1996

A DSMC validation of an extended slip boundary condition for curved boundaries

Kevin W. Tibbs
San Jose State University

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

Recommended Citation

Tibbs, Kevin W., "A DSMC validation of an extended slip boundary condition for curved boundaries" (1996). *Master's Theses*. 1408.

DOI: <https://doi.org/10.31979/etd.bcuf-ynv5>

https://scholarworks.sjsu.edu/etd_theses/1408

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

**A DSMC VALIDATION OF AN EXTENDED SLIP BOUNDARY CONDITION FOR
CURVED BOUNDARIES**

A Thesis

Presented to

The Faculty of the Department of Physics
San Jose State University

In Partial Fulfillment

of the Requirements for the Degree
Master of Science

by

Kevin W. Tibbs

December 1996

UMI Number: 1382612

UMI Microform 1382612
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

APPROVED FOR THE DEPARTMENT OF PHYSICS

Dr. Alejandro Garcia

Dr. Patrick Hamill

Dr. Curt Flory, Hewlett-Packard Labs

APPROVED FOR THE UNIVERSITY

© 1996

Kevin W. Tibbs

ALL RIGHTS RESERVED

ABSTRACT

A DSMC VALIDATION OF AN EXTENDED SLIP BOUNDARY CONDITION FOR CURVED BOUNDARIES

by Kevin W. Tibbs

The Direct Simulation Monte Carlo technique is used to model the behavior of a molecular fluid entrained in the annular region between rotating cylinders. The simulation validates an extension to the slip boundary condition for the Navier-Stokes equation, proposed by Panzer, Liu, and Einzel. Specifically, this simulation has exhibited the anomolous reversal of slope of the angular component of velocity, as a function of radius, for systems with large slip lengths, as predicted by Panzer, et al.

ACKNOWLEDGEMENTS

I would like to thank my wife, Addy, for her continued encouragement, for which this work would likely not have come to fruition, and my parents for being an endless supply of emotional support. I would also like to thank Dr. Alejandro Garcia for his patience and his ability to know when to exchange the carrot for the whip. I would especially like to thank Dr. Curt Flory for his careful evaluation of this manuscript, his keen insight, and his tolerance of my simulations thrashing his computer.

Table of Contents

1. Introduction.	1
2. Theoretical Background.	1
2.1 Theoretical Background to Navier-Stokes	1
2.2 Theoretical Background to Panzer, Liu, and Einzel (PLE) Extension for slip	4
2.3 Theoretical Background to Boltzmann Transport.	5
2.3.1 The Maxwell-Boltzmann Distribution.	6
2.3.2 Collision Frequency and the Mean Free Path	9
2.4 Theoretical Background to Maxwell's Slip.	11
3. The Simulation	12
3.1 The DSMC Technique for the Simulation of a Large Ensemble of Particles.	12
3.2 The Geometry and Mesh of the System	13
3.3 Calculation of Experimental Parameters	14
3.3.1 Mesh Calculation Algorithm	15
3.4 The Random Number Generator.	17
3.4.1 The Uniform Deviate.	17
3.4.2 Invertible Distributions	18
3.4.3 The Gaussian Distribution	19
3.4.4 The Acceptance-Rejection Scheme for Generating Arbitrary Random Distributions	21
3.5 Program Flow and Design	22
3.6 Movement and Wall Interaction Algorithm	23
3.7 The Particle Sorting Technique	24
3.8 Collision Algorithm	25
3.9 Measurements	28
3.9.1 Standard Deviation.	28
3.9.2 Angular Component of Velocity	28
4. Runs	29
4.1 Validation Runs	29
4.1.1 Completely Specular Walls	30
4.1.1.1 Conservation of Energy, and Linear and Angular Momentum.	30
4.1.1.2 Evolution of a Uniform Velocity Distribution to a Maxwell-Boltzmann Distribution.	33
4.1.2 Maxwell-Boltzmann Distribution from Thermal Interactions with Stationary Walls.	34

Table of Contents

4.1.3 The Simulation's Approach to Navier-Stokes	35
4.2 Experimental Runs	37
5. Discussion	42
5.1 Mechanism Involved in the Slope Reversal of v_θ vs. r	42
5.2 Conclusion	46
Bibliography	47
Appendices.	48
A.1 Cartesian and Cylindrical-Polar Conversions for Position and Velocity.	48
A.2 Wall Interaction Calculations.	50
A.2.1 Determining whether a wall interaction will occur in a time step	50
A.2.2 Determining the post-interaction particle velocity.	51
A.3 Integration of the Integral $\int_0^\infty v^2 e^{-av^2} dv$	54
A.4 Program Listings	56

List of Figures

Fig. 2-1 Geometry for a fluid entrained in the annular region between two rotating, infinitely long cylinders.....	2
Fig. 2-2 The Maxwell-Boltzmann probability density distribution for Argon at 300 K..	8
Fig. 2-3 A test particle traveling through stationary particles.....	10
Fig. 3-1 An example mesh and related geometry of the annular region between rotating cylinders.....	14
Fig. 3-2 Pseudocode fragment used to generate two independent Gaussian variables, x and y , with a mean of μ and variance of σ	20
Fig. 3-3 An illustration of the Acceptance/Rejection method for generation of an arbitrary probability distribution.....	21
Fig. 3-4 Relationship between input and output files to the main simulation routine with an example of the file name incrementation scheme.....	22
Fig. 3-5 An illustration of the particle sorting technique.....	25
Fig. 4-1 An example collision between identicle particles that illustrates how the collision algorithm introduces a non-exact conservation of angular momentum.....	32
Fig. 4-2 Evolution of an initially uniform velocity distribution to a Maxwell-Boltzmann distribution from particle-particle interactions only.....	34
Fig. 4-3 Evolution of an initial uniform velocity distribution to a Maxwell-Boltzmann distribution from interaction with thermalizing walls only.....	35
Fig. 4-4 Comparison of velocity profiles for completely thermalizing walls.....	36
Fig. 4-5 The predicted reversal of slope for the angular component of velocity as a function of radius for the PLE extension to the slip boundary condition for various values of α 's for the "small" system.....	38
Fig. 4-6 Angular component of velocity vs. radius for the case of highly specular walls exhibiting a profile with opposite slope to that of the case where the walls were highly thermalizing.....	39
Fig. 4-7 The v_θ vs. r plots for various α 's illustrating the transition from completely thermal walls to highly specular ones.....	41
Fig. 4-8 A plot of average standard deviations of v_θ from the plots found in Fig. 4-7..	41
Fig. 5-1 The initial state of a simulation in which interactions with the walls are switched from completely thermal to completely specular.....	42
Fig. 5-2 This figure shows the effect of switching the particle-wall interactions from completely thermal to completely specular on the pre-spun system.....	43
Fig. 5-3 This figure shows the effect of switching the particle-wall interactions from completely thermal to completely specular on the pre-spun system without particle-particle interactions.....	43
Fig. 5-4 The initial state of the fine mesh simulation in which interactions with the walls are switched from completely thermal to completely specular.....	45

List of Figures

Fig. 5-5	This figure shows the effect of switching the particle-wall interactions from completely thermal to completely specular on the fine mesh pre-spun system..	45
Fig. A2-2	Geometry of a particle's path and its possible intersection with a container wall..	50
Fig. A2-3	Geometry of a point of contact on the surface of a cylindrical wall..	52
Fig. A4-4	Program map for the simulation code and postprocessing code..	57

Section 1: Introduction

This thesis involves the simulation of the flow of a dilute gas within the annular region between two rotating cylinders. The behavior of the velocity field under conditions in which the slip length is comparable to the characteristic length of the system is examined.

An extension to the usual slip boundary condition used in the Navier-Stokes equation was proposed in a paper by Panzer, et al.¹ This extension involved the explicit inclusion of local boundary curvature. Their boundary condition produced a solution for the Navier-Stokes equation that showed an anomalous slope for the angular component of velocity as a function of radius when the boundary surface has a large slip. Being an extension to the Navier-Stokes equation, the velocity profile due to the Panzer, Liu, and Einzel (PLE) boundary condition would naturally only be valid for the continuum approximation. It was the intent of this research to either validate or invalidate the solution by attempting to elicit this phenomenon from a microscopic description.

This paper outlines the theoretical background to the Navier-Stokes equation (including the PLE extension), Boltzmann transport, and Maxwell's slip. It then outlines the various algorithms used to simulate the dilute gas entrained within the annular region between the rotating cylinders. Finally, the paper shows the results of various validation and experimental runs that reveal the anomalous slope, supporting the theoretical extension to the boundary conditions proposed by Panzer, et al.

Section 2: Theoretical Background

2.1 Theoretical Background to Navier-Stokes

At the continuum level, a fluid is described by its local mass density, momentum density and energy density. The latter two may be replaced by the local fluid velocity, defined as

¹ I. D. Einzel, P. Panzer, M. Liu, Physical Review Letters, **64**, No. 19, 2269-2272 (1990); P. Panzer and M. Liu, D. Einzel, Int. J. of Mod. Phys. B, **6**, No. 20, 3251-3278 (1992).

momentum density over mass density, and local temperature, defined as the internal energy density over heat capacity.

In fluid dynamics, the equation of motion for viscous fluids is the Navier-Stokes equation

$$\rho \frac{D\mathbf{V}}{Dt} = -\nabla p + \mathbf{B} + \mu \nabla^2 \mathbf{V} \quad (2.1)$$

where $\mathbf{V}$ is the velocity field, ρ is the density, p is the isotropic mechanical pressure, $\mathbf{B}$ is the body force, and μ is the viscosity. The substantive derivative, $\frac{D}{Dt}$, is a vector operator and applying it to a vector quantity is not equivalent to operating the components of the operator onto the components of the vector, except in cartesian coordinates. A suitable vector expression for the operator is

$$\frac{D\mathbf{V}}{Dt} = \frac{\partial}{\partial t} \mathbf{V} + \nabla \frac{V^2}{2} - \mathbf{V} \times (\nabla \times \mathbf{V}) \quad (2.2)$$

and can be expanded into the components of any coordinate system.²

Fig. 2-1. Geometry for a fluid entrained in the annular region between two rotating, infinitely long cylinders.

2. L. D. Landau and E. M. Lifshitz, *Fluid Mechanics*, Vol. 6 of *Course of Theoretical Physics*, Pergamon Press, 49 (1959).

The Navier-Stokes equation is valid only for systems in the continuum limit, i.e., that limit in which the differential element is large in comparison to the characteristic length between particle collisions.

The system under examination is one in which a fluid is entrained in the annular region between two infinitely long cylinders. Schematically, the geometry describing this system is shown in Fig. 2-1.

Rewriting the Navier-Stokes equation for the situation of rotating cylinders of infinite z extent, one gets for the angular component of velocity³

$$\frac{d^2 v_\theta}{dr^2} + \frac{1}{r} \frac{dv_\theta}{dr} - \frac{v_\theta}{r^2} = 0 \quad (2.3)$$

which has a solution of the form

$$v_\theta = ar + \frac{b}{r}. \quad (2.4)$$

The coefficients a and b are found from the boundary conditions which specify that the angular component of the velocity of the fluid is equal to the wall velocity at the interfaces. Applying this no-slip approximation, the solution becomes

$$v_\theta = \frac{\omega_2 R_2^2 - \omega_1 R_1^2}{R_2^2 - R_1^2} r + \frac{(\omega_1 - \omega_2) R_1^2 R_2^2}{R_2^2 - R_1^2} \frac{1}{r}. \quad (2.5)$$

In a real fluid, the velocity of the fluid does not exactly match the wall's velocity, even for a completely thermalizing wall, so the no-slip boundary condition becomes invalid. A first order fix to this inconsistency involves the application of a modified boundary condition

3. L. D. Landau and E. M. Lifshitz, *Fluid Mechanics*, Vol. 6 of *Course of Theoretical Physics*, Pergamon Press, 60 (1959).

$$v_{\theta}(R) = \zeta_0 \left. \frac{dv_{\theta}}{dr} \right|_{\text{wall}} \quad (2.6)$$

where

$$\zeta_0 = a_D \lambda. \quad (2.7)$$

The intrinsic slip length, ζ_0 , is proportional to the mean free path, λ , and is that distance into the wall the tangential velocity field of the fluid is extrapolated to zero. The value of the proportionality constant, a_D , may be measured experimentally or obtained from theoretical calculations for model problems. For example, for the Kramer's problem (semi-infinite shear flow past a plane wall) a solution of the Boltzmann equation gives $a_D = 1.126$.⁴

2.2 Theoretical Background to Panzer, Liu, and Einzel (PLE) Extension for slip

Panzer, et al. include additional corrections to the traditional slip boundary condition by generalizing the slip condition as

$$v_{\theta}(R) = \zeta \left. \frac{dv_{\theta}}{dr} \right|_{\text{wall}} \quad (2.8)$$

and

$$\frac{1}{\zeta} = \frac{1}{\zeta_0} - \frac{1}{R}. \quad (2.9)$$

Here, the slip length, ζ , is a combination of the intrinsic slip length, ζ_0 , and the radius of curvature of the boundary, R . Applying the boundary condition in equation (2.8) to equation (2.4), the solution for the angular component of velocity with the PLE extension is

4. C. Cercignani, *Theory and Application of the Boltzmann Equation*, Scottish Academic Press, Edinburgh (1975).

$$v_{\theta} = \left(\frac{A\omega_1 + B\omega_2}{A-B} \right) r + \left(\frac{\omega_2 - \omega_1}{A-B} \right) \frac{1}{r}, \quad (2.10)$$

where

$$A = \frac{1}{R_2^2} \left(1 - 2 \frac{\zeta_0}{R_2} \right); \quad B = \frac{1}{R_1^2} \left(1 + 2 \frac{\zeta_0}{R_1} \right). \quad (2.11)$$

Panzer, et. al, also include a modification to the intrinsic slip length by using an enhancement factor that accounts for different ratios of specular versus thermal wall interactions. The intrinsic slip length then becomes

$$\zeta_0 = a\lambda \quad (2.12)$$

where,

$$\begin{aligned} a &= a_D \frac{(1+s-r)}{(1-s+r)} \\ &= a_D \frac{(2-\alpha)}{\alpha} \end{aligned} \quad (2.13)$$

In equation (2.13), the enhancement factor α is defined as the ratio of thermal wall collisions to total wall collisions; whereas, the more general enhancement factor includes both backscattering particles, represented by r , and forward specular collisions, represented by s . For the simulations in this paper, all non-thermal wall collisions are of the forward type (i.e., $r = 0$ and $s = 1 - \alpha$).

2.3 Theoretical Background to Boltzmann Transport

Since the numerical model in this paper simulates only binary collisions (those that occur between only two particles at a time) and uses the hard sphere approximation for the particle-particle interaction, it can only be used to model dilute non-reactive gases. A gas is considered dilute when the mean interparticle spacing, δ , is about an order of magnitude larger than the effective diameter of the particle, d .⁵

The model in this paper uses Argon atoms, and at standard temperature and pressure (STP), the density of Argon is 2.69×10^{25} particles per cubic meter which equates to one atom, on average, in a cube with side dimensions of $33.4 \text{ \AA}$. By assuming this dimension to approximate the nearest neighbor distance between atoms, the ratio to the effective diameter, $3.66 \text{ \AA}$, becomes

$$\frac{\delta}{\lambda} \equiv 9; \quad (2.14)$$

therefore, our criterion for a dilute gas is met.

2.3.1 The Maxwell-Boltzmann Distribution

Since our collision algorithm depends on average velocities, most probable velocity, and mean free path, we need a method of calculating these statistical quantities for dilute gases.

For a system of particles, in thermodynamic equilibrium with a heat reservoir of temperature T , the probability of finding a particle within the range of position $\mathbf{r}$ and $\mathbf{r} + d\mathbf{r}$ and within the range of velocity $\mathbf{v}$ and $\mathbf{v} + d\mathbf{v}$ is

$$P(\mathbf{r}, \mathbf{v}) d\mathbf{r} d\mathbf{v} = C e^{\left(-\frac{E(\mathbf{r}, \mathbf{v})}{kT}\right)} d\mathbf{r} d\mathbf{v} \quad (2.15)$$

where $k = 1.38 \times 10^{-23} \frac{\text{J}}{\text{K}}$ is Boltzmann's constant, E is the particle's energy, and C is a normalization constant. Since the only particle-particle interaction is over an extremely short range (the hard sphere approximation), the energies of the particles are entirely kinetic; therefore, the energy term can be written as

$$E(\mathbf{r}, \mathbf{v}) = E(\mathbf{v}) = \frac{1}{2} m |\mathbf{v}|^2. \quad (2.16)$$

5. G. A. Bird, *Molecular Gas Dynamics*, Claredon Press, Oxford, 17-19 (1972).

6. S. Chapman and T. G. Cowling, *The Mathematical Theory of Non-uniform Gases*, Cambridge Univ. Press, Cambridge, 228 (1990).

Additionally, the positional independence of the energy makes the probability function separable; therefore,

$$P(\mathbf{r}, \mathbf{v}) d\mathbf{r} d\mathbf{v} = P_r(\mathbf{r}) P_v(\mathbf{v}) d\mathbf{r} d\mathbf{v}. \quad (2.17)$$

Equating equations (2.17) and (2.15), substituting the energy term from equation (2.16), and separating the normalization constant, the probability equation becomes

$$P_r(\mathbf{r}) P_v(\mathbf{v}) d\mathbf{r} d\mathbf{v} = C_r C_v e^{\left(-\frac{m|\mathbf{v}|^2}{2kT}\right)} d\mathbf{r} d\mathbf{v}. \quad (2.18)$$

Rearranging, equation (2.18) becomes

$$\frac{P_r(\mathbf{r}) d\mathbf{r}}{C_r d\mathbf{r}} = \frac{C_v e^{\left(-\frac{m|\mathbf{v}|^2}{2kT}\right)} d\mathbf{v}}{P_v(\mathbf{v}) d\mathbf{v}}. \quad (2.19)$$

Since the left hand side of equation (2.19) is independent of $\mathbf{v}$, and the right hand side is independent of $\mathbf{r}$, both sides are equivalent to a separation constant. By equating the right hand side to this constant, rearranging the terms, absorbing the separation constant into C_v , and dropping the subscripts denoting velocity dependence, the velocity dependent probability is

$$P(\mathbf{v}) d\mathbf{v} = C_v e^{\left(-\frac{mv^2}{2kT}\right)} d\mathbf{v}. \quad (2.20)$$

In order to express the probability as a function of speed, instead of depending on the vector quantity, the volume element is transformed into polar coordinates. The probability distribution then becomes

$$P(\mathbf{v}) d\mathbf{v} = C_v v^2 e^{\left(-\frac{mv^2}{2kT}\right)} \sin\theta dv d\theta d\phi. \quad (2.21)$$

Since the integrand in equation (2.21) is independent of direction, the integration over θ and ϕ is straightforward and yields

$$P(v)dv = C4\pi v^2 e^{\left(\frac{mv^2}{2kT}\right)} dv. \quad (2.22)$$

To calculate the normalization constant, equation (2.22) is integrated over all possible velocity magnitudes and set to one (see Appendix A.3). Substituting this normalization constant back into equation (2.22) yields

$$P(v)dv = 4\pi \left(\frac{m}{2\pi kT}\right)^{3/2} v^2 e^{\left(\frac{mv^2}{2kT}\right)} dv, \quad (2.23)$$

which is known as the *Maxwell-Boltzmann distribution* (MB). Fig. 2-2 is the Maxwell-Boltzmann distribution for the system of particles used in the simulation.

Fig. 2-2. The Maxwell-Boltzmann probability density distribution for Argon at 300 K. The dotted line represents the average speed.

The average speed, for a collection of particles obeying the MB distribution, is

$$\begin{aligned} \langle v \rangle &= \int_0^{\infty} v P(v) dv \\ &= \sqrt{\frac{8kT}{\pi m}} \end{aligned} \quad (2.24)$$

and the root mean squared (rms) speed is

$$\begin{aligned}
 v_{\text{rms}} &= \sqrt{\langle v^2 \rangle} = \sqrt{\int_0^\infty v^2 P(v) dv} \\
 &= \sqrt{\frac{3kT}{m}}
 \end{aligned}
 \tag{2.25}$$

The speed at which $P(v)$ is a maximum is the most probable speed, and is

$$\frac{d}{dv}P = 0 \Rightarrow v_{\text{mp}} = \sqrt{\frac{2kT}{m}}.
 \tag{2.26}$$

For a system of Argon atoms at 300 K, the above quantities work out to

$$\begin{aligned}
 \langle v \rangle &= 398.7 \frac{m}{s} \\
 v_{\text{rms}} &= 432.8 \frac{m}{s} \\
 v_{\text{mp}} &= 353.4 \frac{m}{s}
 \end{aligned}
 \tag{2.27}$$

2.3.2 Collision Frequency and the Mean Free Path

In order to calculate the collision frequency, consider the path taken by a moving test particle through a collection of fixed particles as shown in Fig. 2-3. The test particle will travel linearly until it collides elastically with one of the fixed particles, resulting in a deflection from its linear path.

The volume in which this test particle travels is a tube of radius d , where d is the diameter of the particles. This tube will have an approximate volume of

$$V_{\text{tube}} = \pi d^2 l
 \tag{2.28}$$

where l is the length of the particle's total path. A correction for the volume of the tube at the deflection points is not used since the diameter of the simulated particles is much smaller than the mean free path.

Fig. 2-3. A test particle traveling through stationary particles.

Since the gas has uniform density, the number of collisions, M_{tube} , experienced by the test particle, traveling at velocity v , in time t_0 is

$$\begin{aligned} M_{\text{tube}} &= (\text{Gas Density}) \times (\text{Volume of Tube}) \\ &= \left(\frac{N}{V}\right) \times (\pi d^2 v t_0) \end{aligned} \quad (2.29)$$

The collision frequency, the average number of collisions occurring in the sample time, t_0 , is

$$\begin{aligned} f &= \frac{\langle M_{\text{tube}} \rangle}{t_0} \\ &= \frac{N}{V} \pi d^2 \langle v \rangle \end{aligned} \quad (2.30)$$

In order for the above equation to be valid for the situation in which all the other particles are allowed to move, we substitute the average *relative* velocity, $\langle v_r \rangle$, for the average velocity term. The average relative velocity can be derived from the Maxwell-Boltzmann probability distribution as

$$\begin{aligned}
\langle v_r \rangle &= \langle |\mathbf{v}_1 - \mathbf{v}_2| \rangle \\
&= \iint |\mathbf{v}_1 - \mathbf{v}_2| P(\mathbf{v}_1) P(\mathbf{v}_2) d\mathbf{v}_1 d\mathbf{v}_2 \\
&= \sqrt{\frac{16kT}{\pi m}}
\end{aligned} \tag{2.31}$$

The mean free path is

$$\begin{aligned}
\lambda &= (\text{Average Particle Speed}) \times (\text{Average Time Between Collisions}) \\
&= \frac{\langle v \rangle}{f} \\
&= \frac{V}{N\pi d^2 \langle v_r \rangle}
\end{aligned} \tag{2.32}$$

Combining equations (2.24) and (2.31), we get

$$\lambda = \frac{V}{\sqrt{2} N \pi d^2} \tag{2.33}$$

2.4 Theoretical Background to Maxwell's Slip

Maxwell developed the following simple argument to describe the phenomenon of slip. Consider a flat wall in the xy -plane located at $z=0$; the wall moves in the x -direction with speed u_w . There is a dilute gas above the wall and it has fluid velocity $u(z)$ due to the motion of the wall. Call u_b and u_a the mean x -velocity of molecules just before and just after striking the wall. The mean x -velocity of the gas at the wall is

$$u(z=0) = \frac{(u_b + u_a)}{2} \tag{2.34}$$

since the average number of molecules approaching the wall must equal the number leaving the wall (the surface does not create or destroy particles).

It is assumed that a fraction, α , of particles are thermalized on striking the wall while $(1 - \alpha)$ scatter elastically off the surface. The term α will be referred to as the *accommodation factor*. Then

$$u_a = (1 - \alpha)u_b + \alpha u_w \quad (2.35)$$

or

$$u_w = \frac{(u_a + u_b)}{\alpha} - \frac{(2 - \alpha)}{\alpha} u_b. \quad (2.36)$$

Particles arriving at the wall had their last collision with other particles in the fluid at a distance of approximately one mean free path from the wall so u_b can be estimated as $u(z=\lambda)$. Near the wall, $u(z)$ can be expanded in a Taylor series as

$$u_b \equiv u(z=0) + \frac{du}{dz}\lambda = \frac{(u_b + u_a)}{2} + \frac{du}{dz}\lambda. \quad (2.37)$$

Replacing the second u_b term in equation (2.36) with equation (2.37) and rearranging the terms yields

$$\frac{(u_b + u_a)}{2} = u_w + \frac{(2 - \alpha)}{\alpha} \frac{du}{dz}\lambda. \quad (2.38)$$

By directly comparing equations (2.38) and (2.34), the fluid velocity at the wall becomes

$$u(z=0) = u_w + \frac{(2 - \alpha)}{\alpha} \frac{du}{dz}\lambda. \quad (2.39)$$

Equation (2.39) shows that the fluid velocity at the wall differs from the wall's velocity, u_w , by a factor that depends on the accommodation factor, the velocity gradient, and the mean free path.

Section 3: The Simulation

3.1 The DSMC Technique for the Simulation of a Large Ensemble of Particles

A method for the simulation of a large ensemble of particles governed by the Boltzmann transport equation is the Direct Simulation Monte-Carlo (DSMC) technique. This technique involves the imposition of a grid onto the system being simulated such that the

grid size is on the order of or smaller than the mean free path of the particles. The particles within each grid cell are sequentially advanced along a linear path, checked for wall collision, and then collided with each other.

One aspect of the DSMC method that makes it particularly attractive is its ability to simulate systems with a large number of particles. The means by which this is accomplished is by not tracking the trajectories of every “physical” particle but through the use of “parliamentary” particles. In the simulation, parliamentary particles represent a multitude of physical particles and, consequently, carry the statistical information of velocity, mass and position for the ensemble of physical particles they represent. Each parliamentary particle is, therefore, an aggregate of physical particles.

Another aspect of the DSMC technique that makes it computationally tractable is its method of processing collisions. The simultaneous tracking of particle paths and proximity detection for the processing of particle collisions would be computationally intensive, so DSMC selects particle pairs randomly from within a cell and simulates a collision by resetting the particle’s velocities according to the kinematics of an elastic collision. The proper number of collisions within a cell is calculated from the collision rate, particle density, and time between steps. This approximation is a good one since cell sizes are chosen to have a characteristic size smaller than the mean free path, and collisions will most likely occur between particles within a mean free path of each other. The collision algorithm is described in more detail in Section 3.8.

3.2 The Geometry and Mesh of the System

For the simulation, the cells of the annular region were generated by first choosing an integer number of sectors that had a maximum edge length on the order of a mean free path. These sectors were then further broken up by choosing an integer number of tracks that generated cells, near the inner wall, with unity aspect ratio (see Fig. 3-1 for a sample layout of the geometry).

For this problem, it has been assumed that the system extends infinitely in both positive and negative z -directions.

Fig. 3-1. An example mesh and related geometry of the annular region between rotating cylinders. Selected cells have been enumerated to illustrate the cell numbering scheme used in the simulations.

3.3 Calculation of Experimental Parameters

As alluded to earlier, the mesh generation cannot be arbitrary and will relate to intrinsic fluid parameters and the extrinsic system parameters.

The mesh is generated according to the following considerations:

- The slip effects predicted by Panzer, et al. should occur in both compressible and incompressible flow. It is well established that gas flow at speeds below 0.3 Mach is well approximated as incompressible due to the negligible contribution of the $\nabla \frac{V^2}{2} - \mathbf{V} \times (\nabla \times \mathbf{V})$ terms in equation (2.2). Though the simulations are actually easier to perform at higher speeds since the fractional error of the mean value of the local fluid velocity goes as the inverse of Mach number⁷, the simulations were run with wall

7. A. L. Garcia, private communication.

velocities less than 0.3 Mach to allow comparison with analytic solutions which only exist at the lower velocities.

- The largest cell must have a characteristic side length less than or equal to the mean-free-path (mfp). The collision algorithm, described below, selects two particles for collision from within a cell, regardless of location within the cell. Requiring cells to have a characteristic length on the order of the mfp, or less, ensures that collisions happen between likely collision pairs.
- The aspect ratio of the cells must be such that they are not elongated along the radial direction since the angular component of velocity as a function of radius is desired; therefore, the smallest cell is forced to have an aspect ratio near unity in order to minimize any geometric effects occurring in radially long and thin cells.
- The particles, on average, should remain within the smallest cell for at least 4 time steps before leaving the cell. This will provide adequate “mixing” opportunity of the statistical information that the particles carry. Since particles will only exchange this information via inter-particle collisions, and only particles within a given cell are allowed to collide, particles should remain in a given cell for an adequate length of time.
- Each cell is required to have at least 25 representative particles. The DSMC method can adequately reproduce the behavior of a fluid with no less than roughly 25 representative particles.⁸ To prevent any fluctuations below this value, the number of particles in the system is calculated by assuming that the smallest cell contains 100 representative particles.

3.3.1 Mesh Calculation Algorithm

Using the above requirements as a basis, the meshes for the simulations were formulated using the following algorithm:

1. Choose inner radius, r_{inner} , and outer radius, r_{outer} of the system.

8. M.A. Fallavollita, J.D. McDonald and D. Baganoff, Comp. Sci. Eng. 3, 283 (1992).

2. Choose the tangential wall velocities and calculate the corresponding angular velocities for the walls. In all simulations, the outer wall tangential velocity was always set to zero and the inner wall tangential velocity was always positive (counter-clockwise looking in the negative z direction).

$$\omega_{\text{inner}} = \frac{r_{\text{inner}}}{v_{\text{wall_in}}} \quad \omega_{\text{outer}} = 0 \quad (3.1)$$

3. Calculate the angular extent of the cells.

$$\begin{aligned} r_{\text{outer}} \Delta\theta &= \lambda \\ \therefore \Delta\theta &= \frac{\lambda}{r_{\text{outer}}} \end{aligned} \quad (3.2)$$

4. From the consideration that the smallest cell has an aspect ratio of one, calculate the cell length in the radial direction.

$$\Delta r = r_{\text{inner}} \Delta\theta \quad (3.3)$$

5. From the consideration that a particle must, on average, be in a given cell for four time steps, calculate the time step. For this step, the velocity of a particle may be approximated as the sum of the most probable velocity and the velocity of the inner wall. The addition of the wall velocity makes the calculation of Δt more of a worst case value.

$$\Delta t = \frac{1}{4\Delta\theta(v_{\text{mp}} + v_{\text{wall_in}})} \quad (3.4)$$

6. Assuming that the number of particles in the smallest cell is 100, calculate the total number of particles used in the simulation.

$$\begin{aligned} n &= \left(\frac{100}{\text{area of smallest cell}} \right) \times (\text{total area of annular region}) \\ \therefore n &= \left(\frac{100}{r_{\text{inner}} \Delta r \Delta\theta + \frac{1}{2} (\Delta r)^2 \Delta\theta} \right) (\pi(r_{\text{outer}}^2 - r_{\text{inner}}^2)) \end{aligned} \quad (3.5)$$

Table 4-1 illustrates the parameter selections used in the simulations performed.

3.4 The Random Number Generator

In order to model stochastic processes, the generation of random numbers is required. Since computers are deterministic, any program is incapable of generating truly random numbers; therefore, a mathematical formula is used that generates a sequence of numbers such that all values, in the interval of interest, are equally probable. Any value generated by this pseudo random number generator is referred to as a *uniform deviate*.

3.4.1 The Uniform Deviate

One good uniform deviate generator is the linear congruential method.⁹ This method uses an initial integer value, I_1 , and generates a sequence of integers from the iterative formula

$$I_{n+1} = (aI_n + c) \bmod M \quad (3.6)$$

where the constants a , c , and M are specified. The ability of the above formula to generate a good uniform deviate is dependent on the proper selection of the constants. Park and Miller show that $a = 7^5$, $c = 0$, and $M = 2^{31}-1$ generates good results.¹⁰

The uniform deviate that forms the basis for many random distributions is that deviate that spans the interval $[0, 1)$. That deviate is generated by scaling the integer sequence, from equation (3.6), by the constant M .

The above pseudorandom number generator will, after a time, generate the seed that was used to start it. When this happens, the generator will repeat the original sequence. Poorly selected constants or seeds will generate sequences of short periods. An example is the standard ANSI C uniform deviate which will repeat after about 33,000 calls.¹¹ The uniform deviate generator used in this work is the algorithm used in the program ran2

9. D. E. Knuth, *Seminumerical Algorithms*, 2nd ed., vol. 2 of *The Art of Computer Programming*, Addison-Wesley, Reading Mass., section 3.2 (1981).

10. S.K. Park and K.W. Miller, *Comm. A.C.M.* 32, 1192 (1988).

11. S. Harbison and G. Steele, Jr., *C, A Reference Manual*, Prentice-Hall, Englewood Cliffs, 354 (1984).

from *Numerical Recipes in C, 2nd ed.*, which employs a bit shuffling between two generators of differing periods, resulting in a generator with a period of about 2.3×10^{18} .¹² Since `ran2` implements a simpler multiplicative congruential algorithm by setting $c = 0$, seed values of zero must not be allowed; otherwise, a perpetual stream of zeros will be generated. As a consequence, the random deviate, generated from `ran2`, spans the interval $(0, 1)$. Any further reference to intervals derived from random deviates in this paper reflect this; additionally, random deviates will be referenced in following sections by the variable $\mathfrak{R}$.

3.4.2 Invertible Distributions

In the Monte Carlo calculation, it is necessary to generate a random number with a distribution other than that of the uniform deviate. This transformation can be accomplished by mapping the uniform deviate to the new interval through an invertible distribution. A distribution $P(x)$ is called “invertible” if $y = F(x)$ implies $x = F^{-1}(y)$, where $F = \int_a^x P(x)dx$. A simple example is the linear mapping to the interval (a, b)

$$x = a + (b - a)\mathfrak{R}, \quad (3.7)$$

where $\mathfrak{R}$ is a uniform deviate.

Another useful transformation is one that generates a sequence with an exponential distribution

$$P(u) = \frac{1}{\lambda} e^{-u/\lambda} \quad (3.8)$$

where $u \in (0, \infty)$. In order to generate this distribution, consider a new random variable, R , defined as

$$u = -\lambda \ln(1 - R). \quad (3.9)$$

12. W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C, The Art of Scientific Computing*, 2nd ed., Cambridge Univ. Press, 281 (1992).

Solving the equation (3.9) for R and taking the derivative yields

$$dR = \frac{1}{\lambda} e^{-u/\lambda} du. \quad (3.10)$$

Substituting equation (3.8) into (3.10) gives

$$P(u)du = dR = P(R)dR \quad (3.11)$$

which shows that the distribution for $P(R)$ is simply the uniform deviate, $\mathfrak{R}$, and the exponential distribution, $P(u)$, can be calculated by

$$u = -\lambda \ln(1 - \mathfrak{R}). \quad (3.12)$$

3.4.3 The Gaussian Distribution

Another important distribution is the Gaussian distribution

$$P(x)dx = \frac{1}{\sqrt{2\pi}\sigma} e^{-(x-\mu)^2/(2\sigma^2)} dx \quad (3.13)$$

which has a mean value of $\langle x \rangle = \mu$ and variance $\langle (x - \langle x \rangle)^2 \rangle = \sigma^2$.

In order to make the Gaussian distribution invertible, the same polar transformation technique can be employed that is used to integrate it.¹³

Consider the product of two similar Gaussian distributions

$$\begin{aligned} P(x)P(y)dxdy &= \frac{1}{2\pi\sigma^2} e^{-(x-\sigma)^2/(2\sigma^2)} e^{-(y-\sigma)^2/(2\sigma^2)} dxdy \\ &= \frac{1}{2\pi\sigma^2} e^{-((x-\sigma)^2 + (y-\sigma)^2)/(2\sigma^2)} dxdy \end{aligned} \quad (3.14)$$

Applying the polar transformation,

13. F. Reif, *Fundamentals of Statistical and Thermal Physics*, McGraw-Hill, 606 (1965).

$$x = \rho \cos \theta + \mu \quad y = \rho \sin \theta + \mu \quad (3.15)$$

to equation (3.14) gives

$$P(\rho)P(\theta)\rho d\rho d\theta = \frac{1}{2\pi\sigma^2}e^{-\rho^2/(2\sigma^2)}\rho d\rho d\theta. \quad (3.16)$$

Further transforming with

$$u = \frac{\rho^2}{2\sigma^2} \quad a = \frac{1}{2\pi}\theta, \quad (3.17)$$

equation (3.16) becomes,

$$P(u)P(a)du da = (e^{-u}du)(1da). \quad (3.18)$$

The above equation shows that u has an exponential distribution and that a is simply a uniform deviate. This reveals that x and y , two independent variables that follow a Gaussian distribution, can now be generated from two uniform deviates, $\mathcal{R}_1$ and $\mathcal{R}_2$. The following pseudocode fragment illustrates this process.

```

R1 = ran()
R2 = ran()
u = -ln(1-R1)
a = R2
rho = sigma * sqrt(2 * u)
theta = 2 * Pi * a
x = rho * cos(theta) + mu
y = rho * sin(theta) + mu

```

Fig. 3-2. Pseudocode fragment used to generate two independent Gaussian variables, x and y , with a mean of μ and variance of σ^2 .

The above algorithm was used for initializing the particle population at the start of the simulations, and also for generating particle velocities after thermal particle-wall interactions.

3.4.4 The Acceptance-Rejection Scheme for Generating Arbitrary Random Distributions

An arbitrary random distribution can be generated by a technique called *acceptance-rejection* (AR). Consider Fig. 3-3, wherein the curve $P(x)$ represents an arbitrary probability distribution.

Fig. 3-3. An illustration of the Acceptance/Rejection method for generation of an arbitrary probability distribution. Membership of the distribution is denoted by the shaded region. A box, bounded by the region $x \in (a, b)$ and $P \in (0, P_{\max})$, is an area that can be filled with randomly placed “darts” by the use of two independent uniform deviates. Darts that fall in the shaded area are *accepted*; whereas, those points that don’t are *rejected*.

The curve is framed by a box that bounds the range $x \in (a, b)$ and $P \in (0, P_{\max})$. Creating a random variable with the curve’s distribution is implemented by generating a trial point, $(x_{\text{trial}}, y_{\text{trial}})$, that is randomly placed within the box and accepted if $y_{\text{trial}} \leq P(x_{\text{trial}})$.

This technique randomly “throws darts” into the bounding box and keeps only those that land in the shaded area. If the bounding box has a $P_{\max}$ that is less than the maximum of the curve $P(x)$ anywhere along the interval (a, b) , then the clipped portion of the curve can not be filled with darts; therefore, AR can exactly generate the arbitrary distribution only if $P_{\max} \geq P(x) \forall x \in (a, b)$. If $P_{\max}$ is larger than the largest value of the $P(x)$

along (a, b) , then AR is less efficient since more trial points are rejected; however, those points that are accepted still follow the distribution being modeled.

In these simulations, the acceptance-rejection method is used for selecting candidate collision pairs for the collision algorithm, as described in Section 3.8.

3.5 Program Flow and Design

The program was designed to be used in an iterative fashion. A configuration file, which contains all of the configurable parameters, and an optional state file which contains the positions and velocities of all of the particles from the previous run, are read by the main routine. This routine generates a new configuration and state file; additionally, a cell statistics file is generated for that run, which contains the time averaged particle count, velocity, and squared velocity for each cell. As described in Section 3.9, the cell statistics file is used for generating the measures used in characterizing the flow field.

Fig. 3-4. Relationship between input and output files to the main simulation routine with an example of the file name incrementation scheme.

As Fig. 3-4. depicts, the output of one run can be fed directly into the input of the next run. A naming convention was established to facilitate the automatic incrementation of the

filenames allowing for the invocation of a simulation, comprised of many runs, via a script.

The main simulation routine performs the operations of `move`, `sort`, and `collide` in succession. In other words, all particles are moved, including the processing of wall interactions, then sorted, and finally collided. A closer look at the program roadmap and the code itself in Appendix A.4 makes this more clear.

3.6 Movement and Wall Interaction Algorithm

For each time step within a given run, each particle in turn is moved along a linear path, and its trajectory is tested for a possible intersection with either wall. If it is determined that the particle will not collide with either wall over the time step, then the particle is placed at its next location and the next particle's trajectory is examined. If, however, the particle's projected trajectory intersects a wall within the time step, the particle's position is set to its intersect point on the wall, its velocities are reset according to the nature of the wall interaction, and then it is propagated away from the wall over the remainder of the time step.

To determine the rebound velocities at wall interactions, the incoming particle's velocity is resolved into the components normal and tangential to the wall. For specular wall interactions, the tangential component remains constant and the normal component changes sign. For thermal wall interactions, the velocity of the particle is reset according to

$$\mathbf{v} = v_n \hat{\mathbf{n}} + v_t \hat{\mathbf{t}} + v_z \hat{\mathbf{z}} \quad (3.19)$$

where the components are governed by the Gaussian probability distributions

$$\begin{aligned}
P(v_n)dv_n &= \pm \left(\frac{m}{kT_{\text{wall}}} \right) v_n e^{\left(-\frac{m}{2kT_{\text{wall}}} v_n^2 \right)} dv_n \\
P(v_t)dv_t &= \sqrt{\frac{m}{2\pi kT_{\text{wall}}}} e^{\left(-\frac{m}{2kT_{\text{wall}}} (v_t - r_{\text{wall}} \omega_{\text{wall}})^2 \right)} dv_t \\
P(v_z)dv_z &= \sqrt{\frac{m}{2\pi kT_{\text{wall}}}} e^{\left(-\frac{m}{2kT_{\text{wall}}} v_z^2 \right)} dv_z
\end{aligned} \tag{3.20}$$

where the sign of the normal component, v_n , is positive for the inner cylinder wall and negative for the outer. Notice that even though the simulation is for a two dimensional system, the z-component is used to maintain proper velocity statistics.

For the thermal wall interactions, the wall temperatures are set to a default of 300 K. In this project, effects of wall temperature differences (i.e. temperature jump phenomena, viscous heating, etc.) are not examined.

3.7 The Particle Sorting Technique

The program must be able to sequentially step through the cells and randomly select any pair of particles from within that cell in order to process a collision; therefore, all the particles' cell memberships must be known, and additionally, there must exist a random selection mechanism that picks out candidate particles from the cell of interest. In this program, three arrays are generated that facilitate this selection process. Consider that the cells from the annular mesh have been laid out (see Fig. 3-1 for an example of the cell numbering scheme) in a single line in ascending order.

By stepping through all of the cells, three arrays are built that contain all the information necessary to facilitate the selection mechanism.

The array `order` contains a list of particle numbers that are sorted according to cell membership: those particles that live in cell 1 are listed first, those in cell 2 are listed next and so on (the order in the particular cell is not important). The array `cell_n` contains the particle count per cell and the array `index` contains the base index for each cell. The

Fig. 3-5. An illustration of the particle sorting technique.

base index for each cell is simply the cumulative sum of `cell_n` for those cells preceding it plus 1.

Arrays `index` and `cell_n` are used to specify the base and range indices of `order` that span the entries in `order` for particles that lie within the cell of interest. For example, if we wanted to select a particle from cell 4, we should choose one from `order` between 6 (`=index(4)`) and 8 (`=index(4) + cell_n(4) - 1`). In this example, there are three candidates and they are `order(6)=99`, `order(7)= 2`, and `order(8)=86`.

3.8 Collision Algorithm

The DSMC method involves a collision algorithm that randomly selects two particles from within a cell, regardless of position within that cell, and collides them according to a probability proportional to their relative velocities. The collision probability between two particles, i and j , is

$$P(i, j) = \frac{|\mathbf{v}_i - \mathbf{v}_j|}{\sum_{m=1}^{N_{\text{cell}}} \sum_{n=1}^{m-1} |\mathbf{v}_m - \mathbf{v}_n|} \quad (3.21)$$

where the double sum in the denominator, over all possible relative velocities within a cell, normalizes $P(i, j)$.

Repeated calculation of the double sum in the denominator in equation (3.21) for each cell at each time step is computationally expensive; therefore, the acceptance-rejection method is used to generate the collision probability.

In the simulation, a pair of particles, i and j , are collided if the following AR condition is satisfied

$$\frac{|\mathbf{v}_i - \mathbf{v}_j|}{|\mathbf{v}_{\text{rel_max}}|} > \mathfrak{R} \quad (3.22)$$

where $\mathfrak{R}$ is a uniform deviate. Initially, $|\mathbf{v}_{\text{rel_max}}|$ is set to a value that represents a safe overestimate of the possible relative velocities. Comparing equations (2.27) and Fig. 2-2, it can be seen that $3v_{\text{mp}}$ spans most of the range of possible velocities in the gas. Additionally, the collision algorithm must measure a candidate collision pair's relative velocity in order to process the collision; therefore, that relative velocity replaces the current $v_{\text{rel_max}}$ for the cell the pair resides in, should it be greater than its current setting.

To see why the above AR scheme is equivalent to the probability distribution in equation (3.21), consider the following

$$\begin{aligned} P(i, j) &= \frac{|\mathbf{v}_i - \mathbf{v}_j|}{A} \\ P_{\text{max}} &= \frac{|\mathbf{v}_{\text{rel_max}}|}{A} \end{aligned} \quad (3.23)$$

where A is the constant of normalization. Taking the ratio of the two probabilities yield

$$\frac{P(i, j)}{P_{\text{max}}} = \frac{|\mathbf{v}_i - \mathbf{v}_j|}{|\mathbf{v}_{\text{rel_max}}|}. \quad (3.24)$$

The maximum probability for particle collisions within a cell scales the probability for a given pair to $[0, 1]$. Therefore, any uniform deviate will either fall above or below the accept-reject threshold. This process maintains the same distribution as the probability equation as shown above.

Once a pair of particles is selected for collision, their velocities are reset such that the center of mass velocity and the magnitude of the relative velocity remain constant, thus conserving linear momentum and energy, respectively. It should be noted that this conservation of linear momentum is between colliding particle pairs and should not be considered a contradiction to the non-exact conservation of instantaneous linear momentum of the system average due to the rigid wall approximation discussed in Section 4.1.

The relative velocity direction, however, is set to an arbitrary direction in three dimensions. For hard sphere collisions, the direction of the relative velocity after the collision is uniformly distributed in the unit sphere. The new velocities are then

$$\begin{aligned} \mathbf{v}_i^{\text{new}} &= \mathbf{v}_{\text{cm}} + \frac{1}{2} \mathbf{v}_{\text{rel}}^{\text{new}} \\ \mathbf{v}_j^{\text{new}} &= \mathbf{v}_{\text{cm}} - \frac{1}{2} \mathbf{v}_{\text{rel}}^{\text{new}} \end{aligned} \quad (3.25)$$

where,

$$\mathbf{v}_{\text{rel}}^{\text{new}} = v(v_{\text{rel}}, \theta_{\text{new}}, \varphi_{\text{new}}) \quad (3.26)$$

and

$$\begin{aligned} \cos \theta_{\text{new}} &= 1 - 2\mathcal{R}_1 \\ \varphi_{\text{new}} &= 2\pi\mathcal{R}_2 \end{aligned} \quad (3.27)$$

Using the collision frequency, equation (2.30), the total number of collisions within a cell for a given time step is determined to be

$$\begin{aligned} M_{\text{col}} &= \frac{1}{2} N_{\text{cell}} f \Delta t \\ &= \frac{\pi N_{\text{cell}} d^2 \Delta t \langle v_{\text{rel}} \rangle}{2 V_{\text{cell}}} \end{aligned} \quad (3.28)$$

3.9 Measurements

When a simulation is performed, it is comprised of many runs generated iteratively as discussed in Section 3.5, and each run is comprised of many time steps as specified in the configuration files. After each run, a cell statistics file is generated that contains the particle count, components of velocity, and squared velocity for each cell, averaged over the time steps for that run.

When a velocity measure is calculated for comparison with theory, it is compiled from an average over multiple cell statistics files. A benefit of calculating the local fluid velocities from many intermediate statistics files, as opposed to a single statistics file from one long run, is that the statistical distribution of the velocity measures being calculated yields information about the stability of the average velocity measure via the standard deviation.

3.9.1 Standard Deviation

Experience has shown that there exists a degree of ambiguity concerning definitions for statistical quantities, which can lead to subtle differences when comparing with theory. For these experiments, variance is defined as

$$\text{var} = \frac{1}{N} \sum_{i=1}^N (x_i - \langle x \rangle)^2 \quad (3.29)$$

and the standard deviation is

$$\sigma = \sqrt{\text{var}}. \quad (3.30)$$

3.9.2 Angular Component of Velocity

The most important velocity measure employed in this analysis is the angular component of velocity, v_θ . The plots showing the angular component of velocity as a function of radius are calculated first as temporal averages, from the interrune cell statistics files, and then are averaged spatially over the sectors around a given track of constant radius.

Section 4: Runs

All simulations were performed on Hewlett-Packard series 700 workstations. The main simulation routine is a compiled C program and the postprocessing is performed within the MATLAB¹⁴ interpreted programming environment.

All simulations were performed on one of three system sizes: “small”, “medium”, and “large”. Table 4-1 shows the key parameter settings, calculated from the algorithm described in Section 3.3.1, for these systems.

TABLE 4-1: Key Parameter Settings for the Three System Sizes Used in the Simulations. All simulations used a time step of $\Delta t = 1.8 \times 10^{-11}$ seconds and the system mean free path was $\lambda = 6.25 \times 10^{-8}$ meters.

Size	r_{inner} ($\times \lambda$)	r_{outer} ($\times \lambda$)	Kn^a	No. of θ slices	No. of radial shells	No. of particles	ω_{inner} ($\frac{\text{rad}}{\text{sec}}$)
small	3	5	0.2	32	12	51200	5.17×10^8
medium	12	20	.05	126	14	235200	1.29×10^8
large	36	60	.017	377	41	2060934	4.31×10^7

a. The **Knudsen number**, Kn , is a dimensionless number that is the ratio of the mean free path to the *characteristic length* of the system. For this paper, the outer cylinder radius is chosen to be the characteristic length.

4.1 Validation Runs

Prior to any simulation runs designed to test the PLE slip extension results, the software was tested in situations where it is known how the system should evolve. Being able to simulate known phenomena was necessary to establish confidence in the software.

14. MATLAB is a registered trademark of The MathWorks, Inc., 24 Prime Park Way, Natick, Mass. 01760 (<http://www.mathworks.com>).

4.1.1 Completely Specular Walls

4.1.1.1 Conservation of Energy, and Linear and Angular Momentum

To evaluate whether the simulation was able to properly conserve both energy and momentum in a situation where it should, a series of runs were completed for the case where the walls were perfectly specular ($\alpha=0$) and inter-particle collisions were the only interactions that altered the particles' energies.

Since the simulation approximates inter-particle interactions using the hard sphere model and the molecules have no internal degrees of freedom, the energy of the system is wholly translationally kinetic. Additionally, since temperature is proportional to the average kinetic energy of the system, the conservation of energy is represented as the conservation of temperature; therefore, the energy and momenta for a system of N particles is

$$\begin{aligned}\langle T \rangle &= \frac{2}{3k} \langle \text{KE} \rangle \\ \langle |\mathbf{P}| \rangle &= \frac{m}{N} \sum_{i=1}^N \sqrt{(v_x^2 + v_y^2 + v_z^2)_i} \\ \langle L_z \rangle &= \frac{m}{N} \sum_{i=1}^N (xv_y - yv_x)_i\end{aligned}\tag{4.1}$$

where,

$$\langle \text{KE} \rangle = \frac{m}{2N} \sum_{i=1}^N (v_x^2 + v_y^2 + v_z^2)_i.\tag{4.2}$$

The simulation was run in which each run was comprised of 1000 time steps and each time step was 1.8×10^{-11} seconds long. The equations in (4.1) were applied to each of the intermediate state files to generate Table 4-2 to assess to what degree the energy and momenta were conserved.

TABLE 4-2: Interrun Energy and Momenta Statistics for Completely Reflective Walls

time (ns)	$\langle T \rangle$	$\sigma(\langle T \rangle)$	$\langle P \rangle$	$\sigma(\langle P \rangle)$	$\langle L_z \rangle$	$\sigma(\langle L_z \rangle)$
18	300.000000	245.449566	2.645295e-23	1.115888e-23	-9.919573e-33	4.264735e-30
36	300.000000	243.354948	2.648278e-23	1.108788e-23	-1.050476e-33	4.288315e-30
54	300.000000	244.688775	2.645045e-23	1.116479e-23	-1.893699e-33	4.288321e-30
72	300.000000	244.782644	2.644661e-23	1.117389e-23	9.521125e-33	4.257940e-30
90	300.000000	243.840280	2.645669e-23	1.115002e-23	3.311199e-32	4.274007e-30
108	300.000000	244.856748	2.646119e-23	1.113932e-23	5.673546e-32	4.281617e-30
126	300.000000	244.852953	2.644631e-23	1.117461e-23	4.682429e-32	4.278032e-30
144	300.000000	245.083476	2.644984e-23	1.116625e-23	2.678852e-32	4.253554e-30
162	300.000000	244.626428	2.646622e-23	1.112737e-23	1.303620e-32	4.289433e-30
180	300.000000	246.862137	2.642261e-23	1.123053e-23	3.661116e-33	4.275347e-30
198	300.000000	245.814824	2.644076e-23	1.118773e-23	-2.188554e-34	4.270831e-30
216	300.000000	244.874291	2.645210e-23	1.116090e-23	-1.055630e-32	4.280235e-30
234	300.000000	247.139229	2.643067e-23	1.121155e-23	-2.497077e-32	4.265879e-30
252	300.000000	244.205593	2.644800e-23	1.117062e-23	-2.788155e-32	4.269807e-30
270	300.000000	244.161086	2.646341e-23	1.113405e-23	-2.000591e-32	4.263933e-30
288	300.000000	243.652331	2.646753e-23	1.112425e-23	-2.535627e-32	4.294938e-30
interrun mean	300.000000	---	2.645238e-23	---	4.239081e-33	---
interrun sdev	2.720620e-11	---	1.450508e-26	---	2.565230e-32	---

At each interrun time, the table shows the average temperature and linear and angular momentum. The standard deviation columns reflect the variability of the measures between particles at the same instant, whereas the interrun standard deviation reflects variability of the instantaneous averages over time.

It is clearly seen that the temperature is constant to excellent accuracy with a standard deviation thirteen orders of magnitude down from the average. From this observation it is evident that energy is conserved to excellent approximation.

Examination of the interrune deviation of the linear momentum reveals that linear momentum appears to be conserved on average since the interrune standard deviation is 3 orders of magnitude down from the averages; however, there is not precise conservation between runs. This result is not surprising considering that the elastic collisions with stationary walls do not conserve linear momentum.

Angular momentum, too, is conserved on the average. To see this, consider the fact that the average angular momentum should be zero because all of the particles were initialized with the same constant velocity ($v_x=v_{rms}$, $v_y=0$, $v_z=0$) and the particles were placed within the annular region randomly. The variations of the angular momentum values are distributed about zero; however, they aren't strictly zero due to the fact that the collision algorithm selects particles for collision that aren't necessarily in the same location in space. To illustrate this point, consider the two particles in Fig. 4-1 which lie along the same radial line with identical but opposite velocities, separated by some distance δr .

Fig. 4-1. An example collision between identical particles that illustrates how the collision algorithm introduces a non-exact conservation of angular momentum.

These particles, even though separated, can be selected to undergo a collision due to the fact that the selection mechanism can choose any pair in a cell. As Fig. 4-1 depicts, the

initial total angular momentum for this example is $-m\delta r v_\theta$, whereas the final total angular momentum is $m\delta r v_\theta$.

Any inter-particle collision will, in general, not conserve angular momentum; however, if a system is in equilibrium, the distribution of particle velocities within a cell will remain constant, and any large number of collisions between randomly selected pairs within the cell will result in a constant average angular momentum.

The fact that angular momentum is not conserved in DSMC collisions was at one time thought to be a flaw in the method.¹⁵ G. A. Bird introduced a modification to DSMC in which this conservation law is not violated. However, he also performed careful studies that demonstrated that the modification produced no significant effect on the velocity profiles in rotating flows.¹⁶

Apart from these complications with angular momentum in particle-particle interactions, the angular momentum in specular particle-wall interactions is conserved.

Thus from the above considerations, there is conservation of energy and average conservation of linear and angular momentum to excellent approximation.

4.1.1.2 Evolution of a Uniform Velocity Distribution to a Maxwell-Boltzmann Distribution

Another confirmation of the validity of the simulation of inter-particle interactions, in the absence of thermalizing walls, comes from the evolution of the initially uniform velocity distribution to the Maxwell-Boltzmann distribution.

As illustrated in Fig. 4-2, the inter-particle collisions equilibrate the velocity to the Maxwell-Boltzmann distribution within 18 ns.

15. E. Meiburg, Phys. Fluids, **29**, 3107 (1986).

16. G.A. Bird, Phys. Fluids, **30**, 364 (1987).

Fig. 4-2. Evolution of an initially uniform velocity distribution to a Maxwell-Boltzmann distribution from particle-particle interactions only. The initial uniform velocity distribution (a), settles to MB within 18 ns (b). The dotted line represents the MB distribution for a system with an average temperature of 300 K.

4.1.2 Maxwell-Boltzmann Distribution from Thermal Interactions with Stationary Walls

The next test involved shutting off the inter-particle collisions and allowing the particles' energy and momenta to change only through particle-wall interactions.

In Fig. 4-3, the average temperature of the system of particles at the time the timeslice was taken is used to determine the Maxwell-Boltzmann distribution. From the histogram of velocities at different times in the simulation, it can be seen that the initial singular velocity distribution evolves to match a Maxwell-Boltzmann distribution with the characteristic temperature of the walls. This sequence of histograms shows that the particle-wall collision algorithm produces the correct statistics for a gas of homogeneous particles in thermal equilibrium.

Fig. 4-3. Evolution of an initial uniform velocity distribution to a Maxwell-Boltzmann distribution from interaction with thermalizing walls only. The histograms are for timeslices at 0.036 ns (a), 0.36 ns (b), 1.08 ns (c), and 3.6 ns (d). The dotted line represents that MB distribution that would result from a system in equilibrium at the average temperature of the system of particles.

4.1.3 The Simulation's Approach to Navier-Stokes

The final test involved comparing the velocity profiles generated via the DSMC simulation with the no-slip Navier-Stokes velocity profiles. By comparing the results of the “small” ($Kn = 0.2$), “medium” ($Kn = 0.05$), and “large” ($Kn = 0.017$) systems, for fully thermalizing walls ($\alpha = 1$), to the theoretical solution of the no-slip Navier-Stokes, we should expect the velocity profiles of the DSMC solution to approach Navier-Stokes as the system size gets larger since the slip length becomes very small compared to the

characteristic length of the system. The diminished relative size of the slip length makes the slip boundary condition appear more like the no-slip boundary condition.

Fig. 4-4. Comparison of velocity profiles for completely thermalizing walls. As the simulated system size goes from (a) “small” to (b) “medium” and then to (c) “large”, the DSMC velocity profile more closely matches the profile as calculated by the PLE extension to slip, shown as a dashed line. Additionally, both profiles tend to the no-slip Navier-Stokes solution, shown as a dot-dashed line.

As Fig. 4-4 demonstrates, the DSMC simulation (circles) approaches the Navier-Stokes solution (the dot-dashed line) as the system tends to larger dimensions. Additionally, the fact that the Navier-Stokes with the PLE extension solution (the dashed line) follows DSMC very closely lends additional validation to the PLE extension for systems with completely thermalizing walls.

It would be desirable to be able to show an exact correspondence between DSMC and Navier-Stokes; however, it should be noted that the computation becomes progressively more demanding as the system gets larger, and that the “large” system represents the best compromise between size and computation time.

The above comparison with Navier-Stokes and the PLE slip extension is a compelling demonstration of the simulation's ability to correctly model fluid behavior.

4.2 Experimental Runs

As stated earlier, the PLE extension to the slip boundary condition for the Navier-Stokes equation predicts a reversal of slope for the angular component of velocity as a function of radius. This predicted crossover point, for the slope reversal, is encountered when the ratio of slip length to the radius of the outer cylinder becomes unity.

Since slip length can be expressed as a function of the surface's specularity through the parameter α , the predicted angular component of velocity can be plotted as a function of radius for various α 's to reveal a reversal in slope.

Fig. 4-5 shows this slope reversal. It is interesting to note that in their paper, Panzer, et al. infer the extrapolation of the velocity field to zero within the fluid from the zero crossing of the radial derivative of the θ -component of fluid velocity. Fig. 4-5 clearly shows positive angular components of velocity regardless of α , thereby discounting any anomalous reversal of flow as alluded to by Panzer, et al.

A simulation of the “small” system for which $\alpha = 0.01$ is shown, in Fig. 4-6, confirming this reversed slope. The velocity profile of a rotating solid body is also shown for comparison. The solid body curve has a rotation velocity that is calculated as a least squares fit to the DSMC data.

Fig. 4-7 shows a number of velocity profiles for the “small” system from simulations with different α 's. This sequence of plots illustrate how the velocity profile transitions from a system with “sticky” walls to one with “slippery” walls. This sequence illustrates that as

Fig. 4-5. The predicted reversal of slope for the angular component of velocity as a function of radius for the PLE extension to the slip boundary condition for various values of α 's for the "small" system.

the surface accommodation is reduced (large slip at the wall), the velocity profiles do indeed transition from a curve one would expect to the anomalous reversed slope curve.

By averaging the standard deviations of v_θ in Fig. 4-7, the velocity variabilities can be plotted for each value of α . This variability vs α , shown in Fig. 4-8, is presented as an interesting observation that as the walls become more "slippery" the variability of the velocity measure, v_θ , goes up.

Fig. 4-6. Angular component of velocity vs. radius for the case of highly specular walls ($\alpha = 0.01$) exhibiting a profile with opposite slope to that of the case where the walls were highly thermalizing. For comparison, the velocity curve for the Navier-Stokes equation with the PLE extension is shown as a dashed line. Additionally, the solid body velocity curve (with angular velocity of $5.46 \times 10^{-7} \frac{\text{rad}}{\text{sec}}$) is shown as a dot-dashed line.

Fig. 4-7. The v_θ vs. r plots for various α 's illustrating the transition from completely thermal walls to highly specular ones. The values of α were 1.0 (a), 0.7 (b), 0.4 (c), 0.3 (d), 0.2 (e), 0.1 (f), 0.05 (g), and 0.01 (h).

Fig. 4-8. A plot of average (along r) standard deviations of v_θ from the plots found in Fig. 4-7.

Section 5: Discussion

5.1 Mechanism Involved in the Slope Reversal of v_θ vs. r

In an attempt to understand by what mechanism this reversal of slope occurs, an additional simulation was performed where a previously equilibrated system, with completely thermalizing walls ($\alpha = 1$), was instantaneously switched to a system with completely specular and stationary walls. Fig. 5-1 shows the radial dependence of v_θ and $|v|^2$ of the

Fig. 5-1. The initial state of a simulation in which interactions with the walls are switched from completely thermal to completely specular ($\alpha=0$). The angular component of velocity for the pre-spun system is shown in graph (a) along with the PLE solution to the Navier-Stokes equation for comparison. Additionally, the track average of $|v|^2$, as a function of radius, is shown in graph (b) to illustrate that temperature (i.e. energy) is non-uniform throughout the gas.

system prior to the resetting of the walls. Once the walls were made specular, the simulation proceeded as a series runs consisting of single time step of 1.8×10^{-11} sec. Those runs between 40 and 60 time steps were averaged to produce the graphs in Fig. 5-2.

To further isolate the probable source of the slope reversal, another simulation was run that was identical to the above with the exception that inter-particle collisions were shut off. The same time span averages for the radial dependence of v_θ and $|v|^2$ are shown in Fig. 5-3.

It is clear, as evidenced by Fig. 5-2(a) and Fig. 5-3(a), that inter-particle collisions are required to elicit the reversal of slope. This observation supports two possible hypotheses: *slope reversal is either an angular momentum diffusion process, or an artifact of the non-exact angular momentum conservation of the inter-particle collisions.*

Fig. 5-2. This figure shows the effect of switching the particle-wall interactions from completely thermal to completely specular on the pre-spun system. Graph (a) shows a reversal of slope of v_θ along side the velocity curve for a solid body with angular velocity of $9.74 \times 10^7 \frac{\text{rad}}{\text{sec}}$ and graph (b) shows a homogenization of temperature.

Fig. 5-3. This figure shows the effect of switching the particle-wall interactions from completely thermal to completely specular on the pre-spun system without particle-particle interactions. Graph (a) shows a radial dependence of v_θ with a slope like that of the pre-spun state and graph (b) shows a temperature homogenization.

The error introduced into the angular momentum from a processed collision pair, δL , is a function of the distance between the colliding particles (see Section 4.1.1.1). If a system with a finer mesh is used, reducing the magnitude of possible separations, and the coarse mesh plots of v_θ vs. r are qualitatively identical to those of the finer mesh, then the reversal of slope is most likely not due to the non-exact angular momentum conserving collisions.

To evaluate this possible dependence of the slope reversal on the δL error, another simulation was performed in which the cell sizes were shrunk such that the size of Δr and $\Delta\theta$ were each reduced by a factor of 4. This fine meshed simulation was run with the key parameter settings shown in Table 5-1.

TABLE 5-1: Key Parameter Settings for the fine meshed pre-spun system. This simulation used a time step of $\Delta t = 4.5 \times 10^{-12}$ seconds and the system mean free path was $\lambda = 6.25 \times 10^{-8}$ meters.

Size	r_{inner} ($\times \lambda$)	r_{outer} ($\times \lambda$)	Kn	No. of θ slices	No. of radial shells	No. of particles	ω_{inner} ($\frac{\text{rad}}{\text{sec}}$)
small_max	3	5	0.2	128	48	250000	5.17×10^8

With the increase in computational load due to the finer mesh, the smallest number of particles in a cell was chosen to be about 30, which is still above the lower limit of approximately 25 representative particles per cell,¹⁷ as discussed in Section 3.3.

Fig. 5-4 shows the state of the fine mesh simulation prior to the switching of the walls from completely thermal to completely specular. A comparison with Fig. 5-1 reveals the two initial states to be qualitatively the same.

For the fine mesh simulation, the first 16 runs consisted of 10 time steps of 4.5×10^{-12} sec, after which, the number of time step per run was changed to 4 to match the time span per

17. M.A. Fallavollita, J.D. McDonald and D. Baganoff, Comp. Sci. Eng. 3, 283 (1992).

run to that of the coarse mesh simulation. The next 20 runs were then averaged to yield Fig. 5-5

Fig. 5-4. The initial state of the fine mesh simulation in which interactions with the walls are switched from completely thermal to completely specular. Graph (a) is the pre-spun radial dependence of v_θ and graph (b) is the pre-spun radial dependence of $|v|^2$.

Fig. 5-5. This figures shows the effect of switching the particle-wall interactions from completely thermal to completely specular on the fine mesh pre-spun system. Graph (a) shows a reversal of slope of v_θ along side the velocity curve for a solid body with angular velocity of $9.48 \times 10^7 \frac{\text{rad}}{\text{sec}}$ and graph (b) shows a homogenization of temperature. Each particle experienced an average of 3.3 collisions over the timespan the average was performed.

By the comparison of Fig. 5-5 and Fig. 5-2, it can be concluded that the angular momentum non-conservation introduced by the collision algorithm is not a contributing factor to the phenomenon of slope reversal but is most likely due to a diffusion of the angular momentum towards the outer wall.

5.2 Conclusion

This paper has validated the extension to the slip boundary condition for the Navier-Stokes equation as proposed by Panzer, et al. The anomolous reversal of slope for the angular component of velocity due to an inclusion of the radius of curvature of the boundaries in the slip length calculation has been explicitly observed. It is notable that the continuum solution of Panzer et al. can qualitatively model a phenomenon like that of a particle simulation in a realm normally reserved for particle methods ($Kn > 0.1$).

It has also been demonstrated that the mechanism involved in the slope reversal phenomenon is most likely a diffusion of angular momentum towards the outer edges of the annular region. Once this diffusion process equilibrates the system, the gas then rotates as if it were a solid body.

Bibliography

- D. Einzel, P. Panzer, M. Liu, Physical Review Letters, **64**, No. 19 (1990)
- P. Panzer and M. Liu, D. Einzel, Int. J. of Mod. Phys. B, **6**, No. 20 (1992)
- L. D. Landau and E. M. Lifshitz, Fluid Mechanics, Vol. 6 of *Course of Theoretical Physics*, Pergamon Press (1959)
- C. Cercignani, *Theory and Application of the Boltzmann Equation*, Scottish Academic Press, Edinburgh (1975)
- G. A. Bird, *Molecular Gas Dynamics*, Claredon Press, Oxford (1972)
- S. Chapman and T. G. Cowling, *The Mathematical Theory of Non-uniform Gases*, Cambridge Univ. Press, Cambridge (1990)
- M.A. Fallavollita, J.D. McDonald and D. Baganoff, Comp. Sci. Eng. **3** (1992)
- D. E. Knuth, Seminumerical Algorithms, 2nd ed., vol. 2 of *The Art of Computer Programming*, Addison-Wesley, Reading Mass. (1981)
- S.K. Park and K.W. Miller, Comm. A.C.M. **32**, 1192 (1988)
- S. Harbison and G. Steele, Jr., *C, A Reference Manual*, Prentice-Hall, Englewood Cliffs (1984)
- W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C, The Art of Scientific Computing*, 2nd ed., Cambridge Univ. Press (1992)
- F. Reif, *Fundamentals of Statistical and Thermal Physics*, McGraw-Hill (1965)
- E. Meiburg, Phys. Fluids, **29** (1986)
- G.A. Bird, Phys. Fluids, **30** (1987)

Appendices

A.1 Cartesian and Cylindrical-Polar Conversions for Position and Velocity

The coordinate transformations between cartesian and cylindrical polar representations of position and velocity vectors derive from the standard portrayal of the cartesian system.

Fig. A1-1. Geometry for the conversion of position and velocity vectors between cartesian and cylindrical polar-coordinates.

From Fig. A1-1, it is seen that the transformation from cartesian to cylindrical-polar is

$$\begin{aligned} x &= \rho \cos \theta \\ y &= \rho \sin \theta \end{aligned} \quad (\text{A1.1})$$

and the inverse transformation is

$$\begin{aligned} \rho &= \sqrt{x^2 + y^2} \\ \theta &= \text{atan}\left(\frac{y}{x}\right) \end{aligned} \quad (\text{A1.2})$$

The inverse tangent was calculated with the `atan2` function, which returns values for θ that lie in the set $\theta \in [-\pi, \pi]$. In order to maintain all values of θ to within the range $\theta \in [0, 2\pi)$, any negative values of θ are recalculated from

$$\theta \leftarrow \theta + 2\pi. \quad (\text{A1.3})$$

To calculate the tranformation equations for the velocity vector, the time derivative of the equations in (A1.1) is performed to get

$$\begin{aligned} v_x = \dot{x} &= \dot{\rho} \cos\theta - \rho \sin\theta \dot{\theta} \\ v_y = \dot{y} &= \dot{\rho} \sin\theta + \rho \cos\theta \dot{\theta} \end{aligned} \quad (\text{A1.4})$$

Adding the product of the v_x term in equation (A1.4) and $\cos\theta$ to the product of the v_y term and $\sin\theta$ yields

$$\begin{aligned} \cos\theta \dot{x} + \sin\theta \dot{y} &= \dot{\rho} \cos^2\theta - \rho \cos\theta \sin\theta \dot{\theta} + \dot{\rho} \sin^2\theta + \rho \cos\theta \sin\theta \dot{\theta} \\ &= \dot{\rho} \end{aligned} \quad (\text{A1.5})$$

Similarly, subtracting the product of the v_x and $\sin\theta$ from the product of the v_y term and $\cos\theta$ yields

$$\begin{aligned} \cos\theta \dot{y} - \sin\theta \dot{x} &= \dot{\rho} \cos\theta \sin\theta + \rho \cos^2\theta \dot{\theta} - \dot{\rho} \cos\theta \sin\theta + \rho \sin^2\theta \dot{\theta} \\ &= \rho \dot{\theta} \end{aligned} \quad (\text{A1.6})$$

Therefore, the inverse transformation for the velocity vector is

$$\begin{aligned} v_\rho &= \dot{\rho} = \cos\theta \dot{x} + \sin\theta \dot{y} \\ v_\theta &= \rho \dot{\theta} = \cos\theta \dot{y} - \sin\theta \dot{x} \end{aligned} \quad (\text{A1.7})$$

A.2 Wall Interaction Calculations

This section of the appendix outlines the calculations and algorithms used for determining whether a particle interacts with either the inner or outer wall during a time step, and what the resulting velocity of the particle is after an interaction.

A.2.1 Determining whether a wall interaction will occur in a time step

The system of particle and wall can be simplified to the geometry depicted in Fig. A2-2.

Fig. A2-2. Geometry of a particle's path and its possible intersection with a container wall.

Since the simulation performs a move operation on all particles in the system and then performs a collide operation, any particle's path within a given time step will be linear. At the beginning of a move operation on a particle, the particle's future position is calculated from

$$\begin{aligned} x_2 &= x_1 + v_x \Delta t \\ y_2 &= y_1 + v_y \Delta t \end{aligned} \quad (\text{A2.1})$$

The linear path lies on the line

$$y = mx + b \quad (\text{A2.2})$$

where,

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$

$$b = y_1 - \left(\frac{y_2 - y_1}{x_2 - x_1} \right) x_1$$
(A2.3)

Either container wall can be expressed as

$$x^2 + y^2 = \rho_i^2 \quad i = \text{inner, outer.}$$
(A2.4)

Squaring equation (A2.2) and plugging into equation (A2.4) gives

$$(1 + m^2)x^2 + (2bm)x + (b^2 - \rho_i^2) = 0$$
(A2.5)

which has solution

$$x = \frac{-bm \pm \sqrt{(1 + m^2)\rho_i^2 - b^2}}{(1 + m^2)}$$
(A2.6)

The algorithm for determining whether a particle will undergo a wall interaction is

1. Evaluate (A2.6) to determine x^+ and x^- .
2. Evaluate (A2.2) to determine y^+ and y^- .
3. Are the intersect points in front of the particle? If so, proceed, and if not, then process the next particle.
4. Are any of the distances between the start point and the forward intersect points less than the distance to the end point. If not, then there are no wall interactions for this particle in this time step. If so, then reset the current particle position to the closest intersect point, reset the particle's velocity according to the type of wall interaction, and then move the particle to a new position based upon its new velocity and the remainder of the time step.

A.2.2 Determining the post-interaction particle velocity

Once it has been determined that a particle will collide with a wall and its position has been reset to the intercept of its path and the wall, the velocities must be reset to properly

reflect the nature of the wall interaction. For specular interactions, the normal component of the velocity vector changes sign. For thermal interactions, the velocity is reset to a Gaussian that is biased in the direction of the wall rotation.

Specular Interactions

A point on the surface of a cylindrical wall has normal and tangential unit vectors

Fig. A2-3. Geometry of a point of contact on the surface of a cylindrical wall.

$$\begin{aligned}\hat{\mathbf{n}} &= \hat{\mathbf{r}} = \frac{x\hat{\mathbf{x}} + y\hat{\mathbf{y}}}{r} \\ \hat{\mathbf{t}} &= \hat{\boldsymbol{\theta}} = \frac{-y\hat{\mathbf{x}} + x\hat{\mathbf{y}}}{r}\end{aligned}\tag{A2.7}$$

The incoming velocity vector

$$\mathbf{v}_i = v_{ix}\hat{\mathbf{x}} + v_{iy}\hat{\mathbf{y}}\tag{A2.8}$$

can be resolved into normal and tangential components by the following

$$\begin{aligned}v_{in} &= \mathbf{v}_i \cdot \hat{\mathbf{n}} = \frac{xv_{ix} + yv_{iy}}{r} \\ v_{it} &= \mathbf{v}_i \cdot \hat{\mathbf{t}} = -yv_{ix} + xv_{iy}\end{aligned}\tag{A2.9}$$

For specular collisions, the outgoing velocity vector is

$$\mathbf{v}_o = v_{on}\hat{\mathbf{n}} + v_{ot}\hat{\mathbf{t}} \quad (\text{A2.10})$$

where

$$\begin{aligned} v_{on} &= -v_{in} \\ v_{ot} &= v_{it} \end{aligned} \quad (\text{A2.11})$$

By using equations (A2.7), (A2.9), and (A2.11), equation (A2.10) becomes

$$\mathbf{v}_o = \left(\frac{(y^2 - x^2)v_{ix} - 2xyv_{iy}}{r^2} \right) \hat{\mathbf{x}} + \left(\frac{(x^2 - y^2)v_{iy} - 2xyv_{ix}}{r^2} \right) \hat{\mathbf{y}}. \quad (\text{A2.12})$$

Thermal Interactions

Using the geometry shown in Fig. A2-3, the outgoing velocity vector from a thermal interaction is

$$\mathbf{v}_o = v_{on}\hat{\mathbf{n}} + v_{ot}\hat{\mathbf{t}} + v_{oz}\hat{\mathbf{z}} \quad (\text{A2.13})$$

where the components are derived from the following statistical distributions

$$\begin{aligned} P(v_{on})dv_n &= \pm \frac{m}{kT_w} v_{on} e^{-\frac{mv_{on}^2}{2kT_w}} dv_n \\ P(v_{ot})dv_t &= \sqrt{\frac{m}{2\pi kT_w}} e^{-\frac{m(v_{ot} - r\dot{\theta})^2}{2kT_w}} dv_t \\ P(v_{oz})dv_z &= \sqrt{\frac{m}{2\pi kT_w}} e^{-\frac{mv_{oz}^2}{2kT_w}} dv_z \end{aligned} \quad (\text{A2.14})$$

where the sign of $P(v_{on})dv_n$ is positive for the inner wall and negative for the outer wall.

The equations in (A2.14) are calculated using the technique for generating Gaussian distributions described in Section 3.4.3.

A.3 Integration of the Integral $\int_0^{\infty} v^2 e^{-av^2} dv$.

In order to integrate $\int_0^{\infty} v^2 e^{-av^2} dv$, the integral $I = \int_0^{\infty} e^{-av^2} dv$ must first be evaluated. To integrate this integral, consider the following integral

$$I = \int_{-\infty}^{\infty} e^{-av^2} dv. \quad (\text{A3.1})$$

Introducing the change of variable

$$\begin{aligned} v &= a^{-\frac{1}{2}} x, \\ dv &= a^{-\frac{1}{2}} dx \end{aligned} \quad (\text{A3.2})$$

equation (A3.1) becomes

$$I = a^{-\frac{1}{2}} \int_{-\infty}^{\infty} e^{-x^2} dx. \quad (\text{A3.3})$$

Using two independent variables, x and y , the square of equation (A3.3) becomes

$$\begin{aligned} I^2 &= \frac{1}{a} \int_{-\infty}^{\infty} e^{-x^2} dx \int_{-\infty}^{\infty} e^{-y^2} dy \\ &= \frac{1}{a} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} e^{-(x^2 + y^2)} dx dy \end{aligned} \quad (\text{A3.4})$$

Changing the variables in equation (A3.4) to polar coordinates by using the standard cartesian to polar relationships

$$\begin{aligned} x^2 + y^2 &= r^2 \\ dx dy &= r dr d\theta \end{aligned} \quad (\text{A3.5})$$

transforms equation (A3.4) into

$$\begin{aligned}
\mathbf{I}^2 &= \frac{1}{a} \int_0^{2\pi} \int_0^\infty e^{-r^2} r dr d\theta \\
&= \frac{2\pi}{a} \int_0^\infty e^{-r^2} r dr
\end{aligned} \tag{A3.6}$$

Equation (A3.6) can be solved by recognizing that

$$\begin{aligned}
d(e^{-r^2}) &= -2re^{-r^2} dr \\
\therefore e^{-r^2} r dr &= -\frac{1}{2} d(e^{-r^2})
\end{aligned} \tag{A3.7}$$

Substituting (A3.7) into (A3.6) yields

$$\begin{aligned}
\mathbf{I}^2 &= -\frac{\pi}{a} \int_0^\infty d(e^{-r^2}) \\
&= -\frac{\pi}{a} (e^{-r^2}) \Big|_0^\infty \\
&= \frac{\pi}{a}
\end{aligned} \tag{A3.8}$$

Solving (A3.8) for $\mathbf{I}$ gives the value of the integral in equation (A3.1) as

$$\int_{-\infty}^\infty e^{-av^2} dv = \sqrt{\frac{\pi}{a}}. \tag{A3.9}$$

The utility of (A3.9) becomes apparent when it is recognized that

$$\int_0^\infty v^2 e^{-av^2} dv = -\frac{d}{da} \left(\int_0^\infty e^{-av^2} dv \right). \tag{A3.10}$$

Since (A3.9) has an even integrand, (A3.10) becomes

$$\int_0^\infty v^2 e^{-av^2} dv = -\frac{d}{da} \left(\frac{1}{2} \sqrt{\frac{\pi}{a}} \right) \tag{A3.11}$$

$$\therefore \int_0^\infty v^2 e^{-av^2} dv = \frac{\sqrt{\pi}}{4} a^{-\frac{3}{2}}. \tag{A3.12}$$

A.4 Program Listings

The simulation code consists of a main C program with a collection of included C routines. After the data was generated, it was post-processed using a collection of MATLAB M-files. Fig. A4-4 illustrates the relationship between the various components of the C routines and M-files and briefly describes their function. If multiple routines call the same function, then only the first instance of that function is listed in the subsequent program listings.

Simulation Code

polar_couette.c	
struct.h	
- struct system_data	Structure containing system state
- struct file_data	Structure containing open file pointers and names
file_io.c	
- fopenerr()	Prints error messages if file io fails
- open_files()	Opens files and returns filename and pointer data
- close_files()	Closes opened files
struct_init.c	
- system_init()	Initializes particles and state information
- merr()	Prints error message if malloc fails
move.c	
- move()	Moves particles and processes wall interactions
sort.c	
- sort()	Sorts particles according to cell membership
collide.c	
- collide()	Processes inter-particle collisions
sample.c	
- sample()	Samples particle velocities for cell statistics file
random.c	
- ran()	Generates a uniform deviate also used in struct_init(), move(), and collide()

Post-processing Code

compile_sta.m	Compiles state files statistics (e.g. Table 4-2)
next_basename.m	Increments the filename for a compilation of multiple run data.
pcpp.m	Used to generate many different plots, but mainly velocity histograms and single run velocity profiles
bits.m	Used for binary decoding for a case-like situation
vprof_th.m	Panzer, Liu, & Einzel velocity profile
vplot.m	Plots PLE velocity for multiple α 's
vprof_th.m	
compile_cst.m	Compiles interrune cell statistics and used for plotting velocity profiles (e.g. Fig. 4-7)
vprot_th.m	
vprof_th_noslip.m	No-slip Navier-Stokes velocity profile
next_basename.m	

Fig. A4-4. Program map for the simulation code and postprocessing code.

```

/*****
/****
/**** polar_couette.c
/****
/****
/****
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <string.h>
#include "struct.h"
#include "random.c"
#include "file_io.c"
#include "struct_init.c"
#include "move.c"
#include "sort.c"
#include "sample.c"
#include "collide.c"

#ifndef COLLISIONS
# define COLLISIONS      1          /* used to turn collision on or off */
#endif

#define MASS              (6.633526e-26) /* kg */
#define T                 (300)          /* K */
#define PART_DENSITY      (2.686763e25) /* num_particles/m^3 at STP
NOTE: If you change this number
here, then you must change the
same number in the MATLAB routine
pcpp.m */

#define MIN_PART          (20)          /* minimum number of parliamentary
particles within the smallest
cell */

#define DIAMETER          (3.66e-10)    /* meters, effective diameter of
Argon. Some numbers:
1.76e-10 = atomic diameter
4.67e-10 = van Der Waal diameter
NOTE: If you change this number
here, then you must change the
same number in the MATLAB routine
pcpp.m */

#define RSEED              (-1)

int main (int argc, char *argv[]) {
    long seed, npart, ncellr, ncelltheta, eff_parts, num_timesteps, i, burns;
    long recomm_ncelltheta, recomm_ncellr, recomm_npart, timestamp,
        use_init_state, j;
    long *tot_strikes, coltot, init_seed;
    int wait;
    double inner_wall_rad, outer_wall_rad, inner_wall_angvel,
        outer_wall_angvel, eff_mass, mfp, mpv, deltheta, delr, delt, dummy;
    double recomm_delt, alpha, *tot_delvr, *tot_delvtheta;
    char cfg_infile[80], p_str[80];
    struct system_data sys_dat;
    struct file_data file_dat;

```

```

struct timeval tv_start, tv_end, tv_lapsed;
struct timezone tz;

if(!(tot_strikes = (long *) malloc(2 * sizeof(long)))) merr("tot_strikes");
tot_strikes[0] = 0; tot_strikes[1] = 0;

if(!(tot_delvr = (double *) malloc(2 * sizeof(double)))) merr("tot_delvr");
tot_delvr[0] = 0.0; tot_delvr[1] = 0.0;

if(!(tot_delvtheta = (double *) malloc(2 * sizeof(double))))
    merr("tot_delvtheta");
tot_delvtheta[0] = 0.0; tot_delvtheta[1] = 0.0;

/** Process Command Line **/
if (argc == 0) exit(-1);

/* allow a wait state for xdb process snagging */
if (argc > 2) {
    wait = 1;
    printf("waiting . . .");
    while (wait) {};
}

/** Setup input and output files **/
strcpy(cfg_infile, argv[1]);
file_dat = open_files(cfg_infile);

gettimeofday(&tv_start, &tz);
timestamp = (long) tv_start.tv_sec;

/** Setup System Size **/
fscanf(file_dat.fp_cfg_in, "%s %ld", p_str, &burns);
fprintf(file_dat.fp_cfg_out, "%s %ld\n", p_str, burns);

fscanf(file_dat.fp_cfg_in, "%s %lf", p_str, &inner_wall_rad);
fprintf(file_dat.fp_cfg_out, "%s %17.14e\n", p_str, inner_wall_rad);

fscanf(file_dat.fp_cfg_in, "%s %lf", p_str, &outer_wall_rad);
fprintf(file_dat.fp_cfg_out, "%s %17.14e\n", p_str, outer_wall_rad);

fscanf(file_dat.fp_cfg_in, "%s %lf", p_str, &inner_wall_angvel);
fprintf(file_dat.fp_cfg_out, "%s %17.14e\n", p_str, inner_wall_angvel);

fscanf(file_dat.fp_cfg_in, "%s %lf", p_str, &outer_wall_angvel);
fprintf(file_dat.fp_cfg_out, "%s %17.14e\n", p_str, outer_wall_angvel);

mfp = 1.0 / (sqrt(2.0) * PART_DENSITY * M_PI * DIAMETER * DIAMETER);
mpv = sqrt(2.0 * BOLTZ * T / MASS);

/** Setup mesh **/
deltheta = mfp / (3 * inner_wall_rad);
recomm_ncelltheta = (long) (TWOPI / deltheta);
fscanf(file_dat.fp_cfg_in, "%s %ld", p_str, &ncelltheta);
fprintf(file_dat.fp_cfg_out, "%s %ld\n", p_str, ncelltheta);
deltheta = TWOPI / ncelltheta;

```



```

delr = inner_wall_rad * deltheta;
recomm_ncellr = (long) ((outer_wall_rad - inner_wall_rad) / delr);
fscanf(file_dat.fp_cfg_in, "%s %ld", p_str, &ncellr);
fprintf(file_dat.fp_cfg_out, "%s %ld\n", p_str, ncellr);
delr = (outer_wall_rad - inner_wall_rad) / ncellr;

recomm_delt = (mfp * inner_wall_rad) / (12 * mpv * outer_wall_rad);
fscanf(file_dat.fp_cfg_in, "%s %lf", p_str, &delt);
fprintf(file_dat.fp_cfg_out, "%s %17.14e\n", p_str, delt);

fscanf(file_dat.fp_cfg_in, "%s %ld", p_str, &num_timesteps);
fprintf(file_dat.fp_cfg_out, "%s %ld\n", p_str, num_timesteps);

/** Setup Number of particles **/
/* Note: The number of real particles is set by STP and interstitial
   volume. What is calculated here is the number of "parliamentary"
   particles. */
recomm_npart = MIN_PART * (M_PI * (outer_wall_rad * outer_wall_rad -
    inner_wall_rad * inner_wall_rad)) / ((deltheta / 2) *
    ((inner_wall_rad + delr) * (inner_wall_rad + delr) -
    inner_wall_rad * inner_wall_rad));
fscanf(file_dat.fp_cfg_in, "%s %ld", p_str, &npart);
fprintf(file_dat.fp_cfg_out, "%s %ld\n", p_str, npart);

/* eff_mass calculation uses PART_DENSITY. Since we are multiplying
   by the cross sectional area, this calculation assumes that the system
   extends into the z-axis by a unit length */
eff_parts = PART_DENSITY * M_PI * (outer_wall_rad * outer_wall_rad -
    inner_wall_rad * inner_wall_rad) / npart;
eff_mass = MASS * eff_parts;

fscanf(file_dat.fp_cfg_in, "%s %lf", p_str, &alpha);
fprintf(file_dat.fp_cfg_out, "%s %17.14f\n", p_str, alpha);

fscanf(file_dat.fp_cfg_in, "%s %ld", p_str, &use_init_state);
fprintf(file_dat.fp_cfg_out, "%s %ld\n", p_str, 1);

fscanf(file_dat.fp_cfg_in, "%s %ld", p_str, &seed);

/** Initialize random number generator **/
init_seed = seed;
seed = - labs(seed); ran(&seed);
for (i=0; i<burns; i++) {
    dummy = ran(&seed);
}

/** Setup system data structure **/
sys_dat = system_init(npart, ncellr, ncelltheta, inner_wall_rad,
    outer_wall_rad, inner_wall_angvel, outer_wall_angvel,
    eff_mass, MASS, T, &seed, use_init_state);
sys_dat.alpha = alpha;
sys_dat.coeff = 0.5 * eff_parts * M_PI * DIAMETER * DIAMETER * delt;

/** Load previous state file if we are picking up from a previous run **/

```

```

if (use_init_state == 1) {
    for (i=0; i<npart; i++) {
        fscanf(file_dat.fp_sta_in, "%lf %lf %lf %lf %lf %lf",
            &sys_dat.x[i], &sys_dat.y[i], &sys_dat.z[i],
            &sys_dat.vx[i], &sys_dat.vy[i], &sys_dat.vz[i]);
        sys_dat.x0[i] = sys_dat.x[i];
        sys_dat.y0[i] = sys_dat.y[i];
        sys_dat.r[i] = sqrt(sys_dat.x[i] * sys_dat.x[i] +
            sys_dat.y[i] * sys_dat.y[i]);
        sys_dat.theta[i] = atan2(sys_dat.y[i], sys_dat.x[i]);
        if (sys_dat.theta[i] < 0) sys_dat.theta[i] += TWOPI;
        sys_dat.vr[i] = cos(sys_dat.theta[i]) * sys_dat.vx[i] +
            sin(sys_dat.theta[i]) * sys_dat.vy[i];
        sys_dat.vtheta[i] = cos(sys_dat.theta[i]) * sys_dat.vy[i] -
            sin(sys_dat.theta[i]) * sys_dat.vx[i];
    }
}

/* Perform an initial sort to initialize sort dependent values in sys_dat */
sort(&sys_dat);

coltot = 0;
/** Step over all time steps */
for (i=0; i<num_timesteps; i++) {
    move(&sys_dat, delt, i, &seed, MASS, mpv);
    tot_strikes[0] += sys_dat.strikes[0];
    tot_strikes[1] += sys_dat.strikes[1];
    tot_delvr[0] += sys_dat.delvr[0];
    tot_delvr[1] += sys_dat.delvr[1];
    tot_delvtheta[0] += sys_dat.delvtheta[0];
    tot_delvtheta[1] += sys_dat.delvtheta[1];
    sort(&sys_dat);
    if (COLLISIONS) {
        collide(&sys_dat, &seed);
        coltot += sys_dat.col;
    }
    sample(&sys_dat);
    fprintf(file_dat.fp_traj_out,
        "%17.14e %17.14e %17.14e %17.14e %17.14e %17.14e\n",
        sys_dat.x[0], sys_dat.y[0], sys_dat.z[0], sys_dat.vx[0],
        sys_dat.vy[0], sys_dat.vz[0]);
    if ((j = i % 50) == 0) printf("tstep = %ld\n", i);
}

gettimeofday(&tv_end, &tz);
if (tv_start.tv_usec > tv_end.tv_usec) {
    tv_end.tv_usec += 1000000;
    tv_end.tv_sec--;
}
tv_lapsed.tv_usec = tv_end.tv_usec - tv_start.tv_usec;
tv_lapsed.tv_sec = tv_end.tv_sec - tv_start.tv_sec;

/** Finish writing out config file */
fprintf(file_dat.fp_cfg_out, "%s %ld\n", p_str, seed);

```



```

fprintf(file_dat.fp_cfg_out, "Total Elapsed Simulation time:\t%8.5e (sec)\n",
        (double) (tv_lapsed.tv_sec + 1e-6 * tv_lapsed.tv_usec));

/** Write out state file **/
for (i=0;i<npart;i++) {
    fprintf(file_dat.fp_sta_out,
            "%17.14e %17.14e %17.14e %17.14e %17.14e %17.14e\n",
            sys_dat.x[i], sys_dat.y[i], sys_dat.z[i],
            sys_dat.vx[i], sys_dat.vy[i], sys_dat.vz[i]);
}

/** Time avg cell statistics and write to cell statistics file **/
for (i=0;i<sys_dat.ncell;i++) {
    fprintf(file_dat.fp_cst_out,
            "%17.14e %17.14e %17.14e %17.14e %17.14e %17.14e %17.14e\n",
            sys_dat.accum_n[i] / num_timesteps,
            sys_dat.accum_vx[i] / num_timesteps,
            sys_dat.accum_vy[i] / num_timesteps,
            sys_dat.accum_vz[i] / num_timesteps,
            sys_dat.accum_vr[i] / num_timesteps,
            sys_dat.accum_vtheta[i] / num_timesteps,
            sys_dat.accum_v2[i] / num_timesteps);
}

close_files(&file_dat);
}

.....
***
*** struct.h ***
***
.....
#define MAX_STR 80

struct system_data {

    /* Particle data */
    double *x;
    double *y;
    double *z;
    double *r;
    double *theta;
    double *vx;
    double *vy;
    double *vz;
    double *vr;
    double *vtheta;
    long npart;
    double *x0;          /* MUST BE SET TO THE CURRENT X,Y POSITION WHENEVER */
    double *y0;          /* THE VELOCITY OF PARTICLE IS CHANGED! */

    /* Cell data */
    long nr;              /* number of radial shells */
    long ntheta;          /* number of angular wedges */
    long ncell;           /* total number of cells */
}

```

```

long *coll_rng;      /* range of cells which contain particles that are
                     considered possible colliders with a wall. This
                     array has four elements.
                     coll_rng(0) = lowest index of cells with particles
                               that may collide with the inner wall
                     coll_rng(1) = highest index ... inner wall
                     coll_rng(2) = lowest index ... outer wall
                     coll_rng(3) = highest index ... outer wall */

double delr;         /* radial shell spacing */
double deltheta;     /* angular wedge spacing */
double *wall_rad;    /* radius of walls (0 = inner, 1 = outer) */
double *wall_rad2;   /* radius of walls squared */
double *wall_angvel; /* angular velocity of the walls */
double *wall_T;      /* temperature of the walls */
double *wall_vmp;    /* Most probable velocity off each wall */
double *cell_vol;    /* volume of each cell */
long *order;         /* Order of particles along cell progression */
long *cell_n;        /* number of particles in each cell */
long *index;         /* first particle index in each cell */
long *part_cell;     /* cell ID for each particle */
long *temp;          /* temporary holding array */
double coeff;        /* coefficient used in collide routine */

```

/* These arrays hold information accumulated of all time steps and all particles. To calculate cell stats, perform the following:

$$\text{ave density} = \frac{(\text{mass_of_part}) * (\text{accumulated_n} / \text{num_of_timesteps})}{\text{Volume_of_cell}}$$

$$\text{ave vel comp} = \frac{\text{accumulated_velocity_component}}{\text{accumulated_n}}$$

Note: The time averaging for the average velocity component is implicit in the division by the total number of particles over all time.

```

/*
double *accum_n;      /* Accumulated number of particles in a cell */
double *accum_vx;     /* Accumulated x-component of cell velocity */
double *accum_vy;     /* Accumulated y-component of cell velocity */
double *accum_vz;     /* Accumulated z-component of cell velocity */
double *accum_vr;     /* Accumulated r-component of cell velocity */
double *accum_vtheta; /* Accumulated theta-component of cell velocity */
double *accum_v2;     /* Accumulated squared cell velocity */

/* Collide data */
double *vrmax;        /* current maximum relative speed in each cell */
double *selxtra;      /* extra selection carried over to next time step */
long *col;            /* current total number of collisions */

/* Mover data */
double alpha;         /* percentage of collision that are thermalized
                     as opposed to specular */
long *strikes;        /* number of particles striking each wall */

```

```

    double *delvr;          /* change in radial velocity at each wall */
    double *delvtheta;      /* change in radial velocity at each wall */
};

struct file_data {

    char basename[MAX_STR];
    char basename_out[MAX_STR];
    char config_in[MAX_STR];
    char state_in[MAX_STR];
    char config_out[MAX_STR];
    char state_out[MAX_STR];
    char cellstat_out[MAX_STR];
    char traj_out[MAX_STR];

    FILE *fp_cfg_in;
    FILE *fp_sta_in;
    FILE *fp_cfg_out;
    FILE *fp_sta_out;
    FILE *fp_cst_out;
    FILE *fp_traj_out;

};

/*****
***                                     *****/
*** file_io.c                                     *****/
***                                     *****/
*****/
int fopenerr(char *fname) {
    fprintf(stderr, "Error opening file %s\n", fname);
    exit(-1);
}

struct file_data open_files(char *fname) {

    int dot, i, j, carry;
    size_t length;          /* unsigned int */
    char *cfg = ".cfg";
    char *sta = ".sta";
    char *cst = ".cst";
    char *traj = ".traj";
    char c_numin[2], c_numout[2];
    struct file_data file_dat;

    dot = 46;

    /*** Check input config filename validity ***/
    length = strlen(fname);

    if (length == 0) {
        fprintf(stderr, "Nonexistent filename: use format xxxx##.cfg\n");
        exit(-1);
    }
}

```

```

for (i=length;((fname[i] != dot) && (i>0));i--) {}

if (i-2) {
    fprintf(stderr, "Invalid filename: use format xxxx##.cfg\n");
    exit(-1);
}

/** Construct filenames for all input and output files */
for (j=0;j<(i-2);j++) file_dat.basename[j] = fname[j];
file_dat.basename[i-2] = '\0';

c_numin[0] = fname[i-2];
c_numin[1] = fname[i-1];
c_numin[2] = '\0';

carry = 0;
switch (c_numin[1]) {
    case '0':
        c_numout[1] = '1';
        break;
    case '1':
        c_numout[1] = '2';
        break;
    case '2':
        c_numout[1] = '3';
        break;
    case '3':
        c_numout[1] = '4';
        break;
    case '4':
        c_numout[1] = '5';
        break;
    case '5':
        c_numout[1] = '6';
        break;
    case '6':
        c_numout[1] = '7';
        break;
    case '7':
        c_numout[1] = '8';
        break;
    case '8':
        c_numout[1] = '9';
        break;
    case '9':
        c_numout[1] = '0';
        carry = 1;
        break;
    default :
        fprintf(stderr, "Not a valid filename: use format xxxx##.cfg\n");
        exit(-1);
}

if (carry == 1) {
    switch (c_numin[0]) {

```

```

    case '0':
        c_numout[0] = '1';
        break;
    case '1':
        c_numout[0] = '2';
        break;
    case '2':
        c_numout[0] = '3';
        break;
    case '3':
        c_numout[0] = '4';
        break;
    case '4':
        c_numout[0] = '5';
        break;
    case '5':
        c_numout[0] = '6';
        break;
    case '6':
        c_numout[0] = '7';
        break;
    case '7':
        c_numout[0] = '8';
        break;
    case '8':
        c_numout[0] = '9';
        break;
    case '9':
        c_numout[0] = '0';
        break;
    default :
        fprintf(stderr, "Not a valid filename: use format xxxx##.cfg\n");
        exit(-1);
}
}
else {
    c_numout[0] = c_numin[0];
}

c_numout[2] = '\0';

if ((strlen(file_dat.basename) + 7) > MAX_STR) {
    fprintf(stderr, "filename too long\n");
    exit(-1);
}

strcpy(file_dat.config_in, file_dat.basename);
strcat(file_dat.config_in, c_numin);
strcat(file_dat.config_in, cfg);

strcpy(file_dat.state_in, file_dat.basename);
strcat(file_dat.state_in, c_numin);
strcat(file_dat.state_in, sta);

strcpy(file_dat.config_out, file_dat.basename);

```



```

    strcat(file_dat.config_out, c_numout);
    strcat(file_dat.config_out, cfg);

    strcpy(file_dat.state_out, file_dat.basename);
    strcat(file_dat.state_out, c_numout);
    strcat(file_dat.state_out, sta);

    strcpy(file_dat.cellstat_out, file_dat.basename);
    strcat(file_dat.cellstat_out, c_numout);
    strcat(file_dat.cellstat_out, cst);

    strcpy(file_dat.traj_out, file_dat.basename);
    strcat(file_dat.traj_out, c_numout);
    strcat(file_dat.traj_out, traj);

    /** Open all files and check for errors */
    if(!(file_dat.fp_cfg_in = fopen(file_dat.config_in, "r")))
        fopenerr(file_dat.config_in);
    file_dat.fp_sta_in = fopen(file_dat.state_in, "r");
    if(!(file_dat.fp_cfg_out = fopen(file_dat.config_out, "w")))
        fopenerr(file_dat.config_out);
    if(!(file_dat.fp_sta_out = fopen(file_dat.state_out, "w")))
        fopenerr(file_dat.state_out);
    if(!(file_dat.fp_cst_out = fopen(file_dat.cellstat_out, "w")))
        fopenerr(file_dat.cellstat_out);
    if(!(file_dat.fp_traj_out = fopen(file_dat.traj_out, "w")))
        fopenerr(file_dat.traj_out);

    strcpy(file_dat.basename_out, file_dat.basename);
    strcat(file_dat.basename_out, c_numin);
    strcat(file_dat.basename_out, c_numout);
    /** Return structure with file pointers to opened files */
    return(file_dat);
}

void close_files(struct file_data *file_dat) {

    fclose(file_dat->fp_cfg_in);
    if (file_dat->fp_sta_in != NULL) fclose(file_dat->fp_sta_in);
    fclose(file_dat->fp_cfg_out);
    fclose(file_dat->fp_sta_out);
    fclose(file_dat->fp_cst_out);
    fclose(file_dat->fp_traj_out);

    return;
}

/*****
***
*** struct_init.c
***
*****/
/*****
#ifdef RANDOM
# define RANDOM      1          /* initializes particles with

```

```

                                random or constant velocity
                                distribution */

#endif

struct system_data system_init(long npart, long ncellr, long ncelltheta,
                                double inner_wall_rad, double outer_wall_rad,
                                double inner_wall_angvel, double outer_wall_angvel,
                                double eff_mass, double mass, double T,
                                long *seed, long use_init_state) {

    long i, num_of_cells;
    int not_done;
    double rl, temp_var, rad_span, mpv, vrms, m, costerm, sinterm;
    struct system_data sys_dat;

    /** malloc the constituents of the structure */
    if(!(sys_dat.x = (double *) malloc(npart * sizeof(double)))) merr("x");
    if(!(sys_dat.y = (double *) malloc(npart * sizeof(double)))) merr("y");
    if(!(sys_dat.z = (double *) malloc(npart * sizeof(double)))) merr("z");
    if(!(sys_dat.r = (double *) malloc(npart * sizeof(double)))) merr("r");
    if(!(sys_dat.theta = (double *) malloc(npart * sizeof(double))))
        merr("theta");
    if(!(sys_dat.vx = (double *) malloc(npart * sizeof(double)))) merr("vx");
    if(!(sys_dat.vy = (double *) malloc(npart * sizeof(double)))) merr("vy");
    if(!(sys_dat.vz = (double *) malloc(npart * sizeof(double)))) merr("vz");
    if(!(sys_dat.vr = (double *) malloc(npart * sizeof(double)))) merr("vr");
    if(!(sys_dat.vtheta = (double *) malloc(npart * sizeof(double))))
        merr("vtheta");
    sys_dat.npart = npart;
    if(!(sys_dat.x0 = (double *) malloc(npart * sizeof(double)))) merr("x0");
    if(!(sys_dat.y0 = (double *) malloc(npart * sizeof(double)))) merr("y0");

    num_of_cells = ncellr * ncelltheta;
    sys_dat.nr = ncellr;
    sys_dat.ntheta = ncelltheta;
    sys_dat.ncell = num_of_cells;

    if(!(sys_dat.coll_rng = (long *) malloc(4 * sizeof(long))))
        merr("coll_rng");
    if(!(sys_dat.wall_rad = (double *) malloc(2 * sizeof(double))))
        merr("wall_rad");
    if(!(sys_dat.wall_rad2 = (double *) malloc(2 * sizeof(double))))
        merr("wall_rad2");
    if(!(sys_dat.wall_angvel = (double *) malloc(2 * sizeof(double))))
        merr("wall_angvel");
    if(!(sys_dat.wall_T = (double *) malloc(2 * sizeof(double)))) merr("wall_T");
    if(!(sys_dat.wall_vmp = (double *) malloc(2 * sizeof(double))))
        merr("wall_vmp");

    if (outer_wall_rad < inner_wall_rad) {
        temp_var = inner_wall_rad;
        inner_wall_rad = outer_wall_rad;
        outer_wall_rad = temp_var;
    }
    rad_span = outer_wall_rad - inner_wall_rad;

```

```

sys_dat.delr = rad_span / ncellr;
sys_dat.deltheta = TWOPI / ncelltheta;
sys_dat.wall_rad[0] = inner_wall_rad;
sys_dat.wall_rad[1] = outer_wall_rad;
sys_dat.wall_rad2[0] = inner_wall_rad * inner_wall_rad;
sys_dat.wall_rad2[1] = outer_wall_rad * outer_wall_rad;
sys_dat.wall_angvel[0] = inner_wall_angvel;
sys_dat.wall_angvel[1] = outer_wall_angvel;
sys_dat.wall_T[0] = T; sys_dat.wall_T[1] = T;
sys_dat.wall_vmp[0] = sqrt(2 * BOLTZ * sys_dat.wall_T[0] / mass);
sys_dat.wall_vmp[1] = sqrt(2 * BOLTZ * sys_dat.wall_T[1] / mass);

if(!(sys_dat.cell_vol = (double *) malloc(num_of_cells * sizeof(double))))
    merr("cell_vol");
if(!(sys_dat.order = (long *) malloc(npart * sizeof(long))))
    merr("order");
if(!(sys_dat.cell_n = (long *) malloc(num_of_cells * sizeof(long))))
    merr("cell_n");
if(!(sys_dat.index = (long *) malloc(num_of_cells * sizeof(long))))
    merr("index");
if(!(sys_dat.part_cell = (long *) malloc(npart * sizeof(long))))
    merr("part_cell");
if(!(sys_dat.temp = (long *) malloc(num_of_cells * sizeof(long))))
    merr("temp");

if(!(sys_dat.accum_n = (double *) malloc(num_of_cells * sizeof(double))))
    merr("accum_n");
if(!(sys_dat.accum_vx = (double *) malloc(num_of_cells * sizeof(double))))
    merr("accum_vx");
if(!(sys_dat.accum_vy = (double *) malloc(num_of_cells * sizeof(double))))
    merr("accum_vy");
if(!(sys_dat.accum_vz = (double *) malloc(num_of_cells * sizeof(double))))
    merr("accum_vz");
if(!(sys_dat.accum_vr = (double *) malloc(num_of_cells * sizeof(double))))
    merr("accum_vr");
if(!(sys_dat.accum_vtheta = (double *) malloc(num_of_cells * sizeof(double))))
    merr("accum_vtheta");
if(!(sys_dat.accum_v2 = (double *) malloc(num_of_cells * sizeof(double))))
    merr("accum_v2");

if(!(sys_dat.vrmax = (double *) malloc(num_of_cells * sizeof(double))))
    merr("vrmax");
if(!(sys_dat.selxtra = (double *) malloc(num_of_cells * sizeof(double))))
    merr("selxtra");

if(!(sys_dat.strikes = (long *) malloc(2 * sizeof(long)))) merr("strikes");
if(!(sys_dat.delvr = (double *) malloc(2 * sizeof(double)))) merr("delvr");
if(!(sys_dat.delvtheta = (double *) malloc(2 * sizeof(double))))
    merr("delvtheta");

if (use_init_state != 1) {
    /** initialize particle positions and velocities **/
    mpv = sqrt(2 * BOLTZ * T / mass);
    vrms = sqrt(3 * BOLTZ * T / mass);
    for (i=0;i<npart;i++) {

```

```

/** randomly distribute positions with cylindrical walls */
not_done = 1;
while (not_done) {
    sys_dat.x[i] = outer_wall_rad * (2 * ran(seed) - 1);
    sys_dat.y[i] = outer_wall_rad * (2 * ran(seed) - 1);
    sys_dat.r[i] = sqrt(sys_dat.x[i] * sys_dat.x[i] +
                        sys_dat.y[i] * sys_dat.y[i]);
    sys_dat.theta[i] = atan2(sys_dat.y[i], sys_dat.x[i]);
    if (sys_dat.theta[i] < 0) sys_dat.theta[i] = sys_dat.theta[i] + TWOPI;
    costerm = sys_dat.x[i] / sys_dat.r[i];
    sinterm = sys_dat.y[i] / sys_dat.r[i];
    if ((sys_dat.r[i] > outer_wall_rad) && (sys_dat.r[i] < inner_wall_rad)) {
        not_done = 0;
    }
}

sys_dat.z[i] = 0.0;
sys_dat.x0[i] = sys_dat.x[i];
sys_dat.y0[i] = sys_dat.y[i];

if (RANDOM) {
    /** randomly distribute velocities between cylindrical walls */
    sys_dat.vx[i] = mpv * sqrt(-log(1-ran(seed))) * sin(TWOPI * ran(seed));
    sys_dat.vy[i] = mpv * sqrt(-log(1-ran(seed))) * sin(TWOPI * ran(seed));
    sys_dat.vz[i] = mpv * sqrt(-log(1-ran(seed))) * sin(TWOPI * ran(seed));
}
else {
    /** Uniform velocity distribution: For use as MB verification ONLY */
    sys_dat.vx[i] = vrms;
    sys_dat.vy[i] = 0;
    sys_dat.vz[i] = 0;
}

sys_dat.vr[i] = costerm * sys_dat.vx[i] + sinterm * sys_dat.vy[i];
sys_dat.vtheta[i] = costerm * sys_dat.vy[i] - sinterm * sys_dat.vx[i];

}

/** initialize cell dependent arrays */
for (i=0; i<num_of_cells; i++) {
    /** accumulation arrays */
    sys_dat.accum_n[i] = 0.0;
    sys_dat.accum_vx[i] = 0.0;
    sys_dat.accum_vy[i] = 0.0;
    sys_dat.accum_vz[i] = 0.0;
    sys_dat.accum_vr[i] = 0.0;
    sys_dat.accum_vtheta[i] = 0.0;
    sys_dat.accum_v2[i] = 0.0;
    r1 = inner_wall_rad + floor(i / ncelltheta) * sys_dat.delr;
    sys_dat.cell_vol[i] = (sys_dat.deltheta / 2) * sys_dat.delr *
                        (2 * r1 + sys_dat.delr);

    /** selection stuff */
    sys_dat.vrmax[i] = 3 * sqrt(2 * BOLTZ * T / mass);
}

```

```

    sys_dat.selxtra[i] = 0.0;
}

/** initialize misc stuff */
sys_dat.col = 0;
sys_dat.strikes[0] = 0; sys_dat.strikes[1] = 0;
sys_dat.delvr[0] = 0.0; sys_dat.delvr[1] = 0.0;
sys_dat.delvtheta[0] = 0.0; sys_dat.delvtheta[1] = 0.0;

/** tell program for which cells to check for wall collision */
/* for cells that just line the walls
sys_dat.coll_rng[0] = 0;
sys_dat.coll_rng[1] = ncelltheta - 1;
sys_dat.coll_rng[2] = (ncellr - 1) * ncelltheta;
sys_dat.coll_rng[3] = num_of_cells - 1;
*/
/* for all cells */
sys_dat.coll_rng[0] = 0;
sys_dat.coll_rng[1] = (num_of_cells - 1) / 2 - 1;
sys_dat.coll_rng[2] = (num_of_cells - 1) / 2;
sys_dat.coll_rng[3] = num_of_cells - 1;

return(sys_dat);
}

int merr(char *p_str) {

    printf("Unable to malloc space for %s\n", p_str);
    exit(-1);
}

*****
***                                     ***
***  move.c                           ***
***                                     ***
*****

void move(struct system_data *sys_dat, double tau, long tstep, long *seed,
          double mass, double mpv) {

    /* allocate and initialize variables */
    long i;
    double xnew, ynew, znew, delx, dely, m, b, bracket, pt1x, pt1y, pt2x, pt2y;
    double delxp1, delyp1, delxp2, delyp2, tau_remain, vox, voy, voz, xx, yy, xy;
    double von, vot, rho, phi, delvx, delvy, x_on_r, y_on_r, costerm, sinterm;
    int wall;      /* flag indicating which wall a particle may collide with
                    0 = inner    1 = outer */
    int check;     /* flag indicating whether particle is a possible collision
                    candidate
                    0 = no possible collision    1 = possible collision */
    int pip;       /* possible interaction points (pip)
                    0 = neither pt1 or pt2 are possible wall collisions
                    1 = pt1 is a possible wall collision
                    2 = pt2 is a possible wall collision
                    3 = both pt1 and pt2 are possible wall collisions */
    int wall_collision, not_done;

```

```

wall_collision = 0;
sys_dat->strikes[0] = 0; sys_dat->strikes[1] = 0;
sys_dat->delvr[0] = 0.0; sys_dat->delvr[1] = 0.0;
sys_dat->delvtheta[0] = 0.0; sys_dat->delvtheta[1] = 0.0;

/** iterate through every point */
for (i=0; i<sys_dat->npart; i++) {
    tau_remain = tau;

    while (tau_remain > 0) {

        /** advance point along a linear trajectory */
        xnew = sys_dat->vx[i] * tau_remain + sys_dat->x[i];
        ynew = sys_dat->vy[i] * tau_remain + sys_dat->y[i];
        znew = sys_dat->vz[i] * tau_remain + sys_dat->z[i];

        /** decide if we want to test for possible wall interaction */
        /* particles in inner cells are checked for inner wall collision, and
           particles in outer cells are checked for outer wall collisions */
        /* use flag to indicate which wall the particle may collide with */
        check = 0;
        if ( (sys_dat->part_cell[i] >= sys_dat->coll_rng[0]) &&
            (sys_dat->part_cell[i] <= sys_dat->coll_rng[1]) ) {
            check = 1;
            wall = 0;
        }
        if ( (sys_dat->part_cell[i] >= sys_dat->coll_rng[2]) &&
            (sys_dat->part_cell[i] <= sys_dat->coll_rng[3]) ) {
            check = 1;
            wall = 1;
        }

        if (check) {
            /** if x-components of intersection btw circle and line are real then
                proceed */
            delx = xnew - sys_dat->x[i];
            dely = ynew - sys_dat->y[i];
            m = sys_dat->vy[i] / sys_dat->vx[i];
            b = sys_dat->y0[i] - m * sys_dat->x0[i];
            bracket = sys_dat->wall_rad[wall] * sys_dat->wall_rad[wall] *
                (1 + m * m) - b * b;
            pip = 0;
            if (bracket >= 0) {
                /** Find all intersection points btw circle and line */
                pt1x = (-b * m + sqrt(bracket)) / (1 + m * m);
                pt1y = m * pt1x + b;
                pt2x = (-b * m - sqrt(bracket)) / (1 + m * m);
                pt2y = m * pt2x + b;

                /** Find intersection points ahead of particle trajectory */
                /* If the dot product btw the trajectory vector and the
                   intersect vector is positive than the intersect point is in
                   front of the particle path */
                if (wall_collision == 0) {

```

```

delxp1 = pt1x - sys_dat->x[i]; delyp1 = pt1y - sys_dat->y[i];
if ((delx * delxp1 + dely * delyp1) > 0) pip += 1;
delxp2 = pt2x - sys_dat->x[i]; delyp2 = pt2y - sys_dat->y[i];
if ((delx * delxp2 + dely * delyp2) < 0) pip += 2;
}
else { /* we are sitting on an intersect pt btw line/circle */
delxp1 = pt1x - sys_dat->x[i]; delyp1 = pt1y - sys_dat->y[i];
delxp2 = pt2x - sys_dat->x[i]; delyp2 = pt2y - sys_dat->y[i];

/* determine which one we are on and use other for pip calc */
if ((delxp1 * delxp1 + delyp1 * delyp1)
    < (delxp2 * delxp2 + delyp2 * delyp2)) { /* on pt1 */
    if ((delx * delxp2 + dely * delyp2) < 0) pip += 2;
}
else { /* on pt2 */
    if ((delx * delxp1 + dely * delyp1) < 0) pip += 1;
}

wall_collision = 0;
}

/** Proceed if we have intersect points ahead of trajectory */
if (pip > 0) {
    /** Find collision point if collision has occurred */
    /* We can only have one wall interaction per particle per step;
       therefore, only the closest intersection point is chosen */
    /* Note: sqrt eval unnecessary for distance comparisons */
    switch (pip) {
        case 1:
            if ((delx * delx + dely * dely) <
                (delxp1 * delxp1 + delyp1 * delyp1)) {
                wall_collision = 1;
                sys_dat->x[i] = pt1x;
                sys_dat->y[i] = pt1y;
                tau_remain -= sqrt(delxp1 * delxp1 + delyp1 * delyp1) /
                    sqrt(sys_dat->vx[i] * sys_dat->vx[i] +
                        sys_dat->vy[i] * sys_dat->vy[i]);
            };
            break;
        case 2:
            if ((delx * delx + dely * dely) <
                (delxp2 * delxp2 + delyp2 * delyp2)) {
                wall_collision = 1;
                sys_dat->x[i] = pt2x;
                sys_dat->y[i] = pt2y;
                tau_remain -= sqrt(delxp2 * delxp2 + delyp2 * delyp2) /
                    sqrt(sys_dat->vx[i] * sys_dat->vx[i] +
                        sys_dat->vy[i] * sys_dat->vy[i]);
            };
            break;
        case 3:
            if ((delxp1 * delxp1 + delyp1 * delyp1) <
                (delxp2 * delxp2 + delyp2 * delyp2)) { /* pt1 closer */
                if ((delx * delx + dely * dely) <
                    (delxp1 * delxp1 + delyp1 * delyp1)) {

```

```

        wall_collision = 1;
        sys_dat->x[i] = pt1x;
        sys_dat->y[i] = pt1y;
        tau_remain -= sqrt(delxp1 * delxp1 + delyp1 * delyp1) /
            sqrt(sys_dat->vx[i] * sys_dat->vx[i] +
                sys_dat->vy[i] * sys_dat->vy[i]);
    }
}
else { /* pt2 closer */
    if ((delx * delx + dely * dely) <
        (delxp2 * delxp2 + delyp2 * delyp2)) {
        wall_collision = 1;
        sys_dat->x[i] = pt2x;
        sys_dat->y[i] = pt2y;
        tau_remain -= sqrt(delxp2 * delxp2 + delyp2 * delyp2) /
            sqrt(sys_dat->vx[i] * sys_dat->vx[i] +
                sys_dat->vy[i] * sys_dat->vy[i]);
    }
};
break;
}; /* end switch pip */

if (wall_collision) {
    /** decide type of interaction with wall */
    /* be sure to track number of collisions and corresponding
       momentum changes so that pressures can be calculated */
    /* particles have been moved up to the point of a wall
       collision. The velocity is reset depending on the type of
       collision */
    if (ran(seed) < sys_dat->alpha) {
        /** Specular */
        /* Specular reflections occur when surface is extremely
           slippery; therefore, no angular momentum is imparted to
           the particle from the wall and visa-versa. */
        xx = sys_dat->x[i] * sys_dat->x[i];
        yy = sys_dat->y[i] * sys_dat->y[i];
        xy = sys_dat->x[i] * sys_dat->y[i];
        vox = ((sys_dat->vx[i] * (yy - xx) - sys_dat->vy[i] * (2 * xy)) /
            sys_dat->wall_rad2[wall]);
        voy = ((sys_dat->vy[i] * (xx - yy) - sys_dat->vx[i] * (2 * xy)) /
            sys_dat->wall_rad2[wall]);
        sys_dat->strikes[wall] += 1;
        sys_dat->delvr[wall] -= 2 * (sys_dat->x[i] * sys_dat->vx[i] +
            sys_dat->y[i] * sys_dat->vy[i]) /
            sys_dat->wall_rad[wall];

        sys_dat->vx[i] = vox;
        sys_dat->vy[i] = voy;
        sys_dat->x0[i] = sys_dat->x[i];
        sys_dat->y0[i] = sys_dat->y[i];
    }
    else {
        /** Thermalized */
        x_on_r = sys_dat->x[i] / sys_dat->wall_rad[wall];
        y_on_r = sys_dat->y[i] / sys_dat->wall_rad[wall];
    }
}

```



```

von = (1 - 2 * wall) * sys_dat->wall_vmp[wall] *
      sqrt(-(log(1 - ran(seed))));
rho = sys_dat->wall_vmp[wall] * sqrt(-log(1 - ran(seed)));
phi = 2 * M_PI * ran(seed);
vot = rho * cos(phi) + sys_dat->wall_rad[wall] *
      sys_dat->wall_angvel[wall];
voz = rho * sin(phi);

vox = x_on_r * von - y_on_r * vot;
voy = y_on_r * von + x_on_r * vot;

delvx = vox - sys_dat->vx[i];
delvy = voy - sys_dat->vy[i];

sys_dat->strikes[wall] += 1;
sys_dat->delvr[wall] += x_on_r * delvx + y_on_r * delvy;
sys_dat->delvtheta[wall] += x_on_r * delvy - y_on_r * delvx;

sys_dat->vx[i] = vox;
sys_dat->vy[i] = voy;
sys_dat->vz[i] = voz;
sys_dat->x0[i] = sys_dat->x[i];
sys_dat->y0[i] = sys_dat->y[i];
}
} /* end if pip */
} /* end if bracket */
} /* end if check */

/** store updated position and velocity data in polar coordinates */
if (wall_collision == 0) {
    sys_dat->x[i] = xnew; sys_dat->y[i] = ynew; sys_dat->z[i] = znew;
    tau_remain = 0;
}
sys_dat->r[i] = sqrt(sys_dat->x[i] * sys_dat->x[i] +
                   sys_dat->y[i] * sys_dat->y[i]);
sys_dat->theta[i] = atan2(sys_dat->y[i], sys_dat->x[i]);
if (sys_dat->theta[i] < 0) sys_dat->theta[i] += TWOPI;
sys_dat->vr[i] = cos(sys_dat->theta[i]) * sys_dat->vx[i] +
               sin(sys_dat->theta[i]) * sys_dat->vy[i];
sys_dat->vtheta[i] = cos(sys_dat->theta[i]) * sys_dat->vy[i] -
                  sin(sys_dat->theta[i]) * sys_dat->vx[i];
}
} /* end while tau_remain */

if ((sys_dat->r[i] > sys_dat->wall_rad[1]) ||
    (sys_dat->r[i] < sys_dat->wall_rad[0])) {
    fprintf(stderr, "Particle %d has left the annular region\n", i);
    fprintf(stderr, "tstep = %ld x = %e y = %e r = %e theta = %e\n",
            tstep, sys_dat->x[i], sys_dat->y[i], sys_dat->r[i],
            sys_dat->theta[i]);

    /* reassign position and velocity */
    not_done = 1;

```



```

for (i=1;i<sys_dat->ncell;i++) {
    sys_dat->index[i] = sys_dat->index[i-1] + sys_dat->cell_n[i-1];
}

/* Calculate particle order */
for (i=0;i<sys_dat->npart;i++) {
    jcell = sys_dat->part_cell[i];
    j = sys_dat->index[jcell] + sys_dat->temp[jcell];
    sys_dat->order[j] = i;
    sys_dat->temp[jcell] += 1;
}

.....
***
*** collide.c
***
***
.....

void collide(struct system_data *sys_dat, long *seed) {
    long i, number, nsel, isel, k, kk, ip1, ip2;
    double select, crm, cr, delvx, delvy, delvz, vcmx, vcmy, vcmz, costh, sinth,
        phi, vrelx, vrelz, vrelz;

    sys_dat->col = 0;

    /** Process collisions over all cells */
    for (i=0;i<sys_dat->ncell;i++) {
        number = sys_dat->cell_n[i];

        /** Collide if we have greater than one particle in cell */
        if (number > 1) {
            /** Find number of collision attempts to be made */
            select = sys_dat->coeff * number * number * sys_dat->vrmax[i] /
                sys_dat->cell_vol[i] + sys_dat->seixtra[i];
            nsel = floor(select);
            sys_dat->seixtra[i] = select - nsel;

            crm = sys_dat->vrmax[i];
            for (isel=0;isel<nsel;isel++) {
                /** Pick two particles for collision */
                k = floor(ran(seed) * number);
                kk = (long) ceil(k + ran(seed) * (number - 1)) % number;
                ip1 = sys_dat->order[k + sys_dat->index[i]];
                ip2 = sys_dat->order[kk + sys_dat->index[i]];

                /** Calculate relative speed */
                delvx = sys_dat->vx[ip1] - sys_dat->vx[ip2];
                delvy = sys_dat->vy[ip1] - sys_dat->vy[ip2];
                delvz = sys_dat->vz[ip1] - sys_dat->vz[ip2];
                cr = sqrt(delvx * delvx + delvy * delvy + delvz * delvz);
                if (cr > crm) crm = cr;

                /** Accept / Reject collision */
                if ((cr / sys_dat->vrmax[i]) < ran(seed)) {
                    sys_dat->col += 1;
                }
            }
        }
    }
}

```

```

vcmx = 0.5 * (sys_dat->vx[ip1] + sys_dat->vx[ip2]);
vcmy = 0.5 * (sys_dat->vy[ip1] + sys_dat->vy[ip2]);
vcnz = 0.5 * (sys_dat->vz[ip1] + sys_dat->vz[ip2]);

costh = 1 - 2 * ran(seed);
sinth = sqrt(1 - costh * costh);
phi = 2 * M_PI * ran(seed);
vrelx = cr * costh;
vrelx = cr * sinth * cos(phi);
vrelz = cr * sinth * sin(phi);

sys_dat->vx[ip1] = vcmx + 0.5 * vrelx;
sys_dat->vy[ip1] = vcmy + 0.5 * vrelx;
sys_dat->vz[ip1] = vcnz + 0.5 * vrelz;
sys_dat->vx[ip2] = vcmx - 0.5 * vrelx;
sys_dat->vy[ip2] = vcmy - 0.5 * vrelx;
sys_dat->vz[ip2] = vcnz - 0.5 * vrelz;

sys_dat->x0[ip1] = sys_dat->x[ip1];
sys_dat->y0[ip1] = sys_dat->y[ip1];
sys_dat->x0[ip2] = sys_dat->x[ip2];
sys_dat->y0[ip2] = sys_dat->y[ip2];
} /* if accept/reject */
} /* for isel over number of collision attempts */

/** Update maximum relative speed */
sys_dat->vrmax[i] = crm;

} /* if number */
} /* for i over number of cells */
} /* end collide */

.....
***                                     ***
*** sample.c                           ***
***                                     ***
.....

void sample(struct system_data *sys_dat) {

    long i;

    for (i=0;i<sys_dat->npart;i++) {
        sys_dat->accum_n[sys_dat->part_cell[i]] += 1;
        sys_dat->accum_vx[sys_dat->part_cell[i]] += sys_dat->vx[i];
        sys_dat->accum_vy[sys_dat->part_cell[i]] += sys_dat->vy[i];
        sys_dat->accum_vz[sys_dat->part_cell[i]] += sys_dat->vz[i];
        sys_dat->accum_vr[sys_dat->part_cell[i]] += sys_dat->vr[i];
        sys_dat->accum_vtheta[sys_dat->part_cell[i]] += sys_dat->vtheta[i];
        sys_dat->accum_v2[sys_dat->part_cell[i]] += sys_dat->vx[i] * sys_dat->vx[i]
                                                    + sys_dat->vy[i] * sys_dat->vy[i]
                                                    + sys_dat->vz[i] * sys_dat->vz[i];
    }
}

```

```

/*****
***
*** random.c
***
*****/
#define IM1      (2147483563)
#define IM2      (2147483399)
#define AM       (1.0 / IM1)
#define IMM1     (IM1 - 1)
#define IA1      (40014)
#define IA2      (40692)
#define IQ1      (53668)
#define IQ2      (52774)
#define IR1      (12211)
#define IR2      (3791)
#define NTAB     (32)
#define NDIV     (1 + IMM1 / NTAB)
#define EPS      1.2e-7
#define RNMIX    (1.0 - EPS)
#define BOLTZ    (1.380658e-23) /* J/K */
#define M_E      (2.7182818284590452354)
#define M_PI     (3.14159265358979323846)
#define M_2_SQRTPI (1.12837916709551257390)
#define TWOPI    (6.28318530717958647693)

double ran(long *idum) {
    /* Numerical Recipe's ran2 routine */

    int j;
    long k;
    static long idum2 = 123456789;
    static long iy = 0;
    static long iv[NTAB];
    double temp;

    if (*idum <= 0) {
        if (--(*idum) < 1) *idum = 1;
        else *idum = -(*idum);
        idum2 = (*idum);
        for (j=NTAB+7;j>=0;j--) {
            k = (*idum) / IQ1;
            *idum = IA1 * (*idum - k * IQ1) - k * IR1;
            if (*idum < 0) *idum += IM1;
            if (j < NTAB) iv[j] = *idum;
        }
        iy = iv[0];
    }

    k = (*idum) / IQ1;
    *idum = IA1 * (*idum - k * IQ1) - k * IR1;
    if (*idum < 0) *idum += IM1;
    k = idum2 / IQ2;
    idum2 = IA2 * (idum2 - k * IQ2) - k * IR2;
    if (idum2 < 0) idum2 += IM2;
    j = iy / NDIV;

```

```
iy = iv[j] - idum2;  
iv[j] = *idum;  
if (iy < 1) iy += IMM1;  
if ((temp = AM * iy) < RNMX) return RNMX;  
else return temp;  
;
```

```

% *****
% ***                                     ***
% *** compile_sta.m                     ***
% ***                                     ***
% *****
%
% Compiles the state files for a given basename.
%
% Program assumes that the runs start with the dataset pointed to by the
% basename and it will load until no more run data for that basename is
% available.
%
clear;
%
boltz = 1.380658e-23; % Boltzman's constant (J/K)
%
basename = input('Enter run basename: ', 's');
start_basename = basename;
%
% Enter config file stuff
unixcmd = [setstr(39) 'cat ' basename '.cfg' setstr(39)];
eval(['[status, result] = unix(' unixcmd ');']);
colon = findstr(result, ':');
newline = findstr(result, setstr(10));
for i = 1:13
    if (i == 1)
        eval(['result(1:(colon(i)-1)) '=str2num(result((colon(i)+1):(newline(i)-
1));');']);
    else
        eval(['result((newline(i-1)+1):(colon(i)-1))
'=str2num(result((colon(i)+1):(newline(i)-1));');']);
    end
end
%
fidout = fopen([basename '.starpt'], 'w');
n = 0;
no_data = 0;
not_done = 1;
while not_done
    stafile = [basename '.sta'];
    fid = fopen(stafile);
    if fid ~= 0
        fclose(fid);
        n = n + 1;
        % Load state data
        eval(['load -ascii ' stafile]);
        eval(['vel = ' basename '(:,4:6);']);
        eval(['pos = ' basename '(:,1:3);']);
        eval(['clear ' basename]);
        %
        vel = vel'; % Makes velocity and position matrices into
        pos = pos'; % column vectors
        r=sqrt(dot(pos, pos));
        theta=atan2(pos(2,:), pos(1,:));
        % costerm=pos(1,:) ./ r;

```

```

% sinterm=pos(2,:) ./ r;
%
veldot = dot(vel, vel);
veldotsqrt = sqrt(veldot);
% vtheta=costerm .* vel(2,:) - sinterm .* vel(1,:);
%
avg_T(n) = (6.633526e-26 / (3 * boltz)) * mean(veldot);
sdv_T = (6.633526e-26 / (3 * boltz)) * std(veldot);
avg_lin_mom(n) = 6.633526e-26 * mean(veldotsqrt);
sdv_lin_mom = 6.633526e-26 * std(veldotsqrt);
avg_ang_mom(n) = 6.633526e-26 * mean(pos(1,:) .* vel(2,:) - ...
    pos(2,:) .* vel(1,:));
sdv_ang_mom = 6.633526e-26 * std(pos(1,:) .* vel(2,:) - ...
    pos(2,:) .* vel(1,:));
% avg_ang_mom(n) = 6.633526e-26 * mean(r .* vtheta);
% sdv_ang_mom = 6.633526e-26 * std(r .* vtheta);
%
fprintf(fidout, ...
    '%s\t%17.14g\t%17.14g\t%17.14g\t%17.14g\t%17.14g\n', ...
    basename, avg_T(n), sdv_T, avg_lin_mom(n), sdv_lin_mom, ...
    avg_ang_mom(n), sdv_ang_mom);
%
last_basename = basename;
basename = next_basename(basename);
else
    if n == 0;
        no_data = 1;
    end
    not_done = 0;
end
end
end
%
if no_data
    fprintf(fidout, 'No state files for basename %s', basename);
end
%
fprintf(fidout, 'Interrun Statistics - Ignoring the First Run\n');
fprintf(fidout, 'mean:\t%17.14g\t%17.14g\t%17.14g\n', mean(avg_T(2:n)), ...
    mean(avg_lin_mom(2:n)), mean(avg_ang_mom(2:n)));
fprintf(fidout, 'sdev:\t%17.14g\t%17.14g\t%17.14g\n', std(avg_T(2:n)), ...
    std(avg_lin_mom(2:n)), std(avg_ang_mom(2:n)));
%
fclose(fidout);

% *****
% ***
% *** next_basename.m ***
% ***
% *****
%
% Increments filename to the next filename in a sequence
%
% This function will return the next basename in the sequence and the
% status variable will contain either a 0 for a successful incrementation,
% a -1 on carry over, or a -2 on invalid input basename.

```



```

%
function [new_basename, status] = next_basename(basename);
%
status = 0;
%
length = size(basename, 2);
%
if length ~= 2
    new_basename = 'invalid basename';
    status = -2;
    return
end
%
num = str2num(basename((length-1):length));
root = basename(1:(length-2));
%
if num == []
    new_basename = 'invalid basename';
    status = -2;
    return
end
%
num = num + 1;
char = num2str(num);
if (size(char, 2) == 1)
    char = ['0' char];
end
if (size(char, 2) == 3)
    char = char(2:3);
    status = -1;
end
%
new_basename = [root char];
return

%*****
%***                                     ***
%***  pcpp.m                           ***
%***                                     ***
%*****
%
% This program post processes data generated within the polar couette
% simulation.
%
clear;
if ishold
    hold;
end
%
m = 6.633526e-26;      % mass (kg)
T = 300;               % temp (K)
boltz = 1.380658e-23; % Boltzman's constant (J/K)
atomic_diam = 3.66e-10;
part_density = 2.686763e25;
mfp = 1.0 / (sqrt(2.0) * part_density * pi * atomic_diam * atomic_diam);

```

```

avg_speed = (2 * sqrt(2) / pi) * sqrt(boltz * T / m);
%
printdev = ['-dpvc '];
cst_mask = [1 1 1 1 1 1 1 1];
rast_mask = [1 1 1 1 1 1 1 1];
basename = input('Enter run basename: ', 's');
%
% Enter config file stuff
unixcmd = [setstr(39) 'cat ' basename '.cfg' setstr(39)];
eval(['[status, result] = unix(' unixcmd ');']);
colon = findstr(result, ':');
newline = findstr(result, setstr(10));
for i = 1:13
    if (i == 1)
        eval([result(1:(colon(i)-1)) '=str2num(result((colon(i)+1):(newline(i)-
1)))));']);
    else
        eval([result((newline(i-1)+1):(colon(i)-1))
'=str2num(result((colon(i)-1):(newline(i)-1)))));']);
    end
end
deltheta = 6.28318530717958647693 / theta_slices;
delr = (outer_wall_rad - inner_wall_rad) / radial_shells;
%
% Determine what information to display
fprintf('Choose display to be generated\n\n\n');
fprintf('\tAvg Density:\t1\n');
fprintf('\tAvg Vx:\t2\n');
fprintf('\tAvg Vy:\t4\n');
fprintf('\tAvg Vz:\t8\n');
fprintf('\tAvg Vr:\t16\n');
fprintf('\tAvg Vtheta:\t32\n');
fprintf('\tAvg V^2:\t64\n\n');
fprintf('\tVtheta vs. r:\t128\n\n');
fprintf('\tVel Histogram:\t256\n\n\n');
to_do_num = input('Enter numerical total of all graphs to be displayed: ');
to_do_bits = bits(to_do_num);
for i=(max(size(to_do_bits))+1):12
    to_do_bits(i) = 0;
end
%
if any(cst_mask & to_do_bits(1:max(size(cst_mask))))
    %
    % Enter cell statistics data
    cstfile = [basename '.cst'];
    % response = input(['Enter cell stats file (default ' cstfile '): ', 's']);
    % if ~(response == [])
    %     cstfile = response;
    % end
    eval(['load ' cstfile]);
    eval(['cstarray = ' basename ';']);
    %
    if any(rast_mask & to_do_bits(1:max(size(rast_mask))))
        % Generate rasterized images
        N = input('Enter the number of pixels on a side: ');
    end
end

```

```

cmap = gray(256);
colormap(cmap);
cell_area = zeros(size(cstarray,1));
for i=1:N
    for j=1:N
        x = (2 * outer_wall_rad / (N - 1)) * (j - 1) - outer_wall_rad;
        y = (2 * outer_wall_rad / (N - 1)) * (1 - i) - outer_wall_rad;
        r = sqrt(x * x + y * y);
        theta = atan2(y, x);
        if (theta < 0)
            theta = theta + 6.28318530717958647693;
        end
        %
        if ((r > outer_wall_rad) & (r < inner_wall_rad))
            accum_n(i,j) = 0;
            accum_vx(i,j) = 0;
            accum_vy(i,j) = 0;
            accum_vz(i,j) = 0;
            accum_vr(i,j) = 0;
            accum_vtheta(i,j) = 0;
            accum_v2(i,j) = 0;
            ave_density(i,j) = 0;
            ave_vx(i,j) = 0;
            ave_vy(i,j) = 0;
            ave_vz(i,j) = 0;
            ave_vr(i,j) = 0;
            ave_vtheta(i,j) = 0;
            ave_v2(i,j) = 0;
            mask(i,j) = 1;
        else
            cell = floor((r - inner_wall_rad) / delr) * theta_slices + ...
                floor(theta / deltheta) - 1;
            if (cell_area(cell) == 0)
                r0 = (floor((r - inner_wall_rad) / delr)) * delr + inner_wall_rad;
                cell_area(cell) = (deltheta / 2) * delr * (2 * r0 + delr);
            end
            accum_n(i,j) = cstarray(cell,1);
            accum_vx(i,j) = cstarray(cell,2);
            accum_vy(i,j) = cstarray(cell,3);
            accum_vz(i,j) = cstarray(cell,4);
            accum_vr(i,j) = cstarray(cell,5);
            accum_vtheta(i,j) = cstarray(cell,6);
            accum_v2(i,j) = cstarray(cell,7);
            ave_density(i,j) = accum_n(i,j) / cell_area(cell);
            if (accum_n(i,j) == 0)
                ave_vx(i,j) = 0;
                ave_vy(i,j) = 0;
                ave_vz(i,j) = 0;
                ave_vr(i,j) = 0;
                ave_vtheta(i,j) = 0;
                ave_v2(i,j) = 0;
            else
                ave_vx(i,j) = accum_vx(i,j) / accum_n(i,j);
                ave_vy(i,j) = accum_vy(i,j) / accum_n(i,j);
                ave_vz(i,j) = accum_vz(i,j) / accum_n(i,j);
            end
        end
    end
end

```

```

        ave_vr(i,j) = accum_vr(i,j) / accum_n(i,j);
        ave_vtheta(i,j) = accum_vtheta(i,j) / accum_n(i,j);
        ave_v2(i,j) = accum_v2(i,j) / accum_n(i,j);
    end
end
end
end
%
if to_do_bits(1)
    imagesc(ave_density),axis('equal'),colorbar;
    title(['Average Density ' '(' basename ')']);
    s = ['print ' printdev basename];
    eval([s '_ave_dens.ps']);
    imagesc(accum_n),axis('equal'),colorbar;
    title(['Time avg number of Parliamentary Particles ' '(' basename ')']);
    eval([s '_numofpart.ps']);
end
%
if to_do_bits(2)
    imagesc(ave_vx),axis('equal'),colorbar;
    title(['Average x - Velocity ' '(' basename ')']);
    s = ['print ' printdev basename];
    eval([s '_ave_vx.ps']);
end
%
if to_do_bits(3)
    imagesc(ave_vy),axis('equal'),colorbar;
    title(['Average y - Velocity ' '(' basename ')']);
    s = ['print ' printdev basename];
    eval([s '_ave_vy.ps']);
end
%
if to_do_bits(4)
    imagesc(ave_vz),axis('equal'),colorbar;
    title(['Average z - Velocity ' '(' basename ')']);
    s = ['print ' printdev basename];
    eval([s '_ave_vz.ps']);
end
%
if to_do_bits(5)
    imagesc(ave_vr),axis('equal'),colorbar;
    title(['Average r - Velocity ' '(' basename ')']);
    s = ['print ' printdev basename];
    eval([s '_ave_vr.ps']);
end
%
if to_do_bits(6)
    imagesc(ave_vtheta),axis('equal'),colorbar;
    title(['Average theta - Velocity ' '(' basename ')']);
    s = ['print ' printdev basename];
    eval([s '_ave_vtheta.ps']);
end
%
if to_do_bits(7)
    imagesc(ave_v2),axis('equal'),colorbar;

```

```

        title(['Average Squared Velocity ' '(' basename ')']);
        s = ['print ' printdev basename];
        eval([s '_ave_v2.ps']);
    end
end
%
if to_do_bits(8)
    %
    vtheta_avg = zeros(1,radial_shells);
    vtheta2_avg = zeros(1,radial_shells);
    radius_ave = zeros(1,radial_shells);
    cell_part_count = zeros(1,radial_shells);
    track_density = zeros(1,radial_shells);
    track_part_count = zeros(1,radial_shells);
    %
    radius_th = 0:499;
    radius_th = inner_wall_rad + ...
        radius_th * (outer_wall_rad - inner_wall_rad) / 499;
    vel_th = vprof_th(radius_th, inner_wall_angvel, outer_wall_angvel, ...
        inner_wall_rad, outer_wall_rad, mfp, alpha);
    %
    for i = 1:radial_shells
        for j = 1:theta_slices
            cell_ang_avg = cstarray(((i-1)*theta_slices+j),6) / ...
                cstarray(((i-1)*theta_slices+j),1);
            cell_v2_avg = cstarray(((i-1)*theta_slices+j),7) / ...
                cstarray(((i-1)*theta_slices+j),1);
            vtheta_avg(i) = vtheta_avg(i) + cell_ang_avg;
            vtheta2_avg(i) = vtheta2_avg(i) + cell_v2_avg;
            track_part_count(i) = track_part_count(i) ...
                + cstarray(((i-1)*theta_slices+j),1);
        end
        cell_part_count = track_part_count / theta_slices;
        track_density(i) = track_part_count(i) / ...
            (pi * delr * (2 * inner_wall_rad + ...
                delr * (2 * i - 1)));
        radius_ave(i) = inner_wall_rad + (i - 0.5) * delr;
    end
    vtheta_avg = vtheta_avg / theta_slices;
    vtheta2_avg = vtheta2_avg / theta_slices;
    plot(radius_ave, vtheta_avg, 'wo');
    if ~(max(size(vel_th)) == 1) & vel_th(1) == 0
        hold;
        plot(radius_th, vel_th, 'w--');
        hold
    end
    title(['Ang Comp of Vel (per Track Avg) vs. Radius (' basename ')']);
    xlabel('Radius (m)'); ylabel('Angular Component of Velocity (m/s)');
    %
    glabel(.55,.9,['MFP=' num2str(mfp) '(m)'],'lin');
    %
    glabel(.55,.85,['Avg_spd=' num2str(avg_speed) '(m/s)'],'lin');
    %
    glabel(.55,.8, ...
        ['V_wall0=' num2str(inner_wall_rad*inner_wall_angvel) '(m/s)'],'lin');
    %
    glabel(.55,.75, ...
        ['V_wall1=' num2str(outer_wall_rad*outer_wall_angvel) '(m/s)'],'lin');
    %
    glabel(.55,.7,['alpha=' num2str(alpha)],'lin');

```

```

% glabel(.55,.65,['delt=' num2str(delt)],'lin');
s = ['print ' printdev basename];
eval([s '_vtheta_avg.ps']);
keyboard
%
plot(radius_ave, vtheta2_avg, 'wo');
title(['Avg squared velocity vs Radius (' basename ')']);
xlabel('Radius (m)'); ylabel('Squared Velocity (m^2/s^2)');
eval([s '_v2_avg.ps']);
keyboard
%
plot(radius_ave, track_density, 'wo');
title(['Track Particle Density vs Radius (' basename ')']);
xlabel('Radius (m)'); ylabel('Track Avg Particle Density');
eval([s '_track_density.ps']);
keyboard
%
plot(radius_ave, cell_part_count, 'wo');
title(['Avg Cell Part Count per Track vs Radius (' basename ')']);
xlabel('Radius (m)'); ylabel('Avg Cell Particle Count');
eval([s '_cell_part_count.ps']);
end
%
end
%
if to_do_bits(9)
%
no_bins = input('Enter number of bins: ');
eval(['load -ascii ' basename '.sta;']);
eval(['vel = ' basename '(:,4:6);']);
eval(['clear ' basename]);
%
v = sqrt(vel(:,1) .* vel(:,1) + vel(:,2) .* vel(:,2) + vel(:,3) .* vel(:,3));
[N, X] = hist(v, no_bins);
bar(X, N);
title(['Histogram of Particle Speeds (' basename ')']);
xlabel('Speed (m/s)');
ylabel('Count');
hold;
%
X2 = X .* X;
energy = .5 * m * sum(v .* v);
T = 2*(energy/npart)/(3*boltz);
coeff1 = m / (2 * boltz * T);
coeff2 = (4 * pi * size(vel, 1)) * ((coeff1 / pi) ^ (3 / 2));
mb = coeff2 * (exp(-coeff1 * X2) .* X2) * (X(2) - X(1));
plot(X, mb, '--b');
glabel(.6, .7, ['-T=' num2str(T)], 'lin');
hold;
s = ['print ' printdev basename];
eval([s '_vhist.ps']);
end

%*****
%***
***

```

```

%*** bits.m                                     ***
%***                                     ***
%*****
%
% This function takes a number, truncates it to an integer value,
% then converts that to an array that contains ones representing the binary
% value of the number.
%
function bitpattern = bits(number);
number = floor(abs(number));
count = 1;
while ~(number == 0)
    if rem(number, 2)
        bitpattern(count) = 1;
    end
    number = floor(abs(number / 2));
    count = count + 1;
end
return

%*****
%***                                     ***
%*** vprof_th.m                             ***
%***                                     ***
%*****
%
% This function generates the velocity profile between rotating cylinders
% as prediction by Navier-Stokes using the Panzer, Liu, and Einzel extension
% to the slip boundary condition.
%
function vel = vprof_th(radius, inner_wall_angvel, outer_wall_angvel, ...
                        inner_wall_rad, outer_wall_rad, mfp, alpha)
zeta0 = 1.126 * mfp * (2 - alpha) / alpha;
% zeta0 = 1.126 * mfp;
A = (1 - 2 * zeta0 / outer_wall_rad) / (outer_wall_rad * outer_wall_rad);
B = (1 + 2 * zeta0 / inner_wall_rad) / (inner_wall_rad * inner_wall_rad);
a = (A * inner_wall_angvel - B * outer_wall_angvel) / (A - B);
b = (inner_wall_angvel - outer_wall_angvel) / (B - A);
vel = a * radius + b ./ radius;
return

%*****
%***                                     ***
%*** vplot.m                               ***
%***                                     ***
%*****
%
% Plots PLE velocity profiles for a given set of cylinder radii and
% for a range of alphas.
%
m = 6.633526e-26;      % mass (kg)
T = 300;              % temp (K)
boltz = 1.380658e-23; % Boltzman's constant (J/K)
atomic_diam = 3.66e-10;
part_density = 2.686763e25;

```

```

mfp = 1.0 / (sqrt(2.0) * part_density * pi * atomic_diam * atomic_diam);
%
inner_wall_rad = input('Inner radius: ');
outer_wall_rad = input('Outer radius: ');
inner_wall_angvel = input('Inner wall angvel: ');
outer_wall_angvel = input('Outer wall angvel: ');
alphas = input('alphas: ');
radius_th = 0:499;
radius_th = inner_wall_rad + ...
            radius_th * (outer_wall_rad - inner_wall_rad) / 499;
num = length(alphas);
%
hold off;
for i=1:num
    vel_th = vprof_th(radius_th, inner_wall_angvel, outer_wall_angvel, ...
                    inner_wall_rad, outer_wall_rad, mfp, alphas(i));
    plot(radius_th, vel_th, 'w--');
    text(radius_th(length(radius_th)), vel_th(length(vel_th)), ...
        num2str(alphas(i)));
%    [gtex('a') '=' num2str(alphas(i))];
    if (~ishold)
        title('PLE Velocity Profile for Various Alphas');
        xlabel('radius (m)');
        ylabel('theta Component of Velocity (m/s)');
        hold;
    end
end
hold off;

%*****
%***                                     ***
%*** compile_test.m                     ***
%***                                     ***
%*****
%
% Compiles the cell statistics files for a given basename.
%
% Program assumes that the runs start with the dataset pointed to by the
% basename and it will load until no more run data for that basename is
% available.
%
clear;
%
m = 6.633526e-26;      % mass (kg)
T = 300;               % temp (K)
boltz = 1.380658e-23; % Boltzman's constant (J/K)
atomic_diam = 3.66e-10;
part_density = 2.686763e25;
mfp = 1.0 / (sqrt(2.0) * part_density * pi * atomic_diam * atomic_diam);
avg_speed = (2 * sqrt(2) / pi) * sqrt(boltz * T / m);
%
basename = input('Enter run basename: ', 's');
start_basename = basename;
inc_class = 'n';
inc_class = input('Include no-slip Navier-Stokes Solution? (n/y)');

```



```

%
% Enter config file stuff
unixcmd = [setstr(39) 'cat ' basename '.cfg' setstr(39)];
eval(['[status, result] = unix(' unixcmd ');']);
colon = findstr(result, ':');
newline = findstr(result, setstr(10));
for i = 1:13
    if (i == 1)
        eval(['result(1:(colon(i)-1)) '=str2num(result((colon(i)+1):(newline(i)-
1))))';]);
    else
        eval(['result((newline(i-1)+1):(colon(i)-1))
'=str2num(result((colon(i)+1):(newline(i)-1))))';]);
    end
end
deltheta = 6.28318530717958647693 / theta_slices;
delr = (outer_wall_rad - inner_wall_rad) / radial_shells;
radius_th = 0:499;
radius_th = inner_wall_rad + ...
    radius_th * (outer_wall_rad - inner_wall_rad) / 499;
vel_th = vprof_th(radius_th, inner_wall_angvel, outer_wall_angvel, ...
    inner_wall_rad, outer_wall_rad, mfp, alpha);
vel_th_noslip = vprof_th_noslip(radius_th, inner_wall_angvel, ...
    outer_wall_angvel, inner_wall_rad, outer_wall_rad, mfp, alpha);
%
cstarray = zeros(theta_slices * radial_shells, 7);
cst_trackavg = zeros(1, radial_shells);
avg_trackavg = zeros(1, radial_shells);
sdev_trackavg = zeros(1, radial_shells);
cstv2_trackavg = zeros(1, radial_shells);
v2_trackavg = zeros(1, radial_shells);
v2sdev_trackavg = zeros(1, radial_shells);
num_of_parts_trackavg = zeros(1, radial_shells);
num_trackavg = zeros(1, radial_shells);
num_sdev_trackavg = zeros(1, radial_shells);
dens_of_parts_trackavg = zeros(1, radial_shells);
dens_trackavg = zeros(1, radial_shells);
dens_sdev_trackavg = zeros(1, radial_shells);
not_done = 1;
no_data = 0;
n = 0;
while not_done
    cstfile = [basename '.cst'];
    fid = fopen(cstfile);
    if fid ~= 0
        fclose(fid);
        eval(['load ' cstfile]);
        eval(['cstarray = ' basename ');']);
        eval(['clear ' basename]);
    %
    for i = 1:radial_shells
        cst_trackavg(n+1,i)=0;
        cstv2_trackavg(n+1,i)=0;
        num_of_parts_trackavg(n+1,i)=0;
        for j = 1:theta_slices

```

```

        cst_trackavg(n+1,i) = cst_trackavg(n+1,i) + ...
        cstararray(((i-1)*theta_slices+j),6) / cstararray(((i-1)*theta_slices+j),1);
        cstv2_trackavg(n+1,i) = cstv2_trackavg(n+1,i) + ...
        cstararray(((i-1)*theta_slices+j),7) / cstararray(((i-1)*theta_slices+j),1);
        num_of_parts_trackavg(n+1,i) = num_of_parts_trackavg(n+1,i) ...
            + cstararray(((i-1)*theta_slices+j),1);
    end
    cst_trackavg(n+1,i) = cst_trackavg(n+1,i) / theta_slices;
    cstv2_trackavg(n+1,i) = cstv2_trackavg(n+1,i) / theta_slices;
    dens_of_parts_trackavg(n+1,i) = num_of_parts_trackavg(n+1,i) / ...
        (pi * delr * (2 * inner_wall_rad + ...
            delr * (2 * i - 1)));
    num_of_parts_trackavg(n+1,i) = num_of_parts_trackavg(n+1,i) / ...
        theta_slices;
end
%
avg_trackavg = avg_trackavg + cst_trackavg(n+1,:);
v2_trackavg = v2_trackavg + cstv2_trackavg(n+1,:);
num_trackavg = num_trackavg + num_of_parts_trackavg(n+1,:);
dens_trackavg = dens_trackavg + dens_of_parts_trackavg(n+1,:);
n = n + 1;
%
last_basename = basename;
basename = next_basename(basename);
else
    if n == 0;
        no_data = 1;
    end
    not_done = 0;
end
end
%
% NOTE: I DEFINE variance as (1/N)*SUM((xi - xavg)^2,N) and calculate
%       sdev as sqrt(variance).
%
if no_data
    disp(['No runs exist with basename: ' start_basename]);
else
    %
    avg_trackavg = avg_trackavg / n;
    v2_trackavg = v2_trackavg / n;
    num_trackavg = num_trackavg / n;
    dens_trackavg = dens_trackavg / n;
    for i=1:n
        sdev_trackavg = sdev_trackavg + (cst_trackavg(i,:) - avg_trackavg).^2;
        v2sdev_trackavg = v2sdev_trackavg + ...
            (cstv2_trackavg(i,:) - v2_trackavg).^2;
        num_sdev_trackavg = num_sdev_trackavg + ...
            (num_of_parts_trackavg(i,:) - num_trackavg).^2;
        dens_sdev_trackavg = dens_sdev_trackavg + ...
            (dens_of_parts_trackavg(i,:) - dens_trackavg).^2;
    end
    sdev_trackavg = sqrt(sdev_trackavg / n);
    v2sdev_trackavg = sqrt(v2sdev_trackavg / n);
    num_sdev_trackavg = sqrt(num_sdev_trackavg / n);

```

```

dens_sdev_trackavg = sqrt(dens_sdev_trackavg / n);

radius_ave = inner_wall_rad + ([1:radial_shells] - 0.5) * delr;

%
errorbar(radius_ave, avg_trackavg, sdev_trackavg, sdev_trackavg, 'ow');
% title(['Interrun (per cell) Track Avg of V(phi) vs. Radius (' ...
% plot(radius_ave, avg_trackavg, 'ow');
% title(['Interrun Track Avg of Vtheta vs. Radius (' ...
% start_basename '-' last_basename ''));
% xlabel('Radius (m)'); ylabel('Angular Component of Velocity (m/s)');
% legend('--.', 'No-slip Navier-Stokes', '--', 'Navier-Stokes w/PLE Extension', ...
% 'ow', 'DSMC');
hold
plot(radius_th, vel_th, 'w--');
if (inc_class == 'y')
    plot(radius_th, vel_th_noslip, 'w-.');
end
hold
lengs = size(start_basename, 2);
lengl = size(last_basename, 2);
start_to_last_basenum = [start_basename((lengs-1):lengs) 'to' ...
    last_basename((lengl-1):lengl)];
eval(['print -dps vtheta_comp_' start_to_last_basenum '.ps']);
%
% Make output file containing calculated slip lengths
fid = fopen(['compiled_report_' start_to_last_basenum], 'w');
slope = (avg_trackavg(1,2) - avg_trackavg(1,1)) / ...
    (radius_ave(1,2) - radius_ave(1,1));
inter = avg_trackavg(1,1) - slope * radius_ave(1,1);
rv0 = (vel_th_noslip(1,1) - inter) / slope;
pcslip_in = inner_wall_rad - rv0;
fprintf(fid, 'Slip lengths for Polar-Couette Simulation:\n\n');
fprintf(fid, '\tInner = %8.4g\n', pcslip_in);
fclose(fid);
%
keyboard
%
errorbar(radius_ave, v2_trackavg, v2sdev_trackavg, v2sdev_trackavg, 'ow');
% title(['Interrun Track Avg of V^2 vs. Radius (' start_basename '-' ...
% last_basename ''));
% xlabel('Radius (m)'); ylabel('Track Average of V^2 (m^2/s^2)');
% eval(['print -dps v2_comp_' start_to_last_basenum '.ps']);
% keyboard
%
errorbar(radius_ave, num_trackavg, num_sdev_trackavg, num_sdev_trackavg, 'ow');
% title(['Interrun (per cell) Track Particle Count vs Radius (' ...
% start_basename '-' last_basename ''));
% xlabel('Radius (m)'); ylabel('Per Cell avg of Particle Count');
%
% keyboard
%
errorbar(radius_ave, dens_trackavg, dens_sdev_trackavg, dens_sdev_trackavg, 'ow');
% title(['Interrun Track Particle Density vs Radius (' ...
% start_basename '-' last_basename ''));

```

```

    xlabel('Radius (m)'); ylabel('Per Track Particle Density');
    %
end

%*****
%***                                     ***
%***  vprof_th_noslip.m                 ***
%***                                     ***
%*****
%
% This generates the velocity profile for the Navier-Stokes solution for the
% rotating cylinders with the no slip boundary condition.
%
function vel = vprof_th_noslip(radius, inner_wall_angvel, ...
                               outer_wall_angvel, inner_wall_rad, ...
                               outer_wall_rad, mfp, alpha)
A = 1 / (outer_wall_rad * outer_wall_rad);
B = 1 / (inner_wall_rad * inner_wall_rad);
a = (A * inner_wall_angvel - B * outer_wall_angvel) / (A - B);
b = (inner_wall_angvel - outer_wall_angvel) / (B - A);
vel = a * radius - b ./ radius;
return

```