Pranav Raghuram

UID: 004796142

CEE/MAE M20

February 21st 2019

Final Project

1.1 Introduction

The goal of this problem is to a program that seeds a 2D control volume with a number of initial spheres. Each sphere will then randomly move about and collide with one another, accounting for momentum and kinetic energy conservation, while the control volume also evolves over time. In addition, whenever any spheres collide, there is a probability of them spontaneously absorb into one larger sphere. The particles may also collide with the boundaries. The program will also output a video of this entire process over the specified parameters at the end. The simulation is done in a main script with several functions called. Some of the functions involve calling other functions as well. These functions are the specified ones in the problem statement as well as some others I made to perform the tasks required.

1.2 Model and Methods

We begin by clearing terminal and command window in the main script. We then use `rng('shuffle')` to ensure that the results when calling the rand function will be unpredictable. We then follow this up by initializing the variables provided in the problem statement such as number of spheres, radius and velocity. We then procced to initialize the spheres using the seedInitial function specified in the problem statement. All functions used in the main script will be discussed below after I finish covering the main script. We then initialize the video process by using the following code fragment:

```
v = VideoWriter('CollisionVideo','MPEG-4');
v.FrameRate = 60;
open(v);
```

We then move into the timestepping process. This is done by using an enclosing while loop which runs as long as the current time is less than the final time. Moving into the while-loop, we procced to use the fieldEvolution function to calculate the iterative timestep motion of the spheres. We then increment the current time by the timestep dt. Next, we check for collisions. This is done using the detectCollisionfunction which returns an array called collisions, containing the information of all the collisions. If this array is not empty, we revert the time to the first collision. This time, dtprime, is found using the timeCheck function. We then use the fieldEvolution function again, this time with dtprime as the parameter instead of dt. We also

have to revert the velocities to the time of the first collisons and this is done using the revertVelocity function within a for-loop running through the rows of the spheres array. The revertVelocity function returns the new velocities of the particle in question and these new velocities are then updated in the sphere array. Finally, the current time is reverted to the time of the first collision by using:

```
t = t - dtprime;
```

We then move onto plot the particles and create the video. This is done by the following code fragment:

```
  s = size(spheres);
  ss = s(1);
  theta = linspace(0,2*pi,50);
  for i = 1:ss
      x = spheres(i,2) + cos(theta);
      y = spheres(i,3) + sin(theta);
      plot(x,y);
      hold on;
  end
  frame = getframe;
  writeVideo(v,frame);
```

Finally, after the while-loop has run its course, we end the video by using close(v); .

Now I will describe the functions.

First seedInitial. This function is used to seed initial conditions of control volume. The input parameters are : ns is number of spheres, vs is velocity of spheres(1 value for uniform velocity or (ns x 1) array), rs is sphere radius (same as velocity- either 1 value for uniform or (ns x 1) array) and BC is a (4x1) boundary conditions array. The format is:

```
function [spheres] = seedInitial(ns, vs, rs, BC)
```

Firsr, we error check the function inputs for cases like: no. of spheres need to be positive integers, BC needs to be a 4x1 array, etc. Next, we create the spheres array and initialize to zeros so this helps prevent dynamic resizing. We then also modify the rs and vs inputs to vectors if they were uniform single inputs. This is done by:

```
spheres = zeros(ns, 5);

% if radius is a uniform constant, form a vector with ns
elements filled with that value
 if size(rs) == 1
   temp_rs = rs;
   rs = ones(ns,1)*temp_rs;
 end

 if size(vs) == 1
```

```
    temp_vs = vs;
    vs = ones(ns,1)*temp_vs;
 end
```

We then create random starting positions for the spheres, followed by error checking that the sphere volume doesn't exceed the allowed space in the control volume.

We then proceed to use a bool variable seedCorrect and a for-loop to error check the initialization positions of the sphere and reinitialize it if neede. An example for bottom right is:

```
seedCorrect = false; % boolean variable
 %error check for sphere position initialization & re-
initialize if necessary
 for k = 1:ns
   while (seedCorrect == false)
    if (x_pos(k) + rs(k) > BC(2))     % x position exceeds
right boundary
      seedCorrect = false;
      x_pos(k) = (rand * boundaryWidth) + BC(1);
```

we set the bool variable to true if it passes the error checks. We rhen ensure that the generated spehere cannot intersect an existing sphere.

We then proceed to calculate the x and y velocities of the spheres to fill up the 4th and 5th columns of the spheres array by randomly initializing an angle using the rand function.

```
theta = rand(ns, 1) * 2*pi;     % random angle calculations
for n = 1:ns
  x_vel(n) = vs(n)*cos(theta(n));     % x-component of
velocity
  y_vel(n) = vs(n)*sin(theta(n));     % y-component of
velocity
end
```

Finally we assign all the relevant values to the respecfive columns of the spheres arrays.

Next the fieldEvolution has the format:

```
function [spheres] =
fieldEvolution(spheres,dt,ns,density,boundaryWidth,boundary
Height,absRatio,f,f2,f3)
```

we first update the new x and y positions of the spheres

```
s = size(spheres);
  length = s(1);
  % Update new x and y positions of spheres
  for k = 1:length
```

```
    spheres(k,2) = spheres(k,2) + dt*spheres(k,4);
    spheres(k,3) = spheres(k,3) + dt*spheres(k,5);
  end
```

we then use function handles to call the detectCollision function to determine the collison data (i.e. posions and reference numbers of speheres). Once collisions have been located, we determine what type of collisions they were by again using the rand function to determine probability and seeing if it exceeds the absorption ratio. If it does, it will be an absorption using the absorption function.

This function has the format:

```
function [spheres] = absorption(spheres, A, B, density)
```

wand accounts for a new sphere C created by two initial spheres A and B colliding. We calculate the masses of spheres A and B and use it to calculate sphere C data which is then input into the equations provided in the problem statement to calcyulate its relevant data.

```
Avol = (4/3)*pi*((spheres(A,1))^3);
Bvol = (4/3)*pi*((spheres(B,1))^3);
mA = density*Avol;
mB = density*Bvol;


% calculation of sphere C data
 Cvol = Avol + Bvol;  % sphere C volume
 rC = ((3*Cvol)/(4*pi))^(1/3); % sphere C radius
 C(1,1) = rC;
```

The spheres array is then modified to include C while removing A and B.

```
% when spheres A and B collidie they form a single new
sphere C. Hence the spheres array
 % has to be modified to represent that by reducing the
number of rows and updating
 % the data
 s = size(spheres);
 ss = s(1);
 spheres(A:(ss-1),:) = spheres((A+1):ss,:); %shift array up
one row -overwrite A
 for k = 1:5
   spheres((B-1),k) = C(1,k);       % overwrite B with C
data
 end
 spheres(ss,:) = [];
```

If the probabailiy in fieldEvolution does not exceed the absorption ratio, it will be an elastic collision using the respective function which has a format:

```
function [spheres] = elasticCollision(spheres, A, B,
density)
```

We first account for wall collisons. Example:

```
% check if it is a wall collision
  if (A == 0 || B == 0)
    % wallCollision = true;
    if B == 0   % sphere A collides with wall
      angleA_bfr = atan2((spheres(A,5)),(spheres(A,4)));
% angle of collision
      angleA_aft = - angleA_bfr;
      uA = sqrt((spheres(A,4)^2)+(spheres(A,5)^2));     %
initial speed of A
      spheres(A,4) = uA*cos(angleA_aft);                %
new vel in x-dir
      spheres(A,5) = uA*sin(angleA_aft);
```

After calculating the sphere masses, we calculate their speeds and angle of contact:

```
% calculation of velocity magnitudes (speed)
    uA = sqrt((spheres(A,4)^2)+(spheres(A,5)^2));
    uB = sqrt((spheres(B,4)^2)+(spheres(B,5)^2));
% angle of contact
    alpha = atan2((spheres(A,3)-
spheres(B,3)),(spheres(A,4)-spheres(B,4)));
```

We then use the formulas provided in the slides to calculate the final velocities and update then into the spheres array. Back in the fieldEvolution function, we account for the wall collisions by:

```
s2 = size(spheres);
  length2 = s2(1);

  % Boundary collision
  for j = 1:length2
    x_max = spheres(j,2) + spheres(j,1);
    x_min = spheres(j,2) - spheres(j,1);
    y_max = spheres(j,3) + spheres(j,1);
    y_min = spheres(j,3) - spheres(j,1);

    if (x_min <= 0 && spheres(j,4) < 0)
      spheres(j,4) = - spheres(j,4);
    elseif (x_max >= boundaryWidth && spheres(j,4) > 0)
      spheres(j,4) = -spheres(j,4);
    end

    if (y_min <= 0 && spheres(j,5) < 0)
```

```
      spheres(j,5) = -spheres(j,5);
    elseif (y_max >= boundaryHeight && spheres(j,5) > 0)
      spheres(j,5) = -spheres(j,5);
    end
  end
```

The revertVelocity function has a format `function [vAx,vBx,vAy,vBy] = revertVelocity(spheres,A_ref,B_ref)`

and uses the formulas provided in the slides to rotate the plane and calculate the velocities before returning them to the original plane and outputting them.

1.3 Calculations and Results

1.4 Discussion

The functions could have been improved by using a struct method instead of using 2D arrays only.