1995

A hybrid Direct Simulation Monte Carlo and Molecular Dynamics algorithm

David Timothy Lee
San Jose State University

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

Recommended Citation

Lee, David Timothy, "A hybrid Direct Simulation Monte Carlo and Molecular Dynamics algorithm" (1995). *Master's Theses*. 1007.

DOI: <https://doi.org/10.31979/etd.t3ne-mzn3>

https://scholarworks.sjsu.edu/etd_theses/1007

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 HYBRID
DIRECT SIMULATION MONTE CARLO
AND
MOLECULAR DYNAMICS ALGORITHM**

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
David Timothy Lee
May 1995

UMI Number: 1374598

UMI Microform 1374598

Copyright 1995, by UMI Company. All rights reserved.

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

UMI

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

©1995

David T. Lee

ALL RIGHTS RESERVED

APPROVED FOR THE DEPARTMENT
OF PHYSICS

Dr. Alejandro Garcia

Dr. Patrick Hamill

Dr. Berni Alder

APPROVED FOR THE UNIVERSITY

ABSTRACT

A Hybrid Direct Simulation Monte Carlo and Molecular Dynamics Algorithm

by David Timothy Lee

We have developed a hybrid (DSMC-MD) algorithm that takes advantage of the efficiency of Direct Simulation Monte Carlo (DSMC) simulations in dilute regions while utilizing the accuracy of Molecular Dynamics (MD) simulations in dense regions. We tested the algorithm by comparing the results of DSMC, MD, and DSMC-MD simulations of five systems: a system with specular walls, a system with walls held at a constant temperature, a system with a thermal gradient, a Couette flow, and a Poiseuille flow.

For each of these systems, the DSMC-MD scheme produced the same temperature, density, and velocity gradients as both the DSMC and the MD algorithms. In addition, the hybrid scheme ran approximately four times faster than the pure MD algorithm. The DSMC-MD scheme is both consistent with DSMC simulations and faster than a pure MD scheme. Hence, the DSMC-MD scheme can be used to simulate systems with both dilute and dense regions.

Contents

1	INTRODUCTION	1
2	Properties of the Fluid Systems	4
2.1	Fluid System Descriptions	4
2.2	Boundary Interactions	7
3	Programs	10
3.1	Boundary Condition Implementations	11
3.2	Direct Simulation Monte Carlo Program	14
3.3	Molecular Dynamics Program	17
3.4	Hybrid DSMC-MD Program	19
3.5	Statistics Sampling	22
4	Simulation Conditions	24
5	Results and Discussion	27
5.1	Isolated System	27
5.2	Isothermal System	30
5.3	System with Thermal Gradient	31
5.4	Couette Flow	33
5.5	Poiseuille Flow	36

6	Conclusions	39
A	Program Listings	42
A.1	Makefile	43
A.2	globals.h	45
A.3	definitions.h	45
A.4	prototypes.h	46
A.5	calcforces.c	47
A.6	collide.c	50
A.7	dsmc.c	52
A.8	dual.c	56
A.9	files.c	60
A.10	finishmove.c	61
A.11	initglobals.c	62
A.12	insertcrossingparticles.c	64
A.13	mds.c	66
A.14	mover.c	70
A.15	sampler.c	73
A.16	sorter.c	75
A.17	startmove.c	77
A.18	storeglobals.c	80
A.19	urandom.c	84
A.20	utilities.c	86
B	Simulation Plots	89
B.1	Plots from DSMC Simulations	89
B.2	Plots from MD Simulations	95
B.3	Plots from DSMC-MD Simulations	101

List of Tables

4.1	Simulation Parameters for all of the simulations. Wall temperatures are specified as left/right if the walls are at different temperatures. . .	26
5.1	Average temperature and velocities from the Isolated system simulations.	30
5.2	Average Temperature and velocities from the Isothermal system simulations.	31

List of Figures

3.1	A DSMC particle is moved and crosses a wall. The time and coordinates of the wall crossing are calculated. A new velocity is assigned from a biased Boltzmann distribution and the particle is moved off the wall.	12
3.2	A MD particle is moved and crosses a wall. The coordinates of the wall crossing are calculated. A new velocity is assigned from a biased Boltzmann distribution and the particle is left on the wall.	12
3.3	Periodic Boundary Conditions. The central cell is the simulation cell; the surrounding cells are its images. Once the simulation particle (disk) leaves the central cell, its image (circle) becomes the simulation particle.	14
3.4	The left half of the hybrid scheme is a DSMC region; the right half of the hybrid scheme is a MD region. Dashed lines represent cell boundaries. The particle depicted is transitioning from the DSMC to MD region. It will be placed in the MD region with zero potential energy.	20
5.1	Growth in energy as the hybrid simulation progresses. The solid line is generated by equation- 5.0.	28

5.2	Temperature profiles for DSMC, MD, and DSMC–MD simulations of a system with a thermal gradient. The error bars are typical of the MD and DSMC-MD errors. The solid line is a least squares fit of the DSMC data.	32
5.3	Number density profiles for DSMC, MD, and DSMC–MD simulations of a system with a thermal gradient. The error bars are typical of the MD and DSMC-MD errors. The solid line is derived from the ideal gas law and a least squares fit of the DSMC temperature data.	33
5.4	Velocity profiles for DSMC, MD, and DSMC–MD simulations of a Couette cell. The error bars are typical of the MD and DSMC-MD errors. The solid line is a least squares fit of the DSMC data.	34
5.5	Temperature profiles for DSMC, MD, and DSMC–MD simulations of a Couette cell. The error bars are typical of the MD and DSMC-MD errors. The solid line is a least squares fit of the DSMC data.	35
5.6	Number density profiles for DSMC, MD, and DSMC–MD simulations of a Couette cell. The error bars are typical of the MD and DSMC-MD errors. The solid line is derived from the ideal gas law and a least squares fit of the DSMC temperature data.	35
5.7	Velocity profiles for DSMC, MD, and DSMC–MD simulations of a Poiseuille flow. The error bars are typical of the MD and DSMC-MD errors. The solid line is derived from a least squares fit of the DSMC data.	36
5.8	Temperature profiles for DSMC, MD, and DSMC–MD simulations of a Poiseuille flow. The error bars are typical of the MD and DSMC-MD errors.	37

5.9	Number density profiles for DSMC, MD, and DSMC–MD simulations of a Poiseuille flow. The error bars are typical of the MD and DSMC–MD errors.	38
-----	--	----

Chapter 1

INTRODUCTION

Numerical simulations are often used to study fluid systems that are difficult to study via physical experimentation. Such systems include very large systems, such as planetary atmospheres, and very small systems, such as the space between the head and platter of a modern hard disk drive.

A fluid system simulation must account for the characteristics of the fluid. Fluid characteristics may be simulated by either a particle or a continuum method. The method of choice depends on the system of interest. In general, because of their greater computational demands, particle methods are only used when continuum methods fail (Bird, 1976). This occurs, for example, when the mean free path length, λ , of a molecule in the fluid is of the same scale as the characteristic length, L , of the system. Thus, a simulation of a system that is either microscopic (L very small) or rarified (λ very large) must be performed with a particle method.

In a particle method simulation, a fluid is represented as a collection of discrete particles. The position and momentum of each particle are tracked as the system evolves in time. If there are N particles represented in the simulation, the collection of the $6N$ coordinates and momenta form a point in the system's phase space at any instant in time. The trajectory of this point through the phase space is then used to calculate the bulk properties of the fluid. It is the generation of the trajectory

through phase space that makes particle methods computationally expensive.

One common particle method for the simulation of dilute gases is Direct Simulation Monte Carlo (DSMC). DSMC uses a two step probabilistic approach to generate a point on the phase space trajectory; repeated application of these two steps generates the phase space trajectory (Bird, 1976).

In the first step of the DSMC algorithm, the position and momentum of each particle is tracked by integrating the free particle equations of motion over a short time. This simulates the flight of particles between collisions. In the second step of the DSMC algorithm, a probabilistic acceptance-rejection scheme is used to select pairs of particles that will undergo elastic collisions. For each elastic collision, the collision angles are selected at random from the distributions obtained via kinetic theory (Garcia, 1994). These random, elastic collisions simulate the particle-particle interactions of a fluid.

The probabilistic nature of the DSMC method is the root of both its strengths and weaknesses. Calculations made during each time step are simple and independent. This makes the algorithm efficient. However, the DSMC method is limited to dilute gases because the collision probabilities are derived from dilute gas properties (Bird, 1976).

Yet there are many fluid systems that are not comprised of dilute gases. Hence, another type of particle method is required for dense fluids. A very powerful method is Molecular Dynamics (MD); in MD simulations, a point along the phase space trajectory is generated in two steps. First, the force acting on each particle is calculated (Allen & Tildesley, 1992). Then, the Newtonian equations of motion are numerically integrated over a short time. Repeating these two steps generates a trajectory through phase space.

The explicit force calculation in an MD simulation has two effects. First, it makes MD simulations very expensive. Second, to avoid numerical errors, the integration

time between force calculations must be very short. Since particles in a dilute gas rarely interact, MD simulations of dilute gases must be run for a large number of time steps (Reed & Flurchick, 1994). The long runs and short time steps required to simulate a dilute gas make MD simulations of dilute gases prohibitively expensive.

There are many physical systems that contain both dilute and dense regions. For example, the flow of air over a liquid contains a dilute region, the air, and a dense region, the liquid. These systems cannot be simulated by DSMC because of the dense regions, yet they are too expensive to simulate by MD because of the dilute regions. However, a hybrid technique that simulates dilute regions with DSMC and dense regions with MD could simulate these systems.

We have developed such a hybrid Direct Simulation Monte Carlo and Molecular Dynamics (DSMC-MD) technique. The remainder of this thesis details the implementation, testing, and limits of this algorithm. In Chapter 2, the properties of the fluid systems we simulated are discussed. Chapter 3 contains descriptions of the programs that were developed to perform the simulations. The conditions of each simulation are presented in Chapter 4. In Chapter 5, the results of each simulation are analyzed and presented. Finally, Chapter 6 is the summary and conclusion.

Chapter 2

Properties of the Fluid Systems

2.1 Fluid System Descriptions

The fluid system simulations presented in this thesis were performed as tests of the DSMC–MD algorithm. Five scenarios—an isolated system, an isothermal system, a system containing a thermal gradient, a Couette flow, and a Poiseuille flow—were selected for two reasons. First, these simulated fluid systems are well understood so problems with a simulation can be identified. Second, each fluid system exhibits characteristics that test different aspects of the simulation algorithm.

The five systems chosen for simulation have some common properties. For example, they all have a constant volume, one dimensional geometry. In this geometry, the fluid extends to infinity in the y and z directions, but is bound by a pair of parallel planes located at $x = 0$ and $x = L$.

Another common property of the selected systems is the fluid itself. In each case, the simulated fluid obeys the ideal gas law: $P \propto \rho T$. This relation allows for a simple verification of the temperature and density profiles derived from a simulation. The product of temperature and density at each point along the profile must be proportional to the pressure, P . Further, the pressure at a steady state is constant because $\frac{\partial P}{\partial x} \propto u$, the x component of velocity. In our one dimensional system, u is zero. Therefore, P is a constant.

The differences between the systems make each system useful for examining a particular aspect of the simulations. For example, the first two systems are used to test the behavior of the simulations at thermodynamic equilibrium. The last three simulations test the behavior of the simulations under nonequilibrium but steady state (i.e., time independent) conditions.

The first system, an equilibrium system, is an isolated fluid bound, in the x coordinate, by specularly reflecting walls. In this system, total energy and linear momentum—in y and z coordinates—are conserved. Hence, starting from any arbitrary initial condition, the system reaches an equilibrium state characterized by constant temperature, number density, and bulk flow velocity. Note that the bulk flow velocity may be nonzero because it is determined by the initial y and z components of total momentum. These properties are used to test the energy and momentum conservation of the DSMC, MD, and DSMC-MD algorithms.

The second system, also an equilibrium system, is an isothermal system bound by thermal walls. In this system, energy and momentum are exchanged with the walls. Since the walls are at equal temperatures, the system reaches an equilibrium state characterized by constant temperature and number density. In addition, the equilibrium bulk flow velocity is zero because the walls are at rest. This system is used to verify that the DSMC, MD, and DSMC-MD algorithms can simulate an equilibrium system that is in thermal contact with the outside world.

The other three simulations are of nonequilibrium systems that relax to a steady state. In each of these simulations, temperature and density gradients are produced. As mentioned above, ρ and T are related via the ideal gas law. In addition to the temperature and density gradients, the Couette and Poiseuille flows also exhibit velocity gradients.

The first of the steady state systems is bound by thermal walls that are maintained at different temperatures. In this system, heat is conducted from the hot

wall to the cold wall. The steady state reached by this system is characterized by a linear temperature gradient. Since the fluid obeys the ideal gas law, the density gradient is inversely proportional to T . The temperature of the fluid near the wall will not be the same as the temperature of the wall. This phenomenon is called thermal jump (Bird, 1976). In this system, the velocity gradients are zero. This fluid system is used to test the heat transport properties of the simulations.

A Couette cell is another of the steady state systems. In a Couette cell, the fluid is contained between moving, thermal walls. The motion of the wall is perpendicular to the coordinate separating the walls. In these simulations, the wall velocities are equal and opposite, and are parallel to the y coordinate. This system relaxes to a steady state characterized by a linear velocity profile. The velocity gradient is in the coordinate parallel to the motion of the walls. In this case, it is the y component of the fluid velocity, v , that exhibits a gradient. Like the thermal jump exhibited in the system with a thermal gradient, the velocity of the fluid near the walls is lower than the velocity of the walls (Morris, 1992). This phenomenon is known as velocity slip.

Temperature and density gradients are also present in a Couette flow. The temperature gradients are caused by viscous heating. As in the fluid with a thermal gradient, a thermal jump is present (Bird, 1976). The temperature gradients induce density gradients. Again, the temperature and density gradients are related via the ideal gas law.

The Couette flow simulations are used to demonstrate the ability of the DSMC, MD, and DSMC-MD algorithms to predict the velocity gradients, velocity slip length, thermal gradients, thermal jump, and density gradients present in a Couette cell.

The last fluid system, another steady state system, is the Poiseuille flow. In a Poiseuille flow, a constant external force acts upon the particles comprising the fluid.

For these simulations, the force acts in the negative z direction. This fluid system relaxes to an equilibrium that is characterized by a quadratic bulk flow velocity profile. It is the z component of the velocity, w , that exhibits a gradient. Here too, velocity slip is present. The wall velocity is zero, yet, the fluid velocity near the wall is nonzero.

Viscous heating caused by the velocity gradient induces temperature and density gradients in a Poiseuille flow. These profiles are quartic because the velocity profile is quadratic (White, 1986). Hence, the thermal and density gradients are more difficult to observe than those present in a Couette flow.

The Poiseuille flow is the only one of the five fluid systems that is subject to an external force. Hence, it is used to verify the accuracy of simulations in the presence of an external force.

2.2 Boundary Interactions

The differences between the isolated system, the isothermal system, the system with thermal gradient, and the Couette flow are all caused by differences in the system boundaries. Therefore, the different systems are simulated by implementing different models of the fluid–boundary interaction.

Many different types of interaction can occur at a boundary. For example, at a boundary made of a semiporous membrane, heat and fluid are exchanged. In contrast, at a perfectly insulating wall, neither heat nor fluid is exchanged. Intermediate between these extremes is a thermal wall; heat is exchanged with the surroundings but no fluid crosses the boundary. All five systems that are simulated for this study are bound by walls. The isolated system is bound by perfectly insulating walls; the other systems are bound by thermal walls.

Walls are implemented in a numerical simulation by testing the position of each particle at each time step. If the particle is not in a defined interval, e.g., $0 \leq x \leq L$,

it is assumed to have collided with a wall. At this point, the routine that implements the particle-wall interaction model is called.

Two specific models of particle-wall interactions are used in the simulations performed for this project: a specularly reflecting wall, and a thermal wall. The specularly reflecting wall is a model of a perfectly insulating wall. The thermal wall is a model of a wall in perfect thermal contact with an infinitely large heat bath.

The specularly reflecting wall is simple; when a particle reaches a specular wall, the sign of the velocity component perpendicular to the wall is reversed. The other two velocity components are unchanged.

In contrast, when a particle reaches a thermal wall, it momentarily adheres to the wall. During this time, energy and momentum are exchanged with the wall. The particle then leaves the wall with a new velocity. The distribution of particle velocities is a Maxwell-Boltzmann distribution biased towards faster moving particles (Garcia, 1994). The distributions of particle velocity components are given by

$$P(v_x) = \pm \frac{m}{kT_w} v_x e^{-mv_x^2/kT_w} \quad (2.1)$$

$$P(v_y) = \sqrt{\frac{m}{2\pi kT_w}} e^{-m(v_y - u_w)^2/kT_w} \quad (2.2)$$

$$P(v_z) = \sqrt{\frac{m}{2\pi kT_w}} e^{-mv_z^2/kT_w}, \quad (2.3)$$

where k is the Boltzmann constant, m the mass, u_w is the wall velocity, T_w is the wall temperature, and v the particle velocity. The wall velocity is nonzero only in the Couette flow.

The specular and thermal walls described above are physical boundaries. Algorithms are used to mimic the properties of the walls in numerical simulations. Similar algorithms are used in the DSMC and MD programs to mimic the walls.

These boundary algorithms and the DSMC, MD, and DSMC-MD programs are described in the next chapter.

Chapter 3

Programs

Three major C programs were developed for this thesis. Each program has five variants; one variant for each fluid system simulated. The variants were produced by including conditional compilation directives in the main programs. Compilation of all programs was directed from a common Makefile. Appendix-A contains the Makefile, the include files, and all of the code used to build the DSMC, MD, and DSMC-MD programs.

Several of the subroutines were derived from programs written by others. The `Sorter`, `Mover`, and `Collide` subroutines are derivatives of the `dsmcne` routines presented in Numerical Methods for Physics (Garcia, 1994). The memory allocation scheme is from Numerical Recipes in C (Press et al., 1992). The random number generator is `ran2` from the 2nd edition of Numerical Recipes in C (Press et al., 1992).

Where possible, the DSMC, MD, and DSMC-MD programs share subroutines. This was done to simplify the DSMC-MD code. The common subroutines, except for the statistics routines, will be described in the DSMC section.

In addition to sharing some subroutines, the implementation of boundary conditions is similar for all of the programs. Hence, the implementation of the boundary conditions is presented before the programs are presented.

3.1 Boundary Condition Implementations

All five fluid systems implemented for this project are bound by walls located at $x = 0$ and $x = L$. These walls are implemented by testing the position of each particle at each time step. If the particle is not in the interval $0 \leq x \leq L$, it is assumed to have collided with a wall. At this point, the routine that implements the particle-wall interaction model is called.

Two specific models of particle-wall interactions are used in the simulations performed for this project: a specularly reflecting wall, and a thermal wall. The specularly reflecting wall is a model of a perfectly insulating wall. The thermal wall is a model of a wall in thermal contact with an infinitely large heat bath.

The specularly reflecting wall is simplest to implement. First, the time that the particle crossed the boundary is calculated. What happens next differs for DSMC and MD simulations. For DSMC simulations, the sign of u , the x component of velocity, for the impacting particle is reversed. Then, the particle is moved, at the reversed velocity, for the remainder of the timestep. For MD simulations, the y and z coordinates of the particle at the time of the collision are calculated. The calculated positions are assigned to the y and z coordinates of the particle. Finally, the sign of the x component of the particle's velocity is reversed. In MD simulations, the particle remains on the wall for the remainder of the timestep.

The thermal wall implementation starts out the same as the specular wall implementation. The new position of the colliding particle is the same for both wall models. It is the velocity calculation that differs. For the thermal wall model, the new particle velocity is randomly selected from a biased Maxwell-Boltzman distribution (Garcia, 1994). This simulates the exchange of energy and momentum with an infinitely large heat bath. The x , y , and z components of the post collision velocity are given by:

Figure 3.1: A DSMC particle is moved and crosses a wall. The time and coordinates of the wall crossing are calculated. A new velocity is assigned from a biased Boltzmann distribution and the particle is moved off the wall.

Figure 3.2: A MD particle is moved and crosses a wall. The coordinates of the wall crossing are calculated. A new velocity is assigned from a biased Boltzmann distribution and the particle is left on the wall.

$$u = \sqrt{\frac{-2kT_w \ln(\mathcal{R}_1)}{m}}, \quad (3.1)$$

$$v = \sqrt{\frac{-2kT_w \ln(\mathcal{R}_2)}{m}} \sin(\mathcal{R}_3), \quad (3.2)$$

$$w = \sqrt{\frac{-2kT_w \ln(\mathcal{R}_2)}{m}} \cos(\mathcal{R}_3), \quad (3.3)$$

where k is Boltzmann's constant, m is the mass of the particle, T_w is the wall temperature, and $\mathcal{R}_1$, $\mathcal{R}_2$, and $\mathcal{R}_3$ are uniform random deviates on the open interval $(0,1)$. The x component of velocity, u , is always directed away from the wall. Figure- 3.1 is a diagram of the interaction of DSMC particles with a thermal wall. Figure- 3.2 is a diagram of the interaction of MD particles with a thermal wall.

A variant on the thermal wall is present in a Couette flow; the wall is moving. This is treated by simply adding the wall velocity to the velocity of each particle leaving the wall. For these simulations, this is done by adding the wall velocity to equation-3.2.

A wall is a physical boundary. In numerical simulations, there may also be boundaries that reflect the limits of numerical techniques. For example, a one dimensional system is actually a three dimensional system that is of infinite extent in two coordinates. A direct particle method simulation of such a system would require an infinite number of particles! Hence, a numerical limit must be placed on the system size.

In DSMC simulations this limit is particularly simple; the y and z positions are simply ignored. This is possible because changing the y or z coordinate of a particle does not change the probability of a collision. Hence, the system is assumed to be homogeneous in these coordinates.

For MD simulations, however, the forces acting on each particle are functions of

Figure 3.3: Periodic Boundary Conditions. The central cell is the simulation cell; the surrounding cells are its images. Once the simulation particle (disk) leaves the central cell, its image (circle) becomes the simulation particle.

the distance to other particles. Hence, the y and z coordinates cannot be ignored. The infinite extent of the y and z coordinates is simulated via periodic boundary conditions; a particle that leaves one side of the simulation is replaced by an image particle that enters the opposite side of the system. Figure 3.3 depicts periodic boundary conditions.

3.2 Direct Simulation Monte Carlo Program

The DSMC program is composed of an initialization section and a main loop. The initialization section reads the files containing the simulation parameters, initial particle positions, and initial particle velocities. The main loop repeatedly executes the DSMC algorithm. In addition, statistics sampling and data storage procedures are periodically called during main loop execution.

Initialization of a DSMC simulation consists of several steps. First, simula-

tion parameters, such as the number of particles, system size, boundary conditions, timestep size, and number of timesteps, are read from a data file. Next, the `InitializeGlobals` subroutine allocates space for the global variables and loads the initial particle positions and velocities from data files. The `InitializeGlobals` subroutine is used in DSMC, MD, and DSMC–MD codes. Finally, parameters derived from the basic conditions are calculated.

The main loop executes the DSMC algorithm once each timestep. Three major subroutines are called. At the beginning of each timestep, the `Mover` subroutine updates the particle positions. Next, the `Sorter` subroutine creates a list of particles in each cell. Finally, the `Collide` subroutine implements the acceptance–rejection and collision algorithms. Two other procedures, `Sampler` and `StoreGlobals`, are periodically called from inside the main loop. These procedures collect the simulation statistics and store them to disk files. They are described in the Statistics section.

The `Mover` subroutine implements the first half of a DSMC timestep. It updates the position of each particle in the system and implements the boundary conditions.

The `Mover` subroutine is also used in the DSMC–MD program. In this program, the `Mover` subroutine only acts on particles in the DSMC portion of the system. Hence, instead of a simple loop over all of the particles in the system, two nested loops are used to identify the particles in the DSMC portion of the system. The outer loop steps through each cell; the inner loop steps through each particle in a cell. A sorted list of particles is used to determine which particles are in which cells. The `Sorter` subroutine is used to create the sorted list.

As each particle is selected, its new position is calculated. Euler’s method is used to integrate the x component of the free particle equation of motion. The new position of a particle is given by:

$$\mathbf{x}(t + \tau) = \mathbf{x}(t) + \tau \mathbf{v}(t), \quad (3.4)$$

where τ is the timestep, t is the current time, and v is the y component of velocity.

If the simulation is of a Poiseuille flow, the velocity component parallel to the applied force is also upgraded. The new velocity is given by

$$v(t + \tau) = v(t) + \tau a_{ext}, \quad (3.5)$$

where a_{ext} is the acceleration due to the external force.

After a particle is moved, its position is compared to the boundary positions. If the particle has crossed a boundary, **Mover** imposes the boundary conditions appropriate to the selected model system.

After the **Mover** subroutine has executed, the **Sorter** subroutine is called. It creates a list of particles located in each cell. The sorted list of particles is used by **Mover** and **Collide** subroutines.

Collide implements the second half of a DSMC timestep with a three step acceptance-rejection scheme. First, prospective collision pairs are randomly selected. Second, the collision is randomly accepted or rejected. Third, elastic collisions between accepted particle pairs are simulated.

The first step in the acceptance-rejection scheme, prospective collision pair selection, is implemented by randomly selecting pairs of particles that are located in the same region of the system. The **Sorter** routine is used to identify particles that are close to each other. The number of prospective collision pairs selected is derived from the expected collision frequency for a dilute gas (Garcia, 1994).

The second step is the acceptance or rejection of prospective collision pairs. A prospective collision is accepted if

$$\frac{|\mathbf{v}_i - \mathbf{v}_j|}{v_r^{max}} > \mathcal{R} \quad (3.6)$$

where $|\mathbf{v}_i - \mathbf{v}_j|$ is the relative velocity, v_r^{max} is the maximum relative velocity, and $\mathcal{R}$ is a uniform random deviate in $(0, 1)$ (Garcia, 1994).

Finally, **Collide** calculates the post collision velocities of the accepted collision pairs. The post collision velocities of particles i and j are given by

$$\mathbf{v}_i(t + \tau) = \mathbf{v}_{cm} + \frac{1}{2}\mathbf{v}_r(t) \quad (3.7)$$

and

$$\mathbf{v}_j(t + \tau) = \mathbf{v}_{cm} - \frac{1}{2}\mathbf{v}_r(t). \quad (3.8)$$

The center of mass velocity is $\mathbf{v}_{cm}$, and the precollision relative velocity is $\mathbf{v}_r(t)$. The angle of the collision is selected from a distribution that is uniform over the unit sphere. After **Collide** has executed, the timestep is complete.

3.3 Molecular Dynamics Program

The MD program, like the DSMC program, is composed of three major sections: initialization, main loop, and statistics sampling. The initialization section reads the simulation parameters, initial particle positions, and initial particle velocities files. The main loop repeatedly executes the MD algorithm. At specified intervals, the statistics sampling procedure is called.

The initialization procedure is nearly the same for both MD and DSMC programs. The code that reads the data files, allocates memory, and initializes the variables is identical to the corresponding DSMC code. MD initialization has one more step than DSMC initialization; the net force acting on each particle is calculated prior to entering the main loop.

The MD program main loop repeatedly executes the MD algorithm. Four subroutines are sequentially called. First, the **StartMove** subroutine does three things. It calculates the position of each particle at the end of the timestep, calculates the velocity of each particle half way through the timestep, and applies the boundary conditions. Next, the **Sorter** subroutine creates a list of particles in each cell. Then, the **CalcForces** subroutine calculates the acceleration acting on each particle during

the next timestep. Finally, the **FinishMove** subroutine completes the velocity calculation for each particle. As in the DSMC program, the **Sampler** and **StoreGlobals** subroutines are periodically called prior to executing the next timestep.

The **StartMove** subroutine is also used in the DSMC-MD program. Hence, **StartMove** uses the same sorted list method that the DSMC code uses to determine which particles are in the MD section of the hybrid system. For each particle, a velocity Verlet algorithm is used to integrate the kinematic equations of motion (Allen and Tildesley, 1992). The velocity Verlet algorithm is:

$$\mathbf{x}(t + \tau) = \mathbf{x}(t) + \tau \mathbf{v}(t) + \frac{\tau^2}{2} \mathbf{a}(t), \quad (3.9)$$

$$\mathbf{v}(t + \frac{1}{2}\tau) = \mathbf{v}(t) + \frac{1}{2}\tau \mathbf{a}(t), \quad (3.10)$$

and

$$\mathbf{v}(t + \tau) = \mathbf{v}(t + \frac{1}{2}\tau) + \frac{1}{2}\tau \mathbf{a}(t + \tau). \quad (3.11)$$

$\mathbf{x}$ is the position vector, $\mathbf{v}$ is the velocity vector, $\mathbf{a}$ is the acceleration vector, t is the current time, and τ is the timestep. This algorithm has a local truncation error $O(\tau^4)$. The subroutine **StartMove** implements the first two steps, equations 3.9 and 3.10, of the velocity Verlet algorithm. The last step is implemented in **FinishMove** because the new accelerations are required. The last section of the **StartMove** subroutine imposes the boundary conditions described in the Boundary Interactions.

After the boundary conditions are applied, the **CalcForces** subroutine is called from the main loop. This routine calculates the accelerations required for the last half of the velocity calculation and the next position calculation. The model force used in the MD simulations is derived from the Lennard-Jones potential (Allen & Tildesley, 1992). The force on particle i is given by

$$\mathbf{F}_i = \sum_{j=i+1}^N \frac{24\epsilon}{\sigma^2} \left[2 \left(\frac{\sigma}{r} \right)^{14} - \left(\frac{\sigma}{r} \right)^7 \right] \mathbf{r}_{ij}, \quad (3.12)$$

where ϵ is the potential well depth, σ is the atomic diameter, $\mathbf{r}_{ij}$ is the vector from particle i to particle j , and r is the distance between particles i and j . The force is modified if the simulation is of a Poiseuille flow. In this case the external force is added to the interparticle force.

The force calculation is the most expensive part of the MD simulation. The time required to calculate the force is reduced by imposing a cutoff distance. The cutoff distance is implemented by stepping through each cell, selecting each particle in that cell, and calculating the force on that particle. The cell length is the cutoff distance. Therefore, only particles in the same or neighboring cells are within the cutoff distance. If two particles are farther apart than the cutoff distance, their interaction force is not calculated. As in the DSMC program, the sorted particle list generated by `Sorter` is used to identify which particles are in each cell.

After `CalcForces` is done executing, `FinishMove` is called. `FinishMove` performs the second half of the velocity calculation equation— 3.11. A flag set by `StartMove` causes `FinishMove` to skip the velocity calculation for particles that collided with a wall. At the end of `FinishMove`, a single MD timestep is complete.

Again, statistics sampling is periodically performed by the `Sampler` subroutine. Also, the `StoreGlobals` subroutine writes the simulation statistics to disk every 1000 timesteps.

3.4 Hybrid DSMC–MD Program

The left half of the hybrid DSMC–MD system is the DSMC side; the right half of the hybrid system is the MD side. The first cell on the MD side is a transition cell. A picture depicting the geometry of the hybrid DSMC–MD system is presented

Figure 3.4: The left half of the hybrid scheme is a DSMC region; the right half of the hybrid scheme is a MD region. Dashed lines represent cell boundaries. The particle depicted is transitioning from the DSMC to MD region. It will be placed in the MD region with zero potential energy.

in Figure- 3.4. Like the DSMC and MD programs, the DSMC-MD program is composed of three main sections: initialization, main loop, and statistics sampling and storage. Initialization and statistics sampling and storage is the same for all three programs. The DSMC-MD main loop is an amalgamation of the DSMC and MD program main loops.

The DSMC-MD main loop executes five main tasks. First, all particles are moved. Next the particles are sorted into cells. Then the particles that entered the MD region from the DSMC region are assigned y and z coordinates. The particle interactions are then calculated. Finally, the MD velocity calculation is completed.

The first task, moving all of the particles, executes by calling the DSMC `Mover` subroutine for the DSMC particles and the MD `StartMove` subroutine for the MD particles. There is one modification made to the DSMC `Mover` subroutine for use in the DSMC-MD program. The number of any particle that leaves the DSMC region and enters the MD region is pushed onto a list of transition particles. This list is

used by the `InsertCrossingParticles` subroutine.

The second task, sorting the particles into cells, is performed by the `Sorter` subroutine. This subroutine is common to all three programs.

The `InsertCrossingParticles` subroutine assigns y and z coordinates to each particle that enters the MD region. First, y and z coordinates are randomly selected from a uniform distribution. Next, the separation between the transition particle and every other particle in the transition cell and the adjacent MD cell is calculated. If any particle is within the cutoff distance used in the force calculation, the y and z coordinates of the transition particle are discarded and the process is repeated. Hence, a transition particle is inserted into the MD region with zero potential energy. The transition particle list is cleared after the y and z coordinates are assigned to all transition particles.

The fourth task is divided into two parts. The particle interactions in the DSMC region are handled by the `Collide` subroutine; the particle interactions in the MD region are handled by the `CalcForces` subroutine. There is one modification made to the `CalcForces` subroutine for its use in the DSMC-MD program. The particles in the first MD cell only interact with particles in the first and second MD cells. In the pure MD code, the particles in an interior cell interact with particles in both adjacent cells.

The final task is to finish the velocity calculation for particles in the MD region. The `FinishMove` subroutine performs this calculation. At this point the timestep is complete.

As with the DSMC and MD programs, the sampling and storage routines are called at the beginning of selected timesteps. These are described in the next section.

3.5 Statistics Sampling

The `Sampler` subroutine is called periodically during main loop execution. Consideration of the difference between MD and DSMC simulations led to the use of two sampling strategies. MD simulations of dilute gases have two disparate timescales (Reed & Flurchick, 1994). This makes long simulation times impractical. In contrast, there is only one timescale present in DSMC simulations.

A long sampling period requires long simulation times. Since long MD and DSMC-MD simulations are impractical, in those simulations the sampling period was shorter than the correlation time. Therefore, consecutive samples were correlated, so, the sample standard deviation underestimates the standard error of the mean. For this reason, the theoretical fluctuation variance, for an ideal gas, was used to obtain the uncertainty in the means for MD and DSMC-MD simulations (Landau & Lifshitz, 1993). Note that the estimate of the mean is, however, still valid.

In contrast, the sampling period was longer than the correlation time for DSMC simulations. Therefore, both the mean and variance estimates were valid.

When the `Sampler` routine is called, it calculates and maintains running averages and variances of temperature, particle density, and velocity for each cell. The calculation of the number density is simply a temporal average of the instantaneous number of particles in a cell.

The calculation of the steady state (or equilibrium) bulk fluid velocity is more complex. At some timestep, the average velocity is calculated for all of the particles in a cell. This spatial average is added to a running temporal average. The temporal average is the bulk velocity of the fluid in a cell.

The temperature in each cell is derived from the energy equipartition theorem

(Haile, 1992). The temperature in a cell, at any given time is given by

$$T = \frac{M}{3k} (\langle u^2 \rangle + \langle v^2 \rangle + \langle w^2 \rangle) - \langle KE \rangle \quad (3.13)$$

The atomic mass is M , k is Boltzman's constant, $\langle KE \rangle$ is the average kinetic energy, and u , v , and w are the x , y , and z velocity components. The $\langle u^2 \rangle$, $\langle v^2 \rangle$, and $\langle w^2 \rangle$ averages are instantaneous spatial averages. The kinetic energy of the bulk flow in a cell is $\langle KE \rangle$; it is a time average. This instantaneous temperature is then added to a running temporal average. The temporal averages for each cell are the reported temperatures.

The accumulated averages are periodically stored to ascii data files by the subroutine `StoreGlobals`. `StoreGlobals` also stores the instantaneous position and velocity for every particle in the system. This allows a simulation to be restarted if it is interrupted.

Chapter 4

Simulation Conditions

The five fluid systems that were simulated are very similar. All share the same geometry, number of particles, and particle properties. The common properties of all the simulations are detailed first. Initialization of the simulations is described next. Finally, the specific parameters for all of the simulations are presented.

All five fluid system simulations shared the same geometry. Each system was approximately one-third of a mean free path long ($219.6\text{\AA}$). In the MD simulations, periodic boundary conditions were used in the y and z directions. In these directions, the systems were $291.2\text{\AA}$ long. The DSMC simulations did not depend upon the y and z coordinates.

The number of particles, 500, used in each simulation was another common factor. This number of particles was selected because it is large enough for statistical purposes but, it is small enough to be tractable with limited computing power.

The particle properties were selected to mimic Argon gas. The particle diameter was $3.405\text{\AA}$ and the particle mass was $6.63 \times 10^{-23}\text{g}$. The parameter for the Leonard-Jones potential, ϵ/k , was 119.8K (Morris, 1992).

The initial configuration for all simulations was obtained in the following manner. Five hundred particles were placed on a face centered cubic lattice with the lattice spacing set at the diameter of an Argon atom ($3.405\text{\AA}$). Velocities for each

particle were randomly selected from a 400K Maxwell-Boltzmann distribution. The assigned velocities were then adjusted so that the total linear momentum was zero (Haile, 1992). This temperature was hot enough to assure melting, and cool enough that large (10fs) timesteps could be used in the MD simulations. An MD simulation was then run, with periodic boundary conditions in all coordinates, for 3,000 picoseconds. The particle velocities were then reset to the initial velocities. Another 3,000 picosecond run was performed. This allowed the system to reach equilibrium. A third 3,000 picosecond run, starting with the ending positions and velocities of the second run, was then used to collect temperature, density, and velocity profiles. These were evaluated to verify that the lattice had melted and that equilibrium had been reached. The ending positions of this third run became the starting positions for subsequent simulations.

For subsequent simulations, the temperature was set by randomly selecting the velocities from the Maxwell-Boltzmann distribution for a 300K gas. The average velocity was then subtracted from the velocity of each particle. This set the total linear momentum to zero (Haile, 1992).

For this configuration, the ratio of potential to kinetic energy is approximately 10^{-3} . This ratio of potential to kinetic energy is typical of a dilute gas.

The specific boundary conditions for each of the simulations are presented in Table- 4.1. Differences between the isolated, isothermal, thermal gradient, and Couette flow result from differences in boundary conditions. The Poiseuille flow is simulated by applying a constant external force to the particles in an isothermal system. The acceleration due to this force is 10^{12} m/s^2 ; external forces of this magnitude are not unusual in microscopic flow simulations.

System Type	Duration (ps)	Timestep (ps)	Wall Temp	Wall Mach
Specular Walls				
DSMC	1.0×10^6	1.00	N/A	N/A
MD	1.0×10^4	0.01	N/A	N/A
DSMC-MD	1.0×10^4	0.01	N/A	N/A
Thermal Walls				
DSMC	1.5×10^6	1.00	300	0
MD	3.0×10^3	0.01	300	0
DSMC-MD	3.0×10^3	0.01	300	0
Thermal Gradient				
DSMC	1.0×10^6	1.00	300/400	0
MD	1.0×10^4	0.01	300/400	0
DSMC-MD	1.0×10^4	0.01	300/400	0
Couette Flow				
DSMC	1.0×10^6	1.00	300	0.75
MD	1.0×10^4	0.01	300	0.75
DSMC-MD	1.0×10^4	0.01	300	0.75
Poiseuille Flow				
DSMC	1.0×10^6	1.00	300	0
MD	1.0×10^4	0.01	300	0
DSMC-MD	1.0×10^4	0.01	300	0

Table 4.1: Simulation Parameters for all of the simulations. Wall temperatures are specified as left/right if the walls are at different temperatures.

Chapter 5

Results and Discussion

The results from simulations of five fluid systems by DSMC, MD, and DSMC-MD programs are presented, analyzed, and discussed in this chapter. A complete set of temperature, density, pressure, and velocity profiles is presented in Appendix-B.

5.1 Isolated System

The system bound by specular walls is a conservative system; energy and the components of linear momentum parallel to the walls are conserved. This implies that the system will progress to an equilibrium state. Hence, testing energy and linear momentum conservation during a simulation is an evaluation of the simulation method.

The degree to which energy was conserved in each of the three simulations—DSMC, MD, and DSMC-MD—is a test of the limitations of each method. Energy was conserved, to machine precision, over the time spanned by the DSMC simulation. Energy fluctuated over the lifetime of the MD simulation. At the end of the MD simulation the energy was 0.3% higher than the initial energy.

In contrast to the DSMC and MD simulations, energy increased monotonically over the lifetime of the DSMC-MD simulation. This effect is depicted in Figure- 5.1 and is explained by the following. Particles that leave the MD region have potential

Figure 5.1: Growth in energy as the hybrid simulation progresses. The solid line is generated by equation- 5.0.

energy that is ignored in the DSMC region. In contrast, particles that enter the MD region are placed so that they have no potential energy. Hence, particles crossing from the MD to the DSMC side of the system remove negative potential energy from the system. The result is a growth, over time, in the total energy of the system.

This temporal growth in total energy should be predictable; the average potential energy per particle, $\langle PE \rangle$ and the average rate particles leave the MD side of the system, $\dot{\Phi}$, should determine the average rate of energy growth, $\dot{E}$. With this assumption, the average rate of energy growth, normalized to the initial energy of the system, $E(0)$, is given by

$$\dot{E} = -\dot{\Phi} \frac{\langle PE \rangle}{E(0)}. \quad (5.1)$$

Over the duration of the simulation, $\langle PE \rangle / E(0)$ was $-4.38 \times 10^{-6} \pm 8.68 \times 10^{-8}$ per particle and $\dot{\Phi}$ was 2.02 ± 0.014 particle/ps. The predicted average growth of energy during a simulation is depicted by the solid line on Figure- 5.1. Early in the simulation, the agreement between predicted and measured growth is excellent.

As the simulation progresses, the predicted energy growth is too large. Neither $\langle PE \rangle$ nor $\langle \Phi \rangle$ change significantly over the duration of the experiment. Further, the uncertainty in $\dot{E}$ ($\pm 1.87 \times 10^{-7}$) is too small to account for the disagreement.

One possible cause for the disagreement is the discrepancy between the average potential energy of a particle in the first, or last, MD cell and the average potential energy of a particle in any other cell. In the first and last MD cells, only particles in two cells contribute to the potential energy of a particle. In other MD cells, particles in three cells contribute to the potential energy. Hence, the average potential energy lost by a particle moving from the MD to the DSMC side of the system is less than the average potential energy of a particle elsewhere in the MD region.

The monotonic increase in energy observed in the DSMC–MD simulations has side effects. For example, the hybrid DSMC–MD system exhibited a slow increase in temperature. In contrast, the MD and DSMC simulations reached equilibrium.

The energy conservation failure of the DSMC–MD simulation was not catastrophic. The observed temperature, density, pressure, and velocity profiles were all constant; there were no gradients observed. This is consistent with the results of both DSMC and MD simulations (see Appendix-B, Figures B1-B3).

The average temperatures and velocities for the three simulations are presented in Table- 5.1. The standard error of the means are estimated from the theoretical standard deviations (see Chapter 3, Statistics Sampling). The DSMC and MD derived temperatures were the same. The DSMC–MD derived temperature was higher. This is caused by the accumulation of energy during the DSMC–MD run. Note that the average u , v , and w velocities derived from each of the three simulations are statistically indistinguishable. The average number density is always given by N_p/N_c , where N_p is the number of particles and N_c is the number of cells, because the number of particles in the system is conserved.

	Temperature (K)	
DSMC	227.4	± 0.03
MD	227.6	± 0.2
DSMC-MD	235.2	± 0.4
	u (m/s)	
DSMC	9.5×10^{-5}	$\pm 3.2 \times 10^{-2}$
MD	-3.9×10^{-3}	$\pm 5.0 \times 10^{-2}$
DSMC-MD	-2.6×10^{-5}	$\pm 7.6 \times 10^{-2}$
	v (m/s)	
DSMC	-1.0×10^{-2}	$\pm 5.0 \times 10^{-2}$
MD	-1.0×10^{-2}	$\pm 2.7 \times 10^{-1}$
DSMC-MD	-6.1×10^{-5}	$\pm 6.0 \times 10^{-1}$
	w (m/s)	
DSMC	7.8×10^{-3}	$\pm 6.2 \times 10^{-2}$
MD	4.0×10^{-2}	$\pm 4.0 \times 10^{-1}$
DSMC-MD	-1.2×10^{-5}	$\pm 5.3 \times 10^{-1}$

Table 5.1: Average temperature and velocities from the Isolated system simulations.

5.2 Isothermal System

The isothermal system is not a conservative system because the particles exchange both energy and momentum with the walls. It is, however, an equilibrium system. The system approaches a uniform temperature equal to that of the walls. At the same time, the average number density and the average velocity components become constant functions of position.

All three simulations reached equilibrium. As with the isolated system, the temperature, number density, pressure, and velocity profiles are all constant (see Appendix B, Figures B4-B6). The average values for the temperature and velocity components are presented in Table- 5.2. As in the isolated system, the average number density is always given by N_p/N_c because the number of particles in the system is conserved.

The velocities derived from the DSMC simulation agree with theory. The y and z components of velocity derived from both the MD and DSMC-MD simulations are higher than expected. This is most likely an artifact of the periodic boundary

Temperature (K)		
DSMC	300.0	± 0.05
MD	298.6	± 0.4
DSMC-MD	303.9	± 0.6
u (m/s)		
DSMC	1.3×10^{-2}	$\pm 3.8 \times 10^{-2}$
MD	-6.5×10^{-2}	$\pm 8.1 \times 10^{-2}$
DSMC-MD	-2.6×10^{-1}	$\pm 7.6 \times 10^{-1}$
v (m/s)		
DSMC	-3.0×10^{-1}	$\pm 4.1 \times 10^{-2}$
MD	-1.7×10^0	$\pm 6.6 \times 10^{-1}$
DSMC-MD	-6.1×10^0	$\pm 6.0 \times 10^{-1}$
w (m/s)		
DSMC	7.1×10^{-2}	$\pm 4.0 \times 10^{-2}$
MD	3.4×10^{-2}	$\pm 6.0 \times 10^{-1}$
DSMC-MD	-1.2×10^0	$\pm 5.3 \times 10^{-1}$

Table 5.2: Average Temperature and velocities from the Isothermal system simulations.

conditions and short simulation times. The use of periodic boundary conditions imposes an artificial periodicity on the system (Haile, p. 86).

In addition, the system is shorter, in all three coordinates, than one mean free path length. This makes it likely that a particle with large y or z velocities will make several passes through a cell before interacting with another particle. In contrast, a particle with a large x velocity is likely to strike a wall before interacting with another particle. Therefore, the relaxation time for the y and z velocities is longer than the relaxation time for x velocities.

5.3 System with Thermal Gradient

A thermal gradient was introduced by maintaining the left and right walls at different temperatures; the left wall was maintained at 300K and the right wall was maintained at 400K.

The temperature profile for the system with a thermal gradient should be lin-

Figure 5.2: Temperature profiles for DSMC, MD, and DSMC-MD simulations of a system with a thermal gradient. The error bars are typical of the MD and DSMC-MD errors. The solid line is a least squares fit of the DSMC data.

ear. Figure- 5.2 shows the temperature profiles derived from the DSMC, MD, and DSMC-MD simulations. The line plotted over the data is the least squares best fit to the DSMC data. All three simulations produced the same temperature profile.

The thermal gradient induces a density gradient that is inversely proportional to the temperature gradient. The number density profile, given by

$$n(x) = \frac{\langle n(x)T(x) \rangle}{T(x)}, \quad (5.2)$$

is derived from the ideal gas law and the temperature profile. The number density profiles are presented in Figure- 5.3. All three simulations are consistent with the fit generated from equation- 5.2 and the DSMC temperature data. Hence, all three algorithms produce simulated flows that obey the ideal gas law.

Figure 5.3: Number density profiles for DSMC, MD, and DSMC-MD simulations of a system with a thermal gradient. The error bars are typical of the MD and DSMC-MD errors. The solid line is derived from the ideal gas law and a least squares fit of the DSMC temperature data.

5.4 Couette Flow

A Couette flow is induced by moving two parallel walls in opposite directions. In the simulations done for this thesis, the left wall was moved at Mach 0.75 (242 m/s) in the negative y direction and the right wall was moved at the same speed in the positive y direction.

The motion of the walls sets up a linear velocity gradient parallel to the motion. Figure- 5.4 is a graph of the velocity profiles, for the y component of velocity, obtained from all three simulations. The solid line is the linear least squares best fit to the DSMC data. As in the previous profiles, a typical error bar is presented for the MD and DSMC-MD simulation data. All three simulations produced the same velocity profile.

In the directions perpendicular to the motion of the wall, the velocity profiles

Figure 5.4: Velocity profiles for DSMC, MD, and DSMC-MD simulations of a Couette cell. The error bars are typical of the MD and DSMC-MD errors. The solid line is a least squares fit of the DSMC data.

are uniform. These profiles are presented in Appendix-B, Figures B10-B12.

The velocity gradient set up in a Couette flow induces quadratic temperature and number density gradients. Figure- 5.5 is a graph of the temperature profiles obtained in all three simulations. The solid line is a linear least squares fit of the DSMC temperature data to the equation $T(x) = a + c(x - 0.5)^2$. Figure- 5.6 is a graph of the number density profiles. The equation for the solid line in Figure- 5.6 is

$$n(x) = \frac{\langle n(x)T(x) \rangle}{a + c(x - 0.5)^2}. \quad (5.3)$$

This equation is derived from the ideal gas law and the least squares fit to the temperature data. Within the uncertainty of the measurements, all three simulations produced the same results.

Figure 5.5: Temperature profiles for DSMC, MD, and DSMC-MD simulations of a Couette cell. The error bars are typical of the MD and DSMC-MD errors. The solid line is a least squares fit of the DSMC data.

Figure 5.6: Number density profiles for DSMC, MD, and DSMC-MD simulations of a Couette cell. The error bars are typical of the MD and DSMC-MD errors. The solid line is derived from the ideal gas law and a least squares fit of the DSMC temperature data.

Figure 5.7: Velocity profiles for DSMC, MD, and DSMC-MD simulations of a Poiseuille flow. The error bars are typical of the MD and DSMC-MD errors. The solid line is derived from a least squares fit of the DSMC data.

5.5 Poiseuille Flow

A Poiseuille Flow is obtained by imposing a constant acceleration on the bulk fluid. The interaction of the fluid with the walls sets up a quadratic velocity profile. In the simulations performed for this project, an acceleration of 10^{12} m/s^2 was applied in the negative z direction.

The velocity profiles for the z component of velocity, w , are presented in Figure-5.7. The solid line is the least squares fit of the DSMC data to $w(x) = a + c(x - 0.5)^2$. All three simulations produced similar velocity profiles.

In the Poiseuille Flow the temperature and density profiles were, within the certainty of the measurements, uniform. As in the Couette flow, the velocity gradients should induce temperature and density gradients. The temperature should be slightly lower at the walls than in the center; the density should be slightly higher at the walls than in the center. The most likely explanation for the lack of observed

Figure 5.8: Temperature profiles for DSMC, MD, and DSMC-MD simulations of a Poiseuille flow. The error bars are typical of the MD and DSMC-MD errors.

temperature and density gradients involves the magnitude of the velocity gradients. The heating induced by the velocity gradients is dependant on the magnitude of the velocity gradients. The velocity gradients in the Poiseuille flow are less than half as strong as the velocity gradients in the Couette flow. Hence, the heating is less in the Poiseuille flow than in the Couette flow.

Figure 5.9: Number density profiles for DSMC, MD, and DSMC-MD simulations of a Poiseuille flow. The error bars are typical of the MD and DSMC-MD errors.

Chapter 6

Conclusions

The hybrid DSMC–MD program simulation results are consistent with the DSMC and MD results in all of the nonconservative systems. In addition, the temperature, density, and velocity profiles obtained with the DSMC–MD program were, within the limits of the expected variability, the same temperature, density, and velocity profiles obtained with the DSMC and MD programs. In particular, no anomalies were detected at the interface between the DSMC and MD portions of the hybrid system.

In the only conservative system, the system with specular walls, the only discrepancy is the failure of the DSMC–MD algorithm to conserve energy. The lack of energy conservation is explained by the loss of potential energy as particles move from the MD to the DSMC side of the system.

One way to counteract this loss of potential energy is to allow particles moving from the DSMC to the MD side of the system to gain potential energy. The amount of energy gained might be predicated on a dynamic energy balance.

When thermal walls are present, the failure of the DSMC–MD algorithm to conserve potential energy is not significant. This is because the energy flux through the walls is much greater than the potential energy lost as particles move from the MD to the DSMC portions of the system. Hence, the energy of the system is

maintained by the interaction of the fluid with the walls.

The best way to improve the testing of the DSMC-MD algorithm is to increase the length of the production runs. This is not practical until the algorithm is made more efficient. Currently, a $10ns$ simulation takes about twelve hours on a Sun Sparc-10. To decrease the uncertainty in the measurements by a factor of four would require $160ns$ simulations. This would require an eight day run; this is impractical on our computer system.

The most obvious way to improve the efficiency of the DSMC-MD algorithm is to improve the efficiency of the MD portion of the algorithm. The force calculation algorithm is the obvious place to start. Using a three dimensional sort scheme instead of a one dimensional sort could reduce the number of force calculations by about 10 times. This improvement in efficiency would allow longer simulation runs. In turn, this would allow better estimation of the actual errors.

One application of the hybrid DSMC-MD algorithm is the simulation of a gas-liquid interface. The liquid and a thin layer of gas at the surface would be in the MD region; the gas would be in the DSMC region.

Another application is the simulation of a gas flow that chemically reacts on or with a solid surface. The solid and the adjacent gas would be treated with MD; the gas away from the surface would be treated with DSMC.

The hybrid DSMC-MD algorithm is a simple combination of DSMC and MD algorithms. In nonconservative systems, the results of DSMC-MD simulations were indistinguishable from the results of DSMC and MD simulations. Most fluid flows of interest are nonconservative systems. Therefore, the DSMC-MD algorithm can be used to simulate nonconservative systems that contain both dilute and dense regions.

Bibliography

- [1] M.P. Allen and D.J. Tildesley. *Computer simulation of liquids*. (Oxford: Clarendon Press, 1992).
- [2] G.A. Bird. *Molecular Gas Dynamics*. (Oxford: Clarendon Press, 1976).
- [3] A.J. Garcia. *Numerical Methods for Physics*. (New Jersey: Prentice Hall, 1994).
- [4] J.M. Haile. *Molecular Dynamics Simulations: Elementary Methods*. (New York: John Wiley & Sons, 1992).
- [5] L.D. Landau and E.M. Lifshitz. *Statistical Physics*. 3rd. ed. Part 1. (Oxford: Pergamon Press, 1993).
- [6] D.L. Morris *Molecular Dynamics Studies of Slip Length in a Dilute Gas*. (Technical REport, Department of Physics: San Jose State University, 1992).
- [7] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling. *Numerical Recipes in C. The Art of Scientific Computing*. 2nd. ed. (Cambridge: Cambridge University Press, 1992).
- [8] A.S.C. Reed and K.M. Flurchick. *Hybrid Molecular Dynamics: an Approach to Low Density Simulations*. (Computer Physics Communications. **81**:56,1994).
- [9] F.W. White. *Fluid Mechanics*. (New York: McGraw-Hill, 1986).

Appendix A

Program Listings

A.1 Makefile

```
default:
    @echo "Use mds, dsmc, or dual as an argument."

.KEEP_STATE:

##Valid Wall flags: -DPERIODIC -DSPECULAR -D THERMAL -DDUAL -DPOISE
WALLS = -D THERMAL -DPOISE -DDUAL
DEBUG = -g
OPTIM = #-O
FLAGS = $(WALLS) $(DEBUG) $(OPTIM)
CC = cc -c $(FLAGS)

COMM = files.o initglobals.o sampler.o sorter.o storeglobals.o \
      utilities.o urandom.o
TEST = ##maxke.o brutepot.o partpairs.o

MDOBJ = mds.o calcforces.o startmove.o finishmove.o $(COMM) $(TEST)
DSOBJ = dsmc.o collide.o mover.o $(COMM) $(TEST)
DUOBJ = dual.o insertcrossingparticles.o \
      calcforces.o startmove.o finishmove.o \
      collide.o mover.o \
      $(COMM) $(TEST)

mds: $(MDOBJ)
    @echo "Building mds..."
    cc $(FLAGS) -o mds $(MDOBJ) -lm
    cp mds ../bin/mdspoise

dsmc: $(DSOBJ)
    @echo "Building dsmc"
    cc $(FLAGS) -o dsmc $(DSOBJ) -lm
    cp dsmc ../bin/dsmcpoise

dual: $(DUOBJ)
    @echo "Building dual..."
    cc $(FLAGS) -o dual $(DUOBJ) -lm
    cp dual ../bin/dualpoise

## md files
mds.o:
    $(CC) mds.c
calcforces.o:
    $(CC) calcforces.c
startmove.o:
    $(CC) startmove.c
finishmove.o:
```

```

$(CC) finishmove.c

##      dsmc files
dsmc.o:
    $(CC) dsmc.c
collide.o:
    $(CC) collide.c
mover.o:
    $(CC) mover.c

##      dual files
dual.o: $(DSOBS)
    $(CC) dual.c

insertcrossingparticles.o:
    $(CC) insertcrossingparticles.c

##      common files
files.o:
    $(CC) files.c
initglobals.o:
    $(CC) initglobals.c
sampler.o:
    $(CC) sampler.c
sorter.o:
    $(CC) sorter.c
storeglobals.o:
    $(CC) storeglobals.c
utilities.o:
    $(CC) utilities.c
urandom.o:
    $(CC) urandom.c

##      testing files
maxke.o:
    $(CC) maxke.c
brutepot.o:
    $(CC) brutepot.c
partpairs.o:
    $(CC) partpairs.c

```

A.2 globals.h

```
extern double  *x,*y,*z;
extern double  *vx,*vy,*vz;
extern double  *ax,*ay,*az;
extern double  *deltav,*vwall,*xwall,*xcell;
extern double  *dvtot,*dverr;
extern double  T,tau,tausqrd;
extern int     Np,Nc,Nt,Nw,timestep,seed;
extern int     *Xref,*cell,*indx,*offwall,*strikes;
extern int     ToMd[TRPARTS];
extern double  *ave_n,*ave_u,*ave_v,*ave_w,*ave_T,*bulk;
extern double  *var_n,*var_u,*var_v,*var_w,*var_T,nsamps;
extern double  L,Side,Rc,Rcsqrd;
extern double  posorder,kinetic,potential,vx2,vy2,vz2;
extern double  totcol,totsel;

/*the following variables describe the start
 *and stop of mds & dsmc regions
 */
extern int db,de,mb,me;
extern double split;
#ifdef DUAL
extern int todsmc;
#endif
```

A.3 definitions.h

```
/* Boltzman's constant  $ngA^2/(ps^2K)$  */
#define BOLTZ 1.3807e-15
/* mass of argon atom ng */
#define MASS 6.63e-14
/* diameter of argon atom A*/
#define DIAM 3.66
/* mass density of argon at 1 STP  $ng/A^3$ */
#define DENSITY 1.78e-18
/* cross section of argon in A*/
#define SIGMA 3.405
#define SIGMASQRD 11.594025
/* Boltz*epsilon to get joules*/
#define EPSILON 119.8
/*  $A/psd^2$  */
#define ACOEFF 5.1644
/*  $ngA^2/ps^2$  */
#define PCOEFF 6.6163e-13
/*  $ngA^2/ps^2$  */
#define KCOEFF 3.3165e-14
```

```

    /* temperature coefficient */
#define TCoeff MASS/(3*BOLTZ)

#ifdef POISE
    /*A/ps^2 ~10^12 g's*/
#define ACCEL 0.01
#else
#define ACCEL 0.0
#endif

#define PI 3.14159265358979
#define TWOPI 6.2831855
    /*positional order const 4*PI/(1.414*DIAM) */
#define CPO 2.42780488425

#define WRITEPERIOD 1000 /*write every timesteps/WRITEPERIOD */

#define TRPARTS 10

```

A.4 prototypes.h

```

#define Square(A) (A)*(A)
#include <stdio.h>

void Error();
void InitializeGlobals();
FILE *OpenFile();
void CloseFile();
double **DoubleMatrix();
double *DoubleVector();
int *IntVector();
double URandom();
void CalcForces();
void StartMove();
void FinishMove();
void MiddleMover();
double Sampler();
void StoreGlobals();
void FreeGlobals();
FILE *OutputFileInit(); /*filename,variables_list,number_vars*/
void InsertCrossingParticles();
#ifdef DEBUG
int maxke();
void PartPairs();
#endif

```

A.5 calcf forces.c

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include "prototypes.h"
#include "definitions.h"
#include "globals.h"
#define RCOLLIDE DIAM*DIAM

void CalcForces(tau)
    double tau;
{
    int i,j,k,l,p1,p2;
    double rsqrd,den,den1,den2,cutoffpot,dx,dy,dz,tmpforce;
    potential=0.0;
    cutoffpot=(SIGMASQRD/Rcsqrd)*(SIGMASQRD/Rcsqrd)*(SIGMASQRD/Rcsqrd);
    cutoffpot=cutoffpot*cutoffpot-cutoffpot;

    for (i=mb;i<=me;i++){
        /*for each cell*/
        for (j=1;j<=cell[i]-1;j++){
            p1=Xref[indx[i]+j-1];
            /*calc forces due to same cell particles*/
            for(k=j+1;k<=cell[i];k++){
                p2=Xref[indx[i]+k-1];
                dx=x[p1]-x[p2];
#ifdef PERIODIC
                dx=dx-L*anint(dx/L);
#endif
                dy=y[p1]-y[p2];
                dy=dy-Side*anint(dy/Side);
                dz=z[p1]-z[p2];
                dz=dz-Side*anint(dz/Side);
                rsqrd=dx*dx+dy*dy+dz*dz;
                if (rsqrd<=Rcsqrd){
                    den=SIGMASQRD/rsqrd;
                    den1=den*den*den;
                    den2=2.0*den1*den1-den1;
                    /*(r^-12 repulsive & r^-6 attractive potential) */
                    tmpforce=ACOEFF*den*dx*den2;
                    ax[p1]+=tmpforce;
                    ax[p2]-=tmpforce;
                    tmpforce=ACOEFF*den*dy*den2;
                    ay[p1]+=tmpforce;
                    ay[p2]-=tmpforce;
                    tmpforce=ACOEFF*den*dz*den2;
                    az[p1]+=tmpforce;
```

```

        az[p2]-=tmpforce;
        potential+=den1*den1-cutoffpot-den1;
    }
}
}
for (j=1;j<=cell[i];j++){
    p1=Xref[indx[i]+j-1];
    /* for every particle in cell, calculate forces due to
     * particles in cell to right of current cell. Cell 1
     * is to the right of cell Nc iff there are periodic
     * boundaries in the x direction
     */
#ifdef PERIODIC
    l= (i==Nc) ? 1:i+1;    /* used only with periodic bc's */
#else
    /*used only with walls*/
    if (i==Nc)
        break;
    else
        l=i+1;
#endif
    for(k=1;k<=cell[l];k++){
        p2=Xref[indx[l]+k-1];
        dx=x[p1]-x[p2];
#ifdef PERIODIC
        dx=dx-L*anint(dx/L);
#endif
        dy=y[p1]-y[p2];
        dy=dy-Side*anint(dy/Side);
        dz=z[p1]-z[p2];
        dz=dz-Side*anint(dz/Side);
        rsqrd=dx*dx+dy*dy+dz*dz;
        if (rsqrd<=Rcsqrd){
            den=SIGMASQRD/rsqrd;
            den1=den*den*den;
            den2=2.0*den1*den1-den1;
            /*(r^-12 repulsive & r^-6 attractive potential) */
            tmpforce=ACOEFF*den*dx*den2;
            ax[p1]+=tmpforce;
            ax[p2]-=tmpforce;
            tmpforce=ACOEFF*den*dy*den2;
            ay[p1]+=tmpforce;
            ay[p2]-=tmpforce;
            tmpforce=ACOEFF*den*dz*den2;
            az[p1]+=tmpforce;
            az[p2]-=tmpforce;
            potential+=den1*den1-cutoffpot-den1;
        }
    }
}

```

```
    }  
  }  
}  
potential *=PCOEFF;  
return;  
}
```

A.6 collide.c

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "prototypes.h"
#include "definitions.h"
#include "globals.h"

/*-----Collide-----*/

void Collide(vrmax,coeff)
    double *vrmax,coeff;

{
    double vcmx,vcmx,vcmz,vrelx,vrelx,vrelz;
    double select,crm,cr,cos_theta,sin_theta,phi;

    int i,k,kk,n,num,nselect,ip1,ip2,col;

    for (n=db;n<=de;n++){
        col=0;
        num=cell[n];
        if (num >1){
            /*calc. # of candidate collision pairs*/

            select=(double)(num*num)*coeff*vrmax[n];
            /*insures that col probability = select */
            nselect=(int)floor(select+URandom(&seed));
            crm=vrmax[n];
            for (i=1;i<=nselect;i++){
                /*Pick two particles at random*/
                k=(int)floor((double)num*URandom(&seed));
                kk=(k+(int)floor( (double)(num-1)*URandom(&seed) )+1)%num;
                if (k+indx[n]<1 || k+indx[n]>Np){
                    printf("collide error:  index=%ld  k=%ld \n",indx[n],k);
                    exit(1);
                }
                if (kk+indx[n]<1 || kk+indx[n]>Np){
                    printf("collide error:  index=%ld  kk=%ld \n",indx[n],kk);
                    exit(1);
                }

                ip1=Xref[k+indx[n]];
                ip2=Xref[kk+indx[n]];
                if (ip1<1 || ip1>Np || ip2<1 || ip2>Np){
                    printf("ip1=%ld ip2=%ld \n",ip1,ip2);
                    exit(1);
                }
            }
        }
    }
}
```



```

    }
#ifdef DUAL
    if (x[ip1] > split)
        Error("MD particle Collided.\n");
    if (x[ip2] > split)
        Error("MD particle Collided.\n");
#endif
    cr=0.0;
    cr=(vx[ip1]-vx[ip2])*(vx[ip1]-vx[ip2])+
        (vy[ip1]-vy[ip2])*(vy[ip1]-vy[ip2])+
        (vz[ip1]-vz[ip2])*(vz[ip1]-vz[ip2]);
    cr=sqrt(cr);
    crm = cr > crm ? cr:crm;
    if (cr/vrmax[n]>URandom(&seed)){
        col=col+1; /*a collision has occurred*/
        vcmx=0.5*(vx[ip1]+vx[ip2]);
        vcmy=0.5*(vy[ip1]+vy[ip2]);
        vcmz=0.5*(vz[ip1]+vz[ip2]);
        cos_theta=1.0-2.0*URandom(&seed);
        sin_theta=sqrt(1.0-cos_theta*cos_theta);
        phi=2.0*PI*URandom(&seed);
        vrelx=cr*cos_theta;
        vrelx=cr*sin_theta*cos(phi);
        vrelz=cr*sin_theta*sin(phi);
        vx[ip1]=vcmx+0.5*vrelx;
        vx[ip2]=vcmx-0.5*vrelx;
        vy[ip1]=vcmy+0.5*vrelx;
        vy[ip2]=vcmy-0.5*vrelx;
        vz[ip1]=vcmz+0.5*vrelz;
        vz[ip2]=vcmz-0.5*vrelz;
    }
    }
    vrmax[n]=crm;
    totsels+=(double)nrel;
    totcol+=(double)col;
}

return;
}

```

A.7 dsmc.c

```
/*Direct Simulation Monte Carlo for a Couette Cell.
 *
 *
 *Written by: David Lee
 *   Date: March 15, 1993
 * Revisions: 93Nov06 DTL Rewritten to be compatable with mds code.
 *
 */
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <strings.h>

#include "definitions.h"
#include "prototypes.h"
#include "globals.h"

/*Define global variables here.
 * Positions, velocities, distance matrix, accelerations
 * Number of: particles, cells, timesteps done,
 *timesteps to do, walls
 */

double  *x,*y,*z;
double  *vx,*vy,*vz;
double  *ax,*ay,*az;
double  *deltav,*vwall,*xwall,*xcell;
double  *dvtot,*dverr;
double  T,tau,tausqrd;
int      Np,Nc,Nt,timestep,seed;
int      *Xref,*cell,*indx,*offwall,*strikes;
int      ToMd[TRPARTS],ToDs[TRPARTS];
double  *ave_n,*ave_u,*ave_v,*ave_w,*ave_T,*bulk;
double  *var_n,*var_u,*var_v,*var_w,*var_T,nsamps;
double  L,Side,Rc,Rcsqrd;
double  posorder,kinetic,potential,vx2,vy2,vz2;
double  totcol,totsel;

/*
 *the following variables describe the start
 *and stop of mds & dsmc regions
 */
int db,de,mb,me;
double split;

main()
```

```

{
/*-----Declarations-----*/
/*Simulation Variables */

double mpv,mfp,volume,eff_num,Mach,Nparts,time,tsamp,coeff,*vrmax;
int i,startstep;

/*file variables*/
FILE *params,*velocity,*position;

int Nread;

/*-----INITIALIZATION-----*/
#ifdef PERIODIC
printf("THIS RUN IS WITH PERIODIC BOUNDARY CONDITIONS.\n\n");
#endif
#ifdef THERMAL
printf(
"THIS RUN IS WITH THERMAL WALLS in X, PERIODIC BC's in y,z.\n\n");
#endif
#ifdef SPECULAR
printf(
"THIS RUN IS WITH SPECULAR WALLS in X, PERIODIC BC's in y,z.\n\n");
#endif
#ifdef POISE
printf("The external acceleration is %lg A/ps^2.\n",ACCEL);
#endif
params=OpenFile("data.params","r");
velocity=OpenFile("data.velocity","r");
position=OpenFile("data.position","r");

Nread=fscanf(params,"%ld",&seed);
if (Nread!=1) Error("Error reading seed.");
seed = (seed>=0) ? -1*seed : seed;
Nread=fscanf(params,"%ld %ld %ld %ld",&Np,&Nc,&Nt,&timestep);
if (Nread!=4) Error("Error Reading Number Parameters.");
Nread=fscanf(params,"%lg %lg %lg %lg %lg",&L,&Side,&Rc,&T,&Mach);
if (Nread!=5) Error("Error Reading Physical Parameters 1.");
Nread=fscanf(params,"%lg %lg",&tau,&time);
if (Nread!=2) Error("Error Reading Physical Parameters 2.");

startstep=timestep;
InitializeGlobals(velocity,position);

/*set the values for the start and end of mds region here*/
db=1;      mb=0;
de=Nc;     me=0;

```

```

vrmax=DoubleVector(1,Nc);
xwall[1]=0.0;  xwall[2]=L;
volume=L*Side*Side;
Nparts=(double)Np;
eff_num=DENSITY*volume/(Nparts*MASS);

mfp=volume/(sqrt(2.)*PI*DIAM*DIAM*Nparts*eff_num);
mpv=sqrt(2*BOLTZ*T/MASS);
vwall[2]=Mach*sqrt(5./3.*BOLTZ*T/MASS);
vwall[1]=-vwall[2];

for (i=db;i<=de;i++){
    vrmax[i]=3*mpv;
    xcell[i]=((double)i-0.5)*L/(double)Nc;
}
coeff=0.5*eff_num*PI*DIAM*DIAM*tau/(volume/(double)Nc);
time=0.0;

printf("Np=%ld Nc=%ld Nt=%ld timestep=%ld\n",
        Np,Nc,Nt,timestep);
printf("L=%lg Side=%lg Volume=%lg T=%lg Mach=%lg\n",
        L,Side,volume,T,Mach);
printf("tau=%lg time=%lg mpv=%lg\n",tau,time,mpv);
printf("Each simulation particle represents %lg atoms.\n",eff_num);
printf("System length is %lg. Mean free pathlength is %lg.\n",
        L,mfp);
printf("Wall velocity is %lg m/s.\n",
        100.0*vwall[2]);

/*-----MAIN LOOP-----*/
for (i=1;i<=Np;i++) kinetic+=vx[i]*vx[i]+vy[i]*vy[i]+vz[i]*vz[i];
kinetic*=KCOEFF;
potential=0.0;
Sorter();

for (;timestep<=Nt;timestep++){
    if (timestep%100 ==0)
        posorder=Sampler( startstep );
    if ( (timestep%(10*WRITEPERIOD) == 0 )||(timestep==Nt)){
        printf(
            "%ld of %ld steps done. %lg/%lg collisions %ld strikes\n"
            timestep,Nt,totcol,totsel,strikes[1]+strikes[2]);
        StoreGlobals(time,Mach,startstep);
    }
    Mover(mpv);
    Sorter();
    Collide(vrmax,coeff);
}

```

```
    for (i=1;i<=Np;i++) kinetic+=vx[i]*vx[i]+vy[i]*vy[i]+vz[i]*vz[i];  
    kinetic*=KCOEFF;  
    time+=tau;  
  
}  
  
FreeGlobals();  
return(0);  
}
```

A.8 dual.c

```
/*Molecular Dynamics and DSMC Simulation of a Couette Cell.
 * MD uses Velocity-Verlet algorithm for integrating
 * equations of motion. DSMC uses Bird algorithm.
 *
 *Written by: David Lee
 *      Date: Feb. 2, 1994
 *
 */

/* TESTING PROTOTYPES HERE */
#ifdef DEBUG
int maxke();
double brutepot();
#endif

#include <math.h>
#include <stdlib.h>
#include <stdio.h>

#include "definitions.h"
#include "prototypes.h"

/*Define global variables here.
 * Positions, velocities, distance matrix, accelerations
 * Number of: particles, cells, timesteps done,
 *      timesteps to do, walls
 */

double  *x,*y,*z;
double  *vx,*vy,*vz;
double  *ax,*ay,*az;
double  *deltav,*vwall,*xwall,*xcell;
double  *dvtot,*dverr;
double  T,tau,tausqrd;
int      Np,Nc,Nt,Nw,timestep,seed;
int      *Xref,*cell,*indx,*offwall,*strikes;
int      ToMd[TRPARTS];
double  *ave_n,*ave_u,*ave_v,*ave_w,*ave_T,*bulk;
double  *var_n,*var_u,*var_v,*var_w,*var_T,nsamps;
double  L,Side,Rc,Rcsqrd;
double  posorder,kinetic,potential,vx2,vy2,vz2;
double  totcol,totsel;

/*
 *the following variables describe the
 *start and stop of mds & dsmc regions
```

```

    */
    int db,de,mb,me;
    double split;
    /*
    *todsmc counts the number of particles leaving
    *the md going to dsmc side
    */
    int todsmc;

main()
{
    /*file variables*/
    int i,Nread;
    FILE *params,*velocity,*position;

    /*simulation parameter variables*/
    double Mach,mpv,time,coeff,*vrmax;

    /*Simulation Variables */
    double Px=0.0,Py=0.0,Pz=0.0;
    double mfp,volume,mxke;
    int startstep;

    params=OpenFile("data.params","r");
    velocity=OpenFile("data.velocity","r");
    position=OpenFile("data.position","r");

    Nread=fscanf(params,"%ld",&seed);
    if (Nread!=1) Error("Error reading seed.");
    seed = (seed>=0) ? -1*seed : seed;
    Nread=fscanf(params,"%ld %ld %ld %ld",&Np,&Nc,&Nt,&timestep);
    if (Nread!=4) Error("Error Reading Number Parameters.");
    Nread=fscanf(params,"%lg %lg %lg %lg %lg",&L,&Side,&Rc,&T,&Mach);
    if (Nread!=5) Error("Error Reading Physical Parameters 1.");
    Nread=fscanf(params,"%lg %lg",&tau,&time);
    if (Nread!=2) Error("Error Reading Physical Parameters 2.");

    startstep=timestep;
    InitializeGlobals(velocity,position);
    vrmax=DoubleVector(1,Nc);

    /*set the values for the start and end of mds region here*/
    db=1;
    if (Nc%2 !=0)
        Error("Dual system requires even number of cells.\n");
    de=Nc/2;
    mb=de+1;
    me=Nc;

```

```

split= (double)de*Rc;

tausqrd=tau*tau;
mpv=sqrt(2.0*BOLTZ*T/MASS);
Rcsqrd=Rc*Rc;
vwall[2]=Mach*sqrt(5.0/3.0*BOLTZ*T/MASS);
vwall[1]=-vwall[2];
volume=L*Side*Side;
mfp=volume/(sqrt(2.)*PI*DIAM*DIAM*Np);
for (i=db; i<=de; i++){
    vrmx[i]=3*mpv;
    xcell[i]=((double)i-0.5)*L/(double)Nc;
}
coeff=0.5*PI*DIAM*DIAM*tau/(volume/(double)Nc);

#ifdef SPECULAR
    (void)printf("SPECULAR WALLS\n");
#else
    (void)printf("Thermal Walls\n");
#endif
#ifdef POISE
    (void)printf(
        "Acceleration = %lg in the negative z direction.\n",ACCEL);
#endif
(void)printf("Np=%ld Nc=%ld Nt=%ld timestep=%ld\n",
    Np,Nc,Nt,timestep);
(void)printf("L=%lg Side=%lg Volume=%lg T=%lg Mach=%lg\n",
    L,Side,volume,T,Mach);
(void)printf("tau=%lg time=%lg mpv=%lg\n",tau,time,mpv);
(void)printf("System length is %lg. Mean free pathlength is %lg.\n",
    L,mfp);
(void)printf("Wall velocity is %lg m/s.\n",vwall[2]);
(void)printf("\n");
(void)printf("last dsmc cell=%ld    first md cell=%ld\n",de,mb);
(void)printf("split=%lg\n",split);

todsmc = 0;
/*calculate first forces*/
Sorter();
CalcForces();

/*iterate*/
for (; timestep<=Nt;timestep++){
    if ((timestep%100==0)|| (timestep==startstep))
        posorder=Sampler(startstep);
    if ( timestep%(WRITEPERIOD) == 0 ){
        (void)printf(
"%ld of %ld steps done. %lg/%lg collisions %ld strikes %ld todsmc\n",

```



```

    timestep,Nt,totcol,totsel,strikes[1]+strikes[2],todsmc);
    StoreGlobals(time,Mach,startstep);
}
/*DSMC move*/
Mover(mpv);
/*MD move*/
StartMove(mpv);
/*Sort particles into cells*/
Sorter();
/*Assign y,z positions to DSMC->MD particles*/
InsertCrossingParticles();
/*Do the DSMC collisions*/
Collide(vrmax,coeff);
/*Calculate MD forces*/
CalcForces();
/*finish MD velocity calculation*/
FinishMove();

/*Clear the DSMC-MD transition particle list */
for (i=1;(ToMd[i] != -1)&&(i<TRPARTS);i++)
    ToMd[i]=-1;
time+=tau;
}
FreeGlobals();
return(0);
}

```

A.9 files.c

```
/*OpenFiles--routine opens the initial condition
 *   and parameter files.  If successful the passed
 *   *fileptr are initialized and a zero is returned.
 *   A 1 is returned if an error occurs.
 */

#include <stdio.h>
#include <stdlib.h>

FILE *OpenFile(filename,mode)
    char *filename,*mode;
{
    FILE *fileptr;
    fileptr=fopen(filename,mode);
    if (!fileptr){
        (void)fprintf(stderr,"Could not open %s in %s mode.\n",
            filename,mode);
        exit(1);
    }

    return fileptr;
}

void CloseFile(fileptr,filename)
    FILE *fileptr;
    char *filename;
{
    char mssg[534];

    mssg[533]=(char)0;

    (void)strcpy(mssg,"Error closing File: ");
    (void)strcat(mssg,filename);
    if (fclose(fileptr)!=0) Error(mssg);
    return;
}
```

A.10 finishmove.c

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>

#include "prototypes.h"
#include "definitions.h"
#include "globals.h"

void FinishMove()
{
    int i,j,pi,first_index,last_index,crossing;

    first_index=indx[mb]; /*index into Xref of first particle*/
    last_index =indx[me]+cell[me]-1;
    for (i=first_index;i<=last_index;i++){
        crossing = 0;
        pi=Xref[i];
        /*Clear the DSMC-MD transition particle list */
        for (j=1;(ToMd[j] != -1)&&(j<TRPARTS);j++){
            if (pi == ToMd[j]){
                crossing = 1;
                break;
            }
        }
        if ((offwall[pi]!=0)&&(crossing != 1)){
            /*update velocity from t+.5tau to t+tau*/
            vx[pi]+=0.5*tau*ax[pi];
            vy[pi]+=0.5*tau*ay[pi];
            vz[pi]+=0.5*tau*(az[pi]-ACCEL);
        }
    }
    return;
}
```

A.11 initglobals.c

```
#include <stdio.h>
#include "definitions.h"
#include "prototypes.h"
#include "globals.h"

void InitializeGlobals(velocity,position)
    FILE *velocity,*position;
{
    int i,err,Nread;
    err=0;
    kinetic=0.0;
    potential=0.0;
    totscl=0.0;
    totcol=0.0;
    strikes=0;
    nsamps=0;

    xcell=DoubleVector(1,Nc);
    x=DoubleVector(1,Np);
    y=DoubleVector(1,Np);
    z=DoubleVector(1,Np);
    vx=DoubleVector(1,Np);
    vy=DoubleVector(1,Np);
    vz=DoubleVector(1,Np);
    ax=DoubleVector(1,Np);
    ay=DoubleVector(1,Np);
    az=DoubleVector(1,Np);
    offwall=IntVector(1,Np);
    deltav=DoubleVector(1,2);
    vwall=DoubleVector(1,2);
    xwall=DoubleVector(1,2);

    ave_n=DoubleVector(1,Nc);
    ave_u=DoubleVector(1,Nc);
    ave_v=DoubleVector(1,Nc);
    ave_w=DoubleVector(1,Nc);
    ave_T=DoubleVector(1,Nc);

    var_n=DoubleVector(1,Nc);
    var_u=DoubleVector(1,Nc);
    var_v=DoubleVector(1,Nc);
    var_w=DoubleVector(1,Nc);
    var_T=DoubleVector(1,Nc);

    bulk=DoubleVector(1,Nc);
```

```

dvtot=DoubleVector(1,2);
dverr=DoubleVector(1,2);
strikes=IntVector(1,2);

xwall[1]=0.0;  xwall[2]=L;

cell=IntVector(1,Nc);
Xref=IntVector(1,Np);
indx=IntVector(1,Nc);

for (i=1;i<=Np;i++){
    Nread=fscanf(velocity,"%lg %lg %lg",&vx[i],&vy[i],&vz[i]);

    if (Nread!=3) Error("Error Reading velocity data.");
    Nread=fscanf(position,"%lg %lg %lg",&x[i],&y[i],&z[i]);
    if (Nread!=3) Error("Error Reading position data.");
    ax[i]=0;
    ay[i]=0;
    az[i]=0;
}

for (i=1;i<=Nc;i++){
    ave_n[i]=0.0;
    ave_u[i]=0.0;  ave_v[i]=0.0;  ave_w[i]=0.0;
    ave_T[i]=0.0;
    var_n[i]=0.0;
    var_u[i]=0.0;  var_v[i]=0.0;  var_w[i]=0.0;
    var_T[i]=0.0;
    bulk[i]=0.0;
}
for (i=1;i<=TRPARTS;i++)
    ToMd[i]=-1;

return;
}

```

A.12 insertcrossingparticles.c

```
#include <stdio.h>
#include <math.h>
#include "definitions.h"
#include "globals.h"
#include "prototypes.h"

/*Written by: David Lee
 *      Date: 05 July 1994
 */
void InsertCrossingParticles()
{
    int i,j,numparts,pi;
    int redo,fails;
    double dx,dy,dz,r2;
    /* # particles in transition and next cell */
    numparts=cell[mb]+cell[mb+1];
    for (i=1;(ToMd[i] != -1)&&(i<TRPARTS);i++){
        y[ToMd[i]] = URandom(&seed)*Side;
        z[ToMd[i]] = URandom(&seed)*Side;
        fails=0;
        do {

            redo=0;

            for (j=1;j<=numparts;j++){
                /*calculate distance*/
                pi=Xref[indx[mb]+j-1];
                if (pi != ToMd[i]) {
                    dx= x[pi]-x[ToMd[i]];
                    /*
                    dx=dx-L*anint(dx/L); NOT USED IN DUAL because of Walls */
                    dy= y[pi]-y[ToMd[i]];
                    dy=dy-Side*anint(dy/Side);
                    dz= z[pi]-z[ToMd[i]];
                    dz=dz-Side*anint(dz/Side);
                    r2= dx*dx+dy*dy+dz*dz;
                    /*if < Rc then reassign positions */
                    if (r2 < Rcsqrd){
                        redo=1; /*set recheck flag */
                        y[ToMd[i]] = URandom(&seed)*Side;
                        z[ToMd[i]] = URandom(&seed)*Side;
                        fails++;
                    }
                    if (redo) break;
                }
            }
        }
    }
}
```

```
    }while ( (redo!=0)&&(fails<10) );  
    if (fails==10) Error("10 failures in Transition Cell.");  
  }  
}
```

A.13 mds.c

```
/*Molecular Dynamics Simulation of a Couette Cell.
 *      Uses Velocity-Verlet algorithm for
 *      integrating equations of motion.
 *
 *Written by:  David Lee
 *      Date:  March 15, 1993
 * Revisions:  31 March 94  DTL
 *              removed KE calculations from start,finishmove
 *              removed calculations of total momentum
 *              corrected value for k/3M   (TCOEFF)
 *              14 April 94  DTL
 *              changed nsamps from a local static variable
 *              in Sampler to a global variable
 */
#include <math.h>
#include <stdlib.h>
#include <stdio.h>

#include "definitions.h"
#include "prototypes.h"
#ifdef DEBUG
double brutepot();
#endif

/*Define global variables here.
 * Positions, velocities, distance matrix,
 * accelerations
 * Number of: particles, cells, timesteps done,
 * timesteps to do, walls
 */
double  *x,*y,*z;
double  *vx,*vy,*vz;
double  *ax,*ay,*az;
double  *deltav,*vwall,*xwall,*xcell;
double  *dvtot,*dverr;
double  T,tau,tausqrd;
int      Np,Nc,Nt,timestep,seed;
int      *Xref,*cell,*indx,*offwall,*strikes;
int      ToMd[TRPARTS],ToDs[TRPARTS];
double  *ave_n,*ave_u,*ave_v,*ave_w,*ave_T,*bulk;
double  *var_n,*var_u,*var_v,*var_w,*var_T,nsamps;
double  L,Side,Rc,Rcsqrd;
double  posorder,kinetic,potential,vx2,vy2,vz2;
double  totcol,totsel;
```



```

/*
 *the following variables describe the
 *start and stop of mds & dsmc regions
 */
int db,de,mb,me;
double split;

main()
{
    /*file variables*/
    int i,Nread;
    FILE *params,*velocity,*position;

    /*simulation parameter variables*/
    double Mach,mpv,time;

    /*Simulation Variables */
    double mfp,volume;
    int startstep;

    printf("MD simulation.\n");
    #ifdef PERIODIC
        printf(
            "THIS RUN IS WITH PERIODIC BOUNDARY CONDITIONS.\n\n");
    #endif
    #ifdef SPECULAR
        printf(
            "THIS RUN IS WITH SPECULAR WALLS in X, PERIODIC BC's in y,z.\n\n");
    #endif
    #ifdef THERMAL
        printf(
            "THIS RUN IS WITH THERMAL WALLS in X, PERIODIC BC's in y,z.\n\n");
    #endif
    #ifdef POISE
        printf(
            "Acceleration is %lg in the negative z-direction.\n",ACCEL);
    #endif

    params=OpenFile("data.params","r");
    velocity=OpenFile("data.velocity","r");
    position=OpenFile("data.position","r");

    Nread=fscanf(params,"%ld",&seed);
    if (Nread!=1) Error("Error reading seed.");
    seed = (seed>=0) ? -1*seed : seed;
    Nread=fscanf(params,"%ld %ld %ld %ld",&Np,&Nc,&Nt,&timestep);
    if (Nread!=4) Error("Error Reading Number Parameters.");
    Nread=fscanf(params,"%lg %lg %lg %lg %lg",&L,&Side,&Rc,&T,&Mach);

```

```

if (Nread!=5) Error("Error Reading Physical Parameters 1.");
Nread=fscanf(params,"%lg %lg",&tau,&time);
if (Nread!=2) Error("Error Reading Physical Parameters 2.");

InitializeGlobals(velocity,position);

/*set the values for the start and end of mds region here*/
db=0;      mb=1;
de=0;      me=Nc; split = 0.0;

tausqrd=tau*tau;
mpv=sqrt(2.0*BOLTZ*T/MASS);
Rcsqrd=Rc*Rc;
vwall[2]=Mach*sqrt(5.0/3.0*BOLTZ*T/MASS);
vwall[1]=-vwall[2];
volume=L*Side*Side;
mfp=volume/(sqrt(2.)*PI*DIAM*DIAM*Np);

printf("Np=%ld Nc=%ld Nt=%ld timestep=%ld\n",
      Np,Nc,Nt,timestep);
printf("L=%lg Side=%lg Volume=%lg T=%lg Mach=%lg\n",
      L,Side,volume,T,Mach);
printf("tau=%lg time=%lg mpv=%lg\n",tau,time,mpv);
printf("System length is %lg. Mean free pathlength is %lg.\n",
      L,mfp);
printf("Wall velocity is %lg m/s.\n",vwall[2]);

startstep=timestep;

/*calculate first forces*/
/*Sort particles by cells */
Sorter();
CalcForces();

/*iterate*/
for (;timestep<=Nt;timestep++){
    if (timestep%100==0) posorder=Sampler(startstep);
    if ((timestep%WRITEPERIOD == 0)|| (timestep==Nt) ){
        printf("%ld of %ld steps done. %lg collisions %ld strikes\n",
            timestep,Nt,totcol,strikes[1]+strikes[2]);
        StoreGlobals(time,Mach,startstep);
    }

    StartMove(mpv);
    Sorter();
    CalcForces();

    FinishMove();

```

```
        time+=tau;
    }
    FreeGlobals();
    return(0);
}
```

A.14 mover.c

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "prototypes.h"
#include "definitions.h"
#include "globals.h"

/*Mover-- Moves the particles in DMSC
 * xwall[i]=position of walls 1<=i<=3
 * vwall[i]=velocity of walls 1<=i<=3
 * tau=timestep
 * deltav[i]=change in velocity 1<=i<=Np
 */

void Mover(mpv)
    double mpv;
{
    int i,flag,first_index,last_index,pi;
    int last_in_list,dummy=0;
    double x_old,v_parll,dtr;
    double direction[3],phi,rnd;

    direction[1]=1.0;    direction[2]=-1.0;
    last_in_list=1;
    for (i=1;i<=2;i++)
        deltav[i]=0.0;

#ifdef PERIODIC
    flag=0; /* for periodic bc's only */
#endif
    first_index=indx[db]; /*index into Xref of first particle*/
    last_index =indx[de]+cell[de]-1;

    for (i=first_index;i<=last_index;i++){
        pi=Xref[i];
#ifdef DUAL
        if (x[pi]>split)
            Error("MD particle acted on by Mover.");
#endif
        x_old=x[pi];
        x[pi] += vx[pi]*tau;    /*update position*/
#ifdef POISE
        vz[pi] -= ACCEL*tau;
#endif
#ifdef DUAL
        if (x[pi]>split){
```

```

    ToMd[last_in_list]=pi;
/*    printf("ToMd[%ld]=%ld %lg %lg %lg\n",
    last_in_list,pi,vx[pi],vy[pi],vz[pi]); */
    last_in_list++;
    if (last_in_list >= TRPARTS)
        Error("Length of transition particle list exceeded.");
    }
#endif

#ifdef PERIODIC
    if (x[pi] <= 0.0 )        /*check for collision*/
        x[pi] += L;          /*for periodic bc's only */
    else if (x[pi] >= L )
        x[pi] -= L;          /* for periodic bc's only */
#else
    if ( x[pi] <= 0.0)
        flag = 1;
    else if ( x[pi] >= L )
#ifdef DUAL
        Error("DSMC particle struck Right Wall!");
#else
        flag = 2;
#endif
#endif
    else
        flag = 0;
        if (flag>0){
            strikes[flag]++;
            deltav[flag]-=vy[pi];    /*record velocity change*/
#ifdef SPECULAR
            vx[pi]=-1.0*v_x[pi];
#endif
#ifdef THERMAL
            phi=TWOPI*URandom(&seed);
            v_parll=sqrt(-log(1.0-URandom(&seed)))*mpv;
            vx[pi]= direction[flag]*sqrt(-log(1.0-URandom(&seed)))*mpv;
            vy[pi]=v_parll*sin(phi)+vwall[flag];
            vz[pi]=v_parll*cos(phi);
#endif
            /*calc time of flight after leaving wall*/
            dtr=tau*(x[pi]-xwall[flag])/(x[pi]-x_old);
            x[pi]=xwall[flag]+vx[pi]*dtr;
            deltav[flag]+=vy[pi];
        }
#endif
    if ((x[pi]< 0)||x[pi] > L)){
        printf("Error: Particle out of system.  x=%lg\n",x[pi]);
        exit(1);
    }
}

```

```
    }  
    return;  
}
```

A.15 sampler.c

```
/*Revisions: 19 March 94 DTL
*           removed resetting of averages
*           corrected calculation of Temperature
*           use average u,v,w for bulk flow in
*           Temperature calculations
*           14 April 94 DTL
*           changed nsamps from a local static variable
*           in Sampler to a global variable
*/
#include <math.h>
#include "definitions.h"
#include "globals.h"
#include "prototypes.h"

double Sampler(startstep)
    int startstep;
{
    int i,jcell,tmp;
    double *sum_n,*sum_vx,*sum_vy,*sum_vz,*sum_v2,temp;
    double oldavg,positionalorder,newtemp;

    sum_n=DoubleVector(1,Nc); sum_vx=DoubleVector(1,Nc);
    sum_vy=DoubleVector(1,Nc); sum_vz=DoubleVector(1,Nc);
    sum_v2=DoubleVector(1,Nc);
    vx2=0.0;    vy2=0.0;    vz2=0.0;

    positionalorder=0.0;
    for (i=1;i<=Np;i++){
        vx2+=vx[i]*vx[i];
        vy2+=vy[i]*vy[i];
        vz2+=vz[i]*vz[i];
        tmp=(int)floor((double)Nc*x[i]/L)+1;
        jcell= (tmp<=Nc) ? tmp:Nc;
        if (jcell<1 || jcell >Nc){
            printf("sampler error:  jcell=%ld \n",jcell);
            exit(1);
        }
        sum_n[jcell]++;
        sum_vx[jcell]+=vx[i];
        sum_vy[jcell]+=vy[i];
        sum_vz[jcell]+=vz[i];
        sum_v2[jcell]+=vx[i]*vx[i]+vy[i]*vy[i]+vz[i]*vz[i];
        positionalorder +=cos(CP0*x[i])+cos(CP0*y[i])+cos(CP0*z[i]);
    }

    kinetic=KCOEFF*(vx2+vy2+vz2);
```

```

vx2 =KCOEFF*vx2;
vy2 =KCOEFF*vy2;
vz2 =KCOEFF*vz2;
positionalorder= positionalorder/(3.0*(double)Np);

nsamps+=1.0;
for (i=1;i<=2;i++){
    oldavg=deltav[1];
    dvtot[i] = oldavg + (deltav[i]-oldavg)/nsamps;
    dverr[i] = dverr[i];
}
for (i=1;i<=Nc;i++){
    if (sum_n[i] != 0.0){
        oldavg=ave_n[i];
        newtemp=sum_n[i];
        ave_n[i]+=(newtemp-oldavg)/nsamps;
        var_n[i]+=(newtemp-oldavg)*(newtemp-ave_n[i]);

        oldavg=ave_u[i];
        newtemp=sum_vx[i]/sum_n[i];
        ave_u[i]+=(newtemp-oldavg)/nsamps;
        var_u[i]+=(newtemp-oldavg)*(newtemp-ave_u[i]);

        oldavg=ave_v[i];
        newtemp=sum_vy[i]/sum_n[i];
        ave_v[i]+=(newtemp-oldavg)/nsamps;
        var_v[i]+=(newtemp-oldavg)*(newtemp-ave_v[i]);

        oldavg=ave_w[i];
        newtemp=sum_vz[i]/sum_n[i];
        ave_w[i]+=(newtemp-oldavg)/nsamps;
        var_w[i]+=(newtemp-oldavg)*(newtemp-ave_w[i]);

        oldavg=ave_T[i];
        temp=ave_u[i]*ave_u[i]+ave_v[i]*ave_v[i]+ave_w[i]*ave_w[i];
        newtemp=TCOEFF*(sum_v2[i]/sum_n[i]-temp);

        ave_T[i]+=(newtemp-oldavg)/nsamps;
        var_T[i]+=(newtemp-oldavg)*(newtemp-ave_T[i]);
    }
}
free((char *) (sum_v2+1));
free((char *) (sum_n+1));
free((char *) (sum_vx+1));
free((char *) (sum_vy+1));
free((char *) (sum_vz+1));
return positionalorder;
}

```


A.16 sorter.c

```
#include "prototypes.h"
#include <math.h>
#include "definitions.h"
#include "globals.h"

/*Sorter:  Sorts the particles into cells
 *        L=Length of system
 *        i=index to particle
 *        n=index to cell
 *        j=index to Xref
 */

void Sorter()
{
    int i,j,n,tmp;
    int *nx,*temp;
    nx=IntVector(1,Np);
    temp=IntVector(1,Nc);

    /*clear arrays associated with cells*/
    for (n=1;n<=Nc;n++){
        cell[n]=0;
        temp[n]=0;
    }

    for (i=1;i<=Np;i++){ /*find cell i(th) particle is in.*/
        tmp=(int)floor(Nc*x[i]/L)+1;
        nx[i]= (tmp<=Nc) ? tmp:Nc;    /*assure 1<=n,Nc */
        if (nx[i]<1 || nx[i]>Nc){
            (void)printf("sorter error i=%ld nx[i]=%ld\n",
                        i,nx[i]);
            (void)printf("x=%15.10lf   y=%15.10lf   z=%15.10lf\n",
                        x[i],y[i],z[i]);
            exit(1);
        }
        cell[nx[i]]++;    /*add one to cell(n)*/
    }
    indx[1]=1;

    /*First element points to Xref[1]*/
    for (n=2;n<=Nc;n++) indx[n]=indx[n-1]+cell[n-1];
    /*this gives Np+1 for indx[Nc] */

    for (i=1;i<=Np;i++){
        n=nx[i];
        j=indx[n]+temp[n];
    }
}
```

```

temp[n]++;
if (j<1 || j>Np){
    (void)printf("sorter error: temp=%ld index=%ld \n",
                temp[n],indx[n]);
    exit(1);
}
Xref[j]=i;
}

free((char *)(nx+1));
free((char *)(temp+1));
return;
}

```

A.17 startmove.c

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include "prototypes.h"
#include "definitions.h"
#include "globals.h"

void StartMove(mpv)
    double mpv;
{
    int i, pi, flag, offwall_tmp, first_index, last_index, last_in_list;
    double xo, yo, zo, v_parll, dtr, direction[3], phi;

    direction[0]=0.0; direction[1]=1.0; direction[2]=-1.0;
    last_in_list=1;
    /*update positions with velocity verlet-update in place*/

    first_index=indx[mb]; /*index into Xref of first particle*/
    last_index =indx[me]+cell[me]-1;
    for (i=first_index; i<=last_index; i++){
        pi=Xref[i];
        offwall[pi]=1;
        xo=x[pi];
        yo=y[pi];
        zo=z[pi];
        x[pi]+=tau*vx[pi]+0.5*tau*tau*ax[pi];
        y[pi]+=tau*vy[pi]+0.5*tau*tau*ay[pi];
        z[pi]+=tau*vz[pi]+0.5*tau*tau*(az[pi]-ACCEL);

#ifdef DUAL
        if (x[pi]<=split){
            todsmc++;
            /*Do not update velocity because next force not avail.*/
            continue;
        }
#endif
        vx[pi]+=0.5*tau*ax[pi];
        vy[pi]+=0.5*tau*ay[pi];
        vz[pi]+=0.5*tau*(az[pi]-ACCEL);

        /*check for collision with wall, place walls at 0, L */
#ifdef PERIODIC
        /*Check for crossing periodic boundaries*/
        if (x[pi]<0) x[pi]+=L;
        if (x[pi]>=L) x[pi]-=L;
#else
        /*else
        if (x[pi]<0) x[pi]=0;
        if (x[pi]>=L) x[pi]=L;
        if (y[pi]<0) y[pi]=0;
        if (y[pi]>=L) y[pi]=L;
        if (z[pi]<0) z[pi]=0;
        if (z[pi]>=L) z[pi]=L;
        */
#endif
    }
}
```

```

        if ( x[pi] <= 0.0)
#ifdef DUAL
            Error("MD Particle struck left Wall.\n");
#else
            flag = 1;
#endif
        else if ( x[pi] >= L )
            flag = 2;
        else
            flag = 0;

        if (flag!=0){
            strikes[flag]++;
            /*record velocity change--wall momentum sum*/
            deltav[flag]-=vy[pi];
            offwall[pi]=0;

            x[pi]=xwall[flag];
            y[pi]=y[pi]-(x[pi]-xwall[flag])*(yo-y[pi])/(xo-x[pi]);
            z[pi]=z[pi]-(x[pi]-xwall[flag])*(zo-z[pi])/(xo-x[pi]);

#ifdef SPECULAR
            vx[pi]=-1.0*vx[pi];
#endif
#ifdef THERMAL
            phi=TWOPI*URandom(&seed);
            v_parll=sqrt(-log(1.0-URandom(&seed)))*mpv;
            vx[pi]= direction[flag]*sqrt(-log(1.0-URandom(&seed)))*mpv;
            vy[pi]=v_parll*sin(phi)+vwall[flag];
            vz[pi]=v_parll*cos(phi);
#endif
            deltav[flag]+=vy[pi];
        }
    }
#endif
    if (y[pi]<0) y[pi]+=Side;
    if(y[pi]>=Side) y[pi]-=Side;
    if (z[pi]<0) z[pi]+=Side;
    if(z[pi]>=Side) z[pi]-=Side;
    if ((x[pi]<0)||(x[pi]>L)||
        (y[pi]<0)||(y[pi]>=Side)||
        (z[pi]<0)||(z[pi]>=Side)){
        printf("ERROR:::particle %ld\n",i);
        printf("%lg %lg %lg %lg\n",xo,x[pi],y[pi],z[pi]);
        printf("%lg %lg %lg\n",vx[pi],vy[pi],vz[pi]);
        printf("%lg %lg %lg\n\n",ax[pi],ay[pi],az[pi]);
        exit(1);
    }
    ax[pi]=0.0;

```

```
        ay[pi]=0.0;  
        az[pi]=0.0;  
    }  
    return;  
}
```

A.18 storeglobals.c

```
#include <stdlib.h>
#include <stdio.h>

#include "prototypes.h"
#include "definitions.h"
#include "globals.h"

#define SEEK_SET 0
void StoreGlobals(time,Mach,startstep)
    double Mach,time;
    int startstep;
{
    int i,j;
    char varlist[25][25];
    double force1,force2;

    static FILE *Avg_n,*Avg_u,*Avg_v,*Avg_w,*Avg_T,*XCell;
    static FILE *Var_n,*Var_u,*Var_v,*Var_w,*Var_T;
    static FILE *params,*velocity,*position,*KE;

    if (startstep==timestep){
        /* open output files */
        (void)strcpy(varlist[0],"time");
        (void)strcpy(varlist[1],"positional order");
        (void)strcpy(varlist[2],"total collisions");
        (void)strcpy(varlist[3],"total collision pairs");
        (void)strcpy(varlist[4],"wall_force_1");
        (void)strcpy(varlist[5],"wall_force_2");

        (void)strcpy(varlist[6],"kinetic");
        (void)strcpy(varlist[7],"potential");
        (void)strcpy(varlist[8],"vx2");
        (void)strcpy(varlist[9],"vy2");
        (void)strcpy(varlist[10],"vz2");
        (void)strcpy(varlist[11],"nsamps");

        KE=OutputFileInit("misc.stats",varlist,12);
        Avg_n=OpenFile("Avg_n","w");
        Avg_u=OpenFile("Avg_u","w");
        Avg_v=OpenFile("Avg_v","w");
        Avg_w=OpenFile("Avg_w","w");
        Avg_T=OpenFile("Avg_T","w");
        Var_n=OpenFile("Var_n","w");
        Var_u=OpenFile("Var_u","w");
        Var_v=OpenFile("Var_v","w");
        Var_w=OpenFile("Var_w","w");
```

```

    Var_T=OpenFile("Var_T","ws");
    params=OpenFile("new.data.params","w");
    position=OpenFile("new.data.position","w");
    velocity=OpenFile("new.data.velocity","w");
}

if (fseek(params,SEEK_SET,(int)0)) {
    (void)fprintf(stderr,"Could not reposition parameter file.");
    exit(1);
}
if (fseek(position,SEEK_SET,(int)0)) {
    (void)fprintf(stderr,"Could not reposition position file.");
    exit(1);
}
if (fseek(velocity,SEEK_SET,(int)0)) {
    (void)fprintf(stderr,"Could not reposition velocity file.");
    exit(1);
}

force1=MASS*deltav[1]/(tau*WRITEPERIOD);
force2=MASS*deltav[2]/(tau*WRITEPERIOD);
deltav[1]=0.0;
deltav[2]=0.0;

fprintf(KE,
    "%-10.3le\t%-10.3le\t%-10.3le\t%-10.3le\t%-10.3e\t%-10.3e\t",
    time,posorder,totcol,totsel,force1,force2);

fprintf(KE,
    "%-12.7le\t%-12.7le\t%-10.3le\t%-10.3le\t%-10.3le\t%-13.6le\n",
    kinetic,potential,vx2,vy2,vz2,nsamps);

for (j=1;j<=Np;j++){
    fprintf(position,"%-10.3le\t%-10.3le\t%-10.3le\n",
        x[j],y[j],z[j]);
    fprintf(velocity,"%-10.3le\t%-10.3le\t%-10.3le\n",
        vx[j],vy[j],vz[j]);
}
fprintf(params,"%ld\n",seed);
fprintf(params,"%ld %ld %ld %ld\n",Np,Nc,Nt,timestep+1);
fprintf(params,"%-10.3le %-10.3le %-10.3le %-10.3le %-10.3le\n",
    L,Side,Rc,T,Mach);
fprintf(params,"%-10.3le %-10.3le",tau,time);

if ( fflush(position)||
    fflush(KE)||
    fflush(velocity) ||

```

```

        fflush(params) ){
    fprintf(stderr,"Write Error\n");
    exit(1);
}

for (i=1;i<=Nc;i++){
    (void)fprintf(Avg_n,"%-10.3le\t",ave_n[i]);
    (void)fprintf(Avg_u,"%-10.3le\t",ave_u[i]);
    (void)fprintf(Avg_v,"%-10.3le\t",ave_v[i]);
    (void)fprintf(Avg_w,"%-10.3le\t",ave_w[i]);
    (void)fprintf(Avg_T,"%-10.3le\t",ave_T[i]);
    (void)fprintf(Var_n,"%-10.3le\t",var_n[i]);
    (void)fprintf(Var_u,"%-10.3le\t",var_u[i]);
    (void)fprintf(Var_v,"%-10.3le\t",var_v[i]);
    (void)fprintf(Var_w,"%-10.3le\t",var_w[i]);
    (void)fprintf(Var_T,"%-10.3le\t",var_T[i]);
}
fprintf(Avg_n,"\n\n");
fprintf(Avg_u,"\n\n");
fprintf(Avg_v,"\n\n");
fprintf(Avg_w,"\n\n");
fprintf(Avg_T,"\n\n");
fprintf(Var_n,"\n\n");
fprintf(Var_u,"\n\n");
fprintf(Var_v,"\n\n");
fprintf(Var_w,"\n\n");
fprintf(Var_T,"\n\n");

if (fflush(Avg_n) ||
    fflush(Avg_u) ||
    fflush(Avg_v) ||
    fflush(Avg_w) ||
    fflush(Avg_T) ||
    fflush(Var_n) ||
    fflush(Var_u) ||
    fflush(Var_v) ||
    fflush(Var_w) ||
    fflush(Var_T) ){
    fprintf(stderr,"Write Error.\n");
    exit(1);
}

if (timestep>=Nt){
    CloseFile(Avg_n,"Avg_n");
    CloseFile(Avg_u,"Avg_u");
    CloseFile(Avg_v,"Avg_v");
    CloseFile(Avg_w,"Avg_w");
    CloseFile(Avg_T,"Avg_T");

```



```
    CloseFile(Var_n,"Var_n");
    CloseFile(Var_u,"Var_u");
    CloseFile(Var_v,"Var_v");
    CloseFile(Var_w,"Var_w");
    CloseFile(Var_T,"Var_T");
    CloseFile(params,"new.data.params");
    CloseFile(velocity,"new.data.velocity");
    CloseFile(position,"new.data.position");
    CloseFile(KE,"misc.stats");
}

return;
}
```

A.19 urandom.c

```
#include <stdio.h>
#include "prototypes.h"

#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-14
#define RNMX (1.0-EPS)

double URandom(idum)
    long *idum;
{
    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*(*idum - k*IQ2)-k*IR2;
    if (idum2 < 0)
        idum2 += IM2;
    j = iy/NDIV;
    iy = iv[j] - idum2;
    iv[j] = *idum;
    if (iy < 1)
        iy += IMM1;
    temp = AM*iy;
    if (temp > RNMX)
        return RNMX;
    else
        return temp;
}

```

A.20 utilities.c

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include "definitions.h"
#include "prototypes.h"
#include "globals.h"

/* Allocates an nh-nl dimensional vector with starting index nl*/
int *IntVector(nl,nh)
    int nl,nh;
{
    int *p,i;
    p=(int *)malloc( (unsigned)(sizeof(int)*(nh-nl+1) ) );
    if (!p) Error("Memory allocation Failure:  IntVector.");
    p-=nl;
    for (i=nl;i<=nh;i++) p[i]=0;
    return(p);
}

double *DoubleVector(nl,nh)
    int nl,nh;
{
    double *p;
    int i;

    p=(double *)malloc( (unsigned)(sizeof(double)*(nh-nl+1) ) );
    if (!p) Error("Memory allocation Failure:  DoubleVector.");
    p-=nl;
    for (i=nl;i<=nh;i++) p[i]=(double)0.0;
    return(p);
}

double **DoubleMatrix(nl,nh,ml,mh)
    int nl,nh,ml,mh;
{
    double **mat;
    int i,j;

    mat=(double **)malloc((unsigned)((nh-nl+1)*sizeof(double*)));
    if (!mat) Error("Memory Allocation Failure:  DoubleMatrix");
    mat-=nl;
    for (i=nl;i<=nh;i++){
        mat[i]=(double *)malloc((unsigned)((mh-ml+1)*sizeof(double)));
        if (!mat[i]) Error("Memory Allocation Failure:  DoubleMatrix");
        mat[i]-=ml;
    }
}
```

```

    }
    for (i=nl;i<=nh;i++) for (j=ml;j<=mh;j++) mat[i][j]=(double)0.0;
    return(mat);
}

void FreeGlobals()
{
    free( (char *)(x+1));
    free( (char *)(y+1));
    free( (char *)(z+1));
    free( (char *)(vx+1));
    free( (char *)(vy+1));
    free( (char *)(vz+1));
    free( (char *)(ax+1));
    free( (char *)(ay+1));
    free( (char *)(az+1));
    free( (char *)(deltav+1));
    free( (char *)(vwall+1));
    free( (char *)(xwall+1));
    free( (char *)(Xref+1));
    free( (char *)(cell+1));
    free( (char *)(indx+1));
    free( (char *)(ave_n+1));
    free( (char *)(ave_T+1));
    free( (char *)(ave_u+1));
    free( (char *)(ave_v+1));
    free( (char *)(ave_w+1));
}

FILE *OutputFileInit(filename,varlist,nvars)
    char filename[200],varlist[][25];
    int nvars;
{
    int i;
    FILE *fileptr,*varfileptr;
    char varfilename[512];

    (void)strcpy(varfilename,filename);
    if (strlen(filename)>480)
        Error("Output filename too long.");
    (void)strcat(varfilename,".variables",10);
    /* open the output file and variable list file */
    if ( (fileptr=fopen(filename,"w"))==NULL )
        Error("Could not open %s.\n", filename);
    if ( (varfileptr=fopen(varfilename,"w"))==NULL )
        Error("Could not open %s.\n",varfilename);
    for (i=0;i<nvars;i++)
        (void)fprintf(varfileptr,"%s, ",varlist[i]);

```

```

    (void)fprintf(varfileptr,"\n");
    if (fclose(varfileptr)!=0)
        Error("Could not close %s.",varfilename);
    return fileptr;
}

/*---Error()---*/
void Error(mssg)
    char *mssg;
{
    printf("%s\n",mssg);
    exit(1);
}

```

Appendix B

Simulation Plots

B.1 Plots from DSMC Simulations

DSMC Simulation of an Isolated System

DSMC Simulation of an Isothermal System

DSMC Simulation of System with a Thermal Gradient

DSMC Simulation of a Couette Cell

DSMC Simulation of a Poiseuille Flow

B.2 Plots from MD Simulations

MD Simulation of an isolated System

MD Simulation of an Isothermal System

MD Simulation of System with a Thermal Gradient

MD Simulation of a Couette Cell

MD Simulation of a Poiseuille Flow

B.3 Plots from DSMC–MD Simulations

DSMC-MD Simulation of an Isolated System

DSMC-MD Simulation of an Isothermal System

DSMC-MD Simulation of System with a Thermal Gradient

DSMC-MD Simulation of a Couette Cell

DSMC—MD Simulation of a Poiseuille Flow