Stochastic Methods for Modeling Hydrodynamics of Dilute Gases

By

Tristan J. Hayeck

B.S., Mechanical Engineering (2006)

Massachusetts Institute of Technology

Submitted to the Department of Mechanical Engineering in Partial Fulfillment of the Requirements for the Degree of Bachelor of Science in Mechanical Engineering at Massachusetts Institute of Technology

[June 2006]

May 12, 2006

Signature of Author .....

Department of Mechanical Engineering
May 12, 2006

Certified by .....

Nicolas Hadjiconstantinou
Associate Professor of Mechanical Engineering
Thesis Supervisor

Accepted by.....

John H. Lienhard V
Chairman, Undergraduate Thesis Committee

Stochastic Methods for Modeling Hydrodynamics of Dilute Gases

By

Tristan J. Hayeck

Submitted to the Department of Mechanical Engineering
May 12, 2006 in partial fulfillment of the requirements for the Degree of Bachelor of Science at Massachusetts Institute of Technology in Mechanical Engineering

Abstract

When modeling small scale sub-micron gas flows, continuum methods, i.e. Navier Stokes equations, no longer apply. Molecular Dynamics (MD) approaches are then more appropriate. For dilute gases, where particles travel in straight lines for the overwhelming majority of the time, MD methods are inefficient compared to kinetic theory approaches because they require the explicit calculation of each particle's trajectory. An effective way to model the hydrodynamics of dilute gases is a stochastic particle method known as Direct Simulation Monte Carlo (DSMC). In DSMC the motion and collision of particles are decoupled to increase computational efficiency.

The purpose of this thesis is to evaluate a variant of the DSMC algorithm, in which particles have discrete velocities. The most important modification to the DSMC algorithm is the treatment of collisions between particles with discrete velocities in a way which ensures strict conservation of momentum and energy. To achieve that an algorithm that finds all possible pairs of discrete post-collision velocities given a pair of discrete pre-collision velocities was developed and coded. Two important discretization ingredients were introduced: the number of discrete velocities and the maximum discrete velocity allowed.

A number of simulations were performed to compare the discrete DSMC (IDSMC) and the regular DSMC method. Our results show that the difference between the two methods is small when the allowed discrete velocity spectrum extends to high speeds. In this case the error is fairly insensitive to the number of discrete velocities used. On the other hand, when the maximum velocity allowed is small compared to the most probably particle speed (approximately equivalent to the speed of sound), large errors are observed (in our case up to 450% in the stress).

Thesis Supervisor: Nicolas Hadjiconstantinou

Title: Associate Professor of Mechanical Engineering

1. Introduction

Complex physical systems are usually analyzed using computational methods. In statistical mechanics where large numbers of degrees of freedom are typically involved, stochastic methods are common because they are typically more efficient than their deterministic counterparts. This thesis focuses on a stochastic particle simulation method for dilute gases. Such a method is necessary because for the small-scale gaseous flows of interest here continuum, such as the Navier-Stokes description fail.

One of the most frequently used methods for modeling small scale gaseous hydrodynamics is the Direct Simulation Monte Carlo (DSMC) method. DSMC is an efficient Molecular Dynamics (MD) method; however, instead of calculating the trajectory of each individual particle as in MD, in DSMC interactions (collisions) between particles are treated in a stochastic manner.

In this thesis, we study the dynamics of discrete gases, i.e. gases whose molecular velocities are limited to discrete values. We are particularly interested in how these gases differ from "real" gases whose velocities are, of course, continuous. To study these gases, we developed an integer version of the DSMC algorithm (to be described later). One of the most important modifications required is the development of an algorithm for processing collisions between particles, since post-collision velocities need to lie in the set of discrete values but also be such that momentum and energy are conserved.

In the next section, we give an introduction to dilute gases and DSMC. Section 3 discusses the discrete-velocity DSMC, or Integer DSMC (IDSMC), studied for this thesis; the algorithm used to generate integer post-collision velocities is explained in

detail. In section 4, the computational results are presented. The final section (section 5) presents our conclusions.

2. Dilute Gases and DSMC

A gaseous system can be considered as dilute if the relative distance between particles is large compared to the diameter of the particles:

$$d << \sqrt[3]{V/N} \qquad (1)$$

Here d is the diameter of the particles, V is the volume of the gas, and N is the number of particles. [1]

The computational efficiency of the system is increased by using representative particles that correspond to a certain effective number of particles. The particles in the system are considered to be hard spheres; this is a good approximation because particles in dilute gas travel in straight lines for the vast majority of the time and the only interaction occurs during collisions. This behavior leads to the definition of the mean

$$\lambda = \frac{V}{\sqrt{2}N\pi d^2} \qquad (2)$$

free path which is the average distance traveled by a particle between collisions. [2]

This behavior also lends itself naturally to a simulation process where particles can be moved in straight lines without having to numerically integrate their equations of motion as is done in MD. For simulating a dilute system, DSMC is more efficient than MD; the computational effort scales with N instead of N^2 because the DSMC method

---

[1] Alejandro L. Garcia, Numerical methods for physics / Alejandro L. Garcia. Upper Saddle River, N.J. : Prentice Hall, (2000) 341.

[2] Alejandro L. Garcia, Numerical methods for physics / Alejandro L. Garcia. Upper Saddle River, N.J. : Prentice Hall, (2000) 346.

decouples the collisions from the motion of particles. Time and space are discretized; the system advances with time steps of size Δt ,and in space the system is separated into cells of size Δx which are used for processing particle collisions. The magnitudes of these discretization ingredients need to remain small for accuracy. If the time step is too large, the particles will be allowed to travel large distances without colliding, leading to physically large transport. Collision pairs are selected from the same cell, thus if the cell sizes are too large, the randomly selected collision pairs may not represent a realistic collision since collision partners they are far apart.[3]

2.1 DSMC Algorithm

The DSMC can be thought of in terms of four basic functions: moving the particles over a set time step without collisions, appropriately applying boundary conditions of the system, organizing particles into cells, and applying collisions at the correct rate.

The initial velocity for each particle is determined from the Maxwell-Boltzman distribution; then the particles can go through the first step, moving with their initial velocity in a straight path over a time step of Δt . Next boundary conditions are considered; the model used in this thesis uses diffuse walls. Other boundary conditions are possible such as periodic boundaries, specular surfaces, and thermal walls.[4] The

---

[3] Nicolas Hadjiconstantinou, "Dilute Gases and DSMC," Introduction to Modeling and Simulation, 2006, <http://stellar.mit.edu/S/course/3/sp06/3.021J/materials.html> 7.

[4] Francis J. Alexander Alejandro L. Garcia, "The Direct Simulation Monte Carlo Method," Computers in Physics Vol 11 no 6.,(1997):589.

diffuse walls in this thesis move in their plane and with opposite velocities, in other words they simulate a Couette flow.

The collision step begins by selecting the appropriate number of collision candidate pairs as predicted by the gas collision rate. To avoid collisions between particles that are far apart, the system is divided into spatial cells and only particles within the same cell can collide.

Two different particles within the same cell are chosen at random and the collision is processed based on selection criteria which state that particles with higher relative velocity are more likely to collide. More specifically, once particles have been selected, their relative velocity is analyzed to determine if they will collide. For the hard sphere model, the probability of collision between two particles is proportional to their relative speeds, ie: [5]

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

Here the i and j indexes indicate the different particles and N_c is the number of particles in a given cell. Explicit computation of the double summation in the denominator of equation 3 is computationally intensive. Instead, there is an acceptance rejection algorithm is used.

---

[5] Francis J. Alexander Alejandro L. Garcia, "The Direct Simulation Monte Carlo Method," Computers in Physics Vol 11 no 6.,(1997):590.

More specifically a random number r is chosen and the collision is accepted if: [6]

$$r < \frac{|\mathbf{v}_i - \mathbf{v}_j|}{\mathbf{v}_{r,\max}} \qquad (4)$$

Here, $\mathbf{v}_i$- $\mathbf{v}_j$ is the relative velocity between two particles that have been randomly selected, and $v_{r,max}$ is defined as the maximum relative velocity between two particles in a given cell. Rather than taking the computational power to determine $v_{r,max}$ within a cell at a given time step, $v_{r,max}$ is assigned a reasonable high value at the beginning of the simulation. If a larger relative velocity between particles is encountered during the simulation, $v_{r,max}$ is updated to this value. Due to the fact that a number of collisions will be rejected by this acceptance-rejection algorithm, the number of possible collision partners is chosen to be larger by the ratio of accepted collisions to total collision candidates. More details can be found in reference 4.

3. The Integer DSMC Algorithm

Here we briefly describe the modifications required for creating an integer DSMC method (IDSMC). One of the most important aspects of the simulation is that it must conserve energy and momentum. Rounding the velocities from a continuous spectrum would typically lead to lack of conservation. For this reason, post-collision velocities are selected from a table that is calculated at the simulation onset. Conservation of

---

[6] Francis J. Alexander and Alejandro L. Garcia, "The Direct Simulation Monte Carlo Method," Computers in Physics Vol 11 no 6.,(1997):590.

momentum states that the velocity of the center of mass of the two particles remains constant before and after collisions: [7]

$$\mathbf{v}_{cm} = \frac{1}{2}\left(\mathbf{v}_i + \mathbf{v}_j\right) = \frac{1}{2}\left(\mathbf{v}_i^* + \mathbf{v}_j^*\right) = \mathbf{v}_{cm}^* \qquad (5)$$

The stars denote post-collision properties. Conservation of energy requires that the magnitude of relative velocity is not changed by the collision, i.e.: [8]

$$\mathbf{v}_r = \left|\mathbf{v}_i - \mathbf{v}_j\right| = \left|\mathbf{v}_i^* - \mathbf{v}_j^*\right| = \mathbf{v}_r^* \qquad (6)$$

For a given pair of particles that have been identified as colliding, the possible integer post-collision velocities must be determined. Determination of the possible pairs of discrete velocities satisfying (5) and (6) is a complex task. One of the most efficient approaches, and the one adopted here, is to construct a table which contains all possible post-collision velocities for a given pair of pre-collision velocities. This table is constructed at the beginning of the simulation and is essentially a multi-dimensional array which is indexed by the (integer) x, y, and z components of the **relative** pre-collision velocities. Working only with the relative pre-collision velocity is possible because as shown by equations (5) and (6), a collision modifies the direction of the relative velocities while the magnitude of the relative velocity and the center-of-mass velocity are conserved.

The discrete values of relative velocity in each direction are defined as $i^* \delta$, where i ranges from $-N_v$ to N_v, and N_v is an integer. The maximum relative velocity

---

[7] Alejandro L. Garcia, Numerical methods for physics / Alejandro L. Garcia. Upper Saddle River, N.J. : Prentice Hall, (2000) 358.

[8] Alejandro L. Garcia, Numerical methods for physics / Alejandro L. Garcia. Upper Saddle River, N.J. : Prentice Hall, (2000) 358.

considered (in each direction) is $N_v * \delta = \Delta v^*_{max}$. δ can thus be thought of as the velocity increment or step between discrete values. The introduction of a maximum velocity is a discretization effect which introduces some error. (In fact, the error introduced will be investigated below). This form of discretization is introduced in order to make the number of discrete velocities finite and thus make the simulation tractable. Taking advantage of symmetry, we work with $N_v \geq 0$. Negative relative velocities can easily be obtained from the pre-computed table by switching the appropriate relative component signs. The possible pairs of post-collision velocities are found by checking all possible combinations (using nested loops) in the reference frame of one of the particles. The post-collision velocity range searched in every direction can be determined by the following simple argument.

The maximum post-collision velocity, v^*_{max}, can be determined from the maximum relative velocity. Conservation of momentum requires that:

$$\mathbf{v}_i^* = \mathbf{v}_{cm}^* + \frac{1}{2}\mathbf{v}_r^* \qquad (7)$$

$$\mathbf{v}_j^* = \mathbf{v}_{cm}^* - \frac{1}{2}\mathbf{v}_r^*$$

To determine v^*_{max}, the maximum $\mathbf{v}^*_{cm}$ and the maximum $\mathbf{v}^*_r$ must be determined. The maximum relative velocity in the x, y, and z directions has already been defined as $N_v * \delta$. A bond for each component of $\mathbf{v}^*_{cm}$ is $.5 * \delta * N_v$ since we are considering the system from the reference frame particle one. The maximum $\mathbf{v}^*_r$ magnitude is given by:

$$\left|\mathbf{v}_{r,\max}\right| = \sqrt{v^2_{r,x,\max} + v^2_{r,y,\max} + v^2_{r,z,\max}} = \sqrt{3} * \delta * N_v \qquad (8)$$

Plugging equation 8 back into equation 7 we get:

$$v^*_{max} = \frac{1}{2}\Delta v_{max}\left[1+\sqrt{3}\right] \qquad (9)$$

4. Results

To study the effect of discrete particle velocities we simulated a Couette flow (two infinite parallel walls moving in opposite directions). The wall velocities were set to ± 250 m/s. The physical domain was divided into 50 cells with 10^5 particles per cell and the simulation was evolved until steady state was observed. Gaseous Argon was simulated by taking the molecular mass $m=6.63*10^{-26}$ kg and a molecular diameter $d=3.66*10^{-10}$m. The system size was sufficiently larger than the mean free path, leading to collision dominated particle dynamics.

In all cases, the data was collected at steady state and the respective velocity and shear values for each cell were averaged over 1000 time steps. The percent differences between the DSMC and IDSMC results are normalized by the DSMC results. The shear stress is reported as a mean over the 50 spatial cells.

For one set of data the step size, δ, is left constant (37.5m/s) while N_v, the number of velocities in the multi-dimensional array is varied between 10 and 40; for $N_v = 10$ the maximum real velocity represented is 370m/s and for $N_v = 40$ the maximum velocity represented is 1500m/s. For the second set of data the maximum velocity, M_r, is kept constant (1500m/s), while N_v varies. When $N_v = 10$, $\delta = 150$ m/s, while when $N_v = 40$, $\delta = 37.5$m/s. The value of 1500 m/s was arbitrarily chosen, but is considered a reasonable value because it is several times larger than the speed of sound (340m/s).

Below is a plot of the velocity profile for M_r =1500m/s.

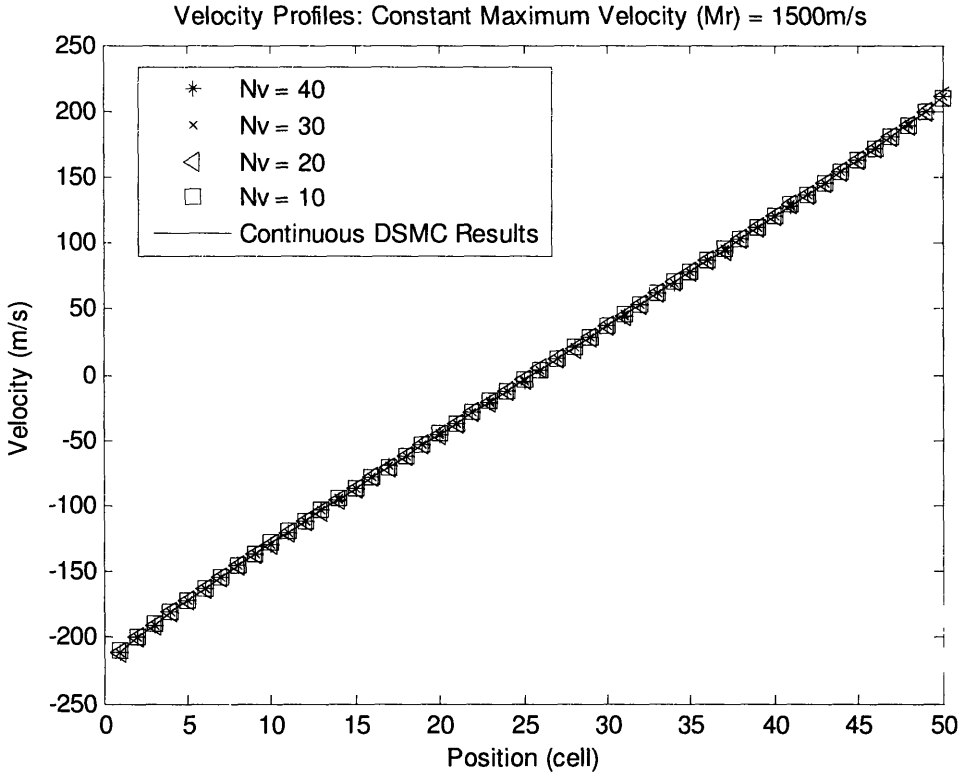

Figure 1: The above plot shows the velocity profiles with mutli dimensional array sizes ranging from N_v = 40 to N_v = 10 in increments of 10; M_r remains constant at 1500m/s.

The uncertainty in flow velocity measurement is approximately 17.3m/s for all simulations.

To further illustrate the results of figure 1, the percent differences between the velocities for different N_v values are plotted below.

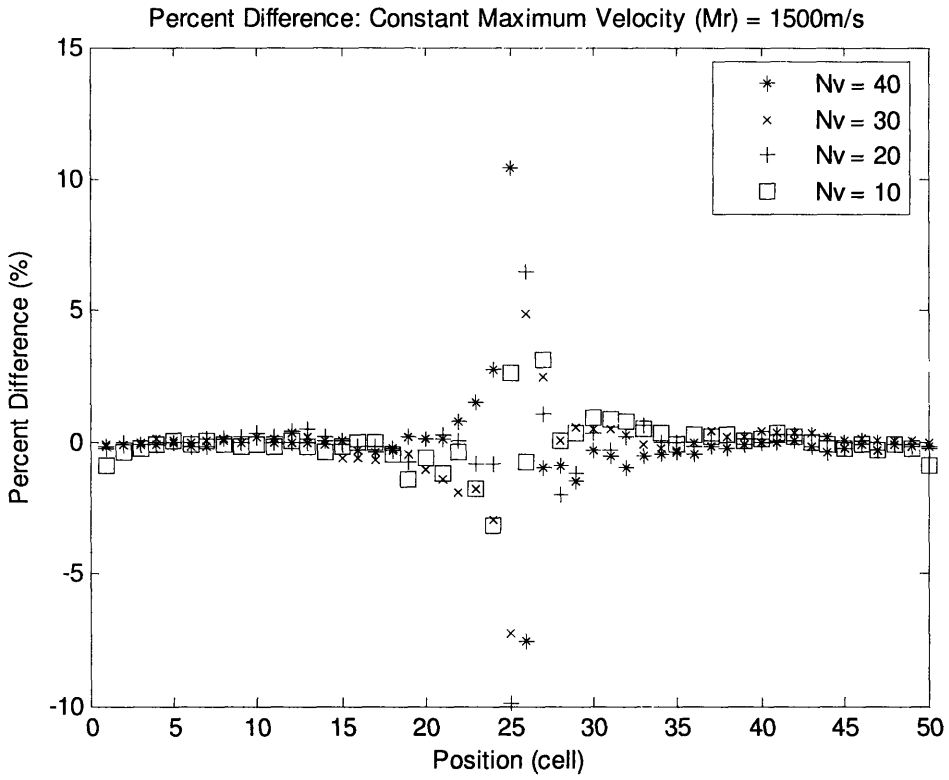

Figure 2: Percent difference between DSMC velocity profiles and IDSMC results for different N_v. The values around cell 25 rise steeply because the DSMC velocity approach zero.

The sharp increase close to cell 25 is because the flow velocity (which is used for normalizing) is close to zero in cell 25.

Below is the plot of shear profiles for constant M_r.

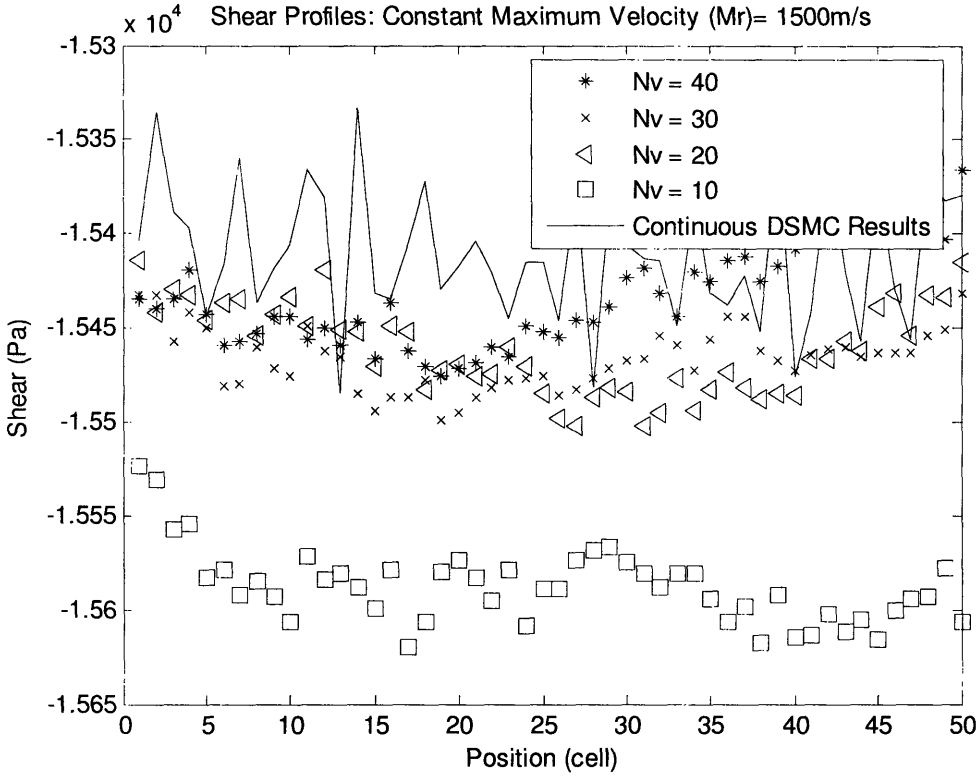

Figure3: Shear stress profiles are plotted above. IDSMC results are shown for constant M_r.

The mean shear values for the respective N_v values from 40 to 10 are: -1.54333e4 Pa, -1.5467e4 Pa, -1.5461e4 Pa, and -1.5588e4 Pa while for the DSMC the mean shear is-1.5411e4 Pa. The uncertainties of the shear values for different three dimensional array sizes from 40 to 10 are: 5.07 Pa, 4.18 Pa, 5.57 Pa, and 3.80 Pa respectively, while for the DSMC the uncertainty is 5.07 Pa.

The next set of data being considered is when step size, δ, is kept constant at 37.5m/s and N_v is varied from 10 to 40 in increments of 10.

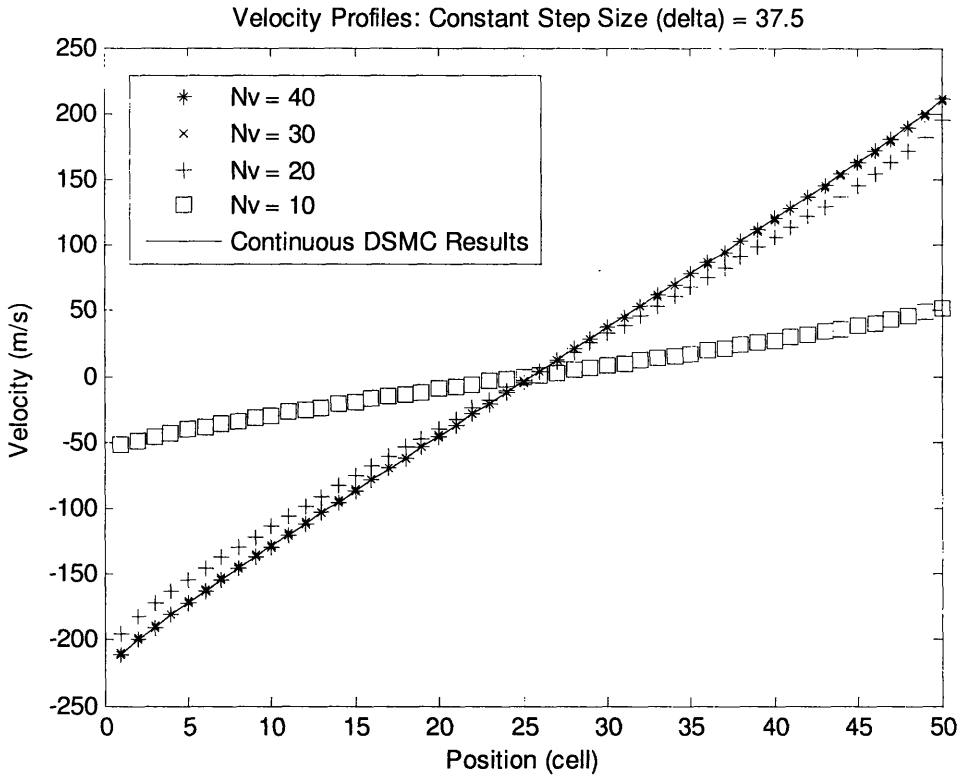

Figure 4: The velocity profiles for the different multi-dimensional array sizes, N_v, with constant step size, $\delta = 37.5$ were compared to DSMC data.

The velocity uncertainties for the different N_v values of 40 to 10 are: 17.28, 17.17, 15.50, and 10.23 m/s respectively, while for the DSMC model the uncertainty is 17.30 m/s.

As above, the percent difference between the flow velocities in each cell for IDSMC simulations and the DSMC simulation were determined to demonstrate the effect of changing the size of N_v while leaving δ constant at 37.5m/s.

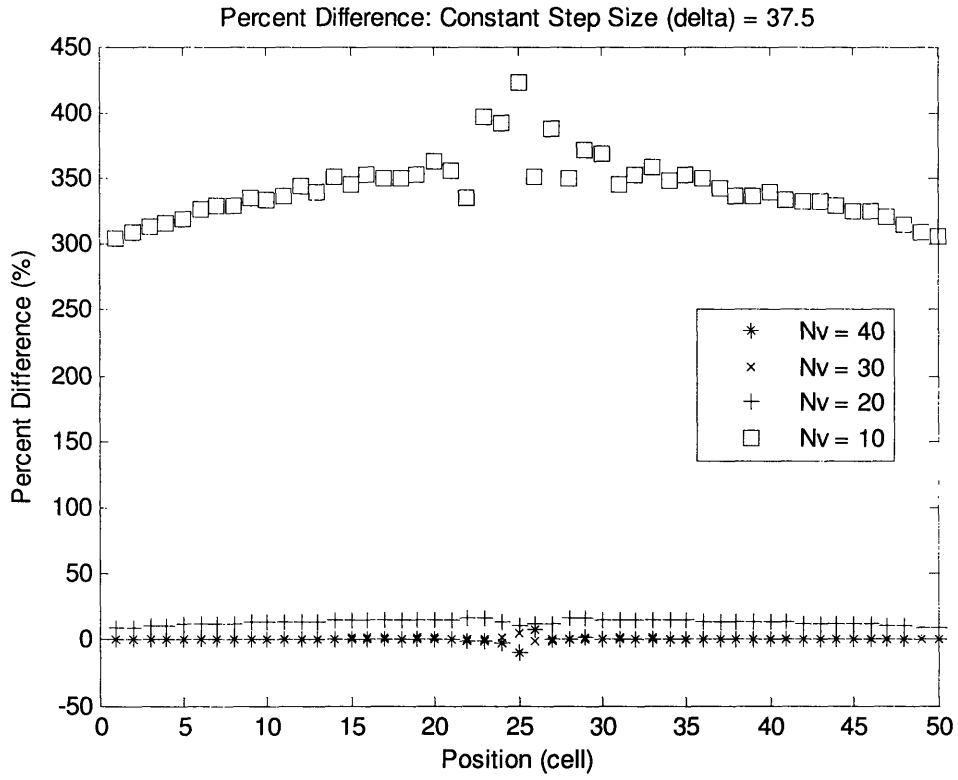

Figure 5: Percent difference between DSMC data and the different N_v values at constant δ. Spikes can be observed in the plots around cell 25 because the flow velocity is close to zero in cell 25.

The shear profiles for the constant δ data sets are plotted below in figure 6.

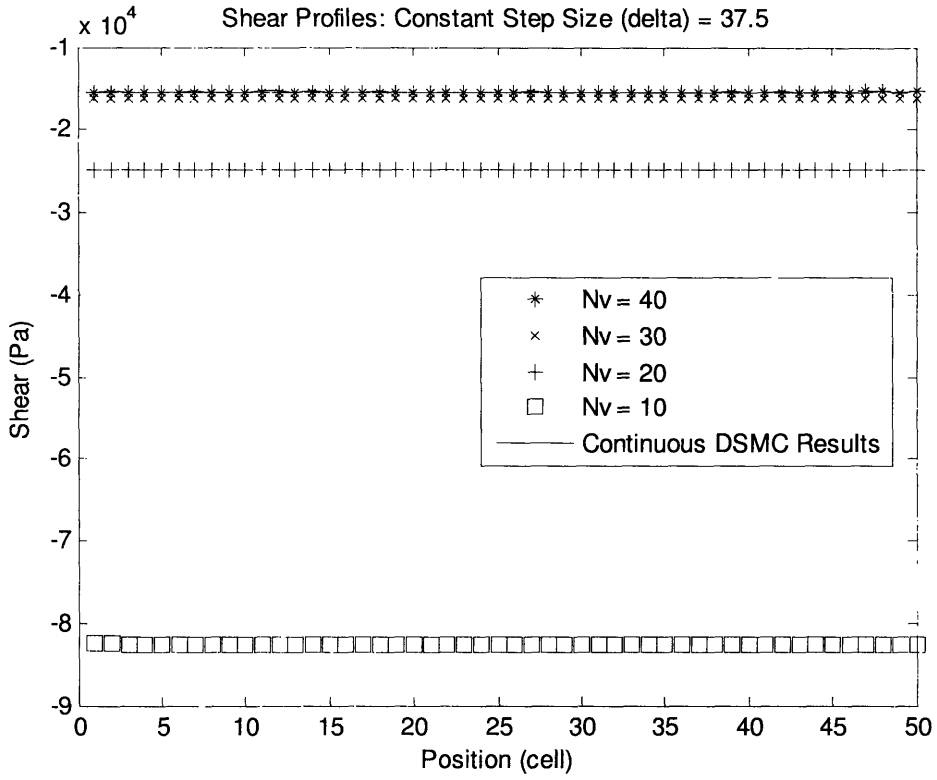

Figure6: Shear profiles over the 50 cells for the different N_v values at constant δ compared to DSMC.

The uncertainties of the shear values for different N_v values of 40 to 10 are: 5.07 Pa, 6.49 Pa, 6.13 Pa, and 4.08 Pa respectively at constant δ, while for the continuous DSMC model the uncertainty is 5.07 Pa. The mean shear values for N_v values of 40 to 10 are: -1.5433e4 Pa, -1.612e4 Pa, -2.4857e4 Pa, and -8.2634e4 Pa, while for the continuous DSMC model the mean shear is-1.5411e4 Pa.

5. Conclusion

For constant maximum velocity, M_r = 1500m/s, N_v did not appear to play as significant a role as when δ remained constant. For the constant M_r data sets the maximum percent difference for mean shear was 1.1% at N_v = 10. Conversely, when δ remained constant the percent difference between the mean shear values for N_v = 30, 20, and 10 are 4.6% , 61.3 %, and 436.2% respectively.

Similar conclusions can be drawn by comparing velocity profiles. The slopes of the velocity profiles appear almost identical for M_r constant and large (figure 1). In contrast, the velocity profiles in figure 4 are very different. The maximum percent error observed for the constant M_r data sets is around 10% which occurs near cell 25 where the velocity values are approaching zero (refer to figure 2). This percent difference appears reasonable and IDSMC can still be considered accurate. The maximum percent difference in figure 5 is, close to 450%, meaning that IDSMC is inaccurate for low M_r.

These results are expected and can be explained by noting that when M_r is small the model does not adequately cover the complete spectrum of possible velocities. In other words, for an accurate result the largest possible relative velocity (and hence also particle velocity) must be sufficiently large (of the order of a few times the speed of sound). Provided this condition is satisfied, the spacing between discrete values appears to have a small effect for the collision dominated simulations we have performed.

References

Francis J. Alexander and Alejandro L. Garcia, "The Direct Simulation Monte Carlo Method," Computers in Physics Vol 11 no 6.,(1997):.

Alejandro L. Garcia, Numerical methods for physics / Alejandro L. Garcia. Upper Saddle River, N.J. : Prentice Hall, (2000) .

Nicolas Hadjiconstantinou, "Dilute Gases and DSMC," Introduction to Modeling and Simulation, 2006, <http://stellar.mit.edu/S/course/3/sp06/3.021J/materials.html>.

Appendix:

Below is a copy of the C++ code for the algorithm that generates the post-collision velocities for the simulation.

```
#include<vector>
#include<iostream>
#include<cmath>
#include"array3d.h"//3dArray Library from Lowell

//www.cppreference.com

using namespace std;


void  CollisionTableGeneratorNew (int rangeV, int rangeDeltaV,
Array3d<vector<vector<int> > >& CollisionTable)
{


        int v1xPrime = 0;
        int v1yPrime = 0;
        int v1zPrime = 0;
        int v2xPrime = 0;
        int v2yPrime = 0;
        int v2zPrime = 0;

        int deltaVx = 0;
        int deltaVy = 0;
        int deltaVz = 0;

        vector <int> velocities;
        velocities.resize(6 , 0);

        //Array3d<vector<vector<int> > > CollisionTable(1+rangeDeltaV*2,
1+rangeDeltaV*2, 1+rangeDeltaV*2);//no longer (3,3,)

        //display variables

        //      int range=3;
        //int NumVectors=0;

        //series of 6 nested loops, loops go through the range of all
discrete values for new velocities with all possibilities of initial
deltaV values this first establishes conservation of momentum
        //after conservation of momentum is established conservation of
energy is checked

        for (int i1 = -rangeV; i1 <= rangeV; i1++) //loop that goes
through range of all v1xPrime values
        {
            cout<<endl<<"IndexValue = "<<i1<<";";

            // int v2xStart;
```

```
            //if(i1 < 0)
            //{
            //  v2xStart=-i1;
            //}
            //else
            //{
            //v2xStart=0;
            //}
                for (int j1 = 0; j1 <= rangeDeltaV; j1++) //loop that
goes through range of all deltaVx values
               {

                   v1xPrime = i1;
                   deltaVx = j1;
                   v2xPrime = deltaVx - v1xPrime;


                       for (int i2 = -rangeV; i2 <= rangeV; i2++) //loop
that goes through range of all v1xPrime values
                       {
                              for (int j2 = 0; j2 <= rangeDeltaV; j2++)
//loop that goes through range of all deltaVx values
                              {
                                  v1yPrime = i2;
                                  deltaVy = j2;
                                  v2yPrime = deltaVy - v1yPrime;



                                        for (int i3 = -rangeV; i3 <=
rangeV; i3++) //loop that goes through range of all v1xPrime values
                                        {
                                               for (int j3 = 0; j3 <=
rangeDeltaV; j3++) //loop that goes through range of all deltaVx values
                                               {
                                                   v1zPrime = i3;
                                                   deltaVz = j3;
                                                   v2zPrime = deltaVz -
v1zPrime;

                                                       //Now Check
Conservation of Energy
                                                       int EnergyIn = (
(deltaVx)*(deltaVx) + (deltaVy)*(deltaVy) + (deltaVz)*(deltaVz) );
                                                       int EnergyOut = (
(v1xPrime-v2xPrime)*(v1xPrime-v2xPrime) + (v1yPrime-
v2yPrime)*(v1yPrime-v2yPrime) + (v1zPrime-v2zPrime)*(v1zPrime-v2zPrime)
);

                                                        velocities[0] =
v1xPrime;
                                                       velocities[1] =
v1yPrime;
                                                       velocities[2] =
v1zPrime;
                                                       velocities[3] =
v2xPrime;
```

```
                                                        velocities[4] =
v2yPrime;
                                                        velocities[5] =
v2zPrime;

        //cout<<velocities[3]<<"value recorded!"<<endl;

                                                        //v1xPrime,
v1yPrime, v1zPrime, v2xPrime, v2yPrime, v2zPrime

                                                        int stop;
                                                        stop=0;
                                                        /*
                                                        if (i1==0 &&
j1!=0)
                                                        {
                                                            stop=1;
                                                        }
                                                        */
                                                        /*
                                                        if (i1==0)// &&
j1==0)
                                                        {
                                                            int vecSize;
                                                            vecSize =
CollisionTable(j1+rangeDeltaV, j2+rangeDeltaV, j3+rangeDeltaV).size();

                                                            for (int index
= 0; index<vecSize; index++)
                                                            {
                                                                int
v1xProxy;
                                                                int
v1yProxy;
                                                                int
v1zProxy;
                                                                int
v2xProxy;
                                                                int
v2yProxy;
                                                                int
v2zProxy;


        v2xProxy=CollisionTable (j1+rangeDeltaV, j2+rangeDeltaV,
j3+rangeDeltaV).at(index).at(0);

        v2yProxy=CollisionTable (j1+rangeDeltaV, j2+rangeDeltaV,
j3+rangeDeltaV).at(index).at(1);

        v2zProxy=CollisionTable (j1+rangeDeltaV, j2+rangeDeltaV,
j3+rangeDeltaV).at(index).at(2);


        v1xProxy=CollisionTable (j1+rangeDeltaV, j2+rangeDeltaV,
j3+rangeDeltaV).at(index).at(3);
```

```
        v1yProxy=CollisionTable (j1+rangeDeltaV, j2+rangeDeltaV,
j3+rangeDeltaV).at(index).at(4);

        v1zProxy=CollisionTable (j1+rangeDeltaV, j2+rangeDeltaV,
j3+rangeDeltaV).at(index).at(5);

                                                                  if
(v1xPrime==v1xProxy && v1yPrime==v1yProxy && v1zPrime==v1zProxy &&
v2xPrime==v2xProxy && v2yPrime==v2yProxy && v2zPrime==v2zProxy)
                                                                    {

        stop=1;
                                                                    }


                                                              }
                                                            }
                                                            */


                                                            if (EnergyIn ==
EnergyOut && stop ==0 )
                                                             {

//cout<<"discrete set found
at("<<deltaVx<<","<<deltaVy<<","<<deltaVz<<")"<<endl;
                                                                CollisionTable
(deltaVx, deltaVy, deltaVz).push_back(velocities);
                                                                //cout<<"entry
in Collisition is "<<CollisionTable (deltaVx+rangeDeltaV,
deltaVy+rangeDeltaV, deltaVz+rangeDeltaV).at(0).at(3)<<endl;

                                                                //double check
Energy and Momentum!


                                                                int Ein;
                                                                int Eout;
                                                                int xMomIn;
                                                                int yMomIn;
                                                                int zMomIn;
                                                                int xMomOut;
                                                                int yMomOut;
                                                                int zMomOut;

                                                                Ein=  (
(deltaVx)*(deltaVx) + (deltaVy)*(deltaVy) + (deltaVz)*(deltaVz) );
                                                                Eout=
v1xPrime*v1xPrime +v1yPrime*v1yPrime + v1zPrime*v1zPrime +
v2xPrime*v2xPrime + v2yPrime*v2yPrime +v2zPrime*v2zPrime;

                                                                xMomIn=
deltaVx;
                                                                yMomIn=
deltaVy;
```

```
                                                                zMomIn=
deltaVz;


xMomOut=v1xPrime+v2xPrime;

yMomOut=v1yPrime+v2yPrime;

zMomOut=v1zPrime+v2zPrime;

                                                                if(Ein!=Eout)
                                                                {

        cout<<endl<<"Energy is not Conserved!"<<endl<<"Energy In  =
"<<Ein<<endl<<"Energy Out = "<<Eout;
                                                                }

                                                                if(xMomIn !=
xMomOut ||  yMomIn != yMomOut ||  zMomIn != zMomOut )
                                                                {

        cout<<endl<<"Energy is not Conserved!"<<endl<<"Energy In  =
"<<Ein<<endl<<"Energy Out = "<<Eout;
                                                                }

                                                            }

                                                        }
                                                    }
                                                }
                                            }
                                        }
        }


        //_____________________________________________________________dis
play_____________________________________________________

        //display
        /*
         int NumVectors;
        int range;

       int k1;
       int k2;
       int k3;
       k1 =0;
       k2 =1;
       k3 =0;

       range = rangeV;

        */


/*
       cout<<endl<<"Number of Vecttors at location: ";
```

```
       NumVectors = CollisionTable(k1+range, k2+range,
k3+range).size();
       cout<<NumVectors;



       cout<<endl;
       for (int k4 = 0; k4 <=  NumVectors-1; k4++)
       {
           cout<<endl;
           cout<<"v1 Prime = (" <<CollisionTable(k1+range, k2+range,
k3+range).at(k4).at(0);
           cout<<", " <<CollisionTable(k1+range, k2+range,
k3+range).at(k4).at(1);
           cout<<", " <<CollisionTable(k1+range, k2+range,
k3+range).at(k4).at(2);
           cout<<")" <<endl<<"v2 Prime = " <<CollisionTable(k1+range,
k2-range, k3+range).at(k4).at(3);
           cout<<", " <<CollisionTable(k1+range, k2+range,
k3+range).at(k4).at(4);
           cout<<", " <<CollisionTable(k1+range, k2+range,
k3+range).at(k4).at(5);
           cout<<")";
           cout<<endl;
       }
        */

        /*
        for (int k1 = -range; k1 <= range; k1++)
        {
            for (int k2 = -range; k2 <= range; k2++)
            {
                for (int k3 = -range; k3 <= range; k3++)
                {
                    //cout<<endl;
                    //cout<<"vectors at
("<<k1<<","<<k2<<","<<k3<<")"<<endl;
                    NumVectors = CollisionTable(k1+range, k2+range,
k3+range).size();
                    //cout<<"Number of Vectors of
vectors:"<<NumVectors<<endl;

                    for (int k4 = 0; k4 <=  NumVectors-1; k4++)
                    {
                        if (k1==1 && k2==2 && k3==3)
                        {

                                cout<<endl<<"v1 Prime = ("
<<CollisionTable(k1+range, k2+range, k3+range).at(k4).at(0);
                                cout<<", " <<CollisionTable(k1+range,
k2+range, k3+range).at(k4).at(1);
                                cout<<", " <<CollisionTable(k1+range,
k2+range, k3+range).at(k4).at(2);
                                cout<<")   v2x Prime = ("
<<CollisionTable(k1+range, k2+range, k3+range).at(k4).at(3);
                                cout<<", " <<CollisionTable(k1+range,
k2+range, k3+range).at(k4).at(4);
```

```
                                cout<<", " <<CollisionTable(k1+range,
k2+range, k3+range).at(k4).at(5);
                                cout<<")";

                                        //      cout<<"x1 Prime = "
<<Table(i1+range, i2+range, i3+range).at(i4).at(1);
                                        //cout<<"x1 Prime = "
<<Table(i1+range, i2+range, i3+range).at(i4).at(2);
                                        //cout<<"x1 Prime = "
<<Table(i1+range, i2+range, i3+range).at(i4).at(3);
                                        //cout<<"x1 Prime = "
<<Table(i1+range, i2+range, i3+range).at(i4).at(4);
                                        //cout<<"x1 Prime = "
<<Table(i1+range, i2+range, i3+range).at(i4).at(5);
                                cout<<endl;
                        }

                    }
                }
            }
         }

        cout<<endl;
        */




        return;
}
```