1997

Simulations of microchannels with sinusoidally rough walls

Joseph Matthew Koning
San Jose State University

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

Recommended Citation

Koning, Joseph Matthew, "Simulations of microchannels with sinusoidally rough walls" (1997). *Master's Theses*. 1506.

DOI: <https://doi.org/10.31979/etd.ja97-9km7>

https://scholarworks.sjsu.edu/etd_theses/1506

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

SIMULATIONS OF MICROCHANNELS WITH
SINUSOIDALLY ROUGH WALLS

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

Joseph Matthew Koning

August, 1997

UMI Number: 1386207

UMI Microform 1386207

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

Joseph Matthew Koning

ALL RIGHTS RESERVED

APPROVED FOR THE DEPARTMENT OF PHYSICS

Dr. Alejandro Garcia

Dr. Patrick Hamill

Dr. David Hash

APPROVED FOR THE UNIVERSITY

ABSTRACT

SIMULATIONS OF MICROCHANNELS
WITH SINUSOIDALLY ROUGH WALLS

by Joseph Matthew Koning

This thesis simulates dilute gas flow through a microchannel, which is on the order of the mean free path (62 nm) in height and 5000 nm in length. Roughness is simulated on the microchannel walls by constructing the upper and lower walls of the 2D microchannel with sinusoids. Two solution methods are used to calculate the pressure profile of the dilute gas flow through the microchannel, solving the Reynolds equation and the direct simulation Monte Carlo method (DSMC). The non-linear partial-differential Reynolds equation describing the flow is solved using the Nigam and White method. The Reynolds equation includes corrections for the slip flow at the walls developed by Fukui and Kaneko. The DSMC method simulates the gas at the molecular level using stochastic algorithms for evaluating collisions and wall interactions. The DSMC method is solved using message passing methods on a massively parallel computer. Comparisons of pressure profiles obtained by the two methods generally show good agreement.

Acknowledgments

The effort, guidance and insight of many people contributed to this thesis. I would like to especially thank Dr. Alejandro Garcia, for his vast effort in educating me and serving as my advisor. I am grateful to the committee, for their time in reading this thesis and their valuable input. Also instrumental in the development of this thesis were Troy Wenski, who provided modified DSMC code for the simulation of the microchannel, and Susan Willner, who created some of the graphics.

I am indebted to my wife, Patricia Koning, for her unflagging support throughout my education and the writing of this thesis, and to my family, for their support and guidance.

I would also like to thank Lawrence Livermore National Laboratory for providing access to the Cray T3D, which was used to run the DSMC simulations.

This thesis would not have been possible without funding from the National Science Foundation.

Table of Contents

	<i>Page</i>
Acknowledgments	v
List of Tables and Figures	viii
1 Introduction	1
2 Reynolds Equation	5
2.1 Lubrication Theory	6
2.1.1 Continuum Flow	8
2.1.2 Slip Flow	9
2.2 Generalized Lubrication An Fukui Kaneko Slip Correction	10
2.3 Factored Implicit Solver	15
3 Direct Simulation Monte Carlo	19
3.1 Maxwell-Boltzmann Distribution	20
3.2 Collision Frequency	22
3.3 Wall Boundary	23
3.3.1 Specular and thermal Collisions	23
3.4 Flux Boundary	25
3.4.1 Velocity Distribution Across a Boundary	26
3.4.2 Acceptance-Rejection	28
3.4.3 Poisson Distribution	29
3.5 DSMC Method	30
3.6 DSMC Theory	35
4 Parallel Processing	42
5 Simulation Results	46
5.1 Validation	47
5.1.1 Factored Implicit Solver	47
5.1.2 Equilibrium Verification	49
5.2 Simulation Results	51
5.2.1 Flat Wall Simulation	51
5.2.2 Wall Separation Variation	53
5.2.3 Phase Variation	57
5.2.4 Amplitude Variation	62
5.2.5 Constrictions	65

Table of Contents (cont.)

	<i>Page</i>
6 Discussion and Conclusion	68
Bibliography	70
Appendix A	72
Appendix B	116

List of Tables

<i>Table</i>		<i>Page</i>
Table 3.1	Outline of the DSMC algorithm in the <code>spdsmc.f</code> steps are attributed to the appropriate worker.	36

List of Figures

<i>Figure</i>		<i>Page</i>
Figure 1.1	Geometry of the microchannel.	2
Figure 2.1	Representation of the microchannel with position and velocity legends.	5
Figure 3.1	Plot of the biased Maxwell distribution equation, of the normal component, for argon at STP.	25
Figure 3.2	Plane AB represents the reservoir-system interface. Particles within the cylinder may cross dS within the time dt .	27
Figure 3.3	The system domain is divided into many cells. A processor will initialize particles into each cell.	31
Figure 4.1	The system domain is separated into columns. The arrows represent the communication system used by the DSMC. program	44
Figure 5.1	Verification of the Factored Implicit solver against a pressure profile for a slider air bearing. $Kn_0=1.912$.	48
Figure 5.2	Verification of the Factored Implicit solver against a pressure profile for a slider air bearing. $Kn_0=4.780$.	49
Figure 5.3	One-dimensional pressure along the microchannel. Pressure is deviating around atmospheric pressure as expected.	50
Figure 5.4a	Pressure profile for a microchannel with flat walls.	52

List of Figures (cont.)

<i>Figure</i>		<i>Page</i>
Figure 5.4b	System geometry for Figure 5.4a.	52
Figure 5.5a	Pressure profile for a microchannel with two periods , $h_0=80$ nm, $\phi=0^\circ$ and $A=20$ nm. Data includes continuum, first order slip corrected, Fukui and Kaneko and DSMC solution.	54
Figure 5.5b	System geometry for Figure 5.5a.	54
Figure 5.6a	Pressure profile for a microchannel with $h_0=200$ nm, two periods, $\phi=0^\circ$ and $A=20$ nm.	55
Figure 5.6b	System geometry for Figure 5.6a.	56
Figure 5.7a	Pressure profile for a microchannel with $h_0=80$ nm, two periods, $\phi=90^\circ$ and $A=20$ nm.	57
Figure 5.7b	System geometry for Figure 5.7a.	58
Figure 5.8a	Pressure profile for a microchannel with $h_0=200$ nm, two periods, $\phi=90^\circ$ and $A=20$ nm.	58
Figure 5.8b	System geometry for Figure 5.8a.	59
Figure 5.9a	Pressure profile for a microchannel with $h_0=80$ nm, two periods, $\phi=180^\circ$ and $A=20$ nm.	60
Figure 5.9b	System geometry for Figure 5.9a.	60
Figure 5.10a	Pressure profile for a microchannel with $h_0=200$ nm, two periods, $\phi=180^\circ$ and $A=20$ nm.	61
Figure 5.10b	System geometry for Figure 5.10a.	
Figure 5.11a	Pressure profile for a microchannel with $h_0=80$ nm, two periods, $\phi=180^\circ$ and $A=30$ nm.	62

List of Figures (cont.)

<i>Figure</i>		<i>Page</i>
Figure 5.11b	System geometry for Figure 5.11a.	63
Figure 5.12a	Pressure profile for a microchannel with $h_0=80$ nm, two periods, $\phi=180^\circ$ and $A=39$ nm.	64
Figure 5.12b	System geometry for Figure 5.12a.	64
Figure 5.13a	Pressure profile for a microchannel with $h_0=80$ nm, two periods, $\phi=180^\circ$ and $A=39.8$ nm.	66
Figure 5.13b	System geometry for Figure 5.13a.	66
Figure 5.14a	Pressure profile for a microchannel with $h_0=80$ nm, six periods, $\phi=0^\circ$ and $A=200$ nm.	67
Figure 5.14b	System geometry for Figure 5.14a.	67

Chapter 1 Introduction

The topic of this thesis is the simulation of the flow of a dilute gas through a microchannel. Microchannels are of interest in the study and design of Micro-Electrical-Mechanical Systems (MEMS). MEMS are miniature structures etched in the surface of silicon wafers. They are manufactured using the same process as integrated circuits. MEMS applications include accelerometers, microvalves, micrometers, micro pressure sensors and micro shear stress sensors [16].

The dilute gas flowing through the microchannel is argon, with a mean free path on the order of the microchannel height. Continuum hydrodynamics is incorrect at these small scales. The discrete particle nature of the gas becomes important at these scales and must be included. The mean free path of a gas (λ) is the length a molecule of the substance can travel before interacting with another molecule. This thesis uses the hard sphere model, which treats all particles as spheres with a diameter d and with interactions like billiard balls. The mean free path in this model is then how far each sphere can travel before coming into contact with another sphere.

In kinetic theory and molecular gas dynamics, the Knudsen number is one of the most useful parameters. The Knudsen number can be used to categorize various types of fluid flows and is defined as $Kn = \frac{\lambda}{h}$, where h is the characteristic length scale of the system, the channel spacing in this case. Some categories of Knudsen numbers include:

- I. $Kn \ll 0.1$: Continuum Flow
- II. $0.1 < Kn < 10$: Transition Flow
- III. $10 \ll Kn$: Free Flow

In this thesis, flows of the second type will be examined.

The microchannel considered is constructed using two sinusoids for the top and bottom walls of the microchannel, Figure 1.1. The two sinusoids are separated by a distance on the order of the mean free path of the dilute gas. The length of the channel is about 100 times the channel height. Flow through the channel is provided by a pressure difference between the left and right sides of the microchannel.

Figure 1.1 The geometry of the microchannel

Sinusoids are used to approximate wall roughness through the channel. The top and bottom wall sinusoids differ only in phase. Several system configurations are examined, including varying channel height, amplitudes and phase differences.

In these simulations the flow is of the Poiseuille type, that is pressure-generated. Other types such as Couette (or shear) flow manifested by moving surfaces and thermal creep, flow induced by temperature gradients along a surface, do not appear in the simulation. The Poiseuille flow is provided by a difference in pressure of 5 to 1 atmospheres at the inlet and outlet respectively.

Two solution methods are used to solve for the pressure flow in the microchannel: solving the Reynolds equation and using the direct simulation Monte Carlo method (DSMC). The Reynolds equation is developed from a more general flow equation called the Navier-Stokes equation. The second method, DSMC, simulates flows on the particle level and may be considered a Monte Carlo method for solving the Boltzmann equation.

The Reynolds equation is a non-linear partial differential equation that can be solved using a specialized PDE solver. Corrections are necessary because of the limitation of the uncorrected Reynolds equations to low Knudsen numbers. There are several approximations used to correct for peculiar effects found in microchannel flow. One such peculiarity is the slip flow at the wall of the microchannel. The Fukui-Kaneko slip correction is based on the Boltzmann equation and is valid for all Knudsen numbers. The corrections show a marked improvement in the pressure profile of the flow through the channel.

The DSMC method is a particle simulation method that uses the vast numbers of molecules within the flow to its advantage. In Molecular Dynamics (MD) simulations, all particles in the system must be tracked. In the DSMC method only a select number of the particles are tracked, many orders of magnitude less than MD. Each simulation particle in the DSMC method represents a much larger group of real particles. This method remains valid for surprisingly low total numbers of representative particles.

The next chapter develops the Reynolds equation and the factored implicit method for solving the Reynolds equation. Chapter 3 explains the DSMC method and its implementation. The parallel methods used in the DSMC simulation are discussed in Chapter 4. The simulation results for the microchannel problem are presented in Chapter 5. The final chapter includes a summary of the results and the conclusion.

Chapter 2 Reynolds Equation

The Reynolds equation can be used to determine the pressure profile of a microchannel [9]. This equation is derived from a more general equation, the Navier-Stokes equation. The Reynolds equation is a non-linear partial differential equation that can be solved numerically using a factored implicit scheme developed by Nigam and White [18]. The following section will outline the derivation of the Reynolds equation. The final Reynolds equation will include corrections for the slip flow determined by Fukui and Kaneko.

Figure 2.1 Representation of the microchannel with position and velocity legends.

Figure 2.1 shows the three-dimensional representation of the microchannel. The microchannel height and width are the quantities h_0 and l , respectively. The position and velocity legends are included for the notation used in this thesis.

2.1 Lubrication Theory

The Reynolds equation is derived through several approximations from the more general equations of hydrodynamics [12]. Consider the equation of continuity

$$\frac{\partial \rho}{\partial t} = -\frac{\partial}{\partial x_i}(\rho u_i) \quad (2-1)$$

and the Navier-Stokes equation.

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

where ρ is the mass density, i, j , and k are the indices for the Cartesian components of the flow velocity and position, μ and ξ are the shear and bulk viscosities and B is the body force. This equation is based on the forces acting on a fluid including inertial and viscous forces, external body force and reversible forces due to pressure gradients. Along with the force equations, the equation of continuity and the equation of state for an ideal gas are used to complete the Navier-Stokes equation.

Two main assumptions, called the “thin film assumptions,” are made in the derivation of the Reynolds equation:

1. The channel height h is much smaller than the channel length l .
2. Inertial forces are negligible.

The first assumption indicates that velocity of the flow along the channel (x -direction) is much greater than in the y or z directions. From this assumption the velocity in the z -direction (w) is set to 0. Assumption 2 indicates that only the viscous forces and pressure gradients need to be included. Along with these assumptions, we will only consider

steady flow $\frac{\partial u}{\partial t} = 0$ and negligible body force $B = 0$. The gas flowing through the

microchannel is assumed to be ideal, giving the ideal gas equation:

$$p = \rho RT$$

where R is the gas constant and T is the temperature of the gas. An ideal gas has zero bulk viscosity. Combining all of the assumptions and simplifications gives:

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

$$\frac{\partial p}{\partial y} = 0 .$$

The equation of continuity, equation (2-1), becomes:

$$\frac{\partial}{\partial x}(\rho u) = 0 .$$

The average equation of continuity is obtained by integrating between the top and bottom walls of the microchannel,

$$\frac{\partial}{\partial x} \left(\int_0^h \rho u dy \right) = 0 . \quad (2-3)$$

Integrating equation (2-2) twice gives the velocity profile $u(y)$. The Reynolds equation is obtained by putting the velocity profile obtained from equation (2-2) into equation (2-3) and integrating.

2.1.1 Continuum Flow

To derive the continuum Reynolds equation the boundary conditions are set to:

$$\begin{aligned} u|_{y=h} &= 0 \\ u|_{y=0} &= 0. \end{aligned}$$

The solution to (2-2) with these boundary conditions is:

$$u = -\frac{1}{2\mu} \frac{\partial p}{\partial x} (h - y) y . \quad (2-4)$$

The right hand term is a flow due to pressure differences in the system, the Poiseuille flow. Substituting equation (2-4) for u into equation (2-3) gives:

$$\frac{d}{dx} \left(\rho h^3 \frac{dp}{dx} \right) = 0 .$$

Under isothermal conditions for compressible fluids $\rho \sim p$, giving the continuum flow Reynolds equation in one dimension:

$$\frac{d}{dx} \left(p h^3 \frac{dp}{dx} \right) = 0 .$$

The Reynolds equations in this thesis will all be in nondimensional form. The above equation becomes:

$$\frac{d}{dX} \left(PH^3 \frac{dP}{dX} \right) = 0 \quad (2-5)$$

where $X = \frac{x}{l}$, $P = \frac{p}{p_a}$, $H = \frac{h}{h_0}$; l is the length of the microchannel, p_a is atmospheric pressure and h_0 is the height at the exit of the microchannel. Because of the choice of boundary conditions, no Couette (shear) flow term is present in equation (2-5). The continuum Reynolds equation corresponds to the case where Kn is very small ($Kn \ll 1$). The dimensions of the system are much larger than the dimensions of individual particles. for example in a propane gas line.

2.1.2 Slip Flow

When $Kn > 0.1$, that is within the slip region, the boundary conditions must include corrections for the velocity discrepancy at the boundaries. The characteristic dimension in these cases is on the order of the mean free path. Individual particle behavior must be taken into account. Maxwell was the first to theorize that the velocity of a gas does not match a moving wall's velocity. By shifting the frame of reference, this can be generalized to the case where the wall's velocity is zero. The velocity gradient of the gas above the wall will not go to zero at the wall interface. Due to this, a slip correction is required in the Reynolds equation and was included by Burgdorfer [3]:

$$u|_{y=h} = -a\lambda \frac{du}{dy}$$

$$u|_{y=0} = a\lambda \frac{du}{dy}.$$

The quantity $a\lambda$ in these boundary conditions is the slip length. The quantity a is defined by $a = \frac{2-\alpha}{\alpha}$, where α is the accommodation coefficient. In this thesis α is equal to one for thermal walls giving $a = 1$. When these boundary conditions are substituted into equation (2-3), a Reynolds equation with first order slip correction is obtained:

$$\frac{d}{dX} \left(PH^3 (1 + 6a\lambda Kn) \frac{dP}{dX} \right) = 0 .$$

This first order slip correction improves the result but is still inaccurate for most situations. The inaccuracy can be related to the basis of the derivation. The Burgdorfer first order slip correction is based on the small Kn assumption. Going to higher Kn numbers ($Kn \approx 1$) leads to inaccuracies.

2.2 Generalized Lubrication Equation and Fukui-Kaneko Slip Correction

Fukui and Kaneko dealt with the inaccuracy of the first order slip correction by deriving the generalized lubrication equation [5][6]. This lubrication equation is based on the Boltzmann equation:

$$\frac{\partial f(X, \xi, t)}{\partial t} + \xi_i \frac{\partial f(X_i, \xi_i, t)}{\partial X_i} = (\text{collision term}) \quad (2-6)$$

and is valid for all Knudsen numbers. The quantity f in the Boltzmann equation is the velocity distribution function and the ξ 's are the velocities of the particles. In an equilibrium state the Maxwell Distribution is the solution to the Boltzmann equation.

$$f_e = \frac{\rho}{(2\pi RT)^{3/2}} e^{-\left(\frac{(\xi_i^2 - v_i^2)}{2RT}\right)}.$$

The density and temperature in the Maxwell distribution are constant values. If the collision term, right hand side of (2-6), of the Boltzmann equation is substituted with a simple collision term proportional to $(f_e - f)$, the Bhatnagar, Gross, Krook (BGK) model[1] is obtained:

$$\frac{\partial f(X, \xi, t)}{\partial t} + \xi_i \frac{\partial f(X_i, \xi_i, t)}{\partial X_i} = A(f_e - f)$$

where A is the characteristic collision frequency.

Using the assumption that the velocity distribution through the microchannel is a perturbation to the static isotropic equilibrium state, the BGK model can be linearized. The flow velocities in the channel must be small compared to the molecular thermal velocities. This means that the Mach number, the ratio of the flow speed to the speed of sound, must be much smaller than one and the temperature variations in the flow must be small. The nondimensional, linearized BGK model is given by:

$$\begin{aligned} \frac{h_0}{l} \varsigma_x \frac{\partial \phi}{\partial X} + \frac{h_0}{l} \varsigma_y \frac{\partial \phi}{\partial Y} + \varsigma_z \frac{\partial \phi}{\partial Z} = \\ \frac{\sqrt{\pi}}{2Kn_0} \left[-\phi + \omega + 2(\varsigma_x V_x + \varsigma_y V_y + \varsigma_z V_z) + \left(\varsigma_x^2 + \varsigma_y^2 + \varsigma_z^2 - \frac{3}{2} \right) \tau \right]. \end{aligned}$$

The quantities ϕ , ω , and τ are the perturbed velocity distribution, density and temperature respectively. X , Y , Z are the nondimensional coordinates. The ς 's are the nondimensional molecular velocities, and V_x , V_y , V_z are the macroscopic velocities.

The value Kn_0 is the characteristic Knudsen number defined by $Kn_0 = \frac{\lambda_a}{h_0}$. This value is usually derived from the height at the exit of the channel and the mean free path at atmospheric pressure.

The following assumptions are made to obtain the flow velocities: the thin film assumptions described above and the temperatures at the boundaries are constant everywhere. Applying these assumptions gives:

$$\frac{h_0}{l} \varsigma_x \frac{\partial \phi}{\partial X} + \frac{h_0}{l} \varsigma_y \frac{\partial \phi}{\partial Y} = \frac{\sqrt{\pi}}{2Kn_0} [-\phi + \omega + 2\varsigma_x V_x] . \quad (2-7)$$

Consider the following equation as a solution to equation (2-7):

$$\phi = \left(\frac{l}{h_0} X \right) \phi_0 + (\varsigma_x) \phi_1(Y, \varsigma_x) .$$

Assuming the walls can be treated as thermal walls (see Chapter 3), the quantity

$\phi_0 = \beta$, where $\beta = \frac{h_0}{l} \frac{dP}{dX}$ is the nondimensional pressure gradient. The equation for ϕ_1

becomes:

$$\varsigma_y \frac{\partial \phi_1}{\partial Y} = \frac{\sqrt{\pi}}{2Kn_0} [-\phi_1 + 2V_x] - \beta . \quad (2-8)$$

The boundary conditions for the quantity ϕ_1 are:

$$\begin{aligned}\phi_1|_{Y=0} &= 2V_0 \quad \zeta_Y > 0 \\ \phi_1|_{Y=H} &= 0 \quad \zeta_Y < 0.\end{aligned}$$

The solutions to equation (2-8) are:

$$\begin{aligned}\phi_1|_{\zeta_Y > 0} &= 2V_0 e^{\frac{\sqrt{\pi}Y}{2Kn_0\zeta_Y}} + \frac{1}{\zeta_Y} e^{\frac{\sqrt{\pi}Y}{2Kn_0\zeta_Y}} \int_0^Y \left(\frac{\sqrt{\pi}V_x(Y')}{Kn_0} - \beta \right) e^{\frac{\sqrt{\pi}Y'}{2Kn_0\zeta_Y}} dY' \\ \phi_1|_{\zeta_Y < 0} &= \frac{1}{\zeta_Y} e^{\frac{\sqrt{\pi}Y}{2Kn_0\zeta_Y}} \int_H^Y \left(\frac{\sqrt{\pi}V_x(Y')}{Kn_0} - \beta \right) e^{\frac{\sqrt{\pi}Y'}{2Kn_0\zeta_Y}} dY' .\end{aligned}$$

The flow velocity in the flow (x) direction, V_x , can be obtained from equation (2-8). The relation is:

$$V_x(Y) = \frac{Kn_0\beta}{\sqrt{\pi}} (1 - \psi_p)$$

where

$$\psi_p = 1 + \frac{1}{2Kn_0} \int_0^H T_{-1} \left(\frac{\sqrt{\pi}}{2Kn_0} |Y - Y'| \right) \psi_p dY'$$

and

$$T_n(x) = \int_0^\infty t^n e^{\left(-t^2 - \frac{x}{t}\right)} dt$$

is the Abramowitz function. Solving the integral equation for ψ_p gives the plane Poiseuille flow velocity V_{xp} . Since the Couette and thermal creep terms were absent in the microchannel application, the Poiseuille flow velocity is the total flow velocity in the x -direction $V_x(Y) = V_{xp}$.

Using the above derivation, a generalized Reynolds lubrication equation can be developed. To obtain this solution it is necessary to estimate the mass flow rate, q , in the

channel and to use the mass flow conservation law. Because the flow speed can be expressed as a sum of the Poiseuille, Couette and thermal creep flows, so can the flow rate. This gives: $q = q_p + q_c + q_t$, where q_p is the Poiseuille flow rate, q_c the Couette flow rate and q_t the thermal creep flow rate. In the microchannel there is no Couette flow and the thermal creep effects will be neglected. This leaves the flow rate $q = q_p$.

The nondimensional Poiseuille flow rate term is defined by:

$$Q_p(Kn) = \frac{-q_p}{h^2 \frac{dp}{dx}} \sqrt{2RT_0} .$$

The relative flow rate is obtained by relating Q_p to the continuum flow rate Q_{con} :

$$\bar{Q}_p(Kn) = \frac{Q_p(Kn)}{Q_{con}} .$$

The relative continuum Poiseuille flow rate is:

$$\bar{Q}_p(Kn) \equiv 1 .$$

The generalized Reynolds lubrication equation for Poiseuille type flow then becomes:

$$\frac{d}{dX} \left(\bar{Q}_p(Kn) PH^3 \frac{dP}{dX} \right) = 0 .$$

For fully accommodating surfaces, $a = 1$, the value $\bar{Q}_p(Kn)$ has been estimated by Robert as [15]:

$$\bar{Q}_p(Kn) = 1 + 6AKn + \frac{12}{\pi} Kn \log(1 + BKn) \quad (2-31)$$

where $A = 1.318,889$, $B = 0.387,361$ are the constants selected to give the best fit to the data tabulated by Fukui and Kaneko.

The Reynolds equation only uses the height variation to define the geometry of the channel. A system with flat walls will have the same height variation as a system with in-phase sinusoidal walls. This leads to the same predicted profile with different system geometries. The Reynolds and generalized lubrication equations derived above both assumed that the flow velocity in the thickness direction (y -direction) is negligible. With the system geometry in this thesis, this may not be the case. The flow velocities in the y -direction can be affected by the undulating nature of the wall geometries.

2.2 Factored Implicit Solver

In order to solve the nondimensional Reynolds equation for complex geometries, a numerical scheme must be developed. The non-linear nature of the Reynolds equation requires a specialized numerical treatment. This derivation is based on the factored implicit scheme developed by Nigam and White [18].

Starting with the nondimensional Reynolds equation:

$$\frac{\partial}{\partial X} \left(\overline{Q}_p(Kn) PH^3 \frac{\partial P}{\partial X} \right) = \frac{12lv_s\mu}{p_a h_o^2} \frac{\partial}{\partial T} (PH) .$$

The time dependence is included on the right hand side. This time dependence will be used in numerically relaxing the system to steady state. If we set $Z = PH$ and use the Robert approximation for $\overline{Q}_p(Kn)$, then this equation becomes:

$$\frac{\partial}{\partial X} \left[HZ \frac{\partial Z}{\partial X} - Z^2 \frac{\partial H}{\partial X} + \overline{Q_p}(Kn) \left(H \frac{\partial Z}{\partial X} - Z \frac{\partial H}{\partial X} \right) \right] = \frac{12lv_s\mu}{\rho_a h_0^2} \frac{\partial Z}{\partial T} .$$

Defining:

$$F = HZZ_x - Z^2 H_x + \overline{Q_p}(Kn)(HZ_x - ZH_x)$$

where $Z_x = \frac{\partial Z}{\partial X}$.

The Reynolds equation is now:

$$F_x = \gamma Z_t$$

where $\gamma = \frac{12lv_s\mu}{\rho_a h_0^2}$.

Using the centered Taylor expansion to expand Z and Z_t :

$$Z^{n+1} = Z^n + \frac{\Delta t}{2} (Z_t^n + Z_t^{n+1}) + O(\Delta t^3)$$

where $Z^n = Z(t = n\Delta t)$.

We can substitute for Z_t giving:

$$Z^{n+1} = Z^n + \frac{\Delta t}{2\gamma} (F_x^n + F_x^{n+1}) + O(\Delta t^3) . \quad (2-9)$$

This equation is not suitable for use in an implicit solver because F is a nonlinear function of Z . The equation for F^{n+1} is obtained by expanding F around the current time level.

$$F^{n+1} = F^n + (H^{n+1} - H^n) \frac{\partial F^n}{\partial H} + (H_{\tau}^{n+1} - H_{\tau}^n) \frac{\partial F^n}{\partial H_{\tau}} \\ + (Z^{n+1} - Z^n) \frac{\partial F^n}{\partial Z} + (Z_{\tau}^{n+1} - Z_{\tau}^n) \frac{\partial F^n}{\partial Z_{\tau}}$$

or:

$$F^{n+1} = F^n + (H^{n+1} - H^n)A + (H_{\tau}^{n+1} - H_{\tau}^n)B + \\ (Z^{n+1} - Z^n)C + (Z_{\tau}^{n+1} - Z_{\tau}^n)D$$

where:

$$A = \frac{\partial F^n}{\partial H} = ZZ_{\tau} + \overline{Q_p}(Kn)Z_{\tau} \\ B = \frac{\partial F^n}{\partial H_{\tau}} = -Z^2 - \overline{Q_p}(Kn)Z \\ C = \frac{\partial F^n}{\partial Z} = HZ_{\tau} - 2ZH_{\tau} - \overline{Q_p}(Kn)H_{\tau} \\ D = \frac{\partial F^n}{\partial Z_{\tau}} = HZ + \overline{Q_p}(Kn)H.$$

Substituting into (2-9):

$$Z^{n+1} = Z^n + \frac{\Delta t}{2\gamma} F_{\tau}^n + \frac{\Delta t}{2\gamma} \\ \left[F_{\tau}^n + (H^{n+1} - H^n)A + (H_{\tau}^{n+1} - H_{\tau}^n)B + (Z^{n+1} - Z^n)C + (Z_{\tau}^{n+1} - Z_{\tau}^n)D \right]$$

and grouping:

$$\left[1 - \frac{\Delta t}{2\gamma} \frac{\partial}{\partial x} C - \frac{\Delta t}{2\gamma} \frac{\partial}{\partial x} D \frac{\partial}{\partial x} \right] Z^{n+1} = \left[1 - \frac{\Delta t}{2\gamma} \frac{\partial}{\partial x} C - \frac{\Delta t}{2\gamma} \frac{\partial}{\partial x} D \frac{\partial}{\partial x} \right] Z^n + \\ \frac{\Delta t}{\gamma} F_{\tau}^n + \frac{\Delta t}{2\gamma} \left[\frac{\partial}{\partial x} (H^{n+1} - H^n)A + \frac{\partial}{\partial x} (H_{\tau}^{n+1} - H_{\tau}^n)B \right].$$

The equation can be rewritten as:

$$[1 - L]Z^{n+1} = [1 - L]Z^n + \phi .$$

Reorganizing we have:

$$Z^{n+1} = Z^n + [1 - L]^{-1} \phi . \quad (2-10)$$

where:

$$\phi = \frac{\Delta t}{2\gamma} \left[\frac{\partial}{\partial x} (H^{n+1} - H^n) A + \frac{\partial}{\partial x} (H_t^{n+1} - H_t^n) B \right]$$

and

$$L = \left[1 - \frac{\Delta t}{2\gamma} \frac{\partial}{\partial x} C - \frac{\Delta t}{2\gamma} \frac{\partial}{\partial x} D \frac{\partial}{\partial x} \right] . \quad (2-11)$$

The original scheme developed by Nigam and White used an uneven grid spacing when calculating the coefficients A, B, C and D. This was used to capture the steep pressure gradients involved with some geometries. This uneven grid spacing was not employed for the system geometries researched in this thesis.

The solution method begins by discretizing equation (2-11) into matrix form. The quantity $[1-L]^{-1}$ is then evaluated as a matrix inverse and multiplied by the quantity ϕ .

The value for Z in equation (2-10) is obtained by iteration until the solution relaxes to a steady state. Once the value of Z is known, the pressure can be extracted by $P = \frac{Z}{H}$,

where H is the known normalized height through the microchannel.

Chapter 3 Direct Simulation Monte Carlo Method

The simulation of the flow of dilute gas through a microchannel in Molecular Dynamics (MD) is accomplished by exactly computing the motion of all the particles within the simulation. To maintain information on the particles, their positions and velocities are constantly monitored $\{\mathbf{r}_i, \mathbf{v}_i\}$ $i = 1, \dots, N$. This entails a lot of computation because each particle's trajectory depends on all of the other particles within the simulation. This gives $\sim N \times N$ operations and makes the MD simulation $O(N^2)$. A collision in the MD simulation can take several timesteps to evaluate.

The DSMC method [2][7] capitalizes on the great number of particles instead of being hindered by them. It does so by treating each simulation particle as a representative of a larger collection of similarly attributed real molecules. This method assumes all particles interact with each other through binary collisions. Each particle's velocity is allowed to remain unchanged unless encountering another particle or boundary. The collisions are selected at random using statistical methods. The interactions are limited to bi-particle collisions. No intra-particle force interactions are included. This limits the simulation to that of an ideal gas and the DSMC method reduces to computation of $O(N)$. In the DSMC method the timestep is on the order of the mean free time, the time between collisions. For MD, the timestep is on the order of the interaction time, the duration of the collision.

Before detailing the DSMC method itself, it is useful to review some kinetic theory. The gas flowing through the microchannel is treated as a dilute gas. This allows us to assume the two body interactions needed for the ideal gas in the DSMC method. The volume of the microchannel is approximately 2 cubic microns at STP so there will be approximately 5×10^7 particles in the system. The DSMC method reduces the amount of computation by using statistical methods to reduce the number of simulation particles.

3.1 Maxwell-Boltzmann Distribution

While the gas in the DSMC method is usually not at thermodynamic equilibrium, various thermodynamic properties (e.g., temperature) can be used out of equilibrium. The properties of equilibrium statistical mechanics are used to develop the DSMC method. The Maxwell-Boltzmann distribution gives a velocity distribution for an ideal gas in an equilibrium state [11]. Assuming a dilute gas so that the interaction between particles is very weak; the total energy of the gas equals the sum of the energies of each particle.

The energy of the particle is taken to be translational energy, $E = \frac{|\mathbf{p}|^2}{2m}$. If the particle's location lies somewhere between $\mathbf{r}$ and $\mathbf{r} + d\mathbf{r}$ and its momentum lies in the range between $\mathbf{p}$ and $\mathbf{p} + d\mathbf{p}$, the volume of phase space corresponding to this region is:

$$d^3r d^3p \equiv (dx dy dz)(dp_x dp_y dp_z) .$$

The probability that the particle has a position and momentum lying in the ranges specified above is:

$P(\mathbf{r}, \mathbf{p}) d^3\mathbf{r} d^3\mathbf{p}$ = (Volume fraction in phase space) * (probability that

$$\text{the molecule is in a cell}) \approx \frac{d^3\mathbf{r} d^3\mathbf{p}}{V_{tot}} * e^{-\beta(\frac{p^2}{2m})}$$

where the second term comes from the canonical distribution and $\beta = \frac{1}{kT}$; k is

Boltzmann's constant. If the probability density does not depend on the position of the particle then:

$$P(\mathbf{p}) d^3\mathbf{p} = \int_{\mathbf{r}} P(\mathbf{r}, \mathbf{p}) d^3\mathbf{r} d^3\mathbf{p} = A e^{-\beta(\frac{p^2}{2m})} d^3\mathbf{p} .$$

The velocity distribution can be obtained from this momentum distribution giving:

$$P(\mathbf{v}) d\mathbf{v} = C v^2 e^{-\beta(\frac{1}{2}mv^2)} dv \sin(\theta) d\theta d\phi .$$

Normalizing this function to determine A gives:

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

This velocity distribution can be used to obtain representative velocities for the ideal gas:

the average velocity

$$\langle v \rangle = \frac{2\sqrt{2}}{\pi} \sqrt{\frac{kT}{m}} \quad (3-1)$$

root mean square velocity $v_{rms} = \sqrt{3} \sqrt{\frac{kT}{m}}$, most probable velocity $v_{mp} = \sqrt{2} \sqrt{\frac{kT}{m}}$, and

average kinetic energy $\langle K \rangle = \frac{3}{2} kT$.

3.2 Collision Frequency

The mean free path has been qualitatively discussed in Chapter 1. This distance is the average length a particle will travel before encountering another particle. The collision frequency (f) of the gas is the average number of collisions per particle per unit time. Both the mean free path (λ) and the collision frequency are related to the average velocity $\langle v \rangle$ by $\lambda = \frac{\langle v \rangle}{f}$. To determine the collision frequency, picture a cylindrical tube traced by a particle of diameter d as it travels through space. The volume of this tube will be $V_{tube} = \pi d^2 l = \pi d^2 \langle v_r \rangle t_0$, where $\langle v_r \rangle$ is the average relative velocity of the particle. The particle will travel $l = \langle v_r \rangle t_0$ in time t_0 . The collision frequency is determined by how many particles are in this tube divided by the time t_0 . The gas is assumed to be homogeneous, therefore the number of particles in the tube is: $M_{tube} = \frac{N}{V} \pi d^2 \langle v_r \rangle t_0$. The quantity N is the number of gas molecules in a volume V . The collision frequency becomes:

$$f = \frac{\langle M_{tube} \rangle}{t_0} = \frac{N}{V} \pi d^2 \langle v_r \rangle .$$

The collision frequency assumes that only one particle moves, this must be corrected by setting:

$$\langle v_r \rangle = \langle |v_1 - v_2| \rangle = \iint dv_1 dv_2 |v_1 - v_2| P(v_1) P(v_2) = \frac{4}{\sqrt{\pi}} \sqrt{\frac{kT}{m}}$$

The explicit value for the mean free path $\lambda = \frac{\langle v \rangle}{f}$ can now be evaluated:

$$\lambda = \frac{\frac{2\sqrt{2}}{\sqrt{\pi}} \sqrt{\frac{kT}{m}}}{\frac{N}{V} \pi d^2 \frac{4}{\sqrt{\pi}} \sqrt{\frac{kT}{m}}} = \frac{V}{\sqrt{2} N \pi d^2}.$$

3.3 Wall Boundary

The interactions of particles with the microchannel boundaries must be considered. Two types of boundaries are present in the channel. The upper and lower walls represented by sinusoids and the left and right flux boundaries. Simple models are used to approximate these boundary interactions.

3.3.1 Specular and Thermal Collisions

Two simple methods are used to approximate the boundary interaction at the top and bottom walls of the microchannel. The specular interaction is akin to a light beam reflected from a mirror. The specular interaction is only used when verifying the program and not in the main simulation. The wall of the microchannel is more massive than the particle so in a specular elastic collision, the normal component of the particle's velocity is changed in sign while the tangential components of the particle's velocity remain unchanged.

A thermal interaction with the wall treats the surface as a thermal reservoir. The particle is assumed to interact sufficiently with the wall so that it becomes thermalized. The particles will have their energy modified so that the post interaction velocities will give the equilibrium distribution of a gas with the temperature of the wall. Once the particle has interacted with the wall, the velocities can be obtained from the Maxwell-Boltzmann distribution. The velocities parallel to the wall will be selected using the normal Maxwell-Boltzmann probability distribution:

$$P(u) = \frac{\beta}{\sqrt{\pi}} e^{-\beta^2 u^2}; P(w) = \frac{\beta}{\sqrt{\pi}} e^{-\beta^2 w^2}; \beta = \sqrt{\frac{m}{2kT_w}}$$

where T_w is the temperature of the wall. The perpendicular velocity is chosen with a biased Maxwell distribution:

$$P(v) = \pm \frac{1}{2\beta^2} v e^{-\beta^2 v^2}.$$

The plus and minus sign in the probability function correspond to surfaces normal in the positive and negative directions respectively. Figure 3.1 shows the biased Maxwell distribution.

The normal to the sinusoidal wall at the interaction point must be calculated to determine the final velocity of the scattered particle. This is accomplished by dividing the path, from initial to final position, into many smaller steps. The intersection of the particle's path and the sinusoidal wall is found by iterating along the path until a collision occurs. The point of interaction with the wall is determined by bisection. The interaction

Figure 3.1 Plot of the biased Maxwell distribution, of the normal component, for argon at STP.

point is used to determine the normal to the surface. The particle's final velocity is then rotated from the wall normal coordinate system to the lab frame coordinates.

Maxwell accommodation is another model for particle-surface interaction. In this model there are given probabilities for a specular or thermal interaction. The probability for a thermal interaction is denoted by the value α , a specular interaction will therefore have a probability of occurring, $1-\alpha$. In this thesis α is set to 1, giving all thermal particle-surface interactions.

3.4 Flux Boundary

The particle interactions at the entrance and exit of the microchannel are handled by a flux boundary. These boundaries consist of open space where particles move in and out without hindrance. The regions outside the boundary are fluid reservoirs with

average fluid velocity and temperature and a constant mass density. In this thesis the reservoir on the left has a higher pressure and therefore a higher density, than the fluid reservoir on the right. This means the reservoir on the left will inject more particles into the channel than the one on the right. To implement the flux boundary, the number and velocity distribution of particles crossing into the channel per timestep must be derived.

3.4.1 Velocity Distribution Across a Boundary

Beginning with the particles present in the reservoir on the left, the wall AB forms a boundary between the reservoir and the system (see Figure 3.2). A particle within the reservoir will have a velocity $\mathbf{c} = (u, v, w)$. A function f is defined by $f(\mathbf{c})d\mathbf{c}dV = n$, 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 the particle in the fluid frame is $\mathbf{C} = \mathbf{c} - \mathbf{c}_0$, where $\mathbf{c}_0$ is the average velocity of the fluid. The velocity of the particle in the AB plane frame is:

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

To determine the number of particles crossing the area dS on the plane AB we will use the function above. For this case the equation becomes:

$$f(\mathbf{C})d\mathbf{C}dV = f(\mathbf{C})d\mathbf{C}u dt dS.$$

Figure 3.2 Plane AB represents the reservoir-system interface. Particles within the cylinder may cross dS within the time dt .

The reservoir fluid is an ideal gas and therefore the distribution of particles within this region will be Maxwellian. The velocity probability distribution for particles crossing the plane AB is:

$$P(\mathbf{C}) = \frac{1}{\alpha} u e^{-\beta^2 c^2} = \left[\frac{1}{\alpha_x} u e^{-\beta^2 (u - u_0)^2} \right] \left[\frac{1}{\alpha_y} e^{-\beta^2 (v - v_0)^2} \right] \left[\frac{1}{\alpha_z} e^{-\beta^2 (w - w_0)^2} \right]$$

where $\beta = \sqrt{\frac{m}{2kT_0}}$. This equation is separable and can be divided into three separate

Maxwell probability functions for each direction $P(\mathbf{C}) = P_x(u)P_y(v)P_z(w)$. Normalizing this equation gives:

$$\alpha_x = \frac{1}{2\beta^2} e^{-\beta^2 u_0^2} + \frac{\sqrt{\pi}}{2\beta} u_0 (1 + \text{erf}(\beta u_0))$$

$$\alpha_y = \alpha_z = \frac{\sqrt{\pi}}{\beta}.$$

where $\text{erf}(x)$ is the error function. The two velocities parallel to the plane are Gaussian distributed, while the velocity perpendicular is non-Gaussian.

3.4.2 Acceptance-Rejection

In the simulation the velocities for the particles injected by the flux boundary must be created at random. The method used for the random generation of the particle velocity in the x direction is called acceptance-rejection. The particle velocities in the y and z direction are obtained by inverting the Gaussian function. This inversion allows the y and z velocities to be randomly selected by a uniform deviate. The velocity probability for the x direction cannot be inverted. The acceptance-rejection method uses two uniform deviates $\mathfrak{R}_1$ and $\mathfrak{R}_2$, distributed between $[0,1)$. The first deviate is used to choose a trial velocity $u_{\text{trial}} = u_{\text{top}} \mathfrak{R}_1$. The velocity u_{top} is chosen by finding the most probable velocity (u_{mp}) of the u velocity distribution. This most probable velocity is used to determine a maximum velocity using the most probable thermal velocity $v_{\text{th}} = \frac{1}{\beta}$:

$$u_{\text{top}} = u_{\text{mp}} + \min(6v_{\text{th}} + u_{\text{mp}}).$$

Once a trial velocity has been chosen the second uniform deviate is used to calculate a random probability $P_{\text{trial}} = P(u_{\text{top}}) \mathfrak{R}_2$, where $P(u_{\text{top}})$ is the highest probability. The velocity u_{trial} is chosen if:

$$P_x(u_{\text{trial}}) > P_{\text{trial}}.$$

The selection process continues until a value of u_{trial} satisfies the above condition.

3.4.3 Poisson Distribution

An equation still must be obtained for the average number of particles crossing the plane separating the reservoir and the channel. The distribution function $f(\mathbf{C})$ used above is Maxwellian and can be written:

$$f(\mathbf{C}) = \frac{\beta^3 n}{\pi^{3/2}} e^{-\beta^2 \mathbf{C}^2}$$

where n is the number of particles in the reservoir per unit volume. Putting $f(\mathbf{C})$ into the equation for the total number gives:

$$\begin{aligned} \langle N_p \rangle &= \int_{t_i}^{t_f} dt \int_0^{L_x} dS \int_0^{L_y} du \int_{-\infty}^{\infty} dv \int_{-\infty}^{\infty} dw (n u f(\mathbf{C})) \\ &= \Delta t L_x L_y n \frac{1}{2\beta\sqrt{\pi}} \left\{ e^{-\beta^2 u_0^2} + \beta\sqrt{\pi} u_0 [1 + \text{erf}(\beta u_0)] \right\} \end{aligned}$$

where $L_x L_y$ is the cross sectional area of the plane AB.

For the simulation, the probability of N_p particles crossing the AB plane in time $\Delta T = t_f - t_i$, must be determined. A distribution function must be determined for the average number of particles crossing the boundary. Each time interval, a discrete number of particles cross the interface. This gives a discrete probability distribution. The probability that N events will occur in a time interval ΔT , is given by the Poisson distribution [4]:

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

Simple and efficient algorithms are known for generating Poisson distributed random numbers (e.g. Press, et. al. [14]).

3.5 DSMC Method

Now that the formulation for the statistical kinetic theory has been presented, the DSMC method can be outlined. The method starts by using the standard method of particle motion in free space $\mathbf{r}_{\text{new}} = \mathbf{r}_{\text{old}} + \mathbf{v}\tau$. The particle's new position is calculated from its old position, old velocity and length of time traveled. All of the particles are moved using the above equation.

After all of the particles have been moved and all surface scattering evaluated, some particles are selected for collision. In MD any two particles would collide based on their trajectories. The DSMC method uses the statistical nature of the gas to select particle collisions. Before any collisions occur, particles are sorted into collision cells. The entire domain of the program is divided into these rectangular cells, Figure 3.3. The collision cells will not all have the same volume because the walls may encroach into the cell, reducing the volume. Each individual cell's volume is calculated by determining the amount of inclusion by each wall. If no walls encroach, the cell will have the maximum volume. If only one of the walls encroaches, the cell volume is calculated by numerically integrating the area of the extruded part and subtracting it from the maximum cell volume. If a cell contains both walls, the volume is calculated by numerically integrating the volume between the two walls. These different methods

Figure 3.3 The system domain is divided into many cells. A processor will initialize particles into each cell.

ensure that all possible cell geometries are calculated appropriately.

Initially, particles are placed into the cells with uniform density. The total number of particles initialized depends on the number of atoms represented by each simulation particle. Each simulation particle represents N_{eff} real particles. This number is given by

$$N_{eff} = \frac{\rho V}{m N_s},$$

where ρ is the density of the gas, V is the volume of the system, m is the mass of

the particle and N_s is the number of simulation particles. It must be ensured that enough particles are initialized into each cell to accurately represent the real system. The size of the cells is determined by the length a particle travels in a single timestep and the mean free path. The cell dimensions cannot be greater than the mean free path of the gas. Both

the cell volume and timestep are chosen so that a particle will not pass through an entire cell in a timestep.

Once the particles are initialized into their cells and free flight has occurred, collisions are handled. The hard sphere model is used for particle interactions. For hard sphere collisions the probability of a collision is proportional to the relative velocity, assuming the positions are uniformly distributed within the cell. This allows all the particles in a particular cell to be candidates for collisions. The probability of a collision for any two particles within a cell is:

$$P_{\text{coll}}(i, j) = \frac{|\mathbf{v}_i - \mathbf{v}_j|}{\sum_{m=1}^{N_c} \sum_{n=1}^{m-1} |\mathbf{v}_m - \mathbf{v}_n|} \quad (3-1)$$

where i and j designate the two different particles and N_c is the number of particles in the cell. The denominator normalizes this discrete distribution.

The explicit normalization of the above equation is computationally expensive. However, the double sum is not required when using the more efficient acceptance-rejection method of obtaining random values from the distribution. The method starts by choosing two candidate particles i and j from the cell at random. Their relative speed is computed. If the particles satisfy the condition:

$$\frac{|\mathbf{v}_i - \mathbf{v}_j|}{v_r^{\text{max}}} > \mathcal{R}$$

they are accepted as collision particles. If the candidate pair is accepted for collision, their velocities are reset. If they are not accepted, another pair is selected until there is

acceptance. This acceptance-rejection scheme exactly selects particles according to (3-1).

The $v_r^{\max}$ in the above equation is used as a normalization term and it must be chosen so that it is greater than or equal to v_r , the relative velocity. The value for $v_r^{\max}$ will be calculated for each cell. The probability is still correct even if the value of $v_r^{\max}$ is overestimated. The collision pair selection process is more efficient if $v_r^{\max}$ is chosen by guessing rather than computing it every timestep.

The next step after accepting a collision pair is to reset each particle's velocity. After collision the particles will have new velocities denoted by $\mathbf{v}_i^*$ and $\mathbf{v}_j^*$. The conservation of momentum states that the center of mass velocity of the two particles will be conserved. This gives:

$$\mathbf{v}_{\text{cm}} = \frac{1}{2}(\mathbf{v}_i + \mathbf{v}_j) = \frac{1}{2}(\mathbf{v}_i^* + \mathbf{v}_j^*) = \mathbf{v}_{\text{cm}}^* .$$

Conservation of energy states that the magnitude of the relative velocity does not change after the collision:

$$v_r = |\mathbf{v}_i - \mathbf{v}_j| = |\mathbf{v}_i^* - \mathbf{v}_j^*| = v_r^* .$$

The two final velocities $\mathbf{v}_i^*$ and $\mathbf{v}_j^*$ represent six unknown quantities. The conservation laws constrain four of the unknowns. The other two unknowns, the angles θ and ϕ , are selected at random. The relative velocity in these coordinates is:

$$\mathbf{v}_r^* = v_r [(\sin \theta \cos \phi)\mathbf{i} + (\sin \theta \sin \phi)\mathbf{j} + (\cos \theta)\mathbf{k}] .$$

The hard sphere scattering model is isotropic in nature. The random directions used to obtain $\mathbf{v}_r^*$ are random in any reference frame. The angles used are uniformly distributed on the unit sphere. All of the angles are equally likely after a particle collision. The angles are chosen using two uniform deviates. The azimuthal angle ϕ is uniformly distributed between 0 and 2π , a value for ϕ is randomly selected by $\phi = 2\pi\mathfrak{R}_1$. The θ angle is chosen according to the probability distribution $P(\theta)d\theta = \sin\theta d\theta$. Using $q = \cos\theta$, the probability distribution becomes $P(q)dq = (\frac{1}{2})dq$. This gives a uniform distribution in the interval $[-1,1]$ for q . Instead of calculating θ , we compute: $q = 2\mathfrak{R}_2 - 1$. $\cos\theta = q$. $\sin\theta = \sqrt{1 - q^2}$. Combining these values along with the azimuthal angle gives equations for the post collision velocities:

$$\mathbf{v}_i^* = \mathbf{v}_{cm} + \frac{1}{2}\mathbf{v}_r^*, \quad \mathbf{v}_j^* = \mathbf{v}_{cm} - \frac{1}{2}\mathbf{v}_r^*.$$

The process of choosing collision pairs will continue until the required number is chosen.

To determine the number of collisions that should take place in any given cell, the collision frequency must be calculated. The total number of collisions in a time (τ) is

given by: $M_{coll} = \frac{1}{2} N_c f \tau$, where N_c is defined above as the number of particles in a cell, f

is the collision frequency. Substituting for the values of N_c and f gives:

$$M_{coll} = \frac{N_c^2 \pi d^2 \langle v_r \rangle \tau}{2V_c}, \text{ where } V_c \text{ is the volume of the cell. The collision pairs go through}$$

an acceptance-rejection scheme, which can be used to obtain a ratio of $\frac{M_{\text{coll}}}{M_{\text{cand}}}$, where

M_{cand} is the number of collision candidates; $\frac{M_{\text{coll}}}{M_{\text{cand}}} \approx \frac{\langle v_r \rangle}{v_r^{\text{max}}}$. Using this in the equation

above gives:

$$M_{\text{coll}} = \frac{N_c^2 \pi d^2 v_r^{\text{max}} \tau}{2V_c}.$$

The quantity v_r^{max} is not calculated from the constituent particles in each cell, but is estimated. If the value for v_r^{max} is overestimated, then more candidate collisions will be tested and rejected. It is still more efficient to overestimate v_r^{max} than to compute it every timestep. The DSMC simulation estimates the value for v_r^{max} by setting it equal to $3 v_{\text{mp}}$, where v_{mp} is the most probable velocity. If the collision candidate's v_r^{max} is found to be greater than the existing v_r^{max} , then v_r^{max} is set to this larger value.

3.6 DSMC Program

The physical domain simulated by the program is separated into equal columns. On a multiprocessor parallel computer, each column is then assigned to an available processor. The system is assumed to be homogenous in the z-direction. The program's first operations are performed by the manager or host process. The manager process has different responsibilities than the worker processes. Table 3.1 shows the main program *spdsmc.f* with indications for the manager's and worker's roles. The manager's role is

strictly data I/O; it will read in the initial conditions and collect and output the data upon completion. The remaining processors will all be workers. These workers will perform the majority of code outlined above. All worker processes will be performing the same routines, but on different data sets in their region.

spdsmc.f

(M) Start MPI and enter a communication group

(M,W) Initialize the system (if restart; *starter.f*)

Main Loop

(W) Create particles at the In and Out flux boundaries (*maker.f*)

(W) Move all particles (*mover.f*)

(W) Check for boundary interactions (*sinehit.f*)

(subroutine of *mover.f*)

(W) Send and Receive particles to and from neighbors (*mover.f*, MPI)

(W) Sort particles into collision cells (*sorter.f*)

(W) Select and evaluate collisions in cells (*collide.f*)

(W) Evaluate pressure at flux boundaries (*pressure.f*)

(W) Take statistical samples in statistical cells (*gridsmf.f*)

End Main Loop

(M/W) Communicate Statistical Data

(M) Write Statistical Data (which may also be used for restart)

Table 3.1 Outline of the DSMC program. Each step is indicated to distinguish between manager (M) and worker (W) roles.

The first step in the simulation is initialization. If the program is being run for the first time, then the manager process will simply distribute the initial conditions and problem data to the workers. Each worker will then divide its domain into the requested number of collision cells. These cells are chosen so that a certain average number of particles will be contained within each cell. The cell width is chosen so that it is a fraction of the mean free path and the timestep is chosen so that the particle rarely crosses

more than one cell. The correct number of particles is put into each cell to obtain uniformity across the whole system. It has been empirically shown that 20 particles in each collision cell will give good results [7]. If the program is being restarted, then the *starter.f* routine is called.

The *starter.f* routine is first run by the manager process. This routine will read in the data written out by the previous run. The data includes average number, temperature and flow velocity for each cell. This data will then be used to initialize particles into each cell in the domain. Each worker will receive the data from the manager and initialize the requisite number of particles into the cell. The velocity of each particle is then randomly set using the Maxwell-Boltzmann distribution $v_x, v_y, v_z = \sqrt{\frac{2kT}{m}} \sqrt{-\ln(1 - \mathfrak{R}_1)} \sin(2\pi\mathfrak{R}_2)$, where $\mathfrak{R}_1$ and $\mathfrak{R}_2$ are uniform deviates. The flow velocity is then added to each particle in the cell giving a close approximation of the previous run's system.

The previous run could be restarted exactly if the position and velocity were recorded for every particle. This would entail large data transfers both between processors and fixed storage media, resulting in costly computation and large data files. Recording only the cell information reduces the restart file size significantly.

Once the system is initialized, the workers at the beginning and end of the system can handle in and out flux boundaries. The *maker.f* routine will create new particles at the entrance and exit. Particles leaving the system are discarded. The entrance and exit are modeled as thermal reservoirs with a fixed temperature, mass density and flow

velocity. Using the theory discussed in Chapter 2, the correct number of particles fluxing into the system through these boundaries is obtained. New particles entering the system are assigned a random height and moved off the boundary interface at a random fraction of the timestep.

After the *maker.f* routine, the *mover.f* routine moves the particles. Each processor will loop through the list of particles and apply $\mathbf{r}_{\text{new}} = \mathbf{r}_{\text{old}} + \mathbf{v}\tau$ to each particle. After the particles are moved they are analyzed for interactions with important boundaries. Along with the entrance and exit boundaries, the system wall boundaries and boundaries formed by the processor interfaces must be considered. Particles leaving a processor's domain are queued into outgoing particle queues to be sent either left or right via interprocessor communication calls. Processors with domains on the ends will handle the outgoing particles differently since particles exiting the system are discarded.

As each particle is moved, it is checked to see if it interacts with the top and bottom walls. This is the function of the *sinehit.f* subroutine. The *sinehit.f* subroutine takes as arguments the geometry and attributes of the walls and the characteristics of the particle. This subroutine will first check which wall the particle will hit. If the particle's initial and final positions are beyond the wall boundary, the particle has escaped and is removed. Otherwise, the post-interaction velocity is calculated based on the thermal interaction. The particle is then moved for the remaining time left in the current timestep, again checking for possible wall or boundary interactions.

Once the boundary interactions are completed, the particles that exit a processor's domain are sent to left and right neighbors. Outgoing particles are put into lists to be sent. Interprocessor communication calls are used to send out the particle information. These send calls are non-blocking, meaning they do not wait for acknowledgment of receipt before proceeding. All processors will then receive the particle information from their left and right neighbors. The receive calls are blocking, meaning the program will not proceed until an incoming packet of data arrives. These blocking receive calls provide synchronization for the program, ensuring that all processors are on the same timestep. The processors at the ends will only receive particle information from one neighbor.

Now that all particles reside in their proper locations, they can be sorted into collision cells. This is the responsibility of the *sorter.f* subroutine. All incoming particles are grouped with other particles into the collision cells based on location. When all the particles are grouped properly, they are collided. The *collide.f* subroutine will then use the lists of particles created by *sorter.f* to collide the particles in each cell based on the DSMC collision algorithm.

After all particles are moved and collided, the pressures at the entrance and exit are evaluated by the *pressure.f* subroutine. This routine samples the thermal energy,

which is the total kinetic energy in the center of mass frame. $\langle E_{th} \rangle = \frac{1}{2} m \sum_{i=1}^N |\mathbf{v}_i - \mathbf{v}|^2$.

where N is the total number of particles in the region being tested and $\mathbf{v}$ is the center of

mass velocity. The temperature is related to the thermal energy by $\langle E_{th} \rangle = \frac{3}{2} NkT$. The correct instantaneous temperature is obtained by correcting for the fluctuations of the center of mass velocity. These fluctuations contribute a mean kinetic energy of $\frac{3}{2} kT$ giving a correction factor $C = \frac{N+1}{N}$. Once the temperature is obtained from $\langle E_{th} \rangle$ the pressure can be found by using the ideal gas law.

Once the correct pressures are obtained, the pressures at the entrance and exit are checked for compliance. These end pressures at the flux boundaries are fixed by the fluid reservoirs. The calculated pressures must be kept close to the expected reservoir pressures. The flow velocity is subsonic; meaning the flow near the flux boundaries are affected by flow further along the microchannel. The flow velocities of the reservoirs are adjusted using a relaxation iteration. This adjustment affects the number of particles entering the system. If the calculated pressures are too high near a flux boundary, the flow velocity will be decreased and vice versa. The flow velocity is adjusted according to the linear relation: $u_{new} = u_{old} + \frac{1}{10} \frac{P_0 - P_r}{P_0} v_{mp}$, where P_0 is the reservoir pressure, P_r is the pressure in the region near the boundary and v_{mp} is the most probable velocity.

The factor $\frac{1}{10}$ is empirically found to give good results.

Now that all particles have been moved, interacted and collided for the current timestep, statistical samples of the system are taken. This is accomplished by dividing

the region into statistical cells. These statistical cells can be larger or smaller than the collision cells; in these simulations they are larger. The *gridsmp.f* subroutine is responsible for the collection of the statistical data. Running sums are collected for the number of particles, particle velocity and squared velocity in each statistical cell. When the simulation is completed, the sums are normalized by the number of particles evaluated. This will give averages for each statistical cell. Each cell's temperature can then be evaluated by calculating the thermal energy and the pressure and temperature can be obtained as before.

When all processors have finished the simulation, they send all of their statistical data to the manager. This manager will then assemble the data and write it out to data files. These data files include information for restart as well as all of the statistical data from the statistical cells. The data files are then post-processed by a Matlab script.

Chapter 4 Parallel Processing

The solution of the DSMC method would take several days to complete on a workstation. This is due to the large number of particles, typically 16,000-100,000, in the system. Tracking and updating the positions of these particles is time consuming.

To reduce the time involved in running the DSMC code, the problem is distributed across many processors. In this case the processors are on a Cray T3D. The Cray T3D is a massively parallel processing computer (MPP) with 256 processing elements (PE). Each PE has a processor and 64 megabytes of random access memory. The processors are DEC alpha EV4 chips running at 156 MHz. The processing elements all run a version of the Cray operating system called Unicos™, a variant of the Unix™ operating system.

The microchannel problem is initially divided into several columns. The columns are then subdivided into cells. Each column is handled by a different processing element. This type of distribution is called domain decomposition. The other type of decomposition is algorithmic decomposition, which distributes parts of the algorithm rather than the data.

Each processing element will work on its own column, maintaining and creating the particles within its domain. Each PE will run an entirely separate part of the simulation. Due to the nature of the simulation, particles will leave one PE's domain and enter another. Accomplishing this transfer requires sending information, a message, to

the receiving processor. This interprocessor communication is handled by a message passing library. In this thesis the message passing was implemented using the Message Passing Interface (MPI) [8] protocols.

MPI is a library of FORTRAN or C functions that allows different processors to communicate in a standard way. This allows a programmer to write a FORTRAN program that will work on many different types of architectures without worrying about the underlying communication strategies employed at the system level. The program needs only to be compiled using an architecture-specific version of the MPI library. The communication strategy employed by MPI centers around messages. These messages can be as simple as an integer or as complex as a combination of many matrices.

To implement a program using MPI, a computational model must first be chosen. In this thesis the data driven model was chosen. This model uses the data to define each processor's workload. The specific data driven model being used is called the manager-worker model. This model is implemented in a Single Program Multiple Data (SPMD) configuration, meaning that the same program is run on every processor.

The program establishes one manager and multiple worker processes. All of the workers will run exactly the same code. The manager handles the initialization and final data output. At initialization each processor will run specific MPI calls that will bind them to a communication group. This communication group exists to limit message traffic to processors needing the information related to the group. In the DSMC program

all of the processors are in the same communication group. MPI gives each processor a rank within the group to distinguish the PE.

A shell script takes the number of requested processors as an argument and starts an instance of the program on each PE. The manager process is the first process to be initialized on the computer. Its rank within the communication group is zero. To generalize the SPMD program to work for a manager or worker, conditionals (i.e., if statements) are included to check the rank of the PE running the code. The PE will then only run the code specific to its job description.

Figure 4.1 The system domain is separated into columns. The arrows represent the communication system used by the DSMC program.

Communication between PEs is accomplished by passing messages. A PE creates a buffer containing the elements to be passed. This message is then sent with a destination processor rank and an optional message tag. Figure 4.1 shows the message communication strategy. The receiving PE will check the rank and tags and if the

message is for it, extract the data. The principle data items (PDI) are a particle's position and velocity. Each PE maintains a list of its own particle's information. If a particle moves out of the domain of a PE, that PE must send the particle's velocity and position information to the PE the particle will enter next. The send function calls used are non-blocking, that is they do not wait for acknowledgment of the receipt of a message by the receiving processor. Send calls in MPI can be made blocking. The receive function calls are blocking, meaning that the PE will not proceed until information is received. If a PE does not have any particle information to distribute it will send a message with no content. This serves as a synchronization signal for all of the worker processes.

The manager process will collect all of the particle information at the end of the program cycle. All the worker processes will send their particle information to be received by the manager. The manager will then process this information to create data files with the assembled data.

The parallel nature of the program serves to limit the work of each PE and therefore speed up the process. This data decomposition of this parallel method allows the program to be scaled to work on larger numbers of PEs. The problems can then be either solved much faster or become much larger.

Chapter 5 Simulation Results

The pressure profiles of the microchannel with rough walls were obtained using two different methods, the factored-implicit scheme of Nigam and White, and the DSMC method. In both methods the same initial conditions were used. The overall length of the microchannel was $L = 5 \mu\text{m}$. The entrance and exit pressures were set at 5 atm and $p_a = 1 \text{ atm}$ and respectively. The ambient temperature was set at $T_0 = 273 \text{ K}$. For the exit pressure the mean free path of argon is 62.6 nm and the viscosity is $\mu = 2.08 \times 10^{-5} \frac{\text{Ns}}{\text{m}^2}$.

Argon was used due to its monatomic configuration and its similarity in mass and diameter to molecular oxygen and nitrogen. The particles in the simulation were modeled as hard spheres with diameters of 3.66 Å, masses of $6.63 \times 10^{-26} \text{ kg}$ and mass density of $1.78 \frac{\text{kg}}{\text{m}^3}$. An average of 16,000 particles were used in the simulation. The number of particles in the system fluctuates due to the creation and removal of particles at the entrance and exit. A simulation run is complete when pressure profiles from successive runs agree within one percent. The separation of the two sinusoidal walls was varied from 80 - 200 nm. The microchannel was configured with varying wall separations, sine amplitudes and phase differences. The upper and lower walls were fully accommodating with $a = 1$.

All of the DSMC simulations were performed on a Cray T3D using 32 processors. With this configuration, the simulation typically takes 2 hours for 15,000 timesteps. The

factored-implicit scheme solutions were run on a Pentium class PC. These solutions used fully accommodating walls and could be configured for different types of Reynolds equations. In this thesis the continuum, first-order slip and Fukui-Kaneko slip corrected Reynolds equations were solved. Solving the Reynolds PDE takes approximately 10 minutes on a Pentium class PC.

5.1 Validation

The factored-implicit solver and the DSMC simulator for rough walls must be verified. For the case of the implicit solver, the solutions obtained can be compared to DSMC solutions obtained previously and known to be accurate. The new DSMC method is an altered version of a program used to simulate slider air bearings [17]. The new method must take into account more complex cell volumes and particle position evaluations. The DSMC method can be also be tested by measuring its divergence from thermodynamic equilibrium when the boundaries are at equilibrium.

5.1.1 Factored-Implicit Solver

The factored-implicit solver is designed to solve problems with linear aspect walls, as discussed in Chapter 2. The DSMC method used in this thesis was adapted from a method to solve for the pressure profiles of slider bearings. Since the implicit solver predated the DSMC method to simulate microchannels with rough walls, the solutions were verified against the slider bearing DSMC method. Several cases were chosen for

the verification. These cases were chosen to cover the range of situations. Figures 5.1 and 5.2 show the pressure profiles for an air bearing system. The three Reynolds equations: continuum, first order slip and Fukui-Kaneko were solved using the factored-implicit method. In the Figures 5.1 and 5.2 the Fukui-Kaneko solution agrees best with the DSMC data, while the continuum and first order slip agree poorly.

Figure 5.1 Verification of the factored-implicit solver against a pressure profile for a slider air bearing. $Kn_0 = 1.912$

Figure 5.2 Verification of the factored-implicit solver against a pressure profile for a slider air bearing. $Kn_0 = 4.780$

5.1.2 Equilibrium Verification

To measure the DSMC method's divergence from thermodynamic equilibrium, all of the system's characteristic quantities are set to equilibrium values. The reservoirs at each end are kept at atmospheric pressure $p_a = 1$ atm and all of the boundary temperatures are set to $T = 273$ K. The relaxation to equilibrium is a test of the cell volume and particle collision routines. The walls are set to be specularly reflecting so they do not influence the relaxation. There will be no average flow through the channel. As the system relaxes, the gas will come into equilibrium with the reservoirs. Any variances brought about by inaccurate cell volumes will become an obvious non-equilibrated spot on graphs of density or pressure. Figure 5.3 shows the plot of Pressure vs. Position(X) in

the system. The entire domain of the system is divided into statistical cells. These cells are then grouped into slabs and the statistical data for each cell in the slab is averaged. As Figure 5.3 shows, the pressure is randomly distributed around atmospheric pressure, as expected.

Figure 5.3 One-dimensional pressure along the microchannel. Pressure is deviating around atmospheric pressure as expected.

5.2 Simulation Results

These simulation results will show the comparison of the factored-implicit scheme of Nigam and White compared to the DSMC method. All simulations have the following parameters: $l = 5 \mu\text{m}$ $\lambda = 62.6 \text{ nm}$.

5.2.1 Flat Wall Simulation

The first microchannel simulation consists of two flat walls separated by 80 nm. Figure 5.4a shows the pressure profile for the geometry shown in Figure 5.4b. The Reynolds equation solver uses the normalized height through the microchannel. Because of the normalized height, the flat walled microchannel will produce the same pressure profile of an in-phase microchannel with the same separation (section 5.2.2). The four solutions shown are three Reynolds solutions and the DSMC solution. The three Reynolds solutions are the continuum, first order slip and Fukui-Kaneko Reynolds solutions. The DSMC solution is the final steady state solution, the error bars are on the order of the size of the markers.

Figure 5.4a Pressure profile for a microchannel with flat walls. Data includes continuum, first order slip, Fukui-Kaneko and DSMC solutions.

Figure 5.4b System geometry for Figure 5.4a.

5.2.2 Wall Separation Variation

The first set of simulations varied the wall separation distance between the upper and lower walls. The upper and lower walls consisted of sinusoids with two periods, a phase difference of $\phi = 0^\circ$ between them and amplitudes of $A = 20$ nm. The separation of the two walls, h_0 , was set at two values 80 nm, which is close to the mean free path and 200 nm which is approximately three times the mean free path. Figure 5.5b shows a basic sinusoidal wall microchannel geometry.

Figure 5.5a shows the pressure profile for the microchannel geometry in Figure 5.5b. The height at the exit is $h_0 = 80$ nm, which gives a Knudsen number at the exit of

$$Kn_0 = \frac{\lambda}{h_0} = 0.7825. \text{ The Fukui-Kaneko solution agrees very well with the DSMC}$$

solution, while the continuum solution compares poorly. The first order slip solution is only slightly different from the Fukui-Kaneko solution.

The factored-implicit solver only takes into consideration the variation in height through the microchannel. A pressure profile of the microchannel with flat walls, Figure 5.4a, gives the same pressure profile as Figure 5.5a.

Figure 5.5a Pressure profile for a microchannel with $h_0 = 80$ nm, two periods, $\phi = 0^\circ$ and $A = 20$ nm.

Figure 5.5b System geometry for Figure 5.5a.

The separation between the two walls is increased to 200 nm, see Figure 5.6b. This gives a Knudsen number at the exit of $Kn_0 = 0.3130$. This value for Kn_0 suggests that the Reynolds equation solutions should be more accurate. This is due to the assumption made in the derivation that the Kn_0 values are much less than 1. The plot of the pressure profile shows that the continuum solution does not compare very well with the DSMC solution, while the first order slip and Fukui-Kaneko are nearly indistinguishable.

Figure 5.6a Pressure profile for a microchannel with $h_0 = 200$ nm, two periods, $\phi = 0^\circ$ and $A = 20$ nm.

Figure 5.6b System geometry for Figure 5.6a.

5.2.3 Phase Variation

The phase between the upper and lower walls was varied. The two values used for the following plots were 90° and 180° . Deviation in the pressure profiles can be seen in both cases with the larger variation in the smaller separation plots. The parameters for these cases are $A = 20$ nm, two periods, $h_0 = 80$ nm or 200 nm. Figures 5.7b and 5.8b show the microchannel with a 90° phase difference and wall separation of $h_0 = 80$ nm and 200 nm respectively. In Figure 5.7a notice that all the Reynolds solutions agree well with the DSMC solution, but in Figure 5.8a, with larger wall separation, the Fukui-Kaneko and the first order slip agree best with the DSMC solution.

Figure 5.7a Pressure profile for a microchannel with $h_0 = 80$ nm, two periods, $\phi = 90^\circ$ and $A = 20$ nm.

Figure 5.7b System geometry for Figure 5.7a.

Figure 5.8a Pressure profile for a microchannel with $h_0 = 200$ nm, two periods, $\phi = 90^\circ$ and $A = 20$ nm.

Figure 5.8b System geometry for Figure 5.8a.

When the phase difference between the walls is increased to $\phi = 180^\circ$, the microchannel exhibits the same oscillating behavior for the pressure profiles with $\phi = 90^\circ$. Figures 5.9b and 5.10b show the microchannel system with phase difference at $\phi = 180^\circ$ and wall separation at $h_0 = 80$ nm and 200 nm respectively. In Figure 5.9a, the continuum solution cannot simulate the system well, while the first order slip solution compares well with the Fukui-Kaneko and DSMC solutions. In Figure 5.10a, the Fukui-Kaneko and first order slip agree well with the DSMC method, while the continuum solution agrees poorly.

Figure 5.9a Pressure profile for a microchannel with $h_0 = 80$ nm, two periods, $\phi = 180^\circ$ and $A = 20$ nm.

Figure 5.9b System geometry for Figure 5.9a.

Figure 5.10a Pressure profile for a microchannel with $h_0 = 200$ nm, two periods, $\phi = 180^\circ$ and $A = 20$ nm.

Figure 5.10b System geometry for Figure 5.10a.

5.2.4 Amplitude Variation

The next set of plots show the comparison of the Reynolds solutions with the DSMC method, when the amplitudes of the sinusoidal walls were varied between 20 nm and amplitude $< \frac{h_0}{2}$. More extreme amplitudes will be discussed in the constrictions section (5.2.5). Figures 5.11a and 5.12a show the variation of the pressure profiles with amplitude for the geometries shown in Figure 5.11b and 5.12b. The parameters for these plots are two periods, $\phi = 180^\circ$, $h_0 = 80\text{nm}$, $Kn_0 = 0.7525$. Figure 5.11a shows a large discrepancy between the three Reynolds solutions and the DSMC solution, while Figure 5.12a shows that all solutions agree very well. The Fukui-Kaneko and first order slip are again indistinguishable.

Figure 5.11a Pressure profile for a microchannel with $h_0 = 80\text{ nm}$, two periods, $\phi = 180^\circ$ and $A = 30\text{ nm}$.

Figure 5.11b System geometry for Figure 5.11a.

Figure 5.12a Pressure profile for a microchannel with $h_0 = 80$ nm, two periods, $\phi = 180^\circ$ and $A = 39$ nm.

Figure 5.12b System geometry for Figure 5.12a.

5.2.5 Constrictions

In this section the amplitude is varied to provide constrictions through the microchannel. The phase and amplitude will also be varied together to provide a sine wave system geometry. Figures 5.13b and 5.14b show the two geometries simulated in this section. The geometry in Figure 5.13b is created by fully out of phase walls with amplitudes set to 99% of the wall spacing. In this system geometry, only a few particles can leave the high pressure section each timestep. The pressure beyond the first contained section is then the atmospheric pressure. All of the Reynolds solutions agree well with the DSMC method in Figure 5.13a. In Figure 5.14b the geometry is much more complex with respect to the DSMC method. The pressure profile will be a simple linear relation due to the constant height (wall separation) throughout the system, making the system easy to solve by the factored-implicit solver. The agreement of the Reynolds solutions is much like Figure 5.4.

Figure 5.13a Pressure profile for a microchannel with $h_0 = 80$ nm, two periods, $\phi = 180^\circ$ and $A = 39.8$ nm.

Figure 5.13b System geometry for Figure 5.13a.

Figure 5.14a Pressure profile for a microchannel with $h_0 = 80$ nm, six periods, $\phi = 0^\circ$ and $A = 200$ nm.

Figure 5.14b System geometry for Figure 5.14a.

Chapter 6 Discussion and Conclusion

The microchannel with rough walls has many applications in the MEMS area. The pressure profiles of these microchannels can be accurately obtained using both the factored-implicit solver and the DSMC method. The DSMC method is a particle simulation method that models gas dynamics while reducing computation time. This method uses the thermal reflections from the walls approximated by sinusoids. Studies have shown that complex surface scattering techniques are not required for representing microchannel walls [17]. The DSMC method is limited by the computation time compared to PDE solvers.

This thesis concentrates on the agreement of the Reynolds equation compared to the DSMC method. The factored-implicit method was developed to solve the Reynolds equation more efficiently. To obtain an accurate result, slip correction was added to the basic Reynolds equation. First order slip correction and a more detailed Fukui-Kaneko slip correction were compared to the continuum Reynolds equations and the DSMC method results.

The Fukui-Kaneko solutions compared the best, qualitatively, to the DSMC method. In some cases the first order slip solution compared very well to the Fukui-Kaneko and DSMC solutions. Some first order solutions were nearly indistinguishable for the Fukui-Kaneko solution. Overall the continuum solutions did not compare well to the DSMC method.

The accuracy of the fits for the Reynolds equation solutions can be traced back to the derivation of the slip flow. The continuum solution is the uncorrected solution and should be inaccurate in all cases. The slip corrections all assumed that the average flow velocity in the y-direction would be negligible. For the sinusoidal walls and system geometries, this is not the case. Even with this inaccuracy, the Reynolds solutions agreed qualitatively with the DSMC data.

Larger wall separations lead to a lower Knudsen number and this leads to problems due to the larger average y-direction flow velocities. Even with this limitation most of the Reynolds solution agreed with the DSMC data. One noticeable discrepancy was the solution for a small constriction, where $A = 30$ nm, $h_0 = 80$ nm, $\phi = 180^\circ$. In this scenario, the continuum and first order slip corrected Reynolds equation could not simulate the microchannel, while the Fukui-Kaneko Reynolds equation agreed with the DSMC data.

The Fukui-Kaneko solution agreed qualitatively with the DSMC method, but not exactly. Future work will include modifications to the Reynolds equation to account for wall curvature as well as spacing. The velocity profiles of the flow in the microchannel will also be explored [10].

Bibliography

- [1] Bhatnagar, P.L., Gross, E.P., Krook, M. (1954), "A Model for Collision Processes in Gases, I," *Phys. Rev.*, 94, 511-525.
- [2] Bird, G. A. (1994), "Molecular Gas Dynamics and Direct Simulation of Gas Flows," Clarendon, Oxford.
- [3] Burgdorfer, A. (1959), "The Influence of the Molecular Mean Free Path on the Performance of Hydrodynamic Gas Lubricated Bearings," *Trans. ASME*, 81, 94-100.
- [4] Feller, W. (1957) "An Introduction to Probability Theory and its Applications." Wiley, New York.
- [5] Fukui, S. and R. Kaneko (1990), "Analysis of ultrathin gas film lubrication based on linear Boltzmann equation: First report derivation of a generalized lubrication equation including thermal creep flow," *J. Tribology*, 110, 253-262.
- [6] Fukui, S., R. Kaneko, (1995), "Molecular Gas Film Lubrication (MGL) ", in "Handbook of Micro/Nanotribology," ed. B. Bushan, CRC Press, Boca Raton, 559-604.
- [7] Garcia, A. L. (1994), "Numerical Methods for Physics," Prentice Hall, Inc.
- [8] Gropp, W., L. Ewing, and A. Skjellum (1994), "Using MPI: Portable Parallel Programming with the Message-Passing Interface." The MIT Press, Cambridge.
- [9] Gross, W.A. (1962), "Gas Film Lubrication," Wiley, New York.

- [10] Huang, Kerson (1987). "Statistical Mechanics," 2nd ed., John Wiley & Sons, New York.
- [11] Kittel C. (1980), "Thermal Physics," W. H. Freeman and Company.
- [12] Landau, L. D. and E. M. Lifshitz (1959), "Fluid Mechanics," Pergamon, Oxford.
- [13] Piekos, Edward S.(1995), "DSMC Modeling of Micromechanical Devices," MIT, Cambridge, FDRL-TR-95-5.
- [14] Press, W., B. Flannery, S. Teukolsky, and W. Vetterling (1992), "Numerical Recipes in FORTRAN," 2nd ed., Cambridge University Press, Cambridge.
- [15] Robert, M., (1993), "New Implementation of Boltzamann Correction to Lubrication Theory," IBM Research Report (internal).
- [16] Senturia, S. D., N. Aluru, J. White (1997), "Simulating the Behavior of MEMS Devices: Computational Methods and Needs," IEEE Comp. Sci. Eng., Jan.-Mar., 30-43.
- [17] Wenski, Troy (1997), "Particle Simulations of Microchannel Flows with Realistic Boundaries," Masters Thesis, San Jose State University.
- [18] White, J. W., A. Nigam (1980), "A Factored Implicit Scheme for the Numerical Solution of the Reynolds Equation at Very Low Spacing," J. of Lubrication Technology, 102, 80-85.

Appendix A

DIRECT SIMULATION MONTE CARLO METHOD SOURCE CODE

```

program SPDSMC
!
! Program to simulate a dilute gas using DSMC algorithm in 2 dimensions
! 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
! This program adjusts the flux of particles so that the pressure
! at the ends matches the reservoir pressure
! This program uses MPI
! Can use C-L scattering
!
! *** PARAMETERS ***
!
!   include 'param.txt'
!
! *** INCLUDES ***
!
!   mpich
!   include '/usr/local/mpi/include/mpif.h'
!   epcc mpi
!   include '/usr/include/mpp/mpif.h'
!
! *** EXTERNAL ***
!
! *** REAL ARRAYS ***
!
!   real*8 x(npmax), v(npmax,3), y(npmax), vrmx(ncmax,ncmax)
!   real*8 xl_out(nptmax), yl_out(nptmax)
!   real*8 xr_out(nptmax), yr_out(nptmax)
!   real*8 vll_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 density(2), phi(2), bval(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
!   real*8 Amp, Np, knum, lambda, h_val, ywall1, ywall2
!   real*8 P_ratio
!
! *** INTEGERS ***
!
!   integer*8 nxcell, nycell, icell, jcell, xgrid, ygrid
!   integer*8 nxcellsys, nycellsys, xgridsys, ygridsys
!   integer*8 nsamp, npart, npart_tot, col, iseed, workers, restart
!   integer*8 crosstop, cross, cross1, cross2
!   integer*8 xgridsys_old, ygridsys_old
!   integer*8 ierr, master
!   integer*8 status(MPI_STATUS_SIZE)
!
! *** CHARACTER ***
!
!   character*12 readfile
!
! *** COMMON ***
!
!   include 'common.txt'
!   common /proc_nums/ myid, 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 /block12/ istep      ! use only if particles escape
!
! *****
!
!   pi = 4.*atan(1.)      ! = 3.14159...
!   master = 0
!
!   Initialize MPI
!   CALL MPI_INIT(ierr)
!   Discovery our identity
!   CALL MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
!   CALL MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs , ierr )
!
!   PROCESSOR 0 INITIALIZATION
!
!   if ( myid .eq. 0 ) then
!     readfile = 'testset.inp'
!     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,*) nxcellsys, nycellsys ! particle cell structure
!       read(11,*) xgridsys, ygridsys ! statistical cell structure
!       read(11,*) Lx2sys, h_val, Amp ! x-dim length, wall separation,
!                                     wave amp
!       read(11,*) Np           ! # of periods
!       read(11,*) phi(1), phi(2) ! phase lower wall, phase upper wall
!       read(11,*) xgridsys_old, ygridsys_old ! Values of old run
!       read(11,*) P_ratio      ! P_left/P_right and P_right = latm
!     close( unit=11 )
!

```

```

workers = nprocs - 1                ! number of workers
master = 0
if(mod(nxcellsys,workers).ne.0 .or.
  & mod(xgridsys,workers).ne.0) then
  write(*,*) 'ERROR: # of xcells / (nprocs-1) must = integer'
  stop
end if
write(*,*) 'Particles/Cell = ',npartsys/nxcellsys/nycellsys
write(*,*) 'Particles/Stat_Cell = ',
  & npartsys/xgridsys/ygridsys
Lx1sys = 0.
Ly2sys = h_val + 2*Amp
Ly1sys = Ly2sys
Lzsys = Lx2sys                      ! Arbitrary z depth
lambda = Lx2sys/Np
knum = 2*pi/lambda
end if

!
!
! Broadcast global variables to workers
! No receive is needed when using MPI_BCAST.
  CALL MPI_BCAST( restart, 1, MPI_INTEGER, 0,
    & MPI_COMM_WORLD, ierr )
  CALL MPI_BCAST( npartsys, 1, MPI_INTEGER, 0,
    & MPI_COMM_WORLD, ierr )
  CALL MPI_BCAST( nstep, 1, MPI_INTEGER, 0,
    & MPI_COMM_WORLD, ierr )
  CALL MPI_BCAST( nxcellsys, 1, MPI_INTEGER, 0,
    & MPI_COMM_WORLD, ierr )
  CALL MPI_BCAST( nycellsys, 1, MPI_INTEGER, 0,
    & MPI_COMM_WORLD, ierr )
  CALL MPI_BCAST( xgridsys, 1, MPI_INTEGER, 0,
    & MPI_COMM_WORLD, ierr )
  CALL MPI_BCAST( ygridsys, 1, MPI_INTEGER, 0,
    & MPI_COMM_WORLD, ierr )
  CALL MPI_BCAST( Lx1sys, 1, MPI_REAL8, 0,
    & MPI_COMM_WORLD, ierr )
  CALL MPI_BCAST( Ly1sys, 1, MPI_REAL8, 0,
    & MPI_COMM_WORLD, ierr )
  CALL MPI_BCAST( Lx2sys, 1, MPI_REAL8, 0,
    & MPI_COMM_WORLD, ierr )
  CALL MPI_BCAST( Ly2sys, 1, MPI_REAL8, 0,
    & MPI_COMM_WORLD, ierr )
  CALL MPI_BCAST( Lzsys, 1, MPI_REAL8, 0,
    & MPI_COMM_WORLD, ierr )
  CALL MPI_BCAST( h_val, 1, MPI_REAL8, 0,
    & MPI_COMM_WORLD, ierr )
  CALL MPI_BCAST( Amp, 1, MPI_REAL8, 0,
    & MPI_COMM_WORLD, ierr )
  CALL MPI_BCAST( knum, 1, MPI_REAL8, 0,
    & MPI_COMM_WORLD, ierr )
  CALL MPI_BCAST( phi, 2, MPI_REAL8, 0,
    & MPI_COMM_WORLD, ierr )
  CALL MPI_BCAST( xgridsys_old, 1, MPI_INTEGER, 0,
    & MPI_COMM_WORLD, ierr )
  CALL MPI_BCAST( ygridsys_old, 1, MPI_INTEGER, 0,
    & MPI_COMM_WORLD, ierr )
  CALL MPI_BCAST( P_ratio, 1, MPI_REAL8, 0,
    & MPI_COMM_WORLD, ierr )

!
!
  if ( myid .eq. 0 ) then

```



```

        if (restart.ne.0) then
            call starter(xgridsys_old, ygridsys_old, 1.0, 1.0, 1.0, 1.0,
&             1.0, 1.0, 1.0, 1.0, 1.0, restart, Amp, knum, phi, iseed)
        end if
    end if
!   WORKERS RECEIVE DATA AND INITIALIZE
!   No receive is needed when using MPI_BCAST.
!
    if ( myid .ne. 0 ) then
        boltz = 1.3806e-23      ! Boltzmann's constant
        iseed = 1
!
!   DATA FOR AIR
!
        mass = 4.81e-26         ! Mass of argon "Air"? atom
        diam = 4.19e-10         ! Effective diameter of argon atom
        density(2) = 1.29       ! Mass density of air at STP
!
!   DATA FOR ARGON
!
        mass = 6.63e-26
        diam = 3.66e-10
        density(2) = 1.78
!
        T = 273.                ! Initial temperature (K)
        P0_right = 101.3e3      ! 1 atm
        P0_left = P_ratio*P0_right
        density(1) = P_ratio*density(2)
        workers = nprocs - 1    ! The number of worker procs.
        master = 0
!
!   EACH WORKER WILL DETERMINE ITS BOUNDARIES AND VOLUMES
!
        do k = 1,workers
            if (myid .eq. k) then
                Lx2 = (Lx2sys*k)/workers      ! x-coord of right boundary
                Lx1 = (Lx2sys*(k-1))/workers  ! x-coord of left boundary
                Ly1 = Ly1sys
                Ly2 = Ly2sys
            end if
        end do
!
        bval(2) = Ly1sys - Amp                ! vertical shift of upper wall
        bval(1) = Amp                        ! vertical shift of lower wall
!
        nxcell = nxcellsys/workers
        nyccell = nyccellsys
        xgrid = xgridsys/workers
        ygrid = ygridsys
        Lz = Lzsys
!
        Volumesys = (bval(2) - bval(1))*(Lx2sys - Lx1sys)*Lzsys
&         + (Amp*Lzsys/knum)*( cos(knum*Lx2sys - phi(1))
&         + cos(knum*Lx1sys - phi(2)) - cos(knum*Lx2sys - phi(2))
&         - cos(knum*Lx1sys - phi(1)) )      ! System volume
!
        Volume = (bval(2) - bval(1))*(Lx2 - Lx1)*Lz
&         + (Amp*Lz/knum)*( cos(knum*Lx2 - phi(1))
&         + cos(knum*Lx1 - phi(2)) - cos(knum*Lx2 - phi(2))
&         - cos(knum*Lx1 - phi(1)) )        ! Worker volume
!
        call cellvol(Lx1sys, Ly1sys, Lx2sys, Ly2sys, Lx1, Ly1,
&         Lx2, Ly2, Lz, nxcell, nyccell, Amp, knum, phi, cell_vol)

```

```

call cellvol(Lx1sys, Ly1sys, Lx2sys, Ly2sys, Lx1, Ly1,
&      Lx2, Ly2, xgrid, ygrid, Amp, knum, phi, statcell_vol)
!
mpv = sqrt(2.*boltz*T/mass)      ! most probable velocity
u0 = 0.      ! reservior velocity
u0_left = u0      ! left reservior velocity
u0_right = u0      ! right reservior velocity
v0 = 0.
w0 = 0.
eff_num = (density(2)/mass)*Volumesys/npartsys !# in each sim. particle
mfp = 1./(sqrt(2.)*pi*diam**2*density(2)/mass) ! Mean free path
if (myid .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 ',(Lx2-Lx1)/nxcell/mfp,
&      ' mean free paths in x_dir'
end if
!
!
INITIALIZE PARTICLES IN EACH WORKER
!
if (restart.ne.0) then
  call starter(xgridsys_old, ygridsys_old, Lx1, Lx2, Ly1, Ly2,
&      Lx1sys, Ly1sys, Lx2sys, Ly2sys, mass, restart,
&      Amp, knum, phi, iseed)
  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)
    ywall1 = Amp*sin(knum*x(i) - phi(1)) + Amp
    ywall2 = Amp*sin(knum*x(i) - phi(2)) + Ly1sys - Amp
    if ( y(i).lt.ywall2 .and. y(i).gt.ywall1 ) then
      do j=1,3
        temp = sqrt(-log(1.-rand(iseed))) *
&      cos(2*pi*rand(iseed))
        v(i,j) = mpv*temp
      end do
      i = i + 1
    end if
  end do
!
end if
!
!
INITIALIZE QUANTITIES TO ZERO
!
do i=1,nycell
  do j = 1,nxcell
    vrmx(i,j) = 3*mpv ! Used by collision routine "collide"
    selxtra(i,j) = 0. ! Used by collision routine "collide"
  end do
end do
do j=1,2
  dtvtot(j) = 0. ! Total momentum change at wall
  dverr(j) = 0. ! Used to estimate error in dtvtot
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 (myid.eq.1 .or. myid.eq.workers) then
  pct = 0.1/10      ! Percent of system Length
  xllit = 0.        ! Left testing coords
  xlb主 = pct*Lx2sys
  if (xlb主.gt.Lx2) then
    xlb主 = Lx2
  end if
  xrlit = Lx2sys*(1.-pct) ! Right testing coords
  if (xrlit.lt.Lx1) then
    xrlit = Lx1
  end if
  xrb主 = Lx2sys

!
  vol_left = (bval(2) - bval(1))*(xlb主 - xllit)*Lz
  &
  & + (Amp*Lz/knum)*( cos(knum*xlb主 - phi(1))
  & + cos(knum*xllit - phi(2)) - cos(knum*xlb主 - phi(2))
  & - cos(knum*xllit - phi(1)) ) ! left edge volume

!
  vol_right = (bval(2) - bval(1))*(xrb主 - xrlit)*Lz
  &
  & + (Amp*Lz/knum)*( cos(knum*xrb主 - phi(1))
  & + cos(knum*xrlit - phi(2)) - cos(knum*xrb主 - phi(2))
  & - cos(knum*xrlit - phi(1)) ) ! right edge volume

!
  u0_old_left = u0_left
  u0_old_right = u0_right
  ipress = 0 ! index used to record test pressure history
end if

!
end if

!
CHECK HOW MANY PARTICLES EACH PROC IS STARTING

```

```

!
! CALL MPI_REDUCE( npart,npart_tot,1,MPI_INTEGER,
& MPI_SUM,0,MPI_COMM_WORLD,ierr)
! if (myid .eq. 0) then
!   write (*,*) 'Total number of initial particles: ',npart_tot
! end if
!
! STOP HERE FOR NOW TO TEST
! STOP
!
! BEGIN MAIN ROUTINE: ALL PROCESSORS *****
!
! nstep = 1
! if(myid.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 (myid.eq.workers) then
!   open(unit=75, file='npart.dat')
! end if
!
! if (myid .ne. 0) then
!
!   do istep = 1,nstep
!
!     DO WORKERS PARTICLE SIMULATION
!
!     if (myid .ne. 0) then
!
!       time = tau
!       CALL movem6(mpv, time, strikes, delv, iseed,
& Lx1sys, Ly1sys, Lx2sys, Ly2sys, Lzsys, mass,
& u0_left, u0_right, v0, w0, FluxOut, density, T,
& Amp, knum, phi)
!
!       SEND OUT LIST OF PARTICLES THAT HAVE LEFT WORKER SYSTEM
!
!       SEND TO RIGHT NEIGHBORS
!
!       if (myid.lt.nprocs-1 ) then
!         CALL MPI_SEND(nr_out,1,MPI_INTEGER,myid+1,11,
& MPI_COMM_WORLD,ierr)
!         CALL MPI_SEND(xr_out,nr_out,MPI_REAL8,myid+1,12,
& MPI_COMM_WORLD,ierr)
!         CALL MPI_SEND(yr_out,nr_out,MPI_REAL8,myid+1,13,
& MPI_COMM_WORLD,ierr)
!         CALL MPI_SEND(v1r_out,nr_out,MPI_REAL8,myid+1,14,
& MPI_COMM_WORLD,ierr)
!         CALL MPI_SEND(v2r_out,nr_out,MPI_REAL8,myid+1,15,
& MPI_COMM_WORLD,ierr)
!         CALL MPI_SEND(v3r_out,nr_out,MPI_REAL8,myid+1,16,
& MPI_COMM_WORLD,ierr)
!       end if
!
!       SEND TO LEFT NEIGHBORS
!
!       if (myid.gt.1) then
!         CALL MPI_SEND(nl_out,1,MPI_INTEGER,myid-1,17,
& MPI_COMM_WORLD,ierr)

```

```

CALL MPI_SEND(xl_out,nl_out,MPI_REAL8,myid-1,18,
& MPI_COMM_WORLD,ierr)
CALL MPI_SEND(yl_out,nl_out,MPI_REAL8,myid-1,19,
& MPI_COMM_WORLD,ierr)
CALL MPI_SEND(v1l_out,nl_out,MPI_REAL8,myid-1,20,
& MPI_COMM_WORLD,ierr)
CALL MPI_SEND(v2l_out,nl_out,MPI_REAL8,myid-1,21,
& MPI_COMM_WORLD,ierr)
CALL MPI_SEND(v3l_out,nl_out,MPI_REAL8,myid-1,22,
& MPI_COMM_WORLD,ierr)
end if
!
! RECEIVE FROM RIGHT NEIGHBORS
!
if (myid.lt.nprocs-1 ) then
CALL MPI_RECV(nr_out,1,MPI_INTEGER,myid+1,17,
& MPI_COMM_WORLD,status,ierr)
CALL MPI_RECV(xr_out,nr_out,MPI_REAL8,myid+1,18,
& MPI_COMM_WORLD,status,ierr)
CALL MPI_RECV(yr_out,nr_out,MPI_REAL8,myid+1,19,
& MPI_COMM_WORLD,status,ierr)
CALL MPI_RECV(v1r_out,nr_out,MPI_REAL8,myid+1,20,
& MPI_COMM_WORLD,status,ierr)
CALL MPI_RECV(v2r_out,nr_out,MPI_REAL8,myid+1,21,
& MPI_COMM_WORLD,status,ierr)
CALL MPI_RECV(v3r_out,nr_out,MPI_REAL8,myid+1,22,
& MPI_COMM_WORLD,status,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 NEIGHBORS
!
if (myid.gt.1) then
CALL MPI_RECV(nl_out,1,MPI_INTEGER,myid-1,11,
& MPI_COMM_WORLD,status,ierr)
CALL MPI_RECV(xl_out,nl_out,MPI_REAL8,myid-1,12,
& MPI_COMM_WORLD,status,ierr)
CALL MPI_RECV(yl_out,nl_out,MPI_REAL8,myid-1,13,
& MPI_COMM_WORLD,status,ierr)
CALL MPI_RECV(v1l_out,nl_out,MPI_REAL8,myid-1,14,
& MPI_COMM_WORLD,status,ierr)
CALL MPI_RECV(v2l_out,nl_out,MPI_REAL8,myid-1,15,
& MPI_COMM_WORLD,status,ierr)
CALL MPI_RECV(v3l_out,nl_out,MPI_REAL8,myid-1,16,
& MPI_COMM_WORLD,status,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)   = vl_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, Lysys, cell_vol, istep) ! Sort into cells
call collide4(vrmax, selxtra, coeff, iseed,col)

!
! WRITE # OF PARTICLES IN LAST PROCESSOR
! USED TO CHECK SYSTEM EQUILIBRIUM
!

    if (myid.eq.workers .and. mod(istep,nstep/100).lt.1) then
        write (75,*) istep, npart
    end if

!
! GET PROGRESS FROM WORKER 1 and NPROCS - 1
!

    if (myid.eq.1 .and. mod(istep,nstep/100).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
    end if
    if(myid.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 (myid.eq.1) then

        call pressure(xllit,xlbig,mass,vol_left,press_left)
        u0_left = u0_old_left +
            &         (P0_left-press_left)/P0_left*0.01*mpv
        u0_old_left = u0_left
        if( mod(istep,nstep/50) .lt. 1 ) then
            ipress = ipress + 1
            pleft(ipress) = press_left      ! record pressures
            step(ipress) = istep
        end if
    end if

!
    if(myid.eq.workers) then

        call pressure(xrlit,xrbig,mass,vol_right,press_right)
        u0_right = u0_old_right -
            &         (P0_right-press_right)/P0_right*0.01*mpv
        u0_old_right = u0_right
    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 (myid .ne. 0) then
        CALL MPI_SEND(xgrid,1,MPI_INTEGER,0,30,
&                     MPI_COMM_WORLD,ierr)
        CALL MPI_SEND(ygrid,1,MPI_INTEGER,0,31,
&                     MPI_COMM_WORLD,ierr)
        CALL MPI_SEND(nxcell,1,MPI_INTEGER,0,32,
&                     MPI_COMM_WORLD,ierr)
        CALL MPI_SEND(nycell,1,MPI_INTEGER,0,33,
&                     MPI_COMM_WORLD,ierr)
        if (myid .eq. 1) then
            CALL MPI_SEND(ipress,1,MPI_INTEGER,0,34,
&                         MPI_COMM_WORLD,ierr)
            CALL MPI_SEND(pleft,ipress,MPI_REAL8,0,35,
&                         MPI_COMM_WORLD,ierr)
            CALL MPI_SEND(step,ipress,MPI_INTEGER,0,36,
&                         MPI_COMM_WORLD,ierr)
        end if
        if (myid.eq.workers) then
            CALL MPI_SEND(pright,ipress,MPI_REAL8,0,37,
&                         MPI_COMM_WORLD,ierr)
        end if
        CALL MPI_SEND(ave_n, ncmx*ncmx,MPI_REAL8,0,38,
&                     MPI_COMM_WORLD,ierr)
        CALL MPI_SEND(ave_u, ncmx*ncmx*3,MPI_REAL8,0,39,
&                     MPI_COMM_WORLD,ierr)
        CALL MPI_SEND(ave_T, ncmx*ncmx,MPI_REAL8,0,40,
&                     MPI_COMM_WORLD,ierr)
        CALL MPI_SEND(ave_pres, ncmx*ncmx,MPI_REAL8,0,41,
&                     MPI_COMM_WORLD,ierr)
        CALL MPI_SEND(statcell_vol, ncmx*ncmx,MPI_REAL8,0,
&                     42,MPI_COMM_WORLD,ierr)
        CALL MPI_SEND(cell_vol, ncmx*ncmx,MPI_REAL8,0,43,
&                     MPI_COMM_WORLD,ierr)
        CALL MPI_SEND(xcell, xgrid,MPI_REAL8,0,44,
&                     MPI_COMM_WORLD,ierr)
        if (myid .eq. 1) then
            CALL MPI_SEND(ycell, ygrid,MPI_REAL8,0,45,
&                         MPI_COMM_WORLD,ierr)
        end if
        CALL MPI_SEND(force(2),1,MPI_REAL8,0,46,
&                     MPI_COMM_WORLD,ierr)
    end if
!
! RECEIVE DATA FROM WORKERS
!
    if (myid.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 MPI_RECV(xgrid,1,MPI_INTEGER,id,30,
&                MPI_COMM_WORLD,status,ierr)
    CALL MPI_RECV(ygrid,1,MPI_INTEGER,id,31,
&                MPI_COMM_WORLD,status,ierr)
    CALL MPI_RECV(nxcell,1,MPI_INTEGER,id,32,
&                MPI_COMM_WORLD,status,ierr)
    CALL MPI_RECV(nycell,1,MPI_INTEGER,id,33,
&                MPI_COMM_WORLD,status,ierr)
    if (id .eq. 1) then
        CALL MPI_RECV(ipress,1,MPI_INTEGER,id,34,
&                    MPI_COMM_WORLD,status,ierr)
        CALL MPI_RECV(pleft,ipress,MPI_REAL8,id,35,
&                    MPI_COMM_WORLD,status,ierr)
        CALL MPI_RECV(step,ipress,MPI_INTEGER,id,36,
&                    MPI_COMM_WORLD,status,ierr)
    end if
    if (id.eq.workers) then
        CALL MPI_RECV(pright,ipress,MPI_REAL8,id,37,
&                    MPI_COMM_WORLD,status,ierr)
    end if
    CALL MPI_RECV(ave_n, ncmax*ncmax,MPI_REAL8,id,38,
&                MPI_COMM_WORLD,status,ierr)
    CALL MPI_RECV(ave_u, ncmax*ncmax*3,MPI_REAL8,id,39,
&                MPI_COMM_WORLD,status,ierr)
    CALL MPI_RECV(ave_T, ncmax*ncmax,MPI_REAL8,id,40,
&                MPI_COMM_WORLD,status,ierr)
    CALL MPI_RECV(ave_pres, ncmax*ncmax,MPI_REAL8,id,41,
&                MPI_COMM_WORLD,status,ierr)
    CALL MPI_RECV(statcell_vol, ncmax*ncmax,MPI_REAL8,id,
&                42,MPI_COMM_WORLD,status,ierr)
    CALL MPI_RECV(cell_vol, ncmax*ncmax,MPI_REAL8,id,43,
&                MPI_COMM_WORLD,status,ierr)
    CALL MPI_RECV(xcell, xgrid,MPI_REAL8,id,44,
&                MPI_COMM_WORLD,status,ierr)
    if (id .eq. 1) then
        CALL MPI_RECV(ycell, ygrid,MPI_REAL8,id,45,
&                    MPI_COMM_WORLD,status,ierr)
    end if
    CALL MPI_RECV(force(2),1,MPI_REAL8,id,46,
&                MPI_COMM_WORLD,status,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
!
        CALL MPI_FINALIZE(ierr)
        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, A, k, phi, cell_vol)
!
! This routine calculates the volume of all cells.
! *** INDIVIDUAL CELLS MAY INTERSECT BOTH WALLS ***
! 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, max_vol, phi(2), bval(2), A, k
      real*8 x_old, y_old, x_new, y_new, pi, direction
      real*8 y_old_bot, y_new_bot, sum_vol_bot
      real*8 ywall2test, ywall1test, yltop, y2bot, count
      real*8 dx, dy, dy_old, dy_new, lambda
      real*8 dy_old_bot, dy_new_bot
      real*8 vol, xint, yint, theta, sum_vol
      real*8 xin1, xin2
      integer nxcell, nycell, i, j, flag(2), flag2, chop_top
      integer xstep, ystep, max_step, min_step
!
      max_vol = Ly1sys*(Lx2-Lx1)*Lz/nycell/nxcell
      min_step = 100
      bval(1) = A
      bval(2) = Ly1sys - A
      pi = 4*atan(1.)
      lambda = 2*pi/k
      y2bot = Ly1sys - 2*A
      yltop = 2*A
!
!      open (unit=14,file='flag.out')
      if (Lx2sys/nxcell .gt. lambda) then
        max_step = min_step*(Lx2sys/nxcell)/lambda
      else
        max_step = min_step
      end if
!
      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
!          chop_top = 0
!          flag(1) = 0
!          flag(2) = 0
!
!          if (j.eq.26 .and. i.eq.1) then
!            flag = 0
!          end if
!
!          if (y4.le.y2bot .and. y1.ge.yltop) then
!            cell_vol(i,j) = max_vol ! cell intersects no walls
!          else
!
!            if (y4.gt.y2bot) then
!              flag = 2 ! cell may cross upper wall

```

```

!           direction = -1.
!           flag(1) = 2      ! cell may cross upper wall
!           direction(1) = -1.
!       end if
!       else if (y1.lt.yltop) then
!       if (y1.lt.yltop) then
!           flag = 1      ! cell may cross lower wall
!           direction = 1.
!           flag(2) = 1      ! cell may cross lower wall
!           direction(2) = 1.
!       end if
!
!       write(14,*) , flag(1), flag(2)
!       dx = (x2 - x1)/max_step
!       sum_vol = 0
!
!       do xstep = 1, max_step
!
!           x_old = x1 + (xstep-1)*dx
!           x_new = x_old + dx
!           dy = y4 - y1
!
!           if( flag(1).eq.0 .or. flag(2).eq.0 ) then
!
!               if(flag(1) .eq. 2) then
!                   flag2 = 2
!                   direction = -1
!               else if(flag(2) .eq. 1) then
!                   flag2 = 1
!                   direction = 1
!               end if
!
!               y_old = A*sin(k*x_old - phi(flag2)) + bval(flag2)
!               y_new = A*sin(k*x_new - phi(flag2)) + bval(flag2)
!               dy_old = y_old - y1
!               dy_new = y_new - y1
!
!               if (y_old.ge.y4 .and. y_new.ge.y4) then
!                   vol = dx*dy
!               else if (y_old.le.y1 .and. y_new.le.y1) then
!                   vol = 0
!                   chop_top = 1
!               else if (y_old.le.y4 .and. y_new.le.y4) then
!                   vol = 0.5*(dy_old + dy_new)*dx
!                   chop_top = 1
!               else
!
!                   chop_top = 1
!
!               if (flag2.eq.2) then
!                   if (y_old.gt.y4 .or. y_new.lt.y1) then
!                       xin1 = x1
!                       xin2 = x2
!                   else if (y_new.gt.y4 .or. y_old.lt.y1) then
!                       xin1 = x2
!                       xin2 = x1
!                   end if
!               else if (flag2.eq.1) then
!                   if (y_new.gt.y4 .or. y_old.lt.y1) then
!                       xin1 = x1
!                       xin2 = x2
!                   else if (y_old.gt.y4 .or. y_new.lt.y1) then
!                       xin1 = x2

```

```

        xin2 = x1
        end if
    end if

    if (y_old.gt.y4 .or. y_new.gt.y4) then

        call intfind(xin1, y4, xin2, y4,
            & A, k, Lylsys, direction, phi(flag2),
            & flag2, xint, yint, theta)

        else if (y_old.lt.y1 .or. y_new.lt.y1) then

            call intfind(xin1, y1, xin2, y1,
                & A, k, Lylsys, direction, phi(flag2),
                & flag2, xint, yint, theta)

        end if

        if (y_old.gt.y4) then

            vol = (xint - x_old)*dy +
                & (x_new - xint)*0.5*(dy_new + dy)

            else if (y_new.gt.y4) then

                vol = 0.5*(dy_old + dy)*(xint - x_old) +
                    & (x_new - xint)*dy

            else if (y_old.lt.y1) then

                vol = 0.5*(dy_new)*(x_new - xint)

            else if (y_new.lt.y1) then

                vol = 0.5*(dy_old)*(xint - x_old)

            end if

        end if

        sum_vol = sum_vol + vol

    else ! cell intersects both walls
        y_old = A*sin(k*x_old - phi(2)) + bval(2) !top
        y_old_bot = A*sin(k*x_old - phi(1)) + bval(1)
        y_new = A*sin(k*x_new - phi(2)) + bval(2) !top
        y_new_bot = A*sin(k*x_new - phi(1)) + bval(1)
        dy_old = y_old - y1
        dy_old_bot = y_old_bot - y1
        dy_new = y_new - y1
        dy_new_bot = y_new_bot - y1

        if(y_old.ge.y4) then
            dy_old = y4 - y1
        end if
        if(y_new.ge.y4) then
            dy_new = y4 - y1
        end if
        if( y_old_bot.ge.y4 .and. y_new_bot.ge.y4) then
            vol = 0
        else if (y_old.le.y1 .and. y_new.le.y1) then
            vol = 0
        end if
    end if

```

```

        else if(y_old.ge.y4 .and. y_new.ge.y4 .and.
&          y_old_bot.le.y1 .and. y_new_bot.le.y1) then
          vol = dx*(y4-y1)
        else if(y_old.le.y4 .and. y_new.le.y4 .and.
&          y_old_bot.ge.y1 .and. y_new_bot.ge.y1) then
          vol = 0.5 * dx * ( (y_old - y_old_bot) +
&            (y_new - y_new_bot) )
        else
          if(y_old.gt.y4) then
            y_old=y4
          end if
          if( y_new.gt.y4) then
            y_new=y4
          end if
          if(y_old_bot.lt.y1) then
            y_old_bot=y1
          end if
          if(y_new_bot.lt.y1) then
            y_new_bot=y1
          end if
          vol = 0.5 * dx * ( (y_old - y_old_bot) +
&            (y_new - y_new_bot) )
        end if

        sum_vol=sum_vol+vol

      end if

    end do

  end if

  if (flag(1).eq.0 .or. flag(2).eq.0) then
    if (flag(1).eq.2 .and. flag(2).eq.0) then
      cell_vol(i,j) = sum_vol*Lz
    else if (flag(2).eq.1 .and. chop_top.eq.1) then
      cell_vol(i,j) = max_vol - sum_vol*Lz
    else if (flag(2).eq.1 .and. chop_top.eq.0) then
      cell_vol(i,j) = 0.
    end if
    else if(flag(1).eq.2 .and. flag(2).eq.1) then
      cell_vol(i,j)=sum_vol*Lz
    end if

  end do

end do

close(unit=14)
return
end

```

```

subroutine collide(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*4 Xref(npmax), index(ncmax,ncmax), cell_n(ncmax,ncmax)
  integer*4 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

```



```
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

```

```

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 gridsmp(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*4 Xref(npmax), index(ncmax,ncmax), cell_n(ncmax,ncmax)
integer*4 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 intfind( x1, y1, x2, y2, A, k, Ly, direction,
&                  phase, flag, xint, yint, theta )
!
!   THIS SUBROUTINE WILL FIND THE INTERSECTION OF A LINE
!   SEGMENT AND A SINE WAVE.
!   XINT AND YINT ARE THE COORDS OF THE INTERSECTION
!   THETA IS THE ANGLE FROM THE Y-AXIS TO THE SURFACE
!   NORMAL WITH CCW BEING POSITVE
!
!   *** REAL VARIABLES ***
!
!       real*8 x1, y1, x2, y2, A, k, direction, phase
!       real*8 xint, yint, theta, xm, ym, ywall, Ly
!
!   *** INTEGER VARIABLES ***
!
!       integer*4 i, flag
!
!
!   ITERATE BISECTION 10 TIMES
!
!   do i = 1,100
!       xm = 0.5*(x1 + x2)
!       ym = 0.5*(y1 + y2)
!       if (flag.eq.1) then
!           ywall = A*sin(k*xm - phase) + A
!       else if (flag.eq.2) then
!           ywall = A*sin(k*xm - phase) + Ly - A
!       end if
!       if ( direction*(ym - ywall).gt.0 ) then
!           x1 = xm
!           y1 = ym
!       else
!           x2 = xm
!           y2 = ym
!       end if
!   end do
!
!   xint = xm
!   yint = ym
!
!   theta = atan( A*k*cos(k*xm - phase) )
!   &      + (1 - direction)*6.283185307/4
!
!   return
!   end

```

```

subroutine maker(rho0,u0,v0,w0,xface,nface,FluxOut,MethodFlag
& ,mpv,pmass,tau,ymin,ymax,Lz,iseed,Lx1,Ly1,Lx2,Ly2,
& 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)
      real*8 delv(2),Lx1,Ly1,Lx2,Ly2,Lz
      integer*4 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,ymin,ymax
      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)
      yface = ymax - ymin
!
!! 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 + minl(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
      vl = 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) + ymin ! assign random entry height
      vx_old = vl
!
      x_new = x_old + vl*dtr
      y_new = y_old + v2*dtr
      call strikem4(x_old,y_old,vx_old,x_new,y_new,v1,v2,v3,dtr,
& 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) = vl
        v(npart,2) = v2
        v(npart,3) = v3
!
        CALCULATE FLUXES
!
        FluxOut(1) = FluxOut(1) + 1
        FluxOut(2) = FluxOut(2) + vl
        FluxOut(3) = FluxOut(3) + v2
        FluxOut(4) = FluxOut(4) + v3
        FluxOut(5) = FluxOut(5) + 0.5*(vl**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 movem(mpv, tau, strikes, delv, iseed,
&               Lx1sys, Ly1sys, Lx2sys, Ly2sys, Lzsys, mass,
&               u0_left, u0_right, v0, w0, FluxOut, density, T,
&               A, knum, phi)
!
!   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)
!   real*8 xl_out(nptmax), yl_out(nptmax), phi(2)
!   real*8 xr_out(nptmax), yr_out(nptmax)
!   real*8 vll_out(nptmax), v2l_out(nptmax), v3l_out(nptmax)
!   real*8 vlr_out(nptmax), v2r_out(nptmax), v3r_out(nptmax)
!   real*8 delv(2), FluxOut(5), height(2), density(2)
!
! *** REAL VARIABLES ***
!
!   real*8 mpv, tau, time, mass, A, knum
!   real*8 x_old, y_old
!   real*8 Lx1sys, Ly1sys, Lx2sys, Ly2sys, Lzsys
!
! *** INTEGER ARRAYS ***
!
!   integer*4 strikes(2)
!
! *** INTEGER VARIABLES ***
!
!   integer*4 keep, repeat, iseed
!
! *** COMMON VARIABLES ***
!
!   include 'common.txt'
!   common /proc_nums/ mypid, nprocs
!   common /pass/ xl_out, yl_out, xr_out, yr_out,
&               vll_out, v2l_out, v3l_out,
&               vlr_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
!   ylmin = A*sin(-phi(1)) + A
!   ylmax = A*sin(-phi(2)) + Ly1sys - A
!   yrmin = A*sin(knum*Lx2sys - phi(1)) + A
!   yrmax = A*sin(knum*Lx2sys - phi(2)) + Ly1sys - A
!   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(1), u0_left, v0, w0, Lx1sys, 1, Fluxout,
&               1, mpv, mass, time, ylmin, ylmax, Lzsys, iseed, Lx1sys,
&               Ly1sys, Lx2sys, Ly2sys, strikes, delv)
!   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)
        vl_out(nl_out) = v(i,1)
        vl_out(nl_out) = v(i,2)
        vl_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)
        vl_out(nr_out) = v(i,1)
        vl_out(nr_out) = v(i,2)
        vl_out(nr_out) = v(i,3)
!
    end if
!
    x(i) = x(npart) ! Replace particle with particle at end of list
    y(i) = y(npart) ! This destroys the particle
    y(i) = y(npart) ! keep = 0 or -1
    do k = 1,3
        v(i,k) = v(npart,k)
    end do
    npart = npart - 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*4 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 sinehit(x_old, y_old, x, y, v1, v2, v3, time,
&                Lx1sys, Ly1sys, Lx2sys, Ly2sys,
&                mpv, strikes, delv, iseed, keep, T, mass,
&                A, k, phi)
!
!   This subroutine handles when a particle strikes the wall.  Its 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 delv(2), v_in(3)
!   real*8 v_out(3), phi(2), bval(2)
!
! *** INTEGER ARRAYS ***
!
!   integer*4 strikes(2)
!
! *** REAL VARIABLES ***
!
!   real*8 x, y, v1, v2, v3, v_par11, time, mpv, m
!   real*8 dtb, dtr, x_old, y_old, vx_old, mass
!   real*8 Lx1sys, Lx2sys, Ly1sys, Ly2sys, xint, yint, rand
!   real*8 A, k, direction, ywall1, ywall2, ywall1_old, ywall2_old
!   real*8 theta, vy_old, y2bot, yltop
!   real*8 direction_new, direction_old
!   real*8 slope, dist, xtest, ytest, ywall, err, slope_max
!
! *** INTEGER VARIABLES ***
!
!   integer*4 flag, repeat, tag_new, tag_old, check_both
!   integer*4 iseed, testpath, testmax
!
! *** COMMON VARIABLES ***
!
!   common /block11/ npart_leave
!   common /proc_nums/ mypid, nprocs
!
! *** DATA ***
!
!   data twopi/6.283185307/
!
!   repeat = 1
!
!   vx_old = v1 ! remember pre-collision velocity
!   vy_old = v2
!
!   bval(1) = A ! vertical center of lower wall
!   bval(2) = Ly1sys - A ! and for upper wall
!
!   pmaxwell = 1. ! Used for maxwell accomodation
!
!   testmax = 20 ! the number of intervals for breaking up path
!
!   err = 1e-12 ! roundoff error bound
!
!   slope_max = A*k ! max particle slope for path check

```



```

!
! do while(repeat.lt.100)
!
! When a particle strikes the wall it is moved off the wall
! 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
!
!   flag = 0
!   tag = 0
!   tag_new = 0
!   tag_old = 0
!   check_both = 0
!   ywall1 = A*sin(k*x - phi(1)) + bval(1)           ! y-coord for lower
wall ywall2 = A*sin(k*x - phi(2)) + bval(2)           ! y-coord for upper wall
!   y_old-coord for lower wall
!   ywall1_old = A*sin(k*x_old - phi(1)) + bval(1)
!   y_old-coord for upper wall
!   ywall2_old = A*sin(k*x_old - phi(2)) + bval(2)
!
!   y2bot = Lylsys - 2*A
!   yltop = 2*A
!
!   Check to see if both walls should be checked for strikes
!   if ( (y2bot.ge.yltop) .or. (0.9*yltop .le. y2bot) ) then
!     check_both = 1
!   end if
!
!   if ( y.ge.ywall2 ) then
!     flag = 2           ! Particle may have struck upper wall
!     direction = -1
!   else if ( y.le.ywall1 ) then
!     flag = 1           ! Particle may have struck lower wall
!     direction = 1
!   end if
!
!   if ( y.gt.y2bot ) then
!     tag_new = 2           ! particle path is in upper wall area
!     direction_new = -1
!   else if ( y.lt.yltop ) then
!     tag_new = 1           ! particle path is in lower wall area
!     direction_new = 1
!   end if
!
!   if ( y_old.gt.y2bot ) then
!     tag_old = 2           ! Particle path is in upper wall area
!     direction_old = -1
!   else if ( y_old.lt.yltop ) then
!     tag_old = 1           ! Particle path is in lower wall area
!     direction_old = 1
!   end if
!
!   if ( y.gt.y2bot .or. y.lt.yltop ) then
!     if ((abs( y-ywall1)) .gt. (abs(y-ywall2)) ) then
!       write(*,*) 'particle near top wall',x,y,ywall2,ywall1
!       tag_new = 2           ! Particle path is near upper wall
!       direction_new = -1
!     else
!       write(*,*) 'particle near bottom wall',x,y,ywall1,ywall2
!       tag_new = 1           ! Particle path is near lower wall

```

```

        direction_new = 1
    end if
!
end if
!
if ( y_old.gt.y2bot .or. y_old.lt.y1top) then
!
    if ((abs( y_old-ywall1_old)) .gt. (abs(y_old-ywall2_old))) then
!
        write(*,*) 'particle near old top wall',x_old,y_old,
!
        & ywall2,ywall1
        tag_old = 2      ! Particle path is near upper wall
        direction_old = -1
    else if ((abs( y_old-ywall1_old)) .lt.
!
        & (abs(y_old-ywall2_old))) then
!
        write(*,*) 'particle near old bottom wall',x_old,y_old,
!
        & ywall1,ywall2
        tag_old = 1      ! Particle path is near lower wall
        direction_old = 1
    end if
!
end if
!
if ( flag.eq.0 .and. (tag_new.eq.0 .and. tag_old.eq.0) ) then
!
    if (x.ge.0. .and. x.le.Lx2sys) then ! Still in system
!
        keep = 1      ! no strike possible and remains in system
        return
!
    else
!
        keep = 0 ! Particle exits entire system
        npart_leave = npart_leave + 1
        return
!
    end if
!
end if
!
CHECK FOR ESCAPED PARTICLE:  if current position AND old position
!
are outside wall, then the particle has escaped and will be punished
!
    ywall1_old = A*sin(k*x_old - phi(1)) + bval(1)
    ywall2_old = A*sin(k*x_old - phi(2)) + bval(2)
!
if ( y.ge.(ywall2+err) .and. y_old.ge.(ywall2_old+err) ) then
    keep = -1      ! particle escaped upper wall
    write(*,*) 'Particle has escaped upper wall, oh shit!'
!
    write(*,*) 'Particle data y:',y,ywall2,y_old,ywall2_old
!
    write(*,*) 'Particle data x:',x,x_old,repeat
!
    write(*,*) 'Flags: ',flag,direction,tag_new,direction_new,
!
    & tag_old,direction_old
    return
else if (y.le.(ywall1-err).and.y_old.le.(ywall1_old-err)) then
    keep = -1      ! particle escaped lower wall
    write(*,*) 'Particle has escaped lower wall, get back here!'
!
    write(*,*) 'Particle data y:',y,ywall1,y_old,ywall1_old
!
    write(*,*) 'Particle data x:',x,x_old,repeat
!
    write(*,*) 'Flags: ',flag,direction,tag_new,direction_new,
!
    & tag_old,direction_old
    return
end if
!

```

```

PARTICLE MAY HAVE STRUCK AND IS STILL IN SYSTEM:
tag_new and tag_old = 1, or 2      and      flag = 0, 1, or 2

If the slope of the path is low enough, the particle can chop the
peak of a wall. Break the path into segments and test for a strike
from the oldest to newest path segments

      if (x_old.ne.x) then
        slope = (y - y_old)/(x - x_old)
        dist = (x - x_old)/testmax      ! segment lengths to test
      else
        slope = slope_max      ! impossible to chop wall
      end if

      if (abs(slope).ge.slope_max) then

        if ( flag.eq.0 ) then      ! unlikely to chop wall at this angle

          if ( x.ge.0. .and. x.le.Lx2sys ) then
            keep = 1      ! keep particle and return
            return
          else
            keep = 0      ! particle has exited
            return
          end if

        else
          testpath = testmax + 1      ! don't break up path
          xtest = x
          ytest = y
        end if

      else

        flag = 0      ! Reset to find strike
        testpath = 1      ! proceed with breakup
        xtest = x_old      ! start at beginning of path
        ytest = y_old

      end if

      do while(testpath.le.testmax)

        xtest = xtest + dist
        ytest = ytest + slope*dist

        if ( tag_old.ne.0 ) then      ! test wall near old end of path

          ywall = A*sin(k*xtest - phi(tag_old)) + bval(tag_old)

          if ( direction_old*(ytest - ywall).lt.0 ) then
            flag = tag_old      ! particle has indeed struck wall
            direction = direction_old
          end if

          if (check_both .eq. 1) then

            if (tag_old .eq. 1) then ! check other wall
              ywall = A*sin(k*xtest - phi(2)) + bval(2)

              if ( -1.0*(ytest - ywall).lt.0 ) then
                flag = 2      ! particle has indeed struck wall
              end if
            end if
          end if
        end if
      end do
    end if
  end if
end if

```

```

        direction = -1
    end if
else
    ywall = A*sin(k*xtest - phi(1)) + bval(1)
!
    if ( 1.0*(ytest - ywall).lt.0 ) then
        flag = 1 ! particle has indeed struck wall
        direction = 1
    end if
end if
!
end if
!
end if
!
if ( tag_new.ne.0 .and. flag.eq.0 ) then ! test wall near new end
!
    ywall = A*sin(k*xtest - phi(tag_new)) + bval(tag_new)
!
    if ( direction_new*(ytest - ywall).lt.0 ) then
        flag = tag_new ! particle has indeed struck wall
        direction = direction_new
    end if
!
    if (check_both .eq. 1) then
!
        if (tag_new .eq. 1) then ! check other wall
            ywall = A*sin(k*xtest - phi(2)) + bval(2)
!
            if ( -1.0*(ytest - ywall).lt.0 ) then
                flag = 2 ! particle has indeed struck wall
                direction = -1
            end if
        else
            ywall = A*sin(k*xtest - phi(1)) + bval(1)
!
            if ( 1.0*(ytest - ywall).lt.0 ) then
                flag = 1 ! particle has indeed struck wall
                direction = 1
            end if
        end if
    end if
!
end if
!
end if
!
if ( flag.eq.0 ) then
    testpath = testpath + 1 ! no strike found, continue test
else
    testpath = testmax + 1 ! strike found, stop testing
end if
!
end do
!
if (flag.eq.0) then
!
    if ( x.ge.0. .and. x.le.Lx2sys ) then
        keep = 1 ! no strike found so keep and return
        return
    else
        keep = 0 ! particle has left system
        return
    end if
!

```

```

else      ! strike was found, now find intersection point
!
      x2 = xtest
      y2 = ytest
      x1 = x_old
      y1 = y_old
      call intfind( x1, y1, x2, y2, A, k, Lylsys, direction,
&          phi(flag), flag, xint, yint, theta )
!
      if (flag .eq. 1) then
!         write(*,*) 'IF_Input',x1, y1, x2,y2
!         write(*,*) 'Lower wall Intfind',xint, yint, theta
!       else
!         write(*,*) 'IF_Input',x1, y1, x2,y2
!         write(*,*) 'Upper wall Intfind',xint, yint, theta
!       end if
!
      end if
!
      if ((xint.lt.0.).or.(xint.gt.Lx2sys)) then
        keep = 0 ! particle has left system before striking wall
        npart_leave = npart_leave + 1
        return
      else
        strikes(flag) = strikes(flag) + 1
      end if
!
      test = rand(iseed) ! Used to draw prob dist. for maxwell
!       If test < pmaxwell use thermal scattering (pmaxwell prob)
!       If test > pmaxwell use specular scattering
!
      MAXWELL ACCOMODATION
!
      if (test.gt.pmaxwell) then ! use specular
        call specang( v1, v2, theta, delv(flag) )
      else ! use thermal
        call thwall( v1, v2, v3, theta, mpv, delv(flag), iseed )
      end if
!
      Dtr = time of flight after leaving wall
      MOVE PARTICLE FOR REMAINING TIME STEP
!
      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-yint)/(y-y_old))
      else ! slope less than 1 or -1
!         USE X VALS
        dtr = abs(time*(x-xint)/(x-x_old))
      end if
      dtb = time - dtr ! time before wall strike
!
      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
!

```

```

        keep = 1
        return
!
    end if
!
!   MOVE PARTICLE OFF WALL WITH REMAINING TIME
!
        y = yint + v2*dtr    ! Position after leaving wall
        x = xint + v1*dtr
!
!   ITERATE TO CHECK FOR ANOTHER STRIKE OR EXIT
!   RESET OLD POSITIONS AND VELOCITIES TO VALUES AFTER STRIKE
!
        x_old = xint        ! reset to wall strike position
        y_old = yint        ! wall strike position
        vx_old = v1         ! reset to post-collision velocity
        vy_old = v2
        time = dtr          ! time step reduced to remainder of previous step
!
    repeat = repeat + 1
!
end do
!
write(*,*) 'Floundering in sinehit'
write(*,*) 'Proc. Num = ',mypid
keep = 0    ! kill particle
return
!
end

```

```

subroutine sorter(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*4 Xref(npmax), index(ncmax,ncmax), cell_n(ncmax,ncmax)
  integer*4 ix(npmax), jy(npmax), temp(ncmax,ncmax),nycell, nxcell
  integer*4 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

```

```

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*4 iseed
    data twopi/6.283185307/

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

    return
end

subroutine thwall( vx, vy, vz, theta, mpv, dv, iseed )
!
! THIS ROUTINE WILL PERFORM A THERMAL COLLISION WITH
! A WALL. THE NORMAL TO THE WALL IS AT AN ANGLE
! THETA WITH RESPECT TO Y (CCW IS POSITIVE)
! ROTATION IS 2D (X AND Y)
!
! *** REAL VARIABLES ***
!
    real*8 vx, vy, vz, theta, mpv, dv, rand
    real*8 vNormal, vPar11Mag, phi
    real*8 vPar111, vPar112
    real*8 vxNew, vyNew, vzNew
!
! *** INTEGER VARIABLES
!
    integer*4 iseed
!
!
    vNormal = sqrt( -log(1 - rand(iseed)) )*mpv
    phi = 6.283185307*rand(iseed)
    vPar11Mag = sqrt( -log(1 - rand(iseed)) )*mpv
    vPar111 = vPar11Mag*sin(phi)
    vPar112 = vPar11Mag*cos(phi)
    vxNew = cos(theta)*vPar111 - sin(theta)*vNormal
    vyNew = sin(theta)*vPar111 + cos(theta)*vNormal
    vzNew = vPar112
    dv = vyNew - vy    ! track change in y-velocity
    vx = vxNew
    vy = vyNew
    vz = vzNew
!
    return
end

```


Include files:

common.txt

```

common /block1/ x
common /block2/ y
common /block3/ v
common /block4/ npart
common /block5/ nxcell
common /block6/ nycell
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*4(i-m)

```

A sample input file:

0			Restart and cloning (0 = none)
16000			Number of particles
20000			Time steps
93	8		Particle Cells: xgrids ygrids
31	24		Statistic Cells: xgrids ygrids
5.e-6	80e-9	20e-9	Platter Length (m); Wall separation; Sine Amp
2			# of periods
3.14159		0	Lower wall phase; Upper wall phase
31	24		Previous stat structure: xgrids ygrids
5.			P_left/P_right and P_right=latm
thermal walls			

Appendix B

FACTORED IMPLICIT SOLVER MATLAB SCRIPT

```
% Program to solve Reynolds equation using implicit method of
% Nigam and White and Fukui-Kaneko slip correction.
% Flow of fluid through a pipe with out of phase, sine wave walls.
clear all; help reyksc;
tic;
cd /Programs/Matlab;

% Solution constants
N      = 75;           % Number of grid points
Nsteps = 6000;         % Number of iterations
slip   = 2;           % Slip correction to use, 0 = continuum,
                     % 1 first order, 2 Fukui and Kaneko
OutputFileName='fig10nw.dat'; % Filename for pressure profile

% Walls are sine waves all values in nanometers
phase = 180 * (pi/180); % Phase difference between the upper and lower walls
Amp    = 30;           % Amplitude
Hdiff  = 80;           % Height difference between the sine waves
NumPer = 2;           % Number of Periods

%General system constants
viscosity = 2.0806e-005; % Viscosity [N*s/m^2]
len        = 5e-6;       % Length of slider [m]
Patmos     = 101325;      % Atmospheric pressure [Pa]
Vsound     = 307.8 ;     % Velocity of sound [m/s]

% Create X vector
X = (0:(N-1))/(N-1);     % Dimensionless horizontal position
X = X';                 % Convert to column vector
sX = (2*NumPer*pi)*X;    % create vector for sine wave (# of periods )
dX = X(2)-X(1);         % Grid spacing

% Create Wall vectors and normalized height vector
htop = (Amp*sin(sX) + Hdiff); % Vector for top wall
hbot = (Amp*sin(sX - phase)); % Vector for bottom wall
h = htop-hbot;              % Height difference between the two walls
H = h./h(N);               % Normalized height at exit
hnaught = min(h)*1e-9;     % Height of slider at normalization position

% Knudsen number at exit
Kn0 = 62.6/h(N);          % Knudsen number at exit mfp/h0 ,mfp =62.6 nm

% Show initial geometry
subplot(2,1,1);
plot(sX,htop,'r-',sX,hbot,'b-');
title(sprintf(' Sine pipe with phase diff. = %g deg',(phase*180/pi)));
drawnow;
subplot(2,1,2);
plot(sX,H);
title(sprintf('Normalized height variation between the walls'));
drawnow;
```

```

% Create initial Pressure vector
P = (1+0.0*rand(N,1)).*ones(N,1);          % Initial pressure
P(1) = 5;  P(N) = 1;

% Initialize Z
Z = H.*P;
Z1 = Z(1);  ZN = Z(N);

% Create normalized height matrix
ii = 2:(N-1);
ipl = ii+1;  iml = ii-1;
H_x(ii,1) = (H(ipl,1)-H(iml,1))/(2*dX);
H_x(1) = (-3*H(1) + 4*H(2) - H(3))/(2*dX);
H_x(N) = (3*H(N) - 4*H(N-1) + H(N-2))/(2*dX);

% Fukui-Kaneko parameters
A = 1.318889;
B = 0.387361;

% Grouping Parameters into constants
gam = (12*viscosity*len*Vsound)/(Patmos*hnaught^2);
gamt = 1/(2*gam);

if(slip == 0)
    beta=0;
elseif(slip == 1)
    beta=6*Kn0;
end

% Set timestep
dT = (1e-5/gamt)*2.75;          % Time step 1e-5 is the largest theoretical
stable time step                % the 2.75 is an empirical optimization

factor

% Prepare for iteration
Zold = Z;
i=1;
Change=2;
iStep=1;

% Iterate until solution has relaxed
%for iStep=1:Nsteps
while ((Change > 1) | (iStep < Nsteps))

    Z(1) = Z1;  Z(N) = ZN;      % Fix boundary conditions
    Knudsen = Kn0./Z;          % Knudsen = Kn0/H*P
    Knudsen(N) = Kn0;

    if(slip==2)
        beta = 6*a*Kn0 + (12/pi) * Kn0 * log(1 + B.*Knudsen);
    end

    Z_x(ii,1) = (Z(ipl,1)-Z(iml,1))/(2*dX);
    Z_x(1) = (-3*Z(1) + 4*Z(2) - Z(3))/(2*dX);
    Z_x(N) = (3*Z(N) - 4*Z(N-1) + Z(N-2))/(2*dX);

    C = H.*Z_x - 2*Z.*H_x - (beta).*H_x;
    D = H.*Z + (beta).*H;

    D_x(ii,1) = (D(ipl,1)-D(iml,1))/(2*dX);
    D_x(1) = (-3*D(1) + 4*D(2) - D(3))/(2*dX);

```

```

D_x(N) = (3*D(N) - 4*D(N-1) + D(N-2))/(2*dX);

% Build the matrix L
L = zeros(N,N);

L= diag( (gamt*dT/(2*dX)*(C(2:N)+D_x(2:N)) + gamt*dT/(dX^2)*D(2:N)),1) + ...
    diag( (-2*gamt*dT/(dX^2)*D),0) + ...
    diag( (-gamt*dT/(2*dX)*(C(1:N-1)+D_x(1:N-1)) + gamt*dT/(dX^2)*D(1:N-1)), -1);

L(1,1) = -gamt*dT/(dX)*(C(1)+D_x(1)) + gamt*dT/(dX^2)*D(1);
L(1,2) = gamt*dT/(dX)*(C(2)+D_x(2)) - 2*gamt*dT/(dX^2)*D(2);
L(1,3) = gamt*dT/(dX^2)*D(3);
L(N,N-2) = gamt*dT/(dX^2)*D(N-2);
L(N,N-1) = -gamt*dT/(dX)*(C(N-1)+D_x(N-1)) - 2*gamt*dT/(dX^2)*D(N-1);
L(N,N) = gamt*dT/(dX)*(C(N)+D_x(N)) + gamt*dT/(dX^2)*D(N);
M = inv(eye(N)-L);

F = H.*Z.*Z_x - Z.*Z.*H_x + (beta).*(H.*Z_x - Z.*H_x);

F_x(ii,1) = (F(ip1,1)-F(im1,1))/(2*dX);
F_x(1) = (-3*F(1) + 4*F(2) - F(3))/(2*dX);
F_x(N) = (3*F(N) - 4*F(N-1) + F(N-2))/(2*dX);
Phi = 2*gamt*dT*F_x;

Z = Z + M*Phi;

if( rem(iStep,Nsteps/10) == 0 )
    if (i == 1)
        subplot(1,1,1);
        i=i+1;
    end
    Change = norm(Z-Zold)/norm(Z) * 100;
    Zold = Z;
    P = Z./H;
    plot(X,P,'*');
    xlabel('X'); ylabel('P');
    title(sprintf('After %g steps, percent change = %g %%',iStep,Change));
    drawnow;
end

iStep = iStep+1;
end

%Calculate P
P = Z./H;

%Display Solution time
time = toc

% Plot profile
sprintf('change = %g %% time=%g', Change,time);
subplot(1,1,1);
plot(X,P,'w-'); % ,Pdatacn3(:,1),Pdatacn3(:,2), 'w*');
xlabel('X'); ylabel('P');
title(sprintf('Viscosity # = %.4e, Knudsen0 # = %g Phase diff.= %g', ...
    viscosity,Kn0,(phase*180/pi)));

% Output Data
FID = fopen(OutputFileName,'wt');
N = length(X);
DATA=[rot90(X);rot90(P)];
fprintf(FID, '%12.6f %12.6f\n',DATA);
fclose(FID);

```