On the Gap-Tooth Direct Simulation Monte Carlo Method

by

Jessica D. Armour

Submitted to the School of Engineering
in partial fulfillment of the requirements for the degree of

Master of Science in Computation for Design and Optimization

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2012

Author .... [illegible]
School of Engineering
January 20, 2012

Certified by .... [illegible]
Nicolas Hadjiconstantinou
Associate Professor of Mechanical Engineering
Thesis Supervisor

Accepted by .... [illegible]
Markus Buehler
Associate Professor of Civil and Environmental Engineering

On the Gap-Tooth Direct Simulation Monte Carlo Method

by

Jessica D. Armour

Submitted to the School of Engineering
on January 20, 2012, in partial fulfillment of the
requirements for the degree of
Master of Science in Computation for Design and Optimization

Abstract

This thesis develops and evaluates Gap-tooth DSMC (GT-DSMC), a direct Monte Carlo simulation procedure for dilute gases combined with the Gap-tooth method of Gear, Li, and Kevrekidis. The latter was proposed as a means of reducing the computational cost of microscopic (e.g. molecular) simulation methods using simulation particles only in small regions of space (teeth) surrounded by (ideally) large gaps. This scheme requires an algorithm for transporting particles between teeth. Such an algorithm can be readily developed and implemented within direct Monte Carlo simulations of dilute gases due to the non-interacting nature of the particle-simulators.

The present work develops and evaluates particle treatment at the boundaries associated with diffuse-wall boundary conditions and investigates the drawbacks associated with GT-DSMC implementations which detract from the theoretically large computational benefit associated with this algorithm (the cost reduction is linear in the gap-to-tooth ratio). Particular attention is paid to the additional numerical error introduced by the gap-tooth algorithm as well as the additional statistical uncertainty introduced by the smaller number of particles. We find the numerical error introduced by transporting particles to adjacent teeth to be considerable. Moreover, we find that due to the reduced number of particles in the simulation domain, correlations persist longer, and thus statistical uncertainties are larger than DSMC for the same number of particles per cell. This considerably reduces the computational benefit associated with the GT-DSMC algorithm.

We conclude that the GT-DSMC method requires more development, particularly in the area of error and uncertainty reduction, before it can be used as an effective simulation method.

Thesis Supervisor: Nicolas Hadjiconstantinou
Title: Associate Professor of Mechanical Engineering

Acknowledgments

This work was completed while under the funding of MIT Lincoln Laboratory's Lincoln Scholars Program. A huge thanks to them for providing me with this opportunity to better myself.

To Professor Hadjiconstantinou, I would like to express special thanks for giving me a chance to work with him and for his never-ending patience with me throughout the entire process.

To my family, thanks for their continued love, support, and encouragement, especially in the face of the hardships they have faced the past couple of years.

And to my roommate, thanks for always encouraging me to laugh and reminding me to take life and education one day at a time.

Contents

Cover page **1**

Abstract **3**

Acknowledgments **5**

Contents **7**

List of Figures **11**

1 Introduction **13**

1.1 Equation-Free Methods: Introduction . . . 15

1.2 Thesis Objective . . . 17

2 Numerical Methods **19**

2.1 Direct Simulation Monte Carlo method . . . 19

2.1.1 Overview . . . 19

2.1.2 Advection Substep . . . 21

2.1.3 Collision Substep . . . 23

2.1.4 Sampling . . . 25

2.2 Gap-Tooth DSMC Method . . . 26

2.2.1 Overview . . . 26

2.2.2 Algorithm Details . . . 27

2.2.3 Boundary Treatment 30

3 Model Problem - Couette Flow **33**

3.1 Problem Setup 33

3.2 Results 35

3.2.1 Implementation I versus II 35

3.2.2 Verification of GT-DSMC for G = 0 36

3.2.3 Further Investigation of Implementation I 36

3.3 Computational Considerations 37

3.3.1 Computational Run Time versus Gap Size 37

3.3.2 Variance in Results versus Gap Size 37

4 Conclusion and Future Work **51**

4.1 Summary 51

4.2 Future Work 52

Appendix **53**

A Matlab Implementation code **53**

A.1 General Files 53

A.1.1 Setup file - setup.m 53

A.1.2 Particle sorting routine - sorter.m 54

A.1.3 Particle collision routine - colider.m 55

A.1.4 Statistical sampling routine - sampler.m 57

A.1.5 Post Processing routine - PostProcessing.m 58

A.2 DSMC specific files 59

A.2.1 Main DSMC wrapper - Truth.m 59

A.2.2 Particle movement routine - mover.m 61

A.3 Dual Bounce implementation files 62

A.3.1 Main GT-DSMC wrapper - GT2.m 62
A.3.2 GT particle movement routine - GT2mover.m 64
A.3.3 Recursive particle movement routine 1 - move.m 69
A.3.4 Recursive particle movement routine 2 - moveExtra.m 69
A.3.5 Recursive particle bounce routine - bounce.m 70

Bibliography **73**

List of Figures

1-1 Patch Dynamics Diagram 16

2-1 DSMC Flow Chart 21

2-2 Fluxes for a single tooth 28

2-3 Internal Gap-tooth flux and spacing diagram 29

2-4 Recursive Internal Tooth Particle Redistribution Diagram 30

2-5 External System Boundary Treatment of Left-Going Fluxes 31

2-6 External System Boundary Treatment of Right-Going Fluxes 32

3-1 Diagram of Couette Flow 34

3-2 Implementation I, $\mathsf{Kn} = 0.0626$ 39

3-3 Implementation I, $\mathsf{Kn} = 1$ 40

3-4 Implementation II, $\mathsf{Kn} = 0.0626$ 41

3-5 Implementation II, $\mathsf{Kn} = 1$ 42

3-6 Errors in Implementation I, $\mathsf{Kn} = 0.0626$ 43

3-7 Errors in Implementation I, $\mathsf{Kn} = 1$ 44

3-8 Errors in Implementation II, $\mathsf{Kn} = 0.0626$ 45

3-9 Errors in Implementation II, $\mathsf{Kn} = 1$ 46

3-10 Error Comparison, DSMC vs. implementation II GT-DSMC at $\mathsf{G} = 0$, $\mathsf{Kn} = 0.0626$ 47

3-11 Scaled Implementation I, $\mathsf{Kn} = 0.0626$ 48

3-12 Computational Time versus Gap Size 49

3-13 Variance over 50 Ensembles versus Gap Size 50

Chapter 1

Introduction

In this thesis we investigate methods for accelerating (reducing the cost) associated with direct Monte Carlo simulations of dilute gases using novel ideas from the field of multiscale modeling and simulation. We begin the introduction with a discussion of Direct Simulation Monte Carlo (DSMC), which is the main ingredient of the present work.

DSMC was developed by G. A. Bird in 1963 [6] and has since become an industry standard used in a variety of software packages for modeling rarefied gas flows [5]. During the early years of DSMC, rarefied flows were of great interest in the context of space and reentry aerodynamics; in fact, this was the driving force behind its conception and has greatly contributed to its development and the rise in its popularity. More recently, DSMC has found considerable applications for modeling gas flows in micro-electronic mechanical systems (MEMS) [12]. More recent work has extended the use of DSMC to cover dense gases [2] and led to the development of variance-reduction methods that reduce the uncertainty associated with property evaluation in DSMC, which is inherently statistical. Variance-reduction methods are crucial in low-signal flows typical of micro and nanoscale applications [[4],[17]].

The Navier-Stokes description, typically referred to as the continuum description in the literature, is expected to fail as the characteristic flow length scale (L) approaches the molecular mean free path (λ). This is typically quantified via the Knudsen number, defined by

$$\mathrm{Kn} = \frac{\lambda}{L} \tag{1.1}$$

For the hard-sphere gas used in this study, the molecular mean free path is given by

$$\lambda = \frac{1}{\sqrt{2}\pi\sigma^2 n} \tag{1.2}$$

where n is the number density, or number of particles per volume and σ is the effective hard sphere diameter of a molecule.

Extensive literature studies [[8],[12],[7]] indicate that, typically, for $\mathrm{Kn} > 0.1$ the Navier-Stokes description with slip boundary conditions is no longer reliable and molecular or kinetic approaches are required. DSMC is one of the most efficient methods for providing solutions [[18],[1]] to the governing kinetic equation known as the Boltzmann equation [8]. In contrast to molecular dynamics, where individual molecular trajectories are numerically computed using Newton's equations of motion, the DSMC method integrates [18] the Boltzmann equation, in which the simplifications arising from the fact that the gas is dilute are already included [17].

In addition to taking advantage of the lack of strong interactions between molecules in a dilute gas, DSMC achieves great computational efficiency compared to molecular dynamics by taking advantage of the form of the Boltzmann equation and formulating time integration using a splitting scheme, in which each time step is split into an advection substep and a collision substep [1]. During the advection substep, the particles are moved (without collisions). During the collision substep, particles are sorted into small computational cells and collisions are stochastically generated between particle pairs within the same cell. By treating collisions stochastically, rather than calculating exact trajectories as in molecular dynamics, DSMC achieves further computational savings.

Despite the enormous computational savings compared to molecular dynamics methods, DSMC is still expensive, especially as the continuum limit is approached (molecular dynamics is prohibitively expensive in this limit). This is due to both the increased number of particles required as the $\mathsf{Kn} << 1$ ($\mathsf{L} >> \lambda$) limit is approached and the fact that larger length scales imply longer duration timescales.

In the present thesis we investigate the use of the gap-tooth method as a means of alleviating one of the above limitations, namely the increase in computational cost due to the large number of particles required to "fill" the computational domain. The gap-tooth method, originally proposed by Gear, Li, and Kevrekidis [10], falls under the broader category of patch dynamics, itself a subset of equation-free methods [15]. These methods provide avenues for using microscopic (e.g. molecular and, in the present case, kinetic) simulators to obtain solutions to macroscopic problems (in our context $\mathsf{Kn} << 1$), which is traditionally very difficult due to the computational limitations described above. Equation-free methods are further discussed below.

1.1 Equation-Free Methods: Introduction

Macroscopic problems are typically solved using continuum formulations, such as the Navier-Stokes description. These formulations are based on conservation laws, such as mass, momentum, and energy, coupled to models for the transport of these quantities, referred to as closures. The resulting models, typically in the form of partial differential equations describing property fields, are solved analytically or numerically. Although these approaches have been used successfully for centuries, the advent of more extreme conditions (e.g. response to laser irradiation, nanoscale devices and systems) leads to situations where the traditional closure assumptions are no longer valid. The traditional way of overcoming this difficulty is to develop new closure relations, using experiments, or more recently, using molecular dynamics simulations (or a kinetic simulation like DSMC, in the case of dilute gases).

The equation-free framework provides an alternative approach to this problem

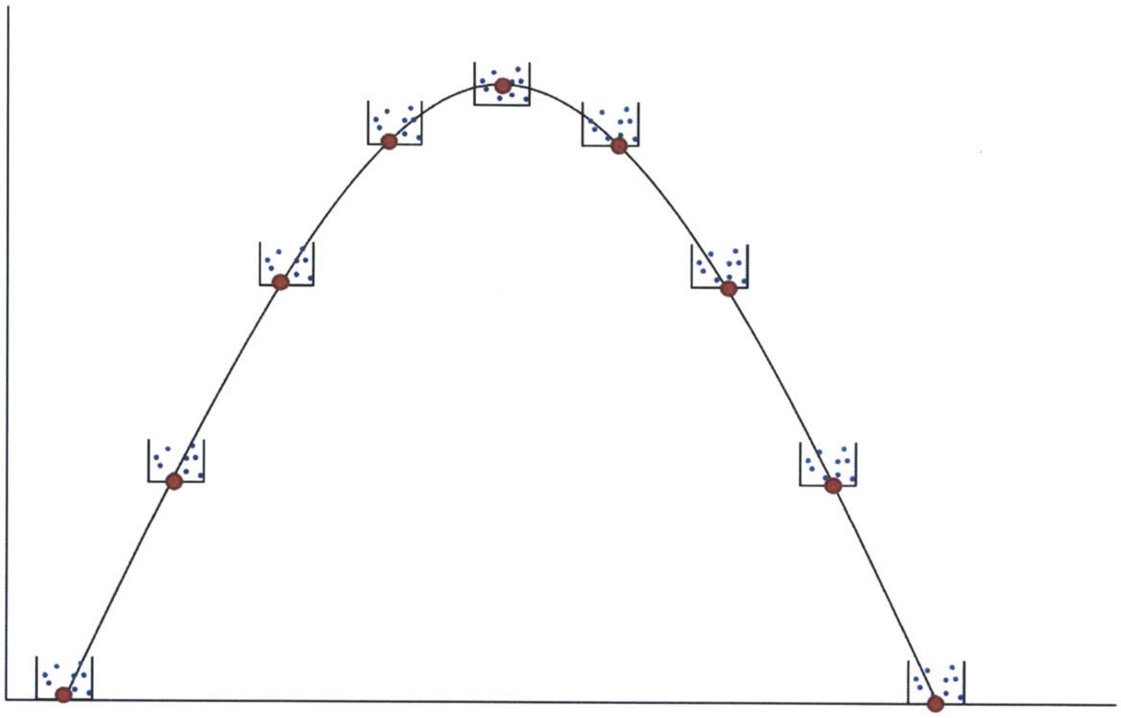


Figure 1-1: Example of patch dynamics. Molecular simulation is only performed within patches (shown here containing particles). Smooth line denotes the solution obtained by interpolating the solutions from the patches.

which sidesteps the need for writing down equations governing the system dynamics [15]. Instead, it uses the molecular simulation as a black box that can provide closure information. What differentiates this approach from a molecular simulation of the complete system is that in the patch-dynamics approach, the molecular simulator is used only in small patches of space, in analogy to the nodes of a numerical solution approach for the governing differential equation (see Figure 1-1). These patches are linked together through interpolation to allow representation of the extended system at the coarse level [16]. In this manner, the computational domain need not be filled with particles. Particles move from one patch to another, thus transferring information for a globally consistent solution.

This approach is expected to be valid provided that the solution field is smooth on the length scale of the patches and associated gaps and that appropriate dynamics are found describing the motion of the particles between patches [9]. In the case of the Boltzmann equation, the weakly-interacting nature of particles lends itself naturally to this approach. This is particularly the case for DSMC, in which, due to the splitting algorithm, particle motion is collisionless and thus completely non-interacting. As shown in section 2.1, this can be exploited to develop precise dynamics for the motion of particles as they move between different patches. These dynamics can be derived by writing down an interpolated from of the hydrodynamic fluxes associated with the particle motion on the numerical grid defined by the patches [13]. This particular case involving particle-flux interpolation between patches is known as the gap-tooth method [10].

1.2 Thesis Objective

In the present thesis we will implement a one-dimensional version of the gap-tooth scheme in which the microscopic solver is DSMC but it has been limited to patches, referred to here as teeth. The resulting simulation method will be referred to as GT-DSMC. As explained in the previous section, this scheme has the potential to provide significant computational savings in the limit $\mathsf{Kn} << 1$, where DSMC is expensive. However, it is currently unknown what the numerical error is associated with solving the governing equation in only a small fraction of the computational domain. As we discuss later, the particle dynamics that allows this process to be used amounts to an interpolation of the particle flux between teeth. One might expect this to be equivalent (in a numerical accuracy sense) to a first-order numerical solution scheme.

The major objective of the present thesis is to thoroughly investigate the error associated with the gap-tooth method, by comparing GT-DSMC solutions with corresponding, highly resolved, DSMC solutions. Another objective is the establishment of appropriate boundary conditions for the presence of diffuse boundaries (walls) in the system. Moreover, we also investigate the effect of the reduced number of particles in the simulation domain in the context of correlations and their effect on the resulting statistical uncertainties in the evaluation of hydrodynamic properties in GT-DSMC (in comparison with DSMC). This is important because if increased correlations between particles in GT-DSMC require longer sampling (or increased number of simulation particles) for achieving the same statistical uncertainty as DSMC, this directly negates some (or all) of the computational gain achieved by GT-DSMC.

In what follows, we describe the DSMC algorithm in more detail. We begin with a physical description of the flows that we will be simulating in this work and the associated model parameters. We then proceed to a more detailed description of the gap-tooth method.

Chapter 2

Numerical Methods

In this chapter, we discuss in detail the DSMC and GT-DSMC algorithms. A Matlab implementation of these algorithms can be found in Appendix A.

2.1 Direct Simulation Monte Carlo method

2.1.1 Overview

In this thesis we will limit our investigation to argon gas, using the hard-sphere model with an effective diameter $\sigma = 3.66 \text{ x } 10^{-10}$ m and molecular mass $m = 6.63 \text{ x } 10^{-26}$ kg. At the reference conditions of $P_0 = 1.013 \text{ x}10^5$ Pa and $T_0 = 273$ K, the number density, n_0, is equal to $2.685 \text{ x } 10^{25}$ m^{-3}, resulting in a mean free path, $\lambda = 6.2579 \text{ x } 10^{-8}$ m. We use two system characteristic lengths, $L = 1 \text{ x } 10^{-6}$ meters, resulting in $Kn = 0.0626$, and $L = 6.2579 \text{ x } 10^{-8}$ meters, resulting in $Kn = 1$. The first length value was chosen because it places the computations in the regime ($Kn << 1$) where kinetic descriptions typically become expensive to solve and techniques such as the Gap-tooth method investigated here are potentially useful. In particular, in the case of DSMC, the computational inefficiency is due to the large number of particles required; this is because for $Kn << 1$, we have $L >> \lambda$. The second length scale was chosen as it is firmly in the range where the continuum description fails.

Unless otherwise stated, our DSMC simulations used 241 cells with 500 particles per cell, for a maximum of 120,500 particles, and was run for 1 x 10^6 time steps of size

$$\Delta t = \frac{w}{4v_{mp}} \tag{2.1}$$

where w is the cell width and v_{mp} is the most probable speed, given by

$$v_{mp} = \sqrt{\frac{2kT_0}{m}} \tag{2.2}$$

In the above, k is the Boltzmann constant ($k = 1.38065 \text{ x } 10^{-23}$ JK^{-1}) and m is the molecular mass. This ensures that the time step, Δt, is a fraction of the mean free time (mean time between collisions). This is required to ensure that the approximation yielded by splitting each time step into distinct advection and collision phases is valid [11].

Correspondingly, the size of cells must be less than the mean free path, λ [3], in order to minimize the error associated with the stochastic processing of collisions. In our work, the cell size was always much smaller than the molecular mean free path since a larger number of cells was used.

When coupled with the requirement that several cells per mean free path are required for accurate solutions [3], this leads to large numbers of simulation particles. By reducing the total number of particles required for the simulation, the gap-tooth method has the potential to significantly extend the range of values of Kn that can be effectively simulated by DSMC.

In our implementation, particles are initially uniformly distributed within the system. Particle velocities are initialized from a Maxwellian distribution parametrized by the system's initial density, temperature and velocity. In our case, the initial condition was taken to be the reference condition stated above, namely $n_0 = 2.685$ x 10^{25} m^{-3}, $T_0 = 273$ K, and zero flow velocity.

The core DSMC algorithm consists of two main substeps, an advection substep, in

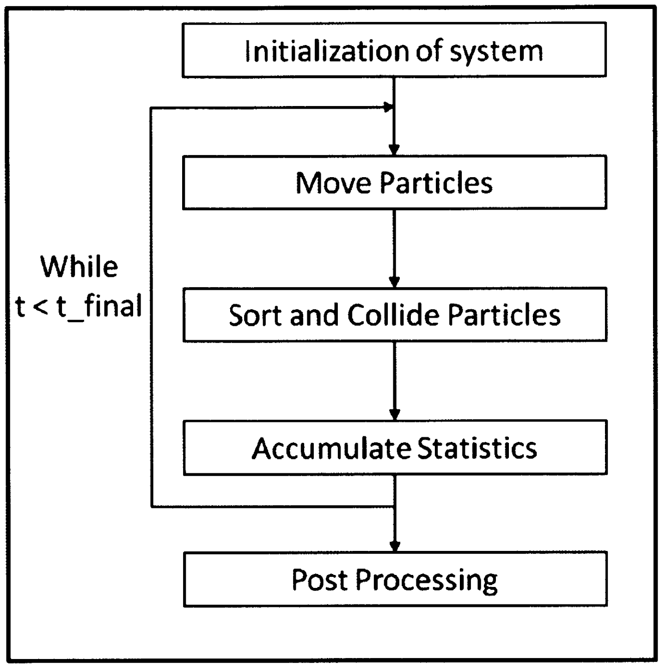


Figure 2-1: DSMC flowchart

which individual particles move without collisions, and a collision substep, in which a subset of the particles pairs within the same cell are selected to collide. Figure 2-1 shows a schematic of a typical algorithm flow. Both substeps are discussed in more detail below.

2.1.2 Advection Substep

The advection substep of the DSMC algorithm updates particle positions based on collisionless advection. For each particle, i, the new position, $\vec{r_i}$, is given by

$$\vec{r_i} = \vec{r_i} + \vec{v_i}\Delta t \tag{2.3}$$

Particles that reach the system boundaries need to be treated according to the boundary conditions prescribed at that location. In this work, we assume all system boundaries, namely at $x = 0$ and $x = L$, to be diffusively reflecting, as in the majority of recent studies [12].

For the particles encountering a boundary, given a point of impact with the wall, $\vec{r}_{wall}$, we calculate the time of flight to the boundary, Δt_{wall}, as follows:

$$\Delta t_{wall} = \frac{(\vec{r}_{wall} - \vec{r_i}) \cdot \hat{n}}{\vec{v_i} \cdot \hat{n}} \tag{2.4}$$

where $\hat{n}$ is the unit normal to the surface at the point r_{wall} and r_i is the particle position before the move that resulted in a boundary crossing.

In the case of diffusively reflecting walls, the components of particle velocity are reset according to a biased Maxwellian distribution that also needs to take into account the translational motion of the walls (if applicable) [1]. The component of molecular velocity normal to the wall is given by the distribution

$$P_\perp(v_\perp) = \frac{m}{kT_w} v_\perp e^{-\frac{mv_\perp^2}{2kT_w}} \tag{2.5}$$

where T_w is the wall temperature [1].

Similarly, each of the molecular velocity components parallel to the wall are described by the distribution [1]

$$P_\parallel = \sqrt{\frac{m}{2\pi kT_w}} e^{\frac{-mv_\parallel^2}{2kT_w}} \tag{2.6}$$

These distributions correspond to a non-moving wall. The case of a wall translating in a direction parallel to its plane can be treated by adding the wall velocity value, v_{wall}, to the molecular velocity drawn from distribution 2.6; the case of a wall moving normal to its plane is more complex and will not be treated here.

To draw molecular velocities from the above distributions, we proceed as follows: If we let z be a uniformly distributed random number in $[0, 1]$ and let x_g, y_g be random numbers that are Gaussian-distributed with zero mean and variance of unity, then the resultant expressions for the tangential and normal components of velocity after a particle strikes the wall are given by [1]

$$v_{\parallel} = \sqrt{\frac{kT_w}{m}} x_g \tag{2.7}$$

$$v_{\perp} = \sqrt{\frac{-2kT_w}{m} \ln(z)} \tag{2.8}$$

respectively. If, in particular, the wall moves in the y direction with velocity, v_{wall}, then

$$v_y = \sqrt{\frac{kT}{m}} y_g + v_{wall} \tag{2.9}$$

The final position of the particle is given by

$$\vec{r} = \vec{r}_{wall} + \vec{v}_{new} t_{after} \tag{2.10}$$

where t_{after} is given by

$$t_{after} = \Delta t - \Delta t_{wall} \tag{2.11}$$

with Δt_{wall} calculated using equation 2.4

2.1.3 Collision Substep

The collision substep employs a random selection process, following rules drawn from kinetic theory, which is used to choose a set of representative collisions to be processed at each time step. It begins by sorting the particles into cells, allowing particles to collide only if they fall within the same spatial cell, since this ensures that in a single time step nearby particles are more likely to collide than largely spatially separated particles.

The algorithm proceeds using the acceptance-rejection scheme outlined by A. Garcia in [1], randomly selecting particle pairs and accepting them as a collision pair if the ratio of their relative speed to the maximum relative particle speed in the cell of interest, $v_{r,max}$, is greater than some uniformly distributed random number, x, in $[0,1]$:

$$\frac{|\vec{v}_i - \vec{v}_j|}{v_{r,max}} > x \tag{2.12}$$

If the pair is selected, the particle velocities are reset. We repeat this process until we have processed M_{cand} collisions given by

$$M_{cand} = \frac{N_c^2 \pi \sigma^2 v_{r,max} N_e \Delta t}{2V_c} \tag{2.13}$$

where N_c is the number of particles per cell, V_c is the volume of the cell, and N_e is the the effective number (number of real physical molecules represented by each computational particle).

Because the acceptance-rejection procedure accepts collisions proportionally to the particle relative velocity, $< v_r >$, the actual number of collisions is related to the number of candidate collisions by the relation

$$\frac{M_{coll}}{M_{cand}} = \frac{< v_r >}{v_{r,max}} \tag{2.14}$$

which leads to a total number of collisions, M_{coll}, given by the following

$$M_{coll} = \frac{N_c^2 \pi \sigma^2 < v_r > N_e \Delta t}{2V_c} \tag{2.15}$$

as required by kinetic theory [1].

The velocity of the center of mass of the particle pair selected to undergo collision will be unchanged by the collision because of conservation of linear momentum. We denote the post collision velocity of the particles by v_i' and v_j', respectively, and the velocity of the center of mass of the particle pair system as v_{cm}. Then

$$\vec{v}_{cm} = \frac{1}{2}(\vec{v}_i + \vec{v}_j) = \frac{1}{2}(\vec{v}_i' + \vec{v}_j') = \vec{v}_{cm}' \tag{2.16}$$

Similarly, conservation of energy requires that the following relation hold for the relative velocities

$$\nu_r = |\vec{v}_i - \vec{v}_j| = |\vec{v}'_i - \vec{v}'_j| = \nu'_r \tag{2.17}$$

Then, for molecules of equal mass we can write

$$\begin{aligned} \vec{v}'_i &= \vec{v}'_{cm} + \frac{1}{2}\vec{v}'_r\hat{\mathbf{e}} \\ \vec{v}'_j &= \vec{v}'_{cm} - \frac{1}{2}\vec{v}'_r\hat{\mathbf{e}} \end{aligned} \tag{2.18}$$

where $\hat{\mathbf{e}}$ denotes a unit vector chosen randomly on the unit sphere since all directions are equally likely for the relative velocity of the particles post collision [1]. In other words,

$$\vec{v}'_r = \nu_r\left[(\sin\theta\cos\phi)\hat{\mathbf{x}} + (\sin\theta\sin\phi)\hat{\mathbf{y}} + (\cos\theta)\hat{\mathbf{z}}\right] \tag{2.19}$$

where ϕ is the azimuth angle, uniformly distributed in the interval $[0, 2\pi]$ and θ is the polar angle, distributed in the interval $[0, \pi]$ from the distribution $P(\theta)d\theta = \frac{1}{2}\sin\theta d\theta$. To generate values of ϕ and θ, we can write $\phi = 2\pi r_1$ and $\theta = \cos^{-1}(2r_2 - 1)$, where r_1, r_2 are uniformly distributed random numbers in the interval $[0, 1]$.

2.1.4 Sampling

As in all molecular simulation methods, hydrodynamic fields are obtained by collecting statistics of microscopic particle properties. These are averaged in time once steady state is reached, and if required, further averaged over a number of independent ensembles.

The properties of interest here, calculated for each cell of the system, are the number of particles, used in the calculation of average number density, velocity of particles, used to compute the mean flow velocity in each cell, and sum of the squares of the velocity of the particles, used to calculate the temperature in each cell.

2.2 Gap-Tooth DSMC Method

2.2.1 Overview

The gap-tooth direct simulation Monte Carlo (GT-DSMC) method that we develop here can be used to extend current molecular modeling techniques to macroscopic problems, which are currently at the limit (or beyond) of our computational capabilities due to the computational cost associated with molecular simulations of large systems. GT-DSMC combines a standard method for molecular modeling of rarified gaseous flows, DSMC, with the gap-tooth method, developed by Gear, Li, and Kevrekidis [10], to perform equation-free multi-scale modeling of systems using microscopic physics-based simulators.

The gap-tooth method is based on the idea that sufficiently smooth problems on a macroscale domain can be accurately solved using molecular methods within small (microscale) domains that "span" but not completely fill the space and that can subsequently be interpolated to obtain a macroscale solution [13]. This alleviates the need to use the molecular description over the full computational domain, which is computationally prohibitive, thus allowing us to solve large scale problems at significantly reduced costs. The difficulty associated with this method lies in the development of an accurate interpolation scheme that moves particles between the microscale domains in a way that captures the underlying physics of the problem.

We implemented a gap-tooth direct simulation Monte Carlo (GT-DSMC) method as described below within Matlab, then set up several model problems on which we could test this formulation. For larger scale problems, it would be advisable to move to a more computationally efficient coding language; however, for the purposes of this thesis, the performance of the Matlab code was sufficient. Vectorization of the Matlab code allowed significant speed up in long runs. Additionally, running the code in parallel for each simulation allowed us to complete simulations in an efficient manner.

We applied the GT-DSMC scheme to a test problem that involves coupled momentum and energy transfer, comparing our results with DSMC solutions. DSMC is known to produce accurate solutions to the Boltzmann equation [18]; we will show that using GT-DSMC, we were able to obtain solutions that match those of the DSMC method, albeit with some numerical error.

2.2.2 Algorithm Details

Given a macroscale domain of length L, we divide the domain into n equally sized cells of width h, as in the DSMC method. In contrast to DSMC, only a subset of these cells will contain particles; let us call these teeth. The remaining cells not containing particles form gaps between the teeth. In this implementation, we ensure that the cells next to the physical domain boundaries are teeth (filled with particles).

The teeth in our GT-DSMC can be thought of as grid points for a finite difference scheme and the solution for the overall system will be an interpolated form of the values (microscale solutions) at these grid points. By imposing appropriate boundary conditions at the tooth level dictating the motion of particles between teeth and suitable boundary conditions to the external boundaries of the system, we can then evolve the molecular (DSMC) description in time, arriving at a globally consistent solution. We derive the scheme for handling particle motion between teeth internal to the system from Gear, Li, and Kevrikidis [10]. Boundary conditions, however, have not been treated before and are treated in this thesis for the first time.

We let F be the hydrodynamic flux between tooth boundaries. Each tooth has right-going and left-going fluxes, both outgoing and incoming, for each boundary of the tooth. In D-dimensions, we will have 2^D boundaries to deal with and the corresponding fluxes. For the purposes of the following discussion, we will label fluxes as follows: $F^i_{R,O}$ is the right, outgoing flux from tooth i, and similarly $F^i_{L,I}$ is the left moving, incoming flux to tooth i. The fluxes associated with a single tooth, i, can be seen in figure 2-2.

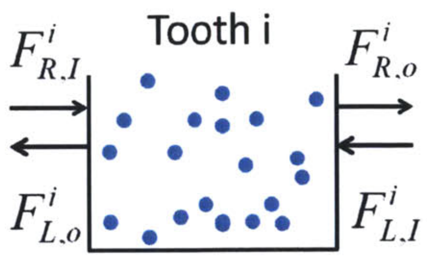


Figure 2-2: Diagram of incoming and outgoing, left-going and right-going fluxes for a single tooth, i, of the system.

Let tooth i be centered a position x and let all teeth be equally sized with width h. Let the centers of the teeth be separated by a distance Δx (see figure 2-3). If we are interested in the right-going flux leaving tooth i through the right hand boundary, given by $F^i_{R,o}$, we can express this as a linear combination of the incoming, right-going flux to the left hand boundary of tooth i, $F^i_{R,I}$, and the incoming, right-going flux at the left boundary of tooth $i+1$, $F^{i+1}_{R,I}$, as follows

$$F^i_{R,o} = (\frac{\Delta x - h}{\Delta x})F^i_{R,I} + (\frac{h}{\Delta x})F^{i+1}_{R,I} \tag{2.20}$$

Then, if we let $\alpha = \frac{h}{\Delta x}$, we can rewrite the above as

$$F^i_{R,o} = (1-\alpha)F^i_{R,I} + \alpha F^{i+1}_{R,I} \tag{2.21}$$

Note that the interpolation coefficients are derived from the geometry of the particular gap-tooth setup. In particular, $\Delta x - h$ is the physical size of the gap. In our results in the following chapter, we label the results in terms of relative gap size, G

$$G = \frac{1}{\alpha} - 1 \tag{2.22}$$

The most important part of the gap-tooth scheme, namely the algorithm for particle motion between teeth, can be derived by interpreting equation 2.21 stochastically

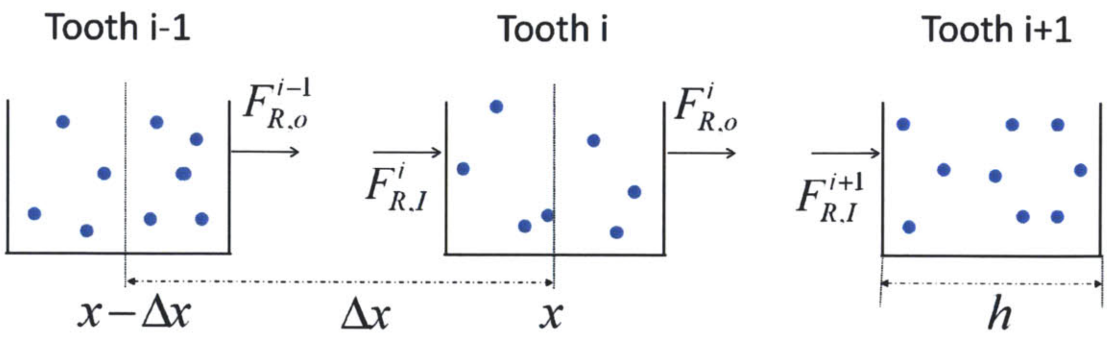


Figure 2-3: Diagram of three teeth internal to the system, with right-going fluxes labelled, and system spacing displayed.

as follows: if a particle is leaving tooth i, moving right, then with probability α, the particle will enter tooth $i+1$ via the left boundary, and with probability $(1-\alpha)$, the particle will instead "loop around" and be redirected to enter the same cell via its left boundary. This process can easily be extended to calculation of left-going fluxes.

Following the example of Gear, Li, and Kevrekidis [10], we will refer to this process of particle movement as flux redistribution, rather than flux interpolation. It is worth noting that the case $G = 0$ ($\Delta x = h$) corresponds to a "pure" DSMC calculation. In this case, $\alpha = 1$ in equation 2.21, so all particles moving to the right enter tooth $i+1$ and no particles are redirected into tooth i by way of its left hand boundary. In this limit any interpolation error vanishes. We will make use of this fact later in our comparison of GT-DSMC to the DSMC results, to make sure that our treatment of external boundaries is accurate.

If we move a particle a total distance $dx_{tot} = v\Delta t$ during the time step Δt, then we can express the total distance traveled as a combination of the distance that the particle moves inside of the tooth to reach the boundary, dx_{in}, and the remaining distance, dx_{out}. If dx_{out} is less then the tooth width, h, than we insert the particle into the receiving tooth a distance dx_{out}; however, if $dx_{out} > h$, the particle passes straight through the receiving tooth and undergoes the flux redistribution process

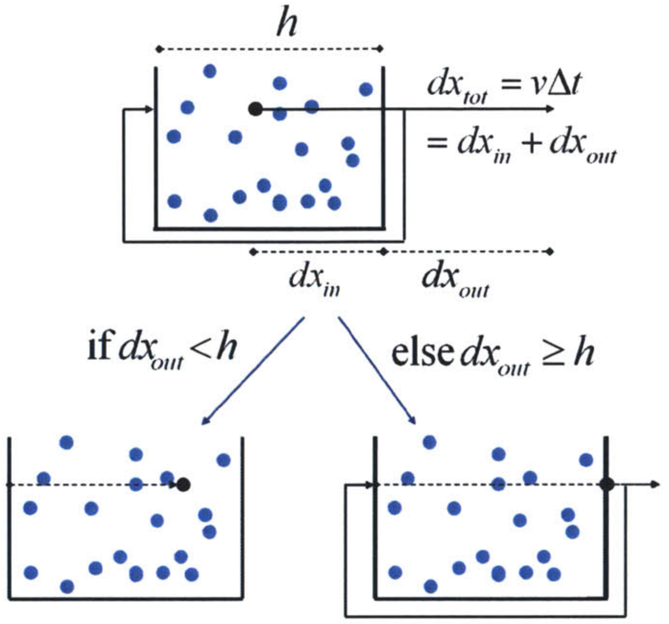


Figure 2-4: Diagram of recursive internal tooth particle redistribution scheme.

again (see figure 2-4). In fact, the flux redistribution is a recursive process, in order to handle any value of $\mathsf{dx_{out}}$.

2.2.3 Boundary Treatment

Teeth adjacent to system physical boundaries need special treatment that amounts to imposition of the external boundary conditions on the gap-tooth simulation. We divide those into two cases, namely, the case of flux towards a boundary and the case of flux away from a boundary. Those are show schematically in figures 2-5 and 2-6 for the case of the left system boundary. The case shown in figure 2-5 can be treated as imposing a diffuse reflection on all particles corresponding to flux $\mathsf{F^1_{L,o}}$; the case shown in figure 2-6 is more subtle.

The subtlety arises from the need to treat particles reentering the tooth (particles with flux $\mathsf{F^1_{R,I}}$ in figure 2-6) using the diffuse boundary conditions, as if they are re-emitted after collision with the wall. Simulations without this subtle treatment of

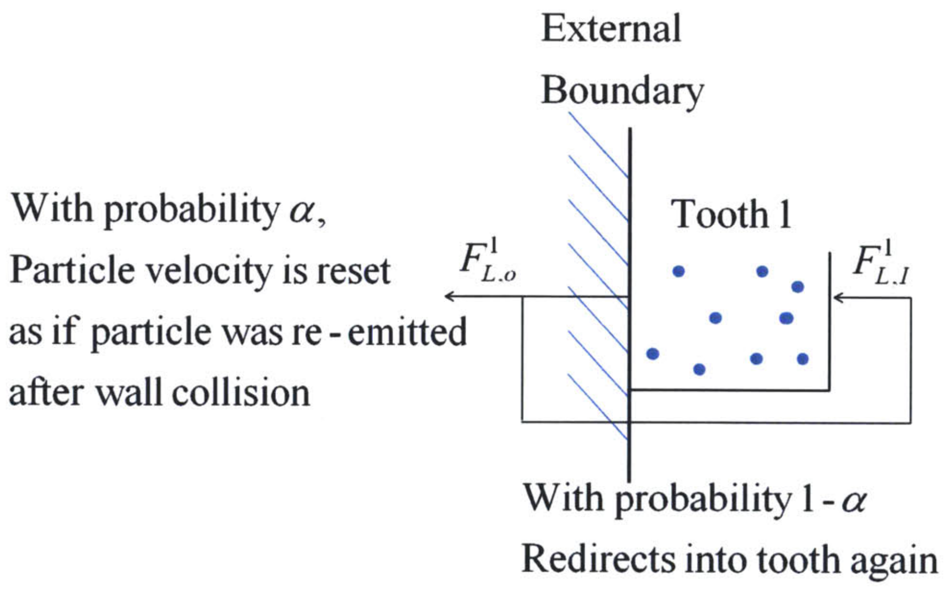

External Boundary
With probability α,
Particle velocity is reset
as if particle was re-emitted
after wall collision
$F^1_{L.o}$
Tooth 1
$F^1_{L.I}$
With probability $1-\alpha$
Redirects into tooth again

Figure 2-5: External System Boundary Treatment of Fluxes towards a Boundary (here, the left)

particles gave inferior results (see section 3.2). Henceforth, we refer to the former (without the additional, subtle treatment) as implementation I and refer to the latter as implementation II.

In the following chapter we apply the GT-DSMC method, for both of the implementations discussed directly, above to a model problem, comparing results with those of DSMC and showing how error scales with gap size.

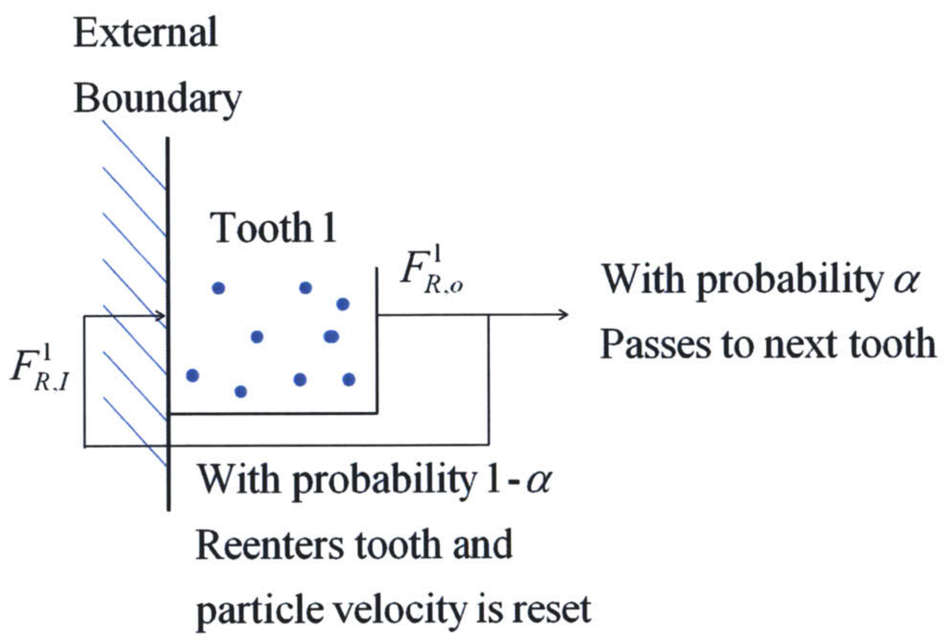


Figure 2-6: External System Boundary Treatment of Fluxes away from a Boundary (here, the left)

Chapter 3

Model Problem - Couette Flow

In this chapter, we discuss the application of the GT-DSMC to a 1-dimensional Couette flow of gaseous argon between two moving thermal plates. We will show results and error levels for a problem with $\mathsf{Kn} = 0.0626$ and with $\mathsf{Kn} = 1$ and for both of the implementations discussed in section 2.2.3, comparing the results with DSMC simulation results.

3.1 Problem Setup

Previous implementations of the gap-tooth method for molecular simulations have had periodic boundary conditions, eliminating the challenges associated with proper treatment of flux interpolation at the boundaries of the system. Here we implement the non-periodic system shown in figure 3-1, with thermal walls at temperatures of 273 K and moving at a velocity of Mach 1 (337 meters per second). The left wall of the system moves in the negative y direction and the right wall moves in the positive y direction. The position of particles in the third (z) direction is not tracked (although their velocity is).

We use 500 particles per tooth, which amounts to a total of $500(\frac{N-1}{G}+1)$ particles in the system where N is the total number of cells in the system and G is the relative

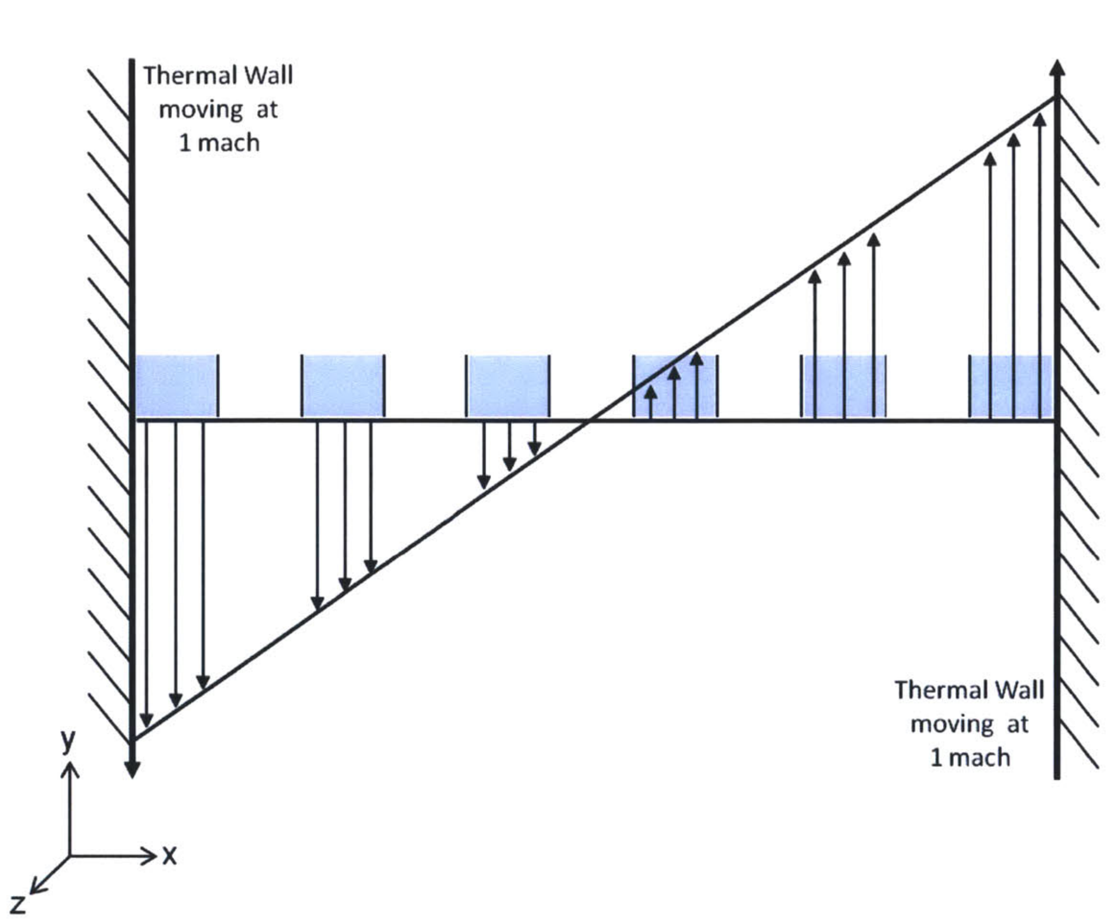


Figure 3-1: Diagram of Couette Flow Model Problem

gap size, given by equation 2.22. Each particle represents a varying number of atoms, which depends on the problem and simulation parameters. In the $\mathsf{Kn} = 1$ case, we have less then 1 atom per particle. In contrast, in the $\mathsf{Kn} = 0.0626$ case, we have around 222 atoms per particle.

We discretize the domain, using 241 equal sized cells, and run simulations for relative gap sizes, G, of 0, 1, 3, 9, and, where appropriate, 39.

We run each simulation for 1 x 10^6 time steps. Samples are collected during the latter 90% of this period, a sufficient period of time for the system to reach an equilibrium state, after which we begin statistical sampling. We calculate time-averaged flow velocity, time-averaged temperature, and time-averaged number density in each cell of the domain from the accumulated statistics.

3.2 Results

3.2.1 Implementation I versus II

In this section we compare the results from implementation I and II to DSMC solutions. We will begin by discussing the results for number density, temperature, and flow velocity in the y-direction as a function of spatial position for implementation I. Looking at figures 3-2A and 3-3A, we notice that the number density values near the walls of the system are low, for both systems, regardless of Knudsen number. The error (compared with the DSMC solution) increases as the gap increases. Figures 3-2B and 3-3B show increasing discrepancies across the system in the temperature values as the gap size increases, while figures 3-2C and 3-3C show discrepancies in the flow velocity close to the wall with increasing gap size.

In contrast, figures 3-4A and 3-5A show that implementation II captures the number density profiles near the walls of the system more accurately. The same holds for the temperature profiles near the walls (see figures 3-4B and 3-5B). It is most apparent with the considerable improvement in the flow profiles (see figures 3-4C and 3-5C compared with figures 3-2C and 3-3C, respectively). It is worth noting that there are still visible differences between the DSMC profile and the GT-DSMC profiles, in the temperature profiles in particular. We believe that this is due to the interpolation scheme that has been implemented to move particles within the internal (bulk) cells of the system, which is, after all, only first-order accurate.

Figures 3-6, 3-7, 3-8, and 3-9 show the same results in terms of the fractional error calculated by treating the DSMC solution as exact. The fractional error of property $X(x)$ was defined as

$$\mathrm{err}(x) = \frac{X(x) - X_{DSMC}(x)}{\tilde{x}} \tag{3.1}$$

where $X_{DSMC}(x)$ is the DSMC value at the same location in space and $\tilde{x}$ is a characteristic magnitude. For all properties, $\tilde{x}$ is taken to be the magnitude of the variation of that quantity across the domain ($|X_{max} - X_{min}|$).

Comparing the errors seen in figuress 3-6 and 3-7 to those of figures 3-8 and 3-9, respectively, shows that implementation II features smaller error levels at the boundaries with some moderate error in the bulk which is consistent with the existence of numerical error due to the gaps. In contrast, implementation I exhibits large errors at the boundaries, suggesting that the boundary conditions are not implemented accurately.

3.2.2 Verification of GT-DSMC for G = 0

Figure 3-10 shows a comparison between DSMC and implementation II GT-DSMC with $G = 0$ for $Kn = 0.0626$. As expected, the difference between the two is random. This further validates implementation II, as no deterministic error is discernible.

3.2.3 Further Investigation of Implementation I

Despite large errors close to the boundaries, implementation I exhibited smaller errors in the temperature values in the bulk than implementation II. This was not because implementation I is more accurate than implementation II in the bulk as their treatment of bulk cells is identical. It is a fortuitous coincidence that the increased boundary slip at the walls present in implementation I leads to a reduced velocity gradient and thus reduced viscous heating, which cancels some of the numerical error associated with the gap-tooth interpolation in the bulk.

To demonstrate this, we run a series of implementation I simulations, of the type presented in section 3.1, but in which the velocity of the walls was artificially increased such that the gas velocity was equal to the value obtained by implementation II GT-DSMC for the same value of G. The rationale behind these investigations is that by eliminating the additional slip velocity associated with implementation I, we can focus on the "true" effect of this implementation on the temperature and compare it to that of implementation II.

Figure 3-11A shows the results for the temperature and figure 3-11B shows the normalized error. These confirm that, when corrected for the additional slip, implementation I exhibits more error in resultant temperature. We see this by comparing those results with the results shown in figures 3-4B and 3-8B.

3.3 Computational Considerations

3.3.1 Computational Run Time versus Gap Size

Figure 3-12 shows how computation time scales with gap size for the GT-DSMC method. Note that $G + 1 = 1$ corresponds to a DSMC simulation. The results correspond to GT-DSMC implementation II, applied to the Couette shearing problem, with 500 particles per cell, run for 1e6 time steps. We see that the computational run time scales linearly with the gap size. This is as expected, as the number of particles simulated scales linearly with gap size and the run time is directly dependent on the number of particles simulated. In other words, the potential for computational savings is enormous, as one would expect. The previous statement, however, neglects the additional cost due to increased statistical uncertainty in the GT-DSMC results due to increased statistical correlations. This is quantified in the following section.

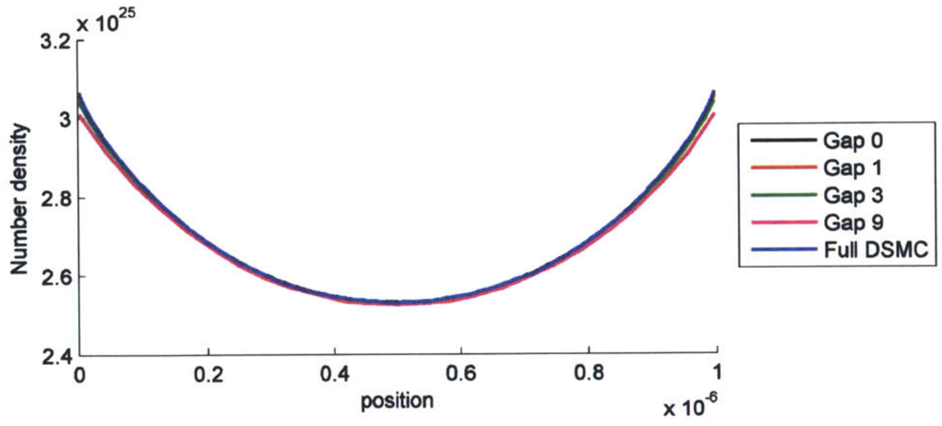
3.3.2 Variance in Results versus Gap Size

Although still a Monte Carlo method that converges as $\beta/\sqrt{n}$, the G-DSMC method is expected to converge more slowly, i. e. $\beta'/\sqrt{n}$ with $\beta' > \beta$, since particles are continuously recycled (reenter teeth), leading to less renewal and increased correlations.

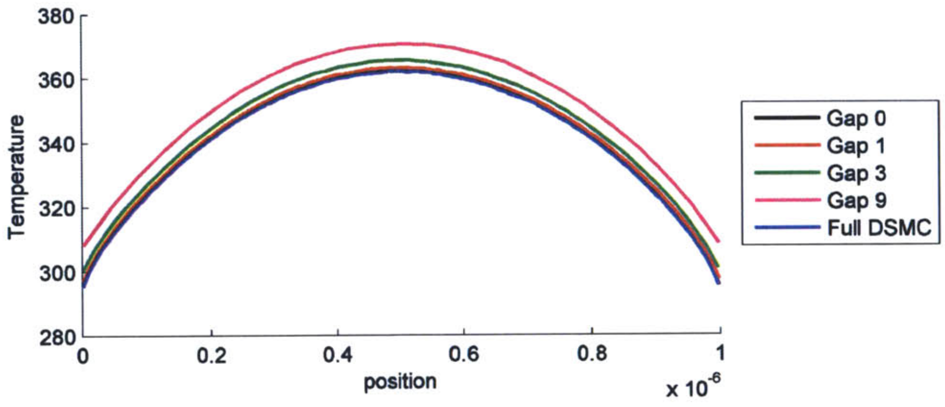
Here we attempt to characterize this phenomenon by plotting variance as a function of gap size. The variance was measured across a 50 ensemble calculation using the GT-DSMC method, implementation II, applied to a system in equilibrium. As before, we used 500 particles per cell and $\mathsf{Kn} = 0.0626$, running each of the ensembles for 1 x 10^5 time steps. Figure 3-13 shows the measured variance in temperature as a function of $\mathsf{G} + 1$.

The variance clearly increases with gap size, as expected. This implies that in order to retain the same statistical uncertainty as a DSMC simulation ($\mathsf{G} + 1 = 1$), simulations with larger gap sizes require longer sampling or an increased number of particles, reducing the overall effectiveness of the GT-DSMC method. In fact, from the figures, we see that for a relative number of particle reduction of 39 ($\mathsf{G} + 1 = 40$), the variance increase is on the order of 20, thus negating the vast majority of the computational benefit.

A.) Number Density



B.) Temperature



C.) Flow Velocity

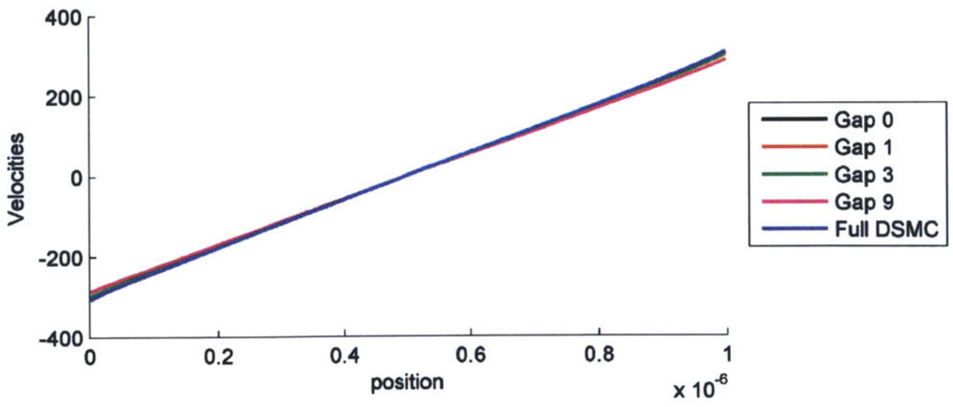
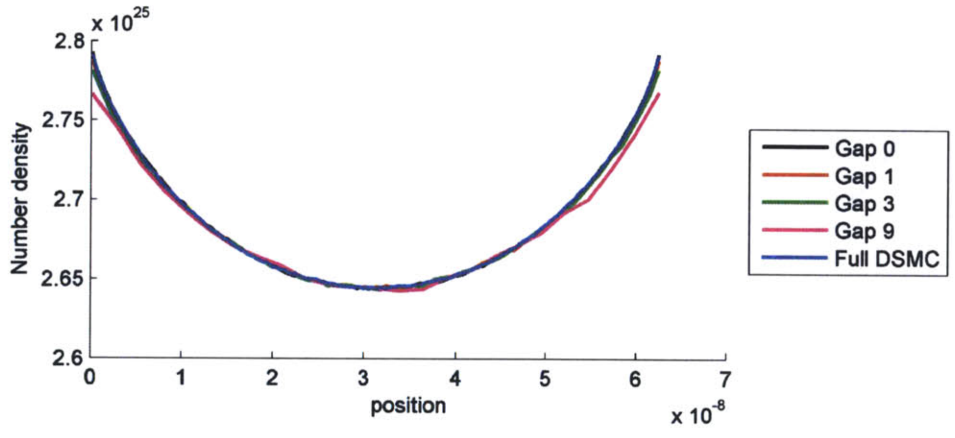
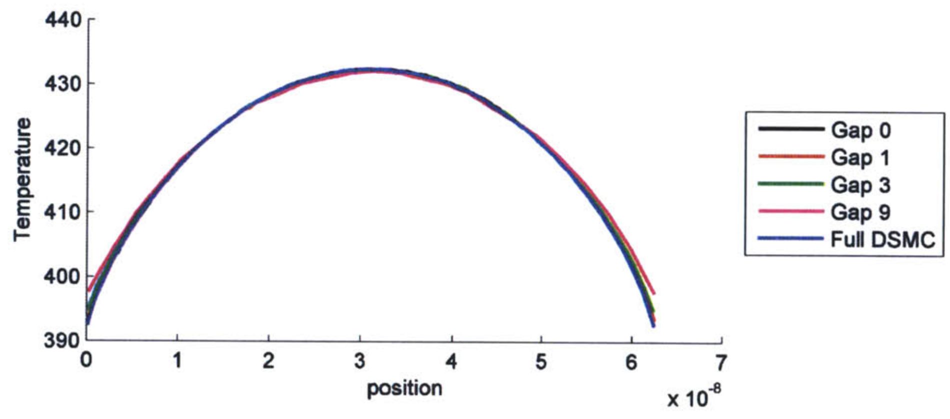


Figure 3-2: Implementation I Results versus Spatial Postion for $\mathsf{Kn} = 0.0626$

A.) Number Density



B.) Temperature



C.) Flow Velocity

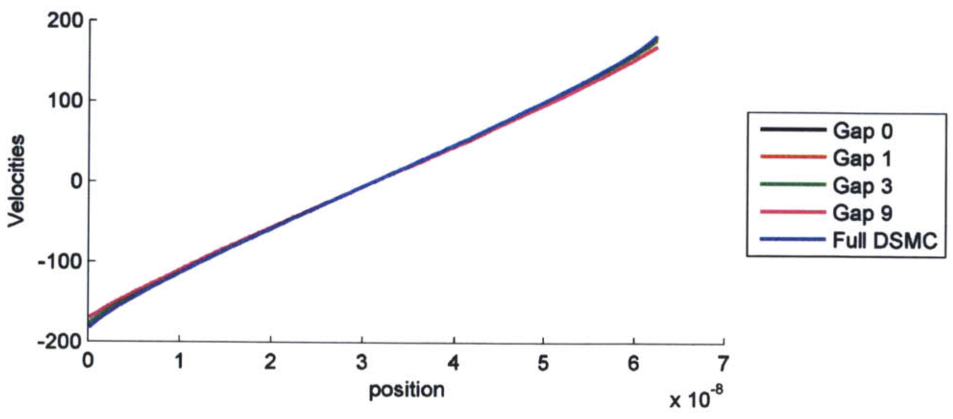
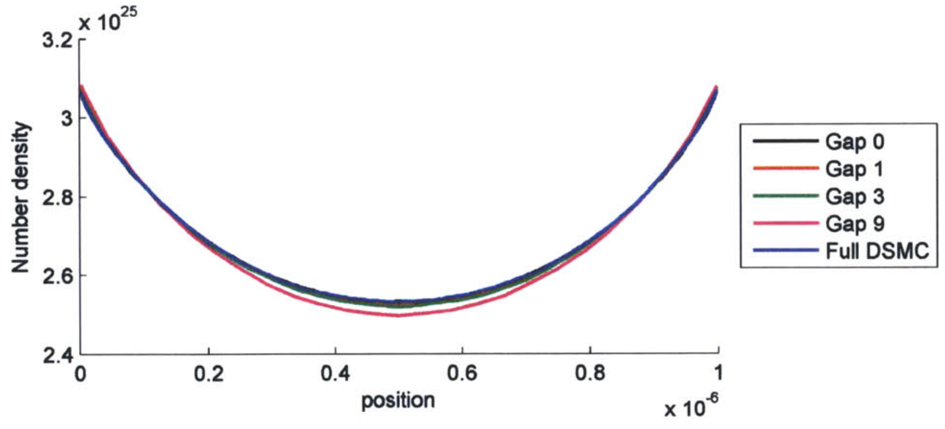
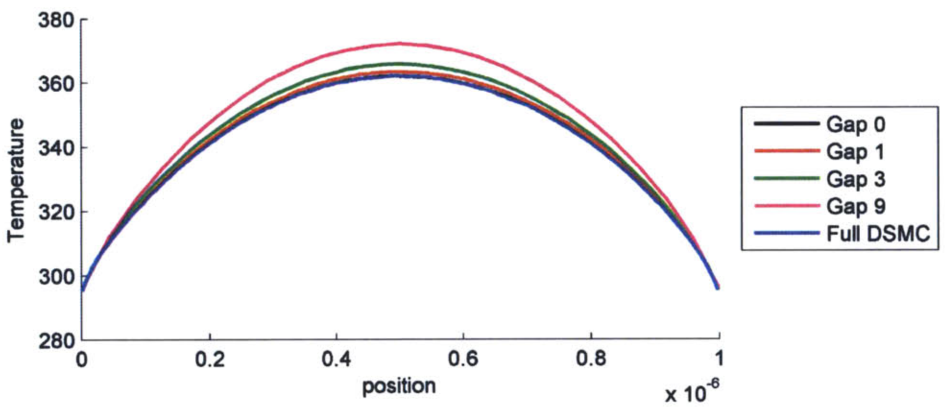


Figure 3-3: Implementation I Results versus Spatial Postion for $\mathsf{Kn} = 1$

A.) Number Density



B.) Temperature



C.) Flow Velocity

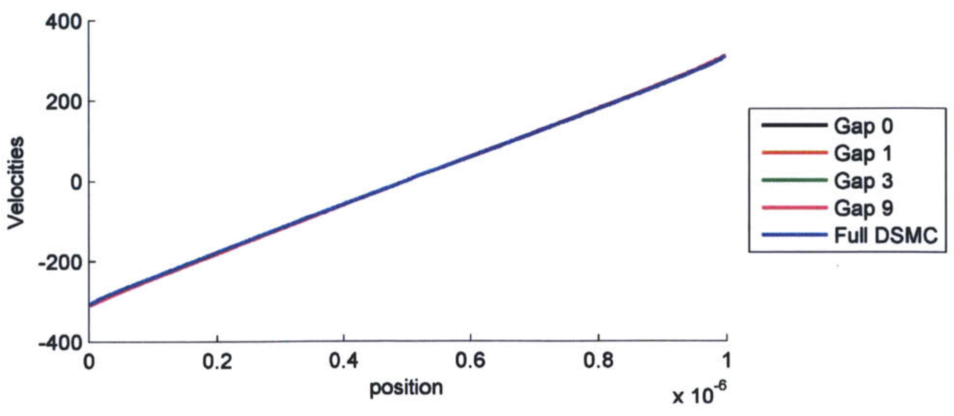
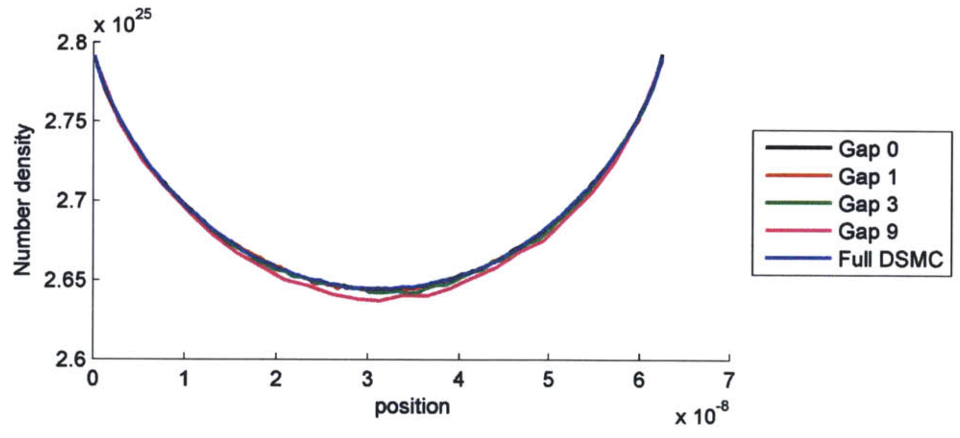
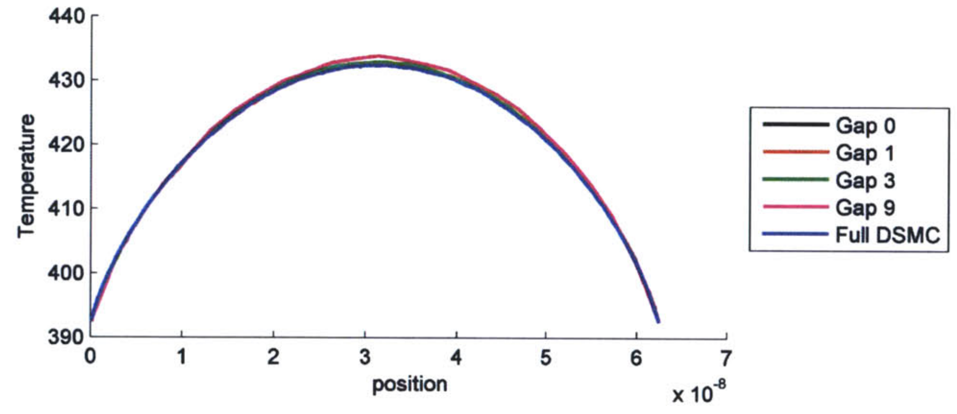


Figure 3-4: Implementation II Results versus Spatial Postion for $\mathsf{Kn} = 0.0626$

A.) Number Density



B.) Temperature



C.) Flow Velocity

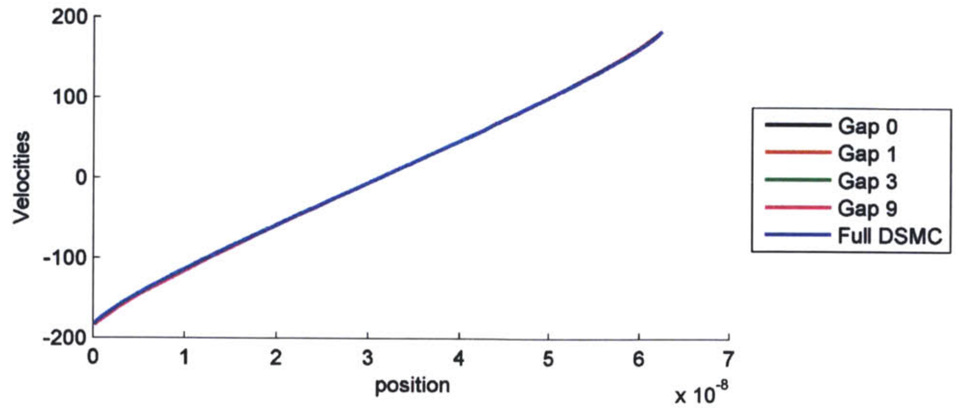
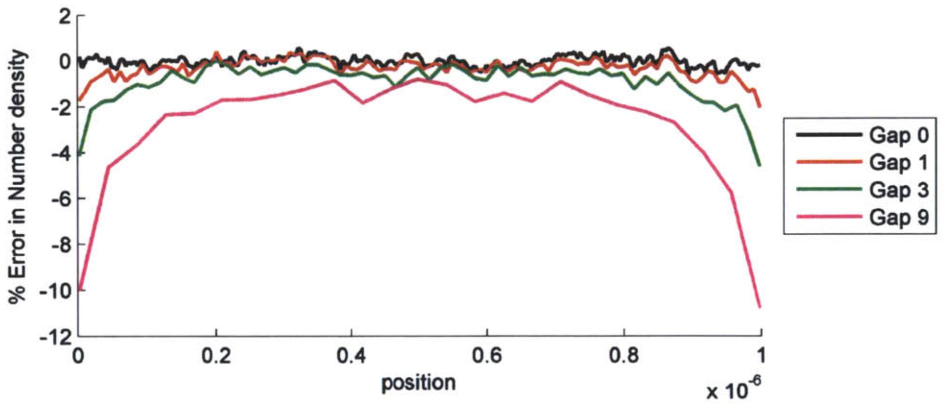
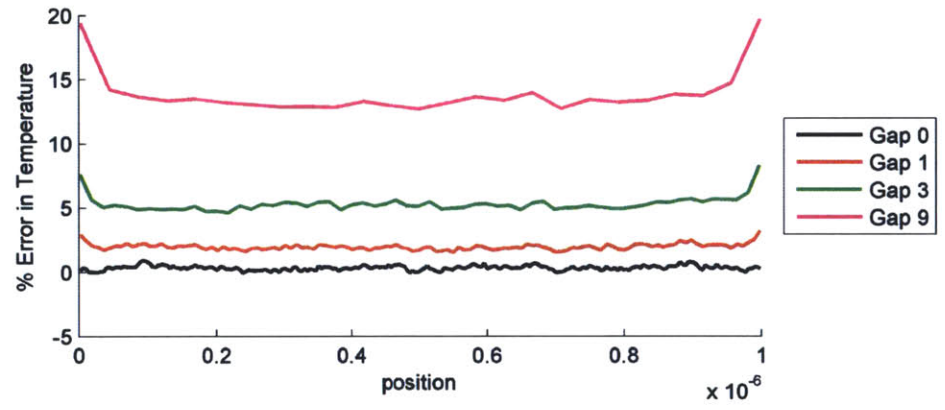


Figure 3-5: Implementation II Results versus Spatial Postion for $\mathrm{Kn} = 1$

A.) Error in Number Density



B.) Error in Temperature



C.) Error in Flow Velocity

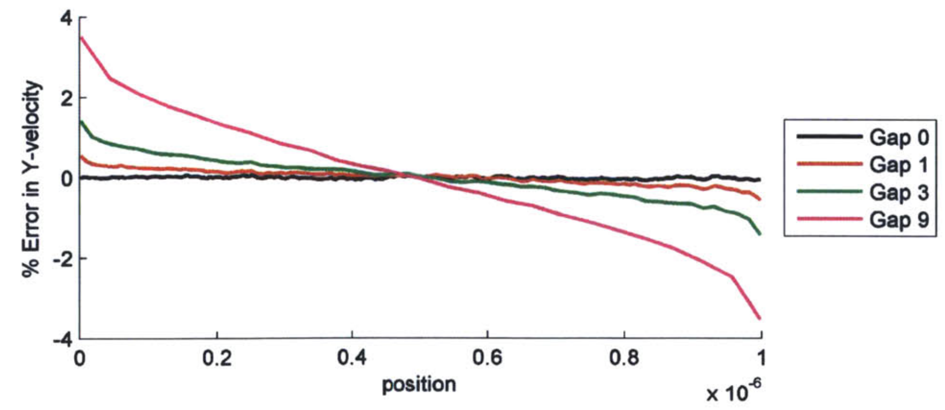
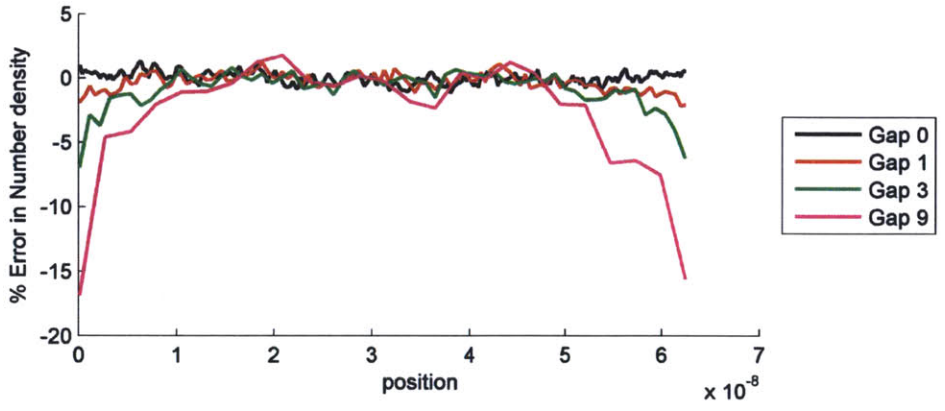
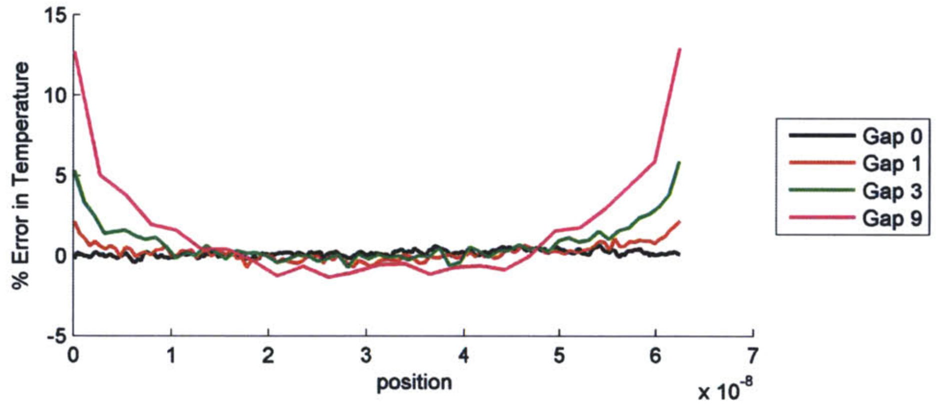


Figure 3-6: Implementation I Error versus Spatial Postion for $\mathsf{Kn} = 0.0626$

A.) Error in Number Density



B.) Error in Temperature



C.) Error in Flow Velocity

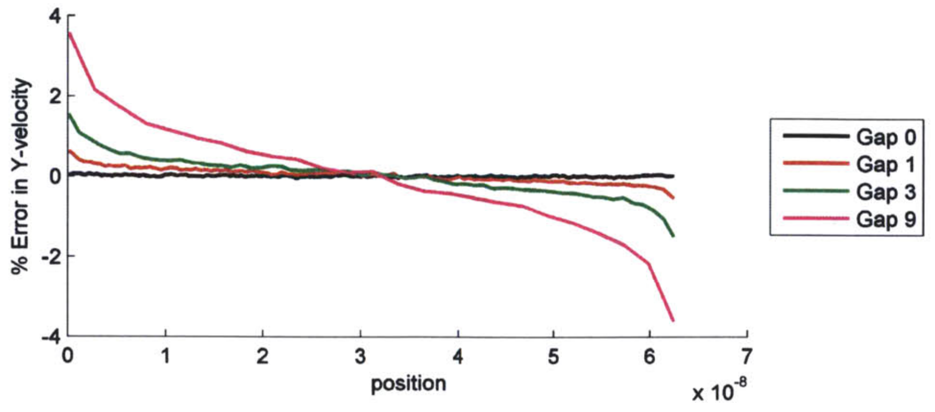
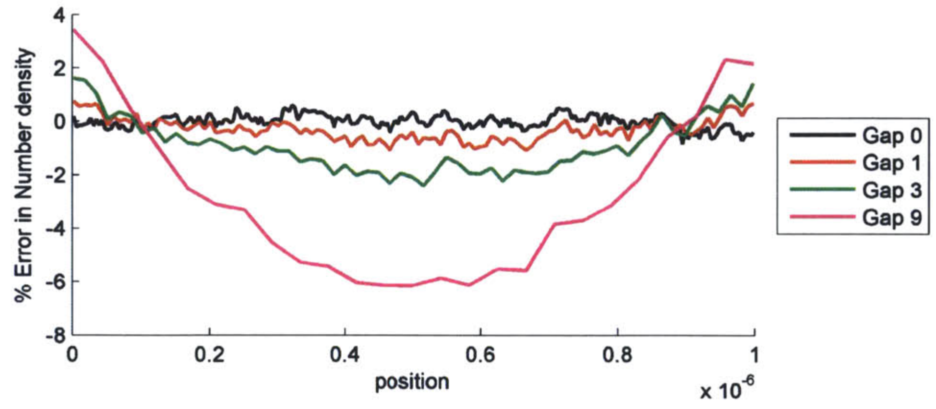
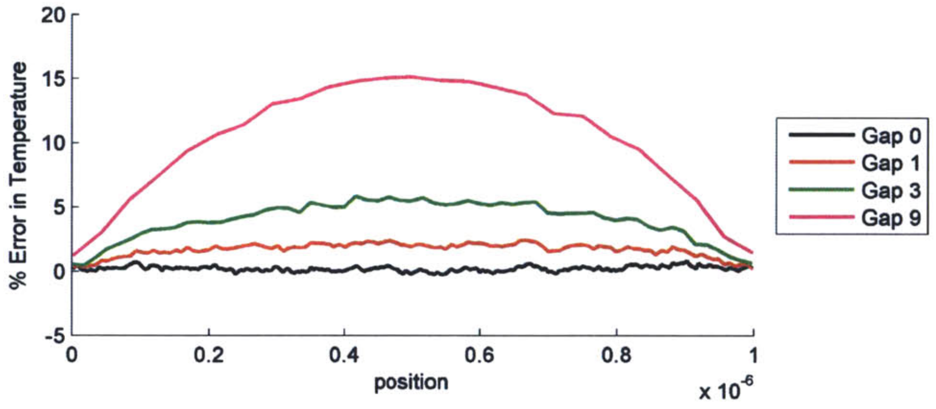


Figure 3-7: Implementation I Error versus Spatial Postion for $\mathsf{Kn} = 1$

A.) Error in Number Density



B.) Error in Temperature



C.) Error in Flow Velocity

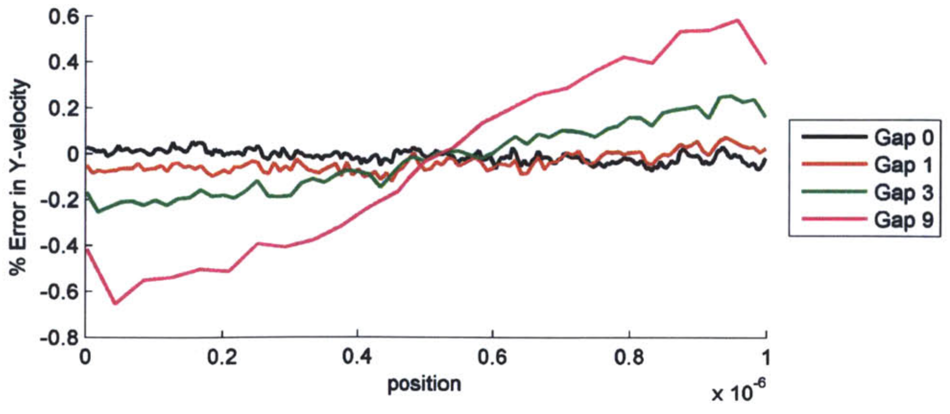
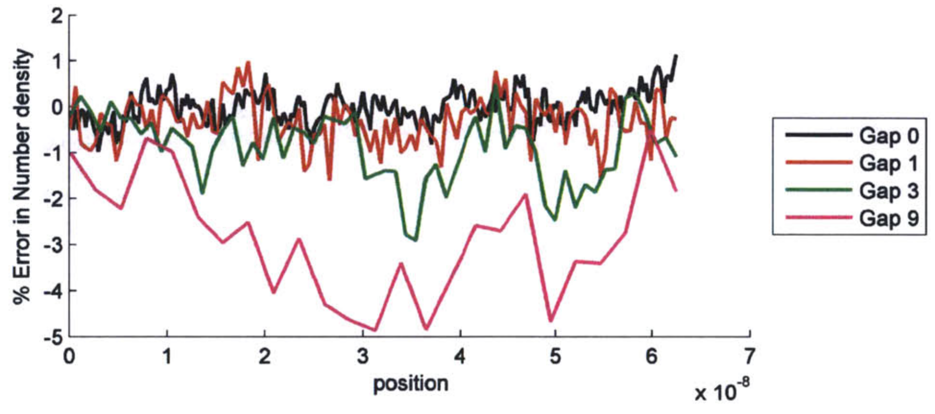
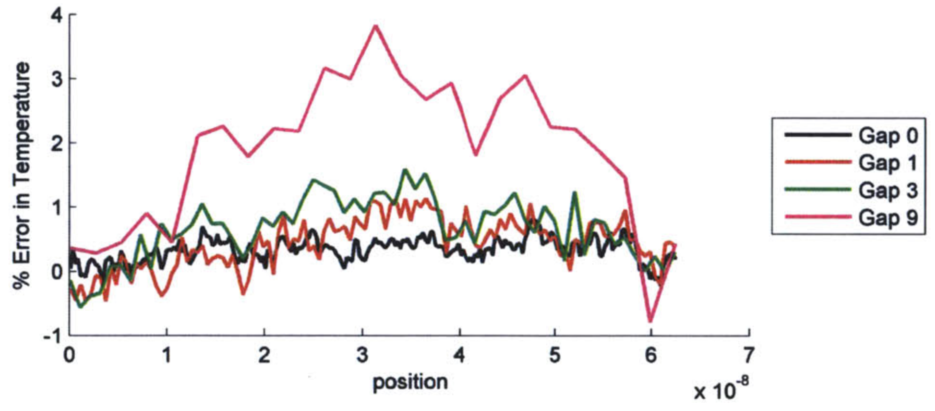


Figure 3-8: Implementation II Error versus Spatial Postion for $\mathsf{Kn} = 0.0626$

A.) Error in Number Density



B.) Error in Temperature



C.) Error in Flow Velocity

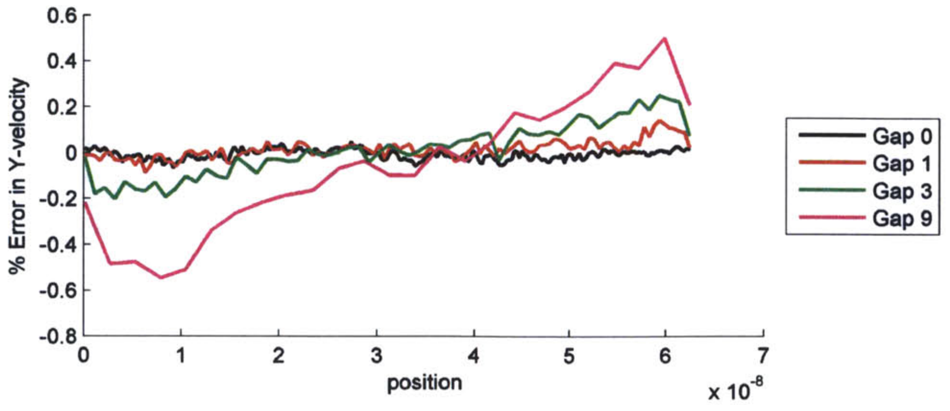
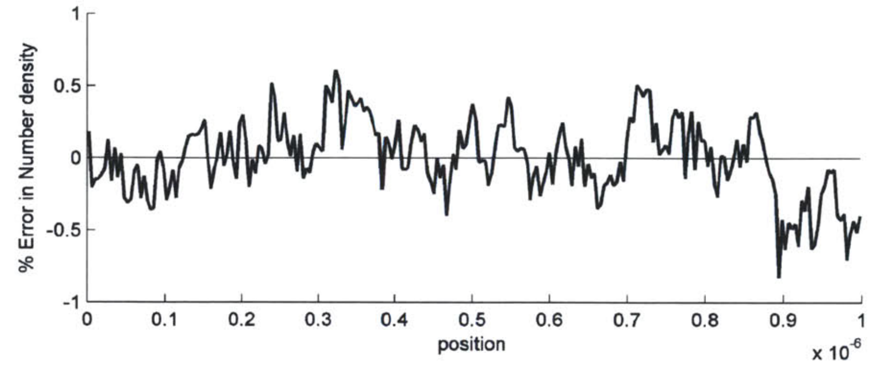
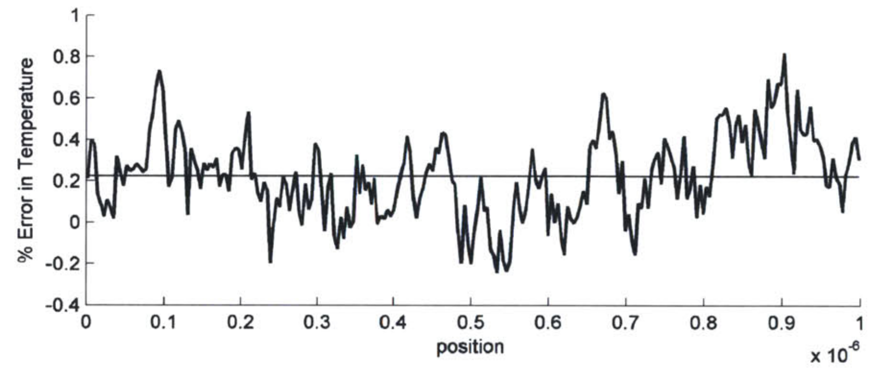


Figure 3-9: Implementation II Error versus Spatial Postion for $\mathsf{Kn} = 1$

A.) Error in Number Density



B.) Error in Temperature



C.) Error in Flow Velocity in the Y-direction

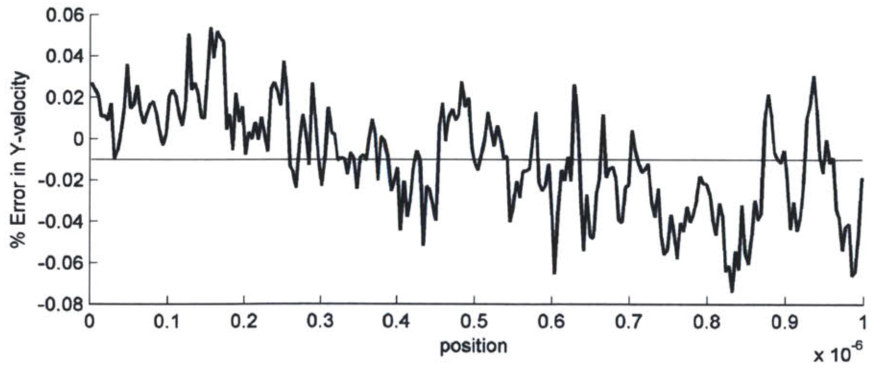
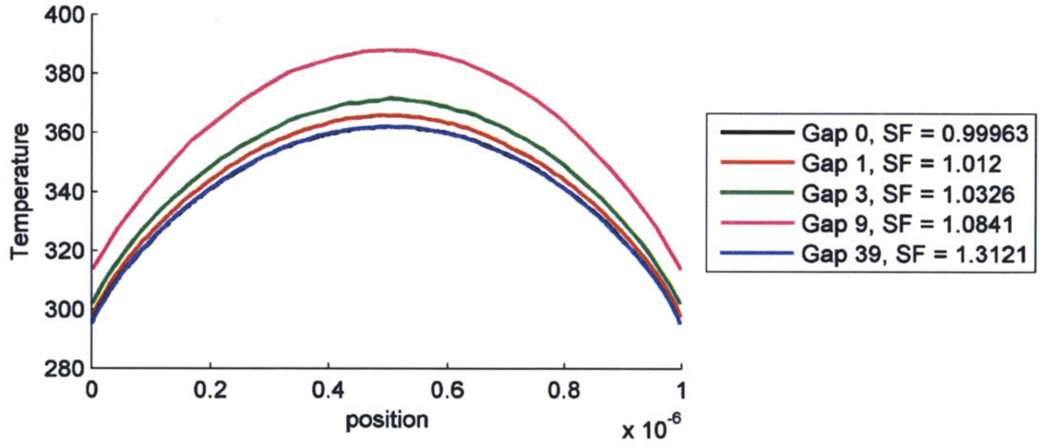


Figure 3-10: Error versus Spatial Position for implementation II GT-DSMC with G = 0 vs. DSMC for Kn = 0.0626

A) Temperature



B) Error in Temperature

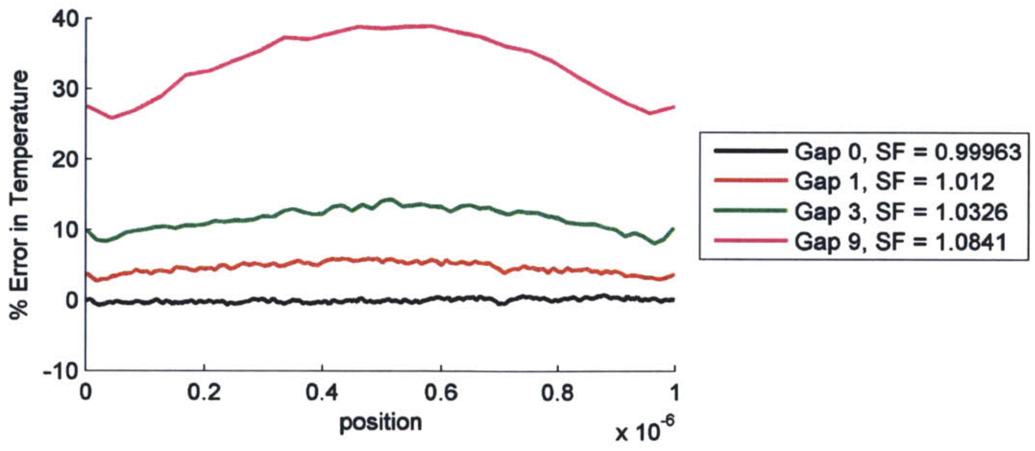


Figure 3-11: Temperature and error in temperature versus spatial position for scaled implementation I GT-DSMC for $\mathsf{Kn} = 0.0626$

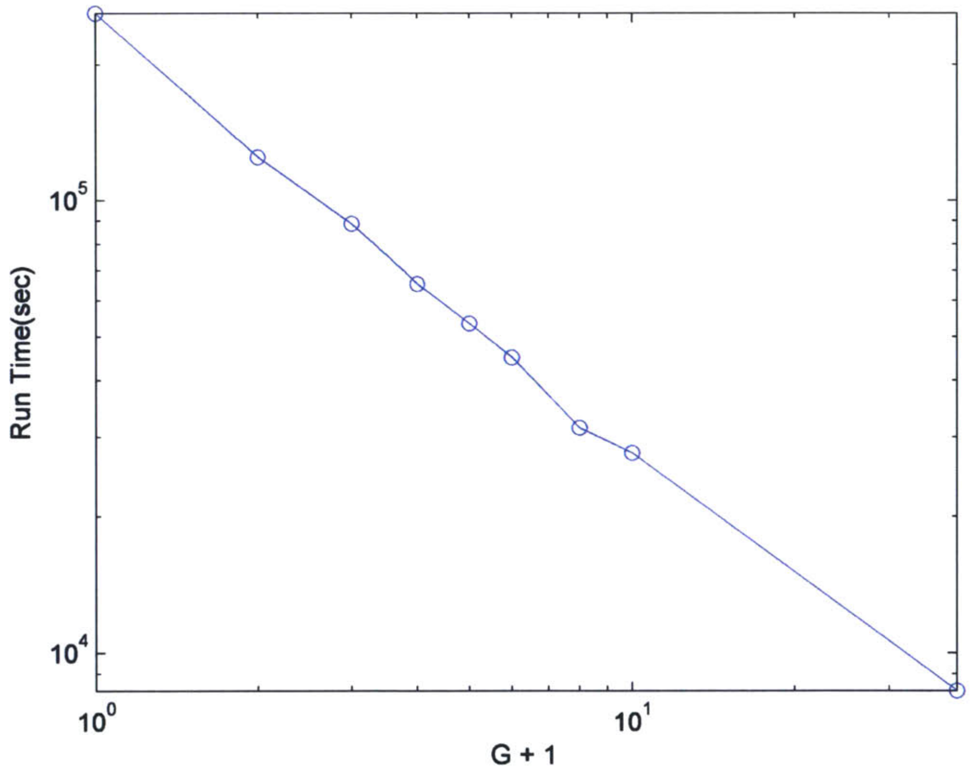


Figure 3-12: Computational time in seconds versus relative gap size plus one, G+1.

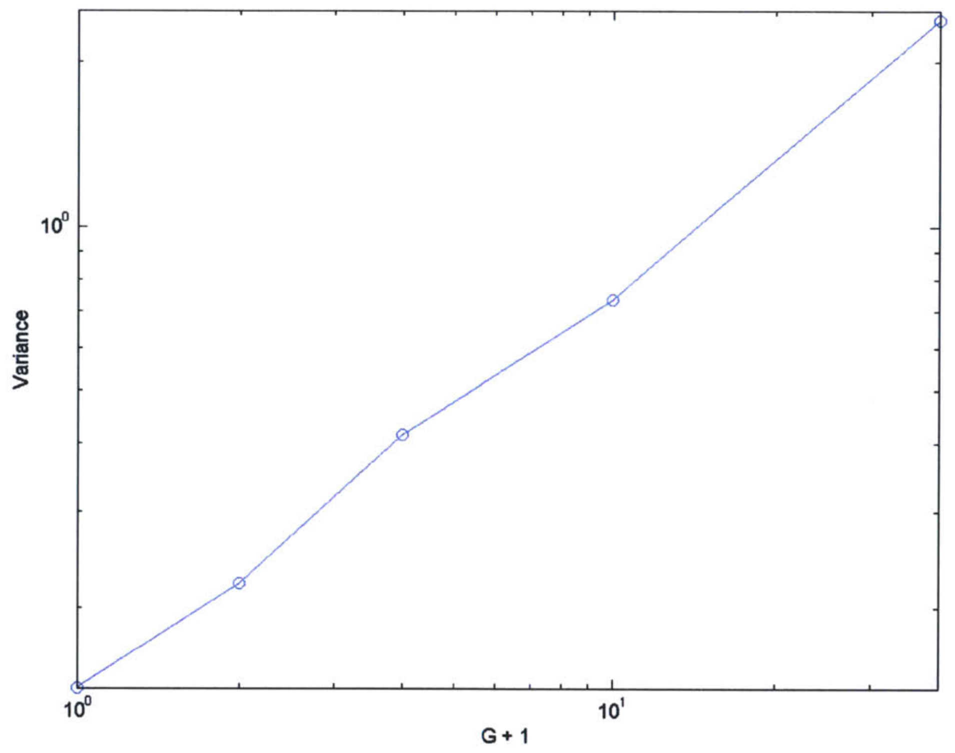


Figure 3-13: Variance in the calculated temperature over 50 ensembles versus gap size plus one, G + 1.

Chapter 4

Conclusion and Future Work

4.1 Summary

We have shown that GT-DSMC is a plausible method for conducting molecular simulations at large length scales and reduced computational cost compared with the DSMC. Application to realistic problems with non-periodic boundary conditions required the development of a boundary condition for diffuse walls. Comparison with DSMC simulations shows that the gap-tooth procedure introduces numerical error that monotonically increases as the gap size increases. Due to the low-order interpolation associated with the flux redistribution scheme used here for transferring particles from one tooth to the other, the numerical error is non-negligible and needs to be considered. Perhaps a higher order interpolation can alleviate this limitation. This, and the challenges associated with it, are discussed in the next section. We also note that using small gaps (the keep the error low) will be more beneficial in higher dimensions (d) where the cost reduction will increase as G^{d} while presumably the numerical error will not increase so fast.

We have also shown that in addition to the number of samples taken, the level of statistical uncertainty in the GT-DSMC results monotonically increases with increased gap size. In other words, for the same number of particles per cell, a GT-

DSMC simulation will exhibit larger statistical uncertainty at larger gap sizes, to the extent that the computational gain from using a smaller number of particles is almost completely lost. This is attributed to the increased correlations [14] between particles that are now continuously recycled.

4.2 Future Work

One of the greatest advantages of the gap-tooth method is its simplicity. As a result, implementation to three dimensions should be pursued in order to investigate whether, as speculated above, the numerical error remains relatively constant with G while the computational gain increases as G^{d}. Moreover, higher dimensions are more likely to reduce the fraction of particles retained within a cell and thus reduce the correlations that lead to high statistical uncertainties.

A higher-order interpolation also appears to be worthwhile since it has the potential to reduce the computational error associated with particle redistribution. As explained in [10], the challenge associated with such approaches is that the associated interpolation polynomial is negative, thus complicating the physical interpretation of the flux interpolation as a probabilistic particle redistribution. It is possible that when coupled with the recently developed variance reduction techniques ([4], [10]) which use positive and negative particles, one can find a natural interpretation for negative polynomial coefficients.

A more exhaustive study into the relationship between statistical uncertainty and gap size needs to be undertaken as well, as it may uncover the precise origin of the increased statistical uncertainty and may suggest ways in which it can be alleviated.

Appendix A

Matlab Implementation code

This appendix contains the matlab code that was used to produce the results of section 3.2. Appendix section A.1 contains routines that are used by both the DSMC method and the GT-DSMC method, section A.2 contains the DSMC only routines, and section A.3 contains the code needed to run implementation II of GT-DSMC. To obtain electronic copies of this, or additional related codes, or for more information feel free to contact the author with requests.

A.1 General Files

A.1.1 Setup file - setup.m

```
%setup.m - code contains all of the constants that will be
%fed into the dsmc or gap-tooth codes.  The idea is to have one
%centralized file in order to ensure that we are passing in identical
%parameters for our runs. Both codes call the same setup file.

nstep = 1000*1000;      % Number of time steps to run the code for.
ncell = 241;            % Number of cells to divide the system into.
npart = ncell*500;      % Number of particles per cell.
vwall_m = 1;            % Wall velocity as Mach number

% Initialize constants  (particle mass, diameter, etc.)
% We are using argon, so the constants below reflect that.
boltz = 1.3806e-23;    % Boltzmann's constant (J/K)
```

```
mass = 6.63e-26;          % Mass of argon atom (kg)
diam = 3.66e-10;          % Effective diameter of argon atom (m)
T = 273;                  % Initial temperature (K)
density = 2.685e25;       % Number density of argon at STP (m^-3)
L = 6.2579e-8;            % System size is one mean free path -> Kn = 1
Volume = L^3;             % Volume of the system (m^3)
tauFactor=0.25;

%Calculating system parameters of interest.  Note, displays to screen.
mfp = 1/(sqrt(2)*pi*diam^2*density);                    % Mean free path
fprintf('System width is %g mean free paths \n',L/mfp);
mpv = sqrt(2*boltz*T/mass);            % Most probable initial velocity
vwall = vwall_m * sqrt(2 * boltz*T/mass);               %Converts to m/s
fprintf('Wall velocities are %g and %g m/s \n',-vwall,vwall);
walls = linspace(0,L,ncell+1);      %Array of wall positions for cells
selxtra = zeros(ncell,1);        % Used by collision routine "colider"
vrmax = 3*mpv*ones(ncell,1);      % Estimated max rel. speed in a cell

tau = tauFactor*(L/ncell)/mpv;                          % Set timestep
disp(['Final Time = ' num2str(tau*nstep)]);

%* Initialize structure and variables used in statistical sampling
sampData = struct('ncell', ncell,    ...
                  'nsamp', 0,    ...
                  'ave_n', zeros(ncell,1), ...
                  'ave_u', zeros(ncell,3), ...
                  'ave_T', zeros(ncell,1));
tsamp = 0;                     % Total sampling time
dvtot = zeros(1,2);            % Total momentum change at a wall
dverr = zeros(1,2);            % Used to find error in dvtot
colSum = 0;  strikeSum = [0 0];
```

A.1.2 Particle sorting routine - sorter.m

```
function sD = sorter(x,L,sD)
% sorter - Function to sort particles into cells
% sD = sorter(x,L,sD)
% Inputs
%    x       Positions of particles
%    L       System size
%    sD      Structure containing sorting lists
% Output
%    sD      Structure containing sorting lists
%
% A. Garcia's particle sorting code, unmodified for J.Armour thesis.
% Publically avaliable, version can be downloaded from:
% http://www.algarcia.org/nummeth/Programs2E.html

%* Find the cell address for each particle
```

```
npart = sD.npart;
ncell = sD.ncell;
jx = floor(x*ncell/L) + 1;
jx = min( jx, ncell*ones(npart,1) );

%* Count the number of particles in each cell
sD.cell_n = zeros(ncell,1);
for ipart=1:npart
  sD.cell_n( jx(ipart) ) = sD.cell_n( jx(ipart) ) + 1;
end

%* Build index list as cumulative sum of the
%  number of particles in each cell
m=1;
for jcell=1:ncell
  sD.index(jcell) = m;
  m = m + sD.cell_n(jcell);
end

%* Build cross-reference list
temp = zeros(ncell,1);     % Temporary array
for ipart=1:npart
  jcell = jx(ipart);       % Cell address of ipart
  k = sD.index(jcell) + temp(jcell);
  sD.Xref(k) = ipart;
  temp(jcell) = temp(jcell) + 1;
end

return;
```

A.1.3 Particle collision routine - colider.m

```
function [v,crmax,selxtra,col] = ...
                     colider(v,crmax,tau,selxtra,coeff,sD)
% colide - Function to process collisions in cells
% [v,crmax,selxtra,col] = colider(v,crmax,tau,selxtra,coeff,sD)
% Inputs
%    v         Velocities of the particles
%    crmax     Estimated maximum relative speed in a cell
%    tau       Time step
%    selxtra   Extra selections carried over from last timestep
%    coeff     Coefficient in computing number of selected pairs
%    sD        Structure containing sorting lists
% Outputs
%    v         Updated velocities of the particles
%    crmax     Updated maximum relative speed
%    selxtra   Extra selections carried over to next timestep
%    col       Total number of collisions processed
%
```

```
% A. Garcia's particle colision code, unmodified for J.Armour thesis.
% Publically avaliable, version can be downloaded from:
% http://www.algarcia.org/nummeth/Programs2E.html

ncell = sD.ncell;
col = 0;          % Count number of collisions

%* Loop over cells, processing collisions in each cell
for jcell=1:ncell

 %* Skip cells with only one particle
 number = sD.cell_n(jcell);
 if( number > 1 )

  %* Determine number of candidate collision pairs
  %  to be selected in this cell
  select = coeff*number^2*crmax(jcell) + selxtra(jcell);
  nsel = floor(select);          % Number of pairs to be selected
  selxtra(jcell) = select-nsel;  % Carry over any left-over fraction
  crm = crmax(jcell);            % Current maximum relative speed

  %* Loop over total number of candidate collision pairs
  for isel=1:nsel

%* Pick two particles at random out of this cell
    k = floor(rand(1)*number);
    kk = rem(ceil(k+rand(1)*(number-1)),number);
    ip1 = sD.Xref(k+sD.index(jcell));      % First particle
    ip2 = sD.Xref(kk+sD.index(jcell));     % Second particle

%* Calculate pair's relative speed
    cr = norm( v(ip1,:)-v(ip2,:) );  % Relative speed
    if( cr > crm )          % If relative speed larger than crm,
      crm = cr;             % then reset crm to larger value
    end

    %* Accept or reject candidate pair according to relative speed
    if( cr/crmax(jcell) > rand(1) )
  %* If pair accepted, select post-collision velocities
      col = col+1;                    % Collision counter
      vcm = 0.5*(v(ip1,:) + v(ip2,:)); % Center of mass velocity
      cos_th = 1 - 2*rand(1);         % Cosine and sine of
      sin_th = sqrt(1 - cos_th^2);    % collision angle theta
      phi = 2*pi*rand(1);             % Collision angle phi
      vrel(1) = cr*cos_th;            % Compute post-collision
      vrel(2) = cr*sin_th*cos(phi);   % relative velocity
      vrel(3) = cr*sin_th*sin(phi);
      v(ip1,:) = vcm + 0.5*vrel;      % Update post-collision
      v(ip2,:) = vcm - 0.5*vrel;      % velocities
    end
  end % Loop over pairs
  crmax(jcell) = crm;     % Update max relative speed
 end
end   % Loop over cells
return;
```

A.1.4 Statistical sampling routine - sampler.m

```
function sampD = sampler(x,v,npart,L,sampD)
% sampler - Function to sample density, velocity and temperature
% Inputs
%    x       Particle positions
%    v       Particle velocities
%    npart   Number of particles
%    L       System size
%    sampD   Structure with sampling data
% Outputs
%    sampD   Structure with sampling data
%
% A. Garcia's particle sampling code.
%  We modified the original code, commenting out line 44.  This is
%  because with the velocity offsets near the walls of our system,
%  this process was causing points near the boundary to be off.  We
%  reimplement what it is essentially doing here in our
%  post-processing instead!
%
% Publically avaliable original version can be downloaded from:
% http://www.algarcia.org/nummeth/Programs2E.html

%* Compute cell location for each particle
ncell = sampD.ncell;
jx=ceil(ncell*x/L);

%* Initialize running sums of number, velocity and v^2
sum_n = zeros(ncell,1);
sum_v = zeros(ncell,3);
sum_v2 = zeros(ncell,1);

%* For each particle, accumulate running sums for its cell
for ipart=1:npart
  jcell = jx(ipart);  % Particle ipart is in cell jcell
  sum_n(jcell) = sum_n(jcell)+1;
  sum_v(jcell,:) = sum_v(jcell,:) + v(ipart,:);
  sum_v2(jcell) = sum_v2(jcell) + ...
            v(ipart,1)^2 + v(ipart,2)^2 + v(ipart,3)^2;
end

%* Use current sums to update sample number, velocity
%  and temperature
```

```
for i=1:3
  sum_v(:,i) = sum_v(:,i)./sum_n(:);
end
sum_v2 = sum_v2./sum_n;
sampD.ave_n = sampD.ave_n + sum_n;
sampD.ave_u = sampD.ave_u + sum_v;
sampD.ave_T = sampD.ave_T + sum_v2 ;%- ...
         %(sum_v(:,1).^2 + sum_v(:,2).^2 + sum_v(:,3).^2);
 %Comment above out as the sum_v(:,2).^2 causes the points near the
 %boundary to be off because of the velocity offsets.
 %Instead, we do this in post-processing.
sampD.nsamp = sampD.nsamp + 1;
return;
```

A.1.5 Post Processing routine - PostProcessing.m

```
%PostProcessing - opens up results from run, calculates things, saves
% into data structure.
function [filename]=PostProcessing(datafile, fileprefix)

if nargin < 1
    [file path] = uigetfile('*.mat','Please select results.');
    datafile = [path file];
end

load(datafile);

if nargin < 2
    fileprefix = 'final';
end

%* Normalize the accumulated statistics
nsamp = sampData.nsamp;
disp(['nsamp: ' num2str(nsamp)]);
const = eff_num/(Volume/ncell);
ave_n = const*sampData.ave_n/nsamp;  %mean number density in each cell
ave_u = sampData.ave_u/nsamp;  %mean velocity in each cell
%correct for modification in sampler.m
ave_T = mass/(3*boltz) * (sampData.ave_T /nsamp) - ...
    mass/(3*boltz) * ave_u(:,1).*ave_u(:,1) -...
    mass/(3*boltz) * ave_u(:,2).*ave_u(:,2)-...
    mass/(3*boltz) * ave_u(:,3).*ave_u(:,3);


% Plot average density, velocity and temperature; optional
figure(1); clf;
xcell = ((1:ncell)-0.5)/ncell * L;
plot(xcell,ave_n); xlabel('position');  ylabel('Number density');
figure(2); clf;
```

```
plot(xcell,ave_u); xlabel('position');  ylabel('Velocities');
legend('x-component','y-component','z-component');
figure(3); clf;
plot(xcell,ave_T); xlabel('position');  ylabel('Temperature');

%Extract important data and save to data structure.
data.Volume = Volume;
data.eff_num = eff_num;
data.ncell = ncell;
data.ave_n = ave_n;
data.ave_u = ave_u;
data.ave_T = ave_T;
data.xcell = xcell;
data.npartStart = npart;
data.npartEnd = npartEnd;
data.nsamp = nsamp;

if strcmp(saveFilePrefix,'truth');
saveFilePrefix = [saveFilePrefix '_'];
gap = 1;
end

data.gap = gap;
data.tauFactor = tauFactor;

%Save Data into new .mat file.
filename = [fileprefix '_' saveFilePrefix num2str(gap) '.mat'];
save(filename, 'data');
disp(['Data converted into data structure and saved in: ' filename]);
```

A.2 DSMC specific files

A.2.1 Main DSMC wrapper - Truth.m

```
function Truth
%  This is a program to simulate a dilute gas using a DSMC algorithm
%  This version simulates planar Couette flow of gaseous argon.
%  We will use the results at truth data to compare GT-DSMC results
%  to.
%
% Modified from A. Garcia's program dsmcne, publically avaliable
% at: http://www.algarcia.org/nummeth/Programs2E.html

clearvars -except ;  help Truth;   % Clear memory and print header

setup; % Initialize constants and info, identical to GT case
eff_num = density*Volume/npart; % Calculate effective number
```

```
x = L*rand(npart,1);     % Assign random positions to particles
% Assign thermal velocities using Gaussian random numbers
v = sqrt(boltz*T/mass) * randn(npart,3);
% Add velocity gradient to the y-component
v(:,2) = v(:,2) + 2*vwall*(x(:)/L) - vwall;

% Initialize variables for evaluating collisions, much of this done
% during setup.
coeff = 0.5*eff_num*pi*diam^2*tau/(Volume/ncell);

%* Declare structure for lists used in sorting
sortData = struct('ncell', ncell,    ...
                 'npart', npart,    ...
                 'cell_n', zeros(ncell,1),  ...
                 'index', zeros(ncell,1),    ...
                 'Xref', zeros(npart,1));

for istep = 1:nstep

  % Move all the particles
  [x, v, strikes, delv] = mover(x,v,npart,L,mpv,vwall,tau);
  strikeSum = strikeSum + strikes;

  % Sort the particles into cells
  sortData = sorter(x,L,sortData);

  % Evaluate collisions among the particles
  [v, vrmax, selxtra, col] = ...
        colider(v,vrmax,tau,selxtra,coeff,sortData);
  colSum = colSum + col;

  % After settle time, accumulate statistical samples, our sampler
  %is slightly different than the original; see sampler for details.
  if(istep > nstep/10)
    sampData = sampler(x,v,npart,L,sampData);
    dvtot = dvtot + delv;
    dverr = dverr + delv.^2;
    tsamp = tsamp + tau;
  end

  % Periodically display the current progress
  if( rem(istep,1000) < 1 )
    fprintf('Finished %g of %g steps, Collisions = %g\n',...
        istep,nstep,colSum);
    fprintf('Total wall strikes: %g (left)  %g (right)\n',...
        strikeSum(1),strikeSum(2));
  end
end

%Saving raw results.
npartEnd = npart; %fluff
```

```
gap = 1; %fluff
saveFilePrefix = 'truth_shear_';
filename = ['raw_' saveFilePrefix '.mat'];
save(filename);
disp(['Results saved in ' filename]);
```

A.2.2 Particle movement routine - mover.m

```
function [x,v,strikes,delv] = mover(x,v,npart, ...
                                  L,mpv,vwall,tau)
% mover - Function to move particles by free flight
%         Also handles collisions with walls
% Inputs
%    x        Positions of the particles
%    v        Velocities of the particles
%    npart    Number of particles in the system
%    L        System length
%    mpv      Most probable velocity off the wall
%    vwall    Wall velocities
%    tau      Time step
% Outputs
%    x,v      Updated positions and velocities
%    strikes  Number of particles striking each wall
%    delv     Change of y-velocity at each wall
%
% A. Garcia's particle moving code, unmodified for J.Armour thesis.
% Publically avaliable, version can be downloaded from:
% http://www.algarcia.org/nummeth/Programs2E.html


%* Move all particles pretending walls are absent
x_old = x;            % Remember original position
x(:) = x_old(:) + v(:,1)*tau;

%* Loop over all particles
strikes = [0 0];  delv = [0 0];
xwall = [0 L];  vw = [-vwall vwall];
direction = [1 -1];   % Direction of particle leaving wall
stdev = mpv/sqrt(2);
for i=1:npart

  %* Test if particle strikes either wall
  if( x(i) <= 0 )
    flag=1; % Particle strikes left wall
  elseif( x(i) >= L )
    flag=2; % Particle strikes right wall
  else
    flag=0;  % Particle strikes neither wall
  end
  %* If particle strikes a wall, reset its position
  %  and velocity. Record velocity change.
  if( flag > 0 )
    strikes(flag) = strikes(flag) + 1;
    vyInitial = v(i,2);
    %* Reset velocity components as biased Maxwellian,
    %  Exponential dist. in x; Gaussian in y and z
    v(i,1) = direction(flag)*sqrt(-log(1-rand(1))) * mpv;
    v(i,2) = stdev*randn(1) + vw(flag); % Add wall velocity
    v(i,3) = stdev*randn(1);
    % Time of flight after leaving wall
    dtr = tau*(x(i)-xwall(flag))/(x(i)-x_old(i));
    %* Reset position after leaving wall
    x(i) = xwall(flag) + v(i,1)*dtr;
    %* Record velocity change for force measurement
    delv(flag) = delv(flag) + (v(i,2) - vyInitial);
  end
end
```

A.3 Dual Bounce implementation files

A.3.1 Main GT-DSMC wrapper - GT2.m

```
function [filename]=GT2(gap,saveFilePrefix)
%function [filename, time]=GTwWallReset(gap,saveFilePrefix)
% GTwWallRest- Program to simulate a dilute gas using DSMC algorithm
% with gap tooth implementation
% This version simulates planar Couette flow
%
% Usage: GTwWallRest(gap,saveFilePrefix)
%

if nargin < 1
    gap = 2;
end

if nargin < 2
    saveFilePrefix = 'gap_wb_';
end

clearvars -except gap saveFilePrefix;  help GTwWallReset;

setup;

%* Assign random positions and velocities to particles
%rand('state',1);        % Initialize random number generators
```

```
%randn('state',1);
x = L*rand(npart,1);    % Assign random positions

% Code added here to only keep those particles initially assigned in
% the teeth.
walls = linspace(0,L,ncell+1);
cell = ceil(x/L*ncell);
a = find(mod(cell-1,gap) == 0);
x= x(a);
npart = size(x,1);

if gap > 1
    usedcells = floor(ncell/(gap))+1; %calc number of cells with parts
else
    usedcells = ncell;
end
eff_num = density*(Volume/ncell*usedcells)/npart;
fprintf('Each simulation particle represents %g atoms\n',eff_num);

% Assign thermal velocities using Gaussian random numbers
v = mpv*sqrt(-log(1-rand(npart,3))).*cos(2*pi*rand(npart,3));

%%IF you want to run equilibrium conditions, uncomment the following.
%vwall = 0;

% Add velocity gradient to the y-component
v(:,2) = v(:,2) + 2*vwall*(x(:)/L) - vwall;

%* Initialize variables used for evaluating collisions
coeff = 0.5*eff_num*pi*diam^2*tau/(Volume/ncell);

%* Declare structure for lists used in sorting
sortData = struct('ncell', ncell,    ...
                  'npart', npart,    ...
                  'cell_n', zeros(ncell,1),  ...
                  'index', zeros(ncell,1),     ...
                  'Xref', zeros(npart,1));

disp(['Starting with npart: ' num2str(npart)]);
disp(['So, ' num2str(npart/usedcells) ' particles per cell.']);

for istep = 1:nstep

  %* Move all the particles
  [x, v, strikes, delv] = GT2mover(x,v,L,mpv,vwall,tau,gap,ncell);
  strikeSum = strikeSum + strikes;

  %* Sort the particles into cells
  sortData = sorter(x,L,sortData);

  %* Evaluate collisions among the particles
  [v, vrmax, selxtra, col] = ...
          colider(v,vrmax,tau,selxtra,coeff,sortData);
  colSum = colSum + col;

  %* After initial transient, accumulate statistical samples
  if(istep > nstep/10)
    sampData = sampler(x,v,npart,L,sampData);
    dvtot = dvtot + delv;
    dverr = dverr + delv.^2;
    tsamp = tsamp + tau;
  end

  %* Periodically display the current progress
  if( rem(istep,1000) < 1 )
    fprintf('Finished %g of %g steps, Collisions = %g\n', ...
                                        istep,nstep,colSum);
    fprintf('Total wall strikes: %g (left)  %g (right)\n', ...
                                  strikeSum(1),strikeSum(2));
    %plottingLocations;
  end
end

%Check that end number of particles in teeth equals initial number of
%particles in teeth: this only works for relative gaps > 0....
cells = ceil(x/L*ncell);
a = find(mod(cells-1,gap) == 0);
xEnd= x(a);
npartEnd = size(xEnd,1);

disp(['Ending with npart == starting npart: ' num2str(npartEnd)]);
if npartEnd ~= npart
    disp('NUMBER OF PARTICLES HAS CHANGED!!!!');
end

%Save data
filename = ['raw_' saveFilePrefix num2str(gap) '.mat'];
save(filename);
disp(['Data saved in ' filename]);
end
```

A.3.2 GT particle movement routine - GT2mover.m

```
function [x,v,strikes,delv] = GT2mover(x,v,L,mpv,vwall,tau,gap,ncell)
% GT2mover - Function to move particles by free flight
%         Also handles collisions with walls thermal walls
% Inputs
%    x       Positions of the particles
%    v       Velocities of the particles
%    L        System length
%    mpv      Most probable velocity off the wall
%    vwall    Wall velocities
%    tau      Time step
%    gap      relative gap size + 1
%    ncell    number of cells in the system
% Outputs
%    x,v      Updated positions and velocities
%    strikes  Number of particles striking each wall
%    delv     Change of y-velocity at each wall

%* Move all particles pretending walls are absent
x_old = x;              % Remember original position
dx = v(:,1)*tau;
x(:) = x_old(:) + dx;
m = sign(dx);
cell = ceil(x/L*ncell);
cell_old = ceil(x_old/L*ncell);
r = m > 0;
walls = linspace(0,L,ncell+1);
dx_wall = walls(cell_old + r)' - x_old;
a = abs(dx) < abs(dx_wall);
dx_in = a.*dx + ~a.*dx_wall;
dx_out = dx - dx_in;

%* Loop over all particles
strikes = [0 0];  delv = [0 0];
xwall = [0 L];  vw = [-vwall vwall];
direction = [1 -1];   % Direction of particle leaving wall
stdev = mpv/sqrt(2);


%% 1.6 times faster with function "move.m" embedded in code. Still
%% need move.m, in case particle indeed needs to move recursively!

%%Vectorized vs for-loop flag assignment. 60 times faster as vector
%%for 10000 particles... :-)
left = find(x <= 0); % Particle strikes left wall
right = find(x >= L);  % Particle strikes right wall
 % otherwise, Particle strikes neither wall
flag = zeros(size(x));
flag(left) = 1; % Particle strikes left wall, set flag 1
flag(right) = 2; % Particle strikes right wall, set flag 2
% otherwise, Particle strikes neither wall, set flag 0

%find particles in the wall cells moving away from the wall that leave
%the cell. We will subject these to the extra initialization step when
%if they loop back into the wall cell when moved.
extraBounce = find((cell_old == 1 & dx_out > 0) ...
    | (cell_old == ncell & dx_out < 0));
```

```
temp = m(extraBounce);
tempB = temp > 0;
tempC = tempB + ~tempB*2;
flag(extraBounce) = tempC;


neither = find(x>0 & x<L & ~((cell_old == 1 & dx_out > 0)...
    | (cell_old == ncell & dx_out < 0)));

for j = 1:size(left,1)
    i = left(j);
    %either hit wall and bounce off or loop around
    itest = rand(1);
    alpha = 1/(gap);
    if itest >= alpha
        %stay
        %reenters the cell at the right edge!
         if abs(dx_out(i)) > L/ncell
          m(i) = sign(dx_out(i));
          dx_out(i) = dx_out(i) - m(i)*(L/ncell);
          delta_t = (dx_in(i)./abs(v(i,1))) + ((L/ncell)/abs(v(i,1)));
          dtr = tau - delta_t;
          bounce; %recursion
         else
            x(i) = (0 + L/ncell) + dx_out(i);
         end
    else
        %"move" to next cell, so bounce off wall!
        %* If particle strikes a wall, reset its position
        %  and velocity. Record velocity change.
        delta_t = abs(dx_in(i)./abs(v(i,1)));
        dtr = tau - delta_t;     % Time of flight after leaving wall
        strikes(flag(i)) = strikes(flag(i)) + 1;
        vyInitial = v(i,2);
        %* Reset velocity components as biased Maxwellian,
        %  Exponential dist. in x; Gaussian in y and z
        v(i,1) = direction(flag(i))*sqrt(-log(1-rand(1))) * mpv;
        v(i,2) = stdev*randn(1) + vw(flag(i)); % Add wall velocity
        v(i,3) = stdev*randn(1);
        %* Reset position after leaving wall
        dx_out(i) = v(i,1)*dtr;
        if abs(dx_out(i)) > L/ncell
            m(i) = sign(dx_out(i));
            dx_out(i) = dx_out(i) - m(i)*(L/ncell);
            move;
        else
            x(i) = xwall(flag(i)) + dx_out(i);
        end
        %* Record velocity change for force measurement
        delv(flag(i)) = delv(flag(i)) + (v(i,2) - vyInitial);
    end
```

```
end

for j=1:size(right,1)
    i = right(j);
    %either hit right wall or loop around
    itest = rand(1);
    alpha = 1/(gap); %replace with function of b
    if itest >= alpha
        %stay
        if abs(dx_out(i)) > L/ncell
          m(i) = sign(dx_out(i));
          dx_out(i) = dx_out(i) - m(i)*(L/ncell);
          delta_t = (dx_in(i)./abs(v(i,1))) + ((L/ncell)/abs(v(i,1)));
          dtr = tau - delta_t;
          bounce; %recursion
        else
            x(i) = (L - L/ncell) + dx_out(i);
        end
    else
        %move
        %* If particle strikes a wall, reset its position
        %  and velocity. Record velocity change.
        delta_t = abs(dx_in(i)./abs(v(i,1)));
        dtr = tau - delta_t;      % Time of flight after leaving wall
        strikes(flag(i)) = strikes(flag(i)) + 1;
        vyInitial = v(i,2);
        %* Reset velocity components as biased Maxwellian,
        %  Exponential dist. in x; Gaussian in y and z
        v(i,1) = direction(flag(i))*sqrt(-log(1-rand(1))) * mpv;
        v(i,2) = stdev*randn(1) + vw(flag(i)); %Add wall velocity bias
        v(i,3) = stdev*randn(1);
        %* Reset position after leaving wall
        dx_out(i) = v(i,1)*dtr;
        if abs(dx_out(i)) > L/ncell
            m(i) = sign(dx_out(i));
            dx_out(i) = dx_out(i) - m(i)*(L/ncell);
            move; %recursion
        else
            x(i) = xwall(flag(i)) + dx_out(i);
        end
        %* Record velocity change for force measurement
        delv(flag(i)) = delv(flag(i)) + (v(i,2) - vyInitial);
    end
end

%moving internal particles
for j = 1:size(neither,1)
  i = neither(j);
  if cell(i) ~= cell_old(i)
    itest = rand(1);
    alpha = 1/gap;
    r = m(i) > 0;
    if itest >= alpha
        %stay
        if abs(dx_out(i)) > (L/ncell)
            m(i) = sign(dx_out(i));
            dx_out(i) = dx_out(i) - m(i)*(L/ncell);
            move;
        else
            x(i) = walls(cell_old(i)+~r) + dx_out(i);
        end
    else
        %move
        if abs(dx_out(i)) > (L/ncell)
            m(i) = sign(dx_out(i));
            dx_out(i) = dx_out(i) - m(i)*(L/ncell);
            move;
        else
        x(i) = walls(cell_old(i)+~r) + (m(i)*gap/ncell*L) + dx_out(i);
        end
    end
  end
end

%moving particles in wall cells. seperate from above routine to aid in
%implementation of extra bounce.
for j = 1:size(extraBounce,1)
  i = extraBounce(j);
  if cell(i) ~= cell_old(i)
    itest = rand(1);
    alpha = 1/gap;
    r = m(i) > 0;
    if itest >= alpha
        %first make note of time passed
        delta_t = (abs(dx_in(i))./abs(v(i,1))); %time to move to wall.
        dtr = tau - delta_t;
        %THIS PART IS DIFFERENT.  Has reinitialization of parts step!
        vyInitial = v(i,2);
        %* Reset velocity components as biased Maxwellian,
        %  Exponential dist. in x; Gaussian in y and z
        v(i,1) = direction(flag(i))*sqrt(-log(1-rand(1))) * mpv;
        v(i,2) = stdev*randn(1) + vw(flag(i)); % Add wall velocity
        v(i,3) = stdev*randn(1);
        %* Reset position after leaving wall
        dx_out(i) = v(i,1)*dtr;
        if abs(dx_out(i)) > L/ncell
            m(i) = sign(dx_out(i));
            dx_out(i) = dx_out(i) - m(i)*(L/ncell);
            moveExtra;
        else
            x(i) = xwall(flag(i)) + dx_out(i);
        end
    else
        %THIS IS THE SAME AS ABOVE LOOPS
        %move
        if abs(dx_out(i)) > (L/ncell)
            m(i) = sign(dx_out(i));
            dx_out(i) = dx_out(i) - m(i)*(L/ncell);
            move;
        else
            x(i) = walls(cell_old(i)+~r) + (m(i)*gap/ncell*L) + dx_out(i);
        end
    end
  end
end
```

A.3.3 Recursive particle movement routine 1 - move.m

```
%move- recursive movement, given some dx_out(i) value...
itest = rand(1);
alpha = 1/gap;
r = m(i) > 0;
if itest >= alpha
    %stay
    if abs(dx_out(i)) > (L/ncell)
        m(i) = sign(dx_out(i));
        dx_out(i) = dx_out(i) - m(i)*(L/ncell);
        move;
    else
        x(i) = walls(cell_old(i)+~r) + dx_out(i);
    end
else
    %move
    if abs(dx_out(i)) > (L/ncell)
        m(i) = sign(dx_out(i));
        dx_out(i) = dx_out(i) - m(i)*(L/ncell);
        move;
    else
        x(i) = walls(cell_old(i)+~r) + (m(i)*gap/ncell*L) + dx_out(i);
    end
end
```

A.3.4 Recursive particle movement routine 2 - moveExtra.m

```
itest = rand(1);
alpha = 1/gap;
r = m(i) > 0;
if itest > alpha
    %first make note of time passed
    delta_t = (L/ncell)/abs(v(i,1)) + delta_t; %time to move to wall.
    dtr = dtr - delta_t;
    %THIS PART IS DIFFERENT.  Has reinitialization of particles step!
    vyInitial = v(i,2);
    %* Reset velocity components as biased Maxwellian,
    %  Exponential dist. in x; Gaussian in y and z
    v(i,1) = direction(flag(i))*sqrt(-log(1-rand(1))) * mpv;
    v(i,2) = stdev*randn(1) + vw(flag(i)); % Add wall velocity
    v(i,3) = stdev*randn(1);
    %* Reset position after leaving wall
    dx_out(i) = v(i,1)*dtr;
    if abs(dx_out(i)) > L/ncell
        m(i) = sign(dx_out(i));
        dx_out(i) = dx_out(i) - m(i)*(L/ncell);
        moveExtra;
    else
        x(i) = xwall(flag(i)) + dx_out(i);
    end
else
    %move
    if abs(dx_out(i)) > (L/ncell)
        m(i) = sign(dx_out(i));
        dx_out(i) = dx_out(i) - m(i)*(L/ncell);
        move;
    else
        x(i) = walls(cell_old(i)+~r) + (m(i)*gap/ncell*L) + dx_out(i);
    end
end
```

A.3.5 Recursive particle bounce routine - bounce.m

```
%bounce.m - for case when particle loops around and THEN recursively
%bounces.
itest = rand(1);
alpha = 1/(gap);
if itest >= alpha
    %stay (ie loop around and reenter)
    if abs(dx_out(i)) > L/ncell
        m(i) = sign(dx_out(i));
        dx_out(i) = dx_out(i) - m(i)*(L/ncell);
        bounce; %in case of recursive bouncing.
    else
        if cell_old(i) == ncell
            x(i) = (L - L/ncell) + dx_out(i);
            %(L- L/ncell) is right hand wall of last cell
        else
            x(i) = 0 - dx_out(i);
        end
    end
```

```
else
    %move
    %* If particle strikes a wall, reset its position
    %  and velocity. Record velocity change.
    strikes(flag(i)) = strikes(flag(i)) + 1;
    vyInitial = v(i,2);
    %* Reset velocity components as biased Maxwellian,
    %  Exponential dist. in x; Gaussian in y and z
    v(i,1) = direction(flag(i))*sqrt(-log(1-rand(1))) * mpv;
    v(i,2) = stdev*randn(1) + vw(flag(i)); % Add wall velocity bias
    v(i,3) = stdev*randn(1);
    %* Reset position after leaving wall
    dx_out(i) = v(i,1)*dtr;
    if abs(dx_out(i)) > L/ncell %if particle passes right through cell
        m(i) = sign(dx_out(i));
        dx_out(i) = dx_out(i) - m(i)*(L/ncell);
        move; %recursively move
    else
        x(i) = xwall(flag(i)) + dx_out(i);
    end
    %* Record velocity change for force measurement
    delv(flag(i)) = delv(flag(i)) + (v(i,2) - vyInitial);
end
```

Bibliography

[1] F. J. Alexander and A. L. Garcia. The direct simulation Monte Carlo method. *Computers in Physics*, 11(6):588 593, 1997. doi: 10.1063/1.168619. URL http://link.aip.org/link/?CIP/11/588/1. 14, 22, 23, 24, 25

[2] F. J. Alexander, A. L. Garcia, and B. J. Alder. A consistent Boltzmann algorithm. *Phys. Rev. Lett.*, 74:5212–5215, Jun 1995. doi: 10.1103/PhysRevLett.74.5212. URL http://link.aps.org/doi/10.1103/PhysRevLett.74.5212. 13

[3] F. J. Alexander, A. L. Garcia, and B. J. Alder. Cell size dependence of transport coefficients in stochastic particle algorithms. *Physics of Fluids*, 10:1540 1542, June 1998. doi: 10.1063/1.869674. 20

[4] L. L. Baker and N. G. Hadjiconstantinou. Variance reduction for Monte Carlo solutions of the Boltzmann equation. *Physics of Fluids*, 17(5):051703, May 2005. doi: 10.1063/1.1899210. 13, 52

[5] G. Bird. *Molecular gas dynamics*. Oxford engineering science series. Clarendon Press, 1976. ISBN 9780198561200. URL http://books.google.com.pe/books?id=0AgpAQAAMAAJ. 13

[6] G. A. Bird. Approach to translational equilibrium in a rigid sphere gas. *Physics of Fluids*, 6:1518, 1963. 13

[7] I. D. Boyd, G. Chen, and G. V. Candler. Predicting failure of the continuum fluid equations in transitional hypersonic flows. *Phys. Fluids*, 7:210 219, 1995. 14

[8] C. Cercignani. *The Boltzmann equation and its applications*. Applied mathematical sciences. Springer-Verlag, 1988. ISBN 9780387966373. URL http://books.google.com/books?id=fnAyzbpYq7IC. 14

[9] C. W. Gear and I. G. Kevrekidis. Boundary processing for Monte Carlo simulations in the gap-tooth scheme. *ArXiv Physics e-prints*, Nov. 2002. 17

[10] C. W. Gear, J. Li, and I. G. Kevrekidis. The gap-tooth method in particle simulations. *Phys. Lett. A*, 316:190 195, 2003. 15, 17, 26, 27, 29, 52

[11] N. G. Hadjiconstantinou. Analysis of discretization in the direct simulation Monte Carlo. *Physics of Fluids*, 12:2634–2638, Oct. 2000. doi: 10.1063/1.1289393. 20

[12] N. G. Hadjiconstantinou. The limits of Navier-Stokes theory and kinetic extensions for describing small-scale gaseous hydrodynamics. *Physics of Fluids*, 18 (11):111301, Nov. 2006. doi: 10.1063/1.2393436. 13, 14, 22

[13] N. G. Hadjiconstantinou and I. G. Kevrekidis. Gap-tooth DSMC: an efficient molecular simulation method for dilute gases. Unpublished, 2002. 17, 26

[14] N. G. Hadjiconstantinou, A. L. Garcia, M. Z. Bazant, and G. He. Statistical error in particle simulations of hydrodynamic phenomena. *Journal of Computational Physics*, 187:274 297, May 2003. doi: 10.1016/S0021-9991(03)00099-8. 52

[15] I. G. Keverekidis, C. W. Gear, J. M. Hyman, P. G. Kevrekidis, O. Runborg, and C. Theodoropoulos. Equation-free multiscale computation: enabling microscopic simulators to perform system-level tasks. *Comm. Math Sci.*, 1, 4:715 762, 2003. 15, 16

[16] I. G. Kevrekidis and G. Samaey. Equation-free multiscale computation: algorithms and applications. *Annual Review of Physical Chemistry*, 60:321 344, 2009. URL `https://lirias.kuleuven.be/handle/123456789/198365`. 16

[17] G. A. Radtke, N. G. Hadjiconstantinou, and W. Wagner. Low-noise Monte Carlo simulation of the variable hard sphere gas. *Physics of Fluids*, 23(3):030606, Mar. 2011. doi: 10.1063/1.3558887. 13, 14

[18] W. Wagner. A convergence proof for Bird's direct simulation Monte Carlo method for the Boltzmann equation. *Journal of Statistical Physics*, 66:1011 1044, 1992. ISSN 0022-4715. URL `http://dx.doi.org/10.1007/BF01055714`. 10.1007/BF01055714. 14, 27

*Note that the blue numbers displayed at the end of each reference are pages within this document where the reference has been cited.