ELEC 4700 Assignment 1

Monte-Carlo Modeling of Electron Transport

Date: 2/5/2022

Student: Samuel (Wendi) Zhu

Student Number: 101088968

Git Repo: https://github.com/Samuelczhu/Monte-Carlo-Modeling-of-Electron-Transport

Table of Contents

1. Electron Modeling

In this part of the assignment, a simple Matlab program is written to model the electrons as particles with a simplistic Monte-Carlo model using the following values [1]:

- Rest mass: $m_o = 9.1093837015 \times 10^{-31}\ KG$
- Effective mass of electrons: $m_n = 0.26 m_0$
- Normal size of the region: 200nm × 100nm
- Boltzmann constant: $K = 1.38064852 \times 10^{-23}\ J \cdot K^{-1}$
- Temperature: 300K

The codes for part 1 are in the folder named "Q1", which contains the following files:

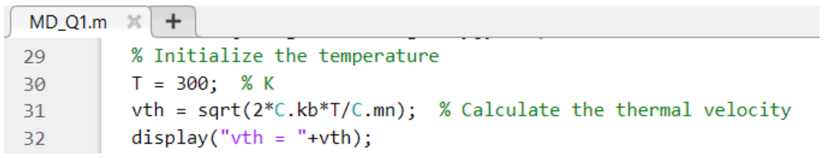
- AddElectrons.m –Function that adds electrons randomly in the region
- globalVars.m – File that groups the constants used in the simulation
- MD_Q1.m – Entry point script for the simulation for part 1
- PlotPoint.m – Function that plots the trajectories of the electrons
- tempDisplay.m – Function that displays the temperature map

1. What is the thermal velocity v_{th}? Assume $T = 300\,\mathrm{K}$.

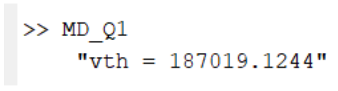
The thermal velocity V_{th} can be calculated using the following equations:

$$\because \frac{1}{2} m v_{th}^2 = \frac{2}{2} KT$$

$$\therefore v_{th} = \sqrt{\frac{2kT}{m}} = 187019.1244\ m/s$$



```
% Initialize the temperature
T = 300;  % K
vth = sqrt(2*C.kb*T/C.mn);  % Calculate the thermal velocity
display("vth = "+vth);
```



```
>> MD_Q1
    "vth = 187019.1244"
```

2. If the mean time between collisions is $\tau_{mn} = 0.2\,\mathrm{ps}$ what is the mean free path?

If the mean time between collisions is $\tau_{mn} = 0.2\ ps$, the mean free path (MFP) can be calculated as follow:

$$MFP = v_{th} \times \tau_{mn} = 3.7404 \times 10^{-8}\ m$$

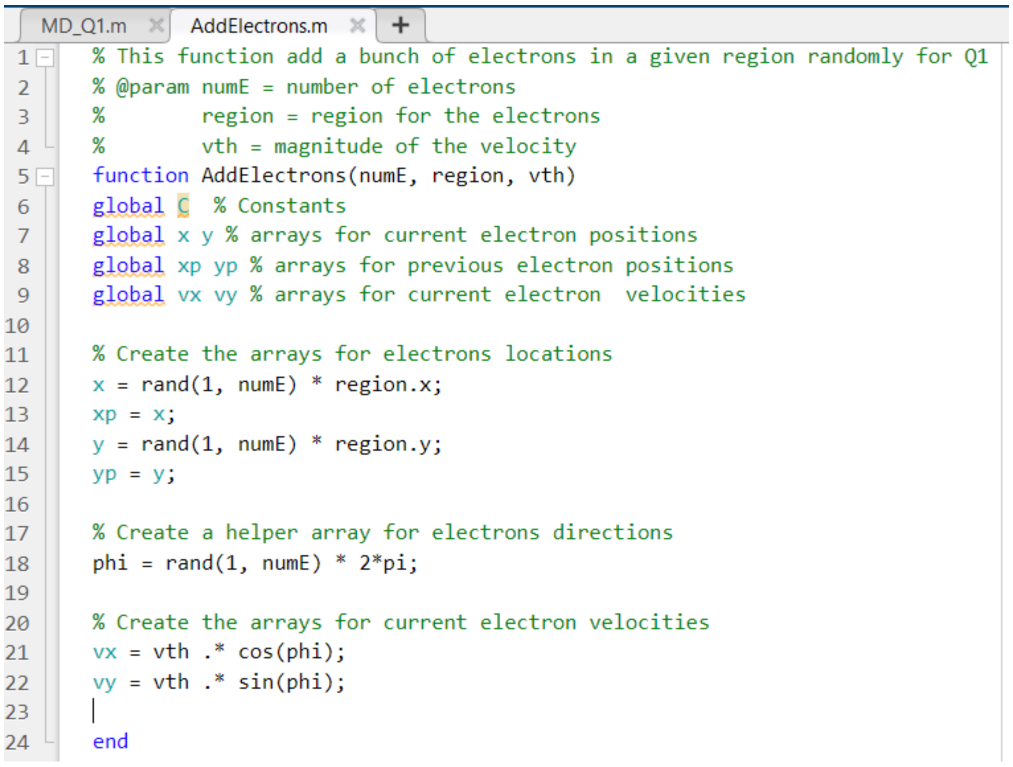
```
    % Initialize the mean time between collision
    Tmn = 0.2e-12;  % 0.2ps
    d = Tmn*vth;  % Calculate the mean free path
    display("Mean path is "+ d);
```

```
>> MD_Q1
    "vth = 187019.1244"

    "Mean path is 3.7404e-08"
```

3. Write a program that will model the random motion of the electrons. The program should do the following:

- Assign each particle a random location in the $x-y$ plane within the region defined by the extent of the Silicon. For simplicity you may use a small number of particles (1000-10000 works well) but you can start much smaller initially if you like.
- Assign each particle a random location in the $x-y$ plane within the region defined by the extent of the Silicon. For simplicity you may use a small number of particles (1000-10000 works well) but you can start much smaller initially if you like.
- Assign each particle with the fixed velocity given by v_{th} but give each one a random direction. To do this pick a random direction ϕ and generate v_x and v_y. Later you will generate these from the Maxwell-Boltzmann distributions.

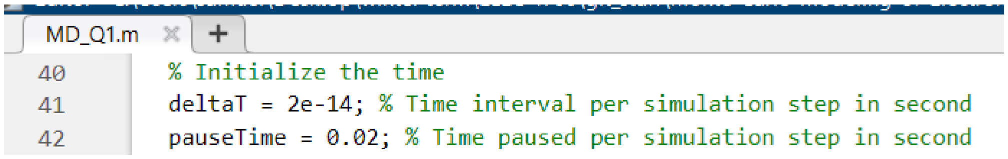
For this part, a function named "AddElectrons()" is programmed to add electrons in a given region randomly.



```
% This function add a bunch of electrons in a given region randomly for Q1
% @param numE = number of electrons
%         region = region for the electrons
%         vth = magnitude of the velocity
function AddElectrons(numE, region, vth)
global C  % Constants
global x y % arrays for current electron positions
global xp yp % arrays for previous electron positions
global vx vy % arrays for current electron  velocities

% Create the arrays for electrons locations
x = rand(1, numE) * region.x;
xp = x;
y = rand(1, numE) * region.y;
yp = y;

% Create a helper array for electrons directions
phi = rand(1, numE) * 2*pi;

% Create the arrays for current electron velocities
vx = vth .* cos(phi);
vy = vth .* sin(phi);

end
```

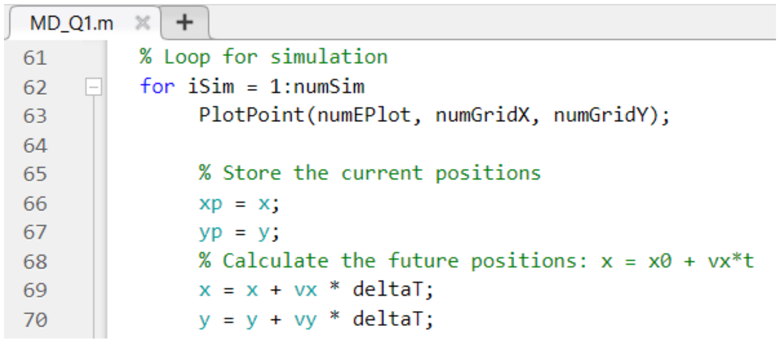
- At a fixed time interval of $\triangle t$, update the particle location using Newton's laws of motion. You will need to pick a time step size that takes into account the velocity of your particles and the size of the region. Typically the spacial step should be smaller than $1/100$ of the region size. Simulate for nominally 1000 timesteps. This should allow each particle to bounce around quite a bit inside the region.

For this part, a variable was created to store the fixed time interval Δt.



```
% Initialize the time
deltaT = 2e-14; % Time interval per simulation step in second
pauseTime = 0.02; % Time paused per simulation step in second
```

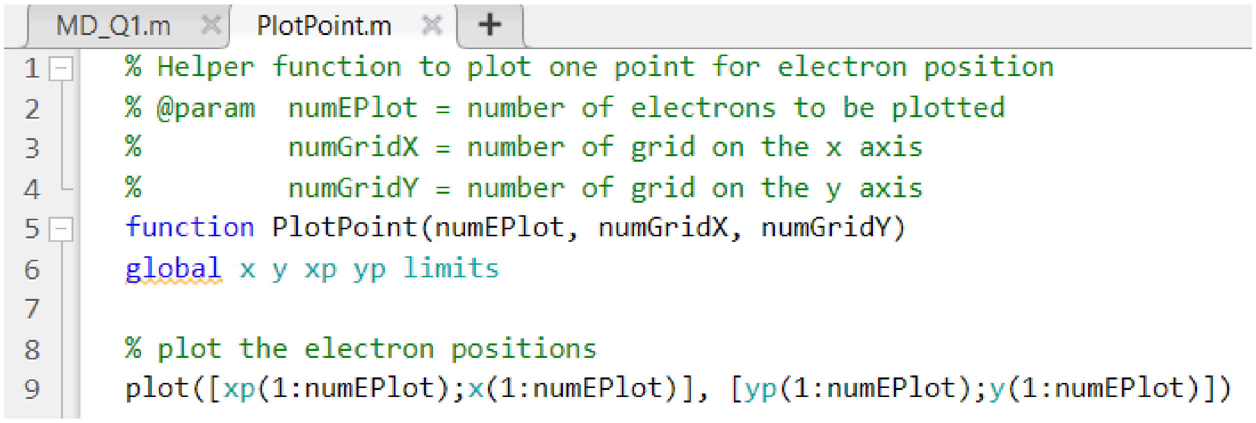
The simulation loop was programmed to update the particle location using Newton's laws of motion.



```
% Loop for simulation
for iSim = 1:numSim
    PlotPoint(numEPlot, numGridX, numGridY);

    % Store the current positions
    xp = x;
    yp = y;
    % Calculate the future positions: x = x0 + vx*t
    x = x + vx * deltaT;
    y = y + vy * deltaT;
```

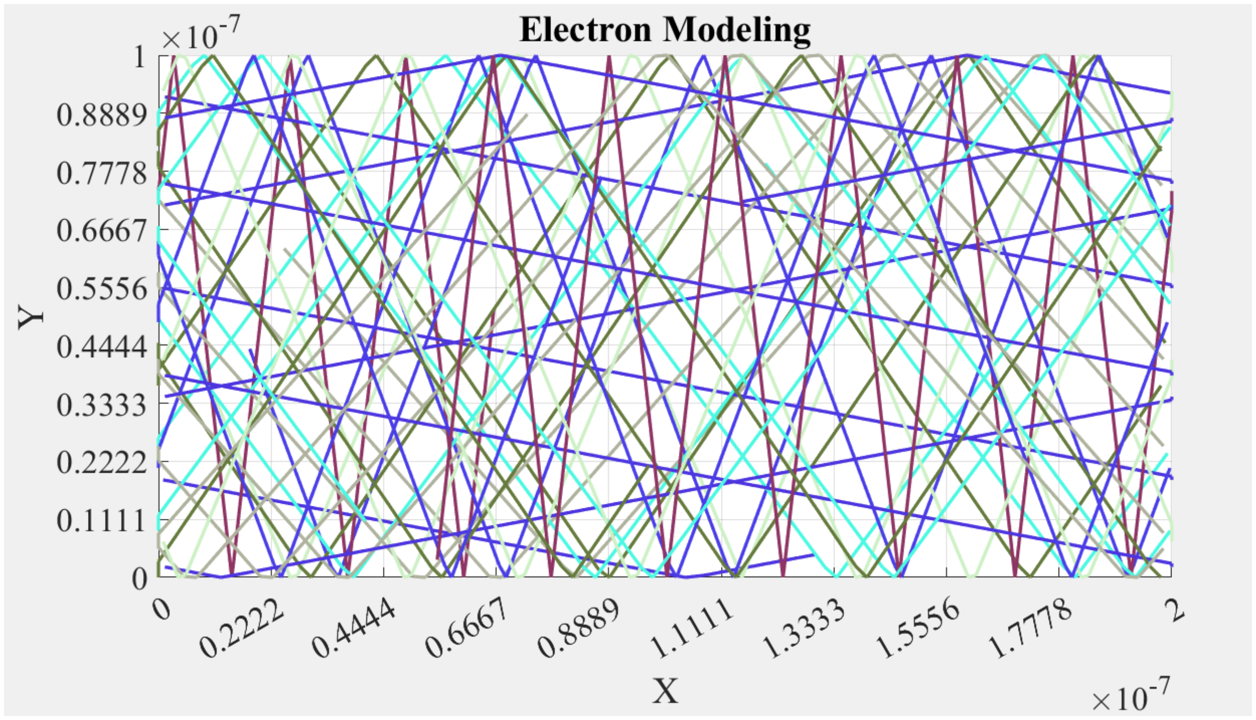
- For a few of the particles trace out their trajectories using the *'plot'* command in Matlab. To plot the trajectories you should keep the previous x and y positions. *Hint: use breakpoints to step through the trajectory and see what is "going on"; use the 'pause' command in Matlab to have the plot update in a loop.*

To plot the trajectories of the particles, a function named "PlotPoint()" is programmed.



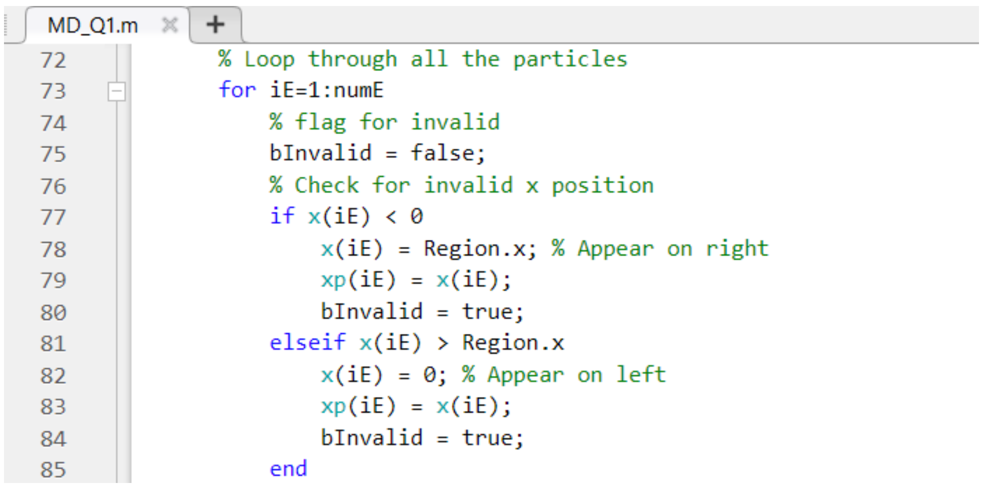
```
% Helper function to plot one point for electron position
% @param  numEPlot = number of electrons to be plotted
%         numGridX = number of grid on the x axis
%         numGridY = number of grid on the y axis
function PlotPoint(numEPlot, numGridX, numGridY)
global x y xp yp limits

% plot the electron positions
plot([xp(1:numEPlot);x(1:numEPlot)], [yp(1:numEPlot);y(1:numEPlot)])
```

- Show a 2-D plot of all (or a subset) of the particles that updates with each time step. *Hint: use the 'pause' command in Matlab to have the plot update in a loop.*

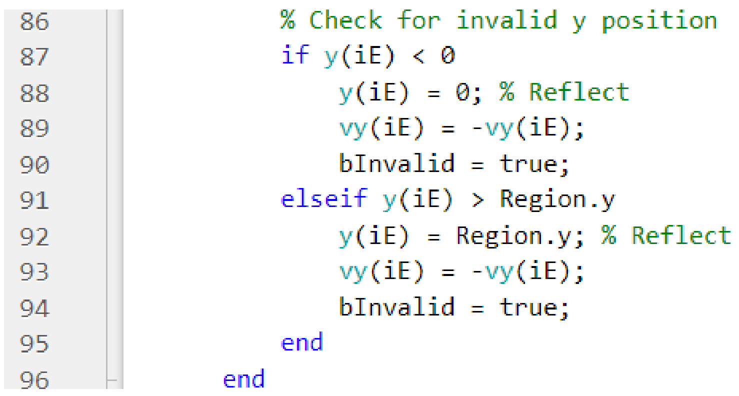


- For the y direction use a boundary condition where the particle reflects at the same angle (specular) and retains its velocity.
- For the x direction use a periodic boundary condition where the particle jumps to the opposite edge. i.e. if it reaches the right side it appears at the left with the same velocity.

The logic for boundary condition was implemented with if statements inside the simulation loop.



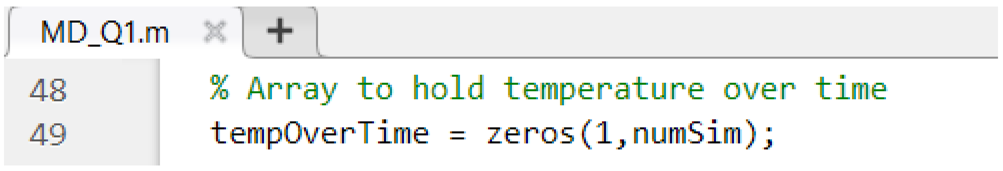
```
        % Loop through all the particles
        for iE=1:numE
            % flag for invalid
            bInvalid = false;
            % Check for invalid x position
            if x(iE) < 0
                x(iE) = Region.x; % Appear on right
                xp(iE) = x(iE);
                bInvalid = true;
            elseif x(iE) > Region.x
                x(iE) = 0; % Appear on left
                xp(iE) = x(iE);
                bInvalid = true;
            end
```



```
            % Check for invalid y position
            if y(iE) < 0
                y(iE) = 0; % Reflect
                vy(iE) = -vy(iE);
                bInvalid = true;
            elseif y(iE) > Region.y
                y(iE) = Region.y; % Reflect
                vy(iE) = -vy(iE);
                bInvalid = true;
            end
        end
```

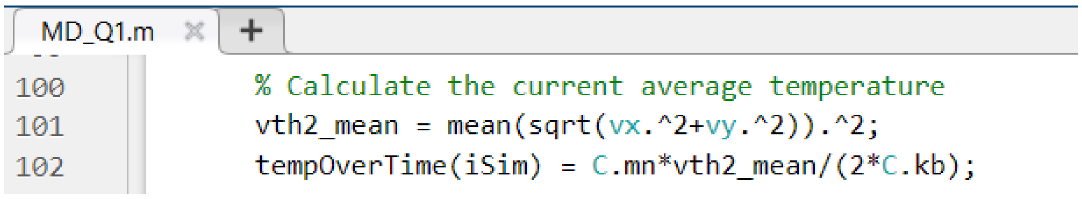
- Calculate and display the semiconductor temperature on the plot at a fixed time interval and verify that it stays constant.

An array was created to hold the temperature to calculate the temperature over time.



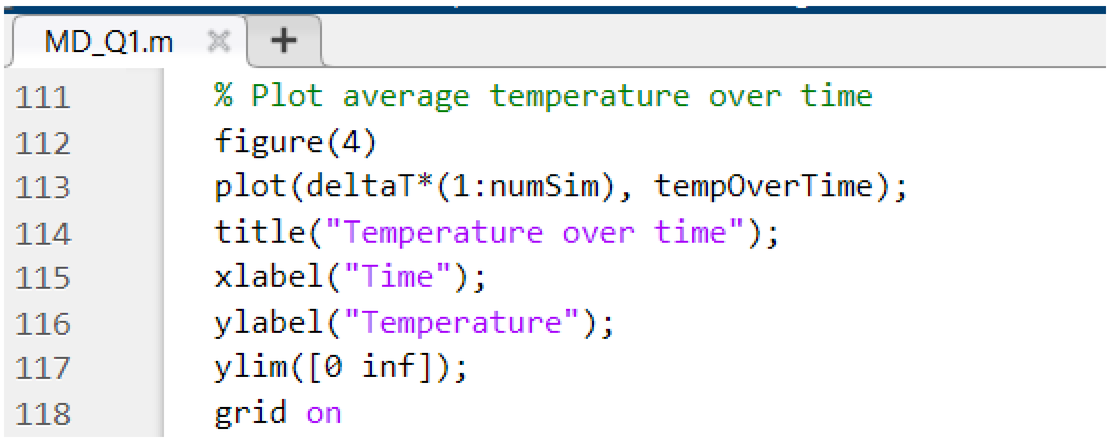
```
% Array to hold temperature over time
tempOverTime = zeros(1,numSim);
```

Inside the simulation loop, the temperature was calculated and stored in the array.



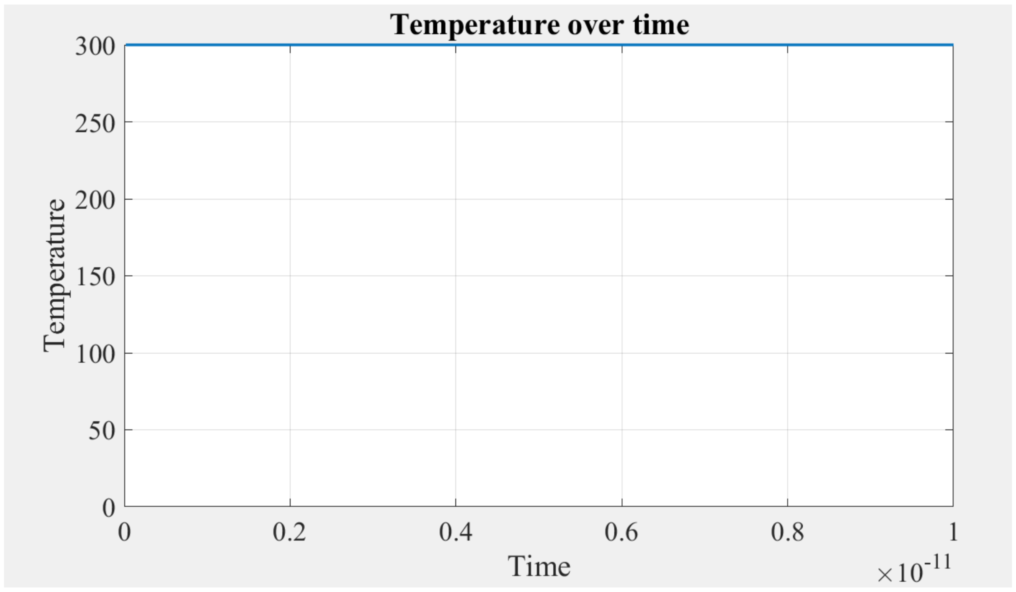
```
    % Calculate the current average temperature
    vth2_mean = mean(sqrt(vx.^2+vy.^2)).^2;
    tempOverTime(iSim) = C.mn*vth2_mean/(2*C.kb);
```

After the simulation loop, the temperature over time is plotted using the following code.



```
% Plot average temperature over time
figure(4)
plot(deltaT*(1:numSim), tempOverTime);
title("Temperature over time");
xlabel("Time");
ylabel("Temperature");
ylim([0 inf]);
grid on
```

The temperature plot shows that the temperature remains constant at 300K throughout the simulation as expected.



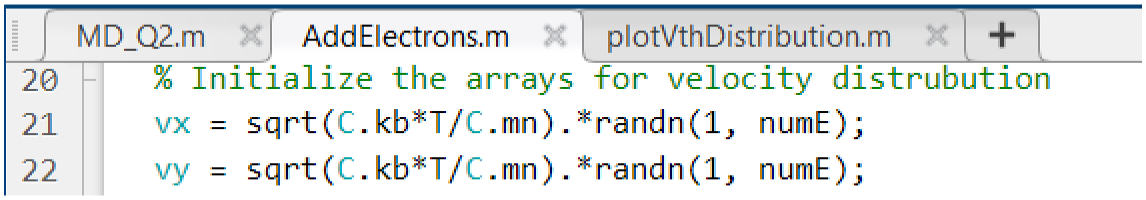
2. Collisions with Mean Free Path (MFP)

In this part, the simulation program from part 1 was updated to calculate the actual Mean Free Path and mean time between collisions of the electrons in the system. The codes for part 2 are in the folder named “Q2”, which contains the following files:

- AddElectrons.m – Function that adds electrons randomly in the region
- globalVars.m – File that groups the constants used in the simulation
- MD_Q2.m – Entry point script for the simulation for part 2
- PlotPoint.m – Function that plots the trajectories of the electrons
- plotTempDistribution.m – Function that plots the initial temperature distribution
- tempDisplay.m – Function that displays the temperature map

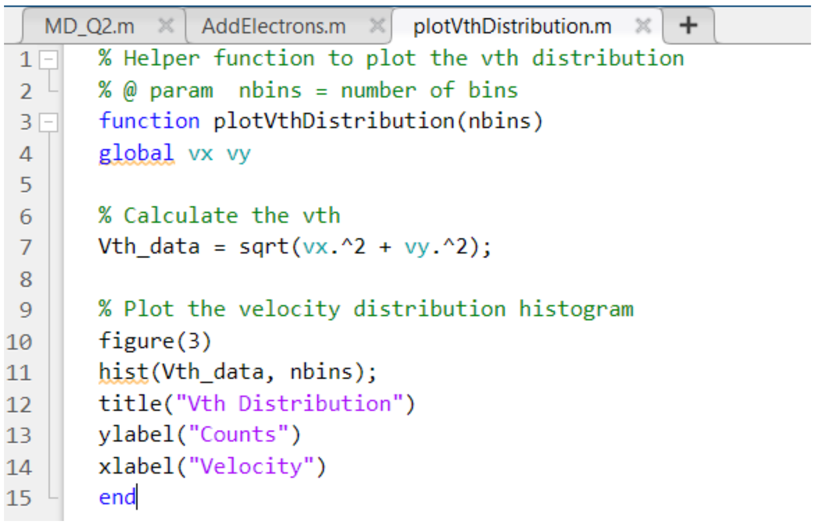
1. Assign a random velocity to each of the particles at the start. To do this you can use a Maxwell-Boltzmann distribution for each velocity component. Ensure that the average of all the speeds will be v_{th}. Plot the distribution in a histogram (Matlab *'hist'* function).

The AddElectrons() function was updated to assign the random velocity to the particles with Maxwell-Boltzmann distribution for each velocity component.



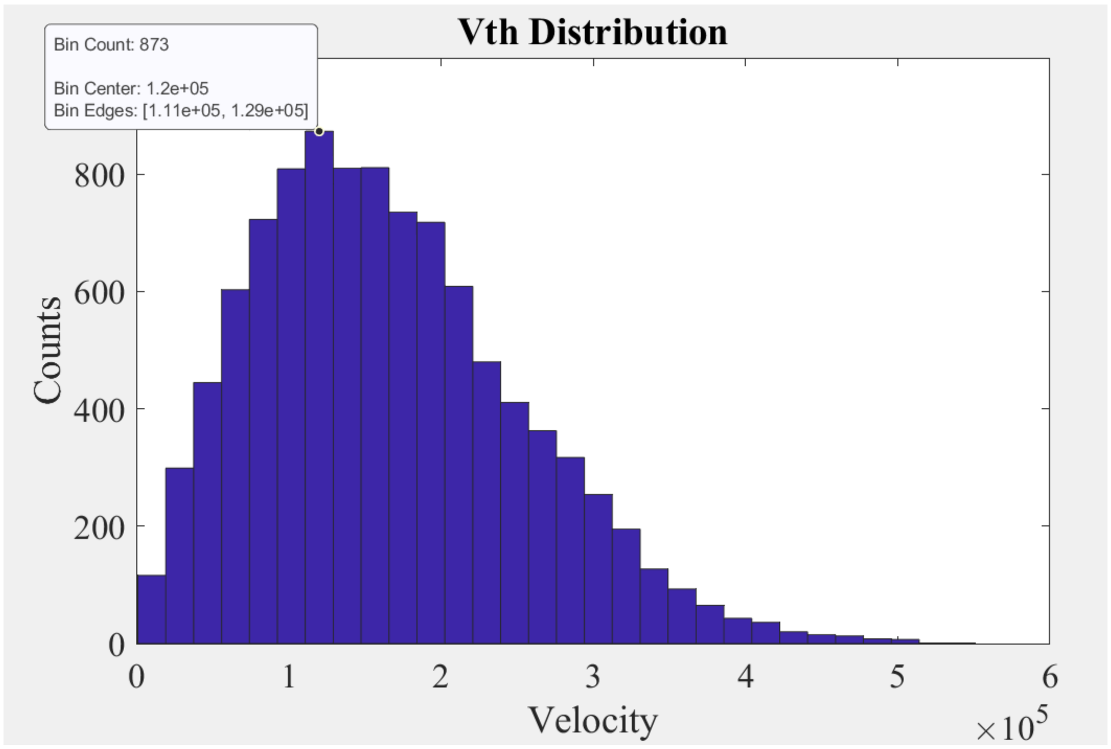
```
% Initialize the arrays for velocity distrubution
vx = sqrt(C.kb*T/C.mn).*randn(1, numE);
vy = sqrt(C.kb*T/C.mn).*randn(1, numE);
```

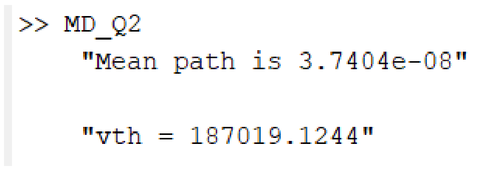
To plot the v_{th} distribution of the system, a function named "plotVthDistribution()" was programmed to plot the distribution in a histogram.



```
% Helper function to plot the vth distribution
% @ param  nbins = number of bins
function plotVthDistribution(nbins)
global vx vy

% Calculate the vth
Vth_data = sqrt(vx.^2 + vy.^2);

% Plot the velocity distribution histogram
figure(3)
hist(Vth_data, nbins);
title("Vth Distribution")
ylabel("Counts")
xlabel("Velocity")
end
```

The histogram shows that the bin center was roughly at $v_{th} = 1.2 \times 10^5\ m/s$. This is similar to the expected value, which is $v_{th} = 187019.1244\ m/s$.

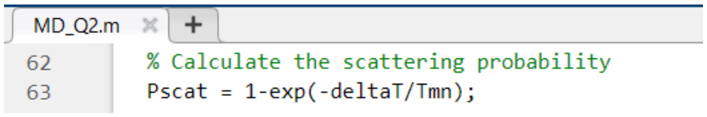




```
>> MD_Q2
    "Mean path is 3.7404e-08"

    "vth = 187019.1244"
```

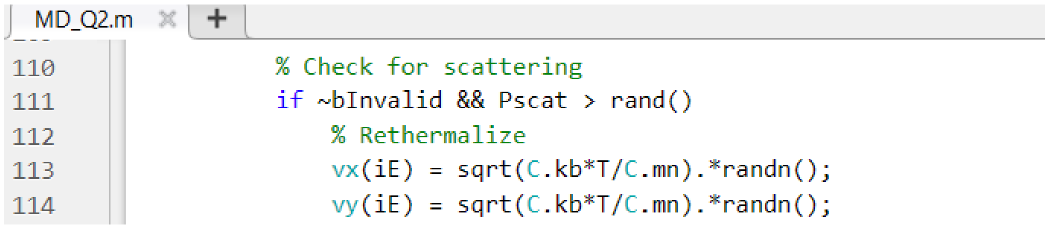
2. Model the scattering of the electrons using an exponential scattering probability: $P_{scat} = 1 - e^{-\frac{dt}{\tau_{mn}}}$, where dt is the time since the last timestep (and P_{scat} calculation), and τ_{mn} is the mean time between collisions. Use the τ_{mn} given above. At every time step (for every electron) use something like this: if $P_{scat} > rand()$ then the particle scatters. When the electron scatters re-thermalize its velocities and assign new velocities V_x, V_y from Maxwell-Boltzmann distributions. Refer to Figure 3 for a sample plot.

In the MD_Q2.m file, the P_{scat} was calculated before the main program loop.



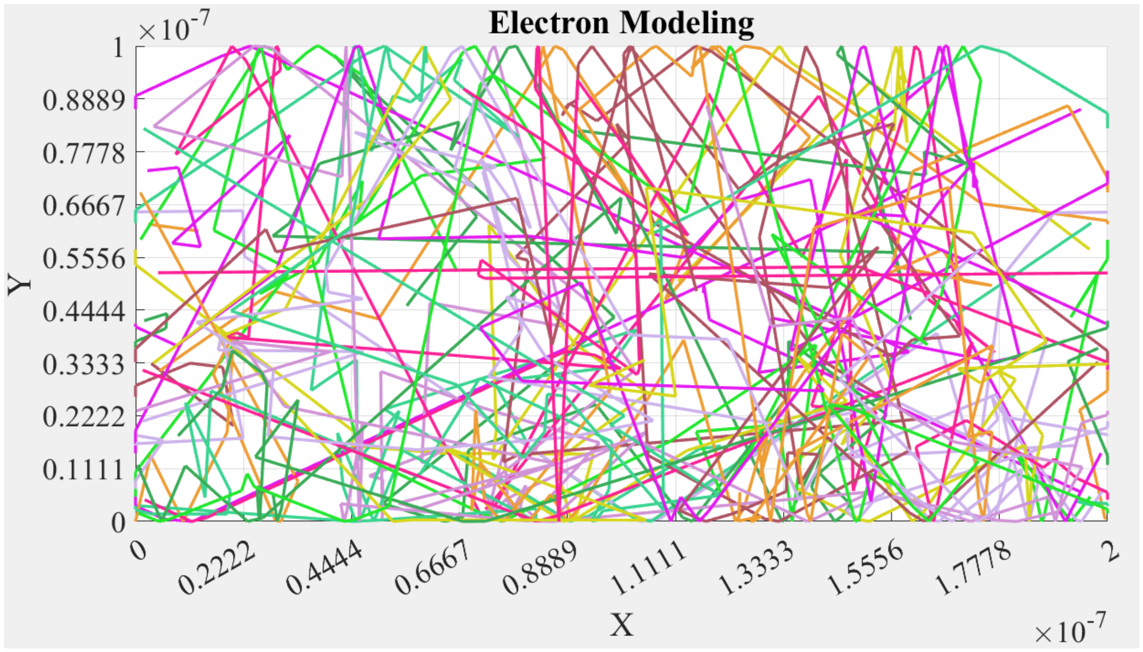
```
% Calculate the scattering probability
Pscat = 1-exp(-deltaT/Tmn);
```

In the main program loop, the condition of $P_{scat} > rand()$ was checked for scattering. If the condition is met, the electron will be re-thermalized with new random velocity values.



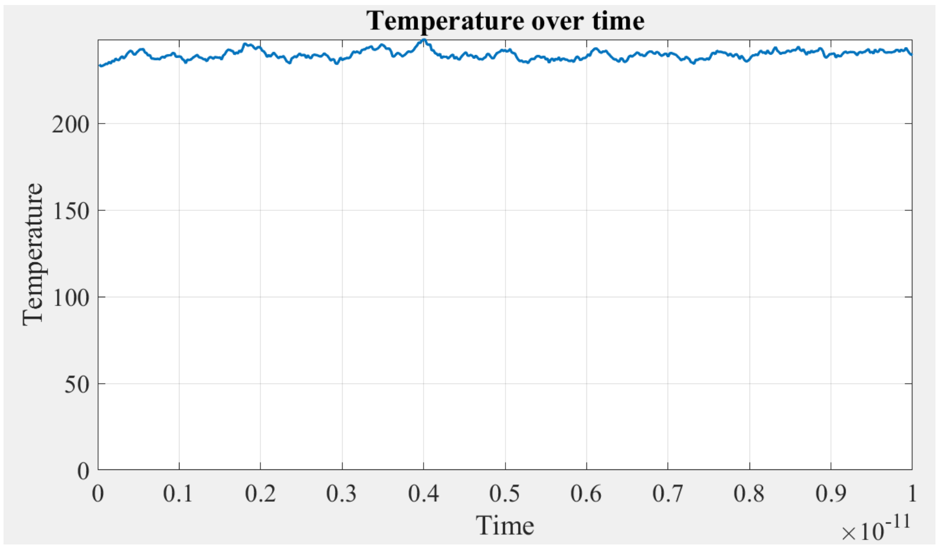
```
            % Check for scattering
            if ~bInvalid && Pscat > rand()
                % Rethermalize
                vx(iE) = sqrt(C.kb*T/C.mn).*randn();
                vy(iE) = sqrt(C.kb*T/C.mn).*randn();
```

The sample scattering plot is shown below.



3. What happens to the average temperature over time?

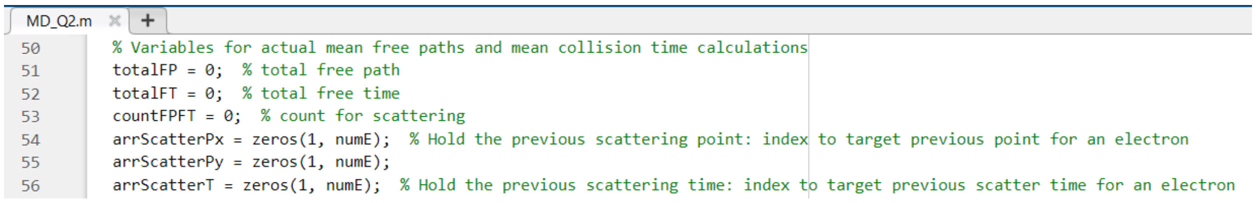
The following plot shows the average temperature over time, which remains roughly constant over time as expected.



In theory, the temperature should remain at roughly 300K over time. However, due to some unknown issue, the temperature from the simulation remains at roughly 250K instead of 300K.

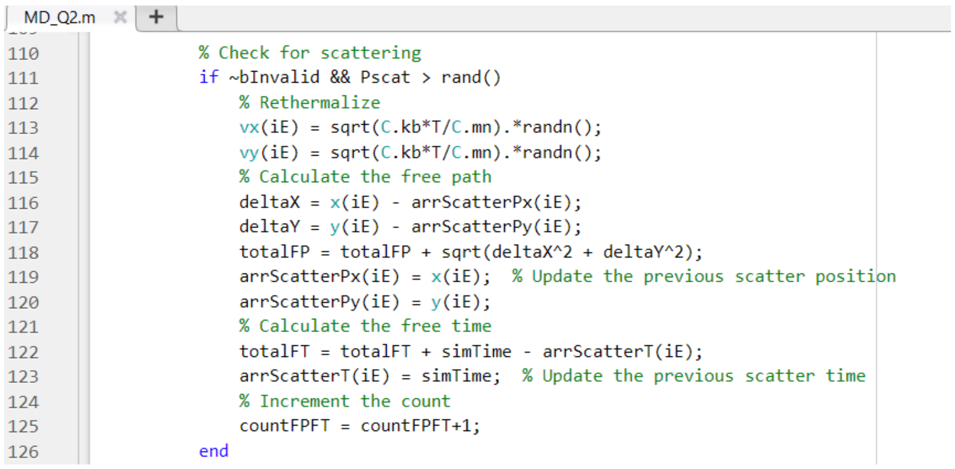
4. Measure the actual Mean Free Path and mean time between collisions to verify your model.

To measure the actual Mean Free Path, the following variables are declared to hold the data for calculation.



```matlab
% Variables for actual mean free paths and mean collision time calculations
totalFP = 0;  % total free path
totalFT = 0;  % total free time
countFPFT = 0;  % count for scattering
arrScatterPx = zeros(1, numE);  % Hold the previous scattering point: index to target previous point for an electron
arrScatterPy = zeros(1, numE);
arrScatterT = zeros(1, numE);  % Hold the previous scattering time: index to target previous scatter time for an electron
```

When the electron scattered, the calculations for Mean Free Path and mean time between collisions are performed and the variables are updated to store the data.



```matlab
        % Check for scattering
        if ~bInvalid && Pscat > rand()
            % Rethermalize
            vx(iE) = sqrt(C.kb*T/C.mn).*randn();
            vy(iE) = sqrt(C.kb*T/C.mn).*randn();
            % Calculate the free path
            deltaX = x(iE) - arrScatterPx(iE);
            deltaY = y(iE) - arrScatterPy(iE);
            totalFP = totalFP + sqrt(deltaX^2 + deltaY^2);
            arrScatterPx(iE) = x(iE);  % Update the previous scatter position
            arrScatterPy(iE) = y(iE);
            % Calculate the free time
            totalFT = totalFT + simTime - arrScatterT(iE);
            arrScatterT(iE) = simTime;  % Update the previous scatter time
            % Increment the count
            countFPFT = countFPFT+1;
        end
```

The actual Mean Free Path and mean time between collisions of the system are very similar to the expected results from calculation.

```
>> MD_Q2
    "Expected Mean free path is 3.7404e-08"

    "vth = 187019.1244"

    "Actual mean free path: 3.875e-08"

    "Mean time between collision: 2.1221e-13"
```

	Actual Result	Expected Result
Mean Free Path	$3.875 \times 10^{-8}\ m$	$3.7404 \times 10^{-8}\ m$
Mean time between collision	$0.21221\ ps$	$0.2\ ps$

3. Enhancement

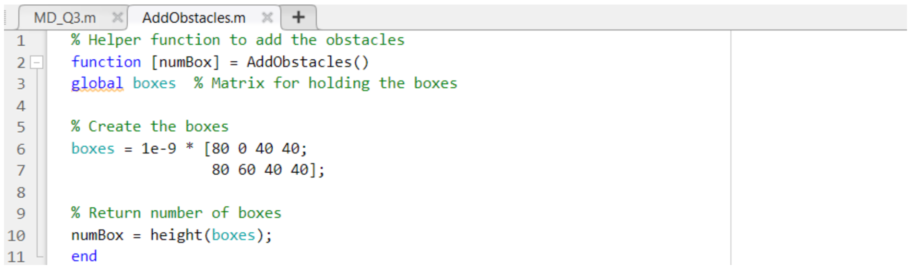
1. Basic Features and Analysis:

For this part, two box obstacles are added to the simulation environment. The codes for part 3 are in the folder named “Q3”, which contains the following files:

- AddElectrons.m – Function that adds electrons randomly in the region
- AddObstacles.m – Function that adds rectangle obstacles
- globalVars.m – File that groups the constants used in the simulation
- MD_Q3.m – Entry point script for the simulation for part 3
- PlotPoint.m – Function that plots the trajectories of the electrons
- plotTempDistribution.m – Function that plots the initial temperature distribution
- tempDisplay.m – Function that displays the temperature map

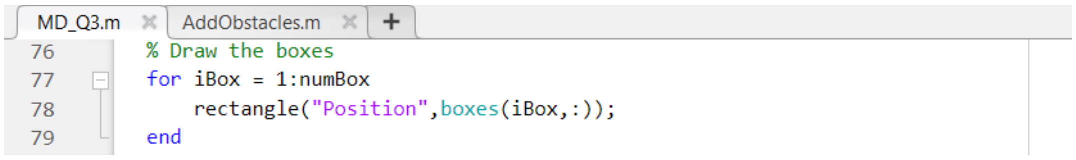
(a) Add in the inner rectangle "bottle neck" boundaries as in Figure 4. There are a number of ways to do this. I did it by defining "boxes" that reflect particles and then adding a number of "boxes" to the region. You will need to be careful not to put electrons in the "boxes" at the start. You should also check that no electrons are leaking through the "boxes".

A function named “AddObstacles()” is implemented to add two rectangular obstacles.



```
% Helper function to add the obstacles
function [numBox] = AddObstacles()
global boxes  % Matrix for holding the boxes

% Create the boxes
boxes = 1e-9 * [80 0 40 40;
                80 60 40 40];

% Return number of boxes
numBox = height(boxes);
end
```

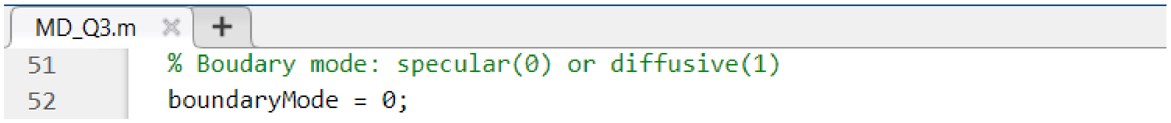
Before the simulation loop, the rectangle boxes are drawn.


MD_Q3.m | AddObstacles.m

```
% Draw the boxes
for iBox = 1:numBox
    rectangle("Position",boxes(iBox,:));
end
```

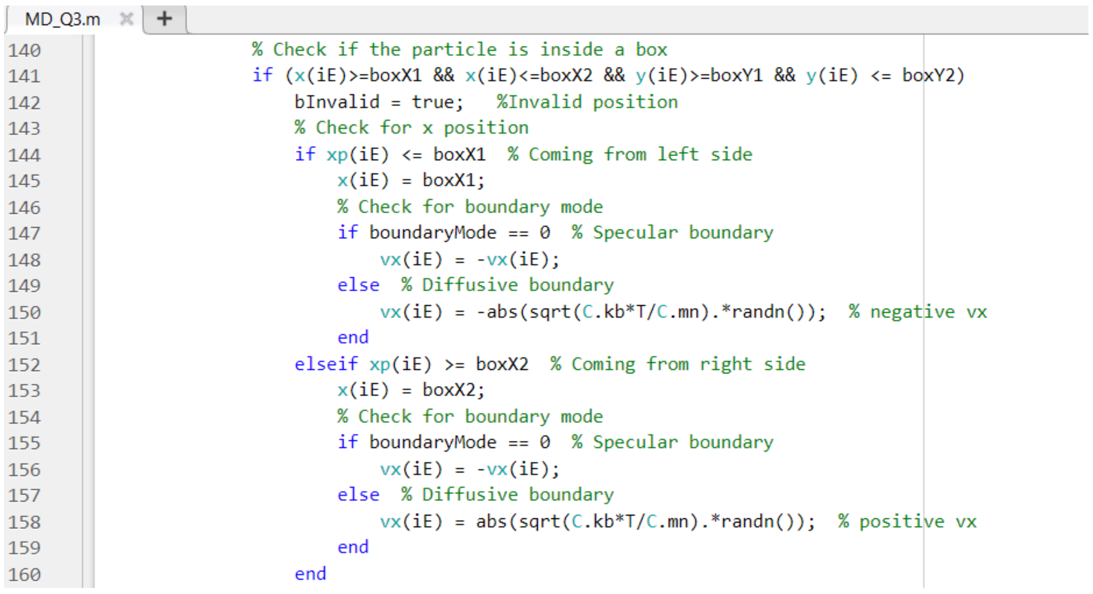
(b) Make all boundaries capable to be either specular or diffusive (ie re-thermalized).

A variable named "boundaryMode" is declared to hold the mode for the boundary.

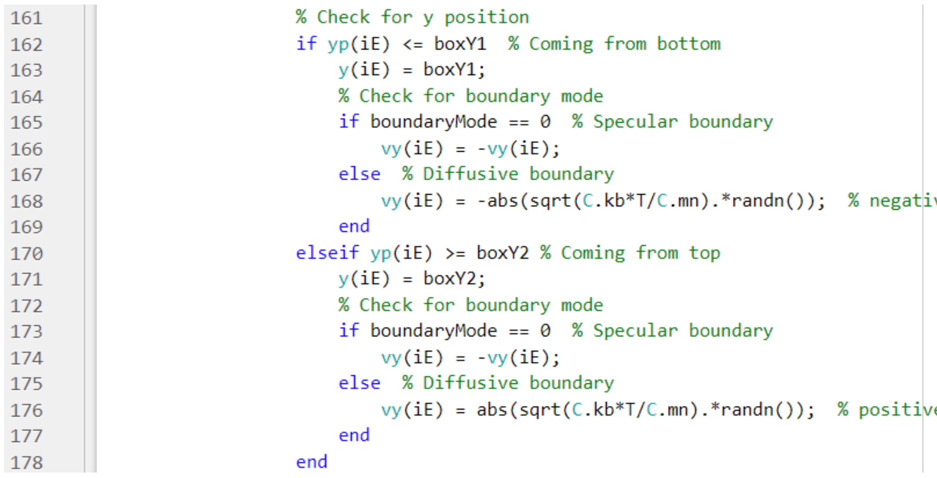

MD_Q3.m

```
% Boudary mode: specular(0) or diffusive(1)
boundaryMode = 0;
```

In the simulation loop, this variable is checked when an electron hit a boundary.

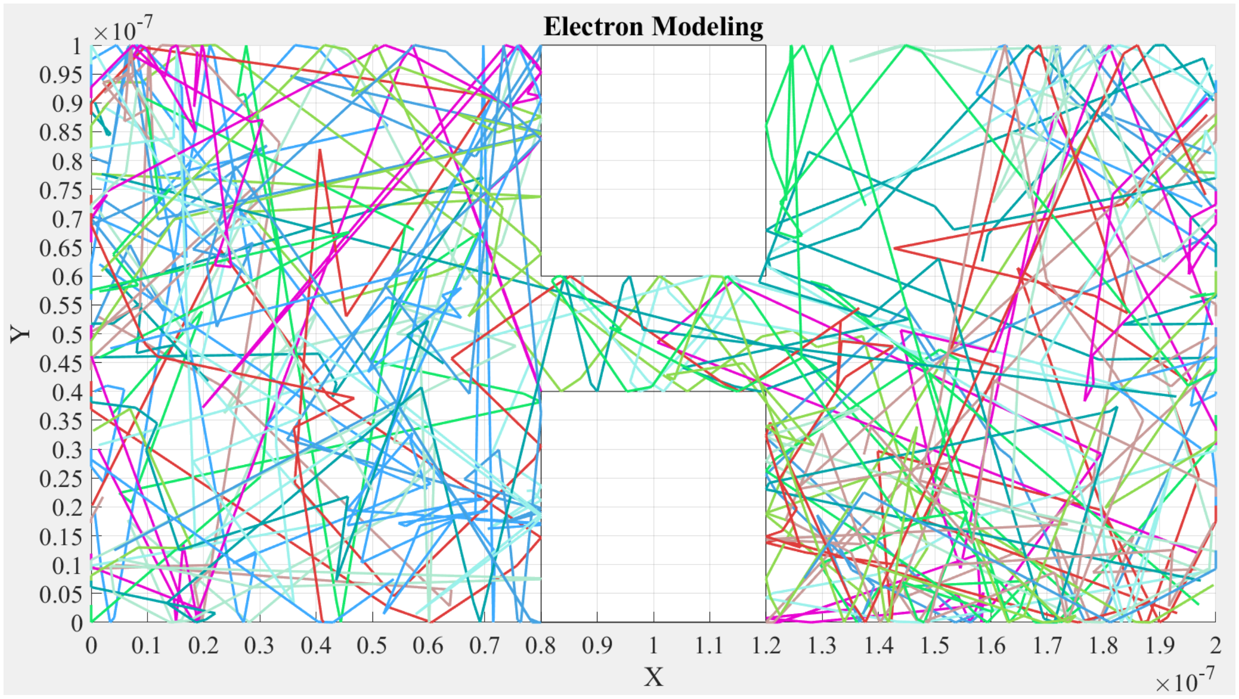

MD_Q3.m

```
            % Check if the particle is inside a box
            if (x(iE)>=boxX1 && x(iE)<=boxX2 && y(iE)>=boxY1 && y(iE) <= boxY2)
                bInvalid = true;   %Invalid position
                % Check for x position
                if xp(iE) <= boxX1  % Coming from left side
                    x(iE) = boxX1;
                    % Check for boundary mode
                    if boundaryMode == 0  % Specular boundary
                        vx(iE) = -vx(iE);
                    else  % Diffusive boundary
                        vx(iE) = -abs(sqrt(C.kb*T/C.mn).*randn());  % negative vx
                    end
                elseif xp(iE) >= boxX2  % Coming from right side
                    x(iE) = boxX2;
                    % Check for boundary mode
                    if boundaryMode == 0  % Specular boundary
                        vx(iE) = -vx(iE);
                    else  % Diffusive boundary
                        vx(iE) = abs(sqrt(C.kb*T/C.mn).*randn());  % positive vx
                    end
                end
```

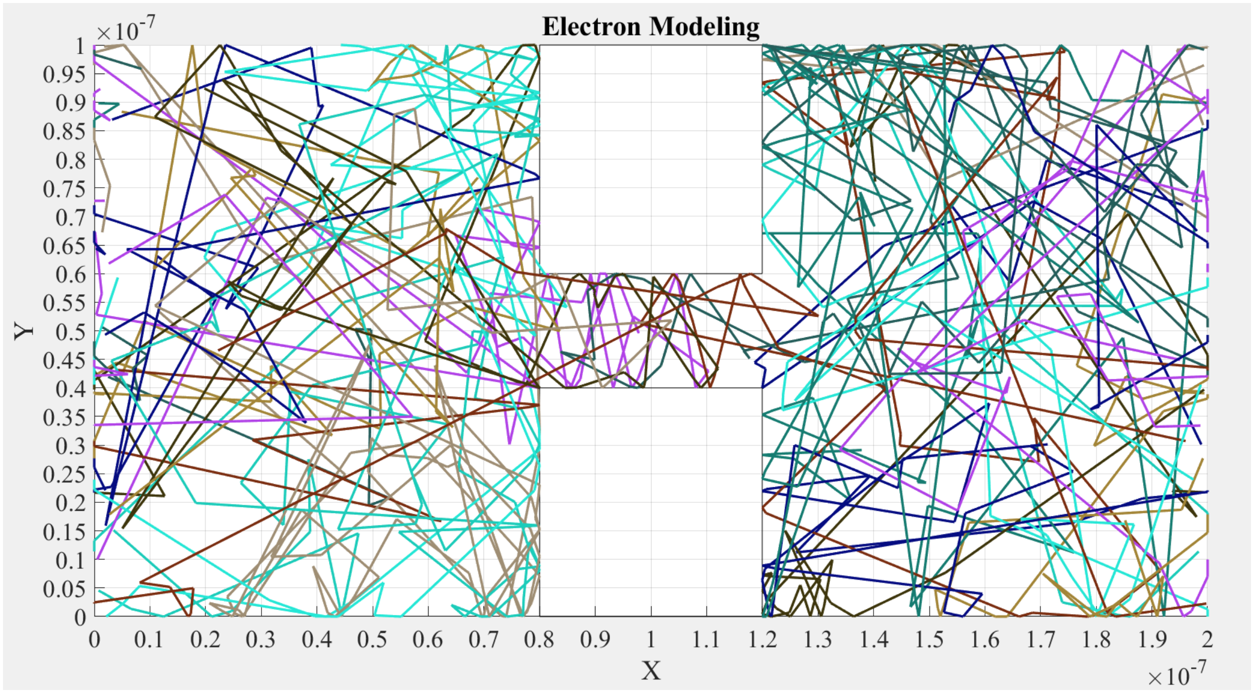


```
                % Check for y position
                if yp(iE) <= boxY1  % Coming from bottom
                    y(iE) = boxY1;
                    % Check for boundary mode
                    if boundaryMode == 0  % Specular boundary
                        vy(iE) = -vy(iE);
                    else  % Diffusive boundary
                        vy(iE) = -abs(sqrt(C.kb*T/C.mn).*randn());  % negative vy
                    end
                elseif yp(iE) >= boxY2 % Coming from top
                    y(iE) = boxY2;
                    % Check for boundary mode
                    if boundaryMode == 0  % Specular boundary
                        vy(iE) = -vy(iE);
                    else  % Diffusive boundary
                        vy(iE) = abs(sqrt(C.kb*T/C.mn).*randn());  % positive vy
                    end
                end
```

The following figure shows the simulation plot when boundaries are specular.

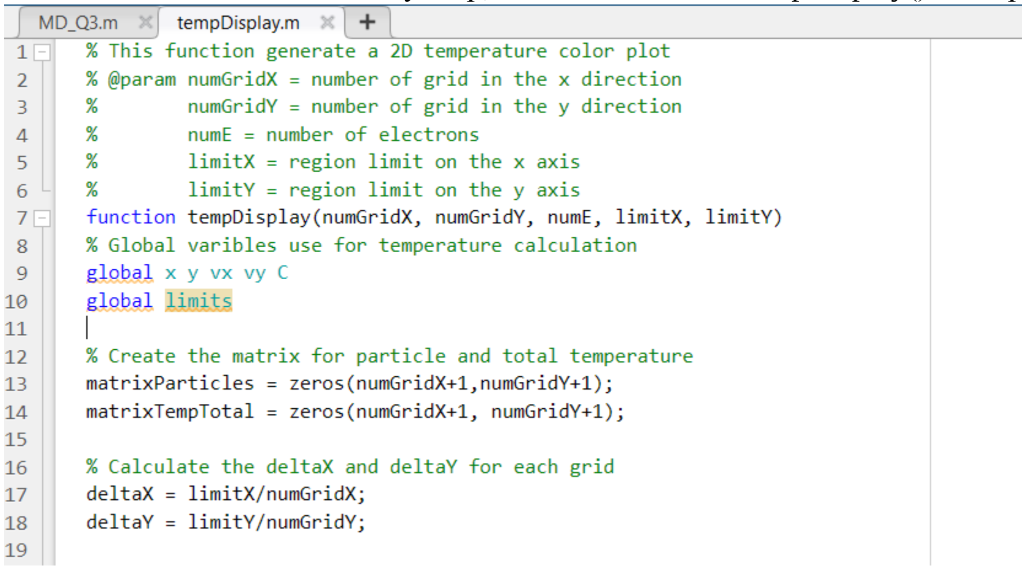


The following figure shows the simulation plot when boundaries are diffusive.

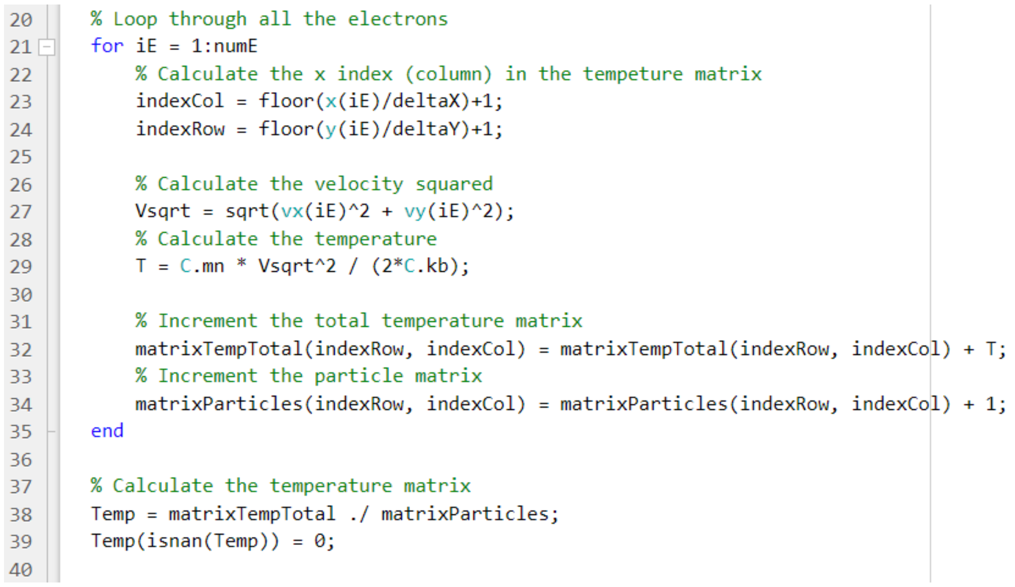


(c) Calculate an electron density map from the final electron positions.

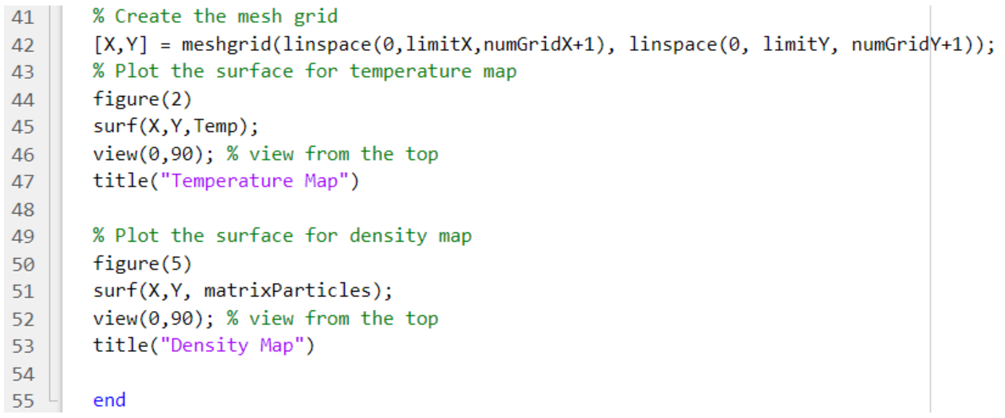
To calculate the electron density map, a function named “tempDisplay()” was programmed.



```matlab
% This function generate a 2D temperature color plot
% @param numGridX = number of grid in the x direction
%        numGridY = number of grid in the y direction
%        numE = number of electrons
%        limitX = region limit on the x axis
%        limitY = region limit on the y axis
function tempDisplay(numGridX, numGridY, numE, limitX, limitY)
% Global varibles use for temperature calculation
global x y vx vy C
global limits

% Create the matrix for particle and total temperature
matrixParticles = zeros(numGridX+1,numGridY+1);
matrixTempTotal = zeros(numGridX+1, numGridY+1);

% Calculate the deltaX and deltaY for each grid
deltaX = limitX/numGridX;
deltaY = limitY/numGridY;

```

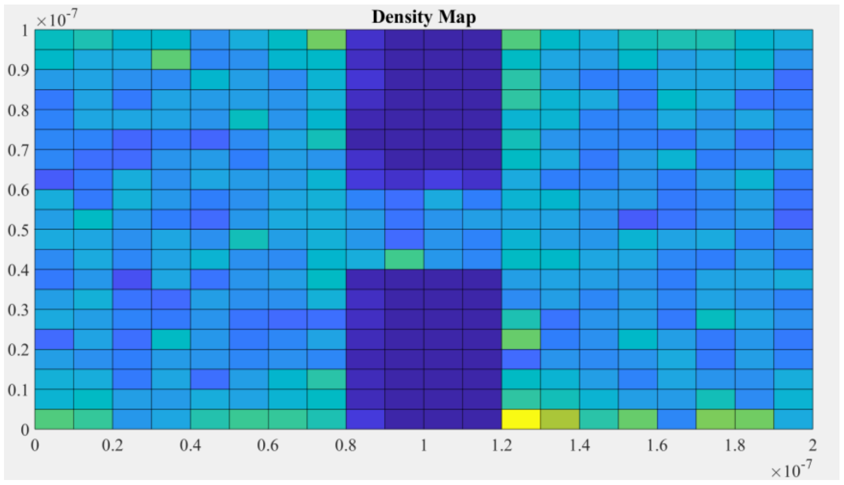


```matlab
% Loop through all the electrons
for iE = 1:numE
    % Calculate the x index (column) in the tempeture matrix
    indexCol = floor(x(iE)/deltaX)+1;
    indexRow = floor(y(iE)/deltaY)+1;

    % Calculate the velocity squared
    Vsqrt = sqrt(vx(iE)^2 + vy(iE)^2);
    % Calculate the temperature
    T = C.mn * Vsqrt^2 / (2*C.kb);

    % Increment the total temperature matrix
    matrixTempTotal(indexRow, indexCol) = matrixTempTotal(indexRow, indexCol) + T;
    % Increment the particle matrix
    matrixParticles(indexRow, indexCol) = matrixParticles(indexRow, indexCol) + 1;
end

% Calculate the temperature matrix
Temp = matrixTempTotal ./ matrixParticles;
Temp(isnan(Temp)) = 0;

```



```matlab
% Create the mesh grid
[X,Y] = meshgrid(linspace(0,limitX,numGridX+1), linspace(0, limitY, numGridY+1));
% Plot the surface for temperature map
figure(2)
surf(X,Y,Temp);
view(0,90); % view from the top
title("Temperature Map")

% Plot the surface for density map
figure(5)
surf(X,Y, matrixParticles);
view(0,90); % view from the top
title("Density Map")

end
```

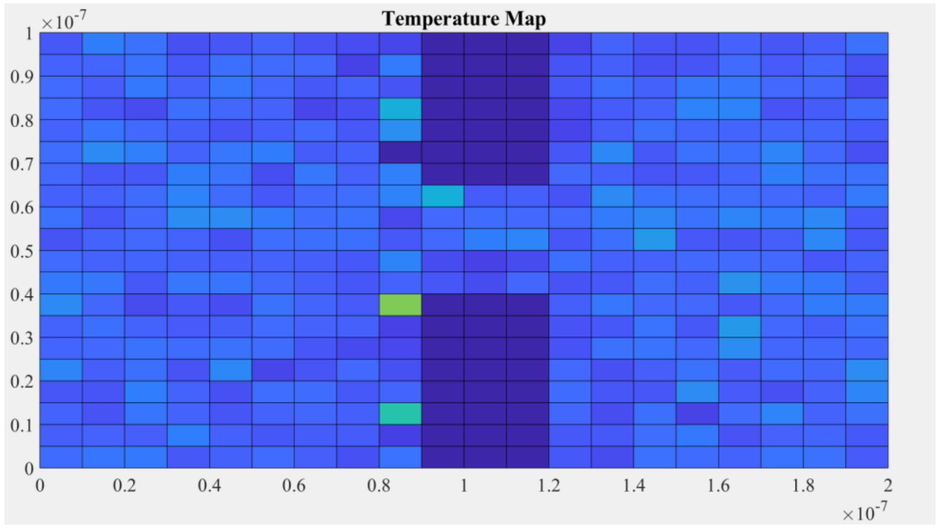
The electron density map was generated by grouping the electrons into a particle matrix and plot the density using the surf() function in MATLAB, which plotted a colored 3-d surface. The 2-D density map is then obtained by viewing on top from the surface.



It should be noted that the left-side edges of the two boxes region are not colored completely blue. This is because the grouping algorithm grouped all the electrons inside a square to the bottom left point of the square, and each square is colored based on the value at the bottom left point. Therefore, the electrons on the left-side edges of the boxes will be grouped to the left side. This similar principle also explained why the color of the bottom edge of the top box is not completely blue.

(d) Calculate a temperature map and display with colors.

The temperature map is calculated inside the "tempDisplay()" function. It was calculated by dividing the total temperature map by the density map. The following figure shows the temperature map.



2. Complete one of the following two tasks. Bonus marks for doing both.

3a. Curved Surfaces

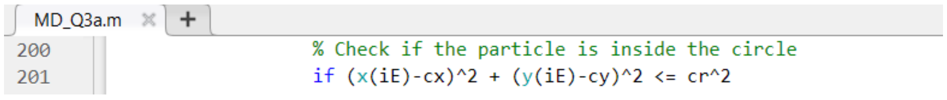
(a) **Curved Surfaces:** Add curvilinear geometry into the simulation. An example would be a circle placed inside the region that reflects electrons. To do this you will need to (for each electron trajectory):

The codes for this part are in the folder named "Q3a", which contains the following files:

- AddElectrons.m – Function that adds electrons randomly in the region
- AddObstacles.m – Function that adds rectangle and circle obstacles
- findCircReflect.m – Function that finds the reflected vector off the circle
- findLineCircIntersect – Function that finds the intersecting point on the circle
- globalVars.m – File that groups the constants used in the simulation
- MD_Q3a.m – Entry point script for the simulation for part 3a
- PlotPoint.m – Function that plots the trajectories of the electrons
- plotTempDistribution.m – Function that plots the initial temperature distribution
- tempDisplay.m – Function that displays the temperature map

- Determine if the electron has moved inside the circular inclusion

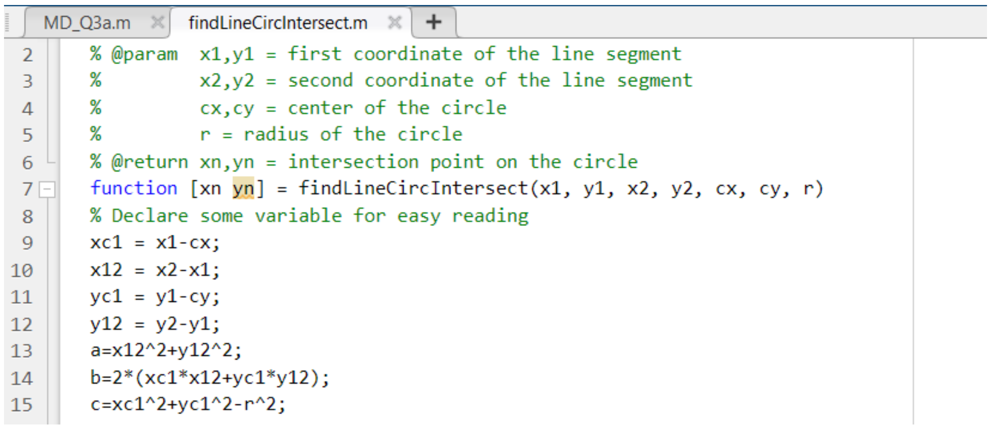
The condition for an electron has moved inside the circle is determined using the equation of the circle.



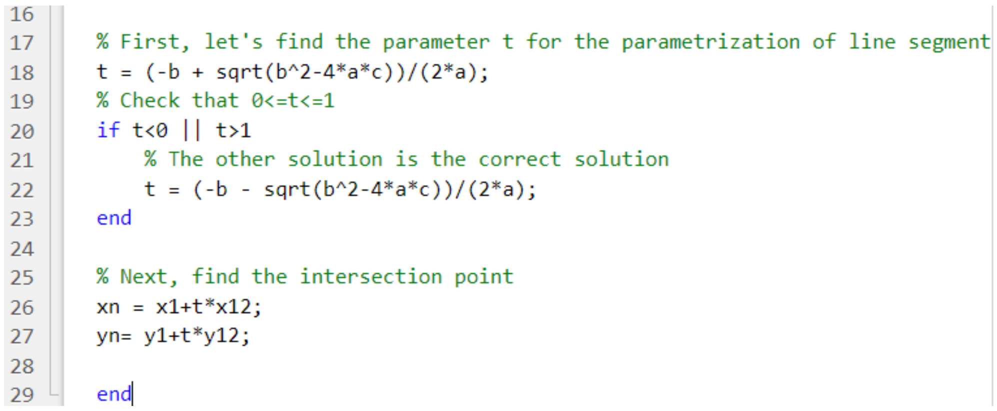
```
                % Check if the particle is inside the circle
                if (x(iE)-cx)^2 + (y(iE)-cy)^2 <= cr^2
```

- If it has calculate the intersection of the trajectory with the circle.

A function named "findLineCircIntersect()" was programmed to find the intersection of the trajectory of the circle.



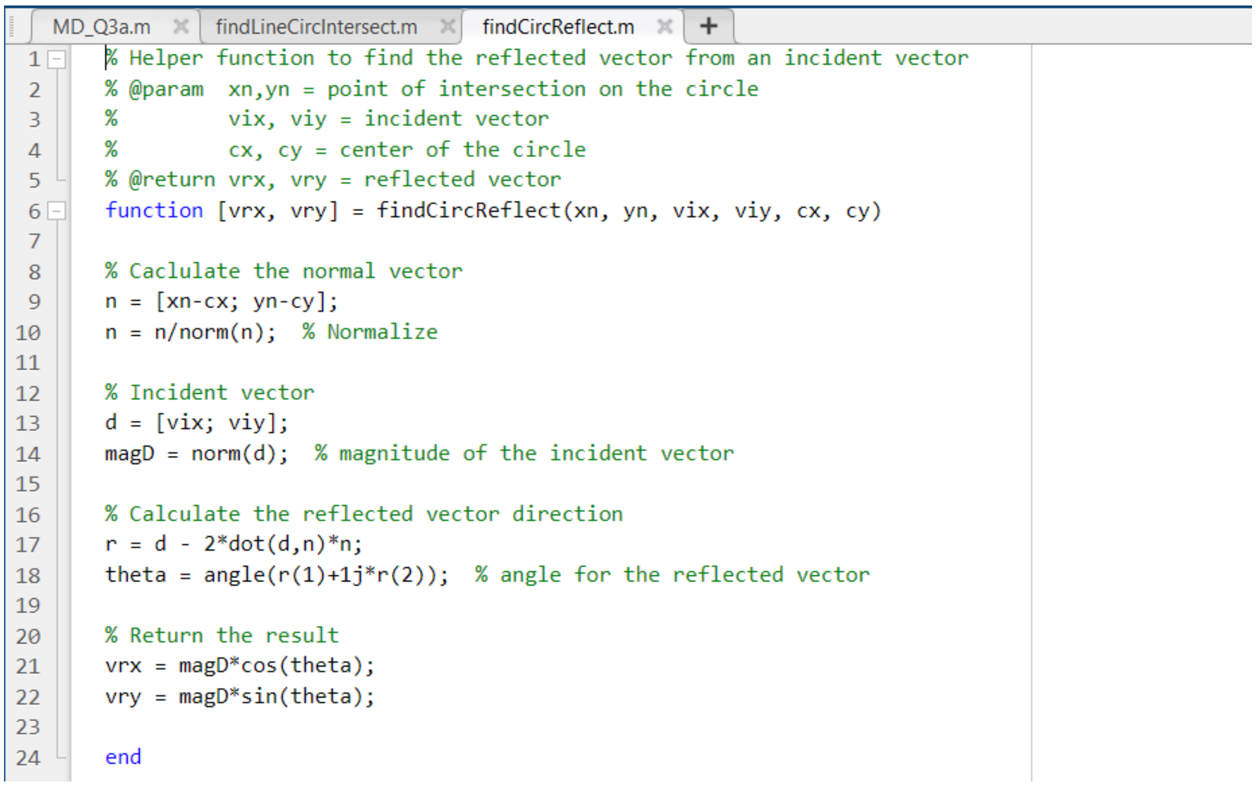
```
% @param  x1,y1 = first coordinate of the line segment
%         x2,y2 = second coordinate of the line segment
%         cx,cy = center of the circle
%         r = radius of the circle
% @return xn,yn = intersection point on the circle
function [xn yn] = findLineCircIntersect(x1, y1, x2, y2, cx, cy, r)
% Declare some variable for easy reading
xc1 = x1-cx;
x12 = x2-x1;
yc1 = y1-cy;
y12 = y2-y1;
a=x12^2+y12^2;
b=2*(xc1*x12+yc1*y12);
c=xc1^2+yc1^2-r^2;
```



```
% First, let's find the parameter t for the parametrization of line segment
t = (-b + sqrt(b^2-4*a*c))/(2*a);
% Check that 0<=t<=1
if t<0 || t>1
    % The other solution is the correct solution
    t = (-b - sqrt(b^2-4*a*c))/(2*a);
end

% Next, find the intersection point
xn = x1+t*x12;
yn= y1+t*y12;

end
```

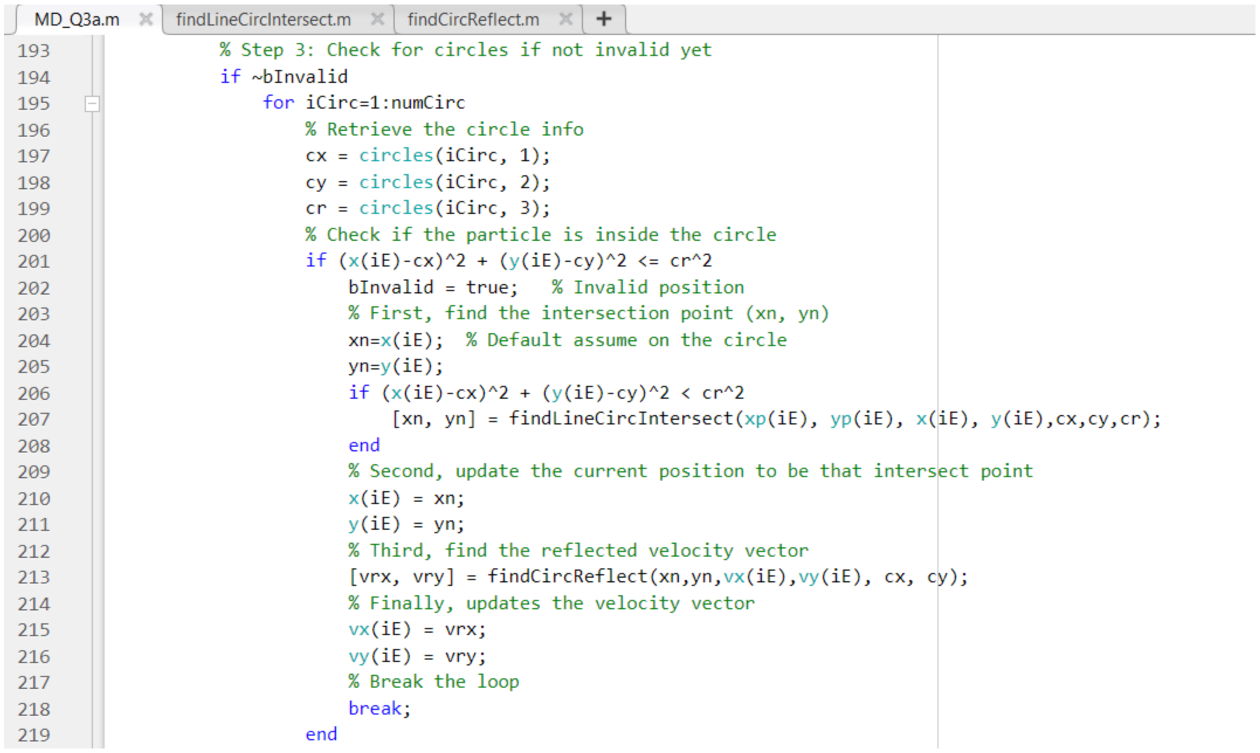
- Then calculate the angle of incidence ϕ with respect to the normal of the surface.
- Finally, calculate the reflection direction using ϕ, assuming specular reflection (angle in equals angle out) and use this to determine a new position and velocity.

A function named "findCircReflect()" was programmed to find the reflection vector.



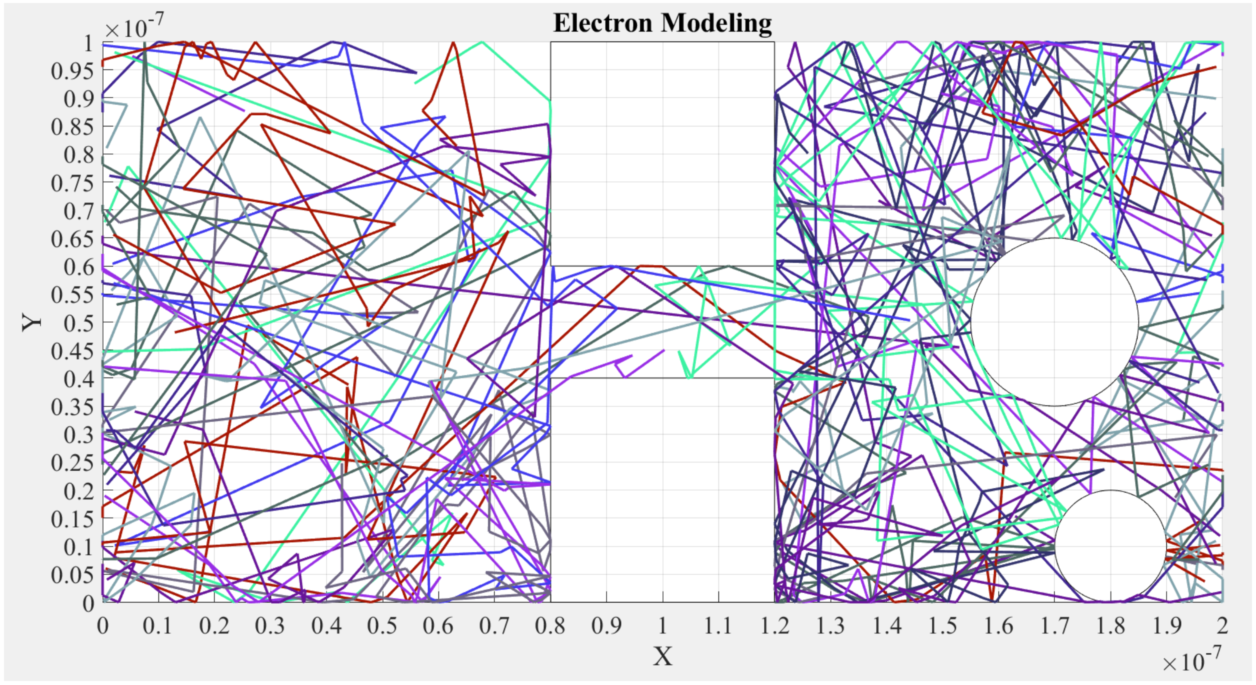
```
% Helper function to find the reflected vector from an incident vector
% @param  xn,yn = point of intersection on the circle
%         vix, viy = incident vector
%         cx, cy = center of the circle
% @return vrx, vry = reflected vector
function [vrx, vry] = findCircReflect(xn, yn, vix, viy, cx, cy)

% Caclulate the normal vector
n = [xn-cx; yn-cy];
n = n/norm(n);  % Normalize

% Incident vector
d = [vix; viy];
magD = norm(d);  % magnitude of the incident vector

% Calculate the reflected vector direction
r = d - 2*dot(d,n)*n;
theta = angle(r(1)+1j*r(2));  % angle for the reflected vector

% Return the result
vrx = magD*cos(theta);
vry = magD*sin(theta);

end
```

The code for implementing the reflection from the circle are shown in follow.



```
            % Step 3: Check for circles if not invalid yet
            if ~bInvalid
                for iCirc=1:numCirc
                    % Retrieve the circle info
                    cx = circles(iCirc, 1);
                    cy = circles(iCirc, 2);
                    cr = circles(iCirc, 3);
                    % Check if the particle is inside the circle
                    if (x(iE)-cx)^2 + (y(iE)-cy)^2 <= cr^2
                        bInvalid = true;   % Invalid position
                        % First, find the intersection point (xn, yn)
                        xn=x(iE);  % Default assume on the circle
                        yn=y(iE);
                        if (x(iE)-cx)^2 + (y(iE)-cy)^2 < cr^2
                            [xn, yn] = findLineCircIntersect(xp(iE), yp(iE), x(iE), y(iE),cx,cy,cr);
                        end
                        % Second, update the current position to be that intersect point
                        x(iE) = xn;
                        y(iE) = yn;
                        % Third, find the reflected velocity vector
                        [vrx, vry] = findCircReflect(xn,yn,vx(iE),vy(iE), cx, cy);
                        % Finally, updates the velocity vector
                        vx(iE) = vrx;
                        vy(iE) = vry;
                        % Break the loop
                        break;
                    end
```

The following figure shows the simulation plot.



3b. Injection

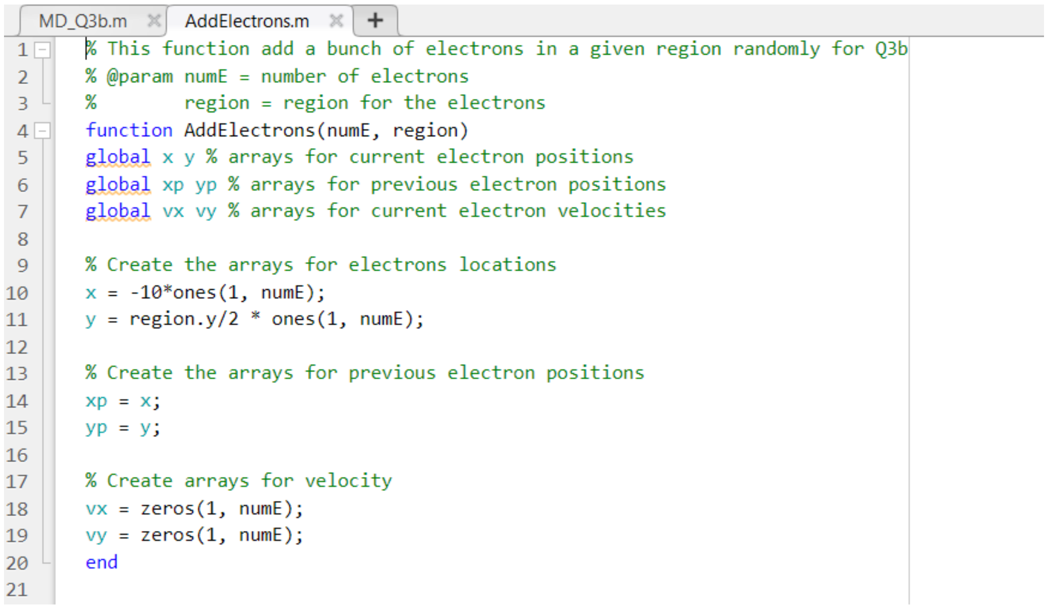
(b) **Injection:** Model electron injection into a region:

The codes for this part are in the folder named “Q3b”, which contains the following files:

- AddElectrons.m – Function that initializes electrons with invalid electrons positions
- AddObstacles.m – Function that adds rectangle and circle obstacles
- findCircReflect.m – Function that finds the reflected vector off the circle
- findLineCircIntersect – Function that finds the intersecting point on the circle
- globalVars.m – File that groups the constants used in the simulation
- MD_Q3b.m – Entry point script for the simulation for part 3b
- PlotPoint.m – Function that plots the trajectories of the electrons
- plotTempDistribution.m – Function that plots the initial temperature distribution
- tempDisplay.m – Function that displays the temperature map

i. Do not have any electrons present at the beginning.

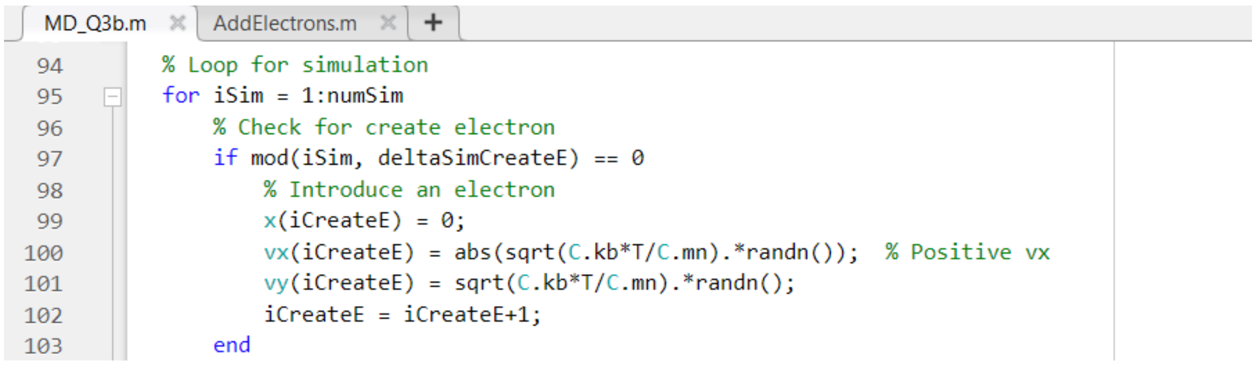
The “AddElectrons()” is modified to initialize electrons with invalid electrons positions and zero velocities.



```matlab
% This function add a bunch of electrons in a given region randomly for Q3b
% @param numE = number of electrons
%        region = region for the electrons
function AddElectrons(numE, region)
global x y % arrays for current electron positions
global xp yp % arrays for previous electron positions
global vx vy % arrays for current electron velocities

% Create the arrays for electrons locations
x = -10*ones(1, numE);
y = region.y/2 * ones(1, numE);

% Create the arrays for previous electron positions
xp = x;
yp = y;

% Create arrays for velocity
vx = zeros(1, numE);
vy = zeros(1, numE);
end

```

ii. Introduce them during the simulation from the left side with a positive v_x – derived from a thermalized velocity within a small central region.

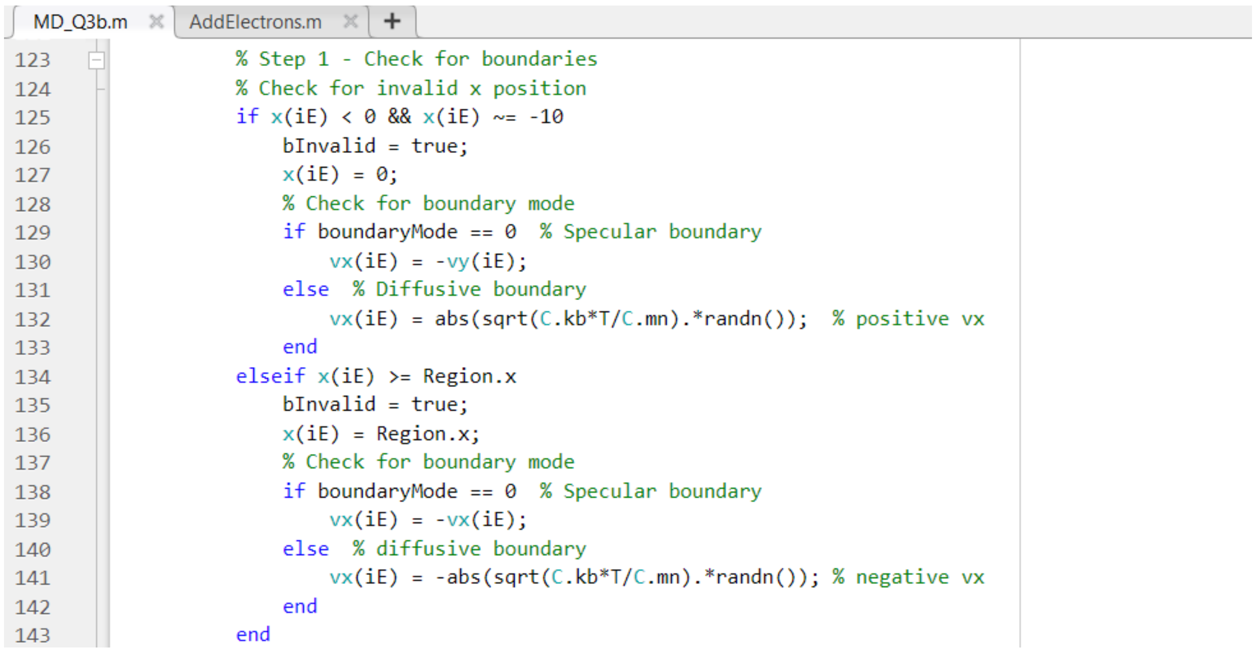
Inside the simulation loop, the electrons are introduced by adjusting the position and thermalizing the velocity.


MD_Q3b.m AddElectrons.m

```
% Loop for simulation
for iSim = 1:numSim
    % Check for create electron
    if mod(iSim, deltaSimCreateE) == 0
        % Introduce an electron
        x(iCreateE) = 0;
        vx(iCreateE) = abs(sqrt(C.kb*T/C.mn).*randn());  % Positive vx
        vy(iCreateE) = sqrt(C.kb*T/C.mn).*randn();
        iCreateE = iCreateE+1;
    end
```

iii. Turn off the periodic BC conditions in x.

The periodic boundary conditions in x are changed to closed boundary conditions.

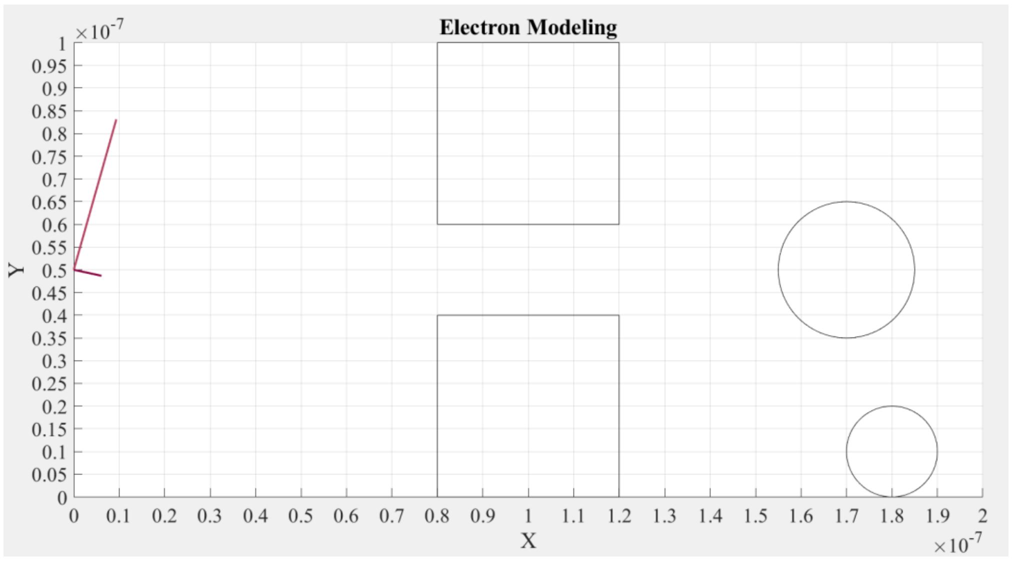

MD_Q3b.m AddElectrons.m

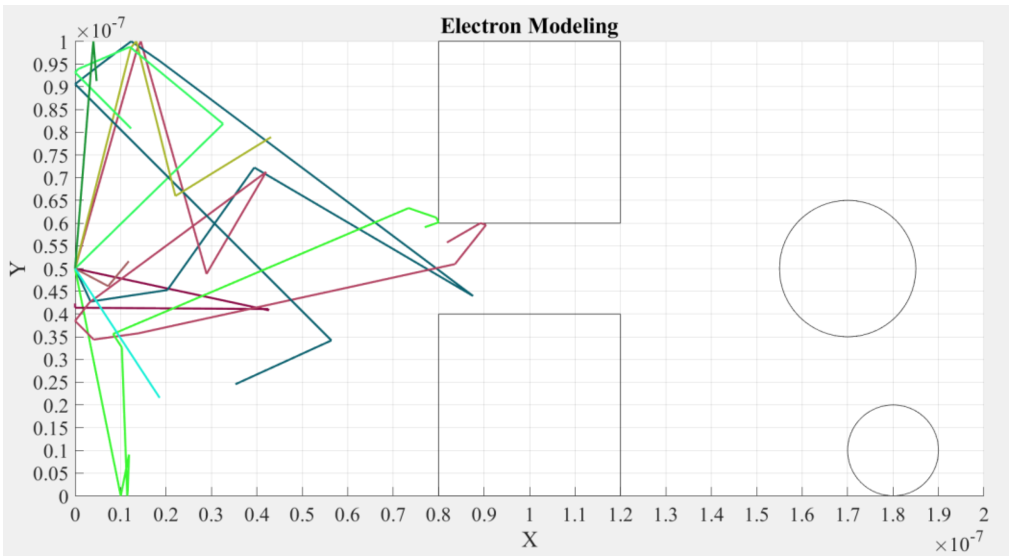
```
        % Step 1 - Check for boundaries
        % Check for invalid x position
        if x(iE) < 0 && x(iE) ~= -10
            bInvalid = true;
            x(iE) = 0;
            % Check for boundary mode
            if boundaryMode == 0  % Specular boundary
                vx(iE) = -vy(iE);
            else  % Diffusive boundary
                vx(iE) = abs(sqrt(C.kb*T/C.mn).*randn());  % positive vx
            end
        elseif x(iE) >= Region.x
            bInvalid = true;
            x(iE) = Region.x;
            % Check for boundary mode
            if boundaryMode == 0  % Specular boundary
                vx(iE) = -vx(iE);
            else  % diffusive boundary
                vx(iE) = -abs(sqrt(C.kb*T/C.mn).*randn()); % negative vx
            end
        end
```

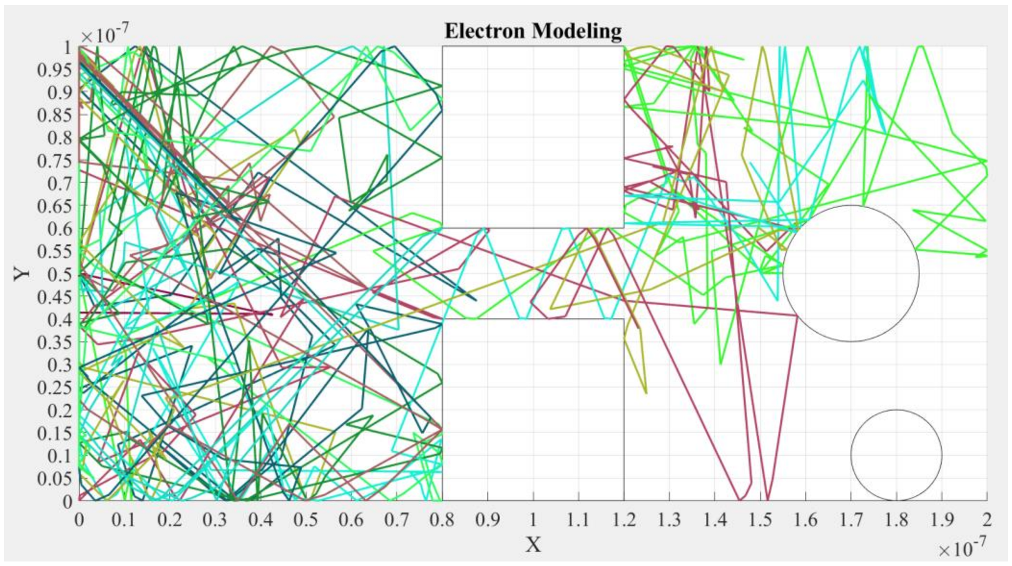
iv. Investigate turning on and off scattering and adding boxes.

v. Play with the initial value of v_x and v_y.

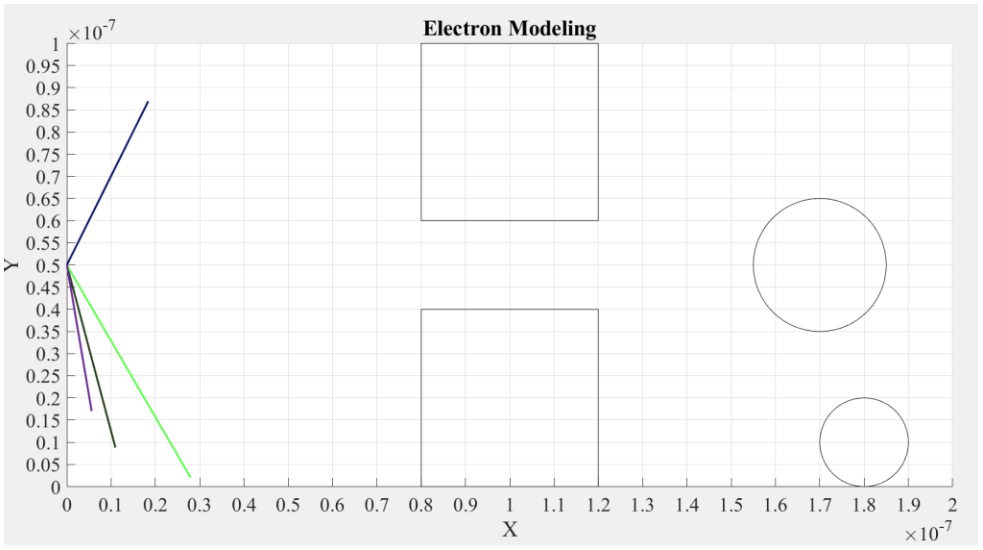
The following figures shows the simulation plots when the scattering is turned on.

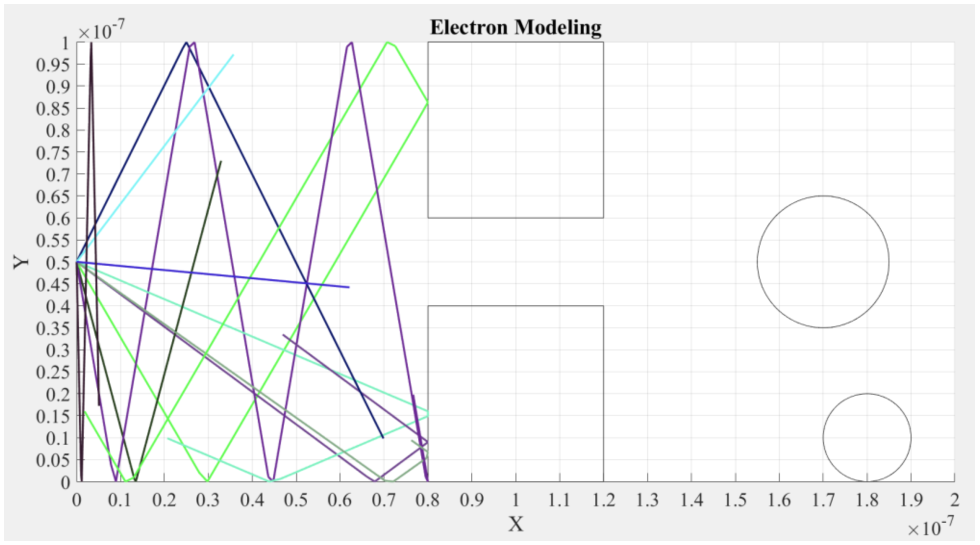

Electron Modeling
×10^-7
Y
X
×10^-7

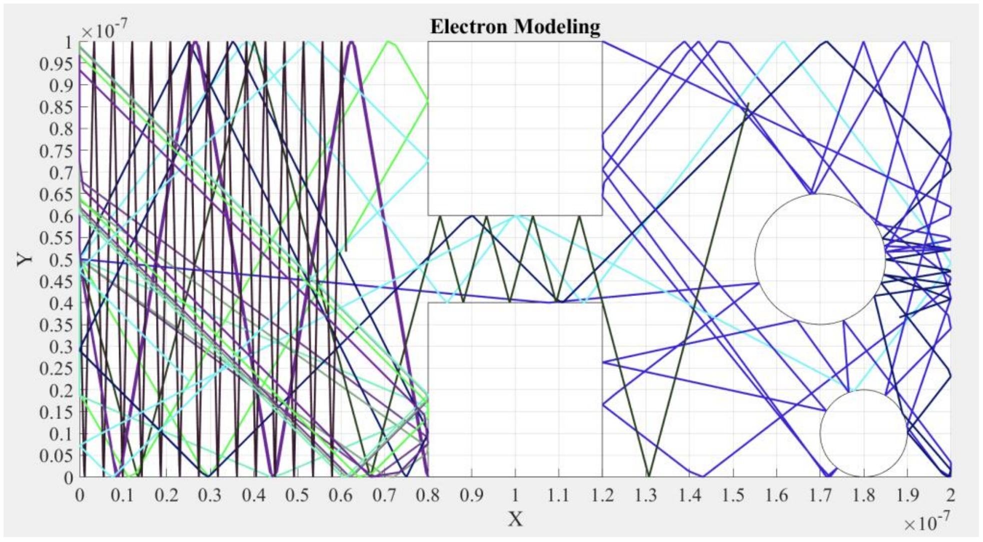

Electron Modeling
×10^-7
Y
X
×10^-7


Electron Modeling
×10^-7
Y
X
×10^-7

The following figures shows the simulation plots when the scattering is turned off.







Reference

[1] 2022. ELEC 4700 Assignment - 1 Monte-Carlo Modeling of Electron Transport. Ottawa: Carleton University.