Avery Edson
404828221
CEE/MAE M20
June 15, 2018

Final Project

1. Introduction

The purpose of this project is to simulate fluid behavior in a 2D model using smoothed particle hydrodynamics (SPH) on a spatially-hashed domain. Essentially a set of particles will mimic the behavior of fluid due to the physical effects and properties of its surrounding "neighbor" particles. To do this, a spatially-hashed domain is used so that the particle in question only takes into account the effects its neighboring particles within a close proximity produce. This is done for several time steps, which provides the result of a fluid-like motion. To achieve this, the properties (constants) of the fluid being simulated were adjusted to achieve the desired behavior.

2. Models and Methods

To begin a grid must be created as a set space for the particles to "live in". Similarly done in the spatial hashing problem of Homework 8, the grid size is set by assigning the maximum axes values, generating the number of rows and columns, and determining the bin dimensions.

```
%set grid size
xmax = 5;
ymax = 5;
w = 0.5;
%number of columns
Nx = floor(xmax/w);
%number of rows
Ny = floor(ymax/w);
%establish bin dimensions
dx = xmax/Nx;
dy = ymax/Ny;
```

Prior to writing the code which simulates the fluid-like behavior, a user input is required to determine whether the script runs a simple "dam break" behavior or runs the complicated setup. To do this, an input is requested of the user using an input function and an if statement is entered to deliver an error if the user does not input a correct input value.

```
method = input('Please enter a method 1 or 2: ');
if method~= 1 && method~=2
    error('method must equal 1 or 2')
end
```

Based on the user input, the script will run one of the two cases distinguished by the switch/case function. The first case is the "dam break" simulation and the second case is the complicated setup. Within each case are the different constants and particles that were used for each simulation. In other words, all the code that is different between the two simulations is within the switch/case function and all the code that the two simulations share is outside of it.

For the “dam break” simulation, the following fluid property constants were chosen, the particle structure was initialized to contain each property, and the particles themselves were initialized in a grid-like manner using a for loop. The time-step and the final time constants were also assigned a value.

```
%model constants;
R = 10 ;
C = 30 ;
h=0.111;
dt = 0.005;
tFinal = 5;
%stiffness coefficient
kappa = 4.6;
%viscosity coefficient
mu = 0.2;
rho_0 = 14;
mass = rho_0/(R*C)*3;

%initialize the particles structure
for k=1:R*C
    particles(k) =
struct('pos',[],'vel',[0,0],'force',[],'density',[],'neigh',[]);
end

%initialize particles
k=1;
x=0.2;
for i=1:R
    y=0.2;
    for j=1:C
        particles(k).pos = [x y];
        y=y+0.1;
        k=k+1;
    end
    x=x+0.1;
end

N=length(particles);
N_new=length(particles);
```

The length of the particles structure was assigned to these variables above as they are referred to in later parts of the script.

For the complicated setup of case 2, the same is done simply with different values. Also, more “particles” are created to act as the stationary “lines” or obstacles that the fluid particles must pass over. After this, the switch/case is ended.

Code is then entered for the purposes of creating a movie when the script is finalized. If I wanted to save a movie, I would set saveMovie equal to true.

```
%movie
```

```
t_steps = tFinal/dt;
saveMovie = false;

if saveMovie == true
    % STEP 1: Create the video object
    vidHandle = VideoWriter('fluidsim2', 'MPEG-4');

    % STEP 2: Set some movie-making options
    vidHandle.FrameRate = 30;
    vidHandle.Quality = 100;

    % STEP 3: Open the file for writing
    open(vidHandle);
end
```

Now the for-loop for the time-steps is started so that it repeatedly runs through the following code to simulate the fluid until the final time is reached. This code is the same for both the "dam break" case and for the complicated setup case.

The first thing that needs to be done within this time for-loop is the particles must be hashed so that each particle is assigned a bin number based on their location. This determines which particle IDs are in a bin at a given time. I performed this by referring to a separate function called 'initializeBins2'.

```
bins=initializeBins2(Nx,Ny,N_new,particles,dx,dy,ymax);
```

We now go to the function 'initialiazeBins2' found in a separate script to analyze the code. As always, the function is created by placing the function header in the first line of the script with the input and output variables. Next, the bin arrangement of the entire grid space we defined earlier is assigned. This is simply done by numbering the grid spaces, starting with 1 in the upper-left corner and counting down the columns.

```
function bins=initializeBins2(Nx,Ny,N_new,particles,dx,dy,ymax)

%calculate the bin arrangement
bin_num=1;
for i=1:Ny
    for j=1:Nx
        binnum(j,i)=bin_num;
        bin_num = bin_num+1;
    end
end
```

Based off of these numbered bins, we must find which bins are adjacent to each other to know which particles are neighbors based off of their bin numbers. This is done in a way similar to the ellipse calculations problem in Homework 2. Each bin is evaluated one at a time, and its neighbors are determined by manipulating the indices. If the calculated index goes beyond the size of the grid, there are boundary conditions that are entered to take them into consideration. Essentially what the code below does is it creates a 'bins' structure that contains a field called 'adjacentBins' which holds the IDs of the bins that are neighboring the bin in question.

```
%find which bins adjacent to each other
for i=1:Ny
    for j=1:Nx
        north = i-1;
        south = i+1;
        west = j-1;
        east = j+1;
        %account for boundaries
        if north<=0
            north = i;
        end
        if south>Ny
            south = i;
        end
        if west<=0
            west = j;
        end
        if east>Nx
            east=j;
        end
        bins(binnum(i,j)).adjacentBins = [binnum(north,j), binnum(south,j),
binnum(north,west), binnum(north,east), binnum(south,east),
binnum(south,west), binnum(i,east), binnum(i,west)];
        %make sure it doesn't include itself
        bins(binnum(i,j)).adjacentBins =
bins(binnum(i,j)).adjacentBins(bins(binnum(i,j)).adjacentBins~=binnum(i,j));
    end
end
```

Due to the way the code above is written, it will calculate repeating bin numbers in the 'adjancentBins' field, thus the unique function is entered below to ensure that no bin numbers are repeated.

```
%ensure no repetition
for i=1:length(bins)
   bins(i).adjacentBins = unique(bins(i).adjacentBins);
end
```

Now that the bin arrangement is determined, we can calculate the bin numbers of each particle using the formula provided to do this. The following for-loop goes through every particle in the 'particles' structure and stores their bin number in a separate array called 'particle_bin'.

```
%calculate bin number of particles
for k = 1:N_new
    x_pos = ceil(particles(k).pos(1)/dx);
    y_pos = ceil((ymax-particles(k).pos(2))/dy);
    particle_bin(k) = (x_pos-1)*Ny + y_pos;
end
```

The next for-loop then iterates for each bin number and evaluates which indices in particle_bin are equal to that bin number and stores those indices in another field of the 'bins' structure called 'particle_ID'. This field contains the IDs of the particles that are currently in that bin.

```
for k=1:N_new
    index=find(particle_bin==k);
    if isempty(index)
        bins(k).particle_ID=[];
    else
        bins(k).particle_ID=index;
    end
end
```

Now that this function has been created and the bins.particleIDs list has been built, the list of neighbors for each particle can be built. This is done using another separate function 'getNeigh2'. In the main script, it is called to as shown below.

```
particles = getNeigh2(particles,bins,h);
```

Within the separate script for the 'getNeigh2' function, the function header is entered at the beginning. The process for determining the list of neighbors for every particle is embedded within several for-loops within each other. What this code does is it calculates the distance between particles if they are located in the same or neighboring bins. For the particles in a given bin, they are compared against all the other particles in that same bin and all the particles in their neighboring bins. If the distance between the particles is less than that of the smoothing radius, h, then the corresponding particle ID is added as a neighbor. It is added as a neighbor in the 'neigh' field of the 'particles' structure.

```
for z=1:length(bins)
    zptcs = bins(z).particle_ID;
    w =[z, bins(z).adjacentBins];
    wparticles = [];
    for i = 1:length(w)
        w_iparticles = [bins(w(i)).particle_ID];
        wparticles = [wparticles, w_iparticles];
    end
    for k = 1:length(zptcs)
        A = zptcs(k);
        particles(A).neigh = [];
        for j = 1:length(wparticles)
            B = wparticles(j);
            x_y=particles(A).pos-particles(B).pos;
            dist = sqrt(x_y(1)^2 + x_y(2)^2);
            if dist<h && k~=j
                particles(A).neigh = [particles(A).neigh , B];
            end
        end
    end
end
```

Now that the list of neighbors for each particle has been built, we can calculate the fluid density represented at each point, as density depends on the neighbors. A for-loop that iterates for every particle is created. For the density of a particle, there are contributions from the neighbors of that particle that are included in the density calculation. To take this into consideration, the sum of the neighbors' contribution 'totaldens' is zeroed for every particle iteration. Then another for-loop is entered to loop through the neighbors of the specified particle, calculating each of their density contributions and adding it to the 'totaldens'. Once this is done for each neighbor, the particle's density is calculated using the formula provided to us.

```
%calculate the density for each particle
for k=1:N_new
    totaldens=0;
    for j=1:length(particles(k).neigh)
        neighbor = particles(k).neigh(j);
        x_y=particles(k).pos-particles(neighbor).pos;
        dist = sqrt(x_y(1)^2 + x_y(2)^2);

        dens = (h^2 - dist^2)^3;
        totaldens = totaldens + dens;
    end
    particles(k).density = (4*mass)/(pi*h^2) + (4*mass)/(pi*h^8)*totaldens;
end
```

Now that the density values are stored for each particle, we can make use of them to calculate the total force on each particle. The force on a particle entails the fluid force contributions from neighbors and the external forces of gravity and wind. Similar to density, force also depends on contributions from its neighbors for fluid force. The same for-loop structure that was used to calculate density is also implemented to calculate the force of each particle. The formula provided to us is used. Each calculated force is stored into the 'force' field of the 'particles' structure.

```
%calculate the total force on each particle
%gravity, pressure, and viscoscity
for k=1:N_new
    totalfluid = 0;
    %gravity, wind forces
    gravity_f = [0,-9.8]*particles(k).density;
    wind = [1,0];
    for j=1:length(particles(k).neigh)
        neighbor = particles(k).neigh(j);

        x_y=particles(k).pos-particles(neighbor).pos;
        dist = sqrt(x_y(1)^2 + x_y(2)^2);

        q_kj = dist/h;
        b = mass*(1-q_kj)/(pi*particles(neighbor).density*h^4);
        c = 15*kappa*(particles(k).density+particles(neighbor).density-
2*rho_0);
        d = (1-q_kj)/q_kj*x_y;
        e = 40*mu*(particles(k).vel - particles(neighbor).vel);

        fluid_force = b*(c*d - e);
```

```
        totalfluid = totalfluid + fluid_force;
    end
    particles(k).force =  gravity_f + wind + totalfluid;

end
```

Now that the total force acting on each particle is known, its position and velocity are updated using the discretized equations that were deducted from the kinematics equations using semi-implicit Euler. This is only done for the particles that are meant to simulate fluid behavior. So for the case of the complicated set-up simulation, it is only done for the fluid particles and not the obstacle particles as we would like these to remain stationary obstacles. This is why the loop is only iterated for particles 1 through N. The fields of the particles are referred to update the kinematics as shown below.

```
%update the velocity and position of each particle
for k=1:N
    particles(k).vel = particles(k).vel +
dt*particles(k).force/particles(k).density;
    particles(k).pos = particles(k).pos + dt*particles(k).vel;
end
```

After updating the kinematics of each particle, there are boundary conditions that must be implemented so that the particles don't exceed the limits of the entire grid. This is done using the formula provided in the problem statement.

```
%boundary conditions
beta=0.75;

for k=1:length(particles)
    if particles(k).pos(1)>xmax
        particles(k).pos(1) = 2*xmax - particles(k).pos(1);
        particles(k).vel(1) = -beta * particles(k).vel(1);

    elseif particles(k).pos(2)>ymax
        particles(k).pos(2) = 2*ymax - particles(k).pos(2);
        particles(k).vel(2) = -beta * particles(k).vel(2);

    elseif particles(k).pos(1)<0
        particles(k).pos(1) = 0 - particles(k).pos(1);
        particles(k).vel(1) = -beta * particles(k).vel(1);

    elseif particles(k).pos(2)<0
        particles(k).pos(2) = 0 - particles(k).pos(2);
        particles(k).vel(2) = -beta * particles(k).vel(2);
    end
end
```

This reflects the particle onto the correct side of the domain and reverses its velocity with an amount of damping(beta).

Now that the kinematics have been updated properly, we can now visualize the results. To do

this, I created xFinal and yFinal vectors that contain the final x and y positions of this time loop. This is done using a for-loop that goes through every particle position and stores the x-positions and y-positions into their respective final vectors.

```
%visualize results
%create vector containing x of each particle
xFinal = [];
for k=1:length(particles)
    x_pos = particles(k).pos(1);
    xFinal =  [xFinal,x_pos];
end

yFinal=[];
for k=1:length(particles)
    y_pos = particles(k).pos(2);
    yFinal = [yFinal, y_pos];
end
```

Now, we can scatter these final position vectors using the scatter function. The axes of the figure are set, the grid is on, and the tick marks are distinguished.

```
scatter(xFinal, yFinal, 'filled')
axis([0 xmax 0 ymax])
grid on

xticks(0:dx:xmax)
yticks(0:dy:ymax)
```

An if statement is then entered to either create a movie if we have saveMovie equal to true or to simply apply drawnow to the scatter plot so that we can simulate the animation.

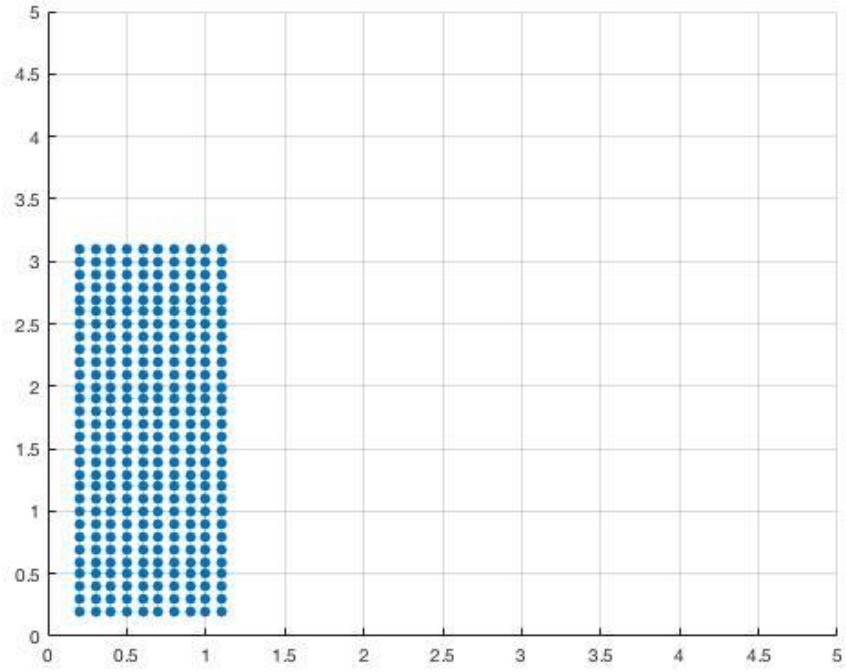
```
if saveMovie == true
    % STEP 4: Add the current frame to the accumulated movie
    writeVideo(vidHandle, getframe(gcf));
else
    % If we're not saving a movie, use drawnow
    drawnow;
end
```

The overarching time loop is then ended and once the movie is finished writing, the file is closed as shown in the code below.

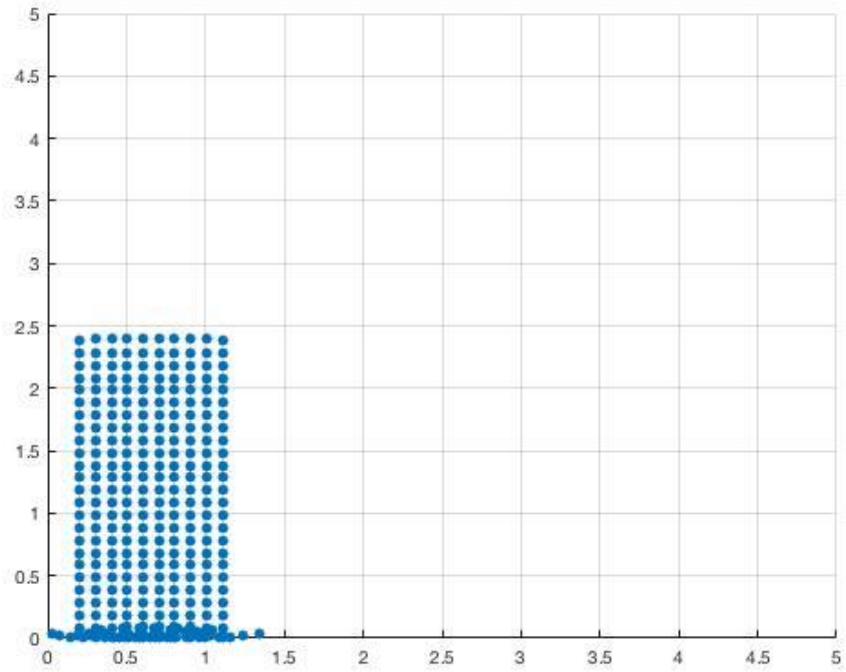
```
if saveMovie == true
    % STEP 5: Close the file when finished writing
    close(vidHandle);
end
```

3. Calculations and Results

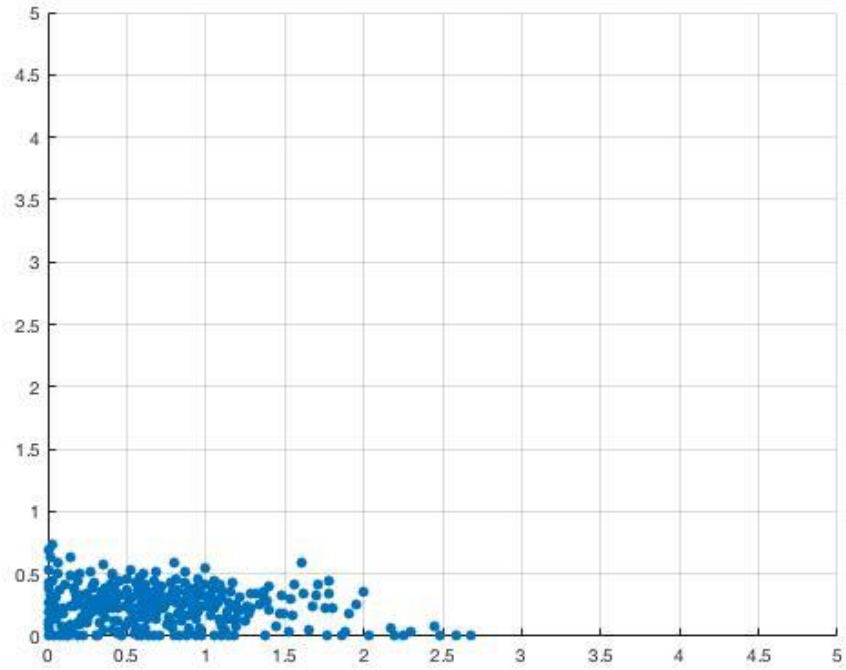
When the script is run for the user input method=1, the “dam break” simulation is produced. At the beginning of the simulation, the particles are positioned in a grid-like manner as shown below.



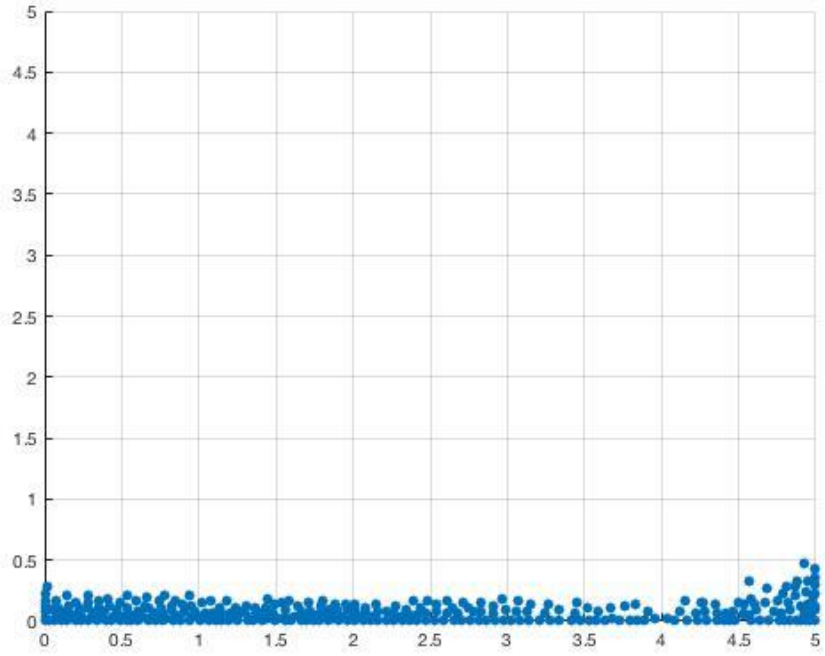
The particles then begin to fall due to gravity, and because rho_0 is very similar to these particles’ densities, the grid of particles does not contract or expand a whole lot when it falls.



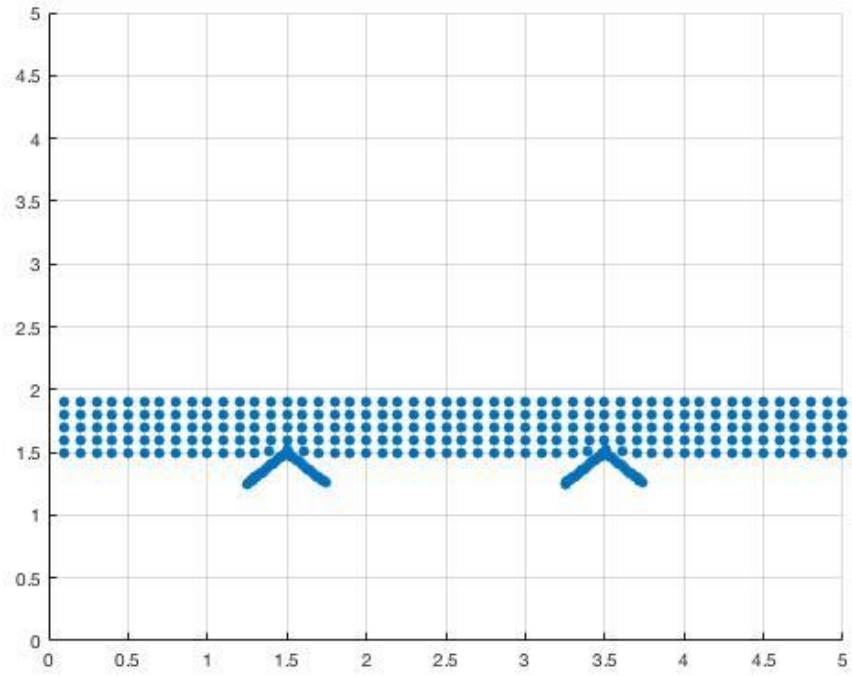
As the particles hit the “ground”, they bounce back up due to the boundary conditions and due to the force exerted on them by their surrounding particles. Since, the stiffness is not very high (kappa=4.6), the bounciness of the particles is present but not so much so that it seems like the particles are floating in air. This was the kappa that provided the best simulation that resembled that of a fluid.



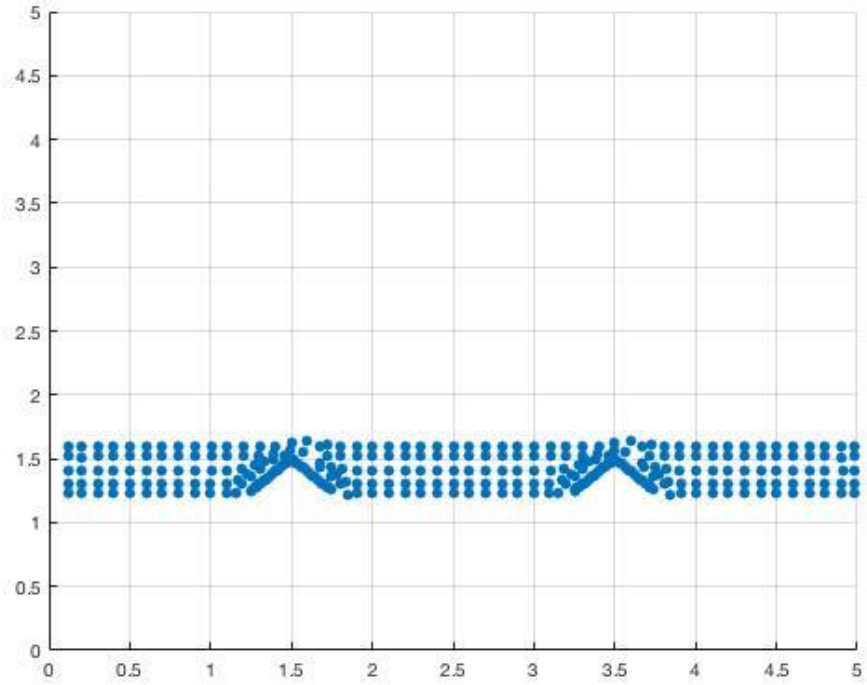
As the simulation continues, the particles eventually spread out along the base of the grid to reach the rest density. They continue until they are spread out evenly, simulating a fluid motion along the way.



When the script is run for the user input method=2, the complicated set-up simulation is produced. My complicated set-up consists of two obstacles shaped like arrow-heads that the particles initially pass over. The beginning of the simulation is as shown below.

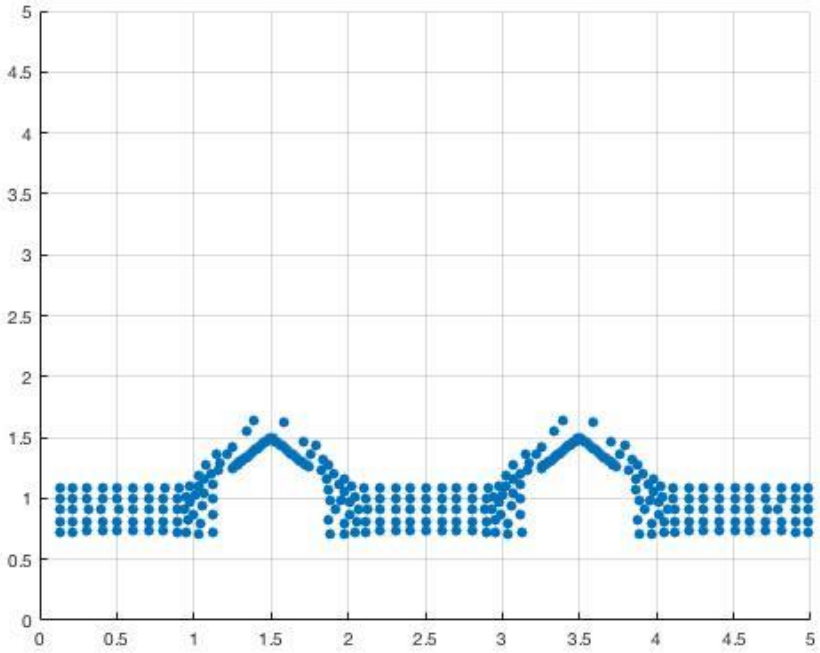


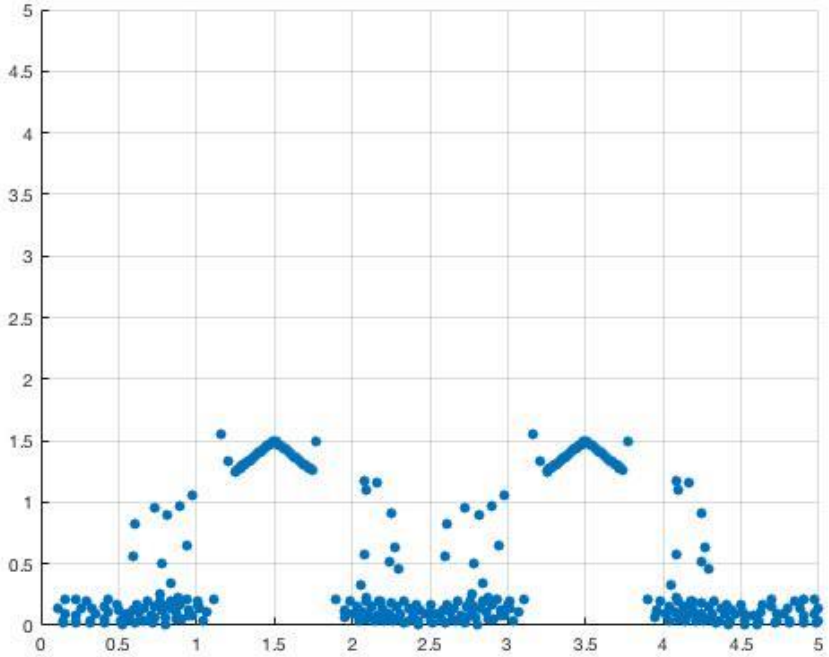
As the particles come into contact with the obstacles, the density of the obstacle particles and the force interactions between the obstacle particles and the fluid particles cause the fluid particles to roll over the sides of the obstacles and not fall straight through them. Since the fluid particles are initialized very close to the obstacles, they haven't acquired enough energy to penetrate the obstacles, which is what we want. A picture of the simulation at a later time is shown below. Notice how the particles roll over the sides of the obstacles.

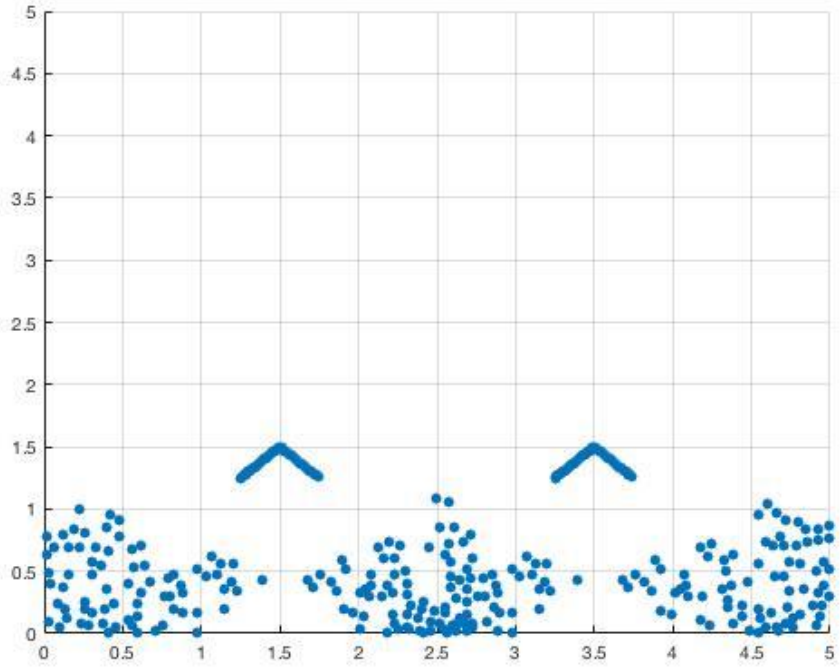


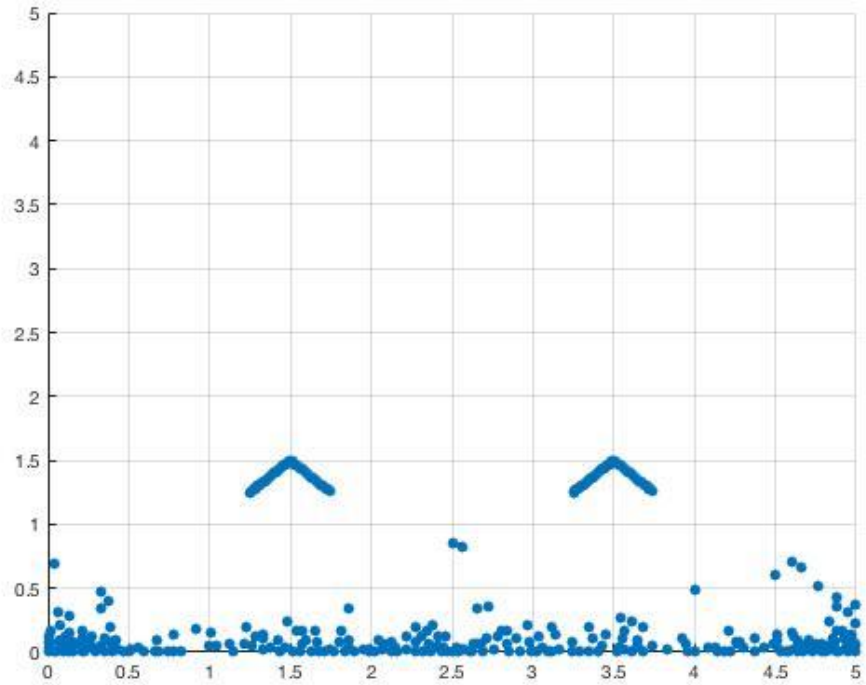
The fluid particles that do not come into contact with the obstacles simply continue to fall due to gravity. They do not contract or expand as their density values are similar to that of rest density.

As the particles continue to fall to the ground, they bounce of the ground due to their boundary conditions and due to their stiffness. Then they continue to simulate based on the interactions with their neighboring particles and based on their set property values. They eventually settle on the bottom of the grid just as a fluid settles. Several still images of the simulation at succeeding times are shown below.


5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5


5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5


5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5


5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5

4. Discussion

The value of the stiffness constant, kappa, influenced my simulation in that when it was a higher value, my particle behavior was "floatier and bouncier". Essentially, my particles wouldn't settle quite nicely like a fluid does. I believe this is because the "stiffer" a particle is, the greater the bounce effect is when two particles come into contact, like two stiff rubber balls do in reality. The value of the viscosity coefficient, mu, influenced my simulation in that when it was a higher value, my particles really stuck together rather than "flowing" over each other in a smooth manner.

My simulation handled the mixing of two different fluids with different rest densities not too greatly, especially if the two fluids were initialized in close proximity of each other. This is because if I don't change the mass of these particles, the particles with a higher rest density will want to contract to reach their rest density, which will also influence the behavior of the other particles with a density already similar to rest density. In order to normalize their behavior, I would have to manipulate the mass of the particles whose rest density I increased, so that their initial densities match their new rest density.

To simulate a fluid inside a circular domain centered at (0, 0), the positions of each particle would be characterized by its distance from the center and by its angular position with respect to the positive x-axis. The circular domain would be hashed by creating concentric circles with different radii and creating extended lines at certain angles from the center (0, 0) to R_{max}. To test the particles crossing a boundary, I would implement an if statement saying that if the particle's distance from the center is greater than R_{max}, then its position and velocity are updated accordingly so that the particle bounces off the boundary. Pseudocode below.

```
if particles(k).distancefromradius>Rmax

        particles(k).distancefromradius = 2*Rmax – particles(k).distancefromradius;

end
```

(and so on)