An Analysis of Direct Simulation Monte Carlo And Its Application to Simulating Supersonic Shockwaves

New Mexico Supercomputing Challenge

Final Report

April 2008

Team 49

Erika DeBenedictis

Saint Pius X High School

Tony Huang

La Queva High School

Project Mentors

Dr. Erik DeBenedictis

Dr. Dan H. Huang

Dr. Michail Gallis

Table of Contents

1. Introduction

When modeling objects moving through a medium it is important to choose a simulation method suited to the problem to ensure the accuracy and efficiency of the simulation. For problems dealing with objects moving through the atmosphere at high altitudes, Direct Simulation Monte Carlo (DSMC) is an appropriate method. DSMC differs from the more frequently used molecular dynamics simulators or fluid models in that it deals with the overall movement of gas as a whole. This allows DSMC to simulate larger regions and more particles, making it the method of choice for simulations in the upper atmosphere.

2. Executive Summary

In this project, we simulate the movement of objects through the upper atmosphere with DSMC, especially focusing on the shockwaves these objects create at supersonic speeds. Supersonic shockwaves in rarified gas are slightly different from those in a fluid, and have a significant effect on the resistance on the object.

In order to accurately simulate this phenomenon, a DSMC program was written and evaluated for both coding and physical accuracy through a series of increasingly demanding tests. It utilizes multi-core processors to address this compute-intensive problem and has demonstrated the wake paths and shockwaves of various objects moving through the upper atmosphere.

3. Problem Statement

How can shockwaves in rarified gas be accurately and efficiently simulated?
Our steps in achieving this goal are:

- Develop a DSMC simulation for Earth's atmosphere at an altitude with rarified gas.
- Validate the simulation for both physical and coding accuracy through test simulations and data analysis.
- Adapt and analyze the simulation for use on multi-core processors.
- Model supersonic gas flow and apply it to a real world problem, observing the unique behavior of shockwaves in rarified gas, and analyzing the results.

4. Overview of Direct Simulation Monte Carlo Methods

4.1 Modeling Gas Particles and Collisions

Direct Simulation Monte Carlo, commonly referred to as DSMC, is a 3-D simulation method that is often used to simulate low density regions such as the upper atmosphere. In DSMC, the simulation region is broken up into many subsections, known as cells. Each cell contains particles that have both velocity and position vectors that define their movement. However, the interactions between particles are not dependent on the position vector, which is used purely to define the particle's trajectory. Particle-particle interactions may only occur if two particles are within the same cell.

When simulating a region, the probability for a collision is applied to all particles in a cell. Within that cell, a certain number of collisions will be performed and new velocity vectors will be assigned to the particles that collide. The determination of a collision is based on the physical and chemical properties of gases. Therefore, the results of collisions will show physical properties, especially those pertaining to density, such as diffusion of particles.

4.2 Knudsen Numbers

The Knudsen number is defined as the ratio between the mean free path, or average travel distance between successive collisions, and the representative length of the simulated object. For example, the Knudsen number for an object that is four times the mean free path would be 0.25. It is the Knudsen number that controls the appropriate model and parameters for that model because it gives the relationship between the density of the region and the size of the simulated object. If we look at an area with very high density, and therefore a small mean free path, the simulated object would most likely be small. By using the same Knudsen number for an area that is not as dense, we could simulate a larger object.

Choosing a smaller Knudsen number for the simulation results in a more accurate simulation, but the compute intensity involved in the simulation would increase drastically. On the other hand, a large Knudsen number would have greater inaccuracies. The Knudsen number 0.1 is commonly used because it gives a balance between accuracy and compute intensity (Gallis). However, depending on the simulation, accuracy and compute time can be exchanged accordingly.

5. Comparison of DSMC to other methods

DSMC methods are particularly suited to researching the characteristics and behavior of large regions with low particle densities. There are differences between DSMC and traditional molecular and fluid dynamics methods.

DSMC vs. Fluid Dynamics

The area modeled by DSMC is usually rarefied gas, such as the gas found in the upper atmosphere. In these cases, the density of the gas is very low. Furthermore, the collision by two gas molecules does not always affect other molecules because the average distance between molecules is so large. Therefore, the gas does not act like a fluid, where the interaction range between molecules is larger than the average distance between molecules.

DSMC vs. Molecular Dynamics

DMSC allows for a much larger area to be simulated. Since the position of gas particles does not affect collisions in DSMC, (we only keep track of the cell number that a particle is in to determine collisions), it is much more efficient to simulate large amounts of particles. This gives DSMC a significant advantage over the compute intensive molecular dynamics models for large regions containing many particles.

All three of these models are effective for certain tasks. Fluid dynamics is most effective in high density environments where the gas behaves as a fluid. Molecular dynamics is useful for small sized simulations and can simulate reactions, especially chemical, at the molecular level. DSMC methods are practical under conditions involving a large simulation region containing low densities.

6. Physical Mechanisms of Simulation

Equations in 6.1 – 6.4 of this section may be found in Mueller's "Direct Simulation Monte Carlo." Equations in 6.5 and 6.6 are from Nave's "Kinetic Theory Concepts."

6.1 Deriving the number of maximum collisions

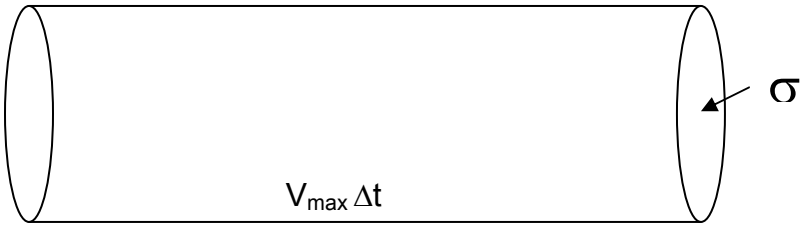
6.1.1 Finding the number of particle pairs

As previously mentioned, the DMSC model treats collisions differently than conventional molecular models. Instead of tracing the orbital equations of each individual particle and checking if the paths of two particles cross in each time step, we look at a cell. Within that cell, we have Nc particles. Next we need to find how many pairs of particles there are in a cell. We do this by taking NcCr2.

6.1.2 Finding probability of collisions

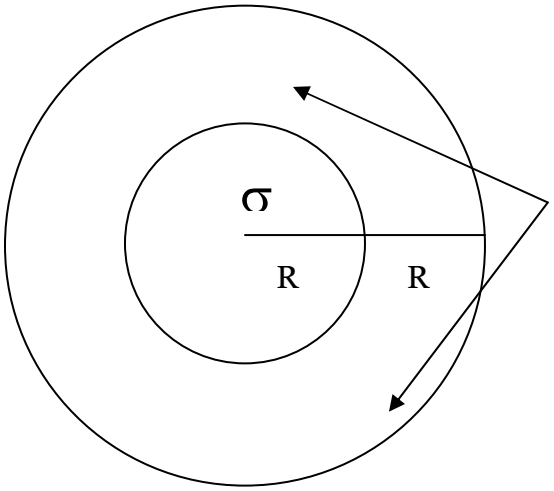
However, not every pair of particles will interact in the simulation. We can narrow down the amount of colliding particles by estimating probability for a collision to happen within a cell. This is done by comparing the volume available for collisions to the volume of the entire cell. First, we need to find the total amount of volume covered by one particle in one time step. This can be found by multiplying the cross-sectional area σ by the maximum distance covered by the particle in one time step.

Distance in one time step = $V_{max}\,\Delta t$



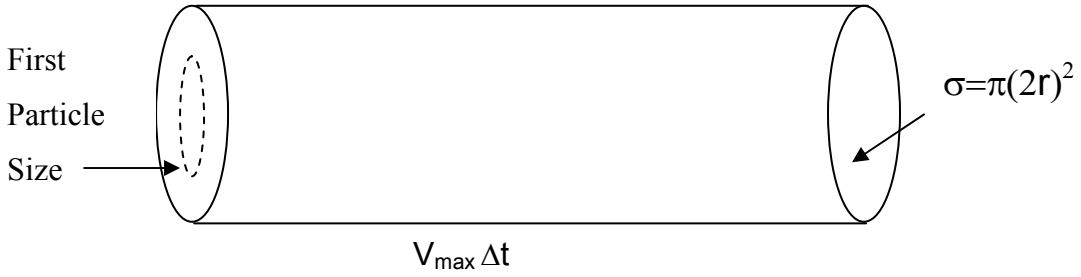
In this equation, V_{max} represents the maximum relative speed a particle can have. In our simulation, V_{max} is found by testing the velocity of the sampled particles from time to time. We take the maximum velocity found and multiply it by 3 to ensure that we have accounted for the highest velocity in the simulation and in turn, the maximum number of collisions possible in one time step.

In order for another particle to collide with this one, the cross-sectional area for the second particle must intersect with the cross-section of the first particle. If the two cross-sections are overlapping or touching, there will be a collision within the next time step.



The center of the second particle must be within this region in order for a collision to occur between the two particles.

From this diagram, we can see that the possible volume for a collision to take place has a cross section of twice the diameter of a particle. The distance that the two particles can travel remains $V_{max}\,\Delta t$. Therefore, the collision volume looks like



If we proceed to divide this collision by the total volume of the cell, we get an estimated probability of a collision occurring between a pair of randomly selected particles.

$$P(c) = \frac{V_{\max}\Delta t(4\pi R^2)}{V_c}$$

6.1.3 Finding the number of maximum colliding particle pairs

Finally, we multiply this rough probability estimate by the number of pairs of particles in a particular cell to determine the number of potential collisions in that cell. Since V_{max} was used

to calculate the volume swept out by a particle, we can ensure that the number of collisions calculated will be the maximum.

Therefore, the maximum number of collisions in a cell with Nc particles will be

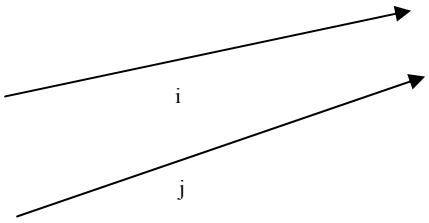
$$M_c = \frac{V_{\max} \Delta t (4\pi R^2)}{V_c} [\frac{N_c (N_c - 1)}{2}]$$

However, we still need to go through each of these "possible collisions" and determine which collisions to actually perform.

6.2 Choosing Colliding Particles

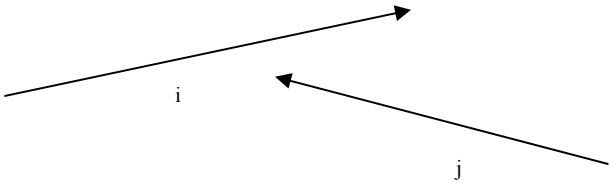
After we derive the maximum number of collisions in a cell within a time step, we must choose which collisions to perform. Not every pair of particles will collide, and we apply an acceptance-rejection method to determine whether the particles do indeed collide for that time step. The actual probability of a pair of particles colliding is dependent on the relative velocity of the two particles. The relative velocity between particles i and j is given by the simple formula $V_{rel} = \left| v_i - v_j \right|$.

If the relative velocity between two particles *i* and *j* is very small, it implies that the two velocity vectors are almost identical. Therefore, there will be a much smaller chance that these two particles will collide.



Velocity vectors *i* and *j* have a small relative velocity.

On the other hand, two particles with a large relative velocity are more likely to collide because their velocity vectors differ drastically.



Velocity vectors *i* and *j* have a large relative velocity.

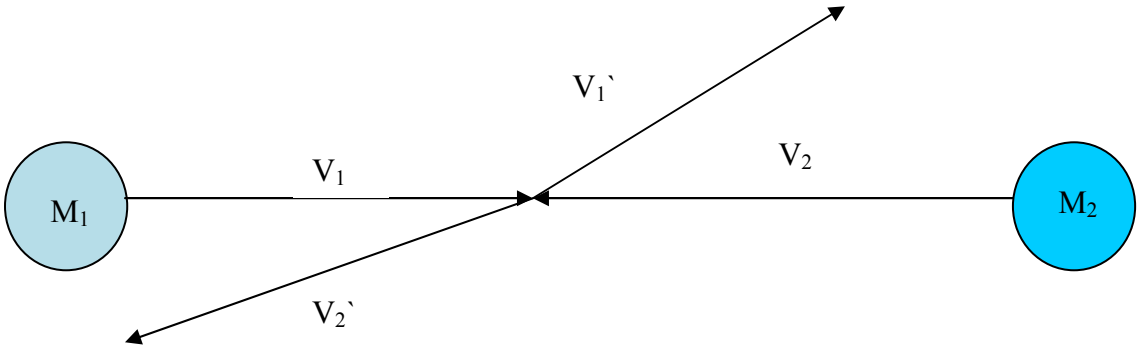
We can now introduce a method to randomly collide particles with a probability depending on their relative velocity. Two particles *i* and *j* will collide if the following equation is satisfied. Here, *Z* is a randomly distributed real number between 0 and 1.

$$\frac{|v_i - v_j|}{v_{\max}} > Z$$

We roll a new *Z* for every new pair of particles. Z is independent of any other variables in the simulation.

6.3 Deriving Post-Collision Velocities

For two particles that have been chosen to collide, we can calculate their post-collision velocities from their pre-collision velocities. The calculation of the post-collision velocities is based on two physical principles that we can apply to the DSMC model.



I. Conservation of Total Momentum

$$m_1 v_1 + m_2 v_2 = m_1 v_1' + m_2 v_2' \qquad (1)$$

II. Elastic Collisions (Conservation of Total Kinetic Energy)

$$\frac{1}{2}m_1 v_1^{\ 2} + \frac{1}{2}m_2 v_2^{\ 2} = \frac{1}{2}m_1 v_1^{\ \prime 2} + \frac{1}{2}m_2 v_2^{\ \prime 2} \qquad (2)$$

If we assume that there is no energy lost due to thermal diffusion, then it is safe to also assume that collisions will be elastic in the DMSC model. By multiplying both sides of Equation (2) by 2 and then rearranging, we get

$$m_1 v_1^{\ 2} - m_1 v_1^{\ \prime 2} = m_2 v_2^{\ \prime 2} - m_2 v_2^{\ 2}$$

We proceed to simplify both sides

$$m_1(v_1^{\ 2} - v_1^{\ \prime 2}) = -m_2(v_2^{\ 2} - v_2^{\ \prime 2})$$

We now divide both sides by $m_1(v_1 - v_1^{\ \prime}) = -m_2(v_2 - v_2^{\ \prime})$

This gives rise to $v_1 + v_1^{\ \prime} = v_2 + v_2^{\ \prime}$

We can now multiply this equation by m_2 on both sides to generate Equation 3.

$$m_2 v_2 - m_2 v_1 = m_2 v_1^{\ \prime} - m_2 v_2^{\ \prime} \qquad (3)$$

We can add Equation 3 to Equation 1 to generate a new equation

$$v_1^{\ \prime}(m_2 + m_1) = (m_1 v_1 + m_2 v_2) + m_2(v_2 - v_1) \qquad (4)$$

Finally, we can solve for one new velocity after the collision.

$$v_1^{\ \prime} = \frac{(m_1 v_1 + m_2 v_2)}{(m_2 + m_1)} + \frac{m_2}{(m_2 + m_1)}(v_2 - v_1)$$

For any system of many particles, the velocity of the center of mass is given by the sum of all the total momentum in the system divided by the total mass of the system. In other words, V_{cm} for an n-body system is given by:

$$v_{cm} = \frac{(m_1 v_1 + m_2 v_2 \ldots + m_n v_n)}{(m_1 + m_2 \ldots + m_n)}$$

For a two-body system as we have in the simulation, the velocity at the center of mass is given by:

$$v_{cm} = \frac{(m_1 v_1 + m_2 v_2)}{(m_1 + m_2)}$$

Therefore, we can rewrite the equation for the post-collision velocity.

$$v_1' = v_{cm} + \frac{m_2}{(m_2 + m_1)}(v_2 - v_1) \tag{5}$$

Using Equation(1), we can solve for the second post-collision velocity vector.

$$v_2' = v_1 + v_1' - v_2$$

$$v_2 = v_{cm} - \frac{m_2}{(m_2 + m_1)}(v_2 - v_1) \tag{6}$$

Finally, we assign random unit sphere direction vectors ($\vec{e}$) to the two resultant velocities. This will give us two final equations for the resultant velocities of two particles in a collision. In these equations, *e* represents the random unit vector.

$$\vec{v}_1' = \vec{v}_{cm} + \frac{m_2}{m_1 + m_2}\left|\vec{v}_2 - \vec{v}_1\right|\vec{e}$$

(5b)

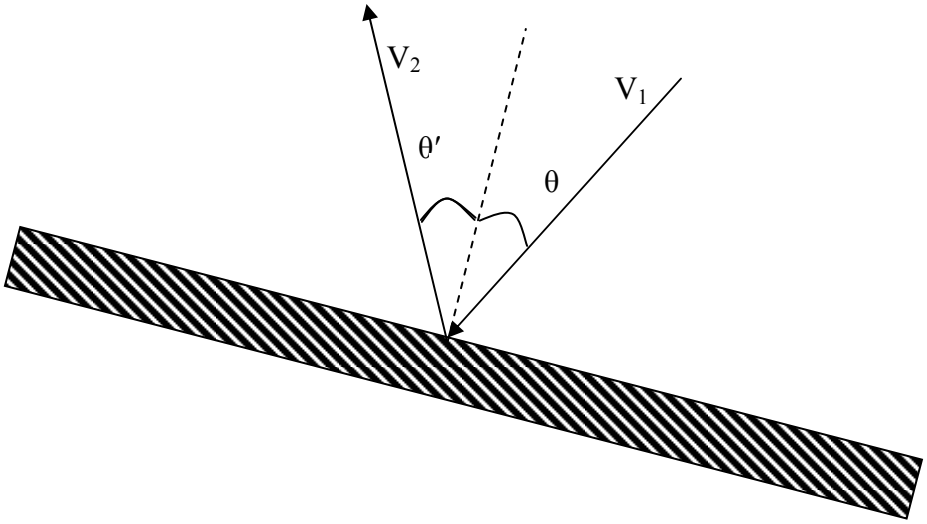
$$\vec{v}_2' = \vec{v}_{cm} - \frac{m_2}{m_1 + m_2}\left|\vec{v}_2 - \vec{v}_1\right|\vec{e}$$

(6b)

6.4 Determining Particle-Object Collisions

When a particle collides with a macro-sized, the equations governing the collision are different. In this case, the mass of the object can be considered to be infinite because the mass of the particle is insignificant compared to the mass of the object. The total energy and momentum of the particle is still conserved.

The conservation of momentum along the surface of the object requires that $V_{1\|} = V_{2\|}$. In other words, the angle (θ) of incidence for the particle will be equal to the angle (θ′) that the particle is reflected at. The basis for the determination of the angle is the normal to the barrier.



Due to the conservation of energy and momentum, the speed of the particle does not change from incidence to reflection. However, the particle will take on a new direction reflected by the surface.

6.5 Finding Thermal Velocity of Particles

All particles have a thermal velocity due to finite temperature of atmospheric gas. The thermal velocity for classical equilibrium is assigned according to the Maxwell-Boltzmann speed distribution. The distribution will be dependent on several factors in the atmosphere, including the molar mass of the particles and the temperature of the system. The distribution is as follows:

$$f(v) = 4\pi\left(\frac{M}{2\pi RT}\right)^{\frac{3}{2}} v^2 e \frac{-Mv^2}{2RT}$$

In this equation, f(v) is the distribution of the velocities (v). M represents the molar mass of the simulators (kg/mol) and T defines the temperature of gas (K). Finally, R is the gas constant (8.3145 J/mol K).

In the simulation, we initially assign speeds of particles based on the equilibrium Boltzmann distribution to speed up convergence. The directions of the particles are randomly assigned.

6.6 Determining Mean-Free Path and Cell Size

In our simulation, we assign our cell size according to the mean free path. It is commonly accepted that the size of the cells should be smaller than the mean free path in order to gain accuracy when counting collisions. The mean free path is defined as the average distance a particle will travel before a collision with another particle occurs. Another way to look at the mean free path is the average speed of the particles divided by the average number of collisions per second, or collision rate. This collision rate can be found by multiplying the average number of particles per unit volume (n_v) by the amount of space available for collisions.

$$\Gamma = n_v(v_{rel}\pi d^2)$$

We use the relative velocity to calculate the volume swept out in a time step because other particles are also moving in the cell. Therefore, it would be inaccurate to use the average molecular velocity. The relative velocity can be calculated to be the square root of 2 times the average molecular velocity. Now, we can define the mean free path as:

$$\lambda(M.F.P.) = \frac{\bar{v}}{\sqrt{2}\pi d^2 \bar{v}(n_v)} = \frac{1}{\sqrt{2}\pi d^2(n_v)}$$

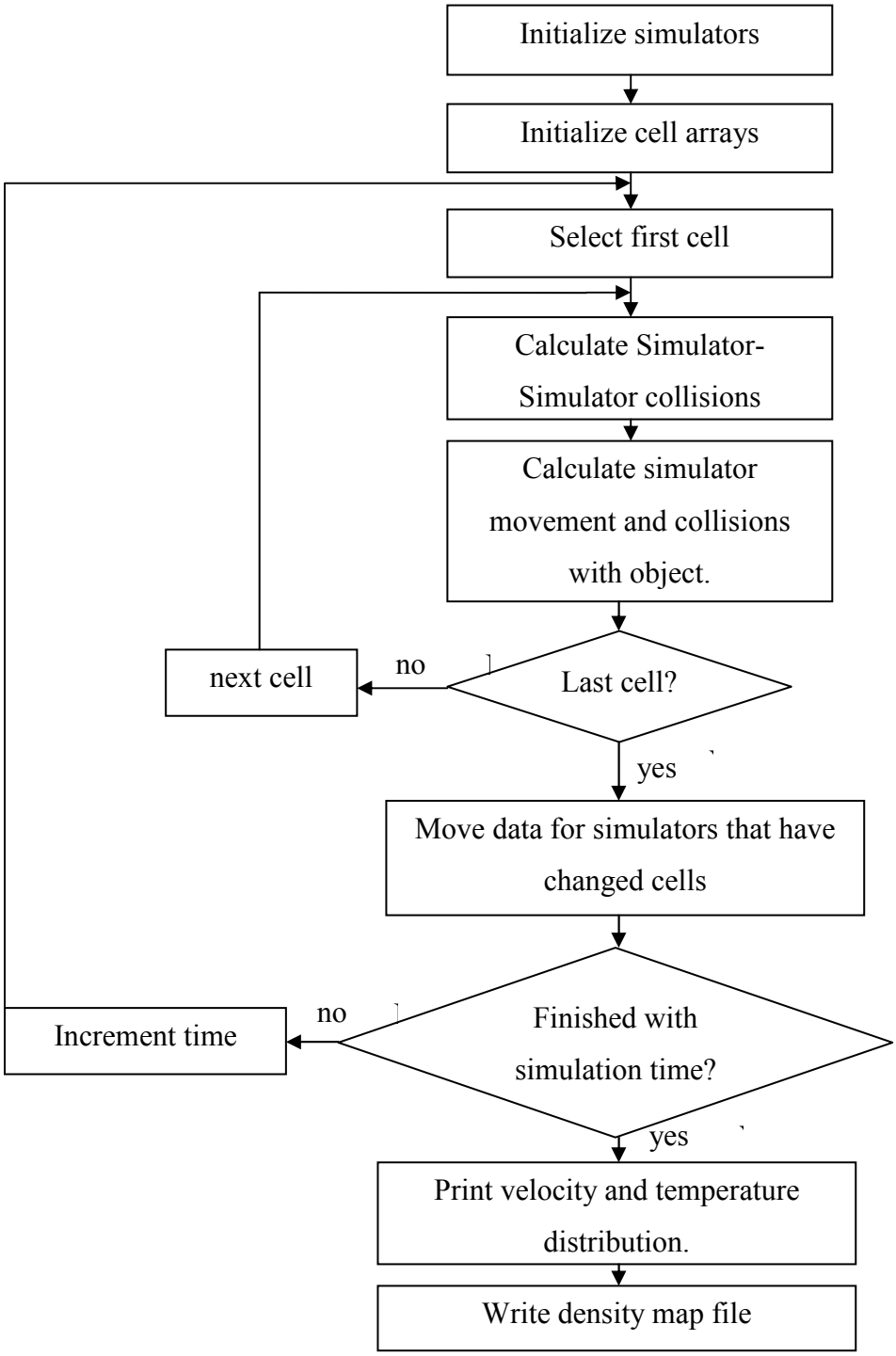
From this equation, we can see that the mean free path of a particle is dependent on temperature, pressure, and the size of the particles since those factors will affect n_v.

The cell size can be chosen to best accommodate the application. A smaller cell size would result in more accurate collisions, but lower efficiency and more intensive computing. On the other hand, a larger cell size would result in less accurate collisions. However, the cell size should always be less than the mean free path. From our calculation, the cell size should be approximately one-third of the mean free path.

From the mean-free path, we can then calculate average collision time T. This is given by the mean free path divided by the average molecular velocity. Our time step Δt should be much smaller than the average collision time. In general, the optimal ratio between time step and collision time is one tenth, which we use in our simulations.

7. Program Description

7.1 Flowchart



7.2 Code Description

The first main goal of this project was the development of a DSMC program. The program was written from scratch in C++, is 3D, can be run on multicore computers, and uses dynamic array allocation to accommodate large scale simulations.

Simulator data is stored in two ways. One structure, `fmol`, is used when initially setting the positions and velocities of the particles when the cell that the particles are in is not yet known. It is also used as an intermediary for moving particles between cells and re-setting particles when they exit the simulation region. The primary data structure for particle data is `postroster`, which contains the data for all the simulators as sorted by their cell. All cells begin the simulation with room to hold zero simulators. As more simulators enter the cell, each cell allocates more room to accommodate them, in increments of 20 simulators. If a simulator moves out of a cell its data is moved to the new cell, and the simulator at the end of that cell's list replaces the vacated slot, allowing the data to remain compact.

This memory grouping method stops cells that continuously have low density from wasting memory, making the code memory more efficient. The computer will also spend less time accessing memory in this configuration because many particles that must be calculated in order will be in the cache line if they are physically grouped together, greatly decreasing compute time. Each simulator requires 60 bytes, allowing thousands of particles to be stored in the cache at one time without accessing memory. For example, on the laptop used as a benchmark, the cache size is 256 KB per core, which could potentially hold 4096 particles. Of course, some amount of this data will be filled with local variables, so the actual number will be somewhat smaller. Consider that if simulators were not stored in order of their cell, processors would have to fetch particle data for collisions from a list that was completely unordered, requiring these processors to access data in main memory almost every time.

Simulators are initialized with the function `msc2` with velocity dependent on the thermal energy of a gas at a specific altitude and temperature as dependent on the Maxwell-Boltzmann speed distribution. `Dataset()` takes these particles and initializes the `postroster` array for holding the simulator data.

There are two primary functions for calculating the movement and collisions of simulators. `Molmolcollide` calculates collisions between molecules using the methods described. `movemolobj` calculates collisions between the simulators and the boundaries. Each

boundary is defined as an instance of the structure line1, which contains the beginning and end coordinates of this line segment. By defining these segments on the simulation region we can create the shape of an object in the region. Both the boundaries and the simulator's beginning and end positions for that timestep are converted into line segments describing their movement. The boundaries are sorted so they are in order of when the simulator's line crosses them and then the function checks to see if the simulator line segment and the line segment of the first boundary cross. If so, the simulator has hit that wall. Its velocity is reflected out with the same angle it hit the boundary with and its position is moved to the intersection point. The process begins again to see if it collides with any other boundaries, discarding the boundary it has just hit.

This method is slightly different than the one commonly used, such as in G. A. Bird's book. Bird prefers to slightly move the simulator off the boundary it has just hit rather than temporarily discarding collisions with that boundary. While it is necessary to make sure that a particle resting on the boundary from a previous collision is not mistaken for colliding again, Bird's method tends to add some small force away from boundaries. For many delicate calibration tests, this force becomes significant, and therefore Bird's methods not used. However, in supersonic tests, such as the ones Bird performs most frequently, the force is insignificant.

When run on multiple cores, each processor is assigned a range of cells to calculate, making the inner loop parallel. All the processors record a list of simulators that must move cells before the next time step in `MoveMolsArray`, and at the end of the calculation of collisions and movement, one processor moves these simulators within `postroster`. `molmolcollide`, `movemolobj`, and the random number generator have all been modified so they are thread safe.

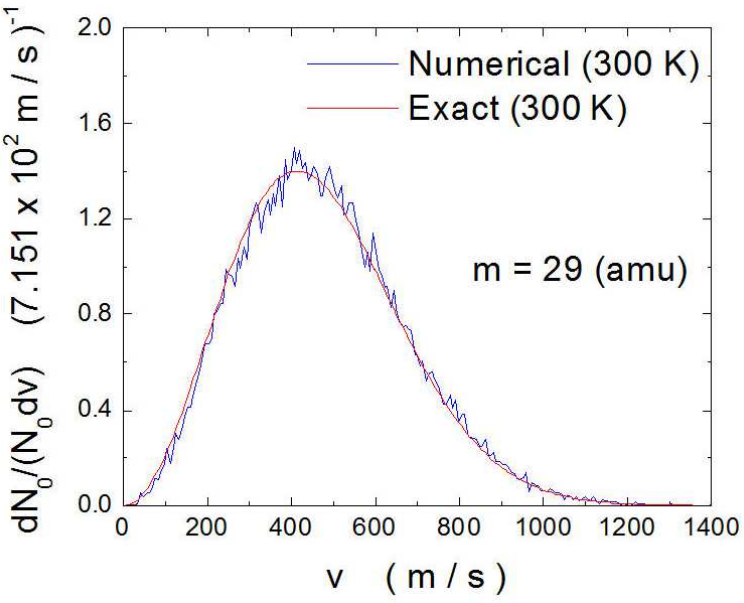
On each timestep, the number of simulators in each cell is added to the array `densityindex`. This array is used to create a gif image that shows the density and velocity map over the simulation region. At the end of the simulation some information parameters are displayed, such as the total number of particle collisions, the number of simulator-boundary collisions, and a histogram of the size of each cell's memory.

8. Calibrating the Simulation

In order to ensure that the simulation is physically realistic, it is necessary to validate the model through a series of tests. The goal is a realistic representation of gas behavior at some atmospheric altitude and therefore must be calibrated for the temperature, velocity, pressure, and density at that altitude. The calculation of gas movement in the simulation must also be realistic to gas behavior, and must not gain or lose energy through collisions between simulators or with objects in the simulation region. Accurately assembling a DSMC program is further complicated by the calibration of the Knudson number, cell size, and simulation region. The tests described below show how we have validated our model.

8.1 Gas in free-flowing steady state

As a preliminary test for calibrating accurate velocities for a given altitude as well as accurate particle-particle collision calculations, we simulated free flowing gas in a steady state without any bounds. In this simulation, the gas flows in from one side and can escape from any four sides. There are no barriers or boundaries involved. The initial conditions agree with the temperature distribution for 300K. After the flow had reached a steady state, where the simulators collide with each other and are evenly cycled through the region, we analyzed the velocity distribution, as shown below.



A comparison of exact and calculated values of speed distribution.

The experimental data we collected from the program matches the equilibrium Boltzmann distribution for the same temperature. This validates the accuracy of our model for particle-particle collisions.

8.2 Equilibrium Box Test

This test is one of the best for verifying a wide range of aspects of our model. Passing this test requires correct molecule-molecule and molecule-object collisions, a balance between particles and cell size, and an efficient and accurate data structure. In the box test, the gas is simply put in a bounded area, initialized with the Boltzmann distribution. The simulators only perform elastic collisions with each other and the bounds of the region. There should be no change in total energy of the system, making this a very good test of the physical accuracy of the collision equations.

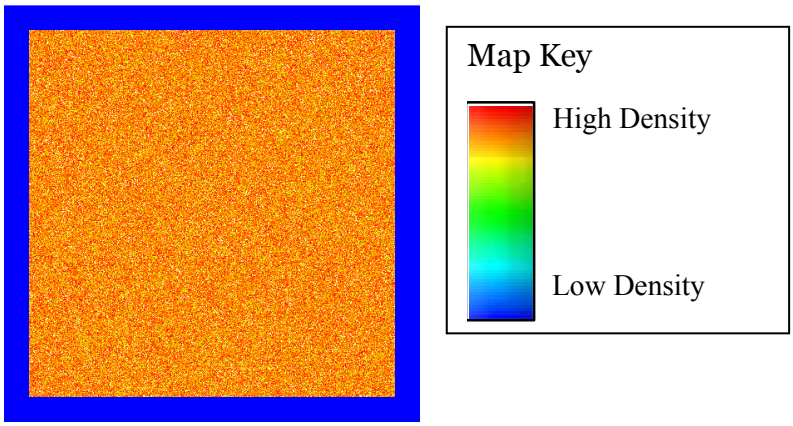
This is also a good test of the efficiency of the data structures and the multi-core operation of the code. In many simulations, the simulators will tend to stay in some region of the area, which reduces the time cost of re-ordering particles.

For example, in the simulation of an object traveling through the region, simulators would tend to follow the direction of the flow, allowing them to stay on the same processor if the region is divided along the flow line. In the box simulation, particles will move throughout the box, causing them to change processors. This demonstrates the advantage of multi-core simulations over the traditional 'node-based' computing in this aspect.

This test was also used to calibrate the parameters of the program to an actual altitude. At 90 km the temperature is 180 K, the pressure is 0.0019 mmHg, and the mean free path is 0.0245415 m, the molecule radius (for normal air) is $3*10^{-10}$ m, and the mass is $4.81727*10^{-26}$ kg (Nave). The total simulation region was 5 meters, with 500 cells in each direction, each being .01 meters in length, or about .4 of the mean free path.

According to the theoretical values, there should be 14771.5 collisions per second for each molecule at this altitude, which corresponds to one collision every .000067698 seconds (Nave). Since we want only a tenth of the particles to collide each time step, the time step is .0000067698 seconds, or one tenth of the theoretical collision time. The total simulation time is .001 seconds.

Below is the density map for this simulation. As shown, all of the simulators stay in the box, and the distribution is essentially uniform.



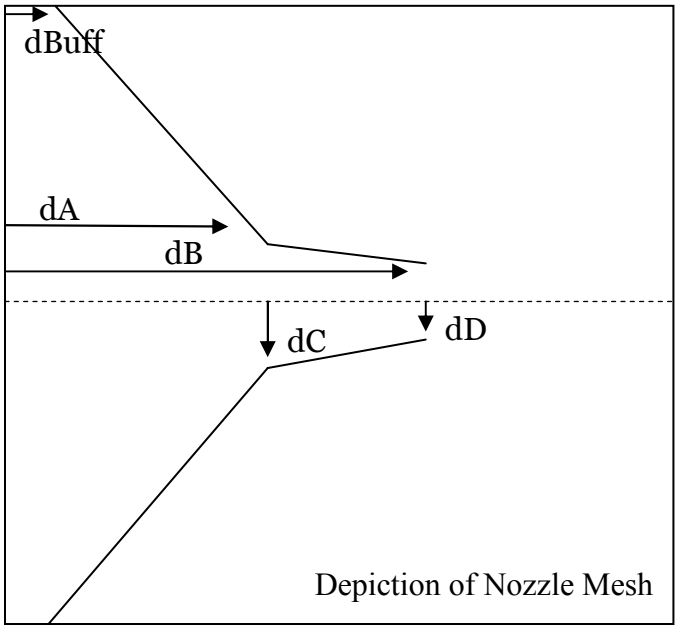
There are 255,000 simulators in the simulation and each represents 600,000,000 particles. There were 25,100 collisions per time step, which meets the recommended $1/10^{th}$ of simulators that should collide on each time step (Gallis). The energy varies by one part in 10^{13}, which corresponds to machine accuracy.

The number of collisions was also validated. Since there should be 14771.5 collisions per second per molecule, with 25000 molecules over .001 seconds there should be 3,766,733 collisions (Nave). There were 3,757,507 collisions, a mere 9,000 off the theoretical value with a .2449% error.

This simulation required only 323,800 KB of memory and took 2 minutes 30 seconds to run on the dual core laptop.

8.3 Nozzle Simulations

Compression and expansion nozzle flows are interesting problems for DSMC. In order to model many different kinds of these problems, we created a dynamic nozzle mesh, which can be adjusted so that the shape changes. Below is a picture of the nozzle mesh.



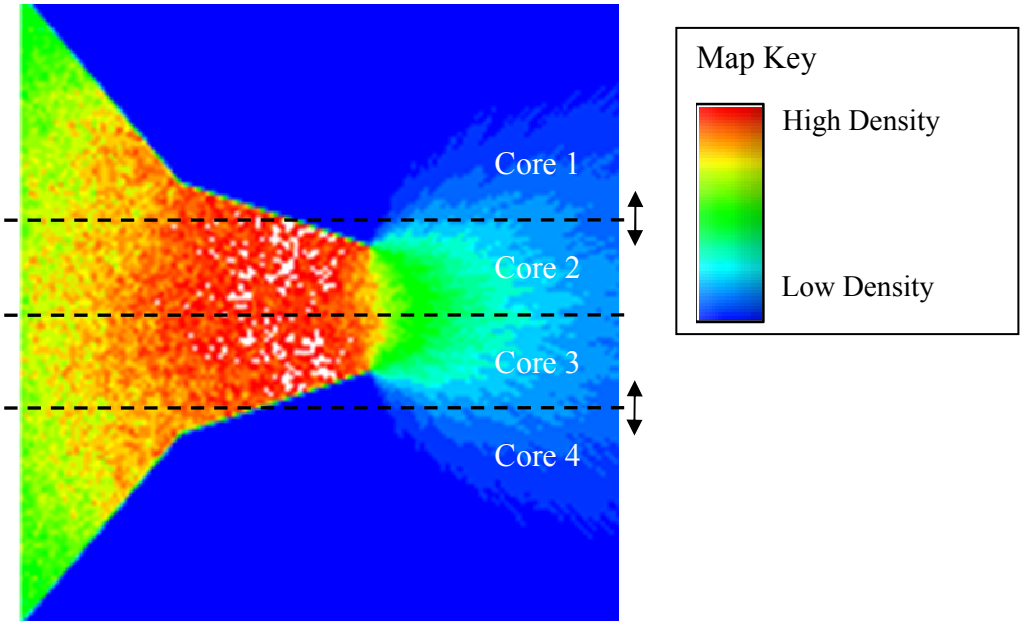
By adjusting the parameters we can change the height of the different portions of the nozzle, the size of the inner and outer necks, or the size of the buffer zone, where molecules are initialized.

8.4 Performance Analysis: Nozzle Test Case

The standard compression nozzle was used as a baseline for computer performance. The test was performed at the same altitude as the box equilibrium test, and therefore had the same dimensional parameters. This test had 150,000 molecules and about 17,000 collisions per time step. The total time was .01 seconds, requiring 1478 total time steps.

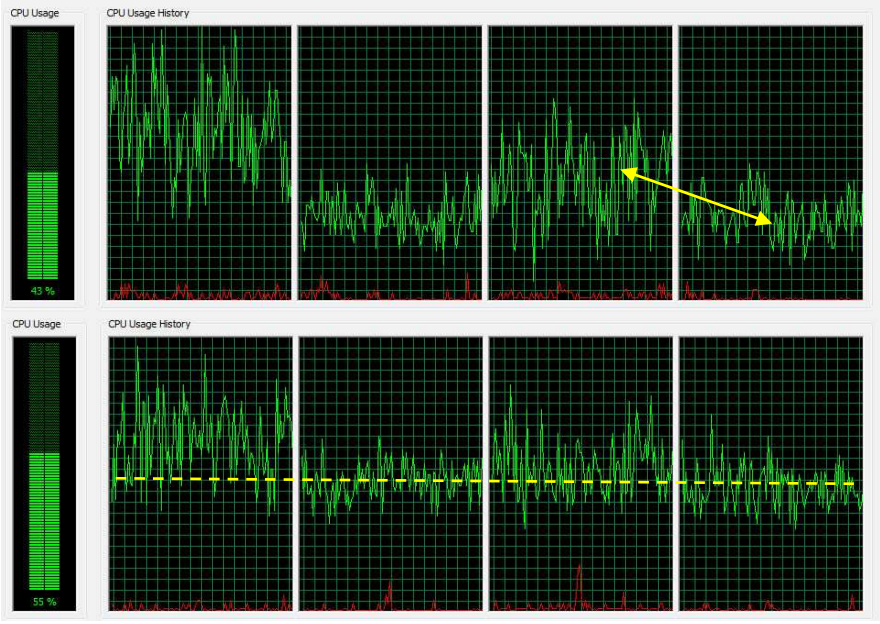
Two computers were used in the test. One was a laptop that has two 1.7 GHz cores, 256 KB L2 cache per core. The other is a quad-core desktop with 2.4 GHz processors and a 2 mb cache size per core.

When using four cores on the quad-core, the program has the option of dynamically allocating how many cells are assigned to each processor. For two cores, the simulation region is divided in half so that both cores have the same number of cells. When using four cores, the simulation region is not symmetric over all four, and so the intermediate boundary lines between cores 1, 2 and 3, 4 change dynamically.



Each core times how long it takes to calculate its section, and then the boundaries are moved so that cores with short calculation times have more cells on the next time step.

Although the proportion of cells on each processor can be manually set at the start, dynamically changing this parameter throughout the simulation is more effective because the distribution of simulators may change over the duration of the simulation. The picture below shows the difference between dynamic and set balancing in the task manager.

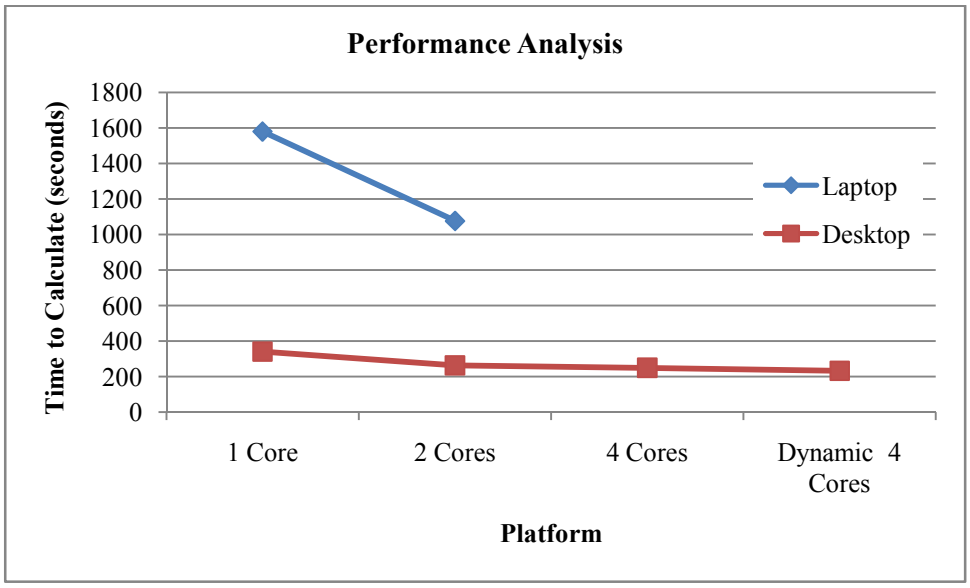


Unbalanced: Large difference between nodes' compute time.

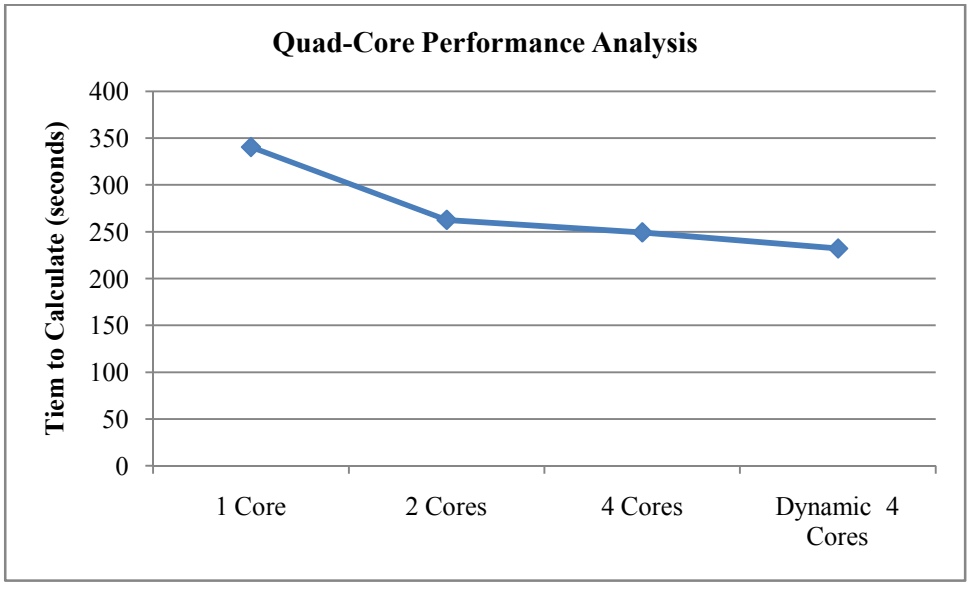
Balanced: Nodes adjust to share compute time

Note: The left-most processor is also the master node, so it has additional processes.

The nozzle simulation was run on different numbers of processors on each computer to analyze performance. The below graph shows the time it took for these computers to run the simulation.



As shown, the laptop benefitted most from running multi-core, while the times on the desktop were already so low they showed less obvious performance benefits. However, the desktop's times were also interesting, as shown below.



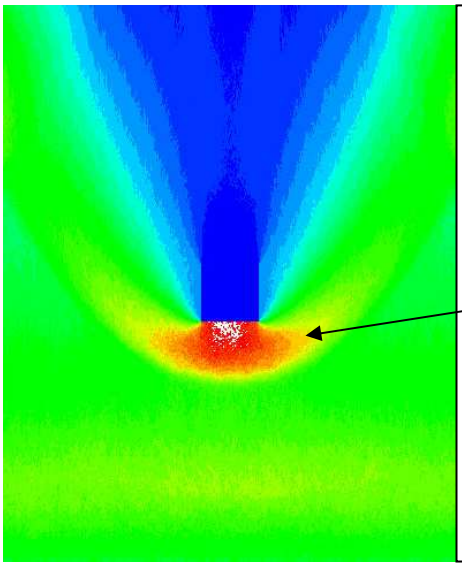
Here we see that the desktop had the largest time decrease when it went to 2 cores, while the second core doubling did not have as great an effect. Dynamically allocating the simulation region to the cores again had a significant positive effect on performance.

9. Applications

A calibrated DSMC model allows us to simulate complex rarified gas flows in the upper atmosphere with confidence that these simulations are accurate. The existing model allows us the opportunity to observe and analyze many familiar objects and gas phenomenon in the upper atmosphere. These applications include viewing shockwaves and modeling a scramjet intake nozzle.

9.1 Shockwave of a Supersonic Box

The program was used to calculate the shockwave of a box traveling at supersonic speeds through the atmosphere. Shockwaves are a difficult phenomenon to capture because they take time to form and require correctly calibrated conditions. This problem is especially significant to spacecraft because it is the primary component in the resistance they experience.



Supersonic or Not?

While this box forms a dense region immediately in front of it and the region takes time to dissipate, this is not a supersonic shockwave because the dense region itself does not further impede particles from free movement.

Notice that this section is dense, but does not *itself* slow down future particles.

This simulation took 30 minutes to complete on the quad core in dynamic mode and has the same parameters as the next simulation.

9.1.1 The Physics of Supersonic Shockwaves

Shockwaves are formed due to disturbances in a medium. Under supersonic conditions, the simulated object creating the disturbance travels faster than the particles that bounce off it. This results in a sudden change in temperature, pressure, and density, causing a shockwave. In rarefied gas, the shockwaves formed are generally thicker, ranging up to ten mean-free paths thick. The shockwave will induce resistance on the simulated object.

The shape of the shockwave is dependent on the speed of the propagating disturbance compared to the speed of sound, which will decrease with decreasing density. Any supersonic disturbance at a particular density will cause a "cone shape" in the particle density distribution. The angle that the cone forms with the normal to the object, or the semi-cone angle, is given by the equation ($\sin(\theta)=c/v$) where c is the speed of sound in the gas and v is the speed of the disturbance (Elert). At subsonic speeds, $\sin(\theta)$ is greater than one, resulting in no real solutions, i.e. no shockwave. With a larger v, the angle between the shockwave and the axis-direction of the cone becomes increasingly small.

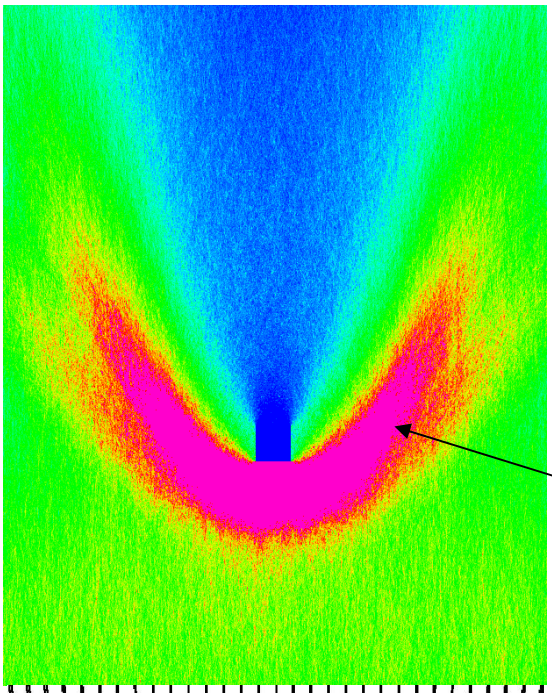
9.1.2 Shockwave Simulation

We simulated the shockwave shape of a box at different mach numbers. As predicted, the shockwave is thicker than at lower altitudes and the angle increases with Mach number. In the following pictures, the average direction and speed of particles in each area is shown by the lines. The blue dots show the starting point for the line, and white lines are supersonic while black lines are not. As shown, the simulators in front of the shockwave are not affected by the flow until they reach the wave, when they diverge to the sides.

The program took 27 minutes to run on the quad core and required 1.4 MB of memory. There were about 21,000 collisions per time step. The simulation area was 10 meters, with 1000 cells on a side. There were 250,000 simulators, each representing 7000000000 molecules. This simulated .04 seconds with a time step of .0000067698 seconds.

The actual size of the box was .4 meters and its speed was 1.2 times the speed of sound, or Mach .95. Note that the speed of sound is 268 m/s at this altitude.

This simulation generated the following density and velocity map:

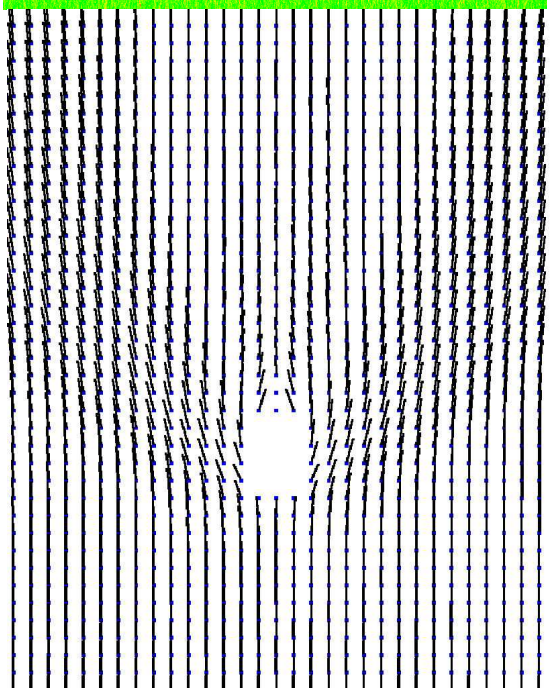


Map Key

High Density

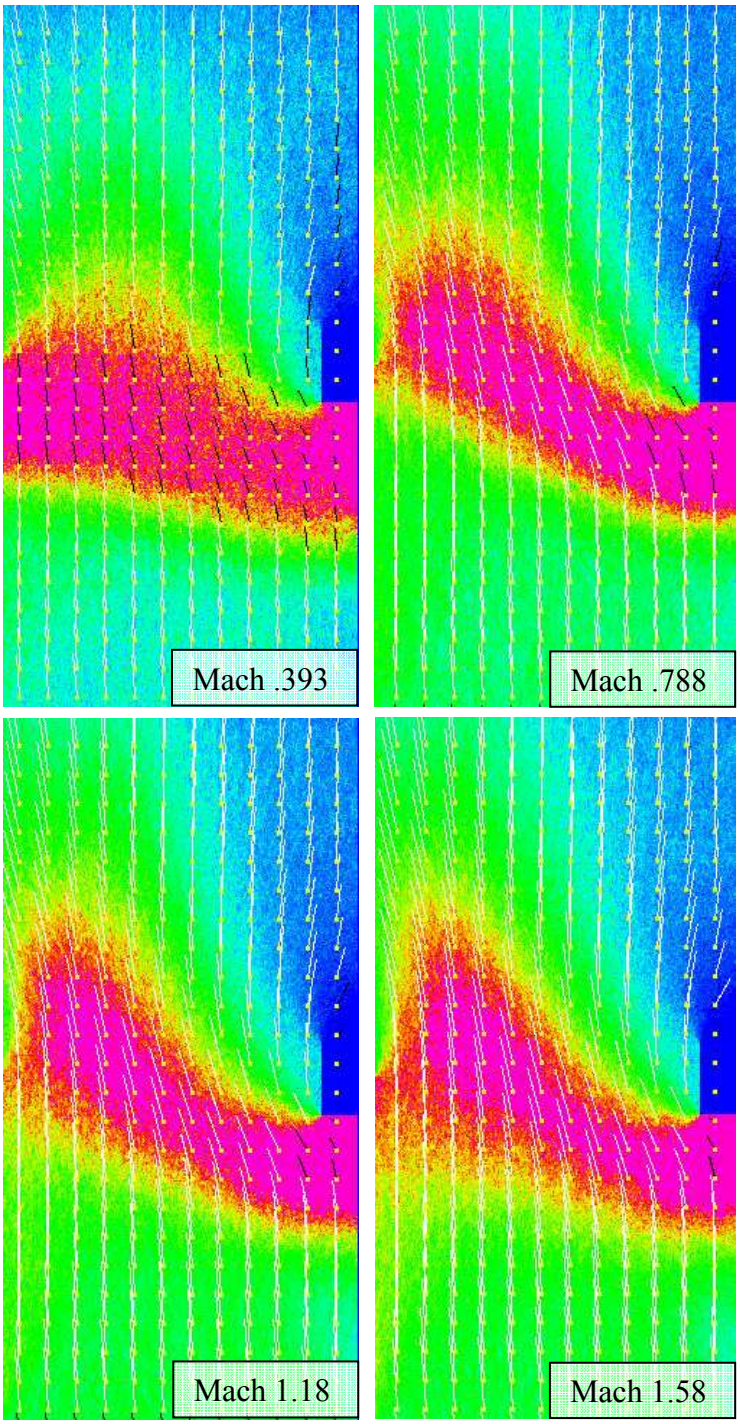
Low Density

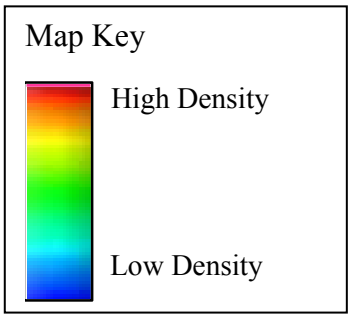
Supersonic shockwave formed from box



This is a velocity map of the simulation region, which shows direction of flow. The blue dots signify the beginning of each vector.
Notice that the simulators do not diverge to the sides until they meet the shockwave, showing this is supersonic.

We also ran a sequence of simulations to show how Mach number changes the form of the shockwave. In these simulations, velocity was incremented by .5 times the speed of sound each time, and the wave deformed for higher velocities as expected.





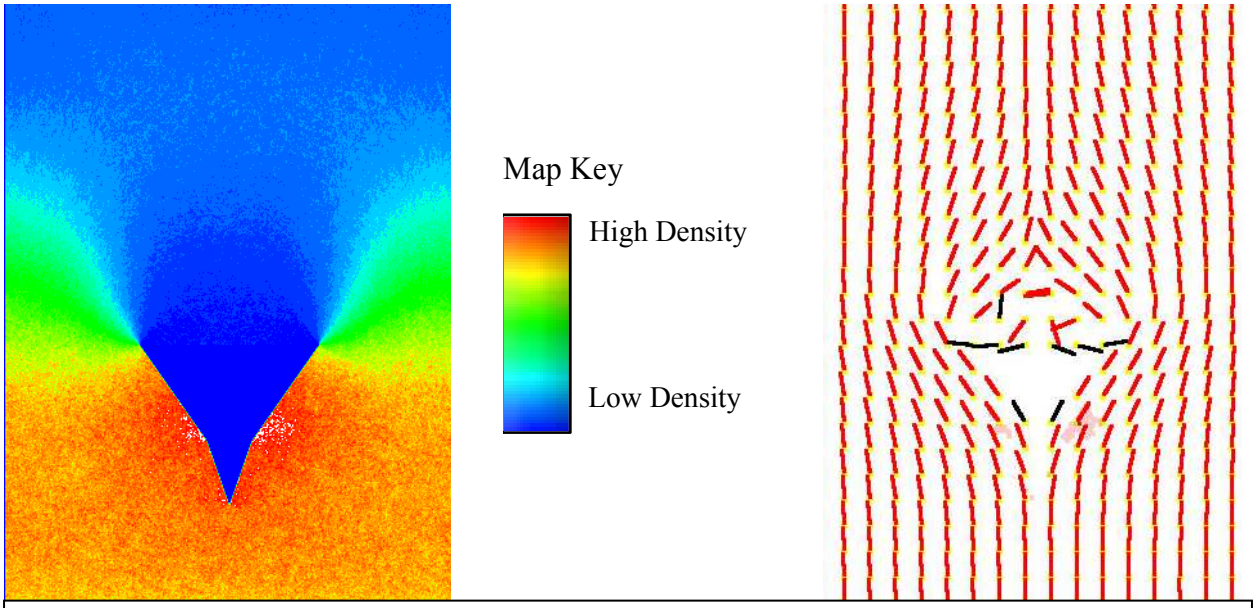
These simulations show half the simulation region of the previous simulation. The box is moving at the speed indicated by the Mach number in the bottom right of each picture.

The lines indicate the speed and direction of the flow. Black lines are below .75 the speed of sound, while lines are above.

By comparing vectors in all the pictures as well as the overall shape of the dense region, we can see the deformation of the shockwave due to speed. This matches the fact that shockwaves at higher velocities will have a 'steeper' angle, such as the Mach 1.58 shockwave.

9.2 Simulation of the Space Shuttle

Using the same simulation region at 90 km, the space shuttle shape was simulated instead of the box. The simulation had the same parameters as the box simulation regarding altitude, temperature and cell size. The simulation is for 2 seconds at mach 1.58. It took 11.5 hours to run in dynamic mode on the quad core. Below is the density and velocity map for the simulation.



Space Shuttle:

The space shuttle has less obvious shockwaves because of the more pointed front.

9.3 Simulation of a Scramjet

Scramjets are a novel form of engine that could someday be used as supersonic passenger jets in the lower atmosphere or as a mode of transport into orbit. They are designed to operate at lower altitudes, primarily gaining their power from the way gas flows into the engine. Understanding more about their operation at orbit altitudes is crucial in evaluating their potential as possible replacements for the aging space shuttle fleet.

9.3.1 Design of a Scramjet

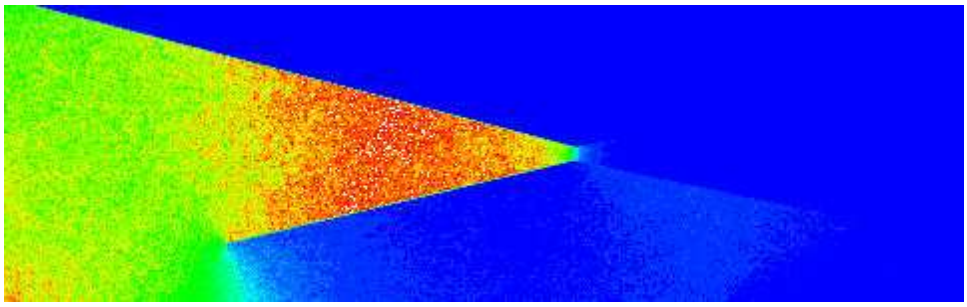
Scramjets are engines that are designed to work at low altitudes at supersonic speeds. It is unique in that instead of carrying fuel onboard for combustion, it collects fuel from the

atmosphere as it travels. The air is forced into the combustion chamber where the speed of the gas remains supersonic. The combustion chamber is shaped like a compression nozzle and constricts the available space for the gas to travel. In the compressed space, fuel is injected into the air (now fuel) and ignited. The gas is under extremely high pressure and bursts through the nozzles, propelling the scramjet forward.

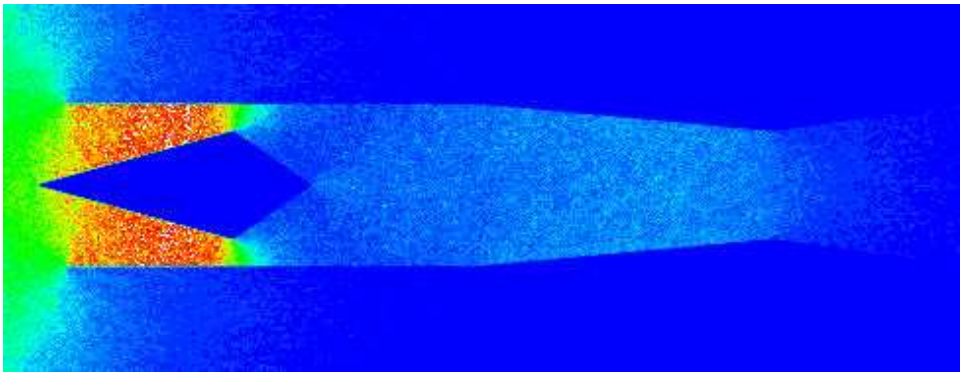
9.3.2 Scramjet Simulation

We used a modification of the existing nozzle mesh to create a scramjet specific nozzle simulation. We then simulated it at supersonic velocities. Since there were two major designs we found, we simulated both[1].

Our two adjustable meshes are as follows. First is the design of the front of the scramjet where the gas is first siphoned out of the region. Note: the mesh has two triangles going out form the midpoint, but since the second has very few molecules it can be difficult to see.



This is a more specific region in the scramjet where the engine is located. Fuel is ignited at the back of the diamond shape.



[1] One design is the top right picture at http://news.bbc.co.uk/media/images/38166000/gif/_38166755_scramjet_inf2300.gif, and the other is at http://www.talkingproud.us/ImagesScience/FalconHypersonic/ScramjetDiagram.jpg

As shown, very few particles get through the neck of the mesh despite different attempts at re-sizing the models. Neither simulation yielded an obvious shockwave. This may be because the scramjet shape will not produce a shockwave at this altitude. However, we have the shape and the program to further model a scramjet, and will perform additional research and analysis both to verify that scramjets should not function at this altitude or that they will function at a lower altitude. We are continuing our investigations in the scramjet area in order to clarify our results.

10. Conclusion

We have studied the physical basis of DSMC and developed a robust program that can simulate a wide variety of space objects. The validity of the model has been established through tests for the correctness of the physical model, accuracy of the code, and efficiency of the program on different computing platforms. We have simulated a variety of objects at different speeds and observed the shockwave patterns they create.

DSMC is an effective method to simulate rarified gas flows and can be used to view the shape of shockwaves for various objects as they change due to velocity.

11. Future Work

There are additional calibration and validation tests, simulations, and concepts we would like to incorporate into the project. The following are goals we may address in the next few weeks.

- Duplicate several of G. A. Bird's simple simulations, such as a vertical bar, as a calibration test.
- Simulate more aerospace vehicles and the shock waves they form.
- Continue simulating scramjets, possibly a very small scramjet at a lower altitude, so that a shock wave forms and the scramjet can operate. Also, perform more research into defining exactly the altitude that a scramjet of a certain size will cease to function.
- Incorporate equations for when conservation of momentum is not in effect for particle-object collisions. These situations may occur when objects are going at high Mach numbers, and the effect would be heat diffusion and a recalculation of the post-collision velocity for the particle.

12. Acknowledgements

We would like to thank Erik DeBenedictis and Dan Huang for being mentors for this project. Their advice was hugely helpful and greatly appreciated!

Special thanks to Dr. Michail Gallis, a Sandia National Laboratory scientist whose specialty is DSMC. Dr. Gallis' explanations and guidance were very much valued. Thank you!

13. References

Bird, G A. "Direct Simulation Monte Carlo Method Visual Programs at GAB Consulting" Jan. 2008. 29 Mar. 2008 <http://www.gab.com.au/>.

Bird, G A. Molecular Gas Dynamics and the Direct Simulation of Gas F Lows. New York: Oxford University P Inc., 1994.

Bird, G A. "Sophisticated DSMC". Notes prepared for a short course at the DSMC07 meeting Santa Fe, 30 Sept. 2007.

Elert, Glenn . "Shock Waves." The Physics Hypertextbook. 1998-1008 1 April. 2008 < http://hypertextbook.com/physics/waves/shock/>.

Gallis, M. (personal interview, March 25, 2008)

Latta, Lutz. Building a Million Particle System. Massive Development GmbH.

Moss, James N. "Direct Simulation Monte Carlo Simulations of Ballute Aerothermodynamics Under Hypersonic Rarefied Conditions." Journal of Spacecraft and Rockets 44 (2007).

Mueller, Matthias. "Direct Simulation Monte Carlo." 14 Dec. 1999. 29 Mar. 2008 <http://www.hlrs.de/people/mueller/papers/honnef99/node4.html>.

Nave, R. "Kinetic Theory Concepts." HyperPhysics. 29 Mar. 2008 <http://hyperphysics.phy-astr.gsu.edu/hbase/kinetic/ktcon.html#c1>.

Okumura, Haruhiko. "Random Number Generator." 26 Jan. 2002. 29 Mar. 2008 <http://oku.edu.mie-u.ac.jp/~okumura/cplusplus/mt19937.cc>.

14. Code

This section is an attempt at allowing the user to download and compile the code, and is meant to be an informal guide to getting started with the program. A couple things to note first: code generation libraries need to be set in multi-threaded mode (and change the number of cores by browsing to NUMCORE). This program also requires a large stack size, which depends on the settings, but give it a few MB. Save a blank file as stdfx.h. Change the compute intensity of the program by raising and lowering NUMMOL and NUMREPRESENT, but remember that molmolcollide should always be 1/10 of NUMMOL in the printout (this takes a few steps to stabilize). Files are exported into an archive file in the directory in gif format. At the end it says "printing densityindex" for a long time- that's qsort helping to decide with the color coding in the pictures, and the time it takes is dwarfed by any large simulation.

14.1 Meshes

The following are meshes that can be used to simulate different shapes. In order to switch the mesh there are a few things that must be done. First, move one of these meshes to above the comment (near MESH SECTION in movemolobj) and scroll down and find where it says "ifcollide[procnum]=0; lineintersection..." and make sure all the obja#'s in the mesh have are listed there. Make sure NUMBOUND matches how many obja#'s there are. To re-set how the molecules are initialized go to setmol: modes 2 and 3 are the ones most often used. Just set the desired mode to 2 (or go through all instances of setmol and fixmol to change it). ENDZONE is used to shrink the simulation area- it's the area that's in use. Set it to .99 if for a square region. The program is set up so that it is calculating the supersonic box. Have fun with the meshes!

```
//supersonic half box for faster simulation
double da=.45;
double db=.1;
line1 obja1;
lineset(obja1, da*WIDTH, HEIGHT*ENDZONE*(1-db)+(1-ENDZONE)*HEIGHT/2, da*WIDTH,
ENDZONE*HEIGHT+(1-ENDZONE)*HEIGHT/2);
line1 obja2;
lineset(obja2, (1-da)*WIDTH, ENDZONE*HEIGHT*(1-db)+(1-ENDZONE)*HEIGHT/2, (1-da)*WIDTH, (1-
db)*ENDZONE*HEIGHT+(1-ENDZONE)*HEIGHT/2);
line1 obja3;
lineset(obja3, da*WIDTH, ENDZONE*HEIGHT*(1-db)+(1-ENDZONE)*HEIGHT/2, (1-da)*WIDTH,
ENDZONE*HEIGHT*(1-db)+(1-ENDZONE)*HEIGHT/2);
line1 obja4;
lineset(obja4, 0, HEIGHT*(3./4.), WIDTH, HEIGHT*(3./4.));
line1 obja5;
lineset(obja5, 0, HEIGHT*(1./4.), WIDTH/2, HEIGHT*(1./4.));
```

```
line1 obja6;
lineset(obja6, WIDTH/2, HEIGHT*(1./4.), WIDTH, HEIGHT*(1./8.));

//scramjet intake nozzle
double dBuffer = .05;
double dA=.6;
double dB=1.0;
double dC=.05;
double dD=.3;
double movelegw=.2;
double movelegh=.8;
line1 obja1;
lineset(obja1, dBuffer*WIDTH, (1-ENDZONE)*HEIGHT/2, dA*WIDTH, (.5-dC)*HEIGHT*ENDZONE+(1-
ENDZONE)*HEIGHT/2 );
line1 obja2;
lineset(obja2, (dBuffer+movelegw)*WIDTH, ENDZONE*(HEIGHT*+movelegh)+(1-
ENDZONE)*HEIGHT/2, dA*WIDTH, (.5+dC)*HEIGHT*ENDZONE+(1-ENDZONE)*HEIGHT/2 );
line1 obja3;
lineset(obja3, dA*WIDTH, (.5-dC)*HEIGHT*ENDZONE+(1-ENDZONE)*HEIGHT/2, dB*WIDTH, (.5-
dD)*HEIGHT*ENDZONE+(1-ENDZONE)*HEIGHT/2 );
line1 obja4;
lineset(obja4, dA*WIDTH, (.5+dC)*HEIGHT*ENDZONE+(1-ENDZONE)*HEIGHT/2, dB*WIDTH,
(.5+dD)*HEIGHT*ENDZONE+(1-ENDZONE)*HEIGHT/2 );
line1 obja5;
lineset(obja5, 0, HEIGHT*ENDZONE+(1-ENDZONE)*HEIGHT/2, WIDTH*.75, HEIGHT*ENDZONE+(1-
ENDZONE)*HEIGHT/2 );
line1 obja6;
lineset(obja6, 0, (1-ENDZONE)*HEIGHT/2, WIDTH, (1-ENDZONE)*HEIGHT/2 );
line1 obja7;
lineset(obja7, WIDTH*.75, HEIGHT*ENDZONE+(1-ENDZONE)*HEIGHT/2 , WIDTH,
HEIGHT*ENDZONE+(1-ENDZONE)*HEIGHT );

//compression nozzle- symmetric
double dBuffer = .05;
double dA=.3;
double dB=.6;
double dC=.2;
double dD=.1;
line1 obja1;
lineset(obja1, dBuffer*WIDTH, 0, dA*WIDTH, (.5-dC)*HEIGHT );
line1 obja2;
lineset(obja2, dBuffer*WIDTH, HEIGHT, dA*WIDTH, (.5+dC)*HEIGHT );
line1 obja3;
lineset(obja3, dA*WIDTH, (.5-dC)*HEIGHT, dB*WIDTH, (.5-dD)*HEIGHT );
line1 obja4;
lineset(obja4, dA*WIDTH, (.5+dC)*HEIGHT, dB*WIDTH, (.5+dD)*HEIGHT );


//compression nozzle- asymmetric : interesting flow patterns on this one
double dBuffer = .05;
double dA=.3;
double dB=.6;
double dC=.2;
double dD=.1;
line1 obja1;
```

```
	lineset(obja1, dBuffer*WIDTH, 0, dA*WIDTH, (.5-dC)*HEIGHT );
	line1 obja2;
	lineset(obja2, dBuffer*WIDTH, HEIGHT, dA*WIDTH, (.5+dC)*HEIGHT );
	line1 obja3;
	lineset(obja3, dA*WIDTH-50*WIDTH/WCELLNUM, (.5-dC)*HEIGHT, dB*WIDTH, (.5-dD)*HEIGHT );
//asymmetric one
	line1 obja4;
	lineset(obja4, dA*WIDTH, (.5+dC)*HEIGHT, dB*WIDTH, (.5+dD)*HEIGHT );

	//shuttle
			double da=.2;
			double db=.3;
			double dc=.45;
			double de=.45;
			double df=.30;
			double w=WIDTH;
			double h=HEIGHT;
	lineset(obja1, w*da, h*.5*ENDZONE+(1-ENDZONE)*HEIGHT/2, w*db, h*de*ENDZONE+(1-
ENDZONE)*HEIGHT/2);
	lineset(obja2, w*db, h*de*ENDZONE+(1-ENDZONE)*HEIGHT/2, w*dc, h*df*ENDZONE+(1-
ENDZONE)*HEIGHT/2);
	lineset(obja3, w*dc, h*df*ENDZONE+(1-ENDZONE)*HEIGHT/2, w*dc, h*(1-df)*ENDZONE+(1-
ENDZONE)*HEIGHT/2);
	lineset(obja4, w*dc, h*(1-df)*ENDZONE+(1-ENDZONE)*HEIGHT/2, w*db, h*(1-de)*ENDZONE+(1-
ENDZONE)*HEIGHT/2);
	lineset(obja5, w*db, h*(1-de)*ENDZONE+(1-ENDZONE)*HEIGHT/2, w*da, h*.5*ENDZONE+(1-
ENDZONE)*HEIGHT/2);
	lineset(obja6, 0, HEIGHT*ENDZONE+(1-ENDZONE)*HEIGHT/2, WIDTH, HEIGHT*ENDZONE+(1-
ENDZONE)*HEIGHT/2);
	lineset(obja7, 0, (1-ENDZONE)*HEIGHT/2, WIDTH, (1-ENDZONE)*HEIGHT/2);

	 //diamond: now a scramjet, the second model
			double h=HEIGHT;
			double w=WIDTH;
			double da=.05;
			double db=.25;
			double dc=.33;
			double de=.4;
			double df=.08;
			double dedgdiff=.05;
			double dg=de-dedgdiff;;
			double dh=.5;
			double di=.8;
			double dj=.4;

// old diamond : non-adjustable
	line1 obja1;
	lineset(obja1, (WIDTH*(1./3.)), (HEIGHT*(2./3.)), (WIDTH*(1./2.)), (HEIGHT*(3./4.)) );
	line1 obja2;
	lineset(obja2, (WIDTH*(1./2.)), (HEIGHT*(3./4.)), (WIDTH*(2./3.)), (HEIGHT*(1./2.)) );
	line1 obja3;
	lineset(obja3, (WIDTH*(1./3.)), (HEIGHT*(1./2.)), (WIDTH*(1./2.)), (HEIGHT*(1./4.)) );
	line1 obja4;
	lineset(obja4, (WIDTH*(2./3.)), (HEIGHT*(1./2.)), (WIDTH*(1./2.)), (HEIGHT*(1./4.)) );
	line1 obja1;
```

```
lineset(obja1, (WIDTH/2), (HEIGHT/2), WIDTH, 0);
line1 obja2;
lineset(obja2, WIDTH, HEIGHT, (WIDTH/2), (HEIGHT/2));
```

14.2 Preliminary2.cpp

```
//This is the main simulator file that was written for the project.
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <time.h>
#include <iostream>
#include <fstream>
#include <conio.h>
#include <process.h>
#include <sys/types.h>
#include <sys/timeb.h>


#include "random.h"
#include "images.h"

#define PI 3.14159265358979323846264338327
#define MAXSTARTVELO (4.)
#define VELOTOCELL (1.)                         //the velocities are VELOTOCELL times that of when displayed in
cells
#define HEIGHT (10.)
#define WIDTH (10.)
#define NUMMOL 250000
#define NUMREPRESENT 7000000000.0
#define HCELLNUM 1000
#define WCELLNUM 1000
#define ENDZONE    .99                  //distance not cut off
#define MAXTIME (.01)              //was .01                    //max timesteps
#define STARTRECORD (MAXTIME/10.)                    //the timestep at which we start taking data
#define DEBUGLEV 0
#define MAXINCC 10000
#define MAXINCP 10000
#define DENSINC 18000                       //when cell is 'full' - max density index
#define NUMBOUND 6                                  // the number of 'walls' in the simulation
#define MACH 1.2              //multiples of supersonic you want the speed to be
double viewt=0;
double dt=.0000067698;       //0.0000067698;
int maxincell=0;
int countsetmol=0;
int countmovemolcollide=0;
int nummolmolcollide=0;
int stepstaken=0;
double alpha=.35;                           //original value for alpha- allocates cells to threads .233
double bufferparambox = 480.*(HEIGHT/HCELLNUM);       //size of gap between box and edge
double densityindex[HCELLNUM*WCELLNUM];
double sortedindex[HCELLNUM*WCELLNUM];
```

```
struct mma {
   int newcell;
   int cell;
   int index;
} MoveMolsArray[NUMMOL];
int ifcollide[NUMCORE];
double calctime[NUMCORE];

typedef unsigned char       BYTE;
typedef unsigned short      WORD;
typedef unsigned long       DWORD;
typedef DWORD   COLORREF;
//#define RGB(r,g,b)
((COLORREF)(((BYTE)(r)|((WORD)((BYTE)(g))<<8))|(((DWORD)(BYTE)(b))<<16)))
//#define min(A,B) ((A)<(B)? (A):(B))
//#define max(A,B) ((A)>(B)? (A):(B))
//(9/50)*NUMMOL

void magicalbreakpoint(int n){    //breaks ALL THE TIME :)  :D  =)  :)  :D  =)
   printf("magical break point called %d", n);
}


/************************
Vector class has +,-,*,/, and rotate (rotates by a radian), and norm (absolute value)
molecules' positions and velocities are vectors
************************/
class vector{
public:

   double x,y,z;
   vector(double i, double j=0, double k=0){
      x= i;
      y= j;
      z= k;
   }

   vector () { x = y = z = 0.0; }

   vector rotate (double theta){
         vector draw3;
         draw3.x= (x *(cos(theta)))+(y*sin(theta));
         draw3.y= (x *(-sin(theta)))+(y*cos(theta));
         return draw3;
   }

   vector operator+(double kh){vector rval= vector (x+kh, y+kh, z+kh); return rval;}
   vector operator-(double kh){vector rval= vector (x-kh, y-kh, z-kh); return rval;}
   vector operator*(double kh){vector rval= vector (x*kh, y*kh, z*kh); return rval;}
   vector operator/(double kh){vector rval= vector (x/kh, y/kh, z/kh); return rval;}

   vector operator+(vector kh){vector rval= vector (x+kh.x, y+kh.y, z+kh.z); return rval;}
   vector operator-(vector kh){vector rval= vector (x-kh.x, y-kh.y, z-kh.z); return rval;}
   //vector operator*(vector kh){vector rval= vector (x*kh.x, y*kh.y); return rval;}
   //vector operator/(vector kh){vector rval= vector (x/kh.x, y/kh.y); return rval;}
```

```
   void operator+=(vector kh){x+=kh.x, y+=kh.y, z+=kh.z;}
   void operator-=(vector kh){x-=kh.x, y-=kh.y, z-=kh.z;}

};
double norm(vector kh){return sqrt (kh.x*kh.x+kh.y*kh.y+kh.z*kh.z);}


struct collidedata{
   double dist;
   double xpos; double ypos;
   double xvelo; double yvelo;
   double dtt;
};

collidedata colorder[NUMCORE][NUMBOUND];       //collision order
void resetcolorder(int procnum){
   for (int i=0; i< NUMBOUND; i++){
      colorder[procnum][i].dist=WIDTH*HEIGHT;
      colorder[procnum][i].xpos=-1.;
      colorder[procnum][i].ypos=-1.;
      colorder[procnum][i].xvelo=-1.;
      colorder[procnum][i].yvelo=-1.;
      colorder[procnum][i].dtt=-1.;
   }
}
vector vectorindex[HCELLNUM*WCELLNUM];
int samplevecindex[HCELLNUM*WCELLNUM];
double newindex[HCELLNUM*WCELLNUM];

/**********************
lines are a structure so I can nicely track the variables
ix, iy= initial pos; fx, fy= final pos
it's Ax+By=C
**********************/
struct line1{
   double ix; double iy; double fx; double fy;
   double A; double B; double C;
};

/****
now for the setting function
http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=geometry2 for how to find the intersecton of lines...
(we love you topcoder!)
****/
void lineset(line1 &ell1, double ixt, double iyt, double fxt, double fyt){
   ell1.ix=ixt;
   ell1.iy=iyt;
   ell1.fx=fxt;
   ell1.fy=fyt;
   ell1.A=ell1.fy-ell1.iy; ell1.B=ell1.ix-ell1.fx;
   ell1.C=ell1.A*ell1.ix+ell1.B*ell1.iy;              //finding ax+by=c
}
```

```
/**********************
Molecule structure has position and velocity.
To add: energy, mass, cross-section, weight        possibly color?
**********************/
#define NOMOL (-2)
struct fmol2 {                      //first molecule structure
   vector pos;                      //position
   vector velo;                    //velocity
   float r;                       //radius
   float m;                        //mass
   int n;                          //mol number
   fmol2 () {}
};

struct fmolarray {
   //char buffer[32768];
   fmol2 data[NUMMOL];
   //char buffer2[32768];
   fmol2 &operator[](int n) {
      if (n < 0 || n >= NUMMOL) {
         magicalbreakpoint(1);
         if(DEBUGLEV>10)
            printf("ARRGH index is %d\n", n);
         n = 0;
      }
      return data[n];
   }
} fmolxxx;

fmol2 fmol[NUMMOL];

struct triple{
   int a, b, c;
};

/**********************
calculates the cell postroster[c][i] is in, returnes cell
**********************/
int changecell(double x, double y){
   double xhold=x/(WIDTH/WCELLNUM);
   double yhold=y/(HEIGHT/HCELLNUM);
   if (xhold<1)
      xhold=1;
   if (yhold<1)
      yhold=1;
   if (xhold>WCELLNUM)
      xhold=WCELLNUM;
   if (yhold>HCELLNUM)
      yhold=HCELLNUM;
   int xcell=int(xhold);
   int ycell=int(yhold);
   int answer = ycell*WCELLNUM+xcell;
   if (answer>HCELLNUM*WCELLNUM || answer<0){
      printf("this is wierd, spot in changecell \n");
   }
```

```
   return answer;
}

/*******
MSC- molecular speed calculator
http://hyperphysics.phy-astr.gsu.edu/hbase/kinetic/kintem.html
msc(double molmass, double temp, double amu)
vp=sqrt(2*R*temp/molmass);        //most probable speed
vhat=(8*R*temp/PI*molmass);       //mean speed
vrms=(3*R*temp/molmass);        //root mean squared speed
MSCv2-evaluates if two random variables are withing the distrubution
*******/
double msc2(double M, double T, int procnum){        //amu is usually 29, making molmass .029, temp in K
   double start=0;
   int stop=0;
   double R= 8.314472;     //gas constant J/(K*mol)
   double vhat=sqrt((8*R*T)/(PI*M));        //mean speed
   while (stop==0){
      double x=vhat*5*genrand_real2(procnum);
      double y= genrand_real2(procnum)*.002003;
      double probo=4*PI*pow((M/(2*PI*R*T)),(3./2))*x*x*exp(((-M*x*x)/(2*R*T)));
      if (probo>y){
         start=x;
         stop=1;
      }
   }
   return start;
}


/*******
sets molecules to positions, differnet methods control type of set
*******/
void setmol(int i, int method, int procnum){
   countsetmol++;
   //fmol[i].pos.y= HEIGHT*ENDZONE*genrand_real2(procnum)+(1-ENDZONE)*HEIGHT/2;
   fmol[i].pos.y= HEIGHT*ENDZONE*genrand_real2(procnum)+(1-ENDZONE)*HEIGHT/2;
   vector start (1.0,0.0);
   double angle1=0;
   if (method==0){
      fmol[i].pos.x= WIDTH*genrand_real2(procnum);                  //we don't like negative positions?
      angle1=genrand_real2(procnum)*2*PI;
   }
   else if (method==1){
      fmol[i].pos.x= 0;//WIDTH/2;
      angle1=(genrand_real2(procnum)*PI)-(PI/2);//*PI-(PI/2);
   }
   else if (method==3){        //NEW METHOD 2- sets in box
      fmol[i].pos.y= (HEIGHT-3*bufferparambox)*genrand_real2(procnum)+1.5*bufferparambox;
      fmol[i].pos.x= (WIDTH-3*bufferparambox)*genrand_real2(procnum)+1.5*bufferparambox;
      angle1=(genrand_real2(procnum)*2*PI);
   }
   else if (method==2){        //OLD METHOD 2- sets along bottom line
      fmol[i].pos.x= 25*(WIDTH/(WCELLNUM))*genrand_real2(procnum);
      //fmol[i].pos.x=0;
```

```
      angle1=(genrand_real2(procnum)*PI)-(PI/2);
   }
   else if (method==4){       //OLD METHOD 2- sets along bottom line
      fmol[i].pos.x= 15*(WIDTH/(WCELLNUM))*genrand_real2(procnum);
      //fmol[i].pos.x=0;
      angle1=(genrand_real2(procnum)*PI)-(PI/2);
   }
   double startv=msc2(.029, 180., procnum);          //we're at 47 km
   start=start.rotate(angle1);
   vector velo=start*startv;
   fmol[i].velo= velo;
   fmol[i].velo.x+=280*MACH;       //supersonic 268.79
   //printf("velo %f\n", norm(fmol[i].velo));
   fmol[i].r= (float)3e-10;
   fmol[i].m=(float)4.81727e-26;
   fmol[i].n=i;
   MoveMolsArray[i].cell=changecell(fmol[i].pos.x, fmol[i].pos.y);
   if (DEBUGLEV>5 && method==1){
      int x= int(fmol[i].velo.x*VELOTOCELL*1.2);              //sets moved positions in aone to 50
      int y= int(fmol[i].velo.y*VELOTOCELL*1.2);
      if (x>=WIDTH || y>= HEIGHT || x<=0 || y<=0){                        //random particle positions
         int p=0;
      }
   }
}

/*******
declaring 'safe' arrays that tell me when I'm jumping array bounds
*******/
class checkarray {
   //char buffer[32768];
   int data[WCELLNUM*HCELLNUM];//MAXINC];
   //char buffer2[32768];
public:
   int operator[](int i){
      if (i<0 || i>=NUMMOL){
         if(DEBUGLEV>10)
            printf("AHHHH #1\n");
      }
      return data[i];
   }
   int set(int i, int v);
};

int checkarray::set(int i, int v) {
   if (i<0 || i>WCELLNUM*HCELLNUM){
      magicalbreakpoint(2);
      if(DEBUGLEV>10)
         printf("AHHHH #2\n");
   }
   if (v > MAXINCC) {
      if(DEBUGLEV>10)
         magicalbreakpoint(3);
         printf("AHHHH #3\n");
      return 1;
```

```
    }
    else {
        data[i] = v;
        return 0;
    }
}


class checkarray2 {
    checkarray data[WCELLNUM*HCELLNUM];
public:
    checkarray &operator[](int i){

        if (i<0 || i>=MAXINCC){
            if(DEBUGLEV>10)
                magicalbreakpoint(4);
                printf("AHHHH #4\n");
        }
        return data[i];
    }
};

#define BUCKET 20
#define BUCKETS 15
static int histogram[BUCKETS];

class cellmemory {
    //fmol2 data[MAXINC];
    fmol2 *data;
    int datasize;
public:
    void checkme(int expect) {
        expect++;
        ASSERT(expect<=datasize);
        if(1)for(int i=0; i< expect; i++){
            ASSERTNE(data[i].n, NOMOL);
        }
        if(1)for(int i=expect; i<datasize; i++){
            ASSERTEQ(data[i].n, NOMOL);
        }
    }
    fmol2 &operator[](unsigned int index) {
        if(index >= datasize){
            int newdatasize=((index+20)/20)*20;
            fmol2 *sufficentdata= new fmol2[newdatasize];
            ASSERT(sufficentdata!=NULL);        //will fail if you run out of memory
            for (int i=0; i<datasize; i++)
                sufficentdata[i]=data[i];
            for (int j = datasize; j < newdatasize; j++)
                sufficentdata[j].n = NOMOL;
            if(data!=NULL) {
                delete data;
                int index = datasize/BUCKET;
                if (index > BUCKETS-1) index = BUCKETS-1;
                histogram[index]--;
```

```
            }
            data=sufficentdata;
            datasize=newdatasize;
            int index2 = datasize/BUCKET;
            if (index2 > BUCKETS-1) index2 = BUCKETS-1;
            histogram[index2]++;
            if(datasize>MAXINCC){
               printf("datasize more than MAXINCC \n");
            }
         }
         return data[index];
      }
      cellmemory(){
         data=NULL;
         datasize=0;
      }
      void printhist() {
         for (int i = 0; i < BUCKETS; i++) {
            printf("bucket %d-%d: %d\n", i*BUCKET, (i+1)*BUCKET-1, histogram[i]);

         }
      }
};

checkarray countpost;

class pr{
   cellmemory *data[HCELLNUM*WCELLNUM];
public:
   void init(){
      for (int i = 0; i < HCELLNUM*WCELLNUM; i++)
         data[i] = new cellmemory;
      printf("executed init \n");
   }
   cellmemory &operator[](unsigned int index) {
      ASSERT(index < HCELLNUM*WCELLNUM);
      return *data[index];
   }
   void check(){
      if (0) for (int i=0; i < HCELLNUM*WCELLNUM; i++){
         data[i]->checkme(countpost[i]);
      }
   }
} postroster;
//fmol2 postroster[HCELLNUM*WCELLNUM][MAXINC];          //postroster[cellnum][molecule
slot]=[NUMMOL]

/****
figures out if and where two lines intersect
ficollide==1 if they collide, and then executes the mol-obj collision code
****/
void lineintersection(line1 &obj, int boundnum, line1 &mol, double xcollide, double ycollide, int c, int i, double dt,
double newdt, int boundtoskip, int procnum){
   if (boundnum==boundtoskip)
      return;
```

```
    xcollide=0;
    ycollide=0;
    newdt=0;
    double det = mol.A*obj.B - mol.B*obj.A;
    if(det == 0){
        if (DEBUGLEV>3){
            printf("parallel lines! wahoo! \n");
        }
    }
    else{
        xcollide = (obj.B*mol.C - mol.B*obj.C)/det;
        ycollide = (mol.A*obj.C - obj.A*mol.C)/det;
    }
    if (xcollide==mol.ix || ycollide== mol.iy){        //if it happens to be right on edge, it doesn't crap out
        xcollide=xcollide+1e-5;
        ycollide=ycollide+1e-5;
    }
    if( (obj.ix==obj.fx) && (fabs(obj.ix-xcollide)<1e-10) ){    //if you have a horizontal boundary, then the
calculation of a collision point can be off in the lower digits
        xcollide=obj.ix;
    }
    if( (obj.iy==obj.fy) && (fabs(obj.iy-ycollide)<1e-10) ){
        ycollide=obj.iy;
    }
    double mminx=min(mol.ix, mol.fx); double mminy=min(mol.iy,mol.fy);
    double mmaxx=max(mol.ix, mol.fx); double mmaxy=max(mol.iy,mol.fy);
    double ominx=min(obj.ix, obj.fx); double ominy=min(obj.iy,obj.fy);
    double omaxx=max(obj.ix, obj.fx); double omaxy=max(obj.iy,obj.fy);
    if( (mminx <= xcollide) && (xcollide <= mmaxx) && (mminy <= ycollide) && (ycollide <= mmaxy)){
        if( (ominx <= xcollide) && (xcollide <= omaxx) && (ominy <= ycollide)){
            //double difference=ycollide-omaxy;
            //printf(" this is the difference %f\n", difference);
            if(ycollide <= omaxy){
                ifcollide[procnum]=1;
                double dist=sqrt((mol.ix-xcollide)*(mol.ix-xcollide)+(mol.iy-ycollide)*(mol.iy-ycollide));
                colorder[procnum][boundnum].dist=dist;
                colorder[procnum][boundnum].xpos=xcollide;
                colorder[procnum][boundnum].ypos=ycollide;
                //postroster[c][i].pos.x=xcollide;
                //postroster[c][i].pos.y=ycollide;
                double angle1 = atan2(obj.iy-obj.fy, obj.ix-obj.fx);
                vector cvelo=postroster[c][i].velo;
                cvelo=cvelo.rotate(angle1);
                cvelo.y=-cvelo.y;
                cvelo=cvelo.rotate(-angle1);
                colorder[procnum][boundnum].xvelo = cvelo.x;
                colorder[procnum][boundnum].yvelo = cvelo.y;
                double timeused=dist/norm(postroster[c][i].velo);
                colorder[procnum][boundnum].dtt=dt-timeused;
                if(DEBUGLEV>10){
                    printf("bounce %d, %d\n", c, i);
                }
            }
        }
    }
```

```
}

/**********************
molcollide assigns cells, finds num of pairs of collisions/cell, and calculates new velocities
**********************/
void molcollide2();
void dataset(){
   if(DEBUGLEV>10)
      printf("checkpoint A\n");
   molcollide2();
   if(DEBUGLEV>10)
      printf("checkpoint B\n");

}
void molcollide2(){        //just sets up the data
   if(DEBUGLEV>10)
      printf("checkpoint in\n");
   int preroster[NUMMOL];
   //checkarray preroster;
   //checkarray2 postroster;
   //fmol2 postroster[HCELLNUM*WCELLNUM][NUMMOL];            //postroster[cellnum][molecule
slot]=[NUMMOL]
   //checkarray countpost;
   //int countpost[NUMMOL];
   if (0) for (int i=0; i<HCELLNUM*WCELLNUM; i++){
      if (1) for (int j=0; j<MAXINCP; j++){
         postroster[i][j].n= NOMOL;
      }
   }
//printf("checkpoint 1\n");
   if (1) for (int i=0; i<NUMMOL; i++){
      preroster[i]= -1;
   }
   if (1) for (int i=0; i<HCELLNUM*WCELLNUM; i++){
      countpost.set(i, -1);
   }

//printf("checkpoint 2\n");
   if (1) for (int n=0; n<NUMMOL; n++){            //assigns cell numbers
      if (fmol[n].pos.x>0 && fmol[n].pos.x<WIDTH && fmol[n].pos.y>0 && fmol[n].pos.y<HEIGHT){
         preroster[n]= changecell(fmol[n].pos.x,fmol[n].pos.y);
      }
      else {
         static int count;
         printf("impossible condition occurs %d times\n", ++count);
      }

      //printf ("%d is in cell %d \n", n, preroster[n]);
      if (preroster[n] >= (0) && preroster[n] <= (WCELLNUM*HCELLNUM-1)){
         countpost.set(preroster[n], countpost[preroster[n]]+1);
         if (countpost[preroster[n]]<MAXINCC){
            fmol2 t = fmol[n];
            postroster[preroster[n]][countpost[preroster[n]]] = t;
            MoveMolsArray[n].cell=preroster[n];
            MoveMolsArray[n].index=countpost[preroster[n]];
```

```
            }
            else
               magicalbreakpoint(5);
         }
      }

      if(DEBUGLEV>10)
         printf("checkpoint out\n");
}

void molmolcollide(double dt, double t, int begincell, int endcell, int procnum){
      if (1) for (int n=begincell; n<=endcell; n++){        //FIXFIXFIX!
         int totalmol=0;
         if (countpost[n]>-1){
            totalmol = countpost[n]+1;
            if(DEBUGLEV>1){
               printf("%d mol num %d in cell %d\n", totalmol, postroster[n][countpost[n]].n, n);
            }
         }
         if(totalmol>1){
            if (maxincell<totalmol){
               maxincell=totalmol;
            }
            if(t>STARTRECORD){
               densityindex[n]+=totalmol;
            }
            double oo=0;
            double speedmax=0;
            if (1) for (int a=0; a<countpost[n]; a++){
               if (norm(postroster[n][a].velo)>speedmax)
                  speedmax=norm(postroster[n][a].velo);
                  oo=PI*4*postroster[n][a].r*postroster[n][a].r;
            }
            speedmax*=2;
            double vc=(WIDTH/WCELLNUM)*(HEIGHT/HCELLNUM);
            //printf ("total= %d, o= %f, speed=%f, vc= %f\n", totalmol, oo, speedmax, vc);
            double mc=0;                                    //number of pairs of particles= mc (escher... tee hee)
            mc=(totalmol*(totalmol-1)*oo*speedmax*dt)*(NUMREPRESENT*NUMREPRESENT)/(vc);
            double newmc=floor(mc);
            if (newmc>(totalmol/2)){
               //printf("newmc>totalmol/2 \n");
               newmc=(totalmol/2);
            }

            //printf("particle pairs=%f cell %d, total=%d \n", newmc, n, totalmol);
            for (int a=0; a<newmc; a+=2){        //now choosing our pairs- do we want to randomly pick i and j?
               if (a<=(NUMMOL-1)){
                  int i=a;
                  int j=i+1;                     //i=a, j=a+1 for now

                  // guarantee i and j to be valid molecule numbers
                  i %= totalmol;
                  j %= totalmol;

                  // focus on the two particles that are colliding
```

```
            fmol2 *p1 = &postroster[n][i], *p2 = &postroster[n][j];
            double vrel= norm(p1->velo-p2->velo);

            double ein = norm(p1->velo)*norm(p1->velo) + norm(p2->velo)*norm(p2->velo);

            // accept reject on particle pair
            double check= vrel/speedmax;
            double z=genrand_real2(procnum);
            if (check>z) continue;                                  //now we execute the particle collision! boom!

            //printf("executing collision between particles %d and %d in cell %d \n", i, j, n);

            //calculate random direction
            vector e;
            double n;
            do {
               e= vector (genrand_real2(procnum)*2.-1.,genrand_real2(procnum)*2.-
1.,genrand_real2(procnum)*2.-1.);       //impact parameter, e=b?
               n = norm(e);
            } while(n>1 || n < 1e-12);
            e=e*1./n;       //3D now
            //e = p1->velo-p2->velo;
            //e = e/norm(e);

            // convert to cm frame
            double mt = p1->m+p2->m;
            vector vcm= (p1->velo*p1->m + p2->velo*p2->m)/mt;

            double icompogn=p2->m/mt;
            p1->velo= vcm + e*icompogn*vrel;

            double jcompogn=p1->m/mt;
            p2->velo= vcm - e*jcompogn*vrel;

            double eout = norm(p1->velo)*norm(p1->velo) + norm(p2->velo)*norm(p2->velo);
            //printf("%.12f %.12f %.12f\n", ein, eout, ein-eout);
            nummolmolcollide++;
         }
      }
   }
}

/*******
has molecules in the cell w/spacecraft colliding
*******/
void movemolobj(double dt, int c, int i, int procnum){
   int boundtoskip=-1;
   while (dt>0){
      int debugmode=0;
      countmovemolcollide++;
      resetcolorder(procnum);
      if (countmovemolcollide>50){
         printf("infinite looping! danger danger! \n");
         magicalbreakpoint(6);
```

```
	}
	if (fabs(postroster[c][i].pos.y-7987.3795298422) < 1e-5){
	   double e = 2.1718;
	   debugmode=1;
	}
	double foo = postroster[c][i].pos.x;
	double foo2 = postroster[c][i].pos.y;
	if(0) if (countmovemolcollide>1){	//why am I doing this?
	   postroster[c][i].pos.x += postroster[c][i].velo.x*(1e-4)*VELOTOCELL;
	   postroster[c][i].pos.y += postroster[c][i].velo.y*(1e-4)*VELOTOCELL;
	}

	//box
	double movebox=0.;
	line1 obja1;		// bottom
	lineset(obja1, bufferparambox+movebox, bufferparambox, WIDTH-bufferparambox+movebox,
bufferparambox);
	line1 obja2;		// right
	lineset(obja2, WIDTH-bufferparambox+movebox, bufferparambox, WIDTH-bufferparambox+movebox,
HEIGHT-bufferparambox );
	line1 obja3;		// top
	lineset(obja3, bufferparambox+movebox, HEIGHT-bufferparambox, WIDTH-bufferparambox+movebox,
HEIGHT-bufferparambox );
	line1 obja4;		// left
	lineset(obja4, bufferparambox+movebox, bufferparambox, bufferparambox+movebox, HEIGHT-
bufferparambox );

	/******
		// MESH SECTION
		//Ok people, in order to switch your mesh you need to do a couple things
		//first, you move the mesh you want outside the comment
		//scroll down and find where it says "ifcollide[procnum]=0; lineintersection..." and make sure all
the obja# ones are there
		//make sure NUMBOUND matches how many obja#'s you have
		//you may want to re-set how the molecules are initializes: so to setmol, 2 and 3 are the ones most
often used
		//ENDZONE is used to shrink the simulation area- it's the area that's in use. set it to .99 if you
want a square region
		//have fun wih the meshes!
	****/

	double molix=postroster[c][i].pos.x;		//initial x and y
	double moliy=postroster[c][i].pos.y;
	postroster[c][i].pos.x += postroster[c][i].velo.x*dt*VELOTOCELL;
	postroster[c][i].pos.y += postroster[c][i].velo.y*dt*VELOTOCELL;
	if(debugmode){
	   printf("new pos is %e,%e \n", postroster[c][i].pos.x, postroster[c][i].pos.y);
	}
	line1 molline;
	lineset(molline, molix, moliy, postroster[c][i].pos.x, postroster[c][i].pos.y);
	double xcollide=0;
	double ycollide=0;
	double newdt=0;
	ifcollide[procnum]=0;
	lineintersection(obja1, 0, molline, xcollide, ycollide, c, i, dt, newdt, boundtoskip, procnum);
```

```
      lineintersection(obja2, 1, molline, xcollide, ycollide, c, i, dt, newdt, boundtoskip, procnum);
      lineintersection(obja3, 2, molline, xcollide, ycollide, c, i, dt, newdt, boundtoskip, procnum);
      lineintersection(obja4, 3, molline, xcollide, ycollide, c, i, dt, newdt, boundtoskip, procnum);
      //lineintersection(obja5, 4, molline, xcollide, ycollide, c, i, dt, newdt, boundtoskip, procnum);
      //lineintersection(obja6, 5, molline, xcollide, ycollide, c, i, dt, newdt, boundtoskip, procnum);
      //lineintersection(obja7, 6, molline, xcollide, ycollide, c, i, dt, newdt, boundtoskip, procnum);
      if(ifcollide[procnum]==1){
         int walltobounce=0;
         double currentdist=WIDTH*HEIGHT;
         for (int j=0; j< NUMBOUND; j++){
            if (colorder[procnum][j].dist<currentdist){
               currentdist=colorder[procnum][j].dist;
               walltobounce=(j);
            }
         }
         postroster[c][i].pos.x=colorder[procnum][walltobounce].xpos;
         postroster[c][i].pos.y=colorder[procnum][walltobounce].ypos;
         double a = colorder[procnum][walltobounce].xvelo;
         double b = colorder[procnum][walltobounce].yvelo;
         double v = sqrt(a*a+b*b);
         if (DEBUGLEV>10){
            printf("collision at (%f, %f) part=%d wall=%d newdt=%f v=%f\n",postroster[c][i].pos.x,
postroster[c][i].pos.y, c, walltobounce, colorder[procnum][walltobounce].dtt, v);
         }
         postroster[c][i].velo.x=colorder[procnum][walltobounce].xvelo;
         postroster[c][i].velo.y=colorder[procnum][walltobounce].yvelo;
         dt=colorder[procnum][walltobounce].dtt;
         boundtoskip = walltobounce;
      }
      else {
         dt = 0.;
         if(0)
if(postroster[c][i].pos.x<2000||postroster[c][i].pos.x>8000||postroster[c][i].pos.y<2000||postroster[c][i].pos.y>8000){
            int confusedbit=1;
            ifcollide[procnum]=0;
            //printf("pos of particle %d in cell %d is (%f, %f)", i, c,postroster[c][i].pos.x, postroster[c][i].pos.y);
         }
      }
   }
}

/********
when it finds a molecule that's out of bounds it resets it until it's in bounds with a legal cell
then assigns particle back into original cell to be moved later
********/
fixmol(int i, int n, int method, int c, int procnum){
   int check =0;
   int celln=0;
   while(check==0){
      setmol(i, method, procnum);
      celln= changecell(fmol[i].pos.x, fmol[i].pos.y);
      if (celln>=0 && celln<WCELLNUM*HCELLNUM){
         check=1;
         postroster[c][n]=fmol[i];
      }
```

```
   }
return celln;
}

/********
this version makes sure the temperature of the molecule stays the same by keeping the velocity components the
same
********/
fixmol2(int i, int n, int method, int c, vector oldvelo, int procnum){
   int check =0;
   int celln=0;
   while(check==0){
      setmol(i, method, procnum);
      celln= changecell(fmol[i].pos.x, fmol[i].pos.y);
      if (celln>=0 && celln<WCELLNUM*HCELLNUM){
         check=1;
         postroster[c][n]=fmol[i];
      }
   }
   if(oldvelo.x<0)
      oldvelo.x=-oldvelo.x;
   double flipbit=genrand_real2(procnum);
   if(flipbit<.5)
      oldvelo.y=-oldvelo.y;
   postroster[c][n].velo=oldvelo;
return celln;
}

/********
calculates in-cell collisions and movement for particles in cells between begincell and endcell
********/
void CalculateCells(int begincell, int endcell, double dt, double t, int procnum){
   struct timeb start;
   struct timeb finish;
   ftime(&start);
   molmolcollide(dt, t, begincell, endcell, procnum);        //molmol collisions
   if(DEBUGLEV>10)
      printf("checkpoint 5\n");

   if (1) for (int c=begincell; c<=endcell; c++){        //moves x and y coordinates of particle
      if (1) for (int i=0; i<=countpost[c]; i++){
         if((postroster[c][i].n) != NOMOL){
            if(t>STARTRECORD){        //waiting for comming to steady state... change back to 10!
               vectorindex[c]+=postroster[c][i].velo;
               samplevecindex[c]++;
            }
            countmovemolcollide=0;
            //double oldtemp= norm(postroster[c][i].velo);
            movemolobj(dt, c, i, procnum);    //mol obj collisions- determined by movement- RECURSIVE
            //double newtemp= norm(postroster[c][i].velo);
            //if(fabs(newtemp-oldtemp)>1e-12){
            //    int tempchanging=1;
            //}
            double x= postroster[c][i].pos.x;              //sets moved positions in aone to 50
            double y= postroster[c][i].pos.y;
```

```
            if (x>=WIDTH || y>= HEIGHT || x<=0 || y<=0){
               //double oldtemp= norm(postroster[c][i].velo);
               fixmol2(postroster[c][i].n, i, 4, c, postroster[c][i].velo, procnum);
               /*double newtemp= norm(postroster[c][i].velo);
               if(fabs(newtemp-oldtemp)>1e-12){
                  int tempchanging=1;
               }
               int n=postroster[c][i].n;*/
               x= postroster[c][i].pos.x;              //sets moved positions in aone to 50
               y= postroster[c][i].pos.y;
               MoveMolsArray[postroster[c][i].n].index=i;
               MoveMolsArray[postroster[c][i].n].cell=c;
            }
            int cell= changecell(x,y);
            //int remainder= cell % WCELLNUM;
            if(cell> HCELLNUM*WCELLNUM){
               printf("what is going on!?\n");
            }
            //if (cell != c){
            MoveMolsArray[postroster[c][i].n].newcell=cell;
            //}
            //else
            //    ASSERTEQ(MoveMolsArray[postroster[c][i].n], -1);

         }
      }
   }
   if (0) for (int j = 0; j < NUMMOL; j++) {       //humongous extra ineffecient function that makes sure all the
molecule data is synchronized
      if (1) for (int c = 0; c < WCELLNUM*HCELLNUM; c++){
         for(int i=0; i<=countpost[c]; i++){
            if (j == postroster[c][i].n) {
               int oughttabe = changecell(postroster[c][i].pos.x, postroster[c][i].pos.y);
               ASSERTEQ(MoveMolsArray[j].newcell, oughttabe);
               ASSERTEQ(MoveMolsArray[j].cell, c);
               ASSERTEQ(MoveMolsArray[j].index, i);
               if (j == 6127) {
                  printf("%f %f\n", postroster[c][i].pos.x, postroster[c][i].pos.y);
                  int x = 34;
               }
               goto away;
            }
         }
      }
      ASSERT(0);
      away: ;
   }
   ftime(&finish);
   calctime[procnum]+=(finish.time + finish.millitm/1000.0) - (start.time + start.millitm/1000.0);
}

/********
checks MoveMolsArray to make sure all particles are in the proper cells
********/
void RearrangeData(){
```

```
   postroster.check();
   for(int i=0; i< NUMMOL; i++){
      int destcell=MoveMolsArray[i].newcell;       //destcell is the new destcell
      int fromcell=MoveMolsArray[i].cell;          //fromcell is the old destcell
      if (destcell == fromcell) continue; //ASSERTNE(destcell, fromcell);
      int fromindx=MoveMolsArray[i].index;         //fromindx is position in old destcell
      if(destcell!=(-1)){                  //if it's not staying in it's destcell then...
         ASSERTEQ(destcell, changecell(postroster[fromcell][fromindx].pos.x,
postroster[fromcell][fromindx].pos.y));
         postroster.check();
         int destindx = countpost[destcell]+1;
         int lastindx = countpost[fromcell];
         int justshorten = fromindx == lastindx;
         ASSERTEQ(postroster[destcell][destindx].n, NOMOL);
         postroster[destcell][destindx]=postroster[fromcell][fromindx];
         if (!justshorten) postroster[fromcell][fromindx]=postroster[fromcell][lastindx];
         MoveMolsArray[i].cell=destcell;
         MoveMolsArray[i].index=destindx;
         if (!justshorten) MoveMolsArray[postroster[fromcell][fromindx].n].index=fromindx;
         postroster[fromcell][lastindx].n= NOMOL;
         countpost.set(destcell, destindx);
         countpost.set(fromcell, lastindx-1);
         MoveMolsArray[i].newcell=-1;
         postroster.check();
      }
   }
   postroster.check();
}

HANDLE start[NUMCORE], end[NUMCORE];
char myid[4][NUMCORE];

void ThreadFuncSecond(int numproc){
   if(NUMCORE==4){
      double beta=1-alpha;
      if(numproc==0)
         CalculateCells(0, HCELLNUM*WCELLNUM*alpha-1, dt, viewt, numproc);
      if(numproc==1)
         CalculateCells(HCELLNUM*WCELLNUM*alpha, HCELLNUM*WCELLNUM/2-1, dt, viewt, numproc);
      if(numproc==2)
         CalculateCells(HCELLNUM*WCELLNUM/2, HCELLNUM*WCELLNUM*beta-1, dt, viewt, numproc);
      if(numproc==3)
         CalculateCells(HCELLNUM*WCELLNUM*beta, HCELLNUM*WCELLNUM-1, dt, viewt, numproc);
   }
   else if (NUMCORE==2){
      if(numproc==0)
         CalculateCells(0, HCELLNUM*WCELLNUM/2-1, dt, viewt, numproc);
      if(numproc==1)
         CalculateCells(HCELLNUM*WCELLNUM/2, HCELLNUM*WCELLNUM-1, dt, viewt, numproc);
   }
   else {
      if(numproc==0)
         CalculateCells(0, 104165, dt, viewt, numproc);       //the middle processor has 1/6 the sim area
      if(numproc==1)
         CalculateCells(104166, 145832, dt, viewt, numproc);
```

```
        if(numproc==2)
            CalculateCells(145833, HCELLNUM*WCELLNUM-1, dt, viewt, numproc);
    }
}
/********
this is the function for each thread, which calls the calculation
********/
void ThreadFunc(char *MyID) {
    int i = (int)MyID;
    int numproc = MyID[2]-'0';

    printf("thread %s %d\n", MyID, numproc);
    for(;;){
        WaitForSingleObject(start[numproc],INFINITE);
        ResetEvent(start[numproc]);
        //printf("doing thing\n");
        ThreadFuncSecond(numproc);
        SetEvent(end[numproc]);
        //printf("thread %d done", numproc);
    }
}

extern "C" {
int cmp(const void *va, const void *vb) {
    double *a = (double *)va, *b = (double *)vb;
    if (*a == *b) return 0;
    return *a < *b ? -1 : 1;
}
}
/******
finds the point where 10% is higher and 90% is lower
******/
setmaxdens(){
    if (1) for (int i=0; i<WCELLNUM*HCELLNUM; i++){
        sortedindex[i]=densityindex[i];
    }
    qsort(sortedindex, WCELLNUM*HCELLNUM, sizeof(double), cmp);
    if (0) for (int i=0; i<WCELLNUM*HCELLNUM; i++){
        printf("%f\n",sortedindex[i]);
    }
    int indextouse=WCELLNUM*HCELLNUM*.97;
    double answer=sortedindex[indextouse];
    return answer;
}

int main (){
    struct timeb starta;
    struct timeb finisha;
    ftime(&starta);
    int *imagedata1[WCELLNUM], id1[WCELLNUM][HCELLNUM]; //image stuff
    int *imagedata2[WCELLNUM], id2[WCELLNUM][HCELLNUM]; //image stuff
    int *imagedata3[WCELLNUM], id3[WCELLNUM][HCELLNUM]; //image stuff
    if(1) for(int i=0; i<NUMMOL; i++)    //if a molecule stays in its cell, this is -1, else it's the new cell num
        MoveMolsArray[i].newcell=-1;
    if(1) for(int i=0; i< NUMCORE; i++)
```

```
    ifcollide[i]=0;
postroster.init();
if (1) for (int i = 0; i < WCELLNUM; i++) {
    imagedata1[i] = id1[i];
    imagedata2[i] = id2[i];
    imagedata3[i] = id3[i];
}
if (1) for (int i = 0; i < WCELLNUM; i++)
    for (int j = 0; j < HCELLNUM; j++) {
        imagedata1[i][j] = RGB(255,255,255);
        imagedata2[i][j] = RGB(50,50,50);
        imagedata3[i][j] = RGB(255,255,255);
    }
if(1) for (int i=0; i<WCELLNUM; i++)
    for(int j=0;j<HCELLNUM; j++){
        int n= (i)*HCELLNUM+j;
        densityindex[n]=0;
        samplevecindex[n]=0;
    }
init_genrand(0, (unsigned long)0);//time(0L));
printf ("hello world\n");

if (1) for (int i=0; i<NUMMOL; i++){                    //set molecule function- random pos+velo
    setmol(i,2, 0);                                     //method 0 for initial positions
}

dataset();      //sets up structures
static lastpercent = -1;

 //initializing threads
for(int i=1; i<NUMCORE; i++){
    start[i] = CreateEvent(NULL, TRUE, FALSE, NULL);
    end[i] = CreateEvent(NULL, TRUE, FALSE, NULL);
    myid[i][0] = 'M';
    myid[i][1] = 'e';
    myid[i][2] = '0' + i;
    myid[i][3] = 0;
    _beginthread( (void (__cdecl *)(void *))ThreadFunc, 0, myid[i]);
}
if (1) for (double t=0; t<MAXTIME; t+=dt){
    viewt=t;
    int percent = int(t*100/MAXTIME);
    if (lastpercent != percent) {
        printf("%d%%%s", lastpercent = percent, lastpercent%10 == 9 ? "\n" : " ");
    }
    // Optimizing Alpha for lowest possible time

    if(1) if(NUMCORE==4){
        double tpercell1= (calctime[0]+calctime[3]);
        double tpercell2= (calctime[1]+calctime[2]);
        if(tpercell1>.1 && tpercell2>.1){
            if(tpercell1>tpercell2){
                alpha = alpha-.01;
            }
            else{
```

```
            alpha= alpha+.01;
        }
        for(int i=0; i<NUMCORE; i++){
            //printf("Core %d has %f s\n", i, calctime[i]);
            calctime[i]=0;
        }
        //printf("alpha is %f\n", alpha);
    }
}

if(1){
    if(1)for (int i=1; i< NUMCORE; i++){
        SetEvent(start[i]);
    }
    ThreadFuncSecond(0);
    if(1)for (int i=1; i< NUMCORE; i++){
        WaitForSingleObject(end[i],INFINITE);
        ResetEvent(end[i]);
    }
}
else{
    CalculateCells(0, HCELLNUM*WCELLNUM-1, dt, t, 0);
}

stepstaken++;
RearrangeData();

double temp[WCELLNUM*HCELLNUM-1];
for(int i=0; i<WCELLNUM*HCELLNUM-1; i++){
    temp[i]=0;
}
double totaltemp=0;
double totalindivtemp=0;
int numparticles=0;
int numindivparticles=0;
if (1) for (int c=0; c<=WCELLNUM*HCELLNUM-1; c++){        //checks temperature distro
    numparticles=0;
    if (1) for (int i=0; i<=countpost[c]; i++){
        if((postroster[c][i].n) != NOMOL){
            numparticles++;
            numindivparticles++;
            temp[c]+=norm(postroster[c][i].velo);
            totalindivtemp+=norm(postroster[c][i].velo)*norm(postroster[c][i].velo);
        }
    }
    if (numparticles!=0){
        temp[c]=temp[c]/(numparticles);
        temp[c]= temp[c]*temp[c];
        totaltemp+=temp[c];
    }
}
totaltemp=totaltemp/WCELLNUM*HCELLNUM;
totalindivtemp=totalindivtemp/numindivparticles;
printf("totalindivtemp=%.12f, molmolcollide=%d\n",totalindivtemp, nummolmolcollide);
```

```
        if(DEBUGLEV>10)
            printf("checkpoint 6\n");
    }


    //Begin Image Writing Section
    if (1) for (int n=0; n<WCELLNUM*HCELLNUM; n++){
        if(samplevecindex[n]>0){
            newindex[n]=norm(vectorindex[n])/samplevecindex[n];
            vectorindex[n].x=vectorindex[n].x/samplevecindex[n];
            vectorindex[n].y=vectorindex[n].y/samplevecindex[n];
            vectorindex[n].z=vectorindex[n].z/samplevecindex[n];
        }
        else{
            newindex[n]=0;
            vectorindex[n]=vector(0,0);
        }
    }
     qsort(newindex, WCELLNUM*HCELLNUM, sizeof(double), cmp);
    int h=WCELLNUM*HCELLNUM*.9;
    if(1) for (int i=0; i<WCELLNUM; i++){
        for(int j=0;j<HCELLNUM; j++){
            int n= (i)*HCELLNUM+j;
            int hue=norm(vectorindex[n])/newindex[h];
            imagedata3[i][j]= HLStoRGB(hue, 240/2, 240);
        }
    }

    printf("printing densityindex\n");
    double maxdensityindex=setmaxdens();
    if(1) for (int i=0; i<WCELLNUM; i++){
        for(int j=0;j<HCELLNUM; j++){
            int n= (i)*HCELLNUM+j;
            if(densityindex[n]>maxdensityindex){
                imagedata1[i][j]=RGB(230,0,255);
                //imagedata2[i][j]=RGB(230,0,255);
            }
            else{
                int h= densityindex[n]*158/(maxdensityindex);
                imagedata1[i][j]= HLStoRGB(h, 240/2, 240);
                //imagedata2[i][j]= HLStoRGB(h, 240/2, 240);
            }
        }
    }

#define VPITCH 20
#define VECLEN 20
    if(1) for (int i=0; i<WCELLNUM; i+=VPITCH){
        for(int j=0;j<HCELLNUM; j+=VPITCH){
            vector sumvector=vector(0,0);
            for(int k=VPITCH/2-VPITCH; k<VPITCH/2; k++){
                for(int m=VPITCH/2-VPITCH; m<VPITCH/2; m++){
                    int realx = i+k;
                    int realy = j+m;
                    if (realx >= 0 && realy >= 0 && realx < WCELLNUM && realy < HCELLNUM)
```

```
                sumvector+=vectorindex[(i+k)*HCELLNUM+j+m];
            }
        }
        int supersonicflag=0;
        if(norm(sumvector)/400.>270)
            supersonicflag=1;
        sumvector.z = 0.0;
        double n = norm(sumvector);
        if (n < 1e-200) continue;
        vector startvec= vector(WCELLNUM-1-i,j);
        vector e = sumvector/n;
        vector e2 = e * VECLEN * (norm(sumvector)/400./270.);
        vector tofindtheta = e2.rotate(-PI/2.);
        //vector tofindtheta=(sumvector/n*VECLEN).rotate(-PI/2.);
        vector endvec=startvec+tofindtheta;
        int newxi=WCELLNUM-1-startvec.x;
        int newyi=startvec.y;
        int newxf=WCELLNUM-1-endvec.x;
        int newyf=endvec.y;
        //LineWrite(imagedata1, WCELLNUM, HCELLNUM, newxi, newyi, newxf, newyf, RGB(0,0,0), 1, 1);
        if(supersonicflag==0)
            LineWrite(imagedata2, WCELLNUM, HCELLNUM, newxi, newyi, newxf, newyf, RGB(0,0,0), 1, 1);
        else
            LineWrite(imagedata2, WCELLNUM, HCELLNUM, newxi, newyi, newxf, newyf, RGB(255,255,255), 1,
1);
        Circle(imagedata2, WCELLNUM, HCELLNUM, newxi, newyi, 3., 12., RGB(0,255, 255));
        imagedata2[newxi][newyi]=RGB(0,255, 255);
    }
  }
  //End Image Writing Section

  if(0){ //checking velocity distrubution
    int velodist[200];
    for (int i=0; i<200; i++){
        velodist[i]=0;
    }
    if(1) for (int i=0; i<WCELLNUM*HCELLNUM; i++){
        for (int j=(-1); j<countpost[i]; j++){
            int done=0;
            for (int h=0; h< 200 && done==0; h++){
                if (norm(postroster[i][j].velo)<7*h && norm(postroster[i][j].velo)>(7*h-7) ){
                    done=1;
                    velodist[h]=velodist[h]+1;
                    //printf("%d \n", 5*h);
                }
            }
            if (done==0){
                 double ne= norm(postroster[i][j].velo)/10;
                printf("%f", ne);
            }
        }
    }
    printf("BEGIN VELOSITY PRINTOUT!\n");
    if(1) for (int i=0; i<200; i++){
        //printf("%d, %d \n", i, velodist[i]);
```

```
            printf("%d \n", velodist[i]);
        }
        printf("END VELOSITY PRINTOUT!\n");
    }
    postroster[0].printhist();
    GifImageStoreAdd(WCELLNUM, HCELLNUM, imagedata1);
    GifImageStoreAdd(WCELLNUM, HCELLNUM, imagedata2);
    GifImageStoreAdd(WCELLNUM, HCELLNUM, imagedata3);
    printf("maxincell= %d \n",maxincell);
    printf("setmolperstep=%f \n", (countsetmol/(MAXTIME/dt)));
    printf("molmolcolide = %d \n", nummolmolcollide);
    ftime(&finisha);
    double totaltime=(finisha.time + finisha.millitm/1000.0) - (starta.time + starta.millitm/1000.0);
    printf("totaltime=%f \n", totaltime);
    magicalbreakpoint(100);
    return (0);
}
```

14.3 Random.cpp

```
//The random number generator that has been modified so that it's threadsafe.
//#include "stdafx.h"
// for Linux, use an empty file or comment out this line

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <time.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <float.h>
#include "random.h"


#ifndef max
#define max(a,b)            (((a) > (b)) ? (a) : (b))
#endif

#ifndef min
#define min(a,b)            (((a) < (b)) ? (a) : (b))
#endif

// random number module

void init_by_array(unsigned long [], int, int);
void init_genrand(int, unsigned long);

unsigned long genrand_int32(int);
double genrand_real2(int);

/*
  A C-program for MT19937, with improved initialization 2002/1/26.

  This is an optimized version that amortizes the shift/reload cost,
  by Eric Landry 2004-03-15.
```

```
   Before using, initialize the state by using init_genrand(seed) or
   init_by_array(init_key, key_length).

   Copyright (C) 1997--2004, Makoto Matsumoto, Takuji Nishimura, and
   Eric Landry; All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions
   are met:

     1. Redistributions of source code must retain the above copyright
        notice, this list of conditions and the following disclaimer.

     2. Redistributions in binary form must reproduce the above copyright
        notice, this list of conditions and the following disclaimer
        in the documentation and/or other materials provided with the
        distribution.

     3. The names of its contributors may not be used to endorse or
        promote products derived from this software without specific
        prior written permission.

   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
   "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
   LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
   A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT
   OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
   SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
   LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
   DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
   THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
   (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
   OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

   Any feedback is very welcome.
   http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
   email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)

   Reference: M. Matsumoto and T. Nishimura, "Mersenne Twister:
   A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number
   Generator", ACM Transactions on Modeling and Computer Simulation,
   Vol. 8, No. 1, January 1998, pp 3--30.
*/

/* Period parameters */
#define N 624
#define M 397
#define MATRIX_A 0x9908b0dfUL   /* constant vector a */
#define UPPER_MASK 0x80000000UL /* most significant w-r bits */
#define LOWER_MASK 0x7fffffffUL /* least significant r bits */

static unsigned long x[NUMCORE][N];      /* the array for the state vector  */
static unsigned long *p0[NUMCORE], *p1[NUMCORE], *pm[NUMCORE];
```

```
/*
   initialize with a seed

   See Knuth TAOCP Vol2. 3rd Ed. P.106 for multiplier.

   In the previous versions, MSBs of the seed affect only MSBs of
   the state.

   2002-01-09 modified by Makoto Matsumoto
*/
void
init_genrand(int procnum, unsigned long s)
{
  int i;

  x[procnum][0] = s & 0xffffffffUL;
  for (i = 1; i < N; ++i) {
    x[procnum][i] = (1812433253UL * (x[procnum][i - 1] ^ (x[procnum][i - 1] >> 30)) + i)
      & 0xffffffffUL;           /* for >32 bit machines */
  }
  p0[procnum] = x[procnum];
  p1[procnum] = x[procnum] + 1;
  pm[procnum] = x[procnum] + M;
}

/*
   initialize by an array with array-length

   init_key is the array for initializing keys

   key_length is its length

   2004-02-26 slight change for C++
*/
void
init_by_array(unsigned long init_key[], int key_length, int procnum)
{
  int i, j, k;

  init_genrand(procnum, 19650218UL);
  i = 1;
  j = 0;
  for (k = (N > key_length ? N : key_length); k; --k) {
    /* non linear */
    x[procnum][i] = ((x[procnum][i] ^ ((x[procnum][i - 1] ^ (x[procnum][i - 1] >> 30)) * 1664525UL))
          + init_key[j] + j) & 0xffffffffUL;  /* for WORDSIZE > 32 machines */
    if (++i >= N) {
      x[procnum][0] = x[procnum][N - 1];
      i = 1;
    }
    if (++j >= key_length) {
      j = 0;
    }
  }
  for (k = N - 1; k; --k) {
```

```
    /* non linear */
    x[procnum][i] = ((x[procnum][i] ^ ((x[procnum][i - 1] ^ (x[procnum][i - 1] >> 30)) * 1566083941UL)) - i)
      & 0xffffffffUL;           /* for WORDSIZE > 32 machines */
    if (++i >= N) {
      x[procnum][0] = x[procnum][N - 1];
      i = 1;
    }
  }
  x[procnum][0] = 0x80000000UL;          /* MSB is 1; assuring non-zero initial array */
}

/* generates a random number on the interval [0,0xffffffff] */
unsigned long
genrand_int32(int procnum)
{
  unsigned long y;

  if (!p0[procnum]) {
    /* Default seed */
    init_genrand(procnum, 5489UL);
  }
  /* Twisted feedback */
  y = *p0[procnum] = *pm[procnum]++ ^ (((*p0[procnum] & UPPER_MASK) | (*p1[procnum] &
LOWER_MASK)) >> 1)
    ^ ((unsigned)-(int)(*p1[procnum] & 1) & MATRIX_A);
  p0[procnum] = p1[procnum]++;
  if (pm[procnum] == x[procnum] + N) {
    pm[procnum] = x[procnum];
  }
  if (p1[procnum] == x[procnum] + N) {
    p1[procnum] = x[procnum];
  }
  /* Temper */
  y ^= y >> 11;
  y ^= y << 7 & 0x9d2c5680UL;
  y ^= y << 15 & 0xefc60000UL;
  y ^= y >> 18;
  return y;
}

/* generates a random number on the interval [0,0x7fffffff] */
long
genrand_int31(int procnum)
{
  return (long) (genrand_int32(procnum) >> 1);
}

/* generates a random number on the real interval [0,1] */
double
genrand_real1(int procnum)
{
  return genrand_int32(procnum) * (1.0 / 4294967295.0);
  /* divided by 2^32-1 */
}
```

```
/* generates a random number on the real interval [0,1) */
double
genrand_real2(int procnum)
{
  return genrand_int32(procnum) * (1.0 / 4294967296.0);
  /* divided by 2^32 */
}

/* generates a random number on the real interval (0,1) */
double
genrand_real3(int procnum)
{
  return (((double) genrand_int32(procnum)) + 0.5) * (1.0 / 4294967296.0);
  /* divided by 2^32 */
}

/* generates a 53-bit random number on the real interval [0,1) */
double
genrand_res53(int procnum)
{
  unsigned long a = genrand_int32(procnum) >> 5, b = genrand_int32(procnum) >> 6;

  return (a * 67108864.0 + b) * (1.0 / 9007199254740992.0);
}

/* 2002-01-09 These real versions are due to Isaku Wada */
/*
int
main(void)
{
  int i;
  unsigned long init[4] = { 0x123, 0x234, 0x345, 0x456 }, length = 4;

  init_by_array(init, length);
  printf("1000 outputs of genrand_int32()\n");
  for (i = 0; i < 1000; ++i) {
    printf("%10lu ", genrand_int32());
    if (i % 5 == 4) {
      printf("\n");
    }
  }
  printf("\n1000 outputs of genrand_real2()\n");
  for (i = 0; i < 1000; ++i) {
    printf("%10.8f ", genrand_real2());
    if (i % 5 == 4) {
      printf("\n");
    }
  }
  return 0;
}
*/

#undef N
#undef M
#undef MATRIX_A
```

```
#undef UPPER_MASK
#undef LOWER_MASK
// end random number module
```

14.4 Random.h

```
//header file
#define NUMCORE 2

long HLStoRGB(int Hue, int Lum, int Sat);

void init_by_array(unsigned long [], int, int);
void init_genrand(int, unsigned long);

unsigned long genrand_int32(int);
double genrand_real2(int);
```

14.5 Images.cpp

```
//File not written for this project: this is code adapted for use in writing lines, circles, and image files.
/* This file is to store all the interface-with-image code */

#include "stdafx.h"                                             // must be present for Visual C++. For Unix,
stdafx.h can be empty
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

#include <crtdbg.h>
#include <direct.h>
#include <time.h>
#include <malloc.h>
#include <winsock.h>
#include "images.h"

void failassert() {
        double p=3.14;                                          // set debugger breakpoint here
        (*(char *)0)++;                                         // crash
}

void asserterrorEQ(int t1, int t2, const char *m1, const char *m2) {
        printf("Assert error %s (= %d) != %s (= %d), but should\n", m1, t1, m2, t2);
        //getchar();
}

void asserterrorNE(int t1, int t2, const char *m1, const char *m2) {
        printf("Assert error %s (= %d) == %s (= %d), but shouldn't\n", m1, t1, m2, t2);
        //getchar();
}
//#define RGB(r,g,b)        (((unsigned char)(r)|((unsigned short)((unsigned char)(g))<<8))|(((unsigned
long)(unsigned char)(b))<<16))

#define H 240                                                   // range of hue
/* Hue helper function
```

```
Mix n1 and n2 according to the following function

High       ------------------
              /                \
             /                  \
            /                    \
Low        /                      ----------------
           0     1/6    1/3     1/2   2/3    5/6    1 hue-->
*/

int HueToRGB(int Low, int High, int Hue) {
        int Weight = max(min(-6*(abs(Hue%H-H/3)-H/3), H), 0);
        return (High*Weight + Low*(H-Weight))/H;
}

/* Convert Hue, Luminance, and Saturation to RGB

As hue chages, two of the colors will be fixed at brightness levels determined
by luminance and saturation, with the third color sweeping between these
intensities.

The average intensity of the two fixed colors is the luminance.

The difference between these two intensities is proportional to saturation and
is scaled never to produce a negative brightness nor a brightness greater than
100% at full saturation

Given these two fixed brightness levels, the helper function HueToRGB mixes
them for the individual red, green, and blue values.

*/
long HLStoRGB(int Hue, int Lum, int Sat) {
        int Spread = Sat*min(Lum, H-Lum)/H;
        int Low = ((Lum - Spread)*255 + H/2)/H;
        int High = ((Lum + Spread)*255 + H/2)/H;
        return RGB(HueToRGB(Low, High, Hue+H/3),
                        HueToRGB(Low, High, Hue),
                        HueToRGB(Low, High, Hue-H/3));
}
#undef H

// End code from Microsoft Article ID: 29240


// This class help draw the dynamic instance of each circuit a different color
// This class is to be implanted into the dynamic record for a circuit
// Subsequently, the drawing function can query it for a color
// The first time called, it randomly generates a color
// Subsequently, it returns the color to draw all the gates in a circuit

#define STARTHUE 0                                      // hue at input of circuit
#define ENDHUE 163                                      // hue at output of circuit
#define MAXGATES 305                                    // number of gates for full-scale color range

class RandomColor {
```

```
	int Color;
public:
	RandomColor() { Color = -1; }

	int MyColor(int gn) {
#if 1
		// form linear weighted average
		int HTarget = (STARTHUE*gn + ENDHUE*(MAXGATES-1-gn))/(MAXGATES-1);
/*
		// test for consistency of the RGB <-> HLS conversion
		if (0) {
			int H, L, S;
			RGBtoHLS(HLStoRGB(HTarget, HLSMAX/2, HLSMAX), H, L, S);
			ASSERT(L == HLSMAX/2);
			ASSERT(S == HLSMAX);
			ASSERT((H-HTarget)*(H-HTarget) <= 2);
		}
*/
		return HLStoRGB(HTarget, 240/2, 240);
#else
		while (Color == -1)
			Color = genrand_int32() & 0xffffff;
		return Color;
#endif
	}
};
#undef STARTHUE
#undef ENDHUE
#undef MAXGATES
#undef HLSMAX
#undef RGBMAX
#undef UNDEFINED

void PolygonWrite(int **image, int maxi, int maxj, double *polygon, int npts, int color) {
	double *dope = new double[npts + 1000];
	int nn;

	for (int i = 0; i < maxi; i++) {

		nn = 0;
		for (int n = 0; n < npts; n++) {

			double *p1 = &polygon[2*n];
			double *p2 = &polygon[2*(n != npts-1 ? n+1 : 0)];

			// skip if no intersection
			if (i < p1[0] && i < p2[0]) continue;
			if (i > p1[0] && i > p2[0]) continue;

			// skip if parallel to scan line
			if (p1[0] == p2[0]) continue;

			// skip top vertex
			if (p1[0] > p2[0] && i == p1[0]) continue;
			if (p1[0] < p2[0] && i == p2[0]) continue;
```

```
                        // compute intersection
                        double frac = (i-p1[0])/(p2[0]-p1[0]);
                        double intersec = p1[1] + frac*(p2[1]-p1[1]);

                        dope[nn++] = intersec;
                }

                // sort
                if (1) for (int ii = 0; ii < nn-1; ii++) {
                        for (int jj = ii+1; jj < nn; jj++)
                                if (dope[ii] > dope[jj]) {
                                        double t = dope[ii];
                                        dope[ii] = dope[jj];
                                        dope[jj] = t;
                                }
                }

                // write
                if (1) for (int ii = 0; ii < nn; ii+=2)
                        if (dope[ii] != dope[ii+1]) {
                                int start = 5000 - (int)(5000.-dope[ii]);
                                int end = 5000 - (int)(5000.-dope[ii+1]);

                                while (start != end) {
                                        if (start >= 0 && start < maxj)
                                                image[i][start] = color;
                                        start++;
                                }
                        }

        }

        delete dope;

}

void Circle(int **image, int maxi, int maxj, double x, double y, double diameter, unsigned int pts, int color) {

if (diameter < 2) diameter = 2;

        if (x + diameter/2 < 0) return;
        if (y + diameter/2 < 0) return;
        if (x - diameter/2 > maxi) return;
        if (y - diameter/2 > maxj) return;

        if (pts > 24)
                pts = 24;

        pts++;
        double poly[50], *p = poly;                             // data storage

        double effrad = diameter/2.;
        effrad *= sqrt(.5);
```

```
	if (1) for (unsigned int i = 0; i < pts; i++) {
		double theta = (0./pts + 360.*i/(pts-1))/(180./3.141592653589793238);
		*p++ =  x + cos(theta) * effrad - sin(theta) * effrad;
		*p++ =  y + sin(theta) * effrad + cos(theta) * effrad;
	}

	PolygonWrite(image, maxi, maxj, poly, pts, color);
}

void LineWrite(int **image, int maxi, int maxj, double x1, double y1, double x2, double y2, int color, int wid,
unsigned int pts) {

	double poly[160], *p = poly;

	// control number of end segments -- permits static buffer allocation
	if (pts < 2)
		pts = 2;
	else if (pts > sizeof(poly)/sizeof(double)/2)
		pts = sizeof(poly)/sizeof(double)/2;

	// compute parameters of line
	double len = sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1));
	if (len == 0) return;
	double vecx = (x2-x1)/len*wid*.5;
	double vecy = (y2-y1)/len*wid*.5;

	if (1) for (unsigned int i = 0; i < pts; i++) {
		double theta = (-90. + 180.*i/(pts-1))/(180./3.141592653589793238);
		*p++ =  x2 + cos(theta) * vecx - sin(theta) * vecy;
		*p++ =  y2 + sin(theta) * vecx + cos(theta) * vecy;
	}

	if (1) for (unsigned int i = 0; i < pts; i++) {
		double theta = (90. + 180.*i/(pts-1))/(180./3.141592653589793238);
		*p++ =  x1 + cos(theta) * vecx - sin(theta) * vecy;
		*p++ =  y1 + sin(theta) * vecx + cos(theta) * vecy;
	}

	PolygonWrite(image, maxi, maxj, poly, pts*2, color);
}

//back to browser
// Allocate an archival disk file, returning an open FILE * for writing its contents
// If link == 1, create a hyperlink to this file from archive.htm
// If stash != NULL, store the (relative) file name
FILE *ArchiveFile(const char *notation, const char *extension, int link, char *stash) {

	time_t long_time;
	time(&long_time);

	for (int i = 0; i < 500; i++) {

		struct tm *newtime = localtime(&long_time);

		const char *mon[12] = {
```

```
                        "jan",
                        "feb",
                        "mar",
                        "apr",
                        "may",
                        "jun",
                        "jul",
                        "aug",
                        "sep",
                        "oct",
                        "nov",
                        "dec",
                };

                char fn[200], fn2[200];
                sprintf(fn, "%s-%02d%s%02d-%02dh%02dm%02ds.%s", notation,
                        newtime->tm_mday, mon[newtime->tm_mon], newtime->tm_year-100, newtime-
>tm_hour, newtime->tm_min, newtime->tm_sec, extension);

                sprintf(fn2, "archive\\%s", fn);

                if (stash != NULL) sprintf(stash, "%s", fn);

#if __GNUC__ == 0
                // create the directory
                _mkdir("archive");
#endif

                // see if the file exists
                FILE *rval = fopen(fn2, "r");

                // doesn't exist, see if we can make it
                if (rval == NULL) {
                        rval = fopen(fn2, "wb+");
                        fclose(rval);
                        rval = fopen(fn2, "wb+");
                }

                // oops, exists, close it and try another
                else {
                        fclose(rval);
                        rval = NULL;
                }

                // got one
                if (rval != NULL) {

                        if (link != 0) {
                                FILE *es = fopen("archive.htm", "a");
                                fprintf(es, "<a href=\"archive/%s\">%s</a><br>\n", fn, fn);
                                fclose(es);
                        }

                        return rval;
                }
```

```
//                      else {
//                              double pi = 3.13;
//                      }

                        long_time++;
                }

                ASSERTALWAYS(0);

                return NULL;
}

// this is a buffer for bitmap images to be rendered semi-independently of pages
const int IMAGESTORELIMIT = 1000;
struct ImageStore {
                char Name[50];                                  // this will be the pathname for access, as in
http://127.0.0.1/bm0.bmp
                int Len;                                        // length of the image file (below)
                const char *MIME;                               // MIME type of data (NOT allocated from
storage pool)
                char *Dope;                                             // data buffer
} Buffer[IMAGESTORELIMIT];

// clear buffers
void ImageStoreClear() {
                for (int i = 0; i < IMAGESTORELIMIT; i++)
                        if (Buffer[i].Dope != NULL)
                                free(Buffer[i].Dope);
}

// search image store for an image matching the url
int ImageStoreSearch(char *n) {
                for (int i = 0; i < IMAGESTORELIMIT; i++)
                        if (strcmp(Buffer[i].Name, n) == 0)
                                return i;
                return -1;
}

void Dither(int x, int y, int **image, int lvr, int lvg, int lvb) {

                unsigned char maptabr[256], maptabg[256], maptabb[256];
                if (1) for (int i = 0; i < 256; i++) {
                        maptabr[i] = (i+255/2/lvr) * lvr / 255 * 255 / lvr;
                        maptabg[i] = (i+255/2/lvg) * lvg / 255 * 255 / lvg;
                        maptabb[i] = (i+255/2/lvb) * lvb / 255 * 255 / lvb;
                }

                for (int i = 0; i < x; i++) {
                        for (int j = 0; j < y; j++) {
                                int c = image[i][j];                    // get the color
                                unsigned char cr = c>>16;
                                unsigned char cg = c>>8;
                                unsigned char cb = c;
                                unsigned char other = c>>24;
```

```
                        unsigned char nr = maptabr[cr];      // cut the range
                        unsigned char ng = maptabg[cg];
                        unsigned char nb = maptabb[cb];

                        image[i][j] = other<<24 | nr<<16 | ng<<8 | nb;

                        char er = cr - nr;                   // the error
                        char eg = cg - ng;
                        char eb = cb - nb;

                        for (int n = 0; n < 15; n++) {
                                int ii = i, jj = j;
                                switch (n&3) {
                                        case 0:
                                                jj++;
                                                break;

                                        case 1:
                                                ii++;
                                                break;

                                        case 2:
                                                ii++;
                                                jj--;
                                                break;

                                        case 3:
                                                ii++;
                                                jj++;
                                                break;
                                }

                                if (ii < 0 || ii >= x || jj < 0 || jj >= y)
                                        continue;

                                if ((rand() & 1) == 0)
                                        continue;

                                c = image[ii][jj];
                                cr = c>>16;
                                cg = c>>8;
                                cb = c;
                                other = c>>24;
                                if (er != 0) nr = min(255, max(0, cr + (er>>2|1))), er -= nr - cr; else nr = cr;
                                if (eg != 0) ng = min(255, max(0, cg + (eg>>2|1))), eg -= ng - cg; else ng = cg;
                                if (eb != 0) nb = min(255, max(0, cb + (eb>>2|1))), eb -= nb - cb; else nb = cb;
                                image[ii][jj] = other<<24 | nr<<16 | ng<<8 | nb;

                                if (er == 0 && eg == 0 && eb == 0)
                                        break;
                        }
                }
        }
}
```

```
// add an image to the store
char *ImageStoreAdd(int x, int y, int **image) {
	int i;
	for (i = 0; i < IMAGESTORELIMIT; i++)
		if (Buffer[i].Dope == NULL)
			break;
	if (i == IMAGESTORELIMIT) return NULL;

	Buffer[i].MIME = "image/bmp";

	Dither(x, y, image, 40, 40, 40);

	FILE *f = ArchiveFile("image", "bmp", 0, Buffer[i].Name);

	int bytes_per_line=(x*24+31)/32*4;
	Buffer[i].Len = 54 + bytes_per_line * y;
	char *p = (char *)malloc(Buffer[i].Len);
	Buffer[i].Dope = p;

	*p++ = 'B';
	*p++ = 'M';
	*(*(long **)&p)++ = 14+40;                          // file size
	*(*(short **)&p)++ = 0;                             // reserved
	*(*(short **)&p)++ = 0;                             // reserved
	*(*(long **)&p)++ = 14+40;                          // offset bits
	*(*(long **)&p)++ = 40;                             // size
	*(*(long **)&p)++ = x;                              // width
	*(*(long **)&p)++ = y;                              // height
	*(*(short **)&p)++ = 1;                             // planes
	*(*(short **)&p)++ = 24;                    // bit count
	*(*(long **)&p)++ = 0;                              // compression
	*(*(long **)&p)++ = bytes_per_line*y;// image size
	*(*(long **)&p)++ = 75*39;                          // x_pixels
	*(*(long **)&p)++ = 75*39;                          // y_pixels
	*(*(long **)&p)++ = 0;                              // number colors
	*(*(long **)&p)++ = 0;                              // colors important

	if (1)
		for (int j = 0; j < y; j++) {
			if (1) for (int i = 0; i < x; i++) {
				*p++ = image[i][j];
				*p++ = image[i][j]>>8;
				*p++ = image[i][j]>>16;
			}
			for (int i = x*3; i < bytes_per_line; i++)
				*p++ = 0;
		}

	fwrite(Buffer[i].Dope, 1, Buffer[i].Len, f);
	fclose(f);

	return Buffer[i].Name;
}
```

```
const int LEVR = 6;                                     // number of levels of red in 256 color
palette
const int LEVG = 6;                                     // number of levels of green in 256 color
palette
const int LEVB = 6;                                     // number of levels of blue in 256 color
palette

void Putword(int w, FILE *f) {
        fputc(w, f);
        fputc(w >> 8, f);
}

void EncodeHeader(int x, int y, RGBQUAD *pPal, FILE *f, int ActualX = -1, int ActualY = -1) {
        fwrite("GIF89a", 1, 6, f);                      // GIF Header

        Putword(ActualX >= 0 ? ActualX : x, f);         // Logical screen descriptor
        Putword(ActualY >= 0 ? ActualY : y, f);

        fputc(0x80 | 7<<5 | 7, f);                      // various fields
        fputc(0, f);                                    // BackGround
        fputc(0, f);                                    // pixel aspect ratio

        for (int i = 0; i < 256; i++) {
                fputc(pPal[i].rgbRed, f);
                fputc(pPal[i].rgbGreen, f);
                fputc(pPal[i].rgbBlue, f);
        }
}

void EncodeExtension(int delay, FILE *f) {

        fputc('!', f);
        fputc(0xF9, f);

        struct anonymous_structure {
                unsigned char transpcolflag:1;
                unsigned char userinputflag:1;
                unsigned char dispmeth:3;
                unsigned char res:3;
                unsigned char delaylow;
                unsigned char delayhigh;
                unsigned char transpcolindex;
        } gg;

        gg.transpcolflag = 0;
        gg.userinputflag = 0;
        gg.dispmeth = 0;
        gg.res = 0;
        gg.delaylow = (unsigned)delay;
        gg.delayhigh = (unsigned)(delay)>>8;
        gg.transpcolindex = -1;
        fputc(sizeof(gg), f);
        fwrite(&gg, sizeof(gg), 1, f);
        fputc(0, f);
}
```

```
void EncodeLoopExtension(int n, FILE *f) {
	fputc('!', f);                                  // byte  1  : 33 (hex 0x21) GIF Extension
code
	fputc(255, f);                                  // byte  2  : 255 (hex 0xFF) Application
Extension Label
	fputc(11, f);                                   // byte  3  : 11 (hex (0x0B) Length of
Application Block (eleven bytes of data to follow)
	fwrite("NETSCAPE2.0", 11, 1, f);
	fputc(3, f);                                    // byte 15  : 3 (hex 0x03) Length of Data
Sub-Block (three bytes of data to follow)
	fputc(1, f);                                    // byte 16  : 1 (hex 0x01)
	Putword(min(65536, n), f);                      // bytes 17 to 18 : 0 to 65535, an unsigned integer in
lo-hi byte format.
                                                                // This indicate the number
of iterations the loop should be executed.
	fputc(0, f);                                    // bytes 19       : 0 (hex 0x00) a Data Sub-
block Terminator.
}

void EncodeComment(const char *m_comment, FILE *f) {
	long n = strlen(m_comment);
	if (n == 0) return;

	if (n > 255) n = 255;

	fputc('!', f);                                  // extension code:
	fputc(254, f);                                  // comment extension
	fputc(n, f);                                    // size of comment
	fwrite(m_comment, n, 1, f);
	fputc(0, f);                                    // block terminator
}

class BitEncoderState {
	FILE *file;

	int InitBits;                                   // Initial number of bits/code, which will be
one more than the # bits/pixel
	int CurBits;                                    // Current number of bits/code, growing

	int BufSize;                                    // GIF needs blocks starting with a count
	char Buf[256];

	unsigned long Acc;                              // Bit accumulator
	int NumBits;                                    // Mumber of bits valid in Acc

public:

	void Init(int ib, FILE *f) {
		file = f;

		CurBits = InitBits = ib;

		BufSize = 0;
```

```
	Acc = 0;
	NumBits = 0;
}

// Output one character, but put into buffer eventually transmitted with a count
void PutChar(int c) {
	Buf[BufSize++] = c;
	if (BufSize >= 254)					// this could be 255??
		Flush();
}

// Transmit characters remaining in buffer
void Flush() {
	fputc(BufSize, file);
	fwrite(Buf, 1, BufSize, file);
	BufSize = 0;
}

// GIF-specific variable bit size code Output
void PutCode(unsigned short code, short FirstFree, int Clear) {

	Acc |= (long)code << NumBits;

	NumBits += CurBits;

	while (NumBits >= 8) {
		PutChar(Acc);
		Acc >>= 8;
		NumBits -= 8;
	}

	// Resetting codes
	if (Clear) {
		CurBits = InitBits;
		return;
	}

	// Calculate maximum code word for this bit size -- have we exceeded?
	short MaxCode = 1 << CurBits;
	if (CurBits < 12) MaxCode--;

	if (FirstFree > MaxCode)		// Increase bit size
		CurBits++;
}

// Close output stream, transmitting residual symbol
void Close() {
	while (NumBits > 0) {
		PutChar(Acc);
		Acc >>= 8;
		NumBits -= 8;
	}

	Flush();
	fflush(file);
```

```
	}

};

const int HSIZE = 5003;

class GIFHashTable {
	long HashTab[HSIZE];
	unsigned short CodeTab[HSIZE];

	int Shift;                                          // shift limit
	long Index;                                         // index into hash table
	long Match;                                         // encoded c..symbol
public:

	// clear the hash table
	void ClearHash() {
		for (int i = 0; i < HSIZE; i++)
			HashTab[i] = -1;
	}

	void Init() {
		Index = 0;
		// maximum number of left shift places to avoid exceeding hash table size
		for (Shift = 0; 255<<(Shift+1) <= HSIZE; Shift++) ;
		ClearHash();
	}

	// Look up (c, Entry) in the cache
	// If found, update Entry with the value in the cache and return TRUE
	int Lookup(long c, short &Entry) {
		Match = (c << 12) + Entry;
		Index = (c << Shift) ^ Entry;

		long Disp = HSIZE - (Index != 0 ? Index : 1);

		// scan through hash table until we find either a match or an empty entry
		while (HashTab[Index] != Match && HashTab[Index] >= 0)
			if ((Index -= Disp) < 0)
				Index += HSIZE;

		if (HashTab[Index] == Match) {     // If a match, update Entry
			Entry = CodeTab[Index];
			return 1;
		}
		else                                                // If not a match, just return
			return 0;
	}

	// Add result of previous lookup to hash
	void Add(int FreeEnt) {
		CodeTab[Index] = FreeEnt;
		HashTab[Index] = Match;
	}
```

```
};

class GIFEncoderState {

	BitEncoderState Encoder;
	GIFHashTable HashTab;

	long CountDown;                                                    // raster scan of image

	// Get the next sequential pixel from the image
	// Does not dither, but reduces to a multi-level 256 color palete
	int GifNextPixel(int x, int y, int **image) {
		if (CountDown == 0) return EOF;
		CountDown--;

		int rgb = image[x - 1 - CountDown%x][CountDown/x];

		unsigned char r = rgb>>16;
		unsigned char g = rgb>>8;
		unsigned char b = rgb;
		return (r * (LEVR-1) / 255)*LEVG*LEVB + (g * (LEVG-1) / 255)*LEVB + (b * (LEVB-1) /
255);
	}

	void CompressLZW(int x, int y, int **image, int InitBits, FILE *f) {

		// Set up the necessary values
		int ClearCode = 1 << (InitBits-1);
		int EOFCode = ClearCode + 1;
		short FreeEnt = ClearCode + 2;

		Encoder.Init(InitBits, f);
		HashTab.Init();
		short Entry = GifNextPixel(x, y, image);

		Encoder.PutCode(ClearCode, FreeEnt, 0);

		for (short c; (c = GifNextPixel(x, y, image)) != EOF; ) {

			if (HashTab.Lookup(c, Entry))
				continue;

			Encoder.PutCode(Entry, FreeEnt, 0);
			if (FreeEnt < 4096)
				HashTab.Add(FreeEnt++);
			else {
				HashTab.ClearHash();
				FreeEnt = ClearCode + 2;
				Encoder.PutCode(ClearCode, FreeEnt, 1);
			}
			Entry = c;
		}
		// Put out the final code.
		Encoder.PutCode(Entry, FreeEnt, 0);
		Encoder.PutCode(EOFCode, FreeEnt, 0);
```

```
		Encoder.Close();
	}

public:
	void EncodeBody(int x, int y, int **image, FILE *f, int OffsetX = 0, int OffsetY = 0) {

		CountDown = (long)x * (long)y;

		fputc(',', f);

		Putword(OffsetX, f);
		Putword(OffsetY, f);
		Putword(x, f);
		Putword(y, f);

		char Flags=0x00;				// non-interlaced (0x40 = interlaced) (0x80 =
LocalColorMap)
		fputc(Flags, f);

		int InitCodeSize = 8;
		fputc(InitCodeSize, f);

		CompressLZW(x, y, image, InitCodeSize+1, f);

		 // fwrite out a Zero-length packet (to end the series)
		fputc(0, f);
	}

};

bool EncodeGIF(int x, int y, int **image1, int **image2, RGBQUAD *pPal, const char *m_comment, FILE *f) {

	class GIFEncoderState es;
	if (image2 != NULL) {
		EncodeHeader(x, y, pPal, f);
		EncodeLoopExtension(100000, f);
		EncodeExtension(50, f);
		es.EncodeBody(x, y, image1, f);
		EncodeExtension(50, f);
		es.EncodeBody(x, y, image2, f);
	}
	else {
		EncodeHeader(x, y, pPal, f);
		EncodeExtension(0, f);
		es.EncodeBody(x, y, image1, f);
	}
	EncodeComment(m_comment, f);
	fputc(';', f);				// fwrite the GIF file terminator
	return true;				// done!
}

// add a GIF image to the store
char *GifImageStoreAdd(int x, int y, int **image1, int **image2) {
	int bi;
  for (bi = 0; bi < IMAGESTORELIMIT; bi++)
```

```
		if (Buffer[bi].Dope == NULL)
			break;
	if (bi == IMAGESTORELIMIT) return NULL;

	Buffer[bi].MIME = "image/gif";

	//Dither(x, y, image1, LEVR, LEVG, LEVB);
			//DO I WANT THIS BACK? !!!!!
	if (image2 != NULL) Dither(x, y, image2, LEVR, LEVG, LEVB);

	FILE *f = ArchiveFile("image", "gif", 0, Buffer[bi].Name);

	// Synthesize a 256 (or less) color palette with some number of levels in each of rgb
	// The standard Internet palette has 6 levels in each color -- for a total of 216 colors
	RGBQUAD Pal[256];
	{
		RGBQUAD rgb;
		rgb.rgbRed = 0;
		rgb.rgbGreen = 0;
		rgb.rgbBlue = 0;
		rgb.rgbReserved = 0;
		for (int i = 0; i < 255; i++)
			Pal[i] = rgb;

		for (int r = 0; r < LEVR; r++)
			for (int g = 0; g < LEVG; g++)
				for (int b = 0; b < LEVB; b++) {
					rgb.rgbRed = r * 255 / (LEVR-1);
					rgb.rgbGreen = g * 255 / (LEVG-1);
					rgb.rgbBlue = b * 255 / (LEVB-1);
					rgb.rgbReserved = 0;
					Pal[r*LEVG*LEVB + g*LEVB + b] = rgb;
				}
	}

	{
		EncodeGIF(x, y, image1, image2, Pal, "Comment Goes Here", f);
		fclose(f);
		char buf[200];
		sprintf(buf, "archive\\%s", Buffer[bi].Name);
		f = fopen(buf, "rb");
		fseek(f, 0, SEEK_END);
		int len = ftell(f);
		Buffer[bi].Dope = new char[Buffer[bi].Len = len];
		fseek(f, 0, SEEK_SET);
		fread(Buffer[bi].Dope, 1, len, f);
	}
	fclose(f);

	return Buffer[bi].Name;
}

/*void GIFHeader(SOCKET theClient) {
	send(theClient, "HTTP/1.0 200\r\n", 14, 0);
	send(theClient, "Content-type: ", 14, 0);
```

```
	send(theClient, "image/gif", 9, 0);
	send(theClient, "\r\n", 2, 0);

	send(theClient, "\r\n", 2, 0);
}

// helper to write a 1x1 pixel gif of a certain color
void SmallGIFHelper(SOCKET theClient, unsigned char *tail) {
	GIFHeader(theClient);
	unsigned char header[73] = { 71, 73, 70, 56, 55, 97, 1, 0, 1, 0, 179, 0, 0, 0, 0, 0, 128, 0, 0, 0, 128, 0, 128,
128, 0, 0, 0, 128, 128, 0, 128, 0, 128, 128, 192, 192, 192, 128, 128, 128, 255, 0, 0, 0, 255, 0, 255, 255, 0, 0, 0, 255,
255, 0, 255, 0, 255, 255, 255, 255, 255, 44, 0, 0, 0, 0, 1, 0, 1, 0, 0, 4, 2, };
	send(theClient, (char *)header, 73, 0);
	send(theClient, (char *)tail, 4, 0);
}

// helper to write a 1x1 pixel gif of a certain color
void BigGIFHelper(SOCKET theClient, unsigned char *data, int len) {
	GIFHeader(theClient);
	send(theClient, (char *)data, len, 0);
}*/

double Clip(double c) {
	c = c + 255;
	c = c/2;
	if (c < 0) return 0;
	if (c > 255) return 255;
	return c;
}

double Clip2(double c) {
	if (c < 0) return 0;
	if (c > 255) return 255;
	return c;
}

class FRGB {
public:
	double r;
	double g;
	double b;
	FRGB() { }
	FRGB(int rr, int gg, int bb) { r = rr; g = gg; b = bb; }
	FRGB(double rr, double gg, double bb) { r = rr; g = gg; b = bb; }
};

void GIFEncode(FILE *out, const char *code) {
	char fname[100];
	sprintf(fname, "%s.gif", code);
	FILE *test = fopen(fname, "rb");
	int len = 0;
	int c;
	while ((c = fgetc(test)) != -1)
		len++;
```

```
	fclose(test);

	test = fopen(fname, "rb");

	fprintf(out, "    else if (stricmp(File, \"%s\") == 0) {\n        static unsigned char %s[%d] = { ", fname, code,
len);

	len = 0;
	while ((c = fgetc(test)) != -1) {
		fprintf(out, "%d, ", c);
		if (++len % 256 == 0)
			fprintf(out, "\n               ");
	}

	fprintf(out, "};\n        BigGIFHelper(theClient, %s, %d);\n    }\n", code, len);


	fclose(test);
}
```

14.5 Images.h

```
//header file
void failassert();

#define ASSERTALWAYS(c) if (!(c)) { fprintf(stderr, "assert failed"); fflush(stderr); failassert(); }

#define ASSERT(c) if (!(c)) { fprintf(stderr, "assert failed"); fflush(stderr); failassert(); }
//#define ASSERT(c)

#define ASSERTEQ(x, y) { int t1 = (x), t2 = (y); if (!(t1 == t2)) asserterrorEQ(t1, t2, #x, #y); }
#define ASSERTNE(x, y) { int t1 = (x), t2 = (y); if (!(t1 != t2)) asserterrorNE(t1, t2, #x, #y); }
#define ASSERTEQptr(x, y) { void *t1 = (x), *t2 = (y); if (!(t1 == t2)) asserterrorEQptr(t1, t2, #x, #y); }
#define ASSERTEQdouble(x, y) { double t1 = (x), t2 = (y); if (!(t1 == t2)) asserterrorEQ(t1, t2, #x, #y); }
void asserterrorNE(int, int, const char *, const char *);
void asserterrorEQ(int, int, const char *, const char *);
void asserterrorEQptr(void *, void *, const char *, const char *);

// define either of these to be nonzero to generate Output in the appropriate graphics format
#define GIF 1
#define BMP 1

const int IMGWID = 750; //HELLO IMAGE SIZE
const int IMGHGT = 750;



FILE *ArchiveFile(const char *notation, const char *extension, int link, char *stash);
void Circle(int **image, int maxi, int maxj, double x, double y, double diameter, unsigned int pts, int color);
char *ImageStoreAdd(int x, int y, int **image);
char *GifImageStoreAdd(int x, int y, int **image1, int **image2 = NULL);
void LineWrite(int **image, int maxi, int maxj, double x1, double y1, double x2, double y2, int color, int wid,
unsigned int pts);
```