ABSTRACT

OBJECT ORIENTED 3D DYNAMIC SIMULATION OF FLOW OF GRANULAR MATERIAL THROUGH HOPPER

Development of object oriented 3D dynamic simulation of the particle is considered. This code is based on the soft sphere model dealing with both normal force , tangential force and the friction. The development of such a model and the code is motivated by the need to understand the flow patterns of granular material. The prime effort while developing this code was focused on the data encapsulation and secured access to all the attributes of all physical parameters of the bulk solid, which might be endangered by the random access of the interface of the software. A dynamically bound array based container has been implemented though needs some more work to be done to make it more robust. Data sorting and searching is also focused. Reduction of computation overhead and increase of speed were tried to resolve through established algorithms on searching and sorting. This is of crucial importance in dealing with problems associated with bulk solids flows, which occur in almost all industries and natural geological events.

OBJECT ORIENTED 3D DYNAMIC SIMULATION OF FLOW OF MATERIAL THROUGH HOPPER

by
Krishnendu Roy

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Mechanical Engineering

Department of Mechanical Engineering

August 1998

APPROVAL PAGE

OBJECT ORIENTED 3D DYNAMIC SIMULATION OF THE FLOW OF GRANULAR MATERIAL THROUGH HOPPER

Krishnendu Roy

Dr. Rajesh N. Dave, Thesis Advisor — Date
Associate Professor of Mechanical Engineering, NJIT

Dr. Pushpendra Singh, Committee Member — Date
Associate Professor of Mechanical Engineering, NJIT

Dr. Rong Y Chen, Committee Member — Date
Graduate Advisor and Professor of Mechanical Engineering, NJIT

BIOGRAPHICAL SKETCH

Author:	Krishnendu Roy
Degree:	Master of Science in Mechanical Engineering
Date:	August 1998

Undergraduate and Graduate Education

- Master of Science in Mechanical Engineering,
  New Jersey Institute of Technology
  Newark, New Jersey, 1998

- Bachelor of Technology in Mining Engineering,
  Indian Institute Of Technology,
  Kharagpur, India, 1992

Major: Mechanical Engineering

This thesis is dedicated to
my family
and my friends

ACKNOWLEDGEMENT

I take this opportunity to express my deep gratitude to Dr. Rajesh Dave, Associate Professor, Mechanical Engineering Department of N.J.I.T. for his valuable guidance and help throughout the course of this work. It was an delightful and enjoyable experience working under his guidance.

I am deeply indebted to Dr. Pushpendra Singh , Assistant Professor, Mechanical Engineering Department of N.J.I.T. for his enormous effort to keep my track throughout the developement of the code and the thesis.

I am very much thankful to Dr. R. Chen, Graduate Advisor and Professor for his constructive advice and guaidance throught my academic pursuit.

I am unhesitatingly thankful to my colleagues in Particle Tchnology Laboratory - Bodhisattwa Choudhuri, Felix Alcocer, Loic Vanel, William C. Dunphy for their friendship and cooperation.

I also unhesitatingly acknowledge Dr. C. Y. Wu for his advice.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 History and Literature Survey

For the development of powder flow technology, it is not enough just to measure the bulk flow properties by uniaxial shear tester or isostatic shear tester or measuring the steady state flow properties by Jenike shear tester(1961, 1964). In many cases,measurement of these properties is good enough for certain process. Often, however, circumstances, such as the surrounding temperature, the humidity, the composition of the raw material, the process conditions and so on, will change with time. This implies that the measurement of the powder flow properties is just a snapshot of a large variety of possibilities.Therefore, measurement of the powder flow properties is limited to certain conditions.

In essence, the behaviour of the powder is determined by the interparticleforces and by their response in time to changing conditions.The future of powder flow technology is dependent on knowledge of this level of expertise. Many theories have been developed in the past, of which the theory of Johnson, Kendall and Roberts (JKR, 1971) is an interesting representative for particle adhesion forces. This theory deals with the case of strong adhesion and particles with low elastic moduli. Another model, with opposite starting points, was developed by Derjanguin, Muller and Toporov (DMT, 1975). Of course, these models are still very limited, and do not include factors such as particle shape, roughness and application of forces. To meet these restrictions, theories including the effect of tangential forces, surface roughness and electrostatic forces have been suggested. Often, this research is limited to two particles in contact. Nowadays, computer is able to simulate thousands of particles in contact and their behaviour with respect to each other based on these and other theories. The main motivation for writing this code is to meet those restrictions with thousand of particles.

Simulation on the microscopic level will be helpful in the future understand certain macroscopic powder flow behaviour. It also will predict behaviour regarding certain conditions and therefore it will provide a tool to manipulate the powder behaviour in a fundamental way, rather than trial and error. Several models have been developed over years, of which Leonard & Jones contact force model for normal forces and Mindlin and Deresiewicz (1953) and a new theory by Thronton (1991) for the tangential forces between particles are utilized. With this model as a guide, the interparticle behaviour was investigated. The purpose of this thesis is to develop a simulation code to find the particle trajectories, velocities and consequently compute the macroscopic quantities which can be great help in understanding the behaviour of flowing granular materials through hopper. Moreover optimization of computer overhead by introducing object oriented concept, and lattice structure for collision detection have been implemented so that the code can handle a significant number of particles.

The force model used in this study is based on the "soft sphere" model introduced by Walton et al. (), who followed nonequilibrium molecular dynamics method proposed by Ashurst and Hoover (). In this force model, the collisions between particles do not occur instantaneously but occur for a finite time duration. Therefore multiple contacts between particles are possible. In order to determine an accurate trajectory change for a single collision, many calculational steps are required.

Numerical Method

The numerical method utilizes a straight forward extension to 3D of the 2D contact force models and integration equations of Walton and Braun (1986), (WB model). Explicit intrigration is accomplished via the "leap-frog" method for each of 6 degrees of freedom. Since the particles are spherical, only the magnitude and direction of the angular velocity is needed to determine the infinitesimal surface displacements between timesteps(i.e., the information necessary to determine changes in tangential friction forces). For the motion of one sphere in the x-direction and rotation about its x-axis the finite difference equations are:

velocity $v_x^{n+1/2} = v_x^{n-1/2} + (F_x^n / m + g_x) \Delta t$. (1.1)

postiton $x^{n+1} = x^n + v_x^{n+1/2} \Delta t$. (1.2)

angular velocity $d\omega_x^{n+1/2}/dt = d\omega_x^{n-1/2}/dt + (N_x / I_0) \Delta t$. (1.3)

Where the subscript n, refers to the current time step; F_x is the x-component of the sum of the all contact forces acting on the particle; g_x is the x-component of the applied body force(i.e., gravity); m is the mass; N is the torque; I_0 is the moment of inertia, and Δt is the time step. Similarly other two directions can be formulated. The orientation of the individual particle (e.g., Euler angles) are not needed to determine the forces or the subsequent motion.

Normal force:

The normal force during collision, F_N is identical to WB model:

for loading $F_N = K_1\alpha$. (1.4)

for unloading $F_N = K_2(\alpha - \alpha_0)$. (1.5)

Where α is the "overlap" between the contacting spheres; $K_2 > K_1$; and α_0 represents the relative "overlap" where the unloading force is set to zero (due to inelastic deformation of the surfaces). This normal-force model produces binary collision with a constant coefficient of restirution given by $e = \sqrt{K_1/K_2}$, independent of the relativevelocity of impact.

Tangential force:

The tangential friction force is a two dimensional (surface) extension to Walton and Braun's one-dimensional approximation to Mindlin's (1949) elastic frictional sphere contact force model. In that WB model the effective tangential stiffness of a contact decreases with tangential displacement until it is zero when full sliding occurs. In the present two-dimensional surface model the tangential displacement *parallel* to the *perpendicular* to the existing friction force are considered separately. They are later combined vectorially and their sum is checked against the total friction force limit, μF_N.

The effective tangential stiffness in the direction *parallel* to the existing friction force is given by:

for T increasing $\qquad K_T = K_0 (1 - (T - T^*)/(\mu F_N - T^*))^\gamma$. $\qquad$ (1.6)

for T decreasing $\qquad K_T = K_0 (1 - (T^* - T)/ (\mu F_N + T^*)^\gamma$. $\qquad$ (1.7)

Where K_0 is the initial tangential stiffness; T is the current tangential force magnitude; T^* starts as zero and is subsequently set to the value of the total tangential force, T, whenever the magnitude changes from increasing to decreasing, or vice versa; γ is a fixed parameter often set to 1/3 to make the model resemblee Mindlin's elastic frictional sphere theory, and μ is the coefficient of friction. A value of 1 or 2 for γ more closely immitates the behaviour of frictional contacts involving plastic deformation in the contact region (Drake and Walton, 1992).

1.2 Overview of the Chapters

The chapter 1 introduces the subject and deals with literature survey and history of simulation of flow of granular material. It also gives the overview of all other chapters.

The chapter 2 deals with the brief description of 3D dynamic simulation of the particles, the force models that has been taken into consideration, the normal force model and the tangential force model used.

The chapter 3 covers the code. It gives the datastructure, used in the code, the objects, the member functions. It also covers the declaration of the different variables and the algorithm used in the code .

The chapter 4 covers the result obtained during filling and emptying the hopper.

The chapter 5 deals with the summary and future works suggested.

CHAPTER 2

3D DYNAMIC SIMULATION OF PARTICLE

2.1 History and Introduction

The particle dynamic simulation has come a long way trailing lots of ideas and its implication in history. In early 1950's the traditional molecular dynamic simulationmethod was developed to study the flow of viscous fluid. Later in order to study the granular particles, particle dynamic simulation method was derived from the molecular dynamics. The major short-coming for molecular dynamics method was, particles are considered to be perfectly elastic. That means total energy of the system is conserved. No kinetic energy is dissipated during collision. The particle dynamic simulation method, tried to solve this drawback by taking into consideration that, a fraction of the initial kinetic energy is lost as heat, elastic or plastic strain energy, acoustic energy or even light. Apart from that, molecular dynamic simulation, theoretically, is supposed to be the core model.

Particle dynamics simulation method uses two types of interactive force models.The "Hard Sphere Model" assumes that particles are rigid bodies with infinite stiffness, ieparticles do not overlap each other during collision and collisions are instanteneous. This model proceeds with irregular time jumps corresponding to the time period between one collision and the next. The velocities of the particles are determined by taking into account the normal and tangential coefficients of restitution, a frictional coefficient, and the velocities of the particles before that time step.

The "Soft Sphere Model", deals with finite time of collision.The particles are allowed to overlap by an amount depending on their stiffness. The post-collision velocity depends on number of parameters, which do not bear the the explicit relationship with the material properties used in hardsphere.

The simulation code developed here, was inspired by molecular dynamics method and contact force model implemented was developed Leonard & Jones. To handle the short-coming of molecular dynamic method, the "Soft Sphere Model" is implemented.

2.2 The Force Models

2.2.1 Normal Force Model

In the "Soft Sphere Model" developed by Walton [], a "Partially Latching-spring Model" is used to estimate the interactive force in collisions, and to approximate the energy loss due to inelastic collision in "normal" direction, i.e., the direction defined by the line joining the center of two ineracting spheres. In this force model, collision process is divided into two regimes. The "Compression" regime and "Restoration" regime. Here, the overlaps between particles are measured as the particle deformations. The "Compression" regime starts at the begining of a collision. In this regime, the relative velocity between two particles decreases as the particles approach each other. When the relative velocity becomes zero and the overlap reaches its maximum value in the collision $\propto_{max}$, the "Compression" regime ends and the "Restoration" periods starts.

$$F_1 = K_1 . \alpha$$

$$F_2 = K_2 . (\alpha - \alpha_0)$$

where F_1 is the force in "Compression" regime, F_2 is the force in the "Restoration" period, K_1 and K_2 are the normal stiffness coeffiecients in the two periods, respectively, α is the overlap between the particles, and α_0 is the residual overlap. α_0 is initialized to zero automatically when the restoring force F_2 goes to zero, because permanent deformations of particles are not allowed. Figure 2.1 (a) (b) demonstrate the relationship between forces and the deformations during collisions.

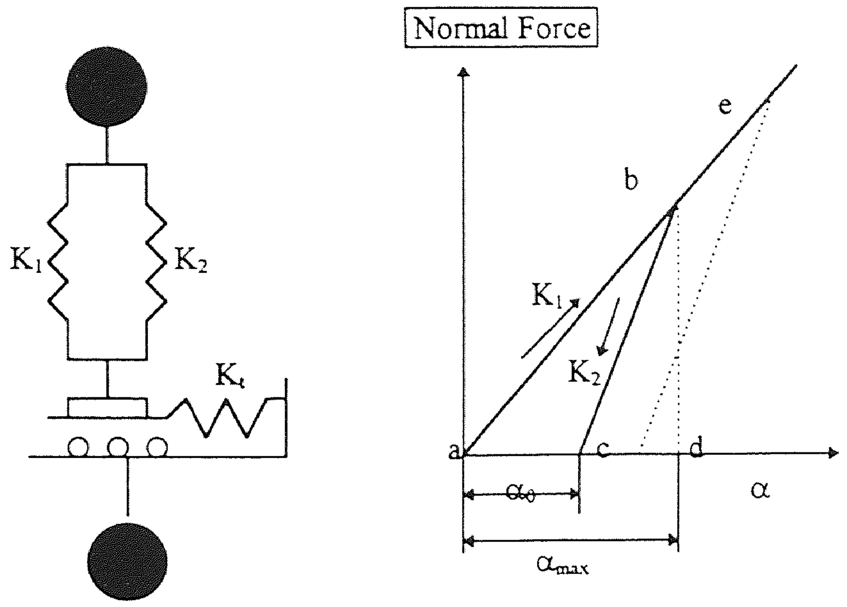


Figure 2.1 (a) The partially Latching-spring Model
(b) The Normal force to Overlap diagram

K_1 is represented by the slope of the line $\overrightarrow{ab}$, K_2 is equal to the slope of the line $\overrightarrow{bc}$. Since the slope bc is greater than the slope ac, the value of K_2 is greater than the value of K_1. The energy loss during this collision is equal to the area of Δabd minus the area Δabc. To determine the interactive force, F_1 and F_2 are both calculated for each time step during the collision, and the smaller one is chosen as normal interactive force.

(a) In the "Compression" period from $\alpha_0 = 0$: $a \longrightarrow b$.

Since K_2 is larger than K_1 and α_0 is equal to zero, then

$F_1 = K_1.\alpha$

$F_2 = K_2.(\alpha - 0) = K_2.\alpha$

It is clear that F_2 is larger than F_1, so the normal interactive force will be equal to F_1.

(b) In the "Restoration" period : $b \longrightarrow c$

In this period, F_2 is always smaller than F_1 as it is shown in the Figure 2.1. F_2 will be chosen as the normal interactive force.

(c) In the "Restoration" period from $\alpha_0 \neq 0$:

If the two particles are recompressed during the "Restoration" period, that is $b \longrightarrow c$, the loading path will go backwards $c \longrightarrow b$, and F_2 will be chosen as the normal force since F_2 is still less than F_1. After reaching point b, F_1 will be chosen as the normal force again because it is now smaller than F_2, and the loading path goes $b \longrightarrow e$.

2.2.2 Tangential Force Model

The tangential interactive force model developed by Walton and Braun [] and used in the code, was derived from Mindlin and Deresiewicz []. In this model, the tangential force at time t is related to the tangential stiffness, the surface displacements and the tangential force at the last time step t-dt. The tangential stiffness decreases with the surface displacement, and when it becomes zero, full sliding takes place. The force model is described in the following page by the Figure 2.2. The Tangential plane at time t will be shifted to different orientation for time step t + dt and that for t-dt will be in different orientation.

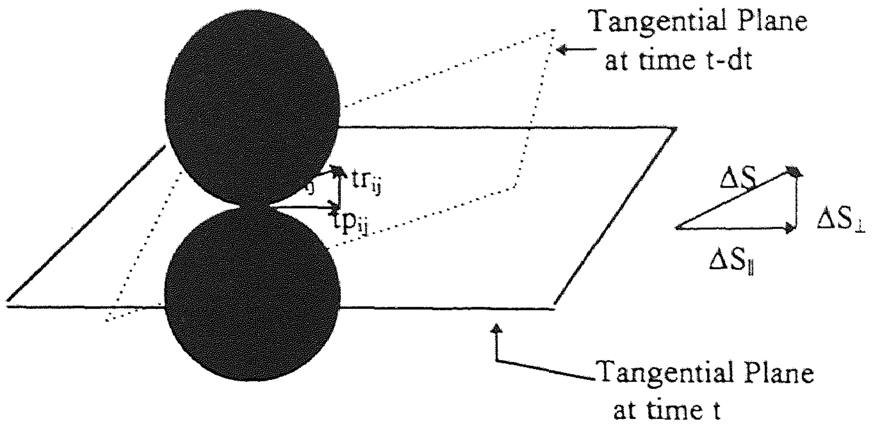


Figure 2.2 The Tangential Force Model

Figure 2.2 demonstrates the collision and the tangential plane between two particles. Thetangential plane is always perpendicular to the line which connects the centers of the two particles, and it will be changed if the particles move to new positions. The tangential force at the current time step ft_{ij} is calculated from the tangential force at the last time step t-dt. In Figure 2.2, tf_{ij} is the tangential force at the last time step, and tp_{ij} is its projection onto the current tangential plane. ΔS is the total tangential displacement at the time t-dt,(ΔS_n, ΔS_t) are its components in the plane's normal and parallel directions. The force ft_{ij} is then calculated as follows:

T_{tot} is the total tangential force acting on a particle. It is obtained by adding two components, normal and the parallel.

$$T_{tot} = T_{\parallel} + T_{\perp} \tag{2.1}$$

where $T_{\parallel}$ is the tangential force parallel to the displacement and $T_{\perp}$ is the tangential forceperpendicular to the displacement.The contribution along the perpendicular direction is given by:

$$T_{\perp} = K_o \,.\, \Delta S_{\perp} \tag{2.2}$$

where K_o is a constant (initial tangential stiffness) and $\Delta S_{\perp}$ is the displacement (from the previous time step) component perpendicular to the tangential force (from the previous time step).The contribution along the parallel direction is given by:

$$T_{\parallel} = T_{proj} + K_T \,.\, \Delta S_{\parallel} \tag{2.3}$$

where T_{proj} is the tangential force from the previous time step, projected onto the current slip plane such that $|T_{proj}|$ is the same as in the previous time step, i.e.

$$\vec{T}_{proj} = \frac{[\, \vec{T}_{old} - \vec{k}_{ij} \,(\, \vec{k}_{ij} \,.\, \vec{T}_{old} \,)\,]}{|\, \vec{T}_{old} - \vec{k}_{ij} \,(\, \vec{k}_{ij} \,.\, \vec{T}_{old} \,)\,|} \,.\, |\, \vec{T}_{old} \,| \tag{2.4}$$

where $\vec{k}_{ij} = (\, \vec{r}_i - \vec{r}_j \,) \,/\, |\, \vec{r}_i - \vec{r}_j \,|$

$\Delta S_{\parallel}$ is the displacement parallel to the tangential force (previous time step) for $|T_{proj}|$ increasing :

$$K_T = \left(1 - (\, |\vec{T}_{proj}| - T^* \,) \,/\, (\, \mu F_N - T^* \,) \right) \tag{2.5}$$

for $|T_{proj}|$ decreasing :

$$K_T = \left(1 - (\, T^* - |\vec{T}_{proj}| \,) \,/\, (\mu F_N + T^*) \right) \tag{2.6}$$

where $\quad T^* = 0 \quad$ in the begining $\quad$ (2.7)

$$= |\vec{T}_{tot}|\ |F_n|^{t-dt} / |F_n|^{t} \quad \text{otherwise} \tag{2.8}$$

$$\vec{\Delta S}^{t-dt} = \vec{\Delta r}_{ij} - \vec{k}_{ij} . (\vec{k}_{ij} . \vec{\Delta r}_{ij}) + [\vec{r}_i (\vec{\omega}_i^{t-dt} \times \vec{k}_{ij}) + \vec{r}_j (\vec{\omega}_j^{t-dt} \times \vec{k}_{ij})]\, \Delta t \tag{2.9}$$

where $\vec{\Delta r}_{ij} = \vec{r}_{ij}^{\,t} - \overrightarrow{r_{ij}}^{t-dt}$

$$\vec{r}_{ij}^{\,t} = \vec{r}_j^{\,t} - \vec{r}_i^{\,t-dt} . \tag{2.10}$$

if $\vec{t} = \vec{T}_{proj} / |\overrightarrow{T_{proj}}|$.

$$\vec{\Delta S}_{\parallel} = [(\overrightarrow{\Delta S^{t-dt}}) . \vec{t}]\, \vec{t} \tag{2.11}$$

$$\vec{\Delta S}_{\perp} = \overrightarrow{\Delta S}^{t-dt} - \overrightarrow{\Delta S}_{\parallel} \tag{2.12}$$

The angular velocity of the i th particle ω_i at the present time step can be calculated from the equation given below:

$$I\ d\vec{\omega}_i/d\vec{t} = \overrightarrow{T_{tot\,i}}$$

$$\Rightarrow I\, (\vec{\omega}_i^{\,t} - \vec{\omega}_i^{\,t-dt}) / \vec{\Delta t} = \vec{T}_{tot\ i} \tag{2.13}$$

$$\Rightarrow \vec{\omega}_i^{\,t} = \overrightarrow{\omega_i}^{t-dt} + \overrightarrow{T_{tot\,i}}\, \Delta t / I. \tag{2.14}$$

Periodic Boundary Condition and Cell-Configuration :

The cell configuration is shown in the figure 2.3

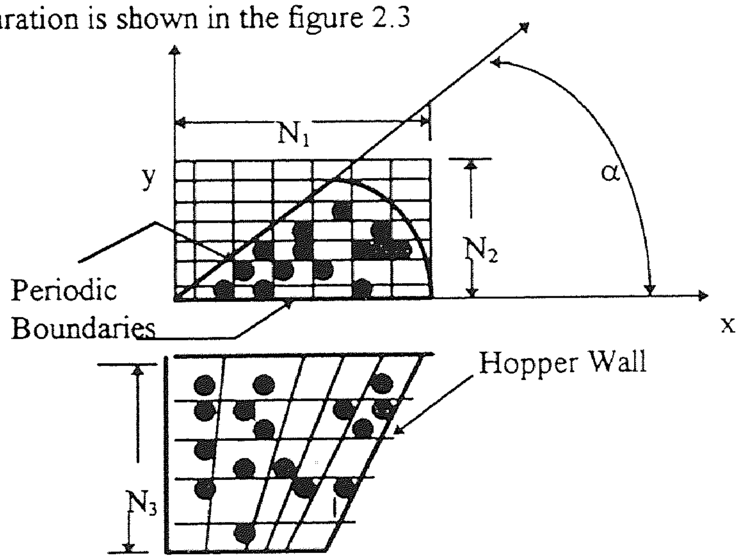


Figure 2.3 The cell distribution in x,y and z directions.

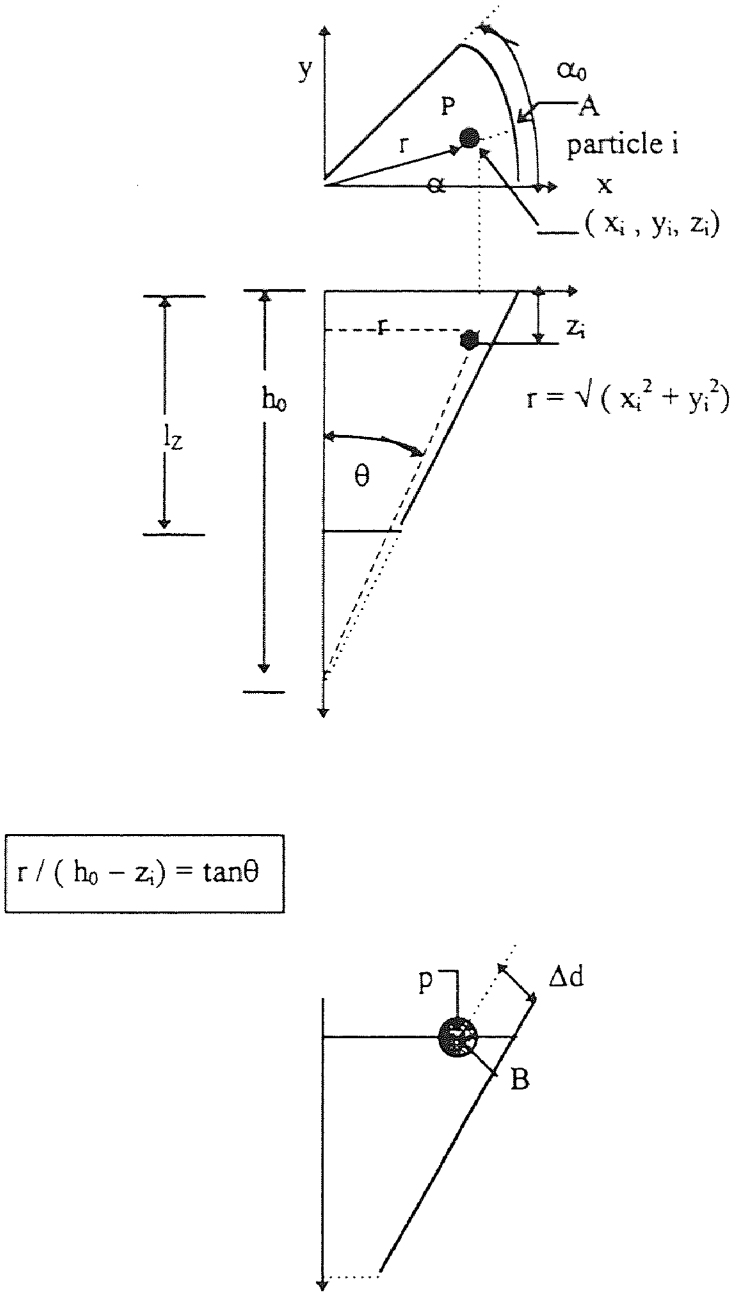


Figure 2.4 The geometry of the hopper and the particle position.

$$PB = \Delta d = (h_0 - z_i)\tan\theta_0 \tag{2.15}$$

The unit vector normal to the inclined slope of the hopper is given by:

$$\vec{r}_{BP} = (x_B - x_P)\vec{i} + (y_B - y_P)\vec{j} + (z_B - z_P)\vec{k} \qquad (2.16)$$

$$\Rightarrow \vec{r}_{BP} = ((PB \cos\theta_0) \cos\alpha)\,\hat{i} + ((PB \cos\theta_0) \sin\alpha)\,\hat{j} - (PB \sin\theta_0)\,\hat{k} \qquad (2.17)$$

$$\Rightarrow |\vec{r}_{BP}| = \sqrt{[(PB \cos\theta_0) \cos\alpha)^2 + (PB \cos\theta_0) \sin\alpha)^2 + (PB \sin\theta_0)^2]} \qquad (2.18)$$

$$\hat{r}_{BP} = \vec{r}_{BP} / |\vec{r}_{BP}| \qquad (2.19)$$

$$v_\perp = \vec{v} \cdot \hat{r}_{BP} \qquad (2.20)$$

$$\vec{v}_{\text{after collisoin}} = \vec{v}_{\text{before collision}} - 2\, v_\perp \hat{r}_{BP} \qquad (2.21)$$

2.3 Deciding on Scaling of Cell Distribution

For a particle having position given by (x_P , y_P , z_P) , the scaling of x , y , z coordinates are done with respect to the local radius R' (the radius of the hopper at z-position z_P).If x_S , y_S , z_S be the scaled coordinate for that particle, then it can be represented by the following relationships:

$$x_S = x_P / R'$$

$$y_S = y_P / R'$$

and $z_S = z_P$ since no scaling takes place in the direction of z-axis.

The cell distribution will take place depending on this scaled coordinates.Based on (x_S , y_S , z_S) decide the cell :The index of cell in the x-axis direction alotted for the particle is given by:

$$n_X = x_S / \Delta x \qquad (2.22)$$

The index of cell in the y-axis direction alotted for the particle is given by:

$$n_Y = y_S / \Delta y \tag{2.23}$$

The index of cell in the z-axis direction alotted for the particle is given by:

$$n_Z = z_S / \Delta z \tag{2.24}$$

Where Δx, Δy, Δz are given by :

$$\Delta x = 1 / N_X \tag{2.25}$$

$$\Delta y = 1 / N_Y \tag{2.26}$$

$$\Delta z = l_Z / N_Z \tag{2.27}$$

N_X, N_Y, N_Z are the number of cells in the x-axis, y-axis, z-axis direction and l_Z is the height of the hopper.

CHAPTER 3

PROGRAMMING, DATASTRUCTURE AND THE CODE

3.1 Datastructure

3.1.1 The Short-Range Force

The total short-range part of the force on a particle i at position x_i is given by the sum of the interparticle short-range forces

$$F^{ST}{}_i = \sum_{j=1}^{Np} f^{ST}{}_{ij} \tag{3.1}$$

The elementary method of evaluating $F^{ST}{}_i$ is to sweep through all particles $j = 1,......N_p$, test whether the separation $r_{ij} = |x_i - x_j|$ is less than r_e, and, if so, compute $f^{ST}{}_{ij}$ and add it to $F^{ST}{}_{ij}$. Such an approach is clearly impractical, since for each of the N_p values of i one would have to test $N_p - 1$ separations r_{ij} giving an operations count scaling as $N^2{}_p$.

3.1.2 The Chaining Mesh

The computational cost of locating those particles j which contribute to the short-range force on particle i is greately reduced if the particle coordinates are ordered such that the tests for locating particles j such that $r_{ij} \leq r_e$ need only be performed over a small subset N_p. It is for this reason that the chaining mesh is introduced. The chaining mesh (in three dimensions) is a regular lattice of ($M_1 \times M_2 \times M_3$) cells, covering the computational box (of side $L_1 \times L_2 \times L_3$) in much the same manner as the ($N_1 \times N_2 \times N_3$) cells of the much finer charge-potential mesh. The number of cells M_S along the s direction is given by the largest integer less than or equal to L_S/r_c. Consequently, the lengths of the sides of the cells of the chaining mesh are always greater than or equal to the cutoff radius r_c.

The figure 1.1 depicts a chaining in two dimensions. Typically, the side lengths of the chaining meshcells HC_S are between three and four times greater than the side lengths H_S of the cells of the charge-potential mesh. The circle of radius r_c centered on particle I in chaining cell q delineates the area in which particles j must lie if they are to have a nonzero contributions to F^{ST}_i must either lie in the same cell q as particle i or in one of the eight neighboring cells. If the particle coordinates are sorted into lists for each chaining cell, then to find the force F^{ST}_i on particle i involves approximately $9N_C$ tests, where N_C ($= N_p/M_1M_2$) is the average number of the particles per chaining cell. Therefore,if Newton's third law is used, the total number of tests in finding all the short-range forces is approximately $N_nN_p \cong 4.5N_CN_p$ as compared with N^2_p for the elementary approach. Similarly, in three dimensions, sorting coordinates into chaining cells gives the number of tests $N_nN_p \sim 13N_CN_p$. The following Figure 3.1 depicts a two dimensional chaining mesh configuration with HC_1 and HC_2 cell size in x and y direction for the cell q.

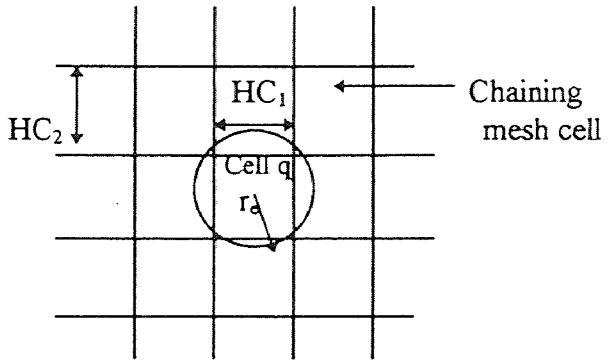


Figure 3.1 Cell configuration of the chaining

3.1.3 The Linked Lists

For serial computers(but not necessarily for vector or array processor machines) it is computationally more efficient to sort the coordinate addresses rather than the coordinates themselves. Address sorting is made possible by introducing the linked-list array LL.

For chaining cell q if, let, HOC[q] be the head-of-chain table entry for chaining cell q, and let LL[i] be the link coordinate for particle i, then the procedure for sorting coordinates into lists for each chaining cells by means of address sorting is summerized as follows:

1. set HOC[q] = -1 for all q.

2. do for all particles i.
   (a) locate cell containing particle

   $$q: = \text{int}(x_1/HC_1,\ x_2/HC_2,\ x_3/HC_3) \tag{3.2}$$

   (b) add particle i to head of list for cell q

   $$LL[i] := HOC[q] \tag{3.3}$$

   $$HOC[q] := i \tag{3.4}$$

In two-dimension third components of q and x are omitted.

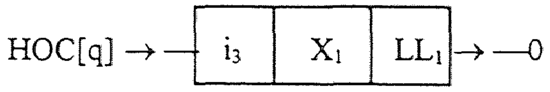
The way the sorting procedure works is illuminated by considering an example. Let's consider the case where three particles I_1, I_2, I_3 lie in the chaining cell q,where $I_1 < I_2 < I_3$. We represent the coordinates by a three-partition box..

i	X_i	LL_i

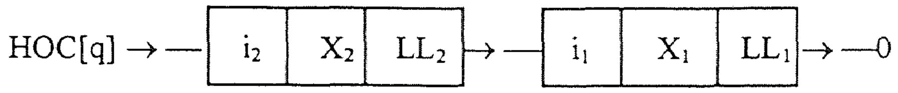
Where i is the address (or array element in FORTRAN), X_i are the physical particle coordinates (x_i, P_i), and LL_i is the linked-list coordinate. If particle coordinates are swept through in increasing i values then the linked list for cell q develops follows:Initially (after step 1)

$HOC[q] \rightarrow —0$

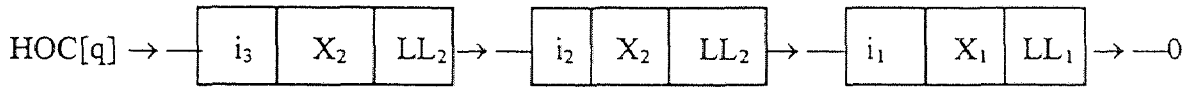
$i_1 < i < i_2$



$i_2 < i < i_3$



$i_3 < i$



The speed and simplicity of creating linked lists from scratch make it pointless saving andupdating them timestep by timestep. The whole sorting process requires only three real arithmatic operation per particle in three dimensions or two in two dimensions.

Once The HOC and LL tables have been filled, a zero entry in HOC[q] indicates that there are no particles in chaining cell q. A nonzero entry gives the address of the coordinates of the next particle in the list, or is zero to indicate the end of the list. Therefore, given HOC and LL, coordinates in each cell can be looked up without any searching. This kind of sorting and reducing the search loops will eventually economise the computation overhead.

3.2 Class and Members

Taking in to consideration that object-oriented programming with c++ is totally structured approach towards programming, the total code has been divided into one Class, called Particle, which in turns contains number of member functions, and a series of datafield. The datafield represents all possible variables required for the code. All the datafield members are transparently accessible by the member functions. The member functions, independently contains several other datafields which are alive only during the operation of the function and are inaccessible by other member functions.

The code contains several datafields for position,velocity and force components acting on a specific particle, which have been defined as double dimension arrays taking into consideration that one dimension of it will represent the co-ordinate axes i.e. x[0][k],v[0][k], F[0][k] will represent the position along x-axis, velocity component along x-axis and the force component along x-axis of particle k.

The member function **init(...)** will initializes the initial position, velocity, force acting on the particles. It also initializes the radius of the particle, mass of the particle, angular velocity in different axes, initial tangential forces, and some datafields to contain the old values, for the comparison of the data.

The member function **march(...)** will start working right after initialization, for a given number of time steps. It will update the position, velocity and forces acting on the particle and checks whether any collision with the wall takes place or not. If the particle collides with the wall, it will keep the particle back to the hopper region applying collision mechanics. And then print out the position data into an output file. Finally it will invoke the member function **search(...)**, for each time steps.

The member function **search(...)**, after getting data from **march(...)** for each particle, will allocate cell to each particle and put them into a link-list, from where the indices of the particle, which are stored in a specific cell, can be found out. Then it will start searching for each particle, 27 neighbouring cells, including that cell. It calculates the forces, both normal and tangential, velocities and angular velocities for particles in contact If, for a cell the loop comes accross more than one particle it will call the member function **listing(...)**.

The member function **listing(...)** is a recursive function which will find out, for a definite cell how many particles are there. The function will calculate the forces, both normal and tangential between particles, which are in contact, and update their normal and tangential force components, velocities, and the angular velocities.

The **main(...)** function will arrange the systematic call of all the member functions. This is done by creating an **object** of the class **Particle**. It will read the input data from the input file(user specified) by calling the member function **init(...)** and then call **march(...)** function. All other funtion calls are done from different functions as specified above.Apart from that the code is having several other functions which include **cross_prod(...),normalize(...)** and **surface(...)** which eventually helps in deciding the cross product of the of two vectors, normalize the vector and calculate the Vs/Vn vs β plot.

3.3 Algorithm used in the Code

3.3.1 User Defined Constants

N1	: The number of cells in x direction.
N2	: The number of cells in y direction.
N3	: The number of cells in z direction.
num	: Number of particles.
del_xx	: Cell length in x direction.
del_yy	: Cell length in y direction.
del_zz	: Cell length in z direction.
theta0	: The semi verticla angle of hopper.
alpha0	: The angle between the periodic boundary.
g	: Acceleration due to gravity.
ht	: Hopper height from the cone vertex.
K	:Stiffness for particle -wall contact.
dt	: Time step.
k1	: Stiffness for loading regime.
k2	: Stiffness for unloading regime.

3.3.2 Description of the Variables:

As Private Member in the *Class*

HOC [N1] [N2] [N3]	: The three dimensional array containing the head of the chain particle index. That means the last search particle in a cell.

LL[num]	: The link list array of maximum num element.
x[3][num]	: Position array of the particle, x[0] along x-direction, y[0] along y-direction and z[0] along z-direction.
xx[num]	: The initial x-positions of the particles are stored in this array
yy[num]	: The initial y-position of the particles are stored in this array
zz[num]	: The initial z-position o fthe particles are stored in this array
X[num]	: The updated x-positions of the particles are stored in this array.
Y[num]	: The updated y-positions of the particles are stored in this array.
Z[num]	: The updated z-position of th eparticles are stored in this array.
v[3][num]	: Initial velocities of the particles are stored in this array
v_x[num]	: Updated velocity along x-direction is stored in this array.
v_y[num]	: Updated velocity along y-direction is stored in this array.
v_z[num]	: Updated velocity along z-direction is stored in this array
Vn[num]	: Normal velocity of the particles
Vn_x[num]	: Normal velocity of the particles along x-direction
Vn_y[num]	: Normal velocity component of the particle along y-direction
Vn_z[num]	: Normal velocity component of the particle along z-direction
F[3][num]	: Three different components of force acting on the particles
f[3][num]	: Old values of three different force component on particle.

Fn_x[num]	: The normal component of the force acting on the particle rosolved in x-direction.
Fn_y[num]	: The normal component of the force acting on the particle resolved in y direction.
Fn_z[num]	: The normal component of the force acting on the particle resolved in z direction.
T_x[num]	: Tangential force resolved in x-direction.
T_y[num]	: Tangential force resolved in y direction.
T_z[num]	: Tangential force resolved in z direction.
T_old_x[num]	: Old value of the tangential component of the force resolved in x-direction.
T_old_y[num]	:Old value of the tangential component of the force resolved in y direction.
T_old_z[num]	: Old value of the tangential component of the force resolved in z direction.
T_tot[3][num]	: Total tangential force acting on the particle in three different direction.
Tp_x[num]	: Parallel component of the tangential force resolved in x axis.
Tp_y[num]	: Parallel component of the tangential force resolved in y axis.
Tp_z[num]	: Parallel component of the tangential force resolved in z axis.
Tn_x[num]	: Normal component of the tangential force resolved in x axis.
Tn_y[num]	:Normal component of the tangential force resolved in y axis.

Tn_y[num]	:Normal component of the tangential force resolved in y axis.
Tn_z[num]	: Normal component of the tangential force resolved in z axis.
w[3][num]	: Angular velocity
w_old[3][num]	: Old value of the angular velocity.
PB[num]	: Normal distance of the center of the particle from the wall of the hopper.
PA[num]	: Distance of the center of the particle from the hopper wall in normal direction to the axis.
BP_x[num]	: The normalized unit vector normal to the hopper wall resolved in x axis.
BP_y[num]	: The normalized unit vector normal to the hopper wall resolved in y axis.
BP_z[num]	: The normalized unit vector normal to the hopper wall resolved in z axis.
mass[num]	: The mass of the hopper.
rad[num]	: The radius of the particle.

In function *normalize(...)*

k_unit[3]	: Normalized unit distance vector between the center of the particles

In function *listing(...)* and *search(...)*

m	: The index of the particle searched in the cell.
K_0	: variable that keeps the value of the stiffness.

k_n[3]	: The variable to that is passed by reference and get the normalized unit distance vector of the particles.
Fn[num]	: Modulus of the force vector
fn[num]	: Modulus of the old force vector.
mod_proj_T[num]	: Modulus of the projected tangential force vector
T_star[num]	: Initialized to zero and takes subsequent total tangential force.
del_x	: Distance between the particle centers in x-direction.
del_y	: Distance between the particle centers in y direction.
del_z	: Distance between the particle centers in z direction.
prod_x	: Variable passed as reference and get the cross product resolved in x direction.
prod_y	: Variable passed as reference and get the cross product resolved in y direction.
prod_z	: Variable passed as reference and get the cross product resolved in z direction.
prod_x2	: Variable passed as reference and get the cross product resolved in x direction.
prod_y2	: Variable passed as reference and get the cross product resolved in y direction.
prod_z2	: Variable passed as reference and get the cross product resolved in z direction.

t_x[num]	: Normalized unit projected tangential force vector in x-direction.
t_y[num]	: Normalized unit projected tangential force vector in y direction.
t_z[num]	: Normalized unit projected tangential force vector in zdirection.
del_sp_x[num]	: Deformation parallel to the plane resolved in x direction.
del_sp_y[num]	:Deformation unit vector paralle to the plane resolved in y direction.
del_sp_z[num]	:Deformation unit vector parallel to the plane resolved in z direction.
del_sn_x[num]	: Deformation unit vector normal to the plane resolved in x direction.
del_sn_y[num]	: Deformation unit vector normal to the plane resolved in y direction.
del_sn_z[num]	:Deformation unit vector normal to the plane resolved in z direction.
R	: The distance between the searched particle and the the particle for which the searching is done.

In function *march(...)*

theta1[num]	: The angle between the line passing through the center

In function ***march(...)***

theta1[num]	: The angle between the line passing through the center of the particle, cutting vertex of the hopper and the axis of the hopper.
alpha1[num]	: The angle between the line passing through the center of the particle, cutting hopper axis and the x axis.
X_a[num]	: x coordinate of the point on the wall cut by the line which radially comes out of the axis of the hopper and passing through the center of the particle.
Y_a[num]	: y coordinate of the point on the wall cut by the line which radially comes out of the axis of the hopper and passing through the center of the particle.
Z_a[num]	: z coordinate of the point on the wall cut by the line which radially comes out of the axis of the hopper and passing through the center of the particle.
nearrad[num]	: Radius of the hopper in the same plane as the particle is and normal to the verticle plane.
v_x_loc[num]	: Velocity along the x-direction, modified for the loop.
v_y_loc[num]	: Velocity along the y-direction, modified for the loop.
v_z_loc[num]	: Velocity along the y-direction, modified for the loop.
F_x_loc[num]	: Modified local force component along x direction.
F_y_loc[num]	: Modified local foorce component along y direction.

F_z_loc[num]	: Modified local force component along z direction.
ndt	: Number oof time steps

The function march(...) can access all the member datafield declared as the private member of the class Particle. After creating an object of class Particle and initializing the required parameters this member funnction is called. This function at the begining opens up output datafile to write the positions of the particles at different time steps. Then it operates the following steps.

1. do for all time steps ndt.

    1.1 do for all particle j.

Determine alpha1 from the equation

$$alpha[j] \approx \tan^{-1} \frac{Y[j]}{X[j]} \tag{3.5}$$

Determine radial distance of the particle (rr) from the axis of the hopper

$$rr = \sqrt{X^2 + Y^2} \tag{3.6}$$

Determine if
$$\begin{matrix} alpha1[j] > alpha0 \\ alpha1[j] < 0 \end{matrix} \tag{3.7}$$

if true
$$alpha1[j] = f\,\mathrm{mod}(alpha1[j], alpha0) \tag{3.8}$$

and
$$\begin{matrix} X[j] = rr \cdot \cos(alpha1[j]) \\ Y[j] = rr \cdot \sin(alpha1[j]) \end{matrix} \tag{3.9}$$

Determine X_a[j] ,Y_a[j] and Z_a[j] from the equations

$$X\_a = \frac{(ht - Z) \cdot \tan(theta0)}{\sqrt{1 + \tan^2(alpha1)} \cdot} \tag{3.10}$$

$$Y\_a = \frac{(ht - Z)\cdot \tan(theta0)\cdot \tan(alpha1)}{\sqrt{1+\tan^2(alpha1)}} \tag{3.11}$$

$$Z\_a = (ht - Z) \tag{3.12}$$

$$nearrad = \sqrt{X\_a^2 + Y\_a^2} \tag{3.13}$$

Determine the normal distance PB of the particle from the wall of the hopper.

$$PB = ((ht - Z)\cdot \tan(theta0) - \sqrt{X^2 + Y^2})\cdot \cos(theta0) \tag{3.14}$$

Determine if PB < radius of the particle (3.15)

if true calculate the unit normal vector $\hat{BP}$

$$\begin{aligned} F_n\_x &= -K(rad - PB)\cdot BP\_x \\ F_n\_y &= -K(rad - PB)\cdot BP\_y \\ F_n\_z &= -K(rad - PB)\cdot BP\_z. \end{aligned} \tag{3.16}$$

Add this force to the existing force on the particle just before collision

$$\begin{aligned} F\_x\_loc &= F[0] + F_n\_x \\ F\_y\_loc &= F[1] + F_n\_y \\ F\_z\_loc &= F[2] + F_n\_z \end{aligned} \tag{3.17}$$

Now update velocities and positions

$$v^n = v^{n-1} + \frac{F_{loc}\cdot dt}{mass} \tag{3.18}$$

$$x^n = x^{n-1} + v_x^n \cdot dt \tag{3.19}$$

where n is the current time step and n–1 is the previous time steps

else
$$\begin{aligned} F\_x\_loc &= F[0] \\ F\_y\_loc &= F[1] \\ F\_z\_loc &= F[2] \end{aligned} \tag{3.20}$$

and position and the velocity is calculated like before.

1.2 Call search(...) for all particle.

The search(...) gets the near radius for each particle as argument and search 27 near neighbot cells to find if there is any particle is there or not if any particle is there then it calculates the distances between the centers of the particle and the particle on which searching is done. If the distance is less than the summation of the radii of the particles then imposes the contact forces on both the particles and go for the next particle. If it finds one more particle in that cell it hands the control to the function listing(...).

```
1. Do for all cells N1, N2, N3
        a. Initialize the head of the chain HOC[][][] = $-1$.
        b. Initialize the link list to $-1$.
2. Do for all particle j
        a. Assign cell indices depending on the positions of the particle
                                        initiating the equation(2.20.2.21,2.22).
        b. Making of the link list is completed.
        c. Head of the chain HOC[][][] contains the most recent particle in the call.
3. Do for all particle j
        a. Get the cell index for the particle j as stored through the step 2.
                Let it be a, b, c
        b. Fix up the boundary conditions for the searching.
        c. Get the value of the HOC[][][] for all 27 cells
                                        m = HOC[a][b][c]     j to be in a,b,c cell        (3.21)
        d. If                           $m \neq -$
                                        $and$                                             (3.22)
                                        $m \neq j$
```

e.Calculate the value of R.

$$R = \sqrt{(X[j] - X[m])^2 + (Y[j] - Y[m])^2 + (Z[j] - Z[m])^2} \qquad (3.23)$$

f. If $$R < rad[j] + rad[m] \qquad (3.24)$$

(1)Apply normal force law.

(2)Calculate the deformation.

(3)Calculate projected tangential force on the slip plane of contact.

(4)Calculate K_T from the equation (2.5) if $|T_{proj}|$ increases or from the equation (2.6) if $|T_{proj}|$ decreases with time steps.

(5)Calculate T* from equation (2.7) if in the begining or or from the equation (2.8) if otherwise.

(6)Determine K_T from the equation (2.5) if T* is increasing or from equation (2.6) if T* is decreasing.

(7) Determine the value of the angular velocity from the equation (2.14).

(8) Determine the normal and tangential displacement (from the previous time step).

(9) Determine normal and tangential component of the tangential force in normal and tangential direction of the slip plane.and calculate the total tangential force from the equation (2.1).

g. Call listing(...)

The function listing(...) takes the particle (j) on which searching is done, and the most updated particle number (m) popped up by the HOC[][][], and searches for any other particle, if it finds that then calculates all forces, normal and tangential, acting on the particle. If it finds more particle then calls the funtion back by recursively.

1. Calculate LL[] $n = LL[m]$ (3.25)

2. If n is not equal to -1, then follow the steps from 3.4.1 to 3.4.2 of searching(...)

3. Get the value of LL[n].

4. If LL[n] is not equal to -1 then call listing(...) replacing the value of m by n.Else stop.

CHAPTER 4

THE SIMULATION RESULTS AND ANALYSIS

4.1 Introduction

The simulation was run varying a list of parameters including:

Particle sizes

Coefficient of friction

Particle stiffnesses

From the recorded data, plots were generated to analyze and interpret the data. In general, the plots showed results which were in qualitative agreement with previous study Walton[], but determination of coefficient of restitution, in normal and tangential, was found to be more subtle than what has been found out by Walton[].To test the code several test cases were run taking into consideration of two particles in two initial positions. The first case deals with the particles in the same height but in offset position to each other in y-direction, and the second case deals with the particles in same height and the same line in both x and y direction and were assigned velocities so that they will collide head on. In both the cases conservation of momentum was checked and found out to be completely in agreement to the Newton's law of inertia. Changing different parameters i.e. particle sizes, coefficient of restituion and particle stiffnesses the results were checked and found out to be in agreement.

4.2 Results

For the case one, the diameter of the first particle was taken to be equal to the diameter of the second particle and was assigned to be 0.1 . The x co-ordinate and the y co-ordinate of the first particle to be 2.0, 0.55 and those for the second particle to be 2.4,0.5. The velocity components along x and y direction were 5.0,0, and that for the second particle to be -5.0,0, the coefficient of friction was 0.3. The stiffnesses for loading and unloading were 900,1710. Time step essentially was assigned to be 0.0001. A loop of 900 time steps were executed and the data were recorded in a file. From the recorded data it is clear that particles are coming close to each other in the straight line and they are having finite time of collision and they are getting separated with an angle with the initial direction of approach.The momentum before collision and the momentum after collision were found to be 0 and 0 which perfectly agrees with the conservation of momentum. The conservation of kinetic energy is conserved for the aboved mentioned stiffnesses.

For the second case the first particle was placed in the position 2.0,0.1 in x and y direction respectively and the second particle was placed in the 2.4 and 0.1 in x and y direction respectively. The z co-ordinate of the particles are kept in constant that is to be 0.1. The velocities in x and y directions are 5.0 and 0 respectively for the first particle and that of -5.0 and 0 respectively for the second particle respectively. The diameter of the particles were taken to be constant and assigned 0.1. Time step to be 0.0001 and for time loops of 900 they were allowed to travel. The output data shows that they are in confidance with law of inertia. The summation of their momentum before collision was to be 0 and the summation of their momentum after collision was 0. The conservation of kinetic energy is found to be conserved for the stiffnesses of the particles 900 and 1710 respectively.

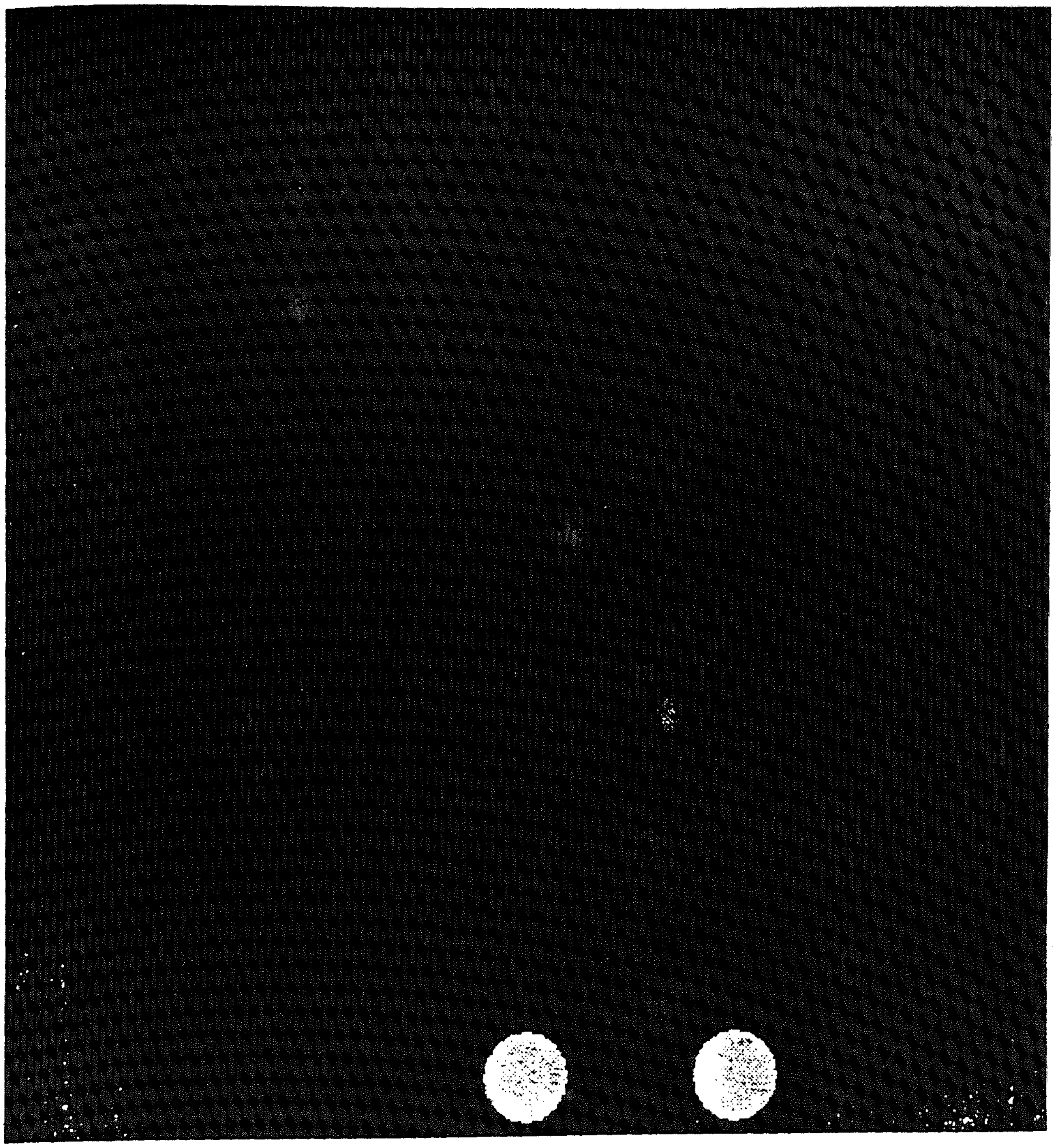


Figure 4.1 Two particles, aligned, started from their initial position

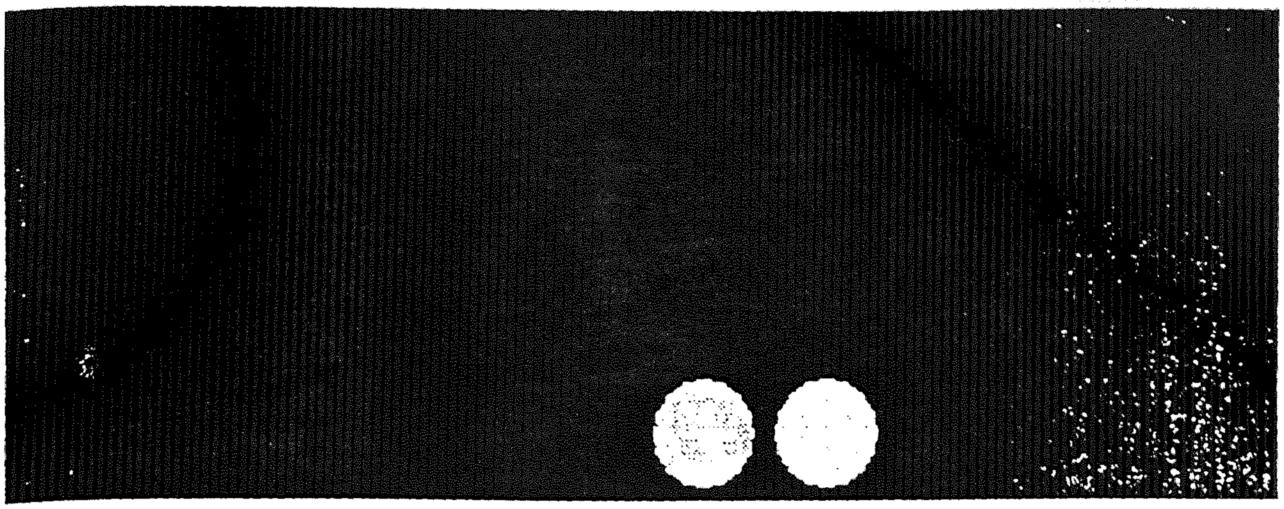


Figure 4.2 They are at the point of collision

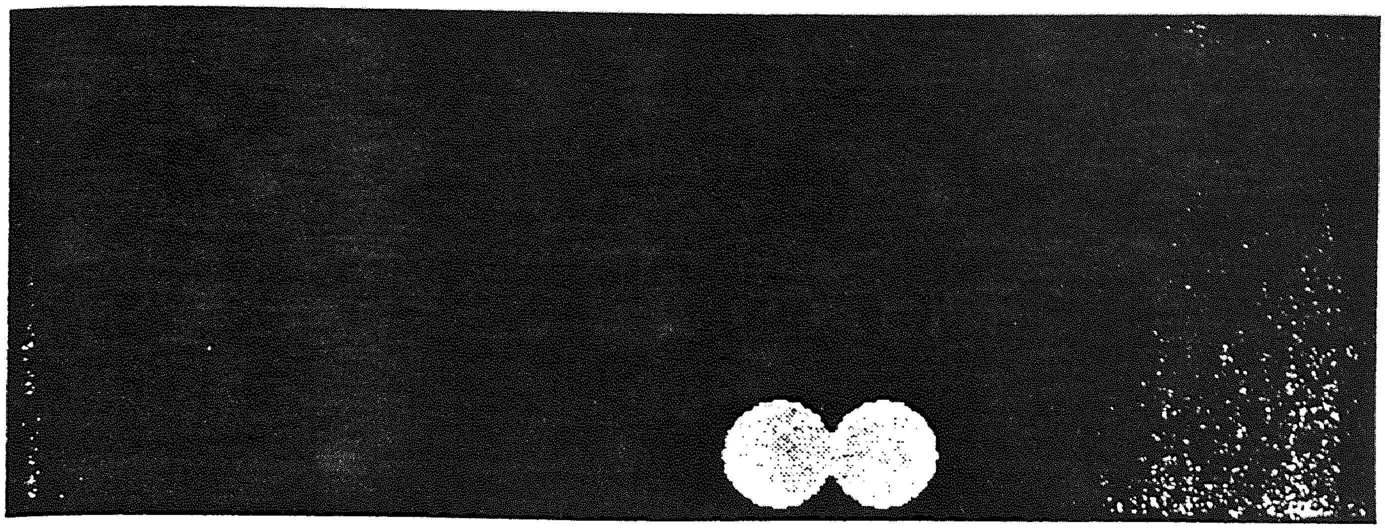


Figure 4.3 collision took place

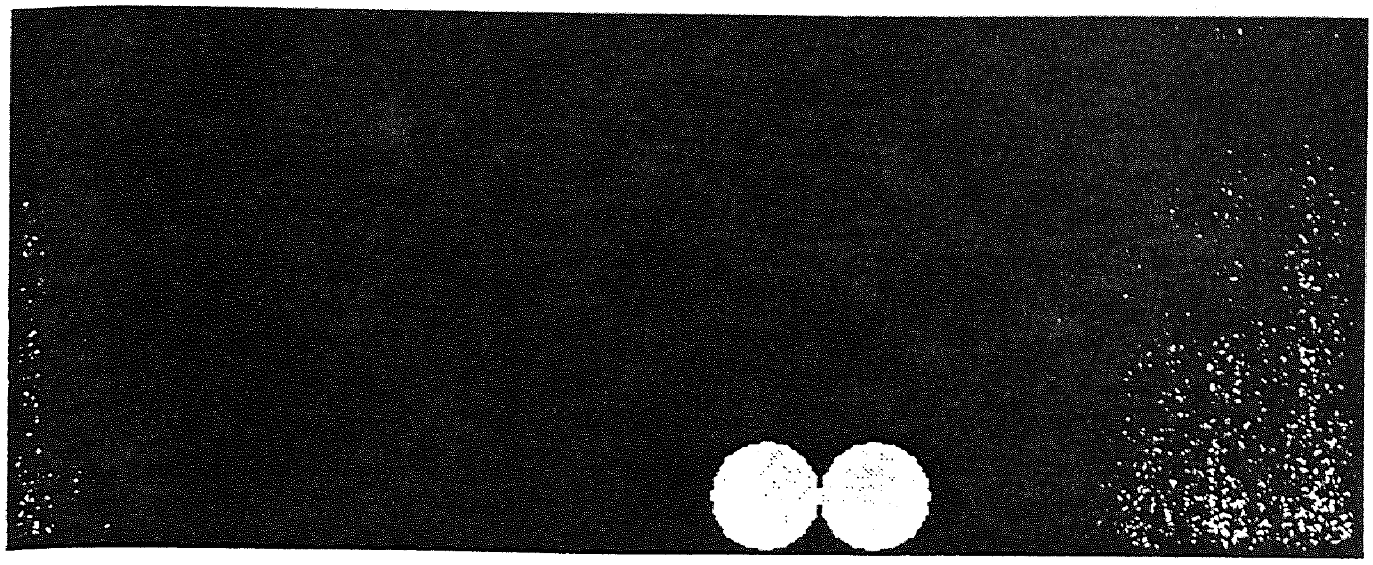


Figure 4.4 Separation going on

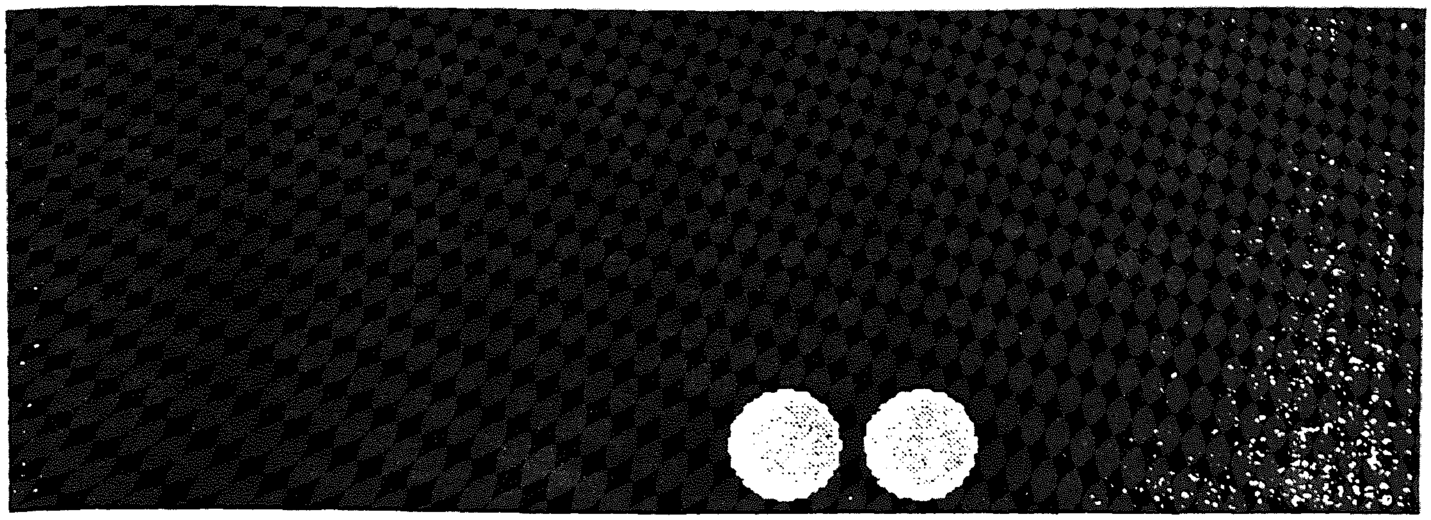


Figure 4.5 They are separated

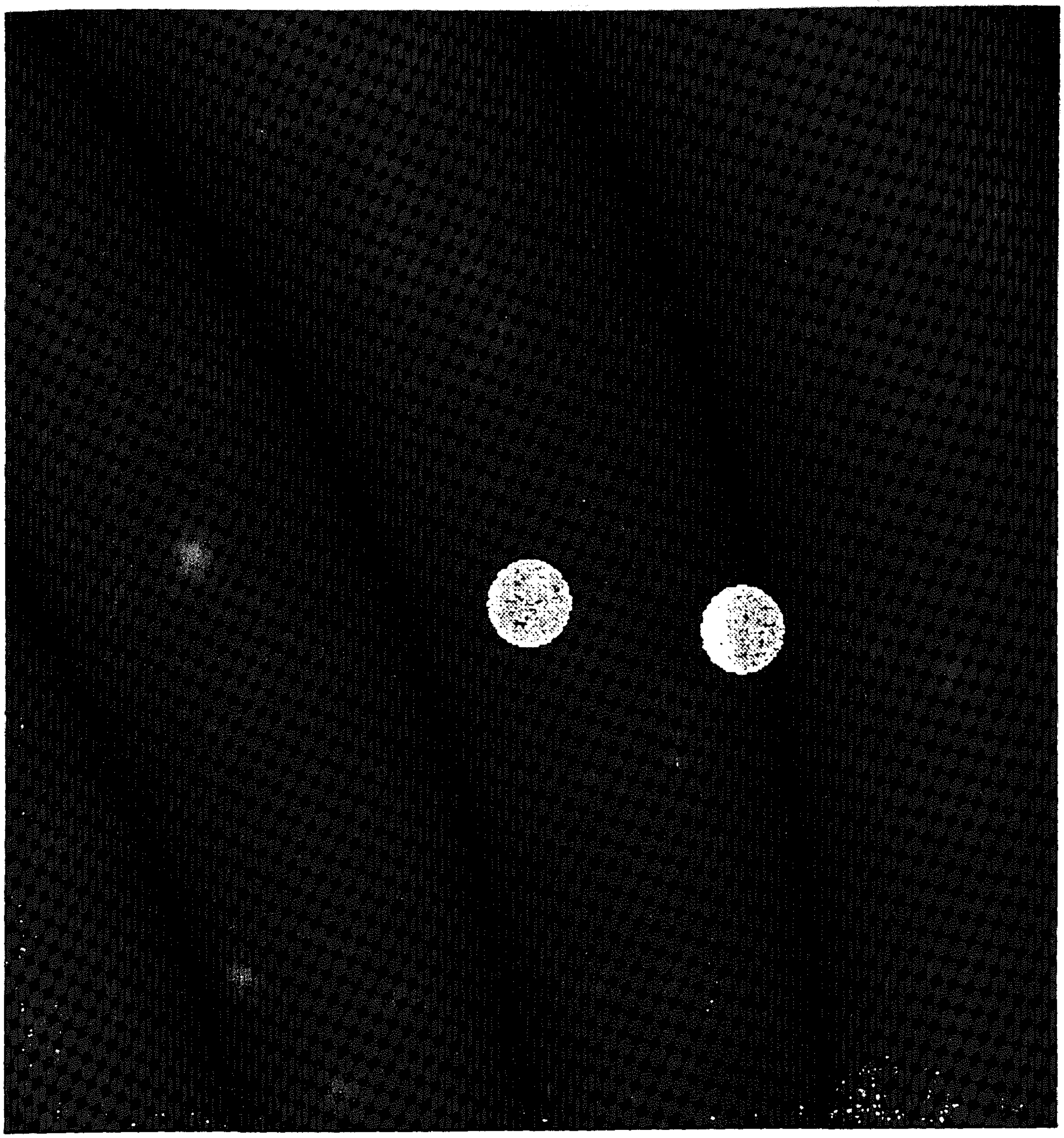


Figure 4.6 Two particles, in offset approaching

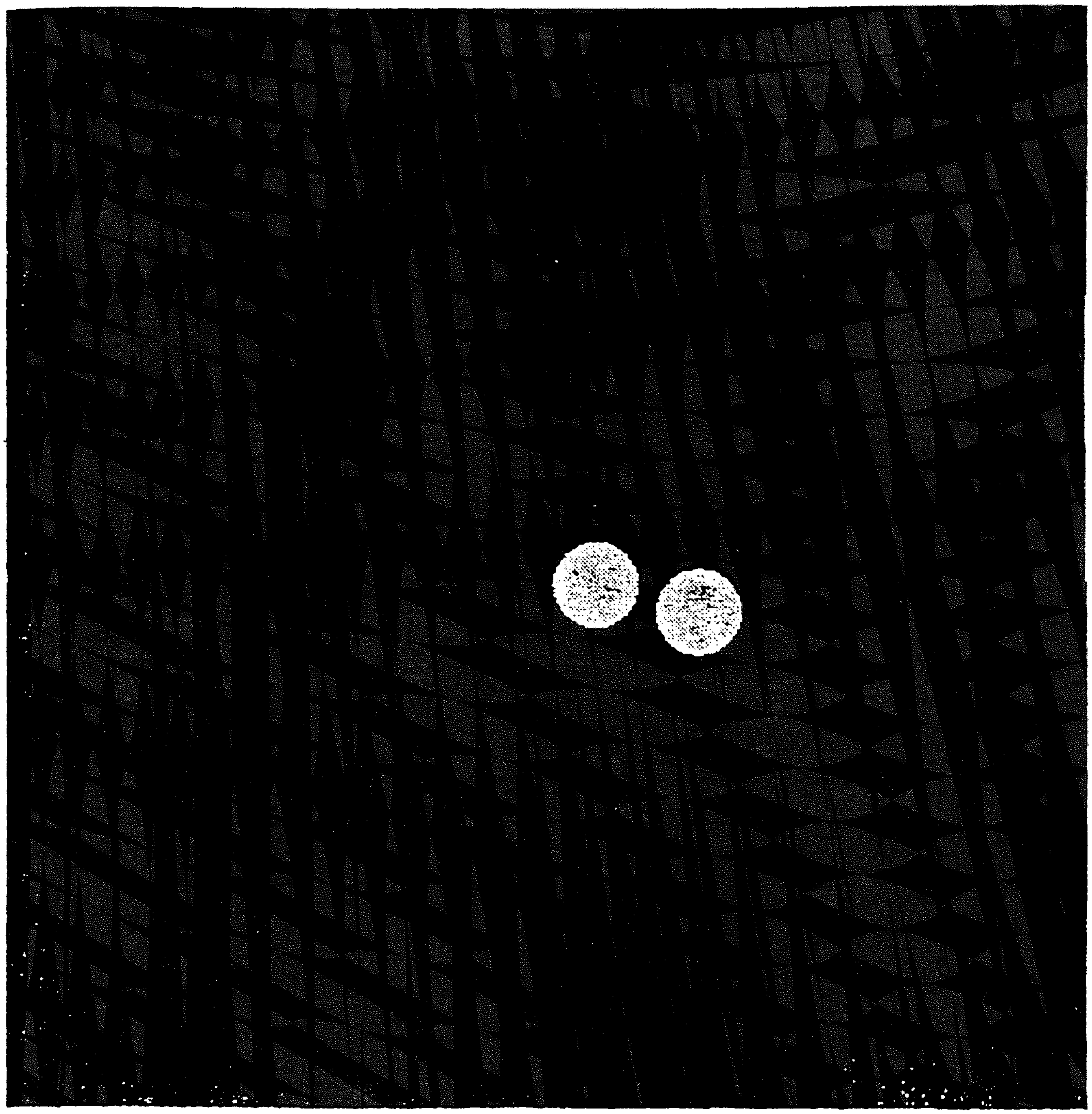


Figure 4.7 Two particles, in offset not yet collided

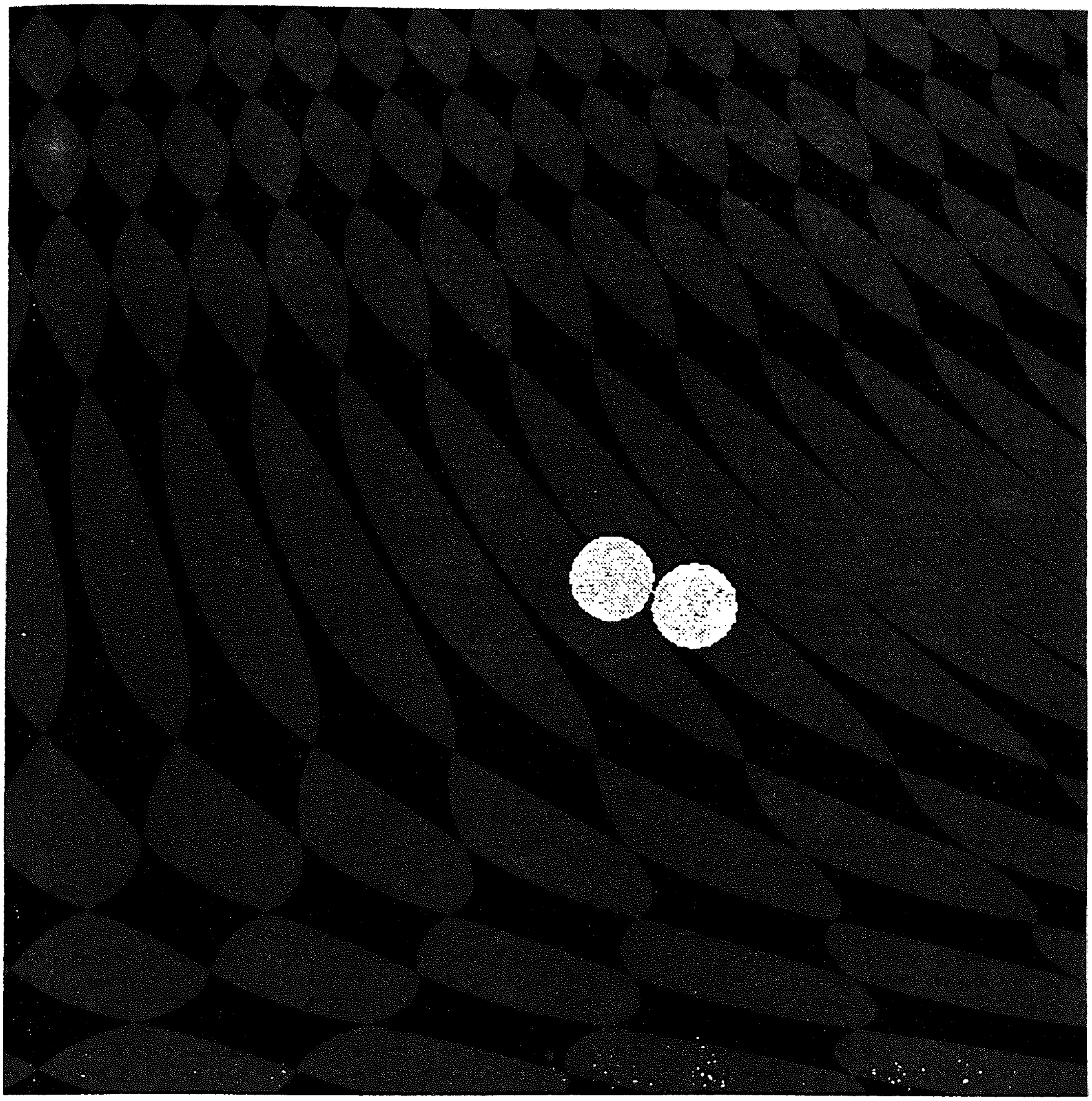


Figure 4.8 Collision took place

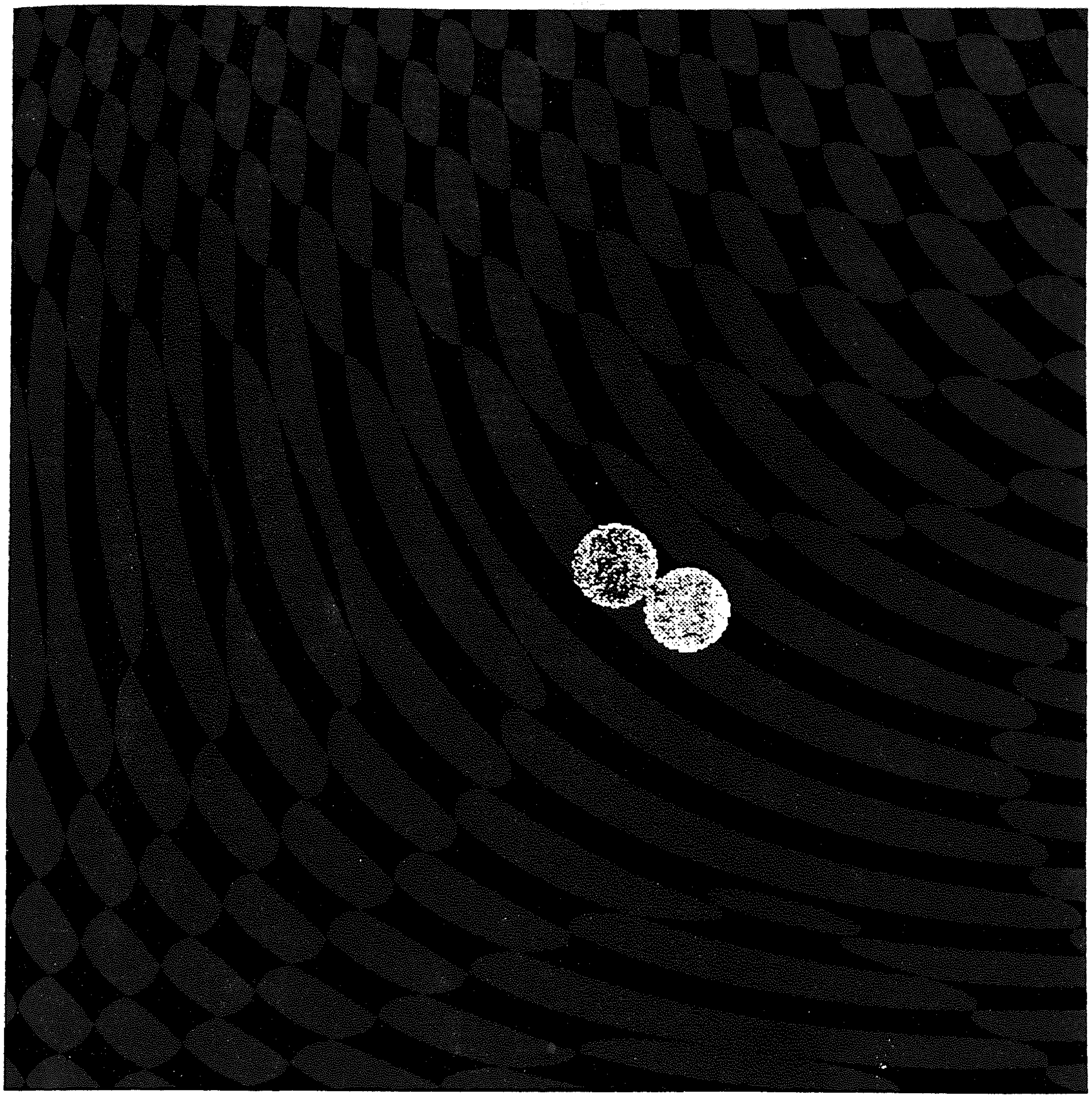


Figure 4.9 They are separating at different angles

CHAPTER 5

SUMMARY AND CONCLUSION

The flow behavior of granular particles is determined by the interactions between the individual particles. Many parameters concerned with the particle itself are of influence of this behavior, such as surface energy, chemical composition, particle shape and particle size distribution. To exactly define particle interaction based on these parameters will be very difficult. However particle interaction can be modelled by parameters such as friction, normal coefficient of restitution, and tangential coefficient of restitution. These factor combines different forces such as tangential forces, normal forces.

The JKR theory (Johnson *et al.*, 1971, Johnson, 1985) makes use of surface energy to model adhesive force. The Hertz theory (Johnson, 1985) is a special case of this theory for non-adhesive particles. Thornton (1991) developed a theory for the influence of the tangential force on the interparticle contact, based on the work by Savkoor and Briggs (1977) and Mindlin and Deresiewicz (1953). These theories except the adhesive force model have been used in this code.

The code shows us, larger the value of the stiffness lesser the number of contact of the particles and there is a tendency of the mass to go upwards.

The following future steps are suggested:

(a) Introduction of adhesive force will improve the flow behaviour of the powder.

To do this one should focus on the surface energy, rather than on the friction between particles. Abief description of *van der waals force* is described below, which can be used as the model for cohesiveness.

van der waals forces:These forces arise from neutral atoms and molecules, and therefore, are always present, like gravitational forces.

The van der waals force, F_{vdW} , between two spheres can be calculated by:

$$F_{vdW} = \frac{h}{8\Pi z^2}\frac{R_1 R_2}{R_1 + R_2} \tag{5.1}$$

This is the force between spheres 1 and sphere 2 with radii R_1 and R_2, where h is the lifshitz-van der waals contant, which is in the order 1.10^{-19} J, and z is the atomic separation between the surfaces.

(b) Introduction to electrostatic force will make the model more realistic.Electrostatics forces are the one that is caused by the charges of the particles. According to Bailey (1984), the coulomb force F_c between the charged and adjacent uncharged particle is given by:

$$F_c = \frac{q^2\left[1 - \frac{z^2}{R^2 + z^2}\right]}{16\Pi\varphi_0\varphi_r z^2} \tag{5.2}$$

here $\varphi_0\varphi_r$ is the dielectric constant of the medium between the particles, R is the radius of the both particles and q is the charge of the particle.

APPENDIX A

MAIN PARTS OF THE SIMULATION CODE

```
//This Simulation Code is Created by Krishnendu Roy, Graduated
Student //of NJIT.All rights reserved.  No parts of this code may
be reproduced, //sorted, or transmitted, in any form or by any
means,electronic,mechanical, //photocopying,recording, or
otherwise, without the prior written permission //of the
creator,or Dr.R.Dave,associated professor,or Dr.P.  Singh,
assistant //professor of ME Dept.  NJIT.


#include <iostream.h> #include <stdlib.h> #include <math.h>
#include <fstream.h> #define N1 5 #define N2 5 #define N3 5
#define num 250 #define del_xx 0.2 #define del_yy 0.2 #define
h_actual 3 //actual hopper height for reference #define del_zz
0.6 #define theta0 30 #define alpha0 30 #define g 9.8 #define PI
3.14159265 #define ht 7.0 //hopper height from the cone vertex
#define K 1000 #define dt 0.01 #define k1 500 #define k2 1500
#define mu 0.3 #define I 100

//////////////////////////////////////////////////////////////////
///////////// The main Class myclass ////////////////////
////////////////////////////////////////////////////////////////


class myclass {

private:  int HOC[N1][N2][N3]; /* head of the chain*/

double x[3][num]; /* position array of the particle,x_1 along x
axis, x_2 along y axis, x_3 along z axis */ double v[3][num]; /*
velocity array of the particle, v_1 vel.  along x axis, v_2 vel.
along y axis, v_3 vel.  along z axis*/ double
v_x[num],v_y[num],v_z[num];

double f[3][num]; /* force array.  f_1 force component along x
axis.  f_2 force component along y axis, f_3 force component
along z */

double rad[num];

int LL[num]; /* link list updating */

int flag; /* index for chosing model */

double PB[num],PA[num]; double BP_x[num],BP_y[num],BP_z[num];
double Vn[num],Vn_x[num],Vn_y[num],Vn_z[num]; double mass[num];
// mass of the particles double F[3][num]; double
X[num],Y[num],Z[num]; // position array of the particle

double xx[num],yy[num],zz[num]; // initial positon stored in this
array

double T_1[num],T_2[num],T_col; // Time of collision of the
particle with wall

double T_minus[num]; // Time that exceeded the collision time
```

```
with wall

double A[num],B[num],C[num],D[num],E[num],G[num];// roots of the
collision function

double Fn_x[num],Fn_y[num],Fn_z[num]; // Normal Force components
in x,y,z direction

double x_t[num][num],x_t1[num][num]; // defomation at this time
step // and the previous time step double
T_x[num],T_y[num],T_z[num]; // Tangential force components in //
in x,y,z direction double T_old_x[num],
T_old_y[num],T_old_z[num]; // Old tangential force // components
double w[3][num], w_old[3][num];// angular velocity in x,y,z
directions and // and values in previous time steps double
T_tot[3][num]; // Total tangential forces acting on particle //
in three different directions.  double x0[num],y0[num],z0[num];
// position of the particle in previous time step

double Tp_x[num],Tp_y[num],Tp_z[num];// Tangential parallel
forces projected on // plane double Tn_x[num],
Tn_y[num],Tn_z[num];// Tangential(normal comp) forces //
projected on plane

public:  void init(char *p); // The member function that
initializes void march(); // Particles that marches with
time,action takes place // in this member funtion void
search(double *r); // The searching of 27 cells are completed //
in this member funtion void listing(int n,int sp); // This member
function helps member funtion // in this member funtion void
normalize(int ,int ,double &,double &,double &); // This member
// function normalizes the position vectors // of the particles
passed as the first two // arguments of it and assigns the
addresses // of it in the last three variables void
cross_prod(double &, double &, double &, double ,double,double,
double *); // This funtion calculates the cross product // of the
vectors passed into it };

// Defining the member function cross_prod(...)  // | i j k | //
AxB = | a1 b1 c1 | // | a2 b2 c2 | // = (b1*c2 - c1*b2)i + (a2*c1
- a1*c2)j + (a1*b2 - a2*b1)k

void myclass ::  cross_prod(double &prodX,double &prodY, double
&prodZ, double rot1,double rot2 ,double rot3, double *k) { prodX
= rot1*k[2] - rot3*k[1]; prodY = rot3*k[0] - rot1*k[2]; prodZ =
rot1*k[1] - rot2*k[0]; }

// Initialize the arrays for particle:  positions, velocity,
force and HOC

void myclass ::  init(char *p) {

ifstream fin; fin.open(p);

for(int i = 0; i < num; i++){ for(int j = 0; j < 3; j++){ fin >>
x[j][i] >> v[j][i] >> f[j][i]; //x0[j][i] = x[j][i]; //
```

```
v[j][i]=0.0;

} v_x[i] = v[0][i]; v_y[i] = v[1][i]; v_z[i] = v[2][i]; rad[i] =
0.05;

X[i] = x[0][i]; Y[i] = x[1][i]; Z[i] = x[2][i];

xx[i] = x[0][i]; yy[i] = x[1][i]; zz[i] = x[2][i];

// initializes the the angular velocities of the particle

w[0][i] = 10.0*PI/180.0; w[1][i] = 10.0*PI/180.0; w[2][i] =
10.0*PI/180.0;

w_old[0][i] = 0.0; w_old[1][i] = 0.0; w_old[2][i] = 0.0;

T_tot[0][i] = 0.0; T_tot[1][i] = 0.0; T_tot[2][i] = 0.0;

T_old_x[i] = 0.0; T_old_y[i] = 0.0; T_old_z[i] = 0.0;

mass[i] = 0.1; F[0][i] = 0.0; F[1][i] = 0.0; F[2][i] =
(mass[i]*g);

// initializes the deformations

for(int k = 0; k < num; k++){ x_t[i][k] = -1.0; x_t1[i][k] =
-1.0; }

} fin.close();

for(int q =0; q < N1; q++){ for(int r = 0; r < N2; r++){ for(int
s = 0; s < N3; s++){ HOC[q][r][s] = -1; } } }


cout <<"Input which contact force model you would like to use:
enter 1 for Otis enter 2 for Lenn.  jones enter 3 for hard sphere
enter 4 for Cohesive model\n"<<"Enter :"; // cin >> flag;

}

// Function that normalizes

void myclass ::  normalize(int nsp,int nm,double &kn1,double
&kn2,double &kn3) { double k_unit[3];

k_unit[0] = (X[nsp] - X[nm])/
sqrt((X[nsp]-X[nm])*(X[nsp]-X[nm])+(Y[nsp]-Y[nm])*(Y[nsp]-Y[nm])
+(Z[nsp]-Z[nm])*(Z[nsp]-Z[nm]));

k_unit[1] = (Y[nsp]-Y[nm])/
sqrt((X[nsp]-X[nm])*(X[nsp]-X[nm])+(Y[nsp]-Y[nm])*(Y[nsp]-Y[nm])
+(Z[nsp]-Z[nm])*(Z[nsp]-Z[nm]));

k_unit[2] =
(Z[nsp]-Z[nm])/sqrt((X[nsp]-X[nm])*(X[nsp]-X[nm])+(Y[nsp]-Y[nm])*(Y[
+(Z[nsp]-Z[nm])*(Z[nsp]-Z[nm]));
```

```
kn1 = k_unit[0]; kn2 = k_unit[1]; kn3 = k_unit[2];

}


// Function for looking at particles in the HOC

void myclass ::  listing(int n,int sp) { int m; double
k_i,k_j,k_k; double Ko; double xt; double mod[num]; double
mod_T[num]; double k_n[3]; double Fn[num]; double fn[num]; double
mod_proj_T[num]; double T_star[num]; double Kt[num]; double
del_x,del_y,del_z; double del_x_old,del_y_old,del_z_old; double
prod_x,prod_y,prod_z; double prod_x2, prod_y2, prod_z2; double
del_s_x, del_s_y, del_s_z; double t_x[num],t_y[num], t_z[num];
double del_sp_x[num],del_sp_y[num],del_sp_z[num]; double
del_sn_x[num],del_sn_y[num],del_sn_z[num]; double
mod_proj_T_old[num]; double k_n1,k_n2,k_n3; double R;


if(LL[n] != -1){ m = LL[n]; if (m!=sp) {

R = sqrt((X[sp]-X[m])*(X[sp]-X[m]) + (Y[sp]-Y[m])*(Y[sp]-Y[m])
+(Z[sp]-Z[m])*(Z[sp]-Z[m]));

if(R < (rad[sp] + rad[m])) { x_t[sp][m] = rad[sp] + rad[m] - R;

if(x_t[sp][m] > x_t1[sp][m]){ Ko = k1; xt = x_t[sp][m];}

else {Ko = k2 - k1; xt = x_t[sp][m]; }

F[0][sp] += Ko*(-xt)*(X[m]-X[sp])/R; F[1][sp] +=
Ko*(-xt)*(Y[m]-Y[sp])/R; F[2][sp] += Ko*(-xt)*(Z[m]-Z[sp])/R;

F[0][m] -= Ko*(-xt)*(X[m]-X[sp])/R; F[1][m] -=
Ko*(-xt)*(Y[m]-Y[sp])/R; F[2][m] -= Ko*(-xt)*(Z[m]-Z[sp])/R;


x_t1[sp][m] = x_t[sp][m];

normalize(sp,m,k_i,k_j,k_k);



mod[sp] = sqrt((T_old_x[sp] -
k_n1*(k_n1*T_old_x[sp]))*(T_old_x[sp] - k_n1*(k_n1*T_old_x[sp]))
+ (T_old_y[sp] - k_n2*(k_n2*T_old_y[sp]))*(T_old_y[sp] -
k_n2*(k_n2*T_old_y[sp])) + (T_old_z[sp] -
k_n3*(k_n3*T_old_z[sp]))*(T_old_z[sp] -
k_n3*(k_n3*T_old_z[sp])));

mod[m] = sqrt((T_old_x[m] - k_n1*(k_n1*T_old_x[m]))*(T_old_x[m] -
k_n1*(k_n1*T_old_x[m])) + (T_old_y[m] -
k_n2*(k_n2*T_old_y[m]))*(T_old_y[m] - k_n2*(k_n2*T_old_y[m])) +
(T_old_z[m] - k_n3*(k_n3*T_old_z[m]))*(T_old_z[m] -
k_n3*(k_n3*T_old_z[m]))) ;
```

```
mod_T[sp] = T_old_x[sp]*T_old_x[sp] + T_old_y[sp]*T_old_y[sp] +
T_old_z[sp]*T_old_z[sp]; mod_T[m] = T_old_x[m]*T_old_x[m] +
T_old_y[m]*T_old_y[m] + T_old_z[m]*T_old_z[m];


T_x[sp] = (T_old_x[sp] - k_n1*(k_n1*T_old_x[sp]))*mod_T[sp] /
mod[sp]; T_y[sp] = (T_old_y[sp] -
k_n2*(k_n2*T_old_y[sp]))*mod_T[sp]/mod[sp]; T_z[sp] =
(T_old_z[sp] - k_n3*(k_n3*T_old_z[sp]))*mod_T[sp]/mod[sp];

T_x[m] = (T_old_x[m] - k_n1*(k_n1*T_old_x[m]))*mod_T[m] / mod[m];
T_y[m] = (T_old_y[m] - k_n2*(k_n2*T_old_y[m]))*mod_T[m]/mod[m];
T_z[m] = (T_old_z[m] - k_n3*(k_n3*T_old_z[m]))*mod_T[m]/mod[m];

k_n[0] = k_n1; k_n[1] = k_n2; k_n[2] = k_n3;



Fn[sp] = F[0][sp]*F[0][sp] + F[1][sp]*F[1][sp] +
F[2][sp]*F[2][sp]; fn[sp] = f[0][sp]*f[0][sp] + f[1][sp]*f[1][sp]
+ f[2][sp]*f[2][sp];


Fn[m] = F[0][m]*F[0][m] + F[1][m]*F[1][m] + F[2][m]*F[2][m];
fn[m] = f[0][m]*f[0][m] + f[1][m]*f[1][m] + f[2][m]*f[2][m];


mod_proj_T[sp] = sqrt(T_x[sp]*T_x[sp] + T_y[sp]*T_y[sp] +
T_z[sp]*T_z[sp]); mod_proj_T[m] = sqrt(T_x[m]*T_x[m] +
T_y[m]*T_y[m] + T_z[m]*T_z[m]);


T_star[sp] = sqrt(T_x[sp]*T_x[sp] + T_y[sp]*T_y[sp] +
T_z[sp]*T_z[sp])*Fn[sp] / fn[sp] ; T_star[m] = sqrt(T_x[m]*T_x[m]
+ T_y[m]*T_y[m] + T_z[m]*T_z[m])*Fn[m] / fn[m] ;

if(mod_proj_T_old[sp] > mod_proj_T[sp])

Kt[sp] = Ko*(1 - (mod_proj_T[sp] - T_star[sp])/(mu*Fn[sp] -
T_star[sp]));

else

Kt[sp] = Ko*(1 - (T_star[sp] - mod_proj_T[sp])/(mu*Fn[sp] +
T_star[sp]));

if(mod_proj_T_old[m] > mod_proj_T[m])

Kt[m] = Ko*(1 - (mod_proj_T[m] - T_star[m])/(mu*Fn[m] -
T_star[m]));

else

Kt[m] = Ko*(1 - (T_star[m] - mod_proj_T[m])/(mu*Fn[m] +
T_star[m]));


del_x = X[sp] - X[m]; del_y = Y[sp] - Y[m]; del_z = Z[sp] - Z[m];

del_x_old = x0[sp] - x0[m]; del_y_old = y0[sp] - y0[m]; del_z_old
= z0[sp] - z0[m];

w[0][sp] = w_old[0][sp] + T_tot[0][sp]*dt/I; w[1][sp] =
w_old[1][sp] + T_tot[1][sp]*dt/I; w[2][sp] = w_old[2][sp] +
T_tot[2][sp]*dt/I;

w[0][m] = w_old[0][m] + T_tot[0][m]*dt/I; w[1][m] = w_old[1][m] +
T_tot[1][m]*dt/I; w[2][m] = w_old[2][m] + T_tot[2][m]*dt/I;



cross_prod(prod_x,prod_y, prod_z,w[0][sp],w[1][sp],w[2][sp],k_n);


cross_prod(prod_x2,prod_y2,prod_z2,w[0][m],w[1][m],w[2][m],k_n);

del_s_x = del_x - del_x_old - k_n1*k_n1*(del_x - del_x_old) +
(X[sp]*prod_x + X[m]*prod_x2)*dt;

del_s_y = del_y - del_y_old - k_n2*k_n2*(del_y - del_y_old); +
(Y[sp]*prod_y + Y[m]*prod_y2)*dt;

del_s_z = del_z - del_z_old - k_n3*k_n3*(del_z - del_z_old) +
(Z[sp]*prod_z + Z[m]*prod_z2)*dt;

t_x[sp] = T_x[sp]/mod_proj_T[sp]; t_y[sp] =
T_y[sp]/mod_proj_T[sp]; t_z[sp] = T_z[sp]/mod_proj_T[sp];

t_x[m] = T_x[m]/mod_proj_T[m]; t_y[m] = T_y[m]/mod_proj_T[m];
t_z[m] = T_z[m]/mod_proj_T[m];


del_sp_x[sp] = del_s_x*t_x[sp]*t_x[sp]; del_sp_y[sp] =
del_s_y*t_y[sp]*t_y[sp]; del_sp_z[sp] = del_s_z*t_z[sp]*t_z[sp];

del_sp_x[m] = del_s_x*t_x[m]*t_x[m]; del_sp_y[m] =
del_s_y*t_y[m]*t_y[m]; del_sp_z[m] = del_s_z*t_z[m]*t_z[m];


del_sn_x[sp] = del_s_x - del_sp_x[sp]; del_sn_x[sp] = del_s_x -
del_sp_y[sp]; del_sn_z[sp] = del_s_z - del_sp_z[sp];

del_sn_x[m] = del_s_x - del_sp_x[m]; del_sn_x[m] = del_s_x -
del_sp_y[m]; del_sn_z[m] = del_s_z - del_sp_z[m];


Tp_x[sp] = T_x[sp] + Kt[sp] * del_sp_x[sp]; Tp_y[sp] = T_y[sp] +
Kt[sp] * del_sp_y[sp], Tp_z[sp] = T_z[sp] + Kt[sp] *
del_sp_z[sp];
```

```
Tp_x[m] = T_x[m] + Kt[m] * del_sp_x[m]; Tp_y[m] = T_y[m] + Kt[m]
* del_sp_y[m]; Tp_z[m] = T_z[m] + Kt[m] * del_sp_z[m];


Tn_x[sp] = Ko * del_sn_x[sp]; Tn_x[sp] = Ko * del_sn_y[sp];
Tn_z[sp] = Ko * del_sn_z[sp];

Tn_x[m] = Ko * del_sn_x[m]; Tn_x[m] = Ko * del_sn_y[m]; Tn_z[m] =
Ko * del_sn_z[m];


T_tot[0][sp] = Tp_x[sp] + Tn_x[sp]; T_tot[1][sp] = Tp_y[sp] +
Tn_y[sp]; T_tot[2][sp] = Tp_z[sp] + Tn_z[sp];

T_tot[0][m] = Tp_x[m] + Tn_x[m]; T_tot[1][m] = Tp_y[m] + Tn_y[m];
T_tot[2][m] = Tp_z[m] + Tn_z[m];


T_old_x[sp] = T_x[sp]; T_old_y[sp] = T_y[sp]; T_old_z[sp] =
T_z[sp];

T_old_x[m] = T_x[m]; T_old_y[m] = T_y[m]; T_old_z[m] = T_z[m];

mod_proj_T_old[sp] = mod_proj_T[sp]; mod_proj_T_old[m] =
mod_proj_T[m];


} listing(m,sp);

}


} else cout<<"No particle left"<<endl; return; }

void myclass::search(double *r) { int m; int i,j,k; int
a,b,c,ek,dui,tin; int boro1,boro2,boro3; int nx,ny,nz; double
k_n1,k_n2,k_n3; double Ko; double xt; double mod[num]; double
mod_T[num]; double k_n[3]; double Fn[num]; double fn[num]; double
mod_proj_T[num]; double T_star[num]; double Kt[num]; double
del_x,del_y,del_z; double del_x_old,del_y_old,del_z_old; double
prod_x,prod_y,prod_z; double prod_x2, prod_y2, prod_z2; double
del_s_x, del_s_y, del_s_z; double t_x[num],t_y[num], t_z[num];
double del_sp_x[num],del_sp_y[num],del_sp_z[num]; double
del_sn_x[num],del_sn_y[num],del_sn_z[num]; double
mod_proj_T_old[num]; double R;

int mm[num],nn[num],pp[num]; double x_s[num],y_s[num];

for( i = 0; i < num; i++){

F[0][i]= 0.0; F[1][i] = 0.0; F[2][i] = (mass[i]*g); LL[i] = -1;
Fn[i] = 0.0; fn[i] = 0.0;

mod_proj_T[i] = 0.0; T_star[i] = 0.0;

t_x[i] = 0.0; t_y[i] = 0.0; t_z[i] = 0.0;
```

```
del_sp_x[i] = 0.0; del_sp_y[i] = 0.0; del_sp_z[i] = 0.0;

del_sn_x[i] = 0.0; del_sn_y[i] = 0.0; del_sn_z[i] = 0.0;


}

////////////////////////////////////////////////////////////////////////
//////////////////// following three for loops assigns -1 to the
/////////////////////// ////////////////// HOC[][][] array
///////////////////////
////////////////////////////////////////////////////////////////////////

for(int ix = 0;ix < N1; ix++){ for(int iy = 0;iy < N2; iy++){
for(int iz = 0; iz <N3; iz++){ HOC[ix][iy][iz] = -1; }}}
////////////////////////////////////////////////////////////////////////
//////////////// The following loop allocates different
/////////////////////////// ///////////////// cells depending upon
the position of the particle ////////////////
////////////////////////////////////////////////////////////////////////


for(int ia = 0; ia< num; ia++){


x_s[ia] = X[ia]/r[ia]; y_s[ia] = Y[ia]/r[ia]; nx = int
(x_s[ia]/del_xx); ny = int (y_s[ia]/del_yy); nz = int
(Z[ia] del_zz); if(nz >= 5) cout <<"Z cells more than 4"<<endl;
if(nx >= 5) cout <<"X cells more than 4"<<endl; if(ny >= 5) cout
<<"Y cells more than 4"<<endl; mm[ia] = nx; nn[ia] = ny; pp[ia] =
nz;

LL[ia] = HOC[nx][ny][nz] ; HOC[nx][ny][nz] = ia; }
////////////////////////////////////////////////////////////////////////
//////////////// search back the indexing of the particle with
//////////////// ///////////////
//////////////////////////////////////////////////////////////////
for(int xyz = 0; xyz < num; xyz++)

for(int ii = 0; ii < num ; ii++){

i = mm[ii]; j = nn[ii]; k = pp[ii];

////////////////////////////////////////////////////////////////////////
/////////////////// k is getting more and more order of
13..///////
//////////////////////////////////////////////////////////////////////
cout<<i<<j<<k<<endl;

////////////////////////////////////////////////////////////////////////
/////////////////// following if statements assigns the boundary
////////// /////////////////// cell configurations and
specification ///////////////
////////////////////////////////////////////////////////////////////////
```

```
if(i == 0){ ek = 0; borol = i+1;} else if(i == 4){ek = i-1;borol
= 4;} else { ek = i-1;borol = i+1;}

if(j == 0) {dui = 0;boro2 = j+1;} else if(j == 4){dui = j-1;boro2
= 4;} else { dui = j-1; boro2 = j+1;} if(k == 0) {tin = 0;boro3 =
k+1;} else if(k == 4){tin = k-1;boro3 = 4;} else{tin = k-1; boro3
= k+1;}


/////////////////////////////////////////////////////////////////////
////////////////////// following is the nesting over the boundary
27 //////////////// ///////////////////// search cells
////////////////
/////////////////////////////////////////////////////////////////////

for(a = ek ; a <= borol;a++){ for(b = dui; b <= boro2;b++){ for(c
= tin; c<= boro3;c++){

m = HOC[a][b][c];


if((m != -1)&&(ii!=m)){


R = sqrt((X[ii] - X[m])*(X[ii] - X[m]) + (Y[ii] - Y[m])*(Y[ii] -
Y[m]) + (Z[ii] - Z[m])*(Z[ii] - Z[m])); // distance of the part.
ii and m is calculated here

if(R < (rad[ii] + rad[m])) {



x_t[ii][m] = rad[ii] + rad[m] - R;

if(x_t[ii][m] > x_t1[ii][m]){ Ko = k1; xt = x_t[ii][m]; }

else {Ko = k2 - k1; xt = x_t1[ii][m]; }

F[0][ii] += Ko*(-xt)*(X[m]-X[ii])/R; F[1][ii] +=
Ko*(-xt)*(Y[m]-Y[ii])/R; F[2][ii] += Ko*(-xt)*(Z[m]-Z[ii])/R;

F[0][m] -= Ko*(-xt)*(X[m]-X[ii])/R; F[1][m] -=
Ko*(-xt)*(Y[m]-Y[ii])/R; F[2][m] -= Ko*(-xt)*(Z[m]-Z[ii])/R;

x_t1[ii][m] = x_t[ii][m];

normalize(ii,m,k_n1,k_n2,k_n3);


mod[ii] = sqrt((T_old_x[ii] -
k_n1*(k_n1*T_old_x[ii]))*(T_old_x[ii] - k_n1*(k_n1*T_old_x[ii]))
+ (T_old_y[ii] - k_n2*(k_n2*T_old_y[ii]))*(T_old_y[ii] -
k_n2*(k_n2*T_old_y[ii])) + (T_old_z[ii] -
k_n3*(k_n3*T_old_z[ii]))*(T_old_z[ii] -
k_n3*(k_n3*T_old_z[ii])));
```

```
mod[m] = sqrt((T_old_x[m] - k_n1*(k_n1*T_old_x[m]))*(T_old_x[m] -
k_n1*(k_n1*T_old_x[m])) + (T_old_y[m] -
k_n2*(k_n2*T_old_y[m]))*(T_old_y[m] - k_n2*(k_n2*T_old_y[m])) +
(T_old_z[m] - k_n3*(k_n3*T_old_z[m]))*(T_old_z[m] -
k_n3*(k_n3*T_old_z[m]))) ;


mod_T[ii] = T_old_x[ii]*T_old_x[ii] + T_old_y[ii]*T_old_y[ii] +
T_old_z[ii]*T_old_z[ii]; mod_T[m] = T_old_x[m]*T_old_x[m] +
T_old_y[m]*T_old_y[m] + T_old_z[m]*T_old_z[m];


T_x[ii] = (T_old_x[ii] - k_n1*(k_n1*T_old_x[ii]))*mod_T[ii] /
mod[ii]; T_y[ii] = (T_old_y[ii] -
k_n2*(k_n2*T_old_y[ii]))*mod_T[ii]/mod[ii]; T_z[ii] =
(T_old_z[ii] - k_n3*(k_n3*T_old_z[ii]))*mod_T[ii]/mod[ii];


T_x[m] = (T_old_x[m] - k_n1*(k_n1*T_old_x[m]))*mod_T[m] / mod[m];
T_y[m] = (T_old_y[m] - k_n2*(k_n2*T_old_y[m]))*mod_T[m]/mod[m];
T_z[m] = (T_old_z[m] - k_n3*(k_n3*T_old_z[m]))*mod_T[m]/mod[m];


k_n[0] = k_n1; k_n[1] = k_n2; k_n[2] = k_n3;



Fn[ii] = F[0][ii]*F[0][ii] + F[1][ii]*F[1][ii] +
F[2][ii]*F[2][ii]; fn[ii] = f[0][ii]*f[0][ii] + f[1][ii]*f[1][ii]
+ f[2][ii]*f[2][ii];


Fn[m] = F[0][m]*F[0][m] + F[1][m]*F[1][m] + F[2][m]*F[2][m];
fn[m] = f[0][m]*f[0][m] + f[1][m]*f[1][m] + f[2][m]*f[2][m];


mod_proj_T[ii] = sqrt(T_x[ii]*T_x[ii] + T_y[ii]*T_y[ii] +
T_z[ii]*T_z[ii]); mod_proj_T[m] = sqrt(T_x[m]*T_x[m] +
T_y[m]*T_y[m] + T_z[m]*T_z[m]);


T_star[ii] = sqrt(T_x[ii]*T_x[ii] + T_y[ii]*T_y[ii] +
T_z[ii]*T_z[ii])*Fn[ii] / fn[ii] ; T_star[m] = sqrt(T_x[m]*T_x[m]
+ T_y[m]*T_y[m] + T_z[m]*T_z[m])*Fn[m] / fn[m] ;

if(mod_proj_T_old[ii] > mod_proj_T[ii])

Kt[ii] = Ko*(1 - (mod_proj_T[ii] - T_star[ii])/(mu*Fn[ii] -
T_star[ii]));

else

Kt[ii] = Ko*(1 - (T_star[ii] - mod_proj_T[ii])/(mu*Fn[ii] +
T_star[ii]));

if(mod_proj_T_old[m] > mod_proj_T[m])
```

```
Kt[m] = Ko*(1 - (mod_proj_T[m] - T_star[m])/(mu*Fn[m] -
T_star[m]));

else

Kt[m] = Ko*(1 - (T_star[m] - mod_proj_T[m])/(mu*Fn[m] +
T_star[m]));


del_x = X[ii] - X[m]; del_y = Y[ii] - Y[m]; del_z = Z[ii] - Z[m];

del_x_old = x0[ii] - x0[m]; del_y_old = y0[ii] - y0[m]; del_z_old
= z0[ii] - z0[m];

w[0][ii] = w_old[0][ii] + T_tot[0][ii]*dt/I; w[1][ii] =
w_old[1][ii] + T_tot[1][ii]*dt/I; w[2][ii] = w_old[2][ii] +
T_tot[2][ii]*dt/I;

w[0][m] = w_old[0][m] + T_tot[0][m]*dt/I; w[1][m] = w_old[1][m] +
T_tot[1][m]*dt/I; w[2][m] = w_old[2][m] + T_tot[2][m]*dt/I;

cross_prod(prod_x,prod_y, prod_z,w[0][ii],w[1][ii],w[2][ii],k_n);
cross_prod(prod_x2,prod_y2,prod_z2,w[0][m],w[1][m],w[2][m],k_n);

prod_x = prod_y = prod_z = 1.0; prod_x2 = prod_y2 = prod_z2 =
1.0;



del_s_x = del_x - del_x_old - k_n1*k_n1*(del_x - del_x_old) +
(X[ii]*prod_x - X[m]*prod_x2)*dt;

del_s_y = del_y - del_y_old - k_n2*k_n2*(del_y - del_y_old); +
(Y[ii]*prod_y + Y[m]*prod_y2)*dt;

del_s_z = del_z - del_z_old - k_n3*k_n3*(del_z - del_z_old) +
(Z[ii]*prod_z + Z[m]*prod_z2)*dt;

t_x[ii] = T_x[ii]/mod_proj_T[ii]; t_y[ii] =
T_y[ii]/mod_proj_T[ii]; t_z[ii] = T_z[ii]/mod_proj_T[ii];

t_x[m] = T_x[m]/mod_proj_T[m]; t_y[m] = T_y[m]/mod_proj_T[m];
t_z[m] = T_z[m]/mod_proj_T[m];


del_sp_x[ii] = del_s_x*t_x[ii]*t_x[ii]; del_sp_y[ii] =
del_s_y*t_y[ii]*t_y[ii]; del_sp_z[ii] = del_s_z*t_z[ii]*t_z[ii];

del_sp_x[m] = del_s_x*t_x[m]*t_x[m]; del_sp_y[m] =
del_s_y*t_y[m]*t_y[m]; del_sp_z[m] = del_s_z*t_z[m]*t_z[m];


del_sn_x[ii] = del_s_x - del_sp_x[ii]; del_sn_x[ii] = del_s_x -
del_sp_y[ii]; del_sn_z[ii] = del_s_z - del_sp_z[ii];
```

```
del_sn_x[m] = del_s_x - del_sp_x[m]; del_sn_x[m] = del_s_x -
del_sp_y[m]; del_sn_z[m] = del_s_z - del_sp_z[m];


Tp_x[ii] = T_x[ii] + Kt[ii] * del_sp_x[ii]; Tp_y[ii] = T_y[ii] +
Kt[ii] * del_sp_y[ii]; Tp_z[ii] = T_z[ii] + Kt[ii] *
del_sp_z[ii];

Tp_x[m] = T_x[m] + Kt[m] * del_sp_x[m]; Tp_y[m] = T_y[m] + Kt[m]
* del_sp_y[m]; Tp_z[m] = T_z[m] + Kt[m] * del_sp_z[m];


Tn_x[ii] = Ko * del_sn_x[ii]; Tn_x[ii] = Ko * del_sn_y[ii];
Tn_z[ii] = Ko * del_sn_z[ii];

Tn_x[m] = Ko * del_sn_x[m]; Tn_x[m] = Ko * del_sn_y[m]; Tn_z[m] =
Ko * del_sn_z[m];


T_tot[0][ii] = Tp_x[ii] + Tn_x[ii]; T_tot[1][ii] = Tp_y[ii] +
Tn_y[ii]; T_tot[2][ii] = Tp_z[ii] + Tn_z[ii];

T_tot[0][m] = Tp_x[m] + Tn_x[m]; T_tot[1][m] = Tp_y[m] + Tn_y[m];
T_tot[2][m] = Tp_z[m] + Tn_z[m];


T_old_x[ii] = T_x[ii]; T_old_y[ii] = T_y[ii]; T_old_z[ii] =
T_z[ii];

T_old_x[m] = T_x[m]; T_old_y[m] = T_y[m]; T_old_z[m] = T_z[m];

mod_proj_T_old[ii] = mod_proj_T[ii]; mod_proj_T_old[m] =
mod_proj_T[m];

} }

if (m!=-1) listing(m,ii); // Listing is called for searching
more than one particles in the cell else cout<<"No other
particle"<<endl; }

}

}

}



}

// Increment time, velocities and positions

void myclass ::march() { double theta1[num]; double alpha1[num];
double X_a[num],Y_a[num],Z_a[num]; double nearrad[num]; double
v_x_loc[num],v_y_loc[num],v_z_loc[num]; double
F_x_loc[num],F_y_loc[num],F_z_loc[num];
```

```
/*************************** OPEN FILES FOR OUTPUT
*************************/

ofstream fout; fout.open("PositionDrop1"); ofstream alph;
alph.open("Alpha"); ofstream anout; anout.open("Angle");

/********************** The folowing loop will do for 10
timestep ******************/

for(int ndt = 0; ndt < 100; ndt++){

/****************************** Following loop is for adjusting
and finding Alpha1 *****/

for(int j = 0; j < num; j++){

double radiant; alpha1[j]= atan2(Y[j],X[j]); double
rr=sqrt(Y[j]*Y[j]+X[j]*X[j]);

// Periodic BC between the angles

if((alpha1[j] > PI/6) || (alpha1[j] < 0)){ alpha1[j] = alpha1[j]
+ PI/6; double bhagsesh = fmod(alpha1[j],PI/6); alpha1[j] =
bhagsesh;

alph<<j<<" "<<alpha1[j]<<endl;

X[j] = cos(alpha1[j])*rr; Y[j] = sin(alpha1[j])*rr; } else
alph<<j <<" "<<alpha1[j]<<endl;

}

// Compute the normal distance from the curved wall of the hopper

for(int k = 0; k < num; k++){ X_a[k]
=(ht-Z[k])*tan(theta0*PI/180)/(sqrt(1 +
tan(alpha1[k]*PI/180)*tan( alpha1[k]*PI/180)));

Y_a[k] =(ht-Z[k])*tan(theta0*PI/180)*tan(alpha1[k]*PI/180)
/(sqrt(1 + tan(alpha1[k]*PI/180)*tan (alpha1[k]*PI/180)));

Z_a[k] = (ht-Z[k]);


nearrad[k] = sqrt(X_a[k]*X_a[k] + Y_a[k]*Y_a[k]);


PA[k] = ( (ht - Z[k]) * tan(theta0*PI/180) - sqrt(X[k]*X[k] +
Y[k] * Y[k]));

PB[k] = PA[k] * cos(theta0*PI/180);



if(PB[k] < rad[k]){
```

```
BP_x[k] = (cos(theta0*PI/180)) * (cos(alpha1[k]*PI/180)); BP_y[k]
= (cos(theta0*PI/180))* (sin(alpha1[k]*PI/180)); BP_z[k] =
-sin(theta0*PI/180);

Fn_x[k] = -K*(rad[k] - PB[k])*BP_x[k]; Fn_y[k] = -K*(rad[k] -
PB[k])*BP_y[k]; Fn_z[k] = -K*(rad[k] - PB[k])*BP_z[k];

F_x_loc[k] = F[0][k] + Fn_x[k]; F_y_loc[k] = F[1][k] + Fn_y[k];
F_z_loc[k] = F[2][k] + Fn_z[k];

v_x[k] += F_x_loc[k]*dt/mass[k]; v_y[k] += F_y_loc[k]*dt/mass[k];
v_z[k] += F_z_loc[k]*dt/mass[k];

X[k] = X[k] + v_x[k] * dt; Y[k] = Y[k] + v_y[k] * dt; Z[k] = Z[k]
+ v_z[k]* dt;


/******** alpha should be checked here
*************************/

double radiant1; alpha1[k]= atan2(Y[k],X[k]); double
rr1=sqrt(Y[k]*Y[k]+X[k]*X[k]);

if((alpha1[k] > PI/6) || (alpha1[k] < 0)){ alpha1[k] = alpha1[k]
+ PI/6; double bhagsesh1 = fmod(alpha1[k],PI/6); alpha1[k] =
bhagsesh1;

alph<<k<<" "<<alpha1[k]<<endl;

X[k] = cos(alpha1[k])*rr1; Y[k] = sin(alpha1[k])*rr1; } else
alph<<k <<" "<<alpha1[k]<<endl;

fout <<(k+1)<<" "<<X[k]<<" "<<Y[k]<<" "<<Z[k]<<endl;

//Particles falling from the bottom are put back on the top // to
be changed if((Z[k] > (4.00-rad[k])) || (Z[k] == (4.00-rad[k]))){

Fn_z[k] = -K*(4.00 - rad[k] - Z[k]); Fn_x[k] = 0.0; Fn_y[k] =
0.0;

F_x_loc[k] = F[0][k] + Fn_x[k]; F_y_loc[k] = F[1][k] + Fn_y[k];
F_z_loc[k] = F[2][k] + Fn_z[k];

v_x[k] += F_x_loc[k]*dt/mass[k]; v_y[k] += F_y_loc[k]*dt/mass[k];
v_z[k] += F_z_loc[k]*dt/mass[k];

// v_x[k] = 0.50*v_x[k]; // v_y[k] = 0.50*v_y[k]; v_z[k] =
0.50*v_z[k];

X[k] = X[k] + v_x[k] * dt; Y[k] = Y[k] + v_y[k] * dt; Z[k] = Z[k]
+ v_z[k]* dt;


}
```

```
}

else {

v_x[k] += F[0][k]*dt/mass[k]; v_y[k] += F[1][k]*dt/mass[k];
v_z[k] += F[2][k]*dt/mass[k];


X[k] = X[k] + v_x[k]*dt; Y[k] = Y[k] + v_y[k]*dt; Z[k] = Z[k] +
v_z[k]*dt;


/******** alpha should be checked here
*************************/

double radiant1; alpha1[k]= atan2(Y[k],X[k]); double
rr1=sqrt(Y[k]*Y[k]+X[k]*X[k]);

if((alpha1[k] > PI/6) || (alpha1[k] < 0)){ alpha1[k] = alpha1[k]
+ PI/6; double bhagsesh1 = fmod(alpha1[k],PI/6); alpha1[k] =
bhagsesh1;

alph<<k<<" "<<alpha1[k]<<endl;

X[k] = cos(alpha1[k])*rr1; Y[k] = sin(alpha1[k])*rr1; } else
alph<<k <<" "<<alpha1[k]<<endl;

fout <<(k+1)<<" "<<X[k]<<" "<<Y[k]<<" "<<Z[k]<<endl; if((Z[k] >
(4.00 - rad[k])) || (Z[k] == (4.00 - rad[k]))){

Fn_x[k] = 0.0; Fn_y[k] = 0.0; Fn_z[k] = -K*(Z[k] - (4.00 -
rad[k]));

F_x_loc[k] = F[0][k] + Fn_x[k]; F_y_loc[k] = F[1][k] + Fn_y[k];
F_z_loc[k] = F[2][k] + Fn_z[k];

v_x[k] += F_x_loc[k]*dt/mass[k]; v_y[k] += F_y_loc[k]*dt/mass[k];
v_z[k] += F_z_loc[k]*dt/mass[k];

// v_x[k] = 0.50*v_x[k]; // v_y[k] = 0.50*v_y[k]; v_z[k] =
0.50*v_z[k];

X[k] = X[k] + v_x[k] * dt; Y[k] = Y[k] + v_y[k] * dt; Z[k] = Z[k]
+ v_z[k]* dt;



}


}
```

```
cout<<"UPTO HERE"<<k <<" "<<ndt<<endl;

} cout<<"Calling Search"<<endl; search(nearrad); /* for(int in
0; in < num; in++){ for(int in1 = 0; in1 < num; in1++){
R[in][in1] = -1.0; }} */ cout<<"NO of loop"<<" "<<ndt<<endl;

} alph.close(); fout.close(); }

main() { char *datafile = "Database1";

cout<<"Input the data file name\n"; // cin >> datafile;

myclass object;

object.init(datafile); object.march();

return 0; }
```

APPENDIX B

MAIN PARTS OF THE OPENGL CODE FOR VISUALIZATION

```
#include <GL/gl.h> #include <GL/glu.h> #include <math.h> #include
<iostream.h> #include <fstream.h> #include "aux.h" #define PI
3.14159265 #define m 24 #define mm 21600/24

float a[m][mm], b[m][mm], c[m][mm]; float r[m];

/* float aa[m] =
{0.9848,0.9396,1.9696,1.8792,2.9544,2.8188,3.9392,3.7584 }; float
bb[m] = { 0.1736,0.342,0.3472,0.684,0.5208,1.02,0.6944,1.368};
float cc[m] = {0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0}; */ float other =
0; float zoom = 0.0; void zoomin() { zoom = zoom +1.0; } void
zoomout() { zoom = zoom - 1.0; }


void mov_ot(){ other = other - 5; } void mov_ott() { other =
  other + 5; }


int move = 0;



void addpos() { move = move + 1; cout<<"x_pos:"<<a[6][move]<<"
y_pos:"<<b[6][move]<<" z_pos:"<<c[6][move]<<endl; } void subpos()
{ move = move - 1; }


int j = 0;

float angle = 0; void addangle() { angle = angle +5; } void
subangle() { angle = angle - 5; }

float rotate = 0; void rotadd() { rotate = rotate + 5; } void
rotsub() { rotate = rotate - 5; }

void light() {

GLfloat ambient[] = { 0.0, 0.0, 0.0, 1.0 }; GLfloat diffuse[] = {
1.0, 0.0, 1.0, 1.0 }; //GLfloat specular[] = { 1.0, 1.0, 1.0, 1.0
}; GLfloat position[] = { 0.0, 3.0, 2.0, 0.0 }; GLfloat
lmodel_ambient[] = { 0.4, 0.4, 0.4, 1.0 }; GLfloat local_view[] =
{ 0.0 };

glEnable(GL_DEPTH_TEST); glDepthFunc(GL_LESS);

glLightfv(GL_LIGHT0, GL_AMBIENT, ambient); glLightfv(GL_LIGHT0,
GL_DIFFUSE, diffuse); glLightfv(GL_LIGHT0, GL_POSITION,
position); glLightModelfv(GL_LIGHT_MODEL_AMBIENT,
lmodel_ambient); glLightModelfv(GL_LIGHT_MODEL_LOCAL_VIEWER,
local_view);

glEnable(GL_LIGHTING); glEnable(GL_LIGHT0);

glClearColor(0.0, 0.0, 0.0, 0.0);

}
```

```
void datain() { extern float a[m][mm], b[m][mm], c[m][mm], r[m];
int num; int l,i,n,time; ifstream fin; fin.open("PositionData");

n = 8; time = 0; while(!fin.eof()){ for( i = 0; i < m; i++){ l =
time; fin>>num>>a[i][l]>>b[i][l]>>c[i][l]; r[i] = 0.15;
cout<<a[i][l]<<" "<<b[i][l]<<" "<<c[i][l]<<endl; } time = time +
1; cout<<time<<endl; } fin.close();

}

/*void myinit() { glShadeModel(GL_FLAT); } */ void myreshape(int
w,int h) { glViewport(0,0,w,h); glMatrixMode(GL_PROJECTION);
glLoadIdentity();

/* if(w < h){
     glOrtho(-15.0,15.0,-15.0*(GLfloat)h/(GLfloat)w,15.0*(GLfloat)h/
     } else
     glOrtho(-15.0*(GLfloat)h/(GLfloat)w,15.0*(GLfloat)w/(GLfloat)h,
     */

gluPerspective(60, 1.0*(GLfloat)w/(GLfloat)h, 0.0,50.0);
glMatrixMode(GL_MODELVIEW); glLoadIdentity(); }

void display() { int k;

GLfloat no_mat[] = { 0.0, 0.0, 0.0, 1.0 }; GLfloat
mat_diffuse_mat[] = { 0.8, 0.0, 0.0, 1.0 };

// GLfloat mat_ambient[] = { 0.7, 0.7, 0.7, 1.0 }; // GLfloat
 mat_ambient_color[] = { 0.8, 0.8, 0.2, 1.0 }; // GLfloat
 mat_specular[] = { 1.0, 1.0, 1.0, 1.0 }; GLfloat mat_diffuse[] =
 { 0.1, 0.4, 0.7, 1.0 }; GLfloat no_shininess[] = { 0.0 }; //
 GLfloat low_shininess[] = { 5.0 }; //GLfloat high_shininess[] =
 { 90.0 }; GLfloat mat_emission_back[] = {0.45, 0.45, 0.45, 0.0};
 GLfloat mat_emission_side1[] = {0.2, 0.2, 0.2, 0.2}; GLfloat
 mat_emission_side2[] = {0.15, 0.15, 0.15, 0.0}; GLfloat
 mat_emission_edge[] = {0.15, 0.15, 0.45, 0.0};


glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, no_mat);
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, mat_diffuse);
glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, no_mat);
glMaterialfv(GL_FRONT_AND_BACK, GL_SHININESS, no_shininess);
glMaterialfv(GL_FRONT_AND_BACK, GL_EMISSION, mat_emission_back);

glPushMatrix(); glTranslatef(0.0,0.0,-29.0 + zoom); glPushMatrix
(); glTranslatef (0.0, 0.0, -39.0); glRectf(-10.0, -10.0, 10.0,
10.0); glPopMatrix ();


glPushMatrix (); glTranslatef (0.0, 0.0, -29.0);
glMaterialfv(GL_FRONT_AND_BACK, GL_EMISSION, mat_emission_side1);
glPushMatrix (); glBegin(GL_QUAD_STRIP);
glVertex3f(10.0,-10.0,10.0); glVertex3f(10.0,-10.0,-10.0);
glVertex3f(-10.0,-10.0,10.0); glVertex3f(-10.0,-10.0,-10.0);
glVertex3f(-10.0,10.0,10.0); glVertex3f(-10.0,10.0,-10.0);
glEnd(); glPopMatrix ();

glPushMatrix (); glMaterialfv(GL_FRONT_AND_BACK, GL_EMISSION,
mat_emission_side2); glBegin(GL_QUAD_STRIP);
glVertex3f(10.0,-10.0,-10.0); glVertex3f(10.0,-10.0,10.0);
glVertex3f(10.0,10.0,-10.0); glVertex3f(10.0,10.0,10.0);
glVertex3f(-10.0,10.0,-10.0); glVertex3f(-10.0,10.0,10.0);
glEnd(); glPopMatrix ();

glMaterialfv(GL_FRONT_AND_BACK, GL_EMISSION, mat_emission_edge);
glPushMatrix (); glLineWidth(4.5); glBegin(GL_LINE_LOOP);
glVertex3f(10.0,-10.0,10.0); glVertex3f(10.0,-10.0,-10.0);
glVertex3f(-10.0,-10.0,-10.0); glVertex3f(-10.0,-10.0,10.0);
glEnd(); glBegin(GL_LINE_STRIP); glVertex3f(-10.0,-10.0,-10.0);
glVertex3f(-10.0,10.0,-10.0); glVertex3f(-10.0,10.0,10.0);
glVertex3f(-10.0,-10.0,10.0); glEnd(); glBegin(GL_LINE_LOOP);
glVertex3f(10.0,-10.0,10.0); glVertex3f(10.0,-10.0,-10.0);
glVertex3f(10.0,10.0,-10.0); glVertex3f(10.0,10.0,10.0); glEnd();
glBegin(GL_LINE_STRIP); glVertex3f(10.0,10.0,-10.0);
glVertex3f(-10.0,10.0,-10.0); glVertex3f(-10.0,10.0,10.0);
glVertex3f(10.0,10.0,10.0); glEnd(); glPopMatrix (); /* drawing
of walls complete */ glPopMatrix ();

// glPopMatrix(); for(k = 0; k < m; k++){ glPushMatrix();
        glTranslatef(0.0,0.0,-29.0);
        glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE,
        mat_diffuse_mat); //
        glTranslatef(a[k][move],b[k][move],c[k][move]);
        glTranslatef( a[k][move],-c[k][move],b[k][move]);
        auxSolidSphere(r[j]); glPopMatrix(); } glPopMatrix();

glFlush();

// glXSwapBuffers(auxXDisplay(),auxXWindow());

}

int main(int argc , char**argv) { auxInitDisplayMode (AUX_SINGLE
| AUX_RGBA); auxInitPosition (0, 0, 800, 800);
auxInitWindow(argv[0]); light(); // myinit(); datain();
auxKeyFunc(AUX_LEFT,addangle); auxKeyFunc(AUX_RIGHT,subangle);
auxKeyFunc(AUX_UP,addpos); auxKeyFunc(AUX_DOWN,subpos);
auxKeyFunc(AUX_r,mov_ot); auxReshapeFunc ( myreshape );
auxKeyFunc(AUX_k,zoomin); auxKeyFunc(AUX_l,zoomout);
auxMainLoop(display); return 0; }
```

REFERENCES

[1] Agrawal, J. H., "Use of Multiple Transmitters for 3-D Non-Intrusive Particle Tracking", M.S. Thesis, New Jersey Institute of Technology, 1995.

[2] Ashok, A. S., "Computational Aspects of a Three Dimensional Non-Intrusive Particle Motion Tracking System", M.S. Thesis, New Jersey Institute of Technology, Newark, New Jersey, 1992.

[3] Dave, R. N., Private Communications, New Jersey Institute of Technology, 1997.

[4] Dave, R. N., A. S. Ashok, and B. G. Bukiet, "On Development of a Three Dimensional Particle Motion Tracking System", *ASME Paper,* Winter Annual Meeting 1992.

[5] Dave R. N., and B. G. Bukiet, "Non-Intrusive Rigid Body Tracking Technique for Dry Particulate Flows, Part I: Theoretical Aspects," Submitted to *Measurement Science and Technology (Formerly Physics E)*, 1994.

[6] Dennis, J. R., R. B. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Prentice-Hall, Engelwood Cliffs, New Jersey, 1983.

[7] Goldstein, H., *Classical Mechanics*, Addison Wesley Publishing Company, 1980.

[8] Moré, J. J., "The Levenberg-Marquardt Algorithm: Implementation and Theory", in *Numerical Analysis*, G. A. Watson, ed., Lecture Notes in Math. 630, Springer-Verlag, Berlin, 1977.

[9] Moré, J. J., B. S. Garbow and K. E. Hillstrom, "User Guide for MINPACK-1", Argonne National Lab Report, 1980.

[10] Parasar, A., Lab Notes, New Jersey Institute of Technology, Newark, New Jersey, 1992.

[11] Savage, S. B., "Flow of Granular Materials with Applications to Geophysical Problems," *Continuum Mechanics in the Environmental Sciences and Geophysics*, IUTAM International Summer School on Mechanics, Udine, Italy, June 22-26, (1992).