Jeffrey Ding
UID: 104928991
CEE M20
June 10th, 2018

Final Project

1. Introduction

The goal of this problem is to study fluid behavior in 2D using smoothed particle hydrodynamics (SPH) on a spatially-hashed domain.

2. Model and Methods

The script begins by setting the number of particles, creating a particles array full of a custom struct, and assigning positions for each particles:

```
N = 100; % number of particles

% create N particles
particles(1:N) =
struct('x',[],'y',[],'vel',struct('v_x',0,'v_y',0),'force',struct('f_x',0,'f_y',0),'rho',[],'neigh',[]);

% assign positions for each particle
for index = 1:N
   particles(index).x = rand();
   particles(index).y = rand();
end
```

For the struct, the script maps each particle to a x and y coordinate. There is also a nested struct called vel that gives each particle directional velocities in x and y directions, and nested struct force that gives each particle directional forces in x and y directions, a scalar rho that gives each particle a density, and a vector neigh that contains neighboring particles to each particle.

Then, the script sets the constant properties of the fluid being simulated, with these values being the default constants:

```
% set the properties of the fluid being simulated
p0 = 1000; % Rest density
m = p0/N; % Mass of single particle
stiffness = 100; % Stiffness constant
mu = 0.1; % Viscosity coefficient
h = 0.25; % smoothing radius
```

Then, the script sets the grid of the hash, calculating the bin dimensions using a grid size:

```
% sets grid
xMax = 1;
yMax = 1;

% bins
Nx = floor(xMax/h); % number of columns
Ny = floor(yMax/h); % number of rows
% bin dimensions
dx = xMax/Nx;
dy = yMax/Ny;
```

Then, the script creates another array of bin structs, with each struct holding a vector, particleIDs, containing ID's of all particles within each bin, and a vector, adjacentBins, containing ID's of all bins adjacent to each bin. It uses a for loop to determine which particles are in each bin:

```
% construct Nx*Ny bins
numBins = Nx*Ny;
bins(1:numBins) = struct('particleIDs',[],'adjacentBins',[]);

% calculate the bin number for every particle
for k = 1:N
   binNum = (ceil(particles(k).x/dx)-1)*Ny + ceil((yMax-particles(k).y)/dy);
   bins(binNum).particleIDs = [bins(binNum).particleIDs, k];
end
```

Afterwards, the script goes through each bin and populates the adjacentBins vector of each bin. The script determines whether or not the bin is a corner bin, a edge bin, or a inner bin and calculates accordingly:

```
% populate adjacentBins
for z = 1:numBins
   if z == 1 % top left corner
      bins(z).adjacentBins = [z+1,z+Ny,z+Ny+1];
   elseif z == Ny % bottom left corner
      bins(z).adjacentBins = [z-1,z+Ny,z+Ny-1];
   elseif z == numBins-Ny+1 % top right corner
      bins(z).adjacentBins = [z+1,z-Ny,z-Ny+1];
   elseif z == numBins % bottom right corner
      bins(z).adjacentBins = [z-1,z-Ny,z-Ny-1];
```

```
    elseif z < Ny % left edge
        bins(z).adjacentBins = [z-1,z+1,z+Ny-1,z+Ny,z+Ny+1];
    elseif rem(z,Ny) == 1 % upper edge
        bins(z).adjacentBins = [z+1,z-Ny,z-Ny+1,z+Ny,z+Ny+1];
    elseif rem(z,Ny) == 0 % bottom edge
        bins(z).adjacentBins = [z-1,z-Ny,z-Ny-1,z+Ny,z+Ny-1];
    elseif z > numBins-Ny+1 % right edge
        bins(z).adjacentBins = [z+1,z-1,z-Ny,z-Ny+1,z-Ny-1];
    else % middle bins
        bins(z).adjacentBins = [z+1,z-1,z+Ny,z+Ny+1,z+Ny-1,z-Ny,z-Ny-1,z-Ny+1];
    end
end
```

Then, the script prepares the timesteps by setting a time variable, t, a dt variable holding difference in time, and a damping variable for collision of walls. The script also sets up movie variables to record the data:

```
% timestep
t = 1;
dt = 0.01;
damping = 0.75; % for wall collisions

% movie variables
vid = VideoWriter('hydrodynamics', 'MPEG-4');
vid.FrameRate = 30;
vid.Quality = 100;
% start video
open(vid);
```

Then, the script runs a while loop to calculate particle movement:

```
while t < 5
    particles = identifyNeighbors(numBins, bins, h, particles, N);
    particles = calculateDensity(m, h, particles, N);
    particles = calculateForce(m, h, mu, particles, N);
    particles = updateKinematics(dt, particles, N, xMax, yMax, damping);

    % hold axes
    hold on;
    cla;

    % display data
    axis([0 xMax 0 yMax]);
```

```
    scatter([particles.x],[particles.y]);
    drawnow;

    % write to video
    writeVideo(vid, getframe(gcf));

    % timestep
    t = t + dt;
end
```

First, the script calls the function identifyNeighbors to populate each particle's neigh data member. The script takes in 5 inputs, a scalar, numBins, representing the number of bins, an array, bins, representing all bins, a scalar, h, representing smoothing radius, an array, particles, representing each individual particle, and a scalar, N, represeenting number of particles.

The function goes through each bin, gets all particles in the bin and a vector of bins adjacent to it. Then, the function iterates through all adjacent bins to collect all particles in each adjacent bins and puts it into another array called closeParticles that has all particles in the current bin and the current adjacent bin. Then, the function iterates through all particles in the current bin and for each particle iterates through all nearby particles. In each iteration, the function calculates the distance between the current particle and the current close particle and determines whether or not they are neighbors. If they are, put the close particle into the current particle's neigh vector:

```
% identify neighbors
for z = 1:numBins
    particlesInZ = bins(z).particleIDs; % get all particles in bin z
    adjacentBinsZ = bins(z).adjacentBins; % get vector of bin IDs adjacent to bin Z
    lengthAdjacentBins = length(adjacentBinsZ);
    for w = 1:lengthAdjacentBins
        particlesInW = bins(adjacentBinsZ(w)).particleIDs; % get all particles in bin W
        closeParticles = [particlesInW, particlesInZ]; % combine particles in bin W and bin Z
        for k = 1:length(particlesInZ) % all particles in bin Z
            for j = 1:length(closeParticles) % all particles in bin W
                % calculate distance
                dist = sqrt((particles(closeParticles(j)).x-particles(particlesInZ(k)).x)^2 +
(particles(closeParticles(j)).y-particles(particlesInZ(k)).y)^2);
                if dist < h && closeParticles(j) ~= particlesInZ(k)
                    % particles k and j are neighbors
                    particleK = particlesInZ(k);
                    particleJ = closeParticles(j);
                    particles(particleK).neigh = [particles(particleK).neigh, particleJ];
```

```
            end
        end
    end
  end
end
```

Then, the function goes through each particles and eliminates any duplicate neighbors:

```
% eliminate duplicate neighbors
for index = 1:N
  particles(index).neigh = unique(particles(index).neigh);
end
```

Back to the main script, the next function called is the calculateDensity function. The function takes in 4 inputs, a scalar, m, representing the mass of a single particle, a scalar, h, representing smoothing radius, an array, particles, representing particles, and a scalar, N, representing number of particles. The function returns a vector of particles with the density values.

The function models a equation and first calculates two constants, a constant, first, to be added, and a constant, secondCo, to be multiplied. Then, the function iterates through each particles and retrieves a vector of the particle's neighbors. Then, the function iterates through the vector of neighbors and uses a number of data members to add to another constant, secondSum. After iterating through each neighbor particle, calculate the current particle's density by adding the first variable to the product of secondCo and secondSum:

```
% calculate density
first = (4*m)/(pi*h*h);
secondCo = (4*m)/(pi*h^8);
for index = 1:N
  currentParticle = particles(index);
  neighborsLength = length(currentParticle.neigh);
  secondSum = 0;
  % iterate through neighbors
  for neighborIndex = 1:neighborsLength
    currentNeighbor = particles(currentParticle.neigh(neighborIndex));
    secondSum = secondSum + (h^2 - (sqrt((currentParticle.x-currentNeighbor.x)^2 +
(currentParticle.y-currentNeighbor.y)^2))^2)^3;
  end
  particles(index).rho = first + secondCo*secondSum;
end
```

The next function the main script calls is calculateForce. The function takes in 5 inputs: a scalar, m, representing the mass of a single particle, a scalar, h, representing smoothing radius, a scalar, mu, representing the viscosity coefficient, an array, particles, representing the particles, and a scalar, N, representing number of particles. The function returns a vector of particles with force values in x and y directions.

The function iterate through each particle and first calculates the external forces of each particle according to its density and setting a variable, sum, to [0, 0]. Then, the function iterates through the current particle's neighbors and calculates the force caused by pressure and the force caused by viscosity and adds each one to the total sum array. After iterating through each neighboring particle, the function adds external forces to the sum force vector and sets them to the current particles respective directional forces:

```
% calculate force
for index = 1:N
   currentParticle = particles(index);
   neighborsLength = length(currentParticle.neigh);
   external = [0,-9.8]*currentParticle.rho + [1,0];
   sum = [0,0];
   % iterate through neighbors
   for neighborIndex = 1:neighborsLength
      currentNeighbor = particles(currentParticle.neigh(neighborIndex));
      % pressure
      first = (-30*m)/(pi*currentNeighbor.rho*h^4);
      second = (currentParticle.rho+currentNeighbor.rho)/2;
      qkj = (sqrt((currentParticle.x-currentNeighbor.x)^2 +
(currentParticle.y-currentNeighbor.y)^2))/h;
      third = ((1-qkj)^2)/qkj;
      fourth = [currentParticle.x - currentNeighbor.x, currentParticle.y -
currentNeighbor.y];
      sum = sum + first*second*third*fourth;
      % viscosity
      first = (-40*m*mu)/(pi*currentNeighbor.rho*h^4);
      second = 1 - qkj;
      third = [currentParticle.vel.v_x - currentNeighbor.vel.v_x, currentParticle.vel.v_y -
currentNeighbor.vel.v_y];
      sum = sum + first*second*third;
   end
   finalVector = external + sum;
   particles(index).force.f_x = finalVector(1);
   particles(index).force.f_y = finalVector(2);
end
```

Finally, the last function the main script calls is the updateKinematics function. This function takes in 6 inputs: a scalar, dt, representing a time step, an array, particles, representing particles, a scalar, N, representing number of particles, a scalar xMax, representing the right boundary, a scalar, yMax, representing the top boundary, and a scalar, damping, for wall collisions. The function returns a vector of particles with updated directional velocities and positions using semi-explicit euler's method to calculate them.

The function iterates through each particles and updates the velocity using the current particle's force and rho values. Then, the function updates the position of the current particle using the newly acquired velocity values. Right before moving on to the next particle, the function checks whether or not the particle went past the boundaries. If the particle went through the right boundary, the particle's position is reflected over the right boundary and its velocity in the x direction is flipped and damped. The same goes for the left boundary, except the position is reflected over the left boundary. If the particle went through the top boundary, the position is reflected over the top boundary and the velocity in the y velocity is flipped and damped. Finally, the same goes for the bottom boundary except the position is reflected over the bottom boundary:

```
for index = 1:N
    % update velocity
   particles(index).vel.v_x = particles(index).vel.v_x +
(dt*particles(index).force.f_x)/particles(index).rho;
   particles(index).vel.v_y = particles(index).vel.v_y +
(dt*particles(index).force.f_y)/particles(index).rho;

   % update position
   particles(index).x = particles(index).x + dt*particles(index).vel.v_x;
   particles(index).y = particles(index).y + dt*particles(index).vel.v_y;

   % check for boundary behavior
   if particles(index).x > xMax % right boundary
      particles(index).x = 2*xMax - particles(index).x;
      particles(index).vel.v_x = -damping*particles(index).vel.v_x;
   end
   if particles(index).x < 0 % left boundary
      particles(index).x = 0 - particles(index).x;
      particles(index).vel.v_x = -damping*particles(index).vel.v_x;
   end
   if particles(index).y > yMax % top boundary
      particles(index).y = 2*yMax - particles(index).y;
      particles(index).vel.v_y = -damping*particles(index).vel.v_y;
   end
```

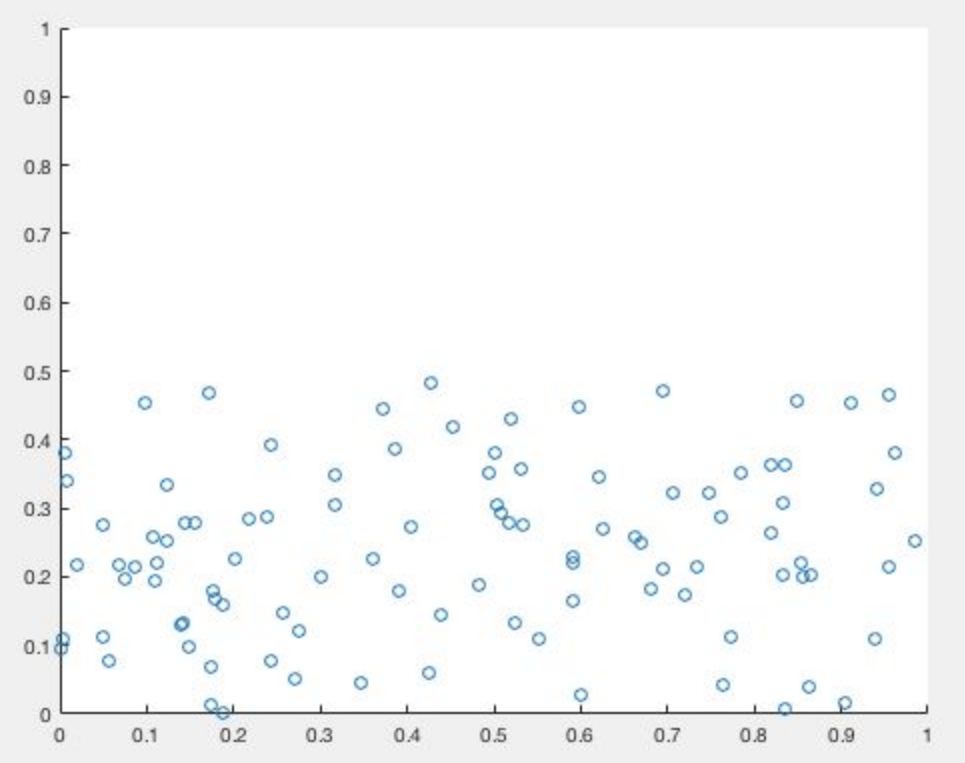
```
  if particles(index).y < 0 % bottom boundary
     particles(index).y = 0 - particles(index).y;
     particles(index).vel.v_y = -damping*particles(index).vel.v_y;
  end
end
```

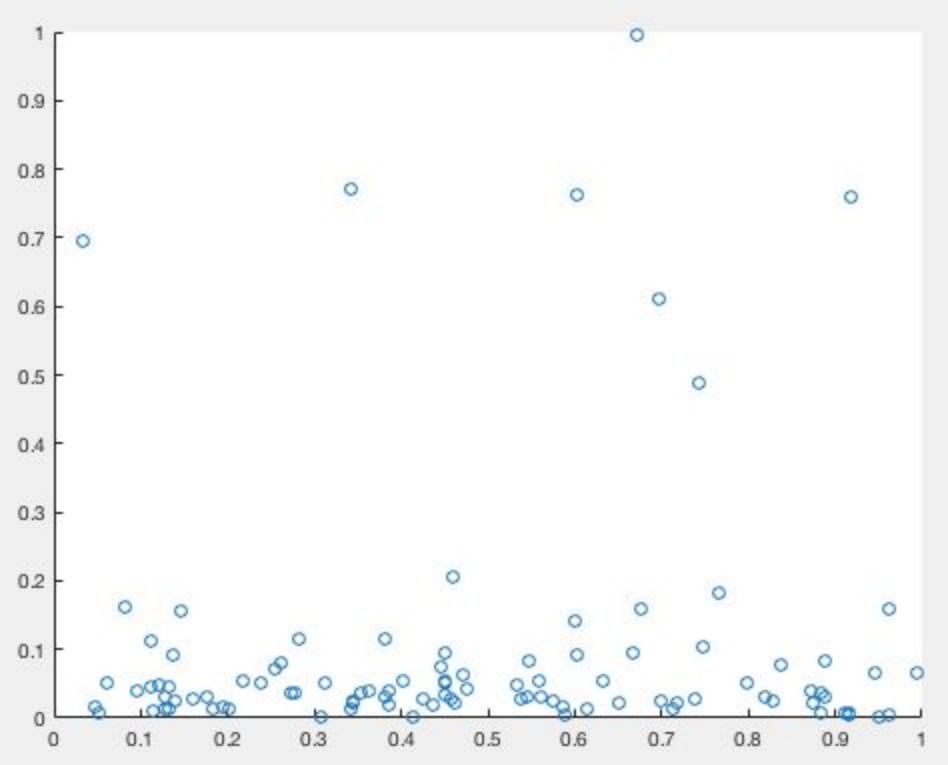
After each function call, the script displays the data on the scatter plot and the video is updated. Then, the next time step is updated. After the loop is done, the video is closed and saved:

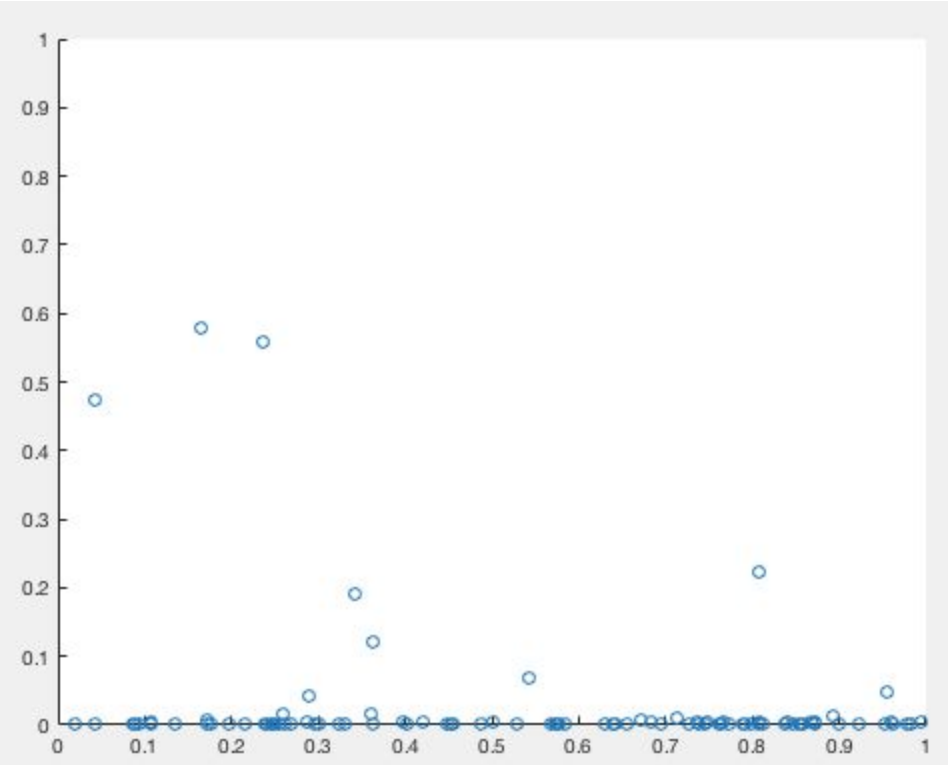
```
% end video
close(vid);
```

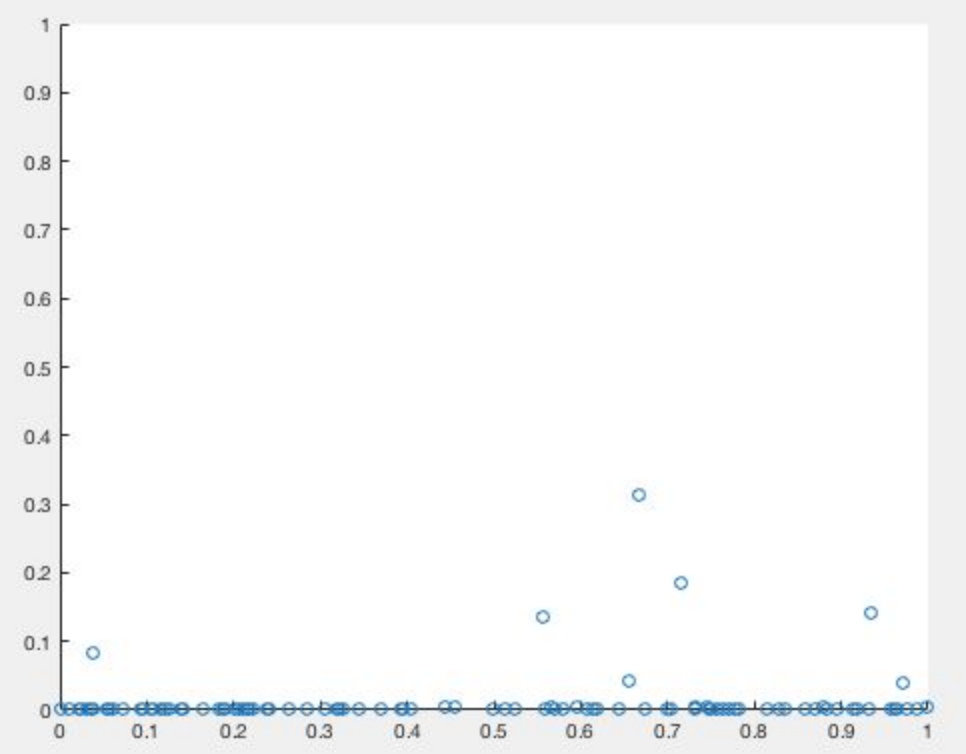
3. Calculations and Results

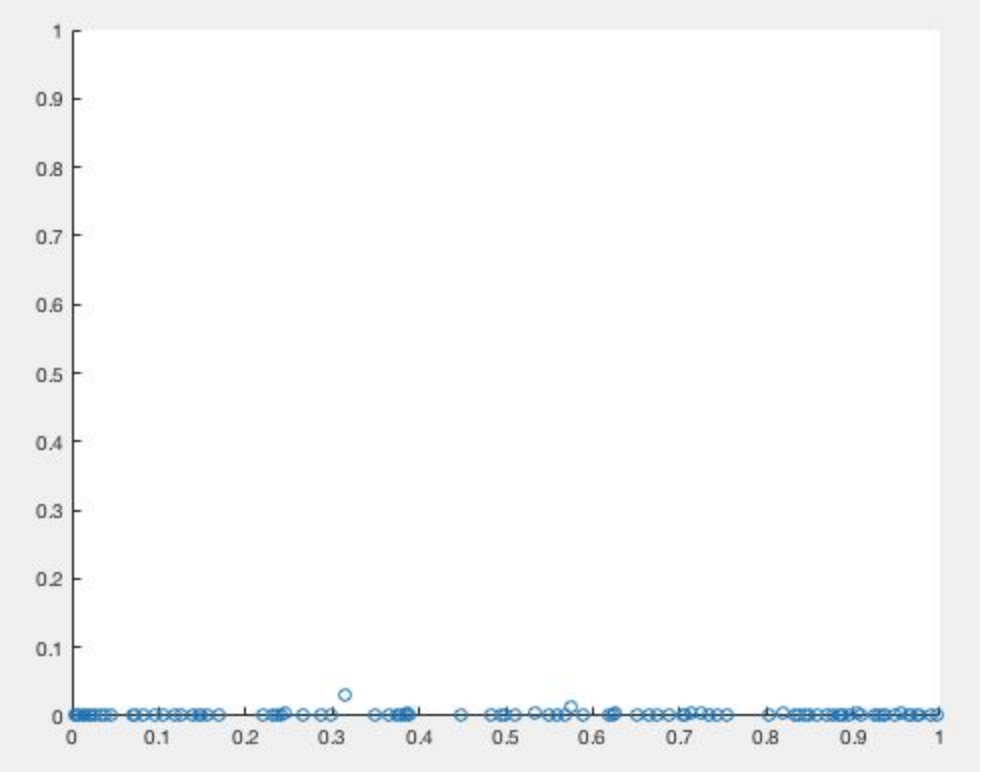
When the program is executed the following stills are created at (t == 1, 2, 3, 4, and 5) for the dam-break model:




1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1

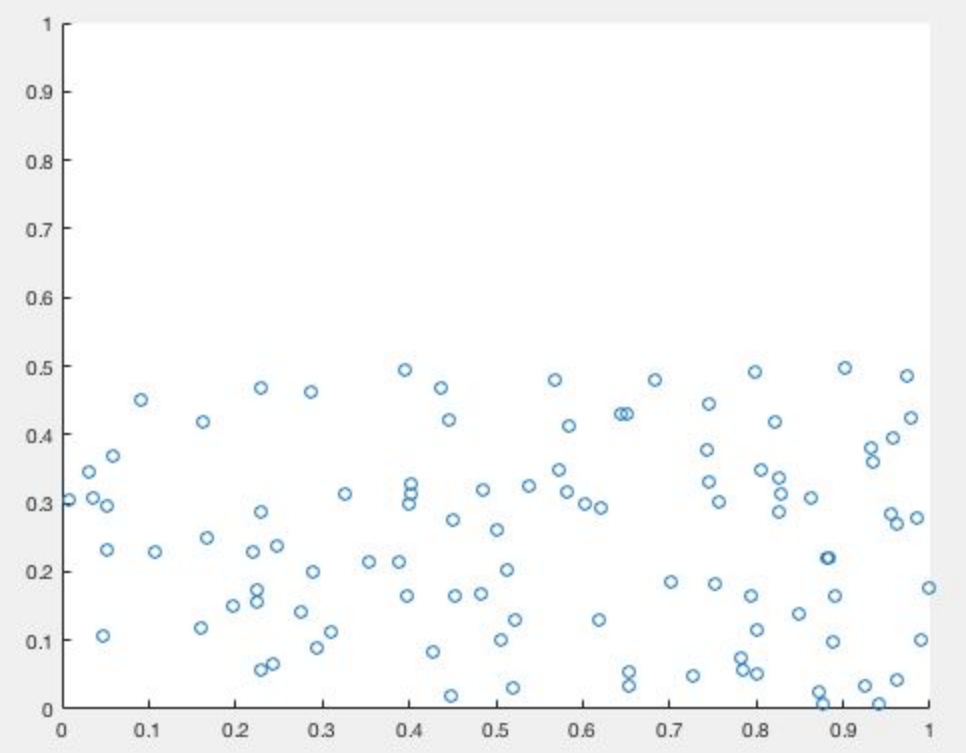

1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1

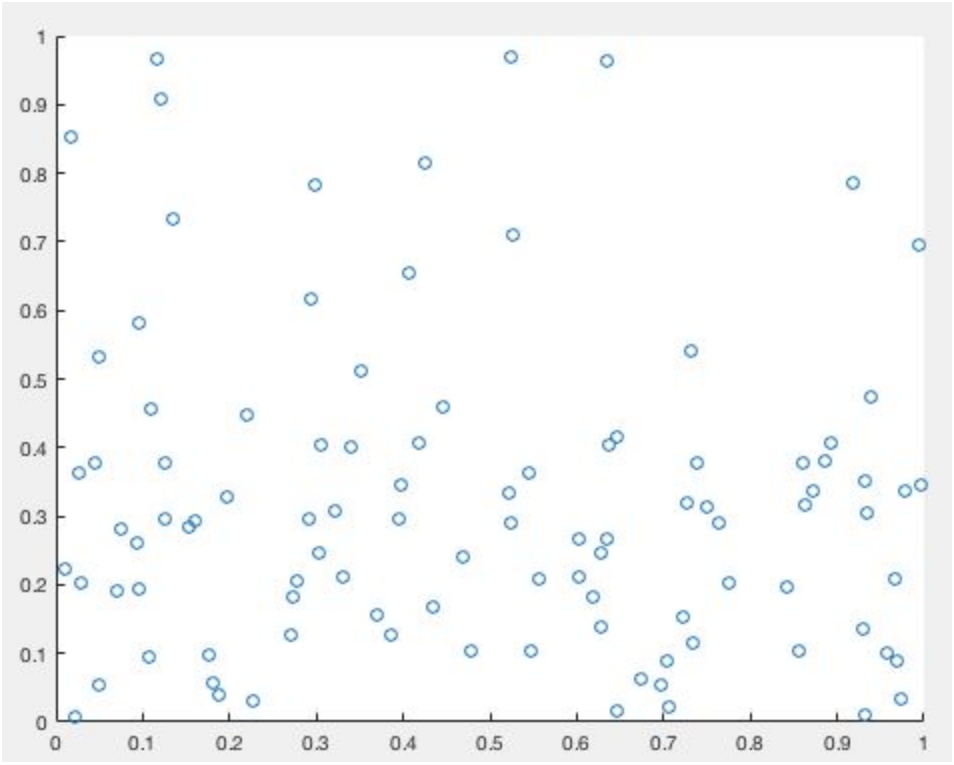

1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1

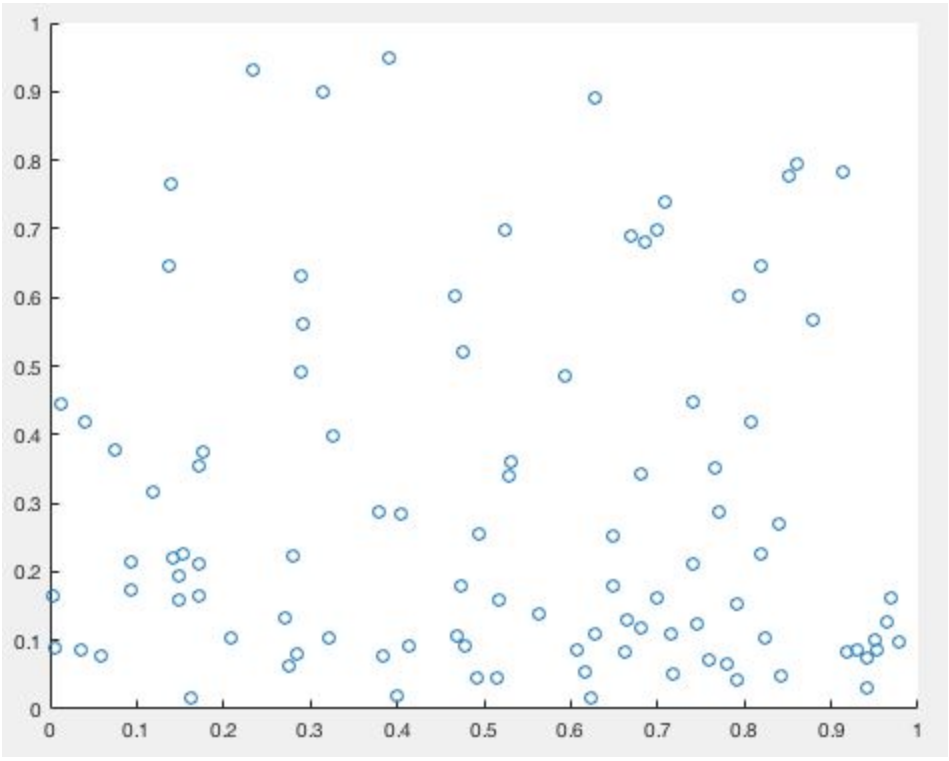

1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1

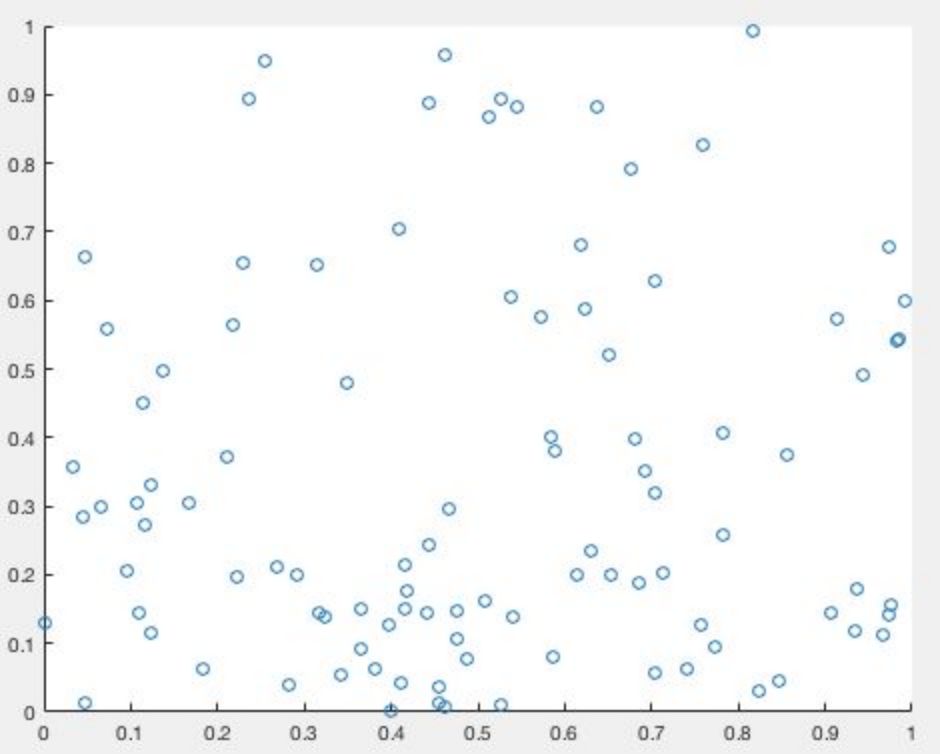
It can clearly be seen that as the particles are affected by gravity and time, more and more particles gravitate and coagulate at the bottom of the hashes. This is the expected behavior.

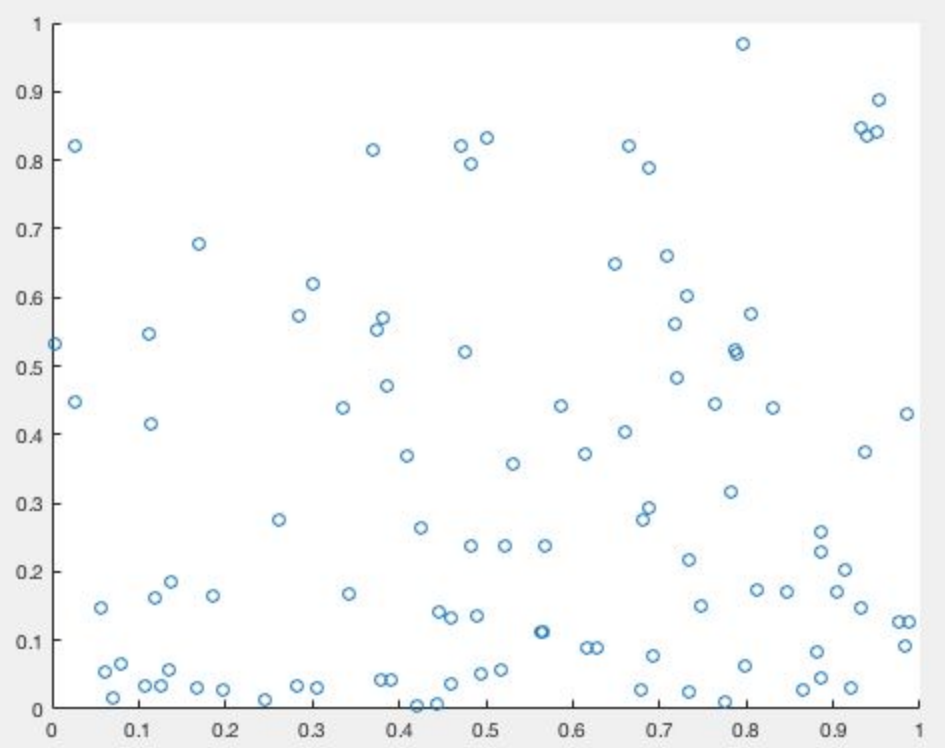
The following images are the same experiment, however, I removed damping so the wall don't affect momentum, thus particles hit walls and bounce off at the same speed that they hit:




0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1


0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1





The results show that without damping, the particles will never settle at the bottom of the hash, instead bouncing around for all of eternity.

4. Discussion

In the first experiment, the particles naturally gravitated to the bottom of the “well.” This is the expected behavior of this simulation because gravity naturally brings the particles in the downward direction. Furthermore, as the particle hit the boundaries, the damping from the wall collision resulted in a decrease in velocity in both vertical and horizontal direction. Thus, the particles eventually settled at the bottom of the well, moving only slightly due to really small forces from each particle acting on each other.

The second experiment represents particles when there is no damping at all. Thus, when a particle hits a wall, the speed of the particle stays constant. Basically, no energy is removed from the system when there is a collision. Thus, the particles just exist in a constant state of flux, moving around for all of eternity.

In both experiments, the stiffness constant and viscosity coefficient influences my simulation in that if I increased the stiffness constant, the particles would have more force applied to them, thus having a greater directional velocity. However, the opposite is true for decreasing stiffness. Furthermore, as I increased the viscosity coefficient, the forces being applied to each particle decreased, thus made all particles slower as more particles resisted the fluid environment. However, as I decreased viscosity, the particles were more free.

When I added two different fluids with different rest densities, the particles would behave slightly differently in that sometimes the particles stick together and are less fluid. Some of the particles end up sticking together and not moving fluidly while some particles get launched at a very high velocity during a collision. I assume this is explained by having two different fluid with different rest densities because some of the particles are very heavy and are not as affected by collisions, thus not moving too wildly. However, if these larger particles hit smaller particles, the smaller particles are launched due to the momentum transfer of collisions.

In order to simulate a fluid inside a circular domain centered at (0,0) with size Rmax, I would redesign the experiment by first modifying the placement of the particles to be in location that can only be in the domain. Afterwards, I would still use a 2Rmax by 2 Rmax grid in order to visualize the domain. In fact, I would even use the same spatial-hashing scheme because it is already developed and that it would still be of use because the script would just not calculate the hashes with no elements in them. In order to test for particles crossing the boundaries, the script would check if the distance between the particle and the center of the domain is greater than Rmax. If it is so, that means that the particle has exceeded the internal radius of the domain and must be reflected. However, the hardest part of this simulation is going to be boundary reflection update because you would need to determine on what boundary the particle hit. Furthermore, it isn’t just the problem of updating one direction of velocity but updating all directions of velocity

because the particle is hitting a circular domain. In order to solve this problem, the script will have to use the final position of the particle and determine at what angle the particle hit the boundary using trigonometry. Then, the script would have to reflect the the particle twice: once over a line from the radius to the location of collision of the particle. Then, the script would have to reflect the particle over the tangent of the location of collision, thus reflecting the particle to where the particle should be. Also, the script needs to re-adjust the velocity of the particle, so we use the same trig principles to recalculate the direction of the particle and apply damping in both x and y directions.