2004

Measurement bias in molecular simulations induced by fluctuations / Martin W. Tysanner

Martin W. Tysanner
San Jose State University

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

Recommended Citation

Tysanner, Martin W., "Measurement bias in molecular simulations induced by fluctuations / Martin W. Tysanner" (2004). *Master's Theses*. 2688.
DOI: <https://doi.org/10.31979/etd.52qm-qxb7>
https://scholarworks.sjsu.edu/etd_theses/2688

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.

**MEASUREMENT BIAS IN MOLECULAR SIMULATIONS
INDUCED BY FLUCTUATIONS**

A Thesis

Presented to

The Faculty of the Department of Physics

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Martin W. Tysanner

December 2004

UMI Number: 1425490

INFORMATION TO USERS

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 bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send 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.

UMI Microform 1425490

Copyright 2005 by ProQuest Information and Learning Company.

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

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

© 2004

Martin W. Tysanner

ALL RIGHTS RESERVED

APPROVED FOR THE DEPARTMENT OF PHYSICS

Dr. Alejandro Garcia

Dr. Patrick Hamill

Dr. Ken Wharton

APPROVED FOR THE UNIVERSITY

ABSTRACT

MEASUREMENT BIAS IN MOLECULAR SIMULATIONS
INDUCED BY FLUCTUATIONS

by Martin W. Tysanner

Two general, common methods for measuring hydrodynamic variables like fluid velocity and temperature are widely considered to be equivalent. It is demonstrated, however, that although the two methods are equivalent at equilibrium, one yields small but systematically biased measurements in systems at the non-equilibrium steady state due to spatial correlations of fluctuations of hydrodynamic variables. The measurement bias can manifest in any non-equilibrium system where fluctuations are present. It can be physically meaningful, as for example fluid velocity or temperature in a system at thermal non-equilibrium, or computationally induced, as in cases where poor simulation algorithm design inadvertently causes fluctuations to become spatially correlated. Examples of both are investigated; analytical expressions are obtained and system characteristics that reduce or magnify the error are determined. Although all investigations employ direct simulation Monte Carlo (DSMC) computations, the same effects will be observed with other particle-based simulation methods such as molecular dynamics.

ACKNOWLEDGEMENTS

I would like to thank my wife, Anne, and children, Valerie and Isaac, for their patience, sacrifice, and supportiveness during the time leading up to and including the preparation of this thesis. I also thank them for their forgiveness of the many hours when they could not use our faster computer because it was busy running simulations. I further wish to thank Prof. Alejandro Garcia, whose patience and willingness to both share his knowledge and guide me with a light touch have helped increase my love of physics and develop my own research and technical writing style. He has truly been an outstanding advisor and teacher, as well as a friend.

CONTENTS

CHAPTER

1	INTRODUCTION	1
1.1	Motivation	1
1.2	System Overview	3
1.3	Anomalous Fluid Flow in a Closed System	6
1.4	Bias in Temperature Measurement at Non-equilibrium	13
1.5	Anomalous Fluid Flow in an Equilibrium System with a Reservoir . .	14
1.6	Thesis Outline	20
2	BACKGROUND	21
2.1	Measurement of Fluid Velocity	21
2.2	Measurement of Fluid Velocity for Multiple Species	23
2.3	Measurement of Temperature	26
2.4	Dependence of Fluctuation Magnitudes on Particle Number	30
3	SIMULATION METHODOLOGY	38
3.1	Classification of Fluid Simulation Methods	38
3.1.1	Continuum versus particle methods	38

3.1.2	Continuum models	39
3.1.3	Molecular dynamics (MD)	40
3.1.4	Direct Simulation Monte Carlo (DSMC)	41
3.1.5	Lattice gases	42
3.1.6	Hybrid models	43
3.2	DSMC Method Details	43
3.2.1	Motivation	43
3.2.2	General comments	44
3.2.3	Movement phase	46
3.2.4	Collision Phase	47
3.2.5	Discretization of space	51
3.2.6	Discretization of time	52
4	SIMULATION SYSTEM	54
4.1	Idealizations and Simplifications	54
4.2	System Details	55
4.3	Simulation Sequence	59
4.4	Simulator Implementation Details	59
4.4.1	General structure of the simulator program	59
4.4.2	DSMC implementation	63
4.4.3	Measuring variances and covariances	64
4.5	Simulator Validation	66
4.5.1	Validation approach	66
4.5.2	Simulator validation results	70

5	ROLE OF FLUCTUATIONS IN FLUID VELOCITY MEASUREMENT	78
5.1	Introduction	78
5.2	Determining the Origin of the Anomalous Velocity	79
5.3	Analysis	90
5.4	Simulation Tests of the Analysis	94
6	ROLE OF FLUCTUATIONS IN TEMPERATURE MEASUREMENT	99
6.1	Theory	99
6.2	Simulation Results	101
6.3	Measurement Bias from Average of Instantaneous Temperatures . . .	107
7	FLUCTUATIONS AND RESERVOIR IMPLEMENTATION	111
7.1	Nature of the Anomalous Fluid Flow with Reservoirs	111
7.2	Characterization of the Effect	119
7.3	Discussion and Additional Simulations	130
8	GENERAL DISCUSSION AND CONCLUDING REMARKS	139
8.1	Summary of Key Results	139
8.2	Directions for Future Study	141
8.2.1	Analytic expressions for fluctuation covariances	141
8.2.2	Simulations of multi-species fluids	145
8.2.3	Two dimensional systems	148
8.2.4	Fluctuations in coupled closed systems with shared rigid walls	149
8.3	Additional Observations and Concluding Remarks	150

BIBLIOGRAPHY	156
---------------------	-----

APPENDIX

A SOME MOLECULAR SPEEDS	158
B MEAN RELATIVE VELOCITY OF PARTICLES IN A GAS	161
C LISTINGS OF DSMC SIMULATOR IMPLEMENTATION (C++)	165
C.1 Header File for Data Structures and Other Definitions	165
C.2 Simulator Initialization, Output and Exit Functions	168
C.3 DSMC Simulation Functions	187
C.4 Covariance Calculation and Output Functions	208
C.5 Random Deviate Generation Functions	211

FIGURES

Figure

1.1	Conceptual diagram of the simulated closed system	4
1.2	Conceptual diagram of the simulated system with one reservoir	4
1.3	Temperature profiles for temperature gradients and Couette flow . . .	7
1.4	Mean fluid velocity measured using Sample Averaging Measurement (SAM) for different temperature gradients	8
1.5	Mean fluid velocity for Couette flow measured using SAM for different shear velocity gradients	9
1.6	Mean fluid velocity measured using Cumulative Average Measurement (CAM) for different temperature gradients	11
1.7	Difference in fluid temperature measurements obtained by SAM and CAM	14
1.8	SAM fluid velocity for an equilibrium system at $T = 1$ with a reservoir at the left boundary and a specular wall at the right boundary	16
1.9	CAM fluid velocity for an equilibrium system at $T = 1$ with a reservoir at the left boundary and a specular wall at the right boundary	17
1.10	SAM fluid velocity for an equilibrium system at $T = 1$ with a reservoir at the left boundary and a specular wall at the right boundary	19

4.1	Conceptual diagram of a simulated closed system	55
4.2	Conceptual diagram of a simulated system with one reservoir	56
4.3	Example of a simulation specification text file	60
4.4	Graphical user interface for simulation result display and plotting . .	61
4.5	Simulation information display within the graphical user interface . .	62
4.6	SAM fluid velocity profiles for the cases of equilibrium, medium tem- perature gradient and large temperature gradient in a closed system .	68
4.7	Gas pressure profiles for the cases of equilibrium, medium, and large temperature gradients in a closed system	71
4.8	Particle number density profiles for equilibrium, medium, and large temperature gradients in a closed system	72
4.9	Particle number profiles from established results	72
4.10	Temperature profiles for cases of equilibrium, medium, and large tem- perature gradients in a closed system	73
4.11	Temperature profiles for closed system from established results	73
4.12	Temperature profiles for Couette flow in a closed system with equal temperature boundaries	75
4.13	Collision frequency profile in a closed Couette flow system with equal temperature boundaries	75
4.14	Covariance of particle number density $\langle \delta n_k \delta n \rangle$ at center of closed sys- tem for different temperature gradients	76
4.15	Covariance $\langle \delta n_k \delta n \rangle$ from established results	76
4.16	Covariance $\langle \delta n_k \delta \mathbf{J} \rangle$ of particle number density and fluid momentum at center of closed system for different temperature gradients	77

4.17	Covariance $\langle \delta n_k \delta \mathbf{J} \rangle$ from established results	77
5.1	SAM mean fluid velocity for different temperature gradients	79
5.2	SAM mean fluid velocity for different shear velocity gradients	80
5.3	SAM mean fluid velocity for two different random deviate generator seeds	82
5.4	SAM mean fluid velocity for different initial particle distributions . . .	84
5.5	SAM mean fluid velocity for different system relaxation times	85
5.6	Fluid pressure profile for a closed system with a large temperature gradient	86
5.7	CAM mean fluid velocity for cases of equilibrium, medium and large temperature gradients	88
5.8	SAM mean fluid velocity for a closed system with a large temperature gradient and with wall temperatures reversed	89
5.9	Effect of time step on SAM mean fluid velocity profile	90
5.10	Effect of total particle number on SAM mean fluid velocity profile . . .	91
5.11	Comparison of theory and simulation for the effect of particle number on SAM mean fluid velocity	95
5.12	Relationship between measured density-momentum covariance and SAM mean fluid velocity	96
5.13	Relationship between SAM mean fluid velocity, variance in number density, and density-momentum covariance	97
6.1	Difference between SAM and CAM temperature profiles for an equi- librium, closed system with thermal walls	102

6.2	SAM-CAM temperature difference profile in a closed system with thermal walls and a medium temperature gradient	104
6.3	SAM-CAM temperature difference profile for medium gradient closed system without contributions from equilibrium fluctuations	105
6.4	Relationship between SAM-CAM temperature difference, variance in number density, and density-momentum covariance	106
6.5	Comparison of two definitions of the SAM temperature with the unbiased CAM temperature for reference	108
7.1	SAM mean fluid velocity for an equilibrium system with a constant generation rate reservoir and a specular wall at opposite boundaries .	112
7.2	Particle density profile for the equilibrium, constant- generation reservoir system	113
7.3	Temperature profile for the equilibrium, constant- generation reservoir system	113
7.4	Effect of particle number on SAM mean fluid velocity in the constant-generation reservoir system	115
7.5	Effect of Poisson distributed generation rate on SAM mean fluid velocity in the equilibrium reservoir system	117
7.6	Histogram of particle flux at the boundary between the system and a constant-generation reservoir in the equilibrium system	118
7.7	Relationship between density-momentum covariance and SAM mean fluid velocity for the constant-generation reservoir system	120
7.8	Relationship between SAM-CAM difference and SAM mean fluid velocity in the constant-generation reservoir system	121

7.9	Effect of reservoir depth on equilibrium system temperature profile in the constant-generation reservoir system	123
7.10	Effect of reservoir depth on equilibrium system temperature profile in the Poisson-generation reservoir system	124
7.11	Effect of time step on SAM mean fluid velocity profile in the constant-generation reservoir system	125
7.12	Effect of time step on histogram of particle flux at system-reservoir boundary in a constant-generation reservoir system	126
7.13	Histogram of particle flux at system-reservoir boundary in a Poisson-generation reservoir system	127
7.14	Effect of time step on histogram of particle flux in a Poisson-generation reservoir system	128
7.15	Effect of particle number on histogram of particle flux in a constant-generation reservoir system	129
7.16	Effect of large increase in reservoir depth on histogram of particle flux a in a constant-generation reservoir system	131
7.17	Effect of reservoir shallowness on histogram of particle flux in a constant-generation reservoir system	132
7.18	Effect of reservoir depth on SAM mean fluid velocity in a Poisson-generation reservoir system	133
7.19	Effect of medium increase in reservoir depth on histogram of particle flux in a constant-generation reservoir system	136
8.1	Histogram of particle flux at a thermal wall of an equilibrium closed system	143

8.2	Histograms comparing entry rates of particles from opposite sides of an interior cell within a closed system with a temperature gradient . .	145
8.3	Histograms comparing rates at which particles enter and leave an in- terior cell through the same boundary at thermal non-equilibrium . .	146
8.4	Histogram of particle number in the interior cell	147

CHAPTER 1

INTRODUCTION

1.1 Motivation

Fluctuations are transient, stochastic deviations from average behavior. In fluids, fluctuations of hydrodynamic or thermodynamic variables are usually very small in magnitude when compared to the mean values of those variables, and consequently they are usually ignored when analyzing or simulating the behavior of a physical system. Yet, there are entire classes of phenomena where fluctuations play an important or even primary role.

For example, there are at least two interesting regimes where fluctuations become more important. First, if the de Broglie wavelength of particles (e.g., photons) passing through a fluid is on the order of the spatial extent of a fluctuation, the particle motion may be affected in a measurable way (e.g., by light scattering). One example is the asymmetric Brillouin peaks observed in light scattering for a fluid under a temperature gradient [1, 2]. Second, if the fluid is a sufficiently dilute gas, or alternatively if the gas is less dilute but the system size is sufficiently small, a relatively small number of particles will be present and this will amplify the relative statistical fluctuations. Nanoscale devices provide one important class of examples [3]; the study of fluctuations in fluids driving Feynman ratchets and Brownian motors is

another [4]. This work focuses on the regime where the system dimensions are small compared to the average distance between molecules.

Fluctuations can affect the measurement of hydrodynamic variables. Temperature measurement provides an illustrative example. Defining temperature in terms of molecular translational kinetic energy requires some care due to the instantaneous fluctuations of the center of mass velocity. Additional, subtle issues exist, partly because temperature is unambiguously defined only at equilibrium [5, 6]. Some of the subtleties of temperature measurement will be covered in more depth in Chapter 6.

In the computational physics arena, molecular simulations may exaggerate the effects of fluctuations. For example, in simulations employing the Direct Simulation Monte Carlo (DSMC) method, one simulation particle may represent many physical particles, and as a result the relatively small number of simulation particles may artificially magnify the fluctuations in hydrodynamic or thermodynamic variables by many orders of magnitude over what would be observable in the corresponding physical system.

Three measurable anomalies that can arise in molecular simulations of fluids due to a failure to consider fluctuations are presented and analyzed in this thesis. All three may lead to bias in measurements of hydrodynamic variables. One of these is a bias in fluid velocity measurement that can occur in systems far from equilibrium. In fact, this bias reveals a non-obvious, previously unexplored ambiguity in the definition of fluid velocity that only becomes apparent in a steady-state system at thermal non-equilibrium. A second is a bias in temperature measurement, which, analogous to the case of fluid velocity measurement, reveals an ambiguity in the definition of temperature. The third is a strictly computational problem that can

arise in simulations of systems that require particle reservoirs; here, certain reservoir algorithms may inadvertently introduce nonphysical correlations between fluctuating variables, and this in turn can lead to measurement anomalies, even at equilibrium.

The remainder of this introduction gives a brief overview of the primary results. In order to provide a context, a brief description of the simulation system used in the investigations is provided first; a more detailed system description will appear in Chapter 4.

1.2 System Overview

All investigations described in this thesis employed a simple, rectangular, essentially one-dimensional system such as that shown in Fig. 1.1 for a closed system or Fig. 1.2 for a system with one particle reservoir. All simulations used the DSMC method, described in Section 3.2, to model a dilute gas. Along one axis, denoted the x -axis in our convention, the variables of interest were sampled and their profiles were generated.

The system had configurable boundaries at the two ends of the system along the x axis. In the y - and z -directions between the walls, the boundaries were periodic. At a periodic boundary, a particle leaving the system immediately reappeared at the opposite side of the system, conserving energy and momentum.

Each boundary for the x -axis could be configured independently in any of the following ways:

- **Thermal wall.** This allowed imposing a temperature gradient across the system in the x -direction. A constant velocity could also be imparted to a thermal wall in the y -direction to allow imposing a shear velocity gradient (i.e., Couette flow) in the x -direction.

Figure 1.1: Conceptual diagram of the simulated closed system. The two walls are perpendicular to the x axis, and they allow imposing a temperature gradient or shear velocity gradient. Boundaries are periodic in the y and z directions. Points represent individual fluid molecules; simulations typically used thousands of particles.

Figure 1.2: Conceptual diagram of the simulated system with one reservoir. The reservoir at the left boundary is shown as two cells deep; the right boundary is a specular wall. Boundaries are periodic in the y and z directions.

- **Specular wall.** A specular wall merely reversed v^x , the particle's x -velocity, with no other effect.
- **Periodic boundary.** The wall was removed; particles reaching the boundary immediately left the system and re-entered at the opposite boundary with the same velocity.
- **Reservoir.** The wall was replaced by a simulated particle reservoir of infinite capacity. The temperature, mean particle density and flow velocity could be set independently for each reservoir.

The system was closed for all cases except when a reservoir was present. It was isolated only if both x boundaries were specular or periodic.

A comment about measurement units used in this thesis may help relate the values shown on various plots to values that are typical of common experience. All units are dimensionless; either they are normalized or they are calculated from normalized quantities. Their magnitudes generally bear little similarity to those often encountered, but conversion to conventional units merely requires a multiplication by the appropriate factor. For example, a particle mass $m = 1$ is typically used; if the unit mass is to represent an atom of ${}^4\text{He}$ the conversion factor is $6.6 \times 10^{-24}g$. The reference temperature $T = 1$ corresponds to the reference temperature in conventional units (e.g., a room temperature of $298K$) rather than $1^\circ C$ or $1K$. The reference for fluid velocity is the speed of sound, but this is a derived quantity which is determined from the assigned values of temperature, molecular mass and fluid density, all normalized and dimensionless, together with a conveniently assigned dimensionless value of Boltzmann's constant ($k_B = 0.5$ in all of our simulations).

1.3 Anomalous Fluid Flow in a Closed System

In molecular simulations of fluids, two scenarios are so simple and have been so thoroughly investigated and understood that they are useful as simple test cases for new simulation programs. The first is the case where thermal walls on opposite sides of a closed system impose a temperature gradient on the fluid. In a rectangular system these walls are parallel; in a cylindrical system the walls are concentric. The second case is Couette flow in a closed system, where the fluid is subjected to shear forces. In a rectangular system the shear forces come from parallel, oppositely moving thermal walls; in the cylindrical system the walls take the form of counter-rotating rings. The investigations considered here employed only rectangular systems.

We start by considering the temperature profiles for the temperature gradient and Couette flow cases in a closed system like that of Fig. 1.1. Fig. 1.3 shows changes in temperature across the system for different gradients. For reference, we also show the equilibrium case. These profiles match expectations completely.

Next we consider fluid velocity measurements for the temperature gradient and Couette flow scenarios in the same system. Clearly, because the system is closed and at the steady state, the fluid must have zero net velocity in the x - and z -directions. For the case where the walls themselves have zero velocity the net fluid velocity must be zero in the y -direction as well. There will likely be measurable fluctuations about zero velocity due to limited numbers of particles; additional fluctuations will arise due to the limited number of samples used to calculate the mean velocity. We will show later (Section 2.4) that the expected mean square velocity fluctuation is proportional to $1/\langle N \rangle$, where $\langle N \rangle$ is the mean number of particles participating in a sample.

Figure 1.3: Temperature profiles for simulations involving temperature gradients and Couette flow; the equilibrium case where $T_L = T_R = 1$ is shown for reference. The left wall temperature and shear velocity are T_L and v_L , respectively; for the right wall these quantities are T_R and v_R . For the Couette flow scenarios, the temperature is highest at the center due to viscous heating.

Figure 1.4: Sample averaging measurement of mean fluid velocity, $\langle u^x \rangle_s$ (labelled simply as $\langle u \rangle_s$ on the plot), for different temperature gradients. Note the apparent net velocity, implying an anomalous flow from left to right that reaches a maximum at the center of the system. Evidently, increasing the number of samples results in a smoother plot but does not change the magnitude of the effect.

Fig. 1.4 shows the fluid velocity profile for the closed system with a large temperature gradient but no shear velocity gradient. Surprisingly, the measured mean velocity across the system was nonzero, in direct conflict with the conservation of mass that we require for a closed system at the steady state. The mean fluid velocity was from left to right, reaching a maximum at the center of the system. This

velocity is statistically meaningful, as the error bars show. The error bars were estimated using results from statistical mechanics [7]. Increasing the number of samples gave a smoother plot but did not change the magnitude of the effect.

We repeated the velocity measurement for a closed, Couette flow system. Fig. 1.5 shows the measured profile. Again, an anomalous fluid velocity appeared, in direct conflict with our expectations. This time the flow was from the walls toward the center of the system where the net shear velocity was zero.

Figure 1.5: Sample averaging measurement of mean fluid velocity, $\langle u^x \rangle_s$, for Couette flow with different shear velocity gradients. Note the anomalous fluid flow from either side toward the center.

Since the measured, nonzero fluid velocities cannot occur in a closed system without violating established physical principles (e.g., conservation of mass), the most likely cause for the anomaly was an incorrectly functioning simulation program. Barring that, the next most likely suspect was a flaw in the simulation algorithm. These were indeed our two strongest candidate explanations for the anomalous flows, and we ran extensive tests to isolate the cause.

Part of the investigation involved careful examination of the method the simulator used to measure the fluid velocity. For both of Figs. 1.4 and 1.5 we had used a method we call Sample Averaging Measurement, or SAM, a method in common use by practitioners of molecular simulations. To determine whether a different method would produce the same fluid velocity profile, we tried an alternative method of fluid velocity measurement which we call Cumulative Average Measurement (CAM); in contrast to SAM, which sums the samples of instantaneous fluid *velocity*, CAM obtains the mean velocity by summing samples of instantaneous fluid *momentum* divided by the cumulative mass over all samples. Fig. 1.6 shows our results with this method. The nonphysical flows disappeared altogether.

In examining the origin of the anomalous fluid velocity with the SAM method, three considerations are key. First, both Figs. 1.4 and 1.5 show measurements in systems far from equilibrium, where significant temperature gradients are present, although the temperature gradient in the Couette system arises indirectly from viscous heating. In the equilibrium plot of Fig. 1.4 the effect is entirely absent.

Second, with a total of 2000 particles distributed over 20 cells, the mean number of simulation particles N in any cell was between 75 and 160, depending on the temperature. With so few particles the thermodynamic fluctuations are significant. As will be shown in Section 2.4, the theoretical root mean square fluctuation of N ,

Figure 1.6: Cumulative average measurement of mean fluid velocity, $\langle u^x \rangle_s$, for different temperature gradients. Note the absence of net fluid velocity, in accordance with expectations for a closed system at the steady state. These measurements were taken from the same simulation runs as those used for Figs. 1.4 and 1.5.

which is also the standard deviation of N , is $\sqrt{\langle N \rangle}$; for a cell containing 100 particles that is approximately 10% of the mean.

Third, when we ran a simulation that increased the total number of particles by a factor of 2.5 without changing any other physical parameters, we observed a decrease in the measured SAM fluid velocity by the same factor. This bears an interesting resemblance to the observation above that the mean square fluctuation is

proportional to the number of particles.

These clues indicate that fluctuations might be causing the measurement bias. The SAM definition, developed in Chapter 2, contains no variables with obvious nonzero variance or covariance at equilibrium. Thus, non-equilibrium fluctuations are the most likely candidates.

After extensively testing the simulator under various scenarios we became convinced that the anomaly had its origin in the SAM method of measuring fluid velocity, whereby a previously unexplored ambiguity in the definition of fluid velocity caused an incorrect measurement with SAM. We found quantitatively that a nonzero covariance of non-equilibrium fluctuations in fluid momentum and density were the source of the incorrect measurement. Furthermore, the covariance is a physical phenomenon rather than a purely computational one; it will be nonzero in a physical system as well. Thus, at non-equilibrium the SAM method is biased, and the correct fluid velocity measurement method is CAM.

Given that both the SAM and CAM measurement methods are apparently in widespread use among those who use molecular simulations to analyze fluid flows, and that the bias in SAM is observable even in very simple non-equilibrium systems, it was somewhat puzzling that apparently others had not previously investigated this phenomenon or recognized that the definition of fluid velocity is ambiguous. The explanation might lie in the magnitude of the effect. For example, in Fig. 1.4 for the moderate gradient case the temperature from the left wall to right wall increases by a factor of three over a distance of only ten mean free paths. Even with this strong gradient, the anomalous fluid velocity from the SAM definition is small (Mach number of approximately 10^{-4}). Thus, the bias is not significant in most molecular simulations of fluid flows, and for that reason it may have been either overlooked

or attributed to a small and unimportant error in program logic. Nevertheless, it is significant enough to have a measurable effect on simulation results in the presence of extreme gradients or a sufficiently rarefied gas. Given that the effect is physical rather than computational in origin, it is also interesting in its own right from a theoretical standpoint.

1.4 Bias in Temperature Measurement at Non-equilibrium

Temperature can be defined loosely as the mean translational kinetic energy per molecule in the center of mass frame of reference [9]. Although it is a thermodynamic variable, it is also a hydrodynamic variable in that it is a property of fluids that appears in the hydrodynamic equations of fluid mechanics. Thus, as with fluid velocity, there may be interesting effects related to temperature measurement in fluids very far from equilibrium.

The measurement of temperature has some well-known difficulties. For example, as will be discussed in Section 6.3, instantaneous fluctuations in the center of mass fluid velocity are part of the definition of temperature, and the temperature measurement process must avoid subtracting it. We found an apparent ambiguity in the definition of temperature itself, related to the method of obtaining the mean translational kinetic energy per molecule. The ambiguity is analogous to that of fluid velocity, and it similarly has an apparent origin in the correlation of non-equilibrium fluctuations. Fig. 1.7 demonstrates it as a nonzero difference between SAM and CAM temperature measurements in a fluid subjected to a sizeable temperature gradient.

Figure 1.7: Difference in fluid temperature measurements obtained by sample averaging measurement and cumulative average measurement, $\langle T \rangle_s - \langle T \rangle_c$, for a closed system with a large temperature gradient ($T_L = 1$, $T_R = 3$). The two measurement methods clearly give slightly different results.

1.5 Anomalous Fluid Flow in an Equilibrium System with a Reservoir

Sometimes it is desirable to perform a molecular simulation of a system that is not closed. The boundary of the simulated system may then be a reservoir of effectively infinite capacity, so that the density of particles in the reservoir does not change as particles are exchanged between the reservoir and the system. The exchange of particles maintains the desired mean density and flow characteristics

while allowing the actual density and fluid momentum within the system to fluctuate stochastically.

There are a variety of ways that are available for implementing reservoirs. One common method is to generate a constant number of particles during each fixed interval of time, randomly assign their positions according to a uniform probability distribution throughout the reservoir, and give them random velocities that are distributed according to the velocity distribution function of interest. For example, at equilibrium the velocity distribution function will be Maxwell-Boltzmann; with a temperature gradient it will be Chapman-Enskog. After being generated, the particles are moved without colliding at their assigned velocities for the entire time interval, and any particles that reach the system then enter the system while the remainder are discarded. Thus, the number of particles entering the system can differ from one interval to the next, although the long term average is constant. Meanwhile, particles from the system that reach the reservoir are simply discarded. A steady state occurs when the mean number of particles leaving the system during a time interval equals the mean number entering the system from the reservoir.

Of course, a virtual reservoir must be sufficiently deep so a particle that is generated with the highest expected velocity and at the greatest depth in the reservoir cannot reach the system within a single time interval. Otherwise the finiteness of the reservoir will bias the distribution of particles entering the system in a nonphysical way; i.e., there will be no occurrences of the highest velocity particles that barely reach the system during a time interval.

When we simulated a system at equilibrium that had a reservoir of this type at the left boundary and a specular wall at the right boundary, we measured an anomalous fluid velocity near the reservoir; see Fig. 1.8. Given that particles were

Figure 1.8: Sample averaging measurement of mean fluid velocity, $\langle u^x \rangle_s$, for an equilibrium system at $T = 1$ with a reservoir at the left boundary and a specular wall at the right boundary. Note the anomalous velocity of fluid from the reservoir.

generated within the reservoir at a constant rate and could only enter and leave the system through the same boundary, it was not physically possible to maintain a net fluid velocity. However, this measurement used SAM. When we used CAM, we obtained the expected zero net fluid velocity, as Fig. 1.9 shows. In light of the SAM anomalous velocity described in the previous section, it was natural to suspect that SAM was once again biased by a correlation of fluctuations in fluid momentum and particle density.

Figure 1.9: Cumulative average measurement of mean fluid velocity, $\langle u^x \rangle_c$, for an equilibrium system at $T = 1$ with a reservoir at the left boundary and a specular wall at the right boundary. The net fluid velocity is zero, as expected.

Several factors indicated that something different was happening with the reservoirs than what we saw in the non-equilibrium systems, however. First, the system was at equilibrium, and as we noted earlier, the covariance of fluctuations that caused the former bias with SAM should not occur at equilibrium. Second, unlike the non-equilibrium systems, this system was open and thus there was the possibility that the reservoir algorithm was somehow introducing a nonzero covariance that would not occur in a physical system. Furthermore, unlike the previous bias

where the anomalous velocity was highest at the center of the system in the case of a temperature gradient, here it decreased with distance from the boundaries.

Analysis of the behavior of a constant, arbitrary volume of fluid shows that the number of particles entering or leaving the volume through an imaginary fixed boundary during a fixed time interval should follow a Poisson distribution. Since this distribution holds for *all* boundaries of the volume, the density of particles within the volume itself must follow a Poisson distribution, regardless of the particle velocity distribution function. Clearly the reservoir implementation described above did not meet this physical requirement because the number of particles that were generated at each time was constant rather than Poisson distributed. Thus, a likely origin of the anomaly was that the constant particle generation rate was somehow introducing an unexpected covariance between fluctuating variables. When we changed the reservoir algorithm so that the number of generated particles at each time followed a Poisson distribution over time intervals, the anomalous SAM fluid velocity disappeared, as Fig. 1.10 shows.

It is important to distinguish between the anomalous fluid velocity we measured in the non-equilibrium system and the one we measured in the equilibrium system with one reservoir. In the non-equilibrium system the SAM method was biased due to the presence of a physically meaningful, non-equilibrium correlation of fluctuations in fluid momentum and particle number within a fixed volume. Eliminating the bias requires changing the measurement method to use CAM instead of SAM. In the equilibrium system with a reservoir, the SAM method is physically correct because at equilibrium there are no non-equilibrium fluctuations present and hence no correlation of them; in our investigations, SAM merely detected an incorrect implementation. Here, elimination of the bias requires changing the reservoir algorithm to more closely

Figure 1.10: Sample averaging measurement of mean fluid velocity, $\langle u_k^x \rangle_s$, for an equilibrium system at $T = 1$ with a reservoir at the left boundary and a specular wall at the right boundary. The number of particles generated at each time was Poisson distributed about the same mean as was used for Fig. 1.8. Note the absence of a net fluid velocity, as expected.

reflect the actual statistical behavior of a fixed volume of gas at equilibrium.

The anomalous flow we found with certain reservoir algorithms is one example of a more general issue in molecular simulations where fluctuations are important. If the stochastic nature of fluctuations is not adequately considered in algorithm design, the algorithm could inadvertently introduce a spatial correlation of fluctuations

that subsequently biases the measurement of hydrodynamic variables. This does not appear to be a topic that others have seriously investigated.

1.6 Thesis Outline

This introduction has presented the general form of the primary results. In subsequent chapters these will be developed in some depth. Chapter 2 provides some general background information, including terminology, quantitative definitions of fluid velocity and temperature, and development of some theory of fluctuations. The following chapter discusses different simulation methodologies, focusing on DSMC in particular because that is the simulation method employed for the investigations. Then Chapter 4 describes in some detail the software implementation of the simulation system, and discusses the validation process that gave confidence that the simulator was producing trustworthy results.

Chapters 5, 6 and 7 form the core of the thesis. These present some illustrative simulations and develop some theory to quantitatively explain the anomalous results for fluid velocity measurement, temperature measurement, and equilibrium reservoirs, respectively. Some results regarding the aforementioned measurement bias of fluid velocity have been presented elsewhere [8], but the present work delves into them in more depth. The discussion and analysis of the results for temperature and particle reservoirs are new.

The present work suggests some further investigations. The conclusion will outline some of these future possibilities, and also give some general remarks about the present work.

CHAPTER 2

BACKGROUND

2.1 Measurement of Fluid Velocity

The terminology and notation conventions established here follow those of the earlier paper [8]. Much of the content of the section of the same name in that paper is repeated here, both for ease of reference and because it directly applies to the present work.

For particle based simulations, there are two common, general methods for calculating fluid velocity. Because differences between these methods form an important part of the context of this thesis, we will define each of them in preparation for an analysis of the actual phenomenon in Chapter 5. In this section we develop the definitions for a single species of molecule, then extend it to an arbitrary number of species in Section 2.2.

We assume a simulation algorithm that partitions the system into K cells of equal volume and gathers statistics separately for each cell, for example like the one in Fig. 1.1. We identify a particular cell with a subscript. Cell k has volume V_k ; during a time t_j it contains $N_k(t_j)$ particles, of mass m , so $M_k = mN_k$ is the fluid mass in the cell. In the general case, the total number of particles in the system, $N_\Sigma(t_j) = \sum_{k=1}^K N_k(t_j)$, will vary with time, although for a closed system it

is necessarily a conserved quantity.

The measurement of an intensive macroscopic variable like instantaneous fluid velocity $\mathbf{u}_k$ or temperature T_k will generally be an average over particles in that cell. Within a given cell, the velocity of particle i is $\mathbf{v}_i$ and its momentum is $m\mathbf{v}_i$. The total fluid momentum in cell k is $\mathbf{J}_k = \sum_{i \in k}^{N_k} m\mathbf{v}_i$. The notation $i \in k$ denotes the index i ranging over the particles within cell k . The instantaneous fluid velocity is the center of mass velocity for the particles in the cell; that is, $\mathbf{u}_k = \mathbf{J}_k/M_k$. Since all particles have equal mass, this fluid velocity equals the average particle velocity $\bar{\mathbf{v}}_k = (1/N_k) \sum_{i \in k}^{N_k} \mathbf{v}_i$, where the overbar indicates an instantaneous average over all particles in a cell. We reserve angle brackets to indicate averages over all samples; for example $\langle N_k \rangle$ is the average number of particles in cell k during the simulation.

The mean fluid velocity $\langle \mathbf{u}_k \rangle$ is an average over samples, which may be taken over time for steady state flows or over an ensemble of systems for time-dependent flows. All of our investigations involved averages over samples, but the bias described in Section 1.3 must also be present when averaging over ensembles. In evaluating $\langle \mathbf{u}_k \rangle$ from samples of particle velocities, there is some flexibility in how statistics are computed. Two methods find common usage, which we now define.

The first, which we will call cumulative average measurement (CAM), sums the velocities of all particles in a cell over all samples, then divides this cumulative sum by the cumulative total number of particles in the cell to yield a mean velocity per particle for the cell. That is, for cell k over S samples and with t_j denoting a particular sample time, the mean fluid velocity by the CAM definition is

$$\langle \mathbf{u}_k \rangle_c = \frac{\frac{1}{S} \sum_{j=1}^S \sum_{i \in k}^{N_k(t_j)} \mathbf{v}_i(t_j)}{\frac{1}{S} \sum_{j=1}^S N_k(t_j)} = \frac{\langle \mathbf{J}_k \rangle}{\langle M_k \rangle}, \quad (2.1)$$

where

$$\langle M_k \rangle = m \langle N_k \rangle = \frac{m}{S} \sum_{j=1}^S N_k(t_j)$$

and

$$\langle \mathbf{J}_k \rangle = \frac{1}{S} \sum_{j=1}^S \mathbf{J}_k(t_j) = \frac{1}{S} \sum_{j=1}^S \sum_{i \in k}^{N_k(t_j)} m \mathbf{v}_i(t_j)$$

are the mean fluid mass and momentum, respectively, in cell k . These averages are similarly defined for each component so $\langle u_k^x \rangle_c = \langle J_k^x \rangle / \langle M_k \rangle$, where the superscript x indicates the x component of a vector.

The second method, which we call sample averaging measurement (SAM), obtains the average particle velocity in a cell at each sample, $\bar{\mathbf{v}}_k(t_j)$, then averages these means over all samples:

$$\langle \mathbf{u}_k \rangle_s = \frac{1}{S} \sum_{j=1}^S \bar{\mathbf{v}}_k(t_j) = \frac{1}{S} \sum_{j=1}^S \frac{1}{N_k(t_j)} \sum_{i \in k}^{N_k(t_j)} \mathbf{v}_i(t_j) = \left\langle \frac{\mathbf{J}_k}{M_k} \right\rangle. \quad (2.2)$$

From the above definitions, $\langle \mathbf{u}_k \rangle_s = \langle \mathbf{J}_k / M_k \rangle$ is evidently different from $\langle \mathbf{u}_k \rangle_c = \langle \mathbf{J}_k \rangle / \langle M_k \rangle$, as a comparison of Eqns. (2.1) and (2.2) shows. The SAM definition is a natural choice when it is useful to measure the instantaneous fluid velocity, for example to monitor the relaxation towards a steady state, and this data is post-processed to obtain mean values.

Although $\langle N_k \rangle$ and $\langle \mathbf{J}_k \rangle$ are the same for both SAM and CAM, this is not true in general for $\langle \mathbf{u}_k \rangle$. Both $\langle \mathbf{u}_k \rangle_s$ and $\langle \mathbf{u}_k \rangle_c$ give the same results in the thermodynamic limit ($\langle N \rangle \rightarrow \infty$) and at thermodynamic equilibrium, but they differ for finite, non-equilibrium systems.

2.2 Measurement of Fluid Velocity for Multiple Species

Now we generalize the CAM and SAM definitions to a fluid containing an arbitrary number A of particle species. It is necessary to extend the notation to

allow identification of a particular species. We do this with the subscript letter a , so that species a has, for example, mass m_a and diameter d_a . When a variable already has a subscript denoting a particular cell we enclose the species in square brackets; e.g., the portion of total fluid mass in cell k comprised of species a molecules is $M_{k[a]}$.

In cell k there are $N_{k[a]}$ particles of species a , so the total number of particles in a cell is $N_k = \sum_{a=1}^A N_{k[a]}$. The total fluid mass in the cell is $M_k = \sum_{a=1}^A M_{k[a]} = \sum_{a=1}^A N_{k[a]} m_a$. The velocity for particle i of species a is $\mathbf{v}_{[a]i}$ and the average particle velocity for species a is $\bar{\mathbf{v}}_{k[a]} = (1/N_{k[a]}) \sum_{i \in k}^{N_{k[a]}} \mathbf{v}_{[a]i}$. Thus, the total fluid momentum in the cell is

$$\begin{aligned} \mathbf{J}_k &= \sum_{a=1}^A \sum_{i \in k}^{N_{k[a]}} m_a \mathbf{v}_{[a]i} \\ &= \sum_{a=1}^A M_{k[a]} \mathbf{u}_{k[a]} = M_k \mathbf{u}_k. \end{aligned} \quad (2.3)$$

The final step in Eqn. (2.3) provides a definition for the instantaneous fluid velocity; i.e., $\mathbf{u}_k = \mathbf{J}_k/M_k = \sum_{a=1}^A \mathbf{u}_{k[a]} (M_{k[a]}/M_k)$. This can be interpreted as the average of the weighted means of the fluid velocities of the individual species obtained independently.

With the definitions given above we can define CAM and SAM for mean fluid velocity, $\langle \mathbf{u}_k \rangle$. As in the single species case, the CAM definition is the most straightforward. As before, for cell k over S samples and with t_j denoting a particular sample time, the mean fluid velocity by the CAM definition is

$$\langle \mathbf{u}_k \rangle_c = \frac{\sum_{j=1}^S \sum_{a=1}^A \sum_{i \in k}^{N_{k[a]}(t_j)} \mathbf{v}_{[a]i}(t_j)}{\sum_{j=1}^S N_k(t_j)} = \frac{\langle \mathbf{J}_k \rangle}{\langle M_k \rangle}, \quad (2.4)$$

where

$$\langle M_k \rangle = \sum_{a=1}^A m_a \langle N_{k[a]} \rangle = \frac{1}{S} \sum_{a=1}^A m_a \sum_{j=1}^S N_{k[a]}(t_j)$$

and

$$\langle \mathbf{J}_k \rangle = \frac{1}{S} \sum_{j=1}^S \mathbf{J}_k(t_j) = \frac{1}{S} \sum_{j=1}^S \sum_{a=1}^A \sum_{i \in k}^{N_{k[a]}(t_j)} m_a \mathbf{v}_{[a]i}(t_j).$$

The CAM definition Eqn. (2.4) can be written in the slightly different form,

$$\langle \mathbf{u}_k \rangle_c = \sum_{a=1}^A \left[\frac{\sum_{j=1}^S \sum_{i \in k}^{N_{k[a]}(t_j)} \mathbf{v}_{[a]i}(t_j)}{\sum_{j=1}^S N_k(t_j)} \right] = \sum_{a=1}^A \frac{\langle \mathbf{J}_{k[a]} \rangle}{\langle M_k \rangle}.$$

This has the form of a weighted sum of contributions from each species, so the CAM fluid velocity for a single species is also

$$\langle \mathbf{u}_k^{[a]} \rangle_c = \frac{\langle \mathbf{J}_{k[a]} \rangle}{\langle M_k \rangle}. \quad (2.5)$$

The SAM definition obtains the average particle velocity in a cell at each sample, $\bar{\mathbf{v}}_{k[a]}(t_j)$, then averages these means over all samples:

$$\begin{aligned} \langle \mathbf{u}_k \rangle_s &= \frac{1}{S} \sum_{j=1}^S \bar{\mathbf{v}}_k(t_j) = \frac{1}{S} \sum_{j=1}^S \sum_{a=1}^A \left(\frac{M_{k[a]}(t_j)}{M_k(t_j)} \right) \bar{\mathbf{v}}_{k[a]}(t_j) \\ &= \frac{1}{S} \sum_{j=1}^S \sum_{a=1}^A \left(\frac{N_{k[a]}(t_j)}{M_k(t_j)} \right) \frac{1}{N_{k[a]}(t_j)} \sum_{i \in k}^{N_{k[a]}(t_j)} m_a \mathbf{v}_{[a]i}(t_j) \\ &= \frac{1}{S} \sum_{j=1}^S \sum_{a=1}^A \frac{1}{M_k(t_j)} \mathbf{J}_{k[a]}(t_j). \end{aligned}$$

The contribution from each molecular species has been weighted according to its mass fraction in the fluid. Now $\langle \mathbf{u}_k \rangle_s$ can be written in either of two forms:

$$\langle \mathbf{u}_k \rangle_s = \frac{1}{S} \sum_{j=1}^S \left[\sum_{a=1}^A \frac{\mathbf{J}_{k[a]}(t_j)}{M_k(t_j)} \right] = \left\langle \frac{\mathbf{J}_k}{M_k} \right\rangle \quad (2.6)$$

or

$$\langle \mathbf{u}_k \rangle_s = \sum_{a=1}^A \left[\frac{1}{S} \sum_{j=1}^S \frac{\mathbf{J}_{k[a]}(t_j)}{M_k(t_j)} \right] = \sum_{a=1}^A \left\langle \frac{\mathbf{J}_{k[a]}}{M_k} \right\rangle. \quad (2.7)$$

Eqn. (2.6) has a form identical to the expression for the single species case. Eqn. (2.7), on the other hand, has the form of a sum of weighted flow velocities for each species.

It is apparent from Eqn. (2.7) that the flow speed of a single species in the mixture can be written as

$$\langle \mathbf{u}_k^{[a]} \rangle_s = \left\langle \frac{\mathbf{J}_{k[a]}}{M_k} \right\rangle.$$

2.3 Measurement of Temperature

Temperature is a thermodynamic state variable that is constant in a system at equilibrium. It is a measure of the mean molecular energy in the center of mass reference frame. For a monatomic gas, all kinetic energy is translational, so the three components of molecular velocity give three degrees of freedom, assuming isotropy in space. For diatomic or polyatomic gases, the gas contains internal energy in the form of rotation and vibration which provide additional degrees of freedom; by equipartition of energy at equilibrium, the translational and ‘internal’ temperatures are equal, and the overall temperature can be defined as the weighted mean of the translational and internal temperatures [9]. In this thesis all gases are monatomic, so the subsequent discussion will assume only translational kinetic energy is present.

The temperature of a system undergoing a change of state is rigorously defined at all times only for reversible changes, i.e., those occurring as a sequence of infinitesimal changes over an infinitely long time. The average translational kinetic energy of the molecules in the center of mass frame can be taken either over samples for an arbitrarily long time or over an arbitrarily large ensemble of identical systems. While the kinetic theory takes into account that fluctuations occur, its requirement for samples over a very long time or very large ensemble presents an obvious practical difficulty in using simulations to study them.

Problems arise if the number of samples or the sampling time are sufficiently reduced. In addition to increasing the variance due to a small sample size, the effect of

fluctuations is magnified. For example, equilibrium fluctuations cause fluctuations in the center of mass velocity, so the instantaneous center of mass velocity will be slightly different at different points in the system at any given time. In a non-equilibrium system at the steady state with a temperature gradient, there is ambiguity in the temperature of a small but finite volume because the long-time mean translational kinetic energy will be different at different points in the volume.

Nonetheless, it can be meaningful to perform an instantaneous temperature measurement as long as the measurement is given as a mean and variance. It will be shown in Section 2.4 that the variance is given by the mean square fluctuation, and that this variance is $\langle(\delta T)^2\rangle = C_V/k_B T^2$, independent of the particle number N (see Eqn. (2.23)). It will also be shown there that T and V are independent at equilibrium, as the expression for $\langle(\delta T)^2\rangle$ confirms, so the variance is also independent of the number density $n = N/V$ under those conditions.

Just as we developed two different definitions of fluid velocity, we now develop the CAM and SAM definitions of temperature for a monatomic gas. Because collisions between molecules are elastic, kinetic energy is conserved in each collision. Hence, kinetic energy is continuously redistributed among the molecules, giving each molecule the same mean kinetic energy, independent of the number of gas species or the species relative masses. From the equipartition theorem this is $\frac{3}{2}k_B T$ for a monatomic gas, so for every species $\frac{1}{2}\langle m(\mathbf{v} - \langle \mathbf{u} \rangle)^2 \rangle = \frac{3}{2}k_B T$. Thus, definitions of temperature will have the same form for both single- and multi-species monatomic gases, although the CAM and SAM definitions are different.

Temperature is always measured in the center of mass frame, so in that frame $\langle \mathbf{u} \rangle = 0$. In the reference frame where the temperature is measured (e.g., laboratory) it is likely that $\langle \mathbf{u} \rangle \neq 0$, so the mean fluid velocity must be measured and subtracted.

Of course, physical thermometers do this implicitly; in a thermometer that measures temperature by an expanding fluid, for example, that fluid is kept mechanically isolated by the thermometer walls and hence is always at rest. But in a simulation the subtraction must be handled computationally, and doing it correctly requires care to avoid biasing the temperature measurement. We discuss this further in Section 6.3.

For the CAM definition,

$$\begin{aligned} \langle T_k \rangle_c &= \frac{2}{3k_B} \frac{\frac{1}{S} \left[\left(\sum_{j=1}^S \sum_{i \in k}^{N_k(t_j)} \frac{1}{2} m_i \mathbf{v}_i^2 \right) - \frac{1}{2 \langle M \rangle} \left(\sum_{j=1}^S \sum_{i \in k}^{N_k(t_j)} m_i \mathbf{v}_i \right)^2 \right]}{\frac{1}{S} \sum_{j=1}^S N_k(t_j)} \\ &= \frac{2}{3k_B} \left[\frac{\langle K_k \rangle}{\langle N_k \rangle} - \frac{1}{2} \frac{\langle M_k \rangle}{\langle N_k \rangle} \left(\frac{\langle \mathbf{J}_k \rangle}{\langle M_k \rangle} \right)^2 \right] \end{aligned} \quad (2.8)$$

$$= \frac{2}{3k_B} \left[\frac{\langle K_k \rangle}{\langle N_k \rangle} - \frac{1}{2} \frac{\langle M_k \rangle}{\langle N_k \rangle} \langle \mathbf{u}_k \rangle_c^2 \right], \quad (2.9)$$

where $\mathbf{v}_i$, the velocity of particle i , is understood to apply to a particular time, i.e., $\mathbf{v}_i(t_j)$. The total fluid momentum and mass at time t_j , $\mathbf{J}_k$ and M_k respectively, have the definitions given in Sections 2.1 and 2.2. The definition of average kinetic energy $\langle K_k \rangle$ in cell k over all S samples is given by

$$\langle K_k \rangle \equiv \frac{1}{S} \sum_{j=1}^S \sum_{i \in k}^{N_k} \frac{1}{2} m_i \mathbf{v}_i^2. \quad (2.10)$$

Because the mean energy $\langle K \rangle$ is an extensive mechanical quantity, its definition is identical for both CAM and SAM. (Note that the symbol K in this context denotes kinetic energy, rather than the number of cells it denoted in Section 2.1. The distinction should be clear from the context.)

There are actually two different ways to define the SAM temperature. The first one measures the instantaneous temperature at each sample, then averages these over all samples. This definition is inherently biased; it systematically underestimates the temperature, even at equilibrium, as we will show in Chapter 6. To distinguish this

fundamentally biased temperature, we denote it as T' . According to the definition of T' ,

$$\begin{aligned}
\langle T'_k \rangle_s &= \frac{2}{3k_B} \frac{1}{S} \sum_{j=1}^S \frac{1}{2} \left[\overline{m_i \mathbf{v}_i(t_j)^2} - \frac{1}{M_k(t_j)/N_k(t_j)} \left(\overline{m_i \mathbf{v}_i(t_j)} \right)^2 \right] \\
&= \frac{2}{3k_B} \frac{1}{S} \sum_{j=1}^S \frac{1}{2} \left[\frac{1}{N_k(t_j)} \sum_{i \in k}^{N_k(t_j)} m_i \mathbf{v}_i^2 - \frac{N_k(t_j)}{M_k(t_j)} \left(\frac{1}{N_k(t_j)} \sum_{i \in k}^{N_k(t_j)} m_i \mathbf{v}_i \right)^2 \right] \\
&= \frac{2}{3k_B} \left[\left\langle \frac{K_k}{N_k} \right\rangle - \frac{1}{2} \left\langle \frac{M_k}{N_k} \left(\frac{\mathbf{J}_k}{M_k} \right)^2 \right\rangle \right] \\
&= \frac{2}{3k_B} \left[\left\langle \frac{K_k}{N_k} \right\rangle - \frac{1}{2} \left\langle \frac{M_k}{N_k} \mathbf{u}_k^2 \right\rangle \right]. \tag{2.11}
\end{aligned}$$

The second SAM temperature definition does not suffer from this inherent bias at equilibrium. It measures the instantaneous mean kinetic energy per particle, then averages these over all samples. Because translational temperature must be measured in the center of mass frame, it is necessary to subtract the kinetic energy due to the center of mass motion. This definition of SAM temperature, then, requires measurement of both the mean kinetic energy per particle and mean particle velocity, both according to SAM:

$$\begin{aligned}
\langle T_k \rangle_s &= \frac{2}{3k_B} \frac{1}{S} \left[\sum_{j=1}^S \frac{1}{2} \left(\overline{m_i \mathbf{v}_i(t_j)^2} \right) - \left\langle \frac{M_k}{N_k} \right\rangle \left(\sum_{j=1}^S \frac{1}{2} \frac{N_k(t_j)}{M_k(t_j)} \overline{m_i \mathbf{v}_i(t_j)} \right)^2 \right] \\
&= \frac{2}{3k_B} \frac{1}{S} \frac{1}{2} \left[\sum_{j=1}^S \frac{1}{N_k(t_j)} \sum_{i \in k}^{N_k(t_j)} m_i \mathbf{v}_i^2 - \left\langle \frac{M_k}{N_k} \right\rangle \left(\sum_{j=1}^S \frac{1}{M_k(t_j)} \sum_{i \in k}^{N_k(t_j)} m_i \mathbf{v}_i \right)^2 \right] \\
&= \frac{2}{3k_B} \left[\left\langle \frac{K_k}{N_k} \right\rangle - \frac{1}{2} \left\langle \frac{M_k}{N_k} \right\rangle \left\langle \frac{\mathbf{J}_k}{M_k} \right\rangle^2 \right] \tag{2.12}
\end{aligned}$$

$$= \frac{2}{3k_B} \left[\left\langle \frac{K_k}{N_k} \right\rangle - \frac{1}{2} \left\langle \frac{M_k}{N_k} \right\rangle \langle \mathbf{u}_k \rangle_s^2 \right] \tag{2.13}$$

for a fluid of any number of species having different molecular masses. For a single constituent species with molecular mass m_a , Eqn. (2.13) takes a slightly more explicit

form:

$$\langle T_k \rangle_s = \frac{2}{3k_B} \left[\left\langle \frac{K_{k[a]}}{N_{k[a]}} \right\rangle - \frac{1}{2} m_a \langle \mathbf{u}_{k[a]} \rangle_s^2 \right]. \quad (2.14)$$

In future discussions of temperature, all references to the SAM temperature will correspond to the definitions given by Eqns. (2.12) or (2.13). If at any time it is necessary to refer to the inherently biased definition given by Eqn. (2.11), we will explicitly refer to that temperature as T' .

2.4 Dependence of Fluctuation Magnitudes on Particle Number

In a macroscopically large but fixed volume of gas at equilibrium containing a fixed number N of molecules with a constant total energy, there will be localized fluctuations in the density and energy of the molecules in any sub-volume due to the random motions of the molecules. Consequently, the thermodynamic variables of temperature, pressure and entropy will also fluctuate about their mean values in the sub-volume. The task here is to show that the variance and covariance of fluctuations of extensive variables like particle number and fluid momentum are proportional to N , while the variance and covariance of fluctuations of intensive quantities like fluid velocity are proportional to $1/N$. The development closely follows that of the references [10, 11], relying first on thermodynamic reasoning, and then drawing the same conclusion from probability arguments.

Consider a macroscopically large, isolated system of fixed total volume V_t whose total energy E_t is distributed among N_t molecules. Let V_t be divided into a small but macroscopic subsystem of volume V , and a much larger “reservoir” V_0 ; V has total energy E distributed among N particles and V_0 has total energy E_0 distributed among N_0 particles. That is, $E_t = E + E_0$ and $N_t = N + N_0$. Because $V \ll V_0$, the relative fluctuations of the various quantities will be much larger in V than V_0 , so V

will be the focus of our analysis.

If we denote $\langle X \rangle$ as the mean of a fluctuating variable X and δX as the fluctuation of X about that mean, then the instantaneous value of X has the form,

$$X = \langle X \rangle + \delta X. \quad (2.15)$$

By definition, $\langle \delta X \rangle = 0$; this can also be seen by taking the mean of both sides of Eqn. (2.15).

The intensive thermodynamic variables for temperature and pressure are denoted T and P , respectively. From their kinetic definitions, they are strictly defined only as statistical averages, either over a long time in a single system at equilibrium or over an ensemble of many identical systems. Nonetheless, we can relax the requirements a little and define the instantaneous temperature as the mean translational kinetic energy of the molecules in the center of mass reference frame, and the pressure as the normal component of momentum flux through an imaginary surface. Thus, the two variables T and P will fluctuate about their equilibrium values in V , and similarly for T_0 and P_0 in V_0 . Because V and V_0 can exchange energy, $\langle T \rangle = \langle T_0 \rangle = T_{eq}$, where the *eq* subscript denotes the equilibrium value. However, because $V \ll V_0$ and V_0 is a reservoir, fluctuations in T_0 and P_0 will be negligible and we can assume $T_0 = T_{eq}$ and $P_0 = P_{eq}$. Ultimately this assumption rests on observations that large deviations in temperature or pressure do not spontaneously occur in macroscopically large volumes containing a large number of molecules.

The entropy S is an extensive variable, and it similarly fluctuates according to Eqn. (2.15). At equilibrium the total entropy S_t of the system is at a maximum, so any fluctuation must reduce S_t . From statistical mechanics [10], this entropy is related to the number of microstates Γ by $S_t = k_B \ln \Gamma$, where Γ corresponds to the

total energy E_t . Thus, the total entropy change due to a fluctuation is

$$\delta S_t \equiv k_B \ln \Gamma_f - k_B \ln \Gamma_{eq};$$

the f and eq subscripts denote the presence of a fluctuation and the equilibrium state, respectively. Since $k_B \ln \Gamma_{eq}$ is constant for constant energy E_t , we have for the probability of a fluctuation,

$$p \propto \Gamma_f \propto e^{\delta S_t/k_B}. \quad (2.16)$$

The particle density $n = N/V$ is what fluctuates, so we obtain the same value for n regardless of whether we fix V and let N vary or fix N and let V vary. Taking the former view, then for N fixed, $P_{eq} dV_0$ is the amount of work done on the reservoir to produce the fluctuation. If dq_0 is the amount of heat transferred from V to the reservoir, then from the first law of thermodynamics, $dq_0 = P_{eq} dV_0$. Dropping the eq subscript on T and P for simplicity and denoting δS as the fluctuation in entropy within the volume V ,

$$\begin{aligned} \delta S_t = \delta S + \delta S_0 &= \delta S + \int_0^f \frac{dq_0}{\langle T_0 \rangle} \\ &= \delta S + \int_0^f \frac{dE_0 + P dV_0}{T} \\ &= (T\delta S)/T - (\delta E + P\delta V)/T. \end{aligned}$$

The last step is justified by noting that $dE_0 = -dE$ and $dV_0 = -dV$. Eqn. (2.16) becomes

$$p \propto \exp \left(-\frac{\delta E - T\delta S + P\delta V}{k_B T} \right). \quad (2.17)$$

Since the magnitude of a fluctuation is small, we can Taylor expand the change in energy δE about the equilibrium value $\langle E \rangle$. Taking E to be a function of S and

V ,

$$\begin{aligned}\delta E &= \left(\frac{\partial E}{\partial S}\right)_{eq} \delta S + \left(\frac{\partial E}{\partial V}\right)_{eq} \delta V \\ &+ \frac{1}{2} \left[\left(\frac{\partial^2 E}{\partial S^2}\right)_{eq} (\delta S)^2 + 2 \left(\frac{\partial^2 E}{\partial S \partial V}\right)_{eq} \delta S \delta V + \left(\frac{\partial^2 E}{\partial V^2}\right)_{eq} (\delta V)^2 \right] + \dots\end{aligned}$$

Using the thermodynamic relationships

$$\left(\frac{\partial E}{\partial S}\right)_{eq} = T, \quad \left(\frac{\partial E}{\partial V}\right)_{eq} = -P$$

and keeping terms up to second order,

$$\begin{aligned}\delta E &\cong T\delta S - P\delta V \\ &+ \frac{1}{2} \left[\delta S \frac{\partial}{\partial S} \left(\frac{\partial E}{\partial S}\right)_{eq} \delta S + \delta S \frac{\partial}{\partial S} \left(\frac{\partial E}{\partial V}\right)_{eq} \delta V \right] \\ &+ \frac{1}{2} \left[\delta V \frac{\partial}{\partial S} \left(\frac{\partial E}{\partial V}\right)_{eq} \delta S + \delta V \frac{\partial}{\partial V} \left(\frac{\partial E}{\partial V}\right)_{eq} \delta V \right] \\ &= T\delta S + P\delta V + \frac{1}{2} \left[\delta S \frac{\partial}{\partial S} \delta E + \delta V \frac{\partial}{\partial V} \delta E \right] \\ &= T\delta S + P\delta V + \frac{1}{2} [\delta S \delta T - \delta V \delta P].\end{aligned}\tag{2.18}$$

Substituting Eqn. (2.18) into Eqn. (2.17),

$$p \propto \exp \left(-\frac{\delta T \delta S - \delta P \delta V}{2k_B T} \right).\tag{2.19}$$

Only two of the variables in Eqn. (2.19) are independent, one of which is intensive and the other extensive, so we can rewrite two of them in terms of the other two. Selecting T and V to be the independent variables and making use of the Maxwell equation $\left(\frac{\partial S}{\partial V}\right)_T = \left(\frac{\partial P}{\partial T}\right)_V$ to eliminate S , we have

$$\delta S = \left(\frac{\partial S}{\partial T}\right)_V \delta T + \left(\frac{\partial S}{\partial V}\right)_T \delta V = \frac{C_V}{T} \delta T + \left(\frac{\partial P}{\partial T}\right)_V \delta V\tag{2.20}$$

and

$$\delta P = \left(\frac{\partial P}{\partial T} \right)_V \delta T + \left(\frac{\partial P}{\partial V} \right)_T \delta V. \quad (2.21)$$

Substituting Eqns. (2.20) and (2.21) into Eqn. (2.19) and cancelling the $\left(\frac{\partial P}{\partial T} \right)_V \delta V \delta T$ terms gives

$$\begin{aligned} p &\propto \exp \left(-\frac{\frac{C_V}{T}(\delta T)^2 - \left(\frac{\partial P}{\partial V} \right)_T (\delta V)^2}{2k_B T} \right) \\ &= \exp \left(-\frac{C_V}{2k_B T^2} (\delta T)^2 \right) \cdot \exp \left(\frac{1}{2k_B T} \left(\frac{\partial P}{\partial V} \right)_T (\delta V)^2 \right) \end{aligned} \quad (2.22)$$

which is a product of two Gaussian distributions, each a function of a single fluctuating variable δT or δV . That is, fluctuations in T and V are statistically independent at equilibrium. From the form of a Gaussian, $e^{-(X^2/2\sigma_X^2)}$, where σ_X^2 is the variance of X , Eqn. (2.22) shows that

$$\sigma_{(\delta T)}^2 = \langle (\delta T)^2 \rangle = \langle (\delta T)^2 \rangle = \frac{C_V}{k_B T^2} \quad (2.23)$$

and

$$\sigma_{(\delta V)}^2 = \langle (\delta V)^2 \rangle = -k_B T \left(\frac{\partial V}{\partial P} \right)_T. \quad (2.24)$$

Eqn. (2.24) is the important result for present purposes. Earlier in this section we noted that it does not matter whether we hold N fixed and let V vary or hold V fixed and let N vary because what is important is the fluctuation in the number density, δn . Now we want to hold V fixed and vary N . Inverting both sides of Eqn. (2.24) and then multiplying by N^2 ,

$$N^2 / \langle (\delta V)^2 \rangle = \langle (\delta n)^2 \rangle = \langle (\delta N)^2 \rangle / V^2 = -\frac{N^2}{k_B T} \left(\frac{\partial P}{\partial V} \right)_T$$

so

$$\langle (\delta N)^2 \rangle = -\frac{N^2 V^2}{k_B T} \left(\frac{\partial P}{\partial V} \right)_T. \quad (2.25)$$

We are interested in the fluctuations in a rarefied gas, which we noted earlier acts very nearly like an ideal gas. Evaluating $(\frac{\partial V}{\partial P})_T$ for $V = Nk_B T/P$ from the ideal gas law, and then substituting this into Eqn. (2.25), gives

$$\langle(\delta N)^2\rangle = -\frac{N^2 V^2}{Nk_B T} \left(\frac{\partial (1/V)}{\partial V} \right)_T = N(k_B T V^2)/k_B T V^2$$

and thus

$$\langle(\delta N)^2\rangle = N. \quad (2.26)$$

Because Eqn. (2.26) used thermodynamic reasoning, it is only valid for large $\langle N \rangle$. If this requirement is removed, the same conclusion results but by different reasoning, as we will now show.

As before, we consider a small but macroscopic volume V containing N particles and a much larger reservoir of volume V_0 and N_0 particles, with $N_t = N + N_0$. If we select one particle, the probability that it will be found in V is V/V_t , assuming all positions in V_t are equally likely. For two distinguishable particles, the probability that both will be in V is $(V/V_t)^2$; for N distinguishable particles the probability is $(V/V_t)^N$. Similarly, the probability that a particular particle will *not* be in V is $(V_t - V)/V_t$; for all of $(N_t - N)$ particles it is $((V_t - V)/V_t)^{N_t - N}$.

There are N_t particles in all and so there are $\binom{N_t}{N} = \frac{N_t!}{N!(N_t - N)!}$ ways we can choose N out of N_t total particles in V , and exactly one way to choose $(N_t - N)$ of the remaining $(N_t - N)$ particles that are not in V . Because $N_t \gg N$, $N_t - N \approx N_t$ and we can approximate $N_t!$ quite well by

$$N_t! = N_t \cdot (N_t - 1) \cdots (N_t - N + 1) \cdot (N_t - N)! \cong N_t^N (N_t - N)!$$

Thus, the probability that N distinguishable particles are in V and the remain-

ing $(N_t - N)$ particles are not in V is

$$\begin{aligned} p &= \frac{N_t!}{N!(N_t - N)!} \left(\frac{V}{V_t}\right)^N \left(1 - \frac{V}{V_t}\right)^{N_t - N} \cong \frac{N_t^N}{N!} \left(\frac{V}{V_t}\right)^N \left(1 - \frac{V}{V_t}\right)^{N_t - N} \\ &= \frac{1}{N!} \left(\frac{N_t V}{V_t}\right)^N \left(1 - \frac{V}{V_t}\right)^{N_t}. \end{aligned}$$

Since $N_t \left(\frac{V}{V_t}\right) = \langle N \rangle$ and $\left(1 - \frac{V}{V_t}\right) = \left(1 - \frac{N_t V}{N_t V_t}\right) = \left(1 - \frac{\langle N \rangle}{N_t}\right)$, the above can be rewritten as

$$p \cong \frac{\langle N \rangle^N}{N!} \left(1 - \frac{\langle N \rangle}{N_t}\right)^{N_t}. \quad (2.27)$$

Taking the limit of very large N_t , the last term in Eqn. (2.27) looks like the definition, $e^x = \lim_{h \rightarrow \infty} \left(1 + \frac{x}{h}\right)^h$; i.e., $e^{-\langle N \rangle} = \lim_{N_t \rightarrow \infty} \left(1 - \frac{\langle N \rangle}{N_t}\right)^{N_t}$. Substituting this into (2.27) the equation reduces to the form of a Poisson probability distribution for $\langle N \rangle$:

$$p = \frac{\langle N \rangle^N e^{-\langle N \rangle}}{N!}. \quad (2.28)$$

The mean and variance of any Poisson-distributed variable are equal, which once again shows that $\langle (\delta N)^2 \rangle = \langle N \rangle$, in agreement with Eqn. (2.25). Although there was no explicit reference to ideal gas behavior, unlike the derivation of Eqn. (2.25), the absence of inter-particle interactions is required for the assumption of independence of particle motion to hold.

That Eqns. (2.26) and (2.28) yield the same relationship between $\langle (\delta N)^2 \rangle$ and N is not surprising in light of the central limit theorem of probability. Given that the correct probability distribution is Poisson, as derived above, then the central limit theorem shows that the distribution must converge to a normal, i.e., Gaussian, distribution for large enough N . That condition certainly holds in the thermodynamic limit where Eqn. (2.26) applies.

In this thesis we are interested in fluctuations of extensive variables like particle number N and fluid momentum $\mathbf{J}$. Fluid momentum is merely the product of the

fluid velocity $\mathbf{u}$ and the total mass of the particles $M = Nm$, so at equilibrium,

$$\langle(\delta\mathbf{J})^2\rangle \propto \langle N\rangle, \quad \langle(\delta N)^2\rangle \propto \langle N\rangle. \quad (2.29)$$

Similarly,

$$\langle\delta\mathbf{J}\delta N\rangle \propto \langle N\rangle. \quad (2.30)$$

We are also interested in fluctuations of intensive variables like fluid velocity $\mathbf{u}$. In general, equilibrium statistical mechanics [10] shows that fluid velocity fluctuates independently of the thermodynamic variables. For example,

$$\langle\delta\mathbf{u}\delta N\rangle = 0, \quad \langle\delta\mathbf{u}\delta T\rangle = 0. \quad (2.31)$$

We can readily obtain the mean square fluctuation of fluid velocity. At equilibrium, the mean fluid momentum $\langle\mathbf{J}\rangle$ of a sub-volume is zero in the center of mass frame of reference, but $\mathbf{J}$ may instantaneously have a nonzero value due to a fluctuation. The kinetic energy of a fluctuation $\delta\mathbf{J}$ is simply $(\delta\mathbf{J})^2/2M = \frac{1}{2}M\mathbf{u}^2$. From equilibrium statistical mechanics, the corresponding probability of a fluctuation $\delta\mathbf{u}$ obeys the relationship

$$p \propto e^{-M\mathbf{u}^2/2k_B T} = \exp\left(\frac{(u^x)^2 + (u^y)^2 + (u^z)^2}{2k_B T}\right),$$

which has the form of a product of Gaussian distributions. The mean square fluctuation of each velocity component is therefore $\sigma_{u^x}^2 = \langle\delta u^x\rangle^2 = k_B T/mN$. From Eqn. (2.31), $\mathbf{u}$ fluctuates independently of T and N , and so its mean square fluctuation depends on the mean rather than instantaneous values of T and N . That is, $\langle\delta u^x\rangle^2 = \frac{1}{\langle N\rangle} (k_B \langle T\rangle/m)$, and so

$$\langle(\delta\mathbf{u})^2\rangle \propto \frac{1}{\langle N\rangle}. \quad (2.32)$$

CHAPTER 3

SIMULATION METHODOLOGY

3.1 Classification of Fluid Simulation Methods

3.1.1 Continuum versus particle methods

For most hydrodynamic systems, either the solution of a set of differential equations or the measurement of mean values of hydrodynamic and thermodynamic variables in a molecular simulation suffice to extract the important physical content from the system, and it is safe to ignore fluctuations entirely. For example, the Navier-Stokes equations of fluid mechanics assume that the temperature, density and momentum are continuous variables that do not fluctuate about their mean values at a given point, although the equations predict how these variables will evolve in time subject to external forces and time varying boundary conditions.

Of course, differential equations like the hydrodynamic equations of fluid dynamics represent an idealization because fluids microscopically consist of discrete molecules. For distance scales and fluid densities that are usually of interest the continuum approach models the physics very well. If fluctuations are of interest in a continuous fluid, Landau fluctuating hydrodynamics [5] models them correctly by incorporating spontaneous local stress and heat source terms into the Navier-Stokes equations. In practical, computational work the spontaneous terms would be gener-

ated through the use of random deviates.

Because of the underlying molecular nature of fluids, continuum models described by differential equations necessarily break down when the distance scale or number density N/V are sufficiently small. It then becomes necessary to treat the system as a collection of discrete particles. Typically a single number, the Knudsen number Kn , characterizes how closely the fluid matches the idealizations of the continuum model. That is, $Kn = \lambda/L$, where λ is the mean free path between discrete collisions of molecules and L is the characteristic length of the system; breakdown occurs [9] for $Kn \gtrsim 0.1$. If the continuum model does not apply, analyzing the system becomes a many-body problem that is intractable except computationally. The molecular simulation techniques most commonly used to investigate such systems are molecular dynamics [12], direct simulation Monte Carlo (DSMC) [9] and lattice gases [13]. In this work we present results obtained using DSMC methods, so the discussion of molecular simulation methods will focus disproportionately on DSMC.

3.1.2 Continuum models

Assuming that Kn is sufficiently small, continuum methods are preferable because they are computationally much less demanding than particle methods. In the continuum approach, the system is partitioned into a grid of cells. Hydrodynamic and thermodynamic variables are calculated for each cell by solving partial differential equations (PDEs) and boundary conditions that model the system. Typically the PDEs will be the Navier-Stokes equations, and the boundary conditions will include geometry, temperatures and fluid velocities at the boundaries, and so on.

The equations are “discretized” in both space and time. The discrete units of space are the cells comprising the system, while the discrete units of time come about

by evolving the system as a series of incremental steps of finite duration, or “time steps.” The time step must be small enough to correctly capture the time dependent behavior of the PDE, but if it is too small the PDE solver will waste a lot of time or become less accurate; the time step can also affect the computational stability of a method. For space discretization, the cell dimensions must be small enough to avoid large changes in variables between adjacent cells or accuracy is compromised; if they are too small solution time or storage requirements may be excessive, or accuracy may suffer as a result of accumulation of round-off errors. There is a large body of literature on the subject of continuum methods, two of which are given in the references [16, 17].

3.1.3 Molecular dynamics (MD)

Molecular dynamics is a particle based method that uses Newton’s laws to move individual particles along a trajectory and collide them with other particles or with the system boundaries in a fully deterministic manner. The time step must be chosen small enough to accurately track the motion of each particle. Molecules have finite size, so the scattering angles after a collision can be accurately determined from the molecule geometry and impact parameter.

Because the particle motions are deterministic, this is the most accurate method for simulation over all possible conditions that might exist in the fluid. For example, there is no requirement for binary collisions (a binary collision is where a particle collides with only one other particle during a time step), so it is very effective for dense gases or liquids where multiple collisions often occur. Its primary disadvantage is computational intensiveness; the motion of each particle must be analyzed at each time, and this grows linearly with the number of particles. It is particularly inefficient

for rarefied gases because collisions are infrequent, yet it is as a result of collisions between particles that much of what is interesting in a fluid occurs. Thus, a simulation of a rarefied gas must run for many more time steps to obtain the steady state behavior of the system.

3.1.4 Direct Simulation Monte Carlo (DSMC)

As the name implies, direct simulation Monte Carlo is a stochastic method. It is a particle based approach that uses statistical averages of particle movement and collisions to simulate fluid behavior, rather than analyzing the detailed motion of each particle the way MD does. It is fully consistent with classic kinetic theory for gases based on the Boltzmann transport equation [9], and both the Boltzmann equation and DSMC procedures share the assumptions of molecular chaos and dilute gases. The Boltzmann equation can even be derived through the DSMC procedures. Thus, it has an established physical basis, and is quite accurate when simulating rarefied gases.

In DSMC, all movement and collision processing occur separately for each cell. Within a cell, all particles are moved along a trajectory, but no effort is made to determine whether particles collide while moving. After movement, a representative set of collision candidate particles is chosen within the cell, and the subsequent scattering angles of the selected particles are determined stochastically rather than analytically. By eliminating the need to handle collisions deterministically it is much faster than MD. Its requirement for binary collisions, however, restricts the method to fluids where this condition can be physically met, i.e., to rarefied gases. We will discuss this method in greater detail in Section 3.2.

3.1.5 Lattice gases

With this approach the system is constructed as a lattice with one lattice point for each possible particle position. When a particle moves, it must move by an integral number of lattice points because the initial and final positions must always be at some point in the lattice. Not only must particle positions be discretized, but so must particle velocities because movement within a time step must be an integral number of lattice points. In a collision, the initial and final angles are also discretized for the same reason. The system is divided into cells, and thermodynamic and hydrodynamic variables are computed at the cell level. To hide the discrete nature of the lattice and particle motions, each cell in the system must contain many lattice points.

The motivation behind the lattice gas method was a desire for high performance, for example by eliminating floating point arithmetic in favor of higher speed integer arithmetic. The idea was to trade off the ability to reproduce fine grained motion of particles, i.e., continuity of velocities and positions, in favor of computational efficiency, a worthwhile goal given the large demands of particle based simulations. In practice, so many approximations are necessary that the correspondence between the simulation and the actual physical system is limited. By making the number of particles in a cell large enough, a reasonable correspondence between the simulation and physical system is possible, but the need to process many particles increases the overhead per cell. Additionally, floating point performance has increased faster than integer performance, undermining one of the original premises. For these and other reasons, lattice gases have limited applicability as a simulation method.

3.1.6 Hybrid models

For completeness, hybrid models are worth mentioning briefly. Hybrid simulations are appropriate when there is a relatively small region of the system where knowledge of the detailed behavior is desired, and this part is best simulated by particle methods. For the rest of the system, it suffices to assume the idealized behavior required by continuum approaches. Rather than suffer the very high overhead of simulating the entire system with particle methods, a hybrid simulation uses particle methods where the added detail is needed, and continuum methods for the rest. The key to a successful hybrid is correctly interfacing the two models so that hydrodynamic and thermodynamic variables do not suffer a discontinuity at the boundary between the two models. This approach is discussed at length in the references [18, 19], and will not be considered further.

3.2 DSMC Method Details

3.2.1 Motivation

Among the choices of simulation methods from Section 3.1, DSMC is optimal for studying fluctuations phenomena in fluids because of its relatively high throughput and accurate reproduction of physical behavior. In fact, we used the DSMC method for all simulations in the present work.

Careful judgement about the validity of our major results requires understanding and evaluating the reliability of the method, and even considering whether the anomalous fluid flows we investigated may in some way result from the DSMC algorithm. This section is an attempt to meet these needs. An entire book has been devoted to discussing the DSMC method and its applications [9], so we can do little

more than touch on some of the highlights here.

3.2.2 General comments

In any molecular simulation, the system is discretized in space, and evolution of the system occurs in discrete time steps. In DSMC, the discrete spatial volumes are called cells. Each cell contains a list of all particles in that cell, including the precise position and velocity of each particle. Typically each cell will have associated statistical information, such as a temperature, fluid velocity and cumulative particle number. These statistics are updated at each sample time, which may or may not be at each time step.

There are two major phases in each time step, the particle movement phase and the collision phase. During movement, each particle is deterministically moved along its current trajectory for the duration of the time step, ignoring the existence of other particles but handling boundary conditions. During the collision phase, a representative subset of the particles collide randomly, conserving energy and linear momentum. While the movement phase is fairly straightforward, the collision processing phase has some peculiar aspects that at first may seem counter-intuitive or even physically incorrect. Nonetheless, the stochastic nature of the DSMC method corresponds nicely with the statistical theory of gases, as long as the basic requirements of the method are met and the number of samples is sufficiently large. Ultimately, the best supporting arguments for the usefulness of DSMC is its excellent agreement with physical theory and other, well-tested simulation methods, together with some proofs that link it to the Boltzmann equation.

A peculiarity of DSMC is that each simulation particle can represent more than one particle in the simulated physical system. The rationale is as follows.

The physical system to be simulated may contain a number of molecules on the order of Avogadro's number, for example when simulating the flow of air over an airfoil in aircraft design. Individually moving and colliding 10^{24} molecules at each time step in a simulation is obviously not feasible, either in storage requirements or computation time, but it is reasonable to simulate such a system if each simulation particle represents a large number of physical particles. The only requirement is that the statistical behavior of particles in the simulation must match that of the physical system being simulated. Thus, in a DSMC simulation there is the total number of simulation particles, denoted by N_Σ , and the effective number of particles, denoted by N_{eff} . Usually $N_{\text{eff}} \gg N_\Sigma$, but in simulations such as those reported in this thesis, the two numbers may be the same. If the simulated gas has mass density ρ , with V as the total system volume and $\langle m \rangle$ as the average mass per particle (weighted by its relative number density), then

$$N_{\text{eff}} = \rho V / \langle m \rangle N_\Sigma. \quad (3.1)$$

For a realistic simulation, the number of simulation particles must be sufficiently large or the duration of the simulation must be sufficiently long to allow accumulation of a statistically representative set of samples. A system containing an average number of particles per cell on the order of 10 to 100 at any given time should work satisfactorily for simulating ideal or nearly ideal gases such as those we studied.

In a study of fluctuations, it is desirable to keep the number of simulation particles reasonably small because the mean square fluctuations in intensive hydrodynamic variables like fluid velocity are inversely proportional to N_Σ (see Eqn. (2.32)). That is, in a DSMC simulation the mean square fluctuations in some key hydrodynamic variables are functions of the number of simulation particles rather than the number of

physical particles they effectively represent, so avoiding large numbers of particles in a cell makes fluctuations more readily observable. In our simulations we maintained a mean of 50 to 200 particles per cell and let $N_{\text{eff}} = 1$.

3.2.3 Movement phase

As previously mentioned, each particle moves deterministically along its current trajectory for the duration of the time step without colliding with other particles. The lists of particles in each cell are updated whenever a particle leaves one cell and enters another. System boundary conditions are imposed during this phase, since the behavior of a particle at a boundary generally depends only on the boundary condition and not on other particles, at least for rarefied gases. External forces like gravity must also be into account during movement, since the acceleration they cause affects the motion. The movement phase is generally very straightforward, with boundaries and system-defined external accelerations being the only complicating factors.

When a particle reaches a boundary, the boundary condition affects the subsequent motion for the remainder of the time step. The behavior of a particle at a boundary is not part of the DSMC algorithm, but it must be consistent with the physical system to yield correct results. The treatment of thermal walls is a good example. Physically, the absorption and re-emission of a molecule at a thermal wall is typically very small, on the order of picoseconds. A simulator can model this faithfully, reducing accordingly the remainder of the time step during which the particle moves after re-emission, or it can approximate the re-emission time as zero. Approximating it as zero will generally yield realistic results because the re-emission time is much smaller than the time step, and thus it will not noticeably modify either the particle density or the collision behavior.

In our simulator, a particle never spent time at a boundary. At a reservoir, the particle was immediately removed from the system. At a thermal wall, the particle was re-emitted immediately upon striking the wall; its energy and momentum were determined from the wall characteristics alone and had no correlation with the particle's prior motion. Regardless of the boundary condition, after reaching a boundary the particle moved for the unused part of the time step with its new velocity.

In a system with a reservoir, the particles in the reservoir are generated and moved during the movement phase. A reservoir particle that reaches the system within the time step then enters the system; it continues to move within the system for whatever portion of the time step it did not consume in reaching the system.

3.2.4 Collision Phase

Particles collide with each other during the second phase. A particle can collide only with other particles that are in the same cell at the end of the movement phase. Only binary collisions are allowed in DSMC, which is the primary reason that DSMC is suitable only for simulations of rarefied gases.

Within a cell, a representative set of particles is selected as candidates for colliding with each other. The number of candidates reflects the likelihood that a collision would occur within the volume V_k of the cell: the larger their relative velocity or cross-section, the greater the volume of space swept out by the particles in a time step and hence the more likely they are to meet. Thus, the number of particles in the cell, the collision cross-sections, the maximum relative velocity between particles $v_{r(\max)}$, and the duration of a time step all enter into the number of candidates.

If the two particles have diameter d_a and d_b , respectively, then for hard spheres

their collision cross-section will be $\sigma_{a,b} = \pi[(d_a + d_b)/2]^2$. Within a DSMC time step of duration τ , then, the number of collision candidates is

$$N_{\text{cand}} = N_k(N_k - 1)v_{\text{r(max)}}\tau\sigma_{a,b}N_{\text{eff}}/2V_k \quad (3.2)$$

with N_{eff} being given by Eqn. (3.1). The factor of two in the denominator corresponds to two particles per pair. Not every pair of collision candidates actually collides, however. Once a pair of candidate particles is selected, the probability that they actually collide is given by

$$P_{\text{coll}} = v_{\text{r}}/v_{\text{r(max)}} \quad (3.3)$$

where v_{r} is the relative velocity of the two particles and $v_{\text{r(max)}}$ is the maximum value of v_{r} that has been encountered up to that time in the simulation. Thus, the reason for using the maximum relative velocity in Eqn. (3.2) is to ensure that the probability in Eqn. (3.3) does not exceed unity.

In general the number of collision candidate pairs N_{coll} will not be an integer. It is not possible to have “part of a collision,” so N_{coll} must be rounded down to the nearest integer. On the other hand, if the remainder is simply discarded, the average number of collisions will be too low. The remainder contains some actual fraction of the possible collisions that may occur over time, and it represents a large number of physical particles if N_{eff} is large.

Accurate modelling of the system requires correct handling of any left-over fraction of a candidate pair. One way is to carry over the fraction to the next time step, and gradually accumulate these fractional amounts until it exceeds unity, then process an additional candidate pair. An alternative is to add a uniformly distributed random deviate $\mathcal{R}$ in the interval $[0, 1)$ to the fraction, and if the result exceeds unity then process an additional candidate pair; otherwise ignore the fraction. Both

ways should work in practice, although the latter method seems slightly preferable because, at least in principle, the carry-over method could introduce a very small correlation between the collision processing of different time steps. When investigating fluctuations, it is desirable to eliminate as many nonphysical correlations as possible.

It is evident from Eqns. (3.2) and (3.3) that if the maximum relative velocity is set too high then a relatively large fraction of the particles in a cell will be selected as candidates for collisions, but the probability that an individual pair actually collides will be correspondingly lower. This leads to inefficient collision processing and hence lower performance. In a single species gas this is typically handled by initially setting

$$v_{r(\max)} = 3 \langle v_r \rangle \quad (3.4)$$

for each cell, and then permanently updating this number whenever a higher value of v_r occurs within the cell during the simulation.

In a multi-species gas involving molecules of significantly different mass, the same kind of inefficiency occurs when colliding two large mass particles. For a given mean molecular energy, determined by the temperature, the maximum relative velocity between two large masses is necessarily much smaller than that between two small mass particles. If the number of candidates and the collision probability are based on the the maximum relative velocity among all particles in the cell, collision processing between large mass particles is correspondingly less efficient. To mitigate this inefficiency in a multi-species gas, particles of similar mass are typically grouped together and the maximum relative velocity is maintained between each mass group in the same way that it is done for a single species gas; see Eqn. (3.4). Setting an initial value for the maximum relative velocity thus requires a separate calculation for each pair of mass groups. Appendix B gives a derivation of $\langle v_r \rangle$, required for Eqn. (3.4) in both single- and multi-species gases.

Although our discussion has been general enough to describe both single- and multi-species gases, both above and in the definitions of SAM and CAM fluid velocity and temperature in Sections 2.2 and 2.3, all of our investigations were of a single species gas. The multi-species cases remain as subjects for future study.

Collision candidates are selected at random from all particles within a cell without regard for their relative position in the cell or their relative velocity. Thus, it is possible in DSMC for two particles on opposite sides of a cell to collide, even when their relative velocity is too small for them to meet within a time step. This collision-at-a-distance scheme is somewhat counter-intuitive because it obviously could not occur in a physical system. Nonetheless, it models a physical system quite well as long as one is interested in statistical behavior rather than the detailed motion of individual particles, just as the kinetic theory of gases faithfully models the aggregate behavior of the many molecules constituting a gas without giving insight into the motions of individual molecules.

The probabilistic method of handling collisions is the origin of the “Monte Carlo” name in DSMC. It is at the heart of the DSMC method, and is responsible for its substantial performance advantage over MD. It is what allows decoupling particle movement from collision processing. It also avoids the need to determine post-collision angles from the impact parameter, since the impact parameter is not even a factor in determining whether a collision occurs. This allows the post-collision angles to be uniformly distributed spherically, i.e., probabilistically but subject to the constraint that both linear momentum and kinetic energy are conserved in every collision.

Because the impact parameter does not enter into collision processing, collisions do not conserve angular momentum. This was a matter of concern in the early days of

DSMC, but studies showed [9] that simulation results did not differ in any important way when the method was modified to conserve angular momentum. Presumably this is because for a statistically meaningful number of samples the aggregate behavior nearly conserves angular momentum within a cell even if individual collisions do not. In a future study of fluctuations with DSMC, it would be interesting to examine whether non-conservation of angular momentum could induce nonphysical rotational Brownian motion of a small sphere that is freely suspended in a gas.

3.2.5 Discretization of space

According to Bird [9], a DSMC simulation becomes exact in the limit of zero cell volume and zero time step duration. In light of the fact that the impact parameter does not enter into collision processing, this claim seems questionable, at least for a hard sphere gas, because angular momentum cannot be conserved with each collision. Nonetheless, in that limit there would be no “smearing” of gradients over finite cell dimensions and collisions would occur at the precise moment when molecules came into contact. Although this ideal is computationally unachievable, practical guidelines exist for discretization of time and space that allow a DSMC simulation to model the physical system very well.

In Section 3.2.4 we explained that collision partners can be taken from anywhere in a cell. In the presence of a temperature or fluid velocity gradient, particles in one part of a cell will have a higher average velocity than particles in a different part of the cell. However, the DSMC collision procedure permits these particles to collide if they are selected at random as collision candidates, and this can have the net effect of transferring some of the fluid momentum or mean molecular kinetic energy from one part of the cell to another, thereby smearing out the gradient in a nonphysical way.

Furthermore, it could even introduce or magnify correlations in the fluid momentum of non-adjacent cells in a way that would not occur in the physical system. Thus, in the presence of large macroscopic gradients, the cell dimensions must be chosen to be sufficiently small in the directions of gradients to minimize these effects.

One way to reduce the smearing problem is to subdivide each cell into subcells, then take collision partners only from the same subcell whenever possible. The total number of candidates is still determined by the number of particles in the cell, but restricting collisions to near neighbors brings excellent correspondence with the physical system. But even without the refinement of division into subcells, the smearing is empirically reduced to an acceptable level if the the local mean free path is at least two times the linear cell dimension in the direction of the gradient [14].

Additionally, statistics are maintained for each cell. In the presence of a large gradient, resolution of temperature and velocity gradients becomes more fine grained with smaller cell dimensions. On the other hand, decreasing the cell dimensions while maintaining the same system volume requires a corresponding increase in the number of cells, and consequently an increase in storage requirements and execution time.

In our simulations, we maintained a cell width of one-half the mean free path, and did not use subcells.

3.2.6 Discretization of time

In general, the time step τ should be much less than the mean time between collisions. For our simulations, it was usually calculated as [15]

$$\tau = \frac{1}{5} \left(\frac{L_k}{v_{\text{mp}}} \right)$$

where L_k was the cell dimension. The most probable velocity v_{mp} was calculated for a mass of species m_a (see Eqn. (A.1) from Appendix A) as

$$v_{\text{mp}} = \sqrt{2k_B T / m_a}.$$

The largest value of v_{mp} of all molecular species and temperatures in the system was used in the calculation of τ .

CHAPTER 4

SIMULATION SYSTEM

4.1 Idealizations and Simplifications

Our gas departed from ideal behavior in that it was a hard sphere gas whose molecules had diameter d and hence finite volume. Although there were no long range intermolecular forces, the molecules interacted by colliding. The consequent reduction in total available volume for movement modifies the equation of state accordingly:

$$P(V - b) = N_{\Sigma} k_B T,$$

where V is the container volume, b is the total volume occupied by the N_{Σ} hard sphere molecules, and P and T have the usual associations with pressure and Kelvin temperature. The correction to the volume is the same as occurs in the van der Waals equation of state, but the pressure has no corresponding change because a hard sphere gas has no intermolecular forces. For a rarefied gas like the ones we considered, $b \ll V$, and so the gas is effectively ideal.

In addition to assuming a (nearly) ideal gas, another simplification was to assume a monatomic gas. Monatomic gases have a particularly simple form of specific heat, $c_v = \frac{3}{2} k_B$, independent of temperature. This eliminated the need during collision processing to distribute the energy among rotational and vibrational modes; at all times, all internal energy was translational kinetic energy.

Movement and collisions occurred in three dimensions, but all measurements and interesting phenomena were along the x -axis only. A one dimensional system fully captured the key results of the present work, justifying the simplification.

4.2 System Details

All the investigations described in this thesis employed a simple, rectangular, essentially one-dimensional system such as that shown in Figs. 4.1 and 4.2.

Figure 4.1: Conceptual diagram of a closed system. The two walls are perpendicular to the x axis; boundaries are periodic in the y and z directions. The vertical dashed lines represent the boundaries of a single cell, where collision processing and sampling of statistics occur.

The primary axis was denoted the x -axis in our convention. To give a context for the simulations described in subsequent chapters, a somewhat detailed overview of this system follows.

The system had configurable boundaries at the two ends of the system along the x axis. In the y - and z -directions between the walls, the boundaries were periodic. At a periodic boundary, a particle leaving the system immediately reappeared at the

Figure 4.2: Conceptual diagram of a system with one reservoir. The reservoir at the left boundary is shown as two cells deep, and its interface with the system is indicated by a heavy dashed line. The right boundary is a specular wall. Boundaries are periodic in the y and z directions.

opposite side of the system, conserving energy and momentum.

Each boundary for the x -axis could be configured independently in any of the following ways:

- **Thermal wall.** This allowed imposing a temperature gradient $\Delta T = T_R - T_L$ across the system in the x -direction. For the left wall, the temperature was T_L ; for the right wall, it was T_R . A thermal wall was also movable in the y -direction. The wall velocities could be independently set to u_L^y and u_R^y for the left and right walls, respectively; this allowed imposing a shear velocity gradient (i.e., Couette flow) $\Delta u^y = u_R^y - u_L^y$, also in the x -direction.

A particle striking a thermal wall was absorbed and then immediately re-injected into the system with a biased Maxwell-Boltzmann velocity distribution [9] that depended on the wall temperature T and shear velocity u^y . For either wall, the velocity components (v^x, v^y, v^z) for particles leaving the wall

were distributed as

$$\begin{aligned} P(v^x) &= \frac{m}{k_B T} v^x e^{-mv^{x2}/2k_B T} \\ P(v^y) &= \sqrt{\frac{m}{2\pi k_B T}} e^{-m(v^y-u^y)^2/2k_B T} \\ P(v^z) &= \sqrt{\frac{m}{2\pi k_B T}} e^{-mv^{z2}/2k_B T} \end{aligned}$$

- **Specular wall.** A specular wall merely reversed the particle's x -velocity v^x with no other effect. If either wall were specular, then it is apparent that no steady-state temperature or velocity gradient could be maintained. If both walls were specular, energy must be conserved at all times.
- **Periodic boundary.** The wall was removed, and particles reaching the boundary immediately left the system and re-entered at the opposite boundary. Of course, both walls needed to be changed to periodic boundaries for this configuration to be usable. This was used primarily to set a flow in motion and then let it proceed indefinitely. Momentum and energy must be conserved at all times.
- **Reservoir.** The wall was replaced by a simulated particle reservoir of infinite capacity. The temperature, mean particle density, flow velocity and depth could be set independently for each reservoir.

The system was closed for all cases except when a reservoir was present. It was isolated only if both x boundaries were specular or periodic.

For simulation purposes, the system was divided into K rectangular cells of equal size that were distributed along the x -axis. The total system length L was also set as a parameter. Statistical sampling of variables like temperature, particle velocity

and particle number occurred separately for each cell over a total of S samples. Typically one sample of every relevant variable was taken at each time step.

Sampling commenced after an initial system relaxation time. Although the initial conditions were close to the steady state, we usually imposed a relaxation period of half the total simulation time to eliminate the potential for the decaying transients to persist and corrupt the results. These transients would manifest as deviations from steady state conditions, arising from imperfections in the system initialization algorithm.

The system contained a monatomic hard sphere gas. The gas consisted of one molecular species of particle mass m and diameter d . The total number of particles N_Σ could be set as an initial condition, and this number remained constant for the duration of the simulation unless a reservoir was present.

As a concrete example, a typical simulation had the following characteristics. The gas was a single species, each particle having mass $m = 1$ and diameter $d = 7.5 \times 10^{-2}$. The total number of particles was $N_\Sigma = 2000$. The boundaries in the x -direction were thermal walls that imposed a temperature and/or shear velocity gradient, so the system was closed. The system contained $K = 20$ cells. The distance between the thermal walls was $L = 2.25$, giving a cell volume $V_k \cong 0.57$. With these parameters, the mean free path was $\lambda = L/10$ and the mean time between collisions was 2.6×10^{-3} at thermal equilibrium when using a gas density of $\rho = M_k/V_k \cong 175$; this corresponds to a volume fraction of 3.4×10^{-2} . Boltzmann's constant was $k_B = 0.5$, so the most probable molecular speed, $\sqrt{2k_B T/m}$ (see Eqn. (A.1)), was unity at the equilibrium reference state of $T = 1$. The sound speed at the reference state was $c = \sqrt{5k_B T/3m} \cong 0.91$.

The typical simulation gathered statistics for each during each of $S = 5 \times 10^6$

successive DSMC time steps of duration 2.25×10^{-2} . This followed an initial system relaxation time of 5×10^6 time steps. Estimating the viscous relaxation time as $\tau_v = L^2/\lambda c$ gives $\tau_v \approx 25$ at the reference state. That is, sampling in this typical case began after about $2000\tau_v$, approximately 10^8 collisions.

4.3 Simulation Sequence

The basic input to our DSMC simulator was a text file containing a list of simulation parameters that acted as a simulation specification; see Fig. 4.3 for an example. The file could also contain some directives to be used in later output formatting, such as a simulation name. A command-line oriented simulator program then constructed an internal representation of the specified one-dimensional system and simulated it for the specified number of time steps, gathering statistics at each sample time.

Once the simulation completed, the results were written out to text files, one for each simulation run. For example, the simulation specified in Fig. 4.3 would have generated two result files. Besides the DSMC simulator, we also wrote an interactive, graphical user interface program in the MATLABTM language. This program took the simulator result files as input, and permitted various kinds of interactive analysis on the data. After generating the selected plots, it allowed saving them to Encapsulated PostScript (EPS) files for inclusion in other documents such as this thesis. Figs. 4.4 and 4.5 show key parts of the graphical user interface for this program.

4.4 Simulator Implementation Details

4.4.1 General structure of the simulator program

Conceptually, the simulator consisted of five general parts.


```

runs      2                                // number of simulations
title     stationary plates with T gradient (no gravity)
          // -----
          // #1: T_L=T_R=1 (equilibrium)
name      Equilibrium,T=1                  // simulation name
boltz     0.5                             // Boltzmann's constant
mass      1                               // molecular mass
density   1                               // simulated gas density
diameter  1                               // molecular diameter
particles 2000                             // number of molecules
length    2.25079                         // total system length
cells     20                              // number of cells
boundaries 2                              // thermal walls
T_lwall   1                               // left wall temperature
T_rwall   1                               // right wall temperature
v_lwall   0                               // left wall y-velocity
v_rwall   0                               // right wall y-velocity
          // #samples=steps-startsample:
steps     10000000                         // total time steps
startsample 5000000                       // time of first sample
end        1
          // #2: T_L=1, T_R=3 (gradient)
name      T_L=1,T_R=3                     // simulation name
boltz     0.5                             // Boltzmann's constant
mass      1                               // molecular mass
density   1                               // simulated gas density
diameter  1                               // molecular diameter
particles 2000                             // number of molecules
length    2.25079                         // total system length
cells     20                              // number of cells
boundaries 2                              // thermal walls
T_lwall   1                               // left wall temperature
T_rwall   3                               // right wall temperature
steps     10000000                         // total time steps
startsample 5000000                       // time of first sample
end        2

```

Figure 4.3: Example of a simulation specification text file for a single species gas, involving two consecutive simulation runs. Not all possible parameters are supplied here; defaults would be used. Comments are preceded by two forward slashes ('//').

Simulations to include

Garcia86.txt

☒ Equilibrium, T=1
☒ T_L=1, T_R=3
☐ T_L=1, T_R=5
☐ T_L=1, T_R=5 with gravity

Plots To Draw

☐ Mean number density profile
☐ SAM fluid momentum profile
☐ Mean temperature profile
☒ SAM/CAM temperature difference
☐ CAM fluid momentum profile
☐ Mean pressure profile
☐ Collision profile
☐ Particle density profile vs. time
☐ Density-density correlation
☐ Density-momentum correlation
☐ Temp.-temp. correlation
☐ Initial number density profile
☐ Initial momentum profile
☐ Initial temperature profile

Plotting Options

☒ Show theory, but :
☐ Center of mass frame
☒ Remove equilb. fluct.
☐ CAM reference plot
☐ With all correlations
☒ With N-N correl. only
☐ With J-N correl. only
☒ With K-N correl. only
☒ Hide graph title
☐ Hide plot legend
☐ Show error bar
☒ Save plots as EPS files

Figure 4.4: Graphical user interface for the simulation result display and plotting part of the simulator. In this example, the simulation specification file ‘Garcia86.txt’ (the example in Fig. 4.3 was part of this file) was used as input to the DSMC simulator, and it contained specifications for four different simulations, all of which were then treated as a related set of simulations for plotting purposes. In the ‘Simulations to Include’ area, two simulations have been selected. They are to be plotted together in a temperature difference plot, selected within the ‘Plots To Draw’ area. Finally, the selected ‘Plotting Options’ specify that two kinds of covariances are to be plotted for each of the selected simulations, so the final plot will contain a total of six lines. Once the plot has been generated, an [S] button appears next to the selected plot type; pressing it brings up a dialog box that asks where the image of the plot is to be saved. In this example, the plotting option ‘Save plots as EPS files’ has been selected, so the image will be saved as an Encapsulated PostScript file; otherwise it would be saved as a TIFF (Tagged Interchange File Format) file.

The first part read the command line, including the name of the simulation specification file, and then extracted the parameters for each simulation. The output

Figure 4.5: Simulation information display within the graphical user interface part of the simulator. This information appears for a simulation in response to its [?] button in the 'Simulations to Include' area. In this example, histogram data has been saved during the DSMC simulation, so it can be plotted using the [Draw] buttons and then saved to files via the [Save] buttons.

of this part was a set of data structures that fully described all simulation runs that were specified by the file.

The second part implemented the DSMC algorithm and various boundary conditions that could be specified. Also included was a loop that sequenced from one simulation run to the next, if more than one simulation run were specified, as well as the procedures for constructing and initializing the simulation system.

The third part implemented the sampling procedures for gathering statistics.

This included calculating means, variances and covariances of the useful quantities. This part, together with the DSMC code above, formed the heart of the simulator.

The fourth part performed limited post-processing of the statistics, then created the output files and wrote the simulation results to them. The output files for a simulation were created immediately on completion of the corresponding run so the simulator could be re-initialized for the next simulation run, if any.

The fifth part read the simulator output files and allowed the user to interactively generate and view different plots from them. The plotting options included showing theoretical expectations based on Eqn. (5.3) on the same plot as the actual measurements, adding estimated error bars, displaying measurements in the center of mass frame, and so on. Any resulting plot could be saved to an Encapsulated PostScript (EPS) file for inclusion in other documents.

The first four parts of the simulator were implemented as a command-line oriented program, written entirely in the ‘C++’ programming language with the goal of high performance simulation. Appendix C contains a nearly complete listing of this program. The fifth part was interactive, and so flexibility and convenience were much more important than maximum performance. That part was written entirely in the MATLAB language as a graphical user interface program; see Figs. 4.4 and 4.5.

4.4.2 DSMC implementation

The DSMC code (the second and third parts of the command-line program described in Section 4.4.1 above) was derived from a MATLAB language implementation by Garcia [16]. Although DSMC is a relatively high performance molecular simulation method, it is still very computationally intensive. The DSMC code was rewritten

in the ‘C++’ language to obtain better performance, and significantly augmented to meet the requirements of our investigations.

Section 3.2 covered the key parts of the DSMC algorithm and some implementation details. One important component that deserves special mention here, however, is the random deviate generator. The correctness of the DSMC algorithm depends on the generator simulating a very high degree of randomness. Additionally, because fluctuations are inherently a small magnitude, stochastic phenomenon, their study would be compromised if the DSMC algorithm did not accurately simulate a gas so that it followed the statistical theory of gases. Thus, a high quality random deviate generator is crucial to studies of fluctuations.

We chose the Mersenne Twister, a freely available set of ‘C’ functions that implement a high quality random deviate generator with good performance. The algorithm was described by Matsumoto and Nishimura [20]; the Mersenne Twister itself was implemented by Nishimura. It generates 64-bit, double precision numbers that are uniformly distributed in the interval $[0, 1)$. Its period is $2^{19937} - 1$, effectively infinite for simulation purposes. See Appendix C.5 for the program listing.

4.4.3 Measuring variances and covariances

In Chapter 5, we will show that the SAM fluid velocity can be expressed in terms of the CAM fluid velocity (see Eqn. (5.3)):

$$\langle \mathbf{u} \rangle_s = \langle \mathbf{u} \rangle_c \left(1 + \frac{\langle (\delta N)^2 \rangle}{\langle N \rangle^2} \right) - \frac{\langle \delta \mathbf{J} \delta N \rangle}{m \langle N \rangle^2} + O(\delta X^3). \quad (4.1)$$

This makes specific reference to the variance $\langle (\delta N)^2 \rangle$ and covariance $\langle \delta \mathbf{J} \delta N \rangle$. Using simulations to verify the correctness of Eqn. (4.1) requires some method of measuring these quantities.

In general, obtaining the variance σ^2 of a quantity X requires measuring both its mean square value $\langle X^2 \rangle$ and its mean $\langle X \rangle$; the variance is then calculated by the well-known formula

$$\sigma_X^2 = \langle (\delta X)^2 \rangle = \langle X^2 \rangle - \langle X \rangle^2.$$

Note that the mean square fluctuation $\langle (\delta X)^2 \rangle$ is just the variance σ_X^2 .

A general method of measuring the mean square value of some quantity in a cell is to measure its instantaneous value at each sample time, square it, and then take the mean of these squares over all samples for that cell. Obtaining the mean is similarly straightforward. Our simulator determined mean square (and mean) values in each cell for particle number N , particle velocity $\mathbf{v}$ and mean particle velocity $\bar{\mathbf{v}}$, i.e., $\langle N^2 \rangle$, $\langle v^2 \rangle$ and $\langle (\bar{\mathbf{v}})^2 \rangle$, respectively. From these the mean square fluctuation $\langle (\delta N)^2 \rangle$ was calculated, as well as the mean kinetic energy $\langle K \rangle$ that was needed for a covariance described below. Each quantity was sampled in each cell at each time step.

Obtaining the covariance σ_{XY} of two quantities X and Y proceeds in similar fashion. The two quantities of interest are multiplied together, then averaged over all samples to give $\langle XY \rangle$. This, together with the product of the means $\langle X \rangle \langle Y \rangle$ of the same variables, allows calculating the covariance with the formula

$$\sigma_{XY}^2 = \langle \delta X \delta Y \rangle = \langle X Y \rangle - \langle X \rangle \langle Y \rangle.$$

The simulator obtained covariances of fluctuations in each cell among the variables of particle number, fluid momentum $\mathbf{J}$ and kinetic energy K , i.e., $\langle \delta \mathbf{J} \delta N \rangle$ and $\langle \delta K \delta N \rangle$.

4.5 Simulator Validation

4.5.1 Validation approach

Although the critical elements of the DSMC algorithm were initially derived as a straightforward rewrite in the C++ language of a basic, existing MATLAB implementation, coding mistakes or minor, unintended algorithm changes could easily occur. Additionally, the DSMC algorithm was only part of the simulator program. It was therefore necessary to verify that the simulator produced trustworthy results.

The first stage of the simulator validation was to compare plots that were created from the output of the simulator with corresponding plots from a few simple published MATLAB DSMC programs, also by Garcia [16]. The parameters and number of samples were kept the same for the published programs and the simulator, and results were compared. Scenarios included equilibrium and shear velocity gradients (Couette flow). All had short run times, on the order of 10^5 time steps. The results to be compared were profile plots for particle number, fluid velocity and temperature across a 20 cell, one dimensional system. The results from both programs, not shown here, matched closely. Minor differences were unavoidable due to the different implementations of the random deviate generators.

The second stage of validation was to replicate some results from an earlier fluctuations study by Garcia [21]. This paper studied spatial correlations of fluctuations in a one dimensional, closed system containing 20 cells and a monatomic gas of 2000 particles. Thermal walls at both ends of the system allowed imposition of different temperature gradients; the gradients that were studied included $\Delta T = T_R - T_L = 0$ (equilibrium), $\Delta T = 2$ ($T_L = 1$, $T_R = 3$), and $\Delta T = 4$ ($T_R = 5$). We viewed reproduction of these results as a sensitive test because of the small magnitude of the

fluctuations and the characteristic profiles of certain covariances of interest. In the next section we will show some of these plots alongside their counterparts from the Garcia paper for illustrative purposes; we observed good agreement.

We added new plot types to meet the needs of this second stage of validation, but retained the plot types from the first stage. The simulation runs during the second stage were necessarily much longer than in the first stage, due to the requirement for more precise statistics. Both of these facts were relevant to what happened next. Although it was not part of the validation plan for the second stage, we looked at fluid velocity profiles for the different temperature gradients and noticed the anomalous fluid flow that we described in Section 1.3. This is shown below in Fig. 4.6.

Although the velocity was very small, having a Mach number of order 10^{-4} , it was indicative of an error in the implementation of the simulator because the system was closed and at the steady state; conservation of total particle number made such a flow physically impossible. Determining the source of the error made it necessary to test the simulator much more exhaustively, adding an unanticipated, third stage of simulator validation. These additional tests included:

- Check for conservation of total particle number at each time step.
- Count the number of particles entering and leaving each cell to look for conservation violations.
- Measure pressure at the left side of each cell and at the boundaries to look for an anomalous pressure that could drive the flow.
- Verify that the ideal gas law holds within each cell.
- Reverse the wall temperatures and look for changes in the anomalous flow.

Figure 4.6: Plot of sample averaging measurement (SAM) fluid velocity $\langle \mathbf{u} \rangle_s$ versus cell position for equilibrium, medium temperature gradient and large temperature gradient with a single species gas in a closed system. In the presence of a temperature gradient, a nonzero fluid velocity clearly appears, indicating a flaw in the simulator or DSMC algorithm, or that some unexpected physical effect is present.

- Run the simulation for a much longer time to look for a gradual diminishing in the amplitude of the effect.
- Initialize the random deviate generator with a different seed.
- Try different time steps, both larger and smaller than the default.

- Change the total number of particles in the system, first increase and then decrease, and look for changes in the flow.
- Run the simulation with periodic boundaries in all directions, and also run with specular walls, in both cases programmatically checking for violations of energy or linear momentum conservation.
- Initialize the system so that all particles were placed at the same point in the system, e.g., 2000 particles at a single point, and let the system evolve to the steady state. This checked for persistence of initial conditions.
- Change the number of cells in the system to change the ratio of cell dimension to mean free path, first increasing the number and then decreasing it.
- Change the method of measuring fluid momentum to conform to the CAM definition instead of the SAM definition.

Some of the tests above were oriented toward isolating the origin of the problem, whether to the simulator implementation, the DSMC algorithm or some unsuspected physical effect. All of these tests seemed to indicate that the simulator was working correctly, and that there were no violations in conservation of particle number, momentum or energy when they should be conserved. Although we ultimately concluded that a physical effect was causing a measurement bias when using SAM, an important result of all of the tests was a high level of confidence in the integrity of the simulator.

4.5.2 Simulator validation results

We first considered the plots of gross thermodynamic behavior in terms of particle number and gas temperature. In a hard sphere gas like we used throughout our investigations, there are no interactions between particles, other than collisions due to the finite particle diameter. Thus, the gas is nearly ideal, the equation of state being

$$P_k(V_{cell} - b_k) = N_k k_B T_k ,$$

where P_k is the pressure, N_k is the number of particles and T_k is the temperature, all for cell k . The cell volume V_{cell} is the same for all cells, and the total volume b_k occupied by the mean number of particles in the cell is simply the particle volume multiplied by N_k . If the volume fraction occupied by the molecules b_k/V_{cell} is small enough to be neglected, the gas can be taken to be ideal. As stated previously, the volume fraction occupied by molecules in our system was usually 3.4×10^{-2} , so the assumption of ideal gas behavior is approximate but quite reasonable.

Since the gas is (nearly) ideal and the pressure is constant throughout the system at the steady state (see Fig. 4.7), the number of particles in a cell should be inversely proportional to the temperature. Figs. 4.8 and 4.10 for the distribution of particle density $n_k = N_k/V_k$ and temperature taken together are consistent with this expectation; compare also with the corresponding plots (Figs. 4.9 and 4.11, respectively) from the Garcia paper. Note that the ‘medium gradient’ case is actually a very large gradient, with the temperature increasing by a factor of 2.5 over a distance of only ten mean free paths.

Figure 4.7: Plot of gas pressure P as a function of cell position k for equilibrium, medium, and large temperature gradients with a single species gas in a closed system. Pressure is constant across the system, as expected.

Figure 4.8: Plot of particle number density $n = N_k/V_{\text{cell}}$ as a function of cell position k for equilibrium, medium, and large temperature gradients with a single species gas in a closed system. The cell position is given in terms of multiples of the minimum mean free path. Particle number density decreases with increasing temperature, as expected from the ideal gas law.

Figure 4.9: Particle number N_k as a function of cell position, from the Garcia 1986 paper [21] (reproduced by permission).

Figure 4.10: Plot of temperature versus cell position for equilibrium, medium, and large temperature gradients with a single species gas.

Figure 4.11: Temperature versus cell position, from Garcia 1986 paper. Note that there is a temperature slip at the wall; given that, the gradients are 0.140 and 0.278 per mean free path for the $T_R = 3$ and $T_R = 5$ scenarios, respectively.

Figure 4.12 shows the temperature profile within a Couette flow system for two different shear velocities. The temperature increases toward the center of the system due to viscous heating, as expected. Collision frequency is determined by the mean free path, so we expect fewer collisions in regions of higher temperature where the density is lower. The total number of collisions in each cell is shown in Fig. 4.13; the profile is consistent with expectations. The Couette flow scenario was not part of the Garcia study.

Figs. 4.14 and 4.16 show covariances $\langle \delta n_k \delta n_{11} \rangle$ and $\langle \delta n_k \delta \mathbf{J}_{11} \rangle$ for the cases of equilibrium, medium temperature gradient and large temperature gradient in a single species gas. The covariance $\langle \delta n_k \delta n_{11} \rangle$ relates the fluctuation in number density of cell 11 to the number density fluctuation in cell k , where $n_k = N_k/V_{\text{cell}}$. The covariance $\langle \delta n_k \delta \mathbf{J}_{11} \rangle$ is similarly defined, but for fluid momentum $\mathbf{J}$ rather than number density. Agreement with their respective counterparts in the Garcia study is excellent, as comparisons with Figs. 4.15 and 4.17 show.

Many additional tests were run to validate the simulator, as previously indicated. Presentation of many of those results are postponed until Chapter 5.

Figure 4.12: Temperature versus cell position for Couette flow in a single species gas; $T_L = T_R = 1$. Two shear velocity gradients are shown along with the equilibrium case. Temperature increases toward the center of the system due to viscous heating.

Figure 4.13: Plot of the number of collisions versus cell position in a Couette flow system with $T_L = T_R = 1$ in a single species gas. It is consistent with expectations that the number of collisions decreases in regions of higher temperature where gas density is lower. The three cases shown correspond to the three cases in Fig. 4.12.

Figure 4.14: Plot of covariance of particle number density $\langle \delta n_k \delta n_{11} \rangle$ for different temperature gradients in a single species gas. The covariance relates number density fluctuations in cell 11 to those in each cell k in the system. For cell 11 the equilibrium component $\langle (\delta n_{11})^2 \rangle = 1$ is comparatively large and has been subtracted. Cell position is given as a multiple of the minimum mean free path.

Figure 4.15: Number-number covariance $\langle \delta n_k \delta n_{11} \rangle$ for cell 11 in the same scenarios as those shown in Fig. 4.14 above; reproduced from the Garcia 1986 paper.

Figure 4.16: Plot of covariance of particle number density and fluid momentum $\langle \delta n_k \delta \mathbf{J}_{11} \rangle$ versus cell position. See caption for Fig. 4.14.

Figure 4.17: Momentum-number covariance $\langle \delta n_k \delta \mathbf{J}_{11} \rangle$ for cell 11 as a function of cell k across the system, from the Garcia 1986 paper.

CHAPTER 5

ROLE OF FLUCTUATIONS IN FLUID VELOCITY MEASUREMENT

5.1 Introduction

In Section 1.3 of the Introduction, we demonstrated that when using the sample averaging measurement (SAM) method, we measured an anomalous fluid velocity. The velocity profiles were presented in Figs. 1.4 and 1.5; they are repeated below as Figs. 5.1 and 5.2 for ease of reference. These fluid velocities were measured in a closed system at the steady state, so from physical arguments they cannot represent actual fluid flows. Rather, they must be due to an erroneous implementation of the simulation method, a previously unknown limitation of the DSMC algorithm, or some unexpected physical effect. Based on physical arguments, our early assumption was that the phenomenon was due to a programming error or computational artifact with no physical basis, and much effort went into characterizing the effect and isolating its cause.

In Section 1.3 we claimed that its origin was a physical effect to which SAM was sensitive, not a merely computational issue. It is the purpose of this chapter to further justify and expand on that argument. The first task is to justify the claim that the phenomenon has its origin in physical effects rather than computational issues. For this we depend heavily on simulations. Following that, we analyze the SAM

method to determine the origin of the measurement error. Finally, we show with simulations that this analysis captures the essence of the anomalous fluid velocity.

Figure 5.1: Sample averaging measurement of mean fluid velocity, $\langle u^x \rangle_s$, for different temperature gradients. This indicates an anomalous flow from left to right that reaches a maximum at the center of the system. Evidently, increasing the number of samples results in a smoother plot but does not change the magnitude of the effect.

5.2 Determining the Origin of the Anomalous Velocity

In Section 4.5 we described the simulator validation process and showed that the simulator duplicated the results of an earlier fluctuations study; this gave confidence

Figure 5.2: Sample averaging measurement of mean fluid velocity, $\langle u^x \rangle_s$, for Couette flow with different shear velocity gradients. The anomalous fluid flow is from both sides toward the center.

that it correctly implemented the DSMC algorithm and contained no obvious flaws elsewhere in the program. Nonetheless, it remained possible that there were subtle programming errors, the simulation was incorrectly set up, or even that the DSMC algorithm itself had a previously unknown problem. A programming error or poor choice of simulation parameters were in fact viewed as by far the most likely causes. A series of simulations tested whether either of these were the cause of the anomalous flow. Section 4.5.1 lists many of those tests, and we will elaborate on some of them

below.

It is worth mentioning that at first all fluid velocities were measured exclusively with the SAM method. We did not recognize the important distinction between SAM and CAM until later; we had used SAM for velocity measurements merely because it seemed to be the most obvious way. Furthermore, as mentioned earlier, there is no distinction between SAM and CAM at equilibrium. For these reasons, some of our tests were unnecessary in retrospect, but at the time it was not at all obvious what was causing the measured nonzero fluid velocity.

The first tests were attempts at changing certain parameters in hope of observing changes in measured flow characteristics. For example, in the simulation of Fig. 5.3, we changed the random deviate generator seed to determine whether there was any noticeable effect. The seed is used to set the initial condition for the random deviate generator; a different seed leads to a different sequence of random deviates. The rationale was that the generator was used extensively by the DSMC algorithm in determining post-collision scattering angles, and any significant effect from changing the seed would indicate an unwanted sensitivity within the simulator. The effect was observable but not statistically meaningful.

One strong candidate cause of the apparent flow was a persistence of initial conditions, as a description of the system initialization may indicate. During initialization, the simulator assigned the N_Σ particle positions randomly but uniformly throughout the system. This was fine for simulations of equilibrium systems, but for a non-equilibrium simulation the resulting profile did not reflect the density gradient that accompanies a thermal gradient. Next, the initialization procedures randomly assigned the particle velocities in each cell according to the expected cell temperature and shear velocity; the temperature and shear velocity were calculated from the sim-

Figure 5.3: Sample averaging measurement (SAM) of mean fluid velocity, $\langle u_k^x \rangle_s$, for two different random deviate generator seeds (initialization values). The effect of this change did not have a statistically significant effect on the measured fluid velocity.

ulation parameters. The resulting velocity distribution within each cell was always Maxwell-Boltzmann, however, even though that is correct only at equilibrium. In the presence of a thermal gradient, a Chapman-Enskog distribution is the appropriate one. For these reasons, the simulator did not correctly initialize a non-equilibrium system to its steady state. To compensate, we allowed the system to relax to its steady state by running the simulation for many time steps without taking samples. Sampling began once the relaxation period was complete.

In the earliest simulations, such as the ones where we first noticed the anomalous flow, the relaxation period was quite short, on the order of 10^4 to 10^5 time steps, giving a total number of collisions on the order of 10^6 . Given the small magnitude of the measured flow, it seemed plausible that initial conditions were persisting, and that the magnitude would monotonically decrease with run time.

We performed several different simulations to test for this. The first, shown in Fig. 5.1, demonstrated that increasing the total sampling time results in a smoother plot but does not change the magnitude of the effect. A second set of simulations tested the effect of initializing the system with all particles in the same position in a single cell, i.e., a Dirac delta function position distribution, then allowing the system to relax for a time before starting sampling. Fig. 5.4 shows that there was no statistically significant memory of the initial position of the particles. Another simulation, not shown here, also initialized all particles at the same initial velocity, again with no statistically significant effect. Finally, we tried different relaxation times to test for a sensitivity to this simulation parameter, but as Fig. 5.5 this had no significant effect.

Although it became apparent that persistence of initial conditions was not causing a flow, one outcome of our concern was to instill a caution in the simulation methodology. In later tests we chose to allow the system to relax for a long time before sampling began, typically half of the total simulation time. The rationale was that fluctuations are generally a small magnitude phenomenon, and we wanted to entirely eliminate initial conditions as a variable.

At this point it was not entirely clear whether or not the simulator was erroneously measuring a fluid velocity or whether the problem lay with the simulation algorithm. Alternatively, there was a small possibility that some unexpected physical

Figure 5.4: Sample averaging measurement of $\langle u_k^x \rangle_s$ for different initial particle distributions. In each case, all $N_\Sigma = 2000$ particles were assigned a Maxwell-Boltzmann velocity distribution and a Dirac delta function position distribution within a system of length L . After the relaxation time of 2×10^6 time steps had elapsed, 2×10^6 samples were taken in all three cases, but no statistically significant difference between them is observable.

effect was entering, although it was impossible on physical grounds that an actual flow was occurring in a closed system. The next few tests helped narrow the range of possibilities somewhat.

First, the simulator was modified to obtain a pressure profile by measuring the momentum flux through the left side of each cell. The flux of momentum leaving the

Figure 5.5: Sample averaging measurement of $\langle u_k^x \rangle_s$ for different system relaxation times. In each case, the system was initialized in the default manner, but the number of elapsed time steps before sampling began differed in each case. In each case, 5×10^6 samples were taken. No statistically significant difference is observable.

cell at its left side was taken to be positive, and the flux in from the cell on the left was taken to be negative. The pressure in the cell at the extreme left was not plotted, since it was the pressure on the left wall where a temperature slip occurred. (This slip can be seen at both the left and right walls in Fig. 4.10; i.e., the temperature of a boundary cell does not match the temperature of the adjacent thermal wall in the presence of a thermal gradient, as originally predicted by Maxwell.) By measuring

pressure, it was possible to verify that no pressure gradient existed in the system, something that would necessary accompany an actual fluid flow. Fig. 5.6 shows that the pressure was constant, as expected in a closed system.

Figure 5.6: Fluid pressure as a function of cell position in a system with a large temperature gradient ($\Delta T = 4$) and a single species gas. The pressure is constant throughout the system within statistical error, as expected from physical arguments.

Second, to check for the possibility that simulator was erroneously moving particles or placing them in the wrong cell after they left a cell, particles were counted as they entered or left each cell. This verified continuity of particle movement between cells and conservation of particle number for each cell. This test, not shown, verified that particles were being moved between cells correctly and no artificial flow was

resulting from a related simulator programming error.

The two preceding tests gave the expected results, which strongly indicated that no real flow existed. To verify this, we attempted to measure the fluid velocity by an alternative method to see whether the two methods yielded similar measurements. This ‘new’ method adhered to our definition of cumulative average measurement (CAM). Now the measured flow disappeared altogether, as Fig. 5.7 shows. This test in particular gave the first unambiguous evidence that the anomalous flow was somehow connected with the SAM method and was not due to an actual flow being erroneously created in the system.

Fig. 5.8 shows the outcome of a simulation that determined the effect of reversing the wall temperatures. In all other simulations, the convention was to set $T_L \leq T_R$, where T_L and T_R were the temperatures of the left and right walls, respectively. Apparently, setting $T_L > T_R$ reversed the direction of the flow; the anomalous fluid velocity is always from cold to hot.

It was possible that the anomalous flow was somehow affected by the coarseness of the sampling interval. For example, a given particle might remain in a cell for too few sample times, or the DSMC algorithm might somehow be sensitive to time step duration under the highly non-equilibrium conditions of our system. We tried different time steps to test for a sensitivity, starting at the algorithmically determined ‘safe’ value and additionally trying smaller values. As Fig. 5.9 shows, varying the time step had no meaningful effect.

The last test we will describe here that helped determine the origin of the anomalous flow with SAM involved changing the total number of particles in the system. Fig. 5.10 shows that the magnitude of the effect decreases as the number of simulation particles increases; it likewise increases when the number of particles

Figure 5.7: Plot of cumulative average measurement of mean fluid velocity $\langle \mathbf{u}_k \rangle_c$ for equilibrium, medium temperature gradient and large temperature gradient cases. The error bars show the fluctuations in fluid velocity across the system are not statistically significant, giving strong evidence that there is no nonphysical flow.

decreases. Measurements with CAM show no such dependence, so clearly SAM is sensitive to the number of particles being sampled.

The dependence of the magnitude of the velocity on the number of particles, together with the fact that the anomalous flow occurred only under non-equilibrium conditions, indicated that non-equilibrium fluctuations might be playing an active role in the phenomenon. Such fluctuations are a physical phenomenon, and would

Figure 5.8: SAM fluid velocity $\langle u_k \rangle_s$ for a large temperature gradient in a single species gas. The two plots show the same magnitude of the gradient, but in opposite directions. Reversing the wall temperatures caused the measured velocity to also reverse, again from the lower to higher temperature wall.

be measurable in any particle based simulation, not just those that use DSMC. We will analyze the role of fluctuations in SAM in the following section, and this will highlight the reason that SAM and CAM can yield different measurements under non-equilibrium conditions.

Figure 5.9: Sample averaging measurement of mean fluid velocity $\langle \mathbf{u}_k \rangle_s$ with different time steps but constant total settling and sample times. For example, the number of samples was doubled when the time step was halved, so that the total sample time was the same. The time step has no obvious effect on the magnitude of the measured flow. The wall temperatures were $T_L = 1$, $T_R = 5$.

5.3 Analysis

For a single species, CAM measures mean particle velocity according to Eqn. (2.1):

$$\langle \mathbf{u}_k \rangle_c = \frac{\frac{1}{S} \sum_{j=1}^S \sum_{i \in k}^{N_k(t_j)} \mathbf{v}_i(t_j)}{\frac{1}{S} \sum_{j=1}^S N_k(t_j)} = \frac{\langle \mathbf{J}_k \rangle}{\langle M_k \rangle}. \quad (5.1)$$

Figure 5.10: Sample averaging measurement (SAM) of mean fluid velocity, $\langle u_k^x \rangle_s$, for different total particle numbers; $T_L = 1$ and $T_R = 5$. The measured anomalous fluid flow decreases as $\langle N_k \rangle$ increases, in agreement with prediction from theory that $\langle (\delta \mathbf{u})^2 \rangle \propto 1/\langle N \rangle$ (see Eqn. (2.32)).

This is a straightforward measure of average fluid velocity in a cell that yields correct measurements under both equilibrium and non-equilibrium conditions, so there is no need to elaborate on it further.

For SAM, which measures mean particle velocity at each sample and then averages these means over all samples according to Eqn. (2.2) or (2.6), we start by

writing

$$\langle \mathbf{u}_k \rangle_s = \left\langle \frac{\mathbf{J}_k}{M_k} \right\rangle = \left\langle \frac{\mathbf{J}_k}{mN_k} \right\rangle. \quad (5.2)$$

Note that the instantaneous total momentum, $\mathbf{J}_k$, may be decomposed into a sum of the mean momentum $\langle \mathbf{J}_k \rangle$ and a fluctuation $\delta \mathbf{J}_k$ about the mean as in Eqn. (2.15); similarly, $N_k = \langle N_k \rangle + \delta N_k$. Dropping the subscripts because cell k is implied and using the binomial expansion,

$$\begin{aligned} \langle \mathbf{u} \rangle_s &= \left\langle \frac{\langle \mathbf{J} \rangle + \delta \mathbf{J}}{m(\langle N \rangle + \delta N)} \right\rangle \\ &= \left\langle \frac{\langle \mathbf{J} \rangle + \delta \mathbf{J}}{m\langle N \rangle} \left[1 - \frac{\delta N}{\langle N \rangle} + \left(\frac{\delta N}{\langle N \rangle} \right)^2 + O([\delta N]^3) \right] \right\rangle \\ &= \frac{\langle \mathbf{J} \rangle}{\langle M \rangle} - \frac{\langle \delta \mathbf{J} \delta N \rangle}{m\langle N \rangle^2} + \frac{\langle \mathbf{J} \rangle \langle (\delta N)^2 \rangle}{m\langle N \rangle^3} + O([\delta X]^3) \\ &= \langle \mathbf{u} \rangle_c \left(1 + \frac{\langle (\delta N)^2 \rangle}{\langle N \rangle^2} \right) - \frac{\langle \delta \mathbf{J} \delta N \rangle}{m\langle N \rangle^2} + O([\delta X]^3), \end{aligned} \quad (5.3)$$

where $O([\delta X]^3)$ indicates terms of cubic order in the fluctuations. This gives an alternative expression for SAM that relates it to the CAM definition.

The foregoing assumed a single species fluid with molecular mass m . To generalize to A species we follow similar reasoning, but there are many new cross terms. We start with Eqn. (5.2) and follow the notation of Section 2.2, which identifies a particular species with the subscript a . Observing that $\mathbf{J} = \sum_{a=1}^A \mathbf{J}_a$ is the instantaneous total fluid momentum and $M = \sum_{a=1}^A M_a$ is the instantaneous total fluid mass in the cell,

$$\begin{aligned} \langle \mathbf{u} \rangle_s &= \left\langle \frac{1}{M} \left(\sum_{a=1}^A \mathbf{J}_a \right) \right\rangle \\ &= \left\langle \frac{1}{\langle M \rangle + \delta M} \sum_{a=1}^A (\langle \mathbf{J}_a \rangle + \delta \mathbf{J}_a) \right\rangle. \end{aligned} \quad (5.4)$$

Now

$$\frac{1}{\langle M \rangle + \delta M} = \frac{1}{M} \left(1 + \frac{\delta M}{\langle M \rangle} \right)^{-1} \cong \frac{1}{\langle M \rangle} \left[1 - \frac{\delta M}{\langle M \rangle} + \left(\frac{\delta M}{\langle M \rangle} \right)^2 \right], \quad (5.5)$$

where

$$\delta M = \delta \left(\sum_{a=1}^A N_a m_a \right) = \sum_a (\delta N_a) m_a$$

and

$$(\delta M)^2 = \sum_{a=1}^A \sum_{b \geq a}^A (\delta N_a \delta N_b) m_a m_b.$$

In the following, we will replace the double summation $\sum_{a=1}^A \sum_{b \geq a}^A$ with a shorthand notation $\sum_{a,b}$ for compactness. Substituting the above expressions for δM and $(\delta M)^2$ into Eqn. (5.5),

$$\frac{1}{\langle M \rangle + \delta M} \cong \frac{1}{\langle M \rangle} \left[1 - \frac{\sum_a (\delta N_a) m_a}{\langle M \rangle} + \frac{\sum_{a,b} (\delta N_a \delta N_b) m_a m_b}{\langle M \rangle^2} \right]. \quad (5.6)$$

For $\mathbf{J}$ we have

$$\sum_a (\langle \mathbf{J}_a \rangle + \delta \mathbf{J}_a) = \langle \mathbf{J} \rangle + \sum_a ((\delta N_a) m_a \bar{\mathbf{v}}_a). \quad (5.7)$$

Substitute Eqns. (5.6) and (5.7) into Eqn. (5.4):

$$\begin{aligned} \langle \mathbf{u} \rangle_s &= \left\langle \frac{1}{\langle M \rangle} \left[1 - \frac{\sum_a (\delta N_a) m_a}{\langle M \rangle} + \frac{\sum_{a,b} (\delta N_a \delta N_b) m_a m_b}{\langle M \rangle^2} \right] \left[\langle \mathbf{J} \rangle + \sum_a (\delta N_a) m_a \bar{\mathbf{v}}_a \right] \right\rangle \\ &= \left\langle \frac{\langle \mathbf{J} \rangle}{\langle M \rangle} \right\rangle - \left\langle \frac{\langle \mathbf{J} \rangle}{\langle M \rangle^2} \sum_a (\delta N_a) m_a \right\rangle + \left\langle \frac{\langle \mathbf{J} \rangle}{\langle M \rangle^3} \sum_a (\delta N_a \delta N_b) m_a m_b \right\rangle \\ &\quad + \left\langle \frac{1}{\langle M \rangle} \sum_a (\delta N_a) m_a \bar{\mathbf{v}}_a \right\rangle + \left\langle \frac{1}{\langle M \rangle^2} \sum_{a,b} (\delta N_a \delta N_b) m_a m_b \bar{\mathbf{v}}_a \right\rangle + O([\delta X]^3) \\ &= \frac{\langle \mathbf{J} \rangle}{\langle M \rangle} - \frac{\langle \mathbf{J} \rangle}{\langle M \rangle^2} \sum_a \langle \delta N_a \rangle m_a + \frac{\langle \mathbf{J} \rangle}{\langle M \rangle^3} \sum_{a,b} \langle \delta N_a \delta N_b \rangle m_a m_b + \frac{1}{\langle M \rangle} \sum_a \mathbf{J}_a \langle \delta N \rangle \\ &\quad + \frac{1}{\langle M \rangle^2} \sum_{a,b} \langle \delta N_a \delta \mathbf{J}_b \rangle m_a + O([\delta X]^3) \\ &= \frac{\langle \mathbf{J} \rangle}{\langle M \rangle} \left(1 + \frac{1}{\langle M \rangle^2} \sum_{a,b} \langle \delta N_a \delta N_b \rangle m_a m_b \right) - \frac{1}{\langle M \rangle^2} \sum_{a,b} \langle \delta N_a \delta \mathbf{J}_b \rangle m_a + O([\delta X]^3). \end{aligned}$$

But $\langle \mathbf{J} \rangle / \langle M \rangle$ is just the definition for the CAM velocity (see (5.1)), so the alternative SAM definition for mean fluid velocity reduces to

$$\begin{aligned} \langle \mathbf{u} \rangle_s &= \langle \mathbf{u} \rangle_c \left(1 + \frac{1}{\langle M \rangle^2} \sum_{a=1}^A \sum_{b \geq a}^A \langle \delta N_a \delta N_b \rangle m_a m_b \right) \\ &\quad - \frac{1}{\langle M \rangle^2} \sum_{a=1}^A \sum_{b \geq a}^A \langle \delta N_a \delta \mathbf{J}_b \rangle m_a + O([\delta X]^3). \end{aligned} \quad (5.8)$$

For a single species gas, $A = 1$ and the expression reduces to Eqn. (5.3), as it must.

5.4 Simulation Tests of the Analysis

We now consider some simulations to test the theory in Section 5.3. Only single species gases will be considered here; verification of the multiple species case remains a task for future study.

Fig. 5.11 shows the SAM fluid velocity for two different total particle numbers. According to fluctuations theory, $\langle (\delta \mathbf{u})^2 \rangle \propto 1/\langle N \rangle$ (see Eqn. (2.32)), so if the anomalous fluid velocity measured by SAM is indeed due to fluctuations then the $N_\Sigma = 2000$ plot should be nearly $5000/2000 = 2.5$ times the magnitude of the $N_\Sigma = 5000$ plot. To test this, a ‘theory’ plot was generated from the data for the 5000 particle plot. Comparison with the 2000 particle plot agrees well with this expectation.

To test the theory more specifically, in particular Eqn. (5.3) that relates the SAM and CAM definitions, we measured the fluctuation variance $\langle (\delta N)^2 \rangle$ and covariance $\langle \delta \mathbf{J} \delta N \rangle$ during the simulations, using the procedure described in Section 4.4.3. Eqn. (5.3) states that

$$\langle \mathbf{u} \rangle_s \cong \langle \mathbf{u} \rangle_c \left(1 + \frac{\langle (\delta N)^2 \rangle}{\langle N \rangle^2} \right) - \frac{\langle \delta \mathbf{J} \delta N \rangle}{m \langle N \rangle^2}. \quad (5.9)$$

In a closed system at the steady state, $\langle \mathbf{u} \rangle_c = 0$, so the above reduces to

$$\langle \mathbf{u} \rangle_s \cong \frac{\langle \delta \mathbf{J} \delta N \rangle}{m \langle N \rangle^2}. \quad (5.10)$$

Figure 5.11: Sample averaging measurement (SAM) of mean fluid velocity, $\langle u_k^x \rangle_s$, for different particle numbers; $T_L = 1$ and $T_R = 5$. The theory plot for the $N_\Sigma = 2000$ case is estimated by multiplying the $N_\Sigma = 2000$ plot by $5000/2000 = 2.5$; it matches the actual plot quite well, within statistical error.

Fig. 5.12 shows the large temperature gradient case ($T_L = 1$ and $T_R = 5$) and large shear velocity Couette flow case ($u_L^y = 0$ and $u_R^y = 4$) plotted with expectations from Eqn. (5.10). Agreement is very good.

As can be seen from Fig. 5.7 which shows the CAM fluid velocity for different temperature gradients, it is apparent that the CAM velocity profile has fluctuations, although they are not statistically significant. Thus, $\langle \mathbf{u} \rangle_c$ in Eqn. (5.9) is not quite

Figure 5.12: Sample averaging measurement of $\langle u_k^x \rangle_s$ for the cases of large temperature gradient and large shear velocity (Couette flow), plotted with the measured covariance term, $-\frac{\langle \delta \mathbf{J} \delta N \rangle}{m \langle N \rangle^2}$ from Eqn. (5.9). Evidently the covariance term captures most of the SAM measurement bias in systems having $\langle \mathbf{u} \rangle_c = 0$.

zero, and thus the $\langle (\delta N)^2 \rangle$ term also contributes. Fig. 5.13 includes the contributions from both the variance and covariance terms. Comparison with Fig. 5.12 shows that the $\langle \delta \mathbf{J} \delta N \rangle$ term by far is the dominant one, consistent with expectations for systems having $\langle \mathbf{u} \rangle_c = 0$.

We can rewrite Eqn. (5.3) in a way that relates it to mode-coupling theory, theory that quantitatively explains the origin of non-equilibrium correlation of fluc-

Figure 5.13: Sample averaging measurement of $\langle u_k^x \rangle_s$ for the large temperature gradient ($T_L = 1$, $T_R = 5$) and shear velocity gradient ($u_L^y = 0$, $u_R^y = 4$) cases, plotted together with the measured variance term, $\langle \mathbf{u} \rangle_c \frac{\langle (\delta N)^2 \rangle}{\langle N \rangle^2}$, and covariance term, $-\frac{\langle \delta \mathbf{J} \delta N \rangle}{m \langle N \rangle^2}$, from Eqn. (5.9). Comparison with Fig. 5.12 shows no significant difference, indicating that the covariance term is by far the dominant one.

tuations. Since $\langle \mathbf{J} \rangle = m \langle N \rangle \langle \mathbf{u} \rangle_c$ and thus

$$\delta \mathbf{J} = m \langle N \rangle \delta \mathbf{u} + m \langle \mathbf{u} \rangle_c \delta N,$$

we can substitute this expression for $\delta \mathbf{J}$ into the $\langle \delta \mathbf{J} \delta N \rangle$ term of Eqn. (5.3) to obtain

$$\langle \mathbf{u} \rangle_s = \langle \mathbf{u} \rangle_c - \frac{\langle \delta \mathbf{u} \delta N \rangle}{\langle N \rangle} + O(\delta X^3). \quad (5.11)$$

The covariance term on the right hand side is zero at equilibrium and in the thermodynamic limit $\langle N \rangle \rightarrow \infty$, but it is nonzero for finite, non-equilibrium systems [10]. The $\langle \delta \mathbf{u} \delta N \rangle$ covariance is in itself theoretically interesting, since the asymmetric Brillouin peaks observed in light scattering for a fluid subjected to a temperature gradient originate from this term [1, 2].

This earlier theoretical work [2] also predicts that the non-equilibrium correlation of fluctuations, which produces a contribution due to the $\langle \delta u_k^x \delta N_k \rangle$ term in Eqn. (5.11), approximately varies as $x(L-x)\Delta T$, where L is the length of the system and x is the position within the system. Relating this to the $\langle u_k^x \rangle_s$ profiles in Fig. 5.1, the expectation is that the profiles will vary approximately quadratically in x , and have a magnitude that is proportional to the temperature gradient. Our results are consistent with these predictions.

In the next chapter we will show that the variance and covariance of non-equilibrium fluctuations lead to an analogous ambiguity in the definition of temperature. That is, it is possible to define both a CAM and SAM temperature, and they are distinguishable in the presence of large temperature gradients.

CHAPTER 6

ROLE OF FLUCTUATIONS IN TEMPERATURE MEASUREMENT

6.1 Theory

In Chapter 2 we obtained CAM and SAM definitions for fluid velocity and temperature (see Sections. 2.1 through 2.3):

$$\langle \mathbf{u}_k \rangle_c = \frac{\langle \mathbf{J}_k \rangle}{\langle M_k \rangle}, \quad (6.1)$$

$$\langle \mathbf{u}_k \rangle_s = \left\langle \frac{\mathbf{J}_k}{M_k} \right\rangle, \quad (6.2)$$

$$\langle T_k \rangle_c = \frac{2}{3k_B} \left[\frac{\langle K_k \rangle}{\langle N_k \rangle} - \frac{1}{2} \frac{\langle M_k \rangle}{\langle N_k \rangle} \left(\frac{\langle \mathbf{J}_k \rangle}{\langle M_k \rangle} \right)^2 \right], \quad (6.3)$$

$$\langle T_k \rangle_s = \frac{2}{3k_B} \left[\left\langle \frac{K_k}{N_k} \right\rangle - \frac{1}{2} \left\langle \frac{M_k}{N_k} \right\rangle \left\langle \frac{\mathbf{J}_k}{M_k} \right\rangle^2 \right] \quad (6.4)$$

where $K_k \equiv \sum_{i \in k}^{N_k} \frac{1}{2} m_i \mathbf{v}_i^2$ is the instantaneous total kinetic energy (center of mass frame) in cell k .

Comparing the CAM and SAM temperature definitions with their fluid velocity counterparts, it is apparent that the SAM definition of temperature will have a bias that is similar in form to that of fluid velocity. That is, in the SAM definition if we let $K = \langle K \rangle + \delta K$ and $N = \langle N \rangle + \delta N$, then we can obtain the SAM definition in

terms of CAM by the same reasoning as was used in deriving Eqn. (5.3):

$$\begin{aligned}
\langle T \rangle_s &= \frac{2}{3k_B} \left\langle \frac{\langle K \rangle + \delta K}{\langle N \rangle + \delta N} \right\rangle \\
&= \frac{2}{3k_B} \left\langle \frac{\langle K \rangle + \delta K}{\langle N \rangle} \left[1 - \frac{\delta N}{\langle N \rangle} + \left(\frac{\delta N}{\langle N \rangle} \right)^2 + O([\delta N]^3) \right] \right\rangle \\
&= \frac{2}{3k_B} \left[\frac{\langle K \rangle}{\langle N \rangle} - \frac{\langle \delta K \delta N \rangle}{\langle N \rangle^2} + \frac{\langle K \rangle \langle (\delta N)^2 \rangle}{\langle N \rangle^3} \right] + O([\delta X]^3) \\
&= \langle T \rangle_c \left(1 + \frac{\langle (\delta N)^2 \rangle}{\langle N \rangle^2} \right) - \frac{2}{3k_B} \frac{\langle \delta K \delta N \rangle}{\langle N \rangle^2} + O([\delta X]^3). \tag{6.5}
\end{aligned}$$

At equilibrium, the average kinetic energy per molecule is the same for all molecules, independent of molecular mass. Thus, Eqn. (6.5) holds for both single- and multi-species fluids.

We now show that the CAM and SAM definitions are equivalent at equilibrium. From the definition above for the instantaneous total kinetic energy, $3k_B T/2 = K/N$ defines the instantaneous temperature T for a monatomic gas. This provides an expression for the fluctuation δK :

$$\delta K = \frac{3k_B}{2} \delta(NT) = \frac{3k_B}{2} (N\delta T + T\delta N)$$

and so

$$\begin{aligned}
\frac{2}{3k_B} \frac{\langle \delta K \delta N \rangle}{\langle N \rangle^2} &= \frac{\langle \langle N \rangle \delta T \delta N \rangle}{\langle N \rangle^2} + \frac{\langle \langle T \rangle (\delta N)^2 \rangle}{\langle N \rangle^2} \\
&= \frac{\langle \delta T \delta N \rangle}{\langle N \rangle} + \langle T \rangle_c \frac{\langle (\delta N)^2 \rangle}{\langle N \rangle^2}.
\end{aligned}$$

The final step in the above identifies $\langle T \rangle$ with the CAM definition because that makes the next step more straightforward; it is reasonable as long as we assume that $\langle T \rangle_c$ and $\langle T \rangle_s$ differ by at most a small amount at equilibrium.

Substituting the above result into Eqn. (6.5),

$$\begin{aligned}\langle T \rangle_s &= \langle T \rangle_c \left(1 + \frac{\langle (\delta N)^2 \rangle}{\langle N \rangle^2} \right) - \left(\frac{\langle \delta T \delta N \rangle}{\langle N \rangle} + \langle T \rangle_c \frac{\langle (\delta N)^2 \rangle}{\langle N \rangle^2} \right) + O([\delta X]^3) \\ &= \langle T \rangle_c - \frac{\langle \delta T \delta N \rangle}{\langle N \rangle} + O([\delta X]^3).\end{aligned}\quad (6.6)$$

At equilibrium or in the thermodynamic limit, T and N are independent [10], so $\langle \delta T \delta N \rangle = 0$. It is clear from Eqn. (6.6) that the CAM and SAM definitions are equivalent under these conditions. But at non-equilibrium the covariance $\langle \delta T \delta N \rangle$ may be nonzero, biasing the SAM definition of temperature.

Because of the usual association of temperature with the mean kinetic energy per molecule over a long time or large ensemble of identical systems, we will prefer the form of Eqn. (6.5) in simulations and comparisons of simulation results with the above analysis. Additionally, under non-equilibrium conditions the definition of $\langle \delta T \delta N \rangle$ is subject to the same kind of ambiguity as the definition of T , but $\langle \delta K \delta N \rangle$ in Eqn. (6.5) is unambiguous under all conditions.

6.2 Simulation Results

In the previous section we showed quantitatively that SAM and CAM both yield a correct temperature measurement at equilibrium. Fig. 6.1 demonstrates this with the simulation of a closed system at equilibrium; within statistical error, the plot of the difference $\langle T \rangle_c - \langle T \rangle_s$ is zero everywhere.

Section 6.1 also showed that SAM is biased at non-equilibrium. Taking the difference between $\langle T \rangle_s$ and $\langle T \rangle_c$ (see Eqns. (6.3) and (6.4)),

$$\langle T \rangle_s - \langle T \rangle_c \cong \langle T \rangle_c \frac{\langle (\delta N)^2 \rangle}{\langle N \rangle^2} - \frac{2}{3k_B} \frac{\langle \delta K \delta N \rangle}{\langle N \rangle^2}.\quad (6.7)$$

At equilibrium, Fig. 6.1 shows no visible difference in the magnitudes of these terms.

Figure 6.1: Difference between SAM and CAM temperature measurements of an equilibrium, closed system with thermal walls at both ends and $T_L = T_R = 1$. The variance term (bottom plot, crosses) and covariance term (top plot, asterisks) in Eqn. (6.7) are shown to indicate their relative magnitude; they must be entirely due to equilibrium fluctuations since the system is at equilibrium.

At non-equilibrium, however, the difference between the variance and covariance terms in Eqn. (6.7) distinguish $\langle T \rangle_s$ from $\langle T \rangle_c$.

That the two fluctuation terms are quantitatively equal can be seen by observing that CAM and SAM are equivalent at equilibrium, and so the right hand side in Eqn. (6.7) must vanish. This requires both terms to be equal.

In a system at equilibrium such as that used for Fig. 6.1, the variance and

covariance are entirely due to equilibrium fluctuations. However, non-equilibrium fluctuations are the source of the measurement bias in SAM. A SAM-CAM difference plot with the variance and covariance terms also shown will allow determining the relative magnitudes of the equilibrium and non-equilibrium fluctuations in a non-equilibrium system. Fig. 6.2 shows the case of a closed system with thermal walls at temperatures $T_L = 1$ and $T_R = 3$, with both fluctuation terms plotted separately. Fig. 6.3 shows the same case, but with the equilibrium contribution to the fluctuation terms subtracted. Apparently, the equilibrium and non-equilibrium contributions are approximately the same.

Finally, Fig. 6.4 shows the simulation results for a non-equilibrium closed system with two temperature gradients, a ‘medium’ gradient with $T_L = 1$ and $T_R = 3$ and a large gradient with $T_L = 1$ and $T_R = 5$. The right hand side of Eqn. (6.7) is plotted along with the SAM-CAM temperature difference. Apparently, Eqn. (6.7) captures quite well the essence of the measurement bias of SAM.

Figure 6.2: SAM-CAM temperature difference in a closed system with thermal walls at both ends and $T_L = 1$ and $T_R = 3$. The fluctuation terms in Eqn. (6.7) are shown to indicate their relative magnitude; they include both the equilibrium and non-equilibrium contributions.

Figure 6.3: SAM-CAM temperature difference for the same conditions as were used for Fig. 6.2, after subtracting equilibrium contributions to the fluctuation terms. Apparently the equilibrium and non-equilibrium contributions are approximately equal.

Figure 6.4: Difference between the SAM and CAM temperatures, $\langle T_k \rangle_s - \langle T_k \rangle_c$, in a closed system with a ‘medium’ gradient of $T_L = 1$ and $T_R = 3$ and a large gradient of $T_L = 1$ and $T_R = 5$. The sum of the variance $\langle (\delta N)^2 \rangle$ and covariance $\langle \delta K \delta N \rangle$ terms from Eqn. (6.7) is also shown for each case; agreement with expectations is excellent.

6.3 Measurement Bias from Average of Instantaneous Temperatures

In Section 2.3 we gave two SAM definitions of temperature and stated that one had an inherent bias, even at equilibrium. That biased definition was denoted as T' (see Eqn. (2.11)):

$$\langle T'_k \rangle_s = \frac{2}{3k_B} \left[\left\langle \frac{K_k}{N_k} \right\rangle - \frac{1}{2} \left\langle \frac{M_k}{N_k} \mathbf{u}_k^2 \right\rangle \right]. \quad (6.8)$$

The SAM definition that we claimed to be correct, at least at equilibrium, is

$$\langle T_k \rangle_s = \frac{2}{3k_B} \left[\left\langle \frac{K_k}{N_k} \right\rangle - \frac{1}{2} \left\langle \frac{M_k}{N_k} \right\rangle \langle \mathbf{u}_k \rangle_s^2 \right]. \quad (6.9)$$

The second term in Eqn. (6.8) is positive definite, so it is clear that the SAM definition $\langle T' \rangle_s$ yields a temperature measurement that is slightly less than expected. Fig. 6.5 confirms this.

The simulation for Fig. 6.5 used a closed system at equilibrium with thermal walls at both ends, each set to $T = 1$. The fluid temperature should therefore also have temperature $T = 1$. The CAM definition is unbiased both at equilibrium and non-equilibrium, so it was the reference. Fig. 6.5 shows that CAM yielded the expected temperature, as also did the SAM definition from Eqn. (6.9). It remains to determine the origin of the bias in $\langle T' \rangle_s$.

Starting from Eqns. (6.8) and (6.9) we can compare the two SAM definitions with help from statistical mechanics [22]. Without reducing the generality of the conclusion, all particles can be taken to have the same mass m , thus simplifying the analysis. Then $M_k/N_k = m$ and $K_k/N_k = \frac{1}{2}m\langle \mathbf{v}^2 \rangle_k$, where the particle velocities $\mathbf{v}$ are measured in cell k . Noting that the instantaneous fluid velocity $\mathbf{u}_k$ is also the

Figure 6.5: Comparison of the two definitions of the SAM temperature, with the unbiased CAM temperature for reference. The system was closed, and had thermal walls with $T_L = T_R = 1$. The SAM temperature definition $\langle T' \rangle_s$ yielded a measurement that was lower than expected by approximately $1/\langle N_k \rangle = 0.01$.

instantaneous mean particle velocity $\bar{\mathbf{v}}_k$, Eqns. (6.8) and (6.9) reduce to

$$\langle T'_k \rangle_s = \frac{2}{3k_B} \left[\frac{1}{2} \langle m \mathbf{v}^2 \rangle - \frac{1}{2} \langle m \mathbf{u}_k^2 \rangle \right] = \frac{m}{3k_B} [\langle \mathbf{v}^2 \rangle - \langle \mathbf{u}_k^2 \rangle], \quad (6.10)$$

$$\langle T_k \rangle_s = \frac{m}{3k_B} [\langle \mathbf{v}^2 \rangle - \langle \mathbf{u}_k \rangle_s^2]. \quad (6.11)$$

Then

$$\begin{aligned}
\langle T_k \rangle_s - \langle T'_k \rangle_s &= \frac{m}{3k_B} [\langle \mathbf{v}^2 \rangle - \langle \mathbf{u}_k \rangle^2 - \langle \mathbf{v}^2 \rangle + \langle \mathbf{u}_k^2 \rangle] \\
&= \frac{m}{3k_B} [\langle \mathbf{u}_k^2 \rangle - \langle \mathbf{u}_k \rangle^2] = \frac{m}{3k_B} \langle (\mathbf{u}_k - \langle \mathbf{u}_k \rangle)^2 \rangle \\
&= \frac{m}{3k_B} \langle (\delta \mathbf{u}_k)^2 \rangle.
\end{aligned} \tag{6.12}$$

That is, the difference between the two measurement methods is just the variance of fluid velocity, which is also the mean square fluctuation. Evidently, the bias in $\langle T' \rangle_s$ arises from removing fluctuations of the center of mass velocity.

From Eqn. (2.32), the mean square fluctuation in fluid velocity is proportional to $1/\langle N_k \rangle$. From equilibrium statistical mechanics, we can use the canonical ensemble to determine the mean square fluctuation. The energy of the fluid velocity fluctuation in cell k is just $\delta E = \frac{1}{2} m N_k (\delta \mathbf{u}_k)^2$, so from the canonical ensemble the probability of a fluctuation $\delta \mathbf{u}_k$ is

$$P(\delta \mathbf{u}_k) = \exp(-\delta E/k_B T_k) = \exp(-M_k (\delta \mathbf{u}_k)^2 / 2k_B T_k). \tag{6.13}$$

So that statistical reasoning applies, we assume T_k is measured in the limit of ensemble size $S \rightarrow \infty$.

Note that the probability $P(\delta \mathbf{u}_k)$ has the form of a Gaussian distribution, $\exp(-(\delta \mathbf{u}_k)^2 / 2\sigma_{\mathbf{u}}^2)$, where $\sigma_{\mathbf{u}}^2 = \langle (\delta \mathbf{u}_k)^2 \rangle$ is the variance of $\mathbf{u}_k$. Comparing with Eqn. (6.13) and associating T_k with the SAM temperature measurement $\langle T_k \rangle_s$ in the limit of a very large ensemble,

$$\sigma_{\mathbf{u}}^2 = \langle (\delta \mathbf{u}_k)^2 \rangle = \frac{k_B T_k}{m \langle N_k \rangle} \cong \frac{k_B \langle T_k \rangle_s}{m \langle N_k \rangle}.$$

Using this result in Eqn. (6.12),

$$\langle T_k \rangle_s - \langle T'_k \rangle_s = \frac{m}{3k_B} \left(\frac{3k_B \langle T_k \rangle_s}{m \langle N_k \rangle} \right) = \frac{\langle T_k \rangle_s}{\langle N_k \rangle}.$$

Rearranging the above shows that

$$\langle T'_k \rangle_s = \langle T_k \rangle_s \left(1 - \frac{1}{\langle N_k \rangle} \right). \quad (6.14)$$

In the simulation for Fig. 6.5, there were 2000 particles and 20 cells, so $\langle N_k \rangle = 100$ in the limit of $S \rightarrow \infty$. Since $\langle T_k \rangle_s = 1$, we expect that $\langle T'_k \rangle_s = 0.99$. The result shown in Fig. 6.5 is very near to this value.

In the more general case where the fluid has multiple species with different masses, the above reasoning applies to each species individually. However, Eqn. (6.14) makes no reference to the species mass or any other species dependent quantity. Thus, the total bias in $\langle T'_k \rangle_s$ for a fluid of an arbitrary number of species will be given by Eqn. (6.14), where N_k is now the total number of particles of all species in cell k .

In this chapter and the previous one, we have shown that correlations of non-equilibrium fluctuations cause ambiguity in the definitions of fluid velocity and temperature, and so it is necessary to make a distinction between the CAM and SAM definitions of these quantities. At equilibrium, the two definitions are equivalent; at non-equilibrium, it is necessary to specify which definition is being used by a molecular simulation.

The next chapter shows a very different situation where correlations between fluctuating variables can lead to incorrect measurements. Unlike the cases we have considered so far, this one arises if a simulation algorithm unintentionally introduces correlations between fluctuations that do not exist in the corresponding physical system. It is a purely computational effect, and it can occur even at equilibrium.

CHAPTER 7

FLUCTUATIONS AND RESERVOIR IMPLEMENTATION

7.1 Nature of the Anomalous Fluid Flow with Reservoirs

In Section 1.5 we demonstrated that SAM measures an anomalous fluid velocity near a particle reservoir when the reservoir algorithm generates the same number of particles at each time. Furthermore, the apparent velocity gradually decreases as the distance from the reservoir, disappearing altogether at the center. Fig. 7.1 shows this clearly, and also demonstrates that the effect is specific to SAM; the accompanying CAM plot shows that $\langle \mathbf{u} \rangle_c = 0$ everywhere, as expected for a system at equilibrium.

Given that the effect occurs with SAM but not at all with CAM, there is a striking correspondence with the anomalous fluid flow we described in Chapter 5. From experience with the phenomena there, it was reasonable to suspect that fluctuations were somehow causing SAM to obtain an incorrect mean velocity. Nonetheless, a crucial difference existed in that the system with a reservoir contained no temperature gradient, a factor that was necessary to the effect we had seen before. In Chapter 5 we determined that the SAM and CAM definitions of fluid velocity are equivalent at equilibrium. Thus, while fluctuations seemed to be causing the apparent flow, the origin appeared to be computational rather than physical in nature.

To establish the origin of the phenomenon, we first determined whether the

Figure 7.1: Sample averaging measurement of mean fluid velocity, $\langle u^x \rangle_s$, for an equilibrium system with a reservoir at the left boundary and a specular wall at the right boundary; the total number of samples was $S = 5 \times 10^6$. The anomalous flow is away from the reservoir. The plot obtained with cumulative average measurement, $\langle u^x \rangle_c$, of the same fluid shows that there was actually no net fluid flow.

thermodynamic variables were as expected. Figs. 7.2 and 7.3 show the particle density and SAM temperature profiles within the system. Although the plot is not shown here, there was no statistically significant pressure variation within the system; this can also be deduced from the ideal gas law and the constancy of T and N within our system.

Fig. 7.4 shows the effect of the number of particles. Both the cases of $\langle N_\Sigma \rangle =$

Figure 7.2: Particle density $n_k = N_k/V_k$ as a function of cell position for the system of Fig. 7.1. As the error bar shows, there is no statistically significant variation in particle density within the system, as expected for an equilibrium system.

Figure 7.3: Temperature $\langle T_k \rangle_s$ as a function of cell position for the system of Fig. 7.1. There is no statistically significant variation in temperature within the system.

2000 and $\langle N_\Sigma \rangle = 5000$ are shown. The magnitude of the $\langle N_\Sigma \rangle = 2000$ plot is approximately $\frac{5000}{2000} = 2.5$ that of the $\langle N_\Sigma \rangle = 5000$ case. This is consistent with the variations in fluid velocity due to fluctuations, since from Eqn. (2.32) we expect that $\langle (\delta \mathbf{u})^2 \rangle \propto 1/N$.

Another observation is possible from Fig. 7.4, similar to one we made in Chapter 5. The $\langle N_\Sigma \rangle = 2000$ plot was obtained from a simulation that was identical to that of Fig. 7.1, except that it was run for 10^8 time steps rather than the 5×10^6 time steps executed for Fig. 7.1. Comparison with that figure shows that the net effect of increasing the number of samples was to yield a smoother plot without changing the magnitude of the effect.

The effect of the reservoir algorithm remained an uncertain factor. The reservoir algorithm generated a constant number of particles at each time, where the constant was chosen such that the particle density in the system at the steady state was the same as the number of particles that were initially in the system. The particles were assigned a random position within the reservoir according to a uniform probability distribution, and then given a random velocity that yielded a Maxwell-Boltzmann distribution. All particles were then moved without colliding, and those that reached the system entered it with its assigned velocity. Because the velocity components were assigned independently and randomly, the number of particles reaching the system at each time would not be constant even though the total number of particles that were moved was constant.

This reservoir algorithm is not at all peculiar; it is apparently in common use by those who perform simulations that involve reservoirs. The algorithm ensures that the average kinetic energy of particles leaving the reservoir is the same as the particles already in the system. The presence of the specular wall at the opposite

Figure 7.4: Sample averaging measurement of mean fluid velocity, $\langle u_k^x \rangle_c$, for the system of Fig. 7.1. The plot of the $\langle N_\Sigma \rangle = 2000$ case (circles) is approximately $5000/2000 = 2.5$ times the magnitude of the $\langle N_\Sigma \rangle = 5000$ case (squares), as expected if the effect is due to fluctuations. Comparison of the $\langle N_\Sigma \rangle = 2000$ case with the corresponding plot in Fig. 7.1 shows that the net effect of increasing the number of samples is to yield a smoother plot.

•

side of the system prevents any net flow of fluid through the system at the steady state. In fact, the particle density and temperature measurements shown in Figs. 7.2 and 7.3 show that measurements of thermodynamic variables yield profiles that are indistinguishable from what are expected for a system at thermodynamic equilibrium. Nevertheless, the fact that SAM yielded a nonzero fluid velocity indicated that

something about the reservoir algorithm was not quite right.

From statistical mechanics, the number of particles in a cell should fluctuate over time according to Poisson statistics. At equilibrium, the number of particles N in a sub-volume of a large container follows a Poisson distribution, as was derived in Section 2.4 (see Eqn. (2.28)):

$$p(N) = \frac{\langle N \rangle^N e^{-\langle N \rangle}}{N!}. \quad (7.1)$$

Since fluctuations in N must arise from the exchange of particles through the boundaries of the sub-volume, it follows from isotropy that at equilibrium the exchange of particles through each boundary separately is Poisson distributed; this necessarily holds true at the boundary between the reservoir and system as well.

To ensure that the number of particles entering the system would be Poisson distributed over time, we modified the reservoir algorithm to vary N according to a Poisson distribution, where N is the number of particles to be created and moved at each time. The mean number $\langle N \rangle$ in Eqn. (7.1) was chosen to be the same as the constant number that was used in the previous, ‘constant N’ reservoir algorithm. Fig. 7.5 shows the effect of this change. SAM measurements of fluid velocity were no longer anomalous; the measured flow completely disappeared.

To make sure that the statistical fluctuations in our measurements were not masking a small but nonzero anomalous fluid velocity in the modified algorithm, we ran another simulation that increased the number of samples to 10^8 . The result is also plotted in Fig. 7.5, and it too indicated that SAM was no longer measuring a flow. Evidently, the anomalous flow was entirely due to the constancy in the number of particles that were generated in the reservoir at each time; only the Poisson distributed number is physically correct.

Figure 7.5: Sample averaging measurement of mean fluid velocity, $\langle u_k^x \rangle_s$, for the system of Fig. 7.1. The number of particles generated in the reservoir at each time was Poisson distributed about the same mean as was used for Fig. 7.1. Comparison with that figure shows that this modification to the reservoir algorithm entirely eliminated the anomalous fluid flow, fully consistent with expectations for a system at equilibrium. The smaller error bar applies to the plot for 10^8 samples.

We performed one more test to determine whether the number of particles entering the system from the ‘constant N ’ reservoir followed a Poisson distribution. We generated a histogram of the number N of particles entering and leaving the system at the reservoir at each time, and then calculated the mean number. From this and Eqn. (7.1) we generated a Poisson distribution of N for comparison with the

actual distribution; Fig. 7.6 shows the results obtained in this manner. The number of particles leaving the system was Poisson distributed, but the number of particles entering was not.

Figure 7.6: Histogram of N at the boundary between the system and a reservoir in a constant temperature system ($T = 5$, $\langle N_\Sigma \rangle = 2000$) at the steady state. The number of particles generated in the reservoir at each time was constant. Apparently the distribution of particles entering the system was not Poisson, in contrast to the number of particles leaving the system.

7.2 Characterization of the Effect

A quick look at the SAM definition for fluid velocity in terms of the CAM velocity shows that either a nonzero covariance $\langle \delta \mathbf{J} \delta N \rangle$ or variance $\langle (\delta N)^2 \rangle$ is the likely origin of the anomalous flow. From Eqn. (5.3),

$$\langle \mathbf{u} \rangle_s = \langle \mathbf{u} \rangle_c \left(1 + \frac{\langle (\delta N)^2 \rangle}{\langle N \rangle^2} \right) - \frac{\langle \delta \mathbf{J} \delta N \rangle}{m \langle N \rangle^2} + O([\delta X]^3).$$

This can be rewritten as a SAM-CAM difference. Dropping the $O([\delta X]^3)$ term and rearranging shows that for a single species gas having particle mass m ,

$$\langle \mathbf{u} \rangle_s - \langle \mathbf{u} \rangle_c \cong \langle \mathbf{u} \rangle_c \frac{\langle (\delta N)^2 \rangle}{\langle N \rangle^2} - \frac{\langle \delta \mathbf{J} \delta N \rangle}{m \langle N \rangle^2}. \quad (7.2)$$

Fig. 7.7 shows a plot of this SAM-CAM difference, together with the covariance term in Eqn. (7.2). As a reminder, the reservoir algorithm generated a constant number of particles at random positions within the reservoir at each time, giving them random velocities with a Maxwell-Boltzmann distribution, then moved them without collisions; those that did not reach the system after moving for the duration of the time step were discarded. Clearly, almost all of the contribution to the SAM-CAM difference arises from the covariance $\langle \delta \mathbf{J} \delta N \rangle$. Fig. 7.8 shows a plot of the entire right hand side of (7.2), and a comparison with Fig. 7.7 shows that the additional contribution from the variance term is insignificant, at least for the case $\langle \mathbf{u} \rangle_c = 0$ we considered.

The reservoir depth can affect simulation results if the maximum distance a particle can travel in one time step is greater than the reservoir depth. That is, a reservoir that is insufficiently deep will not correctly represent an infinitely deep reservoir because those particles that could otherwise exist beyond the end of the reservoir and still barely reach the system when moving sufficiently fast will not be correctly

Figure 7.7: Sample averaging measurement of mean fluid velocity, $\langle u_k^x \rangle_s$, for an equilibrium system at temperature $T = 5$ with a reservoir at the left boundary and a specular wall at the right boundary. The reservoir algorithm generated a constant number of particles at each time. Evidently the contribution to $\langle u_k^x \rangle_s$ from the covariance $\langle \delta \mathbf{J} \delta N \rangle$ accounts almost completely for the anomalous velocity.

represented by a short reservoir. In that case, one of two possible consequences will arise, depending on how the mean number of particles entering the system at each time is determined. If the mean is correctly calculated so that the particle density in the system is correctly maintained, then the density of particles in the reservoir must be artificially increased by a small amount to compensate for the small number of particles that would enter the system if the reservoir were sufficiently deep. The

Figure 7.8: Mean fluid velocity, $\langle u_k^x \rangle_s$, with the entire right side of Eqn. (7.2) shown for comparison. The simulation is the same as that used for Fig. 7.7; comparison with that figure shows that the variance term $\frac{\langle (\delta N)^2 \rangle}{N^2}$ is small.

result would be an artificial under-representation of high energy particles and a consequent slight reduction of temperature within the system. On the other hand, if the reservoir density is not adjusted to compensate for the missing high energy particles, then slightly fewer particles will enter the system over time, slightly reducing the number density within the system.

To determine a safe reservoir depth for our investigations, we simulated the same system with two different depths, $\lambda/2$ and λ , where λ is the mean free path. In light of the above, expressing the depth in terms of the mean free path is somewhat

arbitrary, since the key parameter is the maximum distance a particle can travel in one time step. However, λ was chosen as the parameter for consistency with the way the cell length was determined. As noted earlier, the cell length was chosen as $\lambda/2$, so our two test cases corresponded to reservoir depths of one and two cell lengths, respectively.

Fig. 7.9 compares the system temperatures for a reservoir algorithm that generated a constant number of particles at each time. The temperature of the deeper reservoir is slightly higher; the error bars show that within statistical error the difference is significant, but barely. Fig. 7.10 shows a similar comparison for a reservoir algorithm that generated a Poisson-distributed number of particles at each time. Although the reservoir with a depth of $\lambda/2$ yielded a slightly lower temperature than the reservoir of depth λ , the error bars show that the difference was not statistically significant and hence the comparison is inconclusive.

Considering the results shown in Figs. 7.9 and 7.10 together, it appears that a depth of one mean free path is reasonably safe but a reservoir half that deep may be inadequate, at least at the temperature we simulated ($T = 5$) and the time step τ that we chose. Of course, a lower reservoir temperature would result in a smaller flux of particles with very large speeds, and a smaller time step would result in a shorter maximum distance travelled at each time, so the minimum depth of a reservoir depends on the particular choice of these parameters. All of our reservoir simulations used a depth of one mean free path.

We also considered the effect of the time step. The time step was calculated as $\tau = L/5Kv_{\text{mp}}$, where L is the system length, K is the number of cells, and $v_{\text{mp}} = \sqrt{2k_B T/m}$ is the most probable molecular speed (see Appendix A). The time step determines the frequency at which new particles are created in the reservoir and

Figure 7.9: System temperature $\langle T_k \rangle_s$ for two different reservoir depths. The reservoir algorithm generated a constant number of particles at each time. The temperature in the system with a reservoir of $\frac{1}{2}$ mean free path is slightly lower than the system with a reservoir one mean free path deep, indicating that $\frac{1}{2}$ mean free path may not be quite long enough for a reservoir temperature $T = 5$.

then moved, so changing the time step could potentially affect the fluid velocity if it resulted in the number of particles entering the system at each time becoming more or less Poisson-distributed. Fig. 7.11 shows the results of three different choices of time step, where τ represents the default time step used in all (other) simulations, and $\tau/2$ and $\tau/4$ are time steps of one-half and one-fourth of that default, respectively.

It is apparent from Fig. 7.11 that decreasing the time step results in a lower

Figure 7.10: System temperature $\langle T_k \rangle_s$ for two different reservoir depths. The reservoir algorithm generated a Poisson-distributed number of particles at each time. The temperature for the system with a reservoir of $\frac{1}{2}$ mean free path is slightly lower, but the difference is not statistically significant.

SAM fluid velocity. With a smaller time step, the number of particles to be generated at each time must also decrease if the number density in the system is to remain constant. A reasonable speculation is that the smaller number of particles entering the system at each time causes their distribution to more nearly follow Poisson statistics. The histogram in Fig. 7.12 shows the measurement of the actual distribution for the simulation of the $\tau/4$ case. Now the distribution for particles entering the system from the reservoir is nearly Poisson. Extrapolating from Figs. 7.6 and 7.12 to the

Figure 7.11: SAM fluid velocity for three different time steps in a system with one reservoir. The reservoir algorithm generated a constant number of particles at each time. The default time step was τ ; $\tau/2$ and $\tau/4$ are time steps of one-half and one-fourth of that default, respectively. A shorter time step evidently reduces the anomalous flow.

case $\tau \rightarrow 0$, it appears that a reservoir algorithm which generates a constant number of particles at each time will yield particle numbers that approach the physically correct distribution in that limit.

Other than the time step, almost all system and simulation parameters were identical for all plots in Fig. 7.11. Because the total time was held constant for

Figure 7.12: Histogram showing the distribution in the number of particles entering and leaving the system at each time for a time step of $\tau/4$. The system and reservoir algorithm were the same as that used for Fig. 7.11. The distribution is close to Poisson, both for particles entering and leaving.

all three simulations while the time step changed, the total number of time steps and samples differed for each case. It seems highly unlikely that the number of samples would affect the simulation results in any meaningful way, however, since in all other cases where the number of samples increased the only net effect was to yield a statistically more meaningful result.

It is worthwhile to verify that the reservoir algorithm that generates a Poisson-distributed number of particles at each time always supplies the correct distribution

of particles to the system, regardless of time step. Figs. 7.13 and 7.14 show the cases of the default time step τ and reduced time step $\tau/2$; the distribution in both cases is Poisson, as expected.

Figure 7.13: Histogram showing the distribution in the number N of particles entering and leaving the system during a default time step τ . The reservoir algorithm generated a Poisson-distributed number of particles at each time. The distribution in number of particles is Poisson, both for particles entering and leaving the system.

Interestingly, the approach toward a Poisson distribution for particles entering the system from a ‘constant N ’ reservoir as $\tau \rightarrow 0$ does not appear to have an analogue for varying the number density within the system. That is, if the number of particles in the system is increased with a corresponding increase in the number of particles

Figure 7.14: Histogram showing the distribution in the number of particles N entering and leaving the system at each time for the time step of $\tau/2$. The system and reservoir algorithm were the same as that used for Fig. 7.13. The distribution in N is Poisson.

that must be generated within the reservoir at each time, the deviation from a Poisson distribution of N for particles entering the system appears to be independent of the magnitude of $\langle N \rangle$. Fig. 7.15 shows the histogram for the $\langle N_{\Sigma} \rangle = 5000$ case; comparison with histogram for the $\langle N_{\Sigma} \rangle = 2000$ case in Fig. 7.6 shows little difference.

Figure 7.15: Histogram showing the distribution in the number of particles entering and leaving the system at each time for the case of average total particle number $\langle N_{\Sigma} \rangle = 5000$ with a ‘constant N ’ reservoir algorithm. Comparison with the histogram in Fig. 7.6 for the $\langle N_{\Sigma} \rangle = 2000$ case shows little or no difference in their relative deviation from Poisson statistics.

7.3 Discussion and Additional Simulations

Although our simulations do not provide any conclusive evidence, they give hints that the measurement by SAM of an anomalous fluid velocity may be closely correlated with a deviation from Poisson statistics in the number of particles that enter the system from the reservoir at each time. The simulations that varied the time step provide the most direct evidence for this, in that all system and simulation parameters were nearly identical except for the time step duration. Comparison of relative deviation from Poisson distribution with the magnitude of the SAM fluid velocity near the reservoir appear to indicate a direct relationship.

An alternative algorithm for generating a Poisson-distributed number of particles to the system at each time gave further insight into the relationship between the SAM fluid velocity and the distribution of particle number over time. This algorithm implemented the reservoir as a separate, closed, and thermally isolated system with specular walls and a particle density and temperature that equalled the desired density and temperature of the system.

The particles in this closed reservoir moved and collided, just as they did in the system, but any particle striking the wall between the reservoir and the system ‘entered’ the system. That is, when a reservoir particle struck that wall two things happened: the original particle specularly reflected back into the reservoir; and a new system particle was created at the wall, which then continued moving into the system at the velocity with which the reservoir particle had struck the wall, until the end of the time step. Particles in the system that reached the boundary between the system and reservoir were simply removed from the system without actually entering the reservoir. We expected this scheme to yield both a reservoir and system that were

separately at equilibrium, and thus we expected a Poisson-distribution in the number of particles both entering and leaving the system. The corresponding histogram in Fig. 7.16 supports this expectation.

Figure 7.16: Histogram showing the distribution in the number of particles entering and leaving a system with a single closed reservoir at the left boundary of the system. The reservoir depth was 4λ and the system had temperature $T = 1$. The distribution in number of particles entering the system is very nearly Poisson.

The histogram in Fig. 7.16 was from a reservoir four mean free paths deep. To investigate the effect of reservoir length on particle number distribution, we ran another, nearly identical simulation that differed only in that the closed reservoir was $\frac{1}{2}$ mean free path deep. There was now a noticeable deviation from a Poisson

distribution, as Fig. 7.17 shows.

Figure 7.17: Histogram showing the distribution in the number of particles entering and leaving a system having a single closed reservoir at the left boundary of the system. The reservoir was $\frac{1}{2}$ mean free path deep and the system had temperature $T = 1$. The distribution in number of particles entering the system deviates noticeably from Poisson statistics compared to the longer reservoir of Fig. 7.16.

Surprisingly, as Fig. 7.18 shows, SAM measured a nonzero fluid velocity in the system near the closed reservoir, even in those cases where the number of particles entering and leaving the system was nearly Poisson distributed. As before, the covariance $\langle \delta \mathbf{J} \delta N \rangle$ accounted for almost all of the anomalous flow. While this result does not contradict the hypothesis that a deviation from Poisson statistics is associated

with a nonzero covariance $\langle \delta \mathbf{J} \delta N \rangle$, it clearly allows for additional factors.

Figure 7.18: SAM fluid velocity $\langle \mathbf{u}_k \rangle_s$ in a system having a single closed reservoir at the left boundary of the system of varying depths. Comparison with Fig. 7.16 shows that SAM measures a velocity even when the distribution in number of particles from the reservoir is Poisson. Note that direction of $\langle \mathbf{u} \rangle_s$ depends on reservoir depth, and it even reverses in the ‘medium’ depth case (squares plot).

Fig. 7.18 contains several peculiarities. Most noticeable is that the deepest reservoir gave rise to a measured flow with a direction opposite to what we had seen with the ‘constant N ’ reservoir algorithm. Curiously, this also corresponded to the reservoir that most nearly gave a Poisson distribution in number of particles entering the system; i.e., it was the reservoir whose depth we expected to give the smallest SAM velocity. Also noticeable is that the fluid velocity profile for the ‘medium’

reservoir depth starts with $\langle \mathbf{u}_k \rangle_s < 0$, then rises to $\langle \mathbf{u}_k \rangle_s > 0$ before gradually disappearing at a distance of about 60% of the system length from the reservoir. Finally, the magnitudes of the SAM velocities from the shallowest and deepest reservoirs are approximately the same, but in opposite directions.

From these observations, it remains inconclusive whether a nonzero covariance $\langle \delta \mathbf{J} \delta N \rangle$ is always associated with a deviation from Poisson statistics, or whether the deviation and SAM velocity are merely varying together under less general conditions. The clues in Fig. 7.18, however, indicate that the SAM velocity $\langle u^x \rangle_s$ may be a sum of *two* factors. The first factor is likely to be the deviation of N from a Poisson distribution. This is consistent with the results from ‘constant N ’ reservoir simulations, and also the simulation of a shallow, closed reservoir; furthermore, in all those cases the measured fluid velocity was *away from* the reservoir.

The second factor may be a consequence of the depth of the closed reservoir. It may be significant that the longer the reservoir, the more closely the number of particles entering the system at each time followed a Poisson distribution, but also the more negative the measured velocity became. If $\langle \mathbf{u}_k \rangle_s$ is a sum of two factors, then the first factor tends toward zero as reservoir depth increases, but then the second factor necessarily grows larger with increasing depth. The behavior of the second factor is, in fact, quite reasonable in light of the way collisions are handled in DSMC simulations.

In our simulations, a closed reservoir was treated as a single cell for DSMC collision processing, so any pair of reservoir particles was a potential collision candidate pair, regardless of particle separation. As the reservoir depth increased while keeping the number density constant, the number of collision candidates involving particles very far from each other would necessarily also grow, and so also would the number

of physically unreasonable collisions. This should result in a nonphysical smearing of the spatial distribution of particle velocities throughout the reservoir, similar in concept to the way DSMC collision processing can smear temperature or shear velocity gradients in non-equilibrium systems. It seems plausible that this nonphysical, spatial redistribution of molecular energy would be observable as a non-equilibrium contribution to the covariance $\langle \delta \mathbf{J} \delta N \rangle$.

The two-factor explanation for the SAM velocity is strongly supported by the ‘medium’ reservoir plot in Fig. 7.18. The fact that $\langle \mathbf{u}_k \rangle_s$ changed directions indicates that the relative strength of two factors reversed at some point, so that first one factor dominated and then the other. The other two profiles in that figure show no such inconsistency; they are either consistently positive or negative until they tend toward zero about 60% of the distance across the system. In fact, the histograms for the ‘shallow’ (Fig. 7.17) and ‘deep’ (Fig. 7.16) closed reservoir systems show a significant and insignificant deviation, respectively, from Poisson statistics. Concurrently, we expect the redistribution of molecular energy within the reservoir due to DSMC collision processing to be respectively small and large for the two cases. Fig. 7.19 shows the histogram for the ‘medium’ depth case; deviation from a Poisson distribution is small but between that of the other two cases.

Conclusive evidence for this two-factor explanation probably requires additional simulations. A particularly useful one would consider a system with a deep, closed reservoir comprised of multiple cells, the reservoir cells being similar in dimension to what is used in the rest of the system. We did not perform those simulations.

Conceptually, the existence of correlations between fluctuations in fluid velocity and particle number in a system with a ‘constant N ’ reservoir is reasonable. The cell dimensions in a molecular simulation are generally large enough that particles remain

Figure 7.19: Histogram showing the distribution in the number of particles entering and leaving a system having a single closed, constant generation reservoir at the left boundary of the system. The reservoir was two mean free path deeps and the system had temperature $T = 1$. The distribution in number of particles entering the system deviates slightly from Poisson statistics, but is larger than that of the ‘deep’ reservoir; compare with Fig. 7.16.

in a cell longer than one time step. Thus, because the fluid momentum $\mathbf{J}_k$ explicitly depends on the particle number N_k , $\mathbf{J}_k$ contains some ‘memory’ of the number of particles in cell k at the previous time. Furthermore, according to Eqn. (2.29), $\langle(\delta\mathbf{J}_k)^2\rangle \propto N_k$, so this mean square fluctuation also contains some memory of N_k at the previous time.

When reservoir particles are created at a constant rate, the number of particles

entering the cell from the reservoir at different times is not constant but is related in a non-random way; i.e., it is correlated in time. Given this, it is not surprising that the correlation of fluctuations $\delta\mathbf{J}$ and δN deviates from equilibrium behavior to some degree, even though the cell nearest the reservoir exchanges particles with its adjacent cell as well as the reservoir.

This chapter has focused on how particle reservoir algorithms may inadvertently introduce non-equilibrium correlations of fluctuations that can subsequently bias the measurements of hydrodynamic variables. Rather than being restricted to a special purpose component of molecular simulations that finds use in a limited number of applications, the problem with ‘constant N ’ reservoirs hints at a more general computational issue. Any time an algorithm contains stochastic elements, it inherently introduces fluctuations of some kind into any system where the algorithm is applied. If the fluctuations are not sufficiently stochastic, it is likely that their non-random behavior will be observed and the algorithm corrected. What we have demonstrated, however, is that nonphysical *correlations* of fluctuations can be computationally induced and have measurable consequences, even in apparently correct algorithms, and they may be difficult to detect as program logic errors because of their subtlety. It is therefore important to be especially careful with the stochastic elements of algorithms in those simulations that are sensitive to fluctuations phenomena, and to consider not only the stochasticity of fluctuations but also the interrelationships between different fluctuating variables.

Although the anomalous results in this chapter have a computational origin in simulation algorithms, they may give important clues about the nature of non-equilibrium fluctuations that can occur in a physical system. In particular, we may be able to extend our speculations in this chapter, i.e. regarding the relationship between

anomalous SAM fluid velocity and statistical fluctuations in particle number, to try to better understand the origin of the non-equilibrium phenomena of Chapters 5 and 6. We will explore this further in the next chapter.

CHAPTER 8

GENERAL DISCUSSION AND CONCLUDING REMARKS

8.1 Summary of Key Results

The goal in this thesis was to present our investigations of three different conditions where fluctuations can cause anomalous measurements of hydrodynamic variables in the molecular simulation of a fluid. In Chapter 1 we introduced them, the SAM bias in fluid velocity and temperature measurements at non-equilibrium, and also the measurement of a nonphysical fluid flow in a constant temperature system with one reservoir. In Chapter 2 we developed the sample averaging measurement (SAM) and cumulative average measurement (CAM) definitions of fluid velocity and temperature, both for single and multiple molecular species fluids. The multiple species case was treated for generality, although all of our simulations involved only one species. Chapters 3 and 4 described both the one-dimensional system and molecular simulation method (DSMC) that we used to obtain our results. Then, in Chapters 5-7 we developed our main results in some detail, employing both simulations and quantitative reasoning.

In Chapter 5 we considered the measurement of fluid velocity. There we noted that at non-equilibrium the CAM and SAM definitions of fluid velocity, Eqns. (2.1) and (2.2) respectively, differed due to a nonzero covariance of non-equilibrium fluc-

tuations. We determined on the basis of physical arguments that SAM was biased; i.e., the nonzero SAM fluid velocity was physically impossible in our closed system at the steady state. Our investigations considered non-equilibrium systems with a large temperature gradient, including one scenario where the temperature gradient arose indirectly from viscous heating in a Couette flow system. Nonetheless, we expect that any non-equilibrium system where fluctuations in fluid momentum and particle density are correlated will yield a similar bias in SAM mean fluid velocity. Measurements by the CAM definition, on the other hand, will always be unbiased.

In Chapter 6 we examined the measurement of temperature according to the CAM and SAM definitions, Eqns. (2.8) and (2.12) respectively, in a non-equilibrium system with a temperature gradient. Similar to what we saw with fluid velocity, we found that the SAM definition can be biased by non-equilibrium fluctuations. We were unable to find a clear violation of physical principles by SAM or CAM, unlike the case of mean fluid velocity. Instead, we relied on theoretical arguments to decide that CAM will always yield the correct temperature measurement. That is, the additional variance and covariance terms in the SAM definition, $\langle T \rangle_c \frac{\langle (\delta N)^2 \rangle}{\langle N \rangle^2}$ and $-\frac{2}{3k_B} \frac{\langle \delta K \delta N \rangle}{\langle N \rangle^2}$, have no counterpart in the generally accepted correspondence of temperature with the mean translational kinetic energy per molecule (for a monatomic gas). While those terms are equal and thus cancel at equilibrium, at non-equilibrium their sum is nonzero and the SAM temperature measurement will be biased.

Finally, Chapter 7 described investigations of equilibrium systems that contained one particle reservoir. We used molecular simulations to show that an incorrectly designed reservoir algorithm can lead to biased fluid velocity measurements by SAM, even under equilibrium conditions. We demonstrated that at least one algorithm in apparent widespread use can inadvertently introduce a nonzero covari-

ance of fluctuations in fluid momentum and particle number that would not exist in a physical system. We then considered a simple modification to the algorithm that eliminated the problem: by ensuring that the number of particles entering the system from the reservoir obeyed Poisson statistics, then the system behaved as an equilibrium system in every way that we could determine. We finished by noting that the anomalous results with some reservoir algorithms is but one example of a more general issue. That is, computationally induced, nonphysical correlations of fluctuations can be introduced into simulations by incorrectly designed stochastic algorithms.

Although the simulations of a system with one reservoir showed how purely computational considerations can lead to a subtle bias in measurement, they were also interesting in that they indicated an apparent relationship between the anomalous SAM fluid velocity and deviations from Poisson distribution in particle density fluctuation. This relationship may also occur in the physically meaningful covariance of non-equilibrium fluctuations that underlie the SAM measurement bias. We will consider this possibility further in Section 8.2.1.

8.2 Directions for Future Study

8.2.1 Analytic expressions for fluctuation covariances

Eqn. (2.28) shows that at equilibrium the fluctuation in N in a sub-volume of a large container follows a Poisson distribution:

$$p(N) = \frac{\langle N \rangle^N e^{-\langle N \rangle}}{N!}. \quad (8.1)$$

Since the fluctuation must arise from the exchange of particles through the boundaries of the sub-volume, it follows from isotropy that the exchange of particles through each boundary separately is also Poisson distributed at equilibrium; i.e., all directions are

thermodynamically equivalent. Consequently, the fluctuations in particle density will follow the same statistics in one-, two- and three-dimensional equilibrium systems.

In simulations the sub-volume can be taken as equivalent to a cell. If the simulation is to represent a physical system, the exchange of particles through cell boundaries must obey Poisson statistics at equilibrium. The particle flux histogram in Fig. 7.13 demonstrates this for an equilibrium system with a ‘Poisson N ’ reservoir; Fig. 8.1 does the same for a closed, one-dimensional system with thermal walls at the same temperature.

The above reasoning, i.e. that fluctuations in the number of particles within a fixed sub-volume will obey Poisson statistics, is only valid at equilibrium. At non-equilibrium not all directions within the system are equivalent, and so the flux of particles through different boundaries of a sub-volume may follow different statistics. For example, it is clear that a temperature gradient has a specific direction. More subtly, the flux of particles from a ‘constant N ’ reservoir into the system follows slightly different statistics than the flux of particles leaving the system (see the histogram in Fig. 7.17), disrupting system isotropy.

It is straightforward to show qualitatively that the number of particles within a fixed sub-volume of a system will not obey Poisson statistics at the steady state in the presence of a temperature gradient. Consider a cubic sub-volume with dimensions significantly less than one mean free path, having left and right boundaries at temperatures T_L and T_R , respectively. Assume the sub-volume is embedded in a much larger system with a temperature gradient, such that $T_L < T_R$; assume also the steady state. From the ideal gas law, $N = PV/k_B T$, and so $N_L > N_R$. It is apparent from Eqn. (8.1) that the probability distribution $p(N_L) \neq p(N_R)$ because $N_L \neq N_R$.

Although it is apparent that at non-equilibrium the probability distribution

Figure 8.1: Histogram of N at the thermal wall of a closed one-dimensional system at equilibrium, where $T_L = T_R = 1$. The fluxes of particles both striking and leaving the wall obey Poisson statistics.

$p(N)$ for the number of particles N within a fixed sub-volume over time is not Poisson, we do not have an explicit expression for $p(N)$. One future goal is to obtain this expression in analytic form, and from that derive expressions for the covariances $\langle \delta \mathbf{J} \delta N \rangle$ and $\langle \delta K \delta N \rangle$ at non-equilibrium. The remainder of this section presents some indicative results as well as an outline of an approach that may lead to an expression for $p(N)$.

Considering the non-equilibrium, steady-state system with a temperature gra-

dient, if the distance between the left and right boundaries of a fixed sub-volume is much smaller than one mean free path there will be relatively few collisions within the sub-volume. Hence the statistics for the flux of particles entering the sub-volume through its *right* boundary (from a higher temperature region) and leaving the sub-volume through the *left* boundary (into a lower temperature region) will be highly correlated, although incompletely due to occasional collisions within the sub-volume itself. Fig. 8.2 shows this explicitly for the case where the sub-volume is a single cell.

These statistics are different than those for particles entering and leaving the same sub-volume through the *same* boundary, as the histogram in Fig. 8.3 demonstrates for a single cell. Note that the difference in statistics for Fig. 8.2 is much greater than for Fig. 8.3, presumably due to collisions in the cell. The inequality in statistics for particles entering and leaving a given sub-volume implies that the fluctuation in particle number within the sub-volume does not strictly obey Poisson statistics; Fig. 8.4 confirms this.

We can now outline what may be a promising approach for obtaining $p(N)$ in future work. A key step is to obtain an analytic expression for a probability distribution of the number of particles at any position ' x ' within the system, given a temperature gradient. If we know the expected temperature $\langle T(x) \rangle$, then it is straightforward to obtain a mean square particle speed $\overline{v(x)^2}$. From $\langle T(x) \rangle$ the ideal gas law allows obtaining the particle number $\langle N(x) \rangle$ as a function of position; from $\langle N(x) \rangle$ the position-dependent mean free path $\lambda(x)$ and probability of a collision also follow. We can then determine the expected momentum $\langle \mathbf{J}(x) \rangle$ and particle flux into a cell from the cells both at the left and right, as well as the fluxes out of the cell into cells at the left and right. Putting all of this together, we can hope to obtain an

Figure 8.2: Histograms for $N_{17 \rightarrow 16}$ and $N_{18 \rightarrow 17}$, the number of particles leaving cell 17 for cell 16 (filled circles) and entering cell 17 from cell 18 (diamonds), respectively. The system was closed and had a large temperature gradient imposed by thermal walls with temperatures $T_L = 1$ and $T_R = 5$. The flux of particles entering and leaving the cell follow observably different statistics.

expression for $p(N)$, and subsequently for the covariances $\langle \delta \mathbf{J} \delta N \rangle$ and $\langle \delta K \delta N \rangle$.

8.2.2 Simulations of multi-species fluids

In developing the SAM and CAM definitions of fluid velocity and temperature in Chapter 2 we obtained expressions for both single- and multiple-species fluids. Our simulations only investigated the single species case, however.

Figure 8.3: Histograms for $N_{17 \rightarrow 16}$ (filled circles) and $N_{16 \rightarrow 17}$ (diamonds), the number of particles moving from cell 17 into cell 16 and entering cell 17 from cell 16, respectively. The simulation was the same as that used for Fig. 8.2. The flux of particles both entering and leaving the cell through its left boundary follow slightly different distributions; this can be seen by observing the relative vertical positions of the filled circles (particles entering the cell) within the diamonds (particles leaving).

Fluids with multiple molecular species may yield interesting SAM measurement results, particularly if there is a large disparity in molecular mass. Consider a closed system, for example, containing a fluid with a relatively small number of heavy molecules (e.g., N_{Ar} molecules of argon) mixed with a significantly larger number of light molecules (e.g., N_{He} molecules of helium). Measurements by SAM may yield anomalous fluid velocities that are noticeably larger than what might be expected

Figure 8.4: Histogram for N_{17} , the number of particles in cell 17. The simulation was the same as that used for Fig. 8.2. The Poisson distribution that corresponds to the actual mean number of particles in the cell is shown for reference; the actual distribution clearly deviates from it, especially near the mean.

from a naive application of Eqn. (8.1).

As an example, let $N_{\Sigma} = N_{\text{Ar}} + N_{\text{He}}$, such that the total mass fractions of the argon and helium atoms are equal. Because the molecular masses $m_{\text{Ar}} \neq m_{\text{He}}$, then $p(N_{\Sigma}) \neq p(N_{\text{Ar}}) \neq p(N_{\text{He}})$. Consequently, the mean square fluctuation $\langle (\delta N)^2 \rangle$ and covariance $\langle \delta \mathbf{J} \delta N \rangle$ for the argon molecules will be significantly larger than those for the helium. This and the equality of mass fractions indicate that the anomalous fluid

velocity will be larger for this two-species fluid than would occur in a single species fluid with $N_{\text{Ar}} + N_{\text{He}}$ particles. Simulations are needed to test this expectation.

8.2.3 Two dimensional systems

All of our simulations considered one-dimensional systems. Working in one dimension has the key advantage of maximal simplicity, both in concept and in the simulation efficiencies it allows. With fewer degrees of freedom, profiles of the variables being measured are simpler to interpret, and greater conceptual clarity is possible because the number of variables can be reduced to a minimum.

On the other hand, one dimensional systems are not realistic for most hydrodynamic systems of general interest. Furthermore, restricting the number of dimensions reduces the range of possible phenomena to those that can occur along one axis. Thus, investigation of two dimensional systems is desirable. For example, if the closed, one-dimensional system of Fig. 4.1 were modified so that the system is divided into multiple cells in the y -direction, measurements could be made along that axis and perhaps reveal new phenomena.

We showed in Fig. 6.4 for a system with a temperature gradient and no net fluid velocity that the covariance $\langle \delta K \delta N \rangle$ deviates from its equilibrium value. From this and the fact that in this scenario the average kinetic energy for a single species fluid, $\langle K \rangle = \frac{1}{2}m\langle \mathbf{u}^2 \rangle$, is equally distributed among all three components, we expect that $\langle \delta u^y \delta N \rangle \neq 0$. That is, we expect a nonzero, non-equilibrium correlation of momentum and density fluctuations in the y -direction, even though the system is uniform along that axis; Eqn. (5.11) then implies that $\langle u^y \rangle_s \neq 0$.

8.2.4 Fluctuations in coupled closed systems with shared rigid walls

When investigating the anomalous SAM fluid velocity in a supposedly equilibrium system with one particle reservoir, we considered an algorithm that treated the reservoir as a closed system with specular walls, wherein DSMC particle movement and collision processes occurred in the same way as in the main part of the system. To review, a particle entered the system from the closed reservoir whenever a reservoir particle struck the wall adjoining the system; the reservoir particle remained in the reservoir, and a new one was created in the system with the same velocity and position the reservoir particle would have had if it had not been specularly reflected. Particles reaching the reservoir from the system, on the other hand, were simply removed with no other effect. The intent was to create a constant density reservoir at equilibrium, and thus yield a particle flux into the system that was Poisson distributed over time. Provided the reservoir was sufficiently deep this goal was basically met, as Fig. 7.16 shows. Nevertheless, Fig. 7.18 clearly shows that a reservoir of this type ‘created’ an anomalous fluid velocity, even though no temperature gradient was present.

Although we speculated in Section 8.2.1 that the anomalous flow was due to a confluence of two different computational factors, additional simulations and analysis are necessary to conclusively demonstrate its origin. It is possible that the results in Fig. 7.18 reflect the isolated case of a poorly conceived reservoir algorithm. There was a definite asymmetry in that only the reservoir was closed, since particles could enter and leave the system but not the reservoir. On the other hand, those results could hint of a more general susceptibility of coupled molecular simulations to subtle, computationally induced correlations of fluctuations. If the latter were true, it could be relevant to coupled, closed systems that indirectly interact through their separate

mechanical effects on shared, rigid walls.

Future work should investigate this question. One approach might be to consider a system with a closed reservoir that is similar to what we described above, but significantly deeper and with multiple cells of the same dimension as the main system. This would ensure that the reservoir delivers a Poisson distributed flux of particles into the system without being susceptible to the potential problems of closed reservoirs that we mentioned in Section 8.2.1.

8.3 Additional Observations and Concluding Remarks

We can generalize the definitions of SAM and CAM to an abstract, generalized hydrodynamic variable, $\mathcal{H}(M, \mathbf{J}, K)$:

$$\langle \mathcal{H} \rangle_s \equiv \langle \mathcal{H}(M, \mathbf{J}, K) \rangle, \quad \langle \mathcal{H} \rangle_c \equiv \mathcal{H}(\langle M \rangle, \langle \mathbf{J} \rangle, \langle K \rangle),$$

where $M, \mathbf{J}$ and K are the fluid mass, momentum, and kinetic energy within some fixed volume, respectively. That is, in the SAM definition the mean $\langle \mathcal{H} \rangle_s$ is obtained as the average over samples of the instantaneous values of $\mathcal{H}$, where $\mathcal{H}$ is calculated at each sample from the instantaneous measurements of $M, \mathbf{J}$ and K at that sample time. In the CAM definition, on the other hand, the variables $M, \mathbf{J}$ and K are separately averaged over all samples, and then $\langle \mathcal{H} \rangle_c$ is calculated from the averages $\langle M \rangle, \langle \mathbf{J} \rangle$ and $\langle K \rangle$.

In general, the SAM definition is susceptible to bias because transient correlations among fluctuations in $M, \mathbf{J}$ and K are systematically recorded as part of the calculation of instantaneous $\mathcal{H}$ at each sample. (A transient in this context is a phenomenon that persists on the order of one to a few sample times.) We expect the CAM definition, on the other hand, to be unbiased for any hydrodynamic variable; the

mean fluctuation $\langle \delta X \rangle$ of any variable X is zero by definition, so transient correlations among fluctuations in M , $\mathbf{J}$ and K are not recorded in $\langle \mathcal{H} \rangle_c \equiv \mathcal{H}(\langle M \rangle, \langle \mathbf{J} \rangle, \langle K \rangle)$.

Although the foregoing observations are interesting from a theoretical standpoint, they do not judge the relevance of the SAM bias to practical work. All of our simulations of non-equilibrium systems involved very large gradients in temperature or shear velocity, much larger than is typical for realistic physical systems. For example, consider Fig. 5.1, which shows the SAM fluid velocity profiles that correspond to three different temperature gradients in a closed system with thermal walls. In the moderate gradient case, the temperature from left wall to right wall increases by a factor of three over a distance of only ten mean free paths. Even with this strong gradient, the anomalous fluid velocity from the SAM definition is small, corresponding to a Mach number of approximately 10^{-4} .

For systems far from equilibrium, the magnitude of non-equilibrium fluctuations is usually small when compared to the equilibrium contribution. For example, consider plots of covariance in number density $\langle \delta n_{11} \delta n_k \rangle$ in Fig. 4.14, used to validate the simulator. In the moderate gradient case the mean number of particles $\langle N_k \rangle$ per cell varied between approximately 70 and 160, with the largest number nearest the colder wall; see Fig. 4.8. Near the center of the system a cell had a mean of 100 particles, so Eqn. (2.26) predicts the equilibrium root mean square fluctuation in particle number was $\sigma_N = \sqrt{\langle N_k \rangle} = 10$, corresponding to a mean square density fluctuation of $\sigma_n^2 = 1$; this is about 10% of the mean. Fig. 4.14 indicates that the non-equilibrium mean square fluctuation σ_n^2 in the center was approximately 0.01, or about one percent of the equilibrium component.

For these reasons, the bias is not significant in many and probably most molecular simulations of fluid flows. Nonetheless, for some practical or research problems

it can be quite important.

For example, in simulations of micro-mechanical systems the magnitude of fluctuations is large and the system may be very far from equilibrium. If the purpose of a simulation is to determine whether a Brownian motor will move or a nanoscale pump will create a flow, the bias can be significant, and measurements of fluid velocity may give misleading results with practical consequences. Other examples where fluctuations play an important role include simulations of rarefied gas flows, such as vapor deposition in near-vacuum conditions, aerospace flows in the upper atmosphere and astrophysical flows [25]; or flows with sharp interfaces, such as strong shock waves. Finally, computer simulation programs are usually tested on simple benchmark problems for which the flow properties are known, for example the temperature gradient and Couette flow scenarios we discussed in Sec. 5.2. The anomalous flow measured by the SAM definition could be mistaken as indicating an error in the programming logic, as it was in our initial investigations.

We demonstrated in Chapter 7 that a particle reservoir algorithm can correlate the fluctuations in some measured variables that would be uncorrelated in a physical system with equivalent parameters. This is but an example of a more general issue in molecular simulations which, as far as we know, has not been widely recognized: fluctuations in fluids can become spatially correlated in a nonphysical way due to computational causes, even at thermal equilibrium, if the stochastic nature of fluctuations is not adequately considered in algorithm design. A comparison of Figs. 7.1 and 5.1 shows that the SAM bias arising from an incorrect reservoir algorithm is similar in magnitude to the SAM bias we demonstrated for a ‘moderate’ temperature gradient in a closed system with thermal walls. Whether a bias of this magnitude has practical consequences depends on the nature of the system and the importance

attached to unbiased measurements of hydrodynamic variables. Although the CAM definition is apparently not susceptible to this form of computationally induced bias, it is significant that SAM manifests it even under equilibrium conditions where the SAM and CAM definitions are physically equivalent.

In Section 2.4 it was shown that, at equilibrium, the particle number N in a fixed sub-volume obeys Poisson statistics over time (see Eqn. (2.28)). It is meaningful to question whether a steady-state fluid at constant temperature and with zero flow velocity (as measured by CAM) is at thermodynamic equilibrium if N for a fixed sub-volume does not follow Poisson statistics over time. In Chapter 7 we demonstrated for systems with a ‘constant N ’ reservoir that SAM measures a nonzero fluid velocity under precisely these conditions. This and the fact that SAM does not measure a fluid velocity under truly equilibrium conditions such as exist in a closed system with thermal walls at equal temperature (see Fig. 5.1) imply that such a system is not truly at equilibrium.

It is worth emphasizing that the correlation of non-equilibrium fluctuations is a general, physical phenomenon. Provided that the simulator design does not inadvertently introduce nonphysical correlations like those discussed above, the bias in SAM is not an artifact of a particular method of molecular simulation. Although we demonstrated the effect in simulations of a dilute gas using the DSMC method, similar correlations of non-equilibrium fluctuations will be observed in other particle-based simulations, such as molecular dynamics [26] and lattice gases [27]. Furthermore, while DSMC is a stochastic algorithm, the hydrodynamic fluctuations in DSMC simulations have nothing to do with the Monte Carlo elements of the method, which involve the random selection of collisions among particles as described in Section 3.2.

A technical point peculiar to DSMC simulations is worth reiterating in light of

the inversely proportional relationship in Fig. 5.11 between the particle number and the magnitude of the SAM bias. As mentioned in Section 3.2, each DSMC particle may represent a large number of molecules in the physical system being modelled. The variance of fluctuations for extensive variables is proportional to the number of simulation particles, so this variance in a DSMC simulation differs from that in the physical system by N_{eff} , the ratio of physical molecules to simulation particles. When this ratio is large, the effect of fluctuations in the DSMC simulation exceeds that of the physical system, typically by many orders of magnitude. When the ratio is one-to-one, as was the case in our investigations, then the fluctuations in the simulation have the same magnitude as those in the physical system. For DSMC simulations, then, the bias in SAM depends on the number of simulation particles and not the number of physical particles they represent.

Although the CAM definitions yield unbiased measurements of mean fluid velocity and temperature, there are situations in which the SAM definition is useful. For example, for time varying flows, unless an ensemble of systems is considered, the measured fluid velocity will necessarily be the instantaneous velocity. Another example is in particle/continuum hybrids [18, 19], where the coupling of the two algorithms requires the estimation of instantaneous hydrodynamic quantities. As we have shown, the average of instantaneous fluid velocity is biased, but except for systems where fluctuations are very significant or temperature gradients are very large, this small error is probably acceptable.

In this thesis we have shown that a failure to consider the presence and effect of fluctuations in molecular simulations may lead to biased measurements of hydrodynamic variables in non-equilibrium systems. We have also demonstrated that computationally induced correlations of fluctuations may be inadvertently introduced

into a molecular simulation if the inherent stochasticity of fluctuations is not correctly modelled by an algorithm, and this may affect measurements even at equilibrium. While these anomalous phenomena are of theoretical interest, their practical effects on simulation results will likely be limited to systems very far from thermal equilibrium, those where fluctuations are large, or simulations where measurement accuracy is particularly important. Nonetheless, both researchers and practitioners should be aware of their existence and take them into account as necessary.

BIBLIOGRAPHY

- [1] R. Schmitz, Phys. Rep. 171 (1988) 1.
- [2] M. M. Mansour, J. W. Turner, A. L. Garcia, J. Stat. Phys. 48 (1987) 1157.
- [3] C. Ho, Y. Tai, Annu. Rev. Fluid Mech. 30 (1998) 579.
- [4] P. Reimann, Phys. Reports 361 (2002) 57.
- [5] L. Landau, E. Lifshitz, Fluid Mechanics, Pergamon Press, Oxford, 1959.
- [6] L. Marino, Phys. Fluids 15 (2003) 662.
- [7] N. Hadjiconstantinou, A. Garcia, M. Bazant, G. He, J. Comp. Phys. 187 (2003) 274.
- [8] M. Tysanner and A. Garcia, Measurement Bias of Fluid Velocity in Molecular Simulations, J. Comp. Phys. 196 (2004) 173-83.
- [9] G. A. Bird, Molecular Gas Dynamics and the Direct Simulation of Gas Flows, Clarendon Press, Oxford, 1994.
- [10] L. Landau, E. Lifshitz, Statistical Physics, Part 1, Butterworth-Heinemann, Oxford, 1980.
- [11] R. Pathria, Statistical Mechanics, Butterworth-Heinemann, Oxford, 1996.
- [12] D. Frenkel, B. Smit, Understanding Molecular Simulation, Academic Press, San Diego, 2002.
- [13] J. P. Rivet, J. P. Boon, Lattice Gas Hydrodynamics, Cambridge Univ. Press, Cambridge, 2001.
- [14] F. Alexander, A. Garcia and B. Alder, Cell Size Dependence of Transport Coefficients in Stochastic Particle Algorithms, Phys. Fluids, 10 (1998) 1540.

- [15] A. Garcia and W. Wagner, Time step truncation error in direct simulation Monte Carlo, *Phys. Fluids* 12 (2000) 2621-33.
- [16] A. Garcia, *Numerical Methods for Physics*, Second Edition, Prentice Hall, Upper Saddle River, 2000.
- [17] G. Smith, *Numerical Solution of Partial Differential Equations*, Third Edition, Clarendon Press, Oxford, 1985.
- [18] A. Garcia, J. Bell, W. Crutchfield, B. Alder, *J. Comp. Phys.* 154 (1999) 134.
- [19] E. Flekkoy, G. Wagner, J. Feder, *Europhys. Lett.* 52 (2000) 271.
- [20] M. Matsumoto and T. Nishimura, *ACM Trans. Modeling and Comp. Sim.* 8 (1998) 3.
- [21] A. Garcia, *Phys. Rev. A* 34 (1986) 1454.
- [22] A. Garcia, "Instantaneous Temperature and Velocity Notes", Unpublished work.
- [23] B. Alder, T. Wainwright, *Phys. Rev. A* 1 (1970) 18.
- [24] J. Koplik, J. Banavar, *Annu. Rev. Fluid Mech.* 27 (1995) 257.
- [25] A. Ketsdever, E. Muntz (Eds.), *Proceedings of the 23rd International Symposium on Rarefied Gas Dynamics*, AIP, College Park, MD, 2003.
- [26] M. Mareschal, M. M. Mansour, G. Sonnino, E. Kestemont, *Phys. Rev. A* 45 (1992) 7180.
- [27] A. Suarez, J. Boon, P. Grosfils, *Phys. Rev. E* 54 (1996) 1208.

APPENDIX A

SOME MOLECULAR SPEEDS

We derive equilibrium expressions for the mean molecular speed $\langle v \rangle$, the mean square speed $\langle v^2 \rangle$ and the most probable speed v_{mp} . The mean relative speed $\langle v_r \rangle$ is derived in Appendix B; see Eqn. (B.4) for a single species gas. A molecular mass m is assumed throughout, so all quantities apply to a single species. The probability distribution at equilibrium is Maxwell-Boltzmann, which in spherical coordinates is

$$P_{\mathbf{v}} = \left(\frac{m}{2\pi k_B T} \right)^{3/2} v^2 e^{-mv^2/2k_B T}.$$

The most probable speed is obtainable by finding the maximum of the Maxwell-Boltzmann distribution with respect to molecular speed:

$$0 = \frac{d}{dv} P_{\mathbf{v}} = \left(\frac{m}{2\pi k_B T} \right)^{3/2} \left[2v_{\text{mp}} e^{-mv_{\text{mp}}^2/2k_B T} - v_{\text{mp}}^2 \left(\frac{mv_{\text{mp}}}{k_B T} \right) e^{-mv_{\text{mp}}^2/2k_B T} \right]$$

so

$$v_{\text{mp}} = \sqrt{\frac{2k_B T}{m}}. \quad (\text{A.1})$$

The mean molecular speed is calculated in the usual way. Immediately integrating over all angles because the velocity distribution is isotropic in space,

$$\langle v \rangle = \int_0^{\mathbf{v}_{\text{max}}} P_{\mathbf{v}} d\mathbf{v} = 4\pi \left(\frac{m}{2\pi k_B T} \right)^{3/2} \int_0^\infty dv v^3 e^{-mv^2/2k_B T}. \quad (\text{A.2})$$

Let $A = 4\pi \left(\frac{m}{2\pi k_B T} \right)^{3/2}$ and $\alpha = m/2k_B T$. Then Eqn. (A.2) has the form

$$\langle v \rangle = A \int_0^\infty v^3 e^{-\alpha v^2} dv = -\frac{A}{2\alpha} \int_0^\infty v^2 \cdot \frac{d}{dv} (e^{-\alpha v^2}) dv.$$

This can be integrated by parts to give

$$\begin{aligned} \langle v \rangle &= -\frac{A}{2\alpha} \left[v^2 e^{-\alpha v^2} \right]_0^\infty + \frac{A}{2\alpha} \cdot 2 \int_0^\infty \left(-\frac{1}{\alpha} \right) de^{-\alpha v^2} = -\frac{A}{2\alpha^2} \left[e^{-\alpha v^2} \right]_0^\infty \\ &= \frac{A}{2\alpha^2}. \end{aligned}$$

Substituting the expressions for A and α into the above gives

$$\langle v \rangle = \sqrt{\frac{8k_B T}{\pi m}}. \quad (\text{A.3})$$

The mean square speed is also calculated from the probability distribution:

$$\begin{aligned} \langle v^2 \rangle &= \int_0^{v_{\max}} v^2 P_v dv = A \int_0^\infty dv v^4 e^{-\alpha v^2} = A \int_0^\infty dv \frac{\partial^2}{\partial \alpha^2} (e^{-\alpha v^2}) \\ &= A \frac{\partial^2}{\partial \alpha^2} \left[\int_0^\infty e^{-\alpha v^2} dv \right]. \end{aligned} \quad (\text{A.4})$$

Evaluation of this integral uses the “trick” of moving to polar coordinates.

Denoting the integral as I , square it to obtain a double integral over two variables:

$$I^2 = \left[\int_0^\infty e^{-\alpha x^2} dx \right]^2 = \int_0^\infty e^{-\alpha x^2} dx \int_0^\infty e^{-\alpha y^2} dy = \iint_0^\infty e^{-\alpha(x^2+y^2)} dx dy.$$

Provided the integral is evaluated over the first quadrant of the x - y plane as above, it does not matter whether we use cartesian or plane polar coordinates. Noting that $x^2 + y^2 = \rho^2$ for a circle of radius ρ , change to polar coordinates:

$$\begin{aligned} I^2 &= \int_0^\infty \int_0^{\pi/2} \rho e^{-\alpha \rho^2} d\rho d\phi = \frac{\pi}{2} \int_0^\infty \left(\frac{1}{2\alpha} \right) e^{-\xi} d\xi \\ &= \frac{\pi}{4\alpha}. \end{aligned}$$

Thus, $I = \sqrt{\pi/4\alpha}$. Using this result in Eqn. (A.4), we get the mean square speed,

$$\begin{aligned}\langle v^2 \rangle &= A \frac{\partial^2}{\partial \alpha^2} \left(\frac{\pi}{4\alpha} \right)^{1/2} = A \left(\frac{3\sqrt{\pi}}{8\alpha^{5/2}} \right) \\ &= \frac{3k_B T}{m}\end{aligned}\tag{A.5}$$

and the root mean square speed,

$$v_{\text{rms}} = \sqrt{\frac{3k_B T}{m}}.\tag{A.6}$$

APPENDIX B

MEAN RELATIVE VELOCITY OF PARTICLES IN A GAS

Consider a mixture of dilute hard sphere gases, each comprised of molecules of different mass and diameter. In order to calculate the collision frequency between molecules we need an expression for the mean relative velocity between particles of each pair of molecular species in the gas. The analysis assumes that the gas is at equilibrium.

For a species a , the particle mass is m_a and its number density, $n = N/V$, is n_a . The relative velocity between two particles i and j is $\mathbf{v}_{r,ij}$. For a hard sphere gas, the interaction cross section between two species a and b molecules having diameters d_a and d_b , respectively, is simply

$$\sigma_{ab} = \pi \left(\frac{d_a + d_b}{2} \right)^2,$$

which assumes the cross section is independent of relative velocity.

We want an expression for the average relative velocity between molecules of species a and b , $\langle v_{r,ab} \rangle$, which will be denoted as $\langle v_r \rangle$ for notation simplicity in what follows. If $P_{\mathbf{v}_a}$ is the probability distribution for the velocity of species a molecules and $P_{\mathbf{v}_b}$ is the distribution for species b , then

$$\langle v_r \rangle = \langle |\mathbf{v}_a - \mathbf{v}_b| \rangle = \int_a \int_b d\mathbf{v}_a d\mathbf{v}_b |\mathbf{v}_a - \mathbf{v}_b| P_{\mathbf{v}_a} P_{\mathbf{v}_b}. \quad (\text{B.1})$$

The distribution is Maxwell-Boltzmann, so the probability $P_{\mathbf{v}}$ in cartesian coordinates is

$$P_{\mathbf{v}} = \left(\frac{m}{2\pi k_B T} \right)^{3/2} e^{-m v^2 / 2k_B T}.$$

Eqn. (B.1) becomes

$$\langle v_r \rangle = \left(\frac{\sqrt{m_a m_b}}{2\pi k_B T} \right)^3 \int_a \int_b d\mathbf{v}_a d\mathbf{v}_b |\mathbf{v}_a - \mathbf{v}_b| e^{-m_a v_a^2 / 2k_B T} e^{-m_b v_b^2 / 2k_B T}. \quad (\text{B.2})$$

To evaluate this integral, change to the center of mass coordinates to allow replacing the integration variables $\mathbf{v}_a$ and $\mathbf{v}_b$ with the center of velocity $\mathbf{v}_{cm}$ and relative velocity $\mathbf{v}_r$. Denoting $M \equiv m_a + m_b$, we have

$$\mathbf{v}_a = \frac{m_b}{M} \mathbf{v}_r + \mathbf{v}_{cm}, \quad \mathbf{v}_b = -\frac{m_a}{M} \mathbf{v}_r + \mathbf{v}_{cm}$$

so

$$\begin{aligned} m_a v_a^2 &= m_a \mathbf{v}_a \cdot \mathbf{v}_a = m_a \left(\frac{m_b}{M} \right)^2 v_r^2 + m_a v_{cm}^2 + 2 \left(\frac{m_a m_b}{M} \right) \mathbf{v}_r \cdot \mathbf{v}_{cm} \\ m_b v_b^2 &= m_b \mathbf{v}_b \cdot \mathbf{v}_b = m_b \left(\frac{m_a}{M} \right)^2 v_r^2 + m_b v_{cm}^2 - 2 \left(\frac{m_a m_b}{M} \right) \mathbf{v}_r \cdot \mathbf{v}_{cm}. \end{aligned}$$

Defining the reduced mass, $\mu \equiv (m_a m_b) / M$, and adding the preceding two equations, we get

$$\begin{aligned} m_a v_a^2 + m_b v_b^2 &= \frac{1}{m_a} \left(\frac{m_a m_b}{M} \right)^2 v_r^2 + m_a v_{cm}^2 + \frac{1}{m_b} \left(\frac{m_a m_b}{M} \right)^2 v_r^2 + m_b v_{cm}^2 \\ &= \left(\frac{1}{m_a} + \frac{1}{m_b} \right) \mu^2 v_r^2 + (m_a + m_b) v_{cm}^2 \\ &= \left(\frac{m_a + m_b}{m_a m_b} \right) \mu^2 v_r^2 + M v_{cm}^2 \\ &= \mu v_r^2 + M v_{cm}^2 \end{aligned}$$

Substitute this result into the mean relative velocity integral, Eqn. (B.2):

$$\langle v_r \rangle = \left(\frac{\sqrt{m_a m_b}}{2\pi k_B T} \right)^3 \int_a \int_b d\mathbf{v}_a d\mathbf{v}_b |\mathbf{v}_a - \mathbf{v}_b| e^{-(\mu v_r^2 + M v_{cm}^2) / 2k_B T}.$$

Up until now we have assumed a cartesian coordinate system. For present purposes it is preferable to work in spherical coordinates, since the relative velocity between two particles corresponds to the radial coordinate if the origin is at one of the particles; the angular parts correspond to the angular distribution of relative velocities. In spherical coordinates, then, $v_r \equiv |\mathbf{v}_a - \mathbf{v}_b|$. The velocity distributions are isotropic in space, and thus the integrals over angles are independent of the direction of motion, so the angular part, $\left(\int_0^\pi \sin\theta d\theta \int_0^{2\pi} d\phi\right)^2$ can be immediately integrated to give

$$\langle v_r \rangle = \left(\frac{\sqrt{m_a m_b}}{2\pi k_B T}\right)^3 (4\pi)^2 \int_a \int_b dv_a dv_b v_a^2 v_b^2 v_r e^{-(\mu v_r^2 + M v_{cm}^2)/2k_B T}. \quad (\text{B.3})$$

To evaluate this integral in closed form, two changes are necessary. First, the exponential term has the form of a product of Gaussian distributions, but the range of molecular speeds is necessarily finite because the energy is finite, and thus the range of integration is not infinite. To integrate a Gaussian in closed form, the limits must be $(0, \infty)$. By noting there is zero probability that a molecular speed will be higher than the maximum allowed by the energy, we can extend the upper limit to infinity without changing the result. Second, we need to change the integration variables from v_a and v_b to v_r and v_{cm} . In doing so the limits of integration do not change, because v_r and v_{cm} can each have the same range as v_a or v_b ; this can be seen by letting $v_a = 0$ and thus $v_r = v_b$ on the one hand, or $v_a = v_b$ and thus $v_{cm} = v_a$ on the other. With these changes Eqn. (B.3) becomes

$$\langle v_r \rangle = \frac{2}{\pi} \left(\frac{1}{2k_B T}\right)^3 (m_a m_b)^{3/2} \int_0^\infty \int_0^\infty dv_r dv_{cm} v_r^3 v_{cm}^2 e^{-(\mu v_r^2 + M v_{cm}^2)/2k_B T}.$$

Since v_r and v_{cm} are independent variables for a hard sphere gas, the exponential

terms and thus the integrals are separable:

$$\begin{aligned}\langle v_r \rangle &= \frac{2}{\pi} \left(\frac{1}{k_B T} \right)^3 (m_a m_b)^{3/2} \int_0^\infty dv_r v_r^3 e^{-\mu v_r^2 / 2k_B T} \int_0^\infty dv_{cm} v_{cm}^2 e^{-M v_{cm}^2 / 2k_B T} \\ &= \frac{2}{\pi} \left(\frac{1}{k_B T} \right)^3 (m_a m_b)^{3/2} \left[\frac{1}{2(\mu/2k_B T)^2} \cdot \frac{\sqrt{\pi}}{4(M/2k_B T)^{3/2}} \right].\end{aligned}$$

Evaluation of the two integrals to reach the final step above followed the same procedures as were used to evaluate Eqns. (A.4) and (A.5). This result reduces to

$$\langle v_r \rangle = \sqrt{\frac{8k_B T}{\pi \mu}}. \quad (\text{B.4})$$

For a single species, $m_a = m_b = m$, and so $\mu = m/2$. Hence, the mean relative velocity in this special case is

$$\langle v_r \rangle = 4 \sqrt{\frac{k_B T}{\pi m}}. \quad (\text{B.5})$$

APPENDIX C

LISTINGS OF DSMC SIMULATOR IMPLEMENTATION (C++)

C.1 Header File for Data Structures and Other Definitions

```
/*-----
 * FLUCT.H - Constant and data structure definitions for the DSMC
 * simulator.
 *-----*/

/*-----
 * Constants
 *-----*/

#define X 0 /* Predefined axis names: X, Y, Z */
#define Y 1
#define Z 2
#define PI ((double) 3.141592654)
#define BOLTZ ((double) 1.3806e-23) /* Boltzmann's constant (J/K) */
// Bits to identify particular plots or correlations
#define CORR_NN 0x00000001 /* For density-density correlations */
#define CORR_NP 0x00000002 /* For density-momentum correlations */
#define CORR_TT 0x00000004 /* For temp-temp correlations */
#define CORR_NK 0x00000008 /* For density-energy correlations */
// Macros for common calculations
#define NORM(x,y,z) sqrt((x)*(x)+(y)*(y)+(z)*(z)) /* Norm of a vector */
#define INDEX2(n,y,x) ((y)*(n)+(x)) /* Two-dimensional array index */
// Switches to turn on or off simulator modifications that handle
// particular, special-purpose tests:
#define CONSISTENCY_CHECK 0 /* Do consistency check in 'Mover' function*/
#define INTERNAL_CELL_HIST 0 /* Generate N in/out histogram for cell 17 */
#define COLLISIONLESS 0 /* Inhibit collisions after initial relax */

// Types that specify the kinds of initial and boundary conditions.
typedef enum {
    ic_max_boltz=0, ic_delta_function, ic_uniform, ic_chap_enskog
} InitCondType;
// Types of boundary conditions
typedef enum {
    bc_periodic=0, /* Periodic boundaries everywhere
    bc_specular_wall, /* Specular walls at x=0, x=L
    bc_thermal_wall, /* Thermal walls at x=0, x=L
    bc_reservoir /* Reservoir at boundary (open system)
} BoundaryCondType;
// Reservoir types
typedef enum {
    rt_generate_maxwell_boltz=0, /* Generate particles (M-B dist.)
    rt_generate_chap_enskog, /* Generate particles (Chapman-Enskog)
    rt_generate_fixed, /* Generate constant # particles (M-B)
    rt_closed_system, /* Use closed system for reservoir
```

```

    rt_thermal_wall                // Use thermal wall for reservoir
} ReservoirType;

// Structure to hold the characteristics of a boundary
typedef struct BoundaryType_ {
    double T;                        // Temperature
    double v_parallel;              // Flow velocity in x-dir. (reservoirs)
    double v_shear;                 // Shear velocity
    double mpv;                     // Most probable velocity of particles
    double mpv_stdev;               // Standard deviation in the MPV
    double position;                // Boundary's position on the axis
    double density;                 // Particle density (reservoir only)
    double length;                  // Length of reservoir (reservoir only)
    double mfp;                     // Mean free path in reservoir
    BoundaryCondType boundCond;     // Kind of boundary condition
    ReservoirType resType;          // Reservoir implementation method
    struct ParticleState_ *particles; // List of reservoir particles
    double vrMax;                   // Maximum relative velocity (reservoirs)
    double npart;                   // Number of particles (reservoirs)
} BoundaryType;

// Structure to hold common simulation parameters
typedef struct {
    double boltz;                   // Boltzmann's constant used for sim.
    double density;                 // Gas density (kg/m^3)
    double mass;                    // Particle mass (kg)
    double diam;                    // Effective particle diameter (m)
    double length;                  // System length (m)
    BoundaryType lbound, rbound;    // Specifications for boundaries
    double force;                   // Force on particle (e.g, gravity)
    double initCondParm1;           // Initial condition modifier #1, if any
    double initCondParm2;           // Initial condition modifier #2, if any
    double mfp;                     // Mean free path of particles
    double tau;                     // Time step (sec)
    double effNum;                  // # phys particles each part. represents
    double coeff;                   // Used for evaluating collisions
    long nstep;                     // Number of time steps per run
    long startSampStep;             // Time step when sampling should start
    long showSimProgress;           // Interval for displaying sim. progress
    long saveSnapshot;              // Interval for saving snapshots
    long nsnapshots;                // Total number of snapshots to take
    long boundaryHist;              // # histogram pts — vel. dist. at wall
    long correlations;              // Set of correlations of interest
    unsigned long randSeed;         // Seed for random number generator
    int ncell;                      // Number of cells in system length
    int npart;                      // Initial number of particles
    int nrun;                       // Total number of simulations to run
    int run;                        // Number of this run, starting from zero
    InitCondType initCond;          // Kind of initial conditions
    char name[80];                  // Simulation name
    char title[80];                 // Title for group of simulations
    char filename[128];             // Full name of file containing parms
} SimParmBlock;

// Structure for creating a doubly linked list
typedef unsigned short ListIndex; // Used for indices in linked list
typedef struct {
    ListIndex next;                 // Index to next item in list
    ListIndex prev;                 // Index to previous item in list
} ListLinks;

// Markers for list entry indices:
//   END_LIST denotes a list terminus, either that no entry precedes this
//   one (for 'prev' link) or follows this one (for 'next' link).
//   FREE_LIST denotes that a list entry is in the free list and does not
//   represent a particle in the system; always in 'prev' link.
#define END_LIST ((ListIndex) -1)

```

```

#define FREELIST      ((ListIndex) (1 << (8*sizeof(ListIndex)-1)))

// Structure to hold the state of a particle (periodic boundary conditions
// for y and z boundaries)
typedef struct ParticleState_ {
    double      x;                // Particle position
    double      vx, vy, vz;       // Particle velocity
    ListLinks    list;            // Links together particles in same cell
} ParticleState;

// Structure to hold information about particles entering or leaving
// the system through/at a single boundary, used for constructing histograms
typedef struct {
    long         v_enter;         // Vel. dist — particles entering system
    long         v_leave;        // Vel. dist — particles leaving system
    long         n_enter;        // Number of particles entering system
    long         n_leave;        // Number of particles leaving system
} HistogramData;

// Structure to hold statistics about a system wall
typedef struct {
    HistogramData *hist;         // Vel. dist. for particles leaving wall
    double        dpxTot, dpyTot; // Total momentum change at a wall
    double        dpyErr;        // Used to find error in dpyTot
    double        sum_py, sum_pz; // Net momentum for particles leaving wall
    long          strikeSum;      // Number of times particles struck wall
} WallStats;

// Structure to hold sampling statistics for a cell
typedef struct {
    double        avg_n;         // Average particle number in the cell
    double        avg_px,        // Average momentum/velocity components
                avg_py,         // per particle in the cell
                avg_pz,
                avg_KE;          // Sum of average kinetic energies
    double        sum_px,        // Net momentum in the cell
                sum_py,
                sum_pz,
                sum_KE;          // Cumulative  $m \cdot v^2$  for particles in cell
    double        avg_T;         // Average temperature in the cell
    double        collTot;       // Number of collisions in the cell
    double        l_tot_px;      // Total momentum through left boundary
    long          l_particle_net; // Net particle count thru left boundary
} CellStats;

// Types and structure to hold sampling statistics for correlations; the
// particular of data being correlated is simulation dependent:
// Function for doing sampling during the simulation:
typedef void (*CorrFn) (struct SimState & sim, struct CorrStats & stats);
// Structure to hold sampling statistics for a single kind of correlation.
// An array of these structures is created, one for each kind of
// correlation statistics to be gathered, allowing only the desired
// kinds of data for a particular simulation to be gathered, eliminating
// sampling for unneeded statistics.
typedef struct CorrStats_ {
    double        *avg_ij;       // Correlation data for cell_i, cell_j
    CorrFn        sampFn;        // Function used to perform sampling
    long          id;            // Identifies correlation (ordinal)
} CorrStats;

// Structure to hold information for statistical sampling
typedef struct {
    WallStats      lwall, rwall;  // Sampling statistics for each wall
    CellStats      *cells;        // Sampling statistics for each cell
    CorrStats      *correl;       // Correlation-related statistics
    double         energy;        // Total energy in system after timestep
    double         px, py, pz;    // Total momentum in system after timestep
    double         tsamp;         // Total sampling time
    double         collTot;       // Total number of collisions

```

```

    long      nsamp;                // Total number of samples
    long      ncorr;               // Number of corr. stats being gathered
    long      step;                // Time step number from start of sim.
} SampleStats;

// Structure to hold a cell's state
typedef struct {
    double    px;                  // Sum of x-components of particle momenta
    double    T;                  // Temperature
    double    energy;              // Total energy in the cell
    double    vrMax;               // Maximum relative velocity
    double    selExtra;            // Extra selections carried over to next timestep
    int       n;                   // Number of particles currently in the cell
    ListIndex head;                // Head of list of particles
    ListIndex index;              // Indices to first particle xref
} CellState;

// Structure to hold a simulation state
typedef struct SimState_ {
    SimParmBlock *pb;              // Points to simulation parameters
    CellState *cells;              // Points to state information for each cell
    ParticleState *ps;             // Points to array of particle states
    ParticleState *free;           // Free list for unused particles
    ListIndex *partXref;           // Xref indices for sorting particles into cells
    CellStats *cellInitStats;      // Initial averages for each cell
    SampleStats samp;              // Sampling statistics
    ListIndex npart;               // Current number of particles
    ListIndex maxPart;             // Maximum number of particles
    ListIndex lastPart;            // Index of last particle in the system
} SimState;

// Structure to hold pointers to functions needed to initialize, perform
// gathering of statistics, and saving statistics to a file for analyzing
// a single kind of correlation between cells.
typedef void (*OutputFn) (SimState& sim, struct CorrStats_& cs, FILE *file);
typedef struct {
    CorrFn    sampFn;              // Function used to perform sampling
    CorrFn    initFn;              // Function for initializing data
    OutputFn   saveFn;             // Function for saving data to file
    long      id;                  // Identifies the correlation (CORR_xx)
} CorrFuncList;

```

C.2 Simulator Initialization, Output and Exit Functions

```

/*
 * SIMMAIN.CPP - Initialization and clean-up functions for direct
 * simulation (DSMC) of a rarified gas in a microscopic
 * system. The system boundaries are periodic in two of
 * the dimensions, and two plates bound the system
 * in the third dimension. The plates may be fixed or
 * moving relative to each other, and they may be at
 * different temperatures. The system is divided into
 * cells along this third dimension. A simulation
 * determines correlations of fluctuations among the
 * different cells at steady state, and determines other
 * properties such as pressure on the walls and viscosity.
 *
 * Written by Marty Tysanner
 */

#include <stdlib.h>
#include <stdio.h>
#include <alloc.h>
#include <string.h>
#include <ctype.h>
#include <memory.h>
#include <math.h>

```

```

#include <float.h>
#include <fcntl.h>
#include <fstream.h>
#include <time.h>
#include "fluct.h"

extern CorrFuncList CorrFunctions[];

#ifdef __IBMPC__

#define FULLFILENAME(s) ((s)[1] == ':' || (s)[0] == '\\');
static char PathSepChar = '\\'; // Separates parts of paths to a file
static char ExtensionSepChar = '.'; // Separates parts of a file name
// List of characters that can start a file name or path, other than
// alphabetic characters:
static char *FileStartChar = "_0123456789~!@#%&~+{|}~";
static char *RelativePathStartChar = ".";

#endif

static char *ParmNames[] = {
    "runs", "end", "density", "mass",
    "diameter", "length", "T_lwall", "T_rwall",
    "v_lwall", "v_rwall", "steps", "cells",
    "particles", "initial", "boundaries", "startsample",
    "showprogress", "correlations", "timestep", "boltz",
    "force", "density_lwall", "density_rwall", "randseed",
    "snapshot", "boundary_hist", "reservoir_type", "reservoir_length",
    // Strings MUST be last for SaveSimStats() to work properly; if other
    // options are added, switch statement in GetSimParms() must be modified
    // so that the string cases have the correct number (new options that
    // aren't strings should always be added just before the first parameter
    // that contains a string argument).
    "title", "name",
    // Comment delimiter is last because it is handled as a default case
    // in GetSimParms().
    "/"
};

/*
 * CalcSimParms calculates some useful simulation parameters that are
 * not obtained from the parameter file or defaults. This is a helper
 * function for 'GetSimParms' to reduce clutter in that function.
 * Entry:
 * pb -> simulation parameter block
 * Exit:
 * none
 */

void CalcSimParms (SimParmBlock& pb)
{
    double boltz = pb.boltz; // Boltzmann's constant to be used
    double vol = pb.length * pb.length * pb.length; // System volume
    double mpv, T_mid;
    BoundaryType &lb = pb.lbound, &rb = pb.rbound;

    pb.effNum = pb.density * vol/pb.npart;
    // Convert velocities from Mach numbers to m/s
    lb.v.shear *= sqrt(5./3. * boltz*lb.T/pb.mass);
    rb.v.shear *= sqrt(5./3. * boltz*lb.T/pb.mass);
    // If the boundaries are reservoirs, we will need to calculate the
    // nominal particle density at each wall of the system in the absence
    // of a reservoir (using thermal walls to gather statistics), and
    // then scale this density by any factor provided in the simulation
    // control file. Calculating the nominal density is difficult for a
    // user to do accurately when the two boundary temperatures are unequal,
    // so it is done by the system during the system relaxation time. What
    // is needed here is to determine that the user has provided enough
    // relaxation time, and warn the user if not.

    if (lb.boundCond == bc_reservoir)
        lb.v_parallel = pb.initCondParm1; // x-velocity to add to particles
    if (rb.boundCond == bc_reservoir)

```

```

    rb.v_parallel = -pb.initCondParm1; // x-velocity to add to particles
    if ( (lb.boundCond == bc_reservoir || rb.boundCond == bc_reservoir) &&
        (double)pb.npart*(double)pb.npart > 800*(double)pb.startSampStep )
        cout << "Warning: _at_least_" << 800*pb.startSampStep << "_time_steps_"
            << "settling_time_is_recommended\n";

    switch (pb.initCond) {
        case ic_max_boltz: // ** Maxwell-Boltzmann dist. **
        case ic_delta_function: // ** Delta function position dist. **
            // Most probable velocities off of walls or out of reservoir
            lb.mpv = sqrt(2.0*boltz*lb.T/pb.mass);
            rb.mpv = sqrt(2.0*boltz*rb.T/pb.mass);
            if (pb.initCond == ic_max_boltz)
                break; // Done with Maxwell-Boltzmann dist.
            if (pb.initCondParm1 == 0)
                pb.initCondParm1 = DBL_EPSILON; // Small distance from wall
            else if (pb.initCondParm1 >= 1.0) {
                if (pb.initCondParm1 > 1.0)
                    cout << " _*_Warning_(simulation_#" << (pb.run+1)
                        << "): _position_of_delta_function_initial_condition\n"
                        << " _exceeds_system_length\n";
                pb.initCondParm1 = 1 - DBL_EPSILON; // Small distance from wall
            } /*if*/
            if (pb.initCondParm2 == 0)
                // Default temperature at the point is the value at the point
                // at steady state.
                pb.initCondParm2 = lb.T + pb.initCondParm1*(rb.T - lb.T);
            break;

        case ic_uniform: // ** Uniformly distributed vel. **
            // Most probable velocities
            lb.mpv = sqrt(3.0*boltz*lb.T/pb.mass);
            rb.mpv = sqrt(3.0*boltz*rb.T/pb.mass);
            break;
    } /*switch*/

    lb.mpv_stdev = lb.mpv/M_SQRT2; // Standard deviation for M-B dist.
    rb.mpv_stdev = rb.mpv/M_SQRT2; // M_SQRT=sqrt(2), defined in <math.h>

    pb.mfp = vol/(sqrt(2)*PI*(pb.diam*pb.diam) * pb.npart*pb.effNum);
    if (pb.tau == 0) {
        // For calculating the timestep, use the largest most prob. velocity:
        mpv = (rb.mpv > lb.mpv) ? rb.mpv : lb.mpv;
        pb.tau = 0.2 * (pb.length/pb.ncell)/mpv; // Timestep tau
    } /*if*/
    pb.ccoeff = 0.5 * pb.effNum*PI*(pb.diam*pb.diam) * pb.tau/(vol/pb.ncell);

    // Determine the length of a reservoir having the most probable velocity
    // and timestep. The length is determined by multiplying the calculated
    // cell width by the (specified or default) number of cell widths.
    T_mid = (rb.T + lb.T)/2; // Temperature at middle, approximately
    lb.length *= (lb.mfp = pb.mfp * lb.T/T_mid);
    rb.length *= (rb.mfp = pb.mfp * rb.T/T_mid);

    // If no timestep at which statistics gathering should start was
    // given, set the default system "settling time" to half of the
    // total simulation time.
    if (pb.startSampStep == -1)
        pb.startSampStep = pb.nstep/2; // Use default
} /* CalcSimParms */

/*
 * CleanupSim performs clean-up after completion of a simulation,
 * given a simulation state.
 * Entry:
 *   sim -> simulation state structure
 * Exit:
 *   none
 */
void CleanupSim (SimState& sim)

```

```

{
    free (sim.ps);
    for (int i = 0; i < sim.pb->ncell; i++)
        free (sim.cells[i].snapshot);
    free (sim.cells);
    free (sim.cellInitStats);
    free (sim.samp.cells);
    if (sim.samp.correl != NULL)
        free (sim.samp.correl);
    if (sim.pb->boundaryHist > 0) {
        free (sim.samp.lwall.hist);
        free (sim.samp.rwall.hist);
    } /*if*/
    if (sim.pb->lbound.particles != NULL)
        free (sim.pb->lbound.particles);
    if (sim.pb->rbound.particles != NULL)
        free (sim.pb->rbound.particles);
} /* CleanupSim */



---


/*
 * EvaluateExp evaluates an expression. Currently the only kind of
 * expression that is supported is a simple sum of integers. A more
 * sophisticated expression evaluator could be implemented, but it is
 * not clear that would be especially valuable at this time; if the
 * grammar for defining the input file is made more general purpose
 * or free-form, then a LALR parser could be developed for it, at
 * which time the expression evaluator could be made more sophisticated
 * too. For now, though, a general parser kind of implementation seems
 * like serious overkill.
 * Entry:
 *     buf -> buffered character string containing the expression
 *     kind -> character string describing the expression
 * Exit:
 *     double precision expression result
 */
double EvaluateExp (char *buf, char *kind)
{
    char *p, *s;
    long result, value;
    int error;

    result = 0; error = 0;
    for (s = buf; *s == ' ' || *s == '\t'; s++) ; // Skip whitespace
    while (*s != 0) {
        value = strtol(s, &p, 0);
        if (!isdigit(*s) && *s != '-' && *s != '+') ||
            (value == 0 && *(p-1) != '0') {
            error = 1; // Invalid expression
            break;
        } /*if*/
        result += value; // Assume a valid correlation ID
        for (s=p; *s == ' ' || *s == '\t'; s++) ; // Skip blanks
        if (strncmp(s, "//", 2) == 0)
            break; // Start of a comment; done with line
        else if (*p != 0 && *p != '+') {
            error = 1; // Invalid expression
            break;
        } /*if*/
    } /*while*/
    if (error) { // Check for an error
        cout << "***Invalid expression for_" << kind << "_in_file\n";
        result = 0.0;
    } /*if*/

    return result;
} /* EvaluateExp */



---


/*
 * GetCmdLine extracts all parameters from the command line that are
 * supported by the program. The format of the command line is

```



```

*      SIM.EXE <options> <filename>
*      where <options> is a sequence of options separated by one or more
*      blanks and <filename> is the name of the simulation parameters file.
*      An option always begins with a dash, followed immediately by a
*      single character that identifies the option, followed immediately
*      by the value of the option. An option given on the command line
*      overrides any corresponding value in the parameter file.
*
*      Currently supported options are:
*      -h -> print help message and exit
*      -s -> skip the specified simulation given in the file (integer >= 1)
*      -t -> run all simulations for the specified number of time steps
*
*      Entry:
*      argC -> number of command line arguments
*      argS -> array of command line argument strings
*      pb -> simulation parameter block to be updated (the number of
*      simulations defined in the file that are to be actually
*      run is correctly set in the 'nrun' member, and the set
*      of any simulations to be skipped are set in the
*      'run' member)
*
*      Exit:
*      boolean value, TRUE if command line is valid
*
*-----*/
int GetCmdLine (int argC, char *argS[], SimParmBlock& pb)
{
    static char *help =
        "\n"
        "The format of the command line is\n"
        "SIM.EXE<options><filename>\n"
        "where<options>is a sequence of options separated by one or more\n"
        "blanks and<filename>is the name of the simulation parameters file.\n"
        "An option always begins with a dash, followed immediately by a\n"
        "single character that identifies the option, followed immediately\n"
        "by the value of the option. An option given on the command line\n"
        "overrides any corresponding value in the parameter file.\n"
        "\n"
        "Currently supported options are:\n"
        "-h-> print this help message and exit\n"
        "-s-> skip the specified simulation given in the file (integer >= 1)\n"
        "-t-> run all simulations for the specified number of time steps\n";

    char *p, *s;
    int iarg, skipped, temp;
    char buf[24];
    char fname[sizeof(pb.filename)] = { 0 };
    char hasFullPath;
    fstream pfile;

    if (argC <= 1) {
        cout << "\n (For help: SIM-h)\n"; // No cmd line; print help and exit
        return 0;
    } /* if */

    fname[sizeof(pb.filename)-1] = 0; // In case the file name is truncated
    skipped = 0; // Variable used to mark runs to skip
    pb.nrun = 0;
    pb.run = 0; // Used for returning sims. to skip

    for (iarg=1, s=argS[1]; iarg < argC && *(s=argS[iarg]) == '-'; iarg++) {
        switch (s[1]) {
            case 'h': // ** help **
                cout << help;
                return 0; // Don't want to show any error messages
            case 's': // ** Skip a simulation in the file **
                temp = 1 << (strtol(s+2, NULL, 0) - 1);
                if ((pb.run & temp) == 0)
                    skipped |= temp; // Indicate the simulation to skip
                break;
            case 't': // ** Set the simulation length **

```

```

        pb.nstep = strtol(s+2, NULL, 0);
        break;
    default:
        cout << "Unknown_command_line_option,_" << s << "'\n";
    } /*switch*/
} /*for*/

// The next parameter should be the last one on the command line, and
// it should be the name of the file containing the simulation
// parameters. (Machine-dependent file/path definitions required here
// are defined at the beginning of this source file.)

if (iarg != argC-1)
    cout << "File_name_must_be_last_command_parameter\n";
else {
    hasFullPath = FULLFILE_NAME(s); // True if file name contains full path
    if (hasFullPath)
        strncpy (fname, s, sizeof(pb.filename)-1);
    else if (isalpha(*s) ||
             strchr(FileStartChar, *s) != NULL ||
             strchr(RelativePathStartChar, *s) != NULL) {
        // File name or relative path was given. Assuming the
        // directory containing the program is the starting point for
        // finding the the given file name, use the path to the program,
        // which is presumed to be fully qualified with any path.
        strncpy (fname, argS[0], sizeof(pb.filename)-1); // Program file
        p = strchr(fname, PathSepChar) + 1; // End of path to program
        strncpy (p, s, sizeof(pb.filename)-1-(p-fname)); // File name
    }
    else {
        cout << "Invalid_file_name,_" << s << "'\n";
    } /*if*/
} /*if*/

if (fname[0] == 0)
    return 0; // Something invalid in the command line

// Verify the file exists and do a quick validity check on it. Also
// determine the number of simulations that are defined in the file.

pfile.open(fname, 1); // Try to open the named file
if (pfile.bad())
    cout << "Unable_to_find_parameter_file,_" << fname << "'\n";
else {
    // A parameter file was supplied.
    pfile.get(buf, 20, '_'); buf[pfile.gcount()] = 0;
    if (strcmp(buf, "runs") != 0)
        cout << "Parameter_file_" << fname << "'_has_invalid_format\n";
    else {
        pfile >> temp; // Number of simulations defined in file
        if (temp > 16) temp = 16; // Maximum of 16 runs
        pb.nrun = temp;
        pfile.close(); // Caller can reopen it...
        strcpy (pb.filename, fname); // Save the fully qualified file name
    } /*if*/
} /*if*/

// If any simulations are being skipped, display which ones.
if (*fname != 0 && skipped != 0) {
    cout << "Skipping_simulations:_";
    s = "_";
    for (int i = 0; i < temp; i++)
        if (skipped & (1 << i)) {
            pb.nrun--;
            pb.run |= 1 << i; // Indicate this simulation is skipped
            cout << s << (i+1);
            s = ",_";
        } /*if*/
}

```

```

    cout << "\n";
    if (pb.nrun <= 0) {
        cout << "\n(All simulations in parameter file have been skipped)\n";
        fname[0] = 0; // Indicate a fatal error
    } /* if */
} /* if */

return (fname[0] != 0);
} /* GetCmdLine */

/*
 * GetSimParms reads the simulation parameters from the command line
 * and/or file specified on the command line and fills out a simulation
 * parameter block. Where a parameter is not explicitly given,
 * defaults are used. Additionally, for convenience, some commonly
 * needed values are calculated from these parameters and placed into
 * the parameter block.
 * Entry:
 *   argC -> number of command line arguments
 *   argS -> array of command line argument strings
 *           argS[0] -> program file name (specified by "C" and "C++")
 *           argS[1 .. argC-2] -> program options (optional)
 *           argS[argC-1] -> parameter file name (required)
 * Exit:
 *   pointer to parameter block
 */
SimParmBlock *GetSimParms (int argC, char *argS [])
{
    const SimParmBlock dfltParms = { // Argon at STP
        BOLTZ, // Boltzmann's constant to use
        1.78, // Gas density (kg/m^3)
        6.63e-26, // Particle mass (kg)
        3.66e-10, // Effective particle diameter (m)
        1e-6, // System length (m)
        { // Left boundary (x=0)
            273, // Temperature
            0, // Shear velocity
            0, // Flow velocity in x-dir. (reservoirs)
            0, // Most probable velocity of particles
            0, // Standard deviation in the MPV
            0, // Boundary's position on the axis
            1, // Particle density (reservoir only)
            1, // Length of reservoir (reservoir only)
            0, // Mean free path in reservoir
            { // Info for Chapman-Enskog velocity
                { 0, 0, 0 }, // Heat flux vector
                { { 0,0,0 }, { 0,0,0 }, { 0,0,0 } }, // Stress tensor
                0, // Amplitude of distribution envelope
                0, // Nonzero if x-comp always positive
                0 // Nonzero if stress has nonzero value
            },
            bc_periodic, // Kind of boundary condition
            rt_generate_maxwell_boltz, // Method of reservoir implementation
            NULL, // List of reservoir particles
            0.0, // Maximum relative velocity (reservoirs)
            0, // Number of particles (reservoirs)
        },
        { // Right boundary (x=L)
            273, // Temperature
            0, // Flow velocity in x-dir. (reservoirs)
            0, // Shear velocity
            0, // Most probable velocity of particles
            0, // Standard deviation in the MPV
            1e-6, // Boundary's position on the axis
            1, // Particle density (reservoir only)
            1, // Length of reservoir (reservoir only)
            0, // Mean free path in reservoir
        }
    };
}

```

```

    {
        { 0, 0, 0 }, // Info for Chapman-Enskog velocity
        { {0,0,0},{0,0,0},{0,0,0} }, // Heat flux vector
        0, // Stress tensor
        0, // Amplitude of distribution envelope
        0, // Nonzero if x-comp always positive
        0, // Nonzero if stress has nonzero value
    },
    bc_periodic, // Kind of boundary condition
    rt_generate_maxwell_boltz, // Method of reservoir implementation
    NULL, // List of reservoir particles
    0.0, // Maximum relative velocity (reservoirs)
    0, // Number of particles (reservoirs)
},
0, // Constant force acting on particles
0, 0, // Initial condition modifiers
0, // Mean free path of particles
0, // Time step (s)
0, // # phys particles each part. represents
0, // Used for evaluating collisions
5000, // Number of time steps per run
-1, // Time step when sampling should start
1000, // Interval for displaying sim. progress
0, // Interval for saving snapshots
0, // Total number of snapshots to take
0, // # histogram pts — vel. dist. at wall
0, // Set of correlations of interest
1, // Seed for random number generator
20, // Number of cells in system length
2000, // Initial number of particles
0, // Total number of simulations to run
0, // Number of this run, starting from zero
ic_max_boltz, // Kind of initial conditions
{ 0 } // Simulation name
};

SimParmBlock *pb;
SimParmBlock pbOptions; // Used to save command line options
double temp;
fstream pfile;
char *p, *s;
int i, iBlock, id, isim, skipped;
char buf[100] = {0};

// Process the command line to get the name of the file that contains
// the simulation parameters, as well as any simulation parameter
// overrides.

memcpy (&pbOptions, &dfltParms, sizeof(pbOptions)); // Copy default values
if (!GetCmdLine (argC, argS, pbOptions))
    return (SimParmBlock *)NULL;

skipped = pbOptions.run; // Set of simulations to skip
pfile.open(pbOptions.filename, 1); // Open the simulation parameter file

// Initially create structure using default values. Only those simulation
// parameters explicitly provided in the parameter file or on the command
// line will be overwritten.

pb = (SimParmBlock *) calloc(sizeof(SimParmBlock), pbOptions.nrun);
memcpy ((void *)pb, (void *)&dfltParms, sizeof(dfltParms));
pb->nrun = pbOptions.nrun; // Total number of simulations in file
iBlock = 0; // Current parameter block
pb->run = 0; // Number of this run, starting from zero
isim = 0; // Need to count actual simulation number

while (iBlock < pbOptions.nrun) {
    // Read the next parameter keyword.
    pfile.get(buf, 20, '_'); // Next keyword from file
    buf[pfile.gcount()] = 0;
    for (i = 0;

```

```

        i < sizeof(buf)-1 && (buf[i] == '_' || iscntrl(buf[i]));
        i++;
    if (i > 0)
        strcpy(buf, buf+i); // Left justify the keyword
    for (id = 0; id < sizeof(ParmNames)/sizeof(ParmNames[0]); id++)
        if (strcmp(buf, ParmNames[id]) == 0)
            break; // This is a known parameter

    switch (id) {
    case 0: // ** runs **
        pfile >> temp; // We already know correct run count
        break;

    case 1: // ** end **
        pfile >> temp; // Discard simulation number for now
        if ((skipped & (1 << isim)) == 0) {
            // There was no command line option requesting this
            // simulation to be skipped, so finish the parm block.
            // First set any parameter overrides given on the command
            // line, then calculate remaining simulation values. (If
            // the simulation were being skipped, the next simulation's
            // information would simply overwrite this one's.)
            if (pbOptions.nstep != dfltParms.nstep)
                pb->nstep = pbOptions.nstep;
            strcpy (pb->filename, pbOptions.filename);
            if (pb->title[0] == 0 && pbOptions.title[0] != 0)
                strcpy (pb->title, pbOptions.title); // Default title
            pb->run = isim; // Actual simulation number
            if (*pb->name == 0)
                sprintf (pb->name, "simulation_%d", isim+1);
            CalcSimParms (*pb); // Calculate some useful values
            iBlock++; pb++; // Done with this parm block
        } /*if*/
        if (iBlock < pbOptions.nrun) {
            // Initialize the next parameter block with default values
            // in case some parameters are not supplied in the file.
            memcpy ((void *)pb, (void *)&dfltParms, sizeof(dfltParms));
            pb->nrun = pbOptions.nrun; // Number of simulations to run
            pb->run = iBlock; // Number of this run, starting from zero
            strcpy (pb->filename, pbOptions.filename); // Parm file name
        } /*if*/
        isim++;
        break;

    case 2: // ** density **
        pfile >> pb->density;
        break;

    case 3: // ** mass **
        pfile >> pb->mass;
        break;

    case 4: // ** diameter **
        pfile >> pb->diam;
        break;

    case 5: // ** length **
        pfile >> pb->length;
        pb->rbound.position = pb->length;
        break;

    case 6: // ** T_lwall **
        pfile >> pb->lbound.T;
        break;

    case 7: // ** T_rwall **
        pfile >> pb->rbound.T;
        break;

    case 8: // ** v_lwall **
        pfile >> pb->lbound.v_shear; // Velocity as a Mach number
        break;

    case 9: // ** v_rwall **
        pfile >> pb->rbound.v_shear; // Velocity as a Mach number
        break;

    case 10: // ** steps **

```

```

    pfile >> pb->nstep;
    break;
case 11:                                // ** cells **
    pfile >> pb->ncell;
    break;
case 12:                                // ** particles **
    pfile >> pb->npart;
    break;
case 13:                                // ** initial **
    pfile >> id;
    pb->initCond = (InitCondType) id;
    if (pb->initCond != ic_delta_function) {
        pfile.get(buf, sizeof(buf), '\n'); // Check for initial x-vel.
        if (buf[0])
            pb->initCondParm1 = strtod (buf, &p); // Stream velocity
    }
    else {
        pfile.get(buf, sizeof(buf), '\n'); // Get additional info
        pb->initCondParm1 = strtod (buf, &p); // Location in system
        pb->initCondParm2 = strtod (p, &p); // Temperature if nonzero
    } //if*/
    break;
case 14:                                // ** boundaries **
    pfile >> id;
    pb->lbound.boundCond = pb->rbound.boundCond = (BoundaryCondType)id;
    pfile.get(buf, sizeof(buf), '\n'); // Check for right bound. value
    for (i = 0; i < (int)strlen(buf); i++)
        if (isalnum(buf[i]) || memcmp(buf+i, "/", 2) == 0)
            break;
    if (isalnum(buf[i]))
        // The boundary condition for the right side was explicitly
        // given, so it may be different than that of the left side.
        pb->rbound.boundCond = (BoundaryCondType) strtol(buf+i, &p, 0);
    if (pb->lbound.boundCond == bc_periodic ||
        pb->rbound.boundCond == bc_periodic)
        // Make sure both boundaries are periodic.
        pb->lbound.boundCond = pb->rbound.boundCond = bc_periodic;
    break;
case 15:                                // ** startsample **
    pfile >> pb->startSampStep;
    break;
case 16:                                // ** showprogress **
    pfile >> pb->showSimProgress;
    if (pb->showSimProgress <= 0)
        // Want progress display disabled
        pb->showSimProgress = 0x7FFFFFFF;
    break;
case 17:                                // ** correlations **
    pfile.get(buf, sizeof(buf), '\n'); // Get kinds
    pb->correlations = (long) EvaluateExp(buf, "correlation_type(s)");
    break;
case 18:                                // ** timestep **
    pfile >> pb->tau;
    break;
case 19:                                // ** boltz **
    pfile >> pb->boltz; // Boltzmann's constant to use
    break;
case 20:                                // ** force **
    pfile >> pb->force; // Constant force acting on a particle
    break;
case 21:                                // ** density_lwall **
    pfile >> pb->lbound.density;
    break;
case 22:                                // ** density_rwall **
    pfile >> pb->rbound.density;
    break;
case 23:                                // ** randseed **
    pfile >> pb->randSeed;

```

```

        break;
    case 24:                                // ** snapshot **
        pfile >> pb->saveSnapshot; // Frequency of saving cell state
        pfile >> pb->nsnapshots; // Total number of snapshots to take
        break;
    case 25:                                // ** boundary_hist **
        pfile >> pb->boundaryHist; // histogram length for vel dist at walls
        break;
    case 26:                                // ** reservoir_type **
        pfile >> id; // Reservoir implementation method to use
        pb->lbound.resType = pb->rbound.resType = (ReservoirType) id;
        pfile.get(buf, sizeof(buf), '\n'); // Check for right bound. value
        if (buf[0])
            // The boundary condition for the right side was explicitly
            // given, so it may be different than that of the left side.
            pb->rbound.resType = (ReservoirType) strtol(buf, &p, 0);
        break;
    case 27:                                // ** reservoir_length **
        pfile >> pb->lbound.length; // Length in number of mean free paths
        pfile.get(buf, sizeof(buf), '\n'); // Check for right bound. value
        if (buf[0])
            pb->rbound.length = strtod(buf, &p);
        else
            pb->rbound.length = pb->lbound.length; // Use same as left side
        break;
    case 28:                                // ** title **
    case 29:                                // ** name **
        p = (strcmp(buf, "title") == 0) ? pb->title : pb->name;
        pfile.get(buf, sizeof(buf), '\n');
        for (s = buf; *s == '_' || *s == '\t'; s++) ;
        if (strlen(s) > sizeof(pb->name)-1) {
            memcpy(p, s, sizeof(pb->name)-1); // Truncate the name
            p[sizeof(pb->name)-1] = 0;
        }
        else
            strcpy(p, s); // Copy the entire name
        if (p == pb->title && pbOptions.title[0] == 0)
            strcpy(pbOptions.title, s); // Save the title for others too
        break;
    default:                                // This includes comment prefix "/"
        break;
} /*switch*/

pfile.ignore(65535, '\n'); // Discard rest of line, incl. comments
} /*while*/

pb == pbOptions.nrun; // Start of the parm block array
return pb;
} /* GetSimParms */

/*
 * InitSim initializes the simulation state, given a set of simulation
 * parameters.
 * Entry:
 *   pb -> simulation parameters
 *   sim -> simulation state structure to be initialized
 * Exit:
 *   none
 */
void InitSim (SimParmBlock& pb, SimState& sim)
{
    ParticleState *pp, *ps;
    CellState *cells;
    CellStats *cs;
    CorrStats *corrSt;
    ListIndex **ppNext;
    BoundaryType &lb = pb.lbound, &rb = pb.rbound, *bound;
    double frac, L, mpv, stdDev, T;
    double T_lwall, T_rwall, v_flow, v_lwall, v_rwall;

```

```

double      boltz;
double      adjust, v[3], v2;
long        corr, count;
int         icell, j, ncell;
ListIndex   i, ipart, npart, pcount;

memset ((void *)&sim, 0, sizeof(sim)); // Many members will be zero
sim.pb = &pb;
npart = (ListIndex) pb.npart; // Just for convenience...
ncell = pb.ncell;
boltz = pb.boltz; // Boltzmann's constant to use

// Allocate storage for arrays
count = npart;
if (pb.lbound.boundCond == bc_reservoir ||
    pb.rbound.boundCond == bc_reservoir)
    // There will be a variable number of particles in the system
    // because particles will enter and leave via the reservoir(s) at
    // each timestep during the particle movement phase. We want to
    // make sure there are plenty of extra unused particle
    // states available, so allocate twice as many as the initial number
    // of particles in the system.
    count *= 2;
sim.maxPart = (ListIndex) count;

sim.ps = (ParticleState *) calloc(sizeof(ParticleState), count);
sim.partXref = (ListIndex *) calloc(sizeof(ListIndex), count);
sim.cells = (CellState *) calloc(sizeof(CellState), ncell);
sim.samp.cells = (CellStats *) calloc(sizeof(CellStats), ncell);
for (i = 0; i < ncell; i++)
    sim.cells[i].snapshot =
        (SnapshotInfo *) malloc(sizeof(SnapshotInfo)*pb.nsnapshots);
sim.cellInitStats = (CellStats *) calloc(sizeof(CellStats), ncell);
// Histograms for velocity distribution of particles entering and
// leaving the system at the boundaries (optional):
if (pb.boundaryHist > 0) {
    sim.samp.lwall.hist =
        (HistogramData *) calloc(pb.boundaryHist, sizeof(HistogramData));
    sim.samp.rwall.hist =
        (HistogramData *) calloc(pb.boundaryHist, sizeof(HistogramData));
} /*if*/

// Temporary array used to link together all particles in a given cell:
ppNext = (ListIndex **) malloc(sizeof(ListIndex **)*ncell);

// Initialize the random number generator
sngenrand (pb.randSeed);

// Initialize some arrays
cells = sim.cells;
for (i = 0; i < ncell; i++) {
    mpv = lb.mpv + ((i+0.5)/(double)ncell)*(rb.mpv-lb.mpv);
    cells[i].vrMax = 3.0*mpv; // Est. max relative speed in a cell
    cells[i].selExtra = 0;
    cs = &sim.samp.cells[i]; // Cell statistics
    cs->avg_n = cs->avg_T = cs->collTot
        = cs->avg_px = cs->avg_py = cs->avg_pz = cs->avg_KE
        = cs->sum_px = cs->sum_py = cs->sum_pz = cs->sum_KE
        = 0;
    ppNext[i] = &cells[i].head; // Used to sort particles into cells
    *ppNext[i] = ENDLIST;
} /*for*/

// Initialize other floating point variables to zero.
sim.samp.lwall.dpxTot = sim.samp.rwall.dpxTot = 0; // Wall statistics
sim.samp.lwall.dpyTot = sim.samp.rwall.dpyTot = 0;
sim.samp.lwall.dpyErr = sim.samp.rwall.dpyErr = 0;
sim.samp.lwall.sum_py = sim.samp.lwall.sum_pz = 0;

```



```

sim.samp.rwall.sum_py = sim.samp.rwall.sum_pz = 0;
sim.samp.nsamp = sim.samp.collTot = sim.samp.tsamp = 0;

// Set the initial position of each particle and sort the particles
// into cells. For all initial particle distributions other than
// "delta function" (where all particles are at the same point), weight
// the particle distribution in accordance with any initial temperature
// gradient. Even if periodic boundary conditions are specified, an
// initial temperature gradient can be imposed by setting the "wall"
// (boundary) temperatures. Assuming ideal gas behavior and constant
// pressure and volume, the particle number in the presence of a
// temperature gradient will equal npart/ncell at the center and
//  $N_1/T_1 = N_2/T_2 \Rightarrow N_2 = N_1 * T_1/T_2 = (npart/ncell) * T_1/T_2$ 
// elsewhere, where  $N_1 = npart/ncell$  is the number of particles at
// the center and  $T_1$  is the temperature at the center.

L = pb.rbound.position - pb.lbound.position; // System length
for (i=0, ps=sim.ps; i < npart; i++, ps++) {
    if (pb.initCond == ic_delta_function)
        ps->x = L * pb.initCondParm1; // All particles at same x-coordinate
    else
        // Assign random positions, taking into account any temp. gradient.
        ps->x = L * genrand(); // Assign random positions
        icell = (int) (((double)ncell)*ps->x/L); // Determine the particle's cell
        icell -= (icell >= ncell); // Handle "overflow" case
        ps->list.prev = *ppNext[icell]; // Link previous particle in cell
        *ppNext[icell] = i; // to this one
        ppNext[icell] = &ps->list.next;
        ps->list.next = i; // Needed to set 'prev' of next particle
        cells[icell].n++; // One more particle in this cell
} /*for*/

sim.npart = sim.lastPart = npart; // Current number of particles

// If there are any unused particle structures, link them together and
// put them into the free list.
if (i < count) {
    sim.free = ps; // First unused particle structure
    for (; i < count; i++, ps++)
        ps->list.next = (ListIndex) (i + 1);
    (ps - 1) ->list.next = END_LIST; // Mark the end of the free list
    (ps - 1) ->list.prev = FREE_LIST; // Mark the structure as unused
} /*if*/

for (i = 0; i < ncell; i++)
    *ppNext[i] = END_LIST; // Mark the end of the particle lists
free (ppNext); // Don't need this array anymore

// Set the initial velocities of all particles
ps = sim.ps; // Extract some variables for convenience
T_lwall = lb.T; T_rwall = rb.T;
v_lwall = lb.v_shear; v_rwall = rb.v_shear; // Shear velocities at walls
// Calculate the flow velocity in the x-direction.
if (pb.initCondParm1 != 0)
    v_flow = pb.initCondParm1; // User-specified new flow speed
else
    v_flow = rb.v_parallel - lb.v_parallel; // Net flow from boundaries

for (i=0, pp=ps; i < npart; i++, pp++) {
    frac = pp->x/L; // Fraction of distance to right wall
    T = T_lwall + frac*(T_rwall-T_lwall); // Temperature depends on position
    // Calculate the most probable velocity for uniform distributions,
    // considering any temperature gradient.
    mpv = lb.mpv + frac*(rb.mpv - lb.mpv);

    switch (pb.initCond) {
        case ic_max_boltz: // ** Maxwell-Boltzmann dist. **
        case ic_delta_function: // ** Delta function distributed **
    }
}

```

```

    if (pb.initCond == ic.delta.function)
        T = pb.initCondParm2; // Use specified temperature
        // Assign thermal velocities using Gaussian random numbers
        stdDev = sqrt(boltz*T/pb.mass); // Standard deviation
        pp->vx = stdDev * genrandn();
        pp->vy = stdDev * genrandn();
        pp->vz = stdDev * genrandn();
        break;

    case ic_uniform: // ** Uniformly distributed vel. **
        // Only the x-component of the velocity is nonzero.
        pp->vx = mpv * (1. - 2.0*floor(2.0*genrand()));
        break;
} /*switch*/
} /*for*/

if (pb.initCond != ic.delta.function) {
    // Temperature is measured in the center of mass frame of references, so
    // adjust all velocity components to give a zero net momentum with each
    // cell while maintaining the desired temperature.

    for (icell = 0; icell < ncell; icell++) {
        frac = double(icell)/double(ncell); // Fraction of distance to right wall
        T = T_lwall + frac*(T_rwall-T_lwall); // Temperature depends on position
        pcount = ListIndex(cells[icell].n); // Number of particles in this cell
        if (pcount == 0) continue;

        // First measure the center of mass velocity.
        ipart = cells[icell].head; // Head of list of particles in cell
        v[X] = v[Y] = v[Z] = v2 = 0; // Used for giving zero c.m. velocity
        while (ipart != END_LIST) {
            pp = ps + ipart; // Points to particle state
            v[X] += pp->vx; v[Y] += pp->vy; v[Z] += pp->vz; // Calc c.m. velocity
            ipart = pp->list.next; // Next particle in this cell
        } /*while*/
        v[X] /= pcount; v[Y] /= pcount; v[Z] /= pcount; // Actual c.m. velocity

        // Adjust the velocity components so there is a zero mean velocity in
        // all three directions. Then measure the mean square velocity so
        // that the actual cell temperature can be calculated.
        ipart = cells[icell].head; // Head of list of particles in cell
        while (ipart != END_LIST) {
            pp = ps + ipart; // Points to particle state
            pp->vx -= v[X]; pp->vy -= v[Y]; pp->vz -= v[Z]; // Adjust c.m. vel.
            v2 += pp->vx*pp->vx + pp->vy*pp->vy + pp->vz*pp->vz; // For temp calc
            ipart = pp->list.next; // Next particle in this cell
        } /*while*/

        // It is unlikely that the randomly generated velocities will result
        // in a reservoir temperature that matches what is expected. Adjust
        // all components of all particle velocities so that the temperature
        // in the center of mass frame is what is expected.
        v2 = v2/pcount * pb.mass/(3.0 * boltz); // Actual temperature
        adjust = sqrt(T/v2); // Factor for adjusting mean square speed
        v[X] = v[Y] = v[Z] = 0; // For readjusting the c.m. velocity
        ipart = cells[icell].head; // Head of list of particles in cell
        while (ipart != END_LIST) {
            pp = ps + ipart; // Points to particle state
            // Adjust the particle velocity by increasing (decreasing) the
            // magnitude of the mean square speed by the calculated factor.
            pp->vx *= adjust; pp->vy *= adjust; pp->vz *= adjust;
            v[X] += pp->vx; v[Y] += pp->vy; v[Z] += pp->vz; // Calc c.m. velocity
            ipart = pp->list.next; // Next particle in this cell
        } /*while*/

        // Finally, adjust the mean velocity so that the center of mass
        // velocity has the specified x-component ("flow") and y-component
        // ("shear"); the z-component should have a mean of zero.
        v[X] /= pcount; v[Y] /= pcount; v[Z] /= pcount; // Actual c.m. velocity
    }
}

```

```

ipart = cells[icell].head;          // Head of list of particles in cell
while (ipart != ENDLIST) {
    pp = ps + ipart;                // Points to particle state
    pp->vx += -v[X] + v.flow;        // Add in the flow velocity
    // Include any velocity gradient in the y-component
    pp->vy += -v[Y] + v.lwall + frac*(v.rwall - v.lwall);
    pp->vz -= v[Z];                  // Want zero for mean z-component
    ipart = pp->list.next;           // Next particle in this cell
} /*while*/
} /*for*/

// For each reservoir boundary that is to be implemented as a closed
// system of particles at equilibrium, initialize the positions and
// velocities of the constituent particles using a Maxwell-Boltzmann
// distribution. This requires that the reservoirs be at internal
// equilibrium.
for (bound=&lb, j=0; j < 2; j++, bound=&rb) {
    bound->vrMax = 3.0*bound->mpv;    // Est. max relative speed in reservoir
    if (bound->boundCond != bc_reservoir ||
        bound->resType != rt_closed_system)
        continue;                  // Not a closed system reservoir
    bound->npart = pcount             // Number of particles in the reservoir
        = ListIndex( bound->density * double(npart)
            * bound->length/(rb.position - lb.position) );
    bound->particles = (ParticleState*) calloc(int(pcount), sizeof(*ps));
    L = bound->length;                // Reservoir length
    T = bound->T;                      // Use specified temperature
    stdDev = sqrt(boltz*T/pb.mass);  // Standard deviation
    v[X] = v[Y] = v[Z] = 0;          // Used for giving zero c.m. velocity
    for (i = 0, ps = bound->particles; i < pcount; i++, ps++) {
        // Assign random position, taking into account any temp. gradient.
        ps->x = L * genrand();        // Assign random positions
        // Assign thermal velocity using Gaussian random numbers
        ps->vx = stdDev * genrandn();
        ps->vy = stdDev * genrandn();
        ps->vz = stdDev * genrandn();
        v[X] += ps->vx; v[Y] += ps->vy; v[Z] += ps->vz; // Calc c.m. velocity
    } /*for*/

    // Temperature is measured in the center of mass frame of references,
    // so adjust all velocity components to give a zero net momentum.
    // Then calculate the mean square velocity so that the actual
    // temperature can be determined.
    v2 = 0;                          // Used for adjusting temperature
    v[X] /= pcount; v[Y] /= pcount; v[Z] /= pcount;
    for (i = 0, ps = bound->particles; i < pcount; i++, ps++) {
        ps->vx -= v[X]; ps->vy -= v[Y]; ps->vz -= v[Z]; // Adjust velocity
        v2 += ps->vx*ps->vx + ps->vy*ps->vy + ps->vz*ps->vz; // For temp.
    } /*for*/

    // It is unlikely that the randomly generated velocities will result
    // in a reservoir temperature that matches what is expected. Adjust
    // all components of all particle velocities so that the temperature
    // in the center of mass frame is what was specified.
    v2 = v2/pcount * pb.mass/(3.0 * boltz); // Actual temperature
    adjust = sqrt(T/v2);                  // Factor for adjusting mean square speed
    v[X] = v[Y] = v[Z] = 0;              // For readjusting the c.m. velocity
    for (i = 0, ps = bound->particles; i < pcount; i++, ps++) {
        // Adjust the particle velocity by increasing (decreasing) the
        // magnitude of the mean square speed by the calculated factor.
        ps->vx *= adjust; ps->vy *= adjust; ps->vz *= adjust;
        v[X] += ps->vx; v[Y] += ps->vy; v[Z] += ps->vz; // Calc c.m. velocity
    } /*for*/

    // Finally, adjust the mean velocity so that the center of mass
    // velocity has the specified x-component ("flow") and y-component

```

```

    // ("shear"); the z-component should have a mean of zero.
    v[X] /= pcount; v[Y] /= pcount; v[Z] /= pcount;
    for (i = 0, ps = bound->particles; i < pcount; i++, ps++) {
        ps->vx += -v[X] + bound->v_parallel; // Add in the flow velocity
        ps->vy += -v[Y] + bound->v_shear;    // Add in the shear velocity
        ps->vz -= v[Z];                      // Want zero for mean z-component
    } /*for*/
} /*for*/

// Calculate cell statistics for the initial conditions: number
// of particles, net momentum, and temperature.

ps = sim.ps;
for (i = 0; i < ncell; i++) {
    double sum_vx, sum_vy, sum_vz, sum_v2, vx, vy, vz;
    int sum_n;
    ListIndex ipart;

    if ((sum_n = cells[i].n) <= 0) // Number of particles in cell
        continue;                // No particles in this cell

    ipart = cells[i].head;        // Head of list of particles in cell
    // Initialize running sums
    sum_vx = sum_vy = sum_vz = sum_v2 = 0;
    // For each particle, accumulate running sums for its cell
    while (ipart != ENDLIST) {
        pp = ps + ipart;          // Points to particle state
        sum_vx += (vx = pp->vx);
        sum_vy += (vy = pp->vy);
        sum_vz += (vz = pp->vz);
        sum_v2 += vx*vx + vy*vy + vz*vz;

        ipart = pp->list.next;    // Next particle in this cell
    } /*while*/

    sum_vx /= sum_n; sum_vy /= sum_n; sum_vz /= sum_n; // Take averages
    sim.cellInitStats[i].avg_n = sum_n; // Initial statistics for the cell
    sim.cellInitStats[i].avg_px = sum_vx;
    sim.cellInitStats[i].avg_T =
        sum_v2/sum_n -
        (sum_vx*sum_vx + sum_vy*sum_vy + sum_vz*sum_vz);
} /*for*/

// If statistics for determining correlations between cells are to be
// gathered during the simulation, set up the array of function
// addresses, etc. used to obtain them during the simulation.

if (pb.correlations != 0) {
    for (corr = pb.correlations; corr != 0; corr >>= 1)
        if (corr & 1) sim.samp.ncorr++; // Count the correlation types
    sim.samp.correl = (CorrStats *) calloc(sizeof(CorrStats), sim.samp.ncorr);

    for (corr = pb.correlations, corrSt = sim.samp.correl, i = 0;
        corr != 0;
        corr >>= 1, corrSt++, i++) {
        corrSt->sampFn = CorrFunctions[i].sampFn; // Sampling function
        corrSt->id = i; // Identifies ordinal correlation type
        CorrFunctions[i].initFn(sim, *corrSt); // Corr. specific initialization
    } /*for*/
} /*if*/
} /* InitSim */

/*
* SaveSimStats saves the statistics gathered during a simulation
* into a file. The file name is formed by the simulation number
* concatenated to the simulation control file name, but the "TXT"
* extension of the simulation control file is replaced by "SIM".
* For example, if the simulation control file is "STATS.TXT", the
* output file for the first simulation would be "STATS00.SIM".
* All information is stored as plain text into the file.
*
* If periodic snapshots of each cell's state were taken, these are

```

```

* stored in a separate file in 64-bit IEEE floating point format.
* No attempt is made to standardize the byte ordering, such as the
* little-endian format used by Intel CPUs, so if simulations are
* to be run on one machine architecture and the results displayed
* on another architecture with a different byte ordering scheme,
* this function will need to be modified to standardize the byte
* ordering (probably for little-endian, since PCs are the most
* likely to be used for displaying the data). The file name is
* formed by the simulation control file name as above, but the
* extension is "SMV" (for "simulation movie") rather than "SIM".
*
* Entry:
*   sim -> simulation state descriptor
* Exit:
*   none
*-----*/
void SaveSimStats (SimState& sim)
{
    SimParmBlock  &pb = *sim.pb;
    CorrStats     *corrSt;
    CellStats     *cs, *csInit;
    CellState     *cells;
    char          *blank20 = "                "; // 20 blanks
    char          *p, *s;
    char          **nameList;
    double        *valList;
    double        *buf;
    FILE          *sfile;
    HistogramData *hist;
    double        mass = pb.mass;
    double        nsamp = sim.samp.nsamp;
    int           count, histLen, i, icell, ilist, j, listLen, ncell;
    char          fname[sizeof(pb.filename)+12], sbuf[80];
    // Output file extension
    char          ext[] = { ExtensionSepChar, 's', 'i', 'm', '\0' };
    BoundaryType  &lb = pb.lbound, &rb = pb.rbound;

    double parmVals[] = {
        pb.nrun,          pb.run,          pb.density,    pb.mass,
        pb.diam,          pb.length,      lb.T,         rb.T,
        lb.v_shear,      rb.v_shear,      pb.nstep,     pb.ncell,
        pb.npart,        (int)pb.initCond,
        (int)lb.boundCond, pb.startSampStep,
        pb.showSimProgress, pb.correlations,
        pb.tau,          pb.boltz,        pb.force,
        pb.lbound.density, pb.rbound.density,
        pb.randSeed,     pb.saveSnapshot, pb.boundaryHist,
    };
    char *sysStatNames[] = {
        "tsamp",          "nsamp",          "colltot",
    };
    double sysStats[] = {
        sim.samp.tsamp,    nsamp,
        sim.samp.collTot,
    };
    char *wallStatNames[] = {
        "lwall_dpxTot", "lwall_dpyTot", "lwall_dpyErr", "lwall_strikesum",
        "lwall_sum_py", "lwall_sum_pz", "lwall_type", "lwall_res_len",
        "rwall_dpxTot", "rwall_dpyTot", "rwall_dpyErr", "rwall_strikesum",
        "rwall_sum_py", "rwall_sum_pz", "rwall_type", "rwall_res_len",
    };
    // Prevent division by zero in creating the 'wallStats' array:
    if (sim.samp.lwall.strikeSum == 0) sim.samp.lwall.strikeSum = 1;
    if (sim.samp.rwall.strikeSum == 0) sim.samp.rwall.strikeSum = 1;
    // Now create the 'wallStats' array:
    double wallStats[] = {
        sim.samp.lwall.dpxTot*mass, sim.samp.lwall.dpyTot*mass,
        sim.samp.lwall.dpyErr*mass*mass, sim.samp.lwall.strikeSum,
        sim.samp.lwall.sum_py*mass/sim.samp.lwall.strikeSum,
        sim.samp.lwall.sum_pz*mass/sim.samp.lwall.strikeSum,
    };
}

```

```

        double( int(lb.boundCond) ),
        pb.lbound.length,
        sim.samp.rwall.dpxTot*mass, sim.samp.rwall.dpyTot*mass,
        sim.samp.rwall.dpyErr*mass*mass, sim.samp.rwall.strikeSum,
        sim.samp.rwall.sum_py*mass/sim.samp.rwall.strikeSum,
        sim.samp.rwall.sum_pz*mass/sim.samp.rwall.strikeSum,
        double( int(rb.boundCond) ),
        pb.rbound.length,
    };
    struct { char **names; double *values; int nvals; } dataList[] = {
        { ParmNames,      parmVals,  sizeof(parmVals)/sizeof(*parmVals) },
        { sysStatNames,   sysStats,   sizeof(sysStats)/sizeof(*sysStats) },
        { wallStatNames,  wallStats,  sizeof(wallStats)/sizeof(*wallStats) },
    };
    char *cellStatNames[] = {
        "avg_n",          "avg_px",      "avg_py",      "avg_pz",
        "avg_KE",         "avg_T",
        "avg_sum_px",     "avg_sum_py", "avg_sum_pz", "avg_sum_KE",
        "l_pressure",
        "init_n",         "init_px",   "init_py",     "init_pz", "init_T",
        "vrmax",         "colltot",   "l_particles_into"
    };
    double cellStats [sizeof(cellStatNames)/sizeof(*cellStatNames)];

    // Create or open the output file. The output file name is taken from
    // the input file name, except the run number is incorporated into it to
    // uniquely identify the simulation.

    p = strrchr(pb.filename, PathSepChar) + 1; // End of the path
    s = strchr (p, ExtensionSepChar); // End of first part of file name
    if (s != NULL) p = s; // File name includes an extension
    memcpy (fname, pb.filename, p-pb.filename); // Copy path + part of name
    s = fname + (p-pb.filename);
    if (*p != 0) p = ext; // Output file extension to use
    sprintf (s, "%02d%s", pb.run, p); // Identify the run, add rest of name

    if ((sfile = fopen (fname, "wt")) == NULL) {
        cout << "Unable to open/create file for saving statistics, _\n"
              << fname << "\n\n";
        return;
    }
    /*if*/

    // Output the simulation parameters and all statistics other than the
    // cell statistics.

    for (ilist = 0; ilist < sizeof(dataList)/sizeof(*dataList); ilist++) {
        nameList = dataList[ilist].names;
        valList = dataList[ilist].values;
        listLen = dataList[ilist].nvals;

        for (i = 0; i < listLen; i++) {
            strcpy (sbuf, blank20); // Want to line up values at col. 21
            memcpy (sbuf, nameList[i], strlen(nameList[i]));
            fwrite ( sbuf, strlen(sbuf), 1, sfile );
            fwrite ( sbuf, sprintf(sbuf, "%.16g\n", valList[i]), 1, sfile );
        }
        /*for*/
    }

    if (sim.pb->title[0] != 0)
        fprintf (sfile, "title %s\n", sim.pb->title);
    if (sim.pb->name[0] != 0)
        fprintf (sfile, "name %s\n", sim.pb->name);

    // Output the cell statistics

    ncell = pb.ncell;
    cs = sim.samp.cells; csInit = sim.cellInitStats; cells = sim.cells;
    // Special case for total momentum at left boundary of first cell:
    // this is the same as system's left wall, so get it from there.
    cs->l_tot_px = sim.samp.lwall.dpxTot; // Must get cell 0 value from wall

```

```

for (icell = 0; icell < ncell; icell++, cs++, csInit++) {
    cellStats[0] = cs->avg_n/nsamp;
    // Calculate the mean per-particle momentum:
    cellStats[1] = cs->avg_px * mass/nsamp; // Need to convert from velocity
    cellStats[2] = cs->avg_py * mass/nsamp; // to momentum
    cellStats[3] = cs->avg_pz * mass/nsamp;
    cellStats[4] = cs->avg_KE * 0.5 * mass/nsamp; // SAM kinetic energy
    // Convert an avg-T entry to temperature:
    cellStats[5] = cs->avg_T * mass/(3.0*pb.boltz * nsamp);
    // Calculate the average total cell momentum within a timestep:
    cellStats[6] = cs->sum_px * mass/nsamp;
    cellStats[7] = cs->sum_py * mass/nsamp;
    cellStats[8] = cs->sum_pz * mass/nsamp;
    // Average total kinetic energy in a cell within a timestep:
    cellStats[9] = cs->sum_KE * mass/(2*nsamp);
    // Calculate the average pressure at cell's left boundary, using
    //  $P = (\text{total.momentum})/(\text{total.time} * \text{boundary.area})$ .
    cellStats[10] = cs->l_tot_px*mass/(pb.length*pb.length*sim.samp.tsamp);
    // Do the same for the system initialization averages:
    cellStats[11] = csInit->avg_n;
    cellStats[12] = csInit->avg_px * mass;
    cellStats[13] = csInit->avg_py * mass;
    cellStats[14] = csInit->avg_pz * mass;
    cellStats[15] = csInit->avg_T * mass/(3.0*pb.boltz);
    // Miscellaneous information for each cell:
    cellStats[16] = cells[icell].vrMax; // Maximum relative velocity
    cellStats[17] = cs->collTot;
    cellStats[18] = cs->l_particle_net; // Net particle cnt thru left wall

    for (i = 0; i < sizeof(cellStats)/sizeof(*cellStats); i++) {
        strcpy (sbuf, blank20); // Want to line up values at col. 21
        memcpy (sbuf, cellStatNames[i], strlen(cellStatNames[i]));
        fwrite (sbuf, strlen(sbuf), 1, sfile);
        fwrite (sbuf, sprintf(sbuf, "%.16g\n", cellStats[i]), 1, sfile);
    } /*for*/
} /*for*/

// Output the correlation related statistics
for (i = 0; i < sim.samp.ncorr; i++) {
    corrSt = &sim.samp.correl[i];
    CorrFunctions[corrSt->id].saveFn (sim, *corrSt, sfile);
} /*for*/

fclose (sfile);
cout << "_(Results_written_to_file:_)" << fname << ")\n";

// Create a file containing data about distributions (e.g., distribution
// of velocities of particles entering or leaving the system) if that
// data were collected, intended for use in creating histograms, and write
// them in IEEE double precision floating point format.
if ((count = pb.boundaryHist) > 0) {
    strcpy (fname+strlen(fname)-3, "shg"); // Change the file name extension
    sfile = fopen (fname, "wb");
    // All values for a given histogram are stored consecutively
    histLen = count*sizeof(double) * sizeof(hist[0])/sizeof(hist[0].v_enter);
    buf = (double *) malloc(histLen);
    // For each boundary, store the histogram data into the buffer.
    for (j = 0, hist = sim.samp.lwall.hist;
        j < 2;
        j++, hist = sim.samp.rwall.hist) {
        for (i = 0; i < count; i++) {
            buf[0*count + i] = hist[i].v_enter;
            buf[1*count + i] = hist[i].v_leave;
            buf[2*count + i] = hist[i].n_enter;
            buf[3*count + i] = hist[i].n_leave;
        } /*for*/
    }
}

```

```

        fwrite (buf, histLen, 1, sfile);
    } /*for*/
    free (buf);
    fclose (sfile);
    cout << "_(Results_written_to_boundary_histogram_data_file:_)_"
        << fname << ")\n";
} /*if*/

// If periodic snapshots of each cell's state were taken, write them to
// a separate file in binary (IEEE floating point format, double
// precision). To give flexibility to implementation of the program that
// will display the snapshots, for each snapshot, write out the information
// for all cells, then go on to the next snapshot.

if (pb.nsnapshots > 0) {
    // Allocate a buffer for one snapshot.
    SnapshotOut *buf = (SnapshotOut *) malloc(ncell*sizeof(SnapshotOut));

    strcpy (fname+strlen(fname)-3, "smv"); // Change the file name extension
    sfile = fopen (fname, "wb");
    for (i = 0; i < pb.nsnapshots; i++) {
        for (icell = 0; icell < ncell; icell++) {
            buf[icell].n = (double) cells[icell].snapshot[i].n;
            buf[icell].T = cells[icell].snapshot[i].T;
        } /*for*/
        fwrite (buf, sizeof(SnapshotOut), ncell, sfile);
    } /*for*/
    free(buf);
    fclose (sfile);
    cout << "_(Results_written_to_snapshot_file:_)_" << fname << ")\n";
} /*if*/
} /* SaveSimStats */

/*-----
 * main is the top level function.
 * Entry:
 *   argC -> number of command line arguments
 *   argS -> array of command line argument strings
 * Exit:
 *   zero
 *-----*/
int main (int argC, char *argS[])
{
    SimParmBlock *pb;
    SimState      sim;
    int iRun;

    cout << "SIM_-_Direct_simulation_of_a_rarified_gas_using_the_DSMC_method\n";

    pb = GetSimParms (argC, argS); // Get them from file, or use defaults
    if (pb == NULL)
        return 1; // A fatal error occurred

    // Loop until all simulation runs given in the parm file have completed.

    for (iRun = 0; iRun < pb->nrun; iRun++) {
        InitSim (pb[iRun], sim); // Initialize for the simulation
        RunSim (sim); // Run the simulation
        if (sim.samp.nsamp > 0)
            SaveSimStats (sim); // Save accumulated statistics to file
        CleanupSim (sim); // Deallocate storage, etc.
    } /*for*/

    free (pb);
    return 0;
} /*main*/

```

C.3 DSMC Simulation Functions

```
/*-----
```



```

* SIMRUN.CPP - Functions to run a direct simulation (DSMC) of a
*              rarified gas in a microscopic system. The system
*              boundaries are periodic in two of the dimensions, and
*              two plates bound the system in the third
*              dimension. The plates may be fixed or moving relative
*              to each other, and the may be at different temperatures.
*              The system is divided into cells along this third
*              dimension. A simulation determines correlations of
*              fluctuations among the different cells at steady state,
*              and determines other properties such as pressure on the
*              walls and viscosity.
*
* Written by Marty Tysanner, based on code written by A. Garcia [see
* A. Garcia, Numerical Methods for Physics, 2nd Edition (2000)].
*/

#include <stdlib.h>
#include <stdio.h>
#include <alloc.h>
#include <string.h>
#include <ctype.h>
#include <memory.h>
#include <math.h>
#include <float.h>
#include <fcntl.h>
#include <fstream.h>
#include <time.h>
#include "fluct.h"

/*
* Collider processes collisions in cells and updates the simulation
* state accordingly. Conservation of energy and all three components
* of momentum are verified.
* Entry:
*   sim -> simulation state descriptor
* Exit:
*   error flags
*/

unsigned Collider (SimState& sim)
{
    SimParmBlock *pb;
    ParticleState *p1, *p2, *ps;
    CellState *cells;
    double coeff, npair, select, vr, vrMax;
    double vcm_x, vcm_y, vcm_z, vrel_x, vrel_y, vrel_z;
    double cos_th, phi, sin_th;
    double energy, px, py, pz;
    int isel, jcell, k1, k2, ncand, ncell, number;

    ps = sim.ps; // Array of particle states
    pb = sim.pb; // Simulation parameters
    cells = sim.cells; // Array of cell information structures
    coeff = pb->coeff; // For computing number of selected pairs
    energy = px = py = pz = 0.0;
    ncell = pb->ncell;

    // Loop over cells, processing collisions in each cell
    for (jcell = 0; jcell < ncell; jcell++) {
        number = cells[jcell].n; // Number of particles in the cell
        if (number <= 1)
            continue; // Skip cells with only one particle

        // Determine number of candidate collision pairs to be selected
        // in this cell.
        vrMax = cells[jcell].vrMax; // Current maximum relative speed
        select = coeff*number*(number-1)*vrMax
            + cells[jcell].selExtra; // (Include left-over fraction)
        npair = floor(select); // Number of pairs to be selected now
        cells[jcell].selExtra = select-npair; // Carry over any left-over fraction

        // Loop over total number of candidate collision pairs
        ncand = (int) npair; // Integer value, for convenience
    }
}

```

```

for (isel = 0; isel < ncand; isel++) {
    // Pick two particles at random out of this cell
    k1 = (int) (genrand() * number);
    k2 = (k1 + (int)(genrand()*(number-1)) + 1) % number;
    p1 = ps + sim.partXref[ k1+cells[jcell].index ]; // First particle
    p2 = ps + sim.partXref[ k2+cells[jcell].index ]; // Second particle

    // Calculate pair's relative speed
    vr = NORM (p1->vx - p2->vx, p1->vy - p2->vy, p1->vz - p2->vz);
    if (vr > vrMax) // If this relative speed is largest
        vrMax = vr; // so far, then record it

    // Accept or reject the candidate pair according to relative speed
    if (vr/cells[jcell].vrMax > genrand()) {
        // If pair is accepted, select post-collision velocities.
        // Take collision angle as a random value, with all angles
        // having equal probability.
        sim.samp.collTot++; // Update collision counter
        sim.samp.cells[jcell].collTot++;
        vcm_x = 0.5 * (p1->vx + p2->vx); // Center of mass velocity
        vcm_y = 0.5 * (p1->vy + p2->vy);
        vcm_z = 0.5 * (p1->vz + p2->vz);
        cos_th = 1.0 - 2.0*genrand(); // Cosine and sine of random
        sin_th = sqrt(1.0 - cos_th*cos_th); // collision angle theta
        phi = 2 * PI * genrand(); // Random collision angle phi
        vrel_x = vr*cos_th; // Compute post-collision
        vrel_y = vr*sin_th*cos(phi); // relative velocity
        vrel_z = vr*sin_th*sin(phi);

        // Update the particle velocities and update the variables used
        // to check for energy and momentum conservation.
        px -= p1->vx + p2->vx; py -= p1->vy + p2->vy;
        pz -= p1->vz + p2->vz;
        energy -= p1->vx*p1->vx + p1->vy*p1->vy + p1->vz*p1->vz;
        energy -= p2->vx*p2->vx + p2->vy*p2->vy + p2->vz*p2->vz;
        p1->vx = vcm_x + 0.5*vrel_x; // Update post-collision
        p2->vx = vcm_x - 0.5*vrel_x; // velocities
        p1->vy = vcm_y + 0.5*vrel_y;
        p2->vy = vcm_y - 0.5*vrel_y;
        p1->vz = vcm_z + 0.5*vrel_z;
        p2->vz = vcm_z - 0.5*vrel_z;
        px += p1->vx + p2->vx; py += p1->vy + p2->vy;
        pz += p1->vz + p2->vz;
        energy += p1->vx*p1->vx + p1->vy*p1->vy + p1->vz*p1->vz;
        energy += p2->vx*p2->vx + p2->vy*p2->vy + p2->vz*p2->vz;
    } /*if*/
} /*for loop over pairs*/

cells[jcell].vrMax = vrMax; // Update max relative speed in cell
} /*for loop over cells*/

// In checking for conservation violations, allow for an error that is
// proportional to the number of particles and the number of conservation
// related calculations per particle.
sim.samp.energy += energy; // Update them (adjust for round-off)
sim.samp.px += px; sim.samp.py += py; sim.samp.pz += pz;
coeff = DBLEPSILON * sim.npart * 4.0;
return ( (fabs(energy) > coeff) |
         ((fabs(px) > coeff) << 1) |
         ((fabs(py) > coeff) << 2) |
         ((fabs(pz) > coeff) << 3) );
} /* Collider */

/*
* timeToMoveDist is a helper function for "Mover". It solves the
* quadratic equation for the time a particle takes to reach the wall,
* used for those particles that strike the wall during a timestep.
* The quadratic equation being solved is
* 0.5*a*t^2 + v*t - (x.wall - x) == 0,
* which can be solved using the quadratic equation:

```

```

*      t = 1/(2*0.5*a)* [-v +/- sqrt(v^2 + 2*a*dist)
*      = -v/a +/- sqrt((v/a)^2 + 2*dist/a),
*      where dist = x_wall-x.
*      We want to choose the smallest positive root (a negative time is
*      obviously meaningless, but we don't want to choose a time that is
*      greater than the timestep tau either).
*/
static double timeToMoveDist (double dist, double v, double a)
{
    double part_1, part_2, root_1, root_2, v_over_a;
    part_1 = -(v_over_a = v/a);
    part_2 = sqrt( v_over_a*v_over_a + 2.0*dist/a );
    root_1 = part_1 + part_2;
    if ((root_2 = part_1 - part_2) < 0.0)
        return root_1; // Need positive root
    else
        // Choose the smallest positive root.
        return (root_1 < 0.0 || root_2 < root_1) ? root_2 : root_1;
} /* timeToWall */

/*
* Mover handles movement of all particles by free flight. If
* movement causes a particle to strike a wall, the position and/or
* velocity of the particle is modified according to the boundary
* condition at the wall. Statistics for events at walls are also
* recorded. It is assumed that boundaries are periodic in the
* y- and z-directions. For simulations involving specular
* boundary conditions, the system is bounded by walls in the
* x-direction with simulation-dependent wall behavior.
*
* If a particle moves to a different cell, it is moved into the
* particle list for the new cell so that sorting into cells is
* maintained. The cross-reference of particles in each cell is
* not updated, however; that is done by the "Sorter" function,
* which must be separately called after completion of particle
* movement.
*
* Although conservation of energy and the three components of
* momentum are not checked, the appropriate variables in the
* simulation state are updated so that conservation can be verified
* elsewhere without being confused by this function.
*
* Entry:
*   sim -> simulation state descriptor
* Exit:
*   error flags (always zero)
*/
unsigned Mover (SimState& sim)
{
    SimParmBlock *pb;
    ParticleState *pp, *ps;
    CellState *cells;
    WallStats *ws;
    BoundaryType *bound;
    double a, dist, t;
    double direction, L, stdev, tau;
    double dvx, vx_old, vy_old, vz_old, x, x_old;
    double v_accel, x_accel;
    double scale_n, scale_v;
    long hist_len, n_enter_l, n_enter_r, n_leave_l, n_leave_r;
    int bin;
    int ncell, new_cell, old_cell;
    ListIndex ipart, npart, next, prev;
    BoundaryCondType bc;

    // Extract some commonly needed values to increase performance.
    ps = sim.ps;
    pb = sim.pb;
    cells = sim.cells;
    ncell = pb->ncell; // Number of cells in system
    npart = (ListIndex) sim.maxPart; // Number of particles to look at

```

```

tau    = pb->tau;                // Timestep
L      = pb->length;             // System length

// Histogram-related variables
n_enter_l = n_enter_r = n_leave_l = n_leave_r = 0; // Particle counters
hist_len = pb->boundaryHist;    // Histogram length; may be zero
if (pb->lbound.mpv < pb->rbound.mpv) {
    scale_v = hist_len/(pb->rbound.mpv + 6.0*pb->rbound.mpv_stdev);
    // Expected density of particles in the cell at the right boundary:
    scale_n = pb->npart/L * (pb->rbound.mpv/pb->lbound.mpv)/2.0;
    // Expected number of particles in the cell that will leave the
    // system during a timestep; this will generally equal the average
    // number that will also enter the system during the same timestep.
    // This roughly equals the expected distance covered by a particle
    // (the distance covered in a timestep by a particle moving at the
    // most probable velocity) multiplied by the expected particle
    // density. Allow a range of up to about five standard deviations.
    // We divide the most probable velocity by 2*sqrt(3) because we can
    // assume the x, y and z components of velocity are all equal, so the
    // x-component is 1/sqrt(3) of the magnitude, and the factor of 1/2
    // comes because we are only interested in one of the two directions.
    scale_n *= (pb->rbound.mpv + 5.0*pb->rbound.mpv_stdev)/
                (2.0*sqrt(3.0)) * pb->tau;
}
else {
    scale_v = hist_len/(pb->lbound.mpv + 6.0*pb->lbound.mpv_stdev);
    scale_n = pb->npart/L * (pb->lbound.mpv/pb->rbound.mpv)/2.0;
    scale_n *= (pb->lbound.mpv + 5.0*pb->lbound.mpv_stdev)/
                (2.0*sqrt(3.0)) * pb->tau;
}
// *if*/
// Now calculate the scaling factor from the expected number of particles
// that will leave (or enter) the the system through a boundary during
// a timestep.
scale_n = 1.0;

// Calculate contribution of x-direction motion due to constant external
// force:  $x\_accel = (1/2)*a*t^2 = 0.5*(F_x/m)*tau^2$ .
a = pb->force/pb->mass;          // Acceleration due to external force
x_accel = 0.5*a*tau*tau;        // Zero if pb->force == 0
v_accel = a * tau;              //  $v = a * t$ 

// Move all particles, pretending walls are absent
for (ipart=0, pp=ps; ipart < npart; ipart++, pp++) {
    if (pp->list.prev == FREELIST) continue; // Particle isn't in system
    // Save the particle's velocity components so the system energy
    // and momentum totals can be adjusted for the particle at the end
    // of the loop after the new velocity has been computed.
    vx_old = pp->vx;             // Needed for pressure measurement
    vy_old = pp->vy;             // Needed for drag force measurement
    vz_old = pp->vz;             // Needed to update conservation vars

    // Move particle by free flight under the possible influence of
    // an external, constant force:  $x = x + v_x*t + 0.5*a*t^2$ .
    x = (x_old = pp->x) + pp->vx*tau + x_accel; // Save old position too
    old_cell = (int) (ncell*x_old/L); // Determine the particle's old cell
    new_cell = (int) (ncell*x/L); // Do same for new cell

    if (x >= 0.0 && x < L) {
        pp->x = x; // New particle position
        pp->vx += v_accel; // Adjust the velocity due to any force
    }
    else {
        // The particle is no longer within the system, meaning it has
        // struck a wall. Handle the appropriate boundary condition.
        bc = (x < 0.0) ? pb->lbound.boundCond : pb->rbound.boundCond;

        switch (bc) {
            case bc_periodic: // ** Periodic boundary (no wall) **

```

```

pp->x = fmod(x+L, L); // Periodic boundary conditions
new_cell = (int) (ncell*pp->x/L); // Determine new cell
pp->vx += v_accel; // Adjust the velocity due to any force
if (x < 0.0) {
    n_leave_l++; n_enter_r++;
    ws = &sim.samp.lwall;
}
else {
    n_leave_r++; n_enter_l++;
    ws = &sim.samp.rwall;
} /*if*/
break;

case bc_specular_wall: // ** Specular boundary at wall **
if (x < 0.0) {
    n_leave_l++; n_enter_l++;
    ws = &sim.samp.lwall;
    bound = &pb->lbound;
}
else {
    n_leave_r++; n_enter_r++;
    ws = &sim.samp.rwall;
    bound = &pb->rbound;
} /*if*/

if (a != 0.0) {
    double vx_tmp;
    dist = bound->position - x_old; // Distance to the wall
    t = timeToMoveDist (dist, vx_old, a); // Find time to wall
    vx_tmp = -(vx_old+a*t); // Velocity after hitting the wall
    ws->dpxTot += pp->vx + vx_tmp; // Record for pressure
    t = tau - t; // Remainder of timestep
    pp->vx = vx_tmp + a*t;
    x = bound->position + vx_tmp*t + 0.5*a*t*t; // Distance to wall
}
else {
    pp->vx = -vx_old; // No force; just reverse x-velocity
    ws->dpxTot += pp->vx - vx_old; // Record for pressure calc.
    x = bound->position + (bound->position - x);
} /*if*/
ws->strikeSum++; // Record number of hits on wall
// See NOTE #1 and NOTE #2 in the "thermal wall" case for
// an explanation of the following 'if' statement:
if (x < 0.0)
    x = DBL_EPSILON; // DBL_EPSILON is defined in <float.h>
else if (x > L)
    x = L * (1.0-DBL_EPSILON); // Keep x within system bounds
pp->x = x;
new_cell = (int) (ncell*x/L); // Determine new cell
break;

case bc_thermal_wall: // ** Thermal wall boundary **
if (x < 0.0) {
#ifdef INTERNAL_CELL_HIST
    n_leave_l++; n_enter_l++;
#endif
    ws = &sim.samp.lwall;
    bound = &pb->lbound;
    direction = 1;
}
else {
#ifdef INTERNAL_CELL_HIST
    n_leave_r++; n_enter_r++;
#endif
    ws = &sim.samp.rwall;
    bound = &pb->rbound;
    direction = -1;
} /*if*/

// Reset velocity components as biased Maxwellian,

```

```

// exponential distribution in x; Gaussian in y and z
pp->vx = direction*sqrt(-log(1-genrand())) * bound->mpv;
// Include shear velocity of wall or reservoir particles
stdev = bound->mpv_stdev;
pp->vy = stdev*genrandn() + bound->v_shear;
pp->vz = stdev*genrandn();
ws->sum_py += pp->vy;
ws->sum_pz += pp->vz;
// Reset the position after leaving the wall, assuming
// the new x-velocity for the remainder of the timestep
// (that part of the timestep not used in reaching the
// wall). Also record velocity changes for making
// force measurements.
if (a != 0.0) {
    // Find the time needed to reach the wall so we know how
    // much of the timestep remains after particle hits wall.
    dist = bound->position - x_old; // Distance to the wall
    t = timeToMoveDist (dist, vx_old, a);
    ws->dpxTot += pp->vx - (vx_old+a*t); // Record for pressure
    t = tau - t; // Remainder of timestep after hit wall
    x = bound->position + pp->vx*t + 0.5*a*t*t;
    pp->vx += a * t;
}
else {
    x = bound->position +
        pp->vx*tau*(x-bound->position)/(x-x_old);
    ws->dpxTot += pp->vx - vx_old; // Record for pressure calc.
} /*if*/

// NOTE #1: It can normally be assumed that there will never
// be a time when the velocity on leaving a wall is so great
// that a particle can travel the entire length of the system
// in the remainder of the timestep. For a 64 bit double
// precision value, round-off error in calculating the
// x-velocity on leaving the wall,
// sqrt(-log(1-genrand())),
// should make this assumption valid, since 1-genrand() will
// at a minimum be about 10e-16, and thus
// sqrt(-log(1-genrand())) will be at most about sqrt(37).
// Just to be on the safe side on a machine where this
// assumption about the size of a double precision number is
// greater than 64 bits, or for cases where the system length
// is sufficiently small, do a more robust calculation that
// does not make this assumption.
//
// NOTE #2: In the presence of an external force (e.g.,
// gravity), it is possible that the x-velocity leaving the
// wall will be so low that the acceleration due to the force
// will cause the particle to reverse direction and strike
// the wall again within the same timestep. A really robust
// method of handling this would be to set up a loop where
// the x-velocity is changed each time the particle strikes
// the wall, and allow this to happen an arbitrary number of
// times. However, this adds complexity, and it is only
// important in a small percentage of cases.
//
// Taking the two situations into account, the method of
// handling a particle going out of the system boundaries
// after striking a wall is to simply move the particle to
// within an "infinitesimal" distance of the wall (to avoid
// round-off problems under all conditions, use DBLEPSILON
// as this small distance), but leave the velocity alone.
// The effect on the overall simulation should be
// insignificant because of the low frequency of such events.
if (x < 0.0)
    x = DBLEPSILON; // DBLEPSILON is defined in <float.h>

```

```

else if (x > L)
    x = L * (1.0-DBLEPSILON); // Keep x within system bounds
pp->x = x;
new_cell = (int) (ncell*x/L); // Determine new cell

dvy = pp->vy - vy_old; // Change in y-velocity
ws->dpyTot += dvy; // Record for drag force calculation
ws->dpyErr += dvy*dvy; // Record number of hits on wall
ws->strikeSum++;
break;

case bc_reservoir: // ** Particle reservoir (no wall) **
    if (x < 0.0) {
        n_leave_l++;
        ws = &sim.samp.lwall;
    }
    else {
        n_leave_r++;
        ws = &sim.samp.rwall;
    }
    // if */
    // For reservoirs, we simply take the particle out of the
    // system when it reaches the boundary.
    next = pp->list.next;
    if ((prev = pp->list.prev) == END_LIST)
        cells[old_cell].head = next; // Remove from head of list
    else
        ps[prev].list.next = next; // Remove reference by previous entry
    if (next != END_LIST)
        ps[next].list.prev = prev; // Remove reference by next entry
    cells[old_cell].n--; // One fewer particle in the cell
    sim.npart--; // One fewer particle in the system

    // Place the particle into the free list. We handle the
    // case where all particle structures have been removed from
    // the free list, although this probably should never happen.
    if (sim.free == NULL)
        pp->list.next = END_LIST; // Start of the free list
    else
        pp->list.next = (ListIndex) (sim.free - ps);
    sim.free = pp;
    pp->list.prev = FREE_LIST; // Mark the structure as unused

    // Update the optional histogram for particle velocity
    // distributions; only handle the histogram for particles
    // leaving the system, since the one for particles entering
    // the system are handled within 'ParticlesFromReservoir'.
    if (hist_len > 0) {
        if ((bin = abs((int) (vx_old * scale_v))) >= hist_len)
            bin = hist_len-1; // Too big — lump at tail end
        ws->hist[bin].v.leave++; // Increment count for this velocity
    }
    // if */

    // Finally, update the variables used to check for energy
    // and momentum conservation.
    sim.samp.energy -= vx_old*vx_old + vy_old*vy_old + vz_old*vz_old;
    sim.samp.px -= vx_old;
    sim.samp.py -= vy_old;
    sim.samp.pz -= vz_old;
    continue; // This particle is gone; move next one
} // switch */

// Keep track of the velocity distribution for particles entering
// and leaving the system so histograms can be created.
// @@@ NOTE: The histogram for particles entering the system is not
// correctly updated when an acceleration is present ('a' != 0).
if (hist_len > 0) {
    if ((bin = abs((int) (vx_old * scale_v))) >= hist_len)
        bin = hist_len-1; // Too big — lump at tail end
    ws->hist[bin].v.leave++; // Increment count for this velocity
}

```

```

        if ((bin = abs((int) (pp->vx * scale_v))) >= hist_len)
            bin = hist_len - 1; // Too big — lump at tail end
        ws->hist[bin].v_enter++; // Increment count for this velocity
    } /*if*/
} /*if*/

// If the old and new cells are different, the list of particles
// for those cells must be updated to keep the particles properly
// sorted into cells.

if (old_cell != new_cell) {
#ifdef INTERNAL_CELL_HIST
    if (old_cell == 16 && new_cell == 15)
        { n_leave_l++; n_leave_r++; }
    else if (new_cell == 16) {
        if (old_cell == 15) n_enter_l++;
        else if (old_cell == 17) n_enter_r++;
    } /*if*/
#endif

    // First remove the particle from the old cell's particle list.
    next = pp->list.next;
    if ((prev = pp->list.prev) == END_LIST)
        cells[old_cell].head = next; // Remove from head of list
    else
        ps[prev].list.next = next; // Remove reference by previous entry
    if (next != END_LIST)
        ps[next].list.prev = prev; // Remove reference by next entry
    cells[old_cell].n--; // Update the particle count
    // Update the information needed to calculate the average pressure
    // at the left boundary of a cell. For particles having a
    // positive x-velocity, this is the same as momentum entering the
    // new cell at the left boundary. For particles having a negative
    // x-velocity, this is the same as particles "striking" the left
    // boundary of the old cell, giving an outward pressure there.
    if (pp->vx > 0.0) {
        sim.samp.cells[new_cell].l_tot_px += pp->vx; // Momentum into cell
        sim.samp.cells[new_cell].l_particle_net++; // Particle enters
    }
    else {
        sim.samp.cells[old_cell].l_tot_px -= pp->vx; // "Strikes" left wall
        sim.samp.cells[old_cell].l_particle_net--; // Particle leaves
    }
} /*if*/

#ifdef CONSISTENCY_CHECK
    // This is a good time to make some consistency checks on the
    // simulation code itself. It is intended for debugging purposes,
    // since it is not efficient and will greatly slow down the
    // simulation.

    for (int i = 0; i < ncell; i++) {
        int icell, n, nxt;
        for (n=0, nxt=cells[i].head; nxt != END_LIST;
             nxt=ps[nxt].list.next, n++) {
            if (nxt == ipart)
                cout << "Particle_" << ipart << "_was_deleted_from_cell_"
                     << old_cell << ";_found_in_cell_" << i << "\n";
            icell = (int) (ncell*ps[nxt].x/L); // Determine new cell
            icell -= (icell >= ncell); // Handle "overflow" case
            if (i != icell)
                cout << "Particle_" << ipart << "_is_in_cell_" << i
                     << ";_should_be_in_cell_" << icell << "\n";
        } /*for*/
        if (n != cells[i].n) {
            cout << "Particle_" << ipart << "::_counts_don't_match::_"
                 << n << "_in_list_" << i
                 << ";_" << cells[i].n << "_in_cell_count\n"
                 << "::::0:";
            for (n=0, nxt=cells[i].head; nxt != END_LIST;

```



```

        nxt=ps[nxt].list.next, n++) {
        cout << " " << nxt;
        if ((n+1) % 12 == 0)
            cout << "\n" << (n+1) << ": ";
        } /*for*/
        cout << "\n-1";
        cout << "\n";
    } /*if*/
} /*for*/

if (new_cell > ncell || new_cell < 0) {
    cout << "***Cell number is out of range: cell = " << new_cell
        << "\n";
    pp = (ParticleState *) NULL;
    pp->x = 0.0; // Force a fatal error
} /*if*/
#endif

// Now add the particle to the new cell's particle list. Because
// we want to make this operation fast, insert it at the front.

if ((next = cells[new_cell].head) != END_LIST)
    ps[next].list.prev = ipart;
cells[new_cell].head = ipart;
pp->list.next = next;
pp->list.prev = END_LIST;
cells[new_cell].n++; // Update the particle count
} /*if*/

// Finally, update the variables used to check for energy and momentum
// conservation.
sim.samp.energy += (pp->vx*pp->vx + pp->vy*pp->vy + pp->vz*pp->vz) -
    (vx_old*vx_old + vy_old*vy_old + vz_old*vz_old);
sim.samp.px += pp->vx - vx_old;
sim.samp.py += pp->vy - vy_old;
sim.samp.pz += pp->vz - vz_old;
} /*for*/

// If we have reservoirs at the boundaries, move the particles into
// the system now. (We waited until the end so the particles would
// not be moved in the above movement activity after they entered
// the system).
//n_leave_r = cells[16].n; // Uncomment for histogram: # of particles in cell 17

if (pb->lbound.boundCond == bc_reservoir)
    n_enter_l = ParticlesFromReservoir (sim, pb->lbound);
if (pb->rbound.boundCond == bc_reservoir)
    n_enter_r = ParticlesFromReservoir (sim, pb->rbound);

if (hist_len > 0) {
    // Update the histogram for the number of particles entering and
    // leaving the system through each boundary.
    if ((bin = (double)n_enter_l * scale_n) >= hist_len)
        bin = hist_len - 1;
    sim.samp.lwall.hist[bin].n_enter++;
    if ((bin = (double)n_leave_l * scale_n) >= hist_len)
        bin = hist_len - 1;
    sim.samp.lwall.hist[bin].n_leave++;
    if ((bin = (double)n_enter_r * scale_n) >= hist_len)
        bin = hist_len - 1;
    sim.samp.rwall.hist[bin].n_enter++;
    if ((bin = (double)n_leave_r * scale_n) >= hist_len)
        bin = hist_len - 1;
    sim.samp.rwall.hist[bin].n_leave++;
} /*if*/

return 0;
} /* Mover */

/*
 * ParticlesFromReservoir handles movement of particles into the

```

* system from reservoirs at the boundaries. A reservoir must act
 * like it is infinitely long, so we must make sure it is long
 * enough that a particle at one end will never make it to the other
 * end during a timestep, for any velocity it will ever have (of
 * course, the velocities are Gaussian distributed, so "never" is
 * really "very high probability of never happening during a
 * simulation." This must be balanced against the undesirability of
 * making the reservoir any longer than necessary, since for a given
 * particle density in the reservoir, the longer the reservoir is the
 * greater the number of particles that must be generated each
 * timestep (see below). The implementation makes this compromise:
 * The reservoir length is the distance a particle will be most
 * likely to travel during one timestep, plus six standard deviations
 * of this most probable distance; statistically, that should be
 * enough.

* The reservoir is virtual in that particles are generated anew each
 * time this function is called, with random position in the reservoir
 * and with a Maxwell-Boltzmann velocity distribution; there is no
 * memory between timesteps of particles in a reservoir, keeping
 * unwanted correlations in velocity or position from creeping in and
 * corrupting the reservoir.

* We expect that half the particles will move in each direction, so
 * to save time we only move half the number of needed particles, but
 * all of them are in the same direction. To avoid a systematic
 * undercount or overcount from creeping in when we divide by two,
 * we adjust the count in an alternating manner; the first time
 * the function is called for this boundary, we don't add one before
 * dividing by two, and the next time the function is called for
 * this same boundary we add one before dividing by two.

* Entry:
 * sim -> simulation state descriptor
 * bound-> boundary information
 * Exit:
 * number of particles that entered the system

```
static long ParticlesFromReservoir (SimState& sim, BoundaryType& bound)
{
    ParticleState *pl, *p2, *rp;
    double vcm[3], vrel[3];
    double cos_th, sin_th, phi, select, vr, vrMax;
    int j, ncand, npart, pcount;
    ParticleState *pp, *ps;
    CellState *cells;
    WallStats *wstats;
    SimParmBlock &pb = *sim.pb;
    double L, stdev, tau, v[3], x, xmax;
    double number, pressure;
    double a, direction, dist, t, v_accel, vx_old, x_accel, x_old;
    long n_enter;
    int bin, hist_len;
    int count, i, icell, ncell;
    ListIndex head, ipart;
    char isLeftBound;

    cells = sim.cells; // Array of info for all cells
    tau = pb.tau;
    ncell = pb.ncell;
    xmax = pb.rbound.position - pb.lbound.position; // System length
    stdev = bound.mpv_stdev; // Standard deviation in velocity
    L = bound.length; // Reservoir length
    isLeftBound = (bound.position == pb.lbound.position);
    direction = isLeftBound ? 1.0 : -1.0;
    wstats = isLeftBound ? &sim.samp.lwall : &sim.samp.rwall;

    rp = bound.particles; // First reservoir particle (if 'closed')

    // Calculate contribution of x-direction motion due to constant external
    // force: x_accel = (1/2)*a*t^2 = 0.5*(F_x/m)*tau^2.
    a = pb.force/pb.mass; // Acceleration due to external force
    // The "right" side of the reservoir is always considered to be next to
```

```

// the system, so reverse the acceleration if this reservoir is on the
// right side of the system.
if (!isLeftBound)
    a = -a;
x_accel = 0.5*a*tau*tau;           // Zero if pb->force == 0
v_accel = a * tau;                 // v = a * t

// Variables used to maintain velocity distribution histogram:
hist_len = pb.boundaryHist;       // Histogram length; may be zero
n_enter = 0;

if (bound.npart == 0) {
    // The mean number of particles to be moved at each time step has not
    // yet been determined.
    if (bound.resType == rt_generate_chap_enskog) {
        pressure = 1.0;           // *** THIS MUST BE CALCULATED ***
        ChapmanEskogDistInit (sim, pressure, bound.T, bound.chapEnsk);
        bound.chapEnsk.x_positive = 1; // Negative velocities not needed
    }
    else if (bound.resType == rt_thermal_wall)
        number =
            bound.density * double(wstats->strikeSum)/double(sim.samp.nsamp);
    else if (pb.lbound.T == pb.rbound.T)
        // Calculate the number of particles that would be expected to be
        // in the reservoir if it had the same density as the rest of the
        // system, then multiply by the reservoir density (density of
        // 1.0 means the system's density should be used). Finally, divide
        // this number by two because generated particles will always have
        // a velocity toward the system rather than having velocities in
        // either direction as would occur in a physical system.
        number = bound.length/(pb.rbound.position-pb.lbound.position)
            * pb.npart * bound.density * 0.5;
    else {
        // Calculate the mean number by multiplying the mean fraction of
        // generated particles that actually enter the system (determined
        // empirically here) by the mean number of particles leaving the
        // system through the same boundary. This will keep the same the
        // average number of particles entering and leaving the reservoir,
        // provided the desired density is unity. Divide the result by
        // two for the same reason as in the "equal temperatures" case above.
        number =
            bound.density*double(wstats->strikeSum)/(2*double(sim.samp.nsamp));
        // Determine the mean fraction of generated particles that can be
        // expected to enter the system at each time step.
        for (i = count = 0; i < 10000; i++) {
            x = L * genrand();           // Assign random position in reservoir
            v[X] = stdev*fabs(genrandn()); // Gaussian dist. thermal velocity
            if ((x += (v[X] + bound.v_parallel)*tau + x_accel) > L)
                count++;                 // Particle entered the system
        } //for*
        number /= double(count)/10000; // Divide by fraction to get count
    } //if*
    bound.npart = number;               // Mean particle count to generate
} //if*

switch (bound.resType) {
case rt_generate_maxwell_boltz:
case rt_generate_chap_enskog:
    // Determine the number of particles to "move" within the reservoir.
    // This calculation simply adds a random deviate (call to 'genrand')
    // to the previously determined mean number of particles to move;
    // this compensates by adding an extra particle with the right
    // frequency to accommodate a non-integer mean.
    // The particle count is Poisson distributed about the mean:
    count = genrandp (long (bound.npart + genrand()));
    break;
case rt_generate_fixed:
    // Generate a constant number of particles at each time step. The

```

```

// actual number of particles entering the system will depend on
// the number of the generated particles that have an x-velocity
// and position that permit reaching the system.
count = int (floor(bound.npart + genrand()));
break;

case rt_closed_system:
// The most important task now is to handle collisions between
// particles in the reservoir. Determine number of candidate
// collision pairs to be selected in this cell.
npart = bound.npart; // Number of particles in reservoir
pcount= double(npart);
vrMax = bound.vrMax; // Largest relative speed thus far
select= pb.coeff * pcount*(pcount-1) * vrMax;
ncand = int(floor(select));
// Include the any left-over fraction by adding a uniform deviate
// to it, then incrementing the number of candidate pairs if the
// result is unity or greater:
ncand += int( floor(select - double(ncand) + genrand()) );

// Loop over total number of candidate collision pairs
for (i = 0; i < ncand; i++) {
// Pick two particles at random out of this cell, making sure the
// two particles are not the same. (The algorithm does not
// prevent the same particle from being a collision partner for
// more than one pair, however.)
j = int(genrand() * pcount); // Index of first particle
p1 = rp + j;
p2 = rp + (j + int(genrand()*(pcount-1)) + 1) % npart;

// Calculate pair's relative speed
vr = NORM (p1->vx - p2->vx, p1->vy - p2->vy, p1->vz - p2->vz);
if (vr > vrMax) // If this relative speed is largest
vrMax = vr; // so far, then record it

// Accept or reject the candidate pair according to relative speed
if (vr/bound.vrMax > genrand()) {
// If pair is accepted, select post-collision velocities.
// Take collision angle as a random value, with all angles
// having equal probability.
vcm[X] = 0.5 * (p1->vx + p2->vx); // Center of mass velocity
vcm[Y] = 0.5 * (p1->vy + p2->vy);
vcm[Z] = 0.5 * (p1->vz + p2->vz);
cos_th = 1.0 - 2.0*genrand(); // Cosine and sine of random
sin_th = sqrt(1.0 - cos_th*cos_th); // collision angle theta
phi = 2 * PI * genrand(); // Random collision angle phi
vrel[X] = vr*cos_th; // Compute post-collision
vrel[Y] = vr*sin_th*cos(phi); // relative velocity
vrel[Z] = vr*sin_th*sin(phi);

// Update the particle velocities.
p1->vx = vcm[X] + 0.5*vrel[X]; // Update post-collision
p2->vx = vcm[X] - 0.5*vrel[X]; // velocities
p1->vy = vcm[Y] + 0.5*vrel[Y];
p2->vy = vcm[Y] - 0.5*vrel[Y];
p1->vz = vcm[Z] + 0.5*vrel[Z];
p2->vz = vcm[Z] - 0.5*vrel[Z];
} //*/if*/
} //*/for loop over pairs*/

bound.vrMax = vrMax; // Update max relative speed in cell
count = npart; // Move all particles in reservoir system
#if (CONSISTENCY_CHECK)
// Calculate the temperature in the system and verify it is OK.
double v2;
if (sim.samp.step < pb.startSampStep)
break; // Don't check it yet
v2 = 0; npart = 0; pp = sim.ps;

```

```

for (icell = 0; icell < ncell; icell++) {
    npart += cells[icell].n; // Number of particles in cell
    ipart = cells[icell].head; // Head of list of particles in cell
    while (ipart != ENDLIST) {
        pl = pp + ipart; // Points to particle state
        v2 += pl->vx*pl->vx + pl->vy*pl->vy + pl->vz*pl->vz;
        ipart = pl->list.next; // Next particle in this cell
    } /*while*/
} /*for*/
v2 = v2/npart * pb.mass/(3.0 * pb.boltz); // Actual temperature
if (fabs(v2 - bound.T) > 0.1)
    cout << "\nTemperature is wrong at time_"
    << sim.samp.step << "; temperature =_" << v2 << "\n";

#endif
break;

case rt_thermal_wall:
    // Determine the number of particles to "move" within the reservoir.
    // This calculation simply adds a random deviate (call to 'genrand')
    // to the previously determined mean number of particles to move;
    // this compensates by adding an extra particle with the right
    // frequency to accommodate a non-integer mean.
    // The particle count is Poisson distributed about the mean:
    count = genrandp (long (bound.npart + genrand()));
    break;
} /*switch*/

// Check whether the reservoir should have a Maxwell-Boltzmann particle
// distribution or a Chapman-Enskog distribution; this is specified by
// the user.

ps = sim.ps; // Array of all possible particles
rp--; // First reservoir particle (if 'closed')

for (i = 0; i < count; i++) {
    switch (bound.resType) {
        case rt_generate_maxwell_boltz:
        case rt_generate_fixed:
            x = L * genrand(); // Assign random position in reservoir
            v[X] = stdev * fabs(genrandn()); // Gaussian dist. thermal velocity
            if ((x += (v[X] + bound.v_parallel)*tau + x.accel) <= L)
                continue; // Particle didn't enter the system
            v[X] += v.accel; // Adjust the velocity due to any force
            v[Y] = stdev * genrandn() + bound.v_shear; // Other velocity components
            v[Z] = stdev * genrandn();
            // Set the particle position, depending on whether it is entering
            // from the left or right.
            x = (isLeftBound) ? x - L : bound.position - (x - L);
            break;

        case rt_generate_chap_enskog:
            // The configuration file specifies that a Chapman-Enskog
            // (modified Maxwell-Boltzmann) distribution be used.
            ChapmanEnskogDist(bound.chapEnsk, v);
            v[X] *= stdev; // Chapman-Enskog distributed velocity
            if ((x += (v[X] + bound.v_parallel)*tau + x.accel) <= L)
                continue; // Particle didn't enter the system
            v[X] += v.accel; // Adjust the velocity due to any force
            v[Y] *= stdev; // Turn normalized velocity components
            v[Z] *= stdev; // into actual velocity components
            // Set the particle position, depending on whether it is entering
            // from the left or right.
            x = (isLeftBound) ? x - L : bound.position - (x - L);
            break;

        case rt_closed_system:
            rp++;
            // Move particle by free flight under the possible influence of
            // an external, constant force:  $x = x + v_x * t + 0.5 * a * t^2$ . Any
            // particles that strike the right wall of the reservoir system are

```

```

// assumed to leave the reservoir and enter the system; although
// the particle does not actually leave the reservoir (it is
// specularly reflected, a particle is "created" with the same
// velocity and position the reservoir particle would have if it
// actually moved into the system.
x_old = rp->x; vx_old = rp->vx; // Save old position and x-velocity
x = x_old + vx_old*tau + x_accel;
if (x >= 0.0 && x <= L) {
    rp->x = x; // New particle position
    rp->vx += v_accel; // Adjust the velocity due to any force
    continue; // This particle won't enter the system
} /*if*/

if (a != 0.0) {
    // @@@ This is not implemented correctly!
    double t, vx_tmp;
    dist = bound.position - x_old; // Distance to the wall
    t = timeToMoveDist (dist, vx_old, a); // Find time to wall
    vx_tmp = -(vx_old+a*t); // Velocity after hitting the wall
    t = tau - t; // Remainder of timestep
    rp->vx = vx_tmp + a*t;
    rp->x = bound.position + vx_tmp*t + 0.5*a*t*t; // Distance to wall
} /*if*/

rp->vx = -vx_old; // Reverse the x-velocity
if (x < 0.0) {
    // Specular reflection from the wall farthest from the system:
    rp->x = -x;
    if (rp->x > bound.length) rp->x = bound.length;
    continue; // The particle will not enter the system
} /*if*/

// The particle has struck the wall of the reservoir that is
// nearest the system, so it is effectively entering the system.
x -= L; // How far into the system it went
if ((rp->x = L - x) < 0) // New position in the reservoir
    rp->x = DBLEPSILON; // Special (very unlikely) case
if (!isLeftBound)
    x = bound.position - x; // Reservoir is at right bound of system
v[X] = -rp->vx; v[Y] = rp->vy; v[Z] = rp->vz; // Particle velocity
break;

case rt_thermal.wall:
    // Reset velocity components as biased Maxwellian, exponential
    // distribution in x; Gaussian in y and z.
    v[X] = direction*sqrt(-log(1-genrand())) * bound.mpv;
    // Include shear velocity of wall or reservoir particles
    v[Y] = stdev*genrandn() + bound.v_shear;
    v[Z] = stdev*genrandn();
    // Reset the position after leaving the wall, assuming the new
    // x-velocity for the remainder of the time step (that part of the
    // time step not used in reaching the wall). Also record velocity
    // changes for making force measurements.
    if (a != 0.0) {
        // Find the time needed to reach the wall so we know how much of
        // the timestep remains after particle hits wall.
        dist = bound.position - x_old; // Distance to the wall
        t = timeToMoveDist (dist, vx_old, a);
        t = tau - t; // Remainder of timestep after hit wall
        x = bound.position+DBLEPSILON + v[X]*t + 0.5*a*t*t;
        v[X] += a * t;
    }
    else {
        x = bound.position + v[X]*tau*(x-bound.position)/(x-x_old);
    } /*if*/
    break;
} /*switch*/

// The vector 'v' contains the velocity of a particle entering the

```

```

    // system; 'x' contains its position relative to the boundary.
    if (sim.npart >= sim.maxPart)
        break; // All particles are used up — quit
    pp = sim.free;
    sim.free = ps + pp->list.next; // Update the free list
    ipart = (ListIndex) (pp - ps); // Index to this new particle
    if (ipart > sim.lastPart)
        sim.lastPart = ipart;

    // Set the position and velocity components.
    pp->x = x; // Particle position
    pp->vx = v[X]; pp->vy = v[Y]; pp->vz = v[Z]; // Particle velocity

    // Determine which cell the particle is in
    icell = (int) (((double)ncell)*x/xmax); // Determine particle's cell
    icell -= (icell >= ncell); // Handle "overflow" case
    head = cells[icell].head; // Index to first particle in cell
    pp->list.prev = END_LIST; // Link the new particle into the
    pp->list.next = head; // list of particles for the cell
    if (head != END_LIST)
        ps[head].list.prev = ipart;
    cells[icell].head = ipart;
    cells[icell].n++; // One more particle in this cell
    sim.npart++; // One more particle in the system

    // Keep track of the velocity distribution for particles entering the
    // system so a histogram can be created. (This is optional.)
    if (hist_len > 0) {
        number = hist_len / (bound.mpv + 6.0*bound.mpv_stdev);
        if ((bin = abs((int) (pp->vx * number))) >= hist_len)
            bin = hist_len - 1; // Too big — lump at tail end
        wstats->hist[bin].v_enter++; // Increment count for this velocity
    } /*if*/
    n_enter++; // Increment count of particles entering

    // Finally, update the variables used to check for energy and momentum
    // conservation.
    sim.samp.energy += pp->vx*pp->vx + pp->vy*pp->vy + pp->vz*pp->vz;
    sim.samp.px += pp->vx;
    sim.samp.py += pp->vy;
    sim.samp.pz += pp->vz;
} /*for*/

return n_enter;
} /* ParticlesFromReservoir */

/*
 * ResetStatistics clears statistics gathered by "Mover" and "Collider".
 * Entry:
 *   sim -> simulation state descriptor
 * Exit:
 *   none
 */
void ResetStatistics (SimState& sim)
{
    CellStats *cs = sim.samp.cells;
    WallStats *ws;
    int i, j;

    sim.samp.nsamp = 0;
    sim.samp.collTot = 0.0;
    for (i = 0, ws = &sim.samp.lwall; i < 2; i++) {
        ws->dpxTot = ws->dpyTot = ws->dpyErr = 0.0;
        ws->strikeSum = 0;
        // Clear the optional histograms for collecting the velocity
        // distributions for particles entering and leaving the system; if
        // these optional histograms are not present, the loop won't execute.
        for (j = 0; j < sim.pb->boundaryHist; j++) {
            ws->hist[j].v_enter = ws->hist[j].v_leave = 0;

```

```

        ws->hist[j].n_enter = ws->hist[j].n_leave = 0;
    } /*for*/
    ws = &sim.samp.rwall;
} /*for*/

// Initialize the statistics gathered for individual cells.
for (i = 0; i < sim.pb->ncell; i++, cs++) {
    cs->collTot = 0.0;
    cs->sum_px = cs->sum_py = cs->sum_pz = 0.0;
    cs->l_tot_px = 0.0;
    cs->l_particle_net = 0;
} /*for*/
} /* ResetStatistics */

/*
* Sampler samples mean density, velocity and temperature for each
* cell. Additional calculations or statistics gathering is performed
* as needed for obtaining any correlations needed for the simulation.
* This function assumes that for each cell there exists a valid
* linked list of particles in the cell.
* Entry:
*   sim -> simulation state descriptor
* Exit:
*   error flags
*/
unsigned Sampler (SimState& sim)
{
    SimParmBlock *pb;
    ParticleState *pp, *ps;
    CellState *cells;
    CellStats *cs;
    struct CorrStats_ *corr;
    double energy, px, py, pz, vx, vy, vz;
    double sum_n, sum_vx, sum_vy, sum_vz, sum_v2;
    ListIndex ipart;
    int icell, ncell, ncorr;

    // Extract some commonly needed values to increase performance.
    ps = sim.ps;
    pb = sim.pb;
    cells = sim.cells; // Array of state info for each cell
    ncell = pb->ncell; // Number of cells in system
    cs = sim.samp.cells;
    energy = px = py = pz = 0.0; // Statistics for this timestep only

    // Compute cell location for each particle
    for (icell = 0; icell < ncell; icell++, cs++) {
        // When gathering statistics for a cell, we need to be careful that
        // it contains at least one particle to avoid division by zero. This
        // case can happen with strong gravity or a small number of particles
        // (and it did happen while debugging the program with large gravity!).
        // Obviously, if there are no particles in a cell, the statistics for
        // that cell will not change except for being averaged over one more
        // sample.
        if ((sum_n = cells[icell].n) <= 0.0) // Number of particles in cell
            continue;

        ipart = cells[icell].head; // Head of list of particles in cell
        // Initialize running sums
        sum_vx = sum_vy = sum_vz = sum_v2 = 0.0;
        // For each particle, accumulate running sums for its cell
        while (ipart != END_LIST) {
            pp = ps + ipart; // Points to particle state
            sum_vx += (vx = pp->vx);
            sum_vy += (vy = pp->vy);
            sum_vz += (vz = pp->vz);
            sum_v2 += vx*vx + vy*vy + vz*vz;

            ipart = pp->list.next; // Next particle in this cell
        } /*while*/
    }
}

```



```

    // Use current sums to update sample number, momentum and temperature.
    cells[icell].px = sum_vx;           // Momentum/mass for the cell
    cells[icell].energy = sum_v2;       // Proportional to total energy in cell
    cs->sum_px += sum_vx;               // For obtaining net momentum in cell
    cs->sum_py += sum_vy;
    cs->sum_pz += sum_vz;
    cs->sum_KE += sum_v2;               // For obtaining net energy in the cell
    px += sum_vx; py += sum_vy; pz += sum_vz;
    energy += sum_v2;                  // For sum of particle energies
    sum_vx /= sum_n;
    sum_vy /= sum_n;
    sum_vz /= sum_n;
    sum_v2 /= sum_n;
    cs->avg_n += sum_n;                 // Statistics for the cell
    cs->avg_px += sum_vx;
    cs->avg_py += sum_vy;
    cs->avg_pz += sum_vz;
    cs->avg_KE += sum_v2;
    cells[icell].T = sum_v2 -           // For instantaneous temperature
        (sum_vx*sum_vx + sum_vy*sum_vy + sum_vz*sum_vz);
    cs->avg_T += cells[icell].T;
} /*for*/

// Save statistics that are valid only for a single timestep. These
// are useful for checking for conservation of momentum and energy,
// where appropriate, or for watching the evolution of these quantities
// (although those activities are up to the caller to perform).

sim.samp.energy = energy;
sim.samp.px = px; sim.samp.py = py; sim.samp.pz = pz;

// Perform additional actions needed to calculate any correlations
// that are to be determined in this simulation.

if ((ncorr = sim.samp.ncorr) > 0) {
    for (corr = sim.samp.correl; ncorr > 0; ncorr--, corr++)
        corr->sampFn (sim, *corr);    // Call the sampling function
} /*if*/

sim.samp.tsamp += pb->tau;             // Update total sampling time
sim.samp.nsamp++;                     // Increment sample counter

return 0;
} /* Sampler */

/*
* Sorter creates a list of all particles in each cell, as well as an
* index into this list for each cell that allows rapidly locating the
* set of particle indices for that cell. This function assumes the
* following have already been determined:
* 1. Each particle's cell number
* 2. The number of particles in each cell
*
* Entry:
*   sim -> simulation state descriptor
* Exit:
*   error flags
*/
unsigned Sorter (SimState& sim)
{
    ParticleState *ps;
    CellState *cells;
    ListIndex *pXref;
    ListIndex ipart, iXref;
    int icell, ncell;
    ps = sim.ps;
    cells = sim.cells;
    ncell = sim.pb->ncell;
    pXref = sim.partXref - 1;          // Minus 1 to account for preincrement
    iXref = 0;                        // First particle list index for cell 0

```

```

for (icell = 0; icell < ncell; icell++) {
    cells[icell].index = iXref;
    iXref += (ListIndex) cells[icell].n;
    ipart = cells[icell].head;    // First particle in this cell

    while (ipart != END.LIST) {
        *++pXref = ipart;    // Store the index to the particle
        ipart = ps[ipart].list.next;    // Index of next particle in this cell
    }    /*while*/
}    /*for*/

return 0;
}    /* Sorter */

/*
 * TimeString formats a time into a character string that contains the
 * time and date.
 * Entry:
 *     t -> time to format
 * Exit:
 *     pointer to character string containing the time and date
 */
static const char *TimeString (time_t t)
{
    struct tm *tblock;
    static char sbuf[80];

    tblock = localtime (&t);
    strcpy (sbuf, asctime(tblock));
    sbuf [strlen(sbuf)-1] = 0;    // Delete the newline character
    return (const char *) sbuf;
}    /* TimeString */

/*
 * RunSim runs a single simulation to completion. This is the
 * top-level function for controlling a simulation.
 * Entry:
 *     sim -> simulation state descriptor
 * Exit:
 *     none
 */
void RunSim (SimState& sim)
{
    enum simState_ {
        ss_initial_relax = 0,    // Settle after init (reservoirs only)
        ss_get_stats_for_reservoirs,    // Start collecting stats (reservoirs)
        ss_calc_reservoir_density,    // Set reservoir parameters (reservoirs)
        ss_final_relax,    // Relax system in final configuration
        ss_prepare_to_gather_stats,    // Clear statistics, etc.
        ss_gather_stats    // Actual simulation, gather statistics
    } simState;
    SimParmBlock &pb = *sim.pb;
    BoundaryType &lb = pb.lbound, &rb = pb.rbound;
    time_t start_time = t;
    long action_tstep, showProgress, startStep;
    long istep, nstep;
    unsigned errFlags;
    BoundaryCondType lcond, rcond;

    nstep = pb.nstep;
    showProgress = pb.showSimProgress;    // Interval for displaying progress
    startStep = 0;
    simState = ss_final_relax;    // Assume system without reservoirs
    action_tstep = pb.startSampStep-1;    // Sets time for begin of sampling

    // If there are reservoirs, set the boundaries to thermal walls until the
    // densities have been determined; for systems without reservoirs leave
    // everything alone and just let the system relax.

    lcond = lb.boundCond;    rcond = rb.boundCond;
    if (lb.boundCond == bc_reservoir && lb.resType != rt_closed_system) {
        // We will need to calculate the nominal particle density at each wall

```

```

// of the system in the absence of a reservoir (using thermal walls to
// gather statistics), and then scale this density by any factor
// provided in the simulation control file. Calculating the nominal
// density is difficult for a user to do accurately when the two
// boundary temperatures are unequal, so it is done by the system
// during the system relaxation time. We let the system relax using
// thermal walls at the reservoir temperatures and then obtain the
// density profile by gathering density statistics; in this case the
// first quarter of the relaxation time is devoted to letting the
// initial system settle, the next quarter is used to gather density
// statistics to be used for calculating the reservoir densities, and
// the last half is used to let the system relax using the reservoirs
// with the calculated densities.

lb.boundCond = bc_thermal_wall; // Use thermal walls during relaxation
simState = ss.initial_relax;
action_tstep = pb.startSampStep/4; // Relax system in first quarter
} /*if*/
if (rb.boundCond == bc.reservoir && rb.resType != rt_closed_system) {
    rb.boundCond = bc_thermal_wall;
    simState = ss.initial_relax;
    action_tstep = pb.startSampStep/4; // Relax system in first quarter
} /*if*/

cout << "\nStarting_simulation";
if (pb.nrun > 1)
    cout << "_" << (pb.run+1);
if (pb.name[0] != 0)
    cout << "_(\\" << pb.name << "\\)\n_____";
startTime = time(NULL);
cout << "_at_" << TimeString(startTime) << "\n";

// Loop for the desired number of time steps.

ResetStatistics(sim); // Clear wall and collision statistics
for (istep = 1; istep <= nstep; istep++) {
    sim.samp.step++; // Increment the time step number

    // If it's time to display the simulation progress, do it.
    if (istep % showProgress == 0) {
        cout << "____Finished_" << istep << "_of_" << nstep << "_steps";
        if (istep > startStep) {
            cout << "____collisions_" << sim.samp.collTot;
            if (istep % (10*showProgress) == 0) {
                t = time(NULL);
                t += (time_t) ( (double)(t-startTime)/(double)(istep-startStep)
                               * (double)(nstep-istep) );
                cout << "____(Estimated_completion_of_simulation_#"
                     << (pb.run+1) << ":_:"
                     << TimeString(t) << ")";
            } /*if*/
        } /*if*/
        cout << "\n";
    } /*if*/

    // Handle particle movement and collision, verifying that energy
    // and momentum are conserved where appropriate.

    // Move particles, assuming free flight
    if ((errFlags = Mover (sim) << 8) != 0 ||
        // Build cross-reference for each cell
        (errFlags = Sorter (sim) << 4) != 0 ||
#ifdef COLLISIONLESS == 1)
        simState < ss_final_relax &&
#endif
        // Process representative collisions
        (errFlags = Collider(sim) << 0) != 0) {
        cout << "***_Error:_conservation_violation_at_timestep_"
             << istep << "_when_";
    }
}

```

```

    if ( (errFlags >> 8) != 0) { errFlags >= 8; cout << "moving"; }
    else if ((errFlags >> 4) != 0) { errFlags >= 4; cout << "sorting"; }
    else cout << "colliding";
    cout << ";_violation_in";
    if (errFlags & 1) cout << "_energy";
    if (errFlags & 2) cout << "_p_x";
    if (errFlags & 4) cout << "_p_y";
    if (errFlags & 8) cout << "_p_z";
    cout << "\n";
    return; // Abort the simulation
} /*if*/

// Take the appropriate action for the current state of the simulation
// process.
switch (simState) {
    case ss_initial_relax: // (Only for systems with reservoirs)
        if (istep >= action_tstep) {
            // We want to gather density statistics during the next
            // quarter of the system relaxation time. Allocate temporary
            // buffer to use in getting average particle density.
            simState = ss_get_stats_for_reservoirs;
            action_tstep = pb.startSampStep/2;
            sim.samp.nsamp = 0;
        } /*if*/
        break;

    case ss_get_stats_for_reservoirs: // (Only for systems with reservoirs)
        if (istep <= action_tstep)
            sim.samp.nsamp++;
        else
            simState = ss_calc_reservoir_density;
        break;

    case ss_calc_reservoir_density: // (Only for systems with reservoirs)
        lb.boundCond = lcond; // Restore the desired boundary types
        rb.boundCond = rcond;
        simState = ss_final_relax; // Next state
        action_tstep = pb.startSampStep-1; // Finish relaxation
        break;

    case ss_final_relax: // (Used in all system configurations)
        if (istep >= action_tstep)
            simState = ss_prepare_to_gather_stats;
        break;

    case ss_prepare_to_gather_stats:
        // Prepare to start gathering statistics, which will start on
        // the next timestep.
        startStep = pb.startSampStep; // Used for estimating completion time
        startTime = time(NULL);
        cout << "Statistics_gathering_started_at_time_step_" << istep
             << "_after_" << sim.samp.collTot << "_collisions\n"
             << "_____(" << TimeString(startTime) << ")\n";
        ResetStatistics(sim); // Clear wall and collision statistics
        simState = ss_gather_stats;
        break;

    case ss_gather_stats: // (Used in all system configurations)
        Sampler (sim); // Gather statistics for each cell
        break;
} /*switch*/

} /*for*/

cout << "Simulation_" << (pb.run+1) << "_completed_at_"
     << TimeString(time(NULL)) << "\n";
} /* RunSim */

```

C.4 Covariance Calculation and Output Functions

```

/*-----
* CORRFN.CPP - Functions to handle statistics needed to calculate
* correlations for a direct simulation (DSMC) of a
* rarified gas in a microscopic system. These functions
* include:
*     1. Initialization
*     2. Sampling of simulation state to obtain statistics
*     3. Saving statistics to the simulation output file.
* Each time a new correlation type is defined, it should
* include function to perform each of these activities,
* the function pointers should be added to the
* 'CorrFunctions' array, and a new bit should be defined
* in "FLUCT.H" that is unique to the correlation.
* NOTE: The original file contained much repetitive code, which is
* easily recreated from this skeletal file as indicated in
* some function descriptions. Since the code below is for
* documentation only, only representative routines (for N-K
* covariances) are retained for economy of space, although all
* 'Save' functions are also retained.
*
* Written by Marty Tysanner
*-----*/

#include <stdlib.h>
#include <stdio.h>
#include <alloc.h>
#include <string.h>
#include <ctype.h>
#include <memory.h>
#include <math.h>
#include <float.h>
#include <fcntl.h>
#include <fstream.h>
#include <time.h>
#include "fluct.h"

// The 'CorrFunctions' array is defined at the end of this file.

/*-----
* Corr_NK performs sampling needed to determine density-energy
* correlations.
* NOTE: For other correlation types, the 'cells[i].n' and
*       'cells[j].energy' structure components should be replaced by:
*       CORR_NN -> cells[i].n, cells[j].n
*       CORR_NP -> cells[i].n, cells[j].px
*       CORR_TT -> cells[i].T, cells[j].T
* Entry:
*       sim -> simulation state descriptor
*       stats -> information structure for this correlation type
* Exit:
*       none
*-----*/

void Corr_NK (struct SimState_& sim, struct CorrStats_& stats)
{
    CellState *cells;
    double *avg_NK;
    int i, j, ncell;

    cells = sim.cells;
    ncell = sim.pb->ncell;
    avg_NK = stats.avg_ij;

    for (i=0; i < ncell; i++)
        for (j = 0; j < ncell; j++)
            avg_NK [INDEX2(ncell,i,j)] += cells[i].n * cells[j].energy;
} /* Corr_NK */

/*-----
* Init_NK performs initialization for obtaining density-temperature
* covariances.
* NOTE: For other covariance types, the code is identical.
*-----*/

```

```

*      Entry:
*      sim  -> simulation state descriptor
*      corr -> information structure for this correlation type
*      Exit:
*      none
*/
void Init_NK (struct SimState_& sim, struct CorrStats_& corr)
{
    int i, j, ncell = sim.pb->ncell;

    // Allocate storage for the correlation data
    corr.avg_ij = (double *) malloc(sizeof(double) * ncell*ncell);
    for (i = 0; i < ncell; i++)
        for (j = 0; j < ncell; j++)
            corr.avg_ij[INDEX2(ncell,i,j)] = 0.0;
} /* Init_NK */

/*
* Save_NK saves to an output file the statistics needed to calculate
* density-energy correlations, then performs cleanup including
* freeing of correlation statistics memory.
* Entry:
* sim  -> simulation state descriptor
* stats -> information structure for this correlation type
* sfile -> stream to which statistics are to be output
* Exit:
* none
*/
void Save_NK (struct SimState_& sim, struct CorrStats_& stats, FILE *sfile)
{
    double *avg_NK, coeff;
    int count, i, j, ncell;
    char sbuf[48];

    avg_NK = stats.avg_ij;
    ncell = sim.pb->ncell;
    // Calculate ahead of time the factor for converting an avg_NK entry to
    // a product of density and temperature. The avg_NK entries were
    // calculated from avg-T entries, and an avg-T entry must be converted
    // to a kinetic energy by multiplying by mass/2; the average is the
    // result over 'nsamp' samples:
    // coeff = ( mass/2 )/nsamp.
    coeff = ( sim.pb->mass/2 )/sim.samp.nsamp;

    // Line up values at col. 21
    fwrite (sbuf,
            sprintf(sbuf, "correlation-----%d\n", CORR_NK),
            1,
            sfile);

    // Output the correlation information.
    for (i=0; i < ncell; i++) {
        fwrite ("----", 4, 1, sfile); // Start row at column 5
        for (j = 0 ; j < ncell; j++) {
            count = sprintf( sbuf,
                            "%16g",
                            coeff * avg_NK[INDEX2(ncell,i,j)] ),
            fwrite (sbuf, count, 1, sfile);
        } /*for*/
        fwrite ("\n", 1, 1, sfile);
    } /*for*/

    free (stats.avg_ij); // Release correlation-specific memory
} /* Save_NK */

/*
* Save_NN saves to an output file the statistics needed to calculate
* density-density correlations, then performs cleanup including
* freeing of correlation statistics memory.
* Entry:
* sim  -> simulation state descriptor
* stats -> information structure for this correlation type
*/

```

```

*      sfile -> stream to which statistics are to be output
*      Exit:
*      none
*/
void Save_NN (struct SimState_& sim, struct CorrStats_& stats, FILE *sfile)
{
    double      *avg_nn;
    double      nsamp = sim.samp.nsamp;
    int          i, j, ncell;
    char         sbuf[48];

    avg_nn = stats.avg_ij;
    ncell = sim.pb->ncell;

    // Line up values at col. 21
    fwrite (sbuf,
            sprintf(sbuf, "correlation.....%d\n", CORR_NN),
            1,
            sfile);

    // Output the correlation information
    for (i=0; i < ncell; i++) {
        fwrite ("----", 4, 1, sfile); // Start row at column 5
        for (j = 0 ; j < ncell; j++)
            fwrite ( sbuf,
                    sprintf(sbuf, "%.16g", avg_nn[INDEX2(ncell,i,j)]/nsamp),
                    1,
                    sfile );
        fwrite ("\n", 1, 1, sfile);
    } /*for*/

    free (stats.avg_ij); // Release correlation-specific memory
} /* Save_NN */

/*
* Save_NP saves to an output file the statistics needed to calculate
* density-momentum correlations, then performs cleanup including
* freeing of correlation statistics memory.
* Entry:
*      sim -> simulation state descriptor
*      stats -> information structure for this correlation type
*      sfile -> stream to which statistics are to be output
* Exit:
*      none
*/
void Save_NP (struct SimState_& sim, struct CorrStats_& stats, FILE *sfile)
{
    double      *avg_np;
    double      mass = sim.pb->mass;
    double      nsamp = sim.samp.nsamp;
    int          count, i, j, ncell;
    char         sbuf[48];

    avg_np = stats.avg_ij;
    ncell = sim.pb->ncell;

    // Line up values at col. 21
    fwrite (sbuf,
            sprintf(sbuf, "correlation.....%d\n", CORR_NP),
            1,
            sfile);

    // Output the correlation information, converting the velocity
    // component of avg_np to momentum since avg_np was actually calculated
    // as avg_nv in Corr_NP.
    for (i=0; i < ncell; i++) {
        fwrite ("----", 4, 1, sfile); // Start row at column 5
        for (j = 0 ; j < ncell; j++) {
            count = sprintf( sbuf,
                            "%.16g",
                            avg_np[INDEX2(ncell,i,j)]*mass/nsamp),
            fwrite (sbuf, count, 1, sfile);
        }
    }
}

```

```

    } /*for*/
    fwrite ("\n", 1, 1, sfile);
} /*for*/

free (stats.avg_ij); // Release correlation-specific memory
} /* Save_NP */

/*-----*/
* Save_TT saves to an output file the statistics needed to calculate
* temperature-temperature correlations.
* Entry:
*   sim -> simulation state descriptor
*   stats -> information structure for this correlation type
*   sfile -> stream to which statistics are to be output
* Exit:
*   none
*-----*/
void Save_TT (struct SimState_& sim, struct CorrStats_& stats, FILE *sfile)
{
    double    *avg_TT;
    double    coeff;
    int       i, j, ncell;
    char      sbuf[48];

    avg_TT = stats.avg_ij;
    ncell = sim.pb->ncell;
    // Calculate ahead of time the factor for converting an avg_TT entry to
    // a product of temperatures. The avg_TT entries were calculated from
    // a product of avg_T entries, and an avg_T entry must be converted to
    // an actual temperature by multiplying by mass/(3*boltz); hence to
    // obtain a correct avg_TT entry we must multiply by the square of the
    // same factor, and average the result over 'nsamp' samples:
    //   coeff = ( [ mass/(3*boltz) ]^2 )/nsamp.
    coeff = sim.pb->mass/(3.0 * sim.pb->boltz);
    coeff = coeff*coeff/sim.samp.nsamp;

    // Line up values at col. 21
    fwrite (sbuf,
            sprintf(sbuf, "correlation %d\n", CORR_TT),
            1,
            sfile);

    // Output the correlation information
    for (i=0; i < ncell; i++) {
        fwrite ("----", 4, 1, sfile); // Start row at column 5
        for (j = 0 ; j < ncell; j++)
            fwrite ( sbuf,
                    sprintf(sbuf, "_%.16g", coeff * avg_TT[INDEX2(ncell,i,j)]),
                    1,
                    sfile );
        fwrite ("\n", 1, 1, sfile);
    } /*for*/

    free (stats.avg_ij); // Release correlation-specific memory
} /* Save_TT */

/*-----*/
* Global list of correlation-related functions and identifiers
*-----*/
CorrFuncList CorrFunctions[] = {
    { Corr_NN, Init_NN, Save_NN, CORR_NN },
    { Corr_NP, Init_NP, Save_NP, CORR_NP },
    { Corr_TT, Init_TT, Save_TT, CORR_TT },
    { Corr_NK, Init_NK, Save_NK, CORR_NK },
};

```

C.5 Random Deviate Generation Functions

```

/*-----*/

```



```

* RAND.CPP - Random number generation functions, using the Mersenne
* Twister (MT19937) random number generator. This generator
* was obtained from the Mersenne Twister web page:
* http://www.math.keio.ac.jp/~matumoto/emt.html
* on July 5, 2002. The only changes made to the code
* are style changes to maintain consistency with the style
* of the rest of the code.
*
* The genrandn function is basically the randn function written by
* A. Garcia, from Numerical Methods for Physics, 2nd Edition (2000).
* Very minor changes have been made to make it compatible with
* MT19937.
*
* — Start of comments/code for MT19937 —
*
* A C-program for MT19937: Real number version ([0,1)-interval)
* (1999/10/28)
*
* genrand() generates one pseudorandom real number (double)
* which is uniformly distributed on [0,1)-interval, for each
* call. sgenrand(seed) sets initial values to the working area
* of 624 words. Before genrand(), sgenrand(seed) must be
* called once. (seed is any 32-bit integer.)
* Integer generator is obtained by modifying two lines.
* Coded by Takuji Nishimura, considering the suggestions by
* Topher Cooper and Marc Rieffel in July-Aug. 1997.
*
* This library is free software; you can redistribute it and/or
* modify it under the terms of the GNU Library General Public
* License as published by the Free Software Foundation; either
* version 2 of the License, or (at your option) any later
* version.
* This library is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
* See the GNU Library General Public License for more details.
* You should have received a copy of the GNU Library General
* Public License along with this library; if not, write to the
* Free Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA
* 02111-1307 USA
*
* Copyright (C) 1997, 1999 Makoto Matsumoto and Takuji Nishimura.
* Any feedback is very welcome. For any question, comments,
* see http://www.math.keio.ac.jp/matsumoto/emt.html or email
* matumoto@math.keio.ac.jp
*
* REFERENCE
* M. Matsumoto and T. Nishimura,
* "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform
* Pseudo-Random Number Generator",
* ACM Transactions on Modeling and Computer Simulation,
* Vol. 8, No. 1, January 1998, pp 3--30.
*
*/

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

#define TWO_PI (2.0 * M_PI) /* Used in genrandn() */

// Period parameters
#define N 624
#define M 397
#define MATRIX_A 0x9908b0df /* constant vector a */
#define UPPER_MASK 0x80000000 /* most significant w-r bits */
#define LOWER_MASK 0x7fffffff /* least significant r bits */

// Tempering parameters
#define TEMPERING_MASK_B 0x9d2c5680
#define TEMPERING_MASK_C 0xefc60000
#define TEMPERING_SHIFT_U(y) (y >> 11)
#define TEMPERING_SHIFT_S(y) (y << 7)
#define TEMPERING_SHIFT_T(y) (y << 15)
#define TEMPERING_SHIFT_L(y) (y >> 18)

static unsigned long mt[N]; // the array for the state vector

```

```

static int mti = N+1;           // mti==N+1 means mt[N] is not initialized

/*
 * sgenrand initializes the array with a seed.
 * Entry:
 *     seed -> integer seed (DO NOT USE A SEED OF ZERO)
 * Exit:
 *     none
 */
void sgenrand (unsigned long seed)
{
    int i;
    for (i = 0; i < N; i++) {
        mt[i] = seed & 0xffff0000;
        seed = 69069 * seed + 1;
        mt[i] |= (seed & 0xffff0000) >> 16;
        seed = 69069 * seed + 1;
    } /*for*/
    mti = N;
} /* sgenrand */

/*
 * gammln returns the value
 *     ln [\Gamma(x)]
 * Entry:
 *     x => argument to \Gamma function
 * Exit:
 *     natural logarithm of the \Gamma function for the argument.
 */
double gammln (double x)
{
    static double cof[6] = { 76.18009172947146,      -86.50532032941677,
                             24.01409824083091,      -1.231739572450155,
                             0.1208650973866179e-2,   -0.5395239384953e-5 };

    double ser, tmp, y;
    int j;
    y = x;
    tmp = x + 5.5;
    tmp -= (x + 0.5) * log(tmp);
    ser = 1.000000000190015;
    for (j = 0; j < sizeof(cof)/sizeof(cof[0]); j++)
        ser += cof[j]/++y;
    return -tmp + log(2.5066282746310005*ser/x);
} /* gammln */

/*
 * genrand is a random number generator which provides a uniform
 * distribution in [0,1).
 * Entry:
 *     none
 * Exit:
 *     random number uniformly distributed in [0,1)
 */
double genrand (void)
{
    unsigned long y;
    static unsigned long mag01[2] = { 0x0, MATRIX_A };
    // mag01[x] = x * MATRIX_A for x=0,1

    if (mti >= N) {
        // generate N words at one time
        int kk;
        // if (mti == N+1)           // if sgenrand() has not been called,
        //     sgenrand(4357);       // a default initial seed is used

        for (kk = 0; kk < N-M; kk++) {
            y = (mt[kk] & UPPER_MASK) | (mt[kk+1] & LOWER_MASK);
            mt[kk] = mt[kk+M] ^ (y >> 1) ^ mag01[y & 0x1];
        } /*for*/
    }
}

```

```

    for ( ; kk < N-1; kk++) {
        y = (mt[kk] & UPPER_MASK) | (mt[kk+1] & LOWER_MASK);
        mt[kk] = mt[kk+(M-N)] ^ (y >> 1) ^ mag01[y & 0x1];
    } /*for*/
    y = (mt[N-1] & UPPER_MASK) | (mt[0] & LOWER_MASK);
    mt[N-1] = mt[M-1] ^ (y >> 1) ^ mag01[y & 0x1];
    mti = 0;
} /*if*/

y = mt[mti++];
y ^= TEMPERING_SHIFT_U(y);
y ^= TEMPERING_SHIFT_S(y) & TEMPERING_MASK_B;
y ^= TEMPERING_SHIFT_T(y) & TEMPERING_MASK_C;
y ^= TEMPERING_SHIFT_L(y);

return ( (double)y * 2.3283064365386963e-10 ); /* reals: [0,1)-interval */
} /* genrand */

/*-----
 * genrandn is a random number generator which provides a Normal
 * (Gaussian) distribution.
 * Entry:
 * none
 * Exit:
 * random number, Gaussian distributed
 *-----*/
double genrandn (void)
{
    return sqrt( -2.0*log(1.0 - genrand()) ) * cos( TWO_PI*genrand() );
} /* genrandn */

/*-----
 * genrandp is a random number generator which provides a Poisson
 * distribution of mean xm. The algorithm is adapted from the "poidev"
 * random deviate function in
 * Numerical Recipes In C: The Art Of Scientific Computing
 * Entry:
 * xm => mean
 * Exit:
 * random number, Poisson distributed
 *-----*/
double genrandp (double xm)
{
    static double sq, alxm, g;
    static double oldm=(-1.0); // Flag for whether xm has changed since last call
    double em, t, y;

    if (xm < 12.0) { // Use direct method
        if (xm != oldm) {
            oldm = xm;
            g = exp(-xm); // If xm is new, compute the exponential
        } /*if*/

        // Instead of adding exponential deviates it is equivalent to
        // multiply uniform deviates. We never actually need to take the
        // log, merely compare to the pre-computed exponential.
        em = -1;
        t = 1.0;
        do {
            ++em;
            t *= genrand();
        } while (t > g);
    }
    else {
        // Use the rejection-acceptance method.
        if (xm != oldm) {
            // If xm has changed since the last call, then precompute some
            // functions that occur below.
            oldm = xm;
            sq = sqrt(2.0 * xm);
        }
    }
}

```

```

    alxm = log(xm);
    g = xm * alxm - gammaln(xm+1.0); // gammaln = natural log of Gamma fn.
} /*if*/

do {
    do {
        // y is a deviate from a Lorentzian comparison function.
        y = tan(M_PI*genrand());
        em = sq*y + xm; // em is y, shifted and scaled
    } while (em < 0.0); // Reject if in regime of zero probability
    em = floor(em); // The trick for integer-valued distributions
    // The ratio of the desired distribution to the comparison
    // function; we accept or reject by comparing it to another
    // uniform deviate. The factor 0.9 is chosen so that t never
    // exceeds 1.
    t = 0.9*(1.0 + y*y) * exp(em*alxm - gammaln(em + 1.0) - g);
    } while (genrand() > t);
} /*if*/

return em;
} /* genrandp */

```