CEE/MAE M20
Introduction to Computer Programming with MATLAB

Aanish Patel Sikora
Final Project:
Amazing Collision Simulator!

December 11, 2014

SID: 804028077

Contents

List of Figures

Part I
Problem Statement

The goal of this project is to create a Basic Collision Dynamics Simulator. The project requires mastery of all of the programming concepts and techniques introduced during the quarter. The program itself simulates collisions between N particles and rectangular boundaries. Further functionality is introduced in the Bonus Features section.

The basic simulator can be broken up into 4 parts. The first part involves writing a collision detection function, using the UNIFORM GRID algorithm. The next part involves writing a function to determine collision times. The main script is written in part 3 and is responsible for time-stepping and resolving of collisions. Finally, in part 4, the simulation is created as a movie file.

The bonus features section lists a variety of extra technical and presentation features. For this project, I have implemented the following:

- Gravity ($\star$)
- Inelastic Collisions ($\star$)
- Brownian Dynamics ($\star\star$)
- Heterogenous Disk Mass ($\star\star$)
- Heterogenous Disk Radius ($\star\star\star$)
- Periodic Boundary Conditions ($\star\star\star$)
- LaTeX($\star\star\star$)
- Fancy Data Structures ($\star\star\star\star$)
- Particle Source/Sink ($\star\star\star\star$)

Further details and elaboration is found in the following sections.

Part II
Formulation of Numerical Methods

The project involves 8 helper functions and one main script for a total of 9 MatLab scripts. I will go through each one in logical order. An overview of the scripts:

Scripts	**Function**
Main	Initializes parameters and iterates through time
MakeParticle	Creates structure for each particle
moveObjects	Moves object a certain displacement
moveObjectsGravity	Moves object a certain displacement with Gravity
moveObjectsBrownian	Moves object a certain displacement with Brownian dynamics
Collision_Detection	Implements Uniform Grid Algorithm and detects collisions
Collision_Times	Calculates negative time increment at which particles first touch
Reverse_Velocity	Calculates velocities of particle after collisions
CreateAABBArray	Creates an AABB array for particles

0.1 Initializing Particles

`MakeParticle.m`

Before I began tackling the steps outlined in the problem statment, I made a design decision to use a Structure Array to store the particles. I therefore needed a function to create a structure given information about a particle. I therefore wrote **MakeParticle**. This function is very simple. Due to design decisions that I will elaborate on later, I decided that each particle needs to have the following information associated with it:

Particle P	x position	y position	x Velocity	y Velocity	Radius	Mass	ID

Accordingly, the code was:

```
function P = MakeParticle(x,y,xvel,yvel,radius,mass,ID)
P = struct('x',x,'y',y,'vx',xvel,'vy',yvel,'r',radius,'m',mass,'ID',ID);
```

In the main script, I use this function to create a structure array to hold all of the particles. I name this structure array throughout the code, *particleArray*.

0.2 Creating AABB Arrays

`CreateAABBArray.m`

Again, before I began working on the program logic I needed another helper function. The Uniform Grid Algorithm I implement later needs an Axis-Aligned Bounding Box (AABB) array. I had previously wrote a function to create AABB's for particles in HW7 and I simply need to update it to instead take a structure array of particles as input.

Figure 1 displays the definition of an AABB.

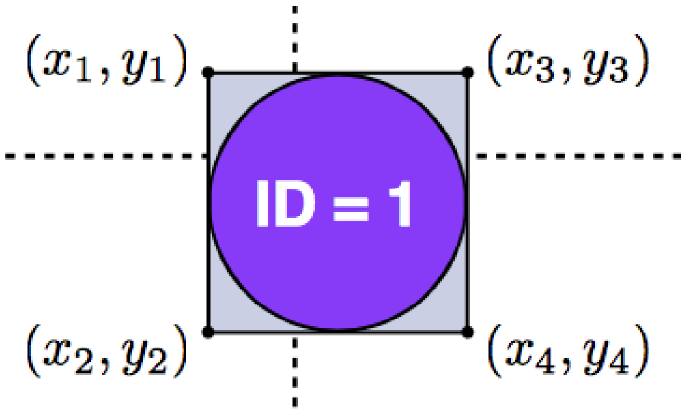
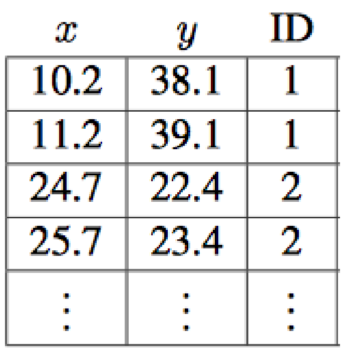


Figure 1: Definition of AABB

I will briefly summarize what AABB array format is. For each particle, there are four rows in the array which correspond to each vertex of that particles AABB. The columns of the array is seen in Figure 2.

The AABB array stores the vertices in the order:

1. Upper Left (x_1, y_1)
2. Bottom Left (x_2, y_2)
3. Upper Right (x_3, y_3)
4. Bottom Right (x_4, y_4)



x	y	ID
10.2	38.1	1
11.2	39.1	1
24.7	22.4	2
25.7	23.4	2
⋮	⋮	⋮

Figure 2: Format of the AABB array

CreateAABBArray is responsible for creating the AABB array. It takes as input the structure array of particles. This array contains the x and y positions of the center and the radius of each particle. This is the only information required. The code is quite straightforward. The AABB array is initialized to have $(4$ x $numberofparticles)$ rows and 3 columns.

The script iterates through the structure array. For each particle it calculates the positions of the 4 vertices. This is quite straightforward. Each vertex is calculated by simply by adding or subtracting the radius to the x and/or y position. In addition, the AABB array stores the particle ID in its third column. Once the loop completes, the AABB array is returned.

0.3 Collision Detection on a Grid

`Collision_Detection.m`

I now begin to tackle the actual project logic. The code is written following the exact procedure and steps outlined in the problem statement. Accordingly, the first part of the program involves writing a Collision Detection function. The function is entitled **Collision_Detection**. The function takes a structure array containing all of the particles as input. The format of this structure was specified earlier. This array is entitled, *particleArray*.

In addition, the function takes as input the systemWidth and systemHeight. The dimensions of the rectangular box confining the disks is given by: $systemWidth$ x $systemHeight$. All other necessary information is contained with *particleArray*.

The function returns a 2 column matrix of structures where each structure corresponds to a particle. A single row corresponds to overlapping particles. This structure array is entitled *collisions*.

The function header is:

```
function [collisions] = Collision_Detection(particleArray,systemWidth,systemHeight)
```

To begin with, the parameters of the simulator are defined. The Uniform Grid algorithm involves breaking up the system into cells. The number of cells is arbitrary, but a larger number of cells is preferred. I have chosen to use 25 cells. Depending on the number of cells, there will be a certain number of rows and columns. Assuming there are n_{cells} cells, there will be $\sqrt{n_{cells}}$ number of rows and columns. The dimension of a single cell is defined by dx and dy, where

$$dx = \frac{systemWidth}{n_{columns}} \tag{1}$$

$$dy = \frac{systemHeight}{n_{rows}} \tag{2}$$

Ideally, $dx = dy$. In addition, the number of particles in the simulator is given by the length of *particleArray*. Finally, **CreateAABBArray** is used to create the AABB array. The details of this were previously provided.

The Uniform Grid Algorithm has several steps as visualized in Figure 3.

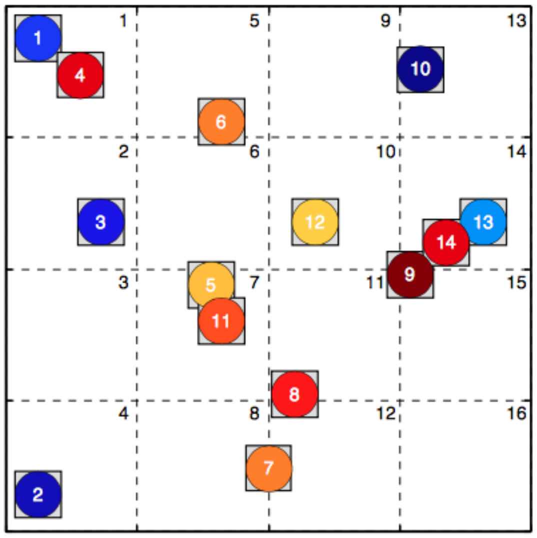


Figure 3: Implementation of the Uniform Grid Algorithm

1. Partition system into a grid of equally sized cells. The number of cells is arbitrary.
2. Use AABB's to determine each particles location in the grid (i.e. the cell number that contains the AABB). A particular particles AABB can overlap a maximum of 4 cells as long as the cell size is larger than the object size. Therefore, each vertex of a particles AABB must be considered seperately and a cell number is obtained for each. A particles's AABB is said to occupy a cell if that cell contains any of its vertices.
3. Particles are said to *broad phase* collide if their AABB's occupy the same cell.
4. Perform exact collision tests only on particles whose AABB's occupy the same cell.

To facilitate this process, the cells of the grid are numbered in the manner of Figure 4. Obviously, numbering adjusts according to the size of the grid.

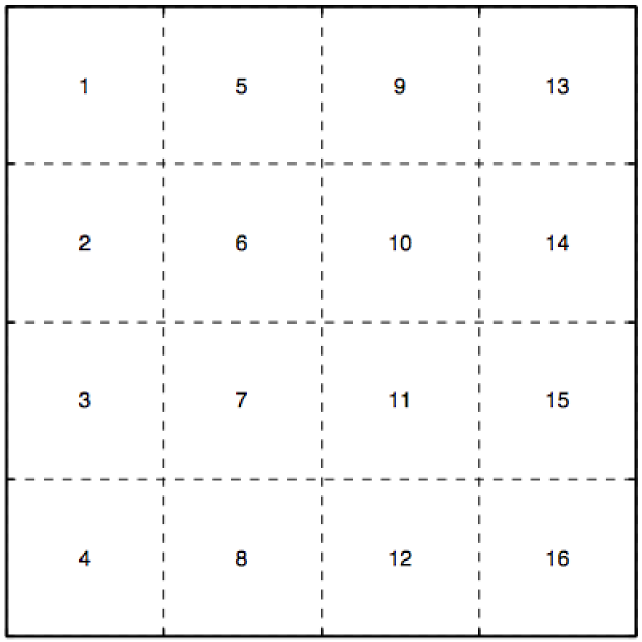
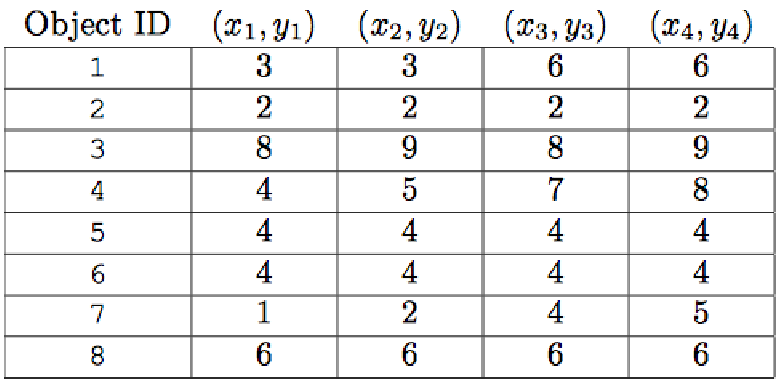


Figure 4: Cell Numbering

I will now delve into the details of the Uniform Grid Algorithm implementation. Step 1 has already been accomplished by defining the number of cells, etc. Step 2 is accomplished by defining a data structure to store the cell number corresponding to each vertex of a particular particle.



Object ID	(x_1, y_1)	(x_2, y_2)	(x_3, y_3)	(x_4, y_4)
1	3	3	6	6
2	2	2	2	2
3	8	9	8	9
4	4	5	7	8
5	4	4	4	4
6	4	4	4	4
7	1	2	4	5
8	6	6	6	6

Figure 5: Format of $cell\_id$ data structure

Figure 5 displays the format of the data structure to be used as well as a representative set of values. Therefore, a $2-D$ array can be used with 5 columns and $n_{particles}$ rows. The array is entitled *cell_id*.

In order to populate *cell_id*, we must iterate through the AABB array. The AABB array contains the particle ID and the vertices of its 4 points. We must therefore derive a formula to relate the Cartesian coordinates of a point to the number of the cell it is located in. The column and row number can be easily obtained. For a given vertex with coordinates, (x, y):

$$column = ceiling(\frac{x}{dx}) \tag{3}$$

$$row = n_{rows} - floor(\frac{y}{dy}) \tag{4}$$

In these formulas, ceiling and floor round a number up and down respectively. Recall, dx and dy are the dimensions of a single cell. At this point, we have a matrix subscript. The formula to convert this into a matrix index is:

$$index = (c - 1) * NR + r \tag{5}$$

c and r are the column and row number respectively. NR is the total number of rows.

We begin iterating through the AABB array in step sizes of 4. This is because each particle has 4 rows corresponding to its vertices. We begin by defining the first column of *cell_id* to be the particle ID obtained from the AABB array. We now consider each vertex in the order specified by the AABB array. Each vertex is checked to ensure it is within the bounds of the rectangular box as defined by systemWidth and systemHeight. If the point is valid, Equations (3), (4) and (5) are used to obtain the cell number that the vertex is located in. The code for one of the vertices is shown:

```
%BOTTOM LEFT
x_cur = AABB_array2(ii+1,1);
y_cur = AABB_array2(ii+1,2);

column_num = ceil(x_cur / dx);
row_num = n_rows - floor(y_cur / dy);

cell_num = (column_num - 1) * n_rows + row_num;
cell_id(index,3) = cell_num;
```

Once this has been done for each vertex of a particle, we move on to the next particle. Eventually, *cell_id* has been completely populated and we know which cells contains which particles.

The goal is now to determine object ID pairs to be exact collision tested. Recall that we test particles only if their AABB's occupy the same cell. At this point, *cell_id* has the cell number information for each particle. We must now check for broad phase collisions by identifying which particles are in the same cell. A cell array is used to store the broad phase collisions occuring in each cell. As such, the cell array has n_{cells} number of rows. Each row contains a single matrix entry where the matrix holds the IDs of pairs of particles overlapping in the corresponding grid cell. The cell array is entitled *broad_phase_collisions*

The procedure involves iterating through *cell_id*. We go through each row to check which cells contain a specific particle. If a particular cell holds a particle, we add the particle ID to the *broad_phase_collisions* cell array in the entry corresponding to that grid cell. In this manner, we populate the cell array such that at the end of the loop, each of the cells contains the ID's of the particles located in that corresponding grid cell. We now know that we must perform exact collisions only on particle pairs in the same cell of *broad_phase_collisions*.

Finally, exact collision testing is performed only on particle pairs specified in each cell of *broad_phase_collisions*. This method is quite familiar. Given the center positions of two particles $(x_1, y_1)\&(x_2, y_2)$ and their radii $r1$ and $r2$, the exact collision test checks if:

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} < r1 + r2 \tag{6}$$

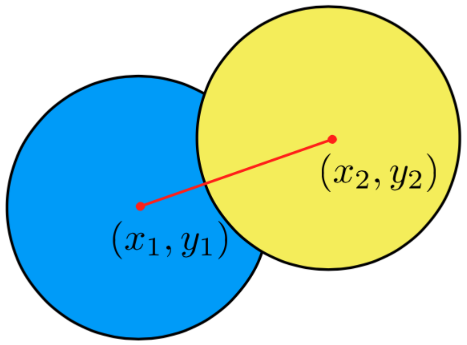


Figure 6: Exact Collision Test

We begin by defining the structure array entitled *collisions* to hold the pairs of particles that are exactly colliding. It is important to note that *collisions* holds much more than simply the ID's of the particle. The reason for this will be clear when I discuss the rest of the project solution.

We then iterate through *broad_phase_collisions*. For each pair of particles, we obtain their center positions using the positions vector and then apply Equation (6). If Equation (6) is satisfied, the particles are exactly colliding and they are added to the *collisions* structure array.

The function returns the *collisions* structure array in the end.

0.4 Indentifying Collision Times

`Collision_Times.m`

This function takes as input, a structure array of particles that have collided and are overlapping. The purpose of this code is to precisely describe a collision by backing up in time to a point at which the particles first initiate contact. This is because if two particles are overlapping, then the frame that this occurs in is not real. In actuality, the particles would have collided and changed direction a moment earlier. The function must return the maximum time increment that it should back up the system by. If the system is backed up to this point, all collisions are resolved and one collision will involve two particles such that the distance between their centers is exactly the radius of one plus the radius of the other.

For example, if the positions were input at a time t and contact occurred at $t - \delta t$, then the function calculates the value of δt. It does this for each of the particle pairs in the *collisions* array and returns the maximum δt. Accordingly, the function header is:

```
function [dtc, first, second] = Collision_Times(collisions)
```

Here, dtc is the maximum time to back up the system by.

Recall Equation (6). We can call the positions of the two particles when they have just collided: (x_{1c}, y_{1c}) and (x_{2c}, y_{2c}). If the particles have just collided, then equality is obtained in Equation (6) such that:

$$\sqrt{(x_{1c} - x_{2c})^2 + (y_{1c} - y_{2c})^2} = r_1 + r_2 \tag{7}$$

Now, the function contains the center positions and x and y velocities of both particles in their collided state. We can call the positions: (Px_1, Py_1) and (Px_2, Py_2) and the velocities: Vx_1, Vy_1, Vx_2 and Vy_2 where 1 corresponds to the first particle and 2 to the second.

The next step is to relate x_{1c},y_{1c},x_{2c} and y_{2c} to the current positions and velocities. It is quite clear that:

$$x_{1c} = Px_1 - tVx_1 \tag{8}$$

$$y_{1c} = Py_1 - tVy_1 \tag{9}$$

$$x_{2c} = Px_2 - tVx_2 \tag{10}$$

$$y_{2c} = Py_2 - tVy_2 \tag{11}$$

Inserting these equations into Equation (25) obtains:

$$(Px_1 - tVx_1 - Px_2 + tVx_2)^2 + (Py_1 - tVy_1 - Py_2 + tVy_2)^2 = (r_1 + r_2)^2 \tag{12}$$

Equation (12) can be rearranged and simplified in t analytically. It is important to note that P is the position of each object, v is the object velocity and crucially, t is the time we need to back the system up by.

I decided to solve Equation (12) using the MatLab function *solve*. The code to do this is as follows:

```
syms t
solve((Px_1 - tVx_1- Px_2 + tVx_2)^2 + (Py_1 - tVy_1 - Py_2 + tVy_2)^2 -
(r_1 + r_2)^2,t)
```

The output of the above code is the two formulas seen for $t1$ and $t2$. Due to the length of the equations, I cannot display them here. However, $t1$ and $t2$ have been verified to be solutions to Equation (12). There are two solutions because we solved a quadratic equation. Practically, we would expect there to be two solutions. Consider two overlapping particles as in Figure 6. It is clear that we can move time backwards or forwards to obtain contact between the boundaries of the particles. The key is to identify which $t1$ and $t2$ correspond to respectively.

It is observed that the following is always true: $t1 > 0$ and $t2 < 0$. As such, based on the way the solution was obtained, the positive results corresponds to the one we want. Therefore, $t2$ can be ignored.

These steps are repeated for each pair of particles. Eventually a maximum $t1$ is found and is assigned to dtc.

0.5 Main Script: Time Stepping and Resolving Collisions

```
Main.m
```

In this Main script, I bring together the collision detection methods and use them to simulate particle dynamics. In a manner similar to the Predator-Prey problem, I discretize time into finite increments.

$$t_1 = 0, t_2 = \Delta t, \ldots, t_n = t_{n-1} + \Delta t. \tag{13}$$

During the script, I iterate through the time steps. In each time step or ”frame”, I update particle positions using the laws of physics. This is done using the function, **moveObjects**. The basic simulator assumes constant velocity between collisions.

Collisions are handled using the ”predictor-corrector” algorithm. For each frame, I predict the motion of the particles using displacement formulas. Collisions are detected using **Collision_Detection**. If any of the particles are predicted to overlap, this collision must be corrected. This is done by going back in time to the earliest point between t_{n-1} and t_n where the two particles came into contact. This is done using the **Collsion_Times** function earlier described.

At the point of collision, the particles must change velocities according to the model of collision physics. The basic simulator assumes a perfectly elastic collision. The function, **Reverse_Velocity** is written to calculate the velocities after the collision.

Finally, the simulator must be visualized by plotting the particles and creating a movie file.

I have provided an overview of the main script. I will go into further detail, but first I will discuss the two helper functions mentioned above: **moveObjects** and **Reverse_Velocity**.

Moving Objects

`moveObjects.m`

This function is called by the Main script and is responsible for updating particle positions between frames. The basic simulator assumes that particles travel at constant velocity between collisions ($\vec{v}$). Therefore, given a position $\vec{x}_{n-1}$ at a time t_{n-1}, if there are no collisions, the position at a later time t_n is

$$\vec{x}^p_n = \vec{x}_{n-1} + \Delta t \vec{v}_{n-1}. \tag{14}$$

$$\vec{y}^p_n = \vec{y}_{n-1} + \Delta t \vec{v}_{n-1}. \tag{15}$$

This is a *prediction* and must be corrected if it causes the particle to overlap with another particle or the boundary. However, these cases will be handled by other parts of the code.

For now, the **MoveObjects** simply implements Equation (14) and Equation (15). It takes as input a structure array of particles and the time step, Δt. It returns the structure array of particles with updated x and y positions. Accordingly, the function header is:

```
function particleArray = moveObjects(particleArray,dt)
```

The code simply iterates through the structure array of particles and applies Equations (14) and (15).

```
for k = 1:length(particleArray)
    particleArray(k).x = particleArray(k).x + dt*particleArray(k).vx;
    particleArray(k).y = particleArray(k).y + dt*particleArray(k).vy;
end
```

Resolving Collisions

`Reverse_Velocity.m`

This function is responsible for calculating the change in velocity of two particles after they have collided. It will be called by the Main script whenever there is a collision. It takes as input [1] structures corresponding to two colliding particles: $first$ and $second$. It will return

[1] Disregard the *inelastic* and *e* for now as they will be explained later.

the new x and y velocities of both particles.

When two particles collided, their collision plane is not necessarily parallel to the x-axis or y-axis. However, we know only the x and y velocities of the particles. As such, it is necessary to convert between reference frames. The reference frame of the collision is the tangential and normal axes. Here, the tangential axis is parallel to the contact plane while the normal axis is perpindicular. There is an angle of rotation associated with the $t-n$ frame and the $x-y$ frame. The steps are:

1. Function takes as input the velocities in the $x-y$ frame.
2. Express velocities in components in the $t-n$ frame.
3. Update velocities using model of collision physics.
4. Rotate velocities back into $x-y$ frame for output.

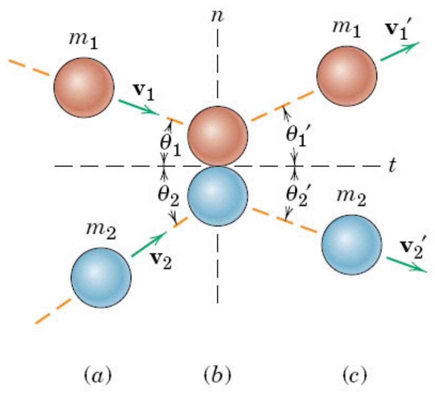


Figure 7: Breaking down velocity components into normal (n) and tangential (t) components for resolving collisions.

The basic simulator assumes perfectly elastic collisions and as such, the velocity tangential to the contact plane is unchanged while the normal components swap:

$$(v_1')_t = (v_1)_t, \qquad (v_2')_t = (v_2)_t, \tag{16a}$$
$$(v_1')_n = (v_2)_n, \qquad (v_2')_n = (v_1)_n. \tag{16b}$$

The rotation angle, θ can be found using the geometry of the problem. It is directly related to the positions of the center of the particles. In fact,

$$\theta = \tan^{-1}(\frac{Px_2 - Px_1}{Py_1 - Py_2}) \tag{17}$$

Once this is obtained, the straightforward and well-known formulas to switch reference frames is applied:

$$v_t = v_x cos(\theta) + v_y sin(\theta) \tag{18a}$$
$$v_n = -v_x sin(\theta) + v_y cos(\theta) \tag{18b}$$

I then applied Equations (16) to determine velocities after the collision. Finally, I convert back into the $x - y$ reference frame.

$$v_x = v_t cos(\theta) - v_n sin(\theta) \tag{19a}$$
$$v_y = v_t sin(\theta) + v_n cos(\theta) \tag{19b}$$

These values are then returned by the function.

Main

At this point, all of the helper functions have been discussed and the Main script can be tackled. I begin by intializing the parameters of the problem. I load position data from a MAT file and manually set the number of particles to consider. As discussed previously, there is a x velocity, y velocity, radius and mass associated with each particle. These are randomly determined with an adjustable maximum value. It is specially noted that the velocity is defined such that positive and negative values are possible. Obviously this is not the case for radius and mass. An array of velocities, radii and masses are created. Also note that it is possible to create fixed density particles. This means that the mass and radius are related to each other. Finally, the time step, Δt and total time are set.

At this point, the structure array is created and entitled *particleArray*. **MakeParticle** is used here and was previously discussed.

The figure on which the particles will be plotted is then configured. It is set up so that the axes are fixed to be a rectangular box defined by $systemWidthxsystemHeight$. In addition, *axis equal* ensures that circular objects appear circular.

Next are the bonus features parameters. These are used to turn on/off certain features. This will be expanded upon further later.

At this point the time stepping begins. A while loop encapsulates all the following code and basically runs until time reaches the final time. The very first thing we do is move the objects. This is done by calling **moveObjects**.

```
particleArray = moveObjects(particleArray,dt);
```

We then increment the time: $t = t + \Delta t$.

At this point I check for collisions using **Collision_Detection**. As described previously the return value is a matrix of particle structures and is stored in the *collisions* variable.

```
collisions = Collision_Detection(particleArray,systemWidth,systemHeight);
```

The goal now is to determine how far back in time we need to go to resolve the collisions. Thus, if *collisions* is not empty, a call is made to **Collision_Times**. The variable dtc is used to hold the time increment we must go backwards by.

```
dtc = Collision_Times(collisions);
```

As such, it is necessary to correct the positions of the objects by moving backwards in time. This is done by calling **moveObjects** again and passing it $-dtc$ as the time increment.

```
particleArray = moveObjects(particleArray,-dtc);
```

I then must change the velocities of colliding particles. This is done by iterating through the *collisions* matrix. For each pair of colliding particles, **Reverse_Velocity** is used to determine the correct velocities of the particles after the collision.

```
[dvx1,dvy1,dvx2,dvy2] = Reverse_Velocity(collisions(j,1),collisions(j,2),inelastic,e);
```

Finally, time is incremented by dtc: $t = t - dtc$.

Boundary collisions are now accounted for. I begin by iterating through all of the objects to check if any have collided with the boundary. I do this by defining a maximum and minimum x and y position for each particle as in Figure 8. The definition of these is exactly simliar to the definition of an AABB for the particle. Using these maximum and minimum points, I can check where intersection has occurred. There are four possibilities:

1. Collision with $x_{minimum} = 0$
2. Collision with $x_{maximum} = systemWidth$
3. Collision with $y_{minimum} = 0$
4. Collision with $y_{maximum} = systemHeight$

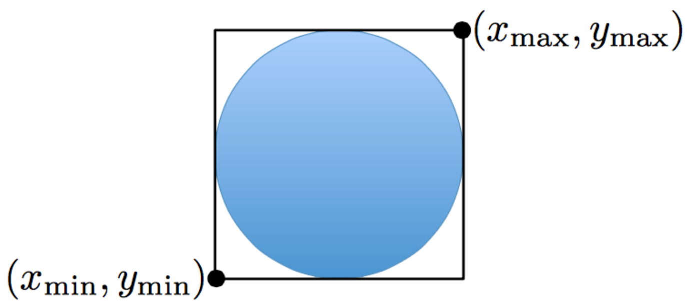

$(x_{\text{max}}, y_{\text{max}})$
$(x_{\text{min}}, y_{\text{min}})$

Figure 8: Minimum and Maximum x and y positions

Boundary collisions are easily solved by simply reversing the direction of velocity normal to the boundary. This means a collisions with the x boundaries involves reversing the x velocities and similarly for y.

Finally, the particles are plotted. This is done by iterating through every object. To plot a particle, the *rectangle* function is used. To use this, a minimum x and y position are needed. This is found simply by subtracting the radius value from x and y. The syntax for the rectangle function is:

```
set(circlefigures(zz),'Position',[x_min,y_min,2*radius_cur,2*radius_cur],
'Curvature',[1 1], 'FaceColor',colors(color));
```

In addition, I will mention how I create videos from my animated plot. I use the videoWriter function. The usage of this is quite straightforward. I simple use getframe to get each frame of the plot and videoWriter creates a movie file from this.

The above continues to repeat as long as the while loop is satisfied.

0.6 Bonus Features

I will now provide details about the bonus features I have implemented.

A. **Gravity** ($\star$)

This bonus feature adds the effect of gravity to the system. Gravity is assumed to act in the downwards y-axis direction. Therefore, it only affects the displacements of the y coordinates in each time step. The eqautions governing the y components of position and velocity are:

$$y_n = y_{n-1} + \dot{y}_{n-1}\Delta t - \frac{g}{2}\Delta t^2, \tag{20}$$

$$\dot{y}_n = \dot{y}_{n-1} - g\Delta t. \tag{21}$$

In order to account for gravity, I write the new function: **moveObjectsGravity**. This function is almost exactly analogous to **moveObjects**. The only difference is that the function takes g as input and Equations (20) are used for the y position and velocity.

```
function particleArray = moveObjectsGravity(particleArray,dt,g)
for k = 1:length(particleArray)
        particleArray(k).x = particleArray(k).x + dt*particleArray(k).vx;
        particleArray(k).y = particleArray(k).y + dt*particleArray(k).vy - (g / 2) * dt
        particleArray(k).vy = particleArray(k).vy - g*dt;
end
```

In the main function I simply must use **moveObjectsGravity** in place of **moveObjects**. I use a flag named *gravity* to specify whether gravity is on or off. If gravity is set to 1, the effects of gravity will be seen. In addition, g is the acceleration due to gravity. Increasing this value will increase gravitational force. For this simulation, I recommend low values of g (around $g = 0.1\frac{m}{s^2}$).

After a collision, I can call **moveObjectsGravity** specifying dtc as the time increment to correct overlapping particles. Please note that no other changes are implemented. The rest of the code is exactly the same as before.

B. **Inelastic Collisions** ($\star$)

Inelastic collisions mean that energy is lost through collisions. The basic simulator assumed elastic collisions and used Equation (16) to determine velocities after collisions. The situation is more complicated with inelastic collisions.

Inelastic collisions are characterized by the coefficient of restitution, e. The normal velocities after a collision are related to the normal velocities before the collision according to:

$$m_1 v_1 + m_2 v_2 = m_1 v_1' + m_2 v_2' \tag{22a}$$
$$v_2' - v_1' = e(v_1 - v_2). \tag{22b}$$

Here v_1 and v_2 are the components of velocity of the two particles *normal* to the contact plane before the collision, and from these equations I can solve for v_1' and v_2' after the collision. Solving these simultaneous equations I obtain:

$$v_1' = v_1 + \frac{m_2}{m_1} v_2 - \frac{m_2}{m_1} v_2' \tag{23a}$$
$$v_2' = \frac{(v_1(1+e) + v_2(\frac{m_2}{m_1} - e))}{1 + \frac{m_2}{m_1}} \tag{23b}$$

Implementing this in MatLab is simple. Instead of swapping components, I apply Equations (23).

```
if inelastic == 1 %Inelastic
    %Inelastic Collision Equations
    v2n_after = (v1n*(1+e)+v2n*((m2/m1) - e))/(1+m2/m1);
    v1n_after = v1n + (m2/m1)*v2n - (m2/m1)*v2n_after;
```

In addition, **Reverse_Velocity** must take two additional inputs, *inelastic* and *e*. *inelastic* is a flag which determines whether inelastic collisions are on or not. If *inelastic* is 1, we assume that collisions are inelastic. Additionally, e is the coefficient of restitution and is used in Equations (23).

When running the simulator with inelastic colllisions on and e set very low, it is observed that particles ”stick” together and eventually all particles come to complete stops. This makes sense because all of the energy has been lost.

C. **Brownian Dynamics** ($\star\star$)

Brownian dynamics is now implemented. A flag named $brownian$ is used to indicate whether to use Brownian dynamics is on or off. If $brownian$ is set equal to 1, it is on. The key difference with this is that the displacement vector changes:

$$\vec{u}_n = \sqrt{2D\Delta t}\,\vec{r} \tag{24}$$

Here, D is the diffusion coefficient and $\vec{r} = (r_x, r_y)$ is a vector of random numbers pulled from a Gaussian distribution with zero mean and unit variance. In addition, the velocity can be taken as $\vec{v}_N = \vec{u}_n/\Delta t$. Just like for gravity, I write a new function to move the particles called **moveObjectsBrownian**. Again, it is very similar to **moveObjects**, only it uses the new displacement vector.

```
        particleArray(k).x = particleArray(k).x + dt*particleArray(k).vx +
            sqrt(2*D*dt) * r_brown(k,1);
        particleArray(k).y = particleArray(k).y + dt*particleArray(k).vy +
            sqrt(2*D*dt) * r_brown(k,2);;

        particleArray(k).vx = sqrt(2*D*dt) * r_brown(k,1) / dt +
            particleArray(k).vx;
        particleArray(k).vy = sqrt(2*D*dt) * r_brown(k,2) / dt +
            particleArray(k).vy;
```

Once again, **moveObjectsBrownian** replaces **moveObjects** except when correcting positions after a collision. After a collision, the same code is used as before.

No other changes were made to the code. Everything remains as it was before.

D. **Heterogenous Disk Mass** ($\star\star$)

This bonus feature means that we must simulate the dynamics of particles with differing masses. This is quite straightforward. As I mentioned previously, particles are stored in structures. Each particle has a mass associate with it. I have chosen to give the particles random masses between 1 and 100.

```
%%%%% Randomized mass %%%%%
max_mass = 100;
min_mass = 1;
mass = min_mass + (rand(numObj,1)) * max_mass;
```

Now, the only other difference is in **Reverse_Velocity**. Recall that Equation (23) was used with the inelastic collisions bonus features. It can be seen that the mass of the particles comes into play through m_1 and m_2. The only difference now is that $m1$ may not equal m_2 and the equation is slightly more complicated to solve.

There are no other differences in the code. Although, it is important to note that I wrote this bonus feature in from the very beginning and so I made design decisions keeping heterogenous masses in mind. In the video attached, if inelastic collisions are turned on, it can be seen that when a heavy particle collides with a small particle (represented by their respective sizes), the small particle is blown away.

E. **Heterogenous Disk Radius** ($\star\star\star$)

This bonus feature simulates collision dynamics with particles of different radii. Recall that the structure for a particle contains a radius value. I have decided to make mass and radius proportional. As such, I define a density value (ρ) and I simply set radius to be: $r = \sqrt{\frac{mass}{\pi\rho}}$. Alternatively, I could have just defined radius to be a random vector independent of mass. I show both methods in the code below.

```
%%%%% Randomized radius %%%%%
max_radius = 5;
min_radius = 1;
% radius = min_radius + (rand(numObj,1)) * max_radius;
radius = sqrt(mass / (rho*pi));
```

Now having different radii for particles comes into play many times. Specifically in: **Collision_Detection** and **Collision_Times**. Here the key difference is in the distance formula. With particles of different radii, r_1 and r_2, the new distance formula is:

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} = r_1 + r_2 \tag{25}$$

There are no other differences in the code. Although, it is important to note that I wrote this bonus feature in from the very beginning and so I made design decisions keeping heterogenous radii in mind.

F. **Periodic Boundary Conditions** ($\star\star\star$)

This bonus feature changes how the boundary behaves. Now, the boundary acts like its infinite and particles don't bounce off the walls. The idea is that we associate a point just past the right boundary of the box $x > systemWidth$ with a point just inside the left boundary, $x' = x - systemWidth$. The same thing is done for y using systemHeight instead. To turn periodic boundary conditions on, set the flag, *periodic* equal to 1.

The code to implement this is:

```
if min_x <= 0 && particleArray(index).vx < 0
     particleArray(index).x = particleArray(index).x + systemWidth;
elseif max_x >= systemWidth && particleArray(index).vx > 0
     particleArray(index).x = particleArray(index).x - systemWidth;
end

if min_y <= 0 && particleArray(index).vy < 0
      particleArray(index).y = particleArray(index).vy + systemHeight;
elseif max_y >= systemHeight && particleArray(index).vy > 0
      particleArray(index).y = particleArray(index).y - systemHeight;
end
```

G. LaTeX($\star\star\star$)

H. **Fancy Data Structures** ($\star\star\star\star$)

I wrote the project code from the beginning using this bonus feature. As I have mentioned previously, I used structure arrays to hold the particles. In addition, my **Collision_Detection** code returns a structure matrix of particles where each row holds particles that are colliding.

My use of structures to define particles had the following benefits:

- Made it very easy to implement and keep track of heterogenous mass and radius for the particles
- Reduced number of variables that I would otherwise have had to keep track of.
- Reduced the number of parameters I would have to pass to functions like **Reverse_Velocity** and others
- When I implemented Sink Boundaries, it was very easy to delete a specific particle from the structure array

I. Particle Source/Sink ($\star\star\star\star$)

This bonus feature changes how the boundaries work. Now, if a particle collides with the boundary, it must dissapear. This involves deleting the particle from the structure array and is quite straightforward.

Now, when I check if a particle has collided with the boundaries, instead of reversing its velocity or shifting it to the other side, I simply note its ID. Then I simply remove it from my structure array. This is done by defining two subarrays. One subarray goes from the beginning of the original array up to right before the particle to delete. The other subarray goes from right after this particle to the end of the array. I then concatenate these two arrays. This array is simply the original array with that particle deleted.

```
%%%%% FOR SINK BOUNDARY CONDITIONS %%%%%
if flag_delete == 1
    beg = 1:index_delete-1;
    back = index_delete+1:numObj;
    indexes_to_keep = horzcat(beg,back);

    for bb = index_delete:numObj
        particleArray(bb).ID = particleArray(bb).ID - 1;
    end
    particleArray = particleArray(indexes_to_keep);
    numObj = numObj - 1;
end
```

Part III
Calculations and Result

The problem statement specifies that two scenarios be displayed. One with two particles and another with N particles.

Scenario 1: Here I simply use two particles and initialize them with random, non equal velocities. They will collide with each other and with the boundaries. I will plot the velocity vs. time and also verify that correct post collision velocities using Equation (23). The bonus features that I have turned on for this demonstration are:

- Gravity, $g = 0.05$
- Inelastic, $e = 0.85$
- Heterogeneous masses
- Heterogeneous radii

I made a plot of the x and y velocities of both particles with time:

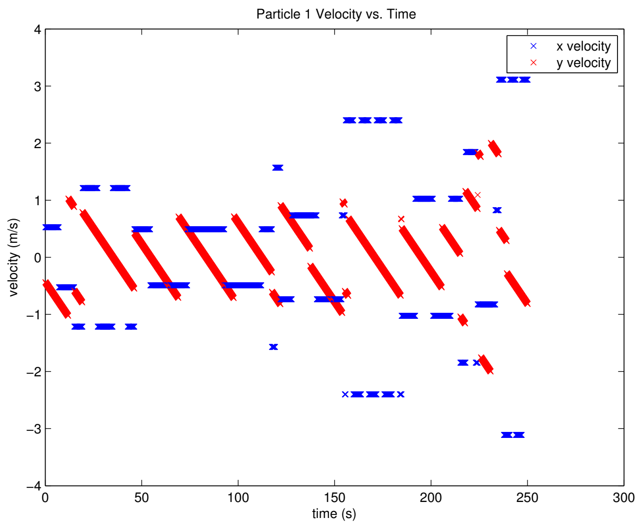


Figure 9: Implementation of the Uniform Grid Algorithm

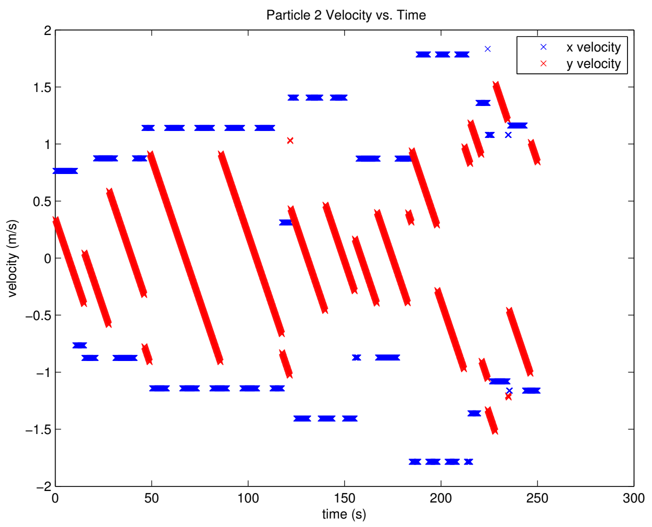


Figure 10: Implementation of the v Grid Algorithm

The results are quite interesting. The discontinuities are obviously where the particle collided with either the other particle or with the boundaries. It can also be seen that the smaller particle, 2, has larger velocities. Additionally, it is seen that the y velocity is always decreasing which is due to gravity. I have also created and attached an animation of this called: **TwoParticles.mp4**.

I then ran a "production run" with a large number of particles. I selected $N > 10$ particles and initialized them with random position, velocities, mass and radii. I run this code for enough time that many collisions can be observed. I have attached several video files corresponding to different run-throughs.

1. A_Collisions.mp4: This is the basic collision simulator. No gravity, no inelastic and normal boundaries.

2. B_Collisions.mp4: Gravity ($g = 0.5$) and Inelastic ($e = 0.5$) are on. Normal Boundaries.

3. C_Collisions.mp4: Gravity ($g = 0.1$) and Inelastic ($e = 0.65$) are on. Normal Boundaries.

4. Brownian.mp4: Brownian Dynamics and Sink Boundaries On.

As can be seen from the videos everything is working quite well. Gravity causes the particles to end up on the ground as time goes to ∞. Inelastic collisions cause the particles to come to complete stops as time goes to ∞. Also, smaller particles are seen to have no effect on larger particles when they collide.

Part IV
Discussion

The project was overall enjoyable. The results were fascinating to watch. I am quite pleased with my code because I managed to get the basic one working and I also implemented most of the bonus features. I wanted to tackle the last two but I ran out of time.