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

1 Introduction 2

2 Theory and Implementation of the Event Driven Simulation 2
   2.1 The Fast Event Driven Algorithm of Marin, Risso and Cordero 3
   2.2 Description of the LMA and DSA 4
   2.3 Pseudo-code for the Event Driven Simulation 7
   2.4 Collision Modelling in the Event Driven Simulation 9

3 Results From the Event Driven Simulation 10

4 Theory and Implementation of the Time Driven Simulation 13
   4.1 Equations of motion for the Time Driven Simulation 13
   4.2 The Isoda ODE Integrator 15

5 Results From the Time Driven Simulation 15

6 Conclusion 17

7 Acknowledgements 17

A Event Driven Simulation Code 18
   A.1 Main Source Code for the Event Driven Algorithm: edriven.c 18
   A.2 blits.c 29
   A.3 eventList.c 29
   A.4 intList.c 32

B Time Driven Simulation Code 33
   B.1 hpart.f 33
   B.2 fcn.inc 40

List of Figures

1 A schematic diagram of a complete binary tree 4
2 An example of the algorithm that picks the minimum event from the FEL 5
3 Elastic collision model in the event driven simulation 10
4 Execution time of the event driven simulation for various number of particles 11
5 Initial conditions for the N=1000 and N=2000 particle simulation 11
6 Final positions of the N=1000 and N=2000 inelastic particle simulation 12
7 Final position for the N=1000 and N=2000 elastic particle simulation 12
8 A plot of the Lennard-Jones potential 14
9 Initial and final conditions for the N=20 particle simulation with oscillations and gravity 16
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\cite{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\cite{1}. If the particles are non-cohesive, the inter-particle forces are purely repulsive\cite{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\cite{2,3,4,5} and size segregation\cite{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\cite{7}. In contrast to the success of the engineers, physicists have not yet developed any theories that adequately describe granular materials\cite{1}. A theory that describes granular materials well would have important practical applications, such as those mentioned above.

Bizon et al.\cite{4} have simulated pattern formations in 3D vertically oscillated granular materials. The patterns found in their simulations match the patterns found in experiments\cite{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.\cite{4}.

There are two main approaches to simulating systems of many hard particles: time driven simulations and event driven simulations\cite{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\cite{8}, which Bizon et al.\cite{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, \(\tau_i\), included with it’s state. \(\tau_i\) 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 \(\tau\) 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 the 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 its 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 2*P* − 1. Each particle in the simulation is associated with a leaf of the *FEL*.

![Binary Tree Diagram](#)

**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 0th 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*; and *lml*, so that a new event will be picked next time through the cycle.

4
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 its sibling leaf and the winner is advanced to the parent node. The parent node is then compared to its 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 \( lmi \) 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:

\[
S_i = (\vec{r}_i, \vec{v}_i, \tau_i, c_i)
\]

(1)

\( \vec{r}_i \) is the position vector. \( \vec{v}_i \) is the velocity vector. \( \tau_i \) 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:

\[
\xi = (t, coltype, partner, c)
\]

(2)

\( t \) is the time at which the event will occur. \( coltype \) is the type of collision (DDC, DWC or VWC). \( partner \) is the disk, wall or virtual wall that the disk will encounter. If \( coltype = DDC, \) then \( c \) is the \( c_{\text{partner}} \) 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, $\infty$ is returned.
- **scheduling(udisk, time, colltype, partner, cpartner):** 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.
- **$\xi = local_{min}(local_i):$** 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*.

```c
begin MAIN FUNCTION
  initialize all positions and velocities randomly
  initialize all $\tau_i$ 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 - $\tau_{disk}$)
      freev(partner, time - $\tau_{partner}$)
  END LOOP
  END MAIN
```

7
diskев(disk, partner)
predictions(disk, colltype)
predictions(partner, colltype)
BREAK
CASE DWC:
freev(disk, time - τ\text{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\_cgt(disk)
IF(colltype == DDC)
   update\_cgt(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\_cgt(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:

\[
\vec{v}_{i_{new}} = \vec{v}_{j_{norm}} + \vec{v}_{j_{tang}}
\]

\[
\vec{v}_{j_{new}} = \vec{v}_{i_{norm}} + \vec{v}_{i_{tang}}
\]

Let \( \hat{r} \) be the unit vector that points from disk \( i \) to \( j \), then \( \vec{v}_{i_{norm}} \) and \( \vec{v}_{j_{norm}} \) are the components of the velocities, prior to the collision, parallel to \( \hat{r} \). \( \vec{v}_{i_{tang}} \) and \( \vec{v}_{j_{tang}} \) 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, \( \mu \). The final velocities of two disks suffering an inelastic collision are:

\[
\vec{v}_{i_{new}} = \mu \vec{v}_{j_{norm}} + \vec{v}_{j_{tang}}
\]

\[
\vec{v}_{j_{new}} = \mu \vec{v}_{i_{norm}} + \vec{v}_{i_{tang}}
\]

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 \( \mu \) 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.
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, \( \mu \) 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 \( \mu \) 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: $D = 0.1$, $N_{x \text{CELLS}} = N_{y \text{CELLS}} = 10$. 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, $\vec{F} = m\vec{a} = m\frac{d^2\vec{r}}{dt^2}$. 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:

$$\frac{\partial \vec{v}}{\partial t} = \vec{a}$$  \hspace{1cm} (7)

$$\frac{\partial \vec{r}}{\partial t} = \vec{v}$$  \hspace{1cm} (8)

The force acting on each particle, $\vec{F}_i$, 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:

$$\vec{f}_{ij} = -\frac{\partial V_{ij}}{\partial \vec{r}_{ij}}$$  \hspace{1cm} (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:

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

(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:

$$\vec{f}_i = -\gamma m (\vec{v}_{ij} \cdot \vec{r}_{ij}) \frac{\vec{r}_{ij}}{|\vec{r}_{ij}|^2} \text{ for } |\vec{r}_{ij}| \leq 2^{1/6} \delta$$

(11)

Where $\vec{v}_{ij} = \vec{v}_j - \vec{v}_i$, and $\gamma$ 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 Isoda ODE Integrator

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

The first argument to Isoda is the address of a function that will set the equations of motion. This function must be of the form: \texttt{fcn(neq,t,y,ydot)}. \texttt{neq} is the number of equations that Isoda has to integrate. \texttt{t} is the time that Isoda starts integrating from. In my simulations, all potentials are time independent, so \texttt{t} is not used at all. \texttt{y} is the array that stores position and velocity information. \texttt{ydot} is an array where the ODEs are specified. As seen in equations 7-11, the derivatives, \texttt{ydot}, are dependent on the positions and velocities of the particles, therefore \texttt{ydot} is specified as a function of the various components of \texttt{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 Isoda. $T_a$ is the absolute tolerance, while $T_r$ is the relative tolerance. While integrating the equations, Isoda evaluates a local error, $e_i$, for each $y_i$. Isoda ensures that $e_i < T_r \ abs(y_i) + T_a$ during the integration.

Other arguments supplied to Isoda are: $y$, \texttt{neq}, \texttt{t_i} and $t_f$, $T_r$. $y$ and \texttt{neq} are the same variables that Isoda passes into \texttt{fcn}. $t_i$ specifies the initial time that the system is at when Isoda is called. $t_f$ is the time that Isoda 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, $4\pi^2 f^2 A \cos(2\pi f t) \ddot{y}$, 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, $-g\ddot{y}$, 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/~wilson/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 $\gamma = 1.0$.](image)

The time driven simulation is sensitive to the time step and tolerance passed to Isoda. 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 Isoda 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 believe 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

[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 <stdio.h>
#include "blibs.h"
#include "eventlist.h"
#include "intlist.h"

const double PI = 3.1415926536; // Diameter of the particles
const double D = 0.09; // Inelastic collision parameter
int N_LOOPS = 1000000000; // # of times through the main loop
const size_t N_PARTICLES = 300; // # of particles
const int N_CELLS = 10; // The # of cells in the x and y direction
const int N_CELLS = 10; // The width of the x and y cells
const double X_SPACING = 1; // The size of the domain in
const double Y_SPACING = 1; // (X_CELLS*Y_SPACING, Y_CELLS*Y_SPACING)

int random_input = 1; // if (random_input) assign IC’s randomly
int time = 1; // if (time) coordinates are output at fixed time steps
// (time steps, dt)
double dt = 0.1; // variable used in discrete time output
const double finish_time = 1000; // if (dt) simulation will stop at finish_time
int flag = 1; // flag for data output
```
// Definition of event types
const int DCI = 1; // Disk-Disk collision
const int DVI = 2; // Disk-Vall collision
const int VCI = 3; // Virtual Vall collision
const int SLI = -1; // Error
double parts[3_PARTICLES][3]; // 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 CELLMATRX_LENGTH = 100; // max # of disks in a cell
const int NCELLS = 0*CELLMATRX_LENGTH; // max # of neighbours a disk can have
int cellmatrx[N_XCELLS][N_YCELLS][CELLMATRX_LENGTH]; // the matrix that keeps track of
// what cell the disks are in

eventNode **localList; // stores all potential events for each disk
int CLI_PARTICLES*N_XCELLS*N_YCELLS+2; // how many hard collisions are associated
// with each particle.
int FEL[2*M_PARTICLES+1]; // Future Events List
Event *elm; // 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 pl, int p2)
{ double a, b, c;
  double pos1[2], pos2[2];
  int i;
  a=0;
  b=0;
  c=0;
  // Values needed to decide whether a collision will happen or not
  // pos1[] and pos2[] array calculation
  for(i=0;i<=i++)
  { pos1[i]=parts[pl][i]+parts[pl][i+2]*{max(parts[pl][i],parts[pl][i+2]−parts[pl][i+2]);
    pos2[i]=parts[p2][i]+parts[p2][i+2]*{max(parts[p2][i],parts[p2][i+2]−parts[p2][i+2]);
  }
  // calculation of b
  for(i=0;i<=i++)
  { b+=pos2[i]+pos1[i][i+2][i+2];
  }
  // calculation of a
  for(i=0;i<=i++)
  { a+=pos2[i][i+2][i+2][i+2];
  }
  // calculation of c
  for(i=0;i<=i++)
  { c+=pos2[i][i+2];
  }
  c=c=0;
  // Decide if the collision will happen or not
  if(b<0&&b−a==0)
  { double time;
    time תורהמ{b−sqrt((b−a))/a};
    if(time<0)
    { cont<<"Negative time for ball-ball collision!\n";
      cont<<"Particle1: \"<<pos1[0]<<","<<pos1[1]<<"\n";
      cont<<"v1: \"<<v1<<","<<v2<<"\n";
      cont<<"t1: \"<<t1<<","<<t2<<"\n";
      cont<<"Particle2: \"<<pos2[0]<<","<<pos2[1]<<"\n";
      cont<<"v2: \"<<v2<<","<<v3<<"\n";
      cont<<"t2: \"<<t1<<","<<t2<<"\n";
    }
  }
cout<<"\ttau: ",parts[p2][4]<<"\n";
cout<<"\n";
cout<<"\n";
exit(1);
}
return time=max(parts[p1][4],parts[p2][4]);
}
else
return FILE_VAL;
}

// Searches the cell matrix for a particle, i
void search_cell_matrix(int idisk, int cell[0]){  
   cell[0]=-1;  
   cell[1]=-1;  
   for(int i=0;i<NUM CELLS;i++) {  
       for(int j=0;j<NUM CELLS;j++) {  
           if(int_search(cell_matrix[idisk[j],CELL_MATRIX_LENGTH])==1) {  
               cell[0]=i;  
               cell[1]=j;  
           }  
       }  
   }  
   if((cell[0]==-1) || (cell[1]==-1)) {  
       cout<<"An invalid disk number was passed to search_cell_matrix!\n";
       cout<<"Invalid disk number is: "<<disk<<"\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 t.wall(int part, int &col_type, 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, n+n, n+n)  
   // The walls along the y-axis first  
   // only consider the right wall  
   if(right){  
       wall_place=(cell[0]+1)*M_SPACING;  
       col_type=DWC;  
       wall=NUM CELLS+NUM_PARTICLES;  
       time=(NUM CELLS+1)*M_SPACING-parts[part][0]-parts[part][2];  
       if((((wall_place-parts[part][0])/parts[part][2])<time){  
           col_type=VWC;  
           time=(wall_place-parts[part][0])/parts[part][2];  
           wall=cell[0]+NUM_PARTICLES;  
       }
   }
}
// only consider the left wall
else
    wall_place=double(cell[0])*x_SPACING;
coltype=SVC;
time=0.0-parts[part][0]+0.0/parts[part][0];
wall=N_PARTICLES;
if((wall_place<parts[part][0]) && (wall_place>parts[part][2])
    coltype=WVC;
time=wall_place+parts[part][0]/parts[part][0];
wall=cell[0]=N_PARTICLES;
}

// walls along x-axis next
// only consider the top wall
if(top){
    if((right.Euler%N_PARTICLES-start.Euler%N_PARTICLES<parts[part][0]-parts[part][0])
        coltype=SVC;
    wall_place=double(cell[1]);
    if((wall_place<parts[part][1]) && (wall_place>parts[part][3])
        coltype=WVC;
    wall=cell[1]=N_PARTICLES+N_CELLs+2;
}
// only consider the bottom wall
else
    if((left.Euler%N_PARTICLES-start.Euler%N_PARTICLES<parts[part][0]-parts[part][0])
        coltype=SVC;
    wall_place=double(cell[1]);
    if((wall_place<parts[part][1]) && (wall_place>parts[part][3])
        coltype=WVC;
    wall=cell[1]=N_PARTICLES+N_CELLs+2;
}
if(t<
    cout<<\n"trouble calculating the wall collision time!\n";
    cout<<\n"particle "<<part<<\n";
    cout<<\n"position ("<<parts[part][0]<<<","<<parts[part][1]<<<") <<";
    cout<<\n"velocity: ("<<parts[part][2]<<<","<<parts[part][3]<<<") <<";
    cout<<\n"time: "<<time<<\n";
    cout<<\n"wall: "<<wall<<\n";
    if(wall<<\n"wall position: x = "<<(wall=N_PARTICLES-N_CELLs)<<\n";
else{
    cout<<\n"wall position: y = "<<(wall=N_PARTICLES-N_CELLs-1)*x_SPACING<<\n";
}
    cout<<\n"\n";
    exit(1);
    return time>>parts[part][4];
}
// Free evolution of update to time tau plus tplus, then update tau
void free(int update, double tplus){
    for(int i=0;i<
        parts[update][i]=parts[update][i]+tplus+parts[update][i+2];
}
}

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

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

// makes instantaneous changes suffered by hard collision of validisk with wall
void waller(int wallid, int wall)
{
    if (wall == int(S_PARTICLES))
    { parts[wallid][2] = parts[wallid][2];
    }
    else if (wall == int(S_PARTICLES + S_XCELLS) || wall == int(S_PARTICLES + S_XCELLS + 1))
    { parts[wallid][3] = parts[wallid][3];
    }
    else
    { cout << "In function waller, an invalid wall number was passed to the function! \n";
        exit(1);
    }

    // The collision of disk1 and disk2 is modeled in this routine.
    void diskcoll(int disk1, int disk2)
    {
        double v1norm[2];
        double v1proj;
        double v2norm[2];
        double v2proj;
        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 += pow(parts[disk2][i] - parts[disk1][i], 2);
            v1norm[i] = v2norm[i] = v1tang[i] = v2tang[i] = 0;
            dist = sqrt(dist);
            for (j = 0; j < 2; j++)
            { norm[j] = (parts[disk2][j] - parts[disk1][j]) / dist;
                v1proj = 0;
                v2proj = 0;
                for (i = 0; i < 2; i++)
                { v1norm[i] = v1proj + parts[disk1][i + 2] * norm[i];
                    v2norm[i] = v2proj + parts[disk2][i + 2] * norm[i];
                    v1tang[i] = parts[disk1][i + 2] - v1norm[i];
                    v2tang[i] = parts[disk2][i + 2] - v2norm[i];
                    parts[disk1][i + 2] = parts[disk1][i + 2] + OEFF * v1tang[i] * v2norm[i];
                    parts[disk2][i + 2] = parts[disk2][i + 2] + OEFF * 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]=i-1;
    cell_of_idisk[1]=i+1;
    search_cell_matrix(idisk, cell_of_idisk);
    for(int j=cell_of_idisk[0];j<cell_of_idisk[1];j++){
        for(int i=cell_of_idisk[0];i<cell_of_idisk[1];i++){
            if(i^k==0) && (j^k==1) && (j<(N_CELLS-1)){
                int_union(narr, cellmatrix[i][j], NARR_LENGTH, CELMATRIX_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_1;
        double min_time;
        fel_1=2*M_PARTICLES-2;
        // Can treat arrays now as having an even number of leaves
        while(fel_1>0){
            min_time=min(fel[fel_1].scheduled_time, fel[fel_1+1].scheduled_time);
            if(min_time==fel[fel_1].scheduled_time){
                fel=(fel_1-2)/2+1=fel_1;
                }
            else{
                fel_1=fel_1+1;
                }
            }
        // 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(lm[disk].event_type==DDC){
                if(lm[disk].c==0(lm[disk].partner)){
                    event_list_remove(localList[disk], lm[disk].scheduled_time);
                    localMin(localList[disk]);
                    }
                }
            cbtp=M_PARTICLES*disk-1;
            // cbtp is odd
            while(cbtp>0){
                if(cbtp%2){
                    mint=min(fel[cbtp].scheduled_time, fel[cbtp+1].scheduled_time);
                    if(mint==fel[cbtp].scheduled_time){
                        fel((cbtp-1)/2+1=fel[cbtp]);
                        }
                    else{
                        fel((cbtp-1)/2+1=fel[cbtp+1]);
                        }
                    cbtp=(cbtp-1)/2;
                    }
                // cbtp is even
                else{
                    mint=min(fel[cbtp].scheduled_time, fel[cbtp-1].scheduled_time);
                    cbtp=(cbtp+1)/2;
                    }
        // cbtp is even
        }
    // cbtp is even
    else{
        mint=min(fel[cbtp].scheduled_time, fel[cbtp-1].scheduled_time);
        cbtp=(cbtp+1)/2;
        }
if(mint==lml[FEL[cbtp]].scheduled_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($PARTICLES))){
      update_cbt(partner);
    } else{
      for(disk=lml[FEL[0]].partner;
          event_type=lml[FEL[0]].event_type;
          time=lml[FEL[0]].scheduled_time;
          // event is invalid, find another one
          if((event_type==ODC)&&(lml[disk].c!=O[partner])){
            update_cbt(disk);
          } else if((event_type==WOC)&&(event_type==DOC)){
            event_list_clear(localList[disk]);
            C[disk]++;
            if(event_type==DOC){
              event_list_clear(localList[partner]);
            } go=0;
          } else if(event_type==WOC){
            event_list_remove(localList[disk], time);
            go=0;
          } while(go);
      } if(event_type==DOC){
        C[partner]++;
      }
      return time;
    } // schedules an event into the local list of disk
    void schedule(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 schedule(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 marr[BARN_LENGTH];
  int i = 0;
  event_list_clear(localList[disk]);
  int Clear(marr, BARN_LENGTH);
  neighborhood(marr, disk);
  while(marr[i] != -1){
    time=ball(disk, marr[i]);
    if(time == huge_val){
      schedule(disk, time, IDC, marr[i], C[marr[i]]);
    }
    i++;
  }
  time=wall(disk, event_type, partner);
  schedule(disk, time, event_type, partner);
  int idisk=localMn(localList[disk]);
}

// moves disk through the wall into a new cell in the cellmatrix
void update(int disk, int wall){
  int cell[2];
  int vert=0;
  search_cell_matrix(disk, cell);
  int remove(cellmatrx[cell[0]][cell[1]], disk, CELLMATRIX_LENGTH);
  // is the virtual wall a top wall or bottom wall
  if(wall>int((N_PARTICLES*W_CELLS))){
    vert=1;
  } else{
    vert=0;
  }
  if(vert){
    if(wall>(cell[1]+int(W_PARTICLES*W_CELLS))){
      // ball passed through the top wall
      int_insert(cellmatrx[cell[0]][cell[1]+1], disk, CELLMATRIX_LENGTH);
    } else{
      // ball passed through the bottom wall
      int_insert(cellmatrx[cell[0]][cell[1]-1], disk, CELLMATRIX_LENGTH);
    }
  } else{
    if(wall>(cell[0]+int(W_PARTICLES))){
      // ball passed through the right wall
      int_insert(cellmatrx[cell[0]+1][cell[1]], disk, CELLMATRIX_LENGTH);
    } else{
      // ball passed through the left wall
      int_insert(cellmatrx[cell[0]-1][cell[1]], disk, CELLMATRIX_LENGTH);
    }
  }
}

// non-random position assignment
void pos_vel_assign(){
  double pos[2];
  pos[0]=2*double(0);
  pos[1]=2*double(0);
  for(int i=0;i<int(W_PARTICLES);i++){
    for(int j=0;j<w;i++){
      parts[i][j]=pos[j];
      ...
```
parts[i][j]=1.0-2.0*(double(rand()))/double(RAND_MAX));
}

pos[0]=pos[0]+2*double(D);
if(pos[0]>=(double(S_XCELLS)*X_SPACING)-2*double(D)) {
    pos[0]=double(D);
    pos[1]=pos[1]+2*double(D);
}
if(pos[1]>=(double(S_YCELLS)*Y_SPACING)-2*double(D)) {
    fprintf(stderr,"Not enough space for these particles\n"");
    fprintf(stderr,"\n\n");
    exit(1);
}
parts[i][j] = 0;
}

// check to see if the disks overlap from the random position 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<0) {
            ok=0;
        }
        if((parts[disk][0]<(D/2)) || (parts[disk][0]>(S_XCELLS*X_SPACING-D/2))) {
            ok=0;
        }
        if((parts[disk][1]<(D/2)) || (parts[disk][1]>(S_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));
            }
            check(i, checked);
        }
        parts[i][i] = 0;
        fprintf(stderr,"initialized particle %d\n", i);
    }
}

// ..................................................................................
// *** *** *** MAIS FUNCTION *** *** ***
// ..................................................................................
int main(){
    // ..................................................................................
    // INITIALIZE THE VARIABLES
    // ..................................................................................
    if(int(4*X_SPACING*Y_SPACING/PI>pow(3,2)+1) > MAIS_LENGTH) {
        fprintf(stderr,"Problems with neighbourhood size: \n"");
        fprintf(stderr,"\nIncrease MAIS_LENGTH\n");
        fprintf(stderr,"\nIncrease 3 of\n");
        fprintf(stderr,"\n\nIncrease X_SPACING and Y_SPACING\n");
        cout<<"Exiting Program\n";
```
exit(0);
}
if(localList=(eventNodePtr *)malloc(sizeof(eventNodePtr))!=NULL){
    fprintf(stderr,"Couldn't allocate memory for the localList array\n"");
    fprintf(stderr,"\n"");
    exit(1);
}
if(lml=(Event *)malloc(sizeof(Event))!=NULL){
    fprintf(stderr,"Couldn't allocate memory for the Local Minima List\n"");
    fprintf(stderr,"\n"");
    exit(1);
}
double time;
int disk;
int partner;
int coll_type;
int wall;
int marr[NARR_LENGTH];
int i, j;
time=-1;
disk=-1;
partner=-1;
coll_type=NULL;
int clear(marr, NARR_LENGTH);
if(random_assign){
    rand_poe_rel_assign();
} else{
    poe_rel_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 xPos;
    int yPos;
    intPos=int(part[i][X]/X_SPACING);
    intPos=int(part[i][Y]/Y_SPACING);
    int insert(cellmatrix[intPos][intPos],i, CELLMATRIX_LENGTH);
} //build the C matrix
for(i=int(N_PARTICLES);i<int(N_PARTICLES+N_XCELLS+N_YCELLS+2);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(marr,i);
    j=0;
    while(marr[j]!=-1){
        time=wall(i,marr[j]);
        if(time!=NONE_VAL){
            head_insert_event(localList[i], time, DDC, marr[j], C[marr[j]]);
        }
        j++;
    }
} // 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+i]=1;
}

// ...........................................................................

// # # # # Data output # # # # #

// # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # 

fprintf(stdout,"%d\n",N_PARTICLES);
for(i=0;i<int(N_PARTICLES);i++){
    fprintf(stdout,"%g\n",FEL);
}

fprintf(stdout,"0.0\n");
if(glbpp == 1){
    fprintf(stdout,"-1\n\n");
}
for(i=0;i<int(N_PARTICLES);i++){
    fprintf(stdout,"%g %g 0.0\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,dink,partner);

    // output the coordinates at discrete times
    if(dtme){
        pdtime = pdtime + dt;
        fprintf(stdout,"%g\n",pdtime);
        if(glbpp == 1){
            fprintf(stdout,"-1\n\n");
        }
    }
    for(j=0;j<int(N_PARTICLES);j++){
        fprintf(stdout,"%g\n",freez[j],pdtime-parts[j][4]);
        fprintf(stdout,"%g %g 0.0\n",freee[j],pdtime-parts[j][4]);
    }
}

if(pdtme>finish_time{
    i=N_LOOPS;
}
}

switch(coll_type){
    case DDC:
        freez(dink,time-parts[dink][4]);
        freez(partner,time-parts[partner][4]);
        dink(dink, partner);    
        predictions(dink, coll_type);
        predictions(partner, coll_type);
        break;

    case DVC:
        freez(dink, time-parts[dink][4]);
        waller(dink, partner);    
        predictions(dink, coll_type);
        break;

28
A.2 blibs.C

#include <stdio.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 <stdio.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 scheduled_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();
    }
    temp_ptr->data.scheduled_time=the_event.scheduled_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 scheduled_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();
    }
    temp_ptr->data.scheduled_time=scheduled_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.scheduled_time)<(returnEvent.scheduled_time)){
            returnEvent=tempPtr->data;
        }
    }
    return returnEvent;
}
// removes an event from the head of the list
void event_list_head_remove(eventNode* head){
    eventNode* 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 *temp;
    int got_it;
    got_it=0;
    if((head->data.scheduled_time)==time){
        event_list_head_remove(head);
        return;
    }
    temp = head;
    while(!got_it){
        previous=temp;
        temp=(temp->link);
        if((temp->data.scheduled_time)==time){
            (previous->link)=(temp->link);
            delete temp;
            got_it=1;
        }
    }
}

// searches for an event with partner in the list
eventNode* event_search(eventNode* head, int partner){
    eventNode* 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;
            }
        }
    return NULL;
}
A.4 intList.C

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

// 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(int 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(int 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(int i=0;i<arr_length;i++){
        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(int i=0;i<copy_length;i++){
        if(i<arr_length){
            copy[i]=arr[i];
        } else{
            copy[i]=-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

This code is heavily based on a gravitational nbody program written by Dr. Matt Choptuik:

The fcn1 routine was rewritten to implement the Lennard-Jones potential, and the energy routine was written to calculate the energy of the system.

program hpart
implicit none
character*5 cdnm
parameter ( cdnm = 'hpart' )
exteger iargc, indlnb, iarg
real*8 r0
real*8 r0_never, imomx, momx, imomy, momy
parameter  ( r0_never = -1.0d-60 )

---
c tfinal: Final integration time
dtout: Output interval
ntout: # of output times (computed)
trace: 1 to monitor conserved quantities
---
real*8 tfinal, dtout
integer ntout
logical trace
---
The cutoff for the Lennard-Jones potential
real*8 r0
---
include 'fcm.inc'

---
integer maxneq, neq
parameter ( maxneq = 6 * mxnp )
---
LSDA Variables.

external fcm, jac
real*8 y(4,2,maxnp)
real*8 thbn, tend
integer itol
real*8 rtol, atol
integer itask, istate, iopt
integer lrv
parameter ( lrv = 22 + maxneq * 16 )
real*8 rvork(lrv)
integer liv
parameter ( liv = 20 + maxneq )
integer ivork(liv)
integer jt
real*8 tol
real*8 default_tol
parameter ( default_tol = 1.0d-6 )
real*8 dist
---
Other locals
integer a, i, j,
& itout, gbpp
---
Parse command line arguments (initial values) ... if( iargc() .lt. 2 ) go to 900
tfinal = r0arg(1,r0_never)
dtout = r0arg(2,r0_never)
tol = r0arg(3,default_tol)
---
Set the trace
---
trace = 1
if( tfinal .eq. r0_never .or. dtout .eq. r0_never
& .or. dtout .le. 0.060 ) go to 900
     c Compute number of uniform time steps requested.
     c---------------------------------------------------------------------
     ntout = tfinal / dtout * 1.540
     c---------------------------------------------------------------------
     c Get particle diameters, initial positions and velocities.
     c---------------------------------------------------------------------
     call getrsv( '*', my, m, m, nxn, np )
     if( np .eq. 0 ) then
       write(0,*) cdsm//' : 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
     tbgm = 0.040
     write(*,*) tbgm
     write(*,*) -1
     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 LSDA parameters ...
     c---------------------------------------------------------------------
     itol = 1
     rtol = 10
     atol = 10
     itask = 1
     iopt = 0
     j = 2
     c---------------------------------------------------------------------
     c Do the integration ...
     c---------------------------------------------------------------------
     do itout = 2, ntout
     c---------------------------------------------------------------------
     c Set next output time.
     c---------------------------------------------------------------------
     tend = tbgm + dtout
     c---------------------------------------------------------------------
     c Integrate to the next output time.
istate = 1
    call linodal,meq,y,tbgn,tend,
    &itol,reltol,atol,iterlimit,
    &istate,iopt,ivode,lrw,lnw,jac,jlt

C Check for LSODA error and exit if one occurred.
    if ( istate .lt. 0 ) then
        write(0,i500) cdmm, istate, itout, ntout,
        &tbgn, tend
        format(' ',a10,': Error return ',i2,
        ', from integrator LSODA, ','
        ', At output time ',i5,,' of ',i5/
        ', interval ',1pe1.3,',' .. ','
        &1pe1.3/
        go to 950
    end if

C Output time and particle coordinates.
    write(*,*), tend
    write(*,*), -1
    write(*,*), -1
    do i = 1, mp
        write(*,*), y(i,1,i), y(2,1,i), y(3,1,i)
    end do
    if ( trace ) then
        momx = 0
        momy = 0
        do i = 1, mp
            momx = momx + y(1,2,i)
            momy = momy + y(3,2,i)
        end do
    end if

C Output the energy and momentum info to stderr.
    call energy(y, m, KE, PE, d, mp)
    write(0,i500) tend, (KE + PE)/(KE + PE)*100.0
    write(0,*), 'pi = p = (',imomx = momx, ', ',imomy = momy, ')
    write(0,*), momx
    end if

C End of integration loop.
    end do

500 continue
    stop

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

C LSODA error exit.
c---------------------------------------------
950 continue
   write(0,*) 'nbody: Error return from LSOFA'
   stop
   stop
end

c---------------------------------------------
c Calls auxiliary
routine 'fcm' which interprets 'y' and 'ydot'
as three dimensional arrays for coding convenience,
and which actually implements equations of motion.
c---------------------------------------------
subroutine fcm(neq,t,y,ydot)
   implicit none
   integer    neq
   real*8     t, y(neq), ydot(neq)
   call fcm(y,ydot,neq/6,t)
   return
end

c---------------------------------------------
c Implements the Lennard-Jones inter-particle
potential. Oscillations and gravity as well.
c---------------------------------------------
subroutine fcm(y,ydot,npart,t)
   implicit none
   include 'fcm.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, amp
   real*8     g
   integer    plain
   plain = 1
   g = 2.040
   dforce = 0.040
   cforce = 0.040

   r0 = m(1)*(2**(1.040/6.040))
   lossy = 1
   freq = 5.040
   amp = 0.2560

c---------------------------------------------
c Implement equations which define velocities,
c and initialize accelerations to 0,
c---------------------------------------------
do i = 1 , npart
   do a = 1 , d
      ydot(a,i) = y(a,2,i)
   if( plain .ne. 1) then
      
37
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 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 = 0.000
  do a = 1, d
    dist = dist + (y[a,1,j] - y[a,1,i])**2
  end do
  if(dist .lt. r0) then
    cforce = -((m[i]**2))/((dist**3)) +
            (12*(m[j]**2))/((dist**4))
  do a = 1,d
    dforce = -gamma*(y[a,1,i] - y[a,1,j])*y(a,2,i)
    cforce = (y[a,1,j] - y(a,1,i))
    dforce = (y[a,1,j] - y(a,1,i))
  end do
end if
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
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 = -((m[i]**2))/((dist**3)) +12*(m[j]**2))/((dist**4))
    ydot(j,2,i) = ydot(j,2,i) - cforce
    dforce = gamma*(dist**j)*y[j,2,i]/((dist**2))
    if( loamy .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 = -((m[i]**2))/((dist**3)) +12*(m[j]**2))/((dist**4))
    ydot(j,2,i) = ydot(j,2,i) + cforce
    dforce = gamma*(dist**j)*y[j,2,i]/((dist**2))
    if(loamy .eq. 1) then
      ydot(j,2,i) = ydot(j,2,i) - dforce*dist
    end if
  end if
end do
38
end if
end do
return
end

c-----------------------------------------------
c Dummy Jacobian routine.
c-----------------------------------------------
subroutine jac
  implicit none
  include 'fcm.inc'
  return
end

c-----------------------------------------------
c Reads the diameters and initial conditions of the
c particles from a file
c-----------------------------------------------
subroutine getivs(fname,m,y0,d,maxp,mp)
  implicit none
  integer indlnb, getu
  character(*) fname
  integer maxp, mp, d
  real*8 m(maxp), y0(d,2,maxp)
  integer ustdin parameter ( ustdin = 5 )
  real*8 mi, x(3), v(3)
  integer ufrom, rc, i

  mp = 0
  if( fname .eq. '-' ) then
    ufrom = ustdin
  else
    ufrom = getu()
    open(ufrom,file=fname(i:indlnb(fname)), &
    form='formatted',status='old',iostat=rc)
    if( rc .ne. 0 ) then
      write(0,*), 'Error opening ',
      & fname(1:indlnb(fname))
      return
    end if
  end if
  continue
  read(ufrom,*),iostat=rc,end=200 mi, x, v
  if( rc .eq. 0 ) then
    mp = mp + 1
    if( mp .gt. maxp ) then
      write(0,*), 'Read maximum of ',
      & maxp, ' from ',
      & fname(1:indlnb(fname))
    mp = maxp
  go to 200
  end if
  m(mp) = mi
  do i = 1, d
    y0(i,1,mp) = x(i)
    y0(i,2,mp) = v(i)
  end do
  continue
end do
end if
200 continue
if( ufrom .ne. utdin ) then
  close(ufrom)
end if
return
end

!----------------------------------------!
! routine to compute kinetic and potential !
! energies in the system                    !
!----------------------------------------!
subroutine energy(y, m, KE, PE, d, np)
  implicit none
  integer d, mp
  real*8 y(d,2,mp), m(mp)
  real*8 xe, pe
  integer i, j, a
  real*8 dist, r0
  r0 = m(1)**(2**((1.0/6.0))
  ke = 0
  pe = 0
  do i=1,mp
    do a=1,d
      xe = xe + (y(a,2,i)**2)
    end do
    do j=1,mp
      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
        end if
      end if
    end do
    xe = xe/2
  end do
return
end

B.2  fcn.inc

!----------------------------------------!
! fcn.inc                                       !
!----------------------------------------!
! Application specific common block for communication with    !
! derivative evaluating routine 'fcn' (optional) ...       !
!----------------------------------------!
integer maxnp
parameter ( maxnp = 1000 )

integer d
parameter ( d = 3 )
real*8 m(maxnp)
integer np
real*8 gamma
real*8 gammad
parameter (gamma = 100.000)
parameter (gammad = 1.000)

common / com_fcn /
& m,
& np