MONTE CARLO CALCULATION OF THE FLOW OF GRANULAR MATERIALS

Thesis by

Kazunori Kawasaki

In Partial Fulfillment
of the Requirements for the Degree of
Aeronautical Engineer

California Institute of Technology
Pasadena, California

1986

(Submitted 5 October 1985)

© 1985

Kazunori Kawasaki

All Rights Reserved

To my wife, Yumiko Kawasaki

ACKNOWLEDGMENTS

I would like to thank sincerely all those who have helped me in this research effort, which was undertaken under the guidance of Professor Bradford Sturtevant. His insight, guidance and patience during this research are very much appreciated. The research benefited from the contributions of many individuals at Caltech: Dr. David Frost and Mr. David Goldstein for their editorial assistance and correction of my English as well as stimulating discussions, Mr. Wen-King Su for the computer assistance and Mrs. Betty Wood for the drawings. The financial support provided by Nissan Motor Corporation is gratefully acknowledged.

Finally, I am grateful to my wife, my parents, my parents-in-law and my friends, at Caltech and elsewhere, who have contributed to this effort by their constant encouragement and support.

ABSTRACT

The flow of granular materials has been investigated theoretically using the Direct Simulation Monte Carlo (DSMC) method for rarefied gas flows implemented on the Caltech concurrent 64-node Cosmic Cube computer. A fundamental understanding of the behavior of a heavily loaded gas under conditions for which collisions among the solid particles suspended in the gas are frequent is very important for a variety of problems, e.g., grain explosions and the performance of metallized solid-propellant rockets. At one extreme of particulate flow (granular material flow), the effect of the interstitial fluid is negligible. In particular, the numerical method has been applied to the problem of sedimentation and channel flow.

Bird's method has been applied to granular material flows by using a hard, rough-sphere particle model and introducing restitution and slip coefficients for particle-particle and particle-boundary collisions, respectively. In the DSMC method, physical space is divided into many cells, each containing several simulated particles. The distance between particles is much greater than the particle diameter. Using the concurrent computer, cells are assigned singly or in groups to individual processors (nodes). Calculations of particle-particle collisions are carried out locally by each node and information is communicated between adjacent nodes. Using the concurrent computer has enabled powerful computational ability to be brought to bear on the DSMC calculation.

For a gas sedimentation calculation simulating the gravitational collapse of a uniform atmosphere, significant thermal and wave effects are observed. For flow in a channel at Mach number 2.76 and Reynolds number 32.6, differences are observed between the behavior of granular material flow and gas flow. For both cases significant "slip" at the

wall is observed. For the flow of granular material, the boundary layer is thin and the velocity reduction near the wall is small.

TABLE OF CONTENTS

Chapter	Title	Page
Chapter	Title	Page
	Copyright	ii
	Dedication	iii
	Acknowledgements	iv
	Abstract	v
	Table of Contents	vii
	List of Figures	ix
1.0	INTRODUCTION	1
2.0	DIRECT SIMULATION MONTE CARLO METHOD FOR GRANULAR MATERIAL FLOWS	5
2.1	Brief Description of the DSMC Method	5
2.2	Calculation of a Particle-Particle Collision	8
2.3	Structure and Summary of the Program	11
2.4	Normalization of parameters used in the following sections	13
3.0	PRELIMINARY INVESTIGATION	14
3.1	One-dimensional Sedimentation	14
3.2	Energy Dissipation and Transfer During a Collision	16
4.0	CONCURRENT FORMULATION FOR THE COSMIC CUBE	18
4.1	Description of the Caltech Cosmic Cube	18
4.2	Implementation of the DSMC Method	19
	4.2.1 Assignment of Nodes	20
	4.2.2 Assignment of Particles	21
	4.2.3 Communication between Nodes	21
	4.2.4 Restricted Particle Movement	22
	4.2.5 Collisions	22

	4.2.6 Host-Node Communication	23
	4.2.7 Data Accumulation	23
4.3	Program Structure	23
4.4	Load Balancing	24
4.5	Limitations of the Caltech Cosmic Cube	25
4.6	Optimal Formulation for the Cosmic Cube	25
5.0	CHANNEL FLOWS OF GRANULAR MATERIALS	26
5.1	Description of the model for channel flow without Gravity	27
	5.1.1 Geometry and Input Data	27
	5.1.2 Reynolds Number and Mach Number	29
5.2	Results and Discussion for the Channel Flow without Gravity	30
5.3	Result and Discussion for the Channel Flow with Gravity	32
6.0	SUMMARY AND CONCLUSIONS	33
	APPENDICES	36
A	NORMALIZATION OF PARAMETERS	36
B	EXACT EQUILIBRIUM SOLUTION	38
C	PROGRAM LISTING	41
	REFERENCES	90
	FIGURES	91

LIST OF FIGURES

Figure	Title	Page
2.1	Diagrams of particle-particle collisions	91
2.2	Flow chart for the sequential computations of the DSMC method	92
3.1(a)	Number density profile for one-dimensional sedimentation with a single type of particle with $\epsilon_p = 1.0$, $\epsilon_s = -1.0$, specular wall and $g = 0.1$	93
3.1(b)	Total internal energy profile for one-dimensional sedimentation with a single type of particle with $\epsilon_p = 1.0$, $\epsilon_s = -1.0$, specular wall and $g = 0.1$	94
3.2	Number density profile for one-dimensional sedimentation with two types of particles where $m_1/m_2 = 1.0$, $d_1/d_2 = 0.464$, with $\epsilon_p = 1.0$, $\epsilon_s = -1.0$, specular wall and $g = 0.1$	95
3.3	Number density profile for one-dimensional sedimentation with two types of particles where $m_1/m_2 = 0.1$, $d_1/d_2 = 1.0$ with $\epsilon_p = 1.0$, $\epsilon_s = -1.0$, specular wall and $g = 0.1$	96
3.4	Number density profile for one-dimensional sedimentation with two types of particles where $m_1/m_2 = 0.n1$ and $d_1/d_2 = 0.464$, with $\epsilon_p = 1.0$, $\epsilon_s = -1.0$, specular wall and $g = 0.1$	97
3.5	Total energy vs. time for different conditions of collision	98
3.6	Total translational and spin energy vs. time	99
	(a) $\epsilon_p = 0.6$, $\epsilon_s = 1.0$	99
	(b) $\epsilon_p = 0.6$, $\epsilon_s = -1.0$	99
	(c) $\epsilon_p = 1.0$, $\epsilon_s = 0.0$	99
	(d) $\epsilon_p = 0.6$, $\epsilon_s = 0.0$	99

4.1 The layout of nodes for minimizing the communication time between nodes 100

4.2 Typical inter-node motion of simulated particles 101

4.3 Message-advance system for sending output to HOST 102

4.4 Flow chart for the concurrent computations of the DSMC method 103

5.1 The simulated flow field and the node and cell layout 104

5.2(a) Number density profile of channel flow of granular material in the steady state with Re = 32.6, M = 2.76 and K_n = 0.125 (ϵ_p = 0.6, ϵ_s = 0.0, ϵ_w = 0.8, ϵ_{sw} = 0.0) 105

5.2(b) Temperature profile of channel flow of granular material in the steady state with Re = 32.6, M = 2.76 and K_n = 0.125 (ϵ_p = 0.6, ϵ_s = 0.0, ϵ_w = 0.8, ϵ_{sw} = 0.0) 106

5.2(c) Pressure profiles of channel flow of granular material in the steady state with Re = 32.6, M = 2.76 and K_n = 0.125 (ϵ_p = 0.6, ϵ_w = 0.0, ϵ_w = 0.8, ϵ_{sw} = 0.0) 107

5.2(d) Velocity field of channel flow of granular material in the steady state with Re = 32.6, M = 2.76 and K_n = 0.125 (ϵ_p = 0.6, ϵ_s = 0.0, ϵ_w = 0.8, ϵ_{sw} = 0.0) 108

5.3(a) Number density profile of channel flow of gas in the in the steady state with Re = 32.6, M = 2.76 and K_n = 0.125 (ϵ_p = 1.0, ϵ_s = -1.0, diffusely reflecting walls, T_w = 1) 109

5.3(b) Temperature profile of channel flow of gas in the steady state with Re = 32.6, M = 2.76 and K_n = 0.125 (ϵ_p = 1.0, ϵ_s = -1.0, diffusely reflecting walls, T_w = 1) 110

5.3(c)	Pressure profile of channel flow of gas in the steady state with Re = 32.6, M = 2.76 and K_n = 0.125 (ϵ_p = 1.0, ϵ_s = -1.0, diffusely reflecting walls, T_w = 1)	111
5.3(d)	Velocity field of channel flow of gas in the steady state with Re = 32.6, M = 2.76 and K_n = 0.125 (ϵ_p = 1.0, ϵ_s = -1.0, diffusely reflecting walls, T_w = 1)	112
5.4(a)	Number density profile of channel flow of intermediate material in the steady state with Re = 32.6, M = 2.76 and K_n = 0.125 (ϵ_p = 1.0, ϵ_s = 0.0, diffusely reflecting walls, T_w = 1)	113
5.4(b)	Tempertature profile of channel flow of intermediate material in the steady state with Re = 32.6, M = 2.76 and K_n = 0.125 (ϵ_p = 1.0, ϵ_s = 0.0, diffusely reflecting walls, T_w = 1)	114
5.4(c)	Pressure profile of channel flow of intermediate material in the steady state with Re = 32.6, M = 2.76 and K_n = 0.125 (ϵ_p = 1.0, ϵ_s = 0.0, diffusely reflecting walls, T_w = 1)	115
5.4(d)	Velocity field of channel flow of intermediate material in the steady state with Re = 32.6, M = 2.76 and K_n = 0.125 (ϵ_p = 1.0, ϵ_s = 0.0, diffusely reflecting walls, T_w = 1)	116
5.5	Comparison of velocity field between intermediate material and gas near the leading edge of diffusely reflecting wall with Re = 32.6, M = 2.76 and K_n = 0.125. Gas (ϵ_p = 1.0, ϵ_s = -1.0). Intermediate material (ϵ_p = 1.0, ϵ_s = 0.0), T_w = 1.	117
5.6	Sedimentation of gas in channel with Re = 16.3, M = 2.76 and K_n = 0.25, ϵ_p = 1.0, ϵ_s = -1.0 specular wall, g = 0.5	118

Chapter 1
INTRODUCTION

A fundamental understanding of the behavior of a heavily loaded gas, under conditions for which collisions among the solid particles suspended in the gas are frequent, is very important for a variety of problems, e.g., dust and grain explosions and the performance of metallized solid-propellant rockets. In such flows, both interstitial fluid effects, e.g., viscous drag and particle-wake interactions, and particle-particle collisions, must be treated. However, at one extreme of particulate flow, the effect of the interstitial fluid is negligible and momentum transfer occurs mainly through particle-particle collisions. A great deal of research has been carried out in the study of granular flows (e.g., Campbell and Brennen, 1985 and Savage and Sayed, 1984). For a complete and current review of the topic, the reader is referred to the paper by Savage. In the following, a granular material flow will be defined as one in which the particle-fluid interaction is negligible. Under this approximation, the model for a granular material is analogous to the molecular model of a gas in which a gas is treated as an assembly of solid particles. Although these are two completely different problems, one from molecular gasdynamics and the other from granular material flows, in some cases, e.g., rarefied flows, the same computational method may be used. Furthermore, even in the case of a two-phase flow such as a plume of a metalized solid-propellant rocket expanding into a vacuum, where both particle-particle collisions and the effect of the interstitial gas on the particles must be taken into account, the entire flow can be regarded as an assembly of different types of particles: small particles representing gas molecules and very large particles corresponding to particles of oxidized metals. Interstitial fluid effects may be modeled in terms of the forces exerted by the small particles on the larger particles. These flows are governed by the Boltzmann equation, derived in the context of molecular gas dynamics for a dilute gas. The mathematical difficulties associated

with the full Boltzmann equation generally preclude a direct approach that would lead to an exact analytical solution.

A conventional numerical approach to solving the Boltzmann equation would be to construct a finite-difference formulation. The distribution function, which is the only dependent variable, becomes six-dimensional in two-dimensional flows when particle spin is included, thus requiring an eight-dimensional array of points in phase space. In this case the number of operations required to calculate the term representing collisions is very large. Therefore, a direct finite-difference approach would require a prohibitive amount of computer time and storage capacity. In an alternative deterministic approach, the exact trajectory of each of the particles is calculated, taking into account some collision probability. For the same reasons, this method can be applied practically only to flows that contain a small number of particles. On the other hand, the computing requirement, especially the computing task for collisions, becomes manageable if probabilistic rather than deterministic procedures are adopted for the computation of collisions. This leads to the Direct Simulation Monte Carlo (DSMC) method, which was originally formulated by Graeme Bird in the 1960's to investigate low-density, high-speed fluid flows.

In the present work, the DSMC method has been applied to granular material flows where particle-particle collisions are assumed to be inelastic. Each of the hard, rough spherical particles has three components of translational velocity and three components of spin velocity. The method includes the introduction of restitution and non-slip coefficients for both particle-particle and particle-wall collisions. As a preliminary investigation, the DSMC method was applied to the one-dimensional, nonsteady gravitational collapse of an atmosphere and the result was compared to the analytical solution. Also, energy dissipation and transfer between translation and spin during a collision was investigated.

In order to extend the method to the solution of, e.g., two-phase or three-dimensional low-Reynolds-number, high-speed continuum flows, considerable ingenuity is required to bring greater computer power to bear efficiently. Even for the relatively simple problem of sedimentation, high-speed computers are required to obtain good statistical solutions. The DSMC method was implemented on the concurrent Cosmic Cube computer developed at Caltech by C. L. Seitz. The Cosmic Cube at Caltech runs in connection with a conventional host computer, e.g., a VAX, which has I/O capability in contrast with the individual nodes of the concurrent machine. Benchmark tests consistently show that a single processor (node) of the Cosmic Cube running at a clock rate of 5 MHz runs at 1/6th the speed of the same program compiled and run on a VAX11/780. Thus the 64-node Cosmic Cube should be expected to run at best about 10 times faster than the VAX11/780.

In the DSMC method, physical space is divided into many cells, each containing several particles. In a conventional (i.e., sequential) computer architecture, the collision calculations proceed from cell to cell, even though the calculations for each cell could, in principle, be done separately and concurrently. The subsequent motion of each of the particles is also treated separately. In the concurrent formulation of the method, cells are assigned singly or in groups to individual processors (nodes) of the Cosmic Cube. Particles that move physically from one cell to another are accounted, for using the message-sending facility of the computer. For an efficient formulation for the Cosmic Cube, it is important to minimize the time that nodes are idle or engaged in work that they would not do in conventional computations. In other words, it is important that the time used to send and receive messages between nodes is small compared with the computation time. It is also important for the work load to be evenly distributed between nodes.

In the following report, the framework of the DSMC method is discussed including the formulation of the method for the Cosmic Cube. As a typical problem of the flow of granular materials, a two-dimensional channel flow problem has been investigated using the Cosmic Cube. The flow of a granular material in a channel is compared to the channel flow of a gas (also calculated by the Cosmic Cube). Also, the two-dimensional sedimentation of gas in a channel has been investigated as the extension of the one-dimensional problem which was done in the preliminary investigation during the first half of the present work. Chapter 2 describes the principal idea behind the DSMC method and the mathematical framework used to calculate the granular material flows, i.e., the motion of particles following collisions. Also, this chapter describes the structure of the program implemented on a sequential calculation. Chapter 3 describes the results of the preliminary investigations including the transfer of spin/translational energy due to the particle-particle collisions and the one-dimensional sedimentation calculation. In the sedimentation calculation, time-evolved profiles of density and temperature are presented for a variety of material properties, e.g., particle diameter and mass. Chapter 4 describes a formulation of the DSMC method for the Cosmic Cube and presents a computational flowchart of a typical problem. Results and discussions of a two-dimensional channel flow calculation obtained using the Cosmic Cube are then given in Chapter 5. A summary of the main results of the present work and the conclusion are given in Chapter 6. Listings of the computer programs of the DSMC method for both the Cosmic Cube and the host processor are given in an appendix.

Chapter 2
DIRECT SIMULATION MONTE CARLO METHOD FOR GRANULAR MATERIAL FLOWS

2.1. Brief Description of the DSMC method.

This method is a computational technique for modeling a real gas at a molecular level with a sample of several thousand simulated molecules. The simulated flow field is divided into a network of such small cells that the state in each cell can be regarded as spatially almost uniform. The extent of each cell is usually taken to be of order λ, the mean-free path. When the gas is initially in thermodynamic equilibrium, the simulated molecules' thermal velocity components are assigned by sampling randomly from a Maxwellian distribution corresponding to the initial temperature. Two position and six velocity coordinates for each simulated molecule are stored. The boundaries are instantaneously inserted into the stream and the subsequent motion of the molecules is computed in the flow field within the imposed boundaries. The molecular paths between collisions are computed exactly but collisions are treated statistically. The calculation procedure consists of holding all simulated molecules motionless for a time interval Δt_m, small compared to the mean collision time per molecule, while collisions are computed everywhere in the flow field. Next the simulated molecules are allowed to move with their new velocities for the time interval Δt_m and are then held motionless in their new positions while another collision cycle takes place. This principle of uncoupling collisions from translational motion is the main characteristic of this method. Collisions are computed by statistical sampling as follows:

1) Pairs of simulated molecules are selected at random from a particular cell and retained for a collision with probability proportional to their relative velocity.

2) For pairs of simulated molecules which have been accepted for collision, a line of impact and an azimuthal angle in the center-of-mass reference frame are defined by selecting the magnitude and direction of an impact vector from appropriate distributions.

3) At each collision, a time counter for the cell, which was set as a random fraction of Δt_m in the initial state is advanced by the amount

$$\Delta t_c = \left[\frac{1}{\pi d^2 n c_r}\right]\left[\frac{2}{N_m}\right] , \tag{2.1}$$

where πd^2 = collision cross section, n = local number density, c_r = relative speed of the pair, and N_m = number of molecules in the cell.

4) When the time counters for all cells have advanced through Δt_m, the representative set of collisions for the time interval is complete.

This sampling scheme produces a collision frequency for pairs of molecules proportional to the product of the local number density and the relative velocity of the pair. Molecules move an average distance between collisions equal to the local mean-free path. It has been argued (Bird, 1963) that this collision process produces the correct collision frequency and mean-free path. Along any particle path, collisions can occur only at intervals that are integral multiples of Δt_c. Since Δt_c is small compared with the mean collision time, the resulting distortion of the path is small. More recently (Deshpande, 1977), refinements of the method have been proposed, but, for simplicity, the original formulation is retained here.

All macroscopic quantities of interest, e.g., number density, temperature associated with randomly fluctuating molecular velocities, flow velocity, and pressure, are obtained as averages of the randomly fluctuating molecular states within each cell. The statistics for all the calculated quantities can be improved by repeating the calculation many times and taking ensemble averages.

The direct simulation procedure is equivalent to a numerical solution of the Boltzmann equation. The Boltzmann equation for hard-sphere molecules may be written as the following:

$$\frac{\partial}{\partial t}(nf) + c\cdot\frac{\partial}{\partial r}(nf) + F\cdot\frac{\partial}{\partial c}(nf) = \pi d^2 \int_{-\infty}^{\infty} n^2(f'f_1' - ff_1)c_r dc_1 \quad , \quad (2.2)$$

where nf, c, c_r, F, and d, are the number density distribution function, local average velocity, magnitude of the relative velocity, body force, and effective molecular diameter, respectively, and primes refer to <u>post</u>-collision values. Implicit in the derivation of the Boltzmann equation (and therefore in the corresponding direct simulation method) is the assumption that the gas is dilute (Bird, 1976).

This equation can also be applied to dilute granular media. The left-hand side of the equation represents the process of following the paths of the individual particles. The collision integral on the right-hand side represents both the sampling of collision partners using the collision probability appropriate to the particular particle model, and the calculation of collisions at a rate consistent with the local collision frequency.

To extend the application of this method to granular material flows, inelastic collisions must be taken into account in the simulation. This was done by introducing restitution coefficients and slip coefficients in the collisions of rough-sphere particles, each of which has three components of translational velocity and three components of

angular velocity. The procedure for calculating the result of a particle-particle collision will now be discussed in more detail.

2.2. Calculation of a Particle-Particle Collision.

Pairs of simulated particles are successively selected at random from a cell and for each pair the probability of collision is evaluated such that the probability is proportional to the relative speed of the two particles. If a collision takes place, the relative location of the two particles is determined and the post-collision velocity of each particle is calculated.

Consider the collision between two simulated particles of the same type, denoted by L and M, in the center-of-mass coordinate system. The vector k connecting their centers at the moment of impact is determined by a miss distance impact parameter vector b, which has a magnitude b and azimuthal angle impact parameter ϵ (see Figure 2.1). The vector 1 is in the direction of the relative velocity vector c_r and joins the center of M to the plane through the center of L normal to the direction of c_r and 1.

The parameters b and ϵ are selected in such a manner that b is distributed between 0 and d, the simulated particle's diameter, with probability proportional to b itself, while ϵ is uniformly distributed between 0 and 2π. Thus b and ϵ can be calculated as follows by using successive random fractions R_f that are uniformly distributed between 0 and 1.

$$b = dR_f^{1/2} \tag{2.3}$$

$$\epsilon = 2\pi R_f \tag{2.4}$$

The simulated particles approach each other with a relative velocity with components U_0 along the vector k and V_0 and W_0 tangential to k. The particles L and M have three components of angular velocity,

$\omega_{L_{x0}}$, $\omega_{L_{y0}}$, $\omega_{L_{z0}}$, and $\omega_{M_{x0}}$, $\omega_{M_{y0}}$, $\omega_{M_{z0}}$, respectively. The collision results in an impulse J inclined at an angle θ with respect to k and an angle ϕ with respect to m normal to k (see Figure 2.1).

The equations governing the collision are:

$$U = U_0 - J\cos\theta \quad , \tag{2.5}$$

$$V = V_0 - J\sin\theta\cos\phi \quad , \tag{2.6}$$

$$W = W_0 - J\sin\theta\sin\phi \quad , \tag{2.7}$$

$$r\omega_{L_x} = r\omega_{L_{x0}} \quad , \tag{2.8}$$

$$r\omega_{M_x} = r\omega_{M_{x0}} \quad , \tag{2.9}$$

$$r\omega_{L_y} = r\omega_{L_{y0}} - \frac{J\sin\theta\sin\phi}{\beta^2} \quad , \tag{2.10}$$

$$r\omega_{M_y} = r\omega_{M_{y0}} - \frac{J\sin\theta\sin\phi}{\beta^2} \quad , \tag{2.11}$$

$$r\omega_{L_z} = r\omega_{L_{z0}} + \frac{J\sin\theta\cos\phi}{\beta^2} \quad , \tag{2.12}$$

$$r\omega_{M_z} = r\omega_{M_{z0}} + \frac{J\sin\theta\cos\phi}{\beta^2} \quad , \tag{2.13}$$

where β is the ratio of the radius of gyration of the particle to the particle radius ($\beta^2 = 0.4$, the value for a uniform sphere, is used throughout the simulation). U, V, W are the post-collision relative velocity components and ω_{L_x}, ω_{L_y}, ω_{L_z}, and ω_{M_x}, ω_{M_y}, ω_{M_z} are the post-collision angular velocity components for the particles L and M, respectively.

The effect of the elasticity and the roughness of the particles enter into the collision process through the introduction of two additional coefficients: the coefficient of restitution ϵ_p, and the coefficient of "non-slip" ϵ_s, respectively. The effect of ϵ_p may be written as:

$$U = -\epsilon_p U_0 \quad . \tag{2.14}$$

The effect of ϵ_s in the y and z directions may be written as:

$$r\omega_{L_z} + r\omega_{M_z} - 2V = \epsilon_s(r\omega_{L_{z0}} + r\omega_{M_{z0}} - 2V_0) \quad , \text{ and} \tag{2.15}$$

$$r\omega_{L_y} + r\omega_{M_y} + 2W = \epsilon_s(r\omega_{L_{y0}} + r\omega_{M_{y0}} + 2W_0) \quad . \tag{2.16}$$

The term on the right-hand side of Equation (2.15) corresponds to the y-component of relative velocity at the point of contact on impact, and the left-hand side corresponds to the y-component of relative velocity upon departure. Similarly Equation (2.16) corresponds to the z-components of relative velocity at impact and upon departure. Equations (2.5) - (2.16) are twelve equations for the twelve unknowns: U, V, W, ω_{L_x}, ω_{L_y}, ω_{L_z}, ω_{M_x}, ω_{M_y}, ω_{M_z}, J, θ, and ϕ. After some algebra the solution for the velocity components may be written as follows:

$$U = -\epsilon_p U_0 \quad , \tag{2.17}$$

$$V = V_0 - \frac{\beta^2(1 - \epsilon_s)}{2(1 + \beta^2)}\left[2V_0 - r(\omega_{L_{z0}} + \omega_{M_{z0}})\right] \quad , \tag{2.18}$$

$$W = W_0 - \frac{\beta^2(1 - \epsilon_s)}{2(1 + \beta^2)}\left[2W_0 + r(\omega_{L_{y0}} + \omega_{M_{z0}})\right] \quad , \tag{2.19}$$

$$\omega_{L_x} = \omega_{L_{x0}} \quad , \tag{2.20}$$

$$\omega_{M_x} = \omega_{M_{x0}} \quad , \tag{2.21}$$

$$\omega_{L_y} = \omega_{L_{y0}} - \frac{(1 - \epsilon_s)}{2r(1 + \beta^2)} \left[2W_0 + r(\omega_{L_{y0}} + \omega_{M_{y0}}) \right] \quad , \tag{2.22}$$

$$\omega_{M_y} = \omega_{M_{y0}} - \frac{(1 - \epsilon_s)}{2r(1 + \beta^2)} \left[2W_0 + r(\omega_{L_{y0}} + \omega_{M_{y0}}) \right] \quad , \tag{2.23}$$

$$\omega_{L_z} = \omega_{L_{z0}} + \frac{(1 - \epsilon_s)}{2r(1 + \beta^2)} \left[2V_0 - r(\omega_{L_{z0}} + \omega_{M_{z0}}) \right] \quad , \tag{2.24}$$

$$\omega_{M_z} = \omega_{M_{z0}} + \frac{(1 - \epsilon_s)}{2r(1 + \beta^2)} \left[2V_0 - r(\omega_{L_{z0}} + \omega_{M_{z0}}) \right] \quad . \tag{2.25}$$

2.3. Structure and Summary of the Program.

This section describes the structure of the sequential program code of the DSMC method for two-dimensional granular material flows. The modification of the program for implementation on the concurrent Cosmic Cube will be discussed in Chapter 4.

The program consists of a main program and many subprograms. Figure 2.2 shows the flow chart. The MAIN program coordinates the various subprograms that calculate the motion and collisions of particles. A brief description of the subprograms is given as follows:

1) GETDAT reads the data file containing the input data, e.g., the calculation time interval ΔT, the width and height of the simulated flow field, the number of cells in each direction, the initial number of simulated particles in each cell, and the initial temperature. The subprogram calculates several parameters which are needed, e.g., the cell height and width and the collision cross

section. The subprogram then divides the simulated flow field into the specified number of cells and calculates the xy-coordinates for each cell.

2) INITIA sets the initial states of all the simulated particles. Six components of thermal velocity are assigned to each particle by sampling randomly from the Maxwell-Boltzmann distribution corresponding to the initial temperature. Position coordinates are associated with each particle as well as an index label that is used by the collision routine. This subprogram also sets the initial cell time as a random fraction of ΔT.

3) MOVE moves all particles through a distance appropriate to ΔT and computes the results of collisions between particles and boundaries.

4) RESET resets all particles' indices after the MOVE routine.

5) COLL selects representative pairs of particles and evaluates the possibility of a collision for each pair. If a collision occurs, the post-collision particle velocities are calculated and the cell time is advanced by an appropriate amount. This procedure continues until the cell time exceeds ΔT.

6) After a specified number of repetitions of the routines MOVE-RESET-COLL, the macroscopic flow properties are calculated by SAMPLE and the results are printed out by PRINT.

7) After the program runs for a specified number of time steps, the calculation is restarted from INITIA with a different initial random number. After the specified number of runs, macroscopic properties are averaged for all runs by ACUM and the results are printed out by PR-ACUM.

The detailed use of the particle index and cell time is described more fully by Bird (1976).

2.4. Normalization of parameters used in the following sections.

The normalization method for a flow consisting of a single type of particle is given by Bird (1976). Distances are normalized by the mean-free path length of the undisturbed initial state, λ_0, whose value is regarded as effectively unity within the program. The vertical extent of the sedimentation field or the height of the channel flow, "ym", and cell size "x_c" and "y_c" are, therefore, stated in terms of the ratio of these quantities to λ_0. Similarly, the most probable thermal speed in the undisturbed initial state, c_{m0}, is regarded as having unit value and is used to normalize the stream velocity 'u' and 'v' in the channel flow in the x and y directions, respectively. The temperature in the undisturbed state, T_0, and particle mass are also regarded as having unit value. The time Δt_m is normalized by λ_0/c_{m0}.

Chapter 3
PRELIMINARY INVESTIGATION

In order to do a quick verification of the DSMC mehod, the collapse of an initially uniform gas atmosphere under the influence of gravity was simulated in a one-dimensional formulation and compared with the analytical solution. The sedimentation of a mixture of two types of particles (with different masses and diameters) was also considered. By restricting the problem to one dimension it is possible to obtain an equilibrium analytical (continuum) solution. The continuum solution, appropriate to the case in which molecule-molecule collisions are perfectly elastic and rough and molecule-boundary collisions are perfectly elastic with slip, is given in Appendix B. The exact equilibrium solution is compared with the results from the DSMC calculation for a variety of initial conditions. The results of the numerical simulation approached the steady-state solution after a time on the order of 100 collision times.

Another preliminary investigation concerns the energy dissipation and energy transfer between translation and spin during collisions.

3.1. One-dimensional sedimentation.

The behavior of a cloud of particles with no interstitial fluid (or of molecules with no ether) is simulated. Initially, the particles have a specified number density and temperature and are in equilibrium (with zero gravity). The typical distance between particles is much greater than their diameter.

At time equal to zero, the particles are released in the 10 mean-free path tall vertical column and the normalized gravity is set to 0.1. Collisions between particles are assumed to be perfectly elastic and perfectly rough whereas particle-boundary collisions are perfectly elastic with perfect slip. Figures 3.1(a) and 3.1(b) show the temporal

and spatial evolution of the number density and temperature fields, respectively, for a flow with a single type of particle. The solid heavy line represents the exact equilibrium solution (see Appendix B). At early times the number density profile (Figure 3.1(a)) approaches, then actually overshoots, the equilibrium solution (after about 10 collision times). The density profile recovers and after 100 collision times the profile is indistinguishable from the equilibrium solution except for small statistical fluctuations. The overshoot behavior, or "bounce" exhibited by the number density profile as it equilibrates toward a steady-state solution suggests the propagation of rarefaction and condensation density waves within the flow field. Figure 3.1(b) shows that the excursion of the temperature field (which is proportional to the thermal energy) from the initially uniform profile is quite pronounced after 10 collision times. As the particles fall under the influence of gravity their potential energy is partially converted to kinetic, or thermal energy which in turn is transferred to other particles through collisions. After 10 collision times, the temperature profile is skewed with the more energetic particles accumulating near the bottom of the flow field. The thermal gradients then begin to decrease and after 50 collision times the temperature profile is close to the uniform equilibrium solution. Note that the final equilibrium thermal energy level is higher than the initial level due to the addition of the gravitational potential energy (when gravity is "turned on") which is partially converted to thermal energy.

Figures 3.2 - 3.4 show the evolution of the number density profiles during the gravitational collapse of an atmosphere consisting of two different types of particles. Three different initial conditions were used to study the effect of varying the mass and diameter ratios. Figure 3.2 shows that for particles of equal mass and different diameter the number density profiles rapidly approach the exact equilibrium solution. The profiles again overshoot the steady-state solution at early times a small amount, as observed earlier (Figure 3.1(a)) during the collapse of an atmosphere consisting of identical particles.

Figures 3.3 and 3.4 show results for which the masses of the two types of particles differ by a factor of ten. The density profile for the heavier particles rapidly approaches the equilibrium solution. In contrast, the profile for the smaller and lighter (or interstitial) particles diverges from the exact solution at early times. At first the lighter particles preferentially settle to the bottom of the flow field; then after some relaxation time (on the order of 50 collision times), the particles "recondense" in the upper part of the flow field and the profile approximates the equilibrium solution. Figure 3.4 shows the effect of changing the ratio of the particle diameters. The profiles in Figures 3.3 and 3.4 show a similar behavior indicating that the mass ratio of the particles is the dominant parameter in determining the relaxation time for the number density profile to approach the equilibrium solution.

3.2. Energy Dissipation and Transfer During a Collision.

As shown in the previous section, the collision model includes the specification of two coefficients, ϵ_p and ϵ_s, which are material properties. ϵ_p ranges between 0 and 1; 0 for perfectly inelastic and 1 for perfectly elastic. ϵ_s may range between -1 and 1; -1 for perfectly rough (rotationally elastic), 0 for fully rough, and 1 for perfect slip (rotationally inelastic). While at low temperatures molecules can be modeled as perfectly elastic and perfectly rough, granular materials are usually both inelastic and rough to some degree. In the simulations of granular material flows, values of $\epsilon_p = 0.6$ and $\epsilon_s = 0$ (corresponding to the values for polystyrene beads, see C. S. Campbell, 1982) were chosen as typical material values. Actually, ϵ_p is not a constant but varies with impact velocity as well as other parameters and ϵ_s may also vary somewhat.

The statement that the granular material is <u>inelastic</u> and <u>fully rough</u> implies that during a collision the surface friction is large enough to bring the relative tangential velocity between the particles to zero. The statement that the molecules are <u>perfectly elastic</u> and

perfectly rough is to be interpreted as follows. When two molecules collide, the points which come into contact will not, in general, possess the same velocity. The two spheres are assumed to grip each other without slipping. Initially, each sphere is strained by the other; then the strain energy is reconverted into translational and rotational kinetic energy with no dissipation. The net effect is that the relative velocity of the spheres at their point of contact is reversed by the elastic impact.

Figures 3.5 and 3.6(a) - 3.6(d) show results using the DSMC method that illustrate the variation of energy dissipation and translational/rotational energy transfer due to collisions as a function of ϵ_p and ϵ_s. In each case the fluid was initially at rest and in an equilibrium state. The total energy is initially equipartitioned between the translational and rotational energy components. The velocity fluctuations are prescribed according to the Maxwell-Boltzmann velocity distribution. As shown in Figure 3.5, for the special case where a collision is perfectly elastic ($\epsilon_p = 1$) with perfect slip ($\epsilon_s = 1$), or perfectly elastic and perfectly rough ($\epsilon_s = -1$), the total energy is conserved. Both inelastic collisions and collisions involving fully rough particles result in energy dissipation. Figures 3.6(a) - 3.6(d) show the details of the conversion of energy between the translational and rotational components. For collisions with perfect slip, the rotational energy, of course, remains unchanged (see Figure 3.6(a)). For inelastic collisions with inelastic restitution and elastic spin coefficients (see Figure 3.6(b)), the translational energy component initially decays most rapidly. After about one collision the rotational energy also decays, as it is converted into translational energy, and an equipartitioned energy state is approached. For the reverse case of elastic restitution and inelastic spin coefficients (see Figure 3.6(c)), the translational energy and the spin energy both begin to decay immediately.

Chapter 4
CONCURRENT FORMULATION FOR THE COSMIC CUBE

This chapter contains a brief description of the structure and operation of the Caltech Cosmic Cube computer including a summary of some of the important terms and conventions used for sending/receiving messages. The formulation of the DSMC method for the Cosmic Cube will be discussed in some detail. Results generated using the concurrent computer will be discussed in Chapter 5 and compared with the corresponding results obtained with a sequential computer.

4.1. Description of the Caltech Cosmic Cube.

At present, software applications written for the Cosmic Cube must be in the programming language *C*. The *C Programmer's Guide to the COSMIC CUBE* (Su et al., 1985) describes the details of *C*-programming for the Cosmic Cube as well as the structure of the hardware. The Cosmic Cube is a collection of $N=2^n$ (n = cube dimension) relatively small computers that operate concurrently and communicate by sending messages between adjacent computers. There is no shared memory; each computer accesses its own dedicated memory. The computers, called *nodes*, are numbered 0, 1, . . ., N-1. Cosmic Cube nodes are connected by bidirectional communication channels that are structured as a binary n-cube. The two machines currently operated by the Caltech Computer Science Department are a 3-cube (8 nodes) and a 6-cube (64 nodes). The 6-cube machine was used in the present work. Another computer (in particular, a DEC VAX) called the Cube Host communicates with the Cosmic Cube through a communication channel connected with node 0. Each node has 128 kbytes of primary storage, and uses an Intel 8086 microprocessor, 8087 floating-point arithmetic coprocessor, and 6 bidirectional channels to communicate with adjacent nodes. Each node also includes some read-only memory used for storing initialization routines. Programming the host computer and the Cosmic Cube is facilitated with the use of several pre-programmed functions. Two fundamental functions,

send, and recv are used for sending and receiving messages, respectively (see the programming manual, Su et al., 1985, for a more complete description).

Although benchmark tests indicate that the 64-node Cosmic Cube should be expected to run at best about 10 times faster than a VAX11/780 computer, the actual speed advantage of the concurrent computer is limited by

1) idle time due to imperfect load balancing,

2) waiting time needed for communication between nodes, and

3) processor time dedicated to processing and forwarding messages between the host computer and the Cosmic Cube.

4.2. Implementation of the DSMC Method.

In the conventional sequential architecture described in Chapter 3, the collision calculations proceed cell by cell, although in principle the collisions and subsequent motion of the particles can be calculated in each cell separately and concurrently. Therefore, to efficiently calculate collisions it is advantageous to use a number of individual computers operating simultaneously. Hence the DSMC method is an ideal candidate for implementation on the Cosmic Cube. In the concurrent formulation of the method, cells are assigned singly or in groups to individual processors (nodes) of the Cosmic Cube. Particles that move from the domain of one node to another are accounted for by communicating the relevant information between the nodes. To implement the DSMC method on the Cosmic Cube much of the sequential computer code may be used, with the addition of several routines that involve communication between nodes and between node 0 and the host computer. The following sections describe the structure of the program that implements the DSMC method on the Cosmic Cube as well as the details of the modification of each subprogram. Complete program listings are given

in Appendix C. Each node of the Cosmic Cube is loaded with the identical program and runs separately and concurrently. Though the program is designed so that rather loose synchronization of the nodes can be employed, in the present implementation lock-step synchronization is enforced by waiting for acknowledgement of every send and receive call.

4.2.1. Assignment of Nodes. The physical space of the simulated flow field is divided into a network of rectangular subregions. Each of the subregions is further subdivided into a number of rectangular cells, each of which contains several particles. If the simulated flow field is assumed to be two-dimensional, each subregion is surrounded by eight adjacent subregions in general. Each node of the Cosmic Cube calculates the motion of particles within one subregion.

During each time step, ΔT, the routine MOVE-GLOB calculates the distance that each particle moves. The subregions are large enough that the majority of particles remain within the same subregion (even within the same cell) in one time step. However, a small number of particles move from one subregion (or node) to another, and the majority of these particles move to an adjacent subregion. The particle velocities are distributed according to the Maxwell-Boltzmann distribution and a particle rarely has a velocity high enough to travel past an adjacent subregion in one time step.

The entire simulated flow field is divided into 64 identical subregions typically consisting of 8 rows and 8 columns. Each subregion is further divided into a number (typically 25) of equally sized cells. Each node is assigned a binary identification (ID) number which is used to label the nodes for communication purposes. The physical computer architecture dictates that the shortest communication channels are between nodes whose ID numbers differ by only a single bit (e.g., node 6 = 0110 and node 14 = 1110). Since the majority of messages are passed between a node and its immediate neighbors, the nodes are laid out as shown in Figure 4.1. For each node or subregion, the 4 closest

subregions correspond to nodes whose ID numbers differ by 2^n, $n = 1, 2, \ldots$ (i.e., 1 bit). The other 4 adjacent nodes have ID numbers that differ by 2 bits. This layout minimizes the communication time between nodes.

4.2.2. Assignment of Particles. In the sequential program, the subprogram INITIA creates all the simulated particles in all the cells for the given initial conditions, i.e., initial temperature and number density. The same subprogram is used with the concurrent Cosmic Cube to assign simulated particles to all the cells in each node. Initially the macroscopic number density is uniform throughout the flow field so that each node has the same number of particles. The particles' velocities follow the Maxwell-Boltzmann velocity distribution corresponding to the initial temperature. Initial data are read in by the cube host computer and subsequently transmitted to each of the nodes. In a channel flow problem subprogram FLOW is used to enter new particles from the left side with a specified number, temperature and stream velocity.

4.2.3. Communication between Nodes. The most important messages that are sent between nodes are those regarding particles that move from one node to another. The subprogram MOVE-GLOB that calculates the motion of particles must be modified to allow for internode particle movement. MOVE-GLOB also discards particles which go out of the simulated field from the right side. Figure 4.2 shows the typical motion of several simulated particles that occurs after collisions have been calculated. When a particle moves from one node to another, all the information about the particle which will be required in the next calculation in the receiving node must be relayed as a message. This is equivalent to physically sending the particle from one subregion to another. The messages consist of the new xy-coordinates and 6 velocity components of the particle. Also, some miscellaneous information that is used in subsequent calculations is sent and will be described in a program listing in the Appendix.

4.2.4. Restricted Particle Movement. After the movement of particles in one time step, the majority of particles either remain in the same node or move to an adjacent node. However, there is a small but finite probability that a particle will have a sufficient velocity to move further away than an adjacent node. This leads to a much more complicated message-sending procedure and a slower computation speed. If the node subregion is sufficiently large and the time step Δt_m is sufficiently small, the number of such particles can be negligible. In the present program, the subprogram MOVE-GLOB restricts the maximum distance a particle can move in one time step to the width of one node subregion. Artificially restricting the movement of a small number of particles introduces a source of error to the simulation. The number of particles that are affected by this procedure depends on the width of the node subregion. The time step Δt_m is chosen to be 1/5 of collision time. It was found that the error that is introduced is negligible if the node width is taken to be larger than about one mean-free path. For a typical case with 16000 particles, in which $T = 1.0$ and the fluid is at rest, no particle is affected by this procedure when the node width is taken to be one mean-free path or longer. Therefore, in the present investigation of the flow in channel, the node width is chosen to be two mean-free paths long for x-direction and one for y-direction. The number of cells per node is chosen to be 25 (limited to be less than 25 because of memory capacity). Then the cell width is smaller than one mean-free path to ensure the validity of the DSMC method.

4.2.5. Collisions. In the sequential program, the subprogram COLL calculates collisions from one cell to the next. With the Cosmic Cube, the routine is the same as for a sequential processor. Although the collision routine runs concurrently on each node, within each node the routine, of course, proceeds from cell to cell sequentially.

4.2.6. Host-Node Communication. Nodes have no I/O capability to peripheral display devices; therefore, to output data (e.g., flow field properties), it is necessary to send messages from individual nodes to the cube host. The host computer then writes the data to an output file or displays the data on a terminal. Since the cube host is connected to the Cosmic Cube only through a communication channel to node 0, it is necessary to route all node-host messages through node 0. This causes node 0 to be quite busy. The messages that are sent from each node to the host computer are about 600 bytes and consist of the following six floating point values: accumulated number of particles, average x and y components of velocity, average translational and angular velocities and collision rate. To avoid a communication "gridlock" (called a fatal deadlock in the programming manual, Su et al., 1985), the message-advance system shown in Figure 4.3 is used. All messages are advanced from one node to the next along the path shown to avoid overloading the memory of any one node (each node has only 10 - 20 kbytes of temporary buffer memory for queuing messages). If messages are sent without specifying the path, the message will be sent automatically by using the shortest available communication channels. However, this method carries the risk of overloading the buffers of node 0 and the adjacent nodes which lead to a deadlock and an interruption in the message-transfer procedure.

4.2.7. Data Accumulation. All the output data sent from every node are stored by the cube host computer and accumulated for a specified number of runs. The data are then averaged and written to an output file. The subprograms ACUM and PR-ACUM from the sequential program version are used directly for this purpose.

4.3. Program Structure.

In applying the DSMC method to the Cosmic Cube some of the programs written for the sequential computer are modified and some are used directly without any change. The subprograms that deal with I/O, and data accumulation and averaging reside in the cube host computer, whereas the routines that calculate the translation and collision of

particles are run by individual nodes. Routines proceed sequentially within each node but concurrently globally. Figure 4.4 shows the program structure for both the cube host and the nodes. The detailed operation of the subprograms is given in terms of comment statements contained within the program listings (see Appendix C).

4.4. Load Balancing.

Idle time due to imperfect load balancing decreases the efficiency of concurrent processing. Particles are initially equally distributed among the nodes, but after some time some nodes might contain a much larger number of particles than other nodes. This problem is particularly severe for problems involving shock waves or sedimentation. Nodes containing many particles require a longer time to calculate collisions and translational motion and therefore the global CPU time is dominated by the most heavily loaded nodes. The key to load balancing lies in assigning a <u>weighting factor</u> to each particle within a node. The weighting factor is defined as the number of real particles or molecules that a simulated particle represents. Each node has its own weighting factor. If the number of particles in a certain node exceeds some specified critical value, some particles are "killed" and the weighting factor of the remaining particles within the node increases correspondingly. Similarly, if the number of particles falls below another specified critical value, new particles are "created" and the weighting factor is adjusted accordingly.

The introduction of the weighting factor dynamically reduces the load unbalance between different nodes. Results with improved statistics are obtained without sacrificing CPU time. Without a weighting factor, average properties from nodes with very few particles have large errors associated with them. The subprogram <u>BALANCE</u> calculates the weighting factors and performs the elimination/duplication of particles. Weighting factors must be taken into account when particles move from one node to another. When a node receives a particle from a node with a different weighting factor, particles must be removed or

duplicated so that the weighting factor of the new particle conforms with the present node. These changes are performed by the subprogram WEIGHT.

4.5. Limitations of the Caltech Cosmic Cube.

The primary storage of each node (128 kbytes) limits the maximum number of simulated particles and cells per node. From the primary storage, 64 kbytes are available for operating system kernel and instruction code of the application program, and the other 64 kbytes are available for data and stack. In addition, buffer space for inter-node messages, i.e., to store particles that are received from other nodes, must be available to the kernel. The present application requires about 40 kbytes for instruction. About 20 kbytes are consumed by the kernel and operating system. Then, considering the queuing space to store particle information, maximum data size available in the present calculation may be about 30 kbytes, i.e., about 25 cells per node and 10 particles per cell. Therefore, totally about 16000 particles can be used in a typical calculation.

4.6. Optimal Formulation for the Cosmic Cube.

To improve the efficiency and accuracy of the concurrent formulation of the DSMC method it is important to optimize the load balancing routine. In the present program, the number of particles is controlled so that each node contains approximately the same number of particles. However, the CPU time consumed by this routine is not negligible. When the number of particles within a node increases or decreases by a factor of 2, particles are destroyed or created by the load-balancing routine. The optimum value of this factor may depend on the type of flow that is being simulated. In some problems (particularly ones involving sedimentation or shock waves) very large particle concentration gradients may develop. In such cases, improved statistical results may be obtained by dynamically changing the cell size or number of cells as well as the weighting factor to accommodate number density inhomogeneities in the flow.

Chapter 5
CHANNEL FLOWS OF GRANULAR MATERIALS

Essentially what differentiates a granular material from a gas is that in a granular material the collisions are inelastic. The system is continually losing energy. In the previous chapters, in order to apply the DSMC method to granular material in an infinite space (uninfluenced by the presence of containing walls), the coefficient of restitution ϵ_p and the non-slip coefficient ϵ_s were introduced. Now in addition to these coefficients, similar coefficients for collisions between a particle and the wall, the coefficient of restitution ϵ_w and non-slip coefficient ϵ_{sw}, are needed to simulate channel flow. Table 5.1 shows several typical permutations of these four coefficients and compares the cases for granular material and gas.

Coefficient	ϵ_p	ϵ_s	ϵ_w	ϵ_{sw}
general inelastic	< 1	0	< 1	0
inelastic spin(no slip)	1	0	1	0
inelastic spin(with diffuse wall)	1	0	diffuse	
elastic spin	1	-1	1	-1
elastic particle(with specular wall)	1	-1	1	1
elastic,no spin	1	1	1	1
gas, elastic with rotation	1	-1	diffuse	

Table 5.1 Comparison of coefficients for several types of materials

"Diffuse" designates a special treatment of wall collisions in which a particle is re-emitted with zero mean velocity but with the wall temperature, independently of the incident energy.

The present work investigates the effects of these coefficients on the flow and tries to understand granular material flows. Channel flows with various conditions of collision for particle-particle and particle-wall without gravity are discussed. Additionally, as an extension of the study of the gravitational collapse of a gas which was done as a one-dimensional problem in the preliminary investigation, the channel flow with gravity will be discussed.

5.1. Description of the Model for Channel Flow without Gravity.

For the channel flow without gravity, in order to understand the differences between granular material flow and gas flow, three combinations of the four coefficients of Table 5.1 were investigated: general inelastic, inelastic spin (with diffusely reflecting wall) and elastic gas. The general inelastic model represents a general granular material flow while the inelastic spin model (with diffusely reflecting wall) artificially represents an intermediate substance with properties somewhere between a granular material and a gas, such that collisions between particles have no slip (as in granular materials) while particles reflect diffusely from the wall (as for the boundary of a gas). For the general inelastic case, values of $\epsilon_p = 0.6$, $\epsilon_s = 0$, $\epsilon_w = 0.8$ and $\epsilon_{sw} = 0$ (corresponding to the values for polystyrene beads, see C. S. Campbell, 1982) were chosen as typical material values. In the diffusely reflecting wall, the wall temperature, T_w, is set to one where T_w is normalized by the temperature of the flow in the undisturbed state.

5.1.1. <u>Geometry and Input Data</u>. Figure 5.1 shows the simulated flow field schematically. For all cases, the same type of geometry of the simulated flow field is used; for the channel flow without gravity the simulated flow field is surrounded by two parallel walls 16 mean-free path lengths long (xm = 16) and two side boundaries 8 mean-free path lengths high (ym = 8) through which particles can penetrate. Particles enter through the left-side boundary and are ejected into a vacuum through the right-side boundary of the simulated flow field. For the channel flow xm is set to 32 and ym is set to 4 so that the

equilibrium state could be observed far downstream.

At time equal to zero and after every time step the particles are released uniformly in the flow field, the normalized temperature is set to 1 and the normalized stream velocity is set to 2.2567, corresponding to a Mach number M = 2.76 and Kn = 0.125, for the x-component and 0 for the y-component. The normalized time interval for the calculation step is set to 0.17725, 1/5 of the mean collision time in the undisturbed initial state. The motions of all particles, including particle-particle and particle-wall collisions (conditions specified by ϵ_p, ϵ_s, ϵ_w and ϵ_{sw}), are begun and continue until a steady state is reached. In the calculations, it was found that after about 20 mean collision times the flow becomes steady such that the number of the particles leaving through the right-side boundary is approximately the same as that entering from the left. The resulting steady-state solutions were printed out after 30 collision time steps. Each plotted value represents the average of the two calculated values (after being averaged over 50 runs) which were geometrically symmetric about the center line. Computing time on the Cosmic Cube for a typical problem is about 13 hours.

The simulated flow field is divided into 64 subregions (8 in the x-direction and 8 in the y-direction) (see Figure 5.1). One subregion is assigned to each node of the 6-D cube so that each node calculates a region 2 mean-free paths long and 1 mean-free path high. Also, each subregion is divided into 25 cells (5 by 5) so that each cell is 0.4 mean-free paths long (x_c = 0.4) and 0.2 mean-free paths high (y_c = 0.2). In the undisturbed initial state, 10 simulated particles are placed in each cell for a total of 16,000 particles in the initial state. After every time step, 10 new particles enter from the left with the stream velocity 2.2567. Derivation of this value follows from:

$$n_{enter} = \frac{u\Delta t_m n_{init}}{x_c} \quad ; \tag{5.1}$$

Δt_m, n_{init} and x_c are the normalized time-interval calculation step, the number of simulated particles per cell initially and the normalized cell size in the x direction, and are set to 0.17725, 10 and 0.4, respectively. To simplify the upstream and downstream boundary conditions the stream velocity u must be higher than about 2, so that the number of particles which move upstream through the side boundaries due to thermal motion can be negligible. Thus, u is set to 2.2567 so that n_{enter} is an integer : 10.

5.1.2. Reynolds Number and Mach Number. The "first approximation" to the coefficient of viscosity μ in a hard-sphere gas is (Bird, 1976)

$$\mu = \frac{5\pi\rho c\lambda}{32} \quad , \tag{5.2}$$

where ρ is the density, c is the average thermal speed and λ is the mean-free path. The relation between c and the most probable thermal speed c_m is:

$$c = \frac{2}{\pi^{1/2}} \cdot c_m \quad . \tag{5.3}$$

Also, c_m can be related to the temperature as the following:

$$c_m = (2RT)^{1/2} \quad . \tag{5.4}$$

Thus, the Reynolds number based on the height, h (h = ym), of the channel is the following:

$$Re_h = \frac{\rho(uc_m)(h\lambda)}{\mu} = \frac{16uh}{5\pi^{1/2}} \tag{5.5}$$

and the Mach number is:

$$M = \frac{uc_m}{(\gamma RT)^{1/2}} = u\left(\frac{2}{\gamma}\right)^{1/2} , \qquad (5.6)$$

and γ can be calculated in terms of the degrees of freedom of the particle, α, as:

$$\gamma = 1 + \frac{2}{\alpha} \quad . \qquad (5.7)$$

For the present model h is 8 and α is 6. Therefore, the Reynolds number based on the height of channel and the Mach number in the simulated flow are 32.6 and 2.76, respectively.

5.2. Results and Discussion for the Channel Flow without Gravity.

Figures 5.2 - 5.5 show the profiles of several variables. Although the contour plots are not smooth, the plots of the same variables vs. y-direction (with x fixed) are relatively smooth curves. Figure 5.2(a), (b), (c) and (d) show the number density, temperature, pressure and velocity fields of the granular material flow (ϵ_p = 0.6, ϵ_s = 0, ϵ_w = 0.8 and ϵ_{sw} = 0), respectively. Figures 5.3 are for the gas flow (ϵ_p = 1, ϵ_s = -1, diffusely reflecting wall) and Figures 5.4 for the intermediate flow (ϵ_p = 1.0, ϵ_s = 0, diffusely reflecting wall). In Figures 5.2 - 5.5, only half of the flow field is displayed; in Figure 5.6 the entire field is exhibited.

In the granular material flow the temperature decreases due to collisional energy loss, while in the gas flow (Figure 5.3(b)), it rises due to viscous dissipation. In the granular material flow the temperature decreases to about 0.35 at the exit of the channel (Figure 5.2(b)). This decrease is the same as that observed in Figure 3.5 for the same medium at rest after 8 collision times. From Figure 5.2(d) it follows that 8 collisions occur in the 16 mean free path length of the channel. Figure 5.2(d) shows the growth of the velocity boundary layer. It is noted that the temperature profile is quite uniform in the y-direction (Figure 5.2(b)); a thermal boundary layer is not

apparent. As expected, pressure is also approximately uniformly distributed (Figure 5.2(c)). Figure 5.2(a) shows that the density tends to be slightly high near the wall (there is no gravity in this problem) but that the particles behave as though the flow were nearly constant density. The slight gathering of particles near the wall causes a reduction of the effective cross-sectional area of the channel so that the velocity u near the central axis increases about 10 % further down stream. Since temperature decreases and velocity increases, the local Mach number increases to about 5. It is remarkable that outside of the boundary layer though the total energy of this system decreases, as does the thermal energy, the kinetic energy increases slightly.

For the general inelastic collision (Figure 5.2) where $\epsilon_{sw} = 0$, a particle-wall collision does not much affect the x-component of velocity of the particle. A diffusely reflecting wall, however, has a great effect. With the diffusely reflecting wall at the same temperature as for the gas, Figures 5.3 and 5.4 show the difference in the growth of the boundary layers due to the difference in the particle-particle collision characteristics. Significant slip flow can be observed in both cases. Note that $\epsilon_s = -1$ for a gas (Figure 5.3) and 0 for granular materials (Figure 5.4).

The growth of the boundary layer for gas collisions seems to be faster than that for granular collisions. In these figures, however, this conclusion is not always clear since the boundary layer from the wall has already grown to the centerline in the early portion of the channel. Figure 5.5 shows the growth of the boundary layers near the leading edge of the wall where the boundary layers from both walls have not yet met. The boundary layers for both cases grow similarly very close to the leading edge of the wall but gradually differences between them appear in the the downstream regions. The growth rate in the granular collision case tends to be slower further downstream. The lower growth rate seems to be caused mainly by the decrease of the collision frequency due to a decrease in the temperature with only a

moderate increase in the density. The collision frequency directly affects momentum transport by shear on the wall. The Reynolds number based on x is of the order of 1 at the point where the boundary layers from the two walls meet (i.e., $x < \lambda$).

Comparing the general inelastic case (Figure 5.2) and the intermediate case (Figure 5.4), all properties (density and temperature etc.), are somewhat different. Though the temperature of the fluid outside the boundary layer in the intermediate case does decrease as in the general inelastic case, the density and temperature profiles are not flat. The Mach number does not increase as much as in the inelastic case. These differences may be due to the different values of ϵ_p as well as to the different wall conditions. This implies that the behavior of the granular material depends very much on the materials' properties, e.g, ϵ_p and ϵ_s.

5.3. Result and Discussion for the Channel Flow with Gravity.

The geometry of the simulated flow field is the same as before (Figure 5.1) with the gas moving between specular walls. This time, xm and ym are set to 32 and 4, respectively, so that an equilibrium state could be obtained further down stream in manageable CPU time. The gravity g is set to 0.5. Re_h is 16.3 and M is 2.76. Figure 5.6 shows the results after 20 collision times. About 20 mean-free paths from the left entrance equilibrium is reached; the temperature is uniform in the x and y directions, the stream is parallel to the walls, and pressure is constant in the x-direction. Before the flow reaches the equilibrium state, overshoot and recovery are exhibited at 10 mean-free paths and 15 mean-free paths, respectively, in analogy to the behavior already commented upon in the atmospheric collapse problem (Chapter 3).

Chapter 6
SUMMARY AND CONCLUSIONS

The flow of granular materials has been investigated using the DSMC method of Bird modified for granular materials. The DSMC method was implemented on the concurrent processing Cosmic Cube computer at Caltech in order to obtain good statistical solutions in manageable CPU time. After preliminary investigations of energy dissipation and energy transfer between translation and spin with several different collision conditions, and of one-dimensional sedimentation with molecular type collisions, the present method is applied to the problem of channel flows.

For the two special cases where a collision is perfectly elastic with perfect slip, or perfectly rough, the total energy is conserved. For inelastic collisions involving perfectly rough particles, the translational energy component is found initially to decay most rapidly while after about one collision the rotational energy is converted onto translational energy as an equipartitioned energy state is approached.

For the gas sedimentation problem at early times, the number density profile approaches and then actually overshoots the equilibrium solution. The density profile then recovers and after a few collision times the profile is indistinguishable from the equilibrium solution except for small statistical fluctuations. The "overshoot" behavior exhibited by the number density profile as it equilibrates toward a steady-state solution suggests the propagation of rarefaction and condensation density waves within the flow field. As the particles fall under the influence of gravity their potential energy is partially converted to kinetic or thermal energy which is in turn transferred to other particles through collisions. The final equilibrium thermal energy level approaches a uniform level higher than the initial level due to the addition of the gravitational potential energy. This

behavior is also observed in the channel flow of gas with gravity. In this case the number density overshoots the equilibrium state in the lower upstream portion of the channel while further downstream it approaches the equilibrium state where the flow direction is uniformly parallel to the wall and the temperature profile is uniform and higher than that upstream.

For the channel flow with no gravity, there are differences between the behavior of granular material flows and gas flows where Mach number is 2.76 and Reynolds number is 32.6. For both cases, significant 'slip' at the wall is observed. For the granular material flow, the boundary layer is thin and the velocity reduction near the wall is small. This especially thin boundary layer for granular materials is mainly caused by the difference between the types of collisions between particles and the wall, diffusely reflecting walls for the gas whereas rough walls for the granular materials. In addition to the different types of particle-wall interactions, the difference in the growth rates of the boundary layers due to the different type of particle-particle interactions is suggested by the comparison between the gas and the intermediate flows, in which particle-particle interactions are of a granular type while particle-wall interaction is of a gas type. The boundary layers for both cases grow similarly very close to the leading edge of the wall but gradually differences between them appear in the downstream regions. The growth rate in the granular collision case tends to be slower further downstream. The lower growth rate seems to be caused mainly by the decrease of the collision frequency due to a decrease in the temperature with only a moderate increase in the density.

Using the concurrent computer on which the DSMC method can be naturally implemented has enabled powerful computational ability to be brought to bear on the DSMC calculation. Future work will be directed at improving the efficiency and accuracy of the concurrent formulation, (e.g., load balancing and dynamic change of cell size), so that it can

be applied to shock-wave or condensation problems where the density difference between different regions may be a couple of orders of magnitude. For further development of the method for two-phase or three-dimensional low-Reynolds-number, high-speed continuum flows, much greater computer power (e.g., more nodes and more memory), will be required.

Appendix A

NORMALIZATION OF PARAMETERS

Distance and time are normalized by the mean-free path and mean-collision time in the initial undisturbed state, respectively. For convenience, the mean-free path and mean-collision time are set to unity within the program. For these values to be unity it is necessary to normalize the diameters and masses of the particles with respect to certain reduced values.

In the initial state, the mean-free path λ_0 for the mixture of two different particles is

$$\lambda_0 = \sum_{p=1}^{2} \frac{n_p}{n} \left[\sum_{q=1}^{2} \frac{1}{4}\pi (d_p + d_q)^2 n_q \left\{ 1 + \frac{m_p}{m_q} \right\}^{1/2} \right]^{-1} , \tag{A.1}$$

where n_i, d_i, and m_i are the particle-number density, diameter, and mass of the ith particle, respectively. Setting λ_0 equal to unity requires that the reduced diameter of species 1 be given by

$$\tilde{d}_1 = \left[\sum_{p=1}^{2} \frac{n_p}{n} \left\{ \sum_{q=1}^{2} \frac{1}{4}\pi \left\{ \frac{d_p}{d_1} + \frac{d_q}{d_1} \right\}^2 n_q \left\{ 1 + \frac{m_p}{m_q} \right\}^{1/2} \right\}^{-1} \right]^{1/2} . \tag{A.2}$$

The mean-collision time is the reciprocal of the mean-collision rate ν_0, given by

$$\nu_0 = \sum_{p=1}^{2} \frac{n_p}{n} \sum_{q=1}^{2} \frac{1}{2}\pi^{1/2} (d_p + d_q)^2 n_q \left\{ 2kT_0 \frac{m_p + m_q}{m_p m_q} \right\}^{1/2} , \tag{A.3}$$

where

$$T_0 = \frac{1}{n}(n_1 T_1 + n_2 T_2) \quad . \tag{A.4}$$

Therefore, setting the mean-collision time to unity is equivalent to setting the mean collision rate to unity. The reduced mean collision rate $\tilde{\nu}_0$ may be expressed in terms of the reduced mass and diameter of a particle of species 1 as follows:

$$\tilde{\nu}_0 = 1 = \sum_{p=1}^{2} \frac{n_p}{n} \sum_{q=1}^{2} \frac{1}{2}\pi^{1/2} \left\{ \frac{d_p}{d_1}\tilde{d}_1 + \frac{d_q}{d_1}\tilde{d}_1 \right\}^2 n_q \left\{ \frac{2k(m_p + m_q)m_1}{m_p m_q \tilde{m}_1} \right\}^{1/2} \quad . \tag{A.5}$$

The reduced mass and diameter of species 2 can then be calculated as follows:

$$\tilde{d}_2 = \frac{d_2}{d_1}\tilde{d}_1 \quad , \tag{A.6}$$

$$\tilde{m}_2 = \frac{m_2}{m_1}\tilde{m}_1 \quad . \tag{A.7}$$

Temperature is also normalized by T_0 so that all the variables are normalized in terms of reduced values. However, it is convenient to retain the number density n as a dimensional quantity based directly on the number of simulated particles within a cell.

Appendix B

EXACT EQUILIBRIUM SOLUTION

Nomenclature

g	gravity
k	Boltzmann's constant
L	length of the flow field in the y-direction
m	mass of molecule
n	number density
T	temperature
α	degrees of freedom
y	coordinate in the direction of gravity

Subscripts

0	initial state
1	species 1
2	species 2
e	final equilibrium state with gravity g

The governing equations for the steady one-dimensional sedimentation of a mixture of two different types of molecules are given as follows:

Mass Conservation:

$$L\, n_{01} = \int_0^L n_{e1} dy \quad , \tag{B.1}$$

$$L\, n_{02} = \int_0^L n_{e2} dy \quad . \tag{B.2}$$

Momentum Conservation:

$$kT_e\frac{d}{dy}(n_{e1} + n_{e2}) + (m_1n_{e1} + m_2n_{e2})g = 0 \quad . \tag{B.3}$$

Equation (B.3) represents the hydrostatic balance between the pressure gradient (first term) and the gravitational body force (second term). The conservation of potential plus kinetic energy is given in the next equation.

Energy Conservation:

$$\frac{1}{2}gL^2(m_1n_{01} + m_2n_{02}) + \frac{1}{2}\alpha kLT_0(n_{01} + n_{02})$$

$$= g\int_0^L (m_1n_{e1} + m_2n_{e2})ydy + \frac{1}{2}\alpha kT_e\int_0^L (n_{e1} + n_{e2})dy \quad . \tag{B.4}$$

Solving equations (B.1) - (B.3) for the number densities gives the following:

$$n_{e1} = n_{01}\frac{m_1gL}{kT_e}\left[\frac{e^{\left\{-\frac{m_1g}{kT_e}y\right\}}}{1 - e^{\left\{-\frac{m_1gL}{kT_e}\right\}}}\right] \quad , \tag{B.5}$$

$$n_{e2} = n_{02}\frac{m_2gL}{kT_e}\left[\frac{e^{\left\{-\frac{m_2g}{kT_e}y\right\}}}{1 - e^{\left\{-\frac{m_2gL}{kT_e}\right\}}}\right] \quad . \tag{B.6}$$

Substituting equations (B.5) and (B.6) into (B.4) and integrating produces the following implicit equation for T_e:

$$\frac{1}{2}gL^2(m_1 n_{01} + m_2 n_{02}) + \frac{1}{2}\alpha kLT_0(n_{01}n_{02}) = kLT_e(n_{01} + n_{02} + \frac{1}{2}\alpha kLT_e(n_{01} + n_{02}))$$

$$-gL^2\left[m_1 n_{01}\left\{\frac{e^{\left\{-\frac{m_1 gL}{kT_e}\right\}}}{1 - e^{\left\{-\frac{m_1 gL}{kT_e}\right\}}}\right\} + m_2 n_{02}\left\{\frac{e^{\left\{-\frac{m_2 gL}{kT_e}\right\}}}{1 - e^{\left\{-\frac{m_2 gL}{kT_e}\right\}}}\right\}\right] \quad . \qquad \text{(B.7)}$$

Equations (B.5) - (B.7) for the number densities and temperature in the final state represent the exact equilibrium solution referred to earlier in the text.

Appendix C

PROGRAM LISTING

PROGRAM	PAGE
HOST PROGRAM	42
DAT	54
CUBE PROGRAM	55

Sep 30 16:33 1985 host_main.c Page 1

```
/*
   This is the main program for the HOST. It reads the data from the data file
   'dat' with the subprogram 'host_getdat' and then send the data to the node
   '0' of the CUBE. It also prints the input data with the subprogram 'prinit'.
   After receiving the results, all data are accumulated by the subprogram
   'acum' through 'nrun' and after all of the averages are calculated and they
   are printed out with the subprogram 'pr_acum'. The results are sent by the
   cube through node '0' in every 'nstep' time steps just as the initial state
   had been sent. In order to save the updated accumulated results in case of
   emergency (computer system down), the subprogram 'pr_backup' prints out the
   accumulated results in every 5 runs.
 */

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

char dat[] = {"dat"};

main()
{
        char fname[80];
        short nx, ny, nrun, outnp;
        short i, j, k, l, m, n, kk, ll, rrr;
        MSGDESC in_d;
        MSGDESC out_ini_d;
        MSGDESC out_step_d;
        MSGDESC pls_sd;
        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);

        cosmic_init(HOST,0);
        host_set();
        host_getdat();
        host_dat();

        htocs(&in.d.nprint,sizeof(struct insht)/sizeof(short));
        htocf(&in.f.Tw,sizeof(struct inflt)/sizeof(float));
        sendb(&in_d);
        while(in_d.lock) flick();
        ctohs(&in.d.nprint,sizeof(struct insht)/sizeof(short));
        ctohf(&in.f.Tw,sizeof(struct inflt)/sizeof(float));

        printf ( "input dat\n" );
        prinit();

        for ( nrun = 0; nrun < in.d.runmax; ) {
            sprintf ( fname,"backup.%d",(nrun/5)%3 );
            freopen ( fname,"w",stdout );
            for ( rrr = 5; rrr-- && nrun < in.d.runmax; nrun++ ) {
                printf( "\nRun number %hd\n", nrun);
                time = 0.;
                outnp = 0;
                for (i = 0; i < chsize; i++) {
                     for ( j = 0; j < cwsize; j++) {
```

Sep 30 16:33 1985 host_main.c Page 2

```
                                sendb(&pls_sd);
                                m = who(j,i);
                                out_ini_d.type = m + 20000;
                                out_ini_d.buf = (char *)&out[m];
                                recvb(&out_ini_d);
                        ctohs(&out[m].d.np,sizeof(struct outsht)/sizeof(short));
                        ctohf(&out[m].f.wt,sizeof(struct outflt)/sizeof(float));
                        }
                }
                acum(outnp);

                /* 'nprint' prints per run */
                for ( outnp = 1; outnp < (in.d.nprint + 1); outnp++ ) {
                        time = outnp * in.d.nstep * in.f.dtm;
                        for (i = 0; i < chsize; i++) {
                             for ( j = 0; j < cwsize; j++) {
                                  sendb(&pls_sd);
                                  m = who(j,i);
                                  out_step_d.type = m + 20000;
                                  out_step_d.buf = (char *)&out[m];
                                  recvb(&out_step_d);
                        ctohs(&out[m].d.np,sizeof(struct outsht)/sizeof(short));
                        ctohf(&out[m].f.wt,sizeof(struct outflt)/sizeof(float));
                             }
                        }
                        acum(outnp);
                }
        }
        pr_backup(nrun);
    }
    pr_acum();
    fflush(stdout);
    cosmic_exit();
}
```

Sep 30 16:33 1985 host_set.c Page 1

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

host_set()
{
      pid = mypid();
      dim = cubedim(); /* dimension of the cube, e.g., 6 for 6D-cube */
      size = 1 << dim; /* number of the nodes, e.g., 64 for the 6D-cube */
      chdim = dim >> 1; /* dimension of the cube in the y-direction
                           of the simulated flow field, typically 3
                           for the 6D_cube */
      chsize = 1 << chdim; /* number of nodes in the y-direction, e.g., 8 */
      cwdim = dim - chdim; /* cube dimension in the x-direction, e.g., 3 */
      cwsize = 1 << cwdim; /* number of nodes in the x-direction, e.g., 8 */
}
```

Sep 30 16:36 1985 host_getdat.c Page 1

```
/* This subroutine opens the data file and reads and prints the data on an
   output file */

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

host_getdat()
{
        int n;
        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( "\n\nrunmax= %5hd  nprint = %5hd  nstep  = %5hd  ",
                in.d.runmax, in.d.nprint, in.d.nstep);
        printf( "ncx    = %5hd  ", in.d.ncx);
        printf( "ncy    = %5hd  mc     = %5hd  seed   = %5hd\n",
                in.d.ncy, in.d.mc, in.d.seed);
        fgets(dum, MAXNAM, datptr);

        fscanf(datptr,"%f\t%f\t%f\n", &in.f.xm, &in.f.ym, &in.f.dtm);
        printf( "xm = %5.2f  ym = %5.2f  dtm = %8.3f\n",
                  in.f.xm, in.f.ym, in.f.dtm);
        fgets(dum, MAXNAM, datptr);

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

        fscanf(datptr, "%f\t%f\t%f\t%f\t%f\t%f\n",
              &in.f.ep, &in.f.es, &in.f.ew, &in.f.esw, &in.f.beta, &in.f.diff);
        printf( "ep     = %5.2f  es     = %5.2f  ew     = %5.2f  ",
                  in.f.ep, in.f.es, in.f.ew);
        printf( "esw    = %5.2f  beta   = %5.2f  diff   = %5.2f\n\n",
                  in.f.esw, in.f.beta, in.f.diff);

}
```

Sep 30 16:37 1985 host_dat.c Page 1

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

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

host_dat()
{
        pi2 = 2 * PI;
        sq2 = sqrt(2.);
        xm_node = in.f.xm / cwsize;
        ym_node = in.f.ym / chsize;
        chx = xm_node / in.d.ncx;
        chy = ym_node / in.d.ncy;
        vmw = sqrt(in.f.Tw);
        fnd = in.d.mc / chx / chy;
        cxs = 1. / ( sq2 * fnd );
        sqd = sq2 * PI * fnd;
        d = 1. / sqrt(sqd);
        ami = 0.25 * in.f.beta / sqd;
        omm = 1. / sqrt(ami);
        omw = sqrt(in.f.Tw / ami);
        acu = in.f.dtm * in.d.nstep * in.d.mc * fnd * cxs * sqrt(2./PI);
        vrm = 2. * sqrt(2./PI);
        pslipf = in.f.beta * (1.0 - in.f.es) / (1.0 + in.f.beta);
        wslipf = (1.0-in.f.esw) / (1.0+in.f.beta);
        collfac =  2.0 * chx * chy / cxs;

   /*   xm_node = length of node region in x-direction, in mean free paths
    *   ym_node = length of node region in y-direction, in mean free paths
    *   chx = cell size in x-direction, in mean free paths
    *   chy = cell size in y-direction, in mean free paths
    *   vmw = most probable thermal speed @ T = Tw
    *   fnd = unnormalized undisturbed gas number density
    *   cxs = hard sphere collision cross section
    *   d   = diameter of the simulated particle
    *   ami = moment of inertia
    *   omm = most probable spin velocity @ T = 1
    *   omw = most probable spin velosity @ T = Tw
    *   acu = number of collisions per cell per sample short interval
    *   vrm = 1st approx to relative speed in initial undisturbed state
    *   pslipf, wslipf, collfac ( see coll.c in the cube program )
    */
}
```

Sep 30 16:37 1985 prinit.c Page 1

```
/* This subprogram prints the parameters used in the following calculations */

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

prinit()
{
        printf( "pi2 = %8.6f sq2 = %8.6f chx = %8.6f", pi2, sq2, chx);
        printf( "chy = %8.6f xm = %8.6f ym = %8.6f\n",
                chy, xm, ym);
        printf( "vmw = %8.6f fnd = %8.6f\n", vmw, fnd);
        printf( "cxs = %8.6f sqd = %8.6f d = %8.6f", cxs, sqd, d);
        printf( "ami = %8.6f omm = %8.6f omw= %8.6f\n",
                ami, omm, omw);
        printf( "NM = %hd acu = %8.6f vrm = %8.6f", NM, acu, vrm);
        printf( "pslipf = %8.6f wslipf = %8.6f\n", pslipf, wslipf);
        printf( "collfac = %8.6f\n\n", collfac);
}
```

Sep 30 16:38 1985 acum.c Page 1

```
/* This suprogram accumulates outputs over the updated run */

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

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

        for ( i = 0; i < chsize; i++ ) {
               for ( j = 0; j < cwsize; j++ ) {
                   m = who(j,i);
                   for ( nx = 0; nx < in.d.ncx; nx++ ) {
                        for ( ny = 0; ny < in.d.ncy; ny++ ) {
                             for ( k = 0; k < D6; k++ )
           acum_sc[k][nx][ny][m][outnp] += out[m].f.wt * out[m].f.sc[k][nx][ny];
                        }
                   }
               }
        }
}
```

Sep 30 16:41 1985 pr_backup.c Page 1

```
/* This subroutine averages the updated accumulated outputs every 5 runs
 * and prints them in order to save the latest updated results in case of
 * emergency.
 */

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

pr_backup(nrun)
     short nrun;
{
     short i, j, nx, ny, k, l, m, np;

     for ( np = 0; np < (in.d.nprint + 1); np ++ ) {
        time = np * in.d.nstep * in.f.dtm;
        printf( "Averaged flow properties at time = %10.5f\n", time);
        for (k = 0; k < chsize; k++) {
           for (l = 0; l < cwsize; l++) {
              m = who(l,k);
              printf( "node = %hd, l=%hd, k=%hd\n", m, l, k);
              printf( " Cell    x     y    samp   dens   x-vel");
              printf( "    y-vel    e-tr    e-sp    etot   coll r\n");
              for ( nx = 0; nx < in.d.ncx; nx++ ) {
                 for ( ny = 0; ny < in.d.ncy; ny++ ) {
                    op[0] = acum_sc[0][nx][ny][m][np];
                    op[1] = op[0] / (nrun * in.d.mc);
                    if ( op[0] > 0. ) {
                       op[2] = acum_sc[1][nx][ny][m][np] / op[0];
                       op[3] = acum_sc[2][nx][ny][m][np] / op[0];
      op[4] = 2.*(acum_sc[3][nx][ny][m][np]/op[0]-op[2]*op[2]-op[3]*op[3])/3.;
                       op[5] =2.*ami*acum_sc[4][nx][ny][m][np]/(3.*op[0]);
                       op[6] = ( op[4] + op[5] ) / 2.;
                    }
                    else {
                       op[2] = 0.;
                       op[3] = 0.;
                       op[4] = 0.;
                       op[5] = 0.;
                       op[6] = 0.;
                    }
                    op[7] = acum_sc[5][nx][ny][m][np]/(nrun * acu);
                    printf( "(%hd, %hd)%5.2f%5.2f",
                    nx, ny, out[m].f.c[0][nx], out[m].f.c[1][ny]);
                    printf( "%6.0f ", op[0]);
                    for ( i = 1; i < 7; i++ )
                       printf("%7.4f ", op[i]);
                    printf( "%7.4f\n", op[7]);
                 }
              }
           }
        }
     }
}
```

Sep 30 16:39 1985 pr_acum.c Page 1

```
/* This subprogram averages the accumulated outputs for all runs and prints
 * averaged values as final results.
 */

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

pr_acum()
{
     short i, j, nx, ny, k, l, m, np;

     for ( np = 0; np < (in.d.nprint + 1); np ++ ) {
        time = np * in.d.nstep * in.f.dtm;
        printf( "Averaged flow properties at time = %10.5f\n", time);
        for (k = 0; k < chsize; k++) {
           for (l = 0; l < cwsize; l++) {
              m = who(l,k);
              printf( "node = %hd, l=%hd, k=%hd\n", m, l, k);
              printf( " Cell    x     y    samp   dens   x-vel");
              printf( "   y-vel    e-tr    e-sp    etot   coll r\n");
              for ( nx = 0; nx < in.d.ncx; nx++ ) {
                 for ( ny = 0; ny < in.d.ncy; ny++ ) {
                    op[0] = acum_sc[0][nx][ny][m][np];
                    op[1] = op[0] / ( in.d.runmax * in.d.mc);
                    if ( op[0] > 0. ) {
                       op[2] = acum_sc[1][nx][ny][m][np] / op[0];
                       op[3] = acum_sc[2][nx][ny][m][np] / op[0];
      op[4] = 2.*(acum_sc[3][nx][ny][m][np]/op[0]-op[2]*op[2]-op[3]*op[3])/3.;
                       op[5] =2.*ami*acum_sc[4][nx][ny][m][np]/(3.*op[0]);
                       op[6] = ( op[4] + op[5] ) / 2.;
                    }
                    else {
                       op[2] = 0.;
                       op[3] = 0.;
                       op[4] = 0.;
                       op[5] = 0.;
                       op[6] = 0.;
                    }
                    op[7] = acum_sc[5][nx][ny][m][np]/( in.d.runmax * acu);
                    printf( "(%hd, %hd)%5.2f%5.2f",
                    nx, ny, out[m].f.c[0][nx], out[m].f.c[1][ny]);
                    printf( "%6.0f ", op[0]);
                    for ( i = 1; i < 7; i++ )
                       printf("%7.4f ", op[i]);
                    printf( "%7.4f\n", op[7]);
                 }
              }
           }
        }
     }
}
```

Sep 30 16:39 1985 host_grey.c Page 1

```
#include <cube/cubedef.h>
#include "host_2d.h"
bin_grey(n)                 /* calculate grey code from binary code */
{
    return(n ^ (n >> 1));
}

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

who(x,y)                  /* calculate node#=whois(x,y) from node location */
{
    x = bin_grey(x);
    y = bin_grey(y);
    x += y << cwdim;
    return(x);
}
```

```
/* This file declares all the external variables in the host program. */

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

#include "2d.def"

char printr[MAXNAM];
FILE *prntptr;
FILE *datptr;

short node,pid,dim,chdim,cwdim,x,y,size,chsize,cwsize;
float pi2, sq2, sqd;
float xm_node, ym_node, chx, chy, cxs, vmw, omm, ami, acu, fnd, time;
float A, d, vrm, omw, pslipf, wslipf, collfac;

float op[14];
/*      op[14] = dummy variables for normalized prshortout
 */

float  acum_sc[D6][NXCELL][NYCELL][NNODES][100];
struct outsht {
        short np,null1;
};
struct outflt {
        float wt;
        float c[DIMEN][NMXCELL], sc[D6][NXCELL][NYCELL];
/*      c[x, y][ncx, ncy] = coord of cell center
 *      sc[0][ncx][ncy] = accum no. parts in cell
 *      sc[1][ncx][ncy] = accum u veloc in cell
 *      sc[2][ncx][ncy] = accum v veloc in cell
 *      sc[3][ncx][ncy] = accum trans enrgy in cell
 *      sc[4][ncx][ncy] = accum spin enrgy in cell
 *      sc[5][ncx][ncy] = accum no. colls in cell
 */
};
struct output {
        struct outsht d;
        struct outflt f;
};
struct output out[NNODES];

/* Input variables      */
struct insht {
        short nprint, nstep, seed;
        short ncx ,ncy, mc, runmax, null2;
};
struct inflt {
        float Tw, beta, uw, u_init, u_enter, mc_enter;
        float xm, ym;
        float dtm, ep, es, ew, esw, g, diff;
};
struct input {
          struct insht d;
          struct inflt f;
};

struct input in;
```

Sep 30 16:41 1985 2d.def Page 1

```
#define MAXNAM  50
#define DIMEN   2       /* # of space dimensions in solution */
#define D3      3       /* # of random velocity components simulated */
#define D6      6       /* # of molecular degrees of freedom simulated */
#define NXCELL  5       /* Max allowed # of x-cells */
#define NYCELL  5       /* Max allowed # of y-cells */
#define NMXCELL 5       /* Max of NXCELL, NYCELL */
#define NM      1200     /* Max allowed number of particles per node */
#define PI      3.141593
#define NNODES  64      /* Number of cosmic cube nodes */
#define NDXMAX  8       /* Number of nodes in x- direction */
#define NDYMAX  8       /* Number of nodes in y- direction */
```

Sep 30 16:42 1985 dat Page 1

```
50      1       100     10      2       10      1
runmax  nprint  nstep   ncx     ncy     mc      seed
16.0    8.0     0.17725
xm      ym      dtm
0.      0.      0.5     2.2567  2.2567  10.
uw      Tw      g       u_init  u_enter mc_enter
1.0     -1.0    1.0     1.0     0.4     0.
ep      es      ew      esw     beta    diff

/*  runmax  = number of runs with a different seed of random number.
    nprint  = number of prints. each print is done every nstep calculations.
    nstep   = number of calculation steps in each print out.
    ncx     = number of cells x-direction in each node.
    ncy     = number of cells y-direction in each node.
    mc      = number of simulated particles in each cell in the initial state.
    seed    = seed value of random number generator, rand().
    xm      = normalized length of x-direction of the simulated flow field.
    ym      = normalized length of y-direction of the simulated flow field.
    dtm     = normalized time step interval.
    uw      = normalized velocity of the bottom wall(boundary).
    Tw      = normalized temperature of the walls(boundaries).
    g       = normalized gravitational acceleration of the simulated field.
    u_init  = x-comp. of stream velocity in the initial state.
    u_enter = x-comp. of new entering stream velocity.
    mc_enter= number of new entering particles per dtm.
    ep      = coefficient of restitution for particle-particle collision.
    es      = coefficient of slip for particle-particle collision.
    ew      = coefficient of restitution for particle-wall collision.
    esw     = coefficient of slip for particle-wall collision.
    beta    = square of the ratio of gyration radius to particle radius.
    diff    = >0 for diffusely reflecting wall, <= for wall with 'ew' and
              'esw' defined.

    These input data are read by 'host_getdat.c' in Host program. They show up
    with in.*, e.g. in.runmax, in Computer Code.

*/
```

Oct 1 23:05 1985 cube_main.c Page 1

```
/* This is the main program of Cube. Each node has an identical code as this
   and each node gets to know its own location in the simulated field and its
   neighbor nodes from subroutine setup(). Then receives data from a neighbor
   node in turn and starts Direct Simulation Monte Carlo calculation. In every
   in.step, updated output is sent to Host through node 0. This output sending
   is done in.nprint times in each of in.runmax runs.
 */
#include <cube/cubedef.h>

#include "2d.h"

main()
{
        int nrun,np,ns,nx,ny;

        setup();                /* setup node network                          */
        cube_getdat();          /* receives input data from HOST               */

        seed = in.seed + mynode(); /* seed has to be initialized in each node
                                      independently */
                                /* independently, for example initial seed */
                                /* = in.seed + mynode() */
        srandom(seed);

        for ( nrun = 0; nrun < in.runmax; nrun++) {
                time = 0.;
                initia();       /* creates in.mc particles in each cell */
                reset();        /* 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 next
                                   node. node 0 sends to Host */

                for ( np = 0; np < in.nprint; np++ ) {
                    for ( ns = 0; ns < in.nstep; ns++ ) {
                        time = ( np * in.nstep + ns + 1 ) * in.dtm;
                        step_type = np * in.nstep + ns + 1;
                        move_glob();            /* moves all particles in in.dtm
                                                   */
                        if (locx == 0) flow();  /* puts entering particles from
                                                   left side(upstream) */
                        recv_ok();          /* receive signal(ready to receive
                                               particles) from neighbor nodes */
                        send_pt();          /* sends particles to neighbors */
                        send_end();         /* sends signal(finish send_pt()) to
                                               neighbor nodes */
                        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
                                               at recv_ok() in the next step(ns)
                                               in this loop */
```

```
                        if ( totmol != 0 ) {
                            if ( totmol > ( 2. * ini_totmol ) ) {
                                  balance();
                            }
                            if ( totmol < ( 0.5 * ini_totmol ) ) {
                                  balance();
                            }
                        }
                        reset();
                        for ( nx = 0; nx < in.ncx; nx++ ) {
                            for ( ny = 0; ny < in.ncy; ny++ ) {
                                if ( ct[0][nx][ny] > time ) continue;
                                if ( ic[0][nx][ny] < 2 ) {
                                    ct[0][nx][ny] = ct[0][nx][ny] + in.dtm;
                                    continue;
                                }
                                coll(nx,ny); /* calculates collision until
                                                celltime(ct[0][nx][ny]) reaches
                                                time */
                            }
                        }
                    }
                    sample();
                    send_out();
                }
                step_type += 1;
                recv_ok();
        }
}
```

Sep 30 15:52 1985 setup.c Page 1

```
#include <cube/cubedef.h>

#include "2d.h"

setup()
{
      MSGDESC in_d;
      sdesc(&in_d,0,0,0,&in,sizeof(in));

      node = mynode();
      pid = mypid();
      dim = cubedim();
      size = 1 << dim;
      recvb(&in_d);       /* receives input data from node (#node-1) */
      if((in_d.node = node + 1) != size) send(&in_d);
                          /* sends input data to node (#node+1) */
      chdim = dim >> 1;
      chsize = 1 << chdim;
      cwdim = dim - chdim;
      cwsize = 1 << cwdim;

/* Each node already knows its node# as its ID. Then we have to give the
*  location coordinate to each node such that its adjacent node# is one
*  bitwize different. The following two equation calculate location coordinate
*  for given node whose # is given by mynode() in <cube/cubedef.h>
*/
      locx = grey_bin(node & (cwsize - 1 ));
      locy = grey_bin((node >> cwdim));
      /* locx and locy are x and y coordinate in the simulated flow field */

/* calclate number of message route for send & recv */
/* number of message route depends on the node's location */
      msg_rt = 8;
      if ((locx == 0) || (locx == (cwsize - 1))) msg_rt = 5;
      if ((locy == 0) || (locy == (chsize - 1))) msg_rt = 5;
      if ((locx == 0) && ((locy == 0) || (locy == (chsize - 1)))) msg_rt = 3;
      if ((locx==(cwsize - 1)) && ((locy==0) || (locy==(chsize - 1)))) msg_rt=3;
}
```

Sep 30 15:54 1985 grey.c Page 1

```
#include <cube/cubedef.h>
#include "2d.h"
bin_grey(n)                     /* calculate grey code from binary code */
{
    return(n ^ (n >> 1));
}

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

who(x,y)                      /* calculate node#=whois(x,y) from node location */
{
    x = bin_grey(x);
    y = bin_grey(y);
    x += y << cwdim;
    return(x);
}
```

Sep 30 15:55 1985 cube_getdat.c Page 1

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

cube_getdat()
{
        int n;

        pi2 = 2 * PI;
        sq2 = sqrt(2.);
        xm_node = in.xm / cwsize;
        ym_node = in.ym / chsize;
        chx = xm_node / in.ncx;
        chy = ym_node / in.ncy;
        vmw = sqrt(in.Tw);
        fnd = in.mc / chx / chy;
        cxs = 1. / ( sq2 * fnd );
        sqd = sq2 * PI * fnd;
        d = 1. / sqrt(sqd);
        ami = 0.25 * in.beta / sqd;
        omm = 1. / sqrt(ami);
        omw = sqrt(in.Tw / ami);
        ini_totmol = in.mc * in.ncx * in.ncy;

        for ( n = 0; n < in.ncx; n++ )
                out.c[0][n] = (n + 0.5) * chx + xm * locx;
        for ( n = 0; n < in.ncy; n++ )
                out.c[1][n] = (n + 0.5) * chy + ym * (chsize - locy - 1);

        acu = in.dtm * in.nstep * in.mc * fnd * cxs * sqrt(2./PI);
        vrm = 2. * sqrt(2./PI);
        pslipf = in.beta * (1.0 - in.es) / (1.0 + in.beta);
        wslipf = (1.0-in.esw) / (1.0+in.beta);
        collfac =  2.0 * chx * chy / cxs;
}
```

Sep 30 15:55 1985 initia.c Page 1

```
/* This subprogram sets the initial states of all the simulated particles. Six
   components of thermal velocity are assigned to each particle by sampling
   randomly from the Maxwell-Boltzmann distribution corresponding to the
   initial temperature and the initial stream velocity in x-direction is set
   by 'in.u_init'. Position coordinates are associated with each particles as
   well as an index label used by the collision routine. This subprogram also
   set the initial cell time used in collision routine as a random fraction of
   'in.dtm'. */

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

initia()
{
        int nx, ny, j, k, l;
        float a, b, bb, aa;
        totmol = ini_totmol;
        out.weight = 1.;
        for ( nx = 0; nx < in.ncx; nx++ ) {
                for ( ny = 0; ny < in.ncy; ny++ ) {
                        ct[0][nx][ny] = rand() * 2. * chx * chy /
                        ( in.mc * in.mc * cxs * vrm );
                        ct[1][nx][ny] = 2. * vrm;
                        for ( j = 0; j < in.mc; j++ ) {
                                k = ny * in.mc + j
                                + nx * in.ncy * in.mc;
                                pt[k].flag = 0;
                                pt[k].wt = out.weight;
                                pt[k].x = out.c[0][nx] + ( rand() - 0.5 )
                                * chx;
                                pt[k].y = out.c[1][ny] + ( rand() - 0.5 )
                                * chy;
                    pt[k].vx = sqrt( -log( rand() ) ) * sin( pi2 * rand() );
                    pt[k].vx += in.u_init;
                pt[k].wx = omm * sqrt( -log( rand() ) ) * sin( pi2 * rand() );
                    pt[k].vy = sqrt( -log( rand() ) ) * sin( pi2 * rand() );
                pt[k].wy = omm * sqrt( -log( rand() ) ) * sin( pi2 * rand() );
                    pt[k].vz = sqrt( -log( rand() ) ) * sin( pi2 * rand() );
                pt[k].wz = omm * sqrt( -log( rand() ) ) * sin( pi2 * rand() );
                        }
                }
        }
}
```

Sep 30 16:04 1985 send_ok.c Page 1

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

send_ok()
{
        MSGDESC sd_ok;
        sdesc (&sd_ok, 0, 0, 0, 0, 0);
        sd_ok.type = step_type + 1;
        if (( locy + 1) < chsize ) {
             sd_ok.node = who(locx,(locy + 1));
             sendb(&sd_ok);
             if ((locx + 1) < cwsize ) {
                  sd_ok.node = who((locx + 1),(locy + 1));
                  sendb(&sd_ok);
             }
             if (locx > 0) {
                  sd_ok.node = who((locx - 1),(locy + 1));
                  sendb(&sd_ok);
             }
        }
        if ((locx + 1) < cwsize ) {
             sd_ok.node = who((locx + 1),locy);
             sendb(&sd_ok);
        }
        if (locx > 0) {
             sd_ok.node = who((locx - 1),locy);
             sendb(&sd_ok);
        }
        if (locy > 0) {
             sd_ok.node = who(locx,(locy - 1));
             sendb(&sd_ok);
             if ((locx + 1) < cwsize) {
                 sd_ok.node = who((locx + 1),(locy - 1));
                 sendb(&sd_ok);
             }
             if (locx > 0) {
                 sd_ok.node = who((locx - 1),(locy - 1));
                 sendb(&sd_ok);
             }
        }
}
```

Sep 30 16:04 1985 sample.c Page 1

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

sample()
{
        int nx, ny;
        int l, nmol;

        for ( nx = 0; nx < in.ncx; nx++ ) {
                for ( ny = 0; ny < in.ncy; ny++ ) {
                        for ( l = 0; l < ic[0][nx][ny]; l++ ) {
                                nmol = lcr[ic[1][nx][ny] + l];
                                out.sc[0][nx][ny] += 1.;
                                out.sc[1][nx][ny] += pt[nmol].vx;
                                out.sc[2][nx][ny] += pt[nmol].vy;
                                out.sc[3][nx][ny] +=
                                        (pt[nmol].vx*pt[nmol].vx
                                        + pt[nmol].vy*pt[nmol].vy +
                                        pt[nmol].vz*pt[nmol].vz);
                                out.sc[4][nx][ny] +=
                                        (pt[nmol].wx*pt[nmol].wx +
                                        pt[nmol].wy*pt[nmol].wy +
                                        pt[nmol].wz*pt[nmol].wz);
#ifdef DEBUG
                                potent += 4. * pt[nmol].y*g / 3.;
#endif
                        }
                }
        }
}
```

```
#include <cube/cubedef.h>

#include "2d.h"

send_out()
{
        int i, j, l, nx, ny;
        MSGDESC out_sd;
        MSGDESC out_rd;
        MSGDESC pls_sd;
        MSGDESC pls_rd;

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

        out_sd.node = HOST;
        if ( node != 0 ) {
               if ( locx == 0 ) out_sd.node = who(( cwsize - 1 ),( locy - 1 ));
               else             out_sd.node = who(( locx - 1 ),locy);
        }
        if ( node != who((cwsize - 1),( chsize - 1)) ) {
               if ( locx == (cwsize - 1) )
                                pls_sd.node = who( 0, (locy + 1) );
               else             pls_sd.node = who( (locx + 1), locy );
        }
        out_sd.type = node + 20000;
        sendb(&out_sd);

        for ( j = locy; j < chsize; j++ ) {
               for ( i = 0; i < cwsize; i++) {
                    if ( ((j == locy) && (i > locx)) || (j > locy)) {
                            sendb(&pls_sd);
                            out_rd.type = who(i,j) + 20000;
                            recvb(&out_rd);
                            out_sd.type = who(i,j) + 20000;
                            recvb(&pls_rd);
                            sendb(&out_sd);
                    }
               }
        }
        recvb(&pls_rd);

        for ( nx = 0; nx < in.ncx; nx++ ) {
                for ( ny = 0; ny < in.ncy; ny++ ) {
                        for ( l = 0; l < 6; l++ ) {
                                out.sc[l][nx][ny] = 0.;
                        }
                }
        }
        for ( nx = 0; nx < in.ncx; nx++ )
                out.c[0][nx] = ( nx + 0.5 ) * chx + xm * locx;
        for ( ny = 0; ny < in.ncy; ny++ )
                out.c[1][ny] = ( ny + 0.5 ) * chy + ym * ( chsize - locy - 1 );
}
```

Sep 30 16:09 1985 move_glob.c Page 1

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

move_glob()
{
        int nmol;
        float dtm1, tt, tty, tttt, ttx, r, ssj, scj, a, b;

        for ( nmol = 0; nmol < totmol; nmol++ ) {
                dtm1 = in.dtm;
                for ( tt = 0.; 0. <= dtm1; dtm1 -= tt ) {
                        if ( in.g == 0. ) {
                           if (pt[nmol].vy == 0.)
                                tty = 1.e+8;
                           else if (pt[nmol].vy > 0.)
                                tty = (in.ym_glob - pt[nmol].y ) / pt[nmol].vy;
                           else
                                tty = - pt[nmol].y / pt[nmol].vy;
                        }
                        else {
                           tttt = pt[nmol].vy * pt[nmol].vy + 2. *
                                        in.g * pt[nmol].y;
                           if ( tttt < 1.e-9 )
                                        tttt = 1.e-9;
                           tty = ( pt[nmol].vy + sqrt( tttt ) ) / in.g;
                           tttt = tttt - 2. * in.g * in.ym_glob;
                           if ( tttt >= 0. && pt[nmol].vy > 0. ) {
                                if (tttt < 1.e-9)
                                        tttt = 1.e-9;
                                tty = (pt[nmol].vy - sqrt(tttt))/in.g;
                           }
                           else;
                        }
                        if (pt[nmol].vx == 0. )
                           ttx = 1.e+8;
                        else if ( pt[nmol].vx > 0. )
                           ttx = ( in.xm_glob - pt[nmol].x ) /pt[nmol].vx;
                        else
                           ttx = -pt[nmol].x /pt[nmol].vx;
                        tt = tty;
                        if ( ttx < tty )
                           tt = ttx;
                        if ( tt > dtm1 )
                           break;
                        pt[nmol].x = pt[nmol].x + pt[nmol].vx * tt *0.9999999;
                        pt[nmol].y = pt[nmol].y + tt * 0.9999999 *
                                (pt[nmol].vy  - 0.5 * tt * in.g);
                        pt[nmol].vy = pt[nmol].vy - in.g * tt;
                        r = 0.5 * d;
                        if ((ttx <= tty) && (pt[nmol].vx > 0.))
                           break;
                        if ( ttx <= tty ) {
                                if (pt[nmol].vx <= 0.) {
                                        /* On the wall at x=0. */
                                        if ( in.diff <= 0. ) {
                                                ssj = wslipf * (pt[nmol].vz
```

```
                                        + pt[nmol].wy * r);
                                scj = wslipf * (pt[nmol].vy
                                        -pt[nmol].wz * r);
                            pt[nmol].vx = - in.ew *pt[nmol].vx;
                                pt[nmol].vy = pt[nmol].vy -
                                        scj * in.beta;
                                pt[nmol].vz = pt[nmol].vz -
                                        ssj * in.beta;
                                pt[nmol].wx = pt[nmol].wx;
                                pt[nmol].wy = pt[nmol].wy -
                                        ssj / r;
                                pt[nmol].wz = pt[nmol].wz +
                                        scj / r;
                                continue;
                        }
                        else {
                                a = vmw * sqrt(- log( rand()
                                        ));
                                b = pi2 * rand();
                                pt[nmol].vy = a * sin(b);
                                pt[nmol].vz = a * cos(b);
                                pt[nmol].vx = vmw * sqrt
                                        (- log( rand()));
                                rot(nmol);
                                continue;
                        }
                }
        }
        else if (pt[nmol].vy > 0. && (in.g == 0. | tttt > 0.)) {
                        /* On the wall at y = in.ym_glob */
                if (in.diff <= 0.) {
                        ssj = wslipf * ( pt[nmol].vz +
                                pt[nmol].wx * r );
                        scj = wslipf * (pt[nmol].vx -
                                pt[nmol].wz * r);
                      pt[nmol].vx =pt[nmol].vx - scj * in.beta;
                        pt[nmol].vy = - in.ew * pt[nmol].vy;
                     pt[nmol].vz = pt[nmol].vz - ssj * in.beta;
                        pt[nmol].wx = pt[nmol].wx - ssj / r;
                        pt[nmol].wy = pt[nmol].wy;
                        pt[nmol].wz = pt[nmol].wz + scj / r;
                        continue;
                }
                else {
                        a = vmw * sqrt(-log(rand()));
                        b = pi2 * rand();
                        pt[nmol].vx = a * sin( b );
                        pt[nmol].vz = a * cos (b );
                        pt[nmol].vy = - vmw * sqrt(-log(
                                rand()));
                        rot(nmol);
                        continue;
                }
        }
        else {
                if (in.diff <= 0.) {
```

Sep 30 16:09 1985 move_glob.c Page 3

```
                                        /* On the wall at y = 0 */
                                        ssj = wslipf * ( pt[nmol].vz -
                                                pt[nmol].wx  * r);
                                        scj = wslipf * (pt[nmol].vx - in.uw +
                                                        pt[nmol].wz * r);
                                      pt[nmol].vx =pt[nmol].vx - scj * in.beta;
                                        pt[nmol].vy = - in.ew *  pt[nmol].vy;
                                     pt[nmol].vz = pt[nmol].vz - ssj * in.beta;
                                        pt[nmol].wx = pt[nmol].wx + ssj / r;
                                        pt[nmol].wy = pt[nmol].wy;
                                        pt[nmol].wz = pt[nmol].wz - scj / r;
                                        continue;
                                }
                                else {
                                        a = vmw * sqrt(-log(rand()));
                                        b = pi2 * rand();
                                        pt[nmol].vx = a * sin(b);
                                        pt[nmol].vz = a * cos(b);
                                        pt[nmol].vy = vmw * sqrt(
                                                -log( rand() ));
                                        rot(nmol);
                                        continue;
                                }
                        }
                }

                pt[nmol].x += ( pt[nmol].vx * dtm1 * 0.999999 );
                pt[nmol].y += ( dtm1 * 0.999999 * ( pt[nmol].vy
                        - 0.5 * in.g * dtm1 * 0.999999 ) );
                pt[nmol].vy -= (in.g * dtm1);
        }
}

rot(nmol)
int nmol;
{
        float aa;

        aa = omw * sqrt(- log( rand()));
        pt[nmol].wx = aa * sin(pi2 * rand());
        aa = omw * sqrt(- log( rand()));
        pt[nmol].wy = aa * sin(pi2 * rand());
        aa = omw * sqrt(- log( rand()));
        pt[nmol].wz = aa * sin(pi2 * rand());
}
```

Sep 30 16:02 1985 flow.c Page 1

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

flow()
{
/* New particles enter from left and go out from right */
/* This subprogram generates new particles entering from left */
/* Stream velocty = ( in.u_enter, 0 ) */

/* in.mc_enter is the exact number of particles entering through left boundary
   per cell per dtm, which may not be integer, therfore integer value 'nflux'
   has to be calculated by the appropriate manner */

        float s, fs1, fs2, v, vn, vp;
        int nmol, nx, ny, l, nflux;
        float a, b, aa, bb, flux;

        s = in.u_enter;
        for ( ny = 0; ny < in.ncy; ny++ ) {
              nflux = 0;
              flux = in.mc_enter;
              if ( flux >= 1. ) {
                    do {
                          nflux += 1;
                          flux -= 1.0;
                    } while ( flux >= 1. );
              }
              a = rand();
              if ( a < flux )  nflux += 1;
              fs1 = s + sqrt( s*s + 2. );
              fs2 = 0.5 * ( 1. + s*( 2.*s - fs1 ) );

              for ( nmol = totmol; nmol < (nflux + totmol); nmol++ ) {
                    do {
                          do {
                                v = 6. * rand() - 3.;
                                vn = v + s;
                          } while ( vn < 0. );

                          a = ( 2. * vn / fs1 ) * exp( fs2 - v*v );
                          b = rand();
                    } while ( a < b );
                    a = pi2 * rand();
                    b = sqrt( -log( rand() ) );
                    vp = b * sin( a );
                    pt[nmol].vz = b * cos( a );
                    pt[nmol].flag = 0;
                    pt[nmol].wt = out.weight;
                    pt[nmol].vx = vn;
                    pt[nmol].vy = - vp;
                    pt[nmol].wx = omm * sqrt(-log(rand())) * sin(pi2 * rand());
                    pt[nmol].wy = omm * sqrt(-log(rand())) * sin(pi2 * rand());
                    pt[nmol].wz = omm * sqrt(-log(rand())) * sin(pi2 * rand());
                    a = rand() * in.dtm;
                    pt[nmol].x = pt[nmol].vx * a;
                    pt[nmol].y = out.c[1][ny]+(rand()-0.5)*chy-0.5*in.g*a*a;
```

Sep 30 16:02 1985 flow.c Page 2

```
                        pt[nmol].vy -= in.g * a;
                        if ( pt[nmol].y <= 0. ) pt[nmol].y = 0.000001;
                }
                totmol += nflux;
        }
}
```

Sep 30 16:16 1985 recv_ok.c Page 1

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

recv_ok()
{
        int count_ok;
        MSGDESC rd_ok;
        sdesc (&rd_ok, 0, 0, 0, 0, 0);
        rd_ok.type = step_type;
        count_ok = 0;
        while ( count_ok < msg_rt )
        {
             recvb(&rd_ok);
             count_ok += 1;
        }
}
```

Sep 30 16:16 1985 send_pt.c Page 1

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

send_pt()
{
	int nmol, nx, ny, node_send;
	int k, m;
	MSGDESC sd;

	sdesc (&sd, 0, 0, 10000, 0, sizeof(struct particle));

	for ( nmol = 0; nmol < totmol; nmol++ ) {
		nx = pt[nmol].x / xm;
		if (nx >= cwsize) {
			bunch(nmol);
			nmol -= 1;
		}
		else {
			ny = pt[nmol].y / ym;
			ny = chsize - 1 - ny;
			if (nx > (locx + 1)) {
				nx = locx + 1;
				pt[nmol].x = (nx + 1) * xm - 0.000001;
			}
			if (nx < (locx - 1)) {
				nx = locx - 1;
				pt[nmol].x = nx * xm + 0.000001;
			}
			if (ny > (locy + 1)) {
				ny = locy + 1;
				pt[nmol].y = (chsize - ny - 1) * ym + 0.000001;
			}
			if (ny < (locy - 1)) {
				ny = locy - 1;
				pt[nmol].y = (chsize - ny) * ym - 0.000001;
			}
			if (nx >= cwsize)	nx = cwsize - 1;
			if (ny < 0)		ny = 0;
			node_send = who(nx,ny);
			if ( node_send != mynode()) {
				sd.node = node_send;
				sd.buf = (char *)&pt[nmol];
				sendb(&sd);
				bunch(nmol);
				nmol -= 1;
			}
		}
	}
}
```

Sep 30 16:18 1985 bunch.c Page 1

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

bunch( nmol )                   /* This subroutine moves the particle
                                   data for particle # totmol-1 into the
                                   memory formerly allocated to particle
                                   nmol, which has just left the node space.
                                   It also reduces totmol and the molecule
                                   index, nmol, by 1.
                                */
int nmol;
{
        pt[nmol].wt = pt[totmol - 1].wt;
        pt[nmol].x = pt[totmol - 1].x;
        pt[nmol].y = pt[totmol - 1].y;
        pt[nmol].vx = pt[totmol - 1].vx;
        pt[nmol].vy = pt[totmol - 1].vy;
        pt[nmol].vz = pt[totmol - 1].vz;
        pt[nmol].wx = pt[totmol - 1].wx;
        pt[nmol].wy = pt[totmol - 1].wy;
        pt[nmol].wz = pt[totmol - 1].wz;
        totmol -= 1;
}
```

Sep 30 16:18 1985 send_end.c Page 1

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

send_end()
{
        MSGDESC sd_end;
        pt[totmol].flag = 1;
        sdesc (&sd_end, 0, 0, 10000,&pt[totmol],sizeof(struct particle));
        if (( locy + 1) < chsize ) {
             sd_end.node = who(locx,(locy + 1));
             sendb(&sd_end);
             if ((locx + 1) < cwsize ) {
                  sd_end.node = who((locx + 1),(locy + 1));
                  sendb(&sd_end);
             }
             if (locx > 0) {
                  sd_end.node = who((locx - 1),(locy + 1));
                  sendb(&sd_end);
             }
        }
        if ((locx + 1) < cwsize ) {
             sd_end.node = who((locx + 1),locy);
             sendb(&sd_end);
        }
        if (locx > 0) {
             sd_end.node = who((locx - 1),locy);
             sendb(&sd_end);
        }
        if (locy > 0) {
             sd_end.node = who(locx,(locy - 1));
             sendb(&sd_end);
             if ((locx + 1) < cwsize) {
                 sd_end.node = who((locx + 1),(locy - 1));
                 sendb(&sd_end);
             }
             if (locx > 0) {
                 sd_end.node = who((locx - 1),(locy - 1));
                 sendb(&sd_end);
             }
        }
}
```

Sep 30 16:18 1985 recv_pt.c Page 1

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

recv_pt()
{
        int count_end, i;
        MSGDESC rd;
        sdesc (&rd, 0, 0, 10000, 0, sizeof(struct particle));
        i = 0;
        count_end = 0;
        while ( count_end < msg_rt )
        {
            rd.buf = (char *)&pt[totmol];
            recvb(&rd);
            if ( pt[totmol].flag == 1 ) count_end += 1;
            if ( pt[totmol].flag == 0 ) {
                if ( pt[totmol].wt == out.weight ) totmol += 1;
                else  weight();
            }
        }
}
```

Sep 30 16:20 1985 weight.c Page 1

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

weight()
{
        int   m, i;
        float a, b;

        b = (float) pt[totmol].wt / out.weight;

        /* pt[totmol].wt is the weighting factor of transferred particle */
        /* weight is the weighting factor of simulated particles in this node */

        m = 0;
        while ( b >= 1. ) {
             m += 1;
             b -= 1.;
        }
        a = rand();
        if ( a < b ) m += 1;
        if ( m == 1 ) {
             pt[totmol].wt = out.weight;
             totmol += 1;
        }
        if ( m > 1 ) {
             pt[totmol].wt = out.weight;

             /* the simulated particles are duplicated in the following loop */
             for ( i = 1; i < m; i++ ) {
                 pt[totmol + i].wt = pt[totmol].wt;
                 pt[totmol + i].x = pt[totmol].x;
                 pt[totmol + i].y = pt[totmol].y;
                 pt[totmol + i].vx = pt[totmol].vx;
                 pt[totmol + i].vy = pt[totmol].vy;
                 pt[totmol + i].vz = pt[totmol].vz;
                 pt[totmol + i].wx = pt[totmol].wx;
                 pt[totmol + i].wy = pt[totmol].wy;
                 pt[totmol + i].wz = pt[totmol].wz;
             }
             totmol += m;
        }
}
```

Sep 30 16:20 1985 balance.c Page 1

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

balance()
{
        int   n, m, i, nmol, i_totmol;
        float a, b, c;

        b = ( float ) ini_totmol / totmol;
        /* want to generate or kill simulated particles with the factor 'b' */
        /* 'b' is a estimated value now, and modified later */

        c = b;
        n = 0;
        while ( b >= 1. ) {
              n += 1;
              b -= 1.;
        }

        i_totmol = totmol;
        /* i_totmol is totmol before balance */

        for ( nmol = 0; nmol < totmol; nmol++ ) {
              m = n;
              a = rand();
              if ( a < b ) m += 1;
              if ( m == 0 ) bunch(nmol);
              if ( m > 1 ) {
             /* the simulated particles are duplicated in the following loop */
                 for ( i = 1; i < m; i++ ) {
                      pt[i_totmol - 1 + i].x = pt[nmol].x;
                      pt[i_totmol - 1 + i].y = pt[nmol].y;
                      pt[i_totmol - 1 + i].vx = pt[nmol].vx;
                      pt[i_totmol - 1 + i].vy = pt[nmol].vy;
                      pt[i_totmol - 1 + i].vz = pt[nmol].vz;
                      pt[i_totmol - 1 + i].wx = pt[nmol].wx;
                      pt[i_totmol - 1 + i].wy = pt[nmol].wy;
                      pt[i_totmol - 1 + i].wz = pt[nmol].wz;
                 }
                 i_totmol += m - 1;
              }
        }
        if ( c > 1 ) totmol = i_totmol;
        /* duplication of particles increments totmol */

        b = c * ( float ) totmol / ini_totmol;
        /* factor 'b' is modified by exact totmol after balance */

        out.weight = out.weight / b;
        /* weighting factor is changed after balance */

        for ( nmol = 0; nmol < totmol; nmol++ ) pt[nmol].wt = out.weight;
}
```

Sep 30 16:20 1985 reset.c Page 1

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

reset()
{
	int nx, ny, nmol;
	int k, m;

	for ( nx = 0; nx < in.ncx; nx++ ) {
		for ( ny = 0; ny < in.ncy; ny++ )
			ic[0][nx][ny] = 0;
	}

	for ( nmol = 0; nmol < totmol ; nmol++ ) {
		nx = (pt[nmol].x - xm * locx) / chx;
		ny = (pt[nmol].y - ym * (chsize - locy - 1)) / chy;
		ic[0][nx][ny] = ic[0][nx][ny] + 1;
	}

	m = 0;
	for ( nx = 0; nx < in.ncx; nx++ ) {
		for ( ny = 0; ny < in.ncy; ny++ ) {
			ic[1][nx][ny] = m;
			m += ic[0][nx][ny];
			ic[0][nx][ny] = 0;
		}
	}

	for ( nmol = 0; nmol < totmol; nmol++ ) {
		nx = (pt[nmol].x - xm * locx) / chx;
		ny = (pt[nmol].y - ym * (chsize - locy - 1)) / chy;
		ic[0][nx][ny] = ic[0][nx][ny] + 1;
		k = ic[1][nx][ny] + ic[0][nx][ny] - 1;
		lcr[k] =  nmol;
		/* Index of first particle in cell [nx][ny] is
		   ic[1][nx][ny].
		   Index of last particle in cell [nx][ny] is
		   ic[1][nx][ny] + ic[0][nx][ny] - 1 .*/
	}
}
```

Sep 30 16:24 1985 coll.c Page 1

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

coll(nx,ny)
int ny, nx;
{
        int k, l, m;
        float vr, a, b, bimp, sinct, cosct, sinph, cosph;
        float e, cose, sinee, xj, yj, zj, al, xl, yl, zl;
        float xk, yk, zk, xkum, ykum, zkum, ab, xkvm, ykvm, zkvm;
        float xkwm, ykwm, zkwm, xkul, ykul, zkul, xkvl, ykvl, zkvl;
        float xkwl,  ykwl, zkwl, uml, vml, wml, r, sum, svm, swm;
        float sul, svl, swl, force1, force2, uuu, vvv, www;


  do {
                        /* Repeat sample coll's till ct[0][nx][ny]]>time */
    do {
                        /* Choose 2 particles from cell [nx][ny] */
                        /* Calculate rel velocity & collision time */
      for ( ; ; ) {
      k = rand() * ic[0][nx][ny] + ic[1][nx][ny];
      l = lcr[k];
      k = rand() * ic[0][nx][ny] + ic[1][nx][ny] ;
      m = lcr[k];
      if ( m != l )
              break;
      }

      vrc[0] = pt[m].vx - pt[l].vx;
      vrc[1] = pt[m].vy - pt[l].vy;
      vrc[2] = pt[m].vz - pt[l].vz;

      vr = sqrt( vrc[0] * vrc[0] + vrc[1] * vrc [1] + vrc[2] * vrc[2] );
      if ( vr > ct[1][nx][ny] )
              ct[1][nx][ny] = vr; /* Largest value of cell rel velocity? */
      a = vr / ct[1][nx][ny];
      b = rand();
    } while ( a < b );

    ct[0][nx][ny] = ct[0][nx][ny] + collfac /
      ( (float) ic[0][nx][ny] * (float) ic[0][nx][ny]  * vr * out.weight ) ;
    out.sc[5][nx][ny] = out.sc[5][nx][ny] + 1.;

    do {
      bimp = d * sqrt( rand() );
      if ( vrc[0] >= vr ) {
              sinct = 0.;
              cosct = 0.;
              sinph = 0.;
              cosph = 0.;
      }
      else {
              cosct = vrc[0] / vr;
              sinct = sqrt( 1. - cosct * cosct );
              cosph = vrc[1] / ( vr * sinct );
```

```
			if ( fabs( cosph ) > 1. )
				cosph = 0.;
			sinph = sqrt( 1. - cosph * cosph );
			if (vrc[2] < 0. )
				sinph = - sinph;
		}
		e = pi2 * rand();
		cose = cos(e);
		sinee = sin(e);
		xj = bimp * cose * sinct;
		yj = - bimp * ( cose * cosct * cosph + sinee * sinph );
		zj = bimp * ( sinee * cosph - cose * cosct * sinph );
		al = sqrt( d * d - bimp * bimp );
		xl = al * vrc[0] / vr;
		yl = al * vrc[1] / vr;
		zl = al * vrc[2] / vr;
		xk = ( xj - xl ) / d;
		yk = ( yj - yl ) / d;
		zk = ( zj - zl ) / d;
	} while ( zk == 0. );

		/* Make axis of molecule m */

		xkum = - xk;
		ykum = - yk;
		zkum = - zk;

		zkvm = zk - 1. / zk;
		ab = sqrt( xk * xk + yk * yk + zkvm * zkvm );
		xkvm = xk / ab;
		ykvm = yk / ab;
		zkvm = zkvm / ab;

		xkwm = yk * zkvm - zk * ykvm;
		ykwm = zk * xkvm - xk * zkvm;
		zkwm = xk * ykvm - yk * xkvm;

		/* Make axis of molecule l */

		xkul = -xkum;
		ykul = -ykum;
		zkul = -zkum;

		xkvl = -xkvm;
		ykvl = -ykvm;
		zkvl = -zkvm;

		xkwl = -xkwm;
		ykwl = -ykwm;
		zkwl = -zkwm;

		/* Calculate coordinates of molecules m and l */
		/* Translational velocity */

		uml = 0.5 * ( xkum * vrc[0] + ykum * vrc[1] + zkum * vrc[2] );
		vml = 0.5 * ( xkvm * vrc[0] + ykvm * vrc[1] + zkvm * vrc[2] );
```

Sep 30 16:24 1985 coll.c Page 3

```
        wml = 0.5 * ( xkwm * vrc[0] + ykwm * vrc[1] + zkwm * vrc[2] );

        /* Spin velocity */
        r = 0.5 * d;
        sum = r * ( pt[m].wx * xkum + pt[m].wy * ykum + pt[m].wz * zkum );
        svm = r * ( pt[m].wx * xkvm + pt[m].wy * ykvm + pt[m].wz * zkvm );
        swm = r * ( pt[m].wx * xkwm + pt[m].wy * ykwm + pt[m].wz * zkwm );
        sul = r * ( pt[l].wx * xkum + pt[l].wy * ykum + pt[l].wz * zkum );
        svl = r * ( pt[l].wx * xkvm + pt[l].wy * ykvm + pt[l].wz * zkvm );
        swl = r * ( pt[l].wx * xkwm + pt[l].wy * ykwm + pt[l].wz * zkwm );

        /* Calculate collisions */

        force1 =  pslipf * ( vml - 0.5 * ( swm + swl ) );
        force2 =  pslipf * ( wml + 0.5 * ( svm + svl ) );

        uml = - in.ep * uml;
        vml = vml - force1;
        wml = wml - force2;

        swl = swl + force1 / in.beta;
        swm = swm + force1 / in.beta;
        svl = svl - force2 / in.beta;
        svm = svm - force2 / in.beta;

        uuu = uml * xkum + vml * xkvm + wml * xkwm;
        vvv = uml * ykum + vml * ykvm + wml * ykwm;
        www = uml * zkum + vml * zkvm + wml * zkwm;

        /* Reset post collision velocity */

        pt[m].vx = 0.5 * ( pt[m].vx + pt[l].vx ) + uuu;
        pt[m].vy = 0.5 * ( pt[m].vy + pt[l].vy ) + vvv;
        pt[m].vz = 0.5 * ( pt[m].vz + pt[l].vz ) + www;

        pt[l].vx = pt[m].vx - 2. * uuu;
        pt[l].vy = pt[m].vy - 2. * vvv;
        pt[l].vz = pt[m].vz - 2. * www;
        pt[m].wx = ( sum * xkum + svm * xkvm + swm * xkwm ) / r;
        pt[m].wy = ( sum * ykum + svm * ykvm + swm * ykwm ) / r;
        pt[m].wz = ( sum * zkum + svm * zkvm + swm * zkwm ) / r;

        pt[l].wx = ( sul * xkum + svl * xkvm + swl * xkwm ) / r;
        pt[l].wy = ( sul * ykum + svl * ykvm + swl * ykwm ) / r;
        pt[l].wz = ( sul * zkum + svl * zkvm + swl * zkwm ) / r;
    } while (ct[0][nx][ny] < time );
}
```

Sep 30 16:28 1985 random.c Page 1

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

#include        <cube/cubedef.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
```

Sep 30 16:28 1985 random.c Page 2

```
#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 + 1 ];
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.
 */
```

Sep 30 16:28 1985 random.c Page 4

```
srandom( x )

    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 )

    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;
        }
```

Sep 30 16:28 1985 random.c Page 5

```
        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 )

    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()
{
	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;
        return( j );
}
```

Sep 30 16:30 1985 2dcube.c Page 1

```
#include <cube/cubedef.h>

#include "2d.h"

char printr[MAXNAM];
short step_type;
short node,pid,dim,chdim,cwdim,x,y,locx,locy,size,chsize,cwsize,msg_rt;
short ic[2][NXCELL][NYCELL], lcr[NM], ini_totmol, b_totmol, totmol;
float pi2, sq2, sqd;
float xm, ym, chx, chy, cxs, um, vmw, time, omm, ami, acu, fnd;
float d, vrm, omw, pslipf, wslipf, collfac;
float ct[2][NXCELL][NYCELL], vrc[D3];
float op[14];
float rand();
unsigned int seed;
#ifdef DEBUG
float potent, totale;
#endif
struct particle pt[NM];
struct output out;
struct input in;
```

Sep 30 16:30 1985 2d.h Page 1

```
/* This file declares all the external variables in the cube program. */
/* Their types are defined in the file '2dcube.c' */

#include <math.h>
#include "2d.def"

extern char printr[MAXNAM];

extern short step_type;
extern short node,pid,dim,chdim,cwdim,x,y,locx,locy,size,chsize,cwsize,msg_rt;

extern short ic[2][NXCELL][NYCELL], lcr[NM], ini_totmol, b_totmol, totmol;
/*      ic[0][nx][ny] = no. particle in cell
 *      ic[1][nx][ny] = summation by columns to nx, ny-1
 *      lcr[nmol] = particle label related to storage  of
 *              particle quantities -- assigned to a particle
 *              during initial problem setup
 */
extern float pi2, sq2, sqd;
extern float xm_node, ym_node, chx, chy, cxs, vmw;
extern float time, omm, ami, acu, fnd;
/*      totmol = total number of molecules in node mesh (MUST ALWAYS
 *              BE LESS THAN NM)
 */
extern float d, vrm, omw, pslipf, wslipf, collfac;
extern float ct[2][NXCELL][NYCELL], vrc[D3];
/*      ct[0][nx][ny] = cell time
 *      ct[1][nx][ny] = maximum relative speed
 *      vrc[x, y, z] = components of collision pair relative speed
 */
extern float op[14];

extern float rand();
extern unsigned int seed;

struct particle {
        int   flag;  /* flag = 1 for end messages, flag = 0 for particles */
        float wt;    /* wt is weighting factor, and initially = 1. */
        float x;
        float y;
        float vx;
        float vy;
        float vz;
        float wx;
        float wy;
        float wz;
};
extern struct particle pt[];

struct output {
        short np,null1;
        float weight;
/*      weight = weighting factor for this node */
        float c[DIMEN][NMXCELL], sc[D6][NXCELL][NYCELL];
/*      c[x, y][ncx, ncy] = coord of cell center
 *      sc[0][ncx][ncy] = accum no. parts in cell
```

```
 *      sc[1][ncx][ncy] = accum u veloc in cell
 *      sc[2][ncx][ncy] = accum v veloc in cell
 *      sc[3][ncx][ncy] = accum trans enrgy in cell
 *      sc[4][ncx][ncy] = accum spin enrgy in cell
 *      sc[5][ncx][ncy] = accum no colls in cell
 */
};
extern struct output out;

/* Input variables      */
struct input {
        short nprint, nstep, seed;
        short ncx ,ncy, mc, runmax, null2;
        float Tw, beta, uw, u_init, u_enter, mc_enter, xm, ym;
        float dtm, ep, es, ev, esw, g, diff;
};

extern struct input in;
```

1. References

BIRD, G. A. 1963 Approach to Translational Equilibrium in a Rigid Sphere Gas. Phys. Fluids 6, p. 1518.

BIRD, G. A. 1976 Molecular Gas Dynamics. Oxford University Press, London.

CAMPBELL, C. S. 1982 Shear Flows of Granular Materials. PhD. Thesis at Caltech.

CAMPBELL, C. S. and BRENNEN, C. E. 1985 Computer Simulation of Granular Shear Flows. J. Fluid Mech. 151, pp. 167-188.

DESHPANDE, S. M., SUBBA RAJU, P. V., RAMANI, N. and NARASIMHA, R. 1978 Monte Carlo Simulation of Low Density Flows. Prod. Indian Academy of Science C 1, pp. 441-458.

SAVAGE, S. B. and SAYED, M. 1984 Stresses Developed by Dry Cohesionless Granular Materials Shered in an Annular Shear Cell. J. Fluid Mech. 142, pp. 391-430.

SU, W. K., FAUCETTE, R. and SEITZ, C. 1985 C Programmer's Guide to the COSMIC CUBE.

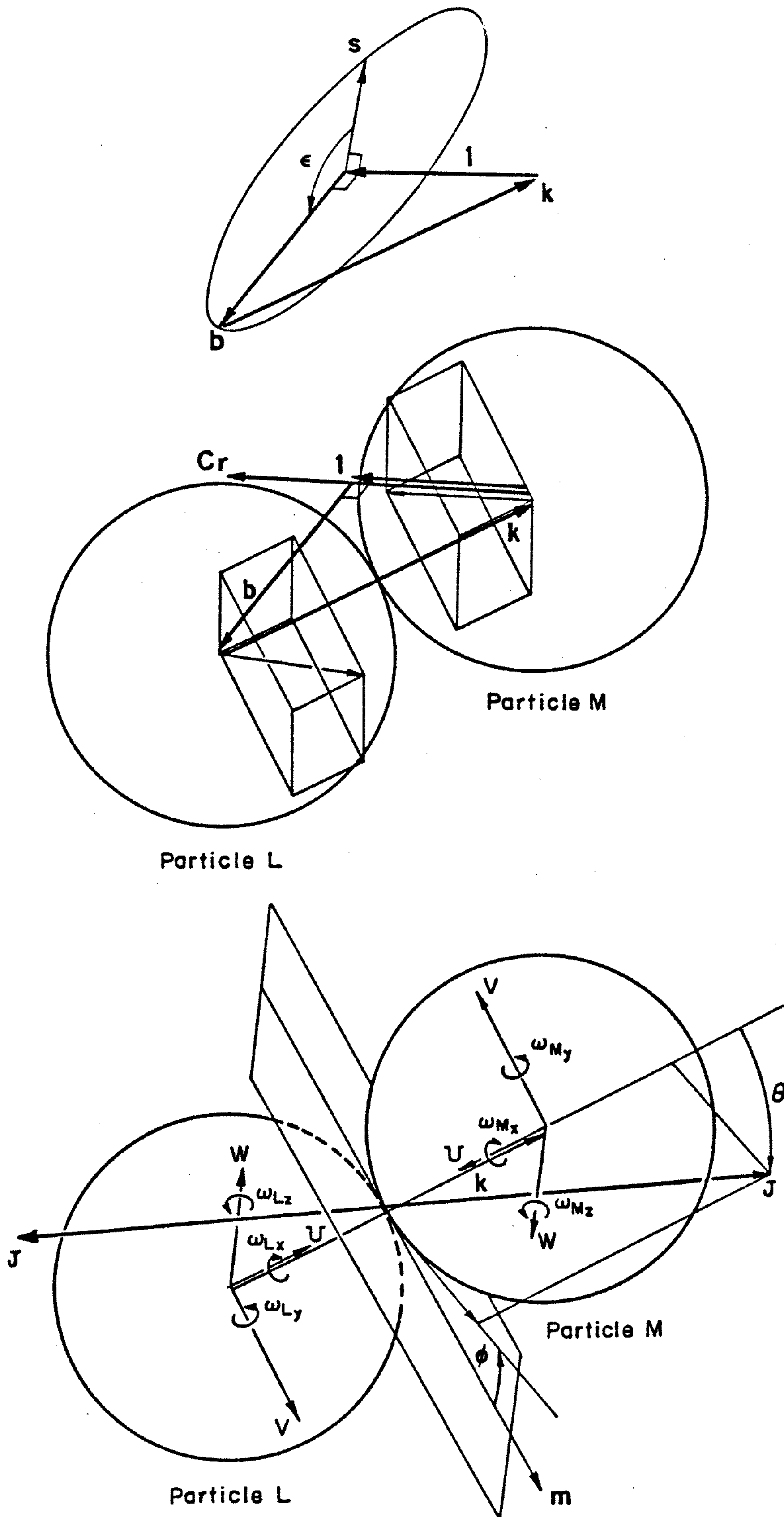


Figure 2.1 Diagrams of particle-particle collisions

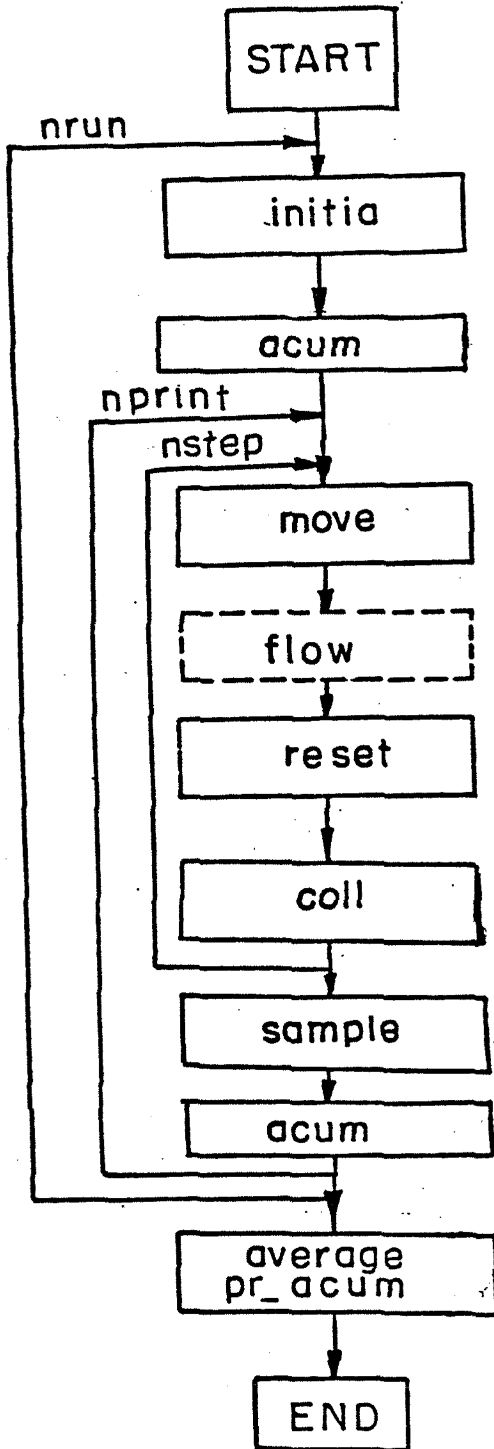


Figure 2.2   Flow chart for the sequential computations of the DSMC method

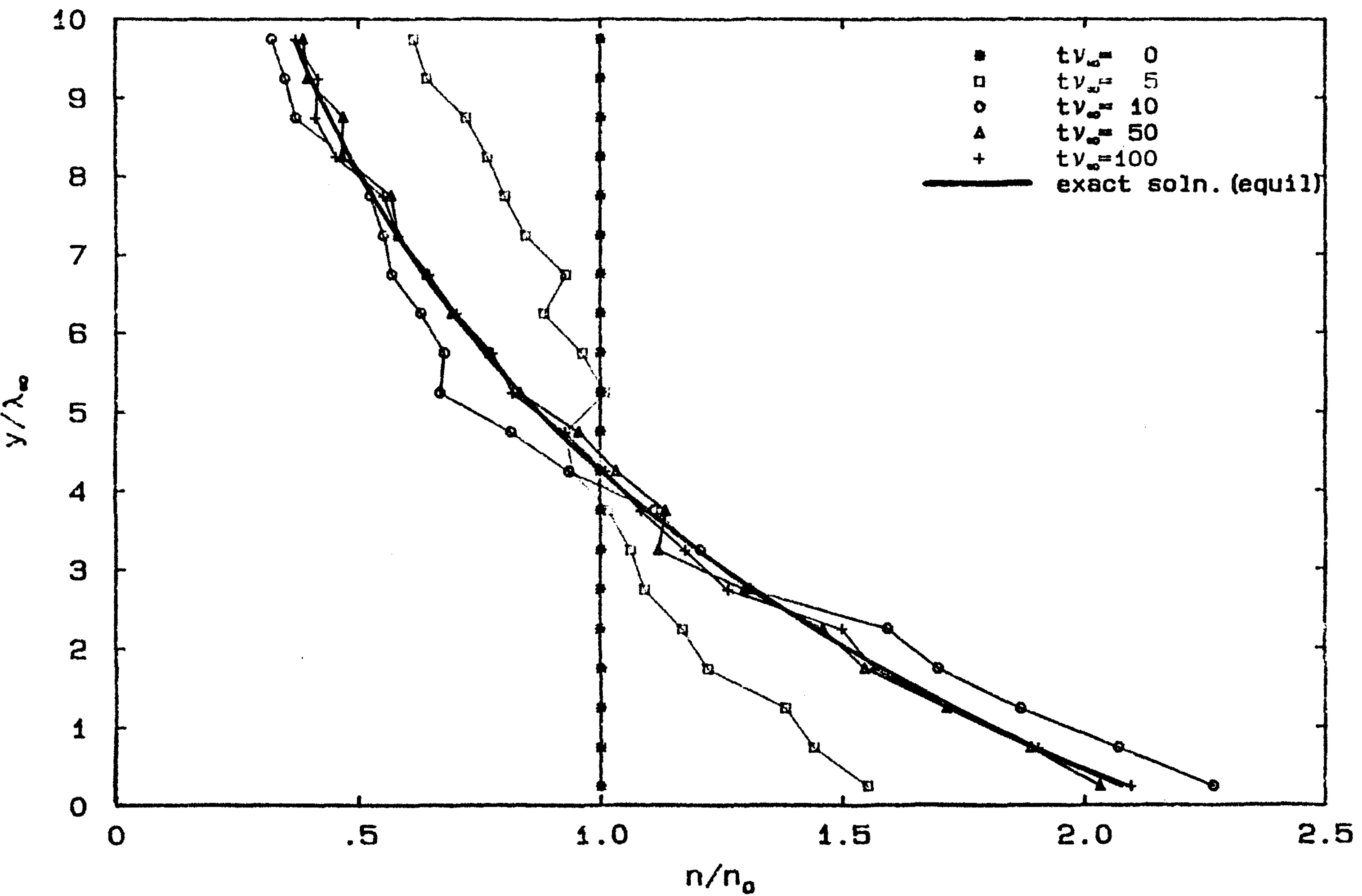


Figure 3.1(a) Number density profile for one-dimensional sedimentation with a single type of particle with $\epsilon_p = 1.0$, $\epsilon_s = -1.0$, specular wall and $g = 0.1$

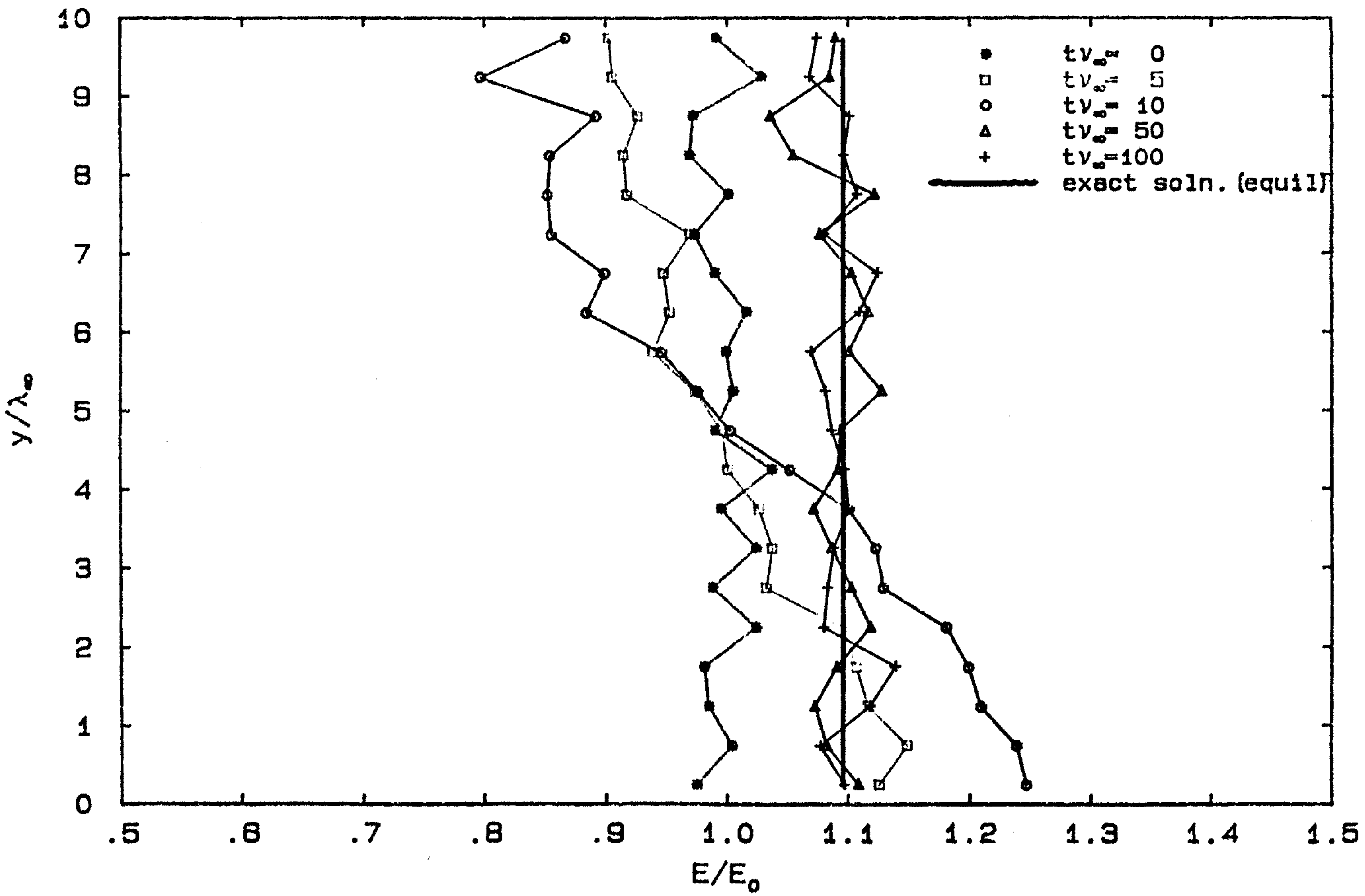


Figure 3.1(b) Total internal energy profile for one-dimensional sedimentation with a single type of particle with $\epsilon_p = 1.0$, $\epsilon_s = -1.0$, specular wall and $g = 0.1$

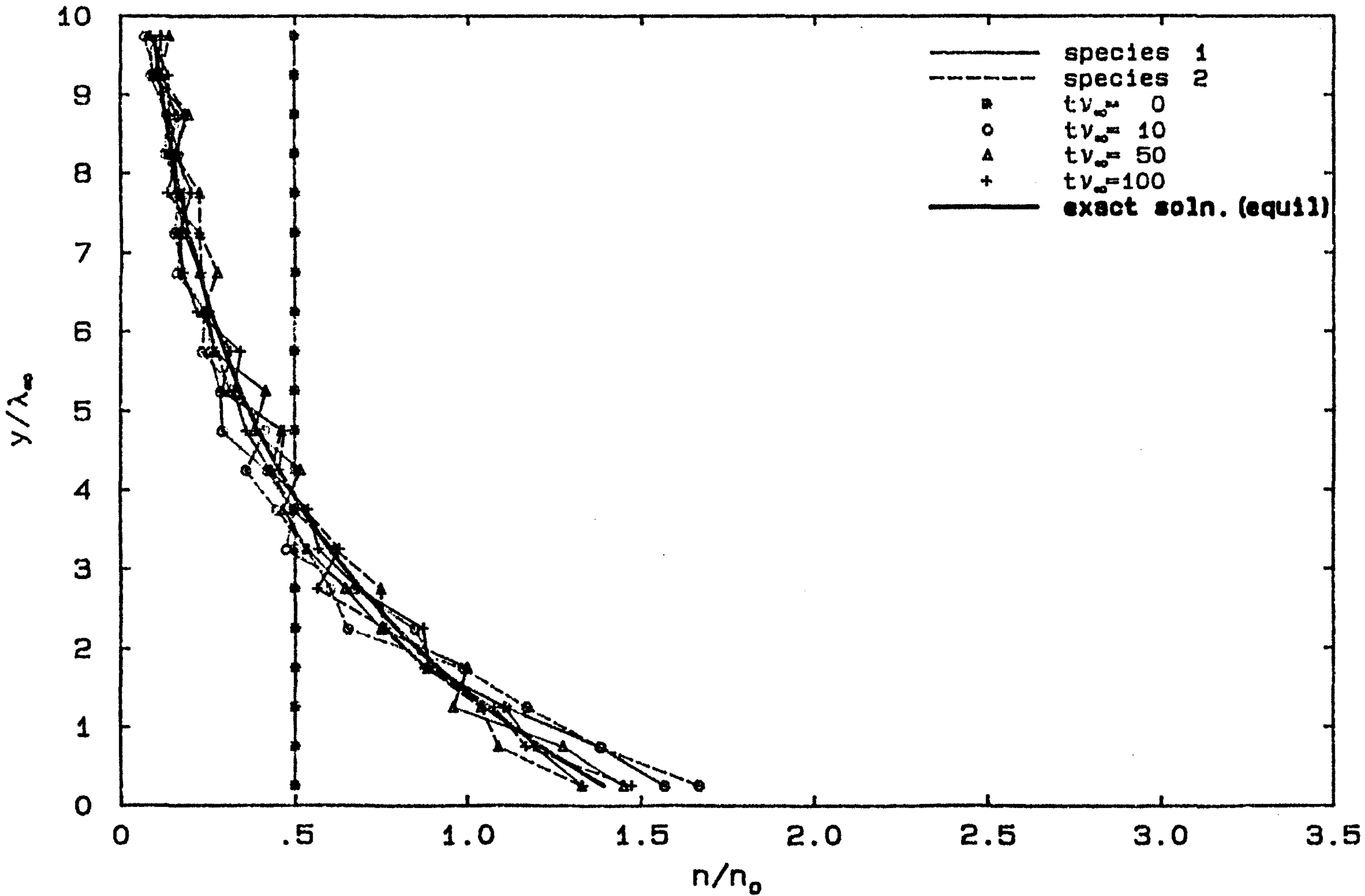


Figure 3.2 Number density profile for one-dimensional sedimentation with two types of particles where $m_1/m_2 = 1.0$, $d_1/d_2 = 0.464$, with $\epsilon_p = 1.0$, $\epsilon_s = -1.0$, specular wall and $g = 0.1$

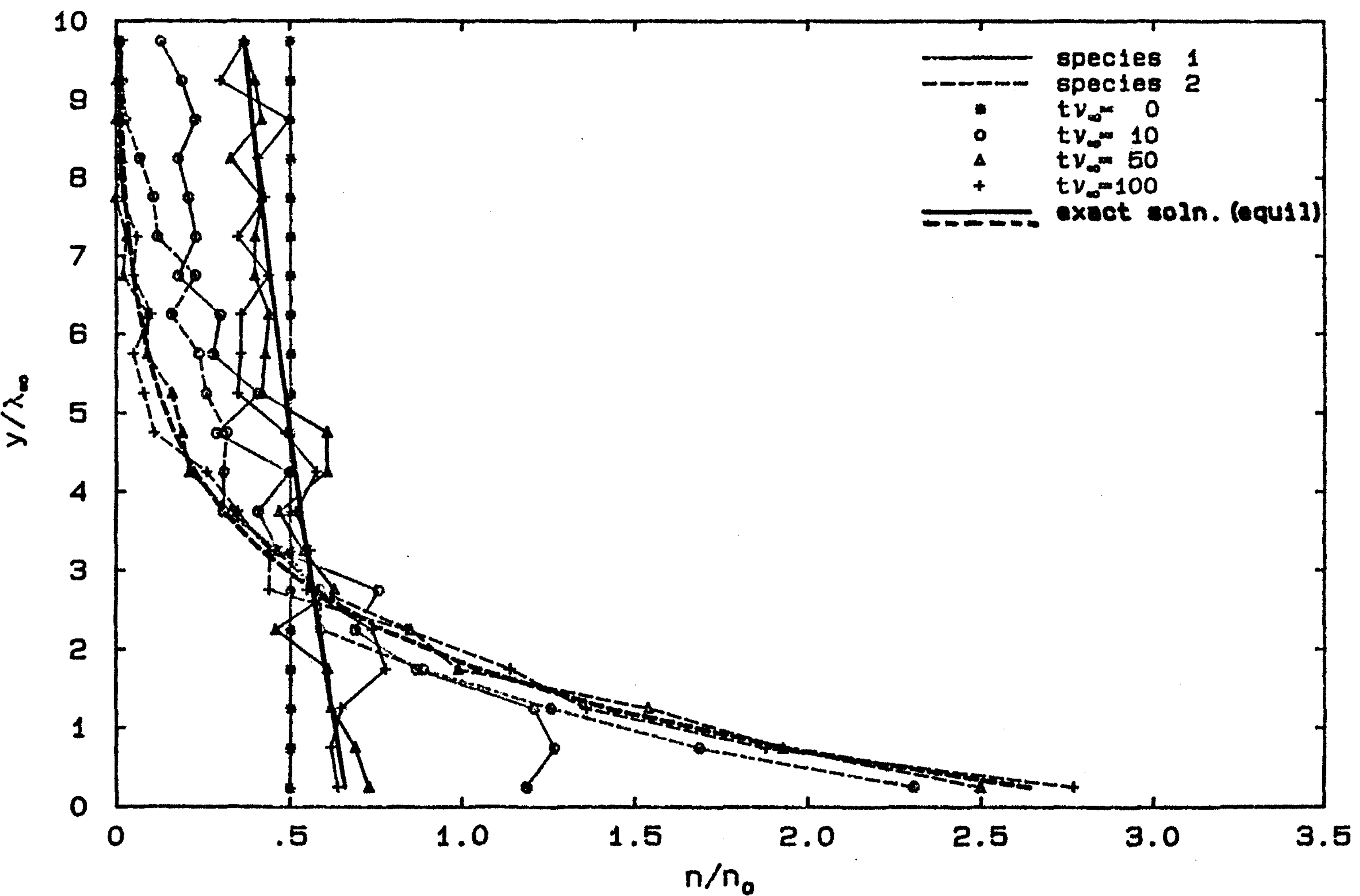


Figure 3.3 Number density profile for one-dimensional sedimentation with two types of particles where $m_1/m_2 = 0.1$, $d_1/d_2 = 1.0$ with $\epsilon_p = 1.0$, $\epsilon_s = -1.0$, specular wall and $g = 0.1$

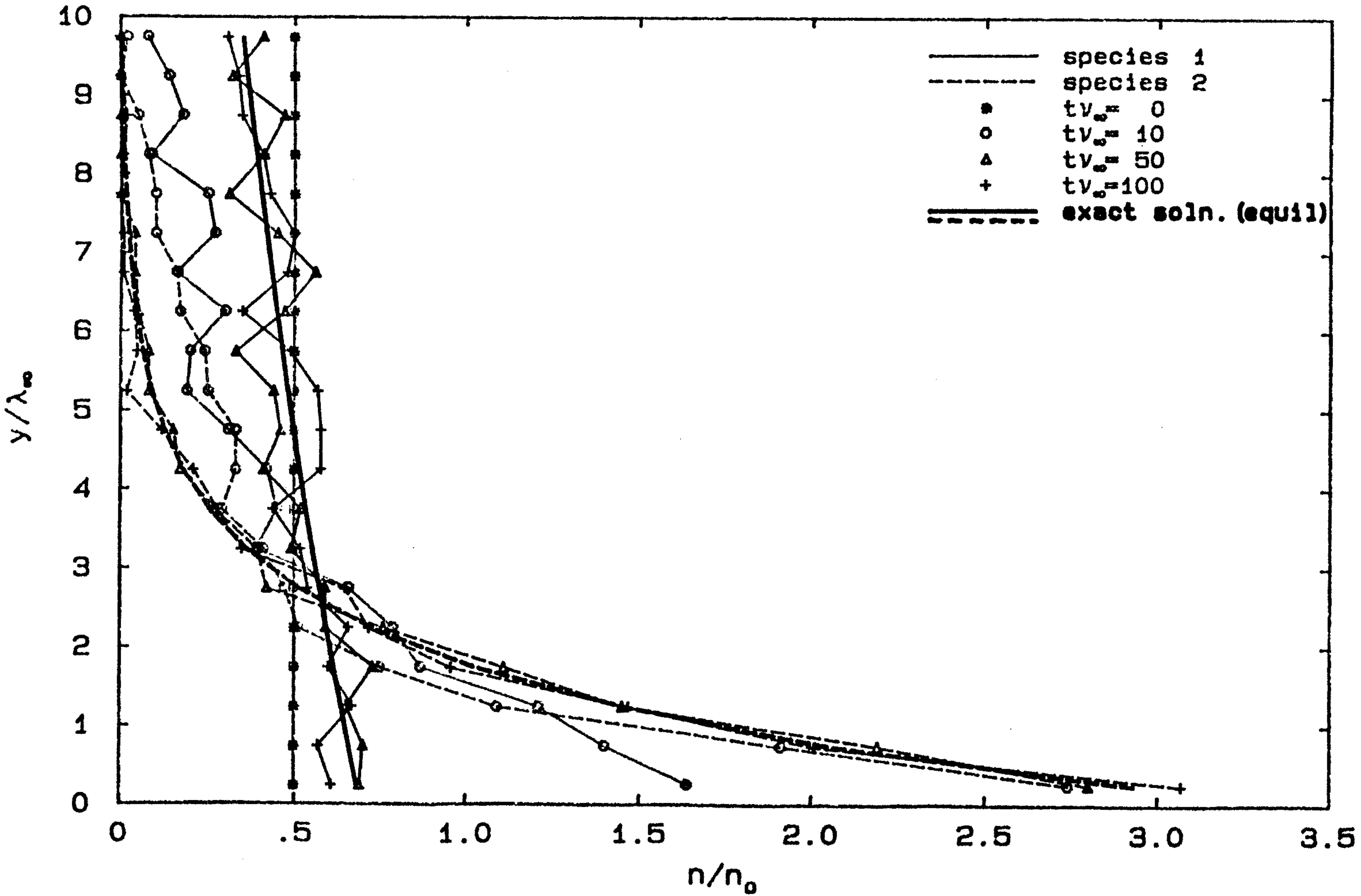


Figure 3.4 Number density profile for one-dimensional sedimentation with two types of particles where $m_1/m_2 = 0.n1$ and $d_1/d_2 = 0.464$, with $\epsilon_p = 1.0$, $\epsilon_s = -1.0$, specular wall and $g = 0.1$

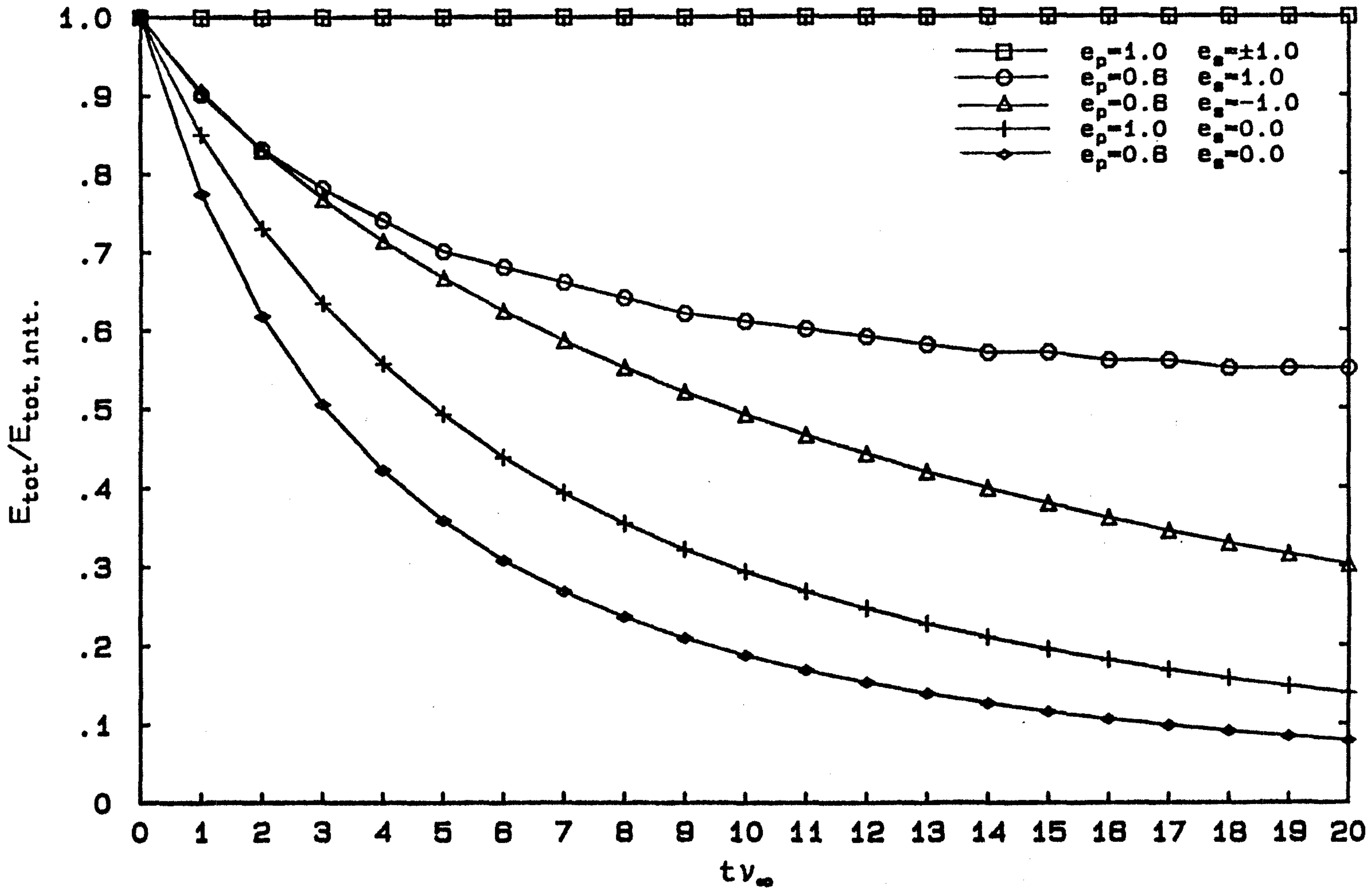
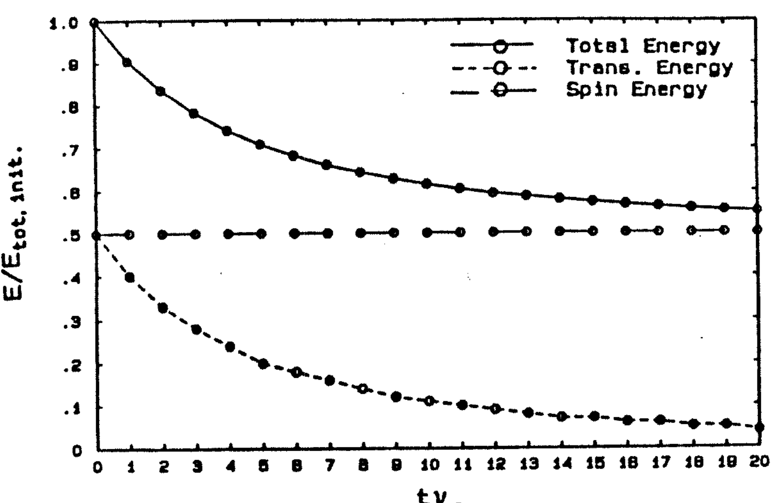
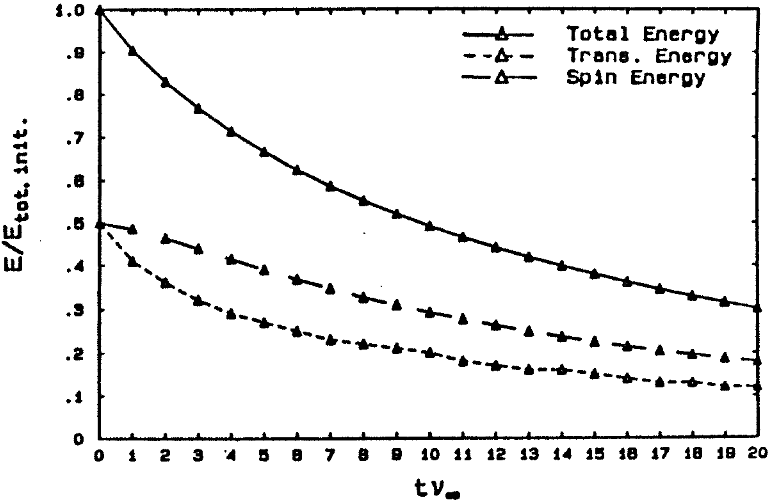


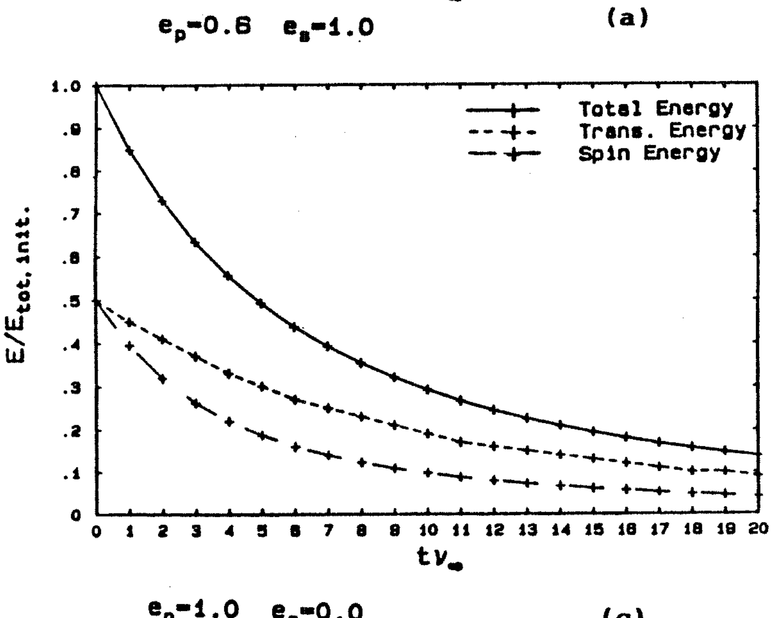
Figure 3.5 Total energy vs. time for different conditions of collision



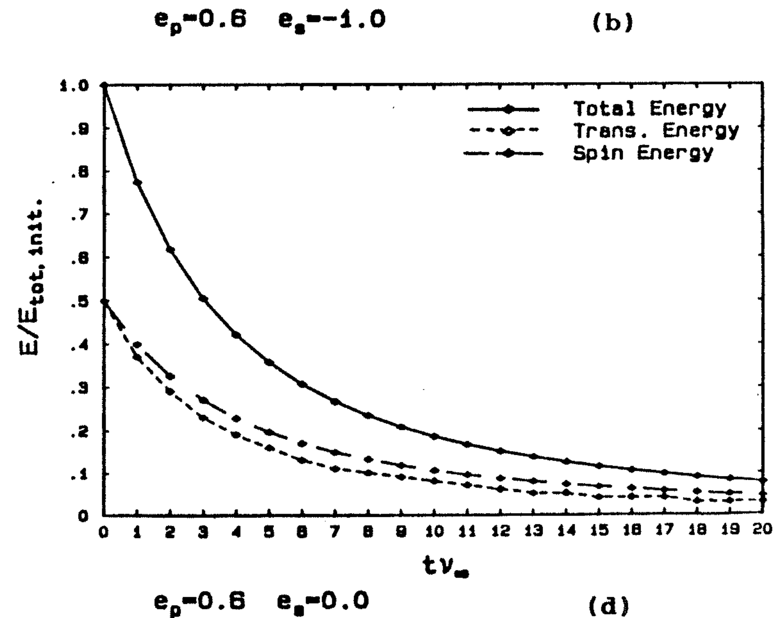
e_p=0.6 e_s=1.0



e_p=0.6 e_s=-1.0



e_p=1.0 e_s=0.0



e_p=0.6 e_s=0.0

Figure 3.6 Total translational and spin energy vs. time

0	1	3	2	6	7	5	4
8	9	11	10	14	15	13	12
24	25	27	26	30	31	29	28
16	17	19	18	22	23	21	20
48	49	51	50	54	55	53	52
56	57	59	58	62	63	61	60
40	41	43	42	46	47	45	44
32	33	35	34	38	39	37	36

←——→ ID numbers differ by a single bit
←- - -→ ID numbers differ by 2 bits

Figure 4.1 The layout of nodes for minimizing the communication time between nodes

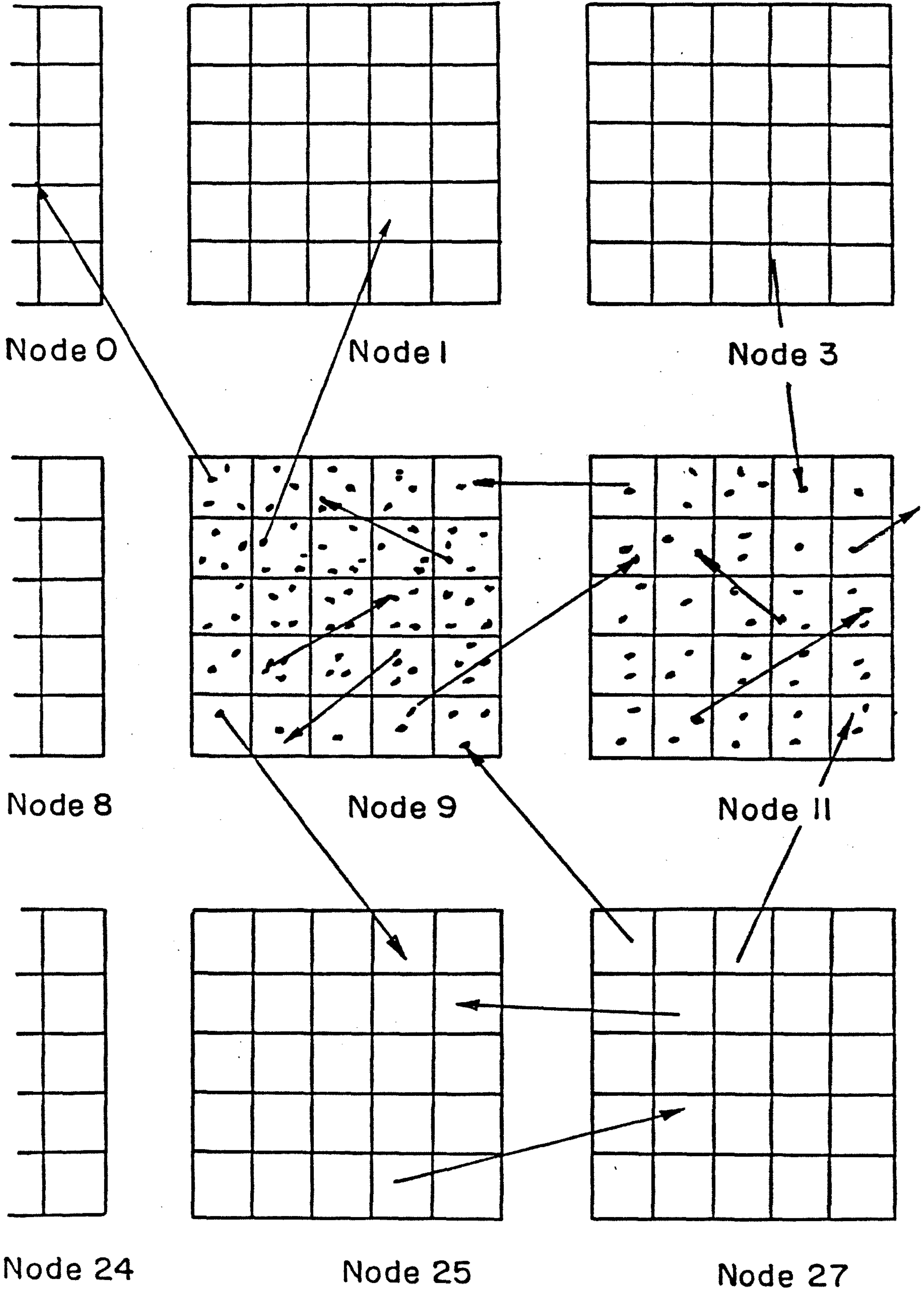


Figure 4.2 Typical inter-node motion of simulated particles

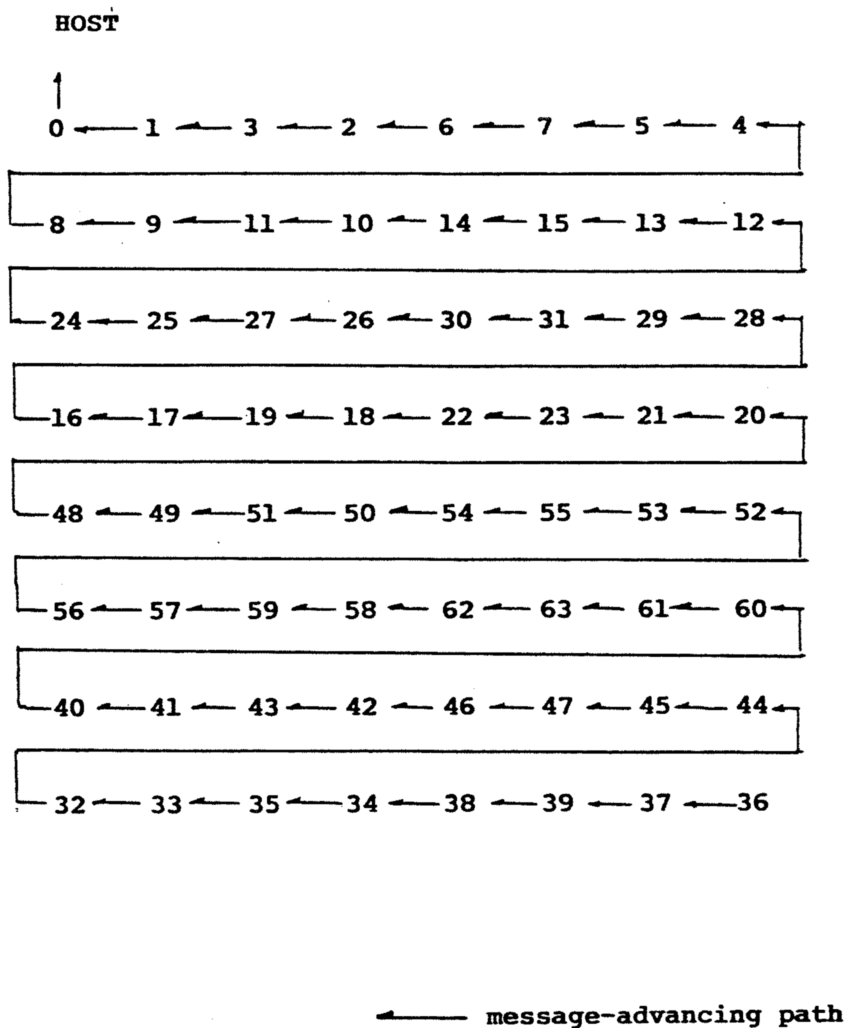


Figure 4.3 Message-advance system for sending output to Host

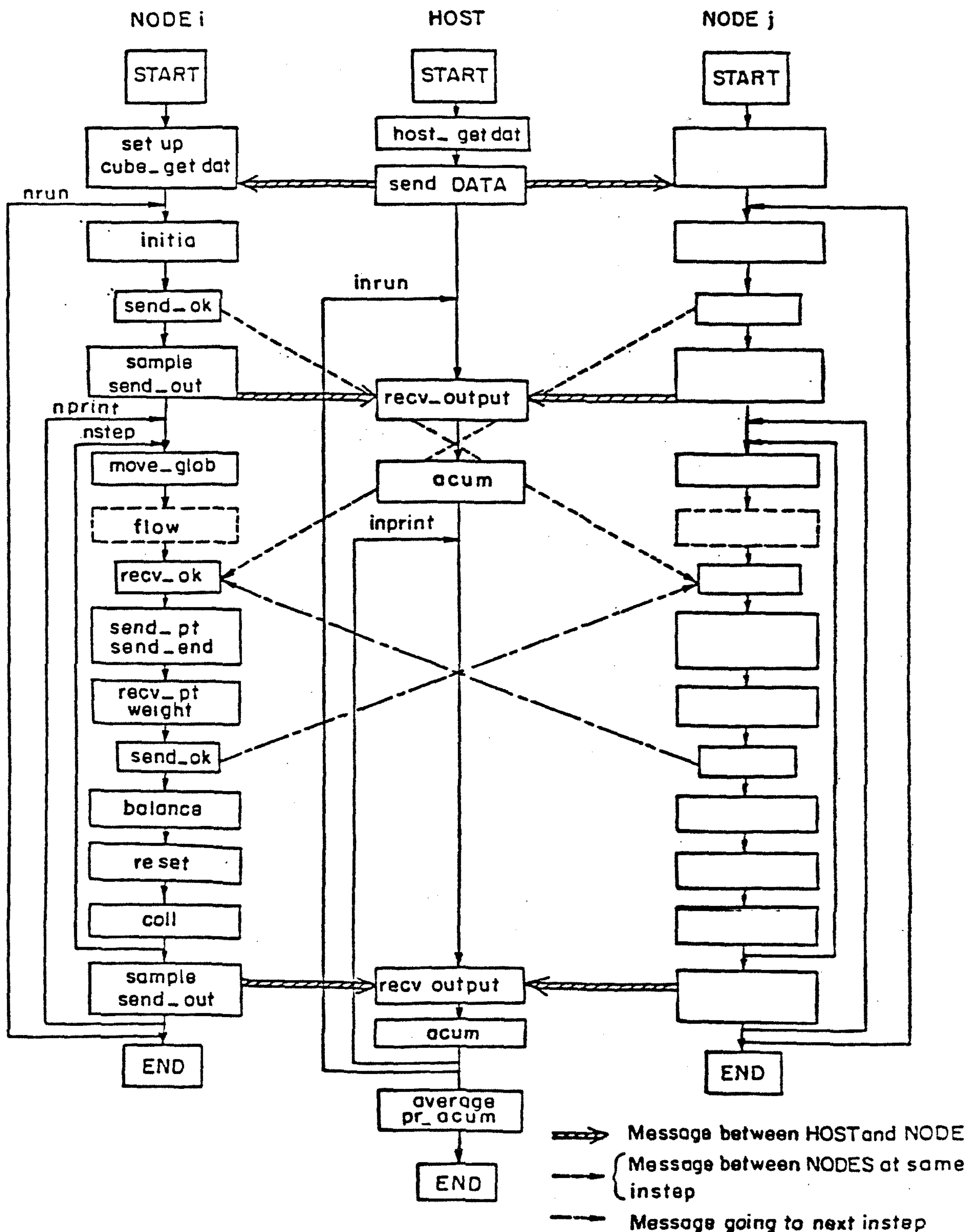


Figure 4.4 Flow chart for the concurrent computations of the DSMC method

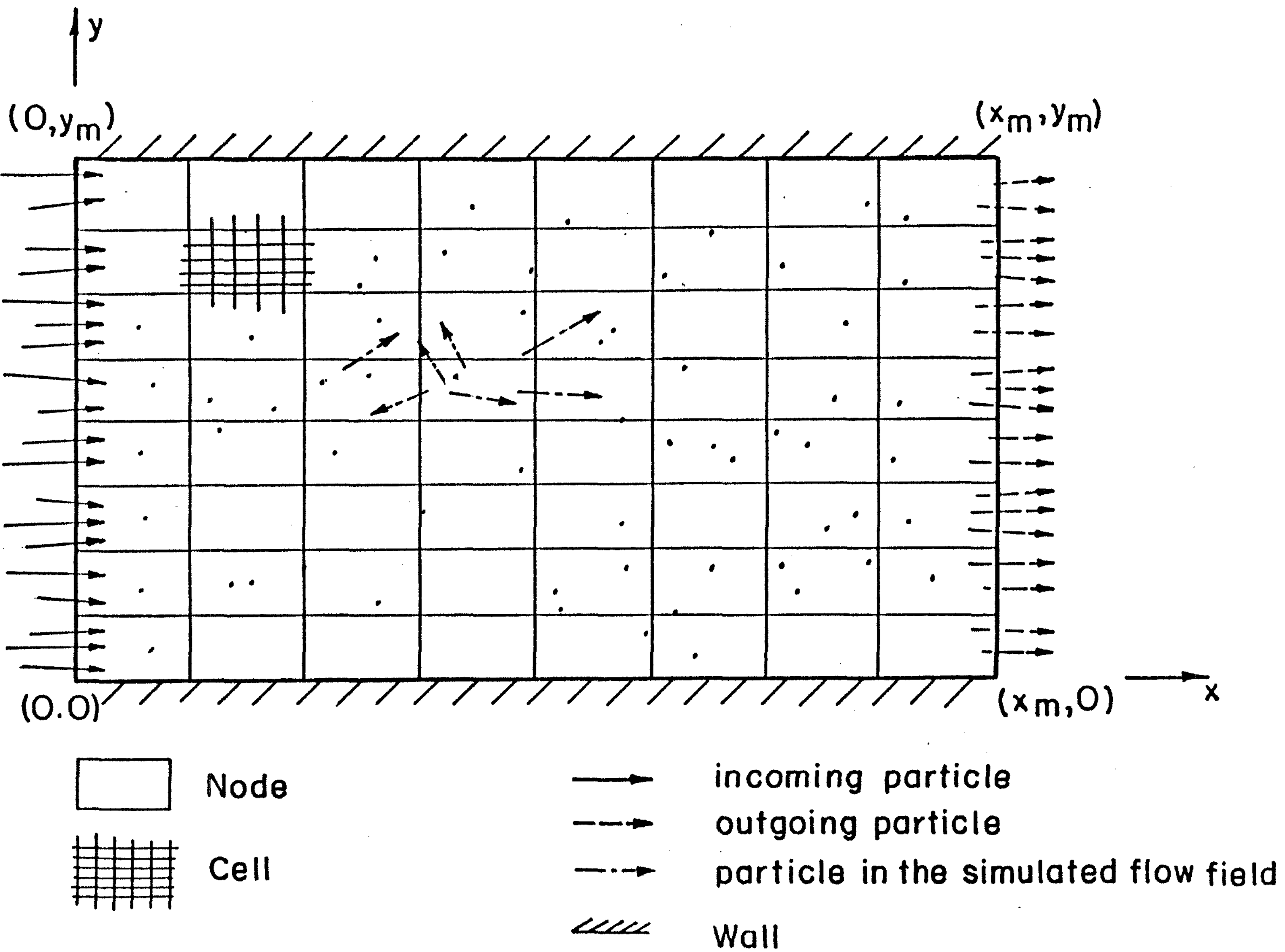


Figure 5.1 The simulated flow field and the node and cell layout

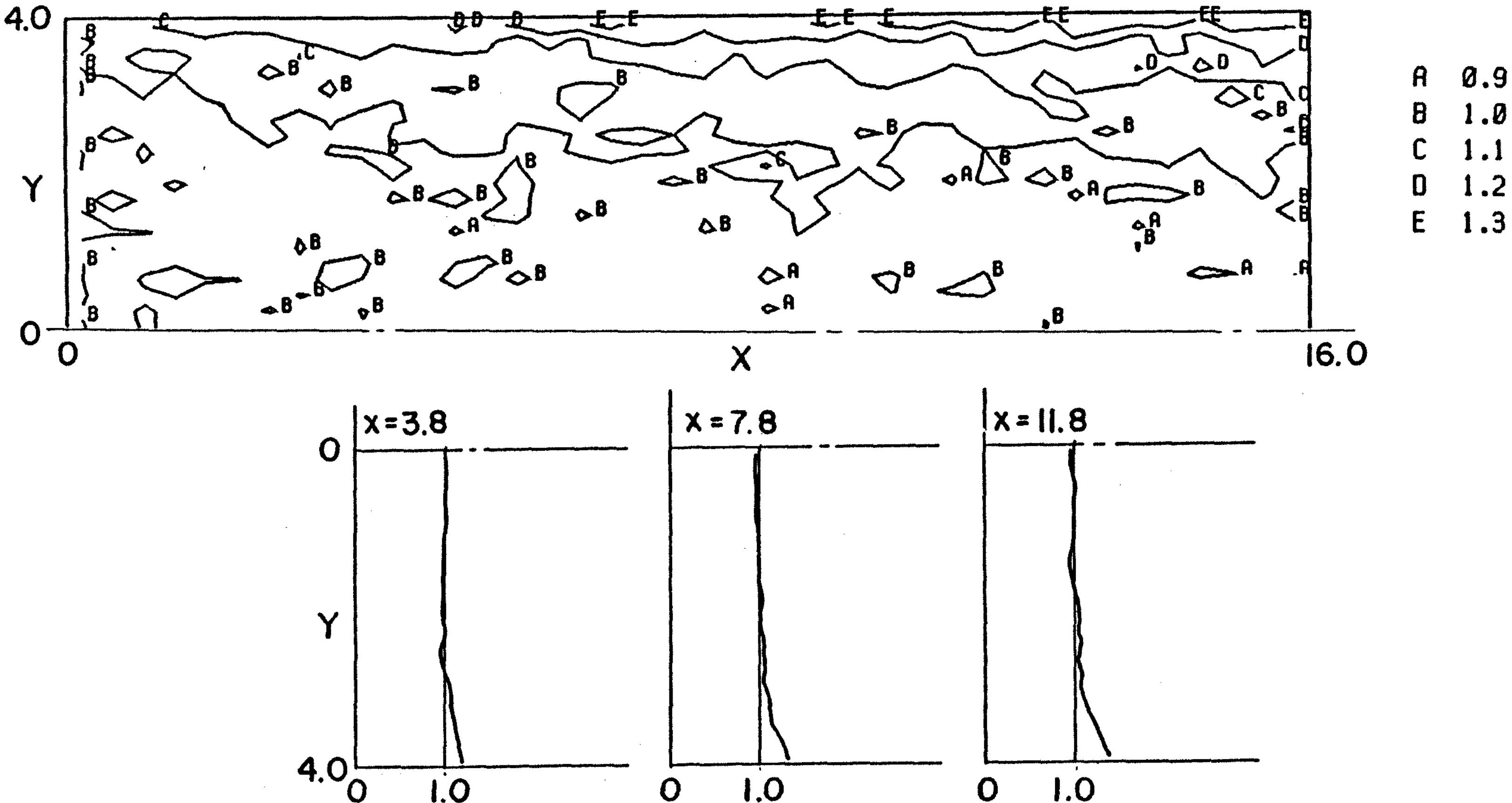


Figure 5.2(a) Number density profile of channel flow of granular material in the steady state with Re = 32.6, M = 2.76 and K_n = 0.125 (ϵ_p = 0.6, ϵ_s = 0.0, ϵ_w = 0.8, ϵ_{sw} = 0.0)

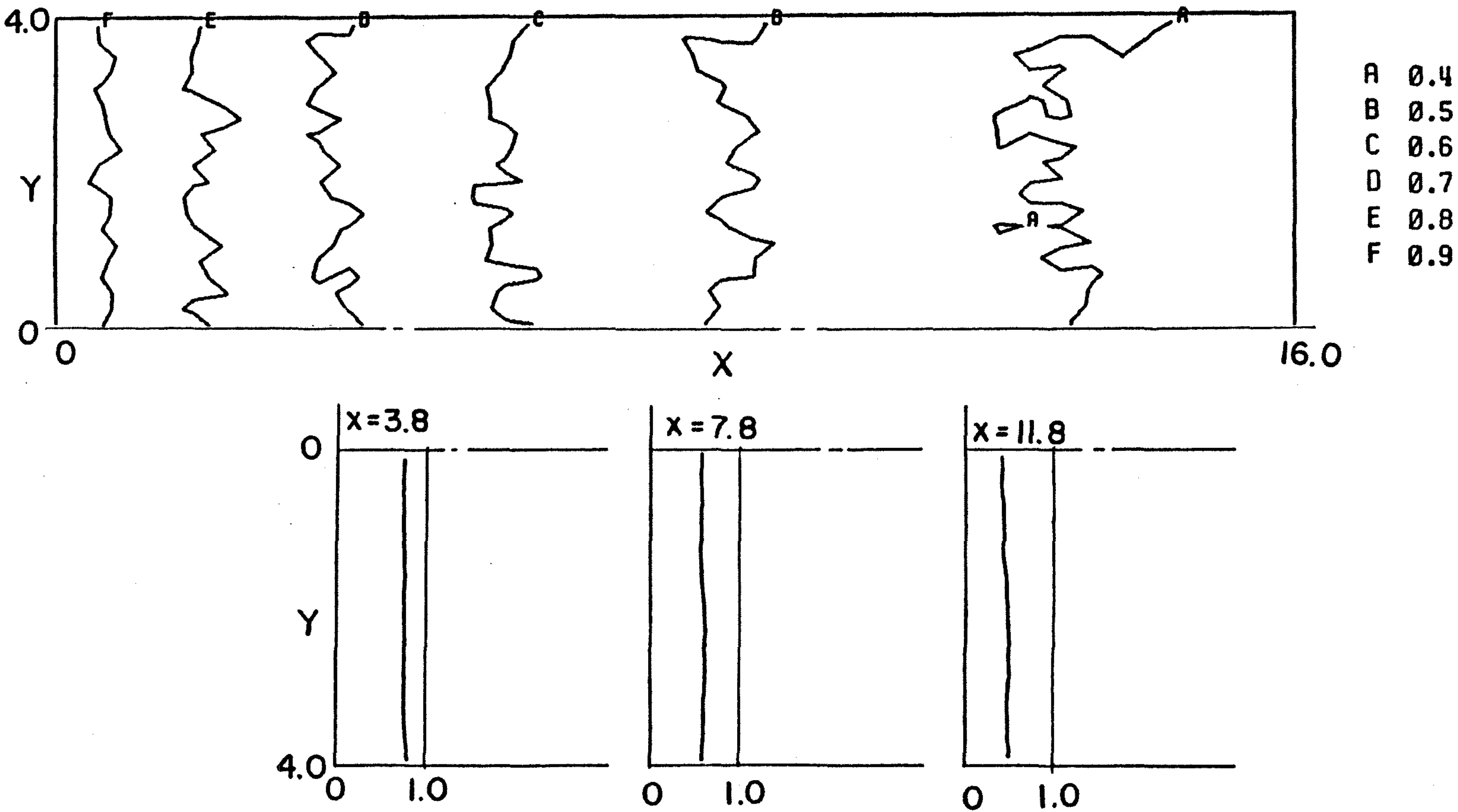


Figure 5.2(b) Temperature profile of channel flow of granular material in the steady state with Re = 32.6, M = 2.76 and K_n = 0.125 (ϵ_p = 0.6, ϵ_s = 0.0, ϵ_w = 0.8, ϵ_{sw} = 0.0)

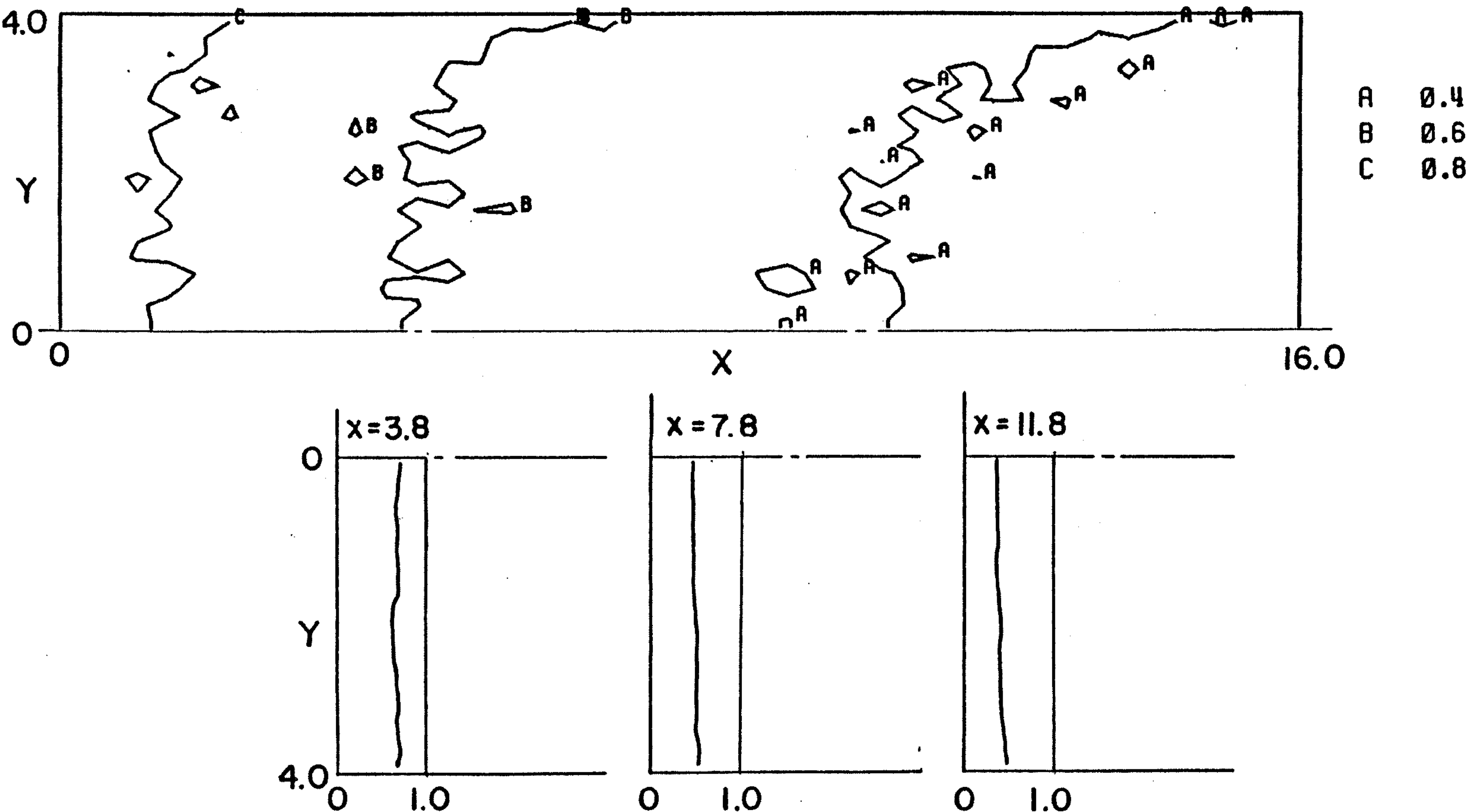


Figure 5.2(c) Pressure profiles of channel flow of granular material in the steady state with Re = 32.6, M = 2.76 and K_n = 0.125 (ϵ_p = 0.6, ϵ_w = 0.0, ϵ_w = 0.8, ϵ_{sw} = 0.0)

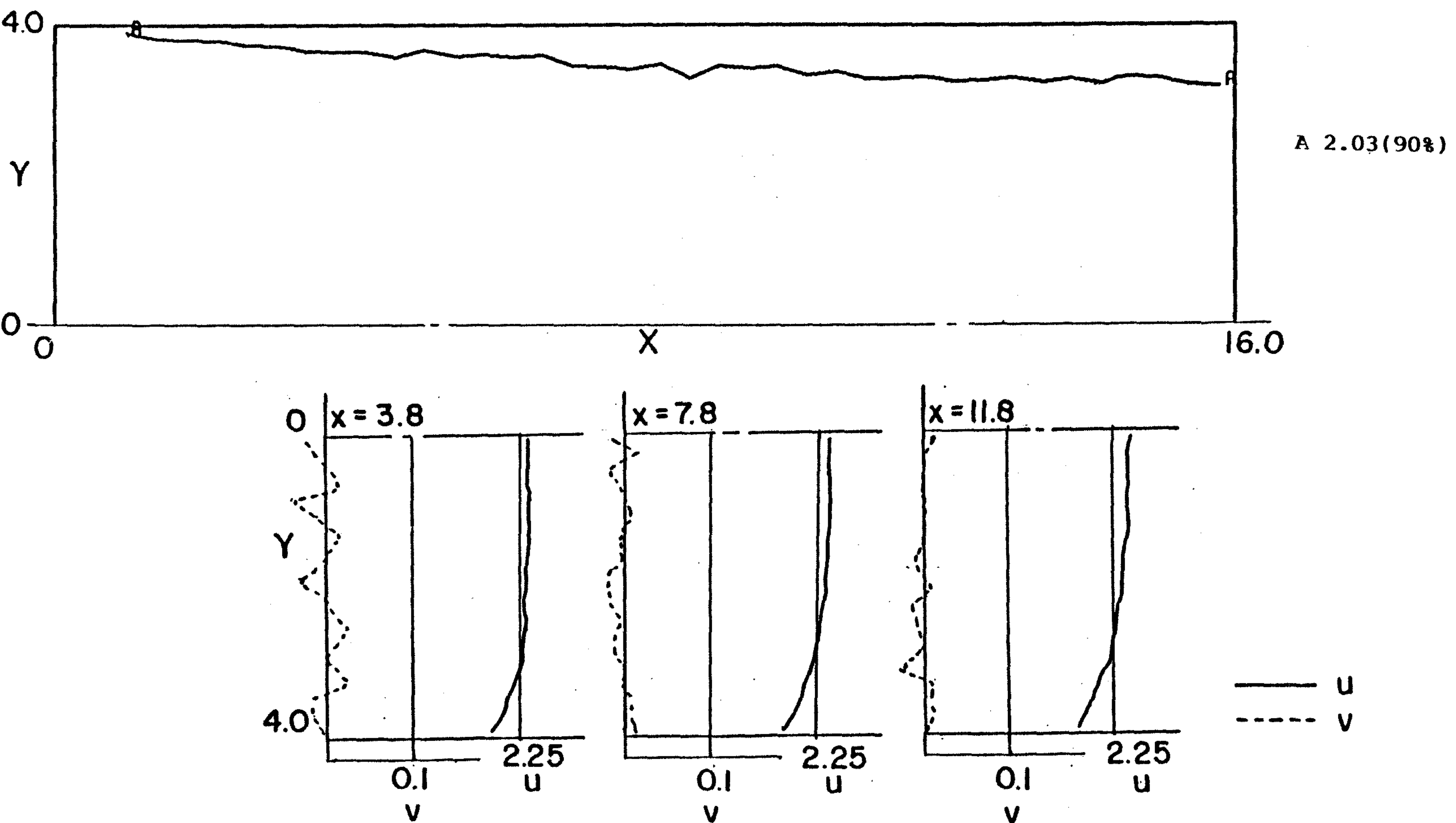


Figure 5.2(d) Velocity field of channel flow of granular material in the steady state with Re = 32.6, M = 2.76 and K_n = 0.125 (ϵ_p = 0.6, ϵ_s = 0.0, ϵ_w = 0.8, ϵ_{sw} = 0.0)

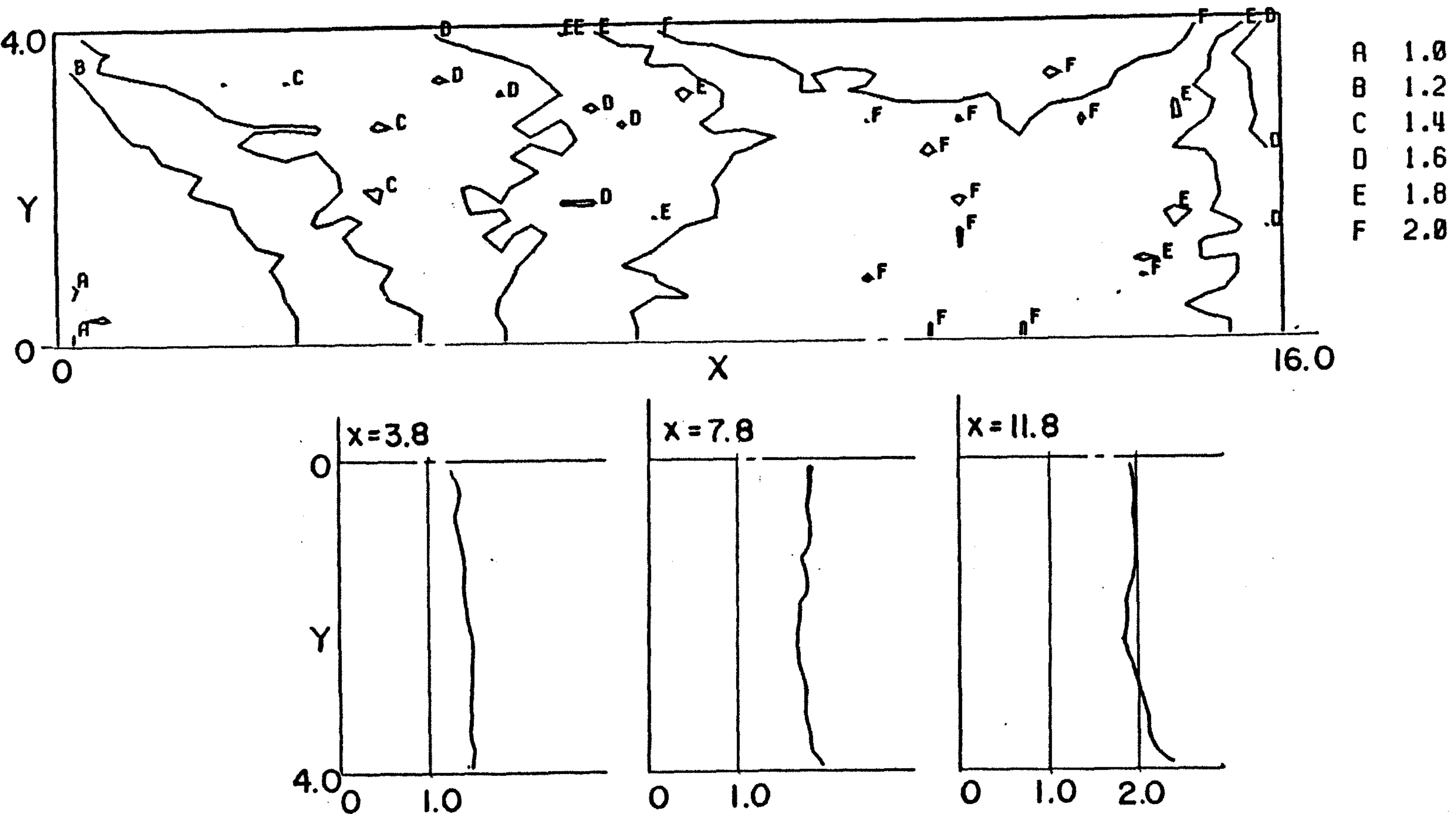


Figure 5.3(a) Number density profile of channel flow of gas in the in the steady state with Re = 32.6, M = 2.76 and K_n = 0.125 (ϵ_p = 1.0, ϵ_s = -1.0, diffusely reflecting walls, T_w = 1)

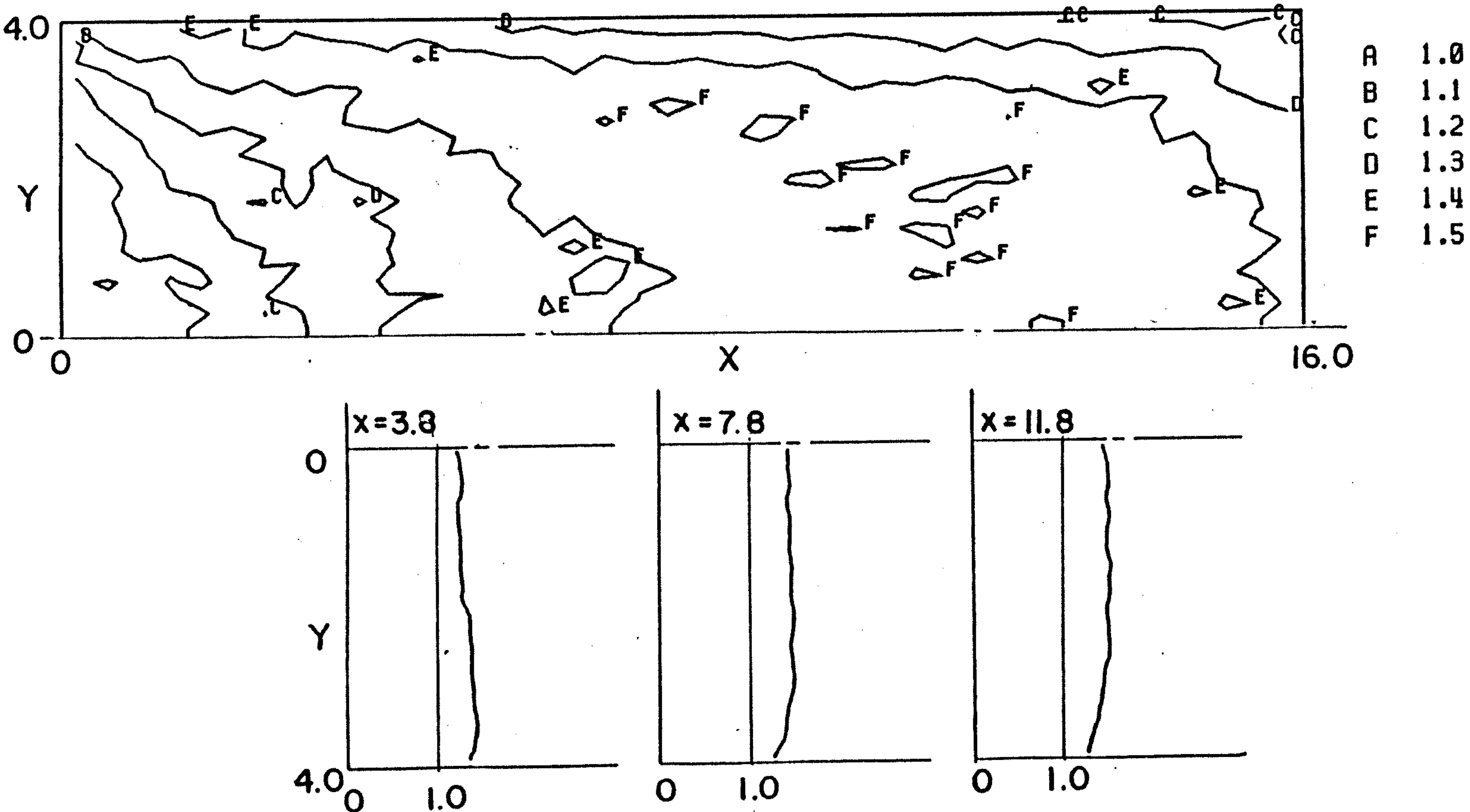


Figure 5.3(b) Temperature profile of channel flow of gas in the steady state with Re = 32.6, M = 2.76 and K_n = 0.125 (ϵ_p = 1.0, ϵ_s = -1.0, diffusely reflecting walls, T_w = 1)

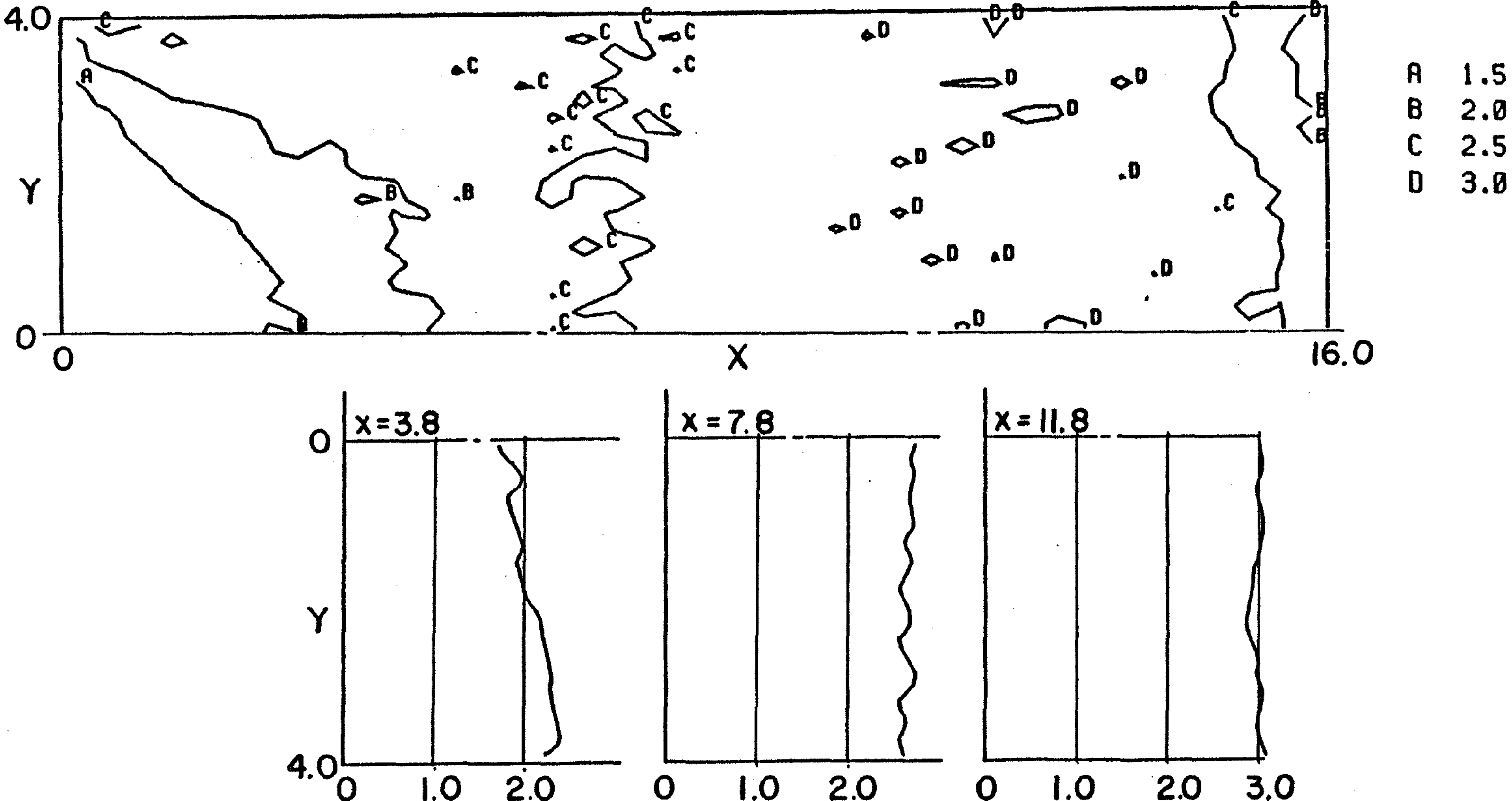


Figure 5.3(c) Pressure profile of channel flow of gas in the steady state with Re = 32.6, M = 2.76 and K_n = 0.125 (ϵ_p = 1.0, ϵ_s = -1.0, diffusely reflecting walls, T_w = 1)

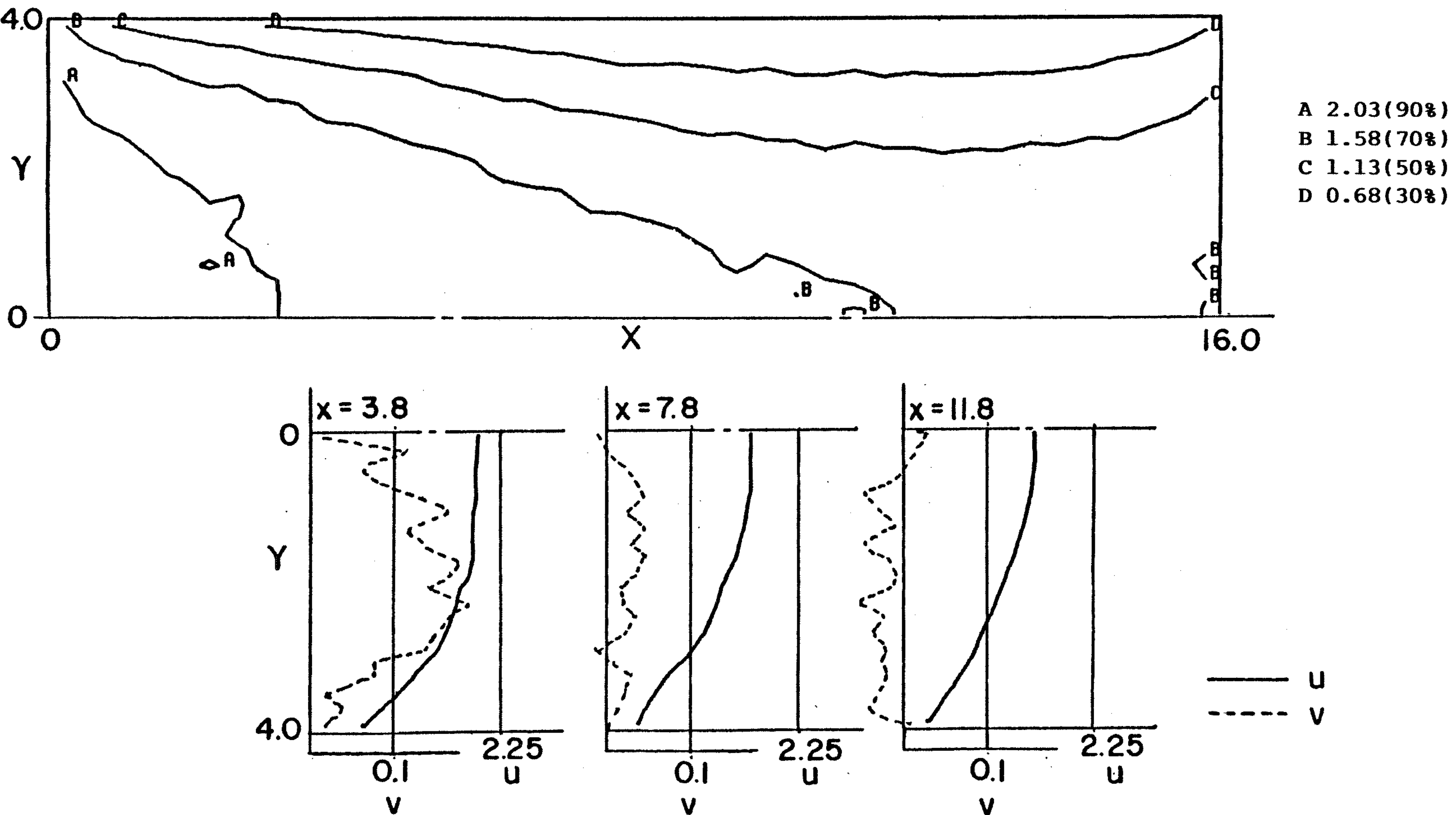


Figure 5.3(d) Velocity field of channel flow of gas in the steady state with Re = 32.6, M = 2.76 and K_n = 0.125 (ϵ_p = 1.0, ϵ_s = -1.0, diffusely reflecting walls, T_w = 1)

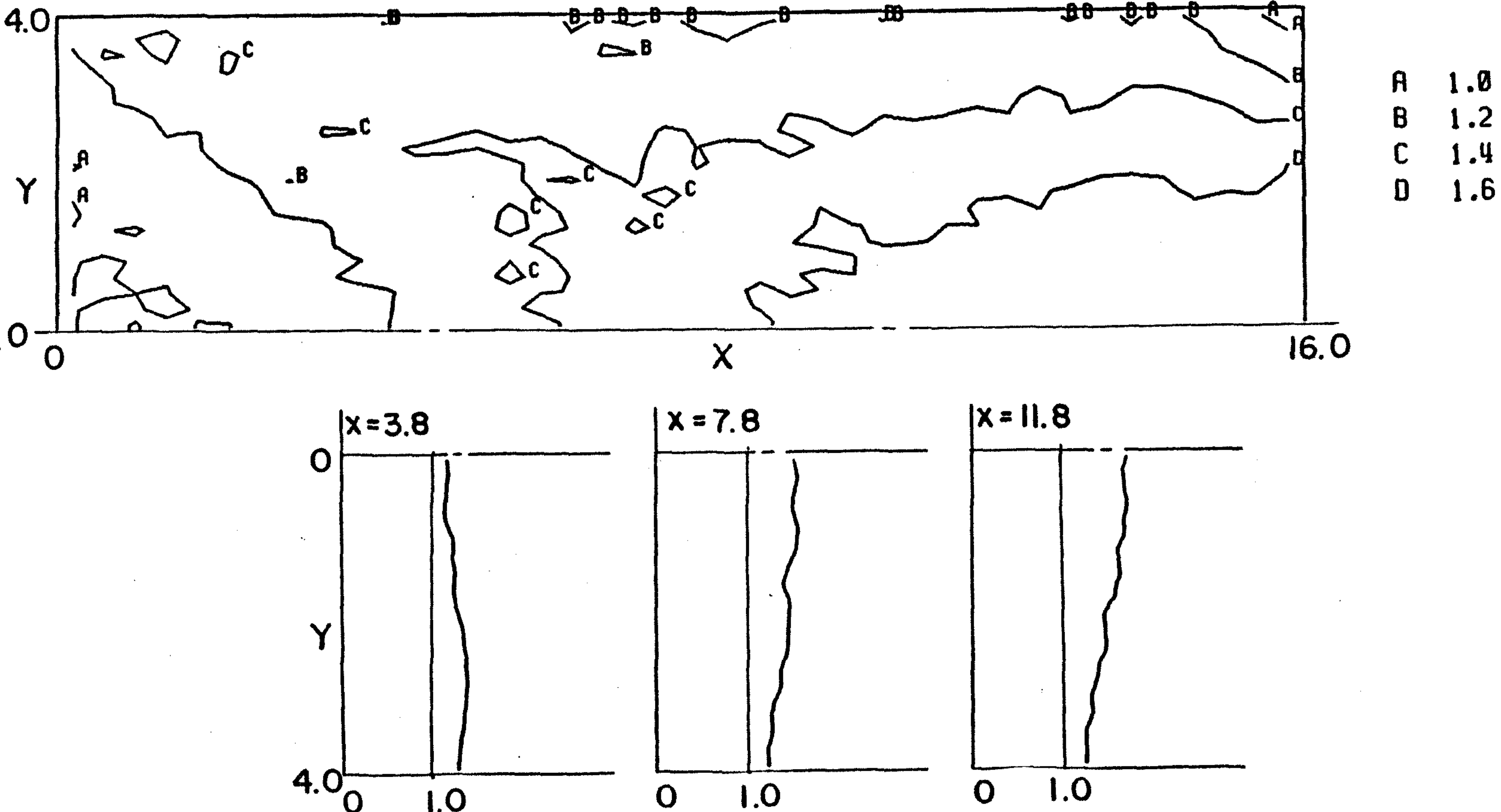


Figure 5.4(a) Number density profile of channel flow of intermediate material in the steady state with Re = 32.6, M = 2.76 and K_n = 0.125 (ϵ_p = 1.0, ϵ_s = 0.0, diffusely reflecting walls, T_w = 1)

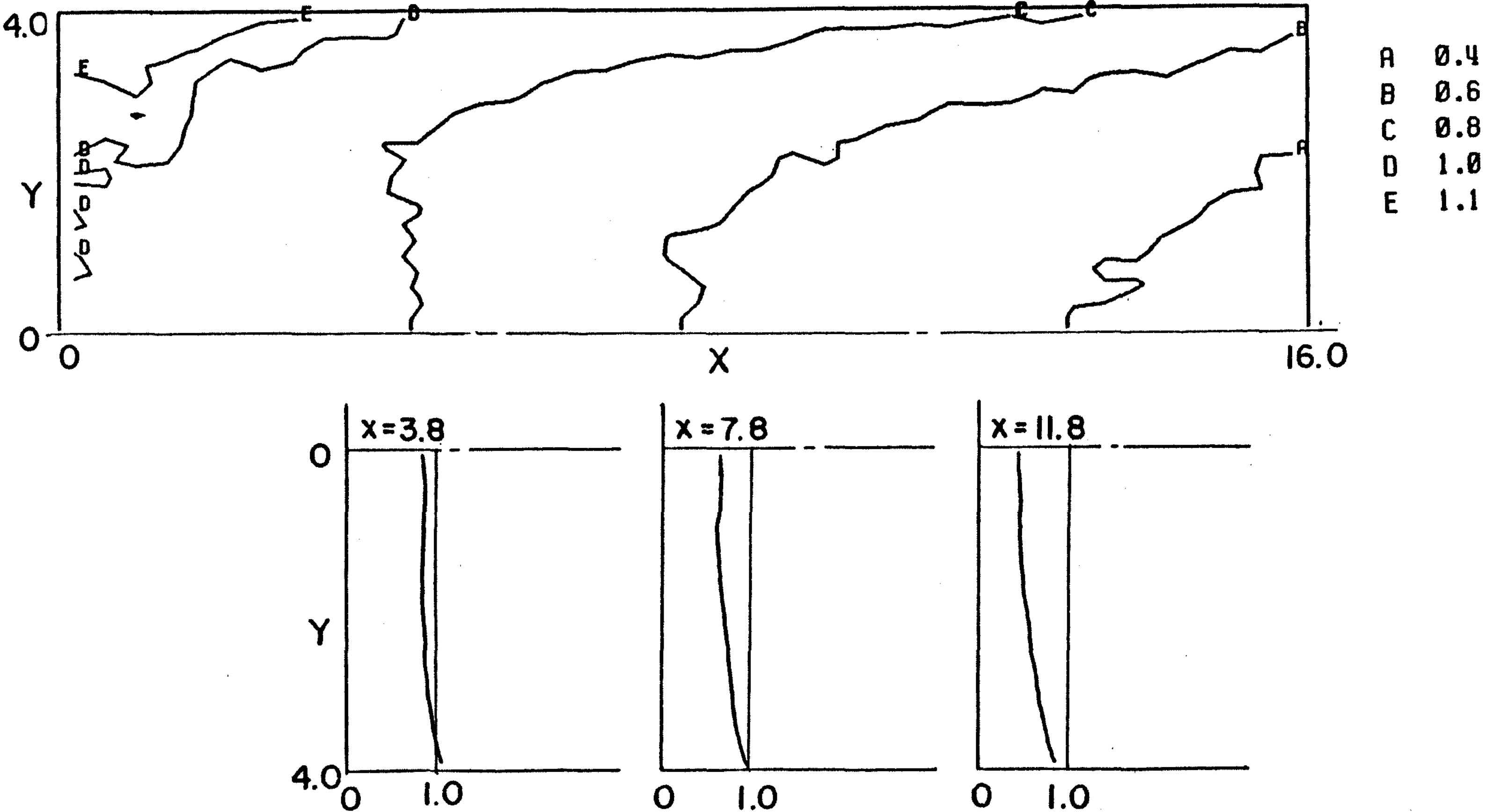


Figure 5.4(b) Tempertature profile of channel flow of intermediate material in the steady state with Re = 32.6, M = 2.76 and K_n = 0.125 (ϵ_p = 1.0, ϵ_s = 0.0, diffusely reflecting walls, T_w = 1)

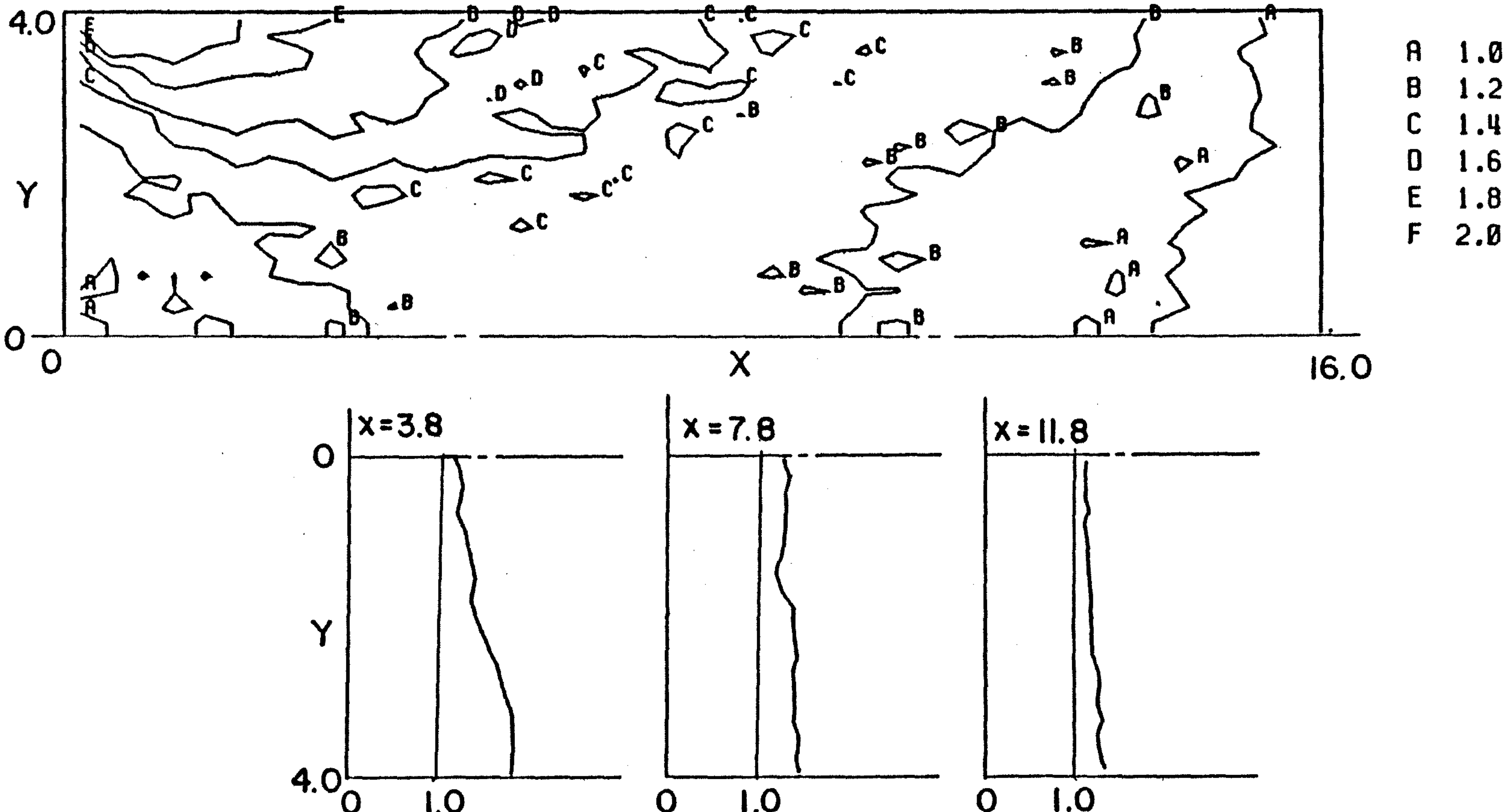


Figure 5.4(c) Pressure profile of channel flow of intermediate material in the steady state with Re = 32.6, M = 2.76 and K_n = 0.125 (ϵ_p = 1.0, ϵ_s = 0.0, diffusely reflecting walls, T_w = 1)

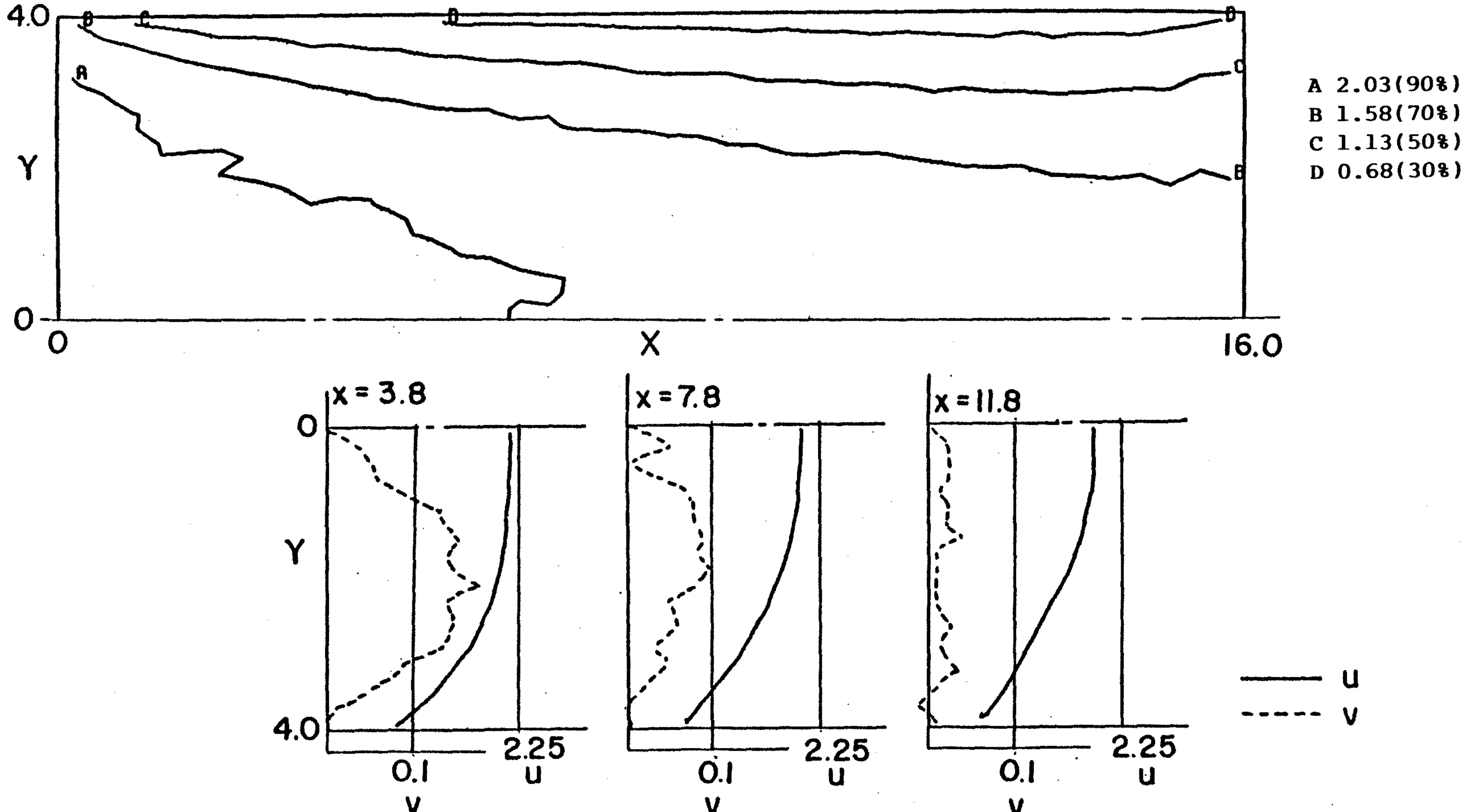


Figure 5.4(d) Velocity field of channel flow of intermediate material in the steady state with Re = 32.6, M = 2.76 and K_n = 0.125 (ϵ_p = 1.0, ϵ_s = 0.0, diffusely reflecting walls, T_w = 1)

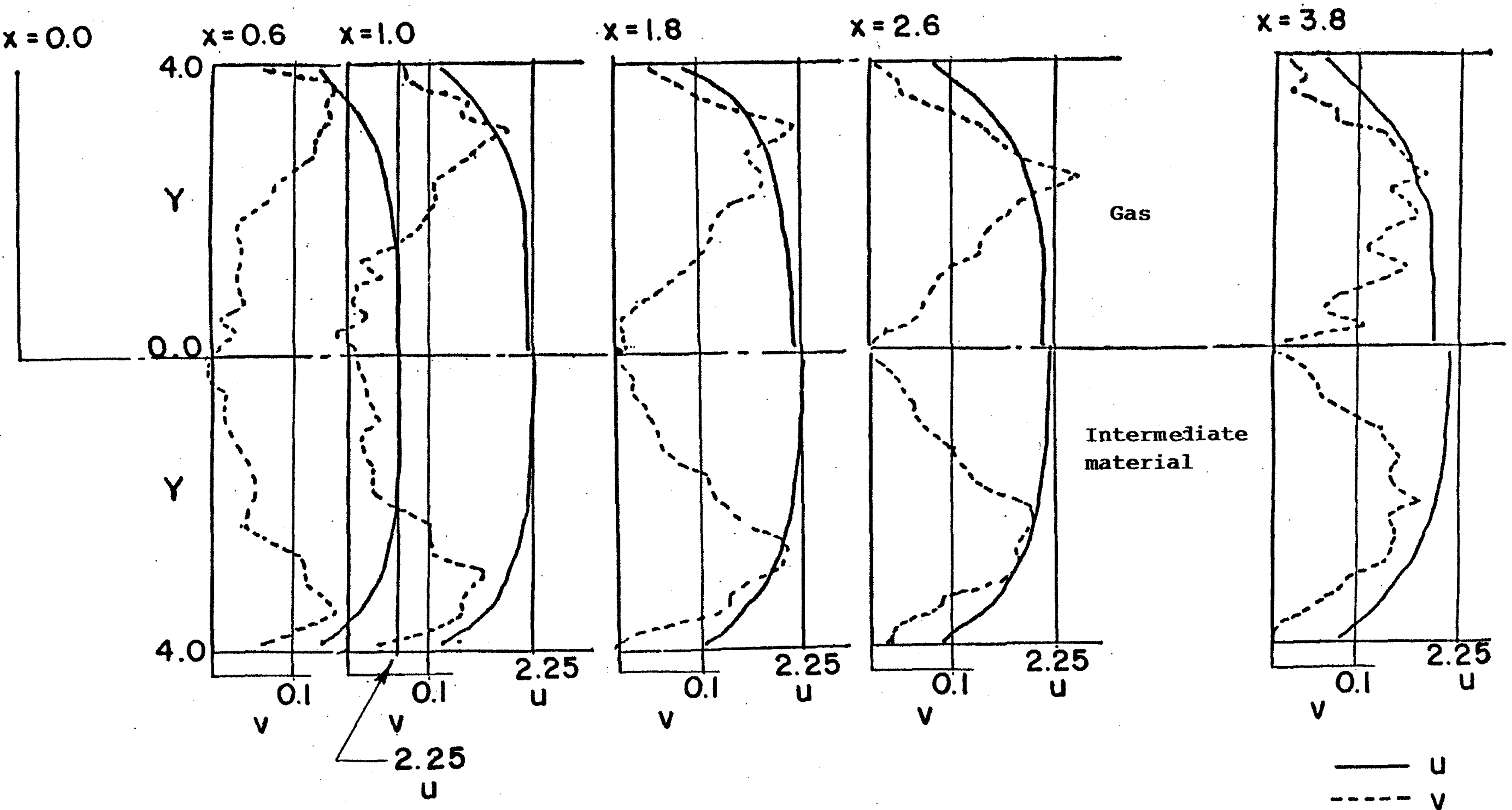


Figure 5.5 Comparison of velocity field between intermediate material and gas near the leading edge of diffusely reflecting wall with Re = 32.6, M = 2.76 and K_n = 0.125. Gas (ϵ_p = 1.0, ϵ_s = -1.0). Intermediate material (ϵ_p = 1.0, ϵ_s = 0.0), T_w = 1.

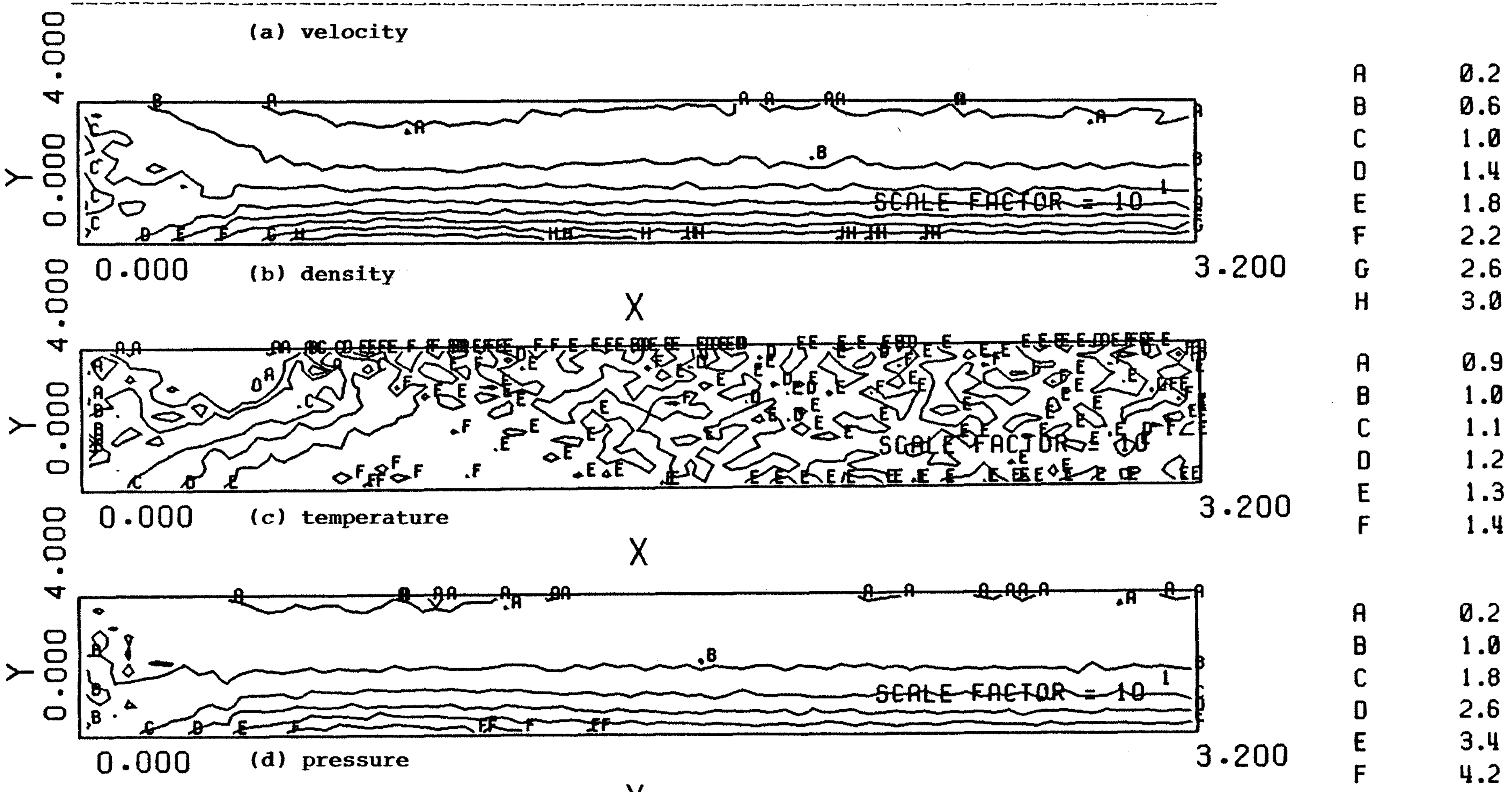


Figure 5.6 Sedimentation of gas in channel with $Re = 16.3$, $M = 2.76$ and $K_n = 0.25$, $\epsilon_p = 1.0$, $\epsilon_s = -1.0$ specular wall, $g = 0.5$