Investigations of a Discrete Velocity Gas

Thesis by

David Benjamin Goldstein

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

California Institute of Technology

Pasadena, California

1990

(Submitted October 27, 1989)

Dedicated to my parents

Acknowledgments

I wish to express my gratitude to my advisor, Professor Bradford Sturtevant for his early support and continuing encouragement and advice throughout the span of our research. I would also like to thank Dr. Gene Broadwell for many hours of fruitful discussion in a field in which his ideas led the way.

Wen-King Su deserves my special thanks for his most generous patience in explaining the intrigues of C programming and parallel processing and for making a stubborn computer work when needed.

I owe a debt of gratitude to my friends at GALCIT for their inspiration both personally and professionally. Notable thanks go to Kazunori Kawasaki, Balu Nadiga and Dr. Takaji Inamuro for penetrating discussions and tangible contributions to the research effort.

Eternal thanks to my mother and father, grandparents, and siblings for their untiring confidence and support during my studies.

And especially, I would like to thank my wife Michal who provided an ongoing source of motivation and endured my late nights with the "machine" with great tolerance.

I wish to express my gratitude to the Submicron Systems Architecture Project, Department of Computer Science, Caltech, for providing computer facilities and expert advice.

Abstract

A new model of molecular gasdynamics with discrete molecular velocity components has been implemented for parallel computation. When the suitably normalized velocity components can take only integer values and time is discretized for digital computation, the particles travel between a regular array of points in physical and velocity space, and the gas is called a "lattice gas." Calculations of molecular motions are thereby simplified. The outcome of binary collisions between particles is determined by reflections about axes of symmetry in the center-of-mass frame of reference. The procedure speeds calculations of collisions. Of interest is the insight the discrete model provides into complex physical behavior and the effect that physically realistic simplifications have on the accuracy and speed of parallel calculations of a flow.

The equilibrium state of a discrete-velocity gas and the influence of limited velocity resolution are explained. It is found that the equilibrium velocity distribution functions of the present model agree with those of the discrete Boltzmann equation at very low velocity resolution and the continuous-velocity Boltzmann equation at higher velocity resolution. The time development of non-equilibrium velocity distribution functions is presented. The model is applied to unsteady flows involving strong shock waves, heat transfer between solid surfaces, and unsteady shear layer development.

When the model is applied to gas mixtures, numerical experiments show that the required number of values of each component of molecular velocity depends strongly upon the mass ratios of the particle species involved. However, fewer than ten values of each velocity component are necessary to produce results of satisfactory accuracy in calculations of a shock wave in a single species gas. A unique, self-adaptive mesh for parallel computation, used either for the present lattice gas model or earlier direct simulation Monte Carlo (Bird, 1976) models, is described. The mesh balances the load between the processors of the multicomputer and maintains the cell size at approximately a fixed number of local mean free paths throughout the flow field.

Contents

List of Figures

List of Tables

List of Symbols

CA	Cellular Automata
DSMC	Direct Simulation Monte Carlo
IDSMC	Integer Direct Simulation Monte Carlo
mfp	Mean Free Paths
b	Impact parameter
c'_m	Most probable speed
c_r	Relative collision speed
c_s	Root mean squared thermal velocity
$\overline{c}'$	Mean thermal speed
$\vec{\mathbf{c}}$	Vector velocity
d	Particle diameter
F	Single particle velocity distribution in phase space
f	Number density of particles in velocity space
j	Ratio of time step to collision time
k	Boltzmann constant

l	Ratio of cell size to mean free path
M_s	Shock Mach number
m	Particle mass
N	Number of particles in a small volume of physical space
n	Particle number density in physical space
P	Pressure
p_{xx}	X or normal component of the pressure tensor
q	Unit particle speed
R	Gas constant
$\vec{\mathrm{r}}$	Location in physical space
s	Slope of symmetry axis
T	Temperature
Δt	Time step
Δt_c	Time increment for one collision
v	Specific volume
(V_x, V_y, V_z)	Candidate point in velocity space
(X, Y, Z)	Cartesian direction
$(\Delta x, \Delta y, \Delta z)$	Cell dimensions
(u, v, w)	Velocity components

(x, y, z)	Cartesian position
β	The inverse most probable speed
χ	Rotation angle of relative velocity during collision
δ	Lattice spacing
λ	Mean free path
ν	Collision frequency
ρ	Density
σ	Collision cross section
τ	Mean free time
θ	Angle between boundary and VW plane
Ω	Collision solid angle
$()^*$	Post collision quantity
$()_0$	Upstream or equilibrium quantity
$()'$	Fluctuating quantity
$()''$	Inverse collision quantity

Chapter 1

Introduction

1.1 Motivation

The objective of the present research is to develop discrete-velocity molecular models that simplify the continuous-velocity model of molecules and make the computation of molecular motions more easily applicable to both rarefied and high density flows. The discrete-velocity model, in which particle velocity components take on only discrete values (*e.g.*, integers), is distinguished from the more physically realistic continuous-velocity molecular models by its elegant simplicity. Implemented with a method for directly simulating particle motions, discrete-velocity models represent complex physical phenomena that can be explored without the difficulty of having to solve the continuous-velocity Boltzmann equation.

1.2 Previous Work

The Boltzmann equation, an integro-differential equation, describes the evolution of the number of molecules having a given velocity in a flowing gas. Evaluation of the collision integral, which describes the gain and loss of particles from each velocity class, is particularly difficult. The equations of radiative transport, which are similar to the Boltzmann equation, can be greatly simplified by supposing that light travels along discrete rays or streams (Chandrasekhar, 1960), thus reducing the equations

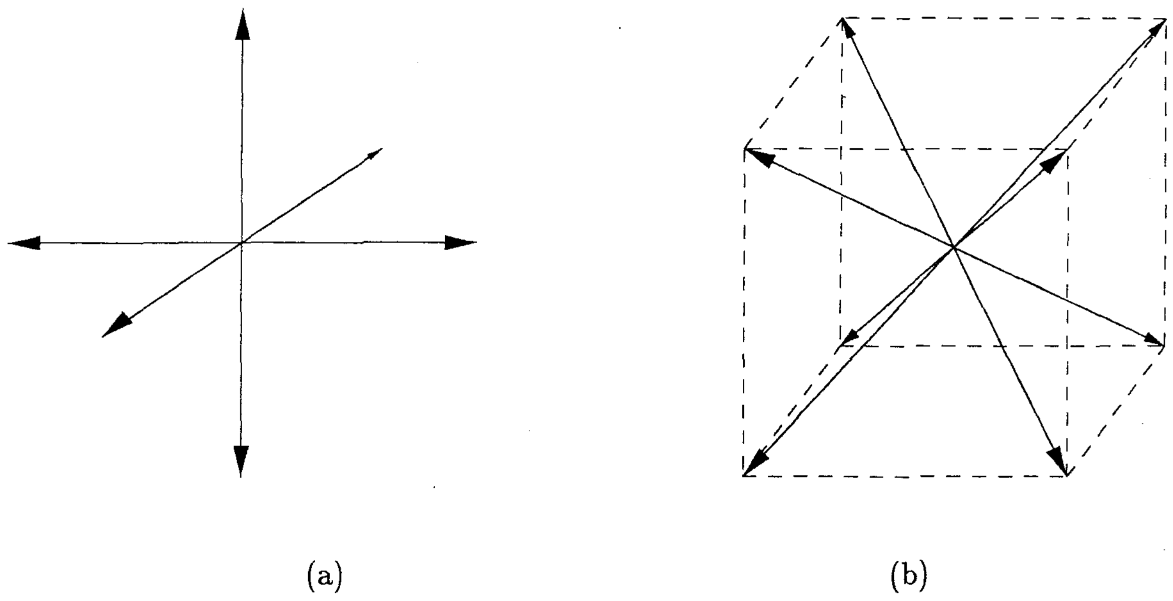

Figure 1.1: Two Broadwell models: (a) single-speed particles move only along Cartesian axes, (b) single-speed particles move at 45° to axes.

to a set of coupled differential equations. That assumption leads to the discrete ordinate method used to compute radiation transport through matter (Duderstadt, 1979). Gross (1960) suggested the possibility of applying a discretization approach to the Boltzmann equation and Broadwell (1963 and 1964a, b), citing such ideas, developed a model of a molecular gas in which particle velocity components are discrete; the molecules can move only with velocities $\mathbf{c}_i$, which belong to a finite set. The evolution of each point in velocity space can then be described by a differential equation. Broadwell used two different discretizations of velocity space; in one the particles move with unit speed in the six directions parallel to the Cartesian axes and in the other they move in the eight directions along lines at 45° to the axes (figure 1.1). Particle collisions (where particles of type $(\mathbf{c}_i, \mathbf{c}_j)$ become particles of type $(\mathbf{c}_k, \mathbf{c}_l)$) were modeled by probabilistic scattering rules that were incorporated into the gain and loss terms of the equation. The collisions, which exactly conserve mass, momentum, and energy, occur between particles that lie on diametrically opposite sides of a unit sphere centered at the origin of velocity space. Broadwell studied shock wave structure and low Mach number Couette and Rayleigh flows; in both cases, unexpectedly good correspondence between the extremely simple model and more refined theory was found. For this model, Caflisch and Papanicolaou (1979) have since derived Euler and Navier-Stokes equations and Caflisch (1979) has investigated shock wave profiles. A general discussion of discrete-velocity gases, including the Broadwell model, is given in the monograph by Gatignol (1975) and the review paper by Platkowski and Illner (1988), with particular emphasis on the mathematical subtleties. Recent studies of the discrete-velocity Boltzmann equation have investigated the relationship between the resulting continuum fluid transport coefficients and their corresponding continuous-velocity values (McNamera, 1988) and have greatly expanded the volume of velocity space that can be considered (Inamuro, 1989).

If space and time are discretized, a discrete-velocity model with only a few velocities may be simulated as a cellular automaton (CA). In such a simulation, particles travel along links between lattice sites in space and collide with other particles encountered at those sites. Consider a pair of two-dimensional models as examples, namely, the so-called HPP model (Hardy, de Pazzis, and Pomeau, 1973, 1976) with single-speed particles on a square lattice, and the FHP model (Frisch, Hasslacher, and Pomeau, 1986) with speed 0 and 1 particles on a triangular lattice. Figure 1.2 indicates the different collision possibilities. The circles drawn for the sample binary collisions are centered at the collision center-of-mass velocity and indicate the points that are possible post-collision velocities that conserve mass, momentum, and energy. (The shaded circles for the three particle collisions indicate the region of velocity space that contains such points. However, since in this work we are con-

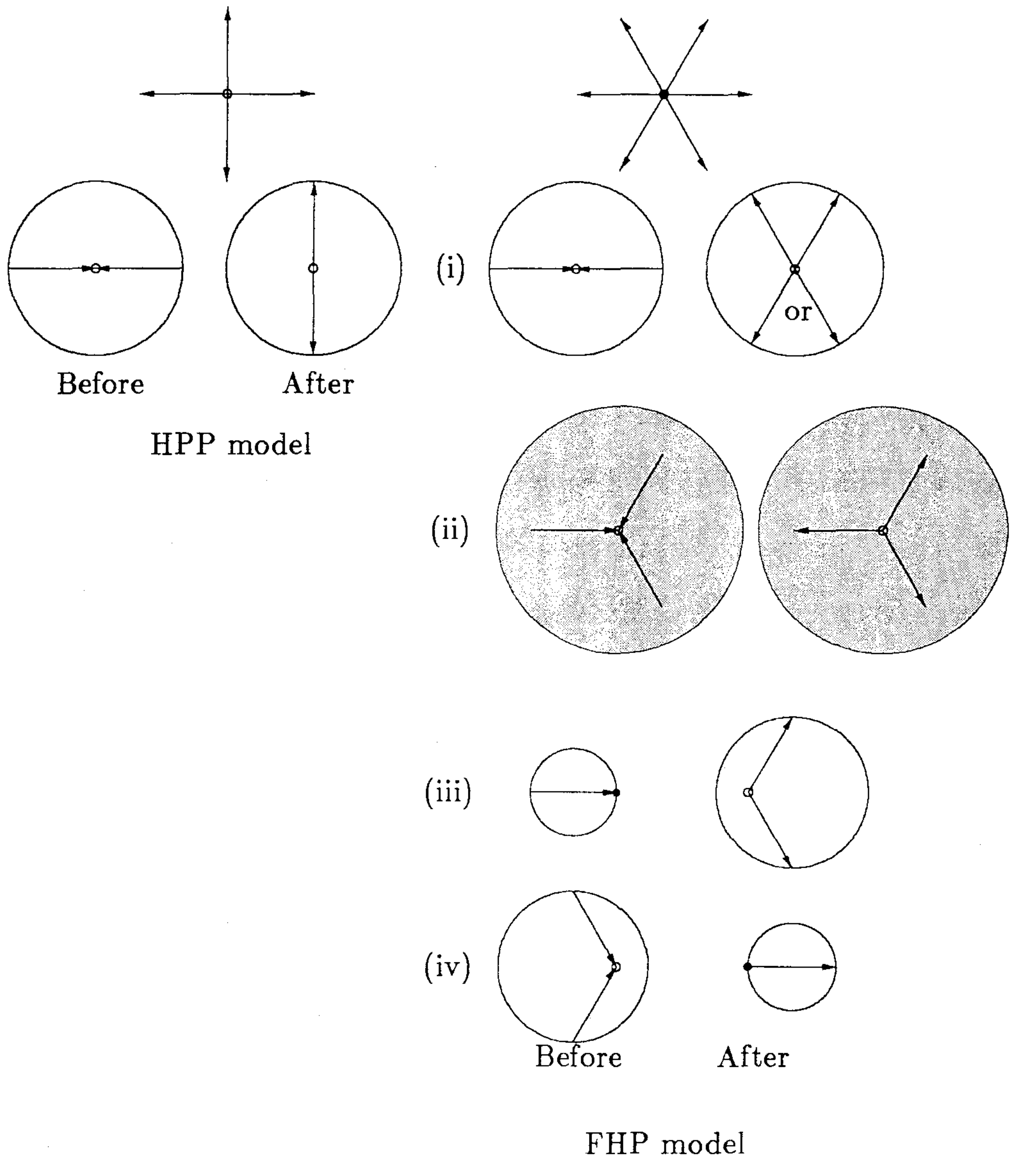

Figure 1.2: HPP and FHP cellular automata models with sample pre- and post-collision states.

cerned with models of dilute gases, ternary collisions are not considered further.) The circles have a diameter equal to the relative speed between the colliding particles. Of course, for the collision to yield discrete velocities, the outcome possibilities are limited to a few points on the circles. In the HPP model all possible circles are of a single diameter, signifying a single type of binary collision, whereas in the FHP model the circles are of three different sizes. Not all collision rules in the FHP model conserve energy during a collision (*e.g.*, FHP collisions (iii) and (iv), which lead to a change of the diameter of the collision circle). The simplicity of the set of collision rules that specify collision outcomes generally restricts the CA models to low velocity resolution, that is, only a few velocities. Also, for technical reasons, an exclusion principle requiring that two particles may not occupy the same point in phase space, is applied to each lattice site. CA simulations model an inherently compressible gas, but generally they have been applied in the low Mach number, incompressible limit, and collision models and particle densities with the lowest practical viscosity have been of primary interest (Henon 1987, Frisch *et al.*, 1986).

CA methods have been applied to several problems, including: shear flow between parallel plates (Kadanoff, 1987), shock waves (Nadiga *et al.*, 1989), the Rayleigh problem and flow over a flat plate normal to the flow (Long *et al.*, 1987), and flow about a cylinder (Doolen, 1988). The advantages of the CA are that simulations can be performed quickly using Boolean algebra and may even be directly computed in specialized hardware. The drawbacks are that the methods may not be more efficient than conventional finite difference techniques for continuum flows and may not yield results consistent with the Navier-Stokes equations; low resolution of phase space can lead to spurious conservation laws (Zanetti, 1988b) and to anisotropy of the stress tensor (Frisch *et al.*, 1986).

To simulate a gas temperature that is independent of flow velocity it is necessary to consider particles having more than one speed. Yet few discrete-velocity models beside those of Nadiga *et al.* (1989) and Inamuro (1989) include multi-speed particles. A purpose of the present work is to measure the accuracy of compressible flow simulations with discrete-velocity models having different levels of velocity resolution.

1.3 Present Approach

1.3.1 Model

A general model of a multi-speed discrete-velocity gas and a method for its implementation are presented in the next sections. The molecular models treated in this work were introduced by Goldstein *et al.* (1989), where the Broadwell model was generalized, and by Goldstein and Sturtevant (1989). In Broadwell's model, one type of collision and one particle speed were considered, whereas, in the present treatment, all collisions which exactly conserve mass, momentum, and energy are considered and any discrete particle velocity is possible. The collision model is discussed in terms of the points that lie on the intersections of the collision spheres and a discrete-velocity grid, and of the symmetries that relate to those points. The discretization of velocity space permits the discretization of particle positions onto a lattice in physical space and, thus, creates a variety of models that can bridge the gaps between the most basic discrete-velocity models, cellular automata, and continuous-velocity approaches. Multi-species particle interactions and diffuse and specular boundary conditions are also considered. The issues involved are (1) how can one practically model collisions between multi-speed particles, (2) what are the effects of the microscopic particle velocity resolution upon the macroscopic flow field quantities, (3) can a spectrum of discrete-velocity models bridge the gap between the most crude single-speed models and the continuous-velocity approach, and (4) can a discrete-velocity model be developed that is general enough to incorporate common real gas effects?

1.3.2 Method

Several recent applications have highlighted the need for further understanding of and a practical method for solving gas dynamic flows in the rarefied and transitional flow regimes. Interest in flows around satellites (Guernsey, 1986) and high altitude hypersonic vehicles (Bird, 1986), in rocket exhaust plumes (Hermina, 1988), and within vacuum chambers (Inamuro, 1989b) has emphasized the shortcomings of present models. These flows may involve gas chemistry, radiation, unsteadiness, phase change and complex geometry. In rarefied flows, no adequate finite difference approach is available because the full integro-differential Boltzmann equation is intractable. A direct particle simulation with the correct treatment of physics and a phenomenological treatment of the chemistry at the microscopic level is currently the best available method for such flows, but a molecular method can be extremely inefficient for calculating continuum flows due to the detail with which the calculations are made. Nevertheless, for the reasons stated above and also because it often happens that some parts of a flow field are rarefied while others are not, there has always been interest in extending the application of molecular simulation methods well into the continuum flow regime.

The methods of this work are patterned after the Direct Simulation Monte-Carlo method (DSMC) of Bird (1963, 1976). Such methods do not solve an explicit finite difference Boltzmann equation so the computational mesh can be independent of the coordinate system, and the calculation of flow over complex bodies may be significantly simplified. Complex molecular models can be incorporated as well to account for radiation, chemistry, and phase change. The present method is implemented with a unique self-adaptive mesh of cells for efficient computation on multi-computers.

The DSMC has previously been limited mainly to computation of rarefied flows by the substantial computational time required. Moderately complicated supersonic flows have been successfully simulated when the computational domain has been divided into sub-regions that could be computed separately (Hermina, 1988). Such a sub-division is impractical if the flow is subsonic or unsteady and the complete flow field must be simulated at once. Also, the grid required for complicated three-dimensional flows can be time consuming to generate. The present research investigates *simplifications* of the molecular models and methods that address these limitations. Emphasis is upon the influence of simplifications on the accuracy and speed of calculation and on the reduction of the memory demands upon the computer.

1.4 Outline of Present Work

In chapter 2, the theoretical underpinnings of the Boltzmann equation are briefly discussed. The present model is then described based on the physics of binary particle collisions both for identical particles and for particles having an integer mass ratio. Symmetries in the collision frame of reference are used to clarify the model and show how such general collisions can produce all possible discrete velocities.

Chapter 3 outlines three approaches that have been used to solve the Boltzmann equation and introduces the basic ideas of the direct simulation particle method. A rapid way to compute the collisions involving a look-up table is discussed and extended to handle a gas of particles that have different masses. The relationship between velocity and spatial resolution is explained and realistic boundary conditions are described. The concurrent computer used for simulations is briefly described and the algorithms for implementing the discrete-velocity and DSMC methods are detailed.

The effects of velocity resolution are then examined in following chapters through simulation and comparison to theoretical solutions or numerical solutions obtained with the standard DSMC method. The first simulations in chapter 4 involve the equilibrium states in a uniform stationary or translating gas. The velocity distribution functions and collision angle distribution are compared to theory. Next, a non-equilibrium distribution function is observed as it relaxes toward equilibrium; the process is compared to a DSMC simulation for particles having both the same and different masses. The discrete-velocity method is then applied to several flows in chapter 5. One-dimensional simulations of shock waves and heat transfer are examined followed by a two-dimensional unsteady shear flow used to demonstrate the application of the method for a complicated problem. Several conclusions about the model and its implementation are drawn in chapter 6. The computer code and instructions detailing its use are in the appendices.

Chapter 2

Theory

2.1 Classical Theory

In classical theory, a gas is considered to be made up of many chaotically moving particles. If the distance between the particles is significantly greater than the particle diameter then the gas is dilute and particle collisions are almost exclusively binary. Even if the particle number density is as high as that in the atmosphere at sea level, the distance between molecules is several times greater than their diameters, and the gas is still dilute. If there are no external force fields, the particles change their velocities only when they collide with each other or with a flow field boundary. In a small volume of space containing N particles, if dN is the number of particles in the velocity space $\vec{\mathbf{c}}$ to $\vec{\mathbf{c}} + \mathrm{d}\vec{\mathbf{c}}$, then $\mathrm{dN} = \mathrm{N}f(\vec{\mathbf{c}})$ where $f(\vec{\mathbf{c}})$ is the distribution function. That is, $f\mathrm{d}\vec{\mathbf{c}}$ is the probability that a given particle at some point in space will have a velocity between $\vec{\mathbf{c}}$ and $\vec{\mathbf{c}} + \mathrm{d}\vec{\mathbf{c}}$. f is normalized so that its integral over all of velocity space is unity. From f, it is possible to compute all the macroscopic variables of interest. For example, $\overline{u} = \int u f(\vec{\mathbf{c}})\mathrm{d}\vec{\mathbf{c}}$.

2.1.1 Boltzmann Equation

The single-particle velocity distribution of a large number of identical particles in a phase space consisting of velocity space, physical space, and time can also be defined as $F(\vec{\mathbf{c}}, \vec{\mathbf{r}}, t) = nf(\vec{\mathbf{c}})$, where n is the particle number density. Molecular chaos in a dilute gas implies that this function is sufficient to describe the probability of finding two particles in a given configuration.

The evolution of f is described by the integro-differential Boltzmann equation,

$$\frac{\partial}{\partial t}(nf) + \vec{\mathbf{c}}.\frac{\partial}{\partial \vec{\mathbf{r}}}(nf) = \int_{-\infty}^{+\infty} \int_{0}^{4\pi} n^2 (f'' f_1'' - f f_1) c_r \sigma d\Omega d\vec{\mathbf{c}}_1 \tag{2.1}$$

(Chapman and Cowling, 1952). The terms on the left-hand side represent the rate of change of the number of molecules in an element of phase space $d\vec{\mathbf{c}}d\vec{\mathbf{r}}$ and the convection of such molecules by $\vec{\mathbf{c}}$ through the surface of the spatial element $d\vec{\mathbf{r}}$. The right-hand side is an integral representation of the effect of collisions upon nf. The double primed terms represent collisions that scatter particles of class $\vec{\mathbf{c}}''$ and $\vec{\mathbf{c}}_1''$ into $\vec{\mathbf{c}}$ and $\vec{\mathbf{c}}_1$, a gain, while the unprimed terms represent particles scattered out of $\vec{\mathbf{c}}$ and $\vec{\mathbf{c}}_1$, a·loss. c_r is the relative particle speed, σ is the collision cross section, and Ω is the collision solid angle for this class of collisions. The formulation can be extended to include mixtures of gases and molecules that have various internal degrees of freedom (Chapman and Cowling, 1952).

It is possible to consider the velocity space to be discretized into a set of velocities, $\vec{\mathbf{c}}_i$, whose distribution is given by f_i (Broadwell, 1963). A discrete Boltzmann equation can then be written for each velocity type i as

$$\frac{\partial}{\partial t}(nf_i) + \vec{\mathbf{c}}_i.\frac{\partial}{\partial \vec{\mathbf{r}}}(nf_i) = \sum_{(\vec{c}_i, \vec{c}_j) \rightleftharpoons (\vec{c}_k, \vec{c}_l)} n^2 A_i (f_k f_l - f_i f_j) \tag{2.2}$$

where A_i incorporates the effects of σ, c_r, and Ω, and nA_i is a collision frequency. The left-hand side is essentially the same as in equation 2.1 but particles only travel with the discrete velocities $\vec{\mathbf{c}}_i$. The right-hand side is a summation over all velocity pairs that can produce or eliminate particles of type i through inverse and forward collisions. Whereas, in a continuous-velocity description a collision can produce an infinite variety of outcomes, it will be shown shortly that an important effect of discretization is to reduce the variety of possible collisions for each particle pair. Broadwell solved this coupled set of differential equations for a single-speed model in which the velocities lie in the six directions parallel to the Cartesian axes or in the eight directions along lines at 45^o to the axes (figure 1.1). The cost of explicitly solving the more general equation is proportional to the square of the volume of velocity space considered, as for the continuous-velocity Boltzmann equation, although the proportionality constant may be substantially smaller. The object of the present research is to indirectly solve the discrete Boltzmann equation by computing the motions of discrete-velocity molecules.

2.2 Discrete Velocity Model

What does it mean to describe molecules that only lie at discrete points in velocity space? An immediate consequence of the discretization is that the velocity components are quantized with the unit of velocity, q. Hence, the distribution of particle velocities is not continuous but approximates the distribution function, f, as a series of delta functions. In any given problem, whether q is 'small' or 'large' depends on the characteristic thermal speed of the molecules, for example, the rms thermal velocity,

$$c_s = (\overline{c'^2})^{1/2} = \sqrt{3RT} \ , \tag{2.3}$$

where R is the gas constant and T is the local temperature. For high-temperature gases, the velocity distribution function is 'wide,' and many molecular speeds occur, so q is small compared to c_s. For cold gases, the velocity distribution function is 'narrow,' so only a few molecular speeds occur, and q may be of the order of c_s.

In early discrete-velocity models, and with cellular automata, the variety of collision possibilities is limited and the number of molecular velocities is fixed. There is only a small range of temperatures in which such models produce realistic solutions and, for practical problems, the velocity distribution function is always narrow. In a gas for which the collision dynamics and molecular speeds are unrestricted and *all* discrete-velocity collisions that conserve mass, momentum, and energy are allowed, the width of the distribution function can increase more realistically with increasing temperature.

2.2.1 Collisions

2.2.1.1 Collision Dynamics

In this section the mechanics of collisions between discrete- or integer-velocity molecules are described. It is required that in every collision mass, momentum, and energy are exactly conserved. For simplicity, collision dynamics are discussed for a two-dimensional gas (all particle velocities lie in a plane). By a generalization to three-dimensions, all of the results presented are obtained with a three-dimensional model of a gas. Only the interaction of hard-sphere molecules is treated. Discussion of collisions between particles of differing mass is deferred until the next section.

A binary elastic collision between identical particles is drawn in the center-of-mass reference frame in physical space in figure 2.1. Particles i and j approach each other with a projected separation between their centers of b, the impact parameter. As viewed in this collision plane, the impact scatters the particles through an angle χ. If the mass of each particle is assumed to be unity, linear momentum conservation leads to,

$$\vec{\mathbf{c}}_i + \vec{\mathbf{c}}_j = \vec{\mathbf{c}}_i^* + \vec{\mathbf{c}}_j^* = 2\vec{\mathbf{c}}_{\mathrm{m}}, \tag{2.4}$$

where $\vec{\mathbf{c}}_{\mathrm{m}}$ is the velocity of the center-of-mass and the starred terms represent the

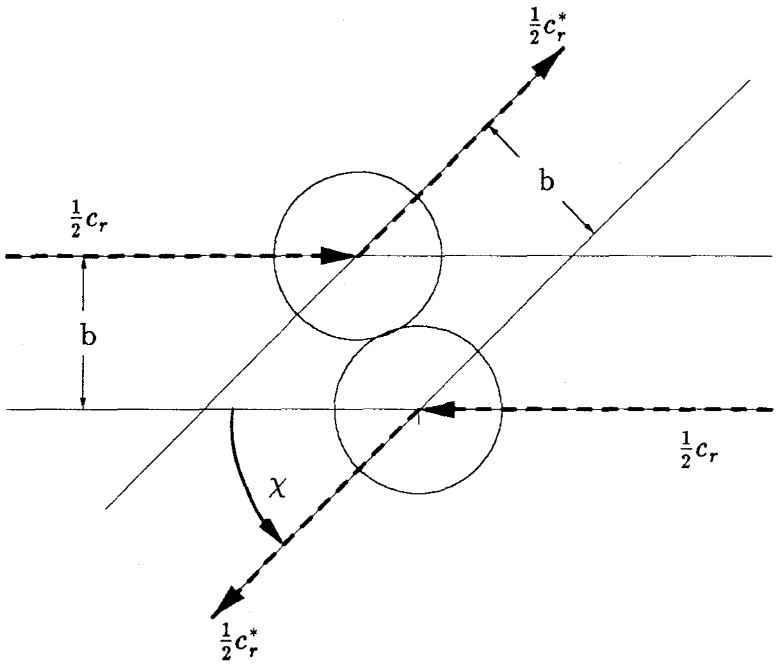

Figure 2.1: Binary particle collision between identical hard-sphere particles in the center-of-mass reference frame, viewed in the collision plane.

post-collision state. Energy conservation is described by,

$$c_i^2 + c_j^2 = c_i^{*2} + c_j^{*2}. \tag{2.5}$$

The relative velocities before and after the collision are

$$\vec{\mathbf{c}}_{\mathbf{r}} = \vec{\mathbf{c}}_i - \vec{\mathbf{c}}_j \tag{2.6}$$

and

$$\vec{\mathbf{c}}_{\mathbf{r}}^* = \vec{\mathbf{c}}_i^* - \vec{\mathbf{c}}_j^*. \tag{2.7}$$

These equations can be combined to show that

$$c_i^2 + c_j^2 = 2c_m^2 + \frac{1}{2}c_r^2 = 2c_m^2 + \frac{1}{2}c_r^{*2} = c_i^{*2} + c_j^{*2} \tag{2.8}$$

and, hence, that $c_r^* = c_r$. Therefore, the consequence of exact momentum and energy conservation is that the relative velocity vector after the collision has the same magnitude and is simply rotated by the collision angle χ about the center-of-mass velocity.

In a discrete-velocity gas, as in a continuous-velocity gas, the probability of a collision between two particles randomly selected from a small region of space is proportional to their relative speed. Figure 2.2 shows a simple quantitative example of a collision in velocity space in which one of the identical colliding particles initially has velocity $(u_i, v_i) = (4q, q)$ and the other $(u_j, v_j) = (2q, 5q)$. The relative velocity is the vector difference between these two velocities and the center-of-mass velocity lies at the center of the relative velocity vector. In a continuous-velocity gas, all points on a large circle centered at the center-of-mass velocity and of a diameter equal to the relative collision speed are equally likely post-collision results. Candidates for outcome discrete velocities in the example of figure 2.2 lie at the intersections of the velocity lattice with the large circle and are indicated by small open circles. If one

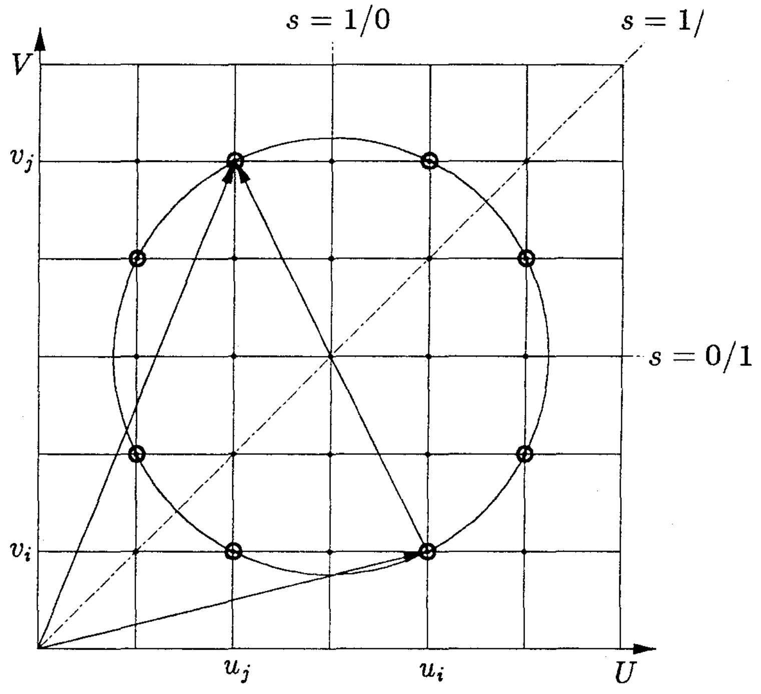

Figure 2.2: Points on collision circle. Reflections about 0/1, 1/1, and 1/0 are possible. Parity: EE (Even Even), Relative velocity: $(-2, 4)$. Lines of symmetry are dotted.

colliding particle is fast and the other is slow, as might occur in a high-temperature gas, the relative velocity will be large, so the diameter of the circle defined by the relative velocity vector will be large, and there might be many possible outcome velocities. By this method, velocities are automatically redistributed between the different regions of velocity space as required by the local temperature and collision dynamics.

In two-dimensions four different cases must be considered depending on the parity of the components of the relative velocity vector. In figure 2.2 both components are even (*e.g.,* (-2q, 4q), designated even-even or EE). In this case, the center-of-mass velocity falls on a lattice point in velocity space. If the parity is EE, generally there are four possible pairs of output velocities (including that in which the initial velocity vectors are simply interchanged with no apparent change on the figure). The three pairs that are different from the input pair can be constructed by reflections about the vertical, the 45^o, and the horizontal axes through the center-of-mass velocity. The three axes are designated by their slopes: 1/0 (vertical), 1/1 (45^o), and 0/1 (horizontal).

If both components of the relative velocity are odd (designated odd-odd or OO, figure 2.3), the center-of-mass falls at the center of a unit cell of the lattice in velocity space, and, again, four outcome pairs obtained by the same reflections as above are possible. On the other hand, when the relative-velocity components are of opposite parity (EO, figure 2.4), the center-of-mass velocity falls on the edge of a unit cell in velocity space, so no reflection about the 1/1 axis occurs, and only two outcome pairs are possible. In the EE and OO cases, the relative velocity vector can lie at 45^o (see the example in figure 2.5). In this case, there are only two outcome pairs possible. On the other hand, in the EE and EO configurations, the relative velocity vector can be vertical or horizontal (figure 2.6). In such a situation, with

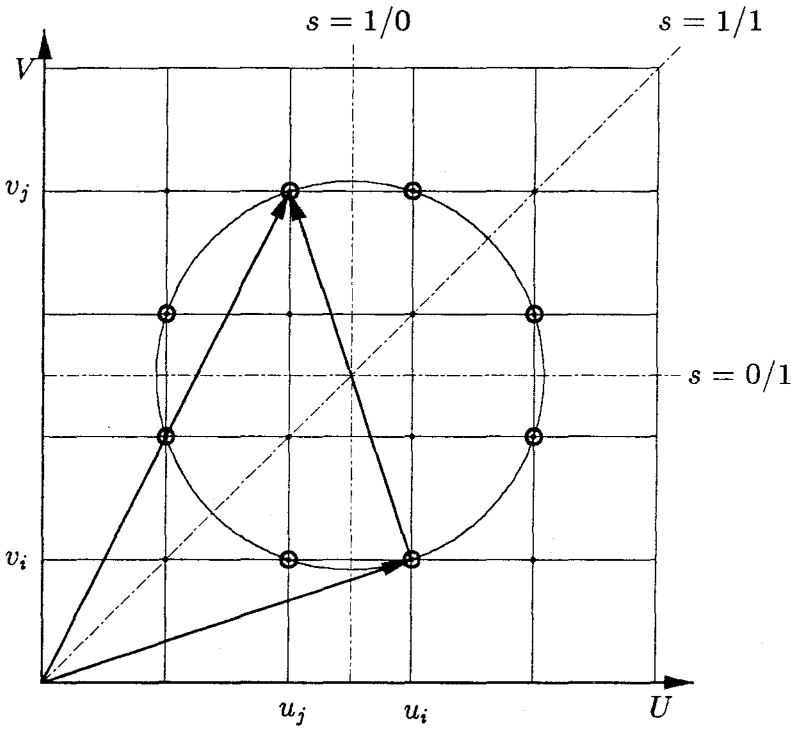

Figure 2.3: Points on collision circle. Reflections about 0/1, 1/1, and 1/0 are possible. Parity: OO, Relative velocity: (-1,3). Lines of symmetry are dotted.

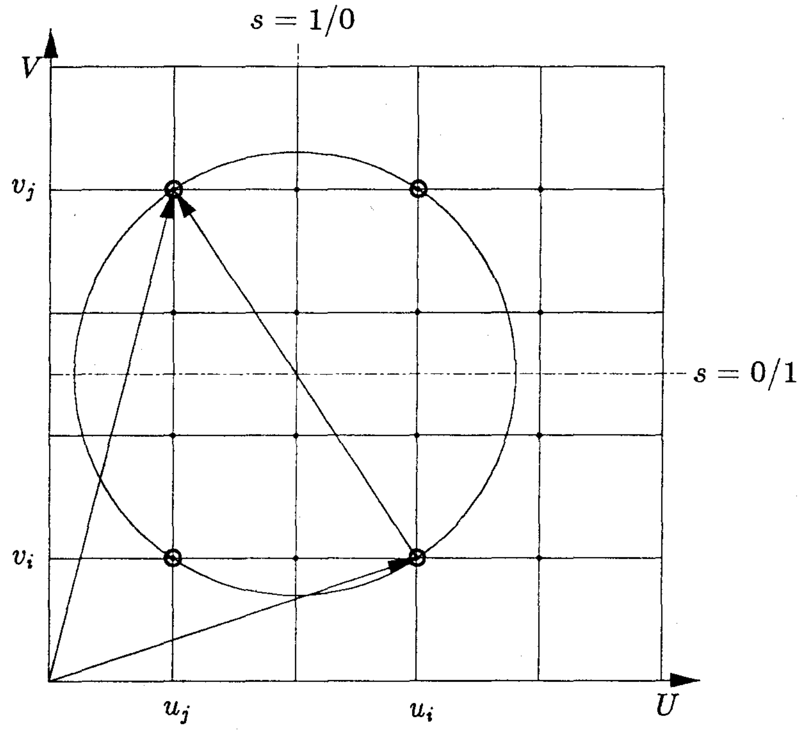

Figure 2.4: Points on collision circle. Reflections about 0/1 and 1/0 only are possible. Parity: EO, Relative velocity: (-2,3).

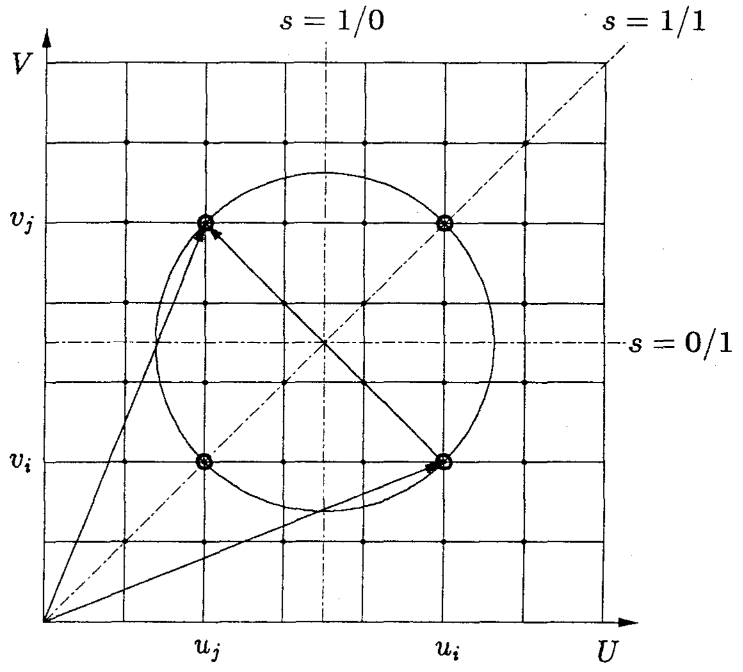

Figure 2.5: Points on collision circle. Reflections about 0/1 and 1/0 only are possible; reflection about 1/1 is degenerate. Parity: OO, Relative velocity: (-3,3).

EE, there are only two outcome pairs, while with EO, there is only one possible outcome - the same as the input.

2.2.1.2 Higher Order Symmetries

At higher relative velocities, *i.e.*, with greater velocity resolution, reflections about other axes can be made so more possibilities for outcomes arise. For example. the circle defined by a squared relative speed of $130q^2$ intersects components ($7q$, $9q$) and ($3q$, $11q$), which cannot be obtained from each other by reflections about

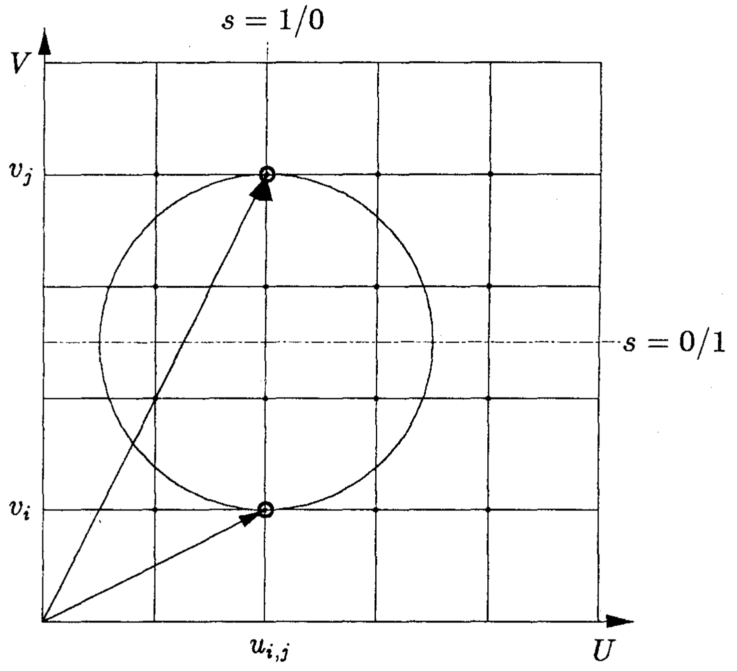

Figure 2.6: Points on collision circle. Reflections about 0/1 and 1/0 are degenerate. Parity: EO, Relative velocity: (0,3).

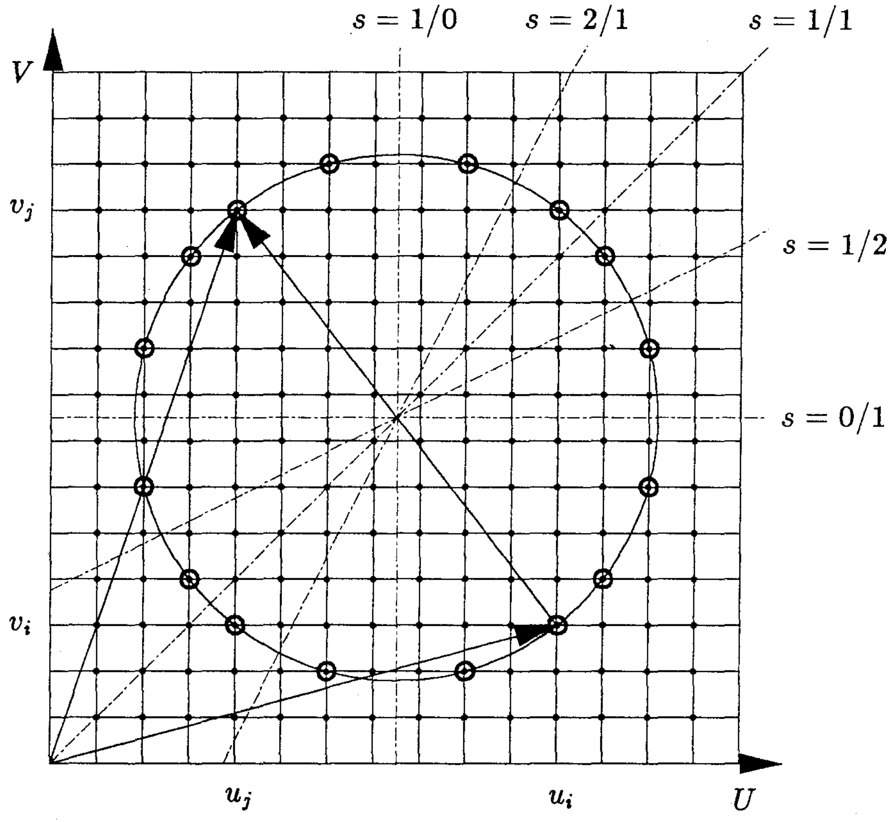

Figure 2.7: Points on collision circle. Reflections about 0/1, 1/1, 1/0, 2/1, and 1/2 are possible. Parity: OO, Relative velocity: (-7,9). Lines of symmetry are dotted.

1/0, 1/1, or 0/1 (see figure 2.7). Both of these points lead to 4 possible outcome pairs so there are a total of 8 possible outcome pairs for these input configurations. These outcomes, however, can be constructed from either input point by reflection about the additional axes of slope 1/2 and 2/1. In general, as the length of the relative velocity vector increases, symmetries about lines of slopes given by ratios of increasing values of whole numbers (*e.g.,* 1/3, 2/3, *etc.*) arise. The number of points in a quadrant that are intersected by the circle about the origin that passes through that point are indicated on the relative velocity lattice in figure 2.8. The boxed-in points are those that participate in symmetries more complex than 0/1, 1/0, and 1/1. If the relative velocity falls along an axis of symmetry, the possible outcomes are correspondingly reduced, and the case is degenerate. In a three-dimensional velocity space, there are $2^3 = 8$ parity cases, each possessing slightly different symmetry possibilities and degeneracies.

As the above discussion suggests, the number of possible collision outcomes increases rapidly as the velocity resolution increases. Figure 2.9 is a plot of the number of integer points in one octant of space intersected by the spheres with radii from 1 to 50. It is clear that with sufficiently large relative velocity many points on the collision spheres are available.

2.2.2 Multi-Species Collisions

The previous discussion has concerned only collisions between identical particles. If the particles have different masses, the collisions are not as simple; the center-of-mass velocity lies closer to the velocity of the more massive particle and not necessarily on an integer or half-integer velocity grid location. The geometry is again most easily seen in a two-dimensional example (figure 2.10). To conserve energy and momentum, each particle velocity after the collision must remain on

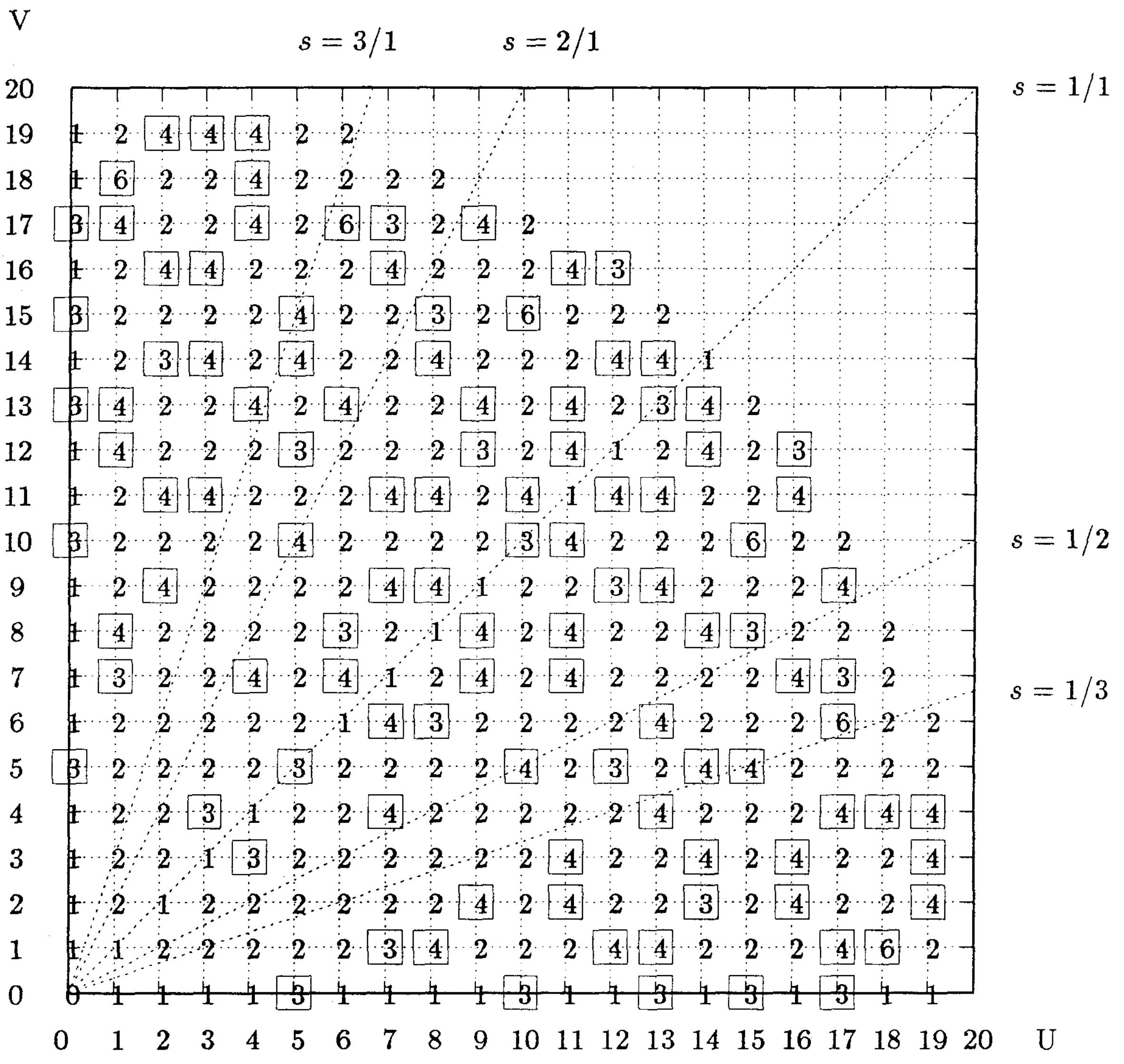

Figure 2.8: Each value indicates the number of points in the quarter of the plane that lie on a circle centered at (0,0) and that pass through the given point. Lines of slope 1/3, 1/2, 1/1, 2/1, and 3/1 are also shown.

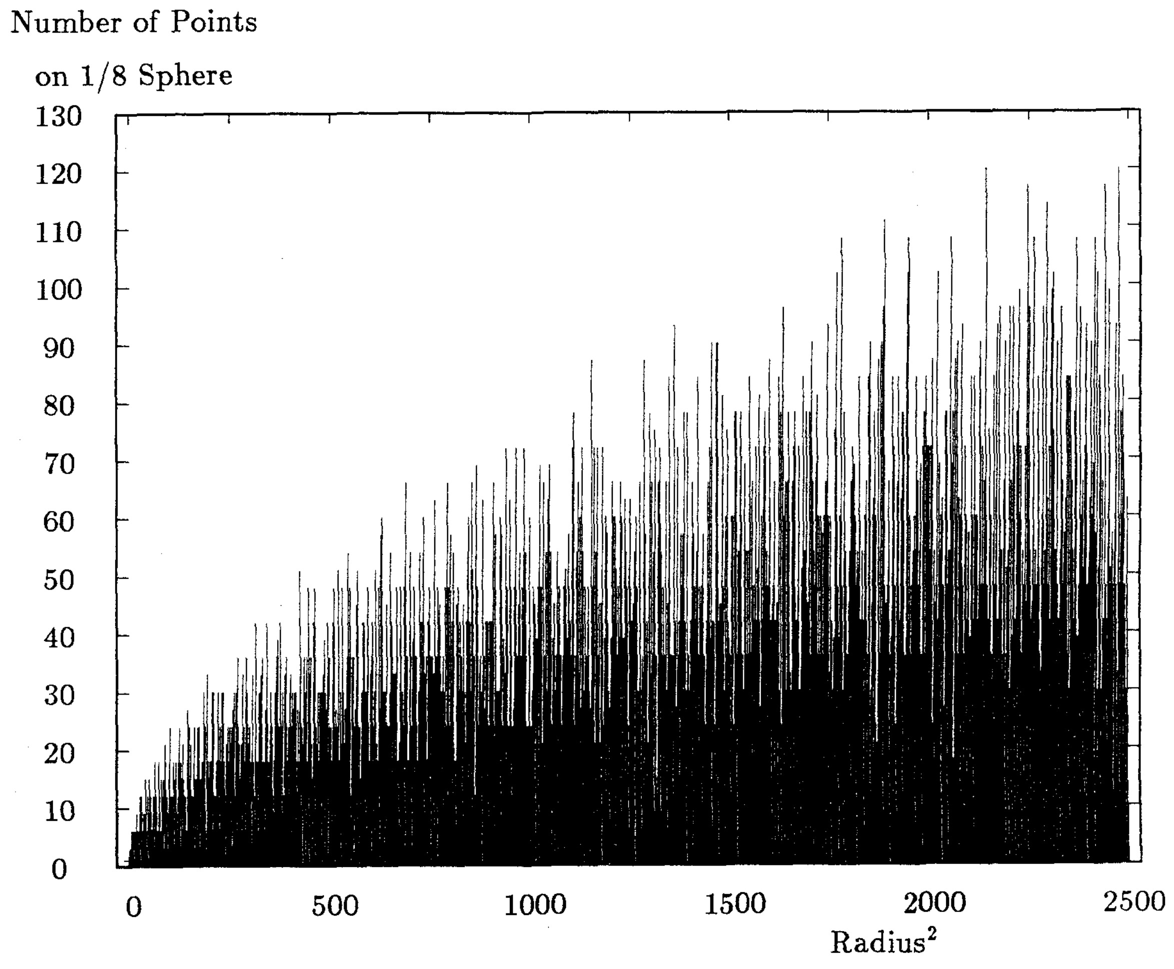

Figure 2.9: The number of the discrete points in one octant on all spheres whose radius is less than 50.

the circle, centered at the center-of-mass velocity, which passes through its own pre-collision velocity; the more massive particle's velocity remains on the smaller, inner circle while the lighter particle's velocity must lie on an opposing point on the large, outer circle. As for a single species collision, in three dimensions the particle velocities lie on spheres instead of circles and all of the discrete points on the spheres are assumed to be equally probable post-collision results.

To create outcome velocities that are integer-valued, the particles must have integer masses; non-integer masses generally lead to only a single intersection of the collision spheres with the velocity axes. At present, only the more restrictive case, in which the ratio of the particle masses is also an integer, is treated. This restriction allows collisions to be performed with a simple change of coordinates, as described below.

2.2.2.1 Velocity Sub-grid

In a discrete-velocity collision, there exists a sub-grid of points in velocity space on which the center-of-mass velocity may lie. For example, if the particles are identical (a mass ratio of one), the sub-grid points lie on integer or half-integer sites corresponding to the relative velocity components being of even or odd parity, respectively. If the particle mass ratio is 2, the sub-grid points lie on integer multiples of $q/3$ as indicated by the intersections of the dotted lines in figure 2.10. In general, the spacing between the sub-grid sites is $q/(m_1 + m_2)$; hence, the higher the mass ratio, the finer the sub-grid.

The sub-grid is used in place of the main grid to find the points on the small sphere of the heavy particle. In the sub-grid coordinate system whose origin lies on the center-of-mass velocity, the coordinates of the heavy particle velocity are the same as the relative velocity between the heavy and light particles on the main grid.

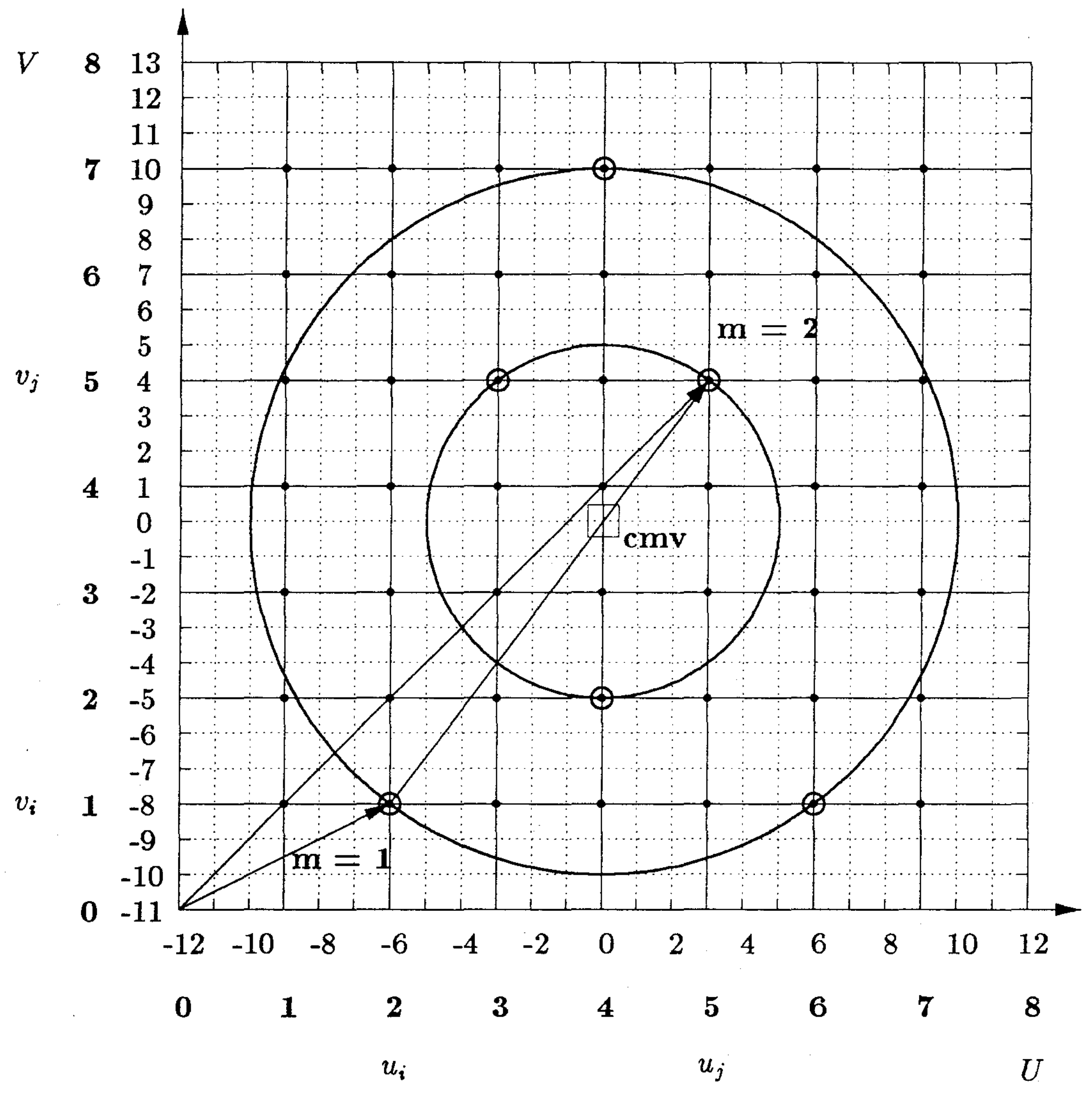

Figure 2.10: Multi-species collision. Mass ratio = 2. Center-of-mass velocity (cmv): (**4,3**$\mathbf{\frac{2}{3}}$), and (0,0) on sub-grid. Relative velocity: (3,4). Main-grid values are in **bold**.

For example, in figure 2.10 the relative velocity between the particles (*i.e.*, $\vec{\mathbf{U}}_{\mathrm{m}=2} - \vec{\mathbf{U}}_{\mathrm{m}=1}$) is (3,4) as are the coordinates of the heavy particle in the sub-grid system. The other points on the sub-grid circle can be found by the reflections described earlier for a single species collision. However, the only such points that are valid post-collision values, are those that also lie on the main grid, *e.g.*, $(4q,\ 2q)$, $(5q,\ 5q)$ and $(3q,\ 5q)$ in this example. Thus, to find the points on the circles, the points in the sub-grid system are found and those that do not also fall on the main grid are rejected. When an acceptable velocity has been found for the heavy particle, the corresponding value for the light particle lies on the opposite side of the outer circle $[(4q,\ 7q)$, $(2q,\ 1q)$ and $(6q,\ 1q)$, respectively$]$.

In contrast to single species collisions, effective multi-species collisions with low velocity resolution can be impossible because the smallest inner sphere on which there can be more than one point is a sphere of radius $\frac{1}{2}q$, corresponding to a relative speed equal to about one half of the mass ratio. Modeling higher mass ratios requires higher velocity resolution to avoid creating a gas with no effective inter-species collisions.

Bellomo and de Socio (1982, 1983) and Bellomo and Monaco (1984) also described a multiple species discrete-velocity gas based on Broadwell's six velocity model. In that model each particle type travels with a single speed that is proportional to the inverse of its mass; hence, the momentum of each particle is the same. In contrast to the present model, collisions only occur between pairs of particles having a stationary center-of-mass and thus collisions simply rotate the relative velocity vector about the origin of velocity space. Those collisions are represented by terms on the right-hand side of a discrete-velocity Boltzmann equation. Such an approach is not suited for the multi-speed model discussed in the present work, however.

2.2.3 Features Of Lattice Gases

Several fundamental differences exist between the present model and other discrete-velocity models. All such models begin from the same premise; particle velocity components only have discrete values. All solve a form of the discrete Boltzmann equation, either by analytic solution (Broadwell) or by direct particle simulation (CA and, as will be discussed in the next chapter, the present model). Previous models have been limited to low velocity resolution. To include more velocities in Broadwell's model would introduce more equations with many more collision terms on the right-hand side (Inamuro, 1988-1989a) while, in the CA models, a greater variety of collisions produces unwieldy rule sets. The present model, however, permits all collisions that conserve mass, momentum, and energy, and can thereby incorporate any level of velocity resolution.

New velocities can be created in all of the models. For example, if a homogeneous gas with velocities equally distributed between $u = \pm 1q$ is followed as it relaxes toward equilibrium, the different models would perform as follows: in the HPP (cubic lattice) model the first collisions would scatter particles into V (and W) $= \pm 1q$, thus creating velocities that were not originally present. As those collisions continue, the inverse collisions that deplete V (and W) would increase until equilibrium is reached. No speed other than $1q$ would occur because such collisions are not included in the model. Similarly, for the FHP model, collisions create the other four equal-speed velocities (five, if one includes speed $0q$). In the nine velocity model of Nadiga *et al.*(1989), speed $0q$ and $\sqrt{2}q$ particles are created by collisions between orthogonally moving speed $1q$ particles. The rule set for the nine-velocity model incorporates this new type of collision and is thus more general than earlier models.

The rules in the present model are less restrictive; indeed, *any* point in the discrete-velocity space is possible and may be reached through some sequence of collisions. Consider the following sequence of collisions that produces high-speed particles from a group of low-speed particles. All particles are again initially given velocities $u = \pm 1q$. Suppose the first collision occupies $(0q, 1q)$ and $(0q, -1q)$ (refer to figure 2.11 as a guide). Another collision occurs, this time $[(0q, 1q)$ and $(1q, 0q)]$ produce $[(0q, 0q)$ and $(1q, 1q)]$ thus occupying two further points in velocity space. There are now three speeds present $(0q, 1q$ and $\sqrt{2}q)$. Similar collisions could also have occupied $(-1q, 1q)$, which can then collide with $(1q, 1q)$ particles to produce particles with the velocity $(2q, 0q)$. $(1q, 2q)$ can then be reached by collisions between $(0q, 2q)$ and $(1q, 1q)$. To continue, similar collisions would occupy $(0q, 2q)$ which can, in turn, collide with particles at $(2q, 0q)$ to occupy $(2q, 2q)$, which has a speed of $\sqrt{8}q$, and so on. A small square region of velocity space can thus be occupied (figure 2.11).

It can be proved by induction that it is possible to build a sequence of such collisions that will occupy every point in velocity space. Consider unoccupied sites (u, v) along the center of the right edge of the square of velocity space already occupied (indicated by an A in the figure). (u, v) can be reached by a collision of the type $[(u - 1q, v - 1q)$ and $(u - 1q, v + 1q)]$, which yields $[(u, v)$ and $(u - 2q, v)]$. Sites adjacent to the corner of the square (given by a B) can then be reached by $[(u - 1q, v)$ and $(u, v - 1q)]$, yielding $[(u, v)$ and $(u - 1q, v - 1q)]$. Finally, the corners (indicated with a C) can be reached by that same type of collision, $[(u - 1q, v)$ and $(u, v - 1q)]$ yielding $[(u, v)$ and $(u - 1q, v - 1q)]$. The original square can, layer by layer, be expanded to fill all of a two-dimensional velocity space while a similar process on the faces of a cube can fill all of three-dimensional space.

An analogous procedure in a continuous-velocity model would eventually lead

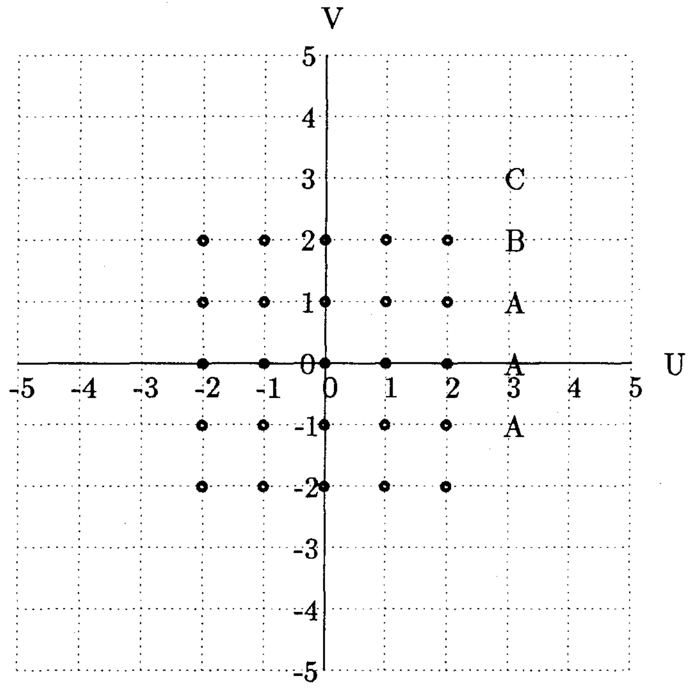

Figure 2.11: Occupied sites in velocity space. Symbols A, B, and C are described in the text.

to all of velocity space being occupied with a Maxwellian distribution of particles. The equilibrium distribution in a discrete-velocity gas can also be found (Nadiga, 1989 and Inamuro, 1988-1989a); it corresponds well with a simple Maxwellian except at the lowest velocity resolution. New velocities are not created during the solution of a particular problem; all velocities are naturally present at some level in an equilibrium gas with sufficiently general collision rules, and the circumstances of the problem to be solved simply alter their distribution.

Chapter 3

Method

A method for applying the model of a discrete-velocity gas to solve complicated problems is presented in this chapter. The collision integral in the continuous-velocity Boltzmann equation (eqn. 2.1) is extremely difficult to evaluate for flows of practical engineering interest, *i.e.*, three dimensional flows with multiple gas species. Three approaches, however, are worth mentioning. The first makes assumptions about the form of f, the second attempts to solve the exact Boltzmann equation, and the third replaces the Boltzmann equation with a simpler model equation. In the Lees (1959) moment method a form of f is written in terms of a multi-sided Maxwellian distribution function representing particles emitted by the different flow field boundaries. Moments of the Boltzmann equation are taken to generate a set of ordinary differential equations that can then be solved more readily. This approximate method is both mathematically complex and produces solutions that may depend on the assumptions made about f. In a second approach, not widely used, the left-hand side of the exact equation is numerically evaluated by a direct finite difference method and the collision integral on the right-hand side is approximated by a Monte Carlo technique (Nordsieck & Hicks, 1967). Steady state solutions are obtained by an iteration procedure from an assumed initial condition. The method is costly and is generally applied to one-dimensional flows. In the third approach (Bhatanagar, Gross, & Krook, 1954), the collision term of the Boltzmann equation is modeled by the expression $n\nu(f_0 - f)$, where ν is a collision frequency, and f_0 is the local equilibrium Maxwellian. Thus, $n\nu f$ represents the loss term while $n\nu f_0$ is the gain. This model of the gain assumes that the number of molecules scattered into $\vec{c}$ and $\vec{c}_1$ is that which would be found from an equilibrium gas with the same velocity and temperature as f. While the model is correct in the collisionless and continuum limits, its validity in the transitional regime is not assured.

None of these methods has proven acceptable for application to flows in which departures from equilibrium are large, for example shock waves, or to flows closer to equilibrium with real gas effects and complex geometry. In fact, the most widely used method bypasses explicit solution of the Boltzmann equation and instead computes the flow by directly simulating the motions and collisions of representative molecules. It is called the direct simulation Monte Carlo (DSMC) method and was introduced by Bird (1963 and 1976).

3.1 Direct Simulation Monte Carlo Method

The present discrete-velocity model is implemented with a method patterned after the DSMC. In the conventional DSMC, a small sample of molecules (typically tens to hundreds of thousands) is taken to represent a flowing gas. The DSMC is a stochastic method and it requires many simulated particles to provide a smooth representation of the distribution function. Fewer particles are needed to produce adequate values of the integrated macroscopic quantities that are generally of greater interest. Space is divided into sampling cells whose size (Δx, Δy, Δz) is small compared to the mean free path λ, and time is discretized into steps Δt that are smaller than the mean molecular collision time τ. Only binary collisions are treated, so the gas is, by definition, dilute. Particles within a cell are randomly selected to collide through an acceptance/rejection scheme that forces the probability of a collision to be proportional to the relative collision speed. A collision time counter in each cell insures the correct collision rate and tracks the cell's progress relative to that of other cells. For the simulation of single species particles there is only one time counter that is incremented by

$$\Delta t_c = \frac{1}{Nn\sigma c_r} \tag{3.1}$$

after each collision and collisions continue until the cell time exceeds the global flow time. This procedure can be shown after many collisions to lead to the correct collision rate (Bird, 1976). The particle cross section is normalized to be

$$\sigma = \frac{1}{n_0 \lambda_0 \sqrt{2}}. \tag{3.2}$$

The subscript 0 refers to the initial equilibrium state. When a binary mixture of species is considered, this normalization must be altered. Four separate cell time counters t_{ab} can be used, one for each type of collision (Bird, 1976). Expanding upon the procedure of equation 3.1, t_{ab} is then incremented by $\Delta t_{c_{ab}}$,

$$\Delta t_{c_{ab}} = \frac{1}{N_a n_b \sigma c_r} + \frac{1}{N_b n_a \sigma c_r} \tag{3.3}$$

for each a,b type of collision. For example, type 1,2 collisions are performed until t_{12} exceeds the flow time at which point another type of collision is considered. From the definition of the mean free path for hard sphere particles in a binary gas (Bird, 1968), the particle cross section is found to be

$$\sigma = \frac{1}{\lambda_0 N_0 \sqrt{2}} [\frac{1}{1 + \frac{n_{02}}{n_{01}}\sqrt{\frac{1}{2}(1 + \frac{m_1}{m_2})}} + \frac{1}{1 + \frac{n_{01}}{n_{02}}\sqrt{\frac{1}{2}(1 + \frac{m_2}{m_1})}}]. \tag{3.4}$$

The subscripts 1 and 2 refer to individual particle species of mass m.

Thus, the Monte Carlo game is played twice; once for the selection of the collision pair and again for the outcome of the collision. In the selection of collision pairs, the distribution of relative velocities is not known *a priori* for a non-equilibrium gas. Known distributions are used, however, for the selection of collision impact parameter, b, and azimuth angle, α; b is chosen to be uniform between 0 and $d/2$ (half the particle diameter) and α to be uniform between 0 and 2π.

After all the cells have completed their collisions, the particles are moved in free flight to locations appropriate for the beginning of the next time step. The resulting computed flow may be unsteady but if the flow does settle down to a steady state, a smooth solution may be obtained as an ensemble average of several late time states of the initial flow. The value of any macroscopic quantity in a cell is found by averaging over the values carried by the particles of the cell. Hence, the statistical scatter in the quantity declines as the inverse square root of the number of particles is considered.

The DSMC has significant advantages over other methods primarily because evaluation of the complex collision integral is replaced by the calculation of many representative collisions, a procedure that turns out to be more efficient. In contrast to the other methods, the cost of the collision calculations does not depend on the volume of velocity space considered but only on the number of particles involved. The free flight translation of the particles during the movement phase replaces the evaluation of the left-hand side of the Boltzmann equation. While other methods must maintain the distribution function for all of velocity space at each point in physical space, the DSMC must only retain several tens of particles in each cell. During a DSMC simulation the distribution function does not need to be reconstituted from the individual particle velocities; only moments of the distribution function are usually determined.

The DSMC method has some significant disadvantages as well. For low Knudson numbers, the physical properties (mass, position, velocity, *etc.*) for many particles must be stored in computer memory thereby limiting the size of the simulation. For complicated problems, the computational time can become excessive if the cell size and time step values are kept less than λ and τ as is required. While exceeding these values has generally been found to simply smooth the flow field gradients, an otherwise correct solution of the Boltzmann equation is not then assured. It has also been found that the DSMC is not easily implemented to take full advantage of vectorization, which is essential for efficient use of today's most powerful computers (McDonald and Baganoff, 1988). Central to the collision cycle is a procedure in which particle pairs are randomly selected and subjected to various tests; both the selection and testing processes inhibit vectorization. Much of the rest of the method, however, can be vectorized (Usami *et al.,* 1989, McDonald and Baganoff, 1988, Feiereisen, 1989, Woronowicz and McDonald, 1989). Conversely, the DSMC is a prime candidate for parallel processing because individual particles are moved independently, there are no long range interparticle forces, and collision partners are drawn only from the same locality.

An early version of the DSMC was adapted to massively parallel computation by writing a new code (Kawasaki, 1985) in the C programming language[1] for Caltech's cosmic computing environment. In the cosmic environment a computation is defined in terms of interacting processes that may reside on a host computer or a parallel processor (Su, 1985 and Athas, 1988). The present research version of the parallel DSMC has been considerably improved and is now run on the Intel iPSC/1, iPSC/2 or the Symult 2010 computers. Though the molecules are presently treated as elastic hard spheres, the phenomenological models of real-gas effects that have

[1]The C language is now the most appropriate for these applications because massively parallel computation is still in the early stages of development and Fortran compilers generally have not been highly developed or are not available.

been developed at other institutions (Borgnake and Larson, 1975, Bird, 1968-1983, Hermina, 1986) could easily be incorporated in the future.

3.2 Integer Direct Simulation Monte Carlo Approach

The discrete-velocity or integer DSMC (IDSMC) method is similar to the DSMC, but the velocity components of the particles are defined as integers. Although, in principle, the number of values that the velocity components can assume is infinite, in practice, one byte (8 bits) provides enough velocity resolution for flows of a single species gas even up to hypersonic Mach numbers.

3.2.1 Discretization of Phase Space

3.2.1.1 Velocity Discretization

Initially the velocity resolution is, in effect, set by the choice of the cell size, Δx, compared to the distance traveled by a particle of unit speed q in time Δt, which shall be called the spatial lattice spacing δ. As in the DSMC, $\Delta t/\tau = j$ and $\Delta x/\lambda = l$ must both be somewhat smaller than unity. Furthermore,

$$\frac{\Delta x}{\delta} = \frac{\Delta x}{q\Delta t} = \frac{l}{j}\frac{\lambda}{q\tau} = \frac{l}{j}\frac{\overline{c'}}{q}, \tag{3.5}$$

where $\overline{c'} = \sqrt{8/3\pi}c_s$ is the average thermal speed. Then for $l \approx j$, if δ is small compared to Δx, q must be small compared to $\overline{c'}$. Thus, the velocity resolution improves as the cell contains more lattice sites. In practice, Δx and Δt are chosen on the basis of λ and τ in the region of highest expected density and temperature in the flow under consideration.[2] In a typical example, if $\delta = 0.1\lambda_0$, $\Delta x = 0.5\lambda_0$,

[2]When, as in the present work, the mesh is coarsened during the calculation in low density regions to keep l roughly constant, the right-hand side of Eq. (3.5) increases because l/j, not $\overline{c'}/q$, increases. Therefore, in this case, the velocity resolution does not increase.

$\Delta t = 0.2\tau_0$, so that $l = 0.5$ and $j = 0.2$, and, from (2.3), $\overline{c'} = 2q$, so $RT = \frac{\pi}{2}q^2$. On the other hand, if δ is halved (to $= 0.05\lambda_0$) and Δt increased so that $l = j = 0.5$, then $\overline{c'} = 10q$, and the initial temperature is 25 times higher. In any application, the dimensional value of q can be chosen to obtain the desired dimensional value of the reference temperature T, *e.g.,* 300K.

When a mixture of species is considered, the velocity resolution is found from the definition of the mean collision rate per molecule to be,

$$\left(\frac{\overline{c'}}{q}\right)^2 = \left(\frac{j}{\Delta t}\right)^2 {n_0}^2 \frac{4}{\sigma^2} \frac{n_0}{m_1 n_1 + m_2 n_2} \frac{1}{\left[{n_1}^2\sqrt{\frac{2}{m_1}} + 2n_1 n_2\sqrt{\frac{m_1+m_2}{m_1 m_2}} + {n_2}^2\sqrt{\frac{2}{m_2}}\right]^2}. \qquad (3.6)$$

3.2.1.2 Spatial Lattice

A further consequence of the discretization of the velocity components is that if particles are initially distributed in space on a regular array aligned with the axes of the co-ordinate system at points with spacing δ, then the particles remain on the array for all time, and the gas is a *lattice gas.* By positioning the particles on the lattice, the spatial resolution is coarsened to a level consistent with the velocity resolution, and the calculation of particle motion is simplified; particle translations during the motion phase are obtained simply by counting lattice sites. Because the particle locations are integer numbers, storage requirements are also reduced. Any number of particles can co-exist at a lattice site, in contrast to the CA method where the number of particles at a site is limited by an exclusion principle (see, for example, Frisch *et al.* 1986).

In the IDSMC, the coarse DSMC mesh of cells is superimposed on the fine lattice, and particles are drawn as candidates for collisions from all the lattice sites within a cell. This hierarchy is seen in figure 3.1. If there are many lattice sites per cell, the discretization of space onto the lattice is no longer significant. At a minimum, there must be at least one lattice site in a cell in order that there be

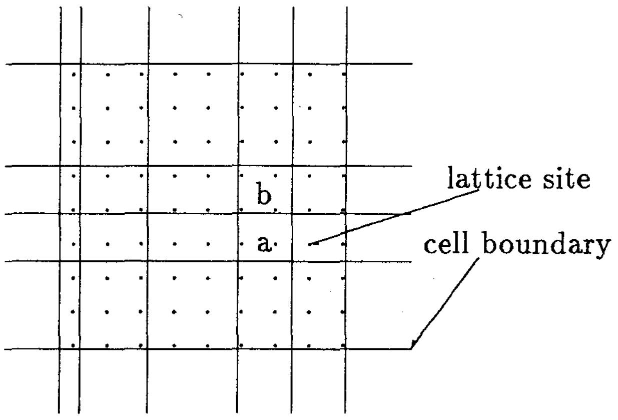

Figure 3.1: Lattice points and cell boundaries in physical space. Cells a and b are the same size but contain different numbers of lattice sites.

any particles there. In the IDSMC, as in the DSMC, a record is kept of the cell in which every particle resides, at the expense of additional storage. Because the cell boundaries may not have any simple relationship to the lattice spacing, two cells of the same size may contain a different number of lattice points (for example, cells a and b in figure 3.1). It is therefore necessary to take the cell volume to be proportional to the number of lattice sites in the cell in order that the time increment for each collision (equations 3.1 or 3.3) is properly computed.

It is worth noting that the integer-velocity method causes a minimum relative velocity, q, between two colliding particles. Thus, the amount the cell time counter can be incremented after a low-energy collision is limited by equations 3.1 or 3.3. In a continuous-velocity gas a particularly low-speed collision can yield large time increments that can, if not compensated for, effectively shut down a cell for long periods and make an unsteady calculation inaccurate because the cell cannot then adapt to changing flow conditions.

To conserve angular momentum it has been suggested (see Meiburg, 1986 and Bird, 1987) that particles be drawn for collisions from small sub-cells within each cell. The lattice sites in each cell of the IDSMC can be used in place of sub-cells; collisions between particles not residing at the same lattice site could be rejected so that angular momentum is exactly conserved in each collision. This procedure has not been implemented in the present study. Of course, if there is only one lattice site per cell there is no need to reject particle pairs. Further implications of restricting collision pairs to individual lattice sites are discussed in chapter 5.

3.2.1.3 Collision Outcomes

It was pointed out in section 2.2.1.2 that the number of possible collision outcomes can be large and the algorithm for finding all of them for a given relative velocity is complicated. Therefore, it is more efficient to do the calculation once and store the results in a look-up table for use during the Monte Carlo calculations. Table 3.1 gives the first few entries of such a look-up table. This table contains the coordinates of the integer points on spheres centered at (u, v, w) = (0,0,0) with different values of the radius squared. Before using the table to compute a collision between particles of the same mass, the velocities of the collision partners are transformed into the relative velocity frame, and the square of the relative speed is found. The outcome relative velocity of the same magnitude is chosen with uniform probability from the table under the condition that the parity of the components of the pre- and post-collision relative velocity vectors is the same. Then the final velocities are re-transformed into the lab frame. It is interesting to note that spheres corresponding to impossible collisions (*e.g.,* squared radius $= 7$) are not intersected at all by the lattice.

The look-up table is also used to find the points on the inner sub-grid sphere

Radius^2	Number of Points on Sphere	Coordinates of Points on Sphere
0	1	(0,0,0)
1	6	(-1,0,0) (0,-1,0) (0,0,-1) (0,0,1) (0,1,0) (1,0,0)
2	12	(-1,-1,0) (-1,0,-1) (-1,0,1) (-1,1,0) (0,-1,-1) (1,1,0) (0,-1,1) (0,1,-1) (0,1,1) (1,-1,0) (1,0,-1) (1,0,1)
3	8	(-1,-1,-1) (-1,-1,1) (-1,1,-1) (-1,1,1) (1,-1,-1) (1,-1,1) (1,1,-1) (1,1,1)
4	6	(-2,0,0) (0,-2,0) (0,0,-2) (0,0,2) (0,2,0) (2,0,0)
5	24	(-2,-1,0) (-2,0,-1) (-2,0,1) (-2,1,0) (-1,-2,0) (2,1,0) (-1,0,-2) (-1,0,2) (-1,2,0) (0,-2,-1) (0,-2,1) (2,0,1) (0,-1,-2) (0,-1,2) (0,1,-2) (0,1,2) (0,2,-1) (2,0,-1) (0,2,1) (1,-2,0) (1,0,-2) (1,0,2) (1,2,0) (2,-1,0)
6	24	(-2,-1,-1) (-2,-1,1) (-2,1,-1) (-2,1,1) (-1,-2,-1) (-1,-2,1) (-1,-1,-2) (-1,-1,2) (-1,1,-2) (2,1,1) (-1,1,2) (-1,2,-1) (-1,2,1) (1,-2,-1) (2,1,-1) (1,-2,1) (1,-1,-2) (1,-1,2) (1,1,-2) (2,-1,1) (1,1,2) (1,2,-1) (1,2,1) (2,-1,-1)
8	12	(-2,-2,0) (-2,0,-2) (-2,0,2) (-2,2,0) (0,-2,-2) (2,2,0) (0,-2,2) (0,2,-2) (0,2,2) (2,-2,0) (2,0,-2) (2,0,2)

Table 3.1 Sample from the top of the look-up table

in a multi-species collision. As described in section 2.2.2.1, outcomes corresponding to points not lying on the main velocity grid are rejected. If the mass ratio is large, most points on the sub-grid sphere will not lie on on the main grid, and many trials may be rejected. This is expensive. The following simple algorithm is used to minimize that cost: a candidate point (V_x, V_y, V_z) is randomly chosen on the sub-grid sphere from the look-up table, and its X component is examined. If the X component lies on the main grid (*i.e.*, $(V_{x_{rel}} - V_x)$ modulus[sum of particle masses] $= 0$), it is acceptable, and the Y and Z components are tested in turn. If the X component did not lie on the main grid, its sign is changed (it is reflected through the $X = 0$ plane), and it is retested. If the X component is now acceptable, the procedure is continued with the Y and Z components being tested in the same way. If V_x is still unacceptable, the candidate point is rejected, and a new point is selected from the table. Even if the mass ratio is as high as 10, this algorithm runs only slightly slower than the single-species algorithm. If separate look-up tables were used for each mass ratio, this rejection process could be avoided, but the memory consumed by the tables might be prohibitive. With the rejection method, a single look-up table can be used for all mass ratios.

The table presently used has entries for spheres up to a radius of 38, a total of 30,415 values. Each value requires three bytes for storage (one for each velocity component) and two bytes are needed for the square of the relative speed that is used for the reference index; hence, the table would consume about 94 kilobytes. Because memory requirements of this magnitude are a matter of concern for users of many present-day parallel-computing machines in which the processors do not share memory, only the entries for the octant $(u, v, w) \geq 0$ of each sphere are stored. The selected outcome configuration of a collision is reflected across the planes $u = 0$, $v = 0$, and $w = 0$, each with 50 percent probability. This procedure reduces the size of the look-up table to 1/8 of that needed for the full sphere, at the expense of a small increase in computational time. Because a look-up table is used, the cost of computing collisions does not vary with the velocity resolution;[3] higher resolution simulations simply require a larger look-up table.

3.2.1.4 Boundary Conditions

Flow boundaries, which for the calculations presented here are straight and parallel to lattice axes, are taken to lie midway between lattice points so that particles may not reside exactly on a boundary. Boundaries that pass through lattice points, however, are also possible. Specular wall collisions are treated in the same way as in the conventional DSMC, by reflecting the particle trajectory across all wall segments necessary to insure that the particle at the end of the time step is inside the flow field, and by reversing its normal velocity after each reflection. For diffuse wall collisions, the velocity components of the emitted particles are chosen from the equilibrium integer distribution (*cf.* fig. 4.4) corresponding to the wall temperature. The trajectories of colliding particles approaching and departing from walls are calculated exactly, but, after the collision, the particle is placed at the nearest lattice site.

3.3 Implementation for Parallel Processing

The integer direct simulation method is well suited for implementation on parallel processors because even two-dimensional flow simulations require substantial computational effort. For example, in the problem discussed in chapter 5, section 4,

[3]For multi-species collisions with the single look-up table, the collision cost varies with the probability of rejection of unacceptable points. That probability depends somewhat on the velocity resolution.

simulation of 3200 time steps, with four million particles, and 100,000 computational cells is required. Three different parallel computers available at the Caltech Computer Science department are suitable for such calculations. They are a 128-processor Intel iPSC/1, a 16-processor iPSC/2 or a 192-processor Symult 2010. Each processor or node in the iPSC/1 contains an independent microcomputer with its own central processing unit (Intel 80286/80287), communications control, and 512 kbyte of local memory. Each node is connected directly to seven others with a hypercube topology and to an external system manager. Together, the nodes are referred to as the "cube." The cube manager is an Intel system 310 supermicrocomputer that, in turn, is connected via an Ethernet to a SUN workstation. The SUN acts as the "host" and runs the program that controls the cube. The nodes communicate by messages sent and received under program control. Although the iPSC/2 has fewer nodes, each node has four megabytes of memory and uses an Intel 80386/80387 central processing unit that is about eight times faster than that in the iPSC/1. The inter-node connections are in the form of a four-dimensional hypercube and each node is also connected to a cube manager. The nodes in the Symult also contain three or four megabytes of memory and use Motorola 68020/68881 central processing units. Thus, the iPSC/2 and Symult nodes are nearly equivalent. The nodes in Symult are connected with a mesh network although, for the present simulations, the message passing time is so small that the machine is used as though the connections were in the form of a hypercube. The iPSC/1 was used for most of the parallel computations although the Symult was used for the largest simulations.

In the simulations, the rectangular physical space of the flow field is divided into rectangular sub-domains, each being allocated to a single node (figure 3.2). The sub-domains contain equal numbers of computational cells each of which, in turn, contains about the same number of particles. When during the calculation a particle moves from the domain of one node to that of another, it is sent as a message. If the node domains are sufficiently large, particles travel only to nearest neighbor nodes in one time step, and message path lengths are minimized when a Grey-code.[4] is used to map the physical domain onto the hypercube topology.

It should be noted that other solutions have been proposed for creating a parallel version of the DSMC. In one case (Furlani, 1989), the background calculation is done on the host but the particles are passed to available processors for the time-consuming move and collide phases of the computation. This method has the advantage that the processors share the work evenly but the severe disadvantage that, for larger simulations, the host quickly becomes overwhelmed as a manager, and the efficiency suffers. Another possibility is called scattered decomposition or random decomposition (Fox *et al.*, 1988). Cells of particles are assigned randomly to processors with the assumption that, while some cells require much computation, some require little and, on average, each processor has about the same workload. This scheme could be inefficient because of the heavy message traffic involved except in machines for which message passing is truly an insignificant expense. Also, it would be necessary for each processor to determine exactly where to send each particle; hence, the position of each cell in the flow must be known by all processors and the computational mesh may consume much memory. A third, perhaps obvious, possibility becomes reasonable if each processor has several megabytes of memory. In this case an ordinary sequential program that computes the entire flow field at once can be run in each processor and the final output ensemble averaged. The advantage would be that a sequential code could be run with minimal alterations and with near perfect load balancing. The disadvantage is that the size of the problems that could be simulated is limited.

[4]See Appendix A.2.

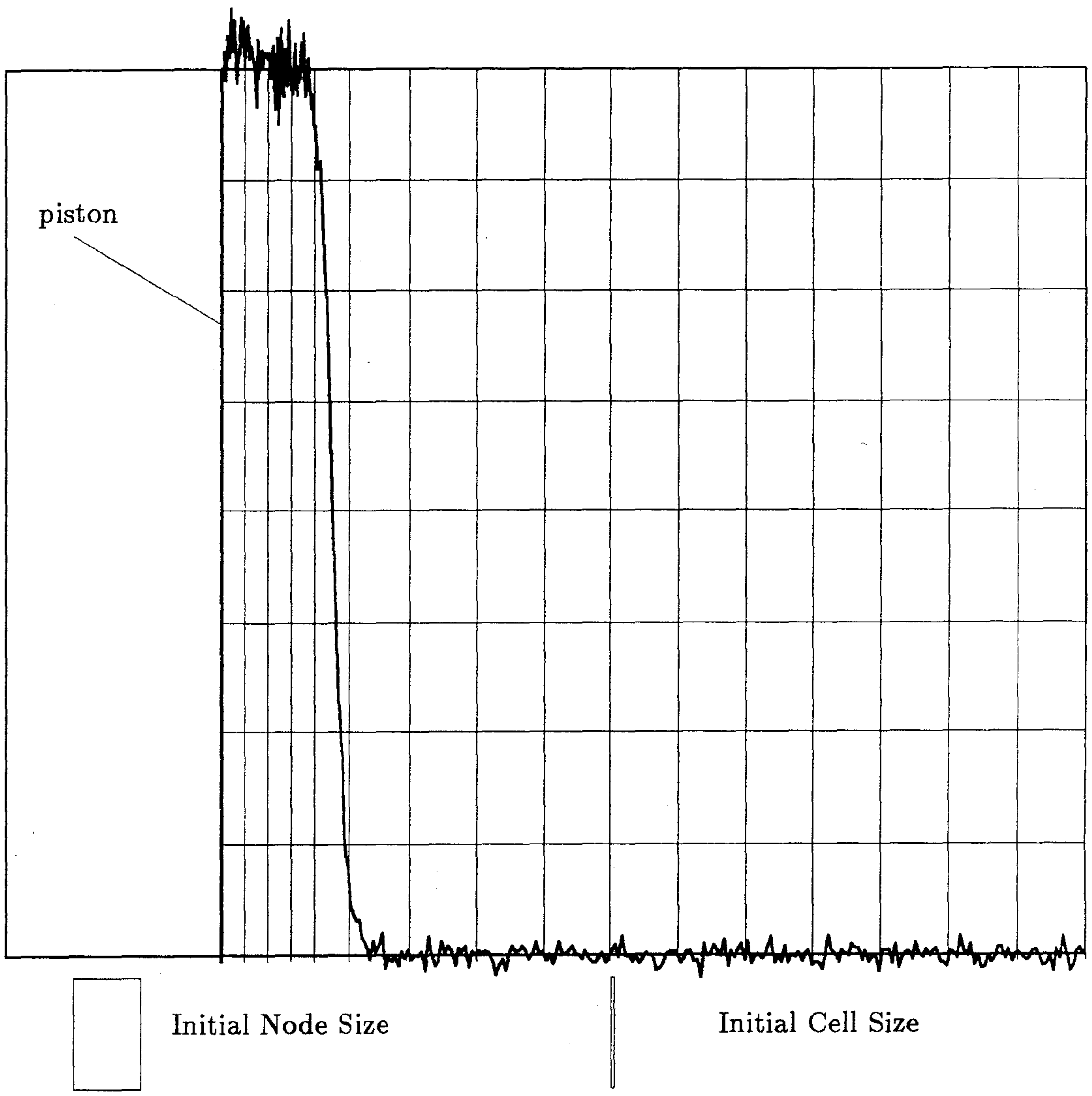

Figure 3.2: Density profile superimposed on node mesh for a typical piston problem. Left-hand wall is impulsively accelerated as a piston into the flow at $t = 0$ creating an unsteady shock wave. Note how the node (and, thus, the cell) size decreases through the shock.

With the present division of the flow field, and because the flows of interest are unsteady, each region of the flow must be computed simultaneously. When a particle is sent between processors, it must arrive at the correct time. Hence, the processors proceed with lock-step synchronization; each processor checks that its neighbors are at the same point in the calculation before it proceeds with the next time step. Such regimented coordination is not inefficient if an adaptive mesh of cells maintains about the same average workload in each processor.

3.3.1 Adaptive Mesh

In all parallel computations reported in this work, an adaptive grid of nodes was used. The grid is remeshed in the vertical and horizontal directions to maintain the average number of particles in the rows and columns of nodes constant. Remeshing has four significant advantages: (i) it balances the work load among the nodes; (ii) it maintains the number of particles per cell and the cell size, measured in local mean free paths, approximately constant; (iii) it compensates for rapidly changing boundary geometry; and (iv) it balances the number of particles in each node, so that the total number of particles used can be maximized. In flows with strong temperature differences, the high temperature areas of the flow have fast moving particles and a high collision rate because the time increment per collision is small (eqns. 3.1 and 3.3). Those regions require the calculation of many more collisions than others, and remeshing based solely on particle number density will lead to some nodes being underutilized. Of course, the remeshing scheme can be easily altered to balance other quantities between processors, such as mean pressure or even computational time, but the features (ii), (iii), and (iv) might then be sacrificed.

For simulations without rectangular symmetry, for example flow about a body in the center of the field, a body-fitted grid would probably be more efficient than a rectangular grid (Bird, 1986). A regular rectangular grid (in which the columns and rows of the mesh remain aligned) does not necessarily place the computing effort where it is most needed. However, the programming time to generate the rectangular mesh is less than that required to create a more complicated mesh and the adaptive mesh permits a useful exchange of some computational time for a gain in programmer efficiency. In the future, the method used for remeshing could be generalized and applied to meshes of other shapes.

To remesh the whole flow field at once either requires clever coordination to determine the node boundaries locally, or a single processor (node zero) to compute the entire mesh on its own. The latter procedure is chosen in the present work because it is simpler and is directly applicable to a sequential version of the code. Nonetheless, a substantial portion of the remeshing computational load is performed in parallel. Each node in the bottom row of the mesh passes its particle number density distribution to the immediate neighbor above. The receiving node adds in its own particle number density distribution and passes the sum further up the column of nodes. The top row of nodes similarly accumulate the data as they pass the distribution to the left to node zero. With the accumulated distribution, node zero creates the new mesh and passes it back to the other nodes. Periodically, just before the complete flow field data is to be sent to the host, the host creates and stores a mesh so that the ensemble averaged data in the host always refers to the correct mesh.

The number of particles required per column of nodes is found simply by summing the entries of the particle number density profile and dividing by the number of nodes per column (see figure 3.2). The mesh generating algorithm in node zero begins at the left side of the flow field and establishes the left boundary of the first column of nodes. It then sums the values of the particle number density profile in the X direction into a counter. When the counter exceeds the desired number of particles in the first column of nodes, the new right boundary for that column is found by interpolating between the two nearest X locations in the profile and then the counter is reset. The process subsequently continues from that point; particles are counted and a node boundary placed whenever a sufficient number of particles are found to fill a column of nodes. The procedure is analogous in the vertical direction.

The final mesh then simply consists of two small vector variables containing the left, right, top, and bottom locations of each column and row of mesh nodes. Each cell's location need not be known by all nodes. When a node receives a new mesh, it re-sorts its particles into its own new grid of cells and sends off those particles that belong to other nodes.

Within each node the sorting of particles into cells is simplified if all the cells are of the same size and if they lie in a regular array (a constant number of rows and columns). The disadvantage of this arrangement is that the cells do not change size within a node to compensate for significant density gradients that can occur there. In principle, the cells within each node could be remeshed to better adjust the cell dimensions to the local mean free path. Such a fine scale remeshing, however, would require the exchange of more data with the host and would consume more time.

Chapter 4

Simulations of a Uniform Gas

Discrepancies between results obtained with the discrete-velocity and continuous-velocity methods may be due either to errors in discrete-velocity particle convection or collisions, that is, an improper discrete representation of either the left or the right-hand side of the Boltzmann equation. Investigation of collisions and convection separately yields physical insight into the implications of the discrete-velocity approximation and may point to procedures that overcome the errors.

In this chapter are presented the calculations of three test problems that exhibit features of the discrete-velocity model in a spatially uniform gas. Only the collision process is considered. In the first example, the equilibrium state of an integer velocity gas is determined. The process of the relaxation of a single-species gas to equilibrium is examined in the second case and relaxation of a binary gas mixture is considered in the third. A discussion of the effects of discretization on the convection of particles is given in chapter 5.

4.1 Equilibrium State of a Discrete Velocity Gas

It is clear from figure 2.9 that for collisions with large relative velocities, typical of gases having moderate velocity resolution, a great number of dynamic interactions is possible. On the other hand, at relatively low temperatures, where fewer velocities are common, the number of possible outcomes is reduced. An important question is whether the discrete representation is qualitatively similar to the continuous representation or whether serious anomalies arise. Figures 4.1 and 4.2 are histogram distribution functions of the rotation angle of the relative velocity vector in an equilibrium three-dimensional IDSMC gas at two different temperatures. The figures are based on data obtained by allowing 32,000 identical particles in a box to collide 64,000 times, a technique similar to the one originally used by Bird (1963). The box contains just one computational cell, and the calculations are performed on a small sequential computer. There is no need for a time counter. The solid lines are theoretical sine distributions appropriate for a hard-sphere continuous-velocity gas and are normalized to include the same number of collisions.

In the low temperature case (fig. 4.1), certain discrete collision angles χ (*i.e.*, 0, 90^o, *etc.*,) occur often, and their resulting distribution does not resemble that of a continuous-velocity gas. To a certain extent, the frequent occurrence of particular angles compensates for the absence of neighboring values. There are, however, more occurrences of ineffective or null deflections ($\chi = 0^o$ and 180^o) than would account for the lack of small non-zero deflections.[1] In low speed collisions there simply are few reflection symmetries and there is a high probability of degenerate collisions. The probability of null collisions occurring in an equilibrium single species gas decreases with increasing velocity resolution (figure 4.3) because the average size of the collision spheres increases (*i.e.*, the mean relative collision energy $= 2kT$). Null collisions effectively distort the time and collision counters because the counters are incremented although nothing is changed. Detailed tailoring of the distribution of collision angles with an acceptance/rejection method was tried but was found to slow

[1] When the particles are identical, a 180^o deflection is also a null collision.

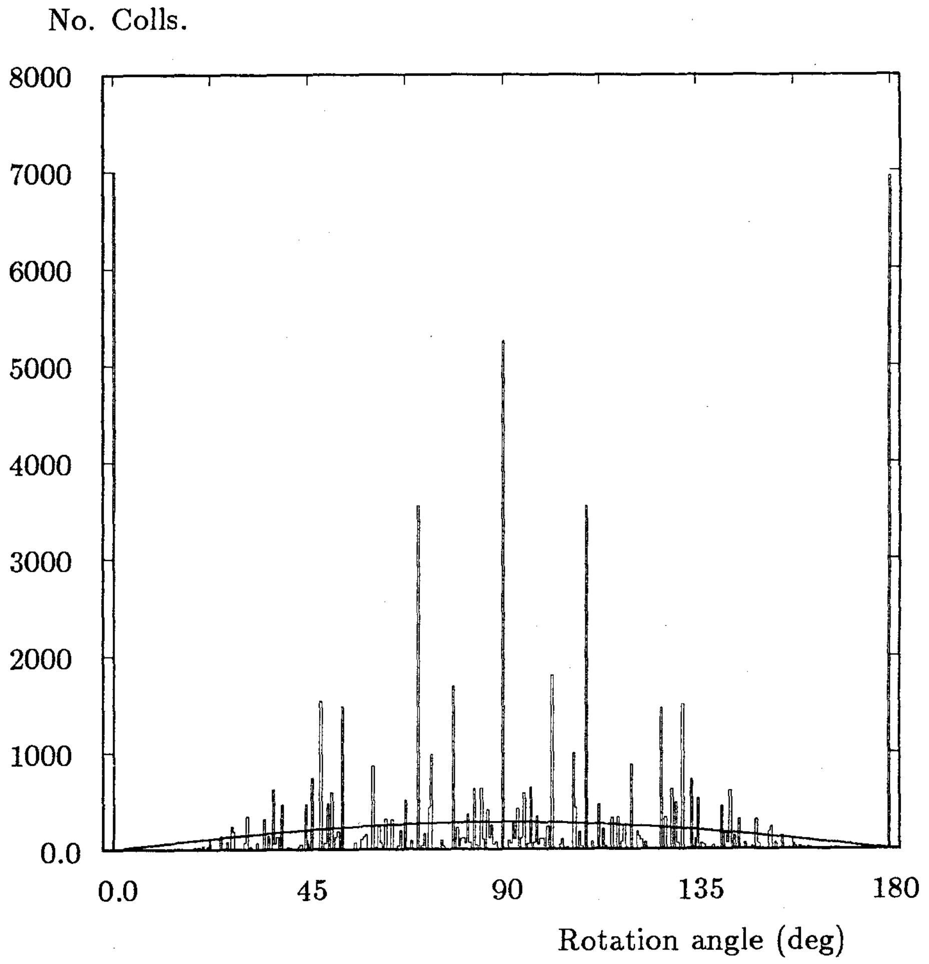

Figure 4.1: Histogram of the angle of rotation, χ, of the relative velocity vector in each collision. The solid line is the theoretical distribution for 64,000 collisions. Particle temperature is $RT = \frac{\pi}{2}q^2$. Bin width = 0.5^o.

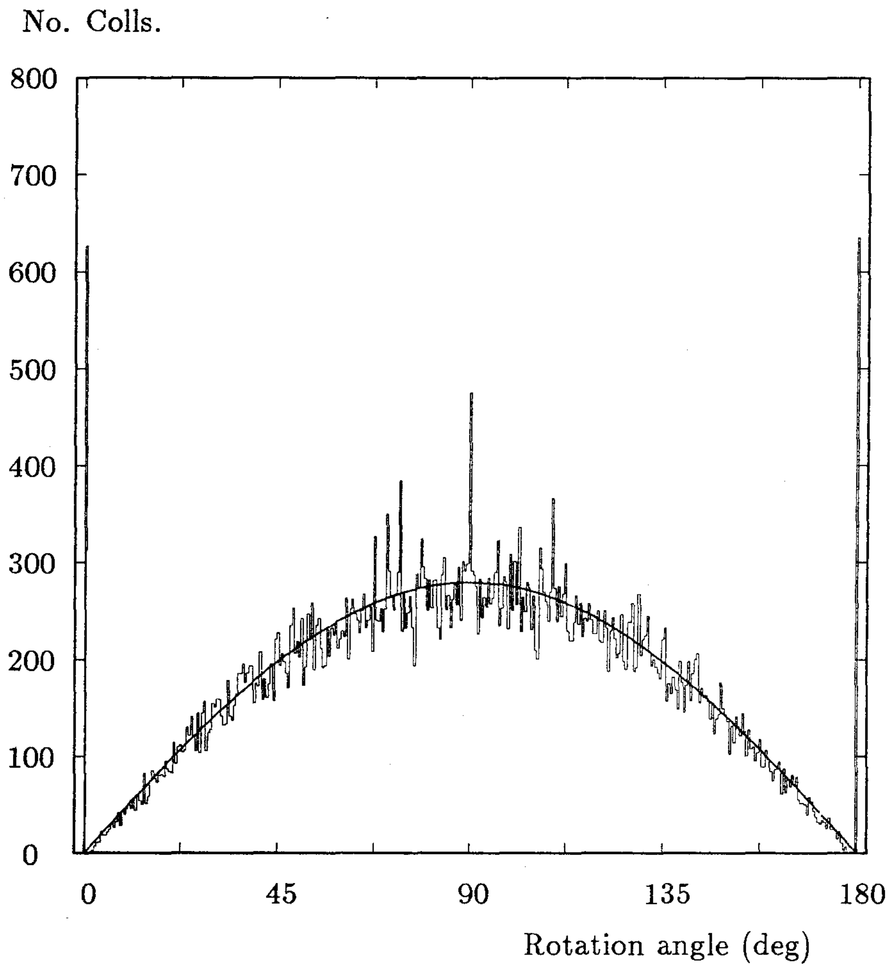

Figure 4.2: Histogram of the angle of rotation where the particle temperature is $RT = 201.3q^2$. Bin width $= 0.5^o$.

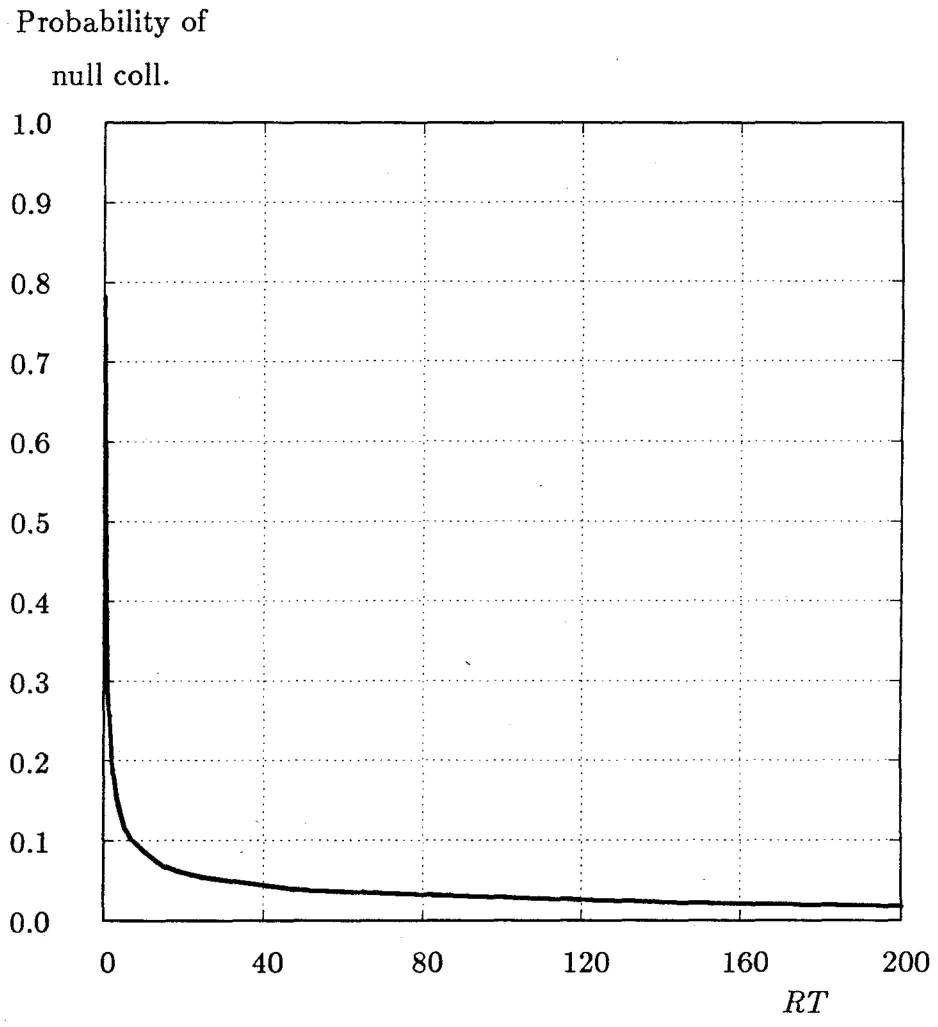

Figure 4.3: Probability of a null (0^o or 180^o) collision occurring in a single species equilibrium gas at different temperatures.

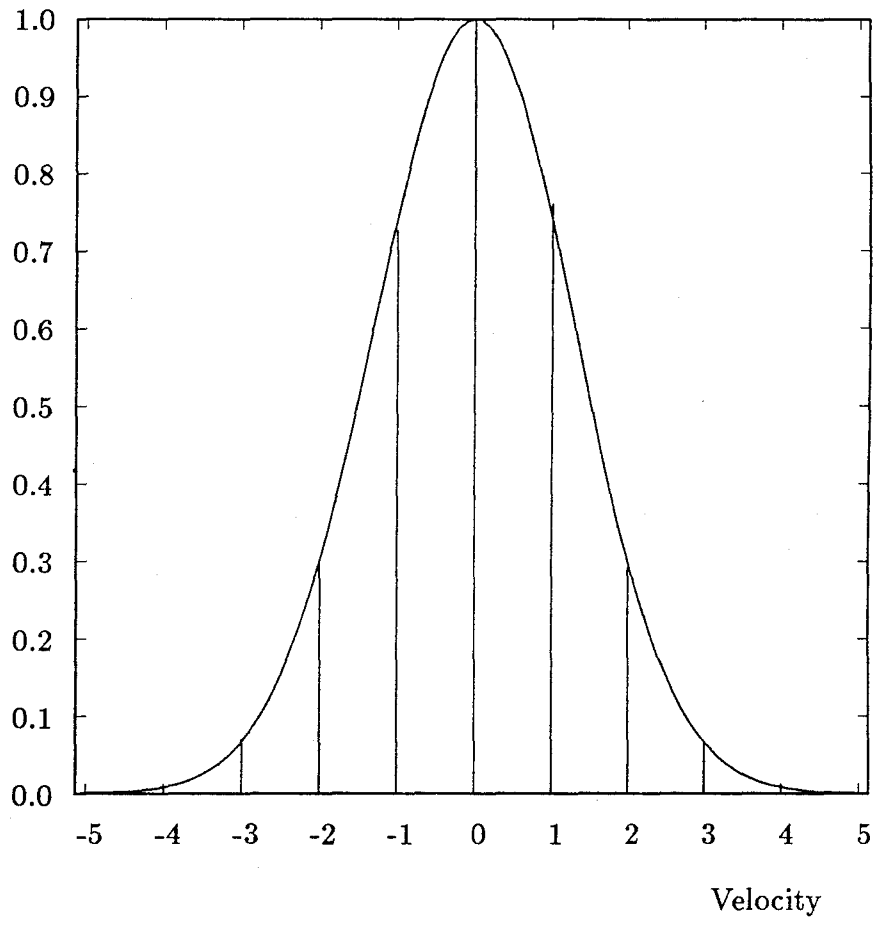

Figure 4.4: Velocity component distribution function normalized to a maximum value of 1. The solid line is the theoretical distribution normalized to cover the same area as the discrete distribution. The particle temperature is $RT = \frac{\pi}{2}q^2$.

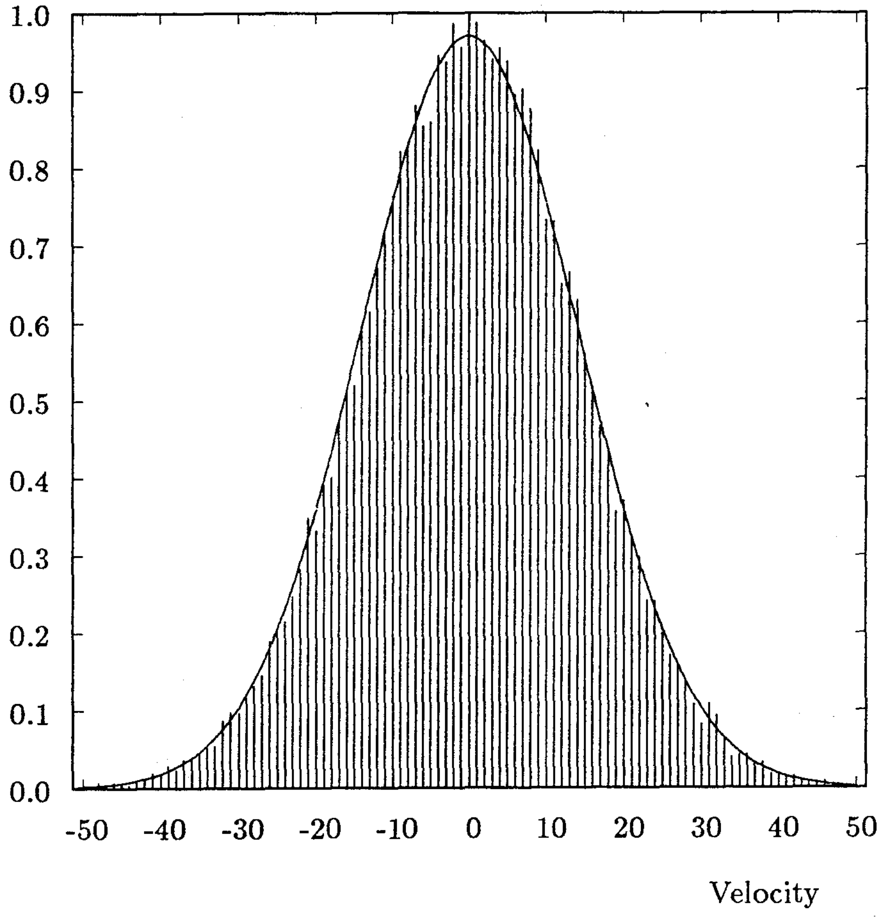

Figure 4.5: Velocity component distribution function for $RT = 201.3q^2$. Note the change in scale from figure 4.4

the simulation substantially. Another possibility, discussed later with respect to multi-species collisions, is to consider a portion of the null collisions to be ineffective and simply not increment the collision counter when they occur.

In the high temperature example (fig. 4.2), the continuous distribution is well modeled except, again, for an excessive number of null collisions. It is not clear *a priori* whether the random choice of outcome intersections of the relative velocity sphere with the lattice produces an adequate distribution of collision angles, because on any given sphere the lattice points are not distributed uniformly. The correspondence found between IDSMC scattering angles and the theoretical sine distribution at high temperatures applies only to the mean gas, not to collisions of a specific relative energy. As will be seen in chapter 5, at low temperatures (*i.e.*, low velocity resolution), the excess of null collisions has the effect of artificially increasing the diffusivity of the gas, while at moderate to high temperatures, the effect on macroscopic quantities is not noticeable.

The velocity distribution function itself provides a more direct means for evaluating the effects of velocity discretization. The IDSMC method produces the correct equilibrium solution to both the discrete Boltzmann equation at very low velocity resolution and to the continuous Boltzmann equation at higher resolution. The sampled distribution of discrete U velocity components at high and low temperatures in an equilibrium gas, figures 4.4 and 4.5, is seen to be in agreement with the theoretical Maxwellian velocity distribution function for a continuous-velocity gas. As few as nine discrete-velocity components are enough to reproduce the Maxwellian distribution, $e^{-\beta^2 u^2}$ ($\beta = 1/\sqrt{2RT}$), with a high degree of accuracy. In the lowest temperature state (figures 4.6 and 4.7), the simulated distribution function is instead compared to a numerical solution for the equilibrium state of a discrete-velocity gas (Nadiga *et al.*, 1989). In both a stationary gas and translating

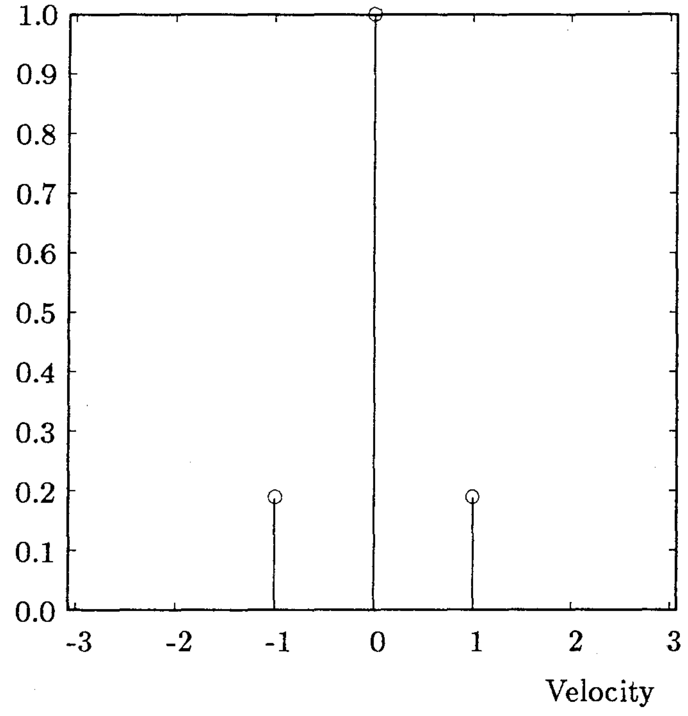

Figure 4.6: Velocity component distribution function normalized to a maximum value of 1. The bar graph is the IDSMC solution and the open circles are the solutions to the discrete Boltzmann equation of Nadiga at $RT = 0.2738q^2$, $\overline{U} = 0$.

gas, the simulated distributions are the same as Nadiga's solution, except that in the Monte Carlo calculations there is some inherent noise.

By varying different portions of the collision process, particularly the symmetry of the points on the collision spheres, it was found that equilibrium distributions are insensitive to the correctness of the process. A more appropriate measure is the relaxation of the velocity distribution function in a highly non-equilibrium gas.

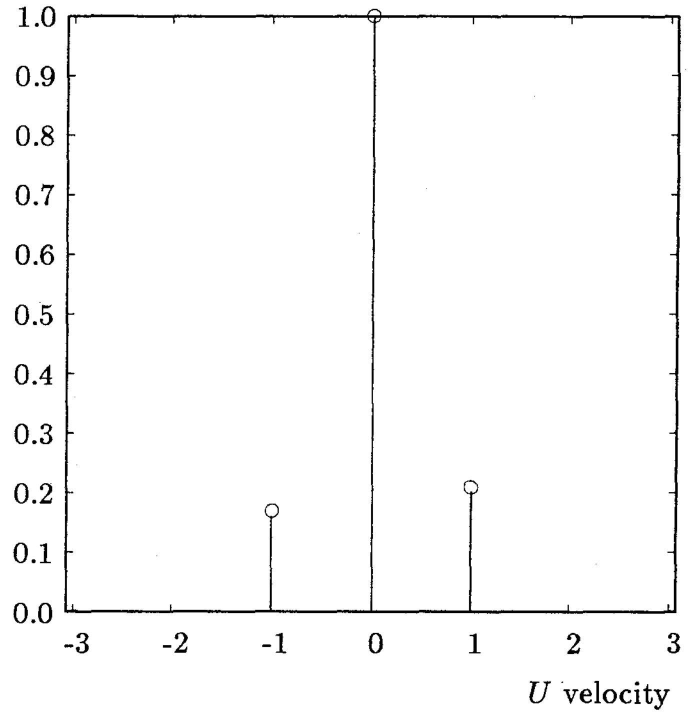

Figure 4.7: U velocity component distribution function for a translating gas. The bar graph is the IDSMC solution and the open circles are the solutions to the discrete Boltzmann equation of Nadiga at $RT = 0.2743q^2$, $\overline{U} = 0.0306$.

4.2 Relaxation to Equilibrium

To observe the relaxation of a non-equilibrium gas, 90,000 particles are simulated in a box consisting of one computational cell. The particles are initially given integer-valued velocity components and are distributed bimodally as indicated in the top rows of figures 4.8 and 4.9. The x-component molecular velocities are distributed in two narrow bands (each with $RT \approx 10q^2$) about $u = \pm 25q$, while the y- and z-component velocities have only one peak of the same width. As for the equilibrium states discussed earlier, the fluid is uniform in these calculations, and all particles in the box may collide with a probability proportional to the relative speed. Only the collisions and velocities are calculated. The particles have no specific position so their movements are not represented. In figure 4.8, the distributions of the DSMC calculation are plotted as histograms with bin size q, while in figure 4.9 the spikes represent the accumulated data at the corresponding discrete values of velocity. Although in the DSMC calculation the initial distribution contains only integer-valued components, after the first collision, the velocities become decimal numbers.

The molecular velocities are sampled after 1 (second row), 2 (third row), and 10 collisions per particle (fourth row). It can be seen that in both of the calculations of figures 4.8 and 4.9, the initial bimodal shape evolves into a Maxwellian distribution with temperature $RT = 218q^2$ in all three directions (indicated by the solid curves in the bottom row), and that the IDSMC results are essentially the same as the DSMC results even in the non-equilibrium state. Using the Bhatanagar, Gross, & Krook (1954) model of the Boltzmann equation for a discrete-velocity gas, Broadwell (1963) showed that the equilibrium distribution has the form of a

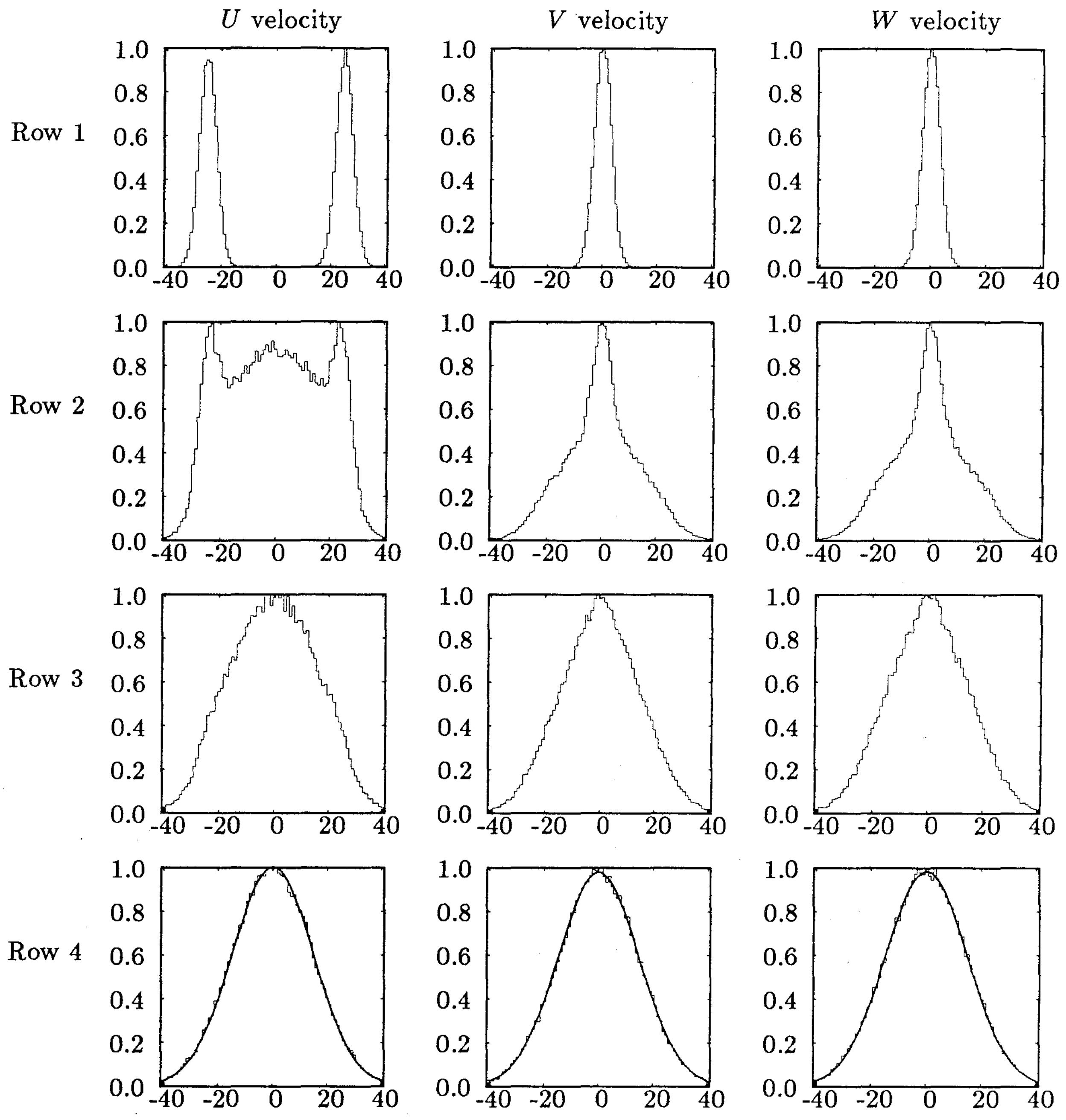

Figure 4.8: DSMC method. Development of velocity distribution function histograms. Initial bimodal distribution spreads to form a Maxwellian. Equilibrium Maxwellians are also drawn in the last row. First row is time zero, second after 1 coll./particle, the third after 2 coll./particle, the last after 10 coll./particle.

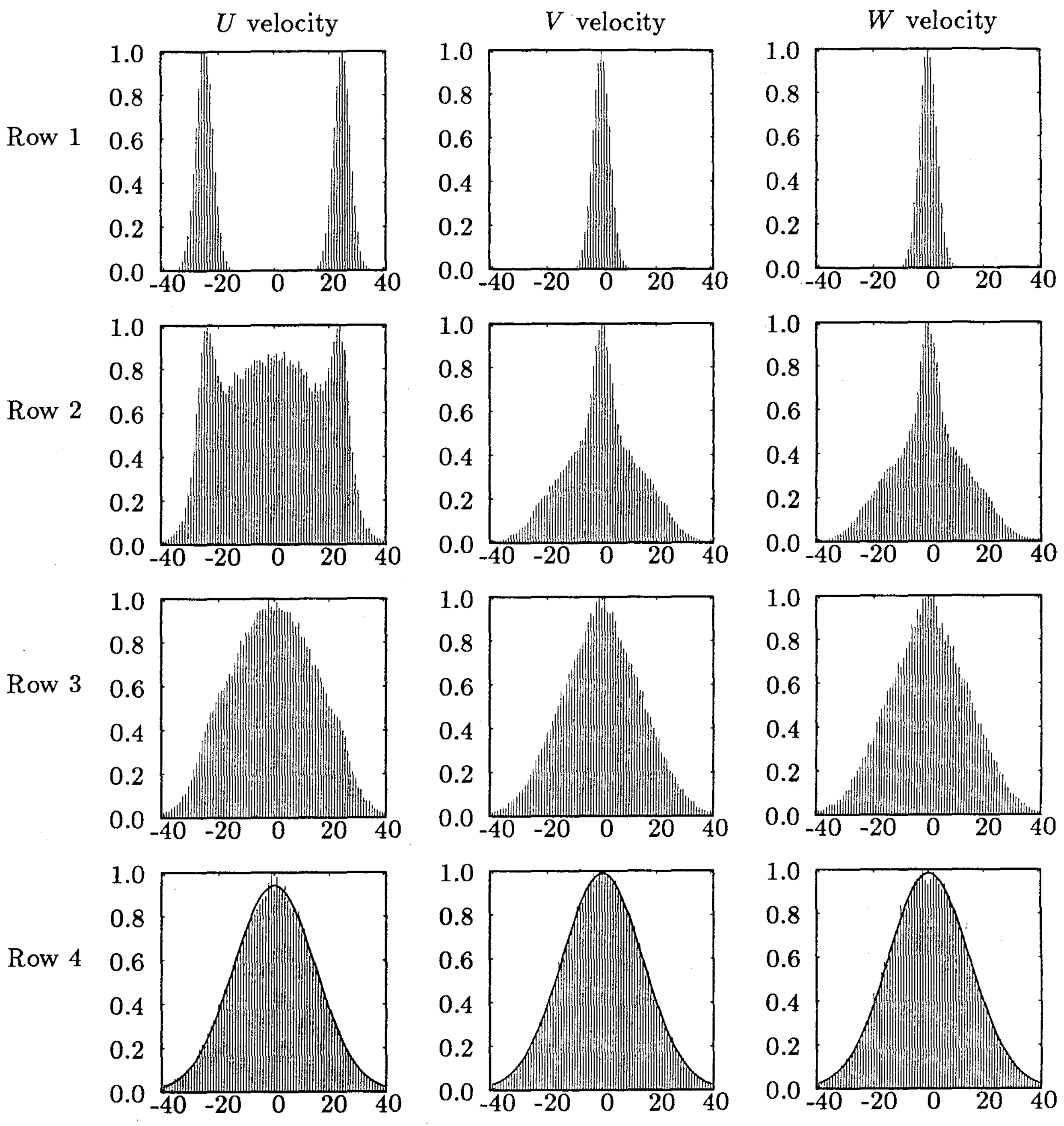

Figure 4.9: IDSMC method. Development of velocity distribution functions. Initial bimodal distribution spreads to form a Maxwellian. Equilibrium Maxwellians are also drawn in the last row. First row is time zero, second after 1 coll./particle, the third after 2 coll./particle, the last after 10 coll./particle.

Maxwellian, *i.e.*, for a two-dimensional stationary gas,

$$n_i \quad \propto \quad \exp[-A({u_i}^2 + {v_i}^2)], \tag{4.1}$$

where the subscript $()_i$ refers to the class of particles with discrete-velocity components u_i and v_i. In the high temperature limit, A becomes $(2RT)^{-1}$. Thus, the distributions shown in the bottom row of figure 4.9 are expected.

If the velocity resolution of the gas is too low, however, there are few points on the discrete-velocity collision spheres, many collisions are null, and the gas relaxes too slowly. This incorrect relaxation rate is due both to the velocity discretization and to the present collision model which incorporates a random choice of points on the collision spheres. The result can be seen in the combined DSMC and IDSMC results of figure 4.10. The individual modes of the initial bimodal distribution have a low temperature of $RT = 0.5q^2$ and are separated by only $4q$ at $u = \pm 2q$. The DSMC particles are assigned floating point velocities and are sorted into bins of size $\frac{1}{10}q$ in the histograms. One hundred forty thousand particles are used. The W component is not shown because it is the same as V. In the discrete solution it is seen that after one collision per particle there are still too many particles with $u = \pm 2$, not enough with $u = \pm 1$, and the V distribution is not sufficiently broad. Both methods, however, eventually produce the same equilibrium state after seven collision times (third row).

4.3 Multi-Species Relaxation

A similar relaxation of a non-equilibrium velocity distribution function occurs in a two-species gas. Consider a spatially uniform gas in which one group of particles moves with $u = +10$ and has a mass of 3, while another group moves with $u = -10$ and has a mass of 1. Thirty thousand particles are used, 15,000 of each species.

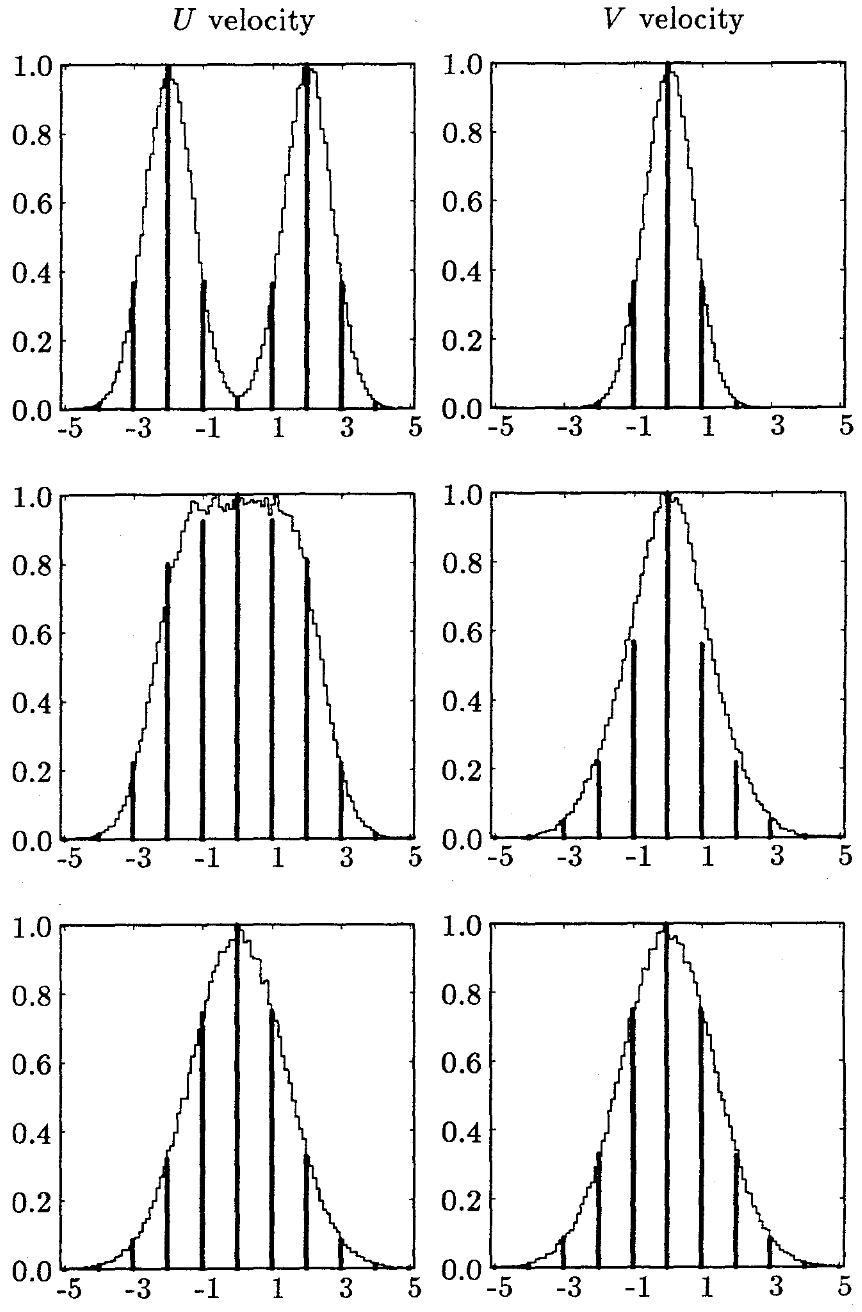

Figure 4.10: DSMC and IDSMC methods. Development of velocity distribution functions after 0 (first row), 1 coll./particle (second row) and 7 coll./particle (third row). DSMC = histogram (bin width $\frac{1}{10}q$), IDSMC = bar. $RT = 1.82q^2$.

At time zero the molecules of each species are in equilibrium with molecules of the same species but not the other; the distribution function of the light particles is considerably wider than that of the heavy particles because the width varies as $\sqrt{kT/m}$. The mass 1 particles have a temperature of $kT/m = 15q^2$ while the mass 3 particles have $kT/m = 5q^2$. The overall gas temperature is $RT = 65q^2$. In figure 4.11, the results of the DSMC relaxation processes in the U and V directions are shown as histogram distributions with a bin width of $1q$. Figure 4.12 shows the results of the same simulation carried out with the IDSMC method.

It can be seen that the initially separate groups merge together to form two overlapping peaks of greater width. Because energy is exactly conserved in each collision, the overall gas temperature remains constant, while the temperature of the individual species' increases as seen by the greater width of each band. Because most of the momentum is carried by the heavy particles, the light particle U distribution function moves toward the heavy particle distribution function. The V distribution function simply broadens with time. The W distribution is the same as V and is not shown.

The IDSMC reproduces the DSMC results well although slight differences can be seen after one collision per particle; the heavy particle distribution function of the IDSMC model has not moved quite as far to the left as in the DSMC model. This discrepancy is caused by the frequent occurrence of null collisions during low-speed inter-species encounters. Poor inter-species coupling is a manifestation of the same influence of velocity resolution as seen in the single species gas except that, with a mass ratio of three, the size of the collision sphere is effectively reduced to one third of that for equal mass particles. This effect is greatly exaggerated if the mass ratio is higher. In figures 4.13 and 4.14 a second comparison is made but here the mass ratio is 10 and the overall temperature is slightly higher ($RT = 76q^2$). Only the

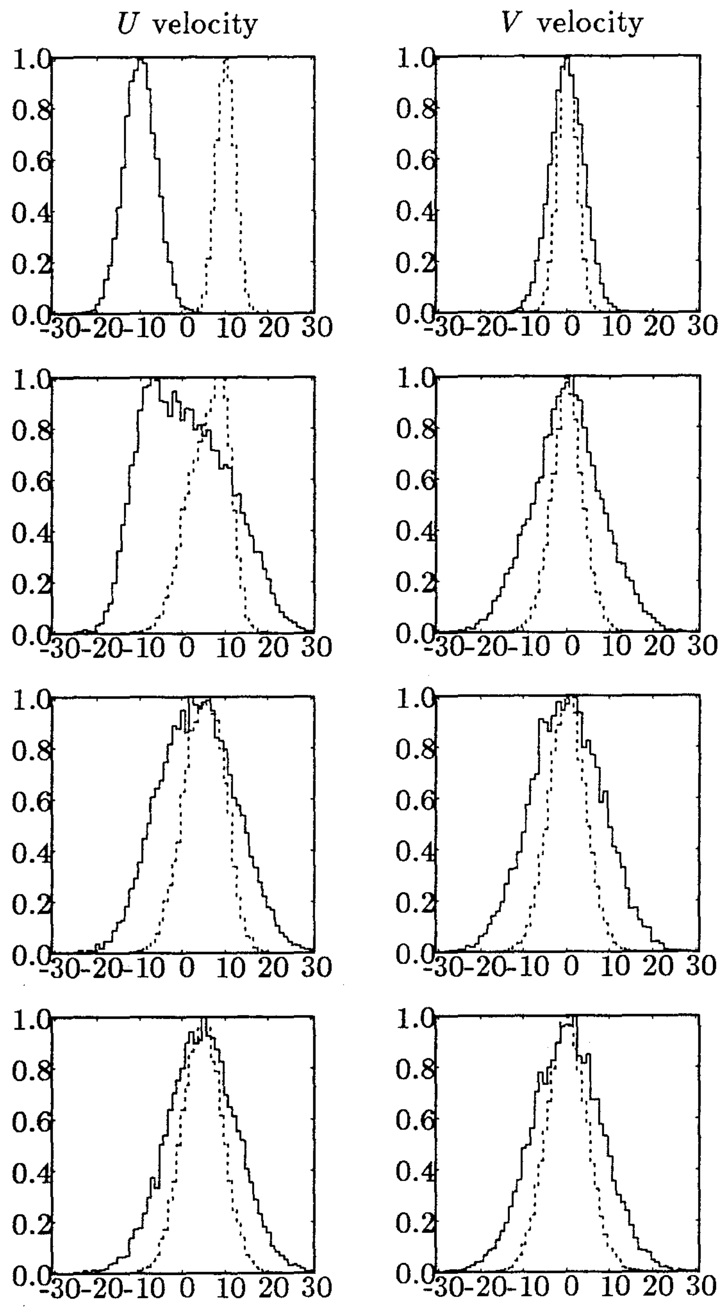

Figure 4.11: DSMC method. Development of velocity distribution functions in a two species gas. Bimodal distribution (mass 1 = solid lines, mass 3 = dotted) spreads to form a Maxwellian. $RT_{total} = 65q^2$. First row is time zero, the second after 1 coll./particle, the third after 2 coll./particle, the last after 10 coll./particle.

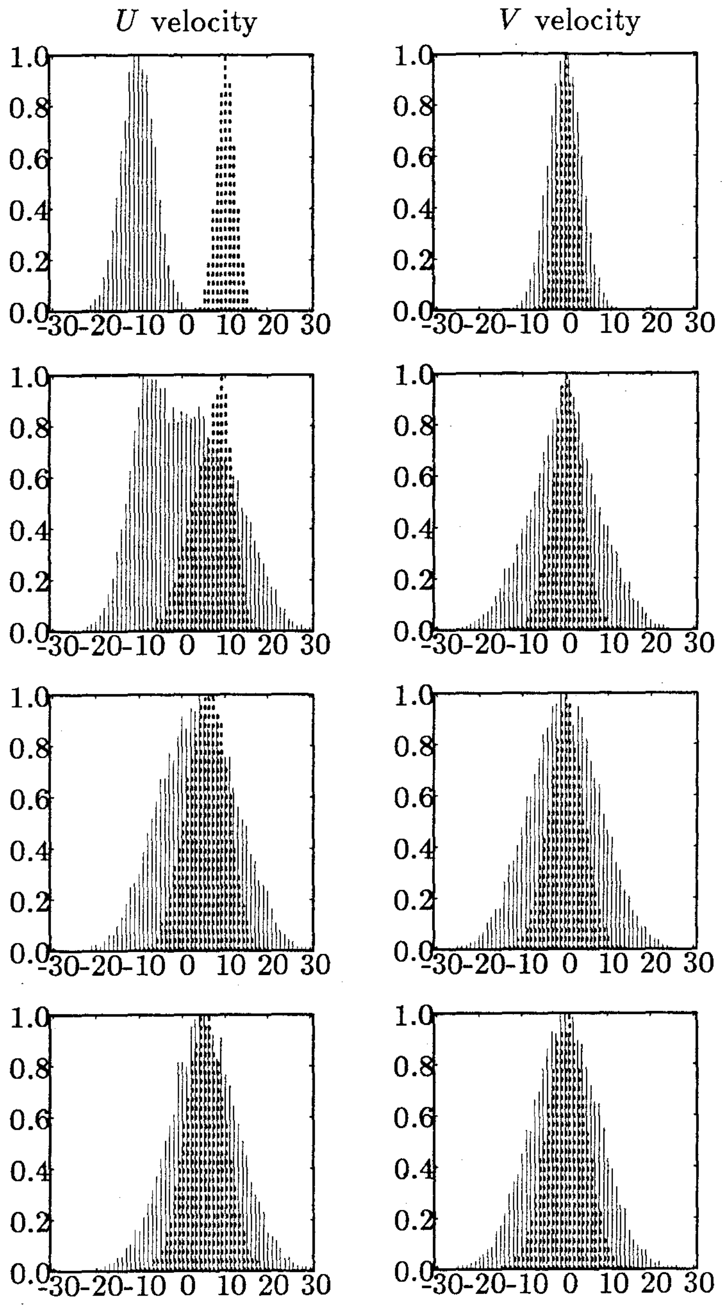

Figure 4.12: IDSMC method. Development of velocity distribution functions in a two species gas. Bimodal distribution (mass 1 = solid lines, mass 3 = dotted) spreads to form a Maxwellian. $RT_{total} = 65q^2$. First row is time zero, the second after 1 coll./particle, the third after 2 coll./particle, the last after 10 coll./particle.

first collision time is shown. With such a high mass ratio, there are so few points on the heavy particle collision spheres that most of the inter-species encounters result in null collisions, and the integer distribution function obviously relaxes too slowly.

Figure 4.15 shows the number of possible outcomes from a collision between a mass 1 and a mass 10 particle versus the square of the relative collision speed. The average, rather than the total, number of outcomes is plotted because the number depends on the location of the collision spheres on the velocity grid and not simply on their size. On each sphere there is, at a minimum, one pair of points — the original pair. For a mass ratio of 10, the minimum collision speed for there to be more than one pair of points on the sphere is $\sqrt{61}q$, a value somewhat larger than half of the mass ratio.[2] Noting the change of scale between figures 2.9 and 4.15, it is clear that there are dramatically fewer points on inter-species collision spheres.

The distribution of collision angles found in the collisions between mass 1 and mass 10 particles in an equilibrium gas at $RT = 76q^2$ is plotted in figure 4.16. Of the 100,000 collisions performed, 53,937 of them occurred between particles having different masses. The huge number of 0^o deflections dominates the distribution. Also note that the other angles are no longer symmetrically distributed about 90^o.

For a simulation with a single species discrete-velocity gas, it is not difficult to use sufficient velocity resolution to reproduce DSMC results. This procedure becomes progressively more difficult, however, for simulations with high species mass ratios because of the large look-up table needed. To alleviate this problem, it may be useful instead to specifically eliminate 0^o collisions, thereby changing the definition of a collision and of the collision time. Figure 4.17 shows results from a simulation with the same initial conditions as figure 4.14; however, if a multi-

[2]Only for odd mass ratios is the minimum relative velocity exactly half the mass ratio plus $\frac{1}{2}q$.

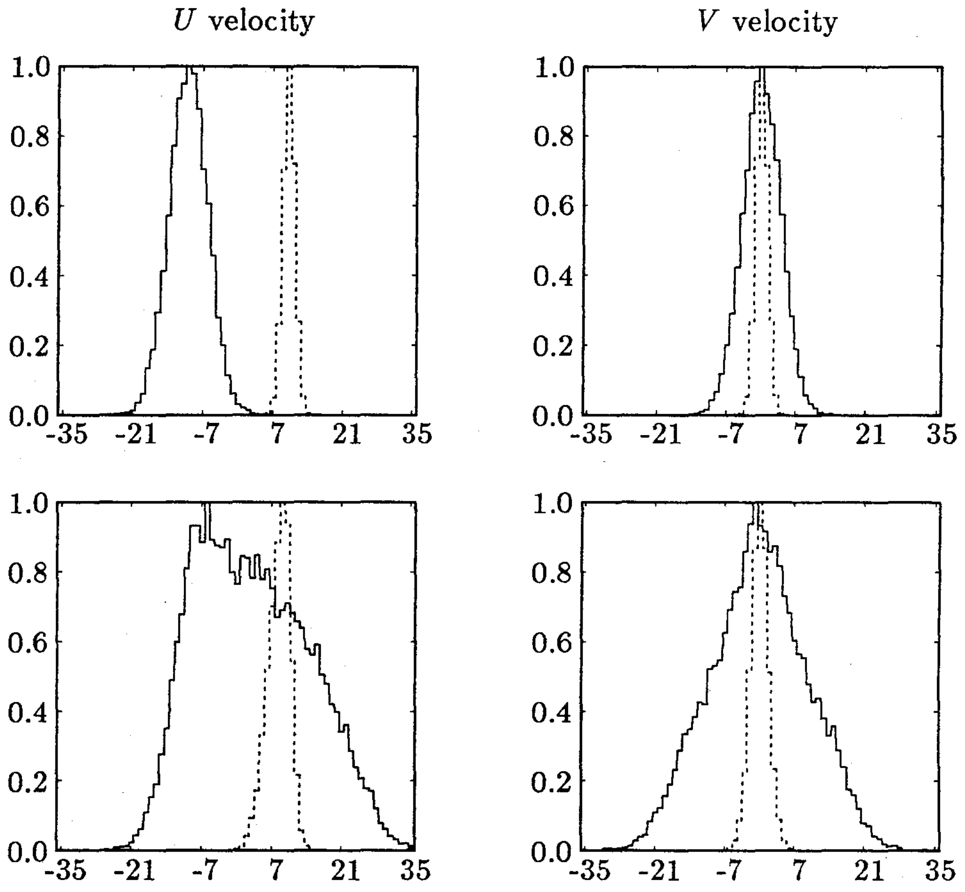

Figure 4.13: DSMC method. Development of velocity distribution function histograms in a gas of mass ratio 10 after 0 (first row) and 1 collision/particle (second row). Mass 1 = solid lines, mass 10 = dotted. $RT_{total} = 76q^2$.

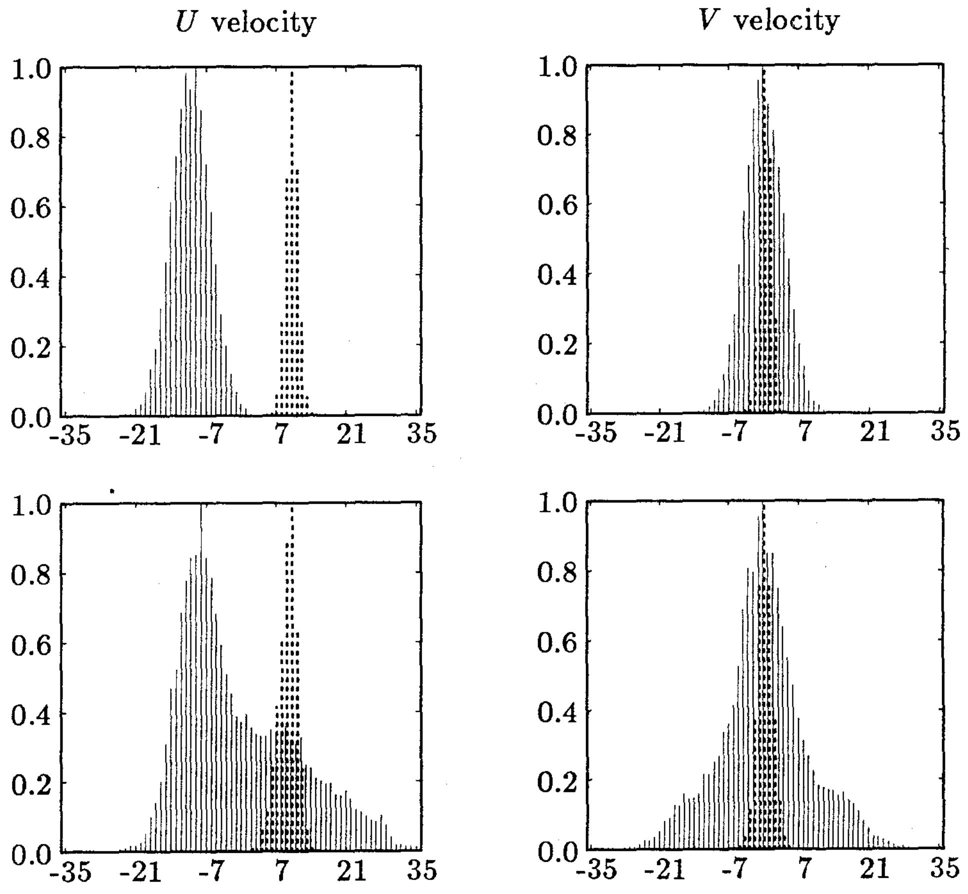

Figure 4.14: IDSMC method. Development of velocity distribution functions in a gas of mass ratio 10 after 0 (first row) and 1 collision/particle (second row). Mass 1 = solid lines, mass 10 = dotted. $RT_{total} = 76q^2$.

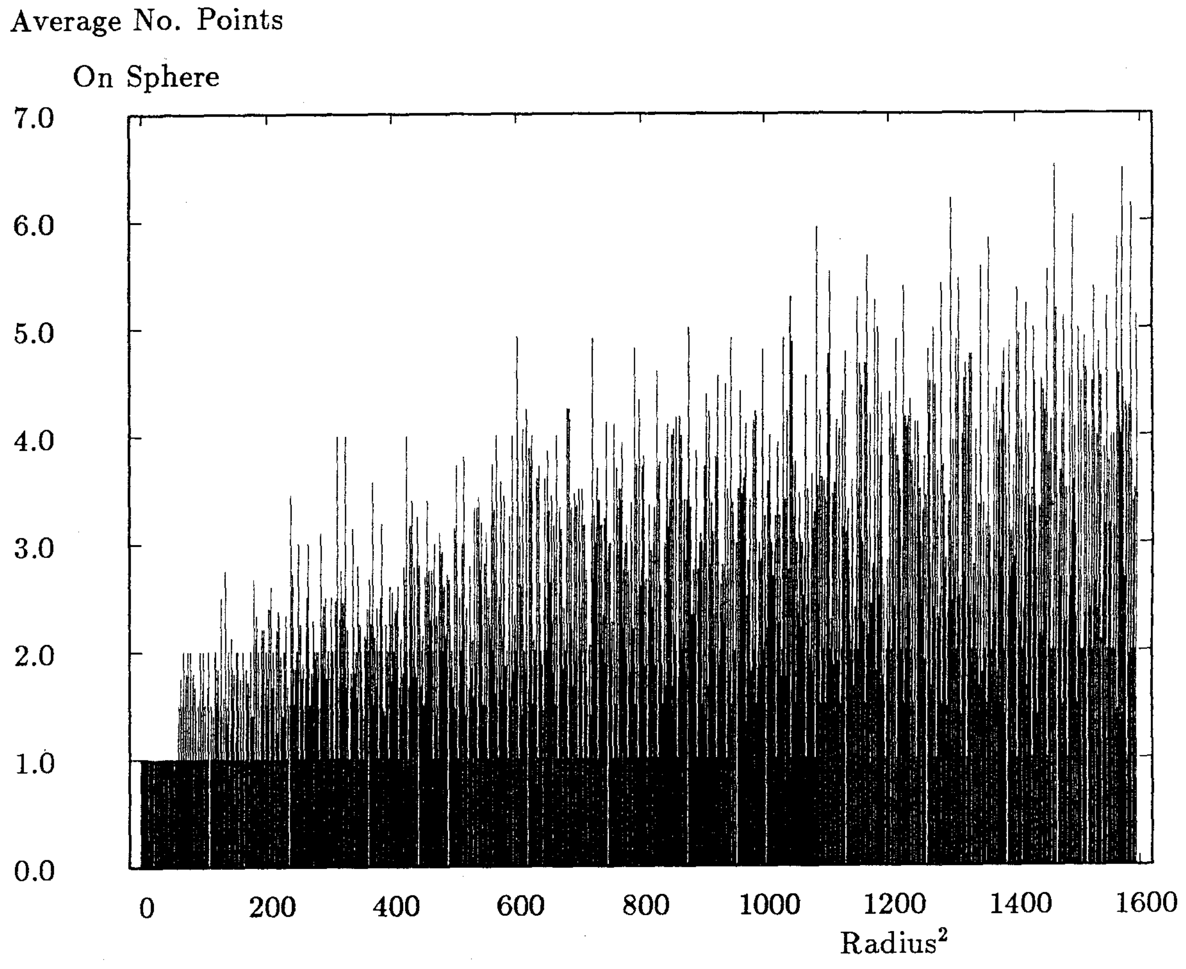

Figure 4.15: The average number of discrete points on all spheres in collisions between mass ratio 10 particles with a relative speed less than 40. Note change in scales from figure 2.9.

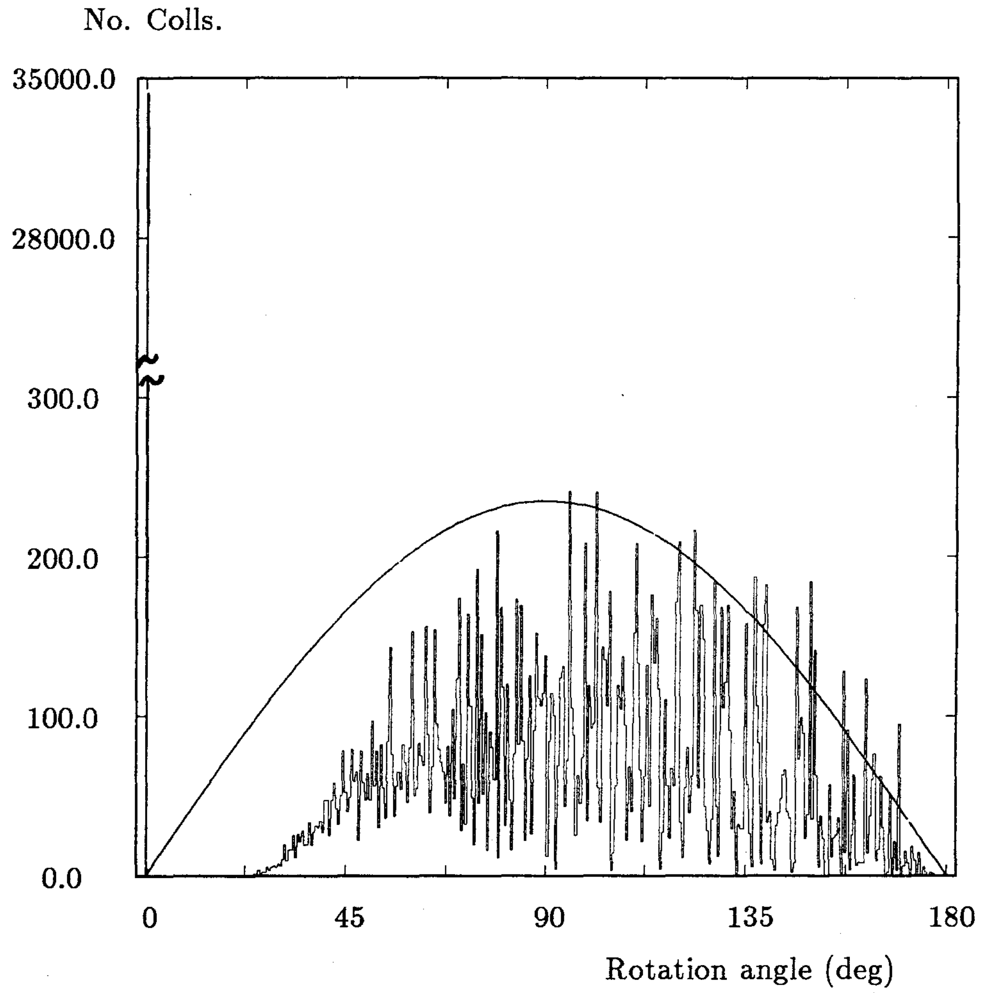

Figure 4.16: Histogram of the angle of rotation, χ, of the relative velocity vector in each multi-species collision in an equilibrium gas. The solid line is the theoretical distribution for 53,937 collisions. Particle temperature is $RT = 76q^2$. Bin width = 0.5^o. Note the two scales along the ordinate.

species collision is null, another multi-species pair is chosen to collide. The collision counter is incremented only after an effective collision has occurred. With the 0^o collisions eliminated, the tendency of the remaining angles toward large deflections causes the IDSMC distribution to relax more rapidly than is desired. It is difficult to strike a balance between the extremes of either permitting or eliminating 0^o collisions because the angle distribution that occurs depends on the particular velocity distribution of the gas. Such a scheme, however, takes the same computational effort to obtain a given result as simply performing more collisions and it is therefore not pursued further. For effective high mass ratio gas simulation, high velocity resolution is required.

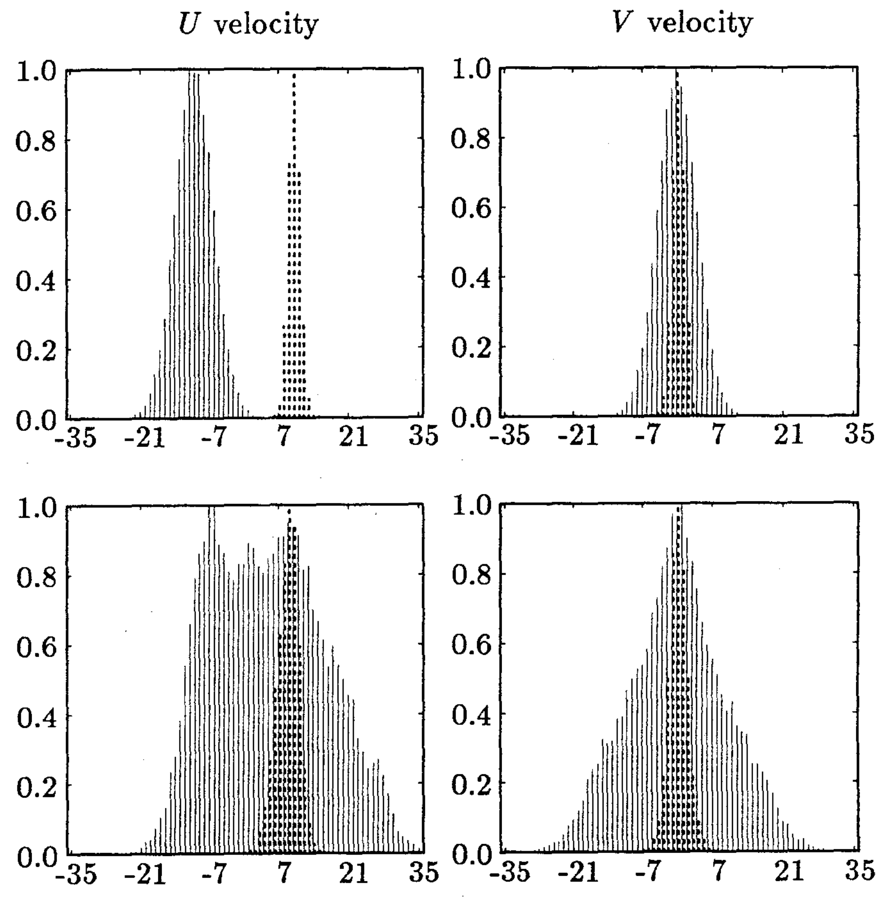

Figure 4.17: IDSMC method. Development of velocity distribution functions in a gas of mass ratio 10 after 0 (first row) and 1 collision/particle (second row). Mass 1 = solid lines, mass 10 = dotted. 0^o deflection multi-species collisions are repeated. $RT_{total} = 76q^2$. 60,000 particles are used.

Chapter 5

One- and Two-Dimensional Applications

Simulations more complicated than those of the equilibrium states or relaxation processes described in chapter 4 will clarify implications of the discrete-velocity model. In this chapter simulations of shock wave structure, heat transfer between parallel plates, and shear layer growth phenomena are used to enumerate some of the macroscopic features of the IDSMC, particularly features related to the convection of discrete-velocity particles. The performance of the DSMC and IDSMC methods on different machines is discussed. The simulations are performed as two-dimensional calculations – particles have positions (x_i, y_i) while the velocities remain three-dimensional as (u_i, v_i, w_i). One-dimensional results are obtained by averaging the two-dimensional data over the width or height of the test volume. Such averaging is equivalent to ensemble averaging many separate one-dimensional runs in which only the x particle positions are updated.

5.1 Normal Shock Wave Simulation

To rigorously test the discrete-velocity method, its solution for the structure of strong shock waves is first compared with the DSMC solution. For this purpose, calculations are carried out on a parallel processor using comparable codes with the same time step, cell size, *etc.*.

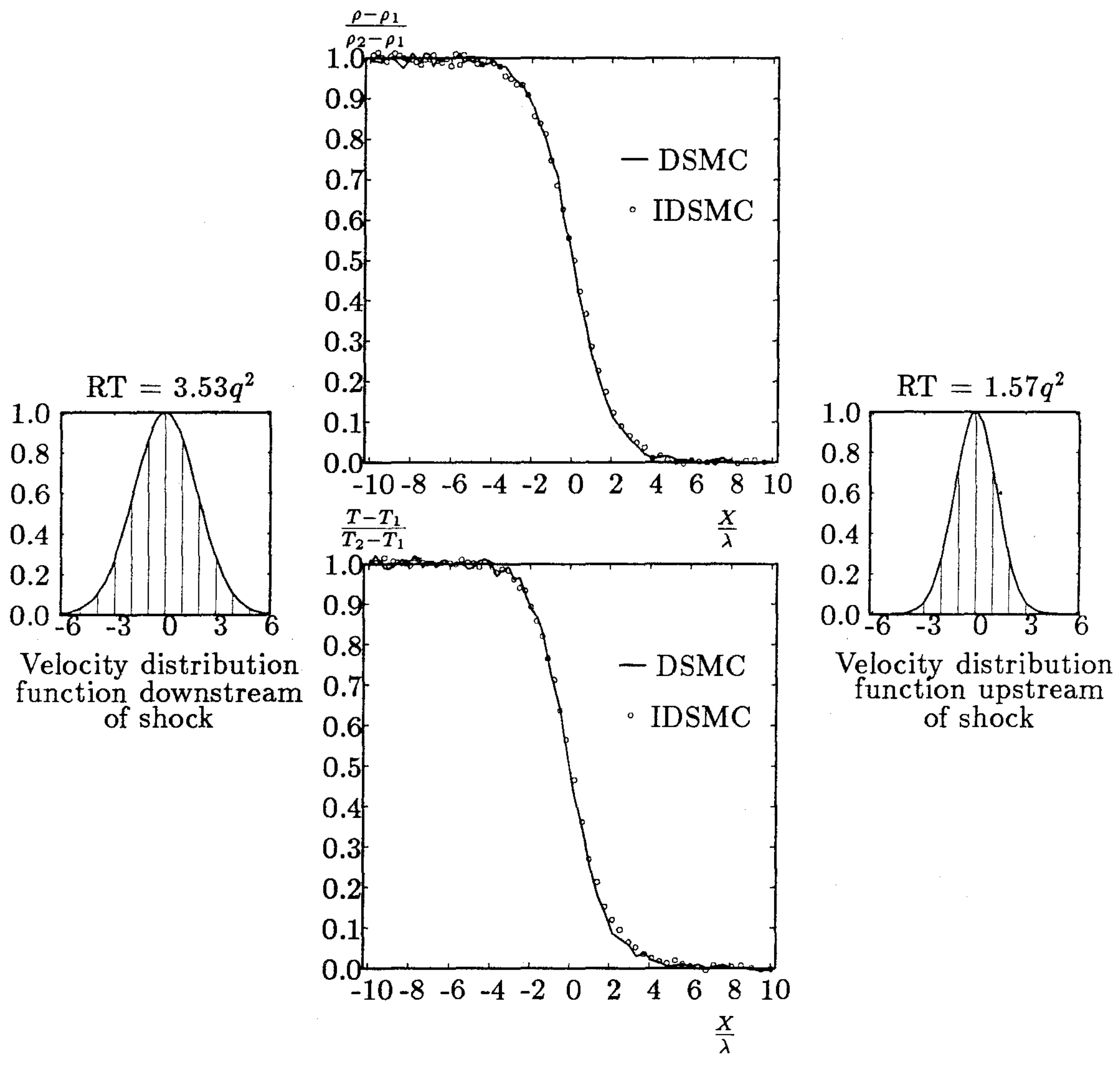

Figure 5.1: $M_s = 2.12$ Shock wave density and temperature profiles. Also shown are the up/downstream thermal velocity distribution functions.

Time step	Cell size	Part./cell	Sites/mfp	# Particles
$0.1\tau_0$	0.5λ	500	20	$5.12*10^6$

Table 5.1 Conditions for simulation of figure 5.1

Figure 5.1 shows the normalized density and temperature profiles obtained by the DSMC (solid line) and the IDSMC (points) for a normal shock wave of strength $M_s = 2.12$ in a perfect hard-sphere gas of identical particles. The space coordinate is normalized with the upstream mean free path. The discrete Maxwellian distributions of molecular thermal velocities (bar plot), corresponding to the measured uniform states upstream and downstream of the shock, together with the continuous Maxwellians (solid curves), are shown for comparison. The calculation is carried out in a nonsteady frame; the left wall of a box is impulsively accelerated to a constant speed of $2q$ at time $t = 0$, and the shock profile is sampled after 300 time steps (138 million collisions). The simulation conditions are given in table 5.1. Thirty-two processors of the Symult 2010 multicomputer are used. It can be seen that excellent agreement with the continuous-velocity DSMC model is achieved starting with only 5 values of each velocity component. In the uniform gas behind the shock, 9 values of each component are found. Similar simulations were performed with the DSMC method alone at different Mach numbers and compared to those done by Bird (1970, 1976, and 1988). The results were the same, thus confirming that the DSMC is correctly implemented.

Figure 5.2 shows results from a similar problem, but with the upstream temperature 4 times smaller and the piston speed half as large as above. In this case, the velocity resolution is poor, so the discrete calculation does not agree as well with the continuous-velocity result. Note that in the upstream and downstream

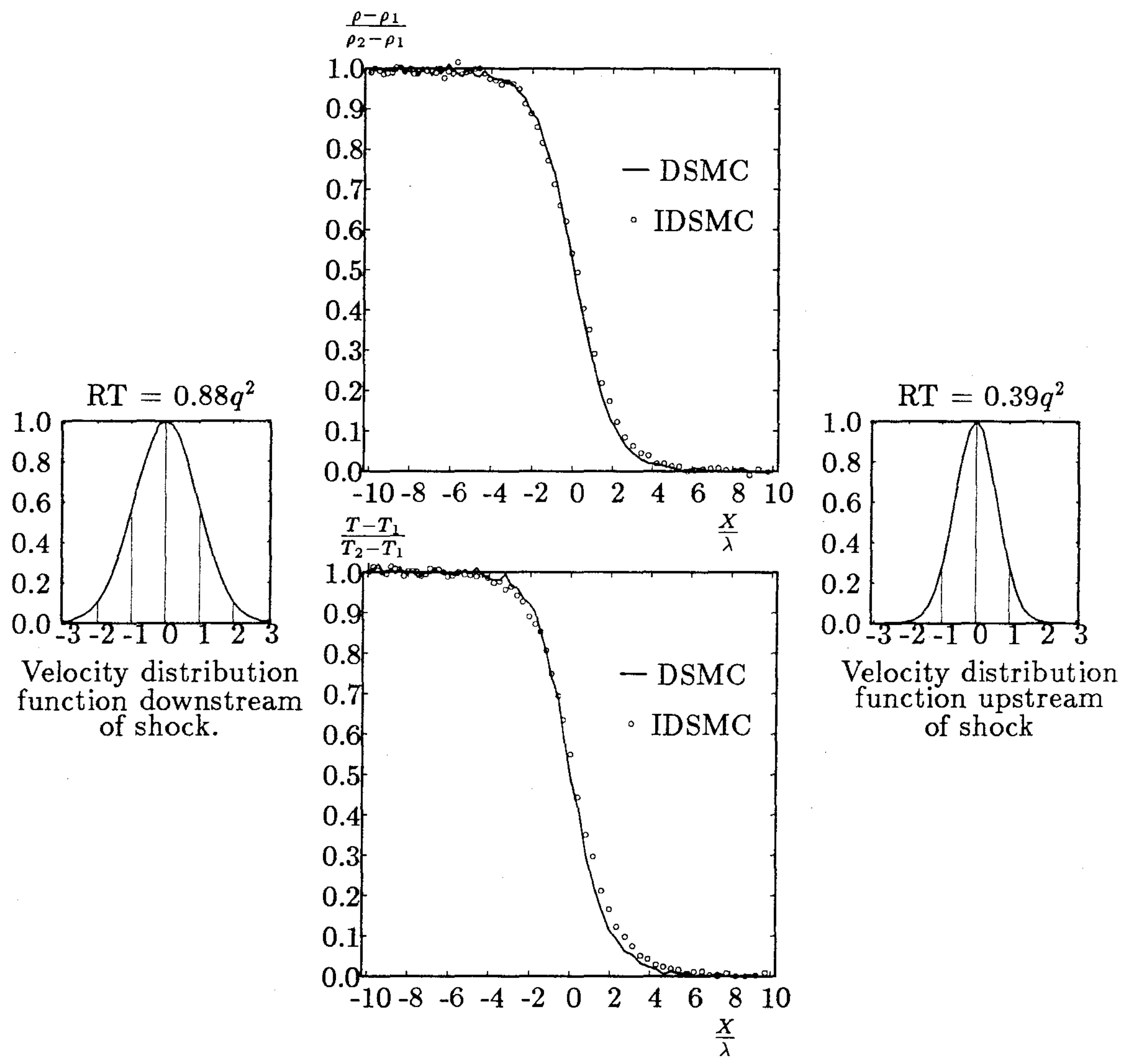

Figure 5.2: $M_s = 2.13$ Shock wave density and temperature profiles. Also shown are the up/downstream thermal velocity distribution functions.

Time step	Cell size	Part./cell	Sites/mfp	# Particles
$0.1\tau_0$	0.5λ	500	10	$5.12*10^6$

Table 5.2 Conditions for simulation of figure 5.2

states the equilibrium distribution functions are well represented, but inside the nonequilibrium shock discrepancies are evident. As discussed in chapters 2 and 4, the particle diffusivity is too large because there are just a few possible velocities and zero deflection collisions occur too often. The hot downstream particles diffuse toward the front of the shock, and the shock becomes too thick. This is a macroscopic manifestation of microscopic differences between the collision terms in the continuous and discrete Boltzmann equations, although possible effects of the convection of particles cannot be absolutely eliminated. Where in figure 5.1 the maximum slope thickness of the IDSMC density profile is 5% greater than the DSMC result, in figure 5.2 the IDSMC solution is 8% thicker. Perhaps, however, this greater discrepancy is still small enough for crude engineering calculations if a low resolution simulation is otherwise desirable.

A sensitive test of the absolute accuracy of shock structure calculations is obtained by plotting the normal component of the pressure tensor, $p_{xx} = \rho\overline{u'^2}$, against the specific volume, v. By integrating the x-momentum equation,

$$\rho u \frac{\partial u}{\partial x} = \frac{\partial p_{xx}}{\partial x} \tag{5.1}$$

it is found that $\frac{\partial p_{xx}}{\partial v}$ should be a straight line – the Rayleigh line. In figure 5.3, p_{xx}, normalized by its upstream value, which by definition is the upstream pressure, is plotted versus ρ_0/ρ for the same shock calculations as presented in figure 5.1. The cluster of points at (1, 1) is from samples near the upstream end of the shock and the cluster near (5.40, 0.42) is from the downstream end. In the calculations reported

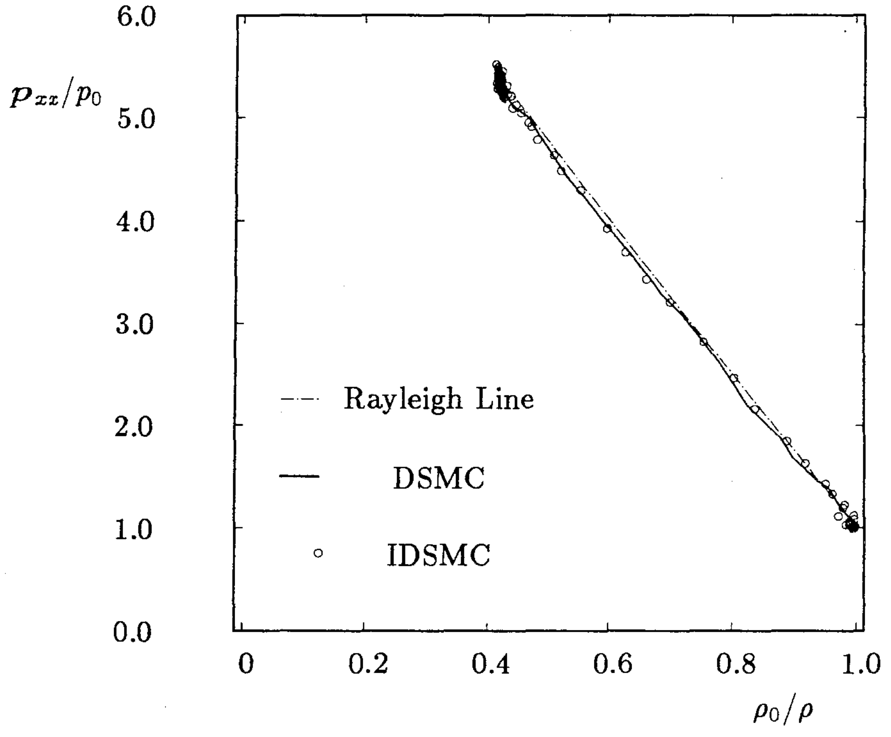

Figure 5.3: $M_s = 2.12$ p_{xx} variation through shock.

here, the cell size and time step are optimized in a trade-off between spatial and temporal resolution and computational time to achieve the performance indicated in the figure; larger values would have resulted in an S-shaped curve that deviated from the straight Rayleigh line. The figure shows that, with the same time step and cell size, the IDSMC and the DSMC give comparable performance.

5.2 Multi-Species Normal Shock

The accuracy of the IDSMC method is sensitive to the species mass ratio in a binary gas as it is to the velocity resolution in a single species gas. In this section the structures of shock waves in binary gases with mass ratios of two and five are compared. In the simulations each species, considered alone, has sufficient velocity

Time step	Cell size	Part./cell	Sites/mfp	# Particles
$0.113\tau_0$	0.5λ	200	40	$3.1*10^6$

Table 5.3 Conditions for simulation in figure 5.4

Time step	Cell size	Part./cell	Sites/mfp	# Particles
$0.08\tau_0$	0.5λ	200	40	$3.1*10^6$

Table 5.4 Conditions for simulation in figure 5.5

resolution to produce accurate results. Yet, because of the small effective size of the collision spheres in the mass ratio = 5 collisions, many such collisions are null and discrepancies with the DSMC method occur. The test Mach numbers are chosen so that the mass five particles have velocity resolution ($RT_{h0} \approx 1.6q^2$ and $RT_{h1} > 3.5q^2$) which was found sufficient to produce good results in figure 5.1.

Figure 5.4 shows a comparison of density profiles for the integer and continuous-velocity methods in a $M_s = 2.01$ shock wave. Upstream, there are equal numbers of both species; the mass ratio is 2. The parameters in table 5.3 are chosen to obtain the desired velocity resolution. The left wall speed is $4q$ and the profiles are sampled at time step 640 (170 million collisions) (figure 5.5). All 128 processors of the iPSC/1 are used. The upstream and downstream thermal velocity distribution functions are shown beside the wave for both the light (solid lines) and the heavy particles (dotted lines). The density profiles (lines) are the heavy, light, and mean DSMC values, while the symbols represent the corresponding quantities obtained with the IDSMC. Both methods show the separation of the species within the wave characteristic of the highly non-equilibrium conditions. The correspondence between the methods is good although there are small differences near the leading edge of the wave where the velocity resolution is not as high as it is downstream. The maximum slope thickness of the IDSMC mean density profile is 8% greater than the DSMC profile. It must be noted that the current version of the DSMC produces a Mach 10 wave, in a gas of 10 percent mass 10 particles, which is only about 2/3 as thick as that obtained by Bird (1976). The reason for the discrepancy is unknown.

For comparison with the mass ratio 2 case, consider a shock in a gas with a particle mass ratio of 5 but in which the maximum velocity resolution for the light species remains essentially the same (figure 5.5). The integer model produces a 32% thicker shock due to insufficient coupling during heavy/light collisions, rather than to inadequate velocity resolution for either species taken on its own. In fact, many heavy/light collisions are completely ineffective because the relative speed of the mass 5 and 1 particles did not exceed the value of $3q$ ($= \frac{1}{2}q + \frac{1}{2}$the mass ratioa), which is the lowest speed for an effective collision. The high mass ratio results in a decrease in the number of points on the small collision spheres of the mass 5 particles during multi-species collisions and, thus, increased diffusivity. Substantially greater velocity resolution would be required for an adequate number of points to occur on the collision spheres in collisions between particles of such disparate masses.

5.3 Heat Transfer Between Solid Surfaces

Heat transfer between parallel plates is discussed in this section. In order to demonstrate the application of a diffuse boundary condition in a lattice gas and to provide an example in which the effect of velocity discretization on the left hand side of the Boltzmann equation is readily apparent, both the unsteady time development and

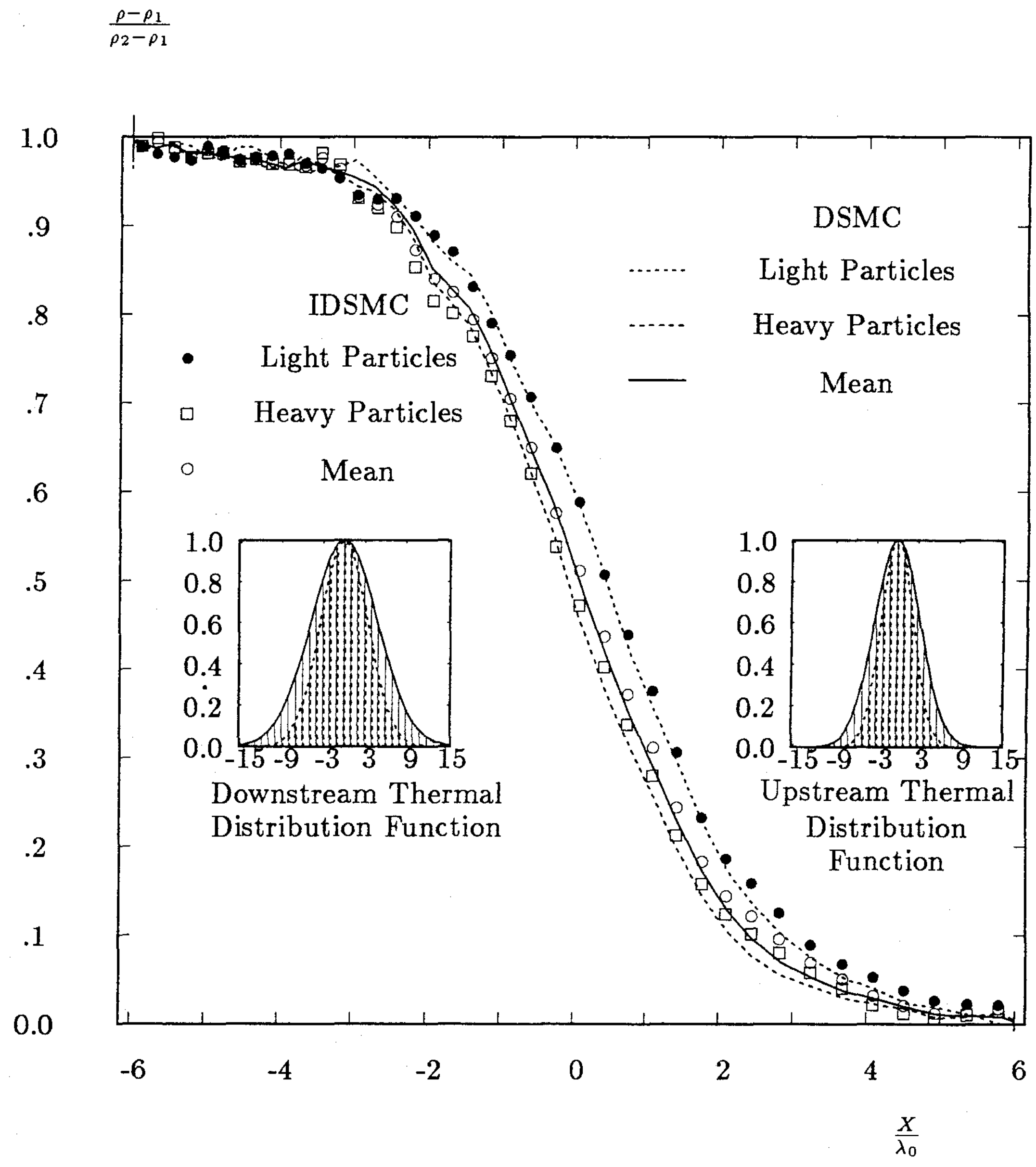

Figure 5.4: $M_s = 2.01$ Shock density profiles. Mass ratio = 2. Number densities are equal. Upstream, $RT_{l0} = 11.18q^2$ and $RT_{h0} = 5.59q^2$. Downstream, $RT_{l1} = 23.25q^2$ and $RT_{h1} = 11.62q^2$. Also shown are up/downstream thermal distribution functions ($m = 2$: dotted lines and bars, $m = 1$: solid lines and bars).

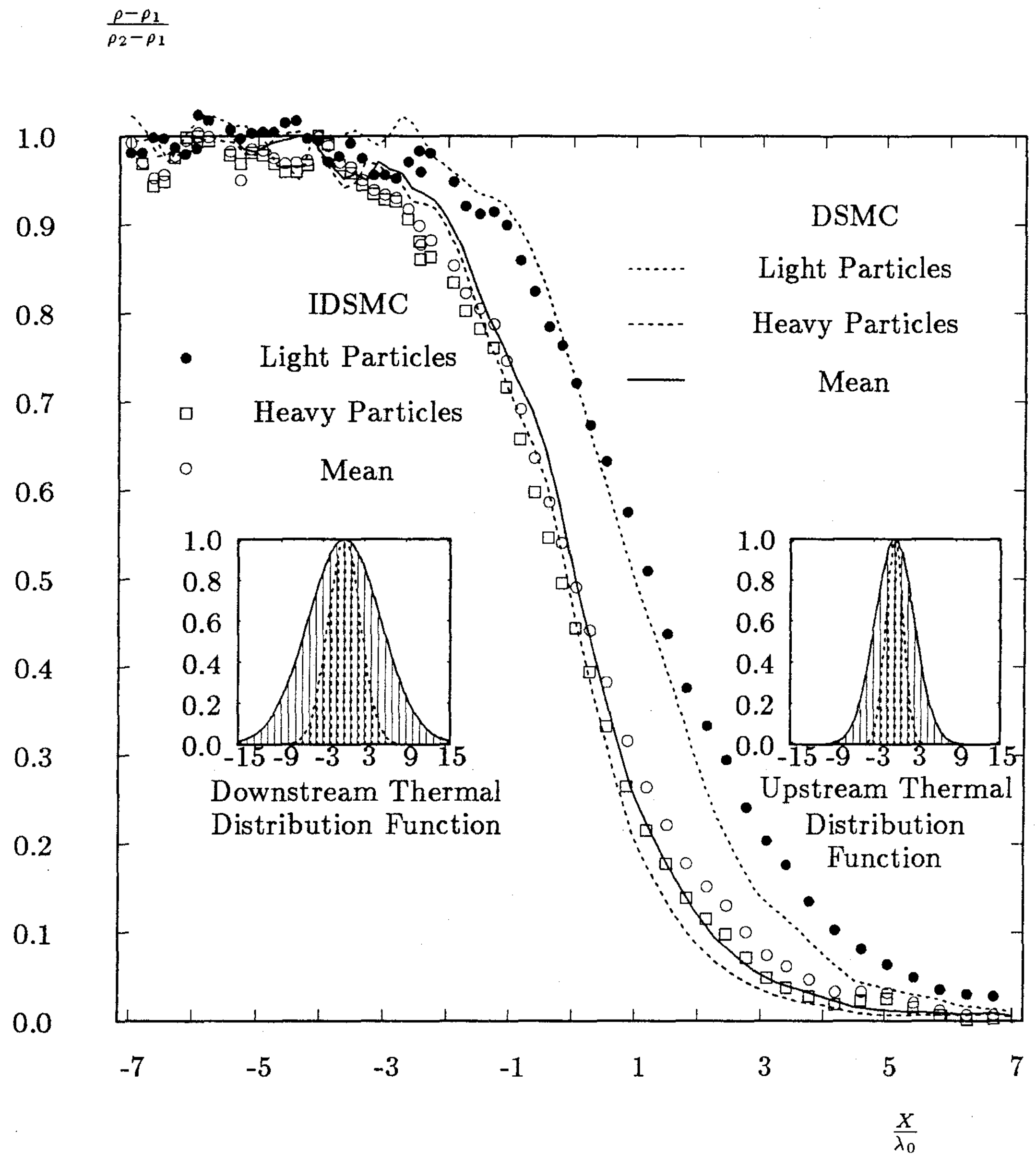

Figure 5.5: $M_s = 2.84$ Shock density profiles. Mass ratio = 5. Number densities are equal. Upstream, $RT_{l0} = 8.25q^2$ and $RT_{h0} = 1.66q^2$. Downstream, $RT_{l1} = 27.77q^2$ and $RT_{h1} = 5.55q^2$. Also shown are up/downstream thermal distribution functions ($m = 5$: dotted lines and bars, $m = 1$: solid lines and bars).

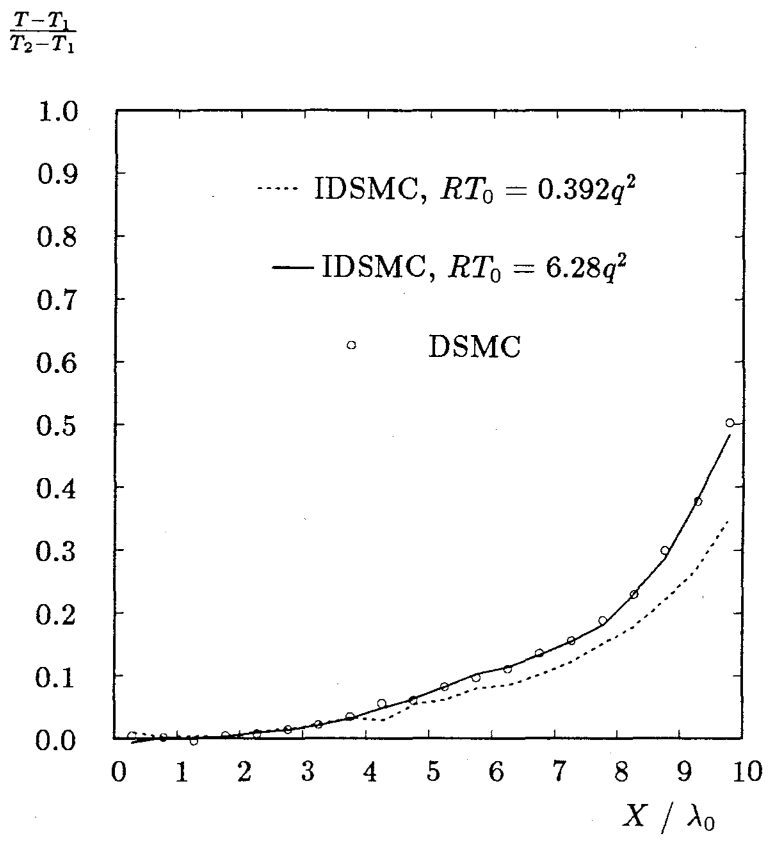

Figure 5.6: Temperature distribution at early times - heat transfer between parallel walls. Wall temperature ratio = 2, 4 collision times.

the steady state solutions are examined. Initially, both plates are at the same temperature and, at $t = 0$, the right hand wall is raised to twice the ambient temperature. All of the particles are identical. The spacing between the walls is 10 mean free paths. Although this configuration involves flat surfaces parallel to the lattice axes, the simulation method can be extended to curved surfaces as well.

Figure 5.6 presents the temperature distribution after four collision times for both the DSMC and the IDSMC methods. The two curves for the IDSMC approach correspond to simulations with initially high velocity resolution ($RT_0 = 2\pi q^2$) and low resolution ($RT_0 = 0.392q^2$) while the circles represent the DSMC results. In

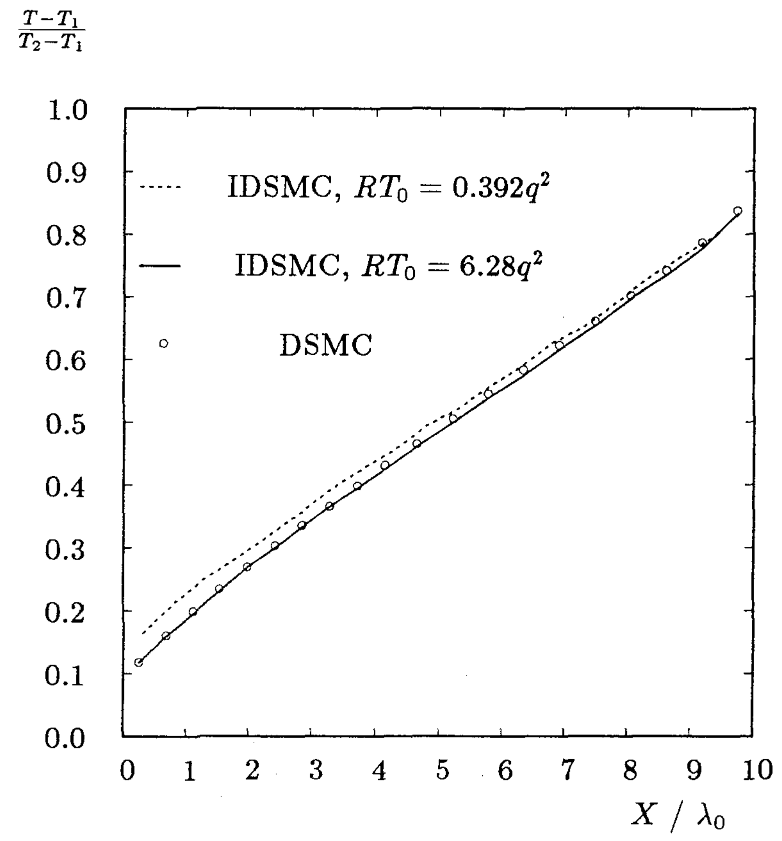

Figure 5.7: Temperature distribution at equilibrium - heat transfer between parallel walls. Wall temperature ratio = 2.

Resolution	Time step	Cell size	Part./cell	Sites/mfp	# Particles
high	$0.2\tau_0$	0.5λ	60	20	$5.1*10^6$
low	$0.1\tau_0$	0.5λ	200	10	$2.6*10^6$

Table 5.5 Conditions for simulation in figures 5.6 and 5.7

the DSMC and high resolution IDSMC simulations, 20 time steps (10.5 million collisions) are calculated to reach 4 collision times. In the low resolution case 40 time steps (5.6 million collisions) are calculated. With sufficient resolution, it is clear that the integer method reproduces the continuous-velocity results well, but that low velocity resolution leads to insufficient heat transfer at the wall as seen by the depressed temperature throughout the domain.

When the gas reaches equilibrium there may be differences between the high and low resolution integer solutions because of null collisions as well as due to the different heat flux from the walls. Null collisions cause a decrease in the slope of the temperature profile so that, from the equilibrium state alone, it is not clear which effect causes the discrepancy with the high resolution calculation. Figure 5.7 shows the steady state solution where the low resolution profile has greater temperature slip than the high resolution profile at the cold boundary where the resolution is lowest. To obtain these profiles a total of 102 million collisions are calculated. The DSMC and high resolution profiles have the distinct curve and inflection point that have been found by others as well (Sone, 1989). Near the hot wall, the velocity resolution improves and the low resolution IDSMC solution corresponds better with the high resolution and DSMC results. Inamuro, applying his method, which is also based on the concepts of points on spheres in a discrete-velocity space, obtains similar results for the equilibrium state (Inamuro, 1989a).

Profiles for the small temperature ratio of 1.05 were computed with the present DSMC method and produced the same temperature jump as found by Sone (1989), thus providing further assurance that the DSMC method is implemented properly. Applications in the linear regime, with their small amplitudes, suffer from substantial noise and will not be discussed further.

The low heat transfer at the wall in a discrete-velocity gas is due to a decreased

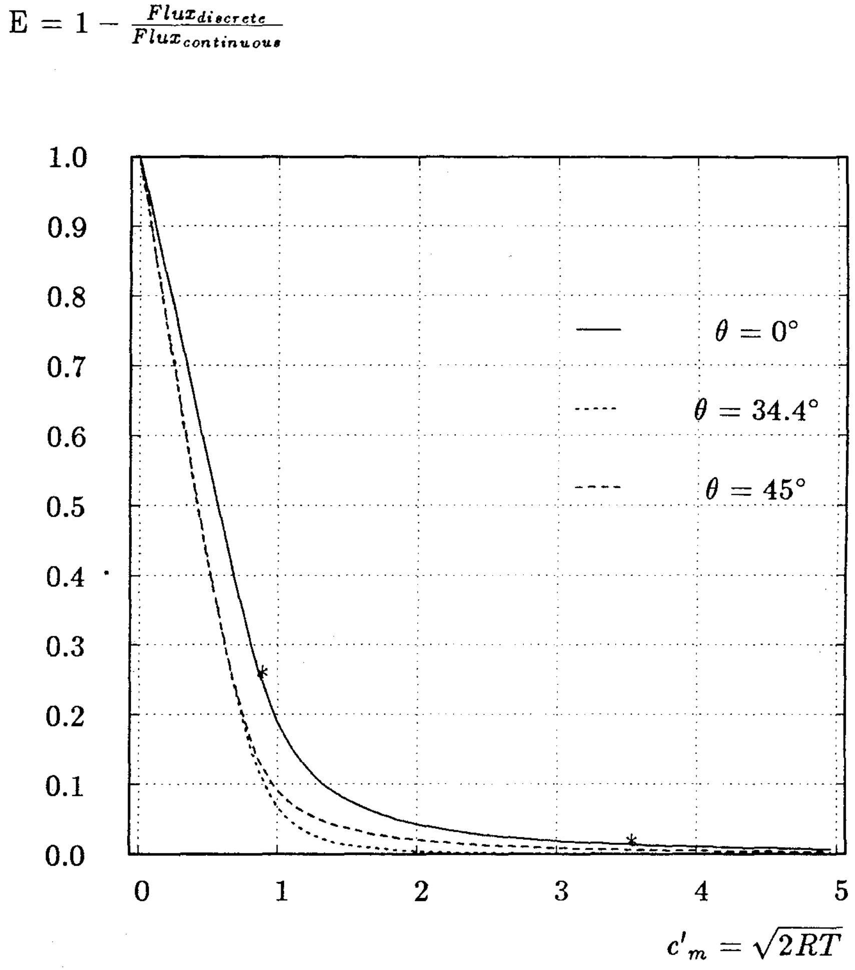

Figure 5.8: Error in particle number flux from an equilibrium gas through a boundary at an angle θ versus velocity resolution. * indicates simulation conditions for figures 5.6 and 5.7

particle flux to the wall (suggested by Inamuro). The particle flux error across a flow boundary at an angle θ with respect to an X or Y axis of the velocity lattice in an equilibrium gas, is plotted in figure 5.8. The error E is defined as

$$E = 1 - \frac{\sum_{i>boundary}\sum_{j=-\infty}^{+\infty}\sum_{k=-\infty}^{+\infty}[i\ cos(\theta) - j\ sin(\theta)]f(i,j,k)q^3}{\int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty}\int_{0}^{+\infty}\vec{c}'_{normal}f(\vec{c}')d\vec{c}'}. \tag{5.2}$$

The summation in the numerator represents the discrete-velocity number flux. The integral in the denominator is the continuous-velocity number flux. The continuous distribution function, $f(\vec{c}')$, is simply a Gaussian while the discrete distribution function, $f(i,j,k)$, is found from the discrete Boltzmann equations (Inamuro, 1989a) for c'_m less than 1.2 and from a Gaussian distribution at higher temperatures. c'_m is the most probable thermal speed, and thus corresponds to the width of the distribution function. The error initially falls off with increasing velocity resolution but falls less rapidly at higher temperatures.

A boundary parallel to the velocity lattice ($\theta = 0$) has the largest flux error; hence, for the computation of a flow around a curved shape, it is simple to determine the velocity resolution needed to keep the error below a desired level. With low velocity resolution, a substantial number of particles move parallel to the boundary (or nearly so) and thus never cross it. For example, if $\theta = 0$, particles with $u = 0$, which represent the region of velocity space $\frac{-1}{2} < u < \frac{+1}{2}$ and $-\infty < v < +\infty$, never cross the boundary and hence, never directly feel its presence. In a continuum velocity gas, no particles have a precisely zero normal velocity component.

That there should be a disparity between the discrete and continuous-velocity solutions solely due to an error in particle convection is not unexpected. Consider the collisionless form of the Boltzmann equation,

$$\frac{\partial}{\partial t}(nf) + \vec{c}.\frac{\partial}{\partial \vec{r}}(nf) = 0, \tag{5.3}$$

which has the simple solution $f(x,t) = f_{init}(x-t/c)$ in one dimension. An imprecise representation of the initial distribution function, f_{init}, introduced by a discrete-velocity representation, can cause "streaming" or "ray" effects for particles having the same velocity in an expanding flow.

The small oscillations of the low resolution temperature profiles (figures 5.6 and 5.7) are not noise but result from a collisionless phenomenon caused by the discretization of space, velocity, and time in a gas in which particles may move with more than a single speed normal to a surface. Before they collide with other particles, those particles leaving the walls with the same normal discrete-velocity will travel along straight, parallel, and uniformly spaced particle paths in an $x - t$ plane (figure 5.9). Where such paths cross at a spatial lattice site, particles of different speeds coexist and there may be a substantial periodic variation of the macroscopic quantities (density, velocity, pressure, *etc.*).

In the heat transfer problems, the Knudson number is 0.1 so particles reflected from a wall may not be deflected by collisions until they are well into the flow. Some low speed collisions do not effectively alter the particle path because it is possible that, even if the v and w components are altered in the collision, the u components of the velocity will remain unchanged. Periodic macroscopic waves can thus extend across the small domain. The low resolution solutions were computed with 10 sites per mean free path and a cell size of half a mean free path. Hence, each value of temperature plotted in figures 5.6 and 5.7 represents an average over about 5 lattice sites. The data are not smooth because 5 sites contain a few oscillation periods and the average of a periodic function over such a small number of cycles is sensitive to the exact number of periods considered. For realistic simulations, each cell should contain enough sites (perhaps 10 or more) to obtain smooth macroscopic results and to ensure that the space-velocity correlation induced by a boundary does not cause an incorrect sampling of velocities during collision calculations.

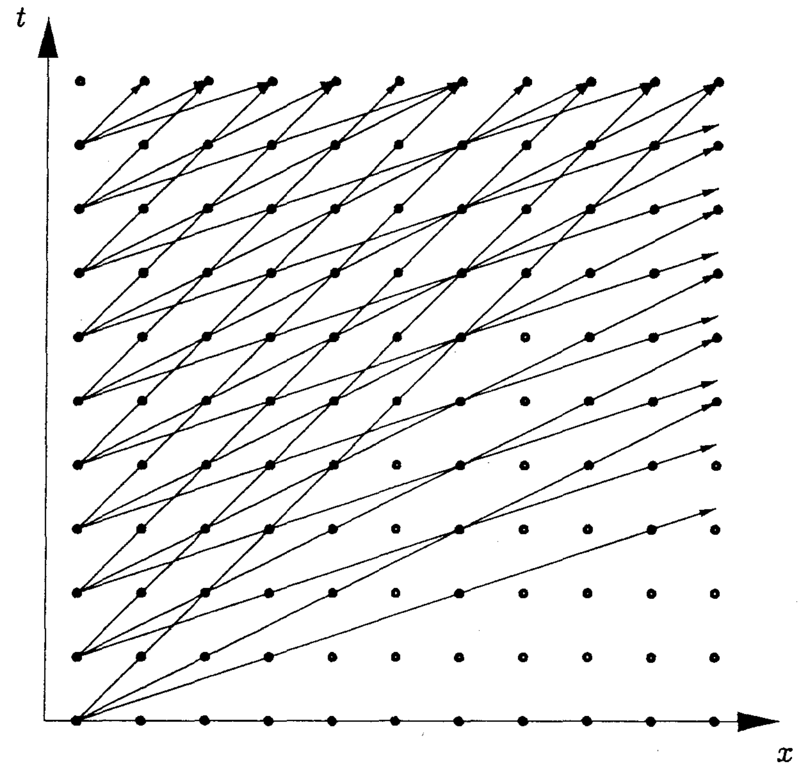

Figure 5.9: $x-t$ plot of particle paths for speed 1, 2, and 3 particles originating from the lattice sites nearest the surface. Where paths cross at a lattice site, particles of the respective speeds will coexist for one time step.

5.4 Two-Dimensional Shear Layer

An unsteady two-dimensional simulation both demonstrates the feasibility of computing a complicated problem on a parallel processor and provides assurance that moderate velocity resolution is sufficient for successful use of the IDSMC. A rectangular region 320 mean free paths wide by 480 high is considered (figure 5.10). The left and right boundaries are periodic; any particle passing out through one side is re-injected through the other. The horizontal boundaries are specular reflectors. The ambient gas temperature, RT_0, is $2\pi q^2$. At time zero there is a sinusoidal division midway through the flow with a peak to peak amplitude of six percent of the flow field height. A velocity perturbation is created by placing 100 point vorticies along the interface and creating the particles in each cell with a mean velocity, due to the vorticies, calculated at the center of the cell. The mean velocity far from the layer is nearly horizontal although slight rarefaction and compression waves may result from small initial inclinations of the flow. The particles throughout the flow have the same mass but are labeled to indicate whether they originated above or below the initial shear layer. Each stream moves with Mach number 0.6; hence, the relative Mach number is 1.2. The cell size is initially taken to be $1.0\lambda_0$ wide and $1.5\lambda_0$ high because the Mach numbers involved are low, and the smallest expected features should be several times greater than λ_0 after the initial shear layer has begun to diffuse. Because the temperature rise (and, thus, the change in the collision frequency) in all regions of the flow is expected to remain small, the time step is chosen to be $0.4\tau_0$. There are 800 cells per processor for each of 128 processors on the Symult 2010. The simulation continues for 3200 time steps (2.9 billion collisions) with data being saved after every 100 steps. Remeshing occurs every 8 steps.

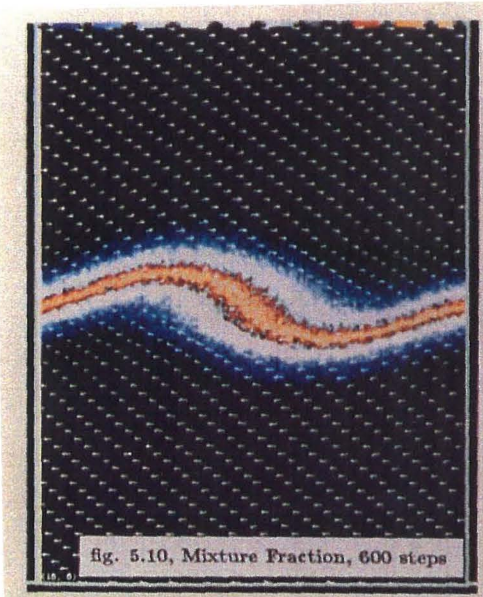
fig. 5.10, Mixture Fraction, 600 steps

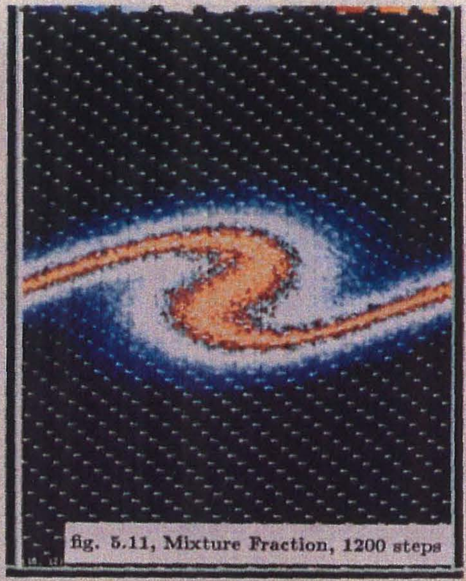
fig. 5.11, Mixture Fraction, 1200 steps

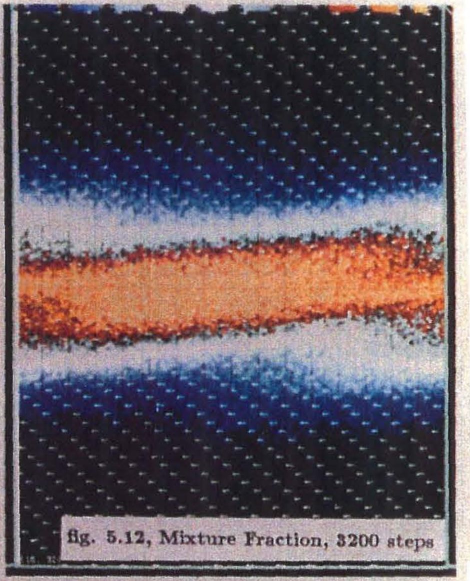
fig. 5.12, Mixture Fraction, 3200 steps

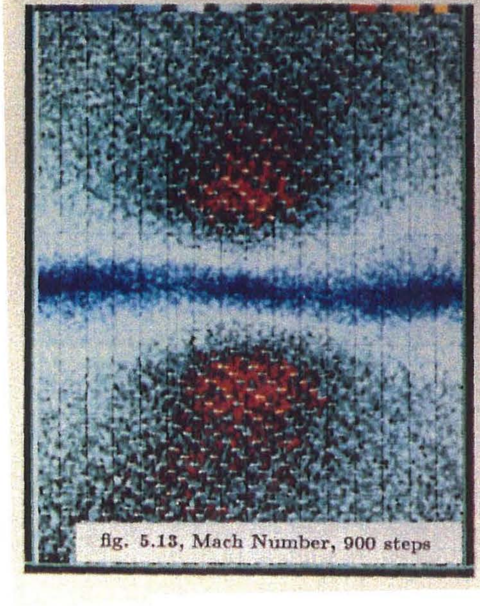
fig. 5.13, Mach Number, 900 steps

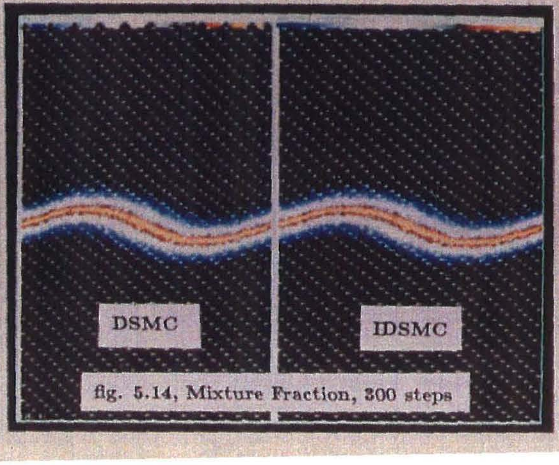

fig. 5.14, Mixture Fraction, 300 steps

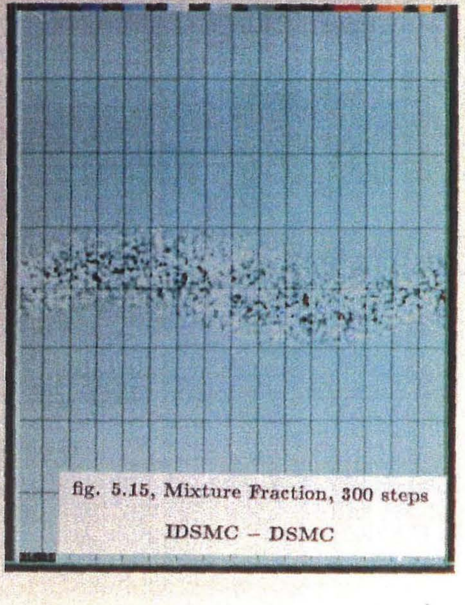
fig. 5.15, Mixture Fraction, 300 steps

IDSMC – DSMC

Time step	Cell size	Part./cell	Sites/mfp	# Particles
$0.4\tau_0$	$1 \times 1.5\lambda_0$	40	10	$4.1*10^6$

Table 5.6 Conditions for simulation of the shear layer.

In figures 5.10 - 5.12 the mixture fraction of particles that began on either side of the shear layer is shown as well as one in every 92 velocity vectors. Black represents unmixed fluid while yellow represents a 50:50 mixture ratio. The color bar is at the top of the figure. Individual cells are difficult to distinguish but the domains computed by each processor are outlined with thin black lines. As soon as the simulation begins, the shear layer diffuses rapidly and very weak compression and rarefaction waves propagate away from the layer. The particles near the perturbation have about 20 velocities available (± 10 in each direction) while those further away have about 16. A low Mach number simulation is more difficult to achieve than one at a high Mach number because the amplitude of the pressure, density, temperature, vorticity or Mach number (figure 5.13)[1] perturbations does not rise much above the noise and because so many time steps are required for the flow to change substantially. The mixture fraction is the least noisy quantity.

As discussed in chapter 3, remeshing is based on the particle number densities. In this simulation, however, the quantity balanced between the processors is a combination of the particle number density and the mean mixture fraction of the two particle types. This causes the mesh to become smaller in the regions of greatest mixing where a fine mesh is desirable. At the final time shown (figure 5.12) the cells in the center are about 1 mean free path wide by 0.96 high while those far from the region of shear are about 1 by 1.86 mean free paths.

[1]Colors (blue $->$ yellow) indicate a scale of Mach number from zero to one.

As time increases, the initial perturbation at first appears to grow as the shear layer rolls up. Yet a substantial rolling motion never develops before the perturbation is damped out by viscosity. The Reynolds number based on the perturbation wavelength and a single stream Mach number is only 316. Other similar simulations have indicated that the close proximity of the horizontal walls probably did not damp the wave as much as did the viscosity. Because the relative Mach number is low and no shocks are seen, the damping is also probably not due to compressibility effects (Papamoschou, 1986).[2]

The same simulation is carried out using the DSMC method and the results of the two methods are compared after 300 time steps in figure 5.14. It is clear that the IDSMC does not produce any noticeable asymmetries in the flow. In fact, if the DSMC results are subtracted from the IDSMC results and the difference plotted (figure 5.15), it is clear that that difference is simply due to random fluctuations within the shear layer. Outside of the layer, the two particle types are not yet mixed and there are no fluctuations. A more quantitative comparison can be made by comparing profiles of the u component of velocity along a specific section through the flow. In figure 5.16 is shown a u profile slicing through the shear layer with a column of cells located at $X = 159\lambda_0$. The IDSMC and DSMC results appear essentially the same. Although the noise level of the IDSMC method has not been specifically quantified, it appears essentially the same as that of the DSMC. Thus, with sufficient velocity and spatial resolution it seems that asymmetries introduced into the Navier-Stokes equations by the lattice gas representation are not significant.

[2]A simulation in a flow field ten times larger with cells also ten times larger, did produce a more substantial roll-up. Such a large cell size, however, is probably unrealistic.

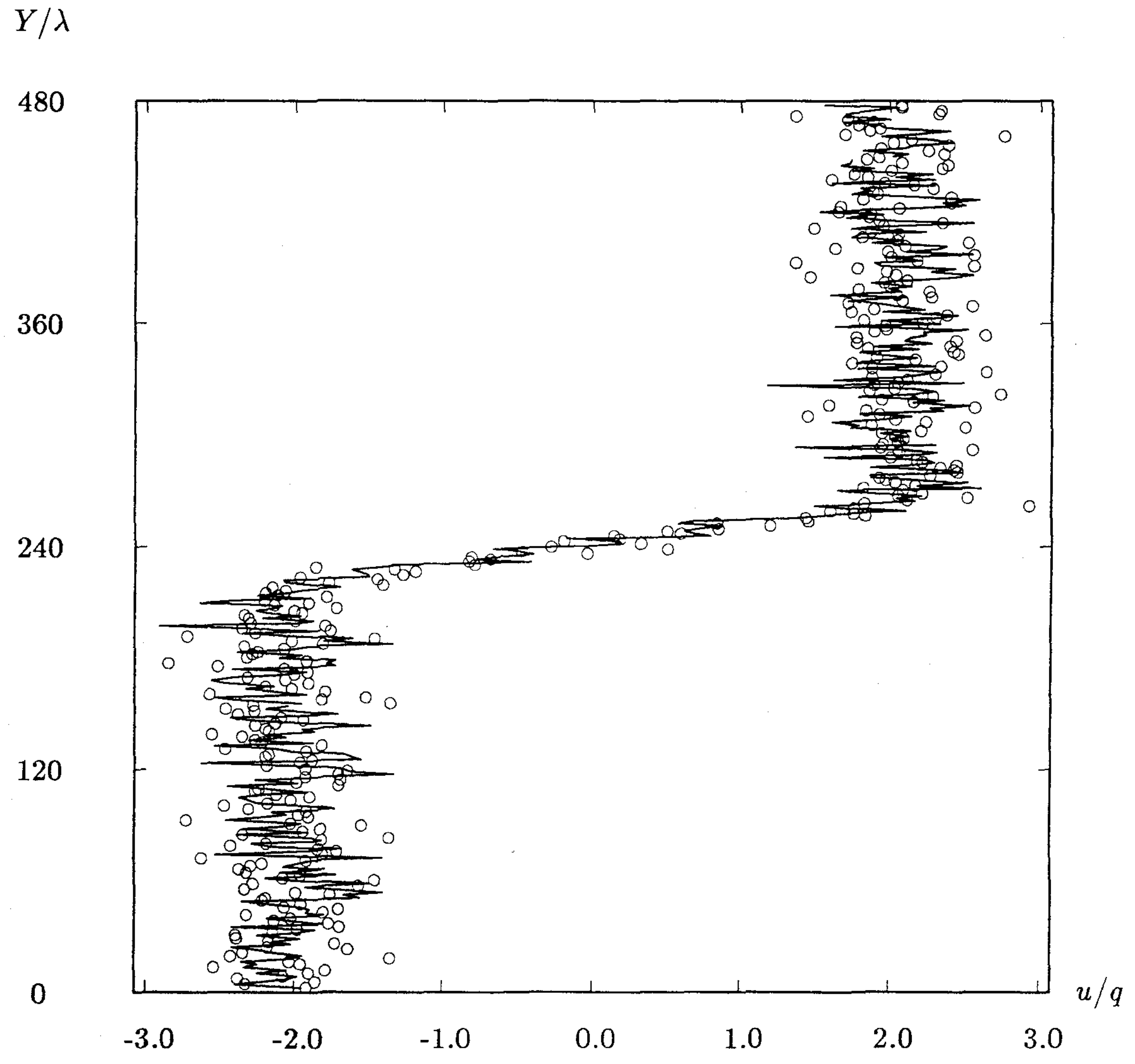

Figure 5.16: u velocity profiles through the shear layer obtained with DSMC (lines) and IDSMC (symbols) at 159 mean free paths from the left hand boundary.

5.5 Performance Comparison

By declaring the molecular velocity components to be one byte long, provision for 256 different values of velocity in the multicomputer used for the present calculations, allows 2.3 times more particles to be treated in a two-dimensional flow than when the particle velocities are stored as real numbers. With real numbers, the storage per particle is 23 bytes (12 for the 3 components of velocity, 8 for 2 positions, 2 for the particle index, and 1 for the particle type), while with the IDSMC method it is 10 bytes (3, 4, 2, and 1). Thus flows with Reynolds numbers 2.3 times greater can be calculated. Clearly, because many fewer than 256 velocity values may be necessary, this result could be improved if the velocities were stored in a more compact manner. For example, if only 5 bits were used for velocities, 10 bits for positions, and 2 bits for the type, the improvement could, in principle, be 3.5 times.

By design, the IDSMC computes primarily with integer arithmetic and the DSMC with floating-point arithmetic; hence, comparison of the relative speed of the two methods may be machine dependent. In the calculations of relaxation to equilibrium reported in Chapter 4, in which only collisions in one large cell are calculated and the particles are not moved, the IDSMC ran about 3 times faster for single species collisions and 2 times faster for multi-species collisions than did the DSMC on a sequential SUN 3/60 microcomputer. This gain results from the relatively slow square roots, random number generation, and trigonometric functions required in the DSMC computation. On the other hand, the simulations that were run on the Intel iPSC multicomputer did not show the same relative performance improvement for the IDSMC: the single species shock showed a 10 percent improvement, the binary gas shocks had 23 percent, and the heat transfer showed no gain. On the Symult, the shear layer simulation showed a 40 percent gain. The relative time each method spent computing collisions remained basically the same as on the SUN. The differences between the parallel and the sequential simulations reflect the proportion of time spent performing collisions. The differences may also reflect the relative performance of the machines for floating point and integer arithmetic. The efficiency of the parallel computation is defined as one minus the ratio of the time spent performing parallel processing tasks (*i.e.*, sending and receiving particles) to the total computational time. On the iPSC and Symult, both codes generally ran with an efficiency of 70 to 80 percent. On the Symult the maximum rate of collisions, using the IDSMC on 128 nodes for the shear layer problem, was 222 million per hour. In view of the fact that the codes have not yet been optimized for speed, it is clear that more work needs to be done to select definitive benchmarks. Improvements can be made to both the DSMC and the IDSMC codes, but the relative performance could remain the same. Nonetheless, the modest improvements in computational time and memory usage realized on the multi-computers with the IDSMC, allow such particle simulations to be extended further toward the continuum flow regime.

Chapter 6

Conclusion

A discrete-velocity model of particle motion in rarefied flows has been studied theoretically and through numerical experiments. The investigation was motivated by a desire to understand the fundamental physics of a discrete-velocity gas and by the need to link discrete-velocity molecular models to accepted continuous velocity models. A new molecular model has been developed to permit such a linkage and has been implemented in the integer direct simulation Monte Carlo method on multi-computers for large scale calculations. The method may also be an acceptable engineering tool to extend particle simulations toward the continuum flow regime. The most important conclusions are

- a computational model, accommodating arbitrary particle velocity resolution, multi-species gases, and diffuse boundary conditions, has been developed;
- moderate velocity resolution is needed to accurately simulate non-equilibrium flows;
- averaging macroscopic quantities over several spatial lattice sites is necessary to avoid "ray" effects;
- lattice gas methods can be successfully applied to rarefied flows;

- the new integer method, as well as the standard DSMC method, can be efficiently implemented on multi-computers.

A summary of the main results follows.

6.1 Summary

An integer velocity model of molecular gasdynamics has been developed and implemented with a direct particle simulation. In the model, each particle possesses integer velocity components (u, v, w) and may collide with other particles. During such a binary collision, mass, momentum, and energy are exactly conserved by determining post-collision particle velocities from the symmetry reflections between points on a sphere in the center-of-mass frame of reference. Multi-species collisions involving two separate spheres and a velocity sub-grid are described. High velocity resolution leads to larger spheres and a large variety of collision possibilities. Through the collision process, all velocity space is filled although the distribution of velocities found depends on the nature of the problem being solved.

The DSMC method is the present state of the art for computing rarefied or transitional flows of real gases around complex geometries. When this method is applied to the discrete-velocity model, each particle resides at a lattice site in space and hops among the sites during the movement phase of the computation. During the collision phase, particles are temporarily held at the lattice sites and are randomly chosen to collide from among the sites contained within each computational cell. For each selected particle pair, the relative speed is used to access the possible outcome relative velocity from a look-up table. In a hard-sphere continuous-velocity gas, the distribution of the relative velocity orientation is spherically symmetric. Hence, the look-up table contains the points on the surface of spheres in a discrete-velocity space and one of those points is chosen randomly as the post collision relative velocity. This concept of points on a sphere and how to compute or find them efficiently is central to the discrete-velocity method.

After the model and method were described, a numerical experiment rather than a theoretical approach was pursued to examine the model's range of validity. A theoretical investigation would have been impractical because it could not be used for applications far from equilibrium. The numerical experiments involved four different types of problems: (i) demonstrating the equilibrium state for a simulated discrete-velocity gas and comparing the result with theory; (ii) proving the correspondence between the IDSMC and DSMC results for unsteady simulations of distribution function relaxation where no exact theory exists; (iii) showing the effective equality of macroscopic quantities using the IDSMC and DSMC methods for one-dimensional shock wave and heat transfer simulations; and (iv) establishing the agreement between the DSMC and IDSMC for a complicated two-dimensional flow field.

It was found by simulation that the discrete velocity distribution function for an equilibrium gas agrees with the continuous-velocity Gaussian distribution at moderate to high velocity resolution and the solution obtained from the discrete Boltzmann equation at low resolution. The simulation of the distribution of collision angles, χ, explained the effect of reduced velocity resolution on the diffusivity of a gas. Whereas in a high temperature gas with good velocity resolution the distribution of χ closely approximates the theoretical $\frac{1}{2}sin(\chi)$ value, at low temperatures many particles are scattered through either zero or 180 degrees. Such null collisions do not provide an exchange of energy or momentum between identical particles and thus allow many particles to diffuse too far through the gas before they collide effectively. The large number of null collisions result from there being few points on the small collision spheres of low speed collisions from which to choose. This paucity of points is due to the reduction of reflection symmetries available and the increased importance of degenerate symmetries on small collision spheres.

The relaxation of a bimodal distribution function that might characterize a region of a flow with extremely high shear was then discussed. Because no theoretical solution is available, the IDSMC simulation was compared to a DSMC simulation with the same initial conditions. It was found that if a gas is made of identical particles or particles of similar mass, the discrete-velocity method reproduces the continuous-velocity results accurately if moderate velocity resolution is used. If, however, the particle mass ratios are large, there are few points in relative velocity space that lie on the small collision spheres of the heavy particles, and substantially greater velocity resolution would be needed for the integer results to be acceptable. A simple correction to the model that ignores null collisions was found to improve the results but did not change the fundamental problem of the null collisions. Together, the equilibrium and relaxation problems demonstrated the effect of discretization on collisions and, thus, on the right-hand side of the discrete Boltzmann equation.

To clarify the implications of the microscopic details of discrete-velocity particle collisions and motions on the macroscopic quantities in a flow, an unsteady shock wave was simulated and profiles of the density and temperature through the shock were compared with DSMC results. A shock wave is a severe test case because it involves highly non-equilibrium flow. Other authors have shown that the DSMC method accurately reproduces experimental results obtained in shock tubes and low density wind tunnels (Pham-Van-Diep, *et al.*, 1989) when the correct inter-molecular potential is used. The present version of the DSMC was found to reproduce published *computational* profiles for a single species hard sphere gas and was thereby concluded to be a correct implementation of the method although there remains some question about the multi-species portion of the code. A more accurate representation of molecular diameters with the Variable Hard Sphere method (Bird, 1983) was superfluous to the investigation of velocity discretization and was not included. Thus, both the IDSMC and DSMC results can only be compared directly to other hard sphere computations but not to experiments.

For the shock waves, moderate velocity resolution was found to be necessary for the IDSMC to reproduce the DSMC results. Low resolution led to greater viscosity than is realistic and, thus, to thick shock waves in a single species gas. In a binary gas the velocity resolution would have to be increased over that of a single species gas by an amount approximately proportional to the species mass ratio in order to obtain the correct viscosity. The shock wave simulations demonstrated the effect of velocity resolution on the right-hand side of the Boltzmann equation.

The heat transfer problem provided insight into the effect of low velocity resolution on particle convection. It was found that decreased velocity resolution leads to lower heat transfer at a diffuse wall while, with sufficient resolution, the heat transfer is correct. The low resolution error is due to an incorrect flux of particles to (and, hence, from) the wall and can thereby imply an error in any flux related quantity, for example, pressure. Also, it is necessary to average macroscopic quantities over 10 or more lattice sites to smooth the space-velocity correlations introduced by the boundaries of a multi-speed discrete-velocity gas. During a diffuse scattering event, the reflected particle's position is rounded to the nearest lattice site. This is a good approximation provided the lattice spacing is substantially less than a mean free path. The diffuse wall model also provides a way to compute a lattice gas flow about a curved surface which would be difficult to simulate as discrete segments between lattice sites. The continuous surface can simply be treated with floating point math and appear as a uniformly diffuse surface on all but the tiny lattice spacing scale.

The final problem involved a complicated two-dimensional flow that was used both to prove the applicability of direct simulation particle methods on a multicomputer and to demonstrate that, in at least one complex flow, the unrealistic terms a lattice gas introduces into the Navier-Stokes equations are insignificant if sufficient velocity resolution is used. The simulation covered several different areas where a problem might have been expected: vorticity diffusion in compressible flow, unsteady curved compression and rarefaction waves, species diffusion, and a real flow instability. The IDSMC and DSMC produced the same results. While many possibilities were obviously not examined, enough were examined to engender confidence that, by using moderate velocity resolution, any anisotropic effects of the discretization are not important for practical problems.

Certain portions of the discrete-velocity model must be expanded to make the method more generally useful. Non-integer species mass ratios should be incorporated with the same type of velocity sub-grid described above. Internal degrees of freedom can be introduced by changing the radii of the collision spheres during some collisions and molecular chemistry can be simulated by introducing three-body collisions. A more accurate representation of molecular diameters can be incorporated with the Variable Hard Sphere model. However, in future applications in the rarefied regime the DSMC should be adequate where the IDSMC would introduce unnecessary complications. In the completely continuum regime, the unaltered IDSMC probably cannot compete with a continuum gas computation. Thus, the practical use of these integer concepts should probably be limited to transitional flow computations.

Chapter 7

Bibliography

References

[1] Athas, W. C., and Seitz, C. L., 1988, Multicomputers: Message-Passing Concurrent Computers. *Computer*, **21**, (8), Aug., 9-24.

[2] Bellomo, N., and de Socio, L. M., 1982, On the Discrete Boltzmann Equation for a Binary Gas. *Progress in Aeronautics and Astronautics - 13th International Symposium on Rarefied Gasdynamics*, July, ed. O. M. Belotserkovskii, 1269-1275.

[3] Bellomo, N., and de Socio, L. M., 1983, The Discrete Boltzmann Equation for Gas Mixtures. A Regular Space Model and a Shock Wave Problem. *Mech. Res. Comm.*, **10**, (4), 233-238.

[4] Bellomo, N., and Monaco, R., 1984, Molecular Gas Flow For Multicomponent Gas Mixtures: Some Discrete Velocity Models of the Boltzmann Equation and Applications. *German-Italian Symposium on the Applications of Mathematics in Technology*, 396-412. March 26-30, Rome, Italy.

[5] Bhatnager, P. L., Gross, E. P., and Krook, M., 1954, A Model for Collision Processes in Gases. I. Small Amplitude Processes in Charged and Neutral One-Component Systems. *Phys. Rev.*, **94**, 511-525.

[6] Bird, G. A., 1963, Approach to Translational Equilibrium in a Rigid Sphere Gas. *Phys. Fluids*, **6**, 1518.

[7] Bird, G. A., 1968, The structure of normal shock waves in a binary gas mixture. *J. Fluid Mech.*, **31**, 657-668.

[8] Bird, G. A., 1970, Aspects of the Structure of Strong Shock Waves. *Phys. Fluids*, **13**(5), 1172-1177.

[9] Bird, G. A., 1976, *Molecular Gas Dynamics*, Clarendon Press, Oxford.

[10] Bird, G. A., 1980, Monte-Carlo simulation in an engineering context. *Progress in Aeronautics and Astronautics - Twelfth International Symposium on Rarefied Gasdynamics, July 7-11, 1980*, 239-255.

[11] Bird, G. A., 1983, Definition of mean free path for real gases. *Phys. Fluids*, **26**(11), 3222-3223.

[12] Bird, G. A., 1986, Direct Simulation of Typical ATOV Entry Flows. *AIAA/ASME 4th Joint Thermophysics and Heat Transfer Conference* June 2-4, 1986, Boston, MA.

[13] Bird, G. A., 1987, Direct simulation of high-vorticity flows. *Phys. Fluids*, **30**(2), 364-366.

[14] Bird, G. A., 1988, private communication.

[15] Borgnakke, C., and Larsen, P. L., 1975, Statistical Collision Model for Monte-Carlo Simulation of Polyatomic Gas Mixture. *J. Comp. Physics*, **18**, 405-420.

[16] Broadwell, J. E., 1963, Study of Rarefied Shear Flows by the Discrete Velocity Method. *Space Technology Laboratories, Inc. report 9813-6001-RU000.*

[17] Broadwell, J. E., 1964a, Study of rarefied shear flow by the discrete velocity method. *J. Fluid Mech.*, **19**(3), 401-414.

[18] Broadwell, J. E., 1964b, Shock Structure in a Simple Discrete Velocity Gas. *Phys. Fluids*, **7**(8), 1243-1247.

[19] Caflisch, R. E., 1979, Navier-Stokes and Boltzmann Shock Profiles for a Model of Gas Dynamics. *Comm. on Pure & Appl. Math*, **32**, 521-544.

[20] Caflisch, R. E., and Papanicolaou, G., 1979, The fluid dynamic limit of a nonlinear Boltzmann equation. *Comm. on Pure & Appl. Math*, **32**, 589-616.

[21] Chandrasekhar, S., 1960, *Radiative Transfer*, Dover Publications, NY.

[22] Chapman, S., and Cowling, T., 1952, *The Mathematical Theory of Non-Uniform Gases.* Cambridge Univ. Press.

[23] Doolen, G., 1987, Lattice Gas Methods for Solving Partial Differential Equations. *Lecture at 3rd SIAM Conf. on Parallel Processing for Scientific Computing*, Los Angeles, Dec. 2.

[24] Duderstadt, J. J., and Martin, W. R., 1979, *Transport Theory*, John Wiley & Sons.

[25] Feiereisen, W. J., 1989, Three Dimensional Discrete Particle Simulation of an ATOV. *Presented at the Thermophysics Conference*, Buffalo, NY, June 12-14 1989. AIAA paper: 89-1711.

[26] Frisch, U., Hasslacher, B., Pomeau, Y., 1986, A Lattice Gas Automaton for the Navier Stokes Equation. *Phys. Rev. Letters*, **56**, (7), April, 1505-1508.

[27] Fox, G., Johnson, M., Lyzenga, G., Otto, S., Salmon, J., and Walker, D., 1988, *Solving Problems on Concurrent Processors.* Prentice-Hall, NJ.

[28] Furlani, T. R., and Lordi, J. A., 1989, A Comparison of Parallel Algorithms for the Direct Simulation Monte Carlo Method: Application to Exhaust Plume Flow Fields. *Rarefied Gas Dynamics: Theoretical and Computational Techniques.* ed. E. P. Muntz, **118**, *In Proceedings of 16th International Symposium on Rarefied Gas Dynamics*, Pasadena, CA, 1988. Progress in Aeronautics and Astronautics. 227-244.

[29] Gatignol, R., 1975, *Théorie cinétique des gaz à répartition discrète de vitesses*, Lecture Notes in Physics, **36**, Springer-Verlag, Berlin.

[30] Goldstein, D., Sturtevant, B., and Broadwell, J. E., 1989, Investigations of the Motion of Discrete-Velocity Gases. *Rarefied Gas Dynamics: Theoretical and Computational Techniques.* ed. E. P. Muntz, **118**, *In Proceedings of 16th International Symposium on Rarefied Gas Dynamics*, Pasadena, CA, 1988. Progress in Aeronautics and Astronautics. 100-117.

[31] Goldstein, D., and Sturtevant, B., 1989, Discrete Velocity Gasdynamics Simulations in a Parallel Processing Environment. *Presented at the Thermophysics Conference*, Buffalo, NY, June 12-14 1989. AIAA paper: 89-1668.

[32] Gross, E. P., 1960, Recent Investigations of the Boltzmann Equation. *Rarefied Gas Dynamics: Proceedings of the First International Symposium held at Nice*, Pergamon Press.

[33] Guernsey, C., March 24, 1986, Application of the Direct Simulation Monte Carlo Method to Rocket Exhaust Plume Analyses. JPL Interoffice memorandum.

[34] Hardy, J., de Pazzis, O., and Pomeau, Y., 1973, Time evolution of a two-dimensional model system. Invariant states and time correlation functions. *J.*

Math. Phys., **14**(12), 1746-1759.

[35] Hardy, J., de Pazzis, O., and Pomeau, Y., 1976, Molecular dynamics of a classical lattice gas: Transport properties and time correlation functions. *Phys. Rev.*, **13** (5) 1949-1961.

[36] Henon, M., 1987a, Isometric Collision Rules for the Four-Dimensional FCHC Lattice Gas. *Complex Systems*, **1**, (3), 475-494.

[37] Henon, M., 1987b, Viscosity of a Lattice Gas. *Complex Systems*, **1**, (4), 763-789.

[38] Hermina, W. L., 1986, Monte-Carlo Simulation of High Altitude Rocket Plumes with Nonequilibrium Molecular Energy Exchange. *AIAA/ASME 4th Joint Thermophysics and Heat Transfer Conf.*, Boston, MA. June. 1-17.

[39] Hermina, W. L., 1988, Direct Simulation Monte-Carlo Model for Space Vehicle Plume Environments. *Sandia Report* **SAND88-8827**, 1-30.

[40] Hilts, P., 1985, Discovery in Flow Dynamics Might Aid Car, Plane Design. *Washington Post*, Nov. 19.

[41] d'Humieres, D., Lallemand, P., and Shimomura, T., 1985, Lattice Gas Cellular Automata, A New Experimental Tool For Hydrodynamics. LA-UR preprint, 85-4051, submitted to *Phys. Rev. Let.*.

[42] Inamuro, T., 1988-1989a, private communication.

[43] Inamuro, T., 1989b, Numerical Studies on Evaporation and Deposition of a Rarefied Gas in a Closed Chamber. *Rarefied Gas Dynamics: Physical Phenomena.* ed. E. P. Muntz, **117**, *In Proceedings of 16th International Symposium on Rarefied Gas Dynamics*, Pasadena, CA, 1988. Progress in Aeronautics and Astronautics. 418-433.

[44] Kadanoff, L. P., McNamera, G., and Zanetti, G., From Automata to Fluid Flow: Comparisons of Simulation and Theory.

[45] Kawasaki, K., 1985, Monte Carlo Calculation of the Flow of Granular Materials. Engineering thesis, Calif. Inst. of Tech..

[46] Lees, L., 1959, A kinetic theory description of rarefied gas flows. *GALCIT Hypersonic Research Project* **15**.

[47] Long, L. N., Coopersmith, R. M., McLachlan, B. G., 1987, Cellular Automatons Applied to Gas Dynamic Problems. *Presented at AIAA 19th Fluid Dynamics, Plasma Dynamics and Lasers Conference*, June 8-10, Honolulu, Hawaii.

[48] McDonald, J. D., and Baganoff, D., 1988, Vectorization of a Particle Simulation Method for Hypersonic Rarefied Flow. *AIAA Thermophysics conf.*, San Antonio, TX. June. AIAA paper: 88-2735.

[49] McNamera, G. R., and Zanetti, G., 1988, Using the Boltzmann Equation to Simulate Lattice Gas Automata. *Phys. Rev. Let.*, **61**, Nov. 14, 2332-2335.

[50] Meiburg, E., 1986, Comparison of the molecular dynamics method and the direct simulation Monte Carlo technique for flows around simple geometries. *Phys. Fluids*, **29**(10), 3107-3113.

[51] Nadiga, B. T., 1988-1989a, private communication.

[52] Nadiga, B. T., Broadwell, J.E., and Sturtevant, B., 1989, Study of a Multi-Speed Cellular Automaton. *Rarefied Gas Dynamics: Theoretical and Computational Techniques.* ed. E. P. Muntz, **118**, *In Proceedings of 16th International Symposium on Rarefied Gas Dynamics*, Pasadena, CA, 1988. Progress in Aeronautics and Astronautics. 155-170.

[53] Nordsieck, A., and Hicks, B., 1967, Monte Carlo Evaluation of the Boltzmann Collision Integral. *In Proceedings of 5th Symposium on Rarefied Gas Dynamics*, Oxford, England. *Rarefied Gas Dynamics*, ed. C. L. Brundin, Academic Press, 695-710.

[54] Papamoschou, D., 1986, Experimental Investigation of Heterogeneous Compressible Shear Layers. PhD thesis presented at Calif. Inst. of Tech..

[55] Pham-Van-Diep, G., Erwin, D., and Muntz, E. P., 1989, Nonequilibrium Molecular Motion in a Hypersonic Shock Wave. *Science*, **245**, Aug., 624-626.

[56] Platkowski, T., and Illner, R., 1988, Discrete Velocity Models of the Boltzmann Equation: A Survey of the Mathematical Aspects of the Theory. *SIAM Review*, **30**, (2), 213-255.

[57] Sone, Y., Ohwada, T., and Aoki, K., 1989, Heat Transfer and Temperature Distribution in a Rarefied Gas Between Two Parallel Plates: Numerical Analysis of the Boltzmann Equation for a Hard Sphere Molecule. *Rarefied Gas Dynamics: Theoretical and Computational Techniques*. ed. E. P. Muntz, **118**, *In Proceedings of 16th International Symposium on Rarefied Gas Dynamics*, Pasadena, CA, 1988. Progress in Aeronautics and Astronautics, 70-81.

[58] Su, W., Faucette, R., and Seitz, C., 1985, C Programmers Guide to the COSMIC CUBE. Computer Science dept. technical report 5203:TR:85, Calif. Inst. Tech..

[59] Usami, M., Fujimoto, T., and Kato, S., 1989, Monte Carlo Simulation on Mass-Flow Reduction due to Roughness of a Slit Surface. *Rarefied Gas Dynamics: Space-Related Studies*. ed. E. P. Muntz, **116**, *In Proceedings of 16th Interna-*

tional Symposium on Rarefied Gas Dynamics, Pasadena, CA, 1988. Progress in Aeronautics and Astronautics, 283-297.

[60] Woronowicz, M. S., and McDonald, J. D., 1989, Application of a Vectorized Particle Simulation in High-Speed Near-Continuum Flow. *Presented at the Thermophysics conference*, Buffalo, NY, June 12-14. AIAA paper: 89-1665.

[61] Zanetti, G., 1988a, The Hydrodynamics of Lattice Gas Automata. PhD thesis, University of Chicago dept. of Physics.

[62] Zanetti, G., 1988b, The Macroscopic Behavior of Lattice Gas Automata.

Appendix A

DSMC/IDSMC Program User's Guide

This is intended to be a discussion of the programs used to simulate rarefied gas-dynamics on the Intel iPSC/1 Hypercube computer at Caltech. The description should be thorough enough to enable a new user, who is familiar with the C programming language, the general layout of a distributed memory message passing computer, and DSMC computations, to run, and perhaps modify, the basic programs. All programs were meant to be run under the Cosmic Environment for the iPSC/1 computer, although they can utilize the Reactive Kernel as well.

A.1 Overall Program Outline

There are separate programs residing on the host computer (generally a SUN 3/60) and the cube. The host acts essentially as a manager while the majority of the simulation is computed by the cube. Within the cube the flow field is divided among the processors; each receives a rectangular sub-domain. Each processor then divides its sub-domain among equally sized computational cells around which the Monte Carlo simulation is based. The particles in a node collide within these cells and move between cells.

At the end of each time step those particles that pass beyond the bounds of their present processor are sent as messages to neighboring processors. Periodically,

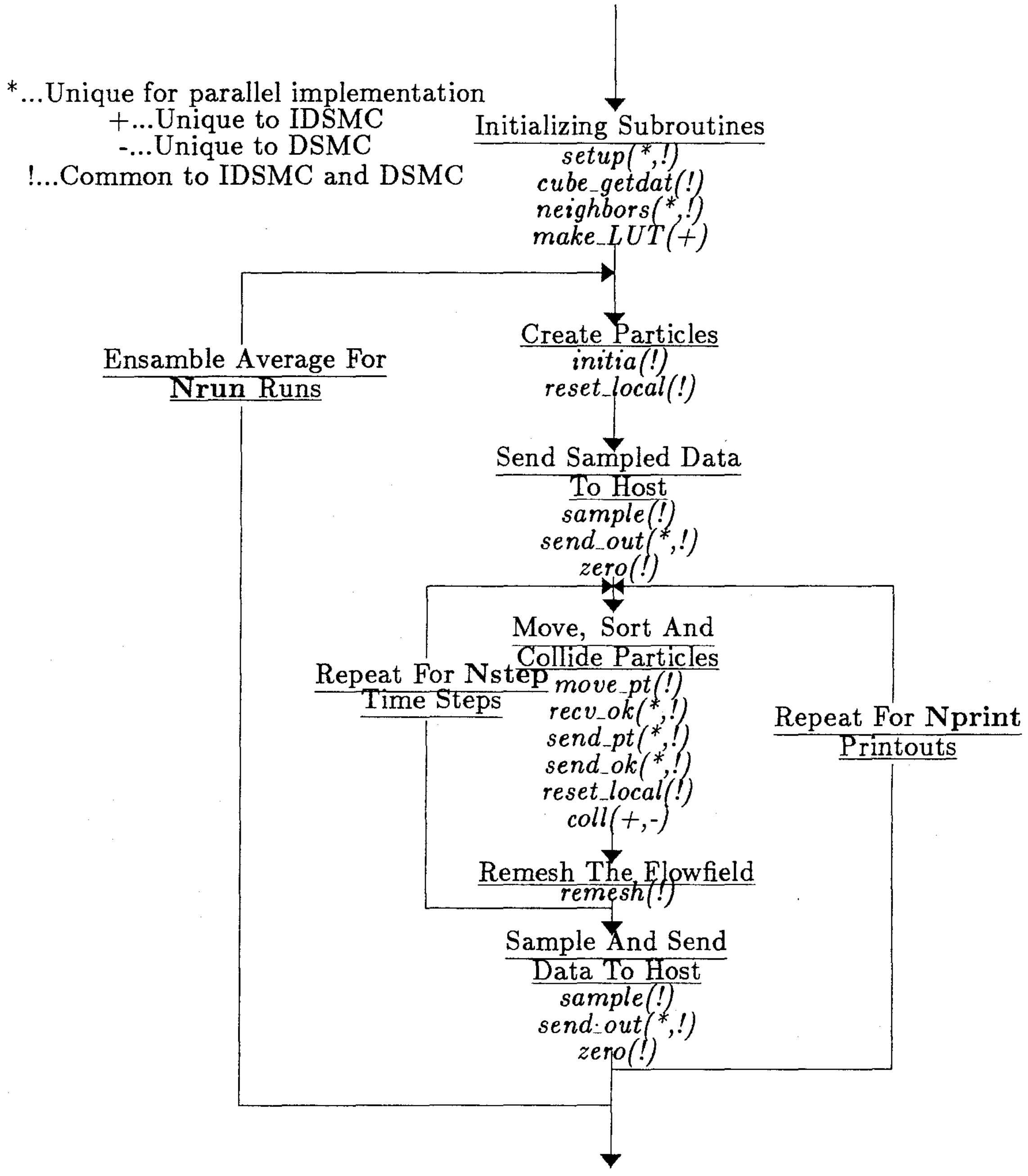

Figure A.1: Flow chart for program in each node. Subroutine names appear in *italics* and variable names in **bold.**

the boundaries separating the processors are moved in order to balance some quantity between the processors, such as the number of particles or the computational time. The boundaries, however, remain regular so that nodes and cells always are aligned in rectilinear rows and columns.

The code is the same for both the DSMC and IDSMC methods except for the collision subroutine. In fact, both simulations begin with exactly the same initial conditions of particles having integer velocities and residing at lattice points in space. In the DSMC simulation the particles quickly collide and take on a continuum of velocities and positions.

In figure A.1 the overall organization of the cube program is outlined. As in the remainder of the discussion, the program/subroutine names are italicized and variable/parameter names are given in boldface.

Upon receiving the start-up data from the host, the cube program calls some initializing subroutines. Results from separate runs are ensemble averaged to increase the number of particles per cell and reduce the noise. At the beginning of each run the cube program creates particles and sends macroscopic flow field data to the host for storage. It then moves the particles, sorts them into the appropriate nodes and cells and collides them. Periodically, the cube remeshes itself and, more infrequently, the host creates or retrieves a mesh for the cube before the cube sends out the next set of flow field data.

What follows are details of the host and cube programs and the important subroutines they contain.

A.2 The Host Program

The host program writes and reads data to and from disk; the cube program only prints out an occasional status message and passes results to the host. The host program begins by reading from disk the flow field parameters and passing them to the cube. Periodically, the cube and the host interact in order to pass out the results, which are then saved on disk in files called *safestep* and *safemesh.* Between those times, however, the host remains idle. Because results are saved periodically, the run can be terminated at almost any time and another run begun with only a minimal loss of information. Not every particle position and velocity can be saved as there is insufficient disk space, but the macroscopic quantities of each cell are retained. Thus, the program must be restarted from time zero.

A.2.1 Beginning the Process

Let us proceed through the host program (appendix B) and describe the function of each portion. The main driving program is called *hostmain.* All communication with the cube and all subroutine calls occur from here. Nearly all of the global variables are defined in files *host-2d.h* and *2d.def,* which are included at the top of *hostmain* and each subroutine. At the beginning of *hostmain* the message descriptors are defined that are later used to send and receive messages from the cube. Like all variables used throughout the code, these descriptors should have names that are indicative of what they do or contain. *Host-main*'s first subroutine calls initialize some variables, read the parameters from the *dat* file and create the first mesh. These data are converted to the cube format (Su, 1985) and sent as a message to node zero, which passes them on to the other nodes. The data are then converted back to the host compatible format.

A.2.2 Receiving Data from the Cube

The run can now begin in earnest. If this is a restart of an earlier simulation, the previous run would have already saved some data and some meshes on disk and these must now be read so that the position of a cell in physical space is the same from run to run. If this is the first run, a new mesh is generated. The correct mesh is sent to the cube. The host then sends a 'please send your data' signal to node zero and waits for node zero to respond. As a safety feature, the host program will only wait for one hour before it gives up and assumes something went wrong in the cube. If it receives data before the hour is up, the host converts the data to host format and continues to ask the other nodes, in sequence, for their data. When all of the data for time zero have been received the host accumulates them with the old data (if they exist) in the subroutine *acum* and then writes them to disk in subroutine *write-safety*.

The host has now completed the first cycle. However, a conflict arises. The host must send a mesh to the cube just before the sending out of data so that the accumulated data for a given time step always refer to the same mesh. Yet if the host were to create every mesh, much host/cube interaction would be required, which would be slow both due to slow communication over the ethernet and to the host itself being slow. This conflict is resolved by having node zero create the mesh every **cremesh** time steps. The mesh during each run is then necessarily different between host remeshings because the flow field is slightly different. However, just before the cube is to send out a new data set, it passes the host the information it needs to make a mesh. The host then makes a mesh if none already exists or reads an old mesh from disk. The host sends whichever mesh it chose back to the cube where each node resorts its particles onto the new mesh before it sends out the macroscopic data. Therefore, the cube generally is free to run and remesh on its own. The host's procedure is seen in the next dozen lines of code in *hostmain* where the host receives the X and Y number density profiles and the cube's most recent mesh and returns a new mesh to the cube.

The host then enters the subroutine *read-safety* where it may read in another old data set and mesh in preparation for the next time it receives output data. The program concludes when the host has received the last data set from the cube.

To summarize the functions of the subroutines called by *hostmain*:

- *Host-set* finds the characteristics of the cube with which the host is working.
- *Host-getdat* opens the *dat* file and reads in all of the flow field parameters for the run.
- *Host-dat* computes several values from those parameters.
- *Makechain* creates the first mesh, which is assumed to be uniform because the initial number density distribution is usually uniform. *Makechain* also sets the components of the accumulated output matrix to zero.
- *Grey* is used to map a two- dimensional grid in physical space onto the hypercube computer architecture so that nearest neighbor sub-domains of physical space are also nearest neighbor processors in the hypercube. This feature is used to reduce the path length traveled by messages.
- *(Host)mkmesh* is the host-based program that creates a new mesh. It is nearly the same as the cube-based *(cube)mkmesh,* which is discussed below.

A.3 The Cube Program

A.3.1 Beginning a Simulation

The cube's code (appendix C) contains the heart of the DSMC/IDSMC methods. As in the host's code, there is a single central driving program, *cube-main*. Most of the variables are specified as external variables and are described in *2d.h*, which is included at the beginning of *cube-main* and each subroutine. Most variables are actually defined in *2dcube*. *2d.def* defines some further quantities that are used in *2dcube* and *2d.h* to dimension arrays.

As soon as the cube program is spawned to a node, *cube-main* calls the subroutine *setup* that receives the startup data sent from either the host or a lower numbered node, and passes the data on to the current node plus one. *Setup* then calculates the same quantities as in *host-set* (above) as well as each node's location in the grid of nodes and how many nearest neighbor nodes each has. *Cube-main* then calls *cube-getdat*, which calculates several variables (as in *host-dat*, above), places the host- generated mesh into the cube mesh, and computes the location of the cells in the node. *Neighbors*, the next subroutine called, creates a small vector containing the node numbers of the eight or so adjacent nodes.

Cube-main next initializes the three random number generators. *Rand* provides a floating point random number between 0 and 1. *Ranbits* supplies as many random bits as are requested. *Ran-int-le* provides a random integer less than or equal to what is specified and greater than or equal to zero. If the IDSMC is being used, *cube-main* will also call *make-LUT* to create the relative velocity look-up table.

A.3.2 The Basic Direct Simulation Algorithm

With the preliminary computation complete, the runs can begin in the cube. First, new particles are created in *initia* based on the specified initial conditions. The look up table is then re-ordered in *scramble()* to insure randomness. Each particle is sorted into the correct cell in *reset-local. Send-ok* tells neighboring nodes that the current node is ready to receive particle message packets. The initial macroscopic cell quantities are found in *sample* and are sent out to the host in *send-out. Zero* resets the array used to send out the macroscopic data. All of these subroutines, used only once at the beginning of each run, do not take much time.

The subsequent subroutines, however, are executed during every time step and consume most of the computational effort. The sequence of events in each processor is as follows: move all particles according to their present velocity components and account for particle/boundary interactions in *move-pt.* If all of the nearest neighbor processors are ready (checked in *recv-ok*), send them (in *send-pt*) any particles that have moved beyond the sub-domain covered by the present node. Receive any particles sent from other nodes in *recv-pt.* Send a 'ready to receive again' signal to the neighbors in *send-ok.* Reset all particle indices in *reset-local* to account for the newly sent or received particles. In *coll* perform all particle collisions. Finally, every **cremesh** time steps, or just before the macroscopic data is to be sent out to the host, create a new mesh with *cube-remesh.* Every **nstep** time steps accumulate the data and send it out to the host with the subroutines *sample, send-out,* and *zero.*

This completes the description of the startup and move/collide cycles inherent to the direct simulation method. The description does, however, gloss over a fair amount of complexity in the individual subroutines, several of which are now discussed more fully in the order in which they are first called.

A.3.3 Details of Several Subroutines

- *Make-LUT().* *Make-LUT* is only called if the IDSMC method is being run and thus it is attached to the end of the IDSMC version of *coll.* *Make-LUT* creates the relative velocity look-up table for the points in the first octant of a sphere $(u, v, w \geq 0)$. *Make-LUT* first calculates the number of points on each sphere within the octant and stores those values in the vector **Enum**[]. It then allocates memory for a pointer to all points on each sphere, accounting for the fact that some spheres have fewer points than others and, thus, require less memory. It organizes those points such that consecutive points on each sphere in the table are placed in 1 group or 3, according to their parity. (See the description of *coll* below for further detail.) Finally, by calling *scramble()*, it scrambles the table while maintaining the parity ordering. *Scramble()* insures that any bias created during the construction of the table is removed.

- *Initia.* *Initia* is called at the beginning of every run to create new particles. The initial temperature and, thus, the velocity resolution of the gas is found from the parameters of the specified flow: the mean free path, time step, and species mass ratios and concentrations. In order that flows computed with the DSMC and IDSMC are directly comparable, exactly the same initial particles are created in both types of simulation. Hence, *initia* next computes the equilibrium integer velocity distribution function for each species and stores it in **temp-array**[][] as an array of integers that are scaled to 2^{13} (for moderate resolution but saving storage space). The width of each distribution function is also found and stored as **temp-width**[]. *Initia* then over-writes the node's mesh with the mesh received from the host. In a somewhat convoluted manner, to account for a possibly non-uniform initial density distribution, the number of lattice sites and particles in each cell are found. The cell time counters and reference maximum relative velocities are set. In every cell each particle is then placed individually at a lattice site, given a mass, (either 1 or **mass-h**, with a probability proportional to **perc-h**, the percentage of heavy particles), and assigned an integer velocity drawn by acceptance/rejection from the distributions in **temp-array**[][].

- *Move-pt*. *Move-pt* is called during every time step to move the particles through space with their current velocity. It is only in this subroutine that particle/boundary interactions occur. Although the particles may obtain positions beyond the limits of their present processor, they are not sent to the other processors until *cube-main* calls *send-pt*.

  In *move-pt*, after the locations of the left-hand wall (the piston) and any obstacle are determined, particles are individually translated. If the boundaries are to be specular the process jumps far down in the subroutine to a portion termed "simple step." There, a particle's position is updated and any specular collisions with a horizontal boundary are performed by reflecting the particle position and reversing the normal velocity component. Then, if the particle's path has carried it through any of the (always specular) vertical surfaces, it is again reflected. To conserve momentum, twice the piston velocity is added to a particle reflected from the piston face.

  If, however, the horizontal boundaries are to be diffuse, particle movements become complicated. In this case, the particle's initial position and velocity are saved in temporary variables and the time until the first impact on either the top or bottom surface is found. If that time is greater than the time step then the particle will not impact at this time and the process jumps down to "simple step." Otherwise, a collision will occur shortly. In that case, the particle is moved to the surface and the time remaining to that particle's time step is found. If the particle had been moving up and is thus sitting on the upper surface, three components of its emitted velocity are found (via acceptance/rejection) from the distribution functions saved in **temp-array**[][]. The normal $(-v)$ velocity component is taken from a different distribution than the parallel components. If the particle sits on the lower surface, its emitted v velocity is similarly positive. If the simulation is with the DSMC method, the particle velocities are drawn from the correct continuous velocity distributions. Finally, the code drops down below "simple step" where the particle is moved off the surface a distance corresponding to its newly assigned velocity and the time remaining in the time step. If this is an IDSMC simulation, the new position is rounded to the nearest lattice site. The particle may then be specularly scattered off the vertical surfaces if it is so destined.

- *Send-pt. Send-pt* sends particles that have been moved beyond the bounds of their present processor to neighboring processors. Generally, a processor covers a sub-domain several mean free paths in each dimension so only a small percentage of particles will leave a processor during each time step. In order to reduce the time required to send particles, *send-pt* has become fairly complicated. It begins by checking whether a particle is to be sent to another processor either to the right or the left. If so, and if the flow field is "flow through" and the particle has passed beyond the flow field limits, the particle's position is corrected by adding or subtracting the flow field width. The X location of the processor to receive the particle is then found by checking one column of nodes to the right, then one column to the left, then two columns to the right, and so on. The procedure in the vertical direction is similar except that the top and bottom boundaries are never "flow through."

If the particle is to be sent from the processor, it is copied into a particle sending structure and removed from the sending node's inventory before its departure. It can be sent either as a single particle or as a member of a group. If the receiving node is not one of the eight or so nearest neighbor nodes, the particle is sent alone with a flag indicating that it came from a distant node. The occurrence of such a very fast particle should be extremely rare if the processors cover a sufficiently large spatial domain. The vast majority of the particles are only sent to nearest neighbor nodes. If the particle sending structure is filled with PTSENT particles, it is immediately sent with a flag indicating that it is not the last such structure of particles to be sent in this time step. Particles are sent in such groups to reduce the total number of messages that must be sent (saving time) and to reduce the size of the message queue that would have to be allocated to allow for the large memory overhead associated with sending many small messages. Finally, after all particles have been placed into sending structures and/or sent, a final structure is sent to all nearest neighbors with a flag informing them of this fact. Distant nodes are not so informed, however.

- *Recv-pt*. *Recv-pt* is the subroutine corresponding to *send-pt* that receives incoming particle packets. *Recv-pt* continues to receive particles until it receives a signal from each of the nearest neighbor nodes that it has sent all particles appropriate to that time step. Upon receiving a particle packet, *recv-pt* checks the flag to see if the packet is from a distant node or a neighbor which will still be sending other packets. In either case it does not count the packet toward fulfilling the signal that all packets have been received. It can happen that a particle is received from a distant node but that the receiving node boundaries have moved beyond the particle position since it was sent. This is extremely rare and the particle is totally discarded. Normal packets are unpacked into individual particles and the total number of particles in the node is incremented.

- *Coll.* *Coll* is the subroutine where all binary particle collisions occur. It is two separate subroutines, in fact, for the DSMC and IDSMC implementations. The correct version is simply swapped into place before the program is compiled. The beginnings of both versions are similar and are discussed together.

  Both versions compute collisions in those cells in which there are two or more particles of the appropriate mass. Collisions are performed separately for each of the four collision types in a binary gas mixture. Two different particles are chosen randomly from among the particles of the correct type within the cell. The particles' relative velocity components and the sum of the squares of the components is found and the collision cross section computed. An account is kept of the highest relative speed found in the last 20 collision attempts and, every 20 collisions, the cell's maximum relative particle speed is decremented by an amount that is half the difference between its present value and the maximum within the last twenty collisions. This procedure compensates for the situation where the fluid within a given cell may cool with time and the actual maximum relative velocity decline appreciably. It also slowly dissipates the effect of a rare, extraordinarily fast collision. In every collision attempt, however, the cell's maximum relative speed is updated if it is less than the collision value. The particles chosen are then rejected for collision if they lie on opposite sides of a flow field obstacle or if the square of their relative speed, normalized by the cell's maximum relative speed squared, is not sufficiently high when compared to the square of a random number. The relative speed itself is not used because it saves time to delay calculating the costly square root until after all possible rejections have occurred. Once a pair has been accepted for collision the appropriate cell time counter and the total collision counter are incremented.

If the particles have a different mass *coll* calls a subroutine, *diff-species-collision()*, to perform the collision. The DSMC and IDSMC versions perform the actual collisions differently both for the same and different species particles. These procedures are discussed separately below.

In the DSMC version the collision is performed by the method Bird (1976) describes. The impact parameter and scattering azimuth angle are chosen randomly. With these, the post-collision relative velocity is computed. Together, these operations are the slow portion of the DSMC collision as they contain four slow function calls: rand(), sqrt(), sin(), and cos(). Next, the center of mass velocity is found and, together with the relative velocity, the post collision particle velocities are finally computed. If the particles had different masses the operations would have been similar except that the computation of the center of mass velocity and the final particle velocities would utilize conservation of momentum to find the correct values.

In the IDSMC version the actual collision is computed very differently. When the particles are identical, *coll* first computes twice the center of mass velocity,

$(u_{2cmv}, v_{2cmv}, w_{2cmv})$, which is an integer vector. The parity of the vector is found and is defined as **par**:

par $= 0$ if u_{2cmv} and v_{2cmv} have the same parity ((o,o) or (e,e)) but w_{2cmv} is different,

par $= 1$ if u_{2cmv} and w_{2cmv} have the same parity but v_{2cmv} is different,

par $= 2$ if w_{2cmv} and v_{2cmv} have the same parity but u_{2cmv} is different,

par $= 3$ if u_{2cmv}, v_{2cmv} and w_{2cmv} have the same parity.

The value of the vectors' parity indicates how far over in the look up table the list of the valid post collision relative velocities begins. If **par** $= 3$, all entries are valid. If **par** $= 0$, only the first third of the entries are valid. If **par** $= 1$, only the second third of the entries are valid and thus the list reference pointer is shifted over so that the point chosen will come from the middle third of the look up table. If **par** $= 2$, only the last third is valid and the list reference pointer is shifted two thirds of the way over. To save memory, only points in one octant of a sphere are stored in the look up table. Thus, to obtain points uniformly distributed through all octants, the point chosen is reflected randomly across the planes u_{2cmv}, v_{2cmv}, and $w_{2cmv} = 0$.

The above look up table procedure is equivalent to the slow portion of the DSMC collision process. Once the relative velocity vector is obtained, the post collision particle velocities are easily found from the center of mass velocity.

If the particles have a different mass, *diff-species-collision()* is called. All points on the sphere of the heavy particle that lie on the sub-grid are considered possible candidates for the post-collision relative velocity. Some, and perhaps many, of the points must be rejected, however, because while they are points on the sphere on a sub-grid they do not lie on integer points on the regular velocity grid. A point is randomly chosen and the rejection process is carried out efficiently as follows: Each velocity component is considered independently in turn. The u component is reflected randomly across the $u = 0$ plane. If it lies on an integer location on the main grid, it is accepted. The test for its acceptability includes a modulus check, which is computed once and stored in a small vector. If u fails the test, it is reflected across the $u = 0$ plane (to become $-u$) and is again tested. If it passes, it is accepted, but if it again fails, the whole point, (u, v, w), is rejected and a new point chosen. When each component individually passes the test the whole point is finally accepted. The ratio of accepted to rejected points depends on the mass ratio of the particles involved. Again, once the relative velocity is obtained, the post collision particle velocities are found simply from conservation of momentum. The long division that is required is computed once and stored in another small vector.

In both the DSMC and IDSMC versions of *coll*, the collision process will continue until the cell time in each cell in the node is brought up to the overall flow time.

- *Cube-remesh.* *Cube-remesh* is called from *cube-main* to remesh the grid of nodes every **cremesh** time steps or just before the host is to be sent the macroscopic data. *Cube-remesh* begins by calling subroutine *send-out-scale*, which accumulates the density profiles into vectors called **xscale**[] and **yscale**[] and sends them to node zero. The other nodes wait for node zero to send back the new mesh. Upon receiving the mesh, each node computes the new size and locations of its cells and then goes through the subroutine sequence *recv-ok*, *send-pt*, *recv-pt*, *send-ok*, and *reset-local* to resort the particles onto the new mesh. This sequence is essentially the same as that carried out by *cube-main* after each movement of the particles.

- *Send-out-scale. Send-out-scale* is a complicated subroutine designed to efficiently accumulate the number density profiles into node zero. It is worth noting here that if another positive quantity, such as the number density of a single species or the particle temperature, is substituted for the number density in *send-out-scale*, the mesh created will be such as to balance that quantity between the nodes. It is assumed that there is a rectangular field of nodes. Each node begins by summing all of the particles in each row and column of cells it contains into the two vectors **myxscale**[] and **myyscale**[], which span the width and height of the entire field of cells. If a node is in the bottom row it then accumulates its **myscale**[] data into the **scale**[] sending structures and passes the structures to the node above. The other nodes wait for that data, accumulate it with their own data, and again pass the combination on to the node above. Nodes in the top row wait for data from the nodes below and to the right and then pass their accumulated data to the node to their left. Ultimately, having received data from the node below and the node to the right, node zero adds in its data and the flow field profiles are complete. The process of accumulating the data is thus performed with moderate parallelism. If the host is to create the mesh, node zero sends the host the profiles and the present mesh and awaits the new mesh. Otherwise, node zero calls *cube-mkmesh* to create the mesh. In either case, node zero finally distributes the new mesh to all the other nodes before returning from *send-out-scale*.

- *Cube-mkmesh. Cube-mkmesh* is the subroutine that actually scans the accumulated profiles to create the new mesh. It begins by calculating where the left-hand flow field boundary, the piston, lies. It then sums all of the entries in the **xscale**[] profile to find the total number of particles (or whatever the profile contains) in the flow field and the number of particles to be placed in each column of nodes. Then, **xscale**[] is scanned to find the first non-zero entry, which should occur at a location coinciding with the most recent position of the piston. The left-most new node boundary, **local-xmark**[**0**], is always placed at the present location of the piston. *Cube-mkmesh* then begins summing particles (into **sum**) for the new first column of nodes by scanning along **xscale**[] until it can proceed no further without **sum** overfilling that column. The right-hand boundary for the column is then found by interpolating between the boundaries of the column of cells which would cause an overflow. **Sum** is corrected and the next column of nodes is considered. No entries in **xscale**[] beyond the first few should be zero and the right-hand side of the flow field should not be reached before all particles have been accounted for. The right-hand boundary of the right-most column of nodes is always placed exactly on the flow field boundary.

The mesh in the X direction is smoothed **hremesh** times. Smoothing consists simply of averaging the width of each column of nodes with that of its neighboring columns. Smoothing is necessary to alleviate an oscillation that was found to occur behind strong unsteady shocks which pass from regions with small cells into regions with large cells. In a smoothed mesh the cell size changes gradually through the shock, but there can be a memory and work load imbalance between processors because a smoothed mesh will not exactly reflect the actual number density profiles. A smoothed node just downstream of a shock may cover too large of a domain and become flooded with particles.

The procedure for remeshing in the vertical direction is the same as in the horizontal direction except that the top and bottom flow boundaries remain fixed and there is no need to search for the first non-empty row of cells (all should contain particles). Also, the mesh is presently not smoothed in the Y direction.

A.4 Running the Code

That completes the description of the different subroutines. This guide concludes with a discussion of how to actually run the program using the IDSMC method to compute a shock wave passing over an obstacle.

A.4.1 Determining the Values in *2d.def*

First, it is necessary to put the IDSMC version of the collision routine in place of *coll.* Then, set the dimensions of the arrays to be used as seen in *2d.def* in appendix B.

The values shown are as they would appear in the file. PTSENT is sized such that each particle package is of a length corresponding the maximum message length that can be sent as a single unit in the Intel iPSC/1 (about 31 for the DSMC and 60 to 70 for the IDSMC). NXCELL and NYCELL are the number of cells in the X and Y directions, respectively, in each node. NM is the maximum number of particles permitted in each node and must be reduced if the DSMC version is used, because each particle would consume more space. NNODES is the number of nodes to be used and NDXMAX and NDYMAX are the number of nodes in the X and Y directions. LATICE and VTYPE are set to 1 as flags indicating that the particles are to remain on a lattice and that they have integer velocities. It is possible to run the program with the particles having integer velocities but not traveling on the spatial lattice. In that case, their positions must be defined as floats and they may be created in *initia* at any point in space. A gas constant, GAMMA, of 1.667 corresponds to a gas of hard smooth spheres having three (VELDIM) translational degrees of freedom. Setting the maximum look up table sphere radius, LUTRAD, as 38 is adequate for a moderate velocity resolution simulation but it should be larger for higher resolution or some high speed collisions will exceed the size of the table. Remember, however, that the memory consumed by the table varies as the cube of the sphere radius. The three typedef lines are used to define what variable type are the velocities, positions, and times.

A.4.2 Determining the Values in *dat*

2d.def must be complete before either the host or the cube programs are compiled. The flow field parameters in the *dat* file (also in appendix B), however, are read by the host program during the run and may be changed at any time prior to that.

In the *dat* file each variable name is placed directly below its value. The whole calculation is repeated for **runmax** runs to obtain a smooth ensemble averaged answer. **Nprint** is the number of times during each run that data is to be saved on disk. Each file may be long (10 to 10000 kbytes) so there must be sufficient disk space available. **Nstep** is the number of time steps between each incidence when data is so saved. **Ncx** and **ncy** are the number of cells in the X and Y directions in each node, respectively, and must be less than or equal to NDXMAX and NDYMAX. **Mc** is the initial number of particles in each cell. It should be large enough that the more rarefied species (if there is more than one) always has at least 20 or so particles per cell to ensure that the time counter is incremented properly. It should also be small enough so that at no time will any node have to contain more than NM particles. In considering this second qualification, the anticipated flow field, remeshing, and statistical fluctuations must all be considered as potential causes of a node becoming swamped. **Seed** is the random number generator seed and should be a moderately sized integer.

Xm-glob and **ym-glob** are the physical right and top boundaries of the flow field. They should be half integers for the IDSMC but may be any floating point number for the DSMC. The left and lower boundaries are similarly specified as **x-left** and **y-bot**. The size of the flow field should be such that each node's domain is at least a few mean free paths in both directions so that in one time step it is very unlikely that a particle will jump completely over a node. The field should also be sized so that, together with the number of cells in each node and the number of nodes, the cells come out as a desired size for the simulation planned. Finally, for the IDSMC, the flow field must be large enough that, no matter how the cells move around and change size during remeshing, at no time will any column or row of cells contain no lattice sites. That is, the width and height of each cell must always be greater than one.

Dtm is the time step. For the DSMC method it may be a float but for a lattice gas it must be an integer (which is invariably 1). **Lam-zero** is the size of a mean free path, (the number of lattice sites per mean free path in the IDSMC), and is used to determine the initial particle velocity resolution. **Ts-ratio** is the ratio of the time step to the mean collision time in the quiescent initial fluid. It too is used to determine the initial particle velocity resolution. It should be sized so that in the hottest portion of the flow to be calculated, where the collision rate is highest, the ratio is still somewhat less than one. For example, if the temperature behind a shock is expected to be about four times the initial temperature, **ts-ratio** should be about half of the 0.5, which should be adequate in a flow with no significant temperature rise.

Tw is the temperature ratio of the top surface to the bottom surface and it may be a floating point number. **Uw** is the X velocity of the top and bottom surfaces; it should be an integer for an IDSMC simulation. **U-enter** is the speed with which the piston will move to the right. For the DSMC it may be a floating point number but for the IDSMC it should be an integer or a half-integer because the particles that rebound from the piston face do so with twice the piston velocity added to their reflected velocity and the final velocity must be an integer. **Diff** is a flag used in the subroutine *move-pt* to indicate whether the horizontal surfaces are to be specular (< 0) or diffuse (≥ 0).

Obst-ht is the height of the obstacle that is placed in the flow. It is set to zero for simple undisturbed flows and to a height less than the flow field height otherwise. The obstacle X location is given in **obst-a**, which should be a half integer for the IDSMC. The obstacle has no thickness and may lie anywhere within a cell. **Pstop** is the X location where the piston will abruptly stop moving. It should be a half integer location in the IDSMC. It must also be somewhat to the left of the obstacle position and the right hand flow boundary to avoid a catastrophic squeezing of the flow.

Hremesh is the number of times the mesh is smoothed. If **hremesh** is set to zero the mesh will not be smoothed. The more the mesh is smoothed the worse the load balancing effect of remeshing becomes. Generally, an **hremesh** value of 1 or 2 is used for high Mach number shocks to allow the mesh at some location to begin to adapt to the presence of the shock before it arrives. Without smoothing, a significant oscillation in the macroscopic quantities can occur behind the shock. **Cremesh** is the number of time steps that occur between remeshings within the cube. For dynamic flow fields it should be 1 or 2 while for more sedate flows it can be 10 or more. If remeshing is not sufficiently frequent and the flow gets far ahead of the mesh, when remeshing finally does occur, huge numbers of particles will be passed between nodes and some nodes' message queues may become saturated and the program may hang. **Time-avg** is the number of printouts after which no new *safestep* files are created and the following results are accumulated into the last such file. This allows for time averaging of an initially unsteady flow (*i.e.*, the heat transfer problem), which later becomes steady.

Mass-h is the mass of the heavy particle species. The light species is always assumed to be of mass 1. **Mass-h** can be any positive value for a DSMC simulation but must be a fairly small integer for a moderate to low velocity resolution IDSMC simulation. A safe rule is that the light gas velocity distribution function at equilibrium should be **mass-h** times as wide as what would be needed for a single species gas or there may be too few possible collisions between the light and the heavy particles. **Perc-h** is the initial percentage of heavy particles. When a particle is created, a random number is compared to **perc-h** to determine if the particle is to be heavy or light; hence, the value of **perc-h** can be any floating point value. It should not be much smaller than 0.05 or larger than .95, however, or there will be so few of the rare species that the time counter will not be incremented properly during a rare species collision. **Dia-rat** is the ratio of the diameters of the heavy to the light particles.

That concludes the description of the program. To run the compiled version simply get the proper sized hypercube with "getcube 7 ipsc cosmic," spawn the *cubemain* code with "spawn cubemain -1 0" and execute the host program with "hostmain."

Appendix B

Program For The Host

```
/* In 2d.def are defined several quantities used as flags or to
 *  dimension arrays in host_2d.h, 2d.h and 2dcube.c.
 */

#define MAXNAM   60       /* Max. number of characters in a line in dat */
#define MAXPRT   50       /* Max number of printouts per nrun * nprint  */
#define PTSENT   31       /* Max # of particles sent in a given message */
#define NXCELL   6        /* Max allowed number of x-cells per node */
#define NYCELL   6        /* Max allowed number of y-cells  per node*/
#define NM       10000    /* Max allowed number of particles per node */
                           /* -max of ~ 6420 for DSMC, 26,000 for IDSMC */
#define NNODES   128      /* Max number of nodes that will be used */
#define NDXMAX   16       /* Number of nodes in x- direction */
#define NDYMAX   8        /* Number of nodes in y- direction */
#define LATICE   1        /* Latice = 1 for pts. to remain on lattice, 0 otherwise */
#define VTYPE    1        /* Vtype is velocity type, 1 = integer, 0 = float */
#define GAMMA    1.666    /* Ratio of specific heats */
#define VELDIM   3        /* Number of random velocity components simulated */
#define LUTRAD   38       /* The max radius of LUT sphere */
typedef char veltype;
typedef int postype;
typedef int timetype;
```

```
1          10         30         6          6          150        973
runmax     nprint     nstep      ncx        ncy        mc         seed
1000.5     1000.5     1.0        10.00      0.2        1.00       1.00
xm_glob    ym_glob    dtm        lam_zer  ts_rat    lat_sp    lat_jump
0.         1.         0.         1.
uw         Tw         u_init     u_enter
-1.0
diff
400.4      500.5
obs_ht     obs_a
400.5      0.         0          2          100
pstop      temp       hremesh    cremesh    time_avg
0.5        0.5        1.         0.         1.
x_left     y_bot      mass_h     perc_h     dia_rat
```

```
/* Host_2d.h is included with the host program to define the
 *   global variables. Many of the variables have the same name
 *   as in the cube program and are not described again.
 */

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

#include "2d.def"

FILE *datptr;

short dim,size,chsize,cwsize;
long tot_field_mol;
float pi2, sq2;
float xm, ym, chx, chy, time;


struct meshlong {
/* these contain the accumulated number of particles in the rows and
 * columns of cells of the whole flow field just before re-meshing
 */
          long xscale[NDXMAX*NXCELL + 1];
          long yscale[NDYMAX*NYCELL + 1];
};
struct meshlong scale;

struct meshflt {
          float xmark[NDXMAX+1], ymark[NDYMAX+1];
/*          xmark and ymark contain the 'new' node positions the mesh should
 *        take and are sent down to the cube
 */
} mesh[MAXPRT];

struct outsht { /* Integer variables output from the cube */
          short npart[NXCELL][NYCELL];
          short npart_h[NXCELL][NYCELL];
/*          short null1; May be needed when there are odd numbers
      .   in NXCELL and NYCELL, like  5x5 but not for 2x20
*/
};

struct outflt { /* Floating pt variables output from the cube */
          float cellx[NXCELL], celly[NYCELL];
          float vx[NXCELL][NYCELL];
          float vy[NXCELL][NYCELL];
          float kinen[NXCELL][NYCELL];
          float kinenx[NXCELL][NYCELL];
          float kineny[NXCELL][NYCELL];
          float kinenz[NXCELL][NYCELL];
};

struct output {
          struct outsht s;
          struct outflt f;
} out[NNODES];


struct acoutlng {
          long npart[NXCELL][NYCELL];
          long npart_h[NXCELL][NYCELL];
};

struct acoutflt {
```

```
            float cellx[NXCELL], celly[NYCELL];
            float vx[NXCELL][NYCELL];
            float vy[NXCELL][NYCELL];
            float kinen[NXCELL][NYCELL];
            float kinenx[NXCELL][NYCELL];
            float kineny[NXCELL][NYCELL];
            float kinenz[NXCELL][NYCELL];
};

struct acoutput {  /* Accumulated variables, longs and floats */
            struct acoutlng l;
            struct acoutflt f;
} acout[NNODES];

struct heading {
            short numruns, numsteps;
} acum_header, mesh_header;

/* Input variables                  */
struct insht {
            short nprint, nstep, seed;
            short mc, runmax, ncx, ncy;
            short hremesh, cremesh, time_avg;
};

struct inflt {
            float xm_glob, ym_glob, x_left, y_bot, mass_h, perc_h, diameter_ratio;
            float Tw, uw, u_init, u_enter, mc_enter;
            float lamda_zero, ts_ratio, lattice_space, lattice_jump;
            float dtm, diff;
            float obs_height, obs_a;
            float pstop, temp;
            float nodeheight[NDYMAX + 1], nodewidth[NDXMAX + 1];
};

struct input {
              struct insht d;
              struct inflt f;
} in;
```

```
/* This is the primary host program which is responsible for: inputing the
 *   run data, sending the run data to the cube, receiving and storing the
 *   actual output data periodically, creating/recovering a new mesh and
 *   sending it down to the cube and finally printing out all the accumulated
 *   data.
 */

#include <stdio.h>
#include <signal.h>
#include <cube/cosmic.h>
#include "host_2d.h"

char dat[] = {"dat"};

alarmed() { fprintf(stderr,"Alarm clock Received\n"); exit(-1); }                 alarmed

main()                                                                                 main
{
        FILE *safestrm;
        char fname[80];
        short nrun, outnp;
        short i, rrr;
        extern short cwdim;
        MSGDESC in_d;
        MSGDESC out_ini_d;
        MSGDESC out_step_d;
        MSGDESC pls_sd;             /* Here are defined the message descriptors */
        MSGDESC new_mesh;           /*  which are used in the Cosmic system */
        MSGDESC last_cube_mesh;
        MSGDESC out_scale_rd;

        signal(SIGALRM,alarmed);

        sdesc (&in_d, 0, 0, 0, &in, sizeof(in));
        sdesc (&out_ini_d, 0, 0, 0, 0, sizeof(struct output));
        sdesc (&out_step_d, 0, 0, 0, 0, sizeof(struct output));
        sdesc (&pls_sd, 0, 0, 100, 0, 0);
        sdesc (&new_mesh, 0, 0, 12345, 0, sizeof(struct meshflt));
        sdesc (&last_cube_mesh, 0, 0, 12345, 0, sizeof(struct meshflt));
        sdesc (&out_scale_rd, 0, 0, 1717, &scale, sizeof(struct meshlong));
                /* These describe what the descriptors consist of */

        cosmic_init(HOST,0);        /* Necessary to use cosmic kernal */
        host_set();                 /* Here are calculated the cube's dimensions */
        host_getdat();              /* Now read in run data from file 'dat' */
        host_dat();                 /* Calculate some parameters from the data */
        mkchain();                  /* Create the first mesh. */
                /* Here are initialized data in the host */

        htocs(&in.d, sizeof(struct insht)/sizeof(short));
        htocf(&in.f, sizeof(struct inflt)/sizeof(float));
                        /* Convert the input data to cube format */
        sendb(&in_d);      /* Send the input data to node zero who will pass
                            *  it on to the other nodes.  */
        ctohs(&in.d, sizeof(struct insht)/sizeof(short));
        ctohf(&in.f, sizeof(struct inflt)/sizeof(float));
                        /* Convert the input data back to host format */


        /* Loop over sample sets, for runmax runs */
    for ( nrun = 0; nrun < in.d.runmax; ) {
```

```
  mkchain();                    /* Just to be sure */
  for ( rrr = 1; rrr-- && nrun < in.d.runmax; nrun++ ) {
          printf("\nRun number %hd\n", nrun); fflush(stdout);
          time = 0.;
          outnp = 0;            /* This is the output number */

                          /* If it exists, read in acum data for
                           *  first time step from the file
                           *  to accumulate new data in
                           *  addition to old data. */
          sprintf(fname, "safestep.%d",outnp);
          if((safestrm = fopen(fname,"r")) != NULL) {
                fread ((char *)&acum_header,sizeof(acum_header),1,safestrm);
                fread ((char *)acout,sizeof(acout),1,safestrm);
                fclose(safestrm);
          }
                          /* If it exists, read in mesh
                           *  data for first time step */
          mesh_header.numsteps = -1;
          sprintf(fname, "safemesh");
          if((safestrm = fopen(fname,"r")) != NULL) {
                fread ((char *)&mesh_header,sizeof(mesh_header),1,safestrm);
                fread ((char *)mesh,sizeof(mesh),1,safestrm);
                fclose(safestrm);
          }
          printf("ached.nrs=%d, mhed.numst= %d\n",
                                  acum_header.numruns, mesh_header.numsteps);
          fflush(stdout);

          /* Now, receive data at time = 0 from each node */
          for (i=0; i<NNODES; i++) {
                          pls_sd.node = i;
                          sendb(&pls_sd);              /* Send signal to node i that
                                                        *  host is ready to receive
                                                        *  data */
                          out_ini_d.type = i + 20000;
                          out_ini_d.buf = (char *)&out[i];
                          alarm(3600); recvb(&out_ini_d); alarm(0);
                                  /* In case nothing is received within
                                   *  1 hour, terminate run in failure. */
                          ctohs( &out[i].s, sizeof(struct outsht)/sizeof(short));
                          ctohf( &out[i].f, sizeof(struct outflt)/sizeof(float));
                                  /* Convert cube data to host format. */
          }
          acum();     /* Accumulate received data in acum array */
          write_safety(outnp); /* Write received results to disk */
          alarm(3600); recvb(&out_scale_rd); alarm(0);
                          /* Receive two mesh scaling vectors within an hour
                           *  or end in failure */
          ctohl(&scale,sizeof(struct meshlong)/sizeof(long));
                                  /* Convert scaling vectors to host format */
          last_cube_mesh.buf = (char *)&mesh[MAXPRT-1];
          alarm(3600); recvb(&last_cube_mesh); alarm(0);
                                          /* Receive the mesh the cube last
                                           *  generated into the last mesh[]
                                           *  space availible, always.
                                           */
          ctohf(&mesh[MAXPRT-1], sizeof(struct meshflt)/sizeof(float));
                                  /* Convert cube's mesh to host format */
          for(i=0; i<cwsize; i++)
                          in.f.nodewidth[i] = mesh[MAXPRT-1].xmark[i];
          for(i=0; i<chsize; i++)
                          in.f.nodeheight[i] = mesh[MAXPRT-1].ymark[i];
```

```
                                        /* Update nodexxx, used in host_mkmesh.c */
                        mkmesh(outnp+1);                /* Make a new mesh */
                        htocf(&mesh[1], sizeof(struct meshflt)/sizeof(float));
                                        /* Convert new mesh to cube format */
                        new_mesh.buf = (char *)&mesh[1];
                        sendb(&new_mesh);               /* Send new mesh to node zero */
                        ctohf(&mesh[1], sizeof(struct meshflt)/sizeof(float));
                                        /* Convert new mesh back to host format */
                        read_safety(outnp);             /* read in  data accumulated
                                                         * from files for safe keeping */

/* In the following section the same procedure as used above for time 0 is used to receive
 *  data from the cube, make meshes and store the data so the comments are
 *  not repeated.
 */
                        /* nprint prints per run */
                        for ( outnp = 1, outnp < (in.d.nprint + 1); outnp++ ) {
                                time = outnp * in.d.nstep * in.f.dtm;
                                /* Here, receive data at other times from each node */
                                for (i=0; i<NNODES; i++) {
                                        pls_sd.node = i;
                                        sendb(&pls_sd);
                                        out_step_d.type = i + 20000;
                                        out_step_d.buf = (char *)&out[i];
                                        alarm(3600);
                                        recvb(&out_step_d),
                                        alarm(0);
                                ctohs( &out[i].s, sizeof(struct outsht)/sizeof(short));
                                ctohf( &out[i].f, sizeof(struct outflt)/sizeof(float));
                                }
                                acum();
                                write_safety(outnp);
                                        /* Write the just accumulated results to disk */
                                if(outnp != in.d.nprint) {
                                    alarm(3600); recvb(&out_scale_rd); alarm(0);
                                    ctohl(&scale,sizeof(struct meshlong)/sizeof(long));
                                    last_cube_mesh.buf = (char *)&mesh[MAXPRT-1];
                                    if(outnp != in d.nprint) {
                                        alarm(3600);
                                        recvb(&last_cube_mesh);
                                        alarm(0),
                                    }   /* Receive two mesh scaling vectors within
                                                 * two hrs or end in failure */
                                ctohf(&mesh[MAXPRT-1],
                                        sizeof(struct meshflt)/sizeof(float));
                                for(i=0; i<cwsize, i++)
                                        in.f.nodewidth[i] = mesh[MAXPRT-1].xmark[i];
                                for(i=0; i<chsize; i++)
                                        in.f.nodeheight[i] = mesh[MAXPRT-1].ymark[i];
                                mkmesh(outnp+1),
                                htocf(&mesh[outnp+1]. sizeof(struct meshflt)/sizeof(float));
                                new_mesh.buf = (char *)&mesh[outnp+1];
                                if(outnp != in d nprint) sendb(&new_mesh);
                                ctohf(&mesh[outnp+1]. sizeof(struct meshflt)/sizeof(float));
                                read_safety(outnp),             /* read in  data accumulated
                                                                 * from files for safe keeping */
                            }
                        }
                }
        }
            cosmic_exit(0);
}
```

```
/* This subroutine calculates characteristics of the Cube to which
 * Host will send data
 */
#include <cube/cosmic.h>
#include "host_2d.h"

short chdim, cwdim;

host_set()
{
        int i, j;

        dim = cubedim();          /* Cube dimension, i.e, 64 nodes, dim = 6 */
        size = 1 << dim,
        chsize = NDYMAX;          /* cube - height - size, num of nodes */
        cwsize = NDXMAX;          /* cube - width  - size, num of nodes */

        j = NDYMAX;
        for(i=0; !(j&1); i++, j>>=1){}
                chdim = i;        /* cube - height - dimension */

        j = NDXMAX;
        for(i=0; !(j&1); i++, j>>=1){}
                cwdim = i;        /* cube - width - dimension */
```

```
/* This subroutine opens the data file and reads and writes on a output file */

#include <cube/cosmic.h>
#include "host_2d.h"

host_getdat()                                                           host_getdat
{
        char dum[MAXNAM];
        extern char dat[], printr[];

        if ( (datptr = fopen(dat, "r")) == NULL ) {
                printf( "Cant open %s\n", dat);
                exit(1);
        }
        fscanf(datptr, "%hd\t%hd\t%hd\t%hd\t%hd\t%hd\t%hd\n",
                &in.d.runmax, &in.d.nprint, &in.d.nstep,
                &in.d.ncx, &in.d.ncy, &in.d.mc, &in.d.seed);
        printf("runmax=%3hd nprint=%3hd nstep =%3hd   ",
                in.d.runmax, in.d.nprint, in.d.nstep);
        printf("ncx=%3hd   ", in.d.ncx);
        printf("ncy=%3hd   mc =%3hd   seed=%3hd\n",
                in.d.ncy, in.d.mc, in.d.seed);
        fgets(dum, MAXNAM, datptr);

        fscanf(datptr, "%f\t%f\t%f\t%f\t%f\t%f\t%f\n", &in.f.xm_glob, &in.f.ym_glob,
                 &in.f.dtm,&in.f.lamda_zero,&in.f.ts_ratio,&in.f.lattice_space,&in.f.lattice_jump);
        printf("xm_glob=%5.2f ym_glob=%5.2f dtm=%5.4f\n",
                 in.f.xm_glob, in.f.ym_glob, in.f.dtm);
        printf("lamda_zero=%5.2f ts_ratio=%5.2f lattice_space=%5.2f lattice_jump=%5.2f\n",
                in.f.lamda_zero,in.f.ts_ratio,in.f.lattice_space,in.f.lattice_jump);
        fgets(dum, MAXNAM, datptr);

        fscanf(datptr, "%f\t%f\t%f\t%f\n",
                 &in.f.uw, &in.f.Tw, &in.f.u_init, &in.f.u_enter);
        printf("uw = %5.2f   Tw = %5.2f\n", in.f.uw, in.f.Tw);
        printf("u_init = %5.4f   u_enter = %5.4f\n", in.f.u_init, in.f.u_enter);
        fgets(dum, MAXNAM, datptr);

        fscanf(datptr, "%f\n", &in.f.diff);
        printf("diff = %5.2f\n", in.f.diff);

        fgets(dum, MAXNAM, datptr);
        fscanf(datptr, "%f\t%f\n", &in.f.obs_height,&in.f.obs_a);
        printf("obs_height= %3.1f obs_a=%3.1f\n", in.f.obs_height,in.f.obs_a);


        fgets(dum, MAXNAM, datptr);
        fscanf(datptr, "%f\t%f\t%hd\t%hd\t%hd\n",
                &in.f.pstop,&in.f.temp,&in.d.hremesh,&in.d.cremesh,&in.d.time_avg);
        printf("pstop = %3.1f temp = %3.3f hremesh = %d cremesh = %d time_avg = %d\n"
                in.f.pstop,in.f.temp,in.d.hremesh,in.d.cremesh,in.d.time_avg);

        fgets(dum, MAXNAM, datptr),
        fscanf(datptr, "%f\t%f\t%f\t%f\t%f\n",
                &in.f.x_left, &in.f.y_bot, &in.f.mass_h, &in.f.perc_h, &in.f.diameter_ratio);
        printf("x_left =%2.2f, y_bot =%2.2f, mass_h =%2.2f, perc_h =%2.2f diam ratio =%2.2f\n\n"
                in.f.x_left, in.f.y_bot, in.f.mass_h,in.f.perc_h, in.f.diameter_ratio);

        printf("Final number of collision times = %f\n",
                in.f.ts_ratio*(float)in.d.nstep*(float)in.d.nprint);
        fflush(stdout);
```

```
/* This subroutine calculates several parameters from data */

#include <cube/cosmic.h>
#include "host_2d.h"

host_dat()                                                     host_dat
{

          pi2  =  2  *  3.14159;
          sq2  =  sqrt(2.);
          xm   =  (in.f.xm_glob-in.f.x_left)  /  cwsize;
          ym   =  (in.f.ym_glob-in.f.y_bot)  /  chsize;
          chx  =  xm  /  in.d.ncx;
          chy  =  ym  /  in.d.ncy;
```

```
#include <cube/cosmic.h>
#include "host_2d.h"


mkchain()                                                         mkchain
{
        int i, m, j, nx, ny;


        /* Here host calculates a first mesh which is passed to the cube.
         */
        in.f.nodeheight[0] = in.f.y_bot;   /* The bottom of the field */

        for (i=0; i<=cwsize; i++)
                mesh[0].xmark[i] = in.f.nodewidth[i] = in.f.x_left + (float)i * xm;
                /* The mesh is evenly spaced in the x direction initially */

        for (i=0; i<=chsize; i++)
                mesh[0].ymark[i] = in.f.nodeheight[i] = in.f.y_bot + (float)i * ym;
                /* The mesh is evenly spaced in the y direction initially */

        /* This is to zero the acout matrix... */
        for ( i = 0; i < chsize; i++ ) {
                for ( j = 0; j < cwsize; j++ ) {
                    m = who(j,i);              /* Node I'm considering */
                    for ( nx = 0; nx < in.d.ncx; nx++ ) {
                        for ( ny = 0; ny < in.d.ncy; ny++ ) {
                                acout[m].l.npart[nx][ny] = 0.;
                                acout[m].l.npart_h[nx][ny] = 0.;
                                acout[m].f.vx[nx][ny] = 0.;
                                acout[m].f.vy[nx][ny] = 0.;
                                acout[m].f.kinen[nx][ny] = 0.;
                                acout[m].f.kinenx[nx][ny] = 0.;
                                acout[m].f.kineny[nx][ny] = 0.;
                                acout[m].f.kinenz[nx][ny] = 0.;
```

```
/* This subroutine accumulates data output from the cube every
 *  nprint time steps. These values, for a given time, are
 *  written to disk in write_safety. */

#include <cube/cosmic.h>
#include "host_2d.h"

acum()                                                              acum
{
        short i, j, m, nx, ny;

        for ( i = 0; i < chsize; i++ ) {  /* For each row of nodes... */
                for ( j = 0; j < cwsize; j++ ) {  /* For each colmn of nodes... */
                     m = who(j,i);              /* Node I'm considering */
                     for ( nx = 0; nx < in.d.ncx; nx++ ) {
                        acout[m].f.cellx[nx] = out[m].f.cellx[nx];
                        for ( ny = 0; ny < in.d.ncy; ny++ ) {
                              /* For each cell [nx][ny].... */
                           acout[m].l.npart[nx][ny] += out[m].s.npart[nx][ny];
                           acout[m].l.npart_h[nx][ny] += out[m].s.npart_h[nx][ny];
                           acout[m].f.vx[nx][ny] += out[m].f.vx[nx][ny];
                           acout[m].f.vy[nx][ny] += out[m].f.vy[nx][ny];
                           acout[m].f.kinen[nx][ny] += out[m].f.kinen[nx][ny];
                           acout[m].f.kinenx[nx][ny] += out[m].f.kinenx[nx][ny];
                           acout[m].f.kineny[nx][ny] += out[m].f.kineny[nx][ny];
                           acout[m].f.kinenz[nx][ny] += out[m].f.kinenz[nx][ny];
                           acout[m].f.celly[ny] = out[m].f.celly[ny];
```

```
/* This routine creates a new mesh based on the particle distribution
 * sent up from the cube.  This routine is nearly the same as that used
 * within the cube, cube_mkmesh.c, which is more fully commented.
 */

#include <cube/cosmic.h>
#include "host_2d.h"

float dif_local_xmark[NDXMAX+1];
float temp_xmark[NDXMAX+1];


mkmesh(outnp)                                                          mkmesh
short outnp;

{

        int i, j, k, i_begin, k_begin, dum;
        int numpercolmn, numperrow, sum;
        float cellwidth, cellheight, stuf, pistonpos;
        float const;


        if(outnp <= mesh_header.numsteps) {
            printf("In host_mkmesh, went to'leave mesh alone',ountp=%d,header=%d\n",
            outnp, mesh_header.numsteps);
            fflush(stdout);
            goto leave_mesh_alone;

                                        /* So that only one set of meshes
                                         *   is used */

        pistonpos = in.f.x_left + ((outnp-1.)*in.d.nstep +
                                        in.d.nstep) * in.f.u_enter * in.f.dtm;
        if(pistonpos > in.f.pstop) pistonpos = in.f.pstop;
        printf("piston at %f\n",pistonpos); fflush(stdout);

        tot_field_mol = 0.;
        for(i=0; i<NDXMAX*in.d.ncx; i++) tot_field_mol += scale.xscale[i];
        numpercolmn = (long)(tot_field_mol) / NDXMAX + 0.5;
                                          /* # of particles
                                        in one column of nodes */
        numperrow = (long)(tot_field_mol) / NDYMAX   + 0.5;
                                          /* # of particles
                                        in one row of nodes */


        for(i=0; i<=cwsize; i++) {  /* xscale is searched for its first non-zero
                                        member and the mesh is started just 2
                                        cells before there */
            cellwidth = (in.f.nodewidth[i+1] - in.f.nodewidth[i])/in.d.ncx;
            for(k=0; k<in.d.ncx; k++) {
                    if(scale.xscale[k + i*in.d.ncx] > 0) {
                        const = k - 2; if(const < 0.) const = 0.;
                        mesh[outnp].xmark[0] = in.f.nodewidth[i]+ const*cellwidth;
                        if(k == 0 && i != 0) mesh[outnp].xmark[0] = in.f.nodewidth[i] -
                            (in.f.nodewidth[i]-in.f.nodewidth[i-1])/in.d.ncx;
                        if(k == 0 && i == 0) mesh[outnp].xmark[0] = in.f.nodewidth[0];

                        if(mesh[outnp].xmark[0] > in.f.pstop) {
                                mesh[outnp].xmark[0] = in.f.pstop;
                                print("Mesh beyond pstop(?), k= %d, i = %d\n",k,i);
                                for(j=0; j<10; j++)
                                print("The first few xscales were, %d, %d\n",
                                                j,scale.xscale[j]);
                        }
                        mesh[outnp].xmark[0] = pistonpos;
                        i_begin = i; k_begin = k; /* these contain the
                                                     location of the first
                                                     nonzero column */
                        goto found_x_begin;
                    }
                }
            }
found_x_begin:

            sum = 0;
            j = 1;
            mesh[outnp].xmark[0] = pistonpos;
            printf("In HOST, pistonpos = %f\n",pistonpos);
            for(i=i_begin; i<cwsize; i++) {         /* now begin counting particles and
                                                       setting down a node boundary when sum
                                                       >= numpercolumn */
                cellwidth = (in.f.nodewidth[i+1] - in.f.nodewidth[i])/in.d.ncx;
                for(k=k_begin; k<in.d.ncx; k++) {
                    k_begin = 0.;
                    if(scale.xscale[k + i*in.d.ncx] == 0) {
                            mesh[outnp].xmark[j] = in.f.nodewidth[i]+ k*cellwidth;
                            printf("Scale.xscale = 0 early \n");
                            goto done_with_x;
                    }
                    if(numpercolmn >= sum + scale.xscale[k + i*in.d.ncx]) {
                            sum += scale.xscale[k + i*in.d.ncx];
                            if(i==cwsize-1 && k==in.d.ncx-1) {
                                    stuf = scale.xscale[k + i*in.d.ncx] /
                                        (float)(numpercolmn / in.d.ncx);
                                    if(stuf < 1.) stuf = 1.;
                                    mesh[outnp].xmark[j] = in.f.nodewidth[i]+
                                                (in.d.ncx-1.+stuf)*cellwidth;
                            }
                    }
                    else {
                        stuf = (float)(numpercolmn-sum)/(float)
                                        (numpercolmn / in.d.ncx);
                                                /* stuf is the linear interpol.
                                                   to the locat. of the node end */
                        mesh[outnp].xmark[j] = in.f.nodewidth[i]+
                                                ((float)k+stuf)*cellwidth;
                        if(k == k_begin)
                                if(i != 0) mesh[outnp].xmark[j] =
                                    in.f.nodewidth[i] + stuf
                                    *(in.f.nodewidth[i]-in.f.nodewidth[i-1])/in.d.ncx;
                                else   mesh[outnp].xmark[j] =
                                    in.f.nodewidth[i] + stuf
                                    *(in.f.nodewidth[i+1]-in.f.nodewidth[i])/in.d.ncx;
                        j += 1;
                        sum = scale.xscale[k+i*in.d.ncx] - (numpercolmn-sum);
                                                /* sum is reduced by the # of
                                                   parts. taken by the last node */
                    }
                }
            }
done_with_x:
            mesh[outnp].xmark[cwsize] = in.f.xm_glob;

            for(i=0; i<cwsize; i++) dif_local_xmark[i] =
                            mesh[outnp].xmark[i+1] - mesh[outnp].xmark[i];
```

```
        /* Now smooth this mesh by smoothing the areas of the nodes */

    for(j=0; j<in.d.hremesh; j++ ) {
        for(i=1; i<cwsize-1; i++) temp_xmark[i] =
            0.3333 *(dif_local_xmark[i-1]+ dif_local_xmark[i]+dif_local_xmark[i+1]);

        dif_local_xmark[0] = (dif_local_xmark[0] + dif_local_xmark[1]) * 0.5;
        dif_local_xmark[cwsize-1] =
                    (dif_local_xmark[cwsize-1] + dif_local_xmark[cwsize-2]) * 0.5;
        for(i=1; i<cwsize-1; i++) dif_local_xmark[i] = temp_xmark[i];
    }

    for(i=1; i<cwsize; i++) mesh[outnp].xmark[i] =
                    mesh[outnp].xmark[i-1] + dif_local_xmark[i-1];



                                                    /* Now the same things are done in
                                                            the Y direction */

        i_begin = k_begin = 0;

    sum = 0;
    j = 1;
    for(i=i_begin; i<chsize; i++) {
        cellheight = (in.f.nodeheight[i+1]-in.f.nodeheight[i])/in.d.ncy;
        for(k=k_begin; k<in.d.ncy; k++) {
                k_begin = 0.;
                if(scale.yscale[k + i*in.d.ncy] == 0) {
                        mesh[outnp].ymark[j] = in.f.nodeheight[i]+
                                                    k*cellheight;
                        goto done_with_y;
                }
                if(numperrow >= sum + scale.yscale[k + i*in.d.ncy]) {
                        sum += scale.yscale[k + i*in.d.ncy];
                        if(i==chsize-1 && k==in.d.ncy-1) {
                                stuf = scale.yscale[k + i*in.d.ncy] /
                                    (float)(numperrow / in.d.ncy);
                                if(stuf < 1 ) stuf = 1.;
                                mesh[outnp].ymark[j] = in.f.nodeheight[i]+
                                            (in.d.ncy-1.+stuf)*cellheight;
                        }
                }
                else {
                        stuf = (float)(numperrow-sum)/(float)
                                (numperrow / in.d.ncy);
                        mesh[outnp].ymark[j] = in.f.nodeheight[i]+
                                            ((float)k+stuf)*cellheight;
                        if(k == k_begin)
                                if(i != 0) mesh[outnp].ymark[j] =
                                  in.f.nodeheight[i] + stuf
                                  *(in.f.nodeheight[i]-
                                  in.f.nodeheight[i-1])/in.d.ncy;
                                else mesh[outnp].ymark[j] =
                                  in.f.nodeheight[i] + stuf
                                  *(in.f.nodeheight[i+1]-
                                  in.f.nodeheight[i])/in.d.ncy;
                        j += 1;
                   sum = scale.yscale[k+i*in.d.ncy] - (numperrow - sum);
                }
        }
    } .
done_with_y:
    mesh[outnp].ymark[chsize] = in.f.ym_glob;

        if(LATICE == 1) {
                for(i=0; i<=cwsize; i++)
                    if(mesh[outnp].xmark[i] ==
                      (int)mesh[outnp].xmark[i]) mesh[outnp].xmark[i] += .01;
                for(i=0; i<=chsize; i++)
                    if(mesh[outnp].ymark[i] ==
                      (int)mesh[outnp].ymark[i]) mesh[outnp].ymark[i] += 0.01;
        }
                        /* Note that the mesh is not smoothed in Y */


leave_mesh_alone:

        for(i=0; i<=cwsize; i++) {  /* nodewidth is updated with new mesh to
                                              be sent down to the cube */
                in.f.nodewidth[i] = mesh[outnp].xmark[i];
                if(i<cwsize) for(k=0; k<in.d.ncx; k++)
                        scale.xscale[k+i*in.d.ncx] = 0;
        }
        for(i=0; i<=chsize; i++) {
                in.f.nodeheight[i] = mesh[outnp].ymark[i];
                if(i<chsize) for(k=0; k<in.d.ncy; k++)
                        scale.yscale[k+i*in.d.ncy] = 0;
        }

        dum = chsize; if(cwsize > dum) dum = cwsize;
        for(i=0; i<=dum; i++) {
                if(i <= cwsize && i <= chsize)
                        printf("HOST:i = %d, xmark = %f, ymark = %f\n",
                                i,mesh[outnp].xmark[i],mesh[outnp].ymark[i]);
                else if(i > cwsize)
                        printf("HOST:i = %d, xmark = ......., ymark = %f\n",
                                i,mesh[outnp].ymark[i]);
                else if(i > chsize)
                        printf("HOST:i = %d, xmark = %f, ymark = ......\n",
                                i,mesh[outnp].xmark[i]);

        }
        fflush(stdout);
```

```
/* These two subroutines, the same in both the host and the cube, are
 *  used to find a node number from its x,y location, or visa versa.
 */

#include <cube/cosmic.h>


grey_bin(n)                    /* calculate binary code from grey code */          grey_bin
short n;
{
    int i;
    for(i = n; n >>= 1; i ^=n);
    return(i);
}

who(x,y)                      /* calculate node#=who(x,y) from node location */        who
short x;
short y;
{
    extern short cwdim;

    x=(x ^ (x >> 1)) + ( (y ^ (y >> 1)) << cwdim );
    return(x);
```

```
/* This subroutine writes or reads to/from disk the accumulated output data.
 */

#include <cube/cosmic.h>
#include "host_2d.h"

write_safety(outnp)                                              write_safety
short outnp;
{
        FILE *safestrm;
        char fname[80];
        int file_write, try_open;


              file_write = outnp;
              if(outnp >= in.d.time_avg) file_write = in.d.time_avg;
                            /* That is, accumulate all data for output numbers greater
                             *  than time_avg into the safestep file #=time_avg
                             */
              acum_header.numruns = 0;
              sprintf(fname, "safestep.%d",file_write);
              if((safestrm = fopen(fname,"r")) != NULL) {
                            fread ((char *)&acum_header,sizeof(acum_header),1,safestrm);
                            fclose(safestrm);
              }             /* Find # of runs at this time which is in header */


                            /* Put new acout data in place of old */
              try_open = 0;
open_safestep:
              sprintf(fname, "safestep.%d",file_write);
              if((safestrm = fopen(fname,"w")) == NULL) {
                            try_open += 1;
                            printf("can't open %s, have tried %d time(s) \n",
                                          fname, try_open);
                            if(try_open > 5) exit(1);
                            else {sleep(60); goto open_safestep;}
                            /* This is a safety mechanism in case system is slow */
              }
              acum_header.numruns += 1;
              fwrite((char *)&acum_header,sizeof(acum_header),1,safestrm);
              fwrite((char *)acout,sizeof(acout),1,safestrm);
              fflush(safestrm);
              fclose(safestrm);

                            /* Put new mesh data in place of old */
             try_open = 0;
             if(outnp > mesh_header.numsteps) {
                            printf("Trying to save mesh, outnp=%d, headernumstep=%d\n",
                                          outnp, mesh_header.numsteps);
                            fflush(stdout);
                            mesh_header.numsteps = outnp,
open_safemesh:
                            sprintf(fname, "safemesh");
                            if((safestrm = fopen(fname,"w")) == NULL) {
                                          try_open += 1;
                                          printf("can't open %s, have tried %d time(s) \n",
                                                        fname, try_open);
                                          if(try_open > 5) exit(1);
                                          else {sleep(60); goto open_safemesh;}
                                          /* This is a safety mechanism in case system is slow */
                            }
                            fwrite((char *)&mesh_header,sizeof(mesh_header),1,safestrm);
                            fwrite((char *)mesh,sizeof(mesh),1,safestrm);
```

```
                fflush(safestrm);
                fclose(safestrm);
        }


}

read_safety(outnp)                                                    read_safety
short outnp;
{
        FILE *safestrm;
        char fname[80];
        int i, j, nx, ny, m;
        int file_read;


        file_read = outnp+1; if(outnp+1 >= in.d.time_avg) file_read = in.d.time_avg;
        /* if it exits, read in data for next time step */
        sprintf(fname, "safestep.%d",file_read);
        if(outnp != in.d.nprint && (safestrm = fopen(fname,"r")) != NULL) {
                fread ((char *)&acum_header,sizeof(acum_header),1,safestrm);
                fread ((char *)acout,sizeof(acout),1,safestrm);
                fclose(safestrm);
        }
        else { /* This is to zero the acout matrix... */

            for ( i = 0; i < chsize; i++ ) {
                for ( j = 0; j < cwsize; j++ ) {
                    m = who(j,i);                  /* Node I'm considering */
                    for ( nx = 0; nx < in.d.ncx; nx++ ) {
                        for ( ny = 0; ny < in.d.ncy; ny++ ) {
                                acout[m].l.npart[nx][ny] = 0.;
                                acout[m].l.npart_h[nx][ny] = 0.;
                                acout[m].f.vx[nx][ny] = 0.;
                                acout[m].f.vy[nx][ny] = 0.;
                                acout[m].f.kinen[nx][ny] = 0.;
                                acout[m].f.kinenx[nx][ny] = 0.;
                                acout[m].f.kineny[nx][ny] = 0.;
                                acout[m].f.kinenz[nx][ny] = 0.;
```

Appendix C

Program For The Cube

```
/* In this file, 2d.h, are declared the global parameters used in
 *  the Cube based program. Each is given a short description.
 *  2d.h is included at the beginning of each subroutine.
 */

#include <math.h>

#include "2d.def"

extern int Enum[1+LUTRAD*LUTRAD];
/*          # of points on spheres in LUT
*/
extern struct Epoint { char Ex, Ey, Ez; } *pE[];
/*          The Look Up Table, LUT, for integer collisions
*/
extern unsigned char number_in_group[1+LUTRAD*LUTRAD][3];
/*          Used to sort LUT based on parity of components
*/
extern char heavy_lite_mod[], heavy_lite_div[];
/*          Used in multi_mass colls to save a bit of time
 *        doing % operation or long division */
extern short step_type;
/*           For lock step coordination between nodes
*/
extern short node;
/*          node # */
extern short locx;
extern short locy;
/*          Node number in x and y directions
 */
extern short chsize;
extern short cwsize;
/*          Number of nodes in x and y directions
 */
extern short msg_rt;
/*          Number of nearest neighbor nodes to route data to
 */
extern short neighbor[8];
/*          The node numbers of those neighbors
 */
extern short npart[NXCELL][NYCELL];
extern short npart_h[NXCELL][NYCELL];
extern short npsum[NXCELL][NYCELL],
/*      npart[nx][ny] = no. particle in cell
 *       npart_h = number of heavy particles in cell
 *      npsum[nx][ny] = summation by columns to nx, ny-1
 *                      index of first particle in cell
 */
extern short vrmaxcount[NXCELL][NYCELL];
/*          used in coll to track when vrm_sq[][] was last incremented */
extern int  lcr[];
 /*     lcr[nmol] = particle label related to storage  of
  *          particle quantities — assigned to a particle
  *          during initial problem setup
  */
extern int temp_width[], temp_width_div2[], temp[];
extern int temp_array[4][101];
/*          temp_width[] = width of IDSMC Gaussian at some temp.
 *        temp_width_div2[] = 1/2 width of IDSMC Gaussian at some temp.
 *        temp_array[][] = The Gaussians at some temp.
 */
extern char mass_of_type[];
```

```
/*          The mass of each type of particle.
 */
extern long ini_totmol;
extern long totmol;
/*          ini_totmol = the initial total number of particles created in node
 *        totmol = the current total number of particles in node
 *          (MUST ALWAYS BE LESS THAN NM)
 */
extern long num_mass[];
/*          the number of particles in each mass group.
 */
extern long    tot_field_mol;
/*          tot_field_mol = total # molecules in whole field
 */

struct meshlong {
/*          these contain the accumulated number of particles in the rows and
 *        columns of cells of the whole flow field just before re-meshing
 */
          long xscale[NDXMAX*NXCELL + 1];
          long yscale[NDYMAX*NYCELL + 1];
};
extern struct meshlong scale;

extern float pi2, sq2;
/*          pi2 = 2 * PI
 *        sq2 = sqrt(2)
 */
extern float xm, ym;
/*        xm = size of node region in x direction
 *        ym = size of node region in y direction
 */
extern float chx, chy, cell_vol;
/*          chx = cell dimension in x-direction
 *        chy = cell dimension in y-direction
 *        cell_vol = cell volume
 */
extern float cxs;
/*          cxs = hard sphere collision cross section
 */
extern float um, vmw;
/*          vmw = most probable molecular speed @ wall temp
 */
extern float fnd;
/*          fnd = unnormalized undisturbed gas number density
 */
extern float vr0;
/*          vr0 = 1st approx to relative speed in undisturbed gas
 */
extern timetype time;
/*          time = flow time
 */
extern float ct[4][NXCELL][NYCELL],vrm_sq[NXCELL][NYCELL],
extern float recentvrmax_sq[NXCELL][NYCELL],
/*          ct[][nx][ny] = cell time
 *        vrm_sq[nx][ny] = maximum relative speed squared
 *.      xsectmax is the max(sum of the cross sections of two particles)
 *        recentvrmax_sq is the most recent value of the
 *                    relative mean velocity squared
 */

extern float rand();

extern postype ptx[], pty[];
extern veltype ptvx[], ptvy[], ptvz[];
extern char type[];
/*           These are the vectors/arrays containing the positions,
 *         velocities and masses of the particles.
 */

struct particle {
          postype x;
          postype y;
          veltype vx;
          veltype vy;
          veltype vz;
          unsigned char type;
};
extern struct particle *ppt[];

extern struct particle_sent_recv {
          int num_sr; /* number sent or received */
          struct particle pt_sr[PTSENT]; /* particle Sent or Received */
} pmesg_sr[];
/*           This is the structure to be Sent/Recved to the neighbors
 */

struct meshflt {
          float xmark[NDXMAX+1], ymark[NDYMAX+1];
/*          these contain the 'new' node heights and widths sent down from the
 *                 host. */
};
extern struct meshflt mesh;

struct output { /* Structure of data output to the host */
          short npart[NXCELL][NYCELL];
          short npart_h[NXCELL][NYCELL];
/*          npart[][] = accum no. parts in cell
 *        npart_h[][] = accum no. heavy parts in cell
 */
/*          short null1;  null needed when NXCELL&NYCELL are odd, same in host
 *                 to account for different formats in cube and host
 */
          float cellx[NXCELL], celly[NYCELL];
/*                   cellx[] = X coord of cell center
 *                 celly[] = Y coord of cell center
 */
          float vx[NXCELL][NYCELL];
          float vy[NXCELL][NYCELL];
          float kinen[NXCELL][NYCELL];
          float kinenx[NXCELL][NYCELL];
          float kineny[NXCELL][NYCELL];
          float kinenz[NXCELL][NYCELL];
/*                   vx[][] = accum u velocity in cell
 *                 vy[][] = accum v velocity in cell
 *                 kinen[][] = accum trans energy in cell
 *                 kinenx[][] = accum x trans energy in cell
 *                 kineny[][] = accum y trans energy in cell
 *                 kinenz[][] = accum z trans energy in cell
 */
};
extern struct output out;

/* Input variables comming from the host */
struct input {
          short nprint;
          short nstep;
          short seed;
```

```
            short mc;
            short runmax;
/*                      nprint = number of printouts per run
 *                   nstep = number of time steps per printout
 *                   mc = initial no. parts per cell
 *                   runmax = number of runs for ensemble averaging
 */
            short ncx ,
            short ncy;
/*                      ncx = number of cells in x direction
 *                   ncy = number of cells in y direction
 */
            short hremesh, cremesh;
/*                      hremesh = amount of smoothing of mesh
 *                   cremesh = #  of time steps between each cube meshing
 */
            short time_avg;
/*                      Used to indicate if code should switch to time averaging
 *                    instead of ensamble averaging after time_avg printouts.
 */

            float xm_glob;
            float ym_glob;
/*                      xm_glob = size of global region in x direction
 *                   ym_glob = size of global region in y direction
 */
            float x_left;
            float y_bot;
/*                      x_left = location of left hand side of flow field
 *                   y_bot =  location of the bottom of the flow field
 */
            float mass_h, perc_h, diameter_ratio;
/*                      mass_h = the mass of the heavier species (light assumed = 1)
 *                   perc_h = the initial percent of heavy species present.
 *                   diameter_ratio = the diameter ratio of heavies to lites.
 *                            ( not presently used )
 */
            float Tw;
            float uw;
/*                      Tw = wall temperature divided by initial temperature
 *                   uw = wall velocity
 */
            float u_init;
            float u_enter;
            float lamda_zero;
            float ts_ratio;
            float lattice_space;
            float lattice_jump;
/*                      u_init = a free velocity variable.
 *                   u_enter = speed of piston
 *                   lamda_zero = initial mfp in neutral gas.
 *                   ts_ratio = ratio of time step (dtm) to initial coll time.
 *                   lattice_space = distance between lattice sites.
 *                   lattice_jump = how far a vel=1 particle can move in one dtm.
 */
           float dtm;
/*                      dtm = time step for calculation, mean upstream collision times
 */
            float diff;
/*                      diff = 0.  ->  rough wall with above parameters set
 *                   diff = 1.  ->  diffuse wall - molecules emitted at
 *                            wall temp
 */
            float obs_height, obs_a;
```

```
/*                      obs refers to some obstacle in the flow. Height is its height
 *                              and obs_a its X location
 */
            float pstop;
/*                      pstop = the x position at which the piston will stop
 */
            float temp;
/*                      a temporary or free variable
 */
            float nodeheight[NDYMAX + 1], nodewidth[NDXMAX + 1];
/*                      nodeheight[0]=y height of the bottom of the lowest node while
 *                  nodeheight[1] is the y height of the top of that node, etc..
 *                  nodewidth[0] = x position of the LHS of the leftmost node while
 *                  nodewidth[1] is the x position of the RHS of that node, etc..
 */
};

extern struct input in;
```

```
/* Here is where most global variables are actually defined. */

#include <cube/cosmic.h>

#include "2d.h"

#define PI          3.141593

int Enum[1+LUTRAD*LUTRAD];
unsigned char number_in_group[1+LUTRAD*LUTRAD][3];
struct Epoint  *pE[LUTRAD*LUTRAD+1];
short step_type;
short node;
short locx;
short locy;
short chsize;
short cwsize;
short msg_rt;
short neighbor[8];
short npart[NXCELL][NYCELL];
short npart_h[NXCELL][NYCELL];
short npsum[NXCELL][NYCELL];
short vrmaxcount[NXCELL][NYCELL];
char mass_of_type[4];
char heavy_lite_mod[401], heavy_lite_div[401];
int lcr[NM];
int temp_width[4], temp_width_div2[4];
int temp_array[4][101];
long ini_totmol;
long totmol;
long num_mass[4];
long  tot_field_mol;
timetype time;
float pi2;
float sq2;
float xm;
float ym;
float chx;
float chy;
float cxs;
float cell_vol;
float vmw;
float fnd;
float vr0;
float ct[4][NXCELL][NYCELL];
float vrm_sq[NXCELL][NYCELL];
float recentvrmax_sq[NXCELL][NYCELL];
float rand();
postype ptx[NM], pty[NM];
veltype ptvx[NM], ptvy[NM], ptvz[NM];
char type[NM];
struct particle  *ppt[1];
struct particle_sent_recv pmesg_sr[9];
struct meshflt mesh;
struct meshlong scale;
struct output out;
struct input in;
```

```
/* This is the main program for the Cube: each node has an identical copy
   Each receives data from a neighboring node and, in turn, starts
   running the (Integer) Direct Simulation Monte Carlo calculation. In every
   nstep time steps, updated output is sent to Host.
   This output sending happens nprint times in each of runmax runs.
*/

#include <cube/cosmic.h>
#include "2d.h"

unsigned long ttime[20];  /* array used to track time spent in each subroutine */
float numbercl, numberc2, numbercl2, jumper;
          /* # of collisions (of each type), jumper = # parts. jumping more than 1 node */

main()                                                                        main
{
          float tottime, efficiency;
          int nrun, np, ns, i, minutes[20];
          unsigned int seed;
          long seedbits, ran_int_seed;

          numbercl = numberc2 = numbercl2 = jumper = tottime = efficiency = 0.;
          for(i = 0; i < 20; i++){
            ttime[i] = 0;
            minutes[i] = 0;
          } /* Set these to zero */

         setup();          /* Establish cube and my node parameters and receive
                            *  initial data from Host */
          cube_getdat(); /* Calculate some flow and particle parameters */
          neighbors();   /* To find who adjacent nodes are */

          seed =  in.seed + 3737 + mynode();
          seedbits  =  2762 + in.seed + mynode();
          srandom(seed);
          set_ranbits(seedbits); ranbits(300);
          ran_int_seed = 1948 + in.seed + mynode(); set_ran_int(ran_int_seed);
          for(i=1; i<1000; i++) ran_int_le(10000);
                              /* Random #s seeds have to be initialized in each node */

          if(VTYPE == 1) {
                    make_LUT();
                    if(node == 0) print("VTYPE = %d, running IDSMC code", VTYPE);
          }
          else      if(node == 0) print("VTYPE = %d, running DSMC code", VTYPE);


         for ( nrun = 0; nrun < in.runmax; nrun++) {     /* Begin first/next run */
                    time = 0.;
                    initia();          /* Creates initial particles in each cell */
                    if(VTYPE == 1) {scramble(); init_heavy_lite();}
                                                  /* To be sure it is freshly mixed */

                    reset_local();    /* puts global and local ID on all particles */
                    step_type = 0;     /* (re)-initialize step_type for new run */

                    send_ok();          /* sends signal(ready to receive particles) to
                                                  neighbor nodes. Signal from the neighbor
                                                  nodes will be received at the 1st recv_ok()
                                                  in the following loop. */
                    sample();           /* calculates macro properties for each cell */
```

```
send_out();          /* sends output(macro properties) to the  Host */
zero();              /* reinitialize macro sample() matrix*/

                       /* Now begin to loop over the time steps and
                        *  the times data is sent out to
                        *  the Host */

for ( np = 0; np < in.nprint; np++ ) { /* data sent out once each nprint */
     for ( ns = 0; ns < in.nstep; ns++ ) { /* increment one time step*/


          if((locx == 0) && (ns == 0)) {
                    tottime = efficiency = 0;
                    for(i=0; i<20; i++) tottime += ttime[i];
                    if(tottime > 0.) efficiency = 1. - (float)(ttime[12]+ttime[4]
                      +ttime[14]+ttime[8]+ttime[9]+ttime[11]+ttime[16]) / tottime;
print("nprint= %d npart= %ld num1= %4.0f, num2= %4.0f, num12= %4.0f, jump= %2.0f, eff=%2.2f",
                    np, totmol, numberc1, numberc2, numberc12, jumper,efficiency);
          }

                    /* Send out some interesting data
                     *  from first column of nodes */

          time = ( np * in.nstep + ns + 1 ) * in.dtm;
                                   /* Actual time of calculation */
          step_type = np * in.nstep + ns + 1;
          move_pt();            /* moves all particles in in.dtm */
          recv_ok();            /* receive signal(ready to receive
                                     particles) from neighbor nodes */
          send_pt();            /* sends particles to neighbors */
          recv_pt();            /* receives particles from neighbor
                                     nodes until receives end-signal */
          send_ok();            /* sends signal(ready to receive pts)
                                     to neighbors. This signal will be recvd
                                     at recv_ok() in the next step(ns)
                                     in this loop */
          reset_local();           /* puts global and local ID on all parts. */
          coll();                  /* perform the collisions in each cell */
          if(ns % in.cremesh==0 || ns==in.nstep-1) remesh(ns,np);
          /* Every cremesh time steps the cube should remesh.
           *  In addition, on the time step just before the data
           *  is to be sent out to the Host the Host is given
           *  (in send_out_scale) a chance to create the mesh */

     }
     sample();                     /* Same as before */
     send_out();
     zero();
}
step_type += 1;
recv_ok();
```

```
/* This subroutine is only called once. It establishes the cube dimensions,
 *  where each node is and how many neighbors each node has.
 *  This routine is much like host_set.c for the host.
 */

#include <cube/cosmic.h>
#include "2d.h"

short chdim, cwdim;

setup()                                                                    setup
{
        int i, j;
        short dim;
        short size;
        MSGDESC in_d;
        sdesc(&in_d,0,0,0,&in,sizeof(in));

        node = mynode();              /* My specific node number, 0 -> 127 */
        dim = cubedim();              /* Cube dimension, 6 or 7, say */
        size = 1 << dim;
        recvb(&in_d);          /* Receive data from node above me or Host if I'm 0 */
        if((in_d.node = node + 1) != size) sendb(&in_d);
                               /* Send data on to node below me if I'm not last node */

        chsize = NDYMAX; /* The number of nodes in X direction */
        cwsize = NDXMAX; /* The number of nodes in Y direction */

        j = chsize;
        for(i=0; !(j&1); i++, j>>=1){}
                chdim = i; /* The dimension of the cube in X, ie, = 3 for 8 nodes */

        j = cwsize;
        for(i=0; !(j&1); i++, j>>=1){}
                cwdim = i; /* The dimension of the cube in Y */


        /* Each node already knows its node# as its ID. We have to give the
         * location coordinate to each node such that its adjacent node# is one
         * bitwize different. The following two equations calculate those locations
         * for given node whose # is given by mynode() above.
         */
        locx = grey_bin(node & (cwsize - 1 )); /* =0 for leftmost node */
        locy = grey_bin((node >> cwdim));       /* =0 for top node */

        /* Calclate number of message routings for send & recv, that is, the number of
         * nodes immediately adjacent to me.
         * For a closed loop system where particles never leave the flow field
         * only the top and bottom nodes have a different # of neighbors.
         */

        msg_rt = 8;
        if ((locy == 0) || (locy == (chsize - 1))) msg_rt = 5; /* Node on top or bottom*/
        if ((locy == 0) && (locy == (chsize - 1))) msg_rt = 2; /* Only 1 row of nodes */
```

```
/* These two subroutines, the same in both the host and the cube, are
 *  used to find a node number from its x,y location, or visa versa.
 */

#include <cube/cosmic.h>

grey_bin(n)                     /* calculate binary code from grey code */          grey_bin
short n;
{
     int i;
     for(i = n; n >>= 1; i ^=n);
     return(i);
}

who(x,y)                       /* calculate node#=who(x,y) from node location */       who
short x;
short y;
{
     extern short cwdim;

     x=(x ^ (x >> 1)) + ( (y ^ (y >> 1)) << cwdim );
     return(x);
```

```
/* Here are calculated several cell or particle parameters. This routine
 *   is very similar to host_dat.c
 */

#include <cube/cosmic.h>
#include "2d.h"


cube_getdat()                                                          cube_getdat
{
          int i;
          float PI;
          float dia_ratio, mass_rat, num_rat, k;

          PI = 3.14159;
          pi2 = 2 * PI;
          sq2 = sqrt(2.);
          xm = (in.xm_glob-in.x_left) / cwsize;
                                        /* node width for average field node */
          ym = (in.ym_glob-in.y_bot) / chsize;
                                        /* node height for average field node*/
          chx = xm / in.ncx;            /* cell width for average cell */
          chy = ym / in.ncy;            /* cell height for average cell */
          cell_vol = chx * chy;
                                       /* Here the cell volume must be the same in every
                                       * node so that the particle cross sections are
                                       * the same. */
          fnd = in.mc / cell_vol;      /* particle density */

          cxs = 1. / (sq2 * fnd * in.lamda_zero);
                                        /* The X-section if identical particles */
          if(in.mass_h > 1 && in.perc_h > 0) {
                                        /* If particles have differ. masses */
                    if(node == 0)
                         print("Single mass particle cross section is %f",cxs);
                    dia_ratio = in.diameter_ratio;
                    mass_rat = 1./in.mass_h;
                    num_rat = in.perc_h/(1.-in.perc_h);
                    k = (1. + 2.*dia_ratio + dia_ratio*dia_ratio)/4.;
                    cxs = (1./fnd/in.lamda_zero)*
                              (1./(sq2 + num_rat*k*sqrt(1+mass_rat)) +
                              1./(sq2 + (1./num_rat)*k*sqrt(1+(1./mass_rat))) );
                    if(node == 0)
                       print("Multi_species particle cross section is %f",cxs);
          }

          ini_totmol = in.mc * in.ncx * in.ncy;
                    /* Initial total num of molecules per node */

                    /* Set mesh to that sent down by the Host */
          for(i=0; i<=cwsize; i++) mesh.xmark[i] = in.nodewidth[i];
          for(i=0; i<=chsize; i++) mesh.ymark[i] = in.nodeheight[i];

          chx = (mesh.xmark[locx+1]-mesh.xmark[locx])/in.ncx;
          chy = (mesh.ymark[chsize-locy]-mesh.ymark[chsize-locy-1])/in.ncy;
                    /* These are now the actual dimensions
                     *  of a cell whereas above, chx and chy were
                     *  dimensions just used to calculate
                     *  the particle size which must
                     *  be common to every node.
                     */
          for(i=0; i<in.ncx; i++) out.cellx[i] = chx * (0.5+i) +
                              mesh.xmark[locx];
          for(i=0; i<in.ncy; i++) out.celly[i] = chy * (0.5+i) +
```

...cube_getdat

```
                    mesh.ymark[chsize-locy-1];
            /* Locate centers of the cells for output */

    vmw = sqrt(in.Tw); /* for wall temperature */


    for(i=0; i<8; i++) pmesg_sr[i].num_sr = 0;
```

```
/* This routine finds 'my' neighboring nodes, eight if 'I'm' in the middle
 * and 5 if 'I'm' on the top or bottom. They are stored in array
 * neighbors[msg_rt].
 * Neighbors is only executed once.
 */

#include <cube/cosmic.h>
#include "2d.h"

neighbors()                                                          neighbors
{
        int i = 0;

        if ((locy + 1) < chsize){   /* If I have a node above me... */
                neighbor[i++] = who(locx,(locy + 1));
                if ((locx + 1) < cwsize)  /* If there's 1 to my right... */
                        neighbor[i++] = who((locx + 1),(locy + 1));
                else    neighbor[i++] = who(0,(locy + 1));
                if (locx > 0)  /* If there's 1 to my left, and so on... */
                        neighbor[i++] = who((locx - 1),(locy + 1));
                else    neighbor[i++] = who((cwsize - 1),(locy + 1));
        }
        if ((locx + 1) < cwsize)
                neighbor[i++] = who((locx + 1),locy);
        else    neighbor[i++] = who(0,locy);
        if (locx > 0)
                neighbor[i++] = who((locx -1),locy);
        else    neighbor[i++] = who((cwsize - 1),locy);
        if (locy > 0){
                neighbor[i++] = who(locx,(locy - 1));
                if ((locx + 1) < cwsize)
                        neighbor[i++] = who((locx + 1),(locy - 1));
                else    neighbor[i++] = who(0,(locy - 1));
                if (locx > 0)
                        neighbor[i++] = who((locx - 1),(locy - 1));
                else    neighbor[i++] = who((cwsize - 1),(locy - 1));
```

```
/* This routine initializes the positions and velocities of all of the
 * particles in the node. Provision has been made to run cases with
 * different initial mean kinetic temperatures and a different mean density in
 * some cells. This is the subroutine to be altered if a different initial
 * condition is sought. Each particle created is placed
 * randomly within each cell (at a lattice site)
 * and is given a temperature drawn from an integer
 * Gaussian distribution. */

#include <cube/cosmic.h>
#include "2d.h"

extern unsigned long ttime[];
float RTemp, RTmix;

initia()                                                                 initia
{
            int nx, ny, i, j, k, m, numpart, howmany_per_cell, extra_per_row;
int ii;
            int num_per_cell, num_sofar;
            int num_sites_in_x, num_sites_in_y;
            int a, b;
            unsigned int tim0;
            float beta, inverse_beta, num_per_row[NYCELL];
            float particle_mass, vel;
            float const;
            float xh, xl, yh, yl, x_low_site, y_low_site;
            float cv, m_avg, m1, m2, d1, d2, dx, n0, n1, n2, brack, lam;
            float nu_12, nu_21, nu_11, nu_22;
            float cbar=0.;

            tim0 = clock();

            RTemp = 0.5*0.25*(3.141593)*
                    (in.lamda_zero*in.ts_ratio/in.lattice_space/in.lattice_jump)
                    *(in.lamda_zero*in.ts_ratio/in.lattice_space/in.lattice_jump);
                    /* That is, for a lattice gas, the temp is initially set as
                     *  RTo.
                     */
            inverse_beta = sqrt(2. * RTemp); /* That is, 1/beta */
            vr0 = 3. * sqrt(2./3.14159) * inverse_beta;
                    /* vr0 should be > highest initial relative velocity */

            if(node == 0) printf("IN INITIA, RTemp(of mass 1s) = %f",RTemp);

    if(in.perc_h > 0 && in.mass_h != 1) {
            m_avg = 1. + in.perc_h*(in.mass_h - 1.);
                                        /* The average particle mass. */
            m1 = 1.;                    /* Mass of light species */
            m2 = in.mass_h * m1;        /* Mass of heavy species */
            cv = (in.xm_glob-in.x_left)*(in.ym_glob-in.y_bot)
                    /in.ncx/in.ncy/NDYMAX/NDXMAX;
                                        /* Average initial cell volume */
            n2 = in.perc_h*in.mc/cv;    /* Number density of heavies */
            n1 = in.mc*(1. - in.perc_h)/cv; /* Number density of lights */
            n0 = in.mc/cv;              /* Number density */
            d1 = sqrt(cxs/3.141593); /* Diameter of light species */
            d2 = in.diameter_ratio * d1;     /* Diameter of heavy species */
            dx = 0.5 * (d1 + d2);       /* Average of two diameters */
            brack = (2./n0)*(2.*d1*d1*n1*n1*sqrt(2./m1) +
                    n1*n2*(d1+d2)*(d1+d2)*sqrt((m1+m2)/m1/m2) +
                    2.*d2*d2*n2*n2*sqrt(2./m2) );
            RTmix = (in.ts_ratio*in.ts_ratio/in.dtm/in.dtm)*
                    2./3.141593/m_avg/brack/brack;
```

...initia

```
                                        /* R*T of the gas mixture */
        lam  =  (1./3.14159/(n1+n2))*(n1/(sqrt(2.)*n1*d1*d1  +
                  n2*((d1+d2)*(d1+d2)/4.)*sqrt(1.+m1/m2)  )  +
                  n2/(sqrt(2.)*n2*d2*d2  +  n1*((d1+d2)*(d1+d2)/4.)*
                  sqrt(1.+m2/m1)  )  );
                                        /* Mean free path in the mixture */
        RTemp  =  RTmix*m_avg/m1;  /* R*T of light species */
        inverse_beta  =  sqrt(2.  *  RTemp);  /* That is, 1/beta */
        vr0  =  3.  *  sqrt(2./3.14159)  *  inverse_beta;
                    /* vr0 should be > highest initial relative velocity */


        nu_12  =  2.*sqrt(3.142)*dx*dx*n2*sqrt(2.*RTemp*(m1+m2)/m1/m2);
        nu_21  =  2.*sqrt(3.142)*dx*dx*n1*sqrt(2.*RTemp*(m1+m2)/m1/m2);
        nu_11  =  2.*sqrt(3.142)*d1*d1*n1*sqrt(2.*RTemp*(m1+m1)/m1/m1);
        nu_22  =  2.*sqrt(3.142)*d2*d2*n2*sqrt(2.*RTemp*(m2+m2)/m2/m2);
                                        /* 4 different collision freqs */

if(node  ==  0)  print("IN  INITIA,  RTmix=%f,  lam0=%f,  cell_vol=%2.4f,  RTemp=%f",
RTmix,lam,  cv,RTemp);
if(node  ==  0)  print("m1=%2.3f,  m2=%2.3f,  d1=%2.3f,  d2=%2.3f,  n1=%2.3f,  n2=%2.3f,  m_avg=%2.3f,  br%
          m1,m2,d1,d2,n1,n2,m_avg,brack);
if(node  ==  0)  print("nu_12  =  %f,  nu_21=%f,  nu_11=%f,  nu_22=%f",
   nu_12,  nu_21,  nu_11,  nu_22);
   }

        mass_of_type[0]  =  1;
        mass_of_type[1]  =  in.mass_h;
        mass_of_type[2]  =  1;
        mass_of_type[3]  =  1;
          /* Consider only a binary gas. */

                              /* We now may create the different distribution functions. If
                               *  the gas contains 2 species only 2 distributions are
                               *  needed, 1 for each species. If, however, the gas
                               *  is of one species but the flow boundaries are diffuse
                               *  4 are needed, 2 are full Gaussians and 2 are the
                               *  nomal VDF, a Gaussian multiplied by the normal component.
                               *  Multi-species and diffuse walls can't be done at
                               *  the same time in this version.
                               *  So, for multi-species:
                               *  m  =  0  ->  Gaussian for mass  =  1 particles
                               *  m  =  1  ->  Gaussian for mass  =  in.mass_h particles
                               *  And, for diffuse boundaries:
                               *  m  =  0  ->  Gaussian for wall at temp  =  1
                               *  m  =  1  ->  Gaussian for wall at temp  =  1*in.Tw
                               *  m  =  2  ->  v*Gaussian for wall at temp  =  1
                               *  m  =  3  ->  v*Gaussian for wall at temp  =  1*in.Tw
                               */
    for(m=0;  m<=1;  m++)  {    /* Use for(m=0; m<=3; m++) for diffuse bndrys */
        if(m  ==  0)  particle_mass  =  1;  else  particle_mass  =  in.mass_h;
        beta  =  1.  /  sqrt(2.*RTemp  /  (float)mass_of_type[m]);
        if(m  ==  1  ||  m  ==  3)  beta  =  1  /  sqrt(2.*RTemp*in.Tw);  /* for heat transfer */
        if(m  <  2)for(i=0;  i<=100;  i++)  {
                              /* For DF in free space place the value (0 -> 8191) (to save space) into
                               *  the array appropriate for that species */
          vel  =  i  -  50.;
          if((-beta)*beta*vel*vel  <  -50.)  temp_array[m][i]  =  0;
          else  temp_array[m][i]  =  (int)(0.5  +  8191  *  exp((-beta)*beta*vel*vel));
        }
        else  {  /* For the DF for the normal velocity component */
           const  =  beta  *  sqrt(2.0)  *  exp(0.5);
           for(i=0;  i<=100;  i++)  {
                vel  =  i  -  50.;
```

...initia

```
                temp_array[m][i] = (int)(0.5 + 8191.*const*fabs(vel)*exp((-beta)*beta*vel*vel));
        }
    }
    if(temp_array[m][0] > 0)
        print("WARNING, temp_array is too narrow, temp_array[%d][0]=%d\n",m,temp_array[m][0]);
    for(i=0; i<=100; i++) {j = i; if(temp_array[m][i] > 0) break; }
    temp_width_div2[m] =  (50 - j);
    temp_width[m] =  2*(50 - j);
    if(node == 0)print("temp_width[%d] = %d , beta = %f, mass=%f\n",
                                m, temp_width[m], beta, mass_of_type[m]);
                        /* The width and half width of the DFs will be used
                         *  when drawing velocities from the DF. */



    numpart = 0;                /* Number of particles counted so far */
    totmol = ini_totmol;        /* (Initial)Total num of molecules */

                                /* Update node boundaries with host mesh */
    for(i=0; i<=cwsize; i++) mesh.xmark[i] = in.nodewidth[i];
    for(i=0; i<=chsize; i++) mesh.ymark[i] = in.nodeheight[i];
    chx = (mesh.xmark[locx+1]-mesh.xmark[locx])/in.ncx;
    chy = (mesh.ymark[chsize-locy]-mesh.ymark[chsize-locy-1])/in.ncy;

                                /* Calculate center of cell locations */
    for(i=0; i<in.ncx; i++) out.cellx[i] =
                                chx * (0.5+i) + mesh.xmark[locx];
    for(i=0; i<in.ncy; i++) out.celly[i] =
                                chy * (0.5+i) + mesh.ymark[chsize-locy-1];


    for ( ny = 0; ny < in.ncy; ny++ ) {
            num_per_row[ny] = in.mc * in.ncx;
                                /* This is the # of particles
                                 * per row of cells within my node. */
    }

    for ( ny = 0; ny < in.ncy; ny++ ) {
            num_sofar = 0;              /* # placed so far */
            yh = (int)((out.celly[ny] + 0.5 * chy)/in.dtm) * in.dtm;
            yl = (int)((out.celly[ny] - 0.5 * chy)/in.dtm) * in.dtm;
            for ( nx = 0; nx < in.ncx; nx++ ) {
                                /* From here...... */
                    /* particles  only on latice sites. */
                    xh = (int)((out.cellx[nx] + 0.5 * chx)/in.dtm) * in.dtm;
                    xl = (int)((out.cellx[nx] - 0.5 * chx)/in.dtm) * in.dtm;
                    num_sites_in_x = (int)((xh-xl)/in.dtm);
                    num_sites_in_y = (int)((yh-yl)/in.dtm);
                    if(num_sites_in_x*num_sites_in_y == 0) goto no_particles_here;
                    if(xl == out.cellx[nx] - 0.5 * chx) x_low_site = xl;
                        else x_low_site = xl + 1;
                    if(yl == out.celly[ny] - 0.5 * chy) y_low_site = yl,
                        else y_low_site = yl + 1;

                    num_per_cell = num_per_row[ny] / in.ncx;
                                /* The lowest (int) number per cell */
                    extra_per_row = num_per_row[ny]-
                            (float)(in.ncx*num_per_cell) + 0.5;
                                /* # needed more than the lowest for
                                 *  a full row. */
                    howmany_per_cell = num_per_cell;
```

```
                                        /* Actual # which will be created in
                                         *  this current cell. */
                              const = (float)(nx+1)*(float)extra_per_row/(float)in.ncx;
                              if(const > num_sofar) {  /* put 1 more pt in this cell */
                                        howmany_per_cell++;
                                        num_sofar++;
                              }
                                        /* ...to here is a way to even out the
                                         *  initial pt distribution if each
                                         *  row doesn't get an evenly divis-
                                         *  ible # of particles. */

                              for(ii=0; ii<4; ii++) ct[ii][nx][ny] = rand() * 2. * chx * chy /
                              ( howmany_per_cell * howmany_per_cell * cxs * vr0 );
                                        /* Set cell time randomly for each spec. */
                              vrm_sq[nx][ny] = 2.*vr0;
                                        /* Set max relative pt velocity to
                                         *  twice the mean value. */
                              recentvrmax_sq[nx][ny] = 0.;        /* Initialize */
                              vrmaxcount[nx][ny] = 0;
                              for ( j = 0; j < howmany_per_cell; j++ ) {
                                        /* Now, create particles. */
                                        k = numpart;       /* To save space... */
                                        numpart++;

                                        /* Place them at sites. */
                                        a = ran_int_le(num_sites_in_x-1);
                                        b = ran_int_le(num_sites_in_y-1);
                                        ptx[k] = x_low_site + a*in.dtm;
                                        pty[k] = y_low_site + b*in.dtm;

                                        /* Assign a mass to them. */
                                        m = 0; if(rand() < in.perc_h) m = 1;
                                        type[k] = m;

                                        /* Assign a velocity component. */
                                        for(i = ran_int_le(temp_width[m]);
                                                  ranbits(13) >= temp_array[m][i-temp_width_div2[m]+50],
                                                  i = ran_int_le(temp_width[m]) );
                                        ptvx[k] = i-temp_width_div2[m];
                                        for(i = ran_int_le(temp_width[m]);
                                                  ranbits(13) >= temp_array[m][i-temp_width_div2[m]+50],
                                                  i = ran_int_le(temp_width[m]) );
                                        ptvy[k] = i-temp_width_div2[m];
                                        for(i = ran_int_le(temp_width[m]);
                                                  ranbits(13) >= temp_array[m][i-temp_width_div2[m]+50],
                                                  i = ran_int_le(temp_width[m]) );
                                        ptvz[k] = i-temp_width_div2[m];

                                        if(VELDIM == 2) ptvz[k] = 0;

                                        cbar += (ptvx[k]*ptvx[k] + ptvy[k]*ptvy[k] + ptvz[k]+ptvz[k]);


                              }
no_particles_here:
                    a = b;
                    }
          }
          totmol = ini_totmol = numpart;
          tot_field_mol = numpart;
          if(node == 0)print("N times particle speed squared = %f",cbar);

          ttime[1] +=(unsigned int)((unsigned int)clock()-tim0);
```

```
/* This routine resets some of the global variables like npart,
 * lcr and npsum.
 */

#include <cube/cosmic.h>
#include "2d.h"

reset_local()                                                        reset_local
{
          int nx, ny, nmol;
          int m;
          int number;
          float invchx, invchy;

          invchx = 0.9999 / chx;
          invchy = 0.9999 / chy;
                  /* These .9999s were used to prevent the
                   *  possibility of a DSMC particle landing
                   *  exactly on the far edge/boundary of a node
                   */
          if(LATICE != 0) { invchx = 1. / chx; invchy = 1. / chy;}

          for ( nx = 0; nx < in.ncx; nx++ ) {
                  for ( ny = 0; ny < in.ncy; ny++ ) {
                          npart[nx][ny] = 0;
                  }
          } /* Set npart[][] to zero. */

          for ( nmol = 0; nmol < totmol ; nmol++ ) {
                  nx = (ptx[nmol] - mesh.xmark[locx]) * invchx;
                  ny = (pty[nmol] - mesh.ymark[chsize - locy - 1]) * invchy;
                  npart[nx][ny] += 1;
          } /* Compute correct number of particles per cell */

          m = 0;
          for ( nx = 0; nx < in.ncx; nx++ ) {
                  for ( ny = 0; ny < in.ncy; ny++ ) {
                          npsum[nx][ny] = m;
                          m += npart[nx][ny];
                          npart[nx][ny] = 0;
                  }
          } /* Calculate correct index of 1st particle in each cell */

          for ( nmol = 0; nmol < totmol; nmol++ ) {
                  nx = (ptx[nmol] - mesh.xmark[locx]) * invchx;
                  ny = (pty[nmol] - mesh.ymark[chsize - locy - 1]) * invchy;
                  npart[nx][ny] += 1;
                          /* Re-compute npart */
                  number = npsum[nx][ny] + npart[nx][ny] - 1;
                  lcr[number] = nmol; /* Find lcr[] index */
                  /* Index of first particle in cell [nx][ny] is
                      npsum[nx][ny].
                      Index of last particle in cell [nx][ny] is
                      npsum[nx][ny] + npart[nx][ny] - 1 .*/
```

```
/* This routine sends a signal to all of the neighboring nodes that it is ok to
 * send "me" messages containing the particles.
 */

#include <cube/cosmic.h>
#include "2d.h"

extern unsigned long ttime[];

send_ok()
{
        int i;
        unsigned int tim0;
        MSGDESC sd_ok;

        tim0 = clock();
        sdesc (&sd_ok, 0, 0, 0, 0, 0);
        sd_ok.type = step_type + 1;

        for(i=0; i < msg_rt; i++){
                sd_ok.node = neighbor[i];
                sendb(&sd_ok);
        }

        ttime[12] +=(unsigned int)((unsigned int)clock()-tim0);
```

send_ok

```
/* This routine samples the individual particle data to create the macroscopic data which it
 *  places in array out... to be sent out to the host.
 */

#include <cube/cosmic.h>
#include "2d.h"

extern unsigned long ttime[];

sample()                                                                        sample
{
        int nx, ny;
        int l, nmol;
        unsigned int tim0;
        veltype xvel, yvel, zvel;

        tim0 = clock();

        for ( nx = 0; nx < in.ncx; nx++ ) {
                for ( ny = 0; ny < in.ncy; ny++ ) {

                        out.npart[nx][ny] = npart[nx][ny];

                        for ( l = 0; l < npart[nx][ny]; l++ ) {
                                nmol = lcr[npsum[nx][ny] + l];
                                xvel = ptvx[nmol];
                                yvel = ptvy[nmol];
                                zvel = ptvz[nmol];
                                out.vx[nx][ny] += xvel;
                                out.vy[nx][ny] += yvel;
                                out.kinen[nx][ny] +=
                                     (xvel*xvel+yvel*yvel+zvel*zvel);
                                out.kinenx[nx][ny] += xvel*xvel;
                                out.kineny[nx][ny] += yvel*yvel;
                                out.kinenz[nx][ny] += zvel*zvel;
                                if(type[nmol] != 0) out.npart_h[nx][ny] += 1;



                /* This first portion of sample() is adequate if the particles
                 *  all have the same mass. If not, a separate subroutine
                 *  is used so as not to confuse the normally simple
                 *  sampling algorithm. In such a case, the meaning of
                 *  kinen, kinenx and kineny are slightly different.
                 */
        if(in.mass_h > 1 && in.perc_h > 0 ) multi_species_temp();
        ttime[3] +=(unsigned int)((unsigned int)clock()-tim0);


multi_species_temp()                                                multi_species_temp


        int nx, ny;
        int l, m, nmol;
        veltype xvel, yvel, zvel;
        float meanvx[2], meanvy[2], meanvz[2], num_mass[2], esum[2];
        float density, mass_avg_vel_x, mass_avg_vel_y mass_avg_vel_z;
        float co_sq, temp[2], temp_wrtcmv[2],
for ( nx = 0; nx < in.ncx  nx++ ) {
      for ( ny = 0; ny < in.ncy; ny++ ) {

                  for(m=0; m<2; m++)
                     num_mass[m]=esum[m]=meanvx[m]=meanvy[m]=meanvz[m]=0;
                                /* Energy_sum and mean velocities, initially */
                  density=mass_avg_vel_x=mass_avg_vel_y=mass_avg_vel_z=0;
                  for ( l = 0; l < npart[nx][ny]; l++ ) {
                              /* Calculate cell density, number density, mean kinetic energy
                               *  and mean velocity of each species
                               */
                              nmol = lcr[npsum[nx][ny] + l];
                              density += mass_of_type[type[nmol]];   /* density */
                              m = type[nmol],
                              num_mass[m]++; /* number density */
                              xvel = ptvx[nmol];
                              yvel = ptvy[nmol];
                              zvel = ptvz[nmol];
                              esum[m] +=
                                   (xvel*xvel+yvel*yvel+zvel*zvel)*mass_of_type[type[nmol]];
                                          /* mean KE */
                              meanvx[m] += xvel; /* mean velocity */
                              meanvy[m] += yvel;
                              meanvz[m] += zvel;
                  }
                  for(m=0; m<2; m++) { /* mean velocity  for species m */
                              if(num_mass[m] > 0){
                                        meanvx[m] /= (float)num_mass[m];
                                        meanvy[m] /= (float)num_mass[m];
                                        meanvz[m] /= (float)num_mass[m];
                              }
                  }

                  for(m=0; m<2; m++) { /* for the mass averaged velocity */
                     if(num_mass[m] > 0) {
                              mass_avg_vel_x += meanvx[m]*num_mass[m]*mass_of_type[m];
                              mass_avg_vel_y += meanvy[m]*num_mass[m]*mass_of_type[m];
                              mass_avg_vel_z += meanvz[m]*num_mass[m]*mass_of_type[m];
                     }
                  }
                  if(density > 0){
                              mass_avg_vel_x /= (float)density;
                              mass_avg_vel_y /= (float)density;
                              mass_avg_vel_z /= (float)density;
                  }

                  for(m=0; m<2; m++) {
                     if(num_mass[m] > 0) {
                              temp[m] = mass_of_type[m]*(esum[m]/
                                   (float)(num_mass[m]*mass_of_type[m])
                                   - meanvx[m]*meanvx[m]-meanvy[m]*meanvy[m]
                                   -meanvz[m]*meanvz[m])/ 3.;
                                          /* The temperature of species m */
                              co_sq = mass_avg_vel_x*mass_avg_vel_x+
                                   mass_avg_vel_y*mass_avg_vel_y+
                                   mass_avg_vel_z*mass_avg_vel_z;
                                          /* The square of c_o */
                              temp_wrtcmv[m] = mass_of_type[m]*(esum[m]
                                   /(float)(num_mass[m]*mass_of_type[m]) +
                                   co_sq - 2.*(mass_avg_vel_x*meanvx[m]+
                                   mass_avg_vel_y*meanvy[m]+
                                   mass_avg_vel_z*meanvz[m]))/3.;
```

...multi_species_temp

```
                    /* The species m temp. WRT the mass avg vel */


out.kinen[nx][ny] = (num_mass[0]*temp_wrtcmv[0]
        + num_mass[1]*temp_wrtcmv[1]);
        /* The total gas temperature */
out.kinenx[nx][ny] = num_mass[0]*temp_wrtcmv[0];
        /* Now, kinenx contains the temp of species 0 */
out.kineny[nx][ny] = num_mass[1]*temp_wrtcmv[1];
        /* and kineny contains the temp of species 1 */
        /* Thus, the meaning on kinenx and kineny
         *  have changed. This saves creating, storing
         *  and sending a lot of extra data for the
         *  multi-species applications chosen. */
```

```
/* This routine sends the output data directly to the host. */

#include <cube/cosmic.h>
#include "2d.h"

extern unsigned long ttime[];

send_out()                                                          send_out
{
        unsigned int tim0;
        MSGDESC out_sd;
        MSGDESC pls_rd;

        tim0 = clock();

            /* set up message descriptor to send output to host */
        sdesc (&out_sd, HOST, 0, 0, &out, sizeof(struct output));
        sdesc (&pls_rd, 0, 0, 100, 0, 0);


        led(1);                         /* Turn on LED light */

                recvb(&pls_rd);         /* Wait until host asks for data */
                out_sd.type = node + 20000;
                sendb(&out_sd);         /* Send the host the data */

        led(0);                         /* Turn off LED light */

        ttime[4] +=(unsigned int)((unsigned int)clock()-tim0);
}
```

```
/* Set the matrix used to send data to the Host back to zero and
 *  reset the cell locations */

#include <cube/cosmic.h>
#include "2d.h"

extern unsigned long ttime[];
zero()
{
        int nx, ny;
        unsigned int tim0;

        tim0 = clock();

        for ( nx = 0; nx < in.ncx; nx++ ) {
                for ( ny = 0; ny < in.ncy; ny++ ) {
                        out.npart[nx][ny] = 0;
                        out.vx[nx][ny] = 0.;
                        out.vy[nx][ny] = 0.;
                        out.kinen[nx][ny] = 0.;
                        out.kinenx[nx][ny] = 0.;
                        out.kineny[nx][ny] = 0.;
                        out.kinenz[nx][ny] = 0.;
                        out.npart_h[nx][ny] = 0;
                }
        }
        for ( nx = 0; nx < in.ncx; nx++ )
                out.cellx[nx] = ( nx + 0.5 ) * chx + mesh.xmark[locx];
        for ( ny = 0; ny < in.ncy; ny++ )
                out.celly[ny] = (ny + 0.5)*chy + mesh.ymark[chsize-locy-1];

        ttime[14] +=(unsigned int)((unsigned int)clock()-tim0);
```

zero

```
/* This routine moves each particle to its next location by an amount
 *  appropriate to the time step.
 *  The time till first impact either on the top or bottom is calculated,
 *  the impact is calculated (perhaps it is diffusive)
 *  and the particle is moved from the wall. If there is time remaining, the
 *  time till the next impact is found and the process repeats until the time
 *  step is used up. At the end are calculated the side wall collisions.
 */


#include <cube/cosmic.h>
#include "2d.h"

extern unsigned long ttime[];

move_pt()                                                    move_pt
{
          int i, nmol, collided_with_wall;
          unsigned int tim0;
          float tt, a, b;
          float dtm_diff, x_diff, y_diff, vx_diff, vy_diff, vz_diff, RTx2;
          float piston_pos, obst_pos;
          extern float RTemp;
          veltype piston_speed;
          postype oldx, oldy, two;
          timetype dtm1;

          tim0 = clock();

          two = 2.;
          RTx2 = RTemp*2.;
          collided_with_wall = 0;
          /* to move a piston from the left */
          piston_pos = in.x_left + time * in.u_enter;
          piston_speed = in.u_enter;
          if(piston_pos > in.pstop) {
                    piston_pos = in.pstop;
                    piston_speed = 0.;
          }
          /* for the obstacle */
          obst_pos = in.obs_a;


          for ( nmol = 0; nmol < totmol; nmol++ ) { /* for all particles/node */
              dtm1 = in.dtm;                          /* the time step */
              oldx = ptx[nmol];
              oldy = pty[nmol];                       /* Starting particle positions */
              if(in.diff <= 0.) goto simple_step;     /* If non-diffusive walls */
                    dtm_diff = in.dtm;
                    x_diff = ptx[nmol];
                    vx_diff = ptvx[nmol];
                    y_diff = pty[nmol];
                    vy_diff = ptvy[nmol];
                    vz_diff = ptvz[nmol];             /* Initial velocities */
                    collided_with_wall = 0;
                    for ( tt = 0.; 0.<=dtm_diff; dtm_diff -= tt ) {
                                              /* tt is the time until the particle
                                               *  has its first/next impact with
                                               *  a horizontal wall. */
                              if (vy_diff == 0.) tt = 1.e+8; /* It never gets there */
                              else if (vy_diff > 0.) { /* pt moving up */
                                        tt = (in.ym_glob-y_diff ) / vy_diff;
                                    }
```

```
            else {          /* pt moving down, will hit bot 1st */
                tt = - (y_diff-in.y_bot) / vy_diff;
                        /* time till impact on bottom */
            }
        if ( tt > dtm_diff ) break; /* If min time till impact > then
                                     *  time step then don't worry
                                     *  about impacts for this pt */

        collided_with_wall = 1;
        y_diff += tt * 0.9999 * vy_diff;
        x_diff += tt * 0.9999 * vx_diff;
                                    /* place particle almost at
                                     *  impact point with impact
                                     *  velocity */

        if(y_diff < in.y_bot) y_diff -= 2.*(y_diff - in.y_bot);
        if(y_diff > in.ym_glob) y_diff -= 2.*(y_diff - in.ym_glob);
                                    /* This is just in case pt
                                     *  exceeded bdrys, it never should */

        if (vy_diff > 0.) {  /* On the wall at y = in.ym_glob */
                                    /* For diffusive wall, assign
                                     *  new velocity components based
                                     *  on top bdry temperature & vel   */
                if(VTYPE != 0) {  /* integer velocities */
                  for(i = ran_int_le(temp_width[1]);
                      ranbits(13) >= temp_array[1][i-temp_width_div2[1]+50];
                      i = ran_int_le(temp_width[1]) );
                      vx_diff = i-temp_width_div2[1] - in.uw;
                  for(i = ran_int_le(temp_width[1]);
                      ranbits(13) >= temp_array[1][i-temp_width_div2[1]+50];
                      i = ran_int_le(temp_width[1]) );
                      vz_diff = i-temp_width_div2[1];
next_1:           for(i = ran_int_le(temp_width[3]);
                      ranbits(13) >= temp_array[3][i-temp_width_div2[3]+50];
                      i = ran_int_le(temp_width[3]) );
                      vy_diff = i-temp_width_div2[3];
                      if(vy_diff >= 0) goto next_1;
                }
                else {  /* DSMC velocities */
                  vmw = sqrt(in.Tw*RTx2);
                  a = vmw * sqrt(-log(rand()));
                          /* Speed of reflected part. */
                  b =   pi2 * rand();
                  vx_diff = a * sin(b) - in.uw;
                  vz_diff = a * cos(b);
                  vy_diff = - vmw * sqrt(-log( rand()));
                }
        continue;
        }
        else {               /* If not, pt. must be on wall at y=in.y_bot
                                      *  assign new velocity components based
                                      *  on bottom bdry temperature & vel *
                if(VTYPE != 0) {  /* IDSMC velocities */
                    for(i = ran_int_le(temp_width[0]);
                        ranbits(13) >= temp_array[0][i-temp_width_div2[0]+50];
                        i = ran_int_le(temp_width[0]) );
                    vx_diff = i-temp_width_div2[0] + in.uw;
                    for(i = ran_int_le(temp_width[0]);
                        ranbits(13) >= temp_array[0][i-temp_width_div2[0]+50];
                        i = ran_int_le(temp_width[0]) );
                    vz_diff = i-temp_width_div2[0];
next_2:            for(i = ran_int_le(temp_width[2]);
                        ranbits(13) >= temp_array[2][i-temp_width_div2[2]+50];
```

```
                                        i = ran_int_le(temp_width[2]) );
                                    vy_diff = i-temp_width_div2[2];
                                    if(vy_diff <= 0) goto next_2;
                                }
                                else { /* DSMC velocities... */
                                            vmw = sqrt(1.*RTx2);
                                            a = vmw * sqrt(-log(rand()));
                                            b = pi2 * rand();
                                            vx_diff = a * sin(b) + in.uw;
                                            vz_diff = a * cos(b);
                                            vy_diff = vmw * sqrt(-log( rand() ));
                                }
                        continue;
                        }
            }
                        /* Particle has now finished its last collision with
                         *  top or bottom walls as allowed by the size of
                         *  the time step and may now collide with other
                         *  things in the flow. */

                        /* First the position is updated */
simple_step:    if(in.diff <= 0 || collided_with_wall == 0) {
                        /* If walls are specular or particle never touched wall */
                        pty[nmol] += dtm1 * ptvy[nmol];
                        ptx[nmol] += dtm1 * ptvx[nmol];
                        if(pty[nmol] < in.y_bot) {
                                pty[nmol] -= (two*pty[nmol]-2.*in.y_bot);
                                ptvy[nmol] = -ptvy[nmol];
                        }
                        else if(pty[nmol] > in.ym_glob) {
                                pty[nmol] -= (two*pty[nmol]-2.*in.ym_glob);
                                ptvy[nmol] = -ptvy[nmol];
                        }
            }
            else { /* particle has been diffusely reflected from a wall */
                        if(LATICE != 0){ /* Now give them floating point positions */
                                x_diff += dtm_diff * (float)ptvx[nmol];
                                y_diff += dtm_diff * (float)ptvy[nmol];

                                    /* Round particles to nearest lattice site */
                                if(x_diff >= 0.) ptx[nmol] = (int)(x_diff + 0.5);
                                else ptx[nmol] = (int)(x_diff - 0.5);
                                if(y_diff >= 0.) pty[nmol] = (int)(y_diff + 0.5);
                                else pty[nmol] = (int)(y_diff - 0.5);
                                if(pty[nmol] < in.y_bot || pty[nmol] > in.ym_glob)
                                        print("particle is out of field, y = %d",
                                                pty[nmol]);
                        }
                        else { /* particles are Bird particles */

                                ptvx[nmol] = vx_diff;
                                ptvy[nmol] = vy_diff;
                                ptvz[nmol] = vz_diff;

                                x_diff += dtm_diff * ptvx[nmol];
                                y_diff += dtm_diff * ptvy[nmol];

                                ptx[nmol] = x_diff;  /* floating point position */
                                pty[nmol] = y_diff;  /* floating point position */

                                if(pty[nmol] < in.y_bot || pty[nmol] > in.ym_glob)
                                        print("particle is out of field, y = %d",
                                                pty[nmol]);
            }


            /* Now to create specularly reflective sidewalls, if needed */
            if(ptx[nmol] < piston_pos) {
                    ptx[nmol] += two*(piston_pos - ptx[nmol]);
                    ptvx[nmol] = -ptvx[nmol] +two*piston_speed;
                                            /* For specular reflection off piston
                                             *  particle gains 2 x pistonspeed */
            }

            if(ptx[nmol] > in.xm_glob) {
                    ptx[nmol] -= (two*ptx[nmol]-2.*in.xm_glob);
                    ptvx[nmol] = -ptvx[nmol];
            }

                    /* To create an obstruction in the flow */
            if(oldx > obst_pos && ptx[nmol] < obst_pos)
                if((obst_pos-oldx)*(pty[nmol]-oldy)/
                    (ptx[nmol]-oldx) + oldy < in.obs_height) {
                                            /* If, in fact particle would have
                                             *  passed through obstacle... */
                    ptx[nmol] += two*(obst_pos - ptx[nmol]);
                    ptvx[nmol] = -ptvx[nmol];
                }
            if(oldx < obst_pos && ptx[nmol] > obst_pos)
                if((obst_pos-oldx)*(pty[nmol]-oldy)/
                    (ptx[nmol]-oldx) + oldy < in.obs_height) {
                                            /* If, in fact particle would have
                                             *  passed through obstacle... */
                    ptx[nmol] += two*(obst_pos - ptx[nmol]);
                    ptvx[nmol] = -ptvx[nmol];
                }
    }
    ttime[6] +=(unsigned int)((unsigned int)clock()-tim0);
```

```
/* This routine waits till it receives an "OK to send particles
 *  to me" signal from all neighbors */

#include <cube/cosmic.h>
#include "2d.h"

extern unsigned long ttime[];

recv_ok()
{
            int count_ok;
            unsigned int tim0;
            MSGDESC rd_ok;

            tim0 = clock();
           sdesc (&rd_ok, 0, 0, 0, 0, 0);

           rd_ok.type = step_type;
           count_ok = 0;
           while ( count_ok < msg_rt )
           {  /* That is, receive messages until  one is received from
                *        every neighbor */
                 recvb(&rd_ok);
                 count_ok += 1;
           }

            ttime[8] +=(unsigned int)((unsigned int)clock()-tim0);
}
```

```
/* This key routine scans through the list of totmol particles and sends
 *  those which have to be moved away out to another node. If the particle
 *  is to be sent to a neighboring node it is packaged with other particles.
 *  If it is to be sent to a distant node it is sent individually.
 */

#include <cube/cosmic.h>
#include "2d.h"

extern unsigned long ttime[];

send_pt()                                                          send_pt
{
  int nmol, nx, ny, node_send, send_to, num_stored, i;
  extern float jumper;
  unsigned int tim0;
  MSGDESC sd;

  tim0 = clock();
  sdesc (&sd, 0, 0, 10000, 0, sizeof(struct particle_sent_recv));


  for ( nmol = 0; nmol < totmol; nmol++ ) { /* Consider particles 1 at a time */
                                     /* nx and ny are the location where the particle */
    nx = locx;                       /* is to go. Particles will generally remain within */
    ny = locy;                       /*  this node if it covers enough area.*/

    if(ptx[nmol] > mesh.xmark[locx + 1] || ptx[nmol] < mesh.xmark[locx]) {
          /* That is, if particle should be to the right or left of this node.. */
          if(ptx[nmol] > mesh.xmark[cwsize]) {
                   /* Particle is to pass through RHS boundary.
                    * This should not happen for a closed flow field. For a free channel
                    *  with no side walls particles passing the right side of the
                    *  field are sent to the left..... */
                   ptx[nmol] -= (float)in.xm_glob;
          }
          else if(ptx[nmol] < mesh.xmark[0]) {
                   /* While particles passing out through the LHS reenter through the RHS */
                   ptx[nmol] += (float)in.xm_glob;
          }

          for(i=1; i<NDXMAX; i++) {   /* To find out to what node it should be sent... */
                   if(locx+1+i <= cwsize) /* if such a node exists to the right and */
                           if(ptx[nmol] > mesh.xmark[locx+1] && ptx[nmol]
                                    < mesh.xmark[locx+1+i])
                                    /* if particle falls within that node... */
                           {nx = locx + i; break;}      /* send particle there. */
                   if(locx-i >= 0) /* if such a node exists to the left and */
                           if(ptx[nmol] < mesh.xmark[locx] && ptx[nmol]
                                    > mesh.xmark[locx-i])
                                    /* if particle falls within that node... */
                           {nx = locx - i; break;}      /* send particle there. */



          /* Now do the same type of thing in the Y direction. */
    if(pty[nmol] > mesh.ymark[chsize-locy] ||
                           pty[nmol] < mesh.ymark[chsize-locy-1]) {
          /* That is, if particle should be above or below this node.. */

          for(i=1; i<NDYMAX; i++) { /* To find out to what node it should be sent... */
```

```
                if(chsize-locy+i <= chsize) /* if node is below the top and */
                        if(pty[nmol] > mesh.ymark[chsize-locy] &&
                                pty[nmol] < mesh.ymark[chsize-locy+i])
                                /* if particle falls within that node... */
                        {ny = locy - i; break;}       /* send particle there. */
                if(chsize-locy-1-i >= 0) /* if node is above the bottom and */
                        if(pty[nmol] < mesh.ymark[chsize-locy-1] &&
                                pty[nmol] > mesh.ymark[chsize-locy-1-i])
                                /* if particle falls within that node... */
                        {ny = locy + i; break;}       /* send particle there. */
        }
}

if ((node_send = who(nx,ny)) != node) {
                        /* If node where particle belongs is not 'me'... */
        for (i=0; i<msg_rt; i++) { /* find neighbor index */
                if(node_send == neighbor[i]) {send_to=i; break;}
                else if(i == msg_rt-1){ /* It was not a neighboring node but a
                                         *  distant one. Send particle individually.
                                         *  This should be very rare. */
                        pmesg_sr[8].pt_sr[0].x = ptx[nmol];
                        pmesg_sr[8].pt_sr[0].y = pty[nmol];
                        pmesg_sr[8].pt_sr[0].vx = ptvx[nmol];
                        pmesg_sr[8].pt_sr[0].vy = ptvy[nmol];
                        pmesg_sr[8].pt_sr[0].vz = ptvz[nmol];
                        pmesg_sr[8].pt_sr[0].type = type[nmol];
                                        /* Copy particle into particle message
                                                sending structure #8. */
                        totmol -= 1;      /* Decrement for departure. */
                        ptx[nmol] = ptx[totmol];
                        pty[nmol] = pty[totmol];
                        ptvx[nmol] = ptvx[totmol];
                        ptvy[nmol] = ptvy[totmol];
                        ptvz[nmol] = ptvz[totmol];
                        type[nmol] = type[totmol];
                                        /* Copy last particle
                                         *  into departing particle's place. */
                        nmol -= 1;
                        pmesg_sr[8].num_sr = -2;
                                        /* This is the flag attached saying that the
                                         *  particle came from afar. */
                        sd.node = node_send;
                        sd.buf = (char *)&pmesg_sr[8];
                        sendb(&sd);
                        jumper += 1.;   /* Increment # of jumpers sent */
                        goto next_particle;


                        /* Normally, the particle is packaged in an array
                         *  which is sent to a neighbor. */
        num_stored = pmesg_sr[send_to].num_sr;
                        /* # pts already stored in structure. */
        pmesg_sr[send_to].pt_sr[num_stored].x = ptx[nmol];
        pmesg_sr[send_to].pt_sr[num_stored].y = pty[nmol];
        pmesg_sr[send_to].pt_sr[num_stored].vx = ptvx[nmol];
        pmesg_sr[send_to].pt_sr[num_stored].vy = ptvy[nmol];
        pmesg_sr[send_to].pt_sr[num_stored].vz = ptvz[nmol];
        pmesg_sr[send_to].pt_sr[num_stored].type = type[nmol];
                                /* Copy particle into particle message
                                        sending structure. */
```

```
            totmol -= 1; /* Switch in last particle and decrement for departure. */
            ptx[nmol] = ptx[totmol];
            pty[nmol] = pty[totmol];
            ptvx[nmol] = ptvx[totmol];
            ptvy[nmol] = ptvy[totmol];
            ptvz[nmol] = ptvz[totmol];
            type[nmol] = type[totmol];
                                    /* Copy last particle
                                     *  into departing particle's place. */
            nmol -= 1;
            pmesg_sr[send_to].num_sr += 1;
                                    /* # Packaged in array is now one greater. */
            if (pmesg_sr[send_to].num_sr == PTSENT) {
                                    /* If pt message is full, send package. */
                    pmesg_sr[send_to].num_sr = -1;
                                    /* Flag means this isnt last pt mesg from me */
                    sd.node = node_send;
                    sd.buf = (char *)&pmesg_sr[send_to];
                    sendb(&sd);
                    pmesg_sr[send_to].num_sr = 0;
                                    /* No pts left in pt messsage. */
            }
next_particle:
i=0;
      }
   }

            /* Finally,  send out particle arrays to all neighbors with whatever
             *  particles are in them. The .num_sr flag is 0 or >0 indicating
             *  that this will be the last message sent. */
   for (i=0; i<msg_rt; i++) {
            sd.node = neighbor[i];
            sd.buf = (char *)&pmesg_sr[i];
            sendb(&sd);
            pmesg_sr[i].num_sr = 0;
   }

   ttime[9] += (unsigned int)((unsigned int)clock()-tim0);
}
```

```
/* This routine receives particle messages from other, generally
 *  nearby, nodes until it has received a concluding message from each of its
 *  nearest neighbors.
 */

#include <cube/cosmic.h>
#include "2d.h"

extern unsigned long ttime[];

recv_pt()                                                              recv_pt
{
        int count_end, i;
        unsigned int tim0;
        MSGDESC rd;

        tim0 = clock();
        sdesc (&rd,0,0,10000, &pmesg_sr[0], sizeof(struct particle_sent_recv));
                        /* Receive particle message into pmesg_sr[0] which
                         *  for the present is unused. */
       count_end = 0;

       while ( count_end < msg_rt )  /* While I have not received messages
                                      *  from all my nearest neighbors.... */
       {
           recvb(&rd);      /* Receive first/next message */

           if (pmesg_sr[0].num_sr < 0) {/* Either this message is not
                                          *  the last from that neighbor
                                          *  or it is from a distant
                                          *  node */
               if (pmesg_sr[0].num_sr == -1)
                       pmesg_sr[0].num_sr = PTSENT;
                               /* It was from a neighbor but was not the
                                *  last and contained the full # of particles
                                *  permitted, PTSENT */
               else {pmesg_sr[0].num_sr = 1;
                               /* Message from distant node had 1 pt. */
                       if(pmesg_sr[0].pt_sr[0].x < mesh.xmark[locx] ||
                           pmesg_sr[0].pt_sr[0].x > mesh.xmark[locx+1] ||
                           pmesg_sr[0].pt_sr[0].y < mesh.ymark[locy] ||
                           pmesg_sr[0].pt_sr[0].y > mesh.ymark[locy+1]) {
                               print("recvng jmp pt outside node, it was discarded");
                               pmesg_sr[0].num_sr = 0;
                       }
                       if(locy < 2) print("recvng jmp pt");
               }
           }
           else  count_end += 1;          /* This was the last message from a neighbor */

                       /* Now read particles from message into internal
                        *  particle list and increment totmol */
           for (i=0; i<pmesg_sr[0].num_sr; i++) {
               ptx[totmol] = pmesg_sr[0].pt_sr[i].x;
               pty[totmol] = pmesg_sr[0].pt_sr[i].y;
               ptvx[totmol] = pmesg_sr[0].pt_sr[i].vx;
               ptvy[totmol] = pmesg_sr[0].pt_sr[i].vy;
               ptvz[totmol] = pmesg_sr[0].pt_sr[i].vz;
               type[totmol] = pmesg_sr[0].pt_sr[i].type;
               totmol += 1;
              if(totmol == NM) {print("totmol > NM\n"); goto too_many;}
           }
       }
too_many:
```

```
pmesg_sr[0].num_sr = 0;
ttime[11] +=(unsigned int)((unsigned int)clock()-tim0);
```

```
/* This is the key routine which calculates collisions between particles.
 *  Particles are chosen at random from within the cell based on their
 *  probability of collision. This particular version, coll.3d_IDSMC_multispec, does
 *  the collisions based on the 3D integer method and assumes
 *  that the velocities have been defined as integers.
 *  It also permits multi-mass particle collisions. */

#include <cube/cubedef.h>
#include "2d.h"

extern unsigned long ttime[];
int npart_cell, npsum_cell;

coll()                                                                coll
{
          int i, j, ny, nx;
          int l, m, b, c, checksame, checkobst, check_even, sum_check;
          int sum_u, sum_v, sum_w;
          int ult, umt, vlt, vmt, wlt, wmt;
          int coll_type, nlite, nheav;
          int num, octant, vx, vy, vz, vt;
          int vrx, vry, vrz;
          int vr_sq;
          int par, shift, Enum_over_3;
          unsigned int tim0;
          float vr;
          float act_sites_per_cell, nom_sites_per_cell, nomcellvol;
          float nom_vol_per_site, xh, xl, yh, yl;
          float a;
          float collfac;
          float compensat;
          extern float numberc1, numberc2, numberc12;
          extern int sphere_radius_sq;

          tim0 = clock();

          if(node == 0) if(VELDIM != 3 || VTYPE != 1)
                    print("USING WRONG VERSION OF COLL");

          nomcellvol = (in.xm_glob-in.x_left)*(in.ym_glob-in.y_bot)
                    /((float)NDYMAX*(float)NDXMAX*(float)in.ncy*(float)in.ncx);
                                        /* ....Nominal cell volume */
          nom_sites_per_cell = (in.xm_glob-in.x_left)*(in.ym_glob-in.y_bot)
                    /((float)NDYMAX*(float)NDXMAX*(float)in.ncy*(float)in.ncx)/in.dtm;

for ( nx = 0; nx < in.ncx; nx++ ) { /* For all cells in X direction */
     for ( ny = 0; ny < in.ncy; ny++ ) { /* For all cells in Y direction */
          for(coll_type=0; coll_type<4; coll_type++) {
                                        /* for the 4 types of multi-spec colls
                                         *  type 0 -> lite/lite
                                         *  type 1 -> lite/heavy
                                         *  type 2 -> heavy/lite
                                         *  type 3 -> heavy/heavy */
                    if ( ct[coll_type][nx][ny] > time ) continue;
                    if(coll_type > 0 && npart_h[nx][ny] < 2) { /* Must be > 2 heavies */
                              ct[coll_type][nx][ny] += in.dtm;
                              continue;
                    }
                    else if ( npart[nx][ny] < 2 ) { /* Must be > 2 lights */
                              ct[coll_type][nx][ny] += in.dtm;
                              continue;
                    }
```

```
                /* We first must determine the volume represented by this cell. */
                if(LATICE != 0) { /* particles on latice sites decides the cell volume. */
                        xh = (int)((out.cellx[nx] + 0.5 * chx)/in.dtm) * in.dtm,
                        xl = (int)((out.cellx[nx] - 0.5 * chx)/in.dtm) * in.dtm,
                        yh = (int)((out.celly[ny] + 0.5 * chy)/in.dtm) * in.dtm;
                        yl = (int)((out.celly[ny] - 0.5 * chy)/in.dtm) * in.dtm;
                        /* these are the positions of the left, right, top and
                         * and bottom rows and columns of lattice sites */
                        act_sites_per_cell = (int)((xh-xl)/in.dtm)*(int)((yh-yl)/in.dtm);
                        /* = the actual number of sites in the cell */
                        cell_vol = act_sites_per_cell * nomcellvol / nom_sites_per_cell;
                }
                else  cell_vol = chx * chy; /* Possible to have integer particles
                                             *  without spatial lattice */
                collfac = 2.0 * cell_vol / cxs; /* operations combined it save time */
                npart_cell = npart[nx][ny]; /* so no need to use an array repeatedly */
                npsum_cell = npsum[nx][ny]; /* so no need to use an array repeatedly */

  do {                          /* Repeat sample coll's till ct[nx][ny] > time */
     checksame = 0;             /* So can only try for collision 100 times */
try_new_pair:
     do {
        checksame++;            /* Choose 2 particles from cell [nx][ny] */
                                /* Calculate rel velocity & collision time */
        sum_check = 1;

        switch(coll_type){
          case 0: { /* lite-lite collision */
                for ( ; ; ) {
                        l = pick_lite();
                        m = pick_lite();
                        if ( m != l ) break;      /* two different particles */
                }
                break;
          }

          case 1: { /* lite-heavy collision */
                l = pick_lite();
                m = pick_heavy();
                break;
          }

          case 2: { /* heavy-lite collision */
                l = pick_heavy();
                m = pick_lite();
                break;
          }
          case 3: {/* heavy-heavy collision */
                for ( ; ; ) {
                        l = pick_heavy();
                        m = pick_heavy();
                        if ( m != l ) break;
                }
                break;
          }
        }

        vrx = ptvx[m] - ptvx[l];
        vry = ptvy[m] - ptvy[l];
        vrz = ptvz[m] - ptvz[l];
                /* vrx, y, z = components of collision pair relative vel */
        vr_sq = vrx*vrx + vry*vry + vrz*vrz; /* relative speed squared*/
        vrmaxcount[nx][ny]++;
        if(vr_sq > recentvrmax_sq[nx][ny]) recentvrmax_sq[nx][ny] = vr_sq;
        if(vrmaxcount[nx][ny] > 20) { /* Update every 20 times */
                vrm_sq[nx][ny] = (recentvrmax_sq[nx][ny]+vrm_sq[nx][ny]) * 0.5;
                        /* = avrg of most recent max vel and old such value */
                recentvrmax_sq[nx][ny] = 0.;            /* reset */
                vrmaxcount[nx][ny] = 0;                 /* reset */
        }
        /* above was to update the max relative velocity as the temperature drops
                while this next statement updates as the temperature rises */
        if ( vr_sq > vrm_sq[nx][ny] ) vrm_sq[nx][ny] = vr_sq; /* Max value of cell rel vel*/

        if(pty[m] < in.obs_height && pty[l] < in.obs_height)
           if((ptx[m] < in.obs_a && ptx[l] > in.obs_a) ||
              (ptx[m] > in.obs_a && ptx[l] < in.obs_a)) sum_check = 0;
                /* sum_check is set = 0 if particles are in the same cell but
                 *  are on opposite sides of the obstacle, if present */

        if(sum_check != 0) {
                sum_check = 0;
                a = rand();
                if(vr_sq/vrm_sq[nx][ny] > a*a) sum_check = 1;
        }
        if(checksame >= 100) sum_check = 1;

    } while (sum_check == 0);



        if(checksame >= 100){
                ct[coll_type][nx][ny] += in.dtm;
                print("Exceeded 100, vr=%d, vrm=%f\n",vr_sq,vrm_sq[nx][ny]);
                goto checked_over_onehundred;
        } /* this should almost never happen */

        vr = sqrt((float)vr_sq); /* The relative speed */
        nlite  = npart[nx][ny] - npart_h[nx][ny];
        nheav  = npart_h[nx][ny];
        if(coll_type==0) ct[0][nx][ny] += collfac / ((float)nlite * (float)nlite * vr);
        else if(coll_type==1) ct[1][nx][ny] += collfac / ((float)nlite * (float)nheav * vr);
        else if(coll_type==2) ct[2][nx][ny] += collfac / ((float)nlite * (float)nheav * vr);
        else if(coll_type==3) ct[3][nx][ny] += collfac / ((float)nheav * (float)nheav * vr);
        /* updated cell time */


        if(vr_sq > sphere_radius_sq || Enum[vr_sq] == 0) {
                /* table is too small */
                print("vr_sq=%d,vel=(%d %d %d)%d (%d %d %d)%d: rejected this collision",
                vr_sq,ptvx[m],ptvy[m],ptvz[m],mass_of_type[type[m]],
                ptvx[l],ptvy[l],ptvz[l],mass_of_type[type[l]]);
                goto checked_over_onehundred;
        }

        if(mass_of_type[type[l]] != mass_of_type[type[m]]) {
                /* particles have a different mass */
                numberc12 += 1; /* the number of inter-species collisions */
                diff_species_collision(l,m,vr_sq);
                goto skip_same_species;
        }
        else {
                if(mass_of_type[type[l]] == 1 || mass_of_type[type[m]] == 1) numberc1 += 1;
                else numberc2 += 1; /* the number of heavy/heavy collisions */
            }

            sum_u = ptvx[m] + ptvx[l]; /* twice the mean u velocity */
            sum_v = ptvy[m] + ptvy[l]; /* twice the mean v velocity */
            sum_w = ptvz[m] + ptvz[l]; /* twice the mean w velocity */

pick_num:
            par = parity(sum_u,sum_v,sum_w); /* determine collision parity */
           if(par != 3){
                    Enum_over_3 = Enum[vr_sq] / 3; /* 1/3 of number of points */
                    shift = par * Enum_over_3;
                    num = shift + 1 + (int)(Enum_over_3*rand());
                            /* num is from correct third of the table */
           }
           else  num = ran_int_le(Enum[vr_sq]); /* any portion is valid */
           if(num == 0) goto pick_num; /* 0 is not valid */


            octant=ranbits(3);

            switch(octant) { /* reflect point across UVW = 0 planes randomly */
               case 0: {vx=pE[vr_sq][num].Ex; vy=pE[vr_sq][num].Ey; vz=pE[vr_sq][num].Ez; break;}
               case 1: {vx= -pE[vr_sq][num].Ex; vy=pE[vr_sq][num].Ey; vz=pE[vr_sq][num].Ez; break;}
               case 2: {vx= -pE[vr_sq][num].Ex; vy= -pE[vr_sq][num].Ey; vz=pE[vr_sq][num].Ez; break;}
               case 3: {vx= -pE[vr_sq][num].Ex; vy= -pE[vr_sq][num].Ey; vz= -pE[vr_sq][num].Ez; break;}
               case 4: {vx= -pE[vr_sq][num].Ex; vy=pE[vr_sq][num].Ey; vz= -pE[vr_sq][num].Ez; break;}
               case 5: {vx=pE[vr_sq][num].Ex; vy= -pE[vr_sq][num].Ey; vz=pE[vr_sq][num].Ez; break;}
               case 6: {vx=pE[vr_sq][num].Ex; vy= -pE[vr_sq][num].Ey; vz= -pE[vr_sq][num].Ez; break;}
               case 7: {vx=pE[vr_sq][num].Ex; vy=pE[vr_sq][num].Ey; vz= -pE[vr_sq][num].Ez; break;}
                       default: break;
            }

            ult = sum_u + vx; /* twice the final u velocity of particle l */
            umt = sum_u - vx;
            vlt = sum_v + vy;
            vmt = sum_v - vy;
            wlt = sum_w + vz;
            wmt = sum_w - vz;

            ptvx[l] = ult>>1; /* >> is faster than /2 or *0.5 */
            ptvx[m] = umt>>1;
            ptvy[l] = vlt>>1;
            ptvy[m] = vmt>>1;
            ptvz[l] = wlt>>1;
            ptvz[m] = wmt>>1;


skip_same_species:
checked_over_onehundred:
           vry = b;

    } while (ct[coll_type][nx][ny] < time );
    } /* For 4 coll types */
  } /* over all y cells */
 } /* over all x cells */
    ttime[13] +=(unsigned int)((unsigned int)clock()-tim0);
}

int sphere_radius, sphere_radius_sq;

make_LUT()
```

make_LUT

```
{

          int i, j, k, l, e, num, num_max, e_num_max;
          int half_sphere_radius_sq;
          int par, shift;
          long tot_num_pts;


          sphere_radius = LUTRAD;

          if(node == 0)
            print("Start to make velocity matrix, assume max rel vel will be %d\n", sphere_radius);

          sphere_radius_sq = sphere_radius*sphere_radius; /* square of sphere radius */
          half_sphere_radius_sq = sphere_radius_sq / 2.; /* half of that */
          num_max = tot_num_pts = 0; /* set counters to 0 */

          for(i=0; i<=sphere_radius_sq; i++) Enum[i] = 0;


          for(i= 0; i<=sphere_radius; i++) {
                    for(j= 0; j<=sphere_radius; j++) {
                              for(k= 0; k<=sphere_radius; k++) {
                                        e = i*i + j*j + k*k;
                                          /* square of sphere radius */
                                        if(e <= sphere_radius_sq) {
                                          /* only those spheres less than max */
                                                  Enum[e] += 1;
                                                            /* number of points */
                                                  tot_num_pts += 1;
                                                            /* total number of points */
                                                  if(Enum[e] > num_max) {
                                                            num_max = Enum[e];
                                                            e_num_max = e;
                                                  }
                                        }
                              }
                    }
          }


          /* To malloc one circle at a time so no block is > 64kbytes */
          /* use xmalloc on s2010 */
          for(i=0; i<=sphere_radius_sq; i++)
                    if(!(pE[i] = (struct Epoint *)malloc(Enum[i]*sizeof(struct Epoint))))
                    {print("malloc failed for i = %d",i); break;}
                    else pE[i]--; /* Want to access the 1st element using index 1 */



          /* place points onto newly malloc'ed circles */
          for(i= 0; i<=sphere_radius; i++) {
                    for(j= 0; j<=sphere_radius; j++) {
                              for(k= 0; k<=sphere_radius; k++) {
                                        e = i*i + j*j + k*k;
                                        if(e <= sphere_radius_sq) {
                                                  par = parity(i,j,k);
                                                  /* shift over in table based on point's parity */
                                                   if(par == 3) par = 0;
                                                   shift = par * Enum[e] / 3;
                                                   number_in_group[e][par] += 1;
                                                   num = shift + number_in_group[e][par];
                                                  pE[e][num].Ex = i;
                                                  pE[e][num].Ey = j;
```

```
                                                pE[e][num].Ez = k;
                                        }
                                }
                        }
                }
                num = 1;
                scramble(); /* scramble the LUT to avoid bias */

if(node == 0)print("Finish making vel matrix, num_max=%d, for e=%d, tot_num_pts=%ld\n",
                        num_max, e, tot_num_pts);

}
```

parity

```
parity(u,v,w)
/* find the parity of the given point */

int u,v,w;

{

        if(((u ^ w)&1) == 0 && ((u ^ v)&1) == 0) return(3); /* all the same parity */
        else if( ((u ^ v)&1) == 0 ) return(0); /* u and v the same but w different */
        else if( ((w ^ v)&1) == 0 ) return(2); /* w and v the same but u different */
        else return(1);                        /* u and w the same but v different */
}
```

scramble

```
scramble()
/* scramble the LUT to avoid bias */

{

        int i, j, k, l, e, num, num_max, e_num_max;
        int num1, num2, hold_x, hold_y, hold_z, par;
        long tot_num_pts;
        extern int sphere_radius, sphere_radius_sq,

        /* Now to scramble the E matrix to aid in getting really random numbers */
        /* pick 2 random points of the same parity and interchange them. Do this
         *   3 times, on average, for each point. That should be pretty random. */


        if(node == 0) print("I will now scramble the LUT");
        for(e=0; e<=sphere_radius_sq; e++) {
                if(Enum[e] > 3) for(i=1; i<3*Enum[e]; i++) {
                        choose_first_num  num1 = ran_int_le(Enum[e]);
                        if(num1 == 0) goto choose_first_num;

                        hold_x = pE[e][num1].Ex, hold_y = pE[e][num1].Ey; hold_z = pE[e][num1].Ez;
                        /* hold the first number in a temporary space */
                        par = parity(hold_x,hold_y,hold_z);

                        choose_second_num  num2 = ran_int_le(Enum[e]);
                        if(num2 == 0 || num2 == num1) goto choose_second_num;
                        if(parity(pE[e][num2].Ex,pE[e][num2].Ey,pE[e][num2].Ez) != par)
                                                        goto choose_second_num;

    pE[e][num1].Ex = pE[e][num2].Ex, pE[e][num1].Ey = pE[e][num2].Ey; pE[e][num1].Ez = pE[e][num2].Ez,
                        /* put 2nd point in place of first */

                        pE[e][num2].Ex = hold_x, pE[e][num2].Ey = hold_y; pE[e][num2].Ez = hold_z.
                        /* put 1st point in place of 2nd */
                }
        }

}


diff_species_collision(l,m,vr_sq)                                    diff_species_collision
int l, m, vr_sq;
{ /* to perform a multi species collision */

char vx, vy, vz, octant;
int num;
int lite, heavy;
int rel_x_plus_400, rel_y_plus_400, rel_z_plus_400;
int ii;

/* I will assume for now that the gas is a binary mixture and one
component is the light gas of mass = 1 */


        if(mass_of_type[type[l]] < mass_of_type[type[m]]) {lite = l; heavy = m;}
        else {lite = m; heavy = l;}

        rel_x_plus_400 = ptvx[heavy] - ptvx[lite] + 400;
        rel_y_plus_400 = ptvy[heavy] - ptvy[lite] + 400;
        rel_z_plus_400 = ptvz[heavy] - ptvz[lite] + 400;
                /* a 400 offset is used to avoid -ve values in []s */

pick_num2:
        num = ran_int_le(Enum[vr_sq]); /* pick any point on sphere, no parity */
        if(num == 0) goto pick_num2;


        octant=ranbits(3); /* to reflect randomly through UVW = 0 */

        if(octant & 1) vx=pE[vr_sq][num].Ex; else vx= -pE[vr_sq][num].Ex;
                /* to reflect randomly through U = 0 */
        if( heavy_lite_mod[rel_x_plus_400 - vx] != 0) {
                /* use a look up to avoid repeating modulus operation */
                vx = -vx; /* reflect through U = 0 */
                if( heavy_lite_mod[rel_x_plus_400 - vx] != 0) goto pick_num2;
                /* if still not good, pick a new point */
        }
        if(octant & 2) vy=pE[vr_sq][num].Ey; else vy= -pE[vr_sq][num].Ey;
                /* to reflect randomly through V = 0 */
        if( heavy_lite_mod[rel_y_plus_400 - vy] != 0) {
                vy = -vy; /* reflect through V = 0 */
                if( heavy_lite_mod[rel_y_plus_400 - vy] != 0) goto pick_num2;
        }
        if(octant & 4) vz=pE[vr_sq][num].Ez; else vz= -pE[vr_sq][num].Ez;
                /* to reflect randomly through W = 0 */
        if( heavy_lite_mod[rel_z_plus_400 - vz] != 0) {
                vz = -vz; /* reflect through W = 0 */
                if( heavy_lite_mod[rel_z_plus_400 - vz] != 0) goto pick_num2;
        }

        ptvx[lite] = ptvx[heavy] - heavy_lite_div[rel_x_plus_400 + vx*mass_of_type[type[heavy]]];
        ptvy[lite] = ptvy[heavy] - heavy_lite_div[rel_y_plus_400 + vy*mass_of_type[type[heavy]]];
        ptvz[lite] = ptvz[heavy] - heavy_lite_div[rel_z_plus_400 + vz*mass_of_type[type[heavy]]];
        ptvx[heavy] -= heavy_lite_div[rel_x_plus_400 - vx];
        ptvy[heavy] -= heavy_lite_div[rel_y_plus_400 - vy];
        ptvz[heavy] -= heavy_lite_div[rel_z_plus_400 - vz];
```

```
                        /* add new relative to mean velocities */
}

init_heavy_lite()                                                  init_heavy_lite
{  /* To initialize the vectors used later. It is only called once */

int i;

        if(node == 0 && (mass_of_type[0] + mass_of_type[1]) > 10)
                print("WARNING, there may be a problem with heavy_lite arrays being too small");
        for(i= -400; i<=400; i++) heavy_lite_mod[i+400] = i%(mass_of_type[0] + mass_of_type[1]);
                /* For the modulus operation */
        for(i= -400; i<=400; i++) heavy_lite_div[i+400] = i/(mass_of_type[0] + mass_of_type[1]);
                /* For the long division */
}

pick_lite()                                                        pick_lite
{  /* Pick a random light particle */
extern int npart_cell, npsum_cell;
int k;

pick_lite:   k = lcr[ran_int_le(npart_cell) + npsum_cell];
        if(mass_of_type[type[k]] != 1) goto pick_lite;
        else return(k);
}


pick_heavy()                                                       pick_heavy
{  /* Pick a random heavy particle */
extern int npart_cell, npsum_cell;
int k;

pick_heavy:   k = lcr[ran_int_le(npart_cell) + npsum_cell];
        if(mass_of_type[type[k]] != in.mass_h) goto pick_heavy;
        else return(k);
}
```

```
/* This is the key routine which calculates collisions between particles.
 *  Particles are chosen at random from within the cell based on their
 *  probability of collision. This particular version, coll.3d_Bird,
 *  performs the normal Bird type of 3D collisions and assumes that
 *  the velocities have been defined as floats. */

#include <cube/cubedef.h>
#include "2d.h"

extern unsigned long ttime[];
int npart_cell, npsum_cell;

coll()                                                              coll
{
        int ny, nx;
        int l, m,  checksame, sum_check;
        int coll_type, nlite, nheav;
        unsigned int tim0;
        float vr, vr_sq, a, bimp;
        float vrx, vry, vrz, mx, my, mz;
        float eps;
        float collfac;
        extern float pslipf, dtmp;
        extern float numberc1, numberc2, numberc12;

        tim0 = clock();

        if(node == 0) if(VELDIM != 3 || VTYPE != 0)
                print("USING WRONG VERSION OF COLL");

for ( nx = 0; nx < in.ncx; nx++ ) { /* For all cells in X direction */
  for ( ny = 0; ny < in.ncy; ny++ ) { /* For all cells in Y direction */
    for(coll_type=0; coll_type<4; coll_type++) {
                                        /* for the 4 types of multi-spec colls
                                         *  type 0 -> lite/lite
                                         *  type 1 -> lite/heavy
                                         *  type 2 -> heavy/lite
                                         *  type 3 -> heavy/heavy */
      . if( ct[coll_type][nx][ny] > time ) continue;
        if(coll_type > 0 && npart_h[nx][ny] < 2) { /* Must be > 2 heavies */
                ct[coll_type][nx][ny] += in.dtm;
                continue;
        }
        else if( npart[nx][ny] < 2 ) { /* Must be > 2 lights */
             ct[coll_type][nx][ny] += in.dtm;
             continue;
        }
        cell_vol = chx * chy; /* Volume of the cell */
        collfac = 2.0 * cell_vol / cxs; /* operations combined it save time */
        npart_cell = npart[nx][ny]; /* so no need to use an array repeatedly */
        npsum_cell = npsum[nx][ny]; /* so no need to use an array repeatedly */

  do {                  /* Repeat sample coll's till ct[coll_type][nx][ny]>time */
     checksame = 0;     /* So can only try for collision 100 times */
     do {
        checksame++;          /* Choose 2 particles from cell [nx][ny] */
                                /* Calculate rel velocity & collision time */
        sum_check = 1;

        switch(coll_type){
          case 0: { /* lite-lite collision */
                for ( ; ; ) {
                        l = pick_lite();
                        m = pick_lite(),
```

...coll

```
                    if ( m != l ) break;          /* two different particles */
            }
            break;
        }

        case 1: { /* lite-heavy collision */
            l = pick_lite();
            m = pick_heavy();
            break;
        }

        case 2: { /* heavy-lite collision */
            l = pick_heavy();
            m = pick_lite();
            break;
        }
        case 3: {/* heavy-heavy collision */
            for ( ; ; ) {
                    l = pick_heavy();
                    m = pick_heavy();
                    if ( m != l ) break;
            }
            break;
        }
    }


    vrx = ptvx[m] - ptvx[l];
    vry = ptvy[m] - ptvy[l];
    vrz = ptvz[m] - ptvz[l];
            /* vrx, y, z = components of collision pair relative vel */


    vr_sq = vrx*vrx + vry*vry + vrz*vrz; /* relative speed */


    vrmaxcount[nx][ny]++;
    if(vr_sq > recentvrmax_sq[nx][ny]) recentvrmax_sq[nx][ny] = vr_sq;
    if(vrmaxcount[nx][ny] > 20) { /* Update every 20 times */
            vrm_sq[nx][ny] = (recentvrmax_sq[nx][ny]+vrm_sq[nx][ny]) * 0.5;
                    /* = avrg of most recent max vel and old such value */
            recentvrmax_sq[nx][ny] = 0.;      /* reset */
            vrmaxcount[nx][ny] = 0;              /* reset */
    }
    /* above was to update the max relative velocity as the temperature drops
            while this next statement updates as the temperature rises */

    if ( vr_sq > vrm_sq[nx][ny] ) vrm_sq[nx][ny] = vr_sq; /* Max value of cell rel vel*/

    if(pty[m] < in.obs_height && pty[l] < in.obs_height)
      if((ptx[m] < in.obs_a && ptx[l] > in.obs_a) ||
         (ptx[m] > in.obs_a && ptx[l] < in.obs_a)) sum_check = 0;
            /* sum_check is set = 0 if particles are in the same cell but
             *  are on opposite sides of the obstacle, if present */

    if(sum_check != 0) {
            sum_check = 0;
            a = rand();
            if(vr_sq/vrm_sq[nx][ny] > a*a) sum_check = 1;
    }
    if(checksame >= 100) sum_check = 1;

} while (sum_check == 0);
```

```
        if(checksame >= 100){
                ct[coll_type][nx][ny] += in.dtm;
                print("Exceeded 100, vr_sq=%f, nx=%d, ny=%d, vrm_sq[][]=%f\n",
                        vr_sq,nx,ny,vrm_sq[nx][ny]);
                goto checked_over_onehundred;

        } /* this should almost never happen */

        vr = sqrt(vr_sq);                       /* The relative speed */

        nlite  = npart[nx][ny] - npart_h[nx][ny];               /* # of lights */
        nheav  = npart_h[nx][ny];
        if(coll_type==0) ct[0][nx][ny] += collfac / ((float)nlite * (float)nlite * vr);
        else if(coll_type==1) ct[1][nx][ny] += collfac / ((float)nlite * (float)nheav * vr);
        else if(coll_type==2) ct[2][nx][ny] += collfac / ((float)nlite * (float)nheav * vr);
        else if(coll_type==3) ct[3][nx][ny] += collfac / ((float)nheav * (float)nheav * vr);
        /* updated cell time */



        if(mass[l] != mass[m]) {
                numberc12 += 1;                 /* Type 1 or 2, heav/lit or lit/heav */
                diff_species_collision(l,m,vr);
                goto skip_same_species;
        }
        else if(mass[l] == 1 || mass[m] == 1) numberc1 += 1; /* type 0 */
              else numberc2 += 1;                      /* type 3 */

        bimp = 1. - 2.*rand();          /* impact parameter, >0, <1 */
        eps = pi2 * rand();             /* The azimuth angle, 0 to 2pi */

        a = sqrt(1. - bimp*bimp);
        vrx = 0.5 * bimp * vr;
        vry = 0.5 * a*cos(eps)*vr;
        vrz = 0.5 * a*sin(eps)*vr;


        /* Reset post collision velocity */

        mx = 0.5 * (ptvx[m] + ptvx[l]);
        my = 0.5 * (ptvy[m] + ptvy[l]);
        mz = 0.5 * (ptvz[m] + ptvz[l]);

        ptvx[m] = (mx + vrx);
        ptvx[l] = (mx - vrx);
        ptvy[m] = (my + vry);
        ptvy[l] = (my - vry);
        ptvz[m] = (mz + vrz);
        ptvz[l] = (mz - vrz);

skip_same_species:
checked_over_onehundred:
        vry = a;

    } while (ct[coll_type][nx][ny] < time );
    } /* For 4 coll types */
   } /* over all y cells */
} /* over all x cells */
   ttime[13] +=(unsigned int)((unsigned int)clock()-tim0);
}

make_LUT()
```

```
{  /* not used for DSMC */
print("ERROR, need to use IDSMC version of coll");
}
scramble()                                                          scramble
{  /* not used for DSMC */
}
init_heavy_lite()                                                   init_heavy_lite
{  /* not used for DSMC */
}

diff_species_collision(i,j,vr_t)                                    diff_species_collision
int  i,j;
float  vr_t;
{

float  sum_of_mass,  rel_mass_i,  rel_mass_j;
float  mx,  my,  mz;
float  vrx,  vry,  vrz;
float  a,  bimp,  eps;


          sum_of_mass  =  mass[i]  +  mass[j];
          rel_mass_i  =  mass[i]/sum_of_mass;
          rel_mass_j  =  mass[j]/sum_of_mass;
          mx  =  ptvx[i]*rel_mass_i  +  ptvx[j]*rel_mass_j;
          my  =  ptvy[i]*rel_mass_i  +  ptvy[j]*rel_mass_j;
          mz  =  ptvz[i]*rel_mass_i  +  ptvz[j]*rel_mass_j;

          bimp  =  1.  -  2.*rand();      /* impact parameter, >0, <1 */
          eps  =  pi2  *  rand();         /* The azimuth angle, 0 to 2pi */

         a  =  sqrt(1.  -  bimp*bimp);
         vrx  =   bimp  *  vr_t;
         vry  =   a*cos(eps)*vr_t;
         vrz  =   a*sin(eps)*vr_t;

          ptvx[i]  =  mx  +  vrx*rel_mass_j;
          ptvx[j]  =  mx  -  vrx*rel_mass_i;
          ptvy[i]  =  my  +  vry*rel_mass_j;
          ptvy[j]  =  my  -  vry*rel_mass_i;
          ptvz[i]  =  mz  +  vrz*rel_mass_j;
          ptvz[j]  =  mz  -  vrz*rel_mass_i;

}

pick_lite() /* Pick a light particle from the list of particles */  pick_lite
{
extern  int  npart_cell,  npsum_cell;
int  k;

     for(k  =  lcr[ran_int_le(npart_cell)  +  npsum_cell],
                    k  !=  1;  k  =  lcr[ran_int_le(npart_cell)  +  npsum_cell]);
     return(k);
}


pick_heavy() /* Pick a heavy particle from the list of particles .*/ pick_heavy
{
extern  int  npart_cell,  npsum_cell;
int  k;

     for(k  =  lcr[ran_int_le(npart_cell)  +  npsum_cell],
                    k  !=  in.mass_h;  k  =  lcr[ran_int_le(npart_cell)  +  npsum_cell]);
     return(k);
```

```
/* This routine sends/receives data, creates the new mesh and then sends
 *  the new mesh from node zero into the other nodes.
 *  It then resorts the particles to fit into the new
 *  mesh   */

#include <cube/cosmic.h>
#include "2d.h"

extern unsigned long ttime[];
remesh(ns,np)                                                              remesh
int ns,np;
{

        int i;
        unsigned int tim0;
        MSGDESC new_mesh;


        sdesc (&new_mesh, 0, 0, 12345, &mesh, sizeof(struct meshflt));

        send_out_scale(ns,np);  /* Send scale vectors up the columns
                                 * and across the rows to node
                                 * zero. From send_out_scale call cube_mkmesh
                                 * if node is node zero */
        tim0 = clock();
        if(node != 0) recvb(&new_mesh);  /* Recieve new mesh from node zero
                                                  */

        chx = (mesh.xmark[locx+1]-mesh.xmark[locx])/in.ncx;
        chy = (mesh.ymark[chsize-locy]-mesh.ymark[chsize-locy-1])/in.ncy;

        for(i=0; i<in.ncx; i++) out.cellx[i] = chx*(0.5+i)+mesh.xmark[locx];
        for(i=0; i<in.ncy; i++) out.celly[i] = chy*(0.5+i) + mesh.ymark[chsize-locy-1];
                /* Update cell sizes and positions with new mesh data */

        ttime[15] +=(unsigned int)((unsigned int)clock()-tim0);


                /* The following portion which sorts the particles back onto
                 *  the new mesh closely reflects the particle send/receive
                 *  section of cube_main()
                 */
        step_type += 1;
        recv_ok();                  /* Recieve signal that ready to receive particles
                                       from neighbor nodes */
        step_type -= 1;

        send_pt();                  /* Send appropriate particles to neighbors */
        recv_pt();                  /* Receive particles from neighbors */
        send_ok();                  /* Send signal of ready to receive particles
                                     *  to neighbors. This signal will be
                                     *  recvd in recv_ok() at next time step
                                     *  in cube_main().
                                     */
        reset_local();              /* Here. sort newly received particles
                                     *  into the new cells.
                                     */
```

```
/* This routine sends the profile data up each column to the node above and
 * then passes them across to node 0. If I am node 0 I'll accumulate
 * the data and create a new mesh. Every node adds in its profile
 * info to the xscale and yscale vectors before passing them on.
 */

#include <cube/cosmic.h>
#include "2d.h"

long myxscale[NDXMAX*NXCELL + 1], myyscale[NDYMAX*NYCELL + 1];
extern unsigned long ttime[];

send_out_scale(ns,np)                                              send_out_scale
int ns, np;
{
        int i, nx, ny;
        unsigned int tim0;
        MSGDESC out_scale_sd;
        MSGDESC out_scale_rd;
        MSGDESC new_mesh;

        tim0 = clock();

       sdesc (&out_scale_sd, 0, 0, 1717, &scale, sizeof(struct meshlong)),
       sdesc (&out_scale_rd, 0, 0, 1717, &scale, sizeof(struct meshlong));
        sdesc (&new_mesh, 0, 0, 12345, &mesh, sizeof(struct meshflt));
                /* Set up message descriptor to send
                 *   output to node above me */

        for(i=0; i<NDXMAX*in.ncx; i++) scale.xscale[i] = myxscale[i] = 0;
        for(i=0; i<NDYMAX*in.ncy; i++) scale.yscale[i] = myyscale[i] = 0;
                /* Set both vectors to zero before the sumations begin */

        for(nx = 0; nx < in.ncx; nx++) {
                for(ny = 0; ny < in.ncy; ny++) {
                        myxscale[nx + in.ncx*locx] += npart[nx][ny];
                        myyscale[ny + in.ncy*(chsize-locy-1)] += npart[nx][ny];
                }

                /* Sum up number of particles in each row
                 *  and column of cells within my node */


        if (locy == NDYMAX - 1 && locy != 0) {
                        /* if I am the bottom node in column but not also the top
                         * I must be the first to send out my vectors. I first
                         * add my scale vectors to the field scale vectors */
                for(i=0; i<NDXMAX*in.ncx, i++) scale.xscale[i] += myxscale[i];
                for(i=0; i<NDYMAX*in.ncy, i++) scale.yscale[i] += myyscale[i];
                out_scale_sd.node = who(locx,locy-1),
                sendb(&out_scale_sd); /* send field vectors up the column of nodes*/
        }
        else { /* a general node (one not in the bottom row) must await the field
                        vectors from the next lowest node in the column */
            if(NDYMAX == 1) goto there_is_only_one_row;
            recvb(&out_scale_rd); /* add my scale vectors to the field scale vectors */
            for(i=0; i<NDXMAX*in.ncx, i++) scale.xscale[i] += myxscale[i];
            for(i=0; i<NDYMAX*in.ncy, i++) scale.yscale[i] += myyscale[i];
            if(locy != 0) { /* I am not a top row node. */
                out_scale_sd.node = who(locx,locy-1);
                sendb(&out_scale_sd); /* send field vectors up column */
```

```
               else {  /* I am a top row node. */
there_is_only_one_row:
                    if(node == who(NDXMAX-1, 0)) {  /* I am top right node. */
                         out_scale_sd.node = who(locx-1,0);
                         out_scale_sd.type = 1718;
                         sendb(&out_scale_sd);  /* send vectors to left */
                    }
                    else {  /* I am a top row node but not the one on the right. */
                         for(i=0; i<NDXMAX*in.ncx; i++) myxscale[i] = scale.xscale[i],
                         for(i=0; i<NDYMAX*in.ncy; i++) myyscale[i] = scale.yscale[i];
                                /* Store field vectors in local vectors. */
                         out_scale_rd.type = 1718;
                         recvb(&out_scale_rd);                /* Recvb from node to my right */
                         for(i=0; i<NDXMAX*in.ncx; i++) scale.xscale[i] += myxscale[i];
                         for(i=0; i<NDYMAX*in.ncy; i++) scale.yscale[i] += myyscale[i];
                                /* add my scale vectors to the field scale vectors */
                         if(node != 0) {
                                out_scale_sd.node = who(locx-1,0);
                                out_scale_sd.type = 1718;
                                sendb(&out_scale_sd);  /* send vectors to left*/
                         }
                         else {  /* I AM NODE 0 and
                                     * now the field scale vectors are complete and should
                                     * represent the number of particles in each row and column
                                     * of cells in the whole flow field */

                                if(ns == in.nstep-1) {  /* if cube is to send out data for
                                             * accumulation on the next time step, node 0
                                             * now sends out field scale vectors and the
                                             * most recent mesh (node boundaries) to the
                                             * host */
                                     out_scale_sd.node = HOST;
                                     sendb(&out_scale_sd);
                                     new_mesh.node = HOST;
                                     sendb(&new_mesh);
                                     recvb(&new_mesh);  /* The host will generate this mesh
                                                          *  (or recall an old one) and send
                                                          *  it back down to node 0. */
                                }
                                else cube_mkmesh(ns,np);
                                                /* In this case, node zero generates
                                                 *  the new mesh */

                                for(i=1; i<NNODES; i++) {
                                     new_mesh.node = i;
                                     sendb(&new_mesh);
                                                      /* Wherever this new mesh came from it is
                                                       * sent on to all nodes. */



          }
          ttime[16] +=(unsigned int)((unsigned int)clock()-tim0);
```

```
/* This routine creates a new mesh based on the particle distribution
 * it receives in the form of two field vectors, xscale and yscale.
 * It is called from send_out_scale.c from node 0 and
 * is nearly the same as host_mkmesh.c
 */

#include <cube/cosmic.h>
#include "2d.h"

float local_xmark[NDXMAX+1], local_ymark[NDYMAX+1];
          /* Local node bdrys, used only in mkmesh */
float dif_local_xmark[NDXMAX+1];
          /* Local node widths */
float temp_xmark[NDXMAX+1];
          /* Temporary node boundaries */

cube_mkmesh(ns,np)                                          cube_mkmesh
int ns, np;

{
          int i, j, k, i_begin, k_begin, dum;
          long numpercolmn, numperrow, sum, act_field_mol;
          float cellwidth, cellheight, stuf;
          float pistonpos;



          pistonpos = in.x_left + time * in.u_enter;
          if(pistonpos > in.pstop) pistonpos = in.pstop;
                                        /* To establish where the piston is: */


          act_field_mol = 0;            /* The actual # of molecules in flow field */
          for(i=0; i<NDXMAX*in.ncx; i++) act_field_mol += scale.xscale[i];
          numpercolmn = (long)(act_field_mol) / NDXMAX + 0.5;
          numperrow = (long)(act_field_mol) / NDYMAX  + 0.5;
                              /* These are the balanced number of particles in
                               *  each colmn/row of nodes. */


          /* The scale.(x,y)scale[] vectors contain the total number of partics.
           *  in each (column,row) of cells throughout the whole flow field.
           *  ie, scale.xscale[0] is the number of particles in the far left
           *  column of cells and should start off being about (NDYMAX*in.ncy*
           *  in.mc) or so. Scale.yscale[0] is the # of part. in bottom-most
           *  row of cells and should start off as (NDXMAX*in.ncx*in.mc.
           */

          for(i=0; i<=cwsize; i++) {  /* xscale is searched for its first
                                       *  non-zero member */
                    for(k=0; k<in.ncx; k++) {
                              if(scale.xscale[k + i*in.ncx] > 0) {
                                        i_begin = i; k_begin = k;
                                                  /* These contain the location
                                                   *  of the first nonzero column
                                                   */
                                        goto found_x_begin;


          }
          found_x_begin:

          local_xmark[0] = pistonpos;
```

```
                /* The LHS boundary is always placed at the present
                 *  piston location */
        sum = 0; /* As I scan from left to right along scale.xscale sum will
                 *  contain the total number of particles seen since the
                 *  first/last node boundary (local_xmark[j]) was placed
                 */
        j = 1;   /* j is just the index of the current node whose right boundary
                 *  I am looking for
                 */
        for(i=i_begin; i<cwsize; i++) {         /* now begin counting particles and
                                                 setting down a node boundary when sum
                                                 >= numpercolumn */
                cellwidth = (mesh.xmark[i+1] - mesh.xmark[i])/in.ncx;
                                /* the width of the current mesh cell */
                for(k=k_begin; k<in.ncx; k++) {
                        k_begin = 0;
                        if(scale.xscale[k + i*in.ncx] == 0) {
                                /* This is just for safety since except for
                                 *    possibly some of the leftmost columns of
                                 *    cells, none of the rest should be empty */
                                local_xmark[j] = mesh.xmark[i]+
                                                        k*cellwidth;
                                /* This will probably cause an error later
                                 *  but I'll try it anyway */
                                goto done_with_x;
                        }
                        if(numpercolmn >= sum + scale.xscale[k + i*in.ncx]) {
                                /* That is, if I have not yet counted up
                                 *  enough particles to lay down the right
                                 *  edge of this node...
                                 */
                                sum += scale.xscale[k + i*in.ncx];
                                /* Sum gets incremented by the number of
                                 *  particles in this column of cells.
                                 */
                                if(i==cwsize-1 && k==in.ncx-1) {
                                /* If something is not quite right and I've
                                 *  reached the far side of the field without
                                 *  having counted up enough particles, place
                                 *  the node boundary now anyway.
                                 */
                                        stuf = scale.xscale[k + i*in.ncx] /
                                            (float)(numpercolmn / in.ncx);
                                        if(stuf < 1) stuf = 1.;
                                            /* So I don't place the boundary
                                             *  short of the end wall.
                                             */
                                        local_xmark[j] = mesh.xmark[i]+
                                                (in.ncx-1.+stuf)*cellwidth;
                                }
                        }
                        else {  /* That is, I have counted enough particles
                                 *  to lay down a new boundary...
                                 */
                                stuf = (float)(numpercolmn-sum)/(float)
                                        (numpercolmn / in.ncx);
                                                /* stuf is the linear interpol.
                                                to the locat. of the node end */
                                local_xmark[j] = mesh.xmark[i]+
                                                ((float)k+stuf)*cellwidth;
                                        /* Lay down the new node boundary.
                                         */
                                if(k == k_begin)
                                        if(i != 0) local_xmark[j] =
                                                        mesh.xmark[i] + stuf
                                                        *(mesh.xmark[i]-mesh.xmark[i-1])/in.ncx;
                                                      else local_xmark[j] =
                                                        mesh.xmark[i] + stuf
                                                        *(mesh.xmark[i+1]-mesh.xmark[i])/in.ncx;
                                              /* If this happened to be inside the first
                                               *   column of cells, lay down the boundary at
                                               *   the right edge of this first column.
                                               */
                                              sum = scale.xscale[k+i*in.ncx] -
                                                                        (numpercolmn-sum);
                                                               /* sum is reduced by the # of
                                                               parts. taken by the last node */
                                              j += 1;
                                              /* now go on to the next (j) node
                                               */
                                }
                        }
                }
done_with_x:

        local_xmark[cwsize] = in.xm_glob;
                /* Set the most rightward node boundary
                 *   equal to the right edge of the flow
                 *   field */

        for(i=0; i<cwsize; i++) dif_local_xmark[i] = local_xmark[i+1] - local_xmark[i];
        /* Now smooth this mesh in X by averaging the width
         *   of each node with that of its neighbors. */

    for(j=0; j<in.hremesh; j++) { /* Smooth hremesh times */
        for(i=1; i<cwsize-1; i++) temp_xmark[i] =
                0.3333*(dif_local_xmark[i-1]+dif_local_xmark[i]+ dif_local_xmark[i+1]);
                        /* For nodes not on LHS or RHS */

        dif_local_xmark[0] = (dif_local_xmark[0] + dif_local_xmark[1]) * 0.5;
                        /* For node on LHS */
        dif_local_xmark[cwsize-1] = (dif_local_xmark[cwsize-1] +
                                         dif_local_xmark[cwsize-2]) * 0.5;
                        /* For node on RHS */
        for(i=1; i<cwsize-1; i++) dif_local_xmark[i] = temp_xmark[i];
    }

        for(i=1; i<cwsize; i++) local_xmark[i] =
                        local_xmark[i-1] + dif_local_xmark[i-1];


/* Now I have finished laying down node boundaries in the X direction and will
 *   go on to do exactly the same thing in the Y direction except that the
 *   Y direction is not smoothed. Thus, the Y direction
 *   proceedure which follows has few if any comments.
 */


        i_begin = k_begin = 0;

        sum = 0;
        j = 1;
        for(i=i_begin; i<chsize; i++) {
                cellheight = (mesh.ymark[i+1]-mesh.ymark[i])/in.ncy;
                for(k=k_begin; k<in.ncy; k++) {
                        k_begin = 0 ;
                        if(scale.yscale[k + i*in.ncy] == 0) {
```

...cube_mkmesh

```
                                        local_ymark[j] = mesh.ymark[i]+
                                                        k*cellheight;
                        print("yscale = 0 early,  k=%d, k_begin=%d i=%d, i_begin= %d",
                                        k,k_begin,i,i_begin);
                                        goto done_with_y;
                                }
                                if(numperrow >= sum + scale.yscale[k + i*in.ncy]) {
                                        sum += scale.yscale[k + i*in.ncy];
                                        if(i==chsize-1 && k==in.ncy-1) {
                                                stuf = scale.yscale[k + i*in.ncy] /
                                                    (float)(numperrow / in.ncy);
                                                if(stuf < 1.) stuf = 1.;
                                                local_ymark[j] = mesh.ymark[i]+
                                                        (in.ncy-1.+stuf)*cellheight;
                                        }
                                }
                                else {
                                        stuf = (float)(numperrow-sum)/(float)
                                                (numperrow / in.ncy);
                                        local_ymark[j] = mesh.ymark[i]+
                                                        ((float)k+stuf)*cellheight;
                                        if(k == k_begin)
                                                if(i != 0) local_ymark[j] =
                                                        mesh.ymark[i] + stuf
                                                  *(mesh.ymark[i]-mesh.ymark[i-1])/in.ncy;
                                                else local_ymark[j] =
                                                        mesh.ymark[i] + stuf
                                                  *(mesh.ymark[i+1]-mesh.ymark[i])/in.ncy;
                                        j += 1;
                                        sum = scale.yscale[k+i*in.ncy] -
                                                                (numperrow - sum);
                                }
                        }
                }
done_with_y:

                local_ymark[chsize] = in.ym_glob;

                        /* Now, in case a lattice is used, we
                         * don't want particles' sites to
                         * land exactly on a node boundary so
                         * we will shift the boundary slightly */
                if(LATICE == 1) {
                        for(i=0; i<=cwsize; i++)
                                if(local_xmark[i] == (int)local_xmark[i])
                                        local_xmark[i] += 0.01;
                        for(i=0; i<=chsize; i++)
                                if(local_ymark[i] == (int)local_ymark[i])
                                        local_ymark[i] += 0.01;
                }



                        /* Here update the actual mesh boundaries from the temporary
                         * variables used in this subroutine and reset the scale.
                         * (x,y)scale variables to zero.
                         */
                for(i=0; i<=cwsize; i++) {
                        mesh.xmark[i] = local_xmark[i];
                        if(i < cwsize) for(k=0; k<in.ncx; k++)
                                        scale.xscale[k+i*in.ncx] = 0;
                }
                for(i=0; i<=chsize; i++) {
                        mesh.ymark[i] = local_ymark[i];
```

```
            if(i < chsize) for(k=0; k<in.ncy; k++)
                                    scale.yscale[k+i*in.ncy] = 0;
}           /* Reset the mesh and scale arrays */

if(ns % (int)(in.cremesh*5) == 0) { /* print out the mesh for reference. */
   print("ns = %d, np = %d",ns,np);
   dum = chsize; if(cwsize > dum) dum = cwsize;
   for(i=0; i<=dum; i++) {
        if(i <= cwsize && i <= chsize)
            print("CUBE:i = %d, xmark = %f, ymark = %f",
                    i,mesh.xmark[i],mesh.ymark[i]);
        else if(i > cwsize)print("CUBE:i = %d, xmark = ......, ymark = %f",
                    i,mesh.ymark[i]);
              else print("CUBE:i = %d, xmark = %f, ymark = ......",
                    i,mesh.xmark[i]);
```

```
#ifndef lint
static char sccsid[] = "@(#)random.c          4.2       (Berkeley)         83/01/02";
#endif

#include        <cube/cosmic.h>
#include        "2d.h"

/*
 * random.c:
 * An improved random number generation package.  In addition to the standard
 * rand()/srand() like interface, this package also has a special state info
 * interface.  The initstate() routine is called with a seed, an array of
 * bytes, and a count of how many bytes are being passed in; this array is then
 * initialized to contain information for random number generation with that
 * much state information.  Good sizes for the amount of state information are
 * 32, 64, 128, and 256 bytes.  The state can be switched by calling the
 * setstate() routine with the same array as was initiallized with initstate().
 * By default, the package runs with 128 bytes of state information and
 * generates far better random numbers than a linear congruential generator.
 * If the amount of state information is less than 32 bytes, a simple linear
 * congruential R.N.G. is used.
 * Internally, the state information is treated as an array of longs; the
 * zeroeth element of the array is the type of R.N.G. being used (small
 * integer); the remainder of the array is the state information for the
 * R.N.G.  Thus, 32 bytes of state information will give 7 longs worth of
 * state information, which will allow a degree seven polynomial.  (Note: the
 * zeroeth word of state information also has some other information stored
 * in it — see setstate() for details).
 * The random number generation technique is a linear feedback shift register
 * approach, employing trinomials (since there are fewer terms to sum up that
 * way).  In this approach, the least significant bit of all the numbers in
 * the state table will act as a linear feedback shift register, and will have
 * period 2^deg - 1 (where deg is the degree of the polynomial being used,
 * assuming that the polynomial is irreducible and primitive).  The higher
 * order bits will have longer periods, since their values are also influenced
 * by pseudo-random carries out of the lower bits.  The total period of the
 * generator is approximately deg*(2**deg - 1); thus doubling the amount of
 * state information has a vast influence on the period of the generator.
 * Note: the deg*(2**deg - 1) is an approximation only good for large deg,
 * when the period of the shift register is the dominant factor.  With deg
 * equal to seven, the period is actually much longer than the 7*(2**7 - 1)
 * predicted by this formula.
 */



/*
 * For each of the currently supported random number generators, we have a
 * break value on the amount of state information (you need at least this
 * many bytes of state info to support this random number generator), a degree
 * for the polynomial (actually a trinomial) that the R.N.G. is based on, and
 * the separation between the two lower order coefficients of the trinomial.
 */

#define         TYPE_0          0               /* linear congruential */
#define         BREAK_0         8
#define         DEG_0           0
#define         SEP_0           0

#define         TYPE_1          1               /* x**7 + x**3 + 1 */
#define         BREAK_1         32
#define         DEG_1           7
#define         SEP_1           3
```

```
#define         TYPE_2          2               /* x**15 + x + 1 */
#define         BREAK_2         64
#define         DEG_2           15
#define         SEP_2           1

#define         TYPE_3          3               /* x**31 + x**3 + 1 */
#define         BREAK_3         128
#define         DEG_3           31
#define         SEP_3           3

#define         TYPE_4          4               /* x**63 + x + 1 */
#define         BREAK_4         256
#define         DEG_4           63
#define         SEP_4           1

/*
 * Array versions of the above information to make code run faster — relies
 * on fact that TYPE_i == i.
 */

#define         MAX_TYPES       5               /* max number of types above */

static  long            degrees[ MAX_TYPES ]    = { DEG_0, DEG_1, DEG_2,
                                                        DEG_3, DEG_4 };

static  long            seps[ MAX_TYPES ]       = { SEP_0, SEP_1, SEP_2,
                                                        SEP_3, SEP_4 };


/*
 * Initially, everything is set up as if from :
 *                      initstate( 1, &randtbl, 128 );
 * Note that this initialization takes advantage of the fact that srandom()
 * advances the front and rear pointers 10*rand_deg times, and hence the
 * rear pointer which starts at 0 will also end up at zero; thus the zeroeth
 * element of the state information, which contains info about the current
 * position of the rear pointer is just
 *          MAX_TYPES*(rptr - state) + TYPE_3 == TYPE_3.
 */

static  long            randtbl[ DEG_3 + 1 ]        = { TYPE_3,
                    0x9a319039, 0x32d9c024, 0x9b663182, 0x5da1f342,
                    0xde3b81e0, 0xdf0a6fb5, 0xf103bc02, 0x48f340fb,
                    0x7449e56b, 0xbeb1dbb0, 0xab5c5918, 0x946554fd,
                    0x8c2e680f, 0xeb3d799f, 0xb11ee0b7, 0x2d436b86,
                    0xda672e2a, 0x1588ca88, 0xe369735d, 0x904f35f7,
                    0xd7158fd6, 0x6fa6f051, 0x616e6b96, 0xac94efdc,
                    0x36413f93, 0xc622c298, 0xf5a42ab8, 0x8a88d77b,
                                0xf5ad9d0e, 0x8999220b, 0x27fb47b9 };

/*
 * fptr and rptr are two pointers into the state info, a front and a rear
 * pointer.  These two pointers are always rand_sep places aparts, as they cycle
 * cyclically through the state information.  (Yes, this does mean we could get
 * away with just one pointer, but the code for random() is more efficient this
 * way).  The pointers are left positioned as they would be from the call
 *                      initstate( 1, randtbl, 128 )
 * (The position of the rear pointer, rptr, is really 0 (as explained above
 * in the initialization of randtbl) because the state table pointer is set
 * to point to randtbl[1] (as explained below).
 */
```

```
static  long        *fptr           = &randtbl[ SEP_3 ];
static  long        *rptr           = &randtbl[ 1 ];


/*
 * The following things are the pointer to the state information table,
 * the type of the current generator, the degree of the current polynomial
 * being used, and the separation between the two pointers.
 * Note that for efficiency of random(), we remember the first location of
 * the state information, not the zeroeth.  Hence it is valid to access
 * state[-1], which is used to store the type of the R.N.G.
 * Also, we remember the last location, since this is more efficient than
 * indexing every time to find the address of the last element to see if
 * the front and rear pointers have wrapped.
 */

static  long        *state          = &randtbl[ -1 ];

static  long        rand_type       = TYPE_3;
static  long        rand_deg        = DEG_3;
static  long        rand_sep        = SEP_3;

static  long        *end_ptr        = &randtbl[ DEG_3 + 1 ];


/*
 * srandom:
 * Initialize the random number generator based on the given seed.  If the
 * type is the trivial no-state-information type, just remember the seed.
 * Otherwise, initializes state[] based on the given "seed" via a linear
 * congruential generator.  Then, the pointers are set to known locations
 * that are exactly rand_sep places apart.  Lastly, it cycles the state
 * information a given number of times to get rid of any initial dependencies
 * introduced by the L.C.R.N.G.
 * Note that the initialization of randtbl[] for default usage relies on
 * values produced by this routine.
 */

srandom( x )                                                          srandom

    unsigned        x;
{
        long            i, j;

        if( rand_type == TYPE_0 ) {
            state[ 0 ] = x;
        }
        else {
            j = 1;
            state[ 0 ] = x;
            for( i = 1; i < rand_deg; i++ ) {
                state[i] = 1103515245*state[i - 1] + 12345;
            }
            fptr = &state[ rand_sep ];
            rptr = &state[ 0 ];
            for( i = 0; i < 10*rand_deg; i++ )  rand();
```

```
/*
 * initstate:
 * Initialize the state information in the given array of n bytes for
 * future random number generation.  Based on the number of bytes we
 * are given, and the break values for the different R.N.G.'s, we choose
 * the best (largest) one we can and set things up for it.  srandom() is
 * then called to initialize the state information.
 * Note that on return from srandom(), we set state[-1] to be the type
 * multiplexed with the current value of the rear pointer; this is so
 * successive calls to initstate() won't lose this information and will
 * be able to restart with setstate().
 * Note: the first thing we do is save the current state, if any, just like
 * setstate() so that it doesn't matter when initstate is called.
 * Returns a pointer to the old state.
 */

char  *
initstate( seed, arg_state, n )                                              initstate

    unsigned            seed;                   /* seed for R. N. G. */
    char                *arg_state;             /* pointer to state array */
    int                 n;                      /* # bytes of state info */
{
        char            *ostate         = (char *)( &state[ -1 ] );

        if( rand_type == TYPE_0 ) state[ -1 ] = rand_type;
        else state[ -1 ] = MAX_TYPES*(rptr - state) + rand_type;
        if( n < BREAK_1 ) {
            if( n < BREAK_0 ) return;
            rand_type = TYPE_0;
            rand_deg = DEG_0;
            rand_sep = SEP_0;
        }
        else {
            if( n < BREAK_2 ) {
                rand_type = TYPE_1;
                rand_deg = DEG_1;
                rand_sep = SEP_1;
            }
            else {
                if( n < BREAK_3 ) {
                    rand_type = TYPE_2;
                    rand_deg = DEG_2;
                    rand_sep = SEP_2;
                }
                else {
                    if( n < BREAK_4 ) {
                        rand_type = TYPE_3;
                        rand_deg = DEG_3;
                        rand_sep = SEP_3;
                    }
                    else {
                        rand_type = TYPE_4;
                        rand_deg = DEG_4;
                        rand_sep = SEP_4;



        state = &( ( (long *)arg_state )[1] );                  /* first location */
        end_ptr = &state[ rand_deg ];           /* must set end_ptr before srandom */
        srandom( seed );
        if( rand_type == TYPE_0 ) state[ -1 ] = rand_type;
        else state[ -1 ] = MAX_TYPES*(rptr - state) + rand_type;
```

```
		return( ostate );
}


/*
 * setstate:
 * Restore the state from the given state array.
 * Note: it is important that we also remember the locations of the pointers
 * in the current state information, and restore the locations of the pointers
 * from the old state information.  This is done by multiplexing the pointer
 * location into the zeroeth word of the state information.
 * Note that due to the order in which things are done, it is OK to call
 * setstate() with the same state as the current state.
 * Returns a pointer to the old state information.
 */

char	*
setstate( arg_state )
```

setstate

```
	char			*arg_state;
{
		long		*new_state		= (long *)arg_state;
		long		type			= new_state[0]%MAX_TYPES;
		long		rear			= new_state[0]/MAX_TYPES;
		char			*ostate		= (char *)( &state[ -1 ] );

		if( rand_type  ==  TYPE_0  )  state[ -1 ] = rand_type;
		else  state[ -1 ] = MAX_TYPES*(rptr - state) + rand_type;
		switch( type ) {
		    case  TYPE_0:
		    case  TYPE_1:
		    case  TYPE_2:
		    case  TYPE_3:
		    case  TYPE_4:
			rand_type = type;
			rand_deg = degrees[ type ];
			rand_sep = seps[ type ];
			break;

		    default:
			print ( "error" );
		}
		state = &new_state[ 1 ];
		if(  rand_type  !=  TYPE_0  )  {
		    rptr = &state[ rear ];
		    fptr = &state[ (rear + rand_sep)%rand_deg ];
		}
		end_ptr = &state[ rand_deg ];			/* set end_ptr too */
		return( ostate );
}


/*
 * random:
 * If we are using the trivial TYPE_0 R.N.G., just do the old linear
 * congruential bit.  Otherwise, we do our fancy trinomial stuff, which is the
 * same in all ther other cases due to all the global variables that have been
 * set up.  The basic operation is to add the number at the rear pointer into
 * the one at the front pointer.  Then both pointers are advanced to the next
 * location cyclically in the table.  The value returned is the sum generated,
 * reduced to 31 bits by throwing away the "least random" low bit.
 * Note: the code takes advantage of the fact that both the front and
 * rear pointers can't wrap on the same call by not testing the rear
 * pointer if the front one has wrapped.
 * Returns a 31-bit random number.
 */

float
rand()                                                              rand
{
        long            i;
        float           j;

        if(  rand_type  ==  TYPE_0  )  {
            i = state[0] = ( state[0]*1103515245 + 12345 )&0x7fffffff;
        }
        else  {
            *fptr = *fptr + *rptr;
            i = (*fptr >> 1)&0x7fffffff;            /* chucking least random bit */
            if(  ++fptr  >=  end_ptr  )  {
                fptr = state;
                ++rptr;
            }
            else  {
                if(  ++rptr  >=  end_ptr  )  rptr = state;
            }
        }
        j = (float)i / 0x7fffffff;
                        /* TO GET A FLOATING POINT NUMBER 0<j<1 */
        return( j );
```

```
/****************************************************************************
* The function ranbits(b) return a random long integer in which the lower b
* bits is significant.  The function set_ranbits set the seed of the random
* number generator.
****************************************************************************

static long seed;

long ranbits(b)
     int b;
{
     register long tt, vv;
     register int bb;

     for(vv = 0, bb = b, tt = seed; bb--; )
           if(tt < 0) { tt ^= 9; tt <<= 1; vv <<= 1; vv |= 1; }
                 else {tt <<= 1; vv <<= 1;}

     seed = tt; return(vv);
}

set_ranbits(newseed) long newseed; { seed = newseed; }
```

```
/* This program returns a random integer between 0 and
the number specified. The number is generated by the
method of ranbits.c written by Wen-king Su. It is
essential that seed be a big random number (like 2345)
or it will take a long time before the sequences
become good and random. It is faster than random.c */


static long seed;

long ran_int_le(num)
     int num;
{
     register long tt, vv;
     unsigned long temp;

     try_again:
          for(temp = (num<<1), vv = 0, tt = seed; temp >>= 1; )
                  if(tt < 0) { tt ^= 9; tt <<= 1; vv <<= 1; vv |= 1; }
                  else {               tt <<= 1; vv <<= 1;            }

          seed = tt;
          if(vv > num) goto try_again;
          return(vv);
}

set_ran_int(newseed) long newseed; { seed = newseed; }
```

set_ran_int

```
# Teach make about .o86 files.

CFLAGS= -O -fswitch
CC = cch


HFILS  =  host_main.c host_set.c host_getdat.c host_dat.c host_mkchain.c acum.c\
          host_mkmesh.c grey.c  pr_safety.c

HOBJ  =   host_main.o host_set.o host_getdat.o host_dat.o host_mkchain.o acum.o\
          host_mkmesh.o grey.o pr_safety.o

GHOBJ  =  host_main.o host_set.o host_getdat.o host_dat.o host_mkchain.o acum.o\
          host_mkmesh.o grey.o pr_safety.o cosmic1.o cosmic2.o cosmic3.o

CFILS  =  2dcube.c cube_main.c setup.c grey.c cube_getdat.c neighbors.c initia.c\
          reset_local.c\
          send_ok.c sample.c send_out.c zero.c\
          move_pt.c recv_ok.c send_pt.c\
          recv_pt.c\
          coll.c send_out_scale.c cube_mkmesh.c cube_remesh.c\
          random.c ran_int_le.c ranbits.c


COBJ  =   cube_main.O86 setup.O86 grey.O86 cube_getdat.O86 neighbors.O86\
          initia.O86 reset_local.O86\
          send_ok.O86 sample.O86 send_out.O86 zero.O86\
          move_pt.O86 recv_ok.O86 send_pt.O86\
          recv_pt.O86\
          coll.O86 cube_remesh.O86 send_out_scale.O86\
          cube_mkmesh.O86 random.O86 ran_int_le.O86 2dcube.O86

GIOBJ  =  cube_main$(OO) setup$(OO) grey$(OO) cube_getdat$(OO) \
          neighbors$(OO)\
          initia$(OO) reset_local$(OO)\
          send_ok$(OO) sample$(OO) send_out$(OO) zero$(OO)\
          move_pt$(OO) recv_ok$(OO) send_pt$(OO)\
          recv_pt$(OO)\
          coll$(OO) cube_remesh$(OO) send_out_scale$(OO)\
          cube_mkmesh$(OO) random$(OO) 2dcube$(OO)\
          ranbits$(OO) ran_int_le$(OO) cosmic1$(OO)\
          cosmic2$(OO) cosmic3$(OO)

IOBJ  =   cube_main.o286 setup.o286 grey.o286 cube_getdat.o286 neighbors.o286\
          initia.o286 reset_local.o286\
          send_ok.o286 sample.o286 send_out.o286 zero.o286\
          move_pt.o286 recv_ok.o286 send_pt.o286\
          recv_pt.o286\
          coll.o286 cube_remesh.o286 send_out_scale.o286\
          cube_mkmesh.o286 random.o286 ran_int_le.o286 2dcube.o286

IOBJ2  =  cube_main.o386 setup.o386 grey.o386 cube_getdat.o386 neighbors.o386\
          initia.o386 reset_local.o386\
          send_ok.o386 sample.o386 send_out.o386 zero.o386\
          move_pt.o386 recv_ok.o386 send_pt.o386\
          recv_pt.o386\
          coll.o386 cube_remesh.o386 send_out_scale.o386\
          cube_mkmesh.o386 random.o386 ran_int_le.o386 2dcube.o386

GOBJ  =   cube_main.gh.o setup.gh.o grey.gh.o cube_getdat.gh.o neighbors.gh.o\
          initia.gh.o reset_local.gh.o\
          send_ok.gh.o sample.gh.o send_out.gh.o zero.gh.o\
          move_pt.gh.o recv_ok.gh.o send_pt.gh.o\
		recv_pt.gh.o\
		coll.gh.o cube_remesh.gh.o send_out_scale.gh.o\
		cube_mkmesh.gh.o random.gh.o ran_int_le.gh.o 2dcube.gh.o

all: cubemain hostmain

contour: contour.o graphic2.o
		cc -o contour ${CFLAGS} contour.o graphic2.o -lsuntool -lsunwindow -lpixrect -lm

contour_safe: contour_safe.o graphic2.o host_getdat.o host_dat.o grey.o
		cch -o contour_safe ${CFLAGS} contour_safe.o graphic2.o\
		host_getdat.o host_dat.o grey.o -lsuntool -lsunwindow -lpixrect -lm

hprofile_safe: hprofile_safe.o host_getdat.o host_dat.o grey.o
		cch -o hprofile_safe ${CFLAGS} hprofile_safe.o\
		host_getdat.o host_dat.o grey.o -lm -lcosmic

vprofile_safe: vprofile_safe.o host_getdat.o host_dat.o grey.o
		cch -o vprofile_safe ${CFLAGS} vprofile_safe.o\
		host_getdat.o host_dat.o grey.o -lm -lcosmic

pv_plot_safe: pv_plot_safe.o host_getdat.o host_dat.o grey.o
		cch -o pv_plot_safe ${CFLAGS} pv_plot_safe.o\
		host_getdat.o host_dat.o grey.o -lm -lcosmic

draw_safe_field: draw_safe_field.o host_getdat.o host_dat.o grey.o
		cch -o draw_safe_field ${CFLAGS} draw_safe_field.o\
		host_getdat.o host_dat.o grey.o -lm -lcosmic

readsafe: readsafe.o host_getdat.o host_dat.o pr_acum.o grey.o smooth_acum.o
		cch -o readsafe ${CFLAGS} readsafe.o host_getdat.o host_dat.o\
		pr_acum.o grey.o smooth_acum.o -lm -lcosmic

dcontour: dcontour.o contour_op.o
		cc -o dcontour ${CFLAGS} dcontour.o contour_op.o -lm

dcontour_safe: dcontour_safe.o contour_op.o host_getdat.o host_dat.o grey.o
		cch -o dcontour_safe ${CFLAGS} dcontour_safe.o\
		contour_op.o host_getdat.o host_dat.o grey.o -lm -lcosmic

tcol: tcol.o
		cch -o tcol -fswitch tcol.c ran_int_le.c -lm -lcosmic

cubemain: $(COBJ)
		cccos -o cubemain ${COBJ} -lmnode -lcosmic -m
		fix86 -s 41000 cubemain

cubemain.ipsc: $(IOBJ)
		ccipsc -o cubemain ${IOBJ} -lcosmic -lm -lmnode

cubemain.ipsc2: $(IOBJ)
		ccipsc2 -o cubemain ${IOBJ2} -lcosmic -lm -lmnode

cubemain.s2010: $(GIOBJ)
		ccs2010 -c cosmic3.c -Dcosmic_timeout=
		ccs2010 -o cubemain ${GIOBJ} -lcube -lm

cubemain.gh: $(GOBJ)
		ccgh -m68020 -f68881 -o cubemain ${GOBJ} -lm -lcosmic

$(COBJ) $(IOBJ):	2d.h

hostmain: $(HOBJ)
		cch -fswitch -o hostmain ${HOBJ} -lm -lcosmic
```

```
ginhostmain: $(GHOBJ)
		cch -fswitch -o ginhostmain ${GHOBJ} -lm -lcube

$(HOBJ):	host_2d.h

2d.h host_2d.h: 2d.def

laser:	host_2d.h 2d.def $(HFILS) 2d.h $(CFILS) makefile
		igrind -Pmaser $? &
		touch laser

listd:	host_2d.h 2d.def $(HFILS) 2d.h $(CFILS) makefile
		print $? &
		touch listd

listb:	host_2d.h 2d.def $(HFILS) 2d.h $(CFILS) makefile
		print $? &
		touch listb

lintc:
		linth	$(CFILS)

linth:
		linth	$(HFILS)

wcc:
		wc host_2d.h 2d.def $(HFILS) 2d.h $(CFILS) makefile
```