density and dV is the differential volume element. The subscript $C.V.$ on the volume integral refers to the control volume fixed in space, or region A , which is bounded by the control surface, $C.S.$ The right side of (3-2) represents the mass rate of flow in and out of the control volume and becomes

$$\begin{aligned} \lim_{\Delta t \rightarrow 0} \frac{m_C(t + \Delta t) - m_B(t + \Delta t)}{\Delta t} &= \int_C \rho v \cos(\gamma) dS - \int_B \rho v \cos(\gamma) dS \\ &= - \int_{C.S.} \rho \mathbf{v} \cdot d\mathbf{S} \end{aligned} \quad (3-4)$$

where $\mathbf{v}$ is the velocity vector of a streamline in the fluid and γ is the angle between the velocity vector and the outward normal, $\mathbf{S}$. Equating the last two expressions yields the integral form of the continuity equation given as

$$\int_{C.S.} \rho \mathbf{v} \cdot d\mathbf{S} = - \frac{\partial}{\partial t} \int_{C.V.} \rho dV \quad (3-5)$$

This result says that the net rate of mass flow out of the control surface is equal to the time rate of decrease of mass inside the control volume.

The left side of continuity equation can be converted into a volume integral using Gauss' theorem and (3-5) becomes

$$\int_{C.S.} \nabla \cdot (\rho \mathbf{v}) dV + \frac{\partial}{\partial t} \int_{C.V.} \rho dV = \int_{C.V.} \left[\nabla \cdot (\rho \mathbf{v}) + \frac{\partial \rho}{\partial t} \right] dV = 0 \quad (3-6)$$

Since the control volume is arbitrary, the integrand must be zero which results in the differential form of the continuity equation as

$$\frac{\partial \rho}{\partial t} + \frac{\partial(\rho u)}{\partial x} + \frac{\partial(\rho v)}{\partial y} + \frac{\partial(\rho w)}{\partial z} = 0 \quad (3-7)$$

where u , v , and w are the Cartesian components of $\mathbf{v}$.

3.3 Conservation of Momentum

The equation of continuity alone is not enough information to determine the flow of a system. Conservation of momentum provides additional information needed to determine the flow. The momentum equation for the control volume will be developed here.

Let $\mathbf{F}$ be the net force acting on a system with a fixed mass. Using Newton's second law

$$\mathbf{F} = \frac{d\mathbf{M}}{dt} \quad (3-8)$$

where $\mathbf{M}$ is the total linear momentum of the system. Assuming that the force is constant over a small time Δt , the total change in momentum is given by

$$\Delta \mathbf{M} = \mathbf{F} \Delta t \quad (3-9)$$

Using Figure 3.1 for the momentum in region A, the total change in momentum may be expressed as

$$\Delta \mathbf{M} = \mathbf{M}_A(t + \Delta t) - \mathbf{M}_C(t + \Delta t) + \mathbf{M}_B(t + \Delta t) - \mathbf{M}_A(t) \quad (3-10)$$

Rearranging and dividing by Δt ,

$$\frac{\Delta \mathbf{M}}{\Delta t} = \frac{\mathbf{M}_A(t + \Delta t) - \mathbf{M}_A(t)}{\Delta t} + \frac{\mathbf{M}_B(t + \Delta t) - \mathbf{M}_C(t + \Delta t)}{\Delta t} \quad (3-11)$$

By taking the limit of (3-11) as Δt goes to 0, the first term on the right becomes

$$\lim_{\Delta t \rightarrow 0} \frac{\mathbf{M}_A(t + \Delta t) - \mathbf{M}_A(t)}{\Delta t} = \frac{\partial}{\partial t} \mathbf{M}_{C.V.} = \frac{\partial}{\partial t} \int_{C.V.} \rho \mathbf{v} dV \quad (3-12)$$

The second term of (3-11) represents the rate at which momentum is crossing into and out of the control volume and it becomes

$$\begin{aligned} \lim_{\Delta t \rightarrow 0} \frac{\mathbf{M}_B(t + \Delta t) - \mathbf{M}_C(t + \Delta t)}{\Delta t} &= \int_B \mathbf{v} \rho \mathbf{v} \cos(\alpha) dS - \int_C \mathbf{v} \rho \mathbf{v} \cos(\alpha) dS \\ &= \int_{C.S.} \mathbf{v} \rho \mathbf{v} \cdot d\mathbf{S} \end{aligned} \quad (3-13)$$

The force in (3-8) can now be rewritten. This force can be separated into a surface force $\mathbf{F}$, and a body force $\mathbf{B}$ and is written as

$$\mathbf{F}_S + \int_{C.V.} \mathbf{B} dV = \frac{\partial}{\partial t} \int_{C.V.} \mathbf{v} \rho dV + \int_{C.S.} \mathbf{v} \rho \mathbf{v} \cdot d\mathbf{S} \quad (3-14)$$

The body force is a force per unit volume which could originate from a central potential such as an electric field or gravitational field.

The differential form of the momentum equation is arrived at by applying the integral form to an elemental volume. Let this volume be a cube in Cartesian coordinates as shown in Figure 3.2.

Figure 3.2 The cube represents a volume element. Each face contains two shear stresses and one normal stress.

Each face of the cube contains three orthogonal stresses of which two are shear stresses parallel to the face and one normal stress which is perpendicular to the face. Each stress, σ_{ij} , can be indexed where the face of action of the stress is perpendicular

to the i axis. The index j gives the direction of the stress. The stresses can be assembled into a stress tensor σ . The stress tensor must be symmetric, $\sigma_{ij} = \sigma_{ji}$, because the fluid is isotropic unlike a solid which has crystal planes. Since the stress tensor composes the surface force $\mathbf{F}$, of (3-14), the momentum balance equation for the cube in the x direction is written as

$$\begin{aligned}
 & (\sigma_{11}|_{x+\Delta x} - \sigma_{11}|_x) \Delta y \Delta z + (\sigma_{21}|_{y+\Delta y} - \sigma_{21}|_y) \Delta x \Delta z + \\
 & \quad (\sigma_{31}|_{z+\Delta z} - \sigma_{31}|_z) \Delta x \Delta y + B_x \Delta x \Delta y \Delta z = \\
 & \frac{\partial}{\partial t} (\rho u) \Delta x \Delta y \Delta z + (\rho u^2|_{x+\Delta x} - \rho u^2|_x) \Delta y \Delta z + \\
 & \quad (\rho uv|_{y+\Delta y} - \rho uv|_y) \Delta x \Delta z + (\rho uw|_{z+\Delta z} - \rho uw|_z) \Delta x \Delta y
 \end{aligned} \tag{3-15}$$

3.4 Navier–Stokes Equations of Motion

The difference terms in (3-15) become partial differentials after dividing by $\Delta x \Delta y \Delta z$ and taking the limit as the cube widths go to zero. The resulting expression may be combined with the equation of continuity (3-7) by expanding all derivatives using the product rule and then combining terms. The result of this process is given by

$$\rho \left(\frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} + v \frac{\partial u}{\partial y} + w \frac{\partial u}{\partial z} \right) = \frac{\partial \sigma_{11}}{\partial x} + \frac{\partial \sigma_{12}}{\partial y} + \frac{\partial \sigma_{13}}{\partial z} + B_x \tag{3-16}$$

This can be extended to the other two coordinates and written in Cartesian tensor notation,

$$\rho \left(\frac{\partial u_i}{\partial t} + u_j \frac{\partial u_i}{\partial x_j} \right) = \frac{\partial \sigma_{ij}}{\partial x_j} + B_i \quad (3-17)$$

For frictional flow, the stress tensor for a Newtonian fluid is expressed as [13]

$$\sigma_{ij} = -p\delta_{ij} + \mu \left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} - \frac{2}{3}\delta_{ij} \frac{\partial u_k}{\partial x_k} \right) + \xi \delta_{ij} \frac{\partial u_k}{\partial x_k} \quad (3-18)$$

where p is the normal stress due to isotropic pressure and δ_{ij} is the Kronecker delta.

The shear and bulk viscosities are μ and ξ , respectively; for a dilute gas $\xi = 0$. This expression may be put into the momentum equation (3-17) resulting in the complete equation of motion for a viscous fluid. These are known as the Navier–Stokes equations of motion and are written as

$$\begin{aligned} \rho \left(\frac{\partial u_i}{\partial t} + u_j \frac{\partial u_i}{\partial x_j} \right) = & -\frac{\partial p}{\partial x_i} + B_i + \frac{\partial}{\partial x_i} \left(\xi \frac{\partial u_k}{\partial x_k} \right) \\ & + \frac{\partial}{\partial x_i} \left[\mu \left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} - \frac{2}{3}\delta_{ij} \frac{\partial u_k}{\partial x_k} \right) \right] \end{aligned} \quad (3-19)$$

3.5 Reynolds Equation

At the continuum level, the equations of motion given by the Navier–Stokes relations tell the complete story for the motion of an arbitrary viscous fluid. These equations can be greatly simplified by considering the geometry of the slider problem, for example, by reducing the treatment to only two dimensions. This result in

conjunction with thin film assumptions will lead to the formulation of the Reynolds equation.

Figure 3.3 The basic slider geometry with the tilt greatly exaggerated. Lower surface moves left to right with a constant velocity U .

The hydrodynamic bearing shown in Figure 3.3 is one of the basic gas film lubrication geometries. In a gas film, non-contact support between two surfaces is achieved by the pressurized gas between them. A simple example of this is when a piece of paper gently slides across a smooth table. The paper floats aerodynamically above the table and is supported by the gas below it. A typical system can be characterized by its system length l and the clearance height h_0 which are shown in Figure 3.3. An important parameter in kinetic theory is the Knudsen number Kn which is defined as

$$Kn = \frac{\lambda}{h} \quad (3-20)$$

where λ is the molecular mean free path and h is a function of position along the channel.

The first type of flow considered is continuum flow. This corresponds to an extremely small Kn or in the limit where it goes to zero. The dimensions of this type of system are much larger than the dimensions of individual particle behavior. For example, air flow past a baseball corresponds to $Kn \approx 10^{-6}$, so sports calculations can be solved using a continuum treatment. In this region the effects of individual fluid particles are washed out.

The second flow category considered is slip flow where the Knudsen number is still small but significantly different from zero. A slip flow theory is one that gives a correction to continuum flow. It is used in problems such as the hydrodynamic wedge shown here where individual particle behavior must be considered. A problem requiring slip correction is one where the dimensions are approaching the particle level.

The Reynolds equation originates from simplifications of the Navier-Stokes equation (3-19) and the equation of continuity (3-7). If the fluid is an ideal gas then the pressure and density are proportional for a given temperature by the ideal gas law,

$$p = \rho RT \quad (3-21)$$

where R is the gas constant and T is the gas temperature. Since the clearance height is much smaller than the system length, it is expected for the flow in the vertical direction to be null since the gas physically has nowhere to go. The vertical gradient for the x -component of velocity, u , will be large since the gas velocity at the bottom

will be close to the platter speed and close to zero at the top of the system. Another assumption is that inertial forces of the gas and any body forces are much smaller than viscous forces. This is a good assumption since the Reynolds number, $Re = \rho u h / \mu$, is small for the system. The last assumption allows the second term on the left-hand side of (3-19) to become negligible; for steady flows the first term is also zero. It is also assumed that no body force acts on the fluid. For a one-dimensional treatment the flow in the z -direction, w , is assumed to be zero. These assumptions are summarized in Table 3.1.

Table 3.1 Summary of “thin film assumptions” used to reduce the Navier-Stokes equations of motion.

Assumption	Result
1.) $h \ll \ell$	$v \approx 0,$ $\frac{\partial u}{\partial y} \gg \frac{\partial u}{\partial x}, \frac{\partial u}{\partial z}$
2.) Inertial forces negligible, $Re \ll 1$	$\rho u_j \frac{\partial u_i}{\partial u_j} = 0$
3.) Steady flow	$\frac{\partial u}{\partial t} = 0$
4.) Negligible body force	$\mathbf{B} = 0$
5.) 2-dimensional treatment	$w = 0$

Putting all of these assumptions into the equations of motion (3-19), the simplified macroscopic equations are

$$\frac{\partial^2 u}{\partial y^2} = \frac{1}{\mu} \frac{\partial p}{\partial x} \quad (3-22)$$

$$\frac{\partial p}{\partial y} = 0 \quad (3-23)$$

and the time-independent equation of continuity, (3-7), becomes

$$\frac{\partial}{\partial x}(\rho u) = 0 \quad (3-24)$$

The average equation of continuity can be obtained by integrating in the y -direction

$$\frac{\partial}{\partial x} \left(\int_0^h \rho u dy \right) = 0 \quad (3-25)$$

The velocity profile $u(y)$ is obtained by integrating (3-22) twice with the appropriate boundary conditions. Finally, the Reynolds equation is obtained by putting the solution of equation (3-22) into equation (3-25) and integrating once again.

Boundary conditions must be considered since the result is a differential equation.

The type of Reynolds equation obtained depends on the treatment of the boundary conditions.

3.5.4 Continuum Flow

The boundary conditions of the slider geometry under continuum flow are

$$\begin{aligned} u|_{y=0} &= U \\ u|_{y=h} &= 0 \end{aligned} \quad (3-26)$$

where U is the velocity of the lower boundary.

Since the left-hand side of (3-22) depends only on y and the right-hand side depends only on x , the equation of motion for u can be treated as a second-order ordinary differential equation. The solution using the boundary conditions of equation (3-26) is given by

$$u = -\frac{1}{2\mu} \frac{\partial p}{\partial x} (h-y)y + U(1-y/h) \quad (3-27)$$

The first term on the right-hand side of (3-27) is a flow due to a pressure gradient which is also called Poiseuille flow. The second term depends only on y and must be a flow due to shear forces and is referred to as Couette flow. Substituting (3-27) into (3-25) gives

$$\frac{\partial}{\partial x} \left(\rho h^3 \frac{dp}{dx} \right) = 6\mu U \frac{\partial}{\partial x} (\rho h) \quad (3-28)$$

Using (3-21) and (3-23), the Reynolds equation for a classical compressible fluid under isothermal conditions ($\rho \propto p$) is given by

$$\frac{d}{dx} \left(p h^3 \frac{dp}{dx} \right) = 6\mu U \frac{d}{dx} (p h) \quad (3-29)$$

This equation can be expressed in non-dimensional form,

$$\frac{d}{dX} \left(PH^3 \frac{dP}{dX} \right) = \Lambda \frac{d}{dX} (PH) \quad (3-30)$$

where

$$P = \frac{p}{p_a}, \quad X = \frac{x}{\ell}, \quad H = \frac{h}{h_o}, \quad \Lambda = \frac{6\mu U \ell}{p_a h_o^2} \quad (3-31)$$

The parameter Λ is a conventional bearing number which denotes the ratio of shear flow rate to pressure flow rate. The height h_o is the exit height or the smallest vertical height in the system. The ambient air pressure is denoted by p_a and the system length is labeled ℓ as in Figure 3.3.

The last result is applicable in the continuum region, that is, in cases where the Knudsen number is practically zero. The Knudsen numbers for the hydrodynamic bearings considered in this thesis are significantly different from zero. As will be shown later, the Reynolds equation for continuum flow does a poor job of predicting the pressure profile in this thin-film application.

3.5.5 First-Order Slip Flow Correction

A slip flow correction must be used when the Knudsen number of the system is small but significantly different from zero. Velocity slip occurs when the flow velocity of the gas at the boundaries does not match the velocity of the boundaries.

When the dimensions of a system are nanoscale, the velocity slip cannot be ignored as with problems of a much larger scale. First-order slip correction is a modification to the boundary conditions that accounts for this difference in velocities at the boundaries. It is based on the Maxwell scattering model for a particle–surface interface.

Figure 3.4 Illustration of a simple system with horizontal walls. A typical particle path is shown as it collides in the gas, hits the surface, and is scattered off in a new direction.

For simplicity, consider a system that has two parallel walls with the upper wall stationary and the lower moving with velocity U as shown in Figure 3.4. The surface interaction shown in the picture represents the average scattering process for all particles that come into contact with the lower boundary. The quantity u_1 represents the average x -velocity of the particles just before colliding with the wall while u_2 is the average x -velocity just after collision.

The average perpendicular distance from the wall to the last point of interparticle collision is roughly a mean free path as show in Figure 3.4. This is the last point in the fluid where particles “feel” the flow of the fluid (i.e., interact with other particles) before colliding with the boundary.

Since there is no tilt on the upper wall, there will be no pressure gradient and consequently no pressure flow. The solution of (3-22) will then show that the flow is a linear function of y as shown below in Figure 3.5. The slope $\frac{du}{dy}$ must negative since the upper wall is fixed and the lower wall is moving; the velocity of the gas at the lower boundary will be moving faster than the gas at the upper boundary.

Figure 3.5 Fluid flow as a function of height in the channel for parallel surfaces. Flow speed is a linear function of height for this geometry.

The average gas velocity at the lower boundary is the average of the particle velocities just before and after collision with the wall,

$$u(0) = \frac{1}{2}(u_1 + u_2) \quad (3-32)$$

From Figure 3.5 the gas velocity at the wall is also given by

$$\begin{aligned} u(0) &= u(\lambda) - \lambda \frac{du}{dy} \\ &= u_1 - \lambda \frac{du}{dy} \end{aligned} \quad (3-33)$$

Parallel momentum accommodation, (2-24), is defined in the reference frame of the scattering surface. In the stationary lab frame, the average parallel velocity coming off of the wall is given by

$$u_2 = (1 - \sigma_p)u_1 + \sigma_p U \quad (3-34)$$

Using (3-32), (3-33), and (3-34), the velocity u_1 can be expressed as

$$u_1 = U + \frac{2\lambda}{\sigma_p} \frac{du}{dy} \quad (3-35)$$

Substituting (3-35) into (3-33), the corrected x -velocity of the gas at the lower wall is

$$u(0) = U + \left(\frac{2 - \sigma_p}{\sigma_p} \right) \frac{du}{dy} \quad (3-36)$$

Note that the gas velocity at the lower boundary $u(0)$ is different from the boundary velocity U . The result says that slip is nonzero and at a minimum if the walls are fully accommodating ($\sigma_p = 1$). Slip increases for lower accommodations. If the channel spacing was large, then $\frac{du}{dy}$ would be very small making slip negligible as with continuum flow.

A similar expression to (3-36) can be found for the gas velocity at the upper boundary using the same treatment. The new modified boundary conditions for a general system with the lower boundary moving are given by

$$\begin{aligned} u|_{y=0} &= U + b\lambda \left. \frac{du}{dy} \right|_{y=0} \\ u|_{y=h} &= -b\lambda \left. \frac{du}{dy} \right|_{y=h} \end{aligned} \quad (3-37)$$

where $b = \frac{2 - \sigma_p}{\sigma_p}$.

The solution of (3-22) with boundary conditions of (3-37) is

$$u = -\frac{1}{2\mu} \frac{dp}{dx} (bh\lambda + hy - y^2) + U \left(1 - \frac{yb\lambda}{h + 2b\lambda} \right) \quad (3-38)$$

Substitution of equation (3-38) into equation (3-25) and assuming an isothermal condition gives

$$\frac{d}{dx} \left[ph^3 \left(1 + \frac{6b\lambda}{h} \right) \frac{dp}{dx} \right] = 6\mu U \frac{d}{dx} (ph) \quad (3-39)$$

The dimensionless version of the last equation, using the normalized quantities of (3-31), is given by

$$\frac{d}{dX} \left[PH^3 (1 + 6bK_n) \frac{dP}{dX} \right] = \Lambda \frac{d}{dX} (PH) \quad (3-40)$$

The last result is a solution of the flow equation using boundary conditions that have been corrected to account for slip flow. Comparing the last expression and the continuum case (3-30), the effect of the slip correction is the presence of the term $(1 + 6bK_n)$. Note that in the limit that K_n goes to zero, the slip-corrected equation reduces to the continuum expression as expected.

In this derivation the assumption is that particles approaching the lower wall had their last collision in the fluid rather than with the upper wall (see Figure 3.4). This assumption limits the slip-corrected Reynolds equation to systems where $K_n < 0.1$. As will be shown in the results, this equation performed poorly since the channels studied have $K_n \approx 1$.

3.6 Generalized Lubrication Equation

Fukui and Kaneko developed a theory to examine the flow characteristics for arbitrary Knudsen numbers [7]. This molecular gas film lubrication equation (MGL) is not a correction to continuum flow like other methods. The MGL equation is,

rather, derived from the Boltzmann equation, the mass flow conservation law, and flow rate estimations. The MGL equation is

$$\frac{d}{dX} \left[\bar{Q}_p(K_n) PH^3 \frac{dP}{dX} \right] = \Lambda \frac{d}{dX} (PH) \quad (3-41)$$

where the function $\bar{Q}_p(K_n)$ is the non-dimensional flow rate to Poiseuille flow. For a fully accommodating surface ($\alpha = 1$), an approximation for the flow rate is

$$\bar{Q}_p(K_n) \approx 1 + 6AK_n + \frac{12}{\pi} \log(1 + BK_n) \quad (3-42)$$

where the parameters $A=1.318889$ and $B=0.387361$ are selected so as to give a best fit to the tabulated database [1].

As will be shown in the results, the MGL equation does very well for all Knudsen numbers that were simulated. It is possible that this equation will fail at higher platter speeds or with a more complex wall geometry such as a sinusoidal surface.

Chapter 4

PARALLEL PROCESSING

4.1 Introduction

Before introducing the DSMC particle simulation method, it will be useful to explain the parallel processing technique used here. A particle simulation essentially keeps track of particle positions and velocities. Since the number of simulation particles is large (typically 16,000 to 50,000 in our simulations), the calculation can take several days for a single processor on a workstation. Parallel processing offers a way to put more processors to work on updating the particle lists thus greatly reducing the total run time of the simulation.

4.2 Parallel Virtual Machine

Parallel Virtual Machine (PVM) is a software system designed to permit a heterogeneous collection of UNIX computers networked together to be used as a single virtual machine. The collection can consist of a local area network of several different types of machines. Or on the other hand, it can consist of massively parallel processors (MPP) such as the Cray T3D used in this thesis, the Intel Paragon, and IBM SP-2. PVM can run on a wide variety of modern platforms. It is well accepted

by the global computing community and is used in solving problems that require enormous computing power.

MPPs are currently the most powerful computers in the world. These machines utilize anywhere from a few hundred to a few thousand processors packed into a single cabinet. They typically have hundreds of gigabits of memory and may cost more than \$10 million. The computer used in this research is the Cray T3D with 256 DEC EV4 Alpha processors with each having 64 Mbytes of local memory. In an MPP, every processor is exactly like every other in capability, resources, software, and communication speed.

Whether using an MPP or distributed computing on a local area network, the notion of message passing is common to both. The central idea in parallel processing is that data must be exchanged in cooperative tasks. PVM software exploits message passing so that parallel applications can be written in a straightforward manner using existing hardware. PVM consists of a library of FORTRAN routines that provide the framework so that any process may communicate and/or synchronize with any other.

The parallel programming paradigm used here is referred to as data decomposition or partitioning. This method assumes that the overall problem consists of computational operations on data structures, such as arrays containing positions and velocities of particles, and that these data structures may be divided and operated upon separately by different processors. A master-slave structure is used to

decompose the data and distribute it among the slave processors. The master reads in or generates the initial data set and then parses out the data to be transformed by the other processors.

Each processor has its own processor identification number (PID) which is obtained by calling a PVM subroutine. This number can be used for master-slave determination and for direction of messages. Separate programs are not written for the master or slave, rather the code compiled and running on each processor is identical. The instructions to be carried out by a particular processor depend on its PID through the logic of the code. For example, consider a program that spawns 8 processors. The PIDs will range from 0 to 7. The code could be written with simple IF statements so that processor 0 is in control of data I/O. Hence, processor 0 is considered to be the master and processors 1 through 7 are the workers.

As a simple example of data decomposition, consider the scalar multiplication of two vectors, $A[1..N]$ and $B[1..N]$, to produce the dot product of A and B . If it is desired to have P processors work on the problem, then data partitioning will allocate N/P elements of each vector to all of the processors. Processor 0, the master processor, may break up the vectors and send the parts to the workers. Each processor will do its part of the product and report the result to the master processor. The master, in turn, will take all of the local products and add them up to form the desired scalar product of the vectors A and B . A working PVM program that performs the dot product described above is presented and explained in appendix A.

Extension of this method of parallel computing to a DSMC particle simulator follows naturally as shown in Figure 4.1. The vectors in this case are long lists of particle positions and velocities. The lists are broken up according to particle position. Since each processor will have its own physical portion of the total system, each will have its own unique list of particles. In the beginning, these lists are formed independently by each processor. All workers are performing identical operations on their own particle lists. When a particle moves from the region handled by processor *X* to that controlled by processor *Y*, a message is passed from *X* to *Y* to add the particle to the list maintained by processor *Y*. In addition, the particle is removed from the list in processor *X* since it is no longer in the region belonging to *X*. The message passing occurs at every time step.

After all workers have completed their part of the simulation they each report their statistical data to the master processor. The master takes the individual data sets and consolidates them to reflect the entire system and then saves these results into output files.

Figure 4.1 Slider system divided by 3 worker processors. The arrows show information paths as particles cross these worker boundaries. The master processor is used for data I/O.

ORNL PVM version 3.3 has been used in this particle simulation. The slider code uses PVM libraries for FORTRAN 77. PVM software is public domain and can be found on the Internet. For a more complete description of PVM and its capabilities see reference [9].

Chapter 5

DIRECT SIMULATION MONTE CARLO METHOD

5.1 Direct Simulation Monte Carlo

To study the flow of a dilute gas using molecular dynamics, the positions and velocities of each particle are needed, $\{\mathbf{r}_i, \mathbf{v}_i\}$ for $i = 1, \dots, N$. Approaching the problem in this way is computationally expensive since the trajectory of one particle depends on all other particles in the simulation. Therefore, molecular dynamics is an $O(N^2)$ method where the problem scaling is nonlinear and an individual collision can take several timesteps to evaluate. The Direct Simulation Monte Carlo (DSMC) method reduces the undesirable nonlinear scaling of the problem.

The DSMC method reduces the problem size by assuming that particles only interact through binary collision. Each particle's velocity remains unchanged until it is allowed to collide with another particle. The post-collision velocities for a pair of particles are instantaneously set to appropriate values using kinetic gas theory. Since interaction only occurs at the instant of collision, the simulation particles otherwise behave like an ideal-gas. At the expense of reducing to an ideal-gas, the DSMC method reduces the scaling of the problem to $O(N)$ and increases the timesteps from

an intra-collision time scale to an inter-collision time scale (i.e., mean free time between collisions).

The simulation consists of moving particles at each time step, and then allowing interactions between them. Particles are first moved as if they have had no interactions according to

$$\mathbf{r}_i^{new} = \mathbf{r}_i^{old} + \mathbf{v}_i \tau \quad (5-1)$$

where τ is the simulation timestep. After all particles have been moved, only some are selected for collision. During the collision process the particle's velocities are reset. This process is iterated for the desired number of steps.

To implement the collision process particles are sorted into cells after they have been moved (see Figure 5.1). The purpose of the cells is to group particles according to their proximity so that only pairs in the same cell are allowed to collide. The size of the cell can be determined from the mean free path of the gas. A particle should not be allowed to skip over an entire cell during a single time step thus skipping over likely interactions. The cell dimensions are not allowed to be greater than a mean free path.

Figure 5.1 Schematic of the collision cells used by the DSMC program. Particles are free to have collisions, scatter off surfaces, and exit through the faces.

Now that the entire system has been moved and grouped into collision cells, collision pairs must be processed. Each particle in the cell is a candidate for collision. In this dilute gas, or hard sphere model, the probability for collision depends not on relative position within the cell but only on relative speed of particles. It has been shown that making use of particle positions within a cell does not matter as long as the cells are no larger than a mean free path in size. The probability of collision for a pair of particles, i and j , is given by

$$P_{coll}(i, j) = \frac{|\mathbf{v}_i - \mathbf{v}_j|}{\sum_{m=1}^{N_c} \sum_{n=1}^{m-1} |\mathbf{v}_m - \mathbf{v}_n|} \quad (5-2)$$

where N_c is the number of particles in the cell. The collision probability for a pair is then proportional to the relative velocity divided by a normalization constant. It is more probable for particles with a high relative velocity to collide within the cell than a pair with a low relative velocity.

The double sum in (5-2) is computationally expensive. This expression can be modified and made more efficient by using an acceptance–rejection scheme to draw from this distribution. Begin the method by choosing a pair of candidates, i and j , at random. The criterion for accepting or rejecting the pair is given by

$$\frac{|\mathbf{v}_i - \mathbf{v}_j|}{v_r^{\max}} > R \quad (5-3)$$

where $v_r^{\max}$ is the maximum relative velocity in the cell, and R is a uniform random deviate. The pair is accepted if (5-3) is satisfied and the collision is processed. The scheme picks the pair exactly according to (5-2). The quantity $v_r^{\max}$ serves a similar purpose as the double sum in (5-2) which normalizes the distribution. The probability for acceptance of a pair is proportional to the relative velocity. The scheme is valid only if $v_r^{\max}$ is indeed greater than the relative velocity of any particle pair in the cell.

Instead of computing $v_r^{\max}$ for each cell, it will be overestimated in such a way that the acceptance–rejection process does not become inefficient.

The number of collisions during the timestep, τ , for each cell can be calculated using the collision frequency (2-17),

$$M_{coll} = \frac{1}{2} N_c f \tau = \frac{N_c^2 \pi d^2 \langle v_r \rangle \tau}{2V_c} \quad (5-4)$$

where V_c is the volume of the cell. Using the fact that all collision pairs go through an acceptance–rejection scheme, the ratio of total accepted pairs to total candidates is

$$\frac{M_{coll}}{M_{cand}} = \frac{\langle v_r \rangle}{v_r^{\max}} \quad (5-5)$$

By plugging (5-5) into (5-4) the number of collision candidates is given by

$$M_{cand} = \frac{N_c^2 \pi d^2 v_r^{\max} \tau}{2V_c} \quad (5-6)$$

This result gives the total number of pairs to pick as candidates for collision in the acceptance–rejection routine

Notice from (5-5) and (5-6) that the selection of $v_r^{\max}$ is somewhat arbitrary since if $v_r^{\max}$ is greatly overestimated there will be many more candidates and hence more pairs will be rejected. In this simulation $v_r^{\max}$ is chosen to be $3v_{mp}$ where v_{mp} is the

most probable velocity of the gas. If a pair is found to have a $v_r^{\max}$ greater than this, the routine will reset $v_r^{\max}$ to the greater value.

Once the collision pair has been chosen, their post-collision velocities need to be computed. Conservation of linear momentum demands that the center of mass velocity of the pair must remain unchanged by the collision,

$$\mathbf{v}_{cm} = \frac{1}{2}(\mathbf{v}_i + \mathbf{v}_j) = \frac{1}{2}(\mathbf{v}_i^* + \mathbf{v}_j^*) = \mathbf{v}_{cm}^* \quad (5-7)$$

where $\mathbf{v}^*$ represents a post-collision velocity. Also, conservation of energy requires that the magnitude of the relative velocity of the pair remain unaltered by the collision,

$$v_r = |\mathbf{v}_i - \mathbf{v}_j| = |\mathbf{v}_i^* - \mathbf{v}_j^*| = v_r^* \quad (5-8)$$

The vectors $\mathbf{v}_i^*$ and $\mathbf{v}_j^*$ represent six unknown quantities that must be determined.

Three constraints for the unknowns are given by (5-7) and one constraint is given by (5-8). The collision process is illustrated in Figure 5.2.

Figure 5.2 Schematic of the collision process. Note that $\mathbf{v}_{cm}$ and v_r are conserved. The direction of $\mathbf{v}_r'$ is randomly assigned.

The remaining two unknowns are fixed by the angles, θ and ϕ , for the relative velocity,

$$\mathbf{v}_r' = v_r [(\sin \theta \cos \phi) \hat{\mathbf{x}} + (\sin \theta \sin \phi) \hat{\mathbf{y}} + (\cos \theta) \hat{\mathbf{z}}] \quad (5-9)$$

For the hard sphere model, these angles are uniformly distributed over the unit sphere. For any particle pair selected for collision, all angles are equally likely for the relative velocity vector after collision. The relative velocity angles are distributed according to the probability densities,

$$P(\phi)d\phi = \frac{1}{2\pi}d\phi \quad (5-10)$$

$$P(\theta)d\theta = \frac{1}{2}\sin(\theta)d\theta \quad (5-11)$$

The azimuth angle, ϕ , is uniformly distributed between 0 and 2π and can be picked as

$$\phi = 2\pi R_1 \quad (5-12)$$

where R_1 is a uniform deviate in $[0, 1)$.

For the angle θ , a convenient change of variable

$$q = -\cos\theta \quad (5-13)$$

gives $P(q)dq = (\frac{1}{2})dq$. The quantity q is uniformly distributed in $[-1, 1]$ and can be generate using another uniform deviate R_2 . To randomly generate the post-collision relative velocity vector (5-9), compute the quantities

$$\begin{aligned} q &= 2R_2 - 1 \\ \sin\theta &= \sqrt{1 - q^2} \end{aligned} \quad (5-14)$$

as well as the angle ϕ . The post collision velocities can now be calculated by

$$\begin{aligned} \mathbf{v}_i^* &= \mathbf{v}_{cm}^* + \frac{1}{2}\mathbf{v}_r^* \\ \mathbf{v}_j^* &= \mathbf{v}_{cm}^* - \frac{1}{2}\mathbf{v}_r^* \end{aligned} \quad (5-15)$$

After all collisions have been evaluated in all cells, the process repeats from the beginning. The particles will be moved, then sorted, and finally collided again. The process continues for the desired number of time steps.

5.2 DSMC Program Description

The system is divided up evenly among processors into subsystems by the main routine (see Figure 4.1). Each subsystem is then divided into collision cells along the x and y -axes (see Figure 5.1). The system is assumed to be homogenous in the z -direction so this position is not recorded.

Each simulation particle represents a larger number of real particles. The effective number of particles depends on the number of simulation particles, N_s , and is given by

$$N_{eff} = \frac{\rho V}{m N_s} \quad (5-16)$$

where ρ is the density of the gas, m is the mass of a gas particle, and V is the volume of the gas. The basic idea is that each simulation particle is considered to represent an effective number of particles located at roughly the same position with roughly the same velocity. Each simulation particle has the mass of one real particle.

Statistically, though, there are N_{eff} real particles that will behave identically to each simulation particle. This quantity along with the collision cell structure determine the quality of the simulation. As a guideline, the collision cell structure and effective

number are chosen such that there are at least 20 particles per cell which experience has shown to be enough to give good results [3].

Collision cell dimensions, L_c , are chosen such that $L_c \leq \lambda$. The time step is set as $\tau = \gamma L_c / \langle v \rangle$. In these simulations $\gamma = 0.2$. The condition imposed on the cell width ensures that particles cannot skip over entire cells in the movement process. In addition, the dimension constraint ensures that every particle within a cell can collide with any other particle in the same cell.

An outline of the main routine, *slider.f*, is given in Table 5.1. Code listings for all the routines are given in Appendix B.

Table 5.1 List outlining the basic steps of the DSMC algorithm in *slider.f*. Subroutine calls are shown in parentheses.

-
- Start PVM
 - Initialize all quantities and particles (*starter.f*)
 - Begin main loop
 - Inject new particles and move all particles (*maker.f* and *mover.f*)
 - Check for particles crossing boundaries (*strike.f*)
 - Send particles to neighbors (*mover.f* and PVM)
 - Receive particles from neighbors (*mover.f* and PVM)
 - Sort particles into cells (*sorter.f*)
 - Collide particles (*collide.f*)
 - Evaluate pressure near end boundaries (*pressure.f*)
 - Take statistical samples (*gridsmp.f*)
 - End main routine
 - Calculate force on upper wall
 - Normalize statistics
 - Save statistical data
-

The task structure of the main routine, *slider.f*, is to first start PVM and initialize quantities. At this point the program has spawned the desired number of processors. There is one processor, the master, designated strictly for data I/O. The remaining processors are considered workers and they handle the particle simulation for their own domain. Each worker has its own list of particles, but all are processing the same code and performing the same functions.

Before the simulation begins, particles must be initialized with positions and velocities. If the simulation is starting with a fresh list of particles then each processor will initialize the correct number of particles to give uniform temperature and pressure

across the system. The program can also restart from a previous run. The routine *starter.f* uses data files to extract the average number, temperature, and flow velocity for each cell. It uses the number density of each cell to determine how many particles to initialize in the cells. The velocity of each particle in a particular cell can be initialized from the Maxwell-Boltzmann distribution

$$v_x, v_y, v_z = \sqrt{2kT/m} \sqrt{-\ln(1-R_1)} \sin(2\pi R_2) \quad (5-17)$$

where R_1 and R_2 are uniform deviates. The flow velocity for the cell is simply added on to the velocity drawn by (5-17).

If the position and velocity of each particle is recorded, then the system can be restarted exactly in its previous state. Unfortunately, the file sizes needed for this would be enormous since the number of particles is large. The method of restart described above allows the file size to be much less since cell information is instead recorded.

Before the *mover.f* routine moves each particle, new particles are first created. These particles will be fluxed in at the left and right boundaries by the *maker.f* routine. *Maker* is called separately for each end boundary. These boundaries are considered to be fluid reservoirs with a fixed mass density, flow velocity, and temperature. By using equation (2-43) the correct number of particles are created at

the boundary interfaces. Each new particle is assigned a random height and is moved off the boundary at a random fraction of the time step.

Since the reservoir pressures are fixed so are the pressures at the boundaries. The simulation forces the pressure of the particles near the boundaries to remain close to the pressures of the reservoirs. This can be done by adjusting the flow velocity of the reservoirs which, in turn, affects the number of particles fluxed into the system by the *maker.f* routine. If the pressure at a region near the boundary is found to have a pressure greater than the reservoir, then the reservoir flow velocity should be decreased. Flow velocity will be increased if the pressure is found to be too low in the boundary region. The main routine, *slider.f*, uses a simple linear relation to adjust the flow velocity parallel to the platter u_0 according to

$$u_0^{new} = u_0^{old} + \frac{P_0 - P_R}{P_0} \left(\frac{1}{10} U \right) \quad (5-18)$$

where P_0 is the reservoir pressure, P_R is the pressure of the region near the boundary, and U is the platter velocity. This adjustment is done at the end of every time step and after all particles are moved and collided.

The *pressure.f* routine is responsible for evaluating the pressure near the boundaries. Pressure is measured by first sampling the thermal energy. Thermal energy is defined as the total kinetic energy in the center of mass frame,

$$\begin{aligned}
\langle E_{TH} \rangle &= \frac{1}{2} m \sum_{i=1}^N |\mathbf{v}_i - \mathbf{v}|^2 \\
&= \frac{1}{2} m \sum_{k=1}^3 \left\{ \sum_{i=1}^N (v_k)_i^2 - \frac{1}{N} \left(\sum_{i=1}^N (v_k)_i \right)^2 \right\}
\end{aligned} \tag{5-19}$$

where N is the number of particles in the testing region and $\mathbf{v}$ is the center of mass velocity of the region. The temperature is related to the thermal energy as

$$\langle E_{TH} \rangle = \frac{3}{2} NkT \tag{5-20}$$

In order to get the correct instantaneous temperature, the thermal energy must be multiplied by a correction factor,

$$C = \frac{N+1}{N} \tag{5-21}$$

The correction factor is needed because even at thermodynamic equilibrium, the instantaneous center of mass velocity fluctuates and contributes a mean kinetic energy of $\frac{3}{2} kT$. The pressure can then be found from the ideal gas law.

Once all the particles have been moved for the current time step, each processor will share information with its neighbors (see Figure 4.1). The processors have lists of particles that have gone beyond the right and left processor boundaries at the end of the movement process. Using PVM routines, each worker will first send particle lists to its corresponding neighbor processors. Each worker will then receive particle lists from its neighbors and add these new particles to its primary particle list. Since

the first and last processors have only one neighbor they are handled differently. This process is done prior to sorting because all collision cells for a particular processor lie within the processor boundaries and therefor all particles must lie within the processor domain.

The *sorter.f* routine sorts particles into collision cells by truncating particle positions. The *collide.f* routine uses lists created by sorter to implement the DSMC collision algorithm.

Synchronization of processors happens only at the time particle information is transferred among neighbors. At this point all processors send information to their neighbors and then wait to receive messages back from their neighbors. The processors are held up until the receiving process is complete. Once the receive is complete, all workers will start sorting, colliding, and moving without any synchronization. However, synchronization is kept in the sense that each processor and its neighbors are on the same iteration number during the message passing sequence.

After all particles have been moved and collided for the current time step, the *gridsmp.f* routine takes statistical samples of the system. The system is divided up into statistical cells and all measurements are done separately within the cells. Statistical cells are similar to collision cells since they merely group particles by proximity. The size of the statistical cells are not constrained as are collision cells. If

the number of particles on average is small in a particular cell, then the statistical measurements for that cell are not reliable. This effect is seen as spikes on a surface plot of any statistical measurement.

For each cell there are running sums for the number of particles, particle velocity, and squared velocity. When the simulation has been completed and the sums are finally set, the sums must be normalized to get time averaged values. The velocity and squared velocity are normalized by dividing by the number of particles that went into the sum. Doing the averages in this way will eliminate the need for any correction factors that were needed for instantaneous measurements. The cell temperature can be calculated using (5-19) and (5-20). Pressure is found from the ideal gas law as before.

The force on the upper wall can be calculated from the total change in momentum at the wall. Each time a particle strikes the upper wall the change in vertical velocity is recorded in the *striker* routine. A running sum is kept over the entire time of the simulation. The force on the upper wall is given by

$$F_y = m \frac{\Delta v_y}{\Delta t} \quad (5-22)$$

At the end of the simulation, data files are created that contain the temperature, pressure, and number density distribution of the entire system. Since each processor has its own slice of the system, its data is unique and must be handled separately. The

processor dedicated to I/O, the master, will take the data from each worker processor and consolidate it. The resulting output files will contain data for the entire system.

Chapter 6

SIMULATION RESULTS

6.1 Simulation Setup

A number of different slider bearing configurations were studied using the DSMC method. The geometry of the slider system included a fixed length $L = 5 \mu\text{m}$. The ratio of the entrance to exit height, $h(x=0)/h_0$ was fixed at two to one for all configurations. Three different exit heights were tested: $h_0 = 10 \text{ nm}$, $h_0 = 25 \text{ nm}$, and $h_0 = 50 \text{ nm}$. Simulation particles consisted of hard spheres with a diameter of $4.19 \text{ \AA}$, a mass of $4.81 \times 10^{-26} \text{ kg}$, and a mass density of 1.29 kg/m^3 . The particle properties are based on average values for the constituents of air. The ambient conditions are standard with pressure $p_0 = 1 \text{ ATM}$ and temperature $T_0 = 273 \text{ K}$. The mean-free path of air at ambient conditions is 47.8 nm and the viscosity of the gas is $\mu = 1.35 \times 10^{-5} \text{ N s/m}^2$. The inlet and outlet pressures are maintained at ambient pressure.

For each geometry variation, four different cases for boundary scattering from the platter were simulated. The Cercignani–Lampis scattering routine incorporated surface accommodation parameters that were derived from experimental scattering data. The simulations include parameters for three different platter materials: glass, an IBM disk platter, and a surface lubricated with z-dol. In addition, a fully

accommodating platter is used in the simulations. The upper boundary, or head, is fully accommodating in each case. The platter speed is set at $U=25$ m/s for each run.

Each simulation initializes with 16,000 particles and uniform ambient conditions or can be restarted from a previous simulation. Steady state flow is reached typically after 0.5 to 1.0 million steps and the number of simulation particles in the system increases and stabilizes at roughly 50,000. All tests were performed on the Cray T3D using 32 processors ($\approx 10^5$ time steps per hour). Steady state is verified by computing variation between the pressure profiles of two consecutive runs. The simulation is complete when the variation is within the expected statistical variation of 0.8 % (see the next section).

In order to compare the DSMC results to theory, the Reynolds equation is solved numerically using finite difference approximations. The resulting set of nonlinear equations are solved iteratively using Newton's method. All Reynolds calculations presented here assume fully accommodation surfaces. Thus for the Fukui-Kaneko generalized lubrication equation, the Poiseuille flow rate is given by (3-42). For each slider configuration, the DSMC results are compared with the continuum Reynolds equation, the first-order slip corrected Reynolds equation, and the Fukui and Kaneko equation.

6.2 Validation of the Program Output

In order to verify that the simulation is working properly and to validate its output, it is first necessary to test the program under conditions where the results are known a priori. The simplest set of conditions is thermodynamic equilibrium with no gas flow through the channel. In this case the lower platter velocity is set to zero. The left and right reservoirs are set to a pressure of 1 atm and to a temperature of 273 K. The upper and lower walls are set to the same reservoir temperature. When the gas is in equilibrium with the reservoirs, temperature and pressure will be uniform down the channel. Using the same equilibrium conditions, the scattering routine can be tested. The implemented scattering model should satisfy the conditions of reciprocity. The equilibrium conditions provide a simple but effective way to test the program.

6.2.1 Verification at Equilibrium with No Flow

Figure 6.1 and Figure 6.2 are one-dimensional plots of pressure and temperature versus position down the channel. These plots are created by taking the average of statistical cells across the height of the channel. Thus, the 1-D plots represent vertical slices of the system along the length of the channel.

Figure 6.1 Plot shows 1-Dimensional pressure along the channel length. Pressure is uniform with deviation about 1 atm as expected.

Figure 6.2 Plot shows 1-Dimensional temperature down the length of the channel. Temperature is uniform with variation about 273 K as expected.

The program has been tested using the prescribed equilibrium conditions. Figure 6.1 shows that the system has come to equilibrium at 1 atm as expected. In addition, Figure 6.2 shows that the system is in thermal equilibrium with the walls at 273 K. It must be verified, though, that the deviations in these measurements are as expected. Specifically, the goal is to find the expected deviations in pressure measurement and verify that the actual deviations of Figure 6.1 are of the same order of magnitude.

The expected deviations in pressure measurements are given by

$$\Delta P^2 = \frac{\langle \delta P^2 \rangle}{M} \quad (6-1)$$

where $\langle \delta P^2 \rangle$ is the thermal fluctuation in pressure and M is the number of independent samples. Using $PV^\gamma = C$ and $PV = NkT$ the thermal fluctuation in pressure is given by [12]

$$\begin{aligned} \langle \delta P^2 \rangle &= -kT \left(\frac{\partial P}{\partial V} \right)_s \\ &= \gamma \frac{PkT}{V} \\ &= \frac{5}{3} \frac{P^2}{N} \end{aligned} \quad (6-2)$$

where $\gamma = \frac{5}{3}$ for an ideal gas and N is the number of particles in the sample. Using the last two equations, the standard deviation in measured pressure is given by

$$\sigma_P = \sqrt{\frac{\Delta P^2}{P^2}} = \sqrt{\frac{5}{3} \frac{1}{NM}} \quad (6-3)$$

The simulation program takes a sample at every time step. These measurements are not all independent since the system may change very little between consecutive steps. It is necessary to estimate the fraction of the number of independent samples to the total samples. Specifically, the number of independent samples in a vertical slice is needed since the results are given in vertical slices.

Figure 6.3 Plot of consecutive instantaneous pressure measurements in a vertical slice of the system. The measurements fluctuate about the mean. Only 100 samples are shown here.

The fraction of independent samples can be estimated by measuring the instantaneous pressure at every time step in a single vertical slice. As consecutive

pressure measurements are taken the values will fluctuate about a mean μ , as shown by Figure 6.3, and will have a standard deviation σ . Two measurements that are located on opposite sides of the mean value should be independent of each other. Specifically, independent measurements are estimated by walking along the plot from one end and counting the number of times that the data series crosses from $\mu + \sigma$ to $\mu - \sigma$.

After measuring the instantaneous pressure for 10,000 samples, the fraction of independent samples was estimated to be 2.3% or $M \approx 230$. Using the last result and $N = 170$ (number of particles in a slice) in (6-3), the expected deviation in pressure is $\sigma_p \approx 5 \times 10^{-3}$ atm. The standard deviation of the data in Figure 6.1 was found to be $\sigma_{data} \approx 8 \times 10^{-3}$ atm. The deviation in the pressure data is the same order of magnitude as the expected deviation.

6.2.2 Verification of Scattering Routine

It is necessary to show that the scattering routine satisfies detail balance. Simply stated, at equilibrium the velocity distribution of the gas should be conserved throughout the scattering process. This can be tested by measuring the velocity distribution of a gas before scattering and comparing it to the post-scattering velocity distribution. The two distributions should be identical if detail balance is satisfied.

The C-L model consists of a normal velocity distribution and a tangential velocity distribution. The velocity of each particle should be resolved into a normal and

tangential component both before and after scattering. Velocity distributions for the normal and tangential velocities for both incident and reflected particles are created by keeping track of the scattering process for each particle. Hence, there is a total of four distributions that will be measured. It should be verified that both the normal and tangential velocity distributions are conserved by the scattering routine.

The simulation was run using the equilibrium condition with no flow. The scattering of 10,000 particles was recorded. Using velocity histograms for the recorded particles, probability distributions were created for the incident and reflected particles.

Figure 6.4 The plot shows the velocity distribution for the normal component of velocity. Incident and reflected velocity distributions are in good agreement. ('o'=incident distribution, 'x'=scattered distribution)

Figure 6.5 Incident and reflected velocity distributions for the parallel component of velocity are in good agreement.

Figure 6.4 and Figure 6.5 show the results of the 10,000 recorded scattering particles. Both plots show that the velocity distributions are conserved for both components of velocity. Thus it has been shown, in a simple manner, that the scattering routine satisfies detail balance.

6.3 Simulation Results for Moving Platter

The simulation program has been verified at equilibrium with the platter speed $U = 0$ m/s in the previous section. In order to develop pressure in the channel, the platter speed is set to a value relevant to disk drives, $U = 25$ m/s. The same speed is used for all slider configurations. A system length of $L = 5 \mu\text{m}$ was used in all simulations.

For each configuration, two sets of DSMC data are presented. The scatter of the data sets qualitatively indicates the statistical uncertainty in the DSMC program results. The fluctuations in the data sets also indicate the state of the system. If the fluctuations are large, then the flow has not yet reached steady-state. Otherwise if the fluctuations are below the expected levels (about 1 %), then the system has most likely reached steady-state flow.

6.3.1 Full Accommodation Thermal Boundary

The following results were produced from the simulation of a fully accommodating platter. In this case, the Cercignani–Lampis scattering routine is disabled. Instead, all particles interacting with the lower platter are scattered thermally.

Figure 6.6 Slider bearing pressure profile for $Kn_0=0.956$, $\Lambda=40.05$, $h_0=50$ nm, and a fully accommodating platter. Data series include continuum solution, first-order slip correction, Fukui-Kaneko solution, and DSMC data.

The first configuration consists of an outlet height $h_0=50$ nm. The Knudsen number at the exit is $Kn_0 = \lambda/h_0 = 0.956$ and the bearing number is $\Lambda = 40.05$.

Figure 6.6 shows the results for this configuration with a fully accommodating platter.

The solutions of the various lubrication equations are plotted against the DSMC simulation results. Both the continuum and slip-corrected Reynolds equations compare poorly with the DSMC data. The Fukui-Kaneko solution and the DSMC data show good agreement.

Figure 6.7 Slider bearing pressure profile for $Kn_0=1.912$, $\Lambda=160.2$, $h_0=25\text{ nm}$, and a fully accommodating platter. Data series include continuum solution, first-order slip correction, Fukui-Kaneko solution, and DSMC data.

Figure 6.7 shows the results for the second configuration with a fully accommodating platter. The second configuration consists of an outlet height $h_0=25\text{ nm}$. The Knudsen number at the exit is $Kn_0 = 1.912$ and the bearing number is $\Lambda = 160.2$. Again, both the continuum and slip-corrected Reynolds equations compare poorly with the DSMC data while the Fukui-Kaneko solution and DSMC show good agreement.

Figure 6.8 Slider bearing pressure profile for $Kn_0=4.780$, $\Lambda=1001$, $h_0=10\text{ nm}$, and a fully accommodating platter. Data series include continuum solution, first-order slip correction, Fukui-Kaneko solution, and DSMC data.

The third and final slider bearing configuration consists of an outlet height $h_0=10\text{ nm}$. The Knudsen number at the exit is $Kn_0 = 4.780$ and the bearing number is $\Lambda = 1001$. In this case, the continuum and slip-corrected Reynolds solutions are in better agreement with the DSMC data than in the two previous cases. The Fukui-Kaneko solution and the DSMC data show good agreement once again.

The three lubrication equations show better agreement in this third case due to the high bearing number. The lubrication equations are all similar in form,

$$\frac{d}{dX} \left[(Q) PH^3 \frac{dP}{dX} \right] = \Lambda \frac{d}{dX} (PH) \quad (6-4)$$

where the expression for Q is different for the continuum Reynolds equation, the slip-corrected Reynolds equation, or the Fukui–Kaneko lubrication equation. In the limit that $\Lambda \rightarrow \infty$, the right-hand-side of (6-4) will dominate and wash out the effect of Q for the various lubrication equations.

6.3.2 Glass Platter

The next set of results were produced from the simulation using a glass platter. In this case, the Cercignani–Lampis scattering routine is enabled and set to use the accommodation parameters for glass. The same three slider configurations of the previous section were studied.

The results of the DSMC simulations for glass are similar to the results of the previous section (see Figure 6.9, Figure 6.10, and Figure 6.11). For all three cases, the Fukui–Kaneko solution was in good agreement with the DSMC data. The continuum and slip-corrected Reynolds equations were in poor agreement at the higher channel spacing and in fair agreement at the lowest channel spacing.

Figure 6.9 Slider bearing pressure profile for $h_0=50\text{ nm}$ using the Cercignani–Lampis scattering routine and accommodation parameters for glass.

Figure 6.10 Slider bearing pressure profile for $h_0=25\text{ nm}$ using the Cercignani–Lampis scattering routine and accommodation parameters for glass.

Figure 6.11 Slider bearing pressure profile for $h_0=10$ nm using the Cercignani–Lampis scattering routine and accommodation parameters for glass.

6.3.3 IBM Disk Platter

The next set of results were produced from the simulation with the Cercignani–Lampis scattering routine set to use the accommodation parameters for an IBM disk platter. The same three slider configurations of the previous sections were studied. The results of the DSMC simulations for the IBM disk platter are similar to the results of the previous sections (see Figure 6.12, Figure 6.13, and Figure 6.14). For all three cases, the Fukui–Kaneko solution was in good agreement with the DSMC data. The continuum and slip-corrected Reynolds equations were in poor agreement at the higher channel spacing and in fair agreement at the lowest channel spacing.

Figure 6.12 Slider bearing pressure profile for $h_0=50\text{ nm}$ using the Cercignani–Lampis scattering routine and accommodation parameters for an IBM disk platter.

Figure 6.13 Slider bearing pressure profile for $h_0=25\text{ nm}$ using the Cercignani-Lampis scattering routine and accommodation parameters for an IBM disk platter.

Figure 6.14 Slider bearing pressure profile for $h_0=10\text{ nm}$ using the Cercignani–Lampis scattering routine and accommodation parameters for an IBM disk platter.

6.3.4 Pt(111) Surface Lubricated with z-dol

The final set of results were produced from the simulation with the Cercignani–Lampis scattering routine set to use the accommodation parameters for a substrate lubricated with z-dol. Three slider configurations were tested giving results similar to the results of the previous sections (see Figure 6.15, Figure 6.16, and Figure 6.17).

Figure 6.15 Slider bearing pressure profile for $h_0=50\text{ nm}$ using the Cercignani-Lampis scattering routine and accommodation parameters for a surface lubricated with z-dol.

Figure 6.16 Slider bearing pressure profile for $h_0=25\text{ nm}$ using the Cercignani-Lampis scattering routine and accommodation parameters for a surface lubricated with z-dol.

Figure 6.17 Slider bearing pressure profile for $h_0=10\text{ nm}$ using the Cercignani–Lampis scattering routine and accommodation parameters for a surface lubricated with z-dol.

*Chapter 7***SUMMARY AND CONCLUSIONS**

The well-founded DSMC method has been adapted to solve the slider bearing problem. DSMC is a numerical simulation method that models gas dynamics at the particle level. In addition, realistic boundary conditions have been incorporated into the simulation by using actual surface scattering data to model particle-surface interactions. To date, most DSMC simulations of rarefield gas flows have assumed diffuse thermal reflection or have adopted the Maxwell accommodation method for reflection of gas particles from a solid surface. Molecular beam studies have shown that the simple scattering models are not realistic.

Only those gas-surface scattering models that satisfy the reciprocity condition can be used in DSMC simulations. The Cercignani–Lampis scattering model is one such model that has been adapted into the DSMC simulation. The C-L model assumes that the reflection process for a molecule's velocity components parallel and normal to the surface are independent. The C-L scattering routine uses parallel and normal accommodation parameters for materials relevant to disk drives. The parameters were extracted from surface study data produced by C. Rettner.

The focus of this research is to test the validity of the Reynolds-type lubrication equations at extremely small flying heights against a DSMC simulation. The fundamental assumption of the continuum Reynolds equation is that the mean free path of a molecule in the channel is small compared to the channel spacing. The continuum equation is valid for systems where the Knudsen number is $Kn \ll 1$. First-order slip correction is an adjustment of the Reynolds equation to account for interactions at the particle level. The correction was derived from the assumption that particles scatter according to a simple model introduced by Maxwell. In addition, the correction assumes that particles approaching the lower wall had their last collision in the fluid rather than with the upper wall. First-order slip is limited to systems where $Kn < 0.1$. The slider configurations simulated in this study had $Kn \approx 1$ thus violating the core assumptions. Both continuum and first-order slip Reynolds equations showed poor or fair agreement with DSMC data.

In the case that the Reynolds equations showed fair agreement with Fukui–Kaneko and DSMC, the bearing number was of the order $\Lambda = 10^3$. The various lubrication equations tend to converge for large bearing numbers. The solutions all share a similar form. The various differences in the solutions are washed out in the limit of large bearing numbers.

The Fukui–Kaneko generalized lubrication equation compared very well with DSMC for all simulated Knudsen numbers. In addition, agreement existed for the

various platter boundary conditions: a fully accommodating platter, a glass platter, an IBM disk platter, and a surface lubricated with z-dol. The agreement indicates that the velocity distributions are relatively unchanged by the use of realistic boundary conditions. In conclusion, for the purpose of disk slider design, the materials may be assumed to act as fully accommodating surfaces.

Bibliography

- [1] Alexander, F. J., A. L. Garcia, and B. J. Alder (1994), *Phys. Fluids*, Vol. 6, 3854
- [2] Allen, B. J. and D. J. Tildesley (1987), "Computer Simulation of Liquids," Clarendon, Oxford..
- [3] Bird, G. A. (1994), "Molecular Gas Dynamics and Direct Simulation of Gas Flows," Clarendon, Oxford.
- [4] Cercignani, C. and M. Lampis (1971), *Trans. Th. Stat. Phys.*, Vol. 1, 101.
- [5] Deckert, K. L. (1990), *IBM J. Res. Develop.*, Vol. 35, 660.
- [6] Feller, W. (1957) "An Introduction to Probability Theory and its Applications," Wiley, New York.
- [7] Fukui, S. and R. Kaneko (1990), "Analysis of ultrathin gas film lubrication based on linearized Boltzmann equation: First report derivation of a generalized lubrication equation including thermal creep flow," *J. Tribology*, 112, 78
- [8] Garcia, A. L. (1994), "Numerical Methods for Physics," Prentice Hall, Inc.
- [9] Geist, A., A. Beguelin (1994), "PVM: Parallel Virtual Machine, A Users' Guide and Tutorial for Network Parallel Computing", The MIT Press, Cambridge
- [10] Gross, W. A., L. A. Matsch, V. Castelli, A. Eshel, J. H. Vohr and M. Wildmann (1980), "Fluid Film Lubrication," Wiley, New York.
- [11] Kittel C. (1980), "Thermal Physics," W. H. Freeman and Company
- [12] Landau, L. D. and E. M. Lifshitz (1980), "Statistical Mechanics," Pergamon, Oxford.
- [13] Landau, L. D. and E. M. Lifshitz (1959), "Fluid Mechanics," Pergamon, Oxford.
- [14] Lord, R. G. (1992), *J. Fluid Mech.*, Vol. 239, 449.
- [15] Press, W., B. Flannery, S. Teukolsky, and W. Vetterling (1992), "Numerical Recipes in FORTRAN", 2nd ed., Cambridge University Press, Cambridge
- [16] Rettner, C. T. (1997) to appear in *J. Tribology*.
- [17] Rettner, C. T. (personal communication)

- [18] Vincenti, W. G. and C. H. Kruger (1965), "Introduction to physical Gas Dynamics," Krieger, Malabar.

*Appendix A***PVM EXAMPLE****A.1 PSDOT Description**

The example program PSDOT performs a scalar product of two vectors, *X* and *Y*, using a parallel processing technique. All program listings for this example are presented with line numbers in the following sections. The vectors are read in from data files by the master processor and then distributed to the worker processors. Each processor, including the master, performs a scalar product on its part of the vectors. The results are then put together by the master to complete the desired task.

PVM is initialized in lines 34 - 36. The routine PVMFGSIZE returns the number of members in the current group. The environment variable, *pvmall*, refers to all processes that were requested at the UNIX command line. The integer, *nodes*, contains the number of total active processes. Each processor has a unique task identification number. This number can be obtained and converted into a processor identification number. When each processor calls PVMFGETPE, its processor identification number is stored in the variable *mypid*. If there are *N* processors active then *mypid* will range from 0 to *N*-1 and this number is unique for each processor.

It has been chosen to designate processor 0 as the master processor in control of data I/O. The IF statement at line 40 allows the code to be broken into two parts;

processor 0 will execute instructions from lines 41 to 151 while all other worker processors will evaluate lines 153 to 203. These two blocks form the master-slave structure of the code.

The master processor reads in the vectors from a data file, *vector.dat*, in lines 42 through 53. The first line of *vector.dat* contains N , the number of elements in each vector. Each subsequent line in the data file contains one element of each vector separated by a tab. Lines 71 to 73 extract parts of each vector that are to be sent to the worker processors.

The message passing sequence begins on line 76 where the send buffer is cleared by `PVMFINITSEND`. The next step in the sequence is to pack each data set one by one. There are five arguments used by the routine `PVMFPACK`. The first specifies the data type of the items to be packed. The second argument points to a memory location of the beginning of the data block that is being sent. This could be any element of an array. The third argument specifies the number of items being packed. The stride of the packing process is specified by the fourth argument. A stride value of one means that elements of an array are not skipped as the data is packed. Finally, the fifth argument returns a status value which may be used as an error flag. Each PVM message passing routine has a status code that is located at the end of the argument list.

Once all the data is packed into the current buffer, the data is then sent off to the desired processor by line 100. The first argument used by PVMFSEND contains the destination processor identification number. This tells the processor where to send the data. The second argument is a message tag that identifies the data being sent. Any receiving process that is expecting this particular data will have a matching message tag.

The block of code containing lines 156 through 176 is where the vector parts are received by each worker. The sequence begins with a call to PVMFRECV. Its first argument specifies the processor identification number of the processor that sent the data. The second argument is a message tag. Both of these arguments should match the first two arguments of the corresponding sending process. This means that the arguments of line 156 should match the arguments of line 100 since these lines involve sending and receiving the same data.

The next step in receiving the data is to unpack it. For every pack statement in a sending sequence there should be an unpack statement in a receiving sequence. Data is packed and unpacked in the same order. The argument list for the PVMFUNPACK routine is the same for the PVMFPACK routine.

Now that each processor has its own set of vectors, the master and workers will compute their own part of the scalar product in lines 115 and 184. The function SDOT accepts two column vectors, X and Y, and returns their scalar product.

Source code for the function SDOT has also been provided. In lines 186 to 198 each worker sends its local dot product to the master. These products are received by the master and added together in lines 120 to 134 producing the final result for the scalar product of the original vectors.

A.2 PSDOT Source Code

```

1      PROGRAM PSDOT
2      *
3      * PSDOT performs a parallel inner (or dot) product, where the
4      * vectors X and Y start out on a master node, which then sets
5      * up the virtual machine, farms out the data and work, and sums
6      * up the local pieces to get a global inner product.
7      *
8      *      .. External Subroutines ..
9      EXTERNAL PVMFMYTID, PVMFPARENT, PVMFSPAWN,
10     PVMFEXIT, PVMFINITSEND
11     EXTERNAL PVMFPACK, PVMFSEND, PVMFRECV, PVMFUNPACK
12     *
13     *      .. External Functions ..
14     REAL*8 SDOT
15     *
16     *      .. Intrinsic Functions ..
17     INTRINSIC MOD
18     *
19     *      .. Parameters ..
20     INTEGER*8 MAXN
21     PARAMETER ( MAXN = 8000 )
22     INCLUDE '/usr/include/mpp/fpvm3.h'
23     *
24     *      .. Scalars ..
25     INTEGER*8 N, LN, MYTID, IBUF, IERR, nodes, mypid
26     INTEGER*8 I, J, K, L
27     REAL*8 LDOT, GDOT
28     *
29     *      .. Arrays ..
30     REAL*8 X( MAXN ), Y( MAXN ), Xsend( MAXN ), Ysend( MAXN )
31     *
32     * Enroll in PVM and get my and get all process task ID numbers
33     *
34     CALL PVMFGSIZE( pvmall, nodes )      ! get # of processors
35     CALL PVMFMYTID( mytid )              ! get each nodes task ID
36     CALL PVMFGETPE( mytid, mypid )      ! convert to processor ID
37     *
38     * If I am node 0, read data file and farm out work to procs.
39     *
40     IF ( mypid .eq. 0 ) THEN
41     *
42         open( unit=11, file='vector.dat' )

```

```

43      read ( 11,* ) N
44      IF ( N .GT. MAXN ) THEN
45          WRITE(*,*) 'N too large.'//
46          $      ' Increase parameter MAXN to run this case.'
47          STOP
48      END IF
49      *
50      do i = 1,N
51          read(11,*) x(i),y(i)
52      end do
53      close(unit=11)
54      write(*,*) 'Data has been Read Successfully.'
55      write(*,*) 'Number of elements: ',N
56      write(*,*) 'Number of nodes: ',nodes
57      *
58      *
59      *      LN is the number of elements of the dot product to do
60      *      locally. Everyone has the same number, with the master
61      *      getting any left over elements. J stores the number of
62      *      elements rest of procs do.
63      *
64      J = N / nodes
65      LN = J + MOD(N, nodes)
66      *
67      *
68      *      Loop over all worker processes, send out vector parts
69      *
70      do K = 1, nodes-1
71          do L = 1, J
72              Xsend(L) = X((K-1)*J + L)
73              Ysend(L) = Y((K-1)*J + L)
74          end do
75          *
76          CALL PVMFINITSEND( PVMDEFAULT, IBUF )
77          *
78          if (IBUF .LT. 0) then
79              write(*,*) 'Error in PVMFINITSEND'
80          end if
81          *
82          CALL PVMFPACK( INTEGER8, J, 1, 1, IERR )
83          *
84          if ( IERR .LT. 0 ) Then
85              write(*,*) 'Error in PVMFPACK for J'
86          end if
87          *
88          CALL PVMFPACK( REAL8, Xsend, J, 1, IERR )
89          *
90          if ( IERR .LT. 0 ) Then
91              write(*,*) 'Error PVMFPACK at Xsend'
92          end if
93          *
94          CALL PVMFPACK( REAL8, Ysend, J, 1, IERR )
95          *
96          if ( IERR .LT. 0 ) Then
97              write(*,*) 'Error PVMFPACK at Ysend'
98          end if

```



```

99      *
100      CALL PVMFSEND( K, 0, IERR )
101      *
102      if ( IERR .LT. 0 ) then
103          write(*,*) 'Error in send to Node:',K
104      end if
105      end do
106      *
107      *
108      *   Figure node 0's part of remaining dot product
109      *
110      do L = 1, LN
111          Xsend(L) = X(J*(nodes-1) + L)
112          Ysend(L) = Y(J*(nodes-1) + L)
113      end do
114      *
115      GDOT = SDOT( LN, Xsend, Ysend)
116      *
117      *   Receive the local dot products from workers, and
118      *   add to get the global dot product
119      *
120      DO 20 K = 1, nodes-1
121          CALL PVMFRECV( K, 1, IBUF )
122      *
123          if ( IBUF .LT. 0 ) Then
124              write(*,*) 'RECV error from node: ',K
125          end if
126      *
127          CALL PVMFUNPACK( REAL8, LDOT, 1, 1, IERR )
128      *
129          if ( IERR .LT. 0 ) Then
130              write(*,*) 'Unpack Err node: ',K,IERR
131          end if
132      *
133          GDOT = GDOT + LDOT
134      20  CONTINUE
135      *
136      *   Print out result
137      *
138      WRITE(*,*) ' '
139      WRITE(*,*) '<x,y> = ',GDOT
140      *
141      *   Do sequential dot product and subtract from
142      *   distributed dot product to get desired error estimate
143      *
144      LDOT = SDOT( N, X, Y, 1 )
145      WRITE(*,*) '<x,y> : sequential dot product. '//
146      $      '<x,y>^ : distributed dot product.'
147      WRITE(*,*) '| <x,y> - <x,y>^ | = ',ABS(GDOT - LDOT)
148      WRITE(*,*) 'Run completed.'
149      *
150      *   If I am a worker process (i.e. node other than zero)
151      *
152      ELSE
153      *
154      *   Receive startup info

```

```

155      *
156      CALL PVMFRECV( 0, 0, IBUF )
157      *
158      if ( IBUF .LT. 0 ) Then
159          write(*,*) 'RECV error at node: ',mypid,IBUF
160      end if
161      *
162      CALL PVMFUNPACK( INTEGER8, LN, 1, 1, IERR )
163      *
164      if ( IERR .LT. 0 ) Then
165          write(*,*) 'Unpack err for LN at node: ',mypid
166      else
167          write(*,*) 'LN Sent LN = ',LN,mypid
168      end if
169      *
170      CALL PVMFUNPACK( REAL8, X, LN, 1, IERR )
171      *
172      if ( IERR .LT. 0 ) Then
173          write(*,*) 'Unpack err for X at node: ',mypid
174      end if
175      *
176      CALL PVMFUNPACK( REAL8, Y, LN, 1, IERR )
177      *
178      if ( IERR .LT. 0 ) Then
179          write(*,*) 'Unpack err for Y at node: ',mypid
180      end if
181      *
182      Figure local dot product and send it in to master
183      *
184      LDOT = SDOT( LN, X, Y )
185      *
186      CALL PVMFINITSEND( PVMDEFAULT, IBUF )
187      *
188      if ( IBUF .LT. 0 ) then
189          write (*,*) 'Unable to INITSEND in node',mypid
190      end if
191      *
192      CALL PVMFPACK( REAL8, LDOT, 1, 1, IERR )
193      *
194      if ( IERR .LT. 0 ) then
195          write (*,*) 'Pack err LDOT at node ',mypid
196      end if
197      *
198      CALL PVMFSEND( 0, 1, IERR )
199      *
200      if ( IERR .LT. 0 ) then
201          write(*,*) 'Unable to send at node: ',mypid
202      end if
203      *
204      END IF
205      *
206      CALL PVMFEXIT( 0 )
207      *
208      STOP
209      *
210      End program PSDOT

```

```
211      *
212      END
213      *
214      include 'SDOT.F'
```

A.3 SDOT Source Code

```
1      REAL*8 FUNCTION SDOT(Elements,vector1,vector2)
2      * This routine returns the dot product of vector1 and vector2
3      * Elements sets the number of elements for the two vectors
4      INTEGER*8 Elements, i
5      REAL*8 vector1(8000), vector2(8000)
6      SDOT = 0.
7      DO i = 1,Elements
8          SDOT = SDOT + vector1(i)*vector2(i)
9      END DO
10     RETURN
11     END
```

Appendix B

SIMULATION SOURCE CODE

```

program PPROTO6
!
! Program to simulate a dilute gas using DSMC algorithm in 2 dimensions
! This version measures the viscosity in a dilute gas with upper and
! lower walls moving with opposite velocities.
! The lower wall is flat and the upper wall is angled.
! It calculates statistics in a different set of grid cells from the
! simulation cells
! It also destroys particles that leave the system
! It creates particles fluxing in at the left and right faces based on
! Poisson statistics
! particles: system size is not constant
! It is capable of continuing a previous run
! It is in double-precision
! This program adjusts the the flux of particles so that the pressure
! at the ends matches the reservoir pressure
! This program uses PVM
!
! *** PARAMETERS ***
!
!       include 'param.txt'
!
! *** INCLUDES ***
!
!       include '/usr/include/pvm3/fpvm3.h'
!
! *** EXTERNAL ***
!
!       external PVMFMYTID, PVMFINITSEND, PVMFPACK
!       external PVMFRECV, PVMFSEND, PVMFUNPACK, PVMFGSIZE, PVMFGETPE
!
! *** REAL ARRAYS ***
!
!       real*8 x(npmax), v(npmax,3), y(npmax), vrmax(ncmax,ncmax)
!       real*8 xl_out(nptmax), yl_out(nptmax)
!       real*8 xr_out(nptmax), yr_out(nptmax)
!       real*8 vl1_out(nptmax), v2l_out(nptmax), v3l_out(nptmax)
!       real*8 vlr_out(nptmax), v2r_out(nptmax), v3r_out(nptmax)
!       real*8 selxtra(ncmax,ncmax), ave_u2(ncmax,ncmax)
!       real*8 ave_n(ncmax,ncmax), ave_u(ncmax,ncmax,3), FluxOut(5)
!       real*8 coeff(ncmax,ncmax), statcell_vol(ncmax,ncmax)
!       real*8 ave_T(ncmax,ncmax), xcell(ncmax), ycell(ncmax)
!       real*8 delv(2), cell_vol(ncmax,ncmax)
!       real*8 dvtot(2), dverr(2), force(2), ferr(2)
!       real*8 ywall(2), vwall(2)
!       real*8 pleft(200), pright(200), ave_pres(ncmax,ncmax)
!
! *** INTEGER ARRAYS ***
!
!       integer*8 Xref(npmax), index(ncmax,ncmax), cell_n(ncmax,ncmax)
!       integer*8 strikes(2), step(200)
!

```

```

! *** REALS ***
!
      real*8 mass, Ly1, Lx1, Ly2, Lx2, mpv, mfp, Lz, rand
      real*8 Lx1sys, Ly1sys, Lx2sys, Ly2sys, Lzsys
!
! *** INTEGERS ***
!
      integer*8 nxccll, nyccll, icell, jcell, xgrid, ygrid
      integer*8 nxcclsys, nycclsys, xgridsys, ygridsys
      integer*8 nsamp, npart, col, iseed, workers, restart
      integer*8 crosstop, cross, cross1, cross2
      integer*8 xgridsys_old, ygridsys_old
!
! *** CHARACTER ***
!
      character*12 readfile
!
! *** COMMON ***
!
      include 'common.txt'
      common /proc_nums/ mypid, nprocs
      common /pass/ xl_out, yl_out, xr_out, yr_out,
&                vl1_out, v2l_out, v3l_out,
&                vlr_out, v2r_out, v3r_out,
&                nl_out, nr_out
      common /block12/ istep      ! use only if particles escape
!
!
*****
!
      pi = 4.*atan(1.)      ! = 3.14159...
!
      ENROLL IN PVM AND GET ID'S
!
      CALL PVMFGSIZE( pvmall, nprocs)      ! Get total # of proc.
      CALL PVMFMYTID( mytid )              ! Get each proc task id
      CALL PVMFGETPE( mytid, mypid )       ! Convert to proc id
!
      PROCESSOR 0 INITIALIZATION
!
      if ( mypid .eq. 0 ) then
        readfile = 'testset.dat'
        open( unit=11, file=readfile )
          read(11,*) restart                ! # 1 = yes 0 = no
          read(11,*) npartsys              ! # of total particles
          read(11,*) nstep                 ! # of time steps
          read(11,*) nxcclsys, nycclsys    ! particle cell structure
          read(11,*) xgridsys, ygridsys    ! statistical cell structure
          read(11,*) vwall_m               ! speed of the lower platter
          read(11,*) Lx2sys, Ly2sys        ! x-dim length, height at exit
          read(11,*) xgridsys_old, ygridsys_old ! Values of old run
        close( unit=11 )
        workers = nprocs - 1                ! number of workers
        if(mod(nxcclsys,workers).ne.0 .or.
&        mod(xgridsys,workers).ne.0) then
          write(*,*) 'ERROR: # of xccll / (nprocs-1) must = integer'
          stop
        end if
        write(*,*) 'Particles/Cell = ', npartsys/nxcclsys/nycclsys
        write(*,*) 'Particles/Stat_Cell = ',
&        npartsys/xgridsys/ygridsys
        Lx1sys = 0.

```

```

Ly1sys = 2.*Ly2sys          ! Entrance height = 2*exit height
Lzsys = Lx2sys              ! Arbitrary z depth
!
  CALL PVMFINITSEND( pvmdefault, ibuf)
  CALL PVMFPACK(INTEGER8, restart, 1, 1, ierr)
  CALL PVMFPACK(INTEGER8, npartsys, 1, 1, ierr)
  CALL PVMFPACK(INTEGER8, nstep, 1, 1, ierr)
  CALL PVMFPACK(INTEGER8, nxcellsys, 1, 1, ierr)
  CALL PVMFPACK(INTEGER8, nycellsys, 1, 1, ierr)
  CALL PVMFPACK(INTEGER8, xgridsys, 1, 1, ierr)
  CALL PVMFPACK(INTEGER8, ygridsys, 1, 1, ierr)
  CALL PVMFPACK-REAL8, vwall_m, 1, 1, ierr)
  CALL PVMFPACK-REAL8, Lx1sys, 1, 1, ierr)
  CALL PVMFPACK-REAL8, Ly1sys, 1, 1, ierr)
  CALL PVMFPACK-REAL8, Lx2sys, 1, 1, ierr)
  CALL PVMFPACK-REAL8, Ly2sys, 1, 1, ierr)
  CALL PVMFPACK-REAL8, Lzsys, 1, 1, ierr)
  CALL PVMFPACK(INTEGER8, Xgridsys_old, 1, 1, ierr)
  CALL PVMFPACK(INTEGER8, Ygridsys_old, 1, 1, ierr)
  CALL PVMFBCAST(PVMALL, 0, ierr)      ! Send data to all workers
!
  if (restart.ne.0) then
    call starter(xgridsys, ygridsys, 1.0, 1.0, 1.0,
&      1.0, 1.0, 1.0, 1.0, 1.0, restart)
    end if
!
  WORKERS RECEIVE DATA AND INITIALIZE
!
  else
!
    CALL PVMFRECV(0, 0, ibuf)
    CALL PVMFUNPACK(INTEGER8, restart, 1, 1, ierr)
    CALL PVMFUNPACK(INTEGER8, npartsys, 1, 1, ierr)
    CALL PVMFUNPACK(INTEGER8, nstep, 1, 1, ierr)
    CALL PVMFUNPACK(INTEGER8, nxcellsys, 1, 1, ierr)
    CALL PVMFUNPACK(INTEGER8, nycellsys, 1, 1, ierr)
    CALL PVMFUNPACK(INTEGER8, xgridsys, 1, 1, ierr)
    CALL PVMFUNPACK(INTEGER8, ygridsys, 1, 1, ierr)
    CALL PVMFUNPACK-REAL8, vwall_m, 1, 1, ierr)
    CALL PVMFUNPACK-REAL8, Lx1sys, 1, 1, ierr)
    CALL PVMFUNPACK-REAL8, Ly1sys, 1, 1, ierr)
    CALL PVMFUNPACK-REAL8, Lx2sys, 1, 1, ierr)
    CALL PVMFUNPACK-REAL8, Ly2sys, 1, 1, ierr)
    CALL PVMFUNPACK-REAL8, Lzsys, 1, 1, ierr)
    CALL PVMFUNPACK(INTEGER8, Xgridsys_old, 1, 1, ierr)
    CALL PVMFUNPACK(INTEGER8, Ygridsys_old, 1, 1, ierr)
!
    boltz = 1.3806e-23      ! Boltzmann's constant
    iseed = 1
!
    DATA FOR AIR
!
    mass = 4.81e-26          ! Mass of air
    diam = 4.19e-10          ! Effective diameter of air
    density = 1.29           ! Mass density of air at STP
!
    DATA FOR ARGON
!
    mass = 6.63e-26
    diam = 3.66e-10
    density = 1.78
!
    T = 273.                ! Initial temperature (K)

```

```

P0 = 101.3e3          ! 1 atm
workers = nprocs - 1 ! The number of worker procs.

!
! EACH WORKER WILL DETERMINE ITS BOUNDARIES AND VOLUMES
!
do k = 1,workers
  if (mypid .eq. k) then
    Lx2 = (Lx2sys*k)/workers      ! x-coord of right boundary
    Lx1 = (Lx2sys*(k-1))/workers ! y-coord of left boundary
    call intersct(Lx1sys, Ly1sys, Lx2sys, Ly2sys, Lx1, 0.,
    &      Lx1, Ly1sys, xint, Ly1, cross1)
    call intersct(Lx1sys, Ly1sys, Lx2sys, Ly2sys, Lx2, 0.,
    &      Lx2, Ly1sys, xint, Ly2, cross2)
    if (cross1.ne.1 .or. cross2.ne.1) then
      write(*,*) 'Could not determine processor volume'
      STOP
    end if
  end if
end do
nxcell = nxcellsys/workers
nycell = nycellsys
xgrid = xgridsys/workers
ygrid = ygridsys
Lz = Lzsys
Volumesys = 0.5*(Ly1sys + Ly2sys)*(Lx2sys - Lx1sys)
&      * Lzsys      ! System volume
Volume = 0.5*(Ly1 + Ly2)*(Lx2 - Lx1)*Lz ! Vol of worker's system
call cellvol(Lx1sys, Ly1sys, Lx2sys, Ly2sys,
&      Lx1, Ly1, Lx2, Ly2, Lz, nxcell, nycell, cell_vol)
call cellvol(Lx1sys, Ly1sys, Lx2sys, Ly2sys,
&      Lx1, Ly1, Lx2, Ly2, Lz, xgrid, ygrid, statcell_vol)
!
ywall(1) = 0. ! Height of lower platter
ywall(2) = 0. ! Striking position of upper boundary --> INTERSECT
mpv = sqrt(2.*boltz*T/mass) ! most probable velocity
vwall(1) = vwall_m ! lower wall moves left to right
vwall(2) = 0. ! upper wall does not move
u0 = vwall(1) ! reservoir velocity
u0_left = u0 ! left reservoir velocity
u0_right = u0 ! right reservoir velocity
v0 = 0.
w0 = 0.
eff_num = (density/mass)*Volumesys/npartsys ! # in each sim. particle
mfp = 1./(sqrt(2.)*pi*diam**2*density/mass) ! Mean free path
if (mypid .eq. 1) then
  write (*,*) 'The desired particle number is: ',npartsys
  write (*,*) 'The number of total time steps is: ',nstep
  write (*,*) 'Each simulation particle represents ',
&      eff_num,' atoms'
  write (*,*) 'Cell width is ',Ly2/nycell/mfp,
&      ' mean free paths in y_dir'
  write (*,*) 'Cell width is ',Ly2/nxcell/mfp,
&      ' mean free paths in x_dir'
  write (*,*) 'The Lower wall velocity is ',vwall(1),' m/s'
end if
!
! INITIALIZE PARTICLES IN EACH WORKER
!
if (restart.ne.0) then
  call starter(xgridsys, ygridsys, Lx1, Lx2, Ly1, Ly2,
&      Lx1sys, Ly1sys, Lx2sys, Ly2sys, mass, restart)
  eff_num = eff_num/restart ! reduce by cloning factor
else

```

```

npart = npartsys*Volume/Volumesys + 0.5 ! Weighted by proc. vol.
i = 1
istep = 0
do while (i .le. npart)
  x(i) = Lx1 + (Lx2 - Lx1)*rand(iseed)! Assign random position
  y(i) = Ly1*rand(iseed)
  call intersct(Lx1sys,Ly1sys,Lx2sys,Ly2sys,
&      0.,0.,x(i),y(i),xint,yint,crosstop)
  if (crosstop.ne.1) then ! Particle is accepted
    do j=1,3
      temp = sqrt(-log(1.-rand(iseed))) *
&      cos(2*pi*rand(iseed))
      v(i,j) = mpv*temp
    end do
    v(i,1) = v(i,1) + vwall(1)+(vwall(2)-vwall(1))*y(i)/Ly1
    i = i + 1
  end if
end do
!
!
!
!
INITIALIZE QUANTITIES TO ZERO
!
do i=1,nycell
  do j = 1,nxcell
    vrmax(i,j) = 3*mpv ! Used by collision routine "colide"
    selxtra(i,j) = 0. ! Used by collision routine "colide"
  end do
end do
do j=1,2
  dvtot(j) = 0. ! Total momentum change at wall
  dverr(j) = 0. ! Used to estimate error in dvtot
end do
nsamp = 0 ! Total number of samples
tsamp = 0. ! Sample time
do icell=1,ygrid
  do jcell = 1,xgrid
    ave_n(icell,jcell) = 0. ! Average number of particles
    ave_T(icell,jcell) = 0. ! Average temperature
    do k=1,3
      ave_u(icell,jcell,k) = 0. ! Average velocity in a cell
    end do
    ave_u2(icell,jcell) = 0. ! Average velocity square
  end do
end do
!
!
if (Ly1sys/nycellsys.lt.Lx2sys/nxcellsys) then
  tau = 0.2*(Ly1sys/nycellsys)/mpv ! Set time step tau using y-dim
else
  tau = 0.2*(Lx2sys/nxcellsys)/mpv ! Set using x-dimension
end if
!
do icell=1,Nycell
  do jcell=1,Nxcell
    if (cell_vol(icell,jcell).ne.0.) then
      coeff(icell,jcell) = 0.5*eff_num*pi*diam**2*tau/
&      cell_vol(icell,jcell)
    else
      coeff(icell,jcell) = 0. ! 0 cellvol = no particles
    end if
  end do
end do
!

```



```

!      SET UP QUANTITIES FOR THE "PRESSURE" ROUTINE
!      USE THE FIRST AND LAST WORKER PROCS
!
      if (mypid.eq.1 .or. mypid.eq.workers) then
        pct = 0.1/10          ! Percent of system Length
        xllit = 0.            ! Left testing coords
        xlbig = pct*Lx2sys
        if (xlbig.gt.Lx2) then
          xlbig = Lx2
        end if
        xrlit = Lx2sys*(1.-pct) ! Right testing coords
        if (xrlit.lt.Lx1) then
          xrlit = Lx1
        end if
        xrbig = Lx2sys
        call intersct(Lx1sys, Ly1sys, Lx2sys, Ly2sys, xlbig, 0.,
&                   xlbig, Ly1, xlint, ylint, cross)
        call intersct(Lx1sys, Ly1sys, Lx2sys, Ly2sys, xrlit, 0.,
&                   xrlit, Ly1, xrint, yrint, cross)
        vol_left = 0.5*(Ly1+ylint)*xlbig*Lz
        vol_right = 0.5*(Ly2+yrint)*(xrbig-xrlit)*Lz
        u0_old_left = u0_left
        u0_old_right = u0_right
        ipress = 0 ! index used to record test pressure history
      end if
!
!      SET UP QUANTITIES TO TEST PRESSURE IN A SLICE OF THE SYSTEM
!
      if (mypid.eq.16) then
        xllit = Lx1
        xlbig = Lx1 + (Lx2-Lx1)/nxcell
        call intersct(Lx1sys, Ly1sys, Lx2sys, Ly2sys, xlbig, 0.,
&                   xlbig, Ly1, xlint, ylint, cross)
        vol_left = 0.5*(Ly1+ylint)*(xlbig-xllit)*Lz
      end if
!
      end if
!
!      CHECK HOW MANY PARTICLES EACH PROC IS STARTING
!
      if (mypid.ne. 0) then
        CALL PVMFINITSEND( pvmdefault, ibuf )
        CALL PVMFPACK( INTEGER8, npart, 1, 1, ierr )
        CALL PVMFSEND( 0, 1, ierr )
      else
        npart = 0
        do m = 1, nprocs-1
          CALL PVMFRECV( M, 1, ibuf )
          CALL PVMFUNPACK( INTEGER8, npartw, 1, 1, ierr )
          npart = npart + npartw
        end do
        write (*,*) 'Total number of initial particles: ',npart
      end if
!
!      STOP HERE FOR NOW TO TEST
c      STOP
!
!      BEGIN MAIN ROUTINE: ALL PROCESSORS *****
!
      nstep = 1
      if(mypid.eq.1) then
        open(unit=69, file='v1bot.dat')
        open(unit=70, file='v2bot.dat')

```

```

!       open(unit=71, file='v3bot.dat')
!       open(unit=72, file='v1top.dat')
!       open(unit=73, file='v2top.dat')
!       open(unit=74, file='v3top.dat')
!       end if
!
!       if(mypid.eq.16) then
!         open(unit=75, file='p_slice.dat')
!       end if
!
do istep = 1,nstep
!
!       DO WORKERS PARTICLE SIMULATION
!
!       if (mypid .ne. 0) then
!
!         time = tau
!         call movem6(ywall, mpv, vwall, time, strikes, delv, iseed,
&                   Lx1sys, Ly1sys, Lx2sys, Ly2sys, Lzsys, mass,
&                   u0_left, u0_right, v0, w0, FluxOut, density, T)
!
!       SEND OUT LIST OF PARTICLES THAT HAVE LEFT WORKER SYSTEM
!
!       SEND TO RIGHT NEIGHBOR
!
!       if (mypid.lt.nprocs-1) then
!         CALL PVMFINITSEND( pvmdefault, ibuf )
!         CALL PVMFPACK( INTEGER8, nr_out, 1, 1, ierr )
!         CALL PVMFPACK( REAL8, xr_out, nr_out, 1, ierr)
!         CALL PVMFPACK( REAL8, yr_out, nr_out, 1, ierr)
!         CALL PVMFPACK( REAL8, v1r_out, nr_out, 1, ierr)
!         CALL PVMFPACK( REAL8, v2r_out, nr_out, 1, ierr)
!         CALL PVMFPACK( REAL8, v3r_out, nr_out, 1, ierr)
!         CALL PVMFSEND( mypid+1, 100, ierr )
!       end if
!
!       SEND TO LEFT NEIGHBOR
!
!       if (mypid.gt.1) then
!         CALL PVMFINITSEND( pvmdefault, ibuf )
!         CALL PVMFPACK( INTEGER8, nl_out, 1, 1, ierr )
!         CALL PVMFPACK( REAL8, xl_out, nl_out, 1, ierr)
!         CALL PVMFPACK( REAL8, yl_out, nl_out, 1, ierr)
!         CALL PVMFPACK( REAL8, v1l_out, nl_out, 1, ierr)
!         CALL PVMFPACK( REAL8, v2l_out, nl_out, 1, ierr)
!         CALL PVMFPACK( REAL8, v3l_out, nl_out, 1, ierr)
!         CALL PVMFSEND( mypid-1, 101, ierr )
!       end if
!
!       RECEIVE LIST OF PARTICLES THAT ENTER WORKER SYSTEM
!
!       RECEIVE FROM RIGHT NEIGHBOR
!
!       if (mypid.lt.nprocs-1) then
!         CALL PVMFRECV( mypid+1, 101, ibuf )
!         CALL PVMFUNPACK( INTEGER8, nr_out, 1, 1, ierr )
!         CALL PVMFUNPACK( REAL8, xr_out(1), nr_out, 1, ierr)
!         CALL PVMFUNPACK( REAL8, yr_out(1), nr_out, 1, ierr)
!         CALL PVMFUNPACK( REAL8, v1r_out(1), nr_out, 1, ierr)
!         CALL PVMFUNPACK( REAL8, v2r_out(1), nr_out, 1, ierr)
!         CALL PVMFUNPACK( REAL8, v3r_out(1), nr_out, 1, ierr)
!
!       ADD NEW PARTICLES TO LISTS

```

```

!
      do i = 1,nr_out
        x(npart + i) = xr_out(i)
        y(npart + i) = yr_out(i)
        v(npart + i, 1) = v1r_out(i)
        v(npart + i, 2) = v2r_out(i)
        v(npart + i, 3) = v3r_out(i)
      end do
!
      npart = npart + nr_out
!
end if
!
RECEIVE FROM LEFT NEIGHBOR
!
if (mypid.gt.1) then
  CALL PVMFRECVR( mypid-1, 100, ibuf )
  CALL PVMFUNPACK( INTEGER8, nl_out, 1, 1, ierr )
  CALL PVMFUNPACK( REAL8, xl_out(1), nl_out, 1, ierr)
  CALL PVMFUNPACK( REAL8, yl_out(1), nl_out, 1, ierr)
  CALL PVMFUNPACK( REAL8, v1l_out(1), nl_out, 1, ierr)
  CALL PVMFUNPACK( REAL8, v2l_out(1), nl_out, 1, ierr)
  CALL PVMFUNPACK( REAL8, v3l_out(1), nl_out, 1, ierr)
!
!
  ADD NEW PARTICLES TO LISTS
!
  do i = 1,nl_out
    x(npart + i) = xl_out(i)
    y(npart + i) = yl_out(i)
    v(npart + i, 1) = v1l_out(i)
    v(npart + i, 2) = v2l_out(i)
    v(npart + i, 3) = v3l_out(i)
  end do
!
  npart = npart + nl_out
!
end if
!
!
call sorter4(Lx1, Lx2, Ly1sys, cell_vol, istep) ! Sort into cells
call colide4(vrmax, selxtra, coeff, iseed,col)
!
GET PROGRESS FROM WORKER 1 and NPROCS - 1
!
if (mypid.eq.1 .and. mod(istep,nstep/10).lt.1) then
  write (*,*) 'Done ',istep,' steps out of ',nstep,
&      ' steps ',col,' collisions'
  write (*,*) ' Strikes on bottom wall = ',strikes(1),
&      ' top wall = ',strikes(2)
  write(*,*) 'Total particles in system:      ',npart
  write(*,*) 'Total particles gone:          ',npart_leave
  write(*,*) 'The left pressure is:          ',press_left
  write(*,*) 'u0_left is:                    ',u0_left
endif
if(mypid.eq.workers .and. mod(istep,nstep/10).lt.1) then
  write(*,*) 'The right pressure is:         ',press_right
  write(*,*) 'u0_right is:                   ',u0_right
  write (*,*) ' '
end if
!
!
TEST PRESSURE AT ENDS AND ADJUST FOR BOUNDARY CONDITION
!
if (mypid.eq.1) then

```



```

        write (*,*) ' '
        write (*,*) 'Total number of FINAL particles: ',npart
    end if
!
! CALCULATE FORCE ON UPPER WALL
!
    if (mypid.ne.0) then
        force(2) = eff_num*dvtot(2)*mass/tau/nstep
        cos_theta = Lx2sys/sqrt((Ly1sys-Ly2sys)**2 + Lx2sys**2)
        force(2) = force(2)*cos_theta ! horizontal projection
    end if
!
! NORMALIZE STATISTICS
!
    if (mypid.ne.0) then
!
        do i=1,ygrid
            do j = 1,xgrid
                if (ave_n(i,j).ne.0.) then
                    do k=1,3
                        ave_u(i,j,k) = ave_u(i,j,k)/ave_n(i,j)
                    end do
                    ave_u2(i,j) = ave_u2(i,j)/ave_n(i,j)
                    ave_T(i,j) = abs(mass*(ave_u2(i,j) - ave_u(i,j,1)**2
& - ave_u(i,j,2)**2 - ave_u(i,j,3)**2)/(3.*boltz))
                    ave_n(i,j) = ave_n(i,j)/nsamp
                end if
                if (statcell_vol(i,j).ne.0.) then
& ave_pres(i,j) = ave_n(i,j)*boltz*ave_T(i,j)*
                    eff_num/statcell_vol(i,j)
                else
                    ave_pres(i,j) = 0.
                end if
            end do
            ycell(i) = (i-0.5)*Ly1/ygrid
        end do
!
        do i = 1,xgrid
            xcell(i) = Lx1 + (i-0.5)*(Lx2-Lx1)/xgrid
        end do
!
    end if
!
! COMBINE ALL RESULTS TO SINGLE DATA FILES
!
! SEND DATA TO PROCESSOR ZERO
!
    if (mypid.ne.0) then
!
        CALL PVMFINITSEND( pvmdefault, ibuf )
        CALL PVMFPACK( INTEGER8, xgrid, 1, 1, ierr )
        CALL PVMFPACK( INTEGER8, ygrid, 1, 1, ierr )
        CALL PVMFPACK( INTEGER8, nxcell, 1, 1, ierr )
        CALL PVMFPACK( INTEGER8, nycell, 1, 1, ierr )
        if (mypid.eq.1) then
            CALL PVMFPACK( INTEGER8, ipress, 1, 1, ierr )
            CALL PVMFPACK( REAL8, pleft, ipress, 1, ierr )
            CALL PVMFPACK( INTEGER8, step, ipress, 1, ierr )
        end if
        if (mypid.eq.workers) then
            CALL PVMFPACK( REAL8, pright, ipress, 1, ierr )
        end if
        CALL PVMFPACK( REAL8, ave_n, ncmax*ncmax, 1, ierr )
    end if

```

```

CALL PVMFPACK( REAL8, ave_u, ncmax*ncmax*3, 1, ierr )
CALL PVMFPACK( REAL8, ave_T, ncmax*ncmax, 1, ierr )
CALL PVMFPACK( REAL8, ave_pres, ncmax*ncmax, 1, ierr )
CALL PVMFPACK( REAL8, statcell_vol, ncmax*ncmax, 1, ierr )
CALL PVMFPACK( REAL8, cell_vol, ncmax*ncmax, 1, ierr )
CALL PVMFPACK( REAL8, xcell, xgrid, 1, ierr )
if (mypid.eq.1) then
    CALL PVMFPACK( REAL8, ycell, ygrid, 1, ierr )
end if
CALL PVMFPACK( REAL8, force(2), 1, 1, ierr )
CALL PVMFSEND( 0, mypid, ierr )
!
end if
!
RECEIVE DATA FROM WORKERS
!
if (mypid.eq.0) then
!
    open(11,file='ave_n.dat')
    open(12,file='ave_vx.dat')
    open(13,file='ave_vy.dat')
    open(14,file='ave_vz.dat')
    open(15,file='ave_T.dat')
    open(16,file='xcell.dat')
    open(17,file='ycell.dat')
    open(20,file='cell.dat')
    open(21,file='statcell.dat')
    open(22,file='pleft.dat')
    open(23,file='pright.dat')
    open(24,file='step.dat')
    open(25,file='ave_pres.dat')
    open(unit=26, file='ave_n_r.dat', form='unformatted')
    open(unit=27, file='ave_u_r.dat', form='unformatted')
    open(unit=28, file='ave_T_r.dat', form='unformatted')
    open(unit=29, file='force.dat')
!
do id = 1,workers
!
    CALL PVMFRECV( id, id, ierr )
    CALL PVMFUNPACK( INTEGER8, xgrid, 1, 1, ierr )
    CALL PVMFUNPACK( INTEGER8, ygrid, 1, 1, ierr )
    CALL PVMFUNPACK( INTEGER8, nxcell, 1, 1, ierr )
    CALL PVMFUNPACK( INTEGER8, nycell, 1, 1, ierr )
    if (id.eq.1) then
        CALL PVMFUNPACK( INTEGER8, ipress, 1, 1, ierr )
        CALL PVMFUNPACK( REAL8, pleft(1), ipress, 1, ierr )
        CALL PVMFUNPACK( INTEGER8, step(1), ipress, 1, ierr )
    end if
    if (id.eq.workers) then
        CALL PVMFUNPACK( REAL8, pright(1), ipress, 1, ierr )
    end if
    CALL PVMFUNPACK( REAL8, ave_n(1,1), ncmax*ncmax, 1, ierr )
    CALL PVMFUNPACK( REAL8, ave_u(1,1,1), ncmax*ncmax*3, 1,
&                                ierr )
    CALL PVMFUNPACK( REAL8, ave_T(1,1), ncmax*ncmax, 1, ierr )
    CALL PVMFUNPACK( REAL8, ave_pres(1,1), ncmax*ncmax, 1,
&                                ierr )
    CALL PVMFUNPACK( REAL8, statcell_vol(1,1), ncmax*ncmax, 1,
&                                ierr )
    CALL PVMFUNPACK( REAL8, cell_vol(1,1), ncmax*ncmax, 1,
&                                ieff )
    CALL PVMFUNPACK( REAL8, xcell(1), xgrid, 1, ierr )
    if (id.eq.1) then

```

```

      CALL PVMFUNPACK( REAL8, ycell(1), ygrid, 1, ierr )
    end if
    CALL PVMFUNPACK( REAL8, force(2), 1, 1, ierr)

!
!
    write(29,*) force(2)

!
    do j=1,xgrid
      do i = 1,ygrid
        write (11,1001) ave_n(i,j)
        write (26) ave_n(i,j)      ! unformatted data for restart
        write (12,1001) ave_u(i,j,1)
        write (13,1001) ave_u(i,j,2)
        write (14,1001) ave_u(i,j,3)
        do k = 1,3
          write (27) ave_u(i,j,k)
        end do
        write (15,1001) ave_T(i,j)
        write (28) ave_T(i,j)
        write (21,1001) statcell_vol(i,j)
        write(25,1001) ave_pres(i,j)
      end do
      write (11,1003)
      write (12,1003)
      write (13,1003)
      write (14,1003)
      write (15,1003)
      write (21,1003)
      write (25,1003)
    end do

!
    do j=1,nxcell
      do i = 1,nycell
        write (20,1001) cell_vol(i,j)
      end do
      write (20,1003)
    end do

!
    do i = 1,xgrid
      write(16,1002) xcell(i)
    end do

!
    if (id.eq.1) then
      do i = 1,ipress
        write(22,*) pleft(i)
        write(24,*) step(i)
      end do
      do i = 1,ygrid
        write(17,1002) ycell(i)
      end do
      close(unit=17)
      close(unit=22)
      close(unit=24)
    end if
    if (id.eq.workers) then
      do i = 1,ipress
        write(23,*) pright(i)
      end do
      close(unit=23)
    end if

!
    end do
!

```

```
!
    close(unit=11)
    close(unit=12)
    close(unit=13)
    close(unit=14)
    close(unit=15)
    close(unit=16)
    close(unit=17)
    close(unit=20)
    close(unit=21)
    close(unit=25)
    close(unit=26)
    close(unit=27)
    close(unit=28)
    close(unit=29)
!
    end if
!
    stop
!
1001 format(e12.6, '          ', $)
1002 format(e12.6)
1003 format(/)
!
    end
```



```

subroutine cellvol(Lx1sys,Ly1sys,Lx2sys,Ly2sys,
&               Lx1,Ly1,Lx2,Ly2,Lz,nxcell,nycell,cell_vol)
! This routine calculates the volume of all cells. It uses the subroutine
! "intersect" to determine if and where any cell walls intersect the upper
! angled wall.
! All cell volumes are stored in a 2D array "cell_vol" and returned

include 'param.txt'
real*8 cell_vol(ncmax,ncmax), Lx1, Lx2, Ly1, Ly2, x1, x2, x3, x4
real*8 Lx1sys, Ly1sys, Lx2sys, Ly2sys, Lz
real*8 y1, y2, y3, y4, xlint, xrint, xtint, xbint, max_vol
integer nxcell, nycell, i, j
integer crossleft, crossright, crosstop, crossbot

max_vol = Ly1sys*(Lx2-Lx1)*Lz/nycell/nxcell
do i = 1,nycell
  do j = 1,nxcell
    x1 = Lx1 + (Lx2-Lx1)*(j-1)/nxcell
    x2 = Lx1 + (Lx2-Lx1)*j/nxcell
    x3 = x2
    x4 = x1
    y1 = Ly1sys*(i-1)/nycell
    y2 = y1
    y3 = Ly1sys*i/nycell
    y4 = y3
    call intersect(Lx1sys,Ly1sys,Lx2sys,Ly2sys,
&               x1,y1,x4,y4,xlint,ylint,crossleft) ! Test left cell wall
    call intersect(Lx1sys,Ly1sys,Lx2sys,Ly2sys,
&               x2,y2,x3,y3,xrint,yrint,crossright) ! Test right cell wall
    call intersect(Lx1sys,Ly1sys,Lx2sys,Ly2sys,
&               x3,y3,x4,y4,xtint,ytint,crosstop) ! Test top cell wall
    call intersect(Lx1sys,Ly1sys,Lx2sys,Ly2sys,
&               x1,y1,x2,y2,xbint,ybint,crossbot) ! Test bottom cell wall
    if (crossleft.ne.1 .and. crossright.ne.1) then
      if (ylint.gt.y4 .and. yrint.gt.y3) then
        cell_vol(i,j) = max_vol ! case 5
      else if (ylint.lt.y1 .and. yrint.lt.y2) then
        cell_vol(i,j) = 0. ! case 6
      else if (crosstop.eq.1 .and. crossbot.eq.1) then
        cell_vol(i,j) = 0.5*(xtint-x4+xbint-x1)*(y4-y1)*Lz ! case 2
      end if
    else if (crossleft.eq.1 .and. crossbot.eq.1) then
      cell_vol(i,j) = 0.5*(xbint-x1)*(ylint-y1)*Lz ! case 1
    else if (crossright.eq.1 .and. crosstop.eq.1) then
      cell_vol(i,j) = max_vol-0.5*(x3-xtint)*(y3-yrint)*Lz ! case 3
    else if (crossleft.eq.1 .and. crossright.eq.1) then
      cell_vol(i,j) = 0.5*(ylint-y1+yrint-y2)*(x2-x1)*Lz ! case 4
    else
      write(*,*) 'COULD NOT DETERMINE CELL VOLUME'
      stop
    end if
  end do
end do
return
end

```

```

subroutine cltrap(v_in,iseed,m,T_surf,v_out)
  ! Routine to draw from Cercignani-Lampis distribution
  real*8 v_in(3),v_out(3)
  real*8 m,k,mpv,na,pa
  integer*8 iseed

      pi = 4.0*atan(1.0)
  !
      T_surf = 273.0      ! Surface temperature
      E = 0.5*m*(v_in(1)**2+v_in(2)**2+v_in(3)**2) ! Incomming energy (J)
      E = E/1.6e-19      ! energy in (eV)
  !
  ! Z-DOL/Pt(111) at 273 K
  !
      na = 0.709397 ! normal accomodation
      pa = 0.796853 ! parallel accomodation
  !
  ! GLASS at 293 K
  !
      na = 0.992389
      pa = 0.651207
  !
  ! STARFIRE at 293 K
  !
      na = 1.0962
      pa = 0.557888
  !
  !
      sigma = 1 - pa*E      ! momentum accommodation
      alphas = sigma*(2-sigma) ! convert to energy accomodation
      alphan = 1 - na*E
  !
  ! PIC ACCOMODATION VALUES
  !
      alphas = 0.7
      alphan = 0.7
  !
      alphas = 0.95      ! Transverse accomodation
      alphan = 0.95      ! Normal accomodation
      alcot = sqrt(1.0 - alphas)
     alcon = sqrt(1.0 - alphan)
      k = 1.38e-23      ! Boltzmann's constant
      mpv = sqrt(2*k*T_surf/m) ! Most probable velocity

      qn = -v_in(2)/mpv*alcon
      r = sqrt(-alphan*alog(1.0-rand(iseed)))
      th = 2*pi*rands(iseed)
      rx = r*cos(th) + qn
      ry = r*sin(th)
      v_out(2) = mpv*sqrt(rx**2 + ry**2)

      qt = sqrt(v_in(1)**2 + v_in(3)**2)/mpv*alcot
      phi = atan2(v_in(3),v_in(1))
      r = sqrt(-alphas*alog(1.0-rand(iseed)))
      th = 2*pi*rands(iseed)
      rx = r*cos(th) + qt*cos(phi)
      ry = r*sin(th) + qt*sin(phi)
      v_out(1) = mpv*rx
      v_out(3) = mpv*ry

      return
end

```

```

subroutine colide4(vrmax,selxtra,coeff,iseed,col)
! Program to simulate a dilute gas using DSMC algorithm
! This version illustrates the approach to equilibrium
  include 'param.txt'
  real*8 x(npmax), v(npmax,3), y(npmax), coeff(ncmax,ncmax)
  real*8 selxtra(ncmax,ncmax), vrmax(ncmax,ncmax)
  integer Xref(npmax), index(ncmax,ncmax), cell_n(ncmax,ncmax)
  integer col, isel, icell, jcell
  real*8 vcm(3), vrel(3), rand
  include 'common.txt'
  data pi/3.141592654/

  col = 0 ! Count the total number of collisions
  do icell=1,nycell
    do jcell = 1,nxcell
      number = cell_n(icell,jcell)
      if( number .gt. 1 ) then
        ! Determine how many candidate collision pairs to be
        !selected
        select = coeff(icell,jcell)*number**2*
&          vrmax(icell,jcell) + selxtra(icell,jcell)
        nsel = int(select)! Number of pairs to be selected
        selxtra(icell,jcell) = select-nsel ! Carry over any
                                          ! left-over fraction
        crm = vrmax(icell,jcell)
        do isel=1,nsel
          ! Pick two particles at random out of this cell
          k = int(rand(iseed)*number)
          kk = mod( int(k + rand(iseed)*(number-1))
+            + 1, number)
          ip1 = Xref(k+index(icell,jcell))
          ip2 = Xref(kk+index(icell,jcell))
          cr = 0.
          do k=1,3
            cr = cr + (v(ip1,k)-v(ip2,k))**2
          end do
          cr = sqrt(cr) ! Relative velocity
          if( cr .gt. crm ) then
            crm = cr ! Max. relative velocity
          end if
          if( cr/vrmax(icell,jcell) .gt. rand(iseed) ) then
            ! Collision occurs
            col = col + 1
            do k=1,3
              vcm(k) = 0.5*(v(ip1,k) + v(ip2,k)) ! Center-
                                                    ! of-mass vel.
            end do
            cos_th = 1. - 2.*rand(iseed) ! Cosine(theta)
            sin_th = sqrt(1. - cos_th**2) ! Sine(theta)
            phi = 2*pi*rand(iseed)
            vrel(1) = cr*cos_th ! Relative velocity
            vrel(2) = cr*sin_th*cos(phi) ! after collision
            vrel(3) = cr*sin_th*sin(phi)
            do k=1,3
              v(ip1,k) = vcm(k) + 0.5*vrel(k) ! Post-
                                                    ! collision
              v(ip2,k) = vcm(k) - 0.5*vrel(k) ! velocities
            end do
          end if
        end do
        vrmax(icell,jcell) = crm !Reset maximum relative velocity
      end if
    end do
  end do

```

```
end do  
return  
end
```

```

Real*8 FUNCTION ERF(X)
!#! Erf - Function returns error function of x
  real*8 GAMMP,X

  IF(X.LT.0.)THEN
    ERF = -GAMMP(.5D0,X**2)
  ELSE
    ERF = GAMMP(.5D0,X**2)
  ENDIF
  RETURN
END

Real*8 FUNCTION GAMMP(A,X)
real*8 A,X,GAMSER,GAMMCF,GLN

IF(X.LT.0..OR.A.LE.0.) Write(*,*) 'ERROR in GAMMP'
IF(X.LT.A+1.)THEN
  CALL GSER(GAMSER,A,X,GLN)
  GAMMP=GAMSER
ELSE
  CALL GCF(GAMMCF,A,X,GLN)
  GAMMP=1.-GAMMCF
ENDIF
RETURN
END

SUBROUTINE GSER(GAMSER,A,X,GLN)
integer ITMAX
real*8 EPS
PARAMETER (ITMAX=100,EPS=3.E-7)
real*8 GAMSER,A,X,GLN,SUM,DEL,AP
integer N
real*8 GAMMLN
GLN=GAMMLN(A)
IF(X.LE.0.)THEN
  IF(X.LT.0.) Write (*,*) 'ERROR in GSER'
  GAMSER=0.
  RETURN
ENDIF
AP=A
SUM=1./A
DEL=SUM
DO 11 N=1,ITMAX
  AP=AP+1.
  DEL=DEL*X/AP
  SUM=SUM+DEL
  IF (ABS(DEL).LT.ABS(SUM)*EPS)GO TO 1
11 CONTINUE
Write (*,*) 'ERROR in GSER: A too large, ITMAX too small'
1  GAMSER=SUM*EXP(-X+A*LOG(X)-GLN)
  RETURN
END

SUBROUTINE GCF(GAMMCF,A,X,GLN)
integer ITMAX
real*8 EPS
PARAMETER (ITMAX=100,EPS=3.E-7)
real*8 GLN,GAMMCF,A,X,GOLD,A0,A1,B0,B1,FAC,AN,ANA,G,ANF
integer N
real*8 GAMMLN

```

```

      GLN=GAMMLN(A)
      GOLD=0.
      A0=1.
      A1=X
      B0=0.
      B1=1.
      FAC=1.
      DO 11 N=1,ITMAX
        AN=FLOAT(N)
        ANA=AN-A
        A0=(A1+A0*ANA)*FAC
        B0=(B1+B0*ANA)*FAC
        ANF=AN*FAC
        A1=X*A0+ANF*A1
        B1=X*B0+ANF*B1
        IF(A1.NE.0.) THEN
          FAC=1./A1
          G=B1*FAC
          IF(ABS((G-GOLD)/G).LT.EPS)GO TO 1
          GOLD=G
        ENDIF
11    CONTINUE
      Write(*,*) 'ERROR in GCF: A too large, ITMAX too small'
1    GAMMCF=EXP(-X+A*log(X)-GLN)*G
      RETURN
      END

      Real*8 FUNCTION GAMMLN(XX)
      REAL*8 COF(6),STP,HALF,ONE,FPF,X,TMP,SER
      real*8 XX
      INTEGER J
      DATA COF,STP/76.18009173D0,-86.50532033D0,24.01409822D0,
&      -1.231739516D0,.120858003D-2,-.536382D-5,2.50662827465D0/
      DATA HALF,ONE,FPF/0.5D0,1.0D0,5.5D0/
      X=XX-ONE
      TMP=X+FPF
      TMP=(X+HALF)*LOG(TMP)-TMP
      SER=ONE
      DO J = 1,6
        X=X+ONE
        SER=SER+COF(J)/X
      END DO
      GAMMLN=TMP+LOG(STP*SER)
      RETURN
      END

```

```

subroutine gridsmp4(Lx1,Ly1sys,Lx2,xgrid,ygrid,ave_n,ave_u,ave_u2,
&
nsamp)
! sampler - function to sample density and velocities
include 'param.txt'
real*8 x(npmax), v(npmax,3), y(npmax)
real*8 Lx, Ly, Lx1, Ly1sys, Lx2
real*8 ave_n(ncmax,ncmax), ave_u(ncmax,ncmax,3)
real*8 ave_u2(ncmax,ncmax)
real*8 sum_n(ncmax,ncmax), sum_v(ncmax,ncmax,3)
real*8 sum_v2(ncmax,ncmax)
integer Xref(npmax), index(ncmax,ncmax), cell_n(ncmax,ncmax)
integer i, j, k, icell, jcell, xgrid, ygrid
include 'common.txt'

do i=1,ygrid      ! Zero the sums used to accumulate
  do j = 1,xgrid
    sum_n(i,j) = 0.          ! statistics on density, velocity
    do k=1,3                ! and energy
      sum_v(i,j,k) = 0.
    end do
    sum_v2(i,j) = 0.
  end do
end do
Lx = Lx2 - Lx1 ! Processor dimension
Ly = Ly1sys    ! Use system height
do ipart=1,npart
  xp = x(ipart) - Lx1 ! Shift of origin
  ! Particle 'ipart' is located in cell 'jcell'
  icell=min0( int(xgrid*xp/Lx)+1, xgrid)
  jcell=min0( int(ygrid*y(ipart)/Ly)+1, ygrid)
  sum_n(jcell,icell) = sum_n(jcell,icell)+1
  do k=1,3
    sum_v(jcell,icell,k) = sum_v(jcell,icell,k) + v(ipart,k)
  end do
  sum_v2(jcell,icell) = sum_v2(jcell,icell) +
&    v(ipart,1)**2 + v(ipart,2)**2 + v(ipart,3)**2
end do
! Accumulate running sums of density, u and T for cells
do i=1,ygrid
  do j = 1,xgrid
    ave_n(i,j) = ave_n(i,j) + sum_n(i,j)
    do k=1,3
      ave_u(i,j,k) = ave_u(i,j,k) + sum_v(i,j,k)
    end do
    ave_u2(i,j) = ave_u2(i,j) + sum_v2(i,j)
  end do
end do
nsamp = nsamp + 1 ! Increment number of samples
return
end

```

```

subroutine intersct(Lx1,Ly1,Lx2,Ly2,x1,y1,x2,y2,xint,yint
& ,cross)
! This subroutine determines if the upper wall formed by Lx1...Ly2 intersects
! with the particle trajectory formed by x1...y2
! xint,yint is the coords of the intersection point
! cross = 1 if the lines intersect else cross = 0

    include 'param.txt'
    real*8 Lx1, Lx2, Ly1, Ly2, x1, x2, y1, y2, xint, yint
    real*8 mw, bw, mp, bp, xbig, xlit, ybig, ylit
    real*8 mp_prime, bp_prime, mw_prime, bw_prime
    integer cross
    common /block12/ istep
    common /block100/ i ! use only if particles escape
    common /proc_nums/ mypid, nprocs

!
!   roundoff = 1.e-16 ! Estimated roundoff error in calcs.
!
!   mw = (Ly2-Ly1)/(Lx2-Lx1) ! Slope of the wall
!
!   DETERMINE PARTICLE SLOPE
!
    if (x1.ne.x2) then
        mp = (y2-y1)/(x2-x1) ! If not verticle use slope intercept
    else
        mp = 1e3 ! Set mp to something big in this case (x2=x1)
    end if

!
!   DETERMINE WHICH SLOPE TO USE
!
    if (abs(mp).gt.1) then ! use inverse slope and x-intercept
        mp_prime = (x2-x1)/(y2-y1)
        mw_prime = 1/mw
        bp_prime = x1-mp_prime*y1 ! x-intercept
        bw_prime = Lx1-mw_prime*Ly1 ! x-intercept
        if (mp_prime.eq.mw_prime) then
            cross = 0 ! parallel lines -no intersection
            return
        end if
    else
        yint = (bw_prime - bp_prime)/(mp_prime - mw_prime)
        xint = mw_prime*yint + bw_prime
    end if

!
!   USE Y-VALS TO DETERMINE UPPER AND LOWER
!   END POINTS
!
    if (y2.gt.y1) then
        ybig = y2
        xbig = x2
        ylit = y1
        xlit = x1
    else
        ybig = y1
        xbig = x1
        ylit = y2
        xlit = x2
    end if

!
!   CHECK FOR ESCAPED PARTICLE
!
    if (istep.ge.1) then ! test only for particle data
        if (mp_prime.gt.mw_prime) then
            if (ybig.ge.yint .and. ylit.ge.yint) then

```



```

        write(*,*) ' '
        write(*,*) 'ERROR! PARTICLE ALREADY OUT OF SYSTEM'
        write(*,*) 'Processor #: ',mypid
        write(*,*) 'Particle #: ',i
        write(*,*) 'Step #: ',istep
        write(*,*) 'xold,yold ',x1,y1
        write(*,*) 'xnew,ynew ',x2,y2
        write(*,*) ' '
        cross = -1 ! This is a tag for the escaped particle
        return
    end if
end if
end if
!
if (abs(ybig-yint).le.roundoff) then
    cross = 1 ! allow an absolute roundoff error of 10^-16
    yint = ybig ! Set the intersection equal to the particle
    xint = xbig ! path end point
    return
else if (abs(ylit-yint).le.roundoff) then
    cross = 1
    yint = ylit
    xint = xlit
    return
else if (xint.ge.0. .and. xint.le.Lx2) then ! intersect in boundary
    if (yint.le.ybig .and. yint.ge.ylit) then ! intersect within path
        cross = 1
        return
    end if
end if
!
!
!
OTHER CASE FOR PARTICLE SLOPE
else ! particle slope < 1 use slope and y-intercept
!
!
    bp = y1-mp*x1 ! y-intercept
    bw = Ly1 ! y-intercept
    if (mp.eq.mw) then
        cross = 0 ! parallel lines
        return
    end if
!
    xint = (bp - bw)/(mw - mp)
    yint = mw*xint + bw
!
!
USE X-VALS TO DETERMINE RIGHT AND LEFT
PATH END POINTS
!
    if (x2.gt.x1) then
        ybig = y2
        xbig = x2
        ylit = y1
        xlit = x1
    else
        ybig = y1
        xbig = x1
        ylit = y2
        xlit = x2
    end if
!
!
CHECK TO SEE IF PARTICLE HAS ESCAPED
!

```

```

if (istep.ge.1) then
  out = 0
  if (mp.gt.mw) then
    if (xbig.ge.xint .and. xlit.ge.xint) then
      out = 1
    end if
  else if (mp.lt.mw) then
    if (xbig.le.xint .and. xlit.le.xint) then
      out = 1
    end if
  end if
  if (out.eq.1) then
    write(*,*) ' '
    write(*,*) 'ERROR! PARTICLE ALREADY OUT OF SYSTEM'
    write(*,*) 'Processor #: ',mypid
    write(*,*) 'Particle #: ',i
    write(*,*) 'Step #: ',istep
    write(*,*) 'xold,yold ',x1,y1
    write(*,*) 'xnew,ynew ',x2,y2
    write(*,*) ' '
    cross = -1
    return
  end if
end if
!

if (abs(xbig-xint).le.roundoff) then
  cross = 1          ! allow an absolute roundoff error of 10^-12
  yint = ybig        ! Set the intersection equal to the particle
  xint = xbig        ! path end point
  return
else if (abs(xlit-xint).le.roundoff) then
  cross = 1
  yint = ylit
  xint = xlit
  return
else if (xint.ge.0. .and. xint.le.Lx2) then ! intersect in boundary
  if (xint.le.xbig .and. xint.ge.xlit) then
    cross = 1
    return
  end if
end if
end if
!

cross = 0
return
end

```

```

subroutine maker(rho0,u0,v0,w0,xface,nface,FluxOut,MethodFlag
& ,mpv,pmass,tau,yface,Lz,iseed,Lx1,Ly1,Lx2,Ly2,ywall,vwall,
& strikes,delv)
!
!##! Maker - Routine to create particles fluxing into DSMC region
!##! across a boundary located at x=xface in direction nface
!##! (i.e. if nface=+1 particles move left to right).
!##! Particles come from a reservoir with density rho0,
!##! fluid velocity (u0,v0,w0) and temperature T0
!##! Total Flux of mass,momentum and energy is returned in FluxOut
!##! If MethodFlag = 1 then use Poisson statistics
!##! else generate average number of particles which flux
!
      include 'param.txt'
      real*8 x(npmax), y(npmax), v(npmax,3)
      real*8 rho0,u0,v0,w0,xface,FluxOut(5),ywall(2),vwall(2)
      real*8 delv(2),Lx1,Ly1,Lx2,Ly2,Lz
      integer nface,MethodFlag,strikes(2)
!
      include 'common.txt'
      common /proc_nums/ mypid, nprocs
!
      real*8 mpv,nden,beta,s_cos,erfsc,umax,pumax,AverageN_f
      real*8 utop,uu,pu,u0Mirror
      integer N_f,ii,k
      real*8 rand,thermal,erf
      integer poidev
!
      do k=1,5
         FluxOut(k) = 0.
      end do
      pi = 4.*atan(1.)
      spi = sqrt(pi)
      beta = 1./mpv
      u0Mirror = nface*u0
      s_cos = u0Mirror*beta
      nden = rho0/pmass ! Number density
      erfsc = erf(s_cos)
!
!##! Determine the number of particles to be created
      AverageN_f = tau * nden/eff_num * yface*Lz * 1./(2.*beta*spi) *
&      (exp(-s_cos**2) + spi*s_cos*(1+erfsc))
      if( MethodFlag .eq. 1 ) then
         N_f = poidev(AverageN_f,iseed)
      else
         N_f = AverageN_f + rand(iseed)
      endif
!
      ! write (*,*) 'Creating ',N_f,' particles'
      if( npart+N_f .gt. npmax ) then
         write (*,*) 'ERROR in maker: Created too many particles'
         write (*,*) 'npart = ',npart,' N_f = ',N_f
         stop
      end if
!
!##! Compute the most probable injection x-velocity
      umax = 0.5*(u0Mirror + sqrt(2*mpv**2+u0Mirror**2))
!##! Compute the probability of the most probable injection x-velocity
      pumax = umax*exp(-(beta*(umax-u0Mirror)**2))
      ! write (*,*) 'Most prob. velocity = ',umax
      ! write (*,*) 'Prob. of most prob. velocity = ',pumax
!
!##! Compute highest x-velocity which may be given to a particle

```

```

      utop = umax + min1(6*mpv,3*mpv+3*umax)
      ! write (*,*) 'Highest possible velocity = ',utop
!
!#! Loop over the number of particles to be injected
      do ii=1,N_f
        !#! Use accept-reject to draw random x-velocity
1111      continue ! FORTRAN until loop
          uu = rand(iseed)*utop ! Pick x-velocity at random
          pu = uu*exp(-(beta*(uu-u0Mirror))**2) ! Compute its probability
          if( pu .lt. pumax*rand(iseed) ) goto 1111 ! Accept or reject
          v1 = uu*nface
!
!#! Set y and z velocities using Maxwell-Boltzmann distributions
!
! Set the x-position to something impossible so that "movem" will
! detect it;
! for left face: x = -999.9
! for right face: x = 999.9
!
      v2 = thermal(mpv,iseed) + v0
      v3 = thermal(mpv,iseed) + w0
!      dtr = tau*rand(iseed) ! assign random fraction of tau
!
      if (nface.eq.1) then
        x_old = -999.9
      else
        x_old = 999.9
      end if
!
      y_old = yface*rand(iseed) ! assign random entry height
      vx_old = v1
!
      x_new = x_old + v1*dtr
      y_new = y_old + v2*dtr
!      call strikem4(x_old,y_old,vx_old,x_new,y_new,v1,v2,v3,dtr,
! & ywall,vwall,Lx1,Ly1,Lx2,Ly2,mpv,strikes,delv,iseed,keep)
!      if (keep.eq.1) then
!
! ADD NEW PARTICLES TO END OF LIST
!
      npart = npart + 1 ! add particle to end of list
      x(npart) = x_old
      y(npart) = y_old
      v(npart,1) = v1
      v(npart,2) = v2
      v(npart,3) = v3
!
! CALCULATE FLUXES
!
      FluxOut(1) = FluxOut(1) + 1
      FluxOut(2) = FluxOut(2) + v1
      FluxOut(3) = FluxOut(3) + v2
      FluxOut(4) = FluxOut(4) + v3
      FluxOut(5) = FluxOut(5) + 0.5*(v1**2 + v2**2 + v3**2)
!    end if
!  end do
!
! do ii=1,5
!   FluxOut(ii) = FluxOut(ii) * nface*pmass*eff_num/(yface*Lz)
! end do
!
! return
! end

```

```

subroutine movem6(ywall, mpv, vwall, tau, strikes, delv, iseed,
&                Lx1sys, Ly1sys, Lx2sys, Ly2sys, Lzsys, mass,
&                u0_left, u0_right, v0, w0, FluxOut, density, T)
!
!   Subroutine to move particles for DSMC code in 2 dimensions
!
! *** PARAMETERS ***
!
!   include 'param.txt'
!
! *** REAL ARRAYS ***
!
!   real*8 x(npmax), y(npmax), v(npmax,3), ywall(2), vwall(2)
!   real*8 xl_out(nptmax), yl_out(nptmax)
!   real*8 xr_out(nptmax), yr_out(nptmax)
!   real*8 v1l_out(nptmax), v2l_out(nptmax), v3l_out(nptmax)
!   real*8 v1r_out(nptmax), v2r_out(nptmax), v3r_out(nptmax)
!   real*8 delv(2), FluxOut(5), height(2)
!
! *** REAL VARIABLES ***
!
!   real*8 mpv, vx_old, tau, time, mass
!   real*8 x_old, y_old
!   real*8 Lx1sys, Ly1sys, Lx2sys, Ly2sys, Lzsys
!
! *** INTEGER ARRAYS ***
!
!   integer strikes(2)
!
! *** INTEGER VARIABLES ***
!
!   integer*8 keep, repeat, iseed
!
! *** COMMON VARIABLES ***
!
!   include 'common.txt'
!   common /proc_nums/ mypid, nprocs
!   common /pass/ xl_out, yl_out, xr_out, yr_out,
!   &              v1l_out, v2l_out, v3l_out,
!   &              v1r_out, v2r_out, v3r_out,
!   &              nl_out, nr_out
!   common /block100/ i ! use only if particles escape
!
!
!   do i=1,2
!       strikes(i) = 0.    ! Strikes counts # of impacts for each wall
!       delv(i) = 0.      ! Delv is change in velocity after impact
!   end do
!   height(1) = Ly1sys
!   height(2) = Ly2sys
!   time = tau ! Dummy var. to pass time-step to subs.
!
!   CREATE PARTICLES AND VELOCITIES FLUX AT ENDS
!
!   if (mypid.eq.1) then
!       call maker(density, u0_left, v0, w0, Lx1sys, 1, Fluxout, 1,
!   &              mpv, mass, time, height(1), Lzsys, iseed, Lx1sys,
!   &              Ly1sys, Lx2sys, Ly2sys, ywall, vwall, strikes, delv)
!   end if
!
!   if (mypid.eq.nprocs-1) then
!       call maker(density, u0_right, v0, w0, Lx2sys, -1, Fluxout, 1,

```

```

&          mpv, mass, time, height(2), Lzsys, iseed, Lx1sys,
&          Ly1sys, Lx2sys, Ly2sys, ywall, vwall, strikes, delv)
end if

!
!
! LOOP OVER PARTICLE LIST FOR MOVING
!
repeat = 1
i = 1
nl_out = 0      ! # of particles going to left neighbor
nr_out = 0      ! # of particles going to right neighbor
do while (repeat.eq.1)    ! Stays in loop until all particles
                           ! are processed
    keep = 1

    !
    ! CHECK TO SEE IF PARTICLE IS NEW (INJECTED)
    !
    time = tau
    if (x(i).eq.-999.9) then
        x(i) = 0.      ! Set position to left entrance
        time = tau*rand(iseed) ! use random fract of time step
    else if (x(i).eq.999.9) then
        x(i) = Lx2sys ! Set position to right entrance
        time = tau*rand(iseed)
    end if

    !
    x_old = x(i)          ! Remember original position
    y_old = y(i)
    x(i) = x_old + v(i,1)*time ! Update x and y position of particle
    y(i) = y_old + v(i,2)*time
    vx_old = v(i,1)       ! Remember x-velocity

    !
    ! STRIKEM will determine if particle is destroyed or strikes
    ! Keep = 0 or -1: particle destroyed
    ! Keep = 1: particle processed further
    !
    call strikem5(x_old, y_old, vx_old, x(i), y(i), v(i,1), v(i,2),
&                v(i,3), time, ywall, vwall, Lx1sys, Ly1sys,
&                Lx2sys, Ly2sys, mpv, strikes, delv, iseed, keep, T, mass)

    !
    !
    ! DETERMINE IN WHICH PROCESSOR THE PARTICLE BELONGS
    !
    if (keep.eq.1) then

        id = min0(int((nprocs-1)*x(i)/Lx2sys)+1, nprocs-1)
        if (id .eq. mypid+1) then
            keep = 3      ! particle goes to right neighbor
        else if (id .eq. mypid-1) then
            keep = 2      ! particle goes to left neighbor
        else if (id.ge.mypid+2 .or. id.le.mypid-2) then
            write(*,*) '*****'
            write(*,*) 'ERROR: PARTICLE BEYOND NEIGHBOR'
            write(*,*) '*****'
            keep = 0      ! Destroy this particle to continue run
            return
        else
            keep = 1      ! id = mypid, remains in this proc
        end if

    end if

    if (keep.eq.1) then

```

```

!
!       i = i + 1           ! Continue and leave this particles in this proc.
!
!       else if (keep.eq.2 .or. keep.eq.3 .or. keep.eq.0
&           .or. keep.eq.-1) then
!
!           if (keep.eq.2) then
!
!               nl_out = nl_out + 1           ! Particle exits to the left
!               xl_out(nl_out) = x(i)
!               yl_out(nl_out) = y(i)
!               vl1_out(nl_out) = v(i,1)
!               v2l_out(nl_out) = v(i,2)
!               v3l_out(nl_out) = v(i,3)
!
!           else if (keep.eq.3) then
!
!               nr_out = nr_out + 1           ! Particle exits to the right
!               xr_out(nr_out) = x(i)
!               yr_out(nr_out) = y(i)
!               vlr_out(nr_out) = v(i,1)
!               v2r_out(nr_out) = v(i,2)
!               v3r_out(nr_out) = v(i,3)
!
!           end if
!
!           x(i) = x(npert) ! Replace particle with particle at end of list
!           y(i) = y(npert) ! This destroys the particle
!           y(i) = y(npert) ! keep = 0 or -1
!           do k = 1,3
!               v(i,k) = v(npert,k)
!           end do
!           npert = npert - 1
!
!       else ! keep not equal to: 0, 1, 2, 3, -1
!           write(*,*) 'ERROR: "movem" cannot place particle'
!       end if
!
!       if (i.gt.npart) then
!           repeat = 0           ! Exit main loop after all particles are proc.
!       end if
!
!   end do
!
!   return
!   end

```

```

integer FUNCTION POIDEV(XM,IDUM)
  real*8 pi
  PARAMETER (PI=3.141592654)
  real*8 xm
  integer*4 idum
  real*8 oldm,g,em,t,sq,alxm,y
  real*8 gammln,rand
  save alxm,g,oldm,sq
  DATA OLDM /-1./
  IF (XM.LT.12.) THEN
    IF (XM.NE.OLDM) THEN
      OLDM=XM
      G=exp(-XM)
    ENDIF
    EM=-1
    T=1.
2    EM=EM+1.
    T=T*RAND(IDUM)
    IF (T.GT.G) GO TO 2
  ELSE
    IF (XM.NE.OLDM) THEN
      OLDM=XM
      SQ=sqrt(2.*XM)
      ALXM=log(XM)
      G=XM*ALXM-GAMMLN(XM+1.)
    ENDIF
1    Y=TAN(PI*RAND(IDUM))
    EM=SQ*Y+XM
    IF (EM.LT.0.) GO TO 1
    EM=int(EM)
    T=0.9*(1.+Y**2)*exp(EM*ALXM-GAMMLN(EM+1.))-G
    IF (RAND(IDUM).GT.T) GO TO 1
  ENDIF
  ! POIDEV=EM
  POIDEV=int(EM+0.5)
  RETURN
END

```



```

subroutine pressure(xlit,xbig,mass,volume,press)
! This subroutine measures the pressure of particles within xlit and xbig
! It first measures the temperature and then calculates the pressure
! from the ideal gas law.
! The volume that the particles occupy must be given to it.

    include 'param.txt'
    real*8 x(npmax), v(npmax,3), sum_v(3), sum_vv(3)
    real*8 mass, KE
    include 'common.txt'

    N = 0
    do k = 1,3
        sum_v(k) = 0.
        sum_vv(k) = 0.
    end do
    do i = 1,npart
        if (x(i).ge.xlit .and. x(i).le.xbig) then
            do k = 1,3
                sum_v(k) = sum_v(k) + v(i,k)
                sum_vv(k) = sum_vv(k) + v(i,k)**2
            end do
            N = N + 1          ! Keep track of number of particles
        end if
    end do
    KE = 0.5*mass*eff_num*(sum_vv(1) + sum_vv(2) + sum_vv(3)
$      - (1/N)*(sum_v(1)**2 + sum_v(2)**2 + sum_v(3)**2))
    cf = (N + 1)/N          ! Correction factor
    press = 2*KE/(3*volume)*cf
    return
end

```

```
real*8 function rand(seed)
  integer*8 seed
  save
  real*8 a,m,temp
  data a,m/16807.0, 2147483647.0/

  temp = a*seed
  seed = mod(temp,m)
  rand = seed/m

  return
end
```

```

subroutine sorter4(Lx1, Lx2, Ly1sys, cell_vol, step)
! Routine to sort particles into a 2D grid of cells
  include 'param.txt'
  real*8 x(npmax), v(npmax,3), y(npmax)
  real*8 cell_vol(ncmax,ncmax)
  integer Xref(npmax), index(ncmax,ncmax), cell_n(ncmax,ncmax)
  integer ix(npmax), jy(npmax), temp(ncmax,ncmax),nycell, nxcell
  integer icell, jcell, npart, step
  real*8 Lx,Ly,Lx1,Lx2,Ly1sys
  include 'common.txt'

  do icell=1,nycell
    do jcell = 1,nxcell
      cell_n(icell,jcell) = 0
      temp(icell,jcell) = 0
    end do
  end do
  Lx = Lx2 - Lx1 ! Shift origin to origin of processor
  Ly = Ly1sys
  do ipart=1,npart
    xp = x(ipart) - Lx1 ! shift of origin
    ix(ipart) = min0(int(nxcell*xp/Lx)+1,nxcell)
    jy(ipart) = min0(int(nycell*y(ipart)/Ly)+1,nycell)
    if (cell_vol(jy(ipart),ix(ipart)).eq.0.) then
!      write(*,*) '**** ERROR!  PARTICLE OUT OF SYSTEM ****'
!      write(*,*) 'Particle #: ',ipart
!      write(*,*) 'Step #: ',step
!      stop
!    end if
    cell_n(jy(ipart),ix(ipart)) = cell_n(jy(ipart),ix(ipart)) + 1
  end do
  m=1
  do icell=1,nycell
    do jcell = 1,nxcell
      index(icell,jcell) = m
      m = m + cell_n(icell,jcell)
    end do
  end do
  do ipart=1,npart
    jcell = ix(ipart)
    icell = jy(ipart)
    temp(icell,jcell)=temp(icell,jcell)+1
    k = index(icell,jcell) + temp(icell,jcell) - 1
    Xref(k) = ipart
  end do
  return
end

```

```

SUBROUTINE STARTER(xgridsys, ygridsys, Lx1, Lx2, Ly1, Ly2,
    &                Lx1sys, Ly1sys, Lx2sys, Ly2sys, m, clone)
!
! This routine reads in cell data (Temperature, Particle #, Center of
! mass flow velocity) and creates new particles for each cell based on
! the statistical data of a previous run.
!
! *** PARAMETERS ***
!
!     include 'param.txt'
!
! *** INCLUDES ***
!
!     include '/usr/include/mpp/fpvm3.h'
!
! *** EXTERNAL ***
!
!     external PVMFMYTID, PVMFINITSEND, PVMFPACK
!     external PVMFRECVR, PVMFSEND, PVMFUNPACK, PVMFGSIZE, PVMFGETPE
!
! *** REAL ARRAYS ***
!
!     real*8 x(npmax), v(npmax,3), y(npmax), m
!     real*8 ave_u2(ncmax,ncmax)
!     real*8 ave_n(ncmax,ncmax), ave_u(ncmax,ncmax,3)
!     real*8 ave_T(ncmax,ncmax)
!
! *** INTEGER ARRAYS ***
!
!
!
! *** REALS ***
!
!     real*8 mass, Ly1, Lx1, Ly2, Lx2, mpv, Lz, rand, Lx
!     real*8 Lx1sys, Ly1sys, Lx2sys, Ly2sys, Lzsys, thermal
!
! *** INTEGERS ***
!
!     integer*8 icell, jcell, xgrid, ygrid
!     integer*8 xgridsys, ygridsys, x_index_origin
!     integer*8 npart, iseed, workers, clone
!     integer*8 crosstop
!
! *** CHARACTER ***
!
!
! *** COMMON ***
!
!     common /block1/ x
!     common /block2/ y
!     common /block3/ v
!     common /block4/ npart
!     common /proc_nums/ mypid, nprocs
!
! *** DATA
!
!     data boltz/1.3806e-23/
!
! READ IN AND SEND STATISTICAL DATA
!
!     workers = nprocs-1
!
!     if (mypid.eq.0) then
!

```

```

if (mod(xgridsys,workers).ne.0) then
  write(*,*) '*** ERROR:  CANNOT RESTART***'
  write(*,*) ' (# of cells)/(workers) must = integer'
  stop
end if

!
open(unit=11, file='ave_n_r.dat',form='unformatted')
open(unit=12, file='ave_T_r.dat',form='unformatted')
open(unit=13, file='ave_u_r.dat',form='unformatted')
!

do i = 1,xgridsys
  do j = 1,ygridsys
    read(11) ave_n(j,i)
    read(12) ave_T(j,i)
    do k = 1,3
      read(13) ave_u(j,i,k)
    end do
  end do
end do

!
close(unit=11)
close(unit=12)
close(unit=13)
!

CALL PVMFINITSEND( pvmdefault, ibuf )
CALL PVMFPACK( REAL8, ave_n, ncmax*ncmax, 1, ierr )
CALL PVMFPACK( REAL8, ave_u, ncmax*ncmax*3, 1, ierr )
CALL PVMFPACK( REAL8, ave_T, ncmax*ncmax, 1, ierr )
do id=1,workers
  CALL PVMFSEND( id, 666, ierr )
end do

!
else ! WORKERS RECEIVE AND INITIALIZE PARTICLES
!
CALL PVMFRECV( 0, 666, ibuf )
CALL PVMFUNPACK( REAL8, ave_n, ncmax*ncmax, 1, ierr )
CALL PVMFUNPACK( REAL8, ave_u, ncmax*ncmax*3, 1, ierr )
CALL PVMFUNPACK( REAL8, ave_T, ncmax*ncmax, 1, ierr )
!

xgrid = xgridsys/workers
ygrid = ygridsys
x_index_origin = (mypid-1)*xgrid ! starting index to read data
Lx = Lx2 - Lx1 ! shift of origin to processor origin
!
!
READ DATA AND GET PROCESSOR CELL INFO
!

npart = 1 ! keep track of created particles
do j = 1,xgrid
  do i = 1,ygrid
    num = ave_n(i,j + x_index_origin)*clone + 0.5
    ! multiply num by cloning factor and round
    T = ave_T(i, j + x_index_origin)
    vxcm = ave_u(i, j + x_index_origin, 1) ! centor of mass x-vel.
    vycm = ave_u(i, j + x_index_origin, 2)
    vzcm = ave_u(i, j + x_index_origin, 3)
    xmax = Lx1 + (Lx2-Lx1)*j/xgrid ! Cell corner coords.
    ymax = Ly1*i/ygrid
    xmin = Lx1 + (Lx2-Lx1)*(j-1)/xgrid
    ymin = Ly1*(i-1)/ygrid
    mpv = sqrt(2*boltz*T/m) ! most probable velocity
  end do
end do

!
!
CREATE CELL PARTICLES AND VELOCITIES
!

```

```

      ipart = 1
      do while(ipart.le.num)
        x(npart) = xmin + (xmax-xmin)*rand(iseed)
        y(npart) = ymin + (ymax-ymin)*rand(iseed)
        call intersct(Lx1sys,Ly1sys,Lx2sys,Ly2sys,
&          0.,0.,x(npart),y(npart),xint,yint,crosstop)
        if (crosstop.ne.1) then      ! Particle is accepted
!
!          GET MB VELOCITY AND ADD CM VELOCITY TO IT
!
          v(npart,1) = vxcm + thermal(mpv,iseed)
          v(npart,2) = vycm + thermal(mpv,iseed)
          v(npart,3) = vzcm + thermal(mpv,iseed)
          ipart = ipart + 1
          npart = npart + 1
        end if
      end do
    end do
  end do
  npart = npart - 1
!
end if
return
!
end

```

```

subroutine strikem5(x_old, y_old, vx_old, x, y, v1, v2, v3, time,
&                  ywall, vwall, Lx1sys, Ly1sys, Lx2sys, Ly2sys,
&                  mpv, strikes, delv, iseed, keep, T, mass)
!
!   This subroutine handles when a particle strikes the wall. Is used by the
!   "movem" subroutine and the "maker" subroutine. It also checks to see if
!   the particle has left the system.
!   "keep" is a flag returned to movem such that:
!       keep = 0;   particle has exited at one of the ends
!       keep = 1;   particle has remained in system
!       keep = -1;  particle has escaped by numerical error
!
! *** REAL ARRAYS ***
!
!   real*8 vwall(2), ywall(2), direction(2), delv(2), v_in(3)
!   real*8 v_out(3)
!
! *** INTEGER ARRAYS ***
!
!   integer*8 strikes(2)
!
! *** REAL VARIABLES ***
!
!   real*8 x, y, v1, v2, v3, phi, v_par11, time, mpv, alpha
!   real*8 dtb, dtr, x_old, y_old, vx_old, x_wall, m, m_mod, mass
!   real*8 Lx1sys, Lx2sys, Ly1sys, Ly2sys, xint, yint, rand
!
! *** INTEGER VARIABLES ***
!
!   integer*8 flag, repeat, crosstop, struck_already
!   integer*8 iseed
!
! *** COMMON VARIABLES ***
!
!   common /block11/ npart_leave
!   common /proc_nums/ mypid, nprocs
!
! *** DATA ***
!
!   data twopi, direction/6.283185307,1.,-1./
!
!
!   repeat = 1
!
! When a particle strikes the wall it is moved off the wall on the next loop
! and on the next loop struck_already will be set to 1
!
!   struck_already = 0      ! Used when moving a particle off the wall
!
!   do while(repeat.lt.100)
!
!   if the program loops 100 times on a single particle obviously there is
!   a problem with the particle. The particle will be destroyed and an
!   error message will be displayed
!
!       DETERMINE WHICH WALL IF ANY THE PARTICLE STRIKES
!
!       if (y.le.0.) then      ! Particle strikes the bottom wall
!           flag = 1
!       else
!
!           call intersct(Lx1sys, Ly1sys, Lx2sys, Ly2sys, x_old, y_old,

```

```

&                                x, y, xint, yint, crosstop)
!
!   if (crosstop.eq.-1) then
!
!       keep = -1 ! Particle has escaped
!       return
!
!   end if
!
!   if (crosstop.eq.1 .and. struck_already.eq.0) then
!
!       flag = 2 ! particle strikes upper wall
!       ywall(flag) = yint ! y-coord of strike
!
!   else if (crosstop.eq.0 .or. struck_already.eq.1) then
!
!       THE PARTICLE HAS NOT STRUCK WALL OR HAS BEEN MOVED OFF
!       THE UPPER WALL FROM A PREVIOUS STRIKE
!
!       if (x.ge.0. .and. x.le.Lx2sys) then
!
!           keep = 1 ! particle is in system
!           return
!
!       else
!
!           keep = 0 ! Particle exits entire system
!           npart_leave = npart_leave + 1
!           return
!
!       end if
!
!   else
!
!       write(*,*) '*** Logical Error in "Strikem" ***'
!       stop
!       end if
!
!   end if
!
!   PARTICLE HAS STRUCK AND IS STILL IN SYSTEM: flag = 1 or 2
!   Process strike data:
!
!   pmaxwell = 1 ! Used for maxwell accomodation
!   test = rand(iseed) ! Used to draw prob dist. for maxwell
!   If test < pmaxwell use thermal scattering (pmaxwell prob)
!   If test > pmaxwell use specular scattering
!
!   if (flag.eq.1) then
!
!       STRIKES LOWER WALL: USE C-L SCATTERING or MAXWELL
!
!       v1 = v1 - vwall(1) ! convert to wall frame
!
!       MAXWELL ACCOMODATION
!
!       if (test.gt.pmaxwell) then ! use specular
!
!           v1 = v1 + vwall(flag)
!           v2 = -v2
!           v3 = v3
!
!       else ! use thermal
!
!           v2 = direction(flag)*sqrt(-log(1.-rand(iseed)))*mpv
!           phi = twopi*rand(iseed)

```



```

!           v_par11 = sqrt(-log(1.-rand(iseed))) * mpv
!           v1 = v_par11*sin(phi) + vwall(flag)
!           v3 = v_par11*cos(phi)
!       end if
!
!       C-L SCATTERING
!
!       v_in(1) = v1
!       v_in(2) = v2
!       v_in(3) = v3
!       call cltrap(v_in, iseed, mass, T, v_out)
!       v1 = v_out(1)
!       v1 = v1 + vwall(1)      ! convert back to lab frame
!       v2 = v_out(2)
!       v3 = v_out(3)
!
!       if (mypid.eq.1) then
!           write(69,*) v1
!           write(70,*) v2
!           write(71,*) v3
!       end if
!
!       else ! flag = 2
!
!           STRIKES UPPER WALL; USE MAXWELL ACCOMODATION
!           OR CERCIGIANI LAMPIS
!
!           vy_old = v2 ! remember pre-collision y velocity
!           alpha = atan((Ly1sys-Ly2sys)/Lx2sys)
!
!           TRANSFORM TO UPPER WALL COORDS
!
!           v1_temp = v1*cos(alpha) - v2*sin(alpha)
!           v2_temp = v2*cos(alpha) + v1*sin(alpha)
!           v2_temp = -v2_temp
!           v1 = v1_temp
!           v2 = v2_temp
!
!           MAXWELL ACCOMODATION
!
!           if (test.gt.pmaxwell) then ! use specular
!               v1 = v1
!               v2 = -v2
!               v3 = v3
!           else ! use thermal
!               v2 = sqrt(-log(1.-rand(iseed)))*mpv
!               phi = twopi*rand(iseed)
!               v_par11 = sqrt(-log(1.-rand(iseed))) * mpv
!               v1 = v_par11*sin(phi) + vwall(flag)
!               v3 = v_par11*cos(phi)
!           end if
!           v2 = -v2      ! 180 degree rotation
!
!       C-L METHOD
!
!       v_in(1) = v1
!       v_in(2) = v2
!       v_in(3) = v3
!       call cltrap(v_in, iseed, mass, T, v_out)
!       v1 = v_out(1)
!       v2 = v_out(2)
!       v3 = v_out(3)
!       v2 = -v2      ! 180 degree rotation

```

```

!
!       if (mypid.eq.1) then
!           write(72,*) v1
!           write(73,*) v2
!           write(74,*) v3
!       end if
!
!       TRANSFORM VELOCITIES FROM UPPER WALL COORDS
!
!       v1_temp = v1*cos(alpha) + v2*sin(alpha)
!       v2_temp = v2*cos(alpha) - v1*sin(alpha)
!       v1 = v1_temp
!       v2 = v2_temp
!       delv(2) = delv(2) - (v2 - vy_old) ! dv at upper wall
!
!   end if
!
!   Dtr = time of flight after leaving wall
!   MOVE PARTICLE FOR REMAINING TIME STEP
!
!   if (flag.eq.2) then
!       struck_already = 1
!       x_wall = xint      ! x-coord of strike
!
!       USE SLOPE TO DETERMINE HOW TO CALCULATE DTR
!
!       if (x.ne.x_old) then
!           m = (y-y_old)/(x-x_old)
!       else
!           m = 1.e3 ! arbitrary large number
!       end if
!
!       if (abs(m).gt.1.0) then
!           USE Y VALS
!           dtr = abs(time*(y-ywall(flag))/(y-y_old))
!       else
!           ! slope less than 1 or -1
!           USE X VALS
!           dtr = abs(time*(x-x_wall)/(x-x_old))
!       end if
!       dtb = time - dtr
!
!   else
!       ! flag = 1
!       ALWAYS USE Y-VALS IN THIS CASE
!       dtr = abs(time*(y-ywall(flag))/(y-y_old))
!       dtb = time - dtr
!       x_wall = x_old + vx_old*dtb
!   end if
!
!   MOVE PARTICLE OFF WALL WITH REMAINING TIME
!
!   y = ywall(flag) + v2*dtr      ! Position after leaving wall
!   x = x_wall + v1*dtr
!
!   if ((x_wall.lt.0.).or.(x_wall.gt.Lx2sys)) then
!       keep = 0 ! particle has left system before striking wall
!       npart_leave = npart_leave + 1
!       return
!   end if
!
!   strikes(flag) = strikes(flag) + 1
!   delv(flag) = delv(flag) + v1 - vx_old ! Record change in velocity
!
!   if (dtr.eq.0.) then

```

```

!
!      Particle is on the wall at the end of this time step and will be
!      moved off at the next time step
!
!      if (x.ge.0. .and. x.le.Lx2sys) then
!
!          keep = 1
!          return
!
!      else ! Particle exits entire system
!          keep = 0
!          npart_leave = npart_leave + 1
!          return
!      end if
!
!  end if
!
!  ITERATE TO CHECK FOR ANOTHER STRIKE OR EXIT
!  RESET OLD POSITIONS AND VELOCITIES TO VALUES AFTER STRIKE
!
!      x_old = x_wall      ! reset to wall strike position
!      y_old = ywall(flag) ! wall strike position
!      vx_old = vl         ! reset to post-collision velocity
!      time = dtr          ! time step reduced to remainder of previos step
!
!  repeat = repeat + 1
!
!  end do
!
!  write(*,*) 'Floundering in strikem'
!  write(*,*) 'Proc. Num = ',mypid
!  keep = 0 ! kill particle
!  return
!
!  end

```

```

real*8 function thermal(mpv,iseed)
!! Thermal - Function returns a velocity from M-B distribution
!! mpv is most probable velocity = sqrt(2*k*T/m)

    real*8 mpv,twopi
    real*8 rand
    integer*8 iseed
    data twopi/6.283185307/

    thermal = mpv*sqrt(-log(rand(iseed)))*cos(twopi*rand(iseed))

    return
end

```

INCLUDE FILES

"Common.txt"

```

common /block1/ x
common /block2/ y
common /block3/ v
common /block4/ npart
common /block5/ nxcell
common /block6/ ny cell
common /block7/ Xref
common /block8/ index
common /block9/ cell_n
common /block10/ eff_num
common /block11/ npart_leave

```

"param.txt"

```

parameter( npmax = 30000, ncmax = 200, nbmax = 11 )
parameter( nptmax = npmax/1 )
implicit real*8 (a-h,o-z)
implicit integer*8 (i-m)

```