Computer Simulations of Pattern Formations in Granular Materials

Brock Wilson
65629966

March 12, 2001

Abstract

Surface patterns form on vertically oscillated granular materials in a vacuum. Two different approaches were developed to simulate these patterns, however neither approach reached this goal. An event driven algorithm was written that did not incorporate oscillations, but can efficiently handle many particles. Clumping was observed in the event driven simulations of inelastic particles free from external forces. A time driven simulation was also developed that included oscillations and gravity. No patterns were found because of the small number of particles simulated. Both techniques show promise for further work on pattern formations in granular materials.

Contents

List of Figures

1 Introduction

Since the study of granular materials is not widespread, an introduction to what they are and how they interact is needed. Granular materials are large collections of discrete macroscopic particles[1]. Some examples are: sand, salt, rice or a collection of marbles. The only inter-particle forces occur when two particles contact each other. If the particles are cohesive, the inter-particle forces can be either attractive or repulsive[1]. If the particles are non-cohesive, the inter-particle forces are purely repulsive[1].

Over the past decade, experiments studying granular materials have become increasingly popular, with most of the interest coming from physicists and engineers. Experiments have shown that granular materials have a number of interesting properties, such as pattern formation[2][3][4][5] and size segregation[1][6]. The knowledge gained from experiments on granular materials has been successfully applied by engineers to fields such as food processing, pharmacology, chemical engineering, industrial processes and geological engineering[7]. In contrast to the success of the engineers, physicists have not yet developed any theories that adequately describe granular materials[1]. A theory that describes granular materials well would have important practical applications, such as those mentioned above.

Bizon et al.[4] have simulated pattern formations in 3D vertically oscillated granular materials. The patterns found in their simulations match the patterns found in experiments[4][3]. In contrast to experiments, computer simulations can study the microscopic and immeasurable effects that lead to pattern formations. For this reason, computer simulations offer insight into the development of a granular theory that experiments can not. The goal of my thesis is to simulate pattern formations in 3D vertically oscillated granular materials, similar to Bizon et al[4].

There are two main approaches to simulating systems of many hard particles: time driven simulations and event driven simulations[8]. In time driven simulations, the system evolves in fixed time steps. In event driven simulations, the system evolves at non constant increments defined by the collision time between particles. Therefore, event driven simulations evolve at times characteristic of the system being simulated.

My initial simulations were based on the event driven algorithm developed by Marin, Risso and Cordero[8], which Bizon et al.[4] used in their simulations of vertically oscillated granular materials in three dimensions. This algorithm is described in section 2, the results from this simulation are given in section 3 and the code for this simulation is given in appendix A.

Due to problems with modeling collisions in the event driven simulation, I also made a simulation based on the time driven method. The time driven simulation is described in section 4, the results are described in section 5 and the code is given in appendix B.

2 Theory and Implementation of the Event Driven Simulation

Event driven simulations are based on two sequential operations.

1. Find the next event that will occur.

2. Update the system to that point in time.

Events are considered to be instantaneous, and are processed in the second operation above. These two operations are repeated as many times as is necessary to complete the simulation.

2.1 The Fast Event Driven Algorithm of Marin, Risso and Cordero

Since simulations of systems of many hard particles are very time intensive, it is desirable to optimize the speed of the simulation. Marin, Risso and Cordero[8], have developed a fast event driven algorithm based on three goals.

1. Minimize the number of events predicted
2. Manage the predicted events efficiently
3. Minimize the number of particle updates per event processed

The following discussion of the event driven algorithm involves a system of P particles in a 2D rectangular domain. Each particle is of the same mass, m, and diameter, D. This discussion is similar to the paper by Marin, Risso and Cordero[8], but hopefully will clarify some of the areas of the paper which lead to errors in the development of my simulation.

The first goal is attained by dividing the original domain up into an array of $N_x \times N_y$ rectangular cells of length L_x and L_y respectively. This technique has long been used to minimize the number of events calculated in event driven simulations[9]. Each particle is associated with a cell and has a group of neighbour particles that are either in the same cell, or one of the closest cells. Events are only calculated between neighbour particles. The event that a particle will cross a cell wall, or hit a domain wall is also computed. In order to predict all events using this technique, $min(L_x, L_y) > D$. If this condition is not met then balls from non-neighbouring cells can suffer collisions with one another.

Now that the domain is divided into cells, three different types of events must be considered: A disk-disk collision (DDC), a disk-wall collision (DWC) and a virtual wall collision (VWC). A DDC occurs when two disks collide. A DWC occurs when a disk collides with a wall. A VWC occurs when the center of a disk passes through a 'virtual wall' to another cell. DDCs and DWCs will be referred to as hard collisions.

The second goal is reached by using local event lists, the local minima list and the future events list. Each event calculated for a particle is stored in it's local event list, $local_i$. The collection of all $local_i$ represents all events that will occur in the near future of the simulation. The event which will occur in the shortest amount of time is picked from each $local_i$, and placed in the local minima list (lml), so lml_i represents the next event that is predicted for the i^{th} disk. The future events list, or FEL, is a structure that helps to efficiently pick the overall minimum event from the lml. The minimization of events predicted and management of predicted events is called the local minima algorithm, or LMA, and will be discussed shortly.

The third goal is reached by only updating the state of a particle after it has experienced a hard collision. Since not all of the disks are updated in each cycle of the simulation, every disk must have a time, τ_i, included with it's state. τ_i is the last time at which the i^{th} disk was updated. If the event processed in a cycle of the algorithm is a VWC, then no disk state updates are performed. That is, the position, velocity and τ of the i^{th} disk are not updated during a VWC. Instead, the disk that is involved in the VWC is moved into a new cell, and events are predicted and added to $local_i$ for it's new neighbours and walls. lml_i is updated after the new events are added to $local_i$. As a result of not updating states during a VWC the round off error incurred from finite precision arithmetic in the simulation is reduced. The technique that minimizes the number of particle updates is called the delayed states algorithm, or DSA.

The two steps in the event driven simulation have now expanded to four steps.

1. Pick the next event using the FEL
2. Update the states of the disks, or update the cell of the disk involved in the event
3. Calculate new events for disks involved in the event
4. Insert the new events into $local$, and update lml

The LMA manages steps 1,3 and 4, while he DSA manages step 2. These steps are described in order below.

2.2 Description of the LMA and DSA

The next event can be efficiently determined if the FEL is implemented as a complete binary tree of fixed size, with P leaves. A binary tree is made up of nodes and leaves, with at most two leaves per node. A complete binary tree has all of it's leaves at the left hand side of the deepest level of the tree[10]. See figure 1 for a picture of a complete binary tree. Since the FEL is a fixed size complete binary tree, it can be implemented as an array of length $2P - 1$. Each particle in the simulation is associated with a leaf of the FEL.

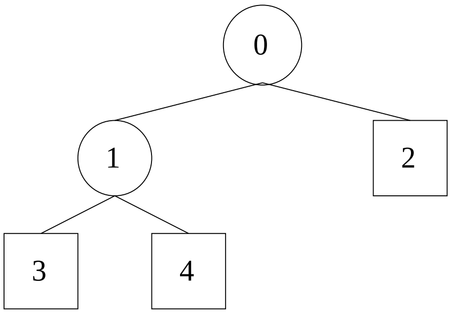

Figure 1: A complete binary tree with three leaves (2-4) and two nodes (0-1). Leaf 3 and 4 are siblings. 0 is the root node.

The next event is determined through a playoff between the local minima of all disks. The algorithm works much like a sports playoff. In the first round, there are P teams. I will consider P to be even, but the argument for P odd is much the same, but with the odd disk in the first round getting passed to the second round with no competition. Two disks from a common node compete, and the disk with the lowest time in the lml is passed onto the next round, so that in the second round there are only $\frac{P}{2}$ disks. The winner of the competition between leaf i and leaf $i + 1$ of the FEL is put into their parent node, $\frac{i-1}{2}$, of the FEL. This process is repeated until a disk is advanced to the 0^{th} node of the FEL. This disk has the globally minimum event. This whole process need not be repeated during each cycle since not more than two particles are updated at a time. The above process is only used to find the first event in the simulation. See figure 2a) for an example of the process described above.Once an event is picked from the FEL, it is removed from $local_i$ and lml_i, so that a new event will be picked next time through the cycle.

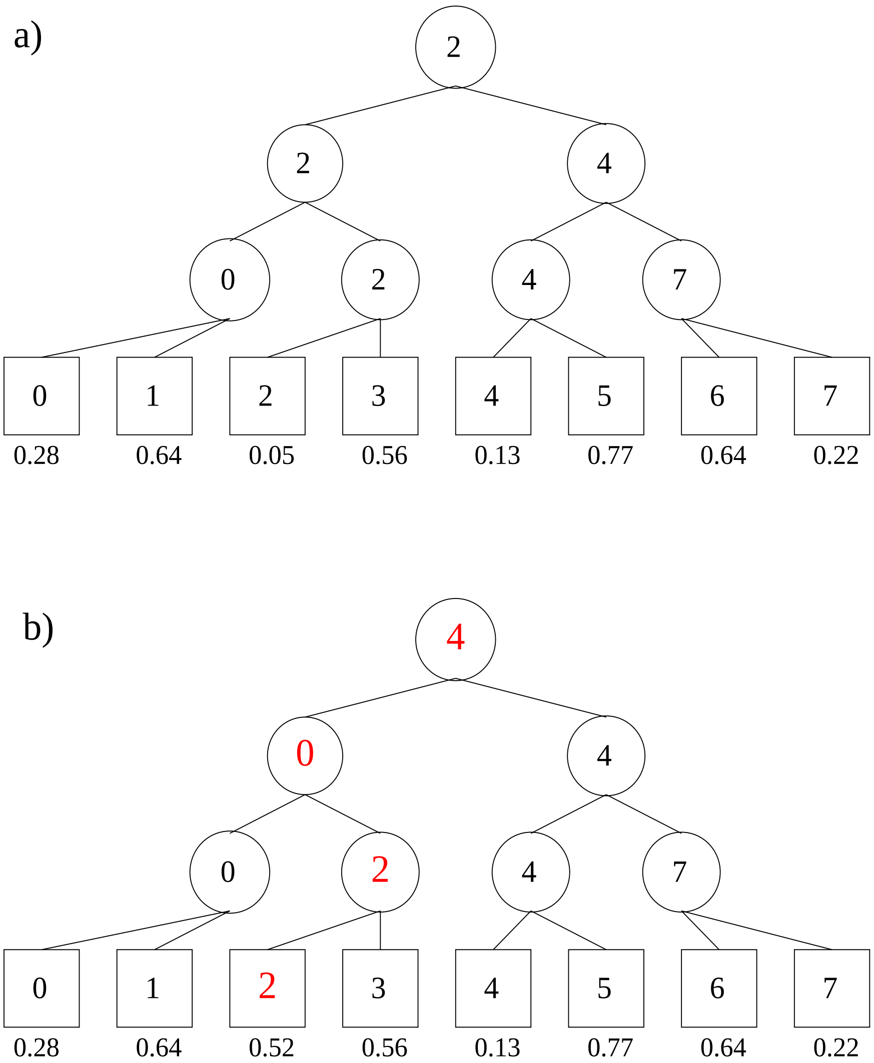

Figure 2: a) An example of finding the first event using the FEL. The numbers in the leaves and nodes represent disk numbers. The numbers below each leaf are the minimum times associated with each leaf. b) An example of finding the next event using the FEL. Disk 2 was the disk involved in the previous event. Disk 4 now has the globally minimum event.

After the first event is processed, the disk or disks involved in the last event will have new events in their local minimum lists. To find the next event, only the disks involved in the last event need to be played off through the FEL. If disk i was involved in the last event, it is compared to it's sibling leaf and the winner is advanced to the parent node. The parent node is then compared to it's sibling node and again the winner is advanced. This process is repeated until the root node is reached. The disk in the root node is the one with the next event. See figure 2b) for an example of this process.

To maximize the execution speed of the algorithm, only the integer label of each disk is entered into the FEL, not the local minimum event for the disk. This makes choosing the next event efficient because only integers are being moved around in the FEL, not the information associated with an event. Local minimum times for disks can be accessed from the lml during the FEL competition.

The next step of the algorithm in the sequence listed above is to update the state or cell of the particle, but before that the validity of the event must be checked. The only events that become invalid are DDCs. If one of the disks involved in the next event has suffered a hard collision since the event was predicted, the event will be invalid. In order to detect invalid events, an integer, c_i, is included in the state of every particle. c_i is the number of hard collisions that the i^{th} disk has suffered. When a DDC is predicted, the c_i of the partner particle is stored as part of the event. If the c_i stored in the event is not equal to c_i at the time the event is going to be processed, the event is invalid. If the next event is not valid, the FEL playoff is repeated until a valid event is chosen.

The complete definition of the state of a particle and an event can now be given. The state of a particle is defined as:

$$S_i = (\vec{r}_i, \vec{v}_i, \tau_i, c_i) \tag{1}$$

$\vec{r}_i$ is the position vector. $\vec{v}_i$ is the velocity vector. τ_i is the time that the particle was last updated. c_i is the number of hard collisions that the disk has suffered. c_i is implemented as a separate array in my program. An event is defined as:

$$\xi = (t, colltype, partner, c) \tag{2}$$

t is the time at which the event will occur. $colltype$ is the type of collision (DDC, DWC or VWC). $partner$ is the disk, wall or virtual wall that the disk will encounter. If $colltype = DDC$, then c is the $c_{partner}$ value at the time the event was predicted.

Once a valid event is chosen, the particle state or cell is updated according to the DSA. If the event predicted is a hard collision, the state of each disk involved in the collision is updated. The local list and local minima list of each disk involved in a hard collision is cleared, since all previously predicted events are invalid after the collision. If the event predicted is a VWC then the cell of the disk is updated.

After the event is processed, new events must be predicted. If the last event was a hard collision, events are predicted for every disk and wall in the neighbourhood of the disks involved in the hard collision. If the last event was a VWC, events are predicted only for new neighbours of the disk involved in the event. DWCs and VWCs are easy to predict because the walls are stationary. The time that a disk will hit a wall is given by the distance to the wall divided by the velocity of the particle normal to the wall. DDCs are more complicated, because the time that two disks hit each other must be predicted. I use a formula, given Lubachevsky[11], that computes if and when two balls will collide. Once new events are predicted, the local minima list of each disk involved in the last event is updated.

This process described above is repeated until the end of the simulation.

2.3 Pseudo-code for the Event Driven Simulation

The process described in section 2.2 is implemented in the following pseudo-code. The actual C code of the event driven simulation is given in appendix A. In addition to the functions described below, the following support functions are also needed:

- **freev(*udisk*, *dt*):** This routine updates the state of the disk, *udisk*, to the time $\tau_{udisk} + dt$, assuming that it does not suffer any hard collisions in this time period.
- **diskev(*udisk*,** *partner*): This routine models the collision of the two disks, *udisk* and *partner*, and updates their positions and velocities accordingly.
- **updateM(*udisk*, *partner*):** This routine updates the cell of the disk, *udisk*. *partner* is the label of one of the virtual walls in the system, and it is an argument to this routine so that the new cell of *udisk* can be determined.
- **event_clear(*local$_i$*):** This routine clears all events from $local_i$.
- **neighbourhood(*neighbours*, *udisk*, *colltype*):** The singly linked list, *neighbours*, is filled with all of the neighbour disks of *udisk*. *colltype* is an argument to this function so that if the last event was a VWC, only the new neighbours of *udisk* are entered into *neighbours*
- ***time* = tball(*udisk*, *partner*):** This routine returns the next time that *udisk* and *partner* will collide, considering their current states. If the two disks will never collide, ∞ is returned.
- **schedual(*udisk*,** *time*, *colltype*, *partner*, $\boldsymbol{c_{partner}}$): This routine enters $\xi = (time, colltype, partner, c_{partner})$ into $local_{udisk}$.
- ***time* = twall(*udisk*, &*colltype*, &*partner*):** This routine returns time that *udisk* will next encounter a wall. The addresses of *colltype* and *partner* are passed into the function so that their value can be changed by the function. *partner* is changed to the wall number that the *udisk* will collide with and *colltype* indicates whether *partner* is a hard or a virtual wall.
- **ξ = local_min(*local$_i$*):** This routine returns the event with the minimum time in $local_i$.
- **update_cbt(*udisk*):** This routine executes the FEL competition algorithm, starting at the leaf associated with *udisk*. See section 2.2 and figure 2b).
- **event_remove(*local$_{disk}$*, *rtime*):** This routine removes the event with time, *time*, from $local_{disk}$.

```
begin MAIN FUNCTION
    initialize all positions and velocities randomly
    initialize all τ_i to zero
    initialize all c_i to zero
    build the cell matrix
    build the local event lists
    build the local minima lists
    enter the disk identifiers into the FEL
    LOOP
        time = next_event(&colltype, &disk, &partner)
        SWITCH(colltype)
            CASE DDC:
                freev(disk,time - τ_disk)
                freev(partner, time - τ_partner)
```

```
                diskev(disk, partner)
                predictions(disk, colltype)
                predictions(partner, colltype)
                BREAK
            CASE DWC:
                freev(disk, time - τ_disk)
                wallev(disk, partner)
                predictions(disk, colltype)
                BREAK
            CASE VWC:
                updateM(disk, partner)
                predictions(disk, colltype)
                BREAK
        end SWITCH
    end LOOP
end MAINFUNCTION

begin PREDICTIONS(disk, colltype)
    IF(colltype != VWC)
        event_clear(local_disk)
    end IF
    neighbourhood(neighbours, disk, colltype)
    FOR(i=1 to length(neighbours))
        time = tball(disk, neighbours_i)
        IF(time < ∞)
            schedual(disk, time, DDC, neighbours_i, C_neighbours_i)
        end IF
    end FOR
    time = twall(disk, &colltype, &partner)
    schedual(disk, time, colltype, partner)
    lml_disk = local_min(local_disk)
end PREDICTIONS

begin NEXT_EVENT(&colltype, &disk, &partner)
    go = true
    update_cbt(disk)
    IF(colltype == DDC)
        updat_cbt(partner)
    end IF
    DO
        disk = FEL_0
        partner = lml_disk.partner
        colltype = lml_disk.colltype
        time = lml_disk.time
        IF((colltype == DDC) and (lml_disk.c != C_partner))
            update_cbt(disk)
```

```
        ELSE IF(colltype != VWC)
            event_clear(local_disk)
            C_disk++
            IF(colltype == DDC)
                event_clear(partner)
            end IF
            go = false
        ELSE
            event_remove(local_disk, time)
            go = false
        end IF
    WHILE( go )
    IF(colltype == DDC)
        C_partner++
    end IF
    RETURN time
end NEXT_EVENT
```

2.4 Collision Modelling in the Event Driven Simulation

The model of elastic collisions in the event driven simulation is taken from the paper by Lubachevsky[11]. The final states of two disks, i and j, involved in the collision are given by:

$$\vec{v}_{inew} = \vec{v}_{jnorm} + \vec{v}_{itang} \tag{3}$$

$$\vec{v}_{jnew} = \vec{v}_{inorm} + \vec{v}_{jtang} \tag{4}$$

Let $\hat{r}$ be the unit vector that points from disk i to j, then $\vec{v}_{inorm}$ and $\vec{v}_{jnorm}$ are the components of the velocities, prior to the collision, parallel to $\hat{r}$. $\vec{v}_{itang}$ and $\vec{v}_{jtang}$ are the components of the velocity, prior to the collision, that are perpendicular to $\hat{r}$. This collision model conserves both energy and momentum. See figure 3 for a picture of the collision model.

Inelastic collisions use the elastic collision model with the outgoing normal velocities multiplied by a dissipative term, μ. The final velocities of two disks suffering an inelastic collision are:

$$\vec{v}_{inew} = \mu\vec{v}_{jnorm} + \vec{v}_{itang} \tag{5}$$

$$\vec{v}_{jnew} = \mu\vec{v}_{inorm} + \vec{v}_{jtang} \tag{6}$$

Momentum is conserved with this model, but energy is not. In my simulation, $\mu < 1$, and is constant.

Unfortunately this inelastic collision model is not very good, because some particles undergo an infinite number of collisions in a finite amount of time[4]. In their simulations of 3D vertically vibrated granular materials, Bizon et al.[4] implement a μ term that is non-constant, and approaches one, below a relative normal cutoff velocity. Above the cutoff velocity, $\mu < 1$ and is constant. They also add a collision operator that models the angular momentum of the particles[4]. I had difficulties implementing the collision operator that is used by Bizon et al.[4], so I decided to develop a time driven approach to modeling the system. See section 4 for a discussion of the time driven simulation.

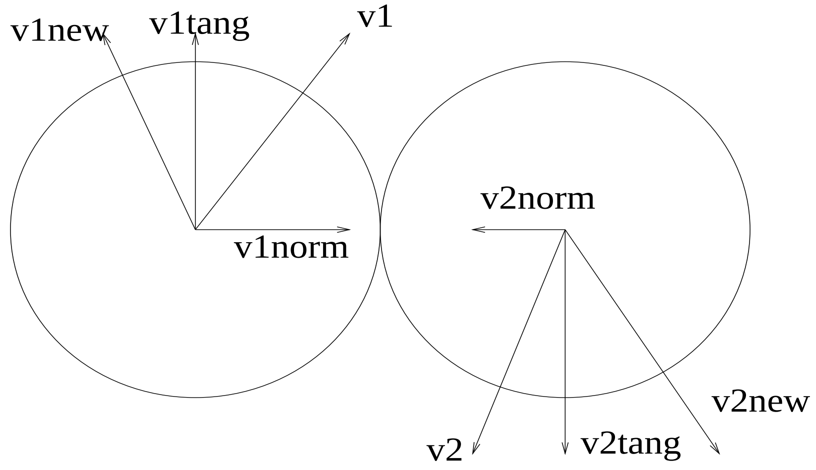

Figure 3: Elastic collision model in the event driven simulation.

3 Results From the Event Driven Simulation

Although I did not reach my goal of simulating 3D vertically vibrated granular materials, the event driven simulation that I developed can efficiently simulate systems of hard particles. I have run 2D simulations of up to 30,000 particle on a Pentium PIII 450MHz processor with 256 Megabytes of RAM. Bizon et al.[4] simulated 3D systems of 30,000 and 60,000 particles, so I believe that my simulation, with the appropriate oscillations and collision operator, is capable of recreating their results. Figure 4 shows the time taken for a system of different number of particles to be simulated to a certain time. I have not extensively investigated the execution time of my simulation, but figure 4 indicates that execution time scales well with the number of particles.

I ran simulations of inelastic particles free from external potentials, and observe that the particles clump together and form strings. The initial positions of the particles in these simulations were randomly distributed with $x \in \{0, 10\}$ and $y \in \{0, 10\}$. The initial velocities were randomly distributed with $v_x \in \{-1, 1\}$ and $v_y \in \{-1, 1\}$. The magnitude of the inelasticity, μ in equations 5 and 6 was 0.4 in these simulations. Simulations were done for 1000 and 2000 particles. A picture of the initial positions the 1000 particle simulation is given in figure 5 a). A picture of the initial positions for the 2000 particle simulation is given in figure 5 b). There is no major clumps or strings of particles in the initial configurations.

After 10 units of system time, the simulations were stopped. Simulations of elastic collisions were also made with the same initial conditions to compare to the inelastic collisions. The results of the 1000 particle inelastic simulation is shown in figure 6 a), and the 1000 particle elastic simulation results are shown in figure 7 a). The results of the 2000 particle inelastic simulation are shown in figure 6 b), and the results of the 2000 particle elastic simulation are shown in figure 7 b). Notice that there is distinct clumping in figure 6 a) and b), whereas there does not appear to be much clumping in figure 7 a) and b). The inelastic collisions that I have modeled seem to cause particles to clump together. I have not investigated this thoroughly, but it would be interesting to study the effects of density, number of particles and the magnitude of the μ term in equations 5 and 6 on the clumping.

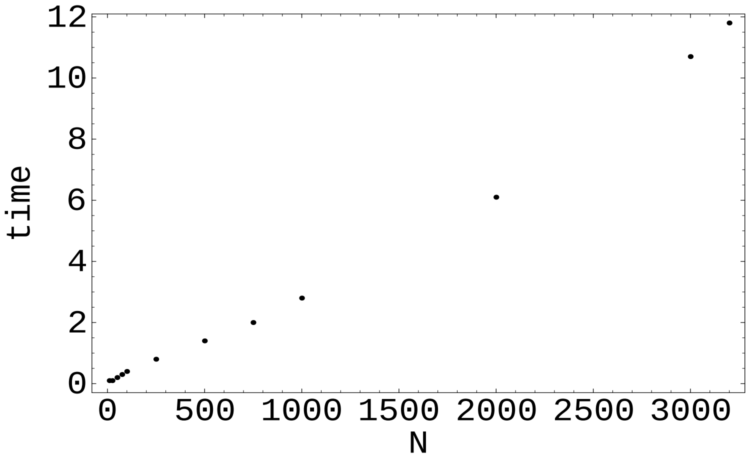

Figure 4: The time in seconds for the event driven simulation to reach a system time of 10, for different numbers of particles. 10 units of system time corresponded to approximately 300 iterations of the main loop. The parameters of this simulation were: $D = 0.1$, $N_X CELLS = N_Y CELLS = 10$. The domain of these simulations was: $x \in \{0, 10\}$, $y \in \{0, 10\}$

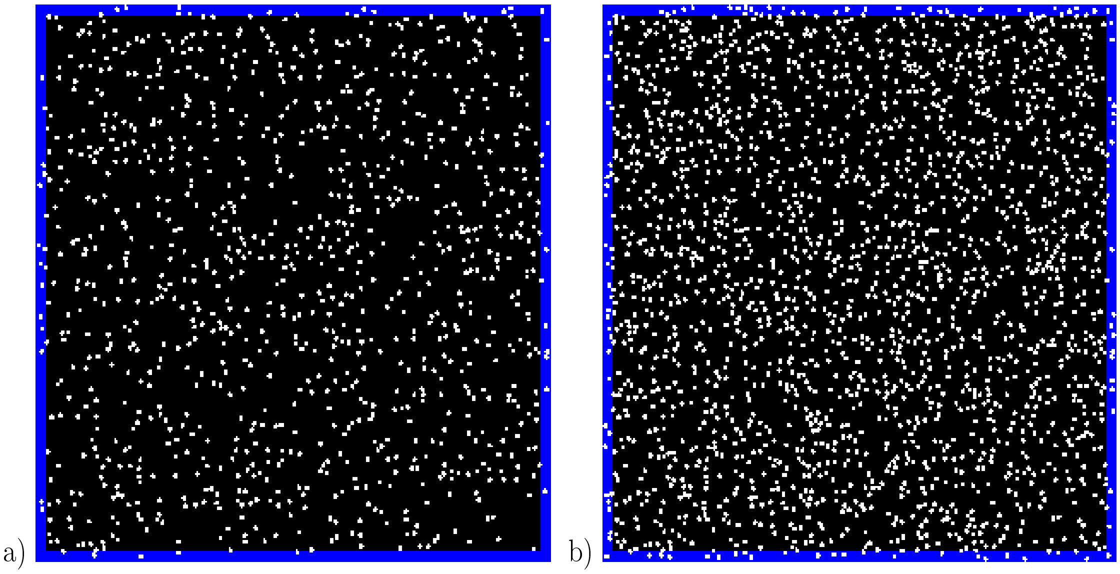

Figure 5: a) Initial conditions for the N=1000 particle simulation. b) Initial conditions for the N=2000 particle simulation.

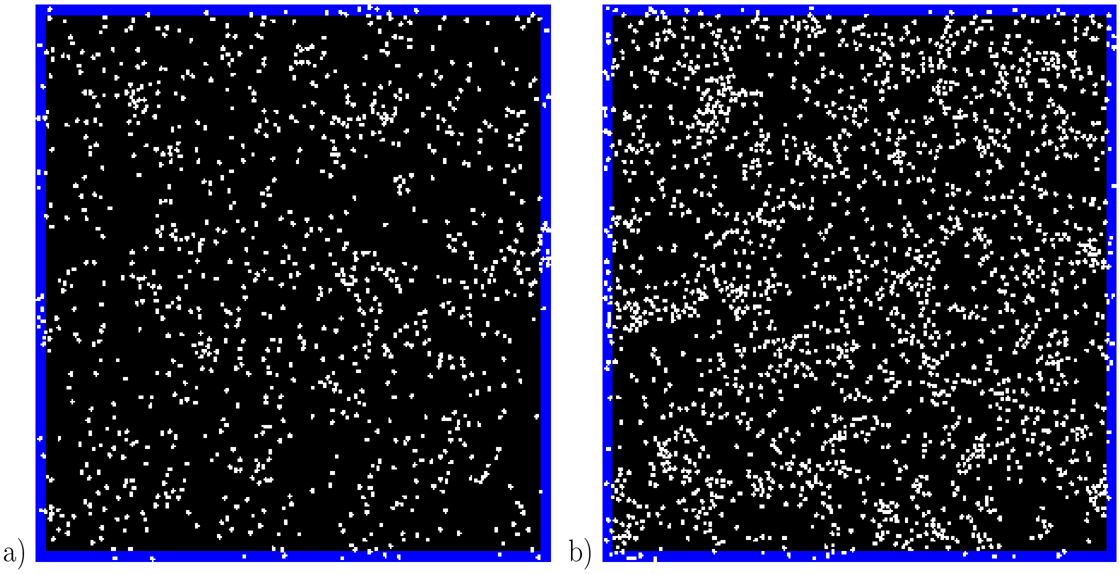

Figure 6: a) Final positions of the inelastic, N=1000 particle simulation. b) Final positions of the inelastic, N=2000 particle simulation.

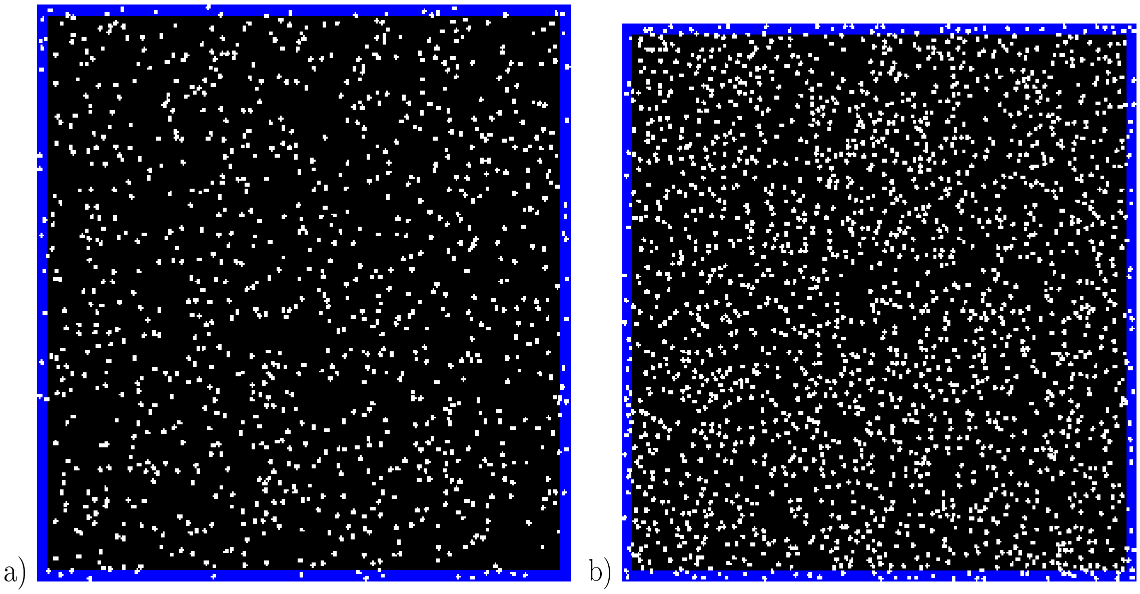

Figure 7: a) Final positions of the elastic, N=1000 particle simulation. b) Final positions of the elastic, N=2000 particle simulation.

Oscillations were not added to the simulation because it is relatively hard to calculate the horizontal wall collision times. Originally I thought that it would be easiest to implement oscillations by keeping the box still and oscillating the particles as if they were charged particles being oscillated by an electric field. This would mean that in order to calculate any horizontal wall(virtual or real) collision time, a nonlinear equation would have to be solved. This would slow the simulation down and add complexity to the implementation of the simulation. An easier way to simulate oscillations is to just oscillate the bottom wall. The bottom hard wall would not be restricted to lie at $y = 0$. Instead the y coordinate of the bottom wall would oscillate up and down in time. This would mean that after each hard collision, the time to hit the bottom wall would need to be calculated no matter what cell the disk is in. Instead of having to solve nonlinear equations for each horizontal wall, only one nonlinear equation would have to be solved after each hard collision to see when the next bottom wall collision would be. Oscillations implemented in this way would not decrease the execution speed of the simulation much, and would be relatively easy to include in the simulation. Adding gravity to the simulation could be easily implemented.

I have not verified numerically that momentum and energy are conserved in the elastic simulations. I have not verified numerically that momentum is conserved in the inelastic simulations.

4 Theory and Implementation of the Time Driven Simulation

Time driven simulations numerically integrate the equations of motion of a system. The equations of motion for my simulation are explained in section 4.1, and the method used for integrating them is described in section 4.2.

4.1 Equations of motion for the Time Driven Simulation

The equations of motion for a system are given by Newton's second law, $\vec{F} = m\vec{a} = m\frac{\partial^2 \vec{r}}{\partial t^2}$. For hard particle simulations, $\vec{a}$ represents the acceleration of each particle in the system. In order to numerically integrate this equation to find $\vec{r}$, this second order equation for $\vec{r}$ must be reduced to two first order equations:

$$\frac{\partial \vec{r}}{\partial t} = \vec{v} \tag{7}$$

$$\frac{\partial \vec{v}}{\partial t} = \frac{\vec{F}}{m} \tag{8}$$

The force acting on each particle,$\vec{F}$, is the sum of all inter-particle forces, plus any external forces. If the potential of two particles is $V_{ij}(\vec{r}_{ij})$, then the inter-particle force is given by:

$$\vec{f}_{ij} = -\frac{\partial V_{ij}}{\partial \vec{r}_{ij}} \tag{9}$$

where $\vec{r}_{ij}$ is the distance between the i^{th} and j^{th} particles. Since the inter-particle forces are given by the gradient of a scalar potential, the particle-particle interactions conserve energy.

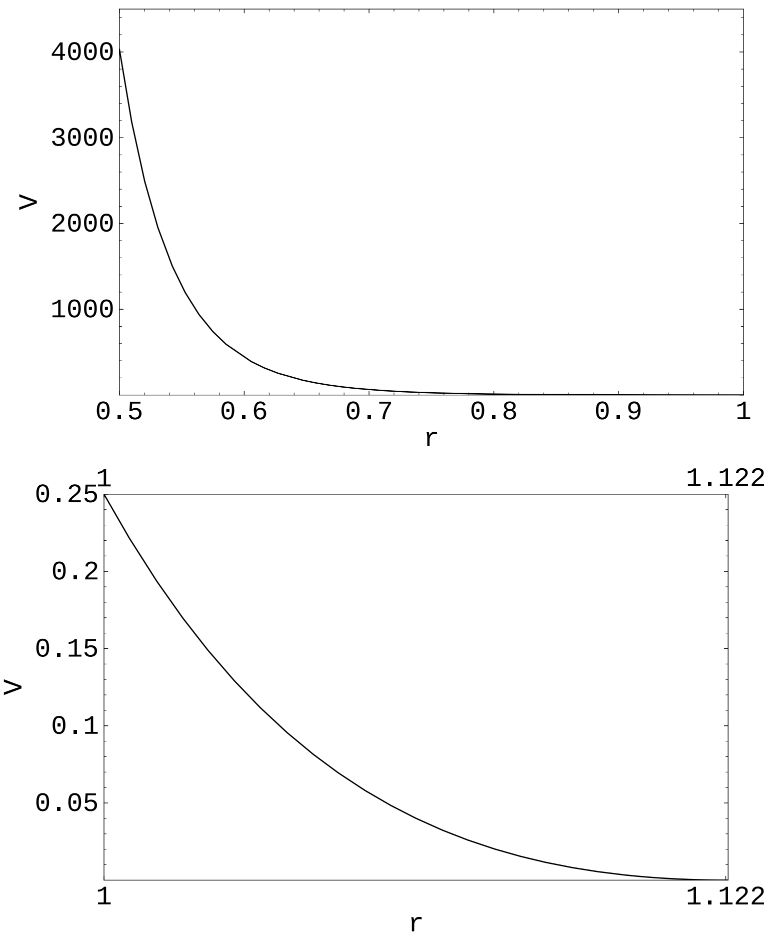

Figure 8: Plots of the Lennard-Jones potential for different ranges of $|\vec{r}_{ij}|$. a) V_{ij} versus $|\vec{r}_{ij}|$ for $0.5 \leq |\vec{r}_{ij}| \leq 1$. b) V_{ij} versus $|\vec{r}_{ij}|$ for $1 \leq |\vec{r}_{ij}| \leq 2^{1/6}$.

In my simulation I use the truncated Lennard-Jones potential, which is a popular potential for hard particle simulations[12]. The equation for the truncated Lennard-Jones potential is:

$$V_{ij} = \begin{cases} 0 & for\ |\vec{r}_{ij}| > 2^{1/6}d \\ 4\varepsilon[(\frac{d}{|\vec{r}_{ij}|})^{12} - (\frac{d}{|\vec{r}_{ij}|})^{6}] + \varepsilon & for\ |\vec{r}_{ij}| \leq 2^{1/6}d \end{cases} \tag{10}$$

This is a purely repulsive potential since it's derivative is negative when $|\vec{r}_{ij}| \leq 2^{1/6}d$ and is zero elsewhere. A plot of V_{ij} versus $|\vec{r}_{ij}|$ is shown in figure 8.

Since this system is conservative, an extra particle-particle force term must be added to model inelastic collisions. The inelastic inter-particle force term I use is from papers by Aoki and Akiyama[13][14].

The inelastic force on the i^{th} particle is:

$$\vec{f}_i = -\gamma m(\vec{v}_{ij} \cdot \vec{r}_{ij})\vec{r}_{ij}/|\vec{r}_{ij}|^2 \quad for\ |\vec{r}_{ij}| \leq 2^{1/6}d \tag{11}$$

Where $\vec{v}_{ij} = \vec{v}_j - \vec{v}_i$, and γ determines the strength of the inelastic force. This force acts in the $\vec{r}_{ij}$ direction, so it is a normal dissipative force. Aoki and Akiyama have used this inelastic term to realistically model pressure and density wave propagation in vertically vibrated granular materials[13]. They have also used this normal dissipative force with an additional dissipative shear force to accurately simulate pattern formations in vertically vibrated granular materials[14].

The code for the time driven simulation is in appendix B. It is based heavily on an gravitational nbody program that Dr. Matt Choptuik wrote[18].

4.2 The lsoda ODE Integrator

My time driven simulation uses the lsoda ODE integrator to integrate the equations of motion. Lsoda is the "Livermore Solver for Ordinary Differential equations, with Automatic method switching for stiff and non-stiff problems."[15]. For non-stiff problems lsoda uses the Adam's method[16] and for stiff problems the backward difference formula[17] is used. When lsoda is called, a number of arguments must be specified.

The first argument to lsoda is the address of a function that will set the equations of motion. This function must be of the form: $fcn(neq, t, y, ydot)$. neq is the number of equations that lsoda has to integrate. t is the time that lsoda starts integrating from. In my simulations, all potentials are time independent, so t is not used at all. y is the array that stores position and velocity information. $ydot$ is an array where the ODEs are specified. As seen in equations 7-11, the derivatives, $ydot$, are dependent on the positions and velocities of the particles, therefore $ydot$ is specified as a function of the various components of y. See appendix B for the routine I have written to set the equations of motion.

Two tolerances, T_a and T_r, are also passed into lsoda. T_a is the absolute tolerance, while T_r is the relative tolerance. While integrating the equations, lsoda evaluates a local error, e_i, for each y_i. lsoda ensures that $e_i < T_r\ abs(y_i) + T_a$ during the integration.

Other arguments supplied to lsoda are: y, neq, t_i and t_f, T_r. y and neq are the same variables that lsoda passes into fcn. t_i specifies the initial time that the system is at when lsoda is called. t_f is the time that lsoda will update the system to, so $t_f - t_i$ is the time step that the system evolves by.

5 Results From the Time Driven Simulation

The time driven simulation that I developed is much less efficient than the event driven simulation. I do not have results for the execution time of the program for different numbers of particles, but from the simulations I have run the execution time does not scale well with the number of particles. The maximum number of particles that I simulated with the time driven simulation was 100. With 100 particles, the time driven simulation was very slow. This is understandable since I did not spend very much time optimizing the time driven simulation to run fast. In the time driven simulation, each particle is compared to each other to see if they will interact in the given time step. The domain of the simulation could be broken into cells, similar to the event driven simulation, in order to reduce the number of particles that are compared to each other. Breaking the domain into cells would increase the execution speed of the time driven simulation.

The time driven algorithm may be slow, but it is easier to include oscillations and gravity in the simulation. In order to add oscillations to the system an extra term, $4\pi^2 f^2 A \cos(2\pi f t)\hat{y}$, is added to equation 8. In this additional term, f is the oscillation frequency and A is one half of the peak to peak amplitude of the oscillations. To add gravity to the simulation another extra term, $-g\hat{y}$, is added to equation 8, where g is the gravitational acceleration.

I used the time driven simulation to simulated 20 particles with oscillations and gravity. The initial positions of the particles were randomly distributed with $x \in \{0, 10\}$ and $y \in \{0, 10\}$. The initial velocities were randomly distributed with $v_x \in \{-1, 1\}$ and $v_y \in \{-1, 1\}$. Each particle in this simulation has a diameter of 1.7. The initial positions of the particles in this simulation are shown in figure 9 a), and the results are shown in figure 9 b). Notice that the balls in figure 9 b) overlap with each other and the walls of the box. This is because the inter-particle potential is not an infinite step well. The potential used, equation 10, in not infinitely steep at the particle boundary, so inter-particle and particle-wall penetration is possible. The disk-disk interactions in this simulation were elastic, but the disk-wall interaction was inelastic. The absolute and relative tolerances were set to 10^{-6} in this simulation. Pattern formations were not observed in this simulation, but gravity and oscillations were quite noticeable. To see an mpeg movie of this simulation, go to http://bh6.physics.ubc.ca/~awilson/tn20.mpg

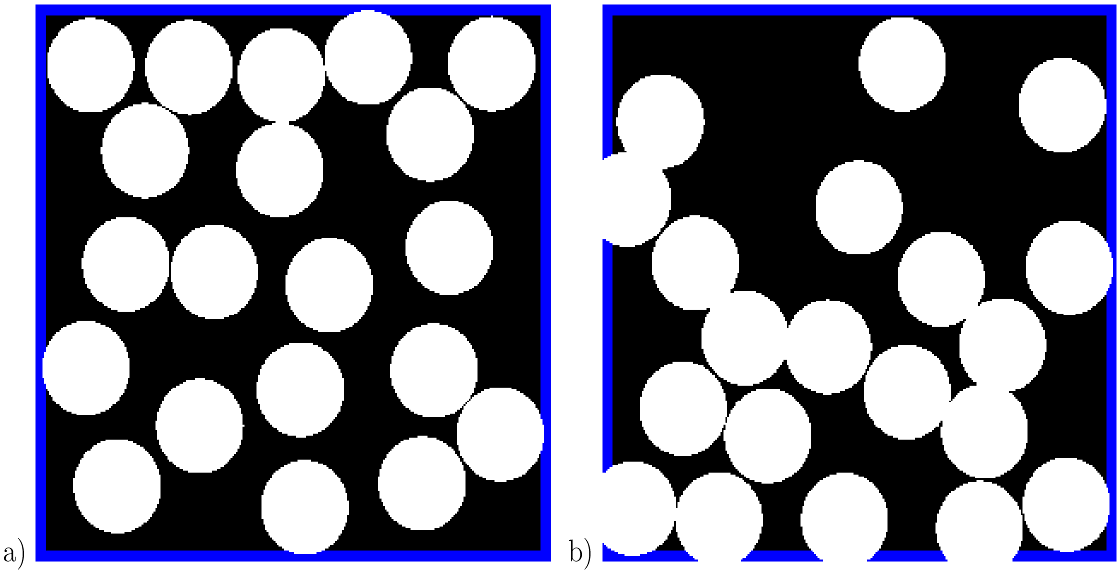

Figure 9: a) Initial conditions of the elastic N=20 particles time driven simulation b) Final condition of the elastic N=20 particles time driven simulation. This simulation include sinusoidal oscillations with $A = 0.25$, $f = 5.0$ and $g = 2.0$. The walls are inelastic with $\gamma = 1.0$.

The time driven simulation is sensitive to the time step and tolerance passed to lsoda. If the time step is too large, or the tolerance is too big, energy increases in the system. In order for energy and momentum, to be conserved, the time step and tolerances passed to lsoda have to be sufficiently small. For the simulation described above, energy was conserved to within 0.8 percent for each ball-ball collision.

6 Conclusion

I did not reach the point of simulating pattern formation in vertically oscillated granular materials. However, I did find that particles in a box tend to form clumps and strings if the particle-particle collisions are inelastic. These clumps are not found with elastic particle-particle collisions.

A fast and efficient event driven algorithm was developed. The event driven algorithm can not simulate oscillations or gravity and does not model collisions well enough to simulate pattern formations[4]. With the modifications discussed in section 3, I belive that the event driven simulation can include oscillations and gravity. A more effective way of modeling collisions must be implemented in order to simulate pattern formations.

A time driven simulation that includes oscillations and gravity was also developed. The time driven simulation is not nearly as fast as the event driven simulation and can not simulate many particles. With the modifications discussed in section 5, I believe that the time driven can be sped up. These modifications may not change the number of particles that the time driven simulation can simulate because of memory restrictions. The collision model in the time driven simulation was easy to implement and can be used to accurately simulate some properties of granular materials[13]. With an additional term in the collision model, I should be able to simulate pattern formations in two dimensions if a large enough number of particles can be simulated[14].

7 Acknowledgements

I would like to thank Matt Choptuik for all the help he has given me. I've learnt a lot about computers in a very short time span. I'd also like to thank Chris James for keeping me sane and vandalizing my computer when I needed it the most.

References

[1] H. M. Jaeger, S. R. Nagel, and R. P. Behringer, Review of Modern Physics, **68**, 1259 (1996)

[2] F. Melo, P.Umbanhowar, and H. L. Swinney, Physical Review Letters, 72, 172 (1994)

[3] H. K. Pak, and R. P. Behringer, Phys. Rev. Lett., **71**, 1832 (1993)

[4] C. Bizon, M. D. Shattuck, J. B. Swift, W. D. McCormick and H. L. Swinney, Physical Review Letters, **80**, 57 (1998)

[5] F. Melo, P. B. Umbanhowar, and H. L. Swinney, Physical Review Letters, **75**, 3838 (1995)

[6] J. B. Knight, H. M. Jaeger, and S. R. Nagel, Phys. Rev. Lett., **70**, 3728 (1993)

[7] P. Evesque and J. Rajchenbach, 1989, Phys. Rev. Lett., **62**, 44 (1989)

[8] M. Marin, D. Risso, and P. Cordero, Journal of Computational Physics, **109**, 306 (1993)

[9] B. J. Alder and T. E. Wainright, Journal of Chemical Physics, **31**, 459 (1959)

[10] http://hissa.nist.gov/dads/

[11] B. D. Lubachevsky, Journal of Computational Physics, **94**, 255 (1991)

[12] R. J. Sadus, **Molecular Simulations of Fluids: Theory, Algorithms and Object-Orientation**, Elsevier Publishers, Amsterdam, 1999

[13] K. M. Aoki and T. Akiyama, Physical Review E, **52**, 3288 (1995)

[14] K. M. Aoki and T. Akiyama, Physical Review Letters, **77**, 4166 (1996)

[15] from the lsoda source code, available from http://laplace.physics.ubc.ca/People/matt/Teaching/98Fall/Phy329/Doc/ode/src_odepack/lsoda.f

[16] http://mathworld.wolfram.com/AdamsMethod.html

[17] http://mathworld.wolfram.com/NewtonsBackwardDifferenceFormula.html

[18] Dr. Matt Choptuik can be contacted at *matt@laplace.physics.ubc.ca*

Appendix

A Event Driven Simulation Code

C++ code for the event driven simulation. Written from scratch by myself, using the algorithm of Marin, Risso and Cordero[8]

A.1 Main Source Code for the Event Driven Algorithm: edriven.C

```
#include "edriven.h"
#include <iostream.h>
#include <math.h>
#include <stdlib.h>
#include <stddef.h>
#include <stdio.h>
#include "blibs.h"
#include "eventList.h"
#include "intList.h"

const double PI = 3.1415926536;
const double D = 0.09;                 // Diameter of the particles
const double COEFF = 1;                // inelastic collision paramter
int N_LOOPS = 1000000000;              // # of times through the main loop
const size_t N_PARTICLES = 300;        // # of particles
const int N_XCELLS = 10;              // The # of cells in the x and y direction
const int N_YCELLS = 10;
const double X_SPACING = 1;          // The width of the x and y cells
const double Y_SPACING = 1;
// The size of the domain is
// (N_XCELLS*X_SPACING, N_YCELLS*Y_SPACING)
int random_assign = 1;                 // if(random_assign) assign IC's randomly
                                       // else fill the domain from the bottom up
int dtime = 1;                         // if(dtime) coordinates are output at fixed
                                       // time steps, dt
double pdtime = 0;                     // variable used in discrete time output
const double dt = 0.1;
const double finish_time = 1000;       // if(dtime), simulation will stop at finish_time
int glbpp = 1;                         // flag for data output
```

```
// Definition of event types
const int DDC = 1;                      // Disk-Disk collision
const int DWC = 2;                      // Disk-Wall collision
const int VWC = 3;                      // Virtual Wall collision
const int NUL = -1;                     // Error
double parts[N_PARTICLES][5];           // The state of the system is kept in parts
                                        // For particle i:
                                        // x = (parts[i][0], parts[i][1])
                                        // v = (parts[i][2], parts[i][3])
                                        // tau = parts[i][4]
const int CELLMATRIX_LENGTH = 100;       // max # of disks in a cell
const int NARR_LENGTH = 9*CELLMATRIX_LENGTH;  // max # of neighbours a disk can have
int cellmatrix[N_XCELLS][N_YCELLS][CELLMATRIX_LENGTH];  // the matrix that keeps track of
                                                        // what cell the disks are in

eventNode **localList;                  // stores all potential events for each disk

int C[N_PARTICLES+N_XCELLS+N_YCELLS+2];// how many hard collisions are associated
                                        // with each particle.

int FEL[2*N_PARTICLES-1];               // Future Events List

Event *lml;                             // The local minima list.
                                        // holds the minima event associated with
                                        // every disk



// Returns the collision time between two particles particle1 and particle2
double tball(int p1, int p2){
  double a, b, c;
  double pos10[2], pos20[2];
  int i;
  a=0;
  b=0;
  c=0;

  // Values needed to decide whether a collision will happen or not
  // pos10[] and pos20[] array calculation
  for(i=0;i<2;i++){
  pos10[i]=parts[p1][i]+parts[p1][i+2]*(max(parts[p1][4],parts[p2][4])-parts[p1][4]);
  pos20[i]=parts[p2][i]+parts[p2][i+2]*(max(parts[p1][4],parts[p2][4])-parts[p2][4]);
  }
  // calculation of b
  for(i=0;i<2;i++){
  b=b+(pos20[i]-pos10[i])*(parts[p2][i+2]-parts[p1][i+2]);
  }
  // calculation of a
  for(i=0;i<2;i++){
    a=a+(parts[p2][i+2]-parts[p1][i+2])*(parts[p2][i+2]-parts[p1][i+2]);
  }
  // calculation of c
  for(i=0;i<2;i++){
    c=c+(pos20[i]-pos10[i])*(pos20[i]-pos10[i]);
  }
  c=c-D*D;
  // Decide if the collision will happen or not
  if((b<=0)&&((b*b-a*c)>=0)){
    double time;
    time=(-b-sqrt(b*b-a*c))/a;
    if(time<0){
      cout<<"Negative time for ball-ball collision!\n";
      cout<<"Particle1:\n"<<"\tposition: ("<<parts[p1][0]<<","<<parts[p1][1]<<")\n";
      cout<<"\tvelocity: ("<<parts[p1][2]<<","<<parts[p1][3]<<")\n";
      cout<<"\ttau:  "<<parts[p1][4]<<"\n";;
      cout<<"\n";
      cout<<"Particle2:\n"<<"\tposition: ("<<parts[p2][0]<<","<<parts[p2][1]<<")\n";
      cout<<"\tvelocity: ("<<parts[p2][2]<<","<<parts[p2][3]<<")\n";
```

```
      cout<<"\ttau:  "<<parts[p2][4]<<"\n";
      cout<<"\n\n";
      cout<<"\a";
      exit(1);
    }
    return time+max(parts[p1][4],parts[p2][4]);
  }
  else
    return HUGE_VAL;
}

// Searches the cell matrix for a particle, i
void search_cell_matrix(int idisk, int cell[]){
  cell[0]=-1;
  cell[1]=-1;
  for(int i=0;i<N_XCELLS;i++){
    for(int j=0;j<N_YCELLS;j++){
      if(int_search(cellmatrix[i][j],idisk, CELLMATRIX_LENGTH)!=-1){
cell[0]=i;
cell[1]=j;
      }
    }
  }
  if((cell[0]==-1)||(cell[1]==-1)){
    cout<<"An invalid disk number was passed tosearch_cell_matrix!\n";
    cout<<"Invalid disk number is: "<<idisk<<"\n";
    cout<<"\a\n\n";
    exit(1);
  }
}


// Returns the collision time predicted for particle with a wall
// Returns type of collision: DWC or VWC
// Returns wall number of the collision
double twall(int part,int& colltype, int& wall){
  double wall_place;
  int cell[2];
  double time;
  int top;
  int right;
  top=0;
  right=0;
  time=-1.0;
  wall=-1;
  search_cell_matrix(part,cell);
  if(parts[part][2]>=0){
    right=1;
  }
  if(parts[part][3]>=0){
    top=1;
  }
  // The 4 walls associated with a disk in cell (n,m)
  // are (N+n, N+n+1, N+N_XCELLS+m+1, N+XCELLS+m+2)

  // The walls along the y-axis first
  // only consider the right wall
  if(right){
    wall_place=(cell[0]+1)*X_SPACING;
    colltype=DWC;
    wall=N_XCELLS+N_PARTICLES;
    time=(N_XCELLS*X_SPACING-parts[part][0]-D/2)/parts[part][2];
    if(((wall_place-parts[part][0])/parts[part][2])<time){
      colltype=VWC;
      time=(wall_place-parts[part][0])/parts[part][2];
      wall=cell[0]+N_PARTICLES+1;
    }
  }
```

```
  // only consider the left wall
  else{
    wall_place=double(cell[0])*X_SPACING;
    colltype=DWC;
    time=(0-parts[part][0]+D/2)/parts[part][2];
    wall=N_PARTICLES;
    if(((wall_place-parts[part][0])/parts[part][2])<time){
      colltype=VWC;
      time=(wall_place-parts[part][0])/parts[part][2];
      wall=cell[0]+N_PARTICLES;
    }
  }

  // walls along x-axis next
  // only consider the top wall
  if(top){
    if(((N_YCELLS*Y_SPACING-parts[part][1]-D/2)/parts[part][3])<time){
      time=(N_YCELLS*Y_SPACING-parts[part][1]-D/2)/parts[part][3];
      colltype=DWC;
      wall=cell[1]+N_PARTICLES+N_XCELLS+2;
    }
    wall_place=double(cell[1]+1)*Y_SPACING;
    if(((wall_place-parts[part][1])/parts[part][3])<time){
      time=(wall_place-parts[part][1])/parts[part][3];
      colltype=VWC;
      wall=cell[1]+N_PARTICLES+N_XCELLS+2;
    }
  }
  // only consider the bottom wall
  else{
    if(((0-parts[part][1]+D/2)/parts[part][3])<time){
      colltype=DWC;
      time=(0-parts[part][1]+D/2)/parts[part][3];
      wall=cell[1]+N_PARTICLES+1+N_XCELLS;
    }
    wall_place=double(cell[1])*Y_SPACING;
    if(((wall_place-parts[part][1])/parts[part][3])<time){
      colltype=VWC;
      time=(wall_place-parts[part][1])/parts[part][3];
      wall=cell[1]+N_PARTICLES+1+N_XCELLS;
    }
  }
  if(time<0){
    cout<<"\n\nTrouble calculating the wall collision time!\n\n";
    cout<<"particle "<<part<<"\n";
    cout<<"position:  ("<<parts[part][0]<<","<<parts[part][1]<<")\n";
    cout<<"velocity:  ("<<parts[part][2]<<","<<parts[part][3]<<")\n";
    cout<<"cell:  ("<<cell[0]<<", "<<cell[1]<<")\n";
    cout<<"tau:  "<<parts[part][4]<<"\n";
    cout<<"time:  "<<time<<"\n";
    cout<<"wall:  "<<wall<<"\n";
    if(wall<=(int(N_PARTICLES)+N_XCELLS)){
      cout<<"wall position:  x = "<<(wall-N_PARTICLES)*X_SPACING<<"\n";;
    }
    else{
      cout<<"wall position:  y = "<<(wall-N_PARTICLES-N_XCELLS-1)*Y_SPACING<<"\n";
    }
    cout<<"\n\n\a";
    exit(1);
  }
  return time+parts[part][4];
}

// Free evolution of update to time tau plus tplus, then  updates tau
void freev(int update, double tplus){
  for(int i=0;i<2;i++){
  parts[update][i]=parts[update][i]+tplus*parts[update][i+2];
  }
```

```
  parts[update][4]=parts[update][4]+tplus;
}

// outputs the x cooridinate of the disk at time tau + tplus
// doesn't update the state of the particle
double freevx(int update, double tplus){
  return parts[update][0]+tplus*parts[update][2];
}
// outputs the y cooridinate of the disk at time tau + tplus
// doesn't update the state of the particle
double freevy(int update, double tplus){
  return parts[update][1]+tplus*parts[update][3];
}

// makes insantaneous changes suffered by hard collision of walldisk with wall
void wallev(int walld, int wall){
  // Hits left or right wall
  if((wall==int(N_PARTICLES))||(wall==int(N_PARTICLES+N_XCELLS))){
    parts[walld][2]=-parts[walld][2];
  }
  // Hits top or bottom wall
  else if((wall==N_PARTICLES+N_XCELLS+1)||(wall==N_PARTICLES+N_XCELLS+1+N_YCELLS)){
    parts[walld][3]=-parts[walld][3];
  }
  else{
    cout<<"In function wallev, an invalid wall number was passed to the function!\n";
    cout<<"\a\n\n";
    exit(1);
  }
}

// The collision of disk1 and disk2 is modeled in this routine.
void diskev(int disk1, int disk2){
  double v1norm[2];
  double v1nproj;
  double v2norm[2];
  double v2nproj;
  double norm[2];
  double v1tang[2];
  double v2tang[2];
  double dist;
  int i;
  // x = (parts[i][0], parts[i][1])
  // v = (parts[i][2], parts[i][3])
  // tau = parts[i][4]
  dist = 0;
  for(i=0;i<2;i++){
    dist = dist + pow(parts[disk2][i]-parts[disk1][i],2);
    v1norm[i] = v2norm[i] = v1tang[i] = v2tang[i] = 0;
  }
  dist = sqrt(dist);
  for(i=0;i<2;i++){
    norm[i]= (parts[disk2][i]-parts[disk1][i])/dist;
  }
  v1nproj = 0;
  v2nproj = 0;
  for(i=0;i<2;i++){
    v1nproj = v1nproj + parts[disk1][i+2]*norm[i];
    v2nproj = v2nproj + parts[disk2][i+2]*norm[i];
  }
  for(i=0;i<2;i++){
    v1norm[i] = v1nproj*norm[i];
    v1tang[i] = parts[disk1][i+2] - v1norm[i];
    v2norm[i] = v2nproj*norm[i];
    v2tang[i] = parts[disk2][i+2] - v2norm[i];
    parts[disk1][i+2] = COEFF*(v1tang[i] + v2norm[i]);
    parts[disk2][i+2] = COEFF*(v2tang[i] + v1norm[i]);
  }
```

```
}

// Returns a list of disks in the neighborhood of a disk, i
void neighborhood(int* narr, int idisk){
  int cell_of_idisk[2];
  int_clear(narr, NARR_LENGTH);
  cell_of_idisk[0]=-1;
  cell_of_idisk[1]=-1;
  search_cell_matrix(idisk, cell_of_idisk);
  for(int i=cell_of_idisk[0]-1;i<cell_of_idisk[0]+2;i++){
    for(int j=cell_of_idisk[1]-1;j<cell_of_idisk[1]+2;j++){
      if((j>=0)&&(i>=0)&&(i<=(N_XCELLS-1))&&(j<=(N_YCELLS-1))){
int_union(narr, cellmatrix[i][j], NARR_LENGTH, CELLMATRIX_LENGTH);
      }
    }
  }
  int_remove(narr, idisk, NARR_LENGTH);
}

// This routine is used to choose the first event only
void create_cbt(){
  // If number of leaves is odd, odd man out gets promoted with
  // no competition in first round
  int fel_l;
  double min_time;
  fel_l=2*N_PARTICLES-2;

  // Can treat arrays now as having an even number of leaves
  while(fel_l>0){
    min_time=min(lml[FEL[fel_l]].schedualed_time,lml[FEL[fel_l-1]].schedualed_time);
    if(min_time==lml[FEL[fel_l]].schedualed_time){
      FEL[(fel_l-2)/2]=FEL[fel_l];
    }
    else{
      FEL[(fel_l-2)/2]=FEL[fel_l-1];
    }
    fel_l=fel_l-2;
  }
}

// This event chooses the next event, only if there have been previous events
void update_cbt(int disk){
  int cbtp;
  double mint;
  // remove invalid DDC events from cbt
  if(lml[disk].event_type==DDC){
    if(lml[disk].c!=C[lml[disk].partner]){
      event_list_remove(localList[disk], lml[disk].schedualed_time);
      lml[disk]=localMin(localList[disk]);
    }
  }

  cbtp=N_PARTICLES+disk-1;
  // cbtp is odd
  while(cbtp>0){
    if(cbtp%2){
      mint=min(lml[FEL[cbtp]].schedualed_time,lml[FEL[cbtp+1]].schedualed_time);
      if(mint==lml[FEL[cbtp]].schedualed_time){
FEL[(cbtp-1)/2]=FEL[cbtp];
      }
      else{
FEL[(cbtp-1)/2]=FEL[cbtp+1];
      }
      cbtp=(cbtp-1)/2;
    }
    // cbtp is even
    else{
      mint=min(lml[FEL[cbtp]].schedualed_time,lml[FEL[cbtp-1]].schedualed_time);
```

```
      if(mint==lml[FEL[cbtp]].schedualed_time){
FEL[(cbtp-2)/2]=FEL[cbtp];
      }
      else{
FEL[(cbtp-2)/2]=FEL[cbtp-1];
      }
      cbtp=(cbtp-2)/2;
    }
  }
}

// Chooses next event using the FEL
double next_event(int& event_type, int& disk, int& partner){
  double time;
  int go=1;
  if((disk==-1)&&(partner==-1)){
    create_cbt();
  }
  else{
    update_cbt(disk);
    if((partner>=0)&&(partner<int(N_PARTICLES))){
      update_cbt(partner);
    }
  }
  do{
    disk=FEL[0];
    partner=lml[FEL[0]].partner;
    event_type=lml[FEL[0]].event_type;
    time=lml[FEL[0]].schedualed_time;
    // event is invalid, find another one
    if((event_type==DDC)&&(lml[disk].c!=C[partner])){
      update_cbt(disk);
    }
    else if((event_type==DWC)||(event_type==DDC)){
      event_list_clear(localList[disk]);
      C[disk]++;
      if(event_type==DDC){
event_list_clear(localList[partner]);
      }
      go=0;
    }
    else if(event_type==VWC){
      event_list_remove(localList[disk], time);
      go=0;
    }
  }while(go);

  if(event_type==DDC){
    C[partner]++;
  }


  return time;
}

// scheduals an event into the local list of disk
void schedual(int disk, double time, int event_type, int partner){
  if(event_search(localList[disk],partner)==NULL){
    head_insert_event(localList[disk], time, event_type, partner, -1);
  }
}
// an overloaded version of the above function
void schedual(int disk, double time, int event_type, int partner, int c){
  if(event_search(localList[disk],partner)==NULL){
    head_insert_event(localList[disk], time, event_type, partner, c);
  }
}
```

```
// predicts future events
void predictions(int disk, int& event_type){
  double time;
  int partner;
  int narr[NARR_LENGTH];
  int i = 0;

  event_list_clear(localList[disk]);
  int_clear(narr, NARR_LENGTH);
  neighborhood(narr, disk);
  while(narr[i]!=-1){
    time=tball(disk, narr[i]);
    if(time!=HUGE_VAL){
      schedual(disk, time, DDC, narr[i], C[narr[i]]);
    }
    i++;
  }
  time=twall(disk,event_type, partner);
  schedual(disk, time, event_type, partner);
  lml[disk]=localMin(localList[disk]);
}

// moves disk through the wall into a new cell in the cellmatrix
void updateM(int disk, int wall){
  int cell[2];
  int vert;
  vert=0;
  search_cell_matrix(disk, cell);

  int_remove(cellmatrix[cell[0]][cell[1]],disk, CELLMATRIX_LENGTH);

  // is the virtual wall a top wall or bottom wall
  if(wall>int((N_PARTICLES+N_XCELLS))){
    vert=1;
  }
  else{
    vert=0;
  }
  if(vert){
    if(wall>(cell[1]+int(N_PARTICLES+N_XCELLS)+1)){
      // ball passed through the top wall
      int_insert(cellmatrix[cell[0]][cell[1]+1],disk, CELLMATRIX_LENGTH);
    }
    else{
      // ball passed through the bottom wall
      int_insert(cellmatrix[cell[0]][cell[1]-1],disk, CELLMATRIX_LENGTH);
    }
  }
  else{
    if(wall>(cell[0]+int(N_PARTICLES))){
      // ball passed through the right wall
      int_insert(cellmatrix[cell[0]+1][cell[1]],disk, CELLMATRIX_LENGTH);
    }
    else{
      // ball passed through the left wall
      int_insert(cellmatrix[cell[0]-1][cell[1]],disk, CELLMATRIX_LENGTH);
    }
  }
}

// non-random position assignment
void pos_vel_assign(){
  double pos[2];
  pos[0]=2*double(D);
  pos[1]=2*double(D);
  for(int i=0;i<int(N_PARTICLES);i++){
    for(int j=0;j<=1;j++){
      parts[i][j]=pos[j];
```

```
      parts[i][j+2]=(1.0-2.0*(double(rand())/double(RAND_MAX)));
    }
    pos[0]=pos[0]+2*double(D);
    if(pos[0]>=((double(N_XCELLS)*X_SPACING)-2*double(D))){
      pos[0]=double(D);
      pos[1]=pos[1]+2*double(D);
    }
    if(pos[1]>=((double(N_YCELLS)*Y_SPACING)-2*double(D))){
      fprintf(stderr,"Not enough space for these particles!\n");
      fprintf(stderr,"\a\n\n");
      exit(1);
    }
    parts[i][4] = 0;
  }
}

// check to see if the disks overlap from the random postion assignment
void check(int disk, int& ok){
  double dist;
  ok=1;
  for(int i=0;i<disk;i++){
    dist=sqrt(pow((parts[disk][0]-parts[i][0]),2)+pow((parts[disk][1]-parts[i][1]),2));
    if(dist<D){
      ok=0;
    }
    if((parts[disk][0]<(D/2))||(parts[disk][0]>(N_XCELLS*X_SPACING-D/2))){
      ok=0;
    }
    if((parts[disk][1]<(D/2))||(parts[disk][1]>(N_YCELLS*Y_SPACING-D/2))){
      ok=0;
    }
  }
}

// random position assignment
void rand_pos_vel_assign(){
  int checked=0;
  for(int i=0;i<int(N_PARTICLES);i++){
    checked=0;
    while(!checked){
      for(int j=0;j<2;j++){
parts[i][j]=10*(double(rand())/double(RAND_MAX));
parts[i][j+2]=(1.0-2.0*(double(rand())/double(RAND_MAX)));
      }
      check(i, checked);
    }
    parts[i][4] = 0;
    fprintf(stderr,"intialized particle %d\n", i);
  }
}



// ##################################################################
// ##                                                              ##
// ##                    MAIN FUNCTION                             ##
// ##                                                              ##
// ##################################################################
int main(){
  //  ##############################################################
  //  INITIALIZE THE VARIABLES
  //  ##############################################################
  if((int(4*X_SPACING*Y_SPACING/PI*pow(D,2))+1)> NARR_LENGTH){
    fprintf(stderr,"Problems with neighbourhood size:\n");
    fprintf(stderr,"\tIncrease NARR_LENGTH\n");
    fprintf(stderr,"\tIncrease D");
    fprintf(stderr,"\tor decrease X_SPACING and Y_SPACING\n");
    cout<<"Exiting Program!\n";
```

```
    exit(0);
  }
  if((localList=(eventNodePtr *)calloc(N_PARTICLES,sizeof(eventNodePtr)))==NULL){
    fprintf(stderr,"Couldn't allocate memory for the localList array!\n");
    fprintf(stderr,"\a");
    exit(1);
  }
  if((lml=(Event *)calloc(N_PARTICLES,sizeof(Event)))==NULL){
    fprintf(stderr,"Couldn't allocate memory for the Local Minima List!\n");
    fprintf(stderr,"\a");
    exit(1);
  }
  double time;
  int disk;
  int partner;
  int coll_type;
  int wall;
  int narr[NARR_LENGTH];
  int i, j;

  time=-1;
  disk=-1;
  partner=-1;
  coll_type=NUL;
  int_clear(narr, NARR_LENGTH);

  if(random_assign){
    rand_pos_vel_assign();
  }
  else{
    pos_vel_assign();
  }
  // make the cell-matrix
  for(i=0;i<N_XCELLS;i++){
    for(j=0;j<N_YCELLS;j++){
      int_clear(cellmatrix[i][j], CELLMATRIX_LENGTH);
    }
  }
  for(i=0;i<int(N_PARTICLES);i++){
    int intPosx;
    int intPosy;
    intPosx=int(parts[i][0]/X_SPACING);
    intPosy=int(parts[i][1]/Y_SPACING);
    int_insert(cellmatrix[intPosx][intPosy],i, CELLMATRIX_LENGTH);
  }
  //build the C matrix
  for(i=0;i<int(N_PARTICLES);i++){
    C[i]=0;
  }
  for(i=int(N_PARTICLES);i<(int(N_PARTICLES+N_XCELLS+N_YCELLS)+2);i++){
    C[i]=-1;
  }
  //build the local event list
  for(i=0;i<int(N_PARTICLES);i++){
    neighborhood(narr,i);
    j=0;
    while(narr[j]!=-1){
      time=tball(i,narr[j]);
      if(time!=HUGE_VAL){
head_insert_event(localList[i], time, DDC, narr[j], C[narr[j]]);
      }
      j++;
    }
    time=twall(i,coll_type, wall);
    head_insert_event(localList[i], time, coll_type, wall, -1);
  }
  // fill the local minima list, lml
  // initialize the FEL
```

```
  for(i=0;i<int(N_PARTICLES);i++){
    lml[i]=localMin(localList[i]);
    FEL[N_PARTICLES-1+i]=i;
  }
  // ##################################################################


  // ##################################################################
  // ##                                                              ##
  // ##         Data output                                          ##
  // ##                                                              ##
  // ##################################################################
  fprintf(stdout,"%d\n",N_PARTICLES);
  for(i=0;i<int(N_PARTICLES);i++){
    fprintf(stdout,"%g\n",D);
  }
  fprintf(stdout,"0.0\n");
  if(glbpp == 1){
    fprintf(stdout,"-1\n-1\n");
  }
  for(i=0;i<int(N_PARTICLES);i++){
    fprintf(stdout,"%g %g 0\n",parts[i][0], parts[i][1]);
  }



  // ##################################################################
  // ##                                                              ##
  // ##          This is where the main loop of                      ##
  // ##          calculations are.                                   ##
  // ##                                                              ##
  // ##################################################################
  time = 0;
  for(i=0;i<N_LOOPS;i++){
    time=next_event(coll_type,disk,partner);

    // output the coordinates at discrete times
    if(dtime){
      while(pdtime+dt<time){
pdtime = pdtime + dt;
fprintf(stdout,"%g\n",pdtime);
if(glbpp == 1){
  fprintf(stdout,"-1\n-1\n");
}
for(j=0;j<int(N_PARTICLES);j++){
  fprintf(stdout,"%g\t",freevx(j,pdtime-parts[j][4]));
  fprintf(stdout,"%g\t0\n",freevy(j,pdtime-parts[j][4]));
}
      }
      if(pdtime>finish_time){
i=N_LOOPS;
      }
    }

    switch(coll_type){
    case DDC:
      freev(disk,time-parts[disk][4]);
      freev(partner, time-parts[partner][4]);
      diskev(disk, partner);
      predictions(disk, coll_type);
      predictions(partner, coll_type);
      break;

    case DWC:
      freev(disk, time-parts[disk][4]);
      wallev(disk, partner);
      predictions(disk, coll_type);
      break;
```

```
    case VWC:
      updateM(disk, partner);
      predictions(disk, coll_type);
      break;
    }
  }
}
```

A.2 blibs.C

```
#include <stddef.h>
#include <iostream.h>

// returns the max of arg1 and arg2
double max(double arg1, double arg2){
  if(arg1>arg2)
    return arg1;
  else if(arg2>arg1)
    return arg2;
  else
    return arg1;
}
// overloaded max with integer arguments
int max(int arg1, int arg2){
  if(arg1>arg2)
    return arg1;
  else if(arg2>arg1)
    return arg2;
  else
    return arg1;
}

// returns min of arg1 and arg2
double min(double arg1, double arg2){
  if(arg1<arg2)
    return arg1;
  else if(arg2<arg1)
    return arg2;
  else
    return arg1;
}
// overloaded min with integer arguments
int min(int arg1, int arg2){
  if(arg1<arg2)
    return arg1;
  else if(arg2<arg1)
    return arg2;
  else
    return arg1;
}
```

A.3 eventList.C

```
#include <stddef.h>
#include <stdlib.h>
#include <iostream.h>
/*
  This file contains the definition of an event and
  also contains all the functions and structs for
  singly linked lists of events
*/

// Definition of an event
struct Event{
  double schedualed_time;
  int event_type;
```

```
  int partner;
  int c;
};

// Definition of eventNode and typedef for pointer
struct eventNode{
  Event data;
  eventNode *link;
};
typedef eventNode* eventNodePtr;

// Inserts a value at the head of the list
void head_insert_event(eventNode*& head, Event the_event){
  eventNodePtr temp_ptr;
  temp_ptr= new eventNode;
  if(temp_ptr==NULL){
    cout<<"Error: Insufficient storage.\n";
    exit(1);
  }
  temp_ptr->data.schedualed_time=the_event.schedualed_time;
  temp_ptr->data.event_type=the_event.event_type;
  temp_ptr->data.partner=the_event.partner;
  temp_ptr->data.c=the_event.c;
  temp_ptr->link=head;
  head=temp_ptr;
}
// overloaded version of the above routine
void head_insert_event(eventNode*& head, double schedualed_time, int event_type, int partner, int c){
  eventNodePtr temp_ptr;
  temp_ptr= new eventNode;
  if(temp_ptr==NULL){
    cout<<"Error: Insufficient storage.\n";
    exit(1);
  }
  temp_ptr->data.schedualed_time=schedualed_time;
  temp_ptr->data.event_type=event_type;
  temp_ptr->data.partner=partner;
  temp_ptr->data.c=c;
  temp_ptr->link=head;
  head=temp_ptr;
}

// returns the length of the list
int event_list_length(eventNodePtr head){
  eventNodePtr tempPtr;
  int returnint;
  returnint=0;
  for(tempPtr=head;tempPtr!=NULL;tempPtr=tempPtr->link){
    returnint++;
  }
  return returnint;
}

// returns the current local minimal event
Event localMin(eventNodePtr head){
  Event returnEvent;
  eventNode *tempPtr;

  tempPtr=head;
  returnEvent=head->data;

  while(tempPtr->link!=NULL){
    tempPtr=tempPtr->link;
    if((tempPtr->data.schedualed_time)<(returnEvent.schedualed_time)){
      returnEvent=tempPtr->data;
    }
  };
  return returnEvent;
```

```
}

// removes an event from the head of the list
void event_list_head_remove(eventNode*& head){
  eventNodePtr rmPtr;

  rmPtr=head;
  head=head->link;
  delete rmPtr;
}

// clears the list
void event_list_clear(eventNode*& head){
  while(head!=NULL)
    event_list_head_remove(head);
}

// removes the event just before the prev pointer
void event_list_remove_previous(eventNode*& prev){
  eventNode *rm_ptr;
  rm_ptr=(prev->link);
  (prev->link)=(rm_ptr->link);
  delete rm_ptr;
}

// removes an event with the specified time from the list
void event_list_remove(eventNode*& head, double time){
  eventNode *previous;
  eventNode *temp;
  int got_it;
  got_it=0;
  if((head->data.schedualed_time)==time){
    event_list_head_remove(head);
    return;
  }
  temp = head;
  while(!got_it){
    previous=temp;
    temp=(temp->link);
    if((temp->data.schedualed_time)==time){
      (previous->link)=(temp->link);
      delete temp;
      got_it=1;
    }
  }
}

// searches for an event with partner in the list
eventNodePtr event_search(eventNodePtr head, int partner){
  eventNodePtr here;
  here=head;
  if(here==NULL){
    return NULL;
  }
  else{
    while((here->data.partner)!=partner && (here->link)!=NULL)
      here=here->link;
    if((here->data.partner)==partner){
      return here;
    }
    else{
      return NULL;
    }
  }
}
```

A.4 intList.C

```
#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>
/*
  This file contains all the functions for the integer arrays needed
  by Jan8array
*/

// Inserts a value at the end of the array
// Returns 1 if the integer was added successfully
// Returns -1 if the integer was not added
int int_insert(int* arr, int the_number, int arr_length){
  int place = 0;
  int insert_flag = 0;
  for(;place<arr_length;place++){
    if(arr[place]<0){
      arr[place]=the_number;
      insert_flag = 1;
    }
    if(insert_flag){
      return 1;
    }
  }
  return -1;
}



// Searches the array for a specific number, target
// returns the array index if target was found.
// returns -1 if target was not found.
int int_search(int* arr, int target, int arr_length){
  int i = 0;
  for(i = 0;i<arr_length;i++){
    if(arr[i]==target){
      return i;
    }
  }
  return -1;
}

// Sets all of the values in the array to -1
void int_clear(int* arr, int arr_length){
  int i = 0;
  for(i=0;i<arr_length;i++){
    if(arr[i]==-1){
      return;
    }
    arr[i]=-1;
  }
}

// copies the content of arr into copy
// if copy_length is too small, returns -1
// if arr is copied fine, returns 1
int int_copy(int* arr, int* copy, int arr_length, int copy_length){
  int i = 0;
  if(copy_length<arr_length){
    return -1;
  }
  for(i=0;i<copy_length;i++){
    if(i<arr_length){
      copy[i]=arr[i];
    }
    else{
      copy[i]=-1;
```

```
    }
  }
  return 1;
}

// places join at the end of arr
// returns 1 if ok, -1 if not.
int int_union(int* arr, int* join, int arr_length, int join_length){
  int index = -1;
  int i =0;
  index = int_search(arr, -1, arr_length);
  if(index==-1){
    return -1;
  }
  if((arr_length-index)<join_length){
    return -1;
  }
  for(i=index;i<(index+join_length);i++){
    arr[i]=join[i-index];
  }
  return 1;
}

// removes number from the array
// returns 1 if removed, -1 if not
int int_remove(int* arr, int number, int arr_length){
  int i = 0;
  int index = -1;
  if(arr[0] == -1){
    return -2;
  }
  index = int_search(arr, number, arr_length);
  if(index<0){
    return -1;
  }
  for(i=index;i<(arr_length-1);i++){
    arr[i]=arr[i+1];
  }
  arr[arr_length-1]=-1;
  return 1;
}
```

B Time Driven Simulation Code

Fortran code for the time driven simulation. This is heavily based on an nbody gravitational simulation that Dr. Matt Choptuik wrote[18].

B.1 hpart.f

```
c-------------------------------------------------------
c     This code is heavily based on a gravitational
c     nbody program written by Dr. Matt Choptuik:
c     matt@laplace.physics.ubc.ca
c     The fcn1 routine was rewritten to implement the
c     Lennard-Jones potential, and the energy routine
c     was written to calculate the energy of the system
c-------------------------------------------------------
      program       hpart

      implicit      none

      character*5   cdnm
      parameter    ( cdnm = 'hpart' )
      integer       iargc,        indlnb,        i4arg
```

```
      real*8         r8arg
      real*8         r8_never, imomx, momx, imomy, momy
      parameter    ( r8_never = -1.0d-60 )

c-------------------------------------------------------------
c     tfinal:        Final integration time
c     dtout:         Output interval
c     ntout:         # of output times (computed)
c     trace:         1 to monitor conserved quantities
c-------------------------------------------------------------
      real*8         tfinal,         dtout
      integer        ntout
      logical        trace

c     The cutoff for the Lennard-Jones potential
      real*8         r0


      include       'fcn.inc'

      integer        maxneq,         neq
      parameter    ( maxneq = 6 * maxnp )

c-------------------------------------------------------------
c     LSODA Variables.
c-------------------------------------------------------------
      external       fcn,        jac

      real*8         y(d,2,maxnp)
      real*8         tbgn,       tend
      integer        itol
      real*8         rtol,       atol
      integer        itask,      istate,     iopt
      integer        lrw

      parameter    ( lrw = 22 + maxneq * 16 )
      real*8         rwork(lrw)

      integer        liw
      parameter    ( liw = 20 + maxneq )
      integer        iwork(liw)
      integer        jt

      real*8         tol
      real*8         default_tol
      parameter    ( default_tol = 1.0d-6  )
      real*8         dist

c-------------------------------------------------------------
c     Other locals
c-------------------------------------------------------------
      integer        a,             i,             j,
     &               itout,         glbpp

c-------------------------------------------------------------
c     Parse command line arguments (initial values) ...
c-------------------------------------------------------------
      if( iargc() .lt. 2 ) go to 900

      tfinal = r8arg(1,r8_never)
      dtout  = r8arg(2,r8_never)
      tol    = r8arg(3,default_tol)
c-------------------------------------------------------------
c     Set the trace
c-------------------------------------------------------------
      trace  = 1

      if( tfinal .eq. r8_never  .or.  dtout .eq. r8_never
```

```
     &    .or. dtout .le. 0.0d0 ) go to 900
c------------------------------------------------------------
c     Compute number of uniform time steps requested.
c------------------------------------------------------------
      ntout = tfinal / dtout + 1.5d0

c------------------------------------------------------------
c     Get particle diamters, initial positions and
c     velocities.
c------------------------------------------------------------
      call getivs('-',m,y,d,maxnp,np)
      if( np .eq. 0 ) then
         write(0,*) cdnm//': No particles'
         stop
      end if
      neq = 6 * np
      r0 = m(1)*(2**(1.0/6.0))

c------------------------------------------------------------
c     Output the initial diameters and positions
c------------------------------------------------------------
      write(*,*) np
      do i = 1 , np
         write(*,*) m(i)
      end do
      tbgn = 0.0d0
      write(*,*) tbgn
      write(*,*) -1
      write(*,*) -1
      do i = 1 , np
         write(*,*) y(1,1,i), y(2,1,i), y(3,1,i)
      end do

c-----------------------------------------------------------
c     Compute initial kinetic, potential and total energy
c-----------------------------------------------------------
      imomx = 0
      imomy = 0
      if( trace) then
         call energy(y, m, iKE, iPE, d, np)
         do i = 1,np
            imomx = imomx + y(1,2,i)
            imomy = imomy + y(2,2,i)
         end do
         write(0,*) 'iE =',iKE + iPE
      end if
c------------------------------------------------------------
c     Set LSODA parameters ...
c------------------------------------------------------------
      itol   = 1
      rtol   = tol
      atol   = tol
      itask  = 1
      iopt   = 0
      jt     = 2

c------------------------------------------------------------
c     Do the integration ...
c------------------------------------------------------------
      do itout = 2 , ntout

c------------------------------------------------------------
c        Set next output time.
c------------------------------------------------------------
         tend = tbgn + dtout

c------------------------------------------------------------
c        Integrate to the next output time.
```

```
c-----------------------------------------------------------
        istate = 1
        call lsoda(fcn,neq,y,tbgn,tend,
     &             itol,rtol,atol,itask,
     &             istate,iopt,rwork,lrw,iwork,liw,jac,jt)
c-----------------------------------------------------------
c       Check for LSODA error and exit if one occured.
c-----------------------------------------------------------
        if( istate .lt. 0 ) then
           write(0,1500) cdnm, istate, itout, ntout,
     &                   tbgn, tend
1500       format(/' ',a,': Error return ',i2,
     &             ' from integrator LSODA.'/
     &             '       At output time ',i5,' of ',i5/
     &             '       Interval ',1pe11.3,' .. ',
     &            1pe11.3/)
           go to 950
        end if


c-----------------------------------------------------------
c       Output time and particle coordinates.
c-----------------------------------------------------------
        write(*,*) tend
        write(*,*) -1
        write(*,*) -1
        do i = 1 , np
           write(*,*) y(1,1,i), y(2,1,i), y(3,1,i)
        end do

        if( trace) then
           momx = 0
           momy = 0
           do i=1,np
              momx = momx + y(1,2,i)
              momy = momy + y(2,2,i)
           end do
c-----------------------------------------------------------
c     Output the energy and momentum info to stderr
c-----------------------------------------------------------
           call energy(y, m, KE, PE, d, np)
           write(0,5000) tend, ((KE + PE)/(iKE+iPE))*100.0
           write(0,*) 'pi - p = (',imomx - momx,', ',imomy-momy,')'
           write(0,*) momx
        end if
c-----------------------------------------------------------
c     End of integration loop.
c-----------------------------------------------------------
      end do

500   continue

      stop

c-----------------------------------------------------------
c     Usage exit.
c-----------------------------------------------------------
900   continue
        write(0,*) 'usage: '//cdnm//
     &             ' <t final> <dt out> [<tol> <trace>]'
        write(0,*) ' '
        write(0,*) '       Masses, initial positions and'
        write(0,*) '       velocities of particles read from'
        write(0,*) '       standard input'
      stop
5000  format('time =', F6.3,'   (E/iE)*100 =', F8.3)
c-----------------------------------------------------------
c     LSODA error exit.
```

```
c-----------------------------------------------------------
950   continue
         write(0,*) 'nbody: Error return from LSODA'
         stop
      stop

      end

c===========================================================
c     Calls auxiliary
c     routine 'fcn1' which interprets 'y' and 'ydot'
c     as three dimensional arrays for coding convenience,
c     and which actually implements equations of motion.
c===========================================================
      subroutine fcn(neq,t,y,ydot)
         implicit    none

         integer     neq

         real*8      t,     y(neq),    ydot(neq)

         call fcn1(y,ydot,neq/6,t)

         return
      end

c===========================================================
c        Implements the Lennard-Jones inter-particle
c        potential.  Oscillations and gravity as well.
c
c        First  index:  dimension (coordinate)
c        Second index:  positions (1) or velocities (2)
c        Third  index:  particle number
c===========================================================
      subroutine fcn1(y,ydot,npart,t)
      implicit    none

      include    'fcn.inc'

      integer     neq,            lossy

      integer     npart
      real*8      y(3,2,npart),   ydot(3,2,npart)
      real*8      t
      integer     a,              i,              j
      real*8      dist,           cforce,             r0
      real*8      dforce
      real*8      freq,           amp
      real*8      g
      integer plain
      plain = 1
      g = 2.0d0
      dforce = 0.0d0
      cforce = 0.0d0


      r0 = m(1)*(2**(1.0d0/6.0d0))
      lossy = 1
      freq = 5.0d0
      amp = 0.25d0
c-----------------------------------------------------------
c     Implement equations which define velocities,
c     and initialize accelerations to 0.
c-----------------------------------------------------------
      do i = 1 , npart
         do a =  1 , d
            ydot(a,1,i) = y(a,2,i)
            if( plain .ne. 1) then
```

```
            if( a .eq. 2) then
               ydot(a,2,i) = -g + amp*(freq**2)*cos(freq*t)
            else
               ydot(a,2,i) = 0
            end if
         else
            ydot(a,2,i) = 0
         end if
      end do
   end do

c-----------------------------------------------------------
c     Loop over pairs of particles, compute pair-wise
c     forces and update accelerations of *both*
c     particles affected by each force.
c-----------------------------------------------------------
      do i = 1 , npart
         do j = i+1 , npart
c-----------------------------------------------------------
c     Compute separation
c-----------------------------------------------------------
            dist = 0.0d0
            do a = 1 , d
               dist = dist + (y(a,1,j) - y(a,1,i))**2
            end do
            dist = sqrt(dist)
            if(dist .lt. r0) then
               cforce = -(6*(m(1)**6))/(dist**8) +
     &                 (12*(m(1)**12))/(dist**14)
               do a =1,d
                  dforce = -gamma*(y(a,1,i)-y(a,1,j))*(y(a,2,i)
     &                 -y(a,2,j))/(dist**2)
               end do
c-----------------------------------------------------------
c     Update accns, making use of F_ji = - F_ij
c-----------------------------------------------------------

               do a = 1 , d
                  ydot(a,2,i) = ydot(a,2,i) +
     &                 cforce * (y(a,1,i) - y(a,1,j))
     &                 -dforce*(y(a,1,i)-y(a,1,j))
                  ydot(a,2,j) = ydot(a,2,j) +
     &                 cforce * (y(a,1,j) - y(a,1,i))
     &                 -dforce*(y(a,1,j)-y(a,1,i))
               end do
            end if
         end do
c-----------------------------------------------------------
c     Walls
c-----------------------------------------------------------
         do j = 1,2
            if((10-y(j,1,i)) .le. r0/2) then
               dist = r0/2 + 10 - y(j, 1,i)
               cforce = -6*(m(1)**6)/(dist**7)+12*(m(1)**12)/(dist**13)
               ydot(j,2,i) = ydot(j,2,i) - cforce
               dforce = gammaw*dist*y(j,2,i)/(dist**2)
               if( lossy .eq. 1) then
                  ydot(j,2,i) = ydot(j,2,i) - dforce*dist
               end if
            end if
            if(y(j,1,i) .le. r0/2) then
               dist = r0/2 +y(j,1,i)
               cforce = -6*(m(1)**6)/(dist**7)+12*(m(1)**12)/(dist**13)
               ydot(j,2,i) = ydot(j,2,i) + cforce
               dforce = gammaw*dist*y(j,2,i)/(dist**2)
               if(lossy .eq. 1) then
                  ydot(j,2,i) = ydot(j,2,i) - dforce*dist
               end if
```

```
            end if
         end do

      end do

      return

      end

c===========================================================
c     Dummy Jacobian routine.
c===========================================================
      subroutine jac
         implicit    none

         include    'fcn.inc'

         return
      end

c===========================================================
c     Reads the diameters and intial conditions of the
c     particles from a file
c===========================================================
      subroutine getivs(fname,m,y0,d,maxnp,np)
      implicit         none

      integer          indlnb,      getu

      character*(*)    fname
      integer          maxnp,       np,         d
      real*8           m(maxnp),    y0(d,2,maxnp)

      integer          ustdin
      parameter      ( ustdin = 5 )

      real*8           mi,          x(3),        v(3)
      integer          ufrom,       rc,          i

      np = 0
      if( fname .eq. '-' ) then
         ufrom = ustdin
      else
         ufrom = getu()
         open(ufrom,file=fname(1:indlnb(fname)),
     &        form='formatted',status='old',iostat=rc)
         if( rc .ne. 0 ) then
            write(0,*) 'dvfrom: Error opening ',
     &           fname(1:indlnb(fname))
            return
         end if
      end if

 100  continue
      read(ufrom,*,iostat=rc,end=200) mi, x, v
      if( rc .eq. 0 ) then
         np = np + 1
         if( np .gt. maxnp ) then
            write(0,*)  'getivs: Read maximum of ',
     &           maxnp, ' from ',
     &           fname(1:indlnb(fname))
            np = maxnp
            go to 200
         end if
         m(np) = mi
         do i = 1 , d
            y0(i,1,np) = x(i)
            y0(i,2,np) = v(i)
```

```
         end do
      end if
      go to 100
 200  continue

      if( ufrom .ne. ustdin ) then
         close(ufrom)
      end if

      return

      end


c---------------------------------------------------
c     routine to compute kinetic and potential
c     energies in the system
c---------------------------------------------------
      subroutine energy(y, m, KE, PE, d, np)
      implicit none
      integer d, np
      real*8 y(d,2,np), m(np)
      real*8 KE, PE
      integer i, j, a
      real*8 dist
      real*8 r0

      r0 = m(1)*(2**(1.0/6.0))
      KE = 0
      PE = 0
      do i=1,np
         do a=1, d
            KE = KE + (y(a,2,i)**2)
         end do
         do j=1+1,np
            if(i .ne. j) then
               dist = 0
               do a = 1,d
                  dist = dist + (y(a,1,j)-y(a,1,i))**2
               end do
               dist = sqrt(dist)
               if(dist .le. r0) then
                  PE = PE +(m(1)/dist)**12 - (m(1)/dist)**6
     &                 + 1.0/4.0
               end if
            end if
         end do
      end do
      KE = KE/2
      return
      end
```

B.2 fcn.inc

```
c===========================================================
c     fcn.inc
c
c     Application specific common block for communication with
c     derviative evaluating routine 'fcn' (optional) ...
c===========================================================
      integer        maxnp
      parameter    ( maxnp = 1 000 )

      integer        d
      parameter    ( d = 3 )

      real*8         m(maxnp)
      integer        np
```

```
      real*8         gamma
      real*8         gammaw
      parameter    ( gamma = 100.0d0)
      parameter    ( gammaw = 1.0d0)


      common / com_fcn /
     &               m,
     &               np
```