Megan McEwen
Student #101003509
2 February 2020

Assignment 1 Report

1.0 Introduction

This report describes the design and function of an electron modelling simulation that was designed in Matlab for the purpose of understanding electron movement and thermal properties of a semiconductor. There are three parts to this report. Section 1 considers a basic model, in which electrons move freely inside a closed environment with constant velocity and constant temperature. Section 2 expands this model to consider random scattering effects, using a Maxwell-Boltzmann distribution. Finally, Section 3 uses this model in a hypothetical scenario with a bottleneck inserted into the semiconductor area, which has diffusive and specular boundaries.

1.1 Electron Modelling

This section considers a simple model, using a set of electrons that are free to move around a closed environment.

The thermal velocity vTH was calcuated using the equation shown:

```
vTH = sqrt(2*k*T/m);
```

where k is Boltzmann's constant, T = temperature in Kelvin, and m = 0.26*electron mass m0. Since this equation uses constants that are not changed throughout the simulation, vTH stays constant at 1.8702e+0.5 m/s.

The mean time between collisions is given as tau = 0.2ps. Given this, the mean free path (MFP) can be calculated as shown:

```
VAvg = mean(Vx.^2 + Vy.^2);
MFP = VAvg*tau;
```

where VAvg is the average overall velocity. For this section, the mean free path also remains constant (as velocities remain unchanged), at a value of MFP = 0.007.

The simulation for this section creates a set of particles of size n and assigns each particle a position (Px,Py) and velocity (Vx,Vy). This is done by creating blank arrays for each particle element, as shown:

```
Px = zeros(n,1);
Py = zeros(n,1);
Vx = zeros(n,1);
Vy = zeros(n,1);
```

Next, each particle is assigned a random initial location within the boundaries:

```
Px(:,1) = xrange*rand(n,1);
```

```
Py(:,1) = yrange*rand(n,1);
```

Each particle is also assigned a fixed velocity vTH, and a random initial direction:

```
randAngle = 2*pi*rand(n,1);
Vx(:,1) = vTH * cos(randAngle);
Vy(:,1) = vTH  *sin(randAngle);
```

The program then loops through the particles for a certain number of iterations (as defined by the user). Each iteration lasts for a specified time interval(timeStep).First, the original position of each particle is saved:

```
Px_old = Px;
Py_old = Py;
```

Next, the position of each particle is updated using Newton's Law of motion (F = ma):

```
Px = Px + Vx*timeStep;
Py = Py + Vy*timeStep;
```

As the particles move, their trajectories are plotted. Figure 1 below shows the movements of 10 particles for 500 iterations:

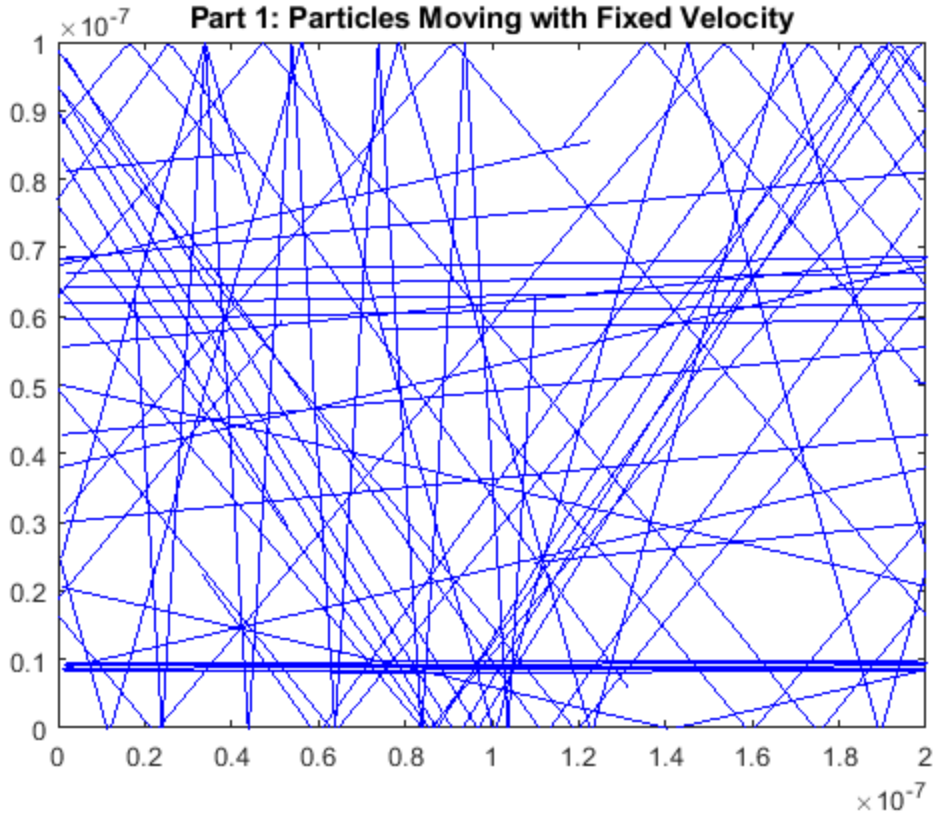

Figure 1: Particle Movement with Constant Velocity

There are a few boundary conditions at play as the particles move around the space. Whenever a particle hits a y boundary (top/bottom of space), it reflects at a specular angle. To make sure only the particles that are hitting the boundary reflect, logical indexing is used to find the particles that are about to go out of range. Then, these particles are redirected at a specular angle:

```
Vy(Py >= yrange) = Vy(Py >= yrange) * -1;
Vy(Py <= 0) = Vy(Py <= 0) * -1;
Py(Py>yrange) = yrange-(Py(Py>yrange)-yrange);
```

Particles approaching an x boundary do not reflect; instead, they move through the boundary and reappear on the opposite of the space. Again, logical indexing is used to only redirect the particles that are approaching a boundary:

```
%x hitting right side
id = Px >= xrange;
Px(id) = Px(id) - xrange;
Px_old(id) = Px_old(id) - xrange;

%x hitting left side
id = Px <= 0;
Px(id) = Px(id) + xrange;
Px_old(id) = Px_old(id) + xrange;
```

The semiconductor temperature is also calculated as the particles move:

```
temp(i) = (1/2)*(m*(VAvg))*(1/k);
```

which is used in a loop, so that temperature is calculated at every timestep. This temperature is plotted as shown in Figure 2:

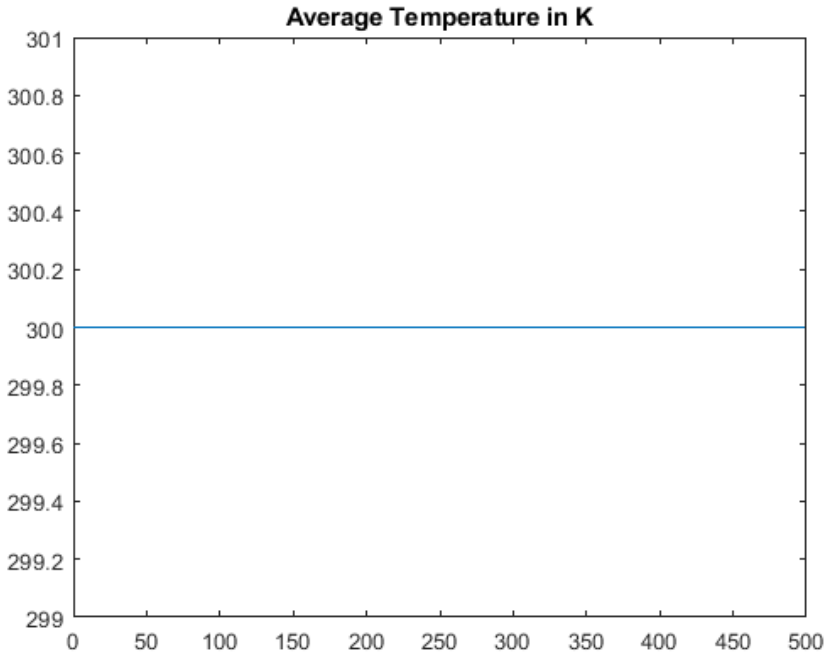

Figure 2: Average Temperature for Part 1 (Constant Velocity)

As expected, since the average velocity remains constant, the temperature remains constant at T = 300K. The particles in this code are all plotted with the same colour, making it difficult to discern individual particle paths as the number of particles increases. Determining a way to give each particle its own colour would be an ideal improvement to this simulation.

1.2 Collisions with Mean Free Path (MFP)

This section repeats much of the work done in Section 1, with some added complexities. Instead of each particle having the same velocity, all particles are now assigned a velocity using a Maxwell-Boltzmann distribution. To do this, randn is used to create a random normal distribution for each velocity component:

```
V_X(:,1) = vTH.*randn(n,1);
V_Y(:,1) = vTH.*randn(n,1);
```

This distribution is required to have an average overall velocity of approximately vTH; to check this, as well as to check the distribution, three histograms are generated. V_X and V_Y have histograms centred at 0 with a normal distribution shape; Avg confirms that the average of all velocities remains vTH. All three histograms for n = 500 particles are shown in Figure 3 below:

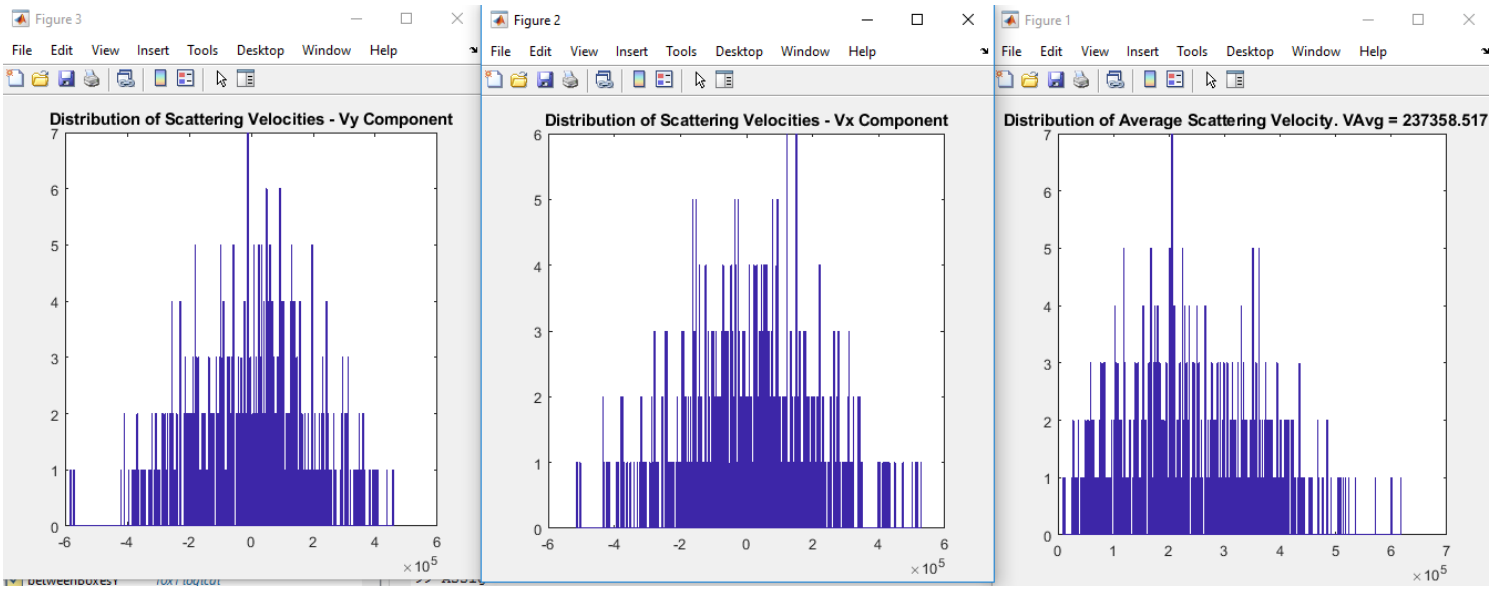

Figure 3: Histograms for Maxwell-Boltzmann Distribution

Next, each particle is given a random probability of scattering. Using logical indexing, particles are selected at random using the Pscat equation. The selected particles are then re-thermalized and are given a new velocity value from the Maxwell-Boltzmann distribution:

```
Pscat = 1-exp(-dt/tau);
ind = Pscat > rand(n,1);

Vx(ind) = sqrt((k*T)/m).*randn(sum(ind),1);
Vy(ind) = sqrt((k*T)/m).*randn(sum(ind),1);
```

The final plot is shown below in Figure 4 (n = 10, 500 iterations):

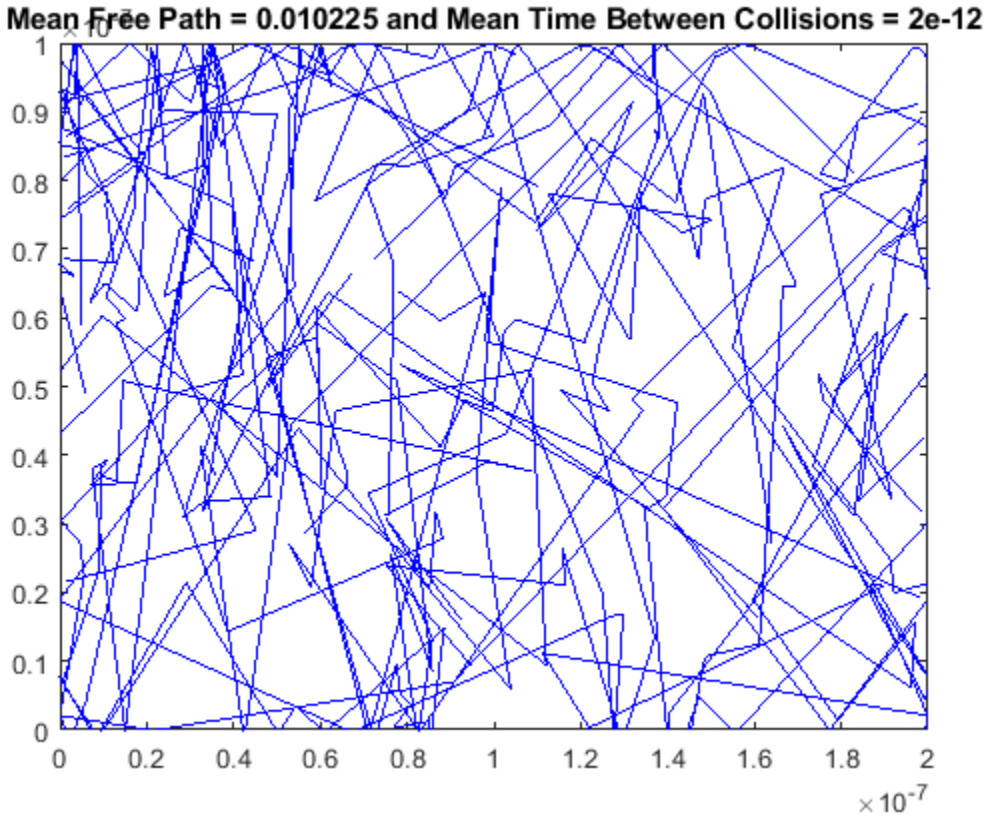

Figure 4: Electron Plot with Scattering

Note that unlike the first plot, the particles do not move in continuous straight lines; occasionally, they randomly begin moving in a new direction, resulting in unpredictable paths. The mean time between collisions remains unchanged; however, the mean free path changes with each iteration, while still remaining relatively close to its original value of 0.007.

The average temperature value is calculated using the same code shown in Section 1; however, since velocity is no longer constant, the temperature has some variation (though its average is still ~300K). Figure 5 shows the temperature plot for Section 2:

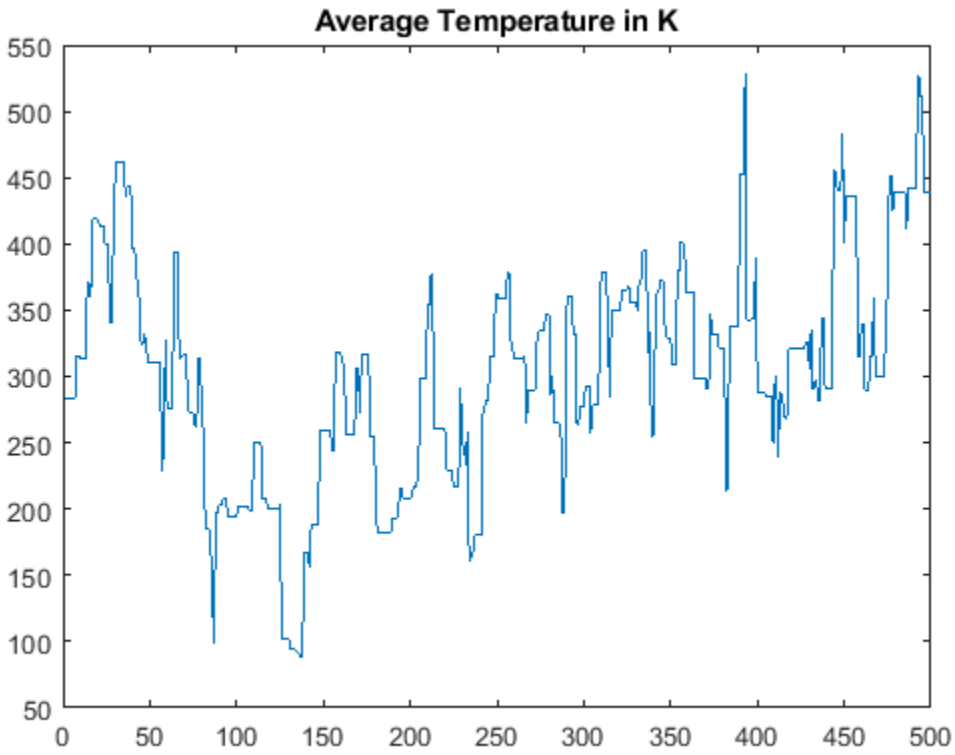

Figure 5: Average Temperature Plot with Scattering

1.3 Enhancements

The final section adds a bottleneck that particles must navigate around. To create the “boxes” that mark the bottleneck, boundaries are defined as shown:

```
%define boxes for bottleneck
boxLeft = 0.8e-7;
boxRight = 1.2e-7;
boxBottom = 0.6e-7;
boxTop = 0.4e-7;

%define inside of boxes
inX = (boxLeft < Px) & (Px < boxRight);
inY = (boxBottom < Py) | (boxTop > Py);
inBox = inX&inY;
```

There are two cases that must be addressed to prevent particles from entering the bottleneck. Outside of the loop, particles must be initialized to not be inside the boxes. This is done using the code below, which uses logical indexing to find particles located inside the boxes and repeatedly removes them to a random new location until all particles are outside the bottleneck:

```
while(numInBox > 0)
    Px(inBox) = xrange*rand(1,numInBox);
    inX = (boxLeft < Px) & (Px < boxRight);
    inY = boxBottom < Py | boxTop > Py;
    inBox = inX&inY;

    numInBox = sum(inBox)
end
```

The second case deals with particles that approach the bottleneck as they iterate and move through the space. This is done inside the loop, again using logical indexing to find particles that are about to cross a box boundary and reversing their velocity so they reflect off of the box edge:

```
%bouncing off boxes if box boundary encountered
   inX = (boxLeft < Px) & (Px < boxRight);
   inY = boxBottom < Py | boxTop > Py;
   inBox = inX&inY;

   betweenBoxes = (Py_old > boxTop)&(Py_old < boxBottom);
   Vy(inBox&betweenBoxes) = -1*Vy(inBox&betweenBoxes);
   Vx(inBox&~betweenBoxes) = -1*Vx(inBox&~betweenBoxes);
```

The plot for this section can be seen in Figure 6 (n = 300, 100 iterations):

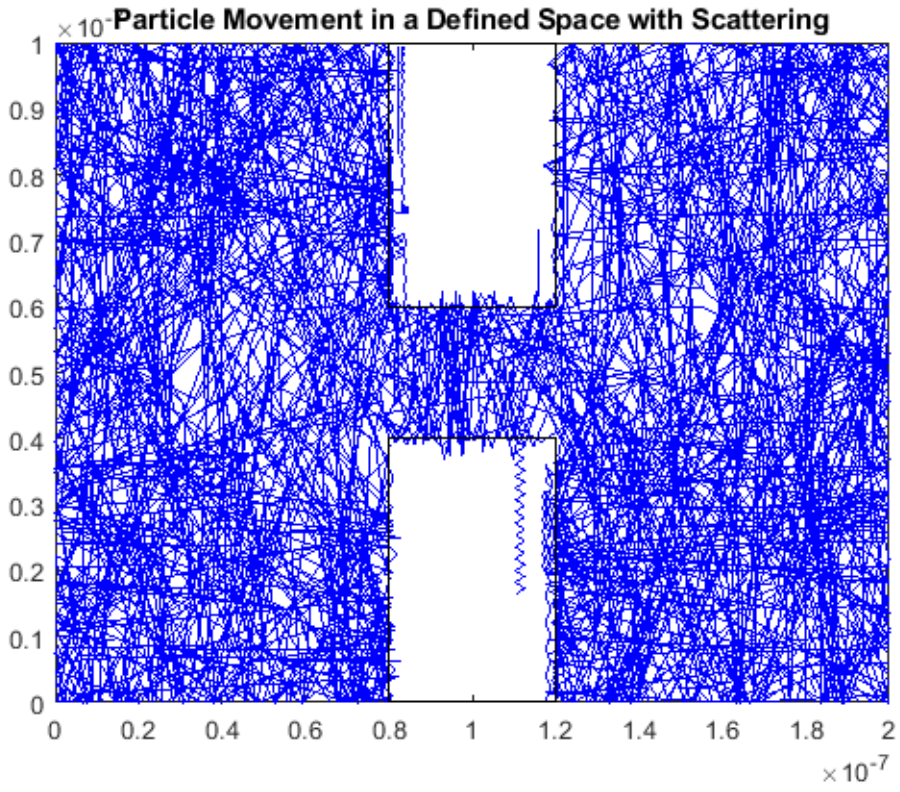

Figure 6: Particle Map with Bottleneck

Note that a few particles are able to get through the bottleneck boundaries and become trapped inside; this is likely due to the boundary conditions, which are only enacted once particles move beyond the box boundary. If the particle moves too far before being corrected, it does not move out of the box, and subsequently becomes trapped. Since this only happens for a few particles, the model as a whole remains accurate.

The temperature plot for this section is shown in Figure 7. Note that it is similar to that of Section 2, with a varying temperature that averages to ~300K:

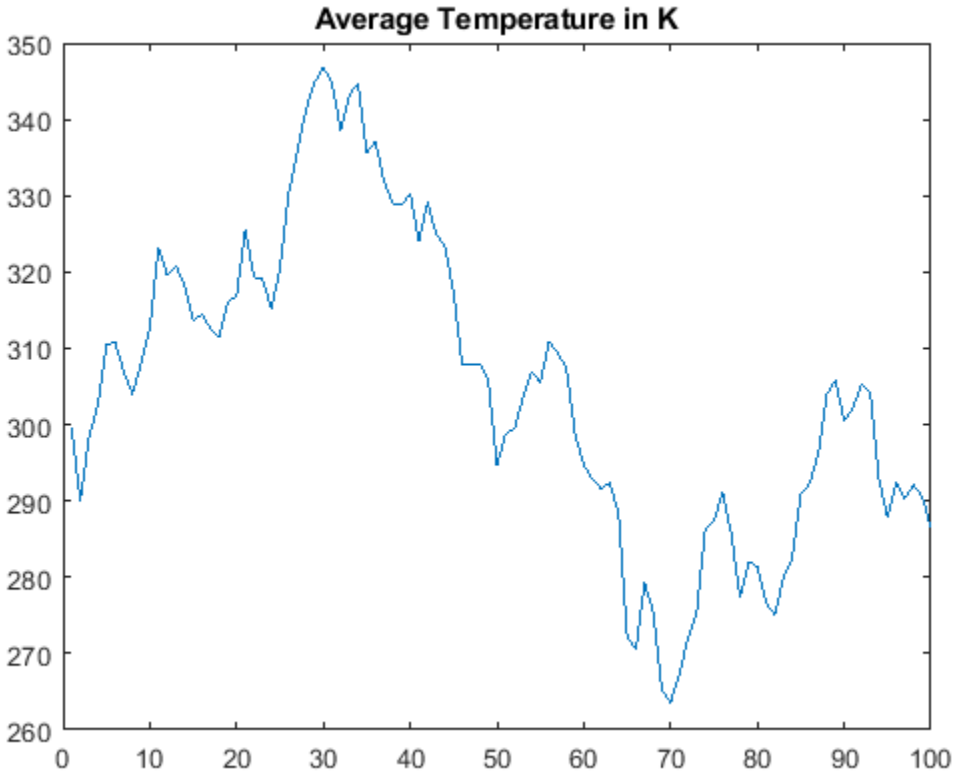

Figure 7: Average Temperature for Section 3

At the end of the simulation, an electron density map is generated showing the final position of particles and their frequency in a grid. This is done using the hist3 function, as shown:

```
%electron density map with final positions
figure(4)
hist3([Px Py],'CdataMode','auto')
title(['Electron Density Map after ', num2str(iter),' iterations for '
,num2str(n),' particles']);
colorbar
view(2)
```

The final electron density map is shown in Figure 8:

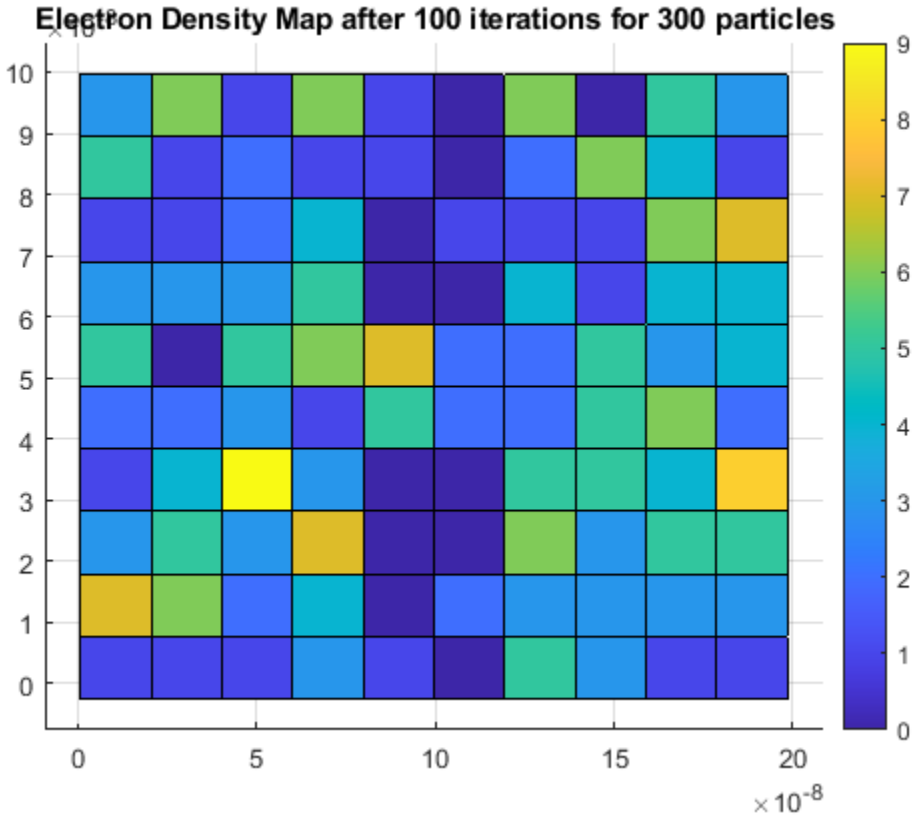

Figure 8: Electron Density Map