Appendix A: Setting up a simulation

The first step of setting up a simulation is making a sample that is going to be simulated. A sample consists of a channel with contacts attached. The **Sample.m** class is used to simplify the construction of a device. Creating a sample is done by invoking **Sample**(*wid,len,eps,t,a*). The initial width and length of the channel has to be specified along with the value of ϵ for the lattice points and the interaction τ. The separation distance between lattice points a can be ignored in the constructor and then it will be set by default to 1 nm.

To simplify the construction of more elaborate channels containing barriers or other features, the append function can simplify construction. **Sample.append**(*M,side*) appends the matrix M to the the desired side of the channel with the default side being right. The size of M needs to match the side it is being attached to. The directions in the context of this code is what is seen when looking at the sample from above or printing out the channel units in the MATLAB command window or when using MATLABs feature *imagesc* with the directions up, down, left or right. An example of a channel of width 3 and length 7 with a barrier in the middle is seen in figure 1 with the directions included in the figure.

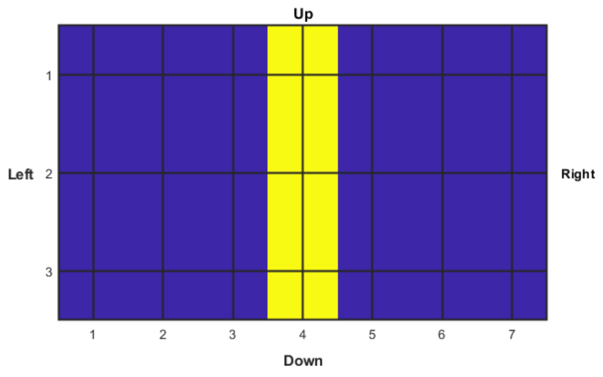

Figure 1: A simple 3 by 7 sample.

The following snippet of code creates a sample with a channel width of 5 lattice points and length of 6 lattice points. The append feature is then used to create a double barrier across the channel with a spacing of 6 lattice points, summing up to a total length of 20 lattice points.

```
load('physical_constants.mat');

Ec = 0.4* eV;
a = 10E-10;
effective_mass = 0.2;
t0 =  h_bar^2/(2*electron_mass*effective_mass* a^2);
t =-t0;
eps = Ec - 4*t;

sample = Sample(5,6,eps,t,a);                   %Width = 5, length = 6.
sample.append(ones(sample.width,1)*eps*1.1);    %Width = 5, length = 7.
sample.append(ones(sample.width,6)*eps);        %Width = 5, length = 13.
sample.append(ones(sample.width,1)*eps*1.1);    %Width = 5, length = 14.
sample.append(ones(sample.width,6)*eps);        %Width = 5, length = 20.
```

After the channel has been implemented, the contacts have to be attached. This is done by using **Sample.addContact**(*M,tau,pos*). M is the size of the contacts unit cell with the amplitude of the contacts ϵ value, *tau* describes the interaction between lattice points and *pos* is the upper position where the contact interacts with the channel. The face direction of the contact also has to be specified, with 1 being the default value facing right, a value of -1 means that the contact is facing left. A contact being attached to the right of the channel needs to be facing left into the channel. Every contact is described by it's own class **Contact.m** which the **Sample.m** class creates in **Sample.addContact**() with a default Fermi level of 1. These contacts are then stored in the samples cell property **Sample.contacts**, accessed by calling **sample.contacts**{*conNmbr*}. The following code snippet adds 2 contacts to the channel made earlier and setting the Fermi levels of the contacts along with the face.

```
sample.addContact(ones(sample.width,1)*eps,t,[1,1]);
sample.addContact(ones(sample.width,1)*eps,t,[1,sample.length]);
sample.contacts{end}.fermi = 0;     %The last contact added to the sample. (Contact 2)
sample.contacts{end}.face = -1;
```

The final addition to the sample is setting the value of the scattering matrix **D** which is by default set to 0. If D is going to have any other value than 0 it needs to be of the same dimension as the Hamiltonian constructed for the channel. This Hamiltonian will have a side of M which is the amount of lattice points in the channel. The value of M is saved as a property in the **Sample.m** class. If noise is desired in the channel this can be added by using the function **Sample.applyNoise**(*amp,corLength*) where *amp* is the standard deviation of the noise relative to the max value in **sample.units**.

```
sample.D = 0;
%sample.D = ones(sample.M)*eps*eV*1E-5;          %Phase relaxation.
%sample.D = eye(sample.M,1)*eps*eV*1E-8;         %Phase + momentum relaxation.
%sample.applyNoise(0.03,2);                      %3% noise, corrLength 10 lattice points.
```

After a sample has been constructed a range of energy values needs to be defined. If desired a magnetic field can also be applied. Before performing the computations to find the self energies $\mathbf{\Sigma}$ and scattering matrices $\mathbf{\Sigma}_0$ the parameters have to be set for the simulations. This is done by constructing a **NEGF_param.m** object which is a simple class to carry data to the simulation functions. Constructing a **NEGF_param** object is done by invoking **NEGF_param**(*sample,E,B,compress*) where the *sample* and E has to be specified. B is 0 by default and compress is false by default. The **NEGF_param** object contains multiple other properties for the simulations that can be changed to adjust the precision of the simulations. The *compress* property is a boolean that decides if the result from the NEGF simulations should be compressed by the lossy compression algorithm before being returned. This slightly increases run-time but scales much better than the NEGF calculations for bigger samples so the added run-time is negligible for larger samples.

```
E = linspace(Ec,Ec + 3*t0, 200);
B = 0;
NEGF_param = NEGF_param(sample,E,B);
NEGF_param.print = true;
NEGF_param.errorMarg = 1E-10;
```

The final line of code in the snippet changes the allowed error margin for the converging algorithms to calculate Σ and Σ_0. Once this has been done it's time to start the calculations to find Σ and, if $D \neq 0$, Σ_0. This is done by calling the function **NEGF.m** or **NEGF_map.m**. If E or B in **NEGF_param** are vectors with multiple values then **NEGF_map** needs to be used, if **NEGF** is used instead, only the first values in E and B will be used. **NEGF** will return a **NEGF_result** which contains the resists from the calculations. **NEGF_map** returns a structure instead with all the **NEGF_result**s stored in a cell array named *NEGF_result*. The cell array is a 2D array with the varying E values along the columns and B along the rows. **NEGF_map** needs to be used in this example since the earlier snippet of code defined E as a range of values between E_c and $E_c + 3 \cdot t_0$.

```
res_struct = NEGF_map(NEGF_param);
```

Once simulations have been completed it is time to analyse the results, for this the functions **NEGF_result_remap**(*NEGF_result,data*) and **NEGF_transmission**(*NEGF_result*) simplify the analysis. **NEGF_result_remap** returns a 2D map of desired type of *data*, default *'electrons'*, at every lattice point for the specified **NEGF_result**. For further information on what different types of data can be extracted in a map look at the documentation of **NEGF_result_remap**. **NEGF_transmission**(*NEGF_result,c*
calculates the transmission as a single value for that *NEGF_result* at contact *con*. The following snippet of code extracts the transmission curve for the simulation prepared above.

```
T = zeros(1,length(res_struct.E));
for i = 1:length(res_struct.E)
    T(i) = NEGF_transmission(res_struct.NEGF_result{i}); %Contact 1 by default.
end
figure(1)
plot(E/eV,T,'LineWidth',1.5)
axis([E(1)/eV, E(end)/eV, 0, max(T)*1.1]);
xlabel("E [eV]"); ylabel("Transmission");
```

```
title("Transmission in 2D wire with double barrier")
grid
set(gca, 'FontWeight', 'bold')
```

This results in the plot found in figure 2. Since **Sample.applyNoise**() is a A simple animation is also able to show how electron density varies in the sample as E changes by using **NEGF_result_remap**().

```
figure(2)
for i = 1:length(res_struct.E)
    imagesc(NEGF_result_remap(res_struct.NEGF_result{i}))
    title("E = " + res_struct.E(i)/eV + "eV")
    pause(0.05)
end
```

A single value can also be shown if the for-loop in the previous snippet is ignored and a single value for E is chosen.

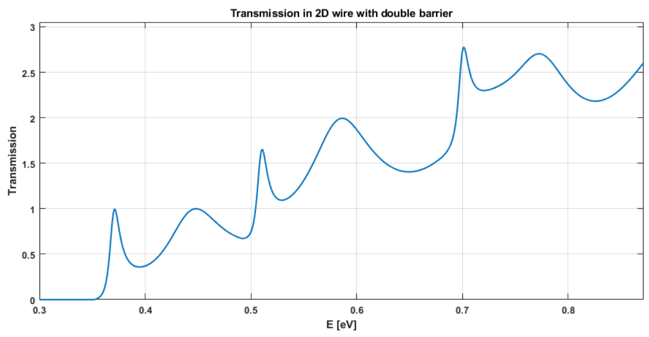

Figure 2: Transmission curve for 2D wire in simulations set up from snippet.