

Handbook for Multi-subband Monte Carlo (MSMC) code

This handbook is to help future developers get familiar with the algorithms and source codes for the MSMC approach for quasi-1DEG system such as Silicon nanowire MOSFETs. The source files were originally written in C language by Tuan, however, there are many of rooms to improve them.

The theoretical treatment of scattering mechanisms and the MSMC method can be found in [Tuan's PhD thesis](#) and the references therein. Here, we devoted to the implementation of the model.

Table of Contents

| | |
|---|-----------|
| 1. An Overview of the MSMC subroutines..... | 10 |
| 1.1. Main program of Single particle MSMC code | 11 |
| 1.2. Main program of an Ensemble MSMC code | 14 |
| 2. The Configuration module..... | 16 |
| 2.1. The <i>material_param()</i> function. See material_param.c | 16 |
| 2.2. The <i>device_structure()</i> function. See device_structure.c | 19 |
| 2.3. The <i>read_voltages_input()</i> function. See read_voltages_input.c | 21 |
| 2.4. The <i>read_simulation_list()</i> function. See read_simulation_list.c..... | 22 |
| 2.5 <i>read_option_for_speed_up()</i> function. See read_option_for_speed_up.c | 24 |
| 2.6 The <i>read_parameters_for_smooth_method()</i> function. See read_parameters_for_smooth_method.c | 24 |
| 2.7 The <i>read_scattering_save_list()</i> function. See read_scattering_save_list.c..... | 25 |
| 2.8 The <i>read_option_for_ANALYSIS()</i> function. See read_option_for_ANALYSIS.c 27 | |
| 2.9 The <i>initial_variables()</i> function. See initial_variables.c | 29 |
| 2.10 The <i>trapezoidal_weights()</i> function. See trapezoidal_weights.c..... | 30 |
| 2.11. 2D Schrodinger and 2D Poisson Solver. | 30 |
| 2.12. The <i>line_charge_density()</i> function. See line_charge_density.c..... | 31 |
| 2.13. The <i>quantum_electron_density()</i> function. See quantum_electron_density.c ... | 31 |
| 2.14. The <i>charge_density()</i> function. See charge_density.c..... | 31 |
| 3. The Scattering table and Normalization module..... | 32 |

| | |
|--|-----------|
| 3.1. The <i>read_data_for_Mobility()</i> function. See <i>read_data_for_Mobility.c</i> | 32 |
| 3.2. The <i>form_factor_calculation()</i> function. See <i>form_factor_calculation.c</i> | 32 |
| 3.3. The <i>SRS_form_factor_calculation()</i> function. See <i>SRS_form_factor_calculation.c</i> | 33 |
| 3.4. The <i>Scattering_table()</i> function. See <i>scattering_table_Mobility.c</i> | 35 |
| 3.4.1. The <i>AC_scattering_table()</i> function. See <i>AC_scattering_table.c</i> | 39 |
| 3.4.2 The <i>OP_scattering_table()</i> function. See <i>OP_scattering_table.c</i> | 43 |
| 3.4.3. The <i>SRS_scattering_table()</i> function. See <i>SRS_scattering_table.c</i> | 44 |
| 3.5. The <i>normalize_table()</i> function. See <i>normalize_table.c</i> | 46 |
| 4. Initial Condition module..... | 50 |
| 4.1. The <i>electrons_initialization_for_Mobility()</i> function. See <i>electrons_initialization_for_Mobility.c</i> | 50 |
| 4.2. The <i>save_electron_distribution()</i> function. See <i>save_electron_distribution.c</i> | 54 |
| 5. Multi-subband Monte Carlo transport for electrons | 55 |
| 5.1. Single particle Monte Carlo..... | 55 |
| 5.1.1. The <i>drift()</i> function. See <i>drift.c</i> | 61 |
| 5.1.2. The <i>scattering()</i> function. See <i>scattering_Mobility.c</i> | 65 |
| 5.1.3. The <i>occupation_electron()</i> function. See <i>occupation_electron.c</i> | 81 |
| 5.1.4. The <i>gather_data_from_all_nodes()</i> function. See <i>gather_data_from_all_nodes.c</i> | 83 |
| 5.2 Ensemble Monte Carlo..... | 89 |
| 5.2.1 Ensemble Monte Carlo (Serial code). See <i>emcd_Mobility()</i> function in <i>EMC_emcd_Mobility.c</i> | 90 |
| 5.2.2 The <i>drift()</i> function. See <i>EMC_drift.c</i> | 95 |
| 5.2.3 The <i>velocity_energy_cumulative_Mobility()</i> function. See <i>EMC_velocity_energy_cumulative_Mobility</i> | 96 |
| 5.2.5 Ensemble Monte Carlo (Parallel code). See <i>emcd_Mobility_paralell()</i> function in <i>EMC_emcd_Mobility.c</i> | 102 |

| | |
|---|------------|
| 6. Multi-subband Monte Carlo transport for holes..... | 107 |
| 6.1. Scattering mechanisms..... | 108 |
| 6.1.1 Acoustic phonon scattering | 108 |
| 6.1.2 Optical phonon scattering..... | 109 |
| 6.1.3 SRS scattering | 109 |
| 6.1.4. The scattering parameters used for hole mobility calculation..... | 110 |
| 6.2 Some changes in MSMC code | 110 |
| 6.2.1 The <i>read_data_for_Mobility()</i> function. | 111 |
| 6.2.2 For other functions | 111 |
| 7. A fully self-consistent MSMC method for the modeling of Silicon Nanowire MOSFETs..... | 112 |
| 7.1. Overall Description of the Model..... | 113 |
| 7.2. The Configuration module | 114 |
| 7.3. The Initialization module | 116 |
| 7.3.1. Initialization in k-space | 116 |
| 7.3.2 Initialization in real-space | 117 |
| 7.3.3 Guess potential | 117 |
| 7.4. The 2D Schrodinger solver..... | 117 |
| 7.5. The scattering table and normalization module..... | 117 |
| 7.6 Ensemble Monte Carlo | 118 |
| 7.6.1. Drift process | 118 |
| 7.6.2. Scattering process..... | 119 |
| 7.7. The <i>check_source_drain_contacts()</i> function | 119 |

| | |
|---|-----|
| 7.8 The delete_particles() function | 120 |
| 7.9. Charge calculation | 121 |
| 7.10. Poisson Equation | 121 |
| 7.11. Current calculation | 121 |

List of Figures

| | |
|--|----|
| Figure 1.1 Typical flowchart of MSMC transport kernel for low-field mobility calculation | 10 |
| Figure 1.2 An example code of <i>main()</i> function using Single particle Monte Carlo method..... | 12 |
| Figure 1.3 An example code of <i>main()</i> function using Ensemble Monte Carlo method | 15 |
| Figure 2.1 List of functions used in the configuration module | 16 |
| Figure 2.2 A part of material parameter source code | 18 |
| Figure 3.1 List of functions used in the scattering tables and normalization module | 32 |
| Figure 3.2 An example source code for making acoustic phonon scattering table | 42 |
| Figure 3.3 An example source code for making optical phonon scattering table | 44 |
| Figure 3.4 An example source code for making SRS scattering table | 46 |
| Figure 3.5 An example code for normalization of scattering table | 48 |
| Figure 3.6. Example of successive summation of scattering rates | 49 |
| Figure 4.1 List of functions used in the initial condition module | 50 |
| Figure 4.2 An example code for electron initialization | 53 |
| Figure 4.3 An illustration of how to initialize electron based on the percentage population | 53 |
| Figure 5.1 List of functions used in the Single particle Monte Carlo | 55 |
| Figure 5.2 An example code for Single particle Monte Carlo | 59 |
| Figure 5. 3 An example code for <i>drift()</i> function | 63 |
| Figure 5.4 The position of transient points are indicated | 64 |

| | |
|--|----|
| Figure 5.5. The scattering() function. Part 1: if only Acoustic scattering is included.... | 67 |
| Figure 5. 6 Illustration of normalized scattering table for only Acoustic phonon | 69 |
| Figure 5. 7 The scattering() function. Part II: if Acoustic and Non-polar optical phonon scatterings are included..... | 70 |
| Figure 5. 8 Illustration of normalized scattering table for Acoustic and Non-polar Optical phonon..... | 71 |
| Figure 5. 9 The scattering() function. Part III: if Acoustic, Non-polar optical phonon and SRS scatterings are included..... | 73 |
| Figure 5. 10 Illustration of nomarlized scattering table for Acoustic, Non-polar Optical phonon and SRS scatterings..... | 74 |
| Figure 5. 11 An example code for isotropic_Acoustic() function..... | 76 |
| Figure 5.12 An example code for isotropic() function | 78 |
| Figure 5.13 An example code for <i>SRS_Forward_or_Backward()</i> function..... | 80 |
| Figure 5. 14 An example code for <i>occupation_electron()</i> function | 82 |
| Figure 5. 15 Schematic of making SMC parallel code for mobility calculation..... | 83 |
| Figure 5. 16 An example code for <i>gather_data_from_all_nodes()</i> function. Part 1: Velocity and Energy | 86 |
| Figure 5. 17 Parallel performance of the SMC and EMC code..... | 89 |
| Figure 5.18 List of functions used for EMC code | 90 |
| Figure 5.19 Flowchart for a particle experiences drift and scattering processes within one time step | 91 |
| Figure 5.20 An examlpe EMC code (serial version)..... | 93 |
| Figure 5. 21 An example code for drift() function used in EMC | 95 |

| | |
|--|-----|
| Figure 5.22 An example code for velocity_energy_cumulative_Mobility() function . | 100 |
| Figure 5.23 An example code for EMC (parallel code) | 105 |
| Figure 7. 1 Flowchart of the typical Multi-subband Monte Carlo for quasi-1DEG systems..... | 113 |
| Figure 7. 2 The device co-ordinate system..... | 115 |
| Figure 7.3 Schematic of mesh layout near the source contact. The shaded area is the charge neutral region..... | 116 |
| Figure 7. 4 A simple method to delete particles | 121 |

List of Tables

| | |
|--|----|
| Table 1. 1 An example of “trick” indices corresponding to real gate voltage values. | 13 |
| Table 2.1 List of material parameters | 18 |
| Table 2.2 List of device parameters | 20 |
| Table 2.3 List of voltage input | 21 |
| Table 2.4 Simulation list..... | 23 |
| Table 2.5 List of option for speed up | 24 |
| Table 2. 6 List of smooth method parameter..... | 25 |
| Table 2.7 Scattering list..... | 26 |
| Table 2. 8 Save list | 27 |
| Table 2.9 List of option for analysis..... | 28 |
| Table 2.10 Indices for wave function | 30 |
| Table 3. 1 Indices for phonon form factor | 32 |
| Table 3.2 Meaning of “v” index for phonon form factor | 33 |
| Table 3. 3. Meaning of “v” index for SRS form factor. | 34 |
| Table 3.4 Meanings of “valley” and “types” indices for scattering table | 38 |
| Table 3.5 The description for flag_mech[types][valley] | 38 |
| Table 3.6 The description for enerAdded[types][valley] | 39 |
| Table 4.1 The meanings of 2D array p[ne][i] | 50 |
| Table 5.1 The meaning of some important variables in the SMC code | 56 |

| | |
|--|-----|
| Table 5.2 Variables to calculate number of scattering events for each type of scattering mechanisms..... | 66 |
| Table 5. 3 Some variables used in velocity_energy_cumulative_Mobility() function .. | 96 |
| Table 6.1 Some variables that have been changed so as not to have subband index ... | 108 |
| Table 6.2 Scattering parameters used for hole mobility calculation | 110 |

1. An Overview of the MSMC subroutines

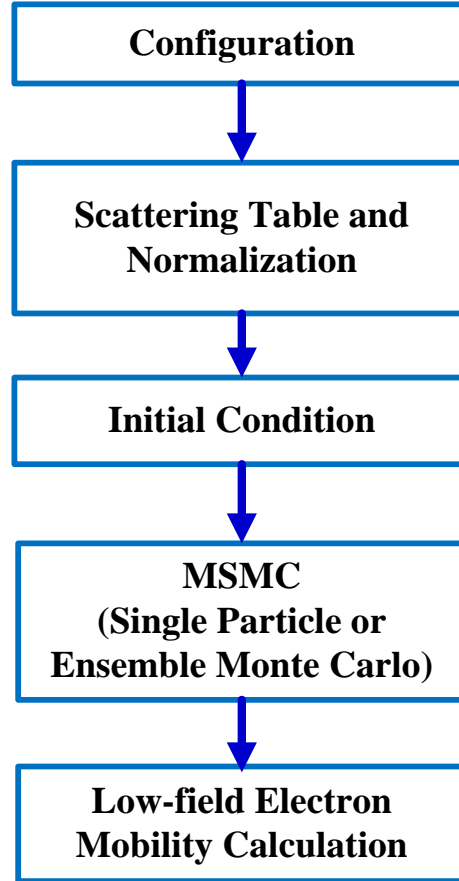


Figure 1.1 Typical flowchart of MSMC transport kernel for low-field mobility calculation

Figure 1.1 shows the typical flowchart of MSMC transport kernel for low-field electron mobility calculation in Silicon nanowire.

Firstly, all various parameters used in the simulator are determined by calling the [Configuration](#) module. Then the scattering table is constructed by using the [Scattering Table and Normalization](#) module. The next step is initialized the electron parameter such as energy and wave vector by calling the [Initial Condition](#) module. After finishing the initialization, the MSMC ([Single particle Monte Carlo \(SMC\)](#) or [Ensemble Monte Carlo \(EMC\)](#)) module is performed in which the single particle or Ensemble Monte Carlo method is used. After each free-flight time (for Single particle Monte Carlo) or each time step (for Ensemble Monte Carlo), the statistical collection is performed to record the value of the

electron parameters. Finally, the [gather data from all nodes](#) for SMC method or [velocity energy cumulative Mobility](#) for EMC approach is called to calculate velocity, energy and mobility.

1.1. Main program of Single particle MSMC code

Figure 1.2 shows an example code of *main()* function in which a Single particle Monte Carlo method is used. For a complete version of this code see [main.c](#).

```

1.  /* *****It is a MSMC code for Single Particle Monte Carlo. ***** */
2.  main(int argc, char **argv)
3.  {
4.      MPI_Init(&argc,&argv);
5.      material_param();
6.      device_structure();
7.      read_voltages_input();
8.      read_simulation_list();
9.      read_option_for_speed_up();
10.     read_parameters_for_smooth_method();
11.     read_scattering_save_list();
12.     read_option_for_ANALYSIS();
13.     int Get_NSELECT();
14.     int NSELECT_2DSchrPoisson = Get_NSELECT();
15.     initial_variables();
16.     Initialize(&argc,&argv);
17.     trapezoidal_weights();
18.     int ReadDataAtGateInput = 0;
19.     int GateInputBegin = Get_GateInputBegin();
20.     int GateInputEnd = Get_GateInputEnd();
21.     int GateInputStep = Get_GateInputStep();
22.     for( ReadDataAtGateInput = GateInputBegin; ReadDataAtGateInput <=GateInputEnd;
        ReadDataAtGateInput = ReadDataAtGateInput + GateInputStep )
23.     {
24.         double Field_start = Get_Field_start();
25.         double Field_end = Get_Field_end();
26.         double Field_step = Get_Field_step();
27.         double Field;
28.         for(Field = Field_start; Field<= Field_end; Field= Field+Field_step)
29.         {
30.             Set_Field_using(Field);
31.             double TsiInput = Get_TsiInput();
32.             double WsiInput = Get_WsiInput();
33.             read_data_for_Mobility(ReadDataAtGateInput, TsiInput, WsiInput );
34.             check_read_data_for_Mobility();// Need for checking process
35.             charge_density();
36.             int ResetNSELECT = Get_ResetNSELECT();
37.             Set_NSELECT(ResetNSELECT);
38.             form_factor_calculation();
39.             SRS_form_factor_calculation();
40.             scattering_table();
41.             if(Get_flag_Run_for_Analysis() ==1)// FOR ANALYSIS: NOT RUN MONTE CARLO
42.             {
43.                 save_form_factor_calculation();
44.                 save_form_factor_calculation_at_SomeFirstSubbands();
45.                 save_eig_at_SomeFirstSubbands();

```

```

46.         save_SRS_form_factor_calculation();
47.         save_SRS_form_factor_at_SomeFirstSubbands();
48.         save_scattering_table();
49.     } // End of if(Get_flag_Run_for_Analysis() ==1)
50. else //Run MONTE CARLO
51. {
52.     normalize_table();
53.     //save_normalized_table(); // Need when checking
54.     electrons_initialization_for_Mobility();
55.     //save_electron_parameters();
56.     save_electron_distribution(fnDistSubband,fnDistEnergy);
57.     emcd_Mobility();// Single particle MC
58.     MPI_Barrier(MPI_COMM_WORLD);
59.     gather_data_from_all_nodes();
60.     save_electron_distribution();
61. }
62.     Set_NSELECT(NSELECT_2DSchrPoisson);
63. }// End of for(Field = Field_start; Field<= Field_end;
64. }//End of for( ReadDataAtGateInput =
65. MPI_Finalize() ;
66. return 0;
67. } // End of main program

```

Figure 1.2 An example code of *main()* function using Single particle Monte Carlo method

- For Figure 1.2.
- Line 4: To establish the MPI environment.
- Line 5 to line 17: See the [Configuration](#) module.
- Line 18 to line 21: Each gate voltage value (V_g) used in 2D Schrodinger and Poisson Solver, which is the Professor Mincheol Shin's source code, has a “trick” index as indicated in the Table 1.1. By using “trick” index, it is more convenient to read data from this solver. The file **ChargevsVg.dat** gives the mapping of the “trick” index and real gate voltage value thus there is no ambiguity about that.

| | Begin | End | Step | Number of points |
|--|----------|----------|-------------|------------------|
| <i>Example 1: Gate voltage used in 2DSchroPoisson</i> | 0.46 [V] | 0.96 [V] | 0.03125 [V] | 17 |
| <i>Example 2: Gate voltage used in 2DSchroPoisson</i> | 0.38 [V] | 0.94 [V] | 0.035 [V] | 17 |
| <i>Example 3: Gate voltage used in 2DSchroPoisson</i> | 0.66 [V] | 1.46 [V] | 0.05 [V] | 17 |
| Gate voltage display in output files from 2DSchroPoisson | -0.4 | 1.2 | 0.1 | 17 |
| Gate Input used in line 18 to line 21 in Figure 1.2 | -4 | 12 | 1 | 17 |

Table 1. 1 An example of “trick” indices corresponding to real gate voltage values.

- There are two “loops” in the *main()* function, one is for varying the gate voltage (meaning that inversion charge) (See line 22), and the other is for varying lateral electric field (F_x) (See line 28). For low-field mobility calculation, we usually fix $F_x = 1\text{kV/cm}$, however, sometime at the checking process, we may want to change F_x . Furthermore, we can draw the velocity versus lateral field by using the second loop (line 28).
- Line 31 to line 40: For each set of V_g and F_x , we can calculate the scattering table. See [the Scattering table and Normalization module](#).
- Line 36 and line 37: We can reset the number of subbands for each valley (**ResetNSELECT**). For example, in 2D Schrodinger and Poisson Solver, we used 30 subbands for each valley (**NSELECT=30**). However, from MSMC transport kernel, we know that this value is much more than enough and about 15 subbands for each valley is very okay. Therefore we can reset **ResetNSELECT=15** to reduce the memory and computational time.

- Line 41 to line 49: If the option **run_for_analysis** is on, some files are saved and then the simulator is finished. See [section 2.8](#) for more details.
- Line 50 to line 61: To run MSMC transport kernel.
 - Line 52: The scattering table is normalized. See [the Scattering table and Normalization module](#).
 - Line 54: To initialize electron parameters for the first time. See [Initial Condition](#) module.
 - Line 57: A single particle Multi-subband Monte Carlo method for mobility calculation is performed. See [Single particle Monte Carlo](#) module.
 - Line 59: To collect data from all nodes and calculate the average velocity, energy and mobility. See [Section 5.1.4](#).
- Line 62: After the simulator is finished for a set of V_g and F_x , we can restore the **NSELECT** as defined in 2D Schrodinger and Poisson Solver.

1.2. Main program of an Ensemble MSMC code

Figure 1.3 shows an example code of *main()* function in which Ensemble Monte Carlo method is used. For a complete version of this code see [EMC_main.c](#). The difference between the SMC and EMC codes are for the functions that perform the SMC and EMC transport models.

```

1.  /* *****It is a MSMC code for Emsemble Monte Carlo. ***** */
2.  main(int argc, char **argv)
3.  {
4.      // This part is the same as the version for SMC code
5.      for( ReadDataAtGateInput = GateInputBegin; ReadDataAtGateInput <=GateInputEnd;
        ReadDataAtGateInput = ReadDataAtGateInput + GateInputStep )
6.      {
7.          for(Field = Field_start; Field<= Field_end; Field= Field+Field_step)
8.          {
9.              // This part is the same as the version for SMC
10.             double dt = Get_dt(); // Observation time step
11.             double tot_time = Get_tot_time();
12.             double transient_time = Get_transient_time();
13.             int iter_total=(int)(tot_time/dt+0.5); // Number of iterations
14.             double time = 0.0; // thời gian chạy của hạt
15.             int j_iter=0;
16.             int iter_reference = 1;
17.
18.             // If USING PARALLEL code
19.             while(j_iter <= iter_total) // For EACH TIME STEP
20.             {
                j_iter += 1;
            }
        }
    }

```

```

21.         time=dt*(double)(j_iter);
22.         emcd_Mobility_parallel();
23.         iter_reference=1;
24.         if(j_iter==iter_total)
25.         {
26.             iter_reference=0;
27.         }
28.         if((node == 0)&&(iter_reference==0))
29.         {
30.             velocity_energy_cumulative_Mobility_parallel();
31.         }
32.     } // End of the while(j_iter <= iter_total)
33. // End of USING PARALLEL code

34. // If Using serial code, please use the italic code
35. while(j_iter <= iter_total) // Chay tung lan cho tat ca cac iterations
36. {
37.     j_iter = j_iter + 1;
38.     time=dt*(double)(j_iter);
39.     emcd_Mobility();
40.     iter_reference=1;
41.     if(j_iter==iter_total)
42.     {
43.         iter_reference=0;
44.     }
45.     velocity_energy_cumulative_Mobility();
46. }
47. // End of If Using serial code, please use the italic code

48. } // End of for(Field = Field_start; Field<= Field_end;
49. } // End of for( ReadDataAtGateInput
50. return 0;
51. }

```

Figure 1.3 An example code of *main()* function using Ensemble Monte Carlo method

- For Figure 1.3.
- Line 4: This part is the same as the version for SMC. See from line 4 to line 21 in Figure 1.2.
- We have two “loops” for each set of V_g and F_x as indicated in line 5 and line 7
- Line 9: This part is the same as the version for SMC. See from line 30 to line 56 in Figure 1.2.
- Line 10 to line 12: To get time step, total time and transient time. See [Section 2.4.](#)
- Line 17 to line 33: The EMC parallel code for each time step Δt is performed. See [Section 5.2.5.](#)
- Line 34 to line 47: If we want to use the EMC serial code for each time step Δt , this part should be used (instead of the EMC parallel code (from line 17 to line 33)). See [Section 5.2.1 to 5.2.3.](#)

2. The Configuration module

This module includes the following functions.

```
material\_param\(\);  
device\_structure\(\);  
read\_voltages\_input\(\);  
read\_simulation\_list\(\);  
read\_option\_for\_speed\_up\(\);  
read\_parameters\_for\_smooth\_method\(\);  
read\_scattering\_save\_list\(\);  
read\_option\_for\_ANALYSIS\(\);  
initial\_variables\(\);  
Initialize(&argc,&argv);  
trapezoidal\_weights\(\);  
  
//Solve 2DSchrodiner and Poisson Solver  
line\_charge\_density\(\);  
quantum\_electron\_density\(\);  
charge\_density\(\);
```

Figure 2.1 List of functions used in the configuration module

2.1. The *material_param()* function. See [material_param.c](#)

Reading #MATERIAL_PARAMETER from Inpudeck

| Description | Name of variable | Value | Unit |
|-------------------------|------------------|-------|----------|
| Temperature | T | 300 | K |
| Electron energy maximum | emax | 0.5 | eV |
| Number of energy steps | n_lev | 10000 | Unitless |
| Band gap of Silicon | Eg | 1.12 | eV |
| Band gap of Oxide | BandgapOxide | 9.0 | eV |

| | | | |
|---------------------------------|---------------------------|---------|-------------------|
| Longitudinal effective mass | ml | 0.98 | Unitless |
| Tranverse effective mass | mt | 0.19 | Unitless |
| Nonparabolicity factor | nonparabolicity_factor | 0.5 | eV ⁻¹ |
| Constant dielectric of Silicon | eps_Si_constant | 11.9 | Unitless |
| Constant dielectric of Oxide | eps_Oxide_constant | 3.9 | Unitless |
| Intrinsic density of Silicon | intrinsic_carrier_density | 1.45e10 | cm ⁻³ |
| gateTYPE(metal_or_nPolysilicon) | gateTYPE | Metal | unitless |
| Affinity of Silicon | affinity_silicon | 4.0 | eV |
| Metal gate workfunction | gateWF | 4.56 | eV |
| Gate Workfunction Offset | GateWorkfunctionOffset | 0.56 | eV |
| Crystal density | crystal_density | 2329.0 | kgm ⁻³ |
| Sound velocity | sound_velocity | 9037.0 | ms ⁻¹ |
| Acoustic deformation potential | DefPot_acoustic | 14.6 | eV |
| f- phonon deformation potential | DefPot_f | 1.1e11 | eVm ⁻¹ |
| f- phonon energy | hw0f_phonon | 0.059 | eV |

| | | | |
|---------------------------------|--------------------|--------|-------------------|
| g- phonon deformation potential | DefPot_g | 8.0e10 | eVm ⁻¹ |
| g- phonon energy | hw0g_phonon | 0.063 | eV |
| SRS r.m.s height | rms_height | 0.3 | nm |
| SRS correlation length | correlation_length | 2.5 | nm |

Table 2.1 List of material parameters

After reading all material variables, some parameters are calculated as shown in Figure 2.2.

```

1.  Vt=kb*T/q; //thermal voltage
2.  eps_sc = eps_Si_constant*eps_0;      // dielectric of Silicon
3.  eps_oxide = eps_Oxide_constant*eps_0; // dielectric of Oxide
4.  DefPot_acoustic = DefPot_acoustic*q;  //[eV] -> [J]
5.  DefPot_f=DefPot_f*q;                  // [eV/m] -> [J/m]
6.  double w0f_phonon=hw0f_phonon*q/hbar;
7.  double nof = 1.0/(exp(hw0f_phonon/Vt)-1.0); // Number of phonon for f-
process.(Bose-Einstein distribution function)
8.  DefPot_g=DefPot_g*q;
9.  double w0g_phonon=hw0g_phonon*q/hbar;
10. double nog = 1.0/(exp(hw0g_phonon/Vt)-1.0); // Number of phonon for g-process
11. constant_acoustic=
(2.0*pi*DefPot_acoustic*DefPot_acoustic*kb*T)/(hbar*crystal_density*sound_velocity*sound_velocity);
12. constant_optical_g_e
=(pi*DefPot_g*DefPot_g)*(nog+1.0)/(crystal_density*w0g_phonon);
13. constant_optical_g_a = constant_optical_g_e*nog/(nog+1.0);
14. constant_optical_f_e =
2.0*(pi*DefPot_f*DefPot_f)*(nof+1.0)/(crystal_density*w0f_phonon);
15. constant_optical_f_a = constant_optical_f_e*nof/(nof+1.0);

```

Figure 2.2 A part of material_param() function

- For Figure 2.2
- Line 1: Thermal voltage is calculated.
- Line 2 and line 3: Dielectric of Silicon and Oxide are calculated
- Line 6 and line 7: Number of phonon for f-process is calculated.
- Line 9 and line 10: Number of phonon for g-process is calculated.
- Line 11: The **constant_acoustic** variable is the first term of the acoustic scattering rate (See [Eq. \(A.28\) – Tuan's thesis](#)).

- Line 12 to line 15: The **constant_optical_g_e**, **constant_optical_g_a**, **constant_optical_f_e** and **constant_optical_f_a** variables are the first term in the optical phonon scattering rate for f- and g- processes with emission and absorption (See [Eq. \(A.32\) – Tuan’s thesis](#)).

2.2. The *device_structure()* function. See [device_structure.c](#)

Reading # DEVICE_PARAMETER from Inpudeck

| Description | Name of variable | Value | Unit |
|--|--------------------|--------------------|-----------------------|
| <i>Device type</i> | <i>device_type</i> | <i>NanowireFET</i> | <i>Not applicable</i> |
| <i>Length of Source and Drain region</i> | <i>LsdInput</i> | <i>0.2</i> | <i>nm</i> |
| <i>Length of channel region</i> | <i>LchInput</i> | <i>0.2</i> | <i>nm</i> |
| <i>Length of gate</i> | <i>LgInput</i> | <i>0.2</i> | <i>nm</i> |
| Tox | ToxInput | 1.0 | nm |
| Tbox | TboxInput | 1.0 | nm |
| Wox | WoxInput | 1.0 | nm |
| Tsi | TsiInput | 6.0 | nm |
| Wsi | WsiInput | 6.0 | nm |
| Tgate | TgateInput | 8.0 | nm |

| | | | |
|------------------------------------|-------------------------------|------------------|------------------------|
| Wgate | WgateInput | 4.0 | nm |
| Mesh size in x-direction | mesh_size_x | 0.2 | nm |
| Mesh size in y-direction | mesh_size_y | 0.1 | nm |
| Mesh size in z-direction | mesh_size_z | 0.1 | nm |
| Number of discrete nodes for Tox | Tox_mesh | 20 | unitless |
| Number of discrete nodes for Tbox | Tbox_mesh | 20 | unitless |
| Number of discrete nodes for Wox | Wox_mesh | 20 | unitless |
| <i>Number of regions</i> | <i>n_region</i> | <i>3</i> | <i>Not applicable</i> |
| <i>Source junction doping type</i> | <i>cvar</i> | <i>N type</i> | <i>Not applicable</i> |
| <i>Doping density at Source</i> | <i>doping_densityInput[1]</i> | <i>1.0e+20</i> | <i>cm⁻³</i> |
| <i>Drain junction doping type</i> | <i>cvar</i> | <i>N type</i> | <i>Not applicable</i> |
| <i>Doping density at Drain</i> | <i>doping_densityInput[2]</i> | <i>1.0e+20</i> | <i>cm⁻³</i> |
| <i>Channel Doping Type</i> | <i>cvar</i> | <i>P type</i> | <i>Not applicable</i> |
| <i>Doping density at Channel</i> | <i>doping_densityInput[3]</i> | <i>-1.45e+10</i> | <i>cm⁻³</i> |

Table 2.2 List of device parameters

It is worth to note that if we calculate the low-field carrier mobility, the parameters for transport direction (x-direction) are not needed thus we do not need to care the *italic and blue color part in Inputdeck*.

If we use 2D Schrodinger and Poisson Solver written by Professor Mincheol Shin, the device parameters should be matched with these in the Inputdeck from this solver.

2.3. The `read_voltages_input()` function. See [read_voltages_input.c](#)

Reading #VOLTAGES_INPUT from Inputdeck

| Description | Name of variable | Value | Unit |
|---|---|----------------------|----------|
| <i>Voltage at Source</i> | <i>V_source_input</i> | <i>0.0</i> | <i>V</i> |
| <i>Drain voltage: Initial_Final_Step</i> | <i>Vd_start, Vd_end, Vd_step</i> | <i>0.0, 0.0, 0.1</i> | <i>V</i> |
| <i>Gate voltage: Initial_Final_Step</i> | <i>Vg_start, Vg_end, Vg_step</i> | <i>0.5, 0.5, 0.1</i> | <i>V</i> |
| Lateral Field: Initial_Final_Step | Field_start, Field_end, Field_step | 1.0, 1.0, 1.0 | kV/cm |
| VgInput: Initial_Final_Step (trick to run) | GateInputBegin, GateInputEnd, GateInputStep | -4, 12, 1 | Unitless |

Table 2.3 List of voltage input

In our case, we don't need to care the real values of source, drain and gate voltages in this Inputdeck (*indicated as italic and blue color in Inputdeck*) since we used 2D Schrodinger and Poisson Solver written by Professor Mincheol Shin. We only need to determine the **lateral field** (F_x), which is applied in the transport direction, and the **VgInput: Initial_Final_Step (trick to run)** parameters as described in [Section 1.1](#) (see Table 1.1).

2.4. The *read_simulation_list()* function. See [read_simulation_list.c](#)

Reading #SIMULATION_LIST from Inpudeck

| Description | Name of variable | Value | Unit |
|---|------------------------------|---------------|-----------------------------|
| <i>Solve 2Dschrodinger 2Dpoisson SelfConsistently</i> | <i>FlagPoissonSChrSolver</i> | <i>1 or 0</i> | <i>Not applicable</i> |
| <i>Relaxation_constants</i> | <i>relaxation_constant</i> | <i>0.08</i> | <i>unitless</i> |
| <i>Solve_Mobility_Calculation</i> | <i>FlagMobility</i> | <i>1 or 0</i> | <i>1 is YES 0 is NO</i> |
| dt...MC_time_step | dt | 0.2 | fs |
| Total_time | tot_time | 60.0 | ps |
| Transient time after which results are calculated | transient_time | 20.0 | ps |
| <i>Length where 3D plots in cross section are plotted</i> | <i>length_plotted</i> | <i>6.0</i> | <i>nm</i> |
| <i>Depth where the 3D plots are plotted</i> | <i>depth_plotted</i> | <i>2.0</i> | <i>nm</i> |
| <i>Width where the 3D plots are plotted</i> | <i>width_plotted</i> | <i>2.0</i> | <i>nm</i> |

| | | | |
|--|--------------------------------------|---------------|-----------------------|
| Number Valley pairs | n_valley | 3 | Not applicable |
| Number Subbands for 2D Schrodinger | NSELECT | 20 | Not applicable |
| <i>Number of time steps to Solve 2D Schrodinger Equation</i> | <i>NTimeSteps</i> | <i>10</i> | <i>Not applicable</i> |
| <i>Artificial factor of cell volume</i> | <i>artificial_factor_cell_volume</i> | <i>3.0e-6</i> | <i>Not applicable</i> |
| Total number of particles for MOBILITY | inum | 200000 | Not applicable |

Table 2.4 Simulation list

For low-field mobility calculation, we do not need to care the *italic and blue color part in Inputdeck*.

If the **FlagPoissonSChrSolver** is **ON**, the 2D Schrodinger and Poisson Solver developed by Tuan is run, however, for a big cross-section it is very slow.

- For Table 2.4
- Time step **dt** is used only for the Ensemble Monte Carlo method
- **tot_time** is the simulation time.
- **transient_time** is the pre-defined time after which the quantities of interest are calculated. See [Figure 4.6-Tuan's thesis](#).
- **n_valley**: number of valley. it is equal to three for electron transport.
- **NSELECT** is the number of subbands for each valley.
- **inum** is the total particles used in the simulator.

2.5 *read_option_for_speed_up()* function. See [read_option_for_speed_up.c](#)

Reading #OPTION_FOR_SPEED_UP from Inputdeck

| Description | Name of variable | Value |
|---|---------------------------------|----------------------|
| Write Velocity vs. Time | flag_write_Velocity_vs_Time | 1 (YES) or 0 (NO) |
| Using PARABOLIC BAND (in transport direction) | flag_using_PARABOLIC_Band | 1 or 0 |
| Calculate average velocity for All Time Steps | flag_calculate_VeloAllTimeSteps | 1 or 0 |
| Reset number of Subbands at each valley in MSMC (Reset_NSELECT) | ResetNSELECT | 12 |

Table 2.5 List of option for speed up

- **Write Velocity vs. Time:** It is needed at the first stage of code development to make sure that the simulator run well, however, it is time consuming.
- **Using PARABOLIC BAND:** We can used parabolic band by reset the non-parabolic factor is equal to zero.
- **Calculate average velocity for All Time Steps:** The average velocity and energy are calcualted for all time (including the transient time).
- **Reset number of Subbands at each valley:** We can reset the number of subband getting from 2D Schrodinger and Poisson Solver to reduce memory for storing the scattering table and computational time. Of couse, **ResetNSELECT** variable should be smaller or equal to **NSELECT**.

2.6 The *read_parameters_for_smooth_method()* function. See [read_parameters_for_smooth_method.c](#)

Reading #SMOOTH_METHOD_PARAMETER_LIST from Inputdeck

| Description | Name of variable | Value |
|--|-----------------------------|----------|
| Number of points to use as first point for transient | NumTransientPoints | 2 |
| <i>Step of transient points (1_means_1dt,2_2dt)</i> | <i>StepOfTransientPoint</i> | <i>1</i> |
| Length of transient point (Multiply by 1picosecond) | LengthTransPoint | 0.0001 |

Table 2. 6 List of smooth method parameter

There is no rule to choose the first point (t_0) as a transient point. We usually choose t_0 when electrons are in the steady state (See [Fig.4.6 – Tuan’s thesis](#)). It is a little bit more accurate if we choose several points as the transient points (See Figure 5.4).

- **E.g.** For SMC code, **NumTransientPoints = 2** and **LengthTransPoint = 0.0001 ps**: we choose two point t_0 and t_1 as transient points and the time difference between them is 0.0001 ps.
- For EMC code, we use **StepOfTransientPoints** instead of **LengthTransPoint**. The **StepOfTransientPoints** is multiple of time step Δt . E.g. **StepOfTransientPoints = 2**, meaning that the time difference between two adjacent points is $2 \cdot \Delta t$.

It should be noted that if we choose number of particles (**inum**) and total simulation time (**tot_time**) are big enough, the ensemble average of velocity and energy almost do not depend on the t_0 . In this situation, however, this method is also very good to understand the Monte Carlo method.

2.7 The `read_scattering_save_list()` function. See [read_scattering_save_list.c](#)

Reading #SCATTERING_LIST and #SAVE_LIST from Inputdeck.

#SCATTERING_LIST

| Description | Name of variable | Value |
|---|---------------------------------|--------------------------------|
| Ballistic or Diffusive transport | flag_ballistic_transport | 1 (ballistic) or 0 (diffusive) |
| Acoustic scattering | flag_acoustic | 1 or 0 |
| Zero Order Non Polar Optical Phonon | flag_zero_order_optical | 1 or 0 |
| <i>First Order Non Polar Optical Phonon</i> | <i>flag_first_order_optical</i> | <i>0</i> |
| <i>Coulomb scattering</i> | <i>flag_Coulomb</i> | <i>0</i> |
| Surface roughness scattering | flag_Surface_roughness | 1 or 0 |
| SRS Model 1 (IEEE2006) or 2 (JAP(2008)) | flag_SRS_model | 1 or 2 |

Table 2.7 Scattering list

- If **flag_ballistic_transport = 1**: The carriers will transport without experiencing any scatterings (Note that here is classical ballistic transport).
- If **flag_acoustic = 1**: The acoustic scattering will be included in the model, otherwise (**flag_acoustic = 0**), it is not included in the model. The same manner for **flag_zero_order_optical** and **flag_Surface_roughness**.
- At this time coulomb scattering and first order non polar optical phonon are still not included in the source code. Thus just setting their flags to zero.
- If SRS scattering is included in the model (**flag_Surface_roughness=1**), we can choose two methods for calculating the SRS form factor. For Model 1 and 2, please see references “*Ramayya, et al., IEEE Trans. Nanotechnol. 6, 113, (2006)*” and “*Ramayya, et al., J. Appl. Phys., 104, p.063711 (2008)*”, respectively.

#SAVE_LIST

| Description | Name of variable | Value |
|--|---|-----------------------------|
| <i>Save_eig_wavefunction</i> | <i>save_eig_wavefunction</i> | <i>1(YES) or 0 (NO)</i> |
| <i>Save_doping_potential_initialization</i> | <i>save_doping_potential_init</i> | <i>1 or 0</i> |
| <i>Save_electron_initlization_distribution</i> | <i>save_electron_initlization</i> | <i>1 or 0</i> |
| <i>Save_electron_population</i> | <i>save_electron_population</i> | <i>1 or 0</i> |
| <i>Save_electron_Density</i> | <i>save_electron_density</i> | <i>1 or 0</i> |
| <i>Save_velo_ener_curr_cumu_all_time_steps</i> | <i>save_VelEnerCurr_all_time_steps</i> | <i>1 or 0</i> |
| <i>Save_velo_ener_curr_cumu_after_transie</i> | <i>save_VelEnerCurr_after_transient</i> | <i>1 or 0</i> |
| <i>Save_init_free_flight</i> | <i>save_init_free_flight</i> | <i>1 or 0</i> |

Table 2. 8 Save list

This SAVE_LIST is NOT used now.

2.8 The *read_option_for_ANALYSIS()* function. See [read_option_for_ANALYSIS.c](#)

Reading #OPTION_FOR_ANALYSIS from Inputdeck.

| Description | Name of variable | Value |
|---|---|---|
| Run for Analysis | flag_Run_for_Analysis | 1 (YES) or 0 (NO) |
| Save scattering table | flag_save_scattering_table | 1 (YES) or 0 (NO) |
| (save scattering table) from_nth_subband | flag_nth_subband | 1 (Eg. 1 meaning first subband) |
| For SRS save scat table at VALLEY | flag_For_SRS_save _scat_table_at_VALLEY | 1 or 2 or 3 (for valley 1 or 2 or 3) |
| Save PHONON formfactor | flag_save_PHONON_formfactor | 1 (YES) or 0 (NO) |
| Save PHONON forfactor SomeFirstSub | flag_save_PHONON_forfactor _SomeFirstSub | 1 (YES) or 0 (NO) |
| Save eig SomeFirstSub | flag_save_eig_SomeFirstSub | 1 (YES) or 0 (NO) |
| Save SRS formfactor | flag_save_SRS_formfactor | 1 (YES) or 0 (NO) |
| Save SRS formfactor SomeFirstSub | flag_save_SRS_formfactor _SomeFirstSub | 1 (YES) or 0 (NO) |

Table 2.9 List of option for analysis

- If **flag_Run_for_Analysis = 1**: The program will run for analysis (not run Monte Carlo). In coding development, it is very important to make sure that some critical parts are correct before running the simulator to have final results.

- If **flag_save_scattering_table=1**, the function [save_scattering_table.c](#) is called and then the scattering table for n^{th} subband will be save in a file. The subband n^{th} is defined in **flag_nth_subband** (For example, **flag_nth_subband=1**, the scattering table for first subband is saved).
- If **flag_For_SRS_save_scat_table_at_VALLEY=1**, the surface roughness scattering at valley 1 is saved, and similarly for valley 2 and 3.
- If **flag_save_PHONON_formfactor = 1**, the function [save_form_factor_calculation.c](#) is called and then phonon form factor at particular gate voltage is saved.
- If **flag_save_PHONON_formfactor_SomeFirstSub = 1**, the function [save_form_factor_calculation_at_SomeFirstSubbands.c](#) is called and then phonon form factors at some first subbands with different gate voltages are saved.
- If **flag_save_eig_SomeFirstSub = 1**, the function [save_eig_at_SomeFirstSubbands.c](#) is called and then the eigen energies at some first subbands with different gate voltages are saved.
- If **flag_save_SRS_formfactor = 1**, the function [save_SRS_form_factor_calculation.c](#) is called and then the SRS form factor factor for Top, Bottom, Right and Left interfaces at particular gate voltage is saved.
- If **flag_save_SRS_formfactor_SomeFirstSub = 1**, the function [save_SRS_form_factor_at_SomeFirstSubbands.c](#) is called and then the SRS form factors at some first subbands for Top, Bottom, Right and Left interfaces with different gate voltages are saved.

2.9 The *initial_variables()* function. See [initial_variables.c](#)

It is better, in my opinion, if at the begining of the program, all global variables are inilized to have values (usually zero).

In this source code, the random number is generated by using function [random2\(long *idum\)](#) (See [ran2.c](#) – it is a code from “Numerical Recipes in C” book). The **idum** is initilized by the function [void init_idum\(\)](#) to have different values for each processor at every running time.

2.10 The *trapezoidal_weights()* function. See [trapezoidal_weights.c](#)

In this source code the double integrals is calculated by using “double trapezoid” rule. See, for example, lecture 24 in “[Introduction to Numerical Methods and Matlab Programming for Engineers by Todd Young and Martin J. Mohlenkamp - Ohio University – 2010](#)”. This ebook is included in the Reference folder.

2.11. 2D Schrodinger and 2D Poisson Solver.

At the first stage, I have used 2D Schrodinger and 2D Poisson Solver written by myself. It is, however, too slow for a big cross-section such as $10 \times 10 \text{ nm}^2$, thus I used the Solver written by Professor Mincheol Shin. If someone wants to develop a small simulator from the beginning, however, this program is maybe a good starting point to learn something. See [poisson_SOR.c](#), [Solved 2D Schro for MSMC.c](#), and [Chapter 2 – Tuan’s thesis](#) for detail.

The outputs of the solver are wave function, eigen energy, confining potential, and line charge density.

- Wave function is 5D array **wave[s][v][i][j][k]**

| Index | Description |
|----------|--|
| s | Indicating s-th section in the transport direction. For low-field mobility calculation, only one section is considered thus s=0 . |
| v | Indicating valley index (v=1 to n_valley). For conduction band of Silicon n_valley= 3 |
| i | Indicating number of subbands for each valley (i=1 to NSELECT) |
| j | Indicating number of points in y-direction in the Silicon domain |
| k | Indicating number of points in z-direction in the Silicon domain |

Table 2.10 Indices for wave function

- Eigen energy is 3D array **eig[s][v][i]**. The meanings of “s”, “v” and “i” indices are indicated in Table 2.10.
- Line Density and Percentage Population are 2D arrays. **LineDensity[v][i]** and **PercentagePopulation[v][i]**. The meanings of “v” and “i” indices are indicated in Table 2.10.
- Confining potential is 2D array **fai[j][k]**. The meanings of “j” and “k” indices are indicated in Table 2.10.

2.12. The *line_charge_density()* function. See [line_charge_density.c](#)

Electron line density in valley **v** and subband **i**, (**N_{v,i}**), is calculated by using [Equation 2.3 \(Tuan’s thesis\)](#) and the percentage population for each subband (**P_{v,i}**) is given by

$$P_{v,i} = \frac{N_{v,i}}{\sum_{v=1}^{n_valley} \sum_{i=1}^{NSELECT} N_{v,i}} \quad (2.1)$$

2.13. The *quantum_electron_density()* function. See [quantum_electron_density.c](#)

Quantum electron density is calculated using [Equation 2.2 \(Tuan’s thesis\)](#).

2.14. The *charge_density()* function. See [charge_density.c](#)

Charge density (**N_{inv}**) is calculated using [Equation 2.18 \(Tuan’s thesis\)](#).

3. The Scattering table and Normalization module

This module is included the following functions.

```
read\_data\_for\_Mobility\(ReadDataAtGateInput, TsilInput, WsilInput \);  
check\_read\_data\_for\_Mobility\(\); // For checking process only  
form\_factor\_calculation\(\); // Phonon formfactor  
SRS\_form\_factor\_calculation\(\); // Surface roughness form_factor  
scattering\_table\(\);  
normalize\_table\(\);
```

Figure 3.1 List of functions used in the scattering tables and normalization module

3.1. The *read_data_for_Mobility()* function. See [read_data_for_Mobility.c](#)

This function is to get wave function, eigen energy, confining potential, and line charge density obtained from 2D Schrodinger and 2D Poisson Solver.

In the checking process, we may want to check the results that are read from 2D Schrodinger and Poisson Solver. The function [check_read_data_for_Mobility\(\)](#) is created for this purpose.

3.2. The *form_factor_calculation()* function. See [form_factor_calculation.c](#)

- Phonon form factor is 4D array **form_factor[s][n][m][v]**

| Index | Description |
|----------|---|
| s | Indicating s-th section. For low-field mobility, only one section is considered thus s=0 |
| n | Indicating the scattering from n-th subband (n=1 to NSELECT) to m-th subband |
| m | (m=1 to NSELECT) |
| v | Indicating type of scattering mechanisms. The value of v and its meaning are given in the Table 3.2. |

Table 3. 1 Indices for phonon form factor

| Scattering mechanism | v | Description |
|---|----------|---------------------------|
| Acoustic (Intra-valley scattering) | 1 | At valley 1 |
| | 2 | At valley 2 |
| | 3 | At valley 3 |
| Non-polar Optical phonon (Inter-valley scattering) | 4 | From valley 1 to valley 2 |
| | 5 | From valley 1 to valley 3 |
| | 6 | From valley 2 to valley 1 |
| | 7 | From valley 2 to valley 3 |
| | 8 | From valley 3 to valley 1 |
| | 9 | From valley 3 to valley 2 |

Table 3.2 Meaning of “v” index for phonon form factor

Phonon form factor is calculated using [Equation A.25 \(Tuan’s thesis\)](#).

3.3. The *SRS_form_factor_calculation()* function. See [SRS form factor calculation.c](#)

- Surface roughness form factor is 4D array **SRS_form_factor[s][n][m][v]**. The meanings of “s”, “n” and “m” indices are given in the Table 3.1. The meaning of “v” index is given in the Table 3.3.

| SRS scattering (Intra-valley) | v index | Description |
|--------------------------------------|----------------|--------------------|
| Top interface | 1 | At valley 1 |
| | 2 | At valley 2 |
| | 3 | At valley 3 |
| Bottom interface | 4 | At valley 1 |
| | 5 | At valley 2 |
| | 6 | At valley 3 |
| Right side interface | 7 | At valley 1 |
| | 8 | At valley 2 |
| | 9 | At valley 3 |
| Left side interface | 10 | At valley 1 |
| | 11 | At valley 2 |
| | 12 | At valley 3 |

Table 3. 3. Meaning of “v” index for SRS form factor.

It is worth to note that the [Equation \(3.22\) \(Tuan’s thesis\)](#) for SRS form factor has a mistake. It is corrected, as below.

$$F_{ii'}^{SRS} = \int_0^{T_{Si}} \int_0^{W_{Si}} \left[-\frac{\hbar^2}{eT_{Si}m_y^v} \Psi_{v,i}(y,z) \frac{\partial^2 \Psi_{v,i'}(y,z)}{\partial y^2} + \Psi_{v,i}(y,z) \varepsilon_y(y,z) \left(1 - \frac{y}{T_{Si}}\right) \Psi_{v,i'}(y,z) \right. \\ \left. + \Psi_{v,i}(y,z) \left(\frac{E_{v,i} - E_{v,i'}}{e} \right) \left(1 - \frac{y}{T_{Si}}\right) \frac{\partial \Psi_{v,i'}(y,z)}{\partial y} \right] dy dz, \quad (3.1)$$

Numerical differentiation: The “central first derivative” and “three-point formular” methods are used to calculate the first and second derivatives, respectively. See Chapter 3 in the book [*“COMPUTATIONAL PHYSICS by M. Hjorth-Jensen – 2003”*](#) for detail.

3.4. The *Scattering_table()* function. See [scattering_table_Mobility.c](#).

This function is included the following functions [AC_scattering_table\(\)](#), [OP_scattering_table\(\)](#) and [SRS_scattering_table\(\)](#).

The scattering table is 5D array **scat_table [valley][n][m][e_step][types]**, the meaning of the “**n**” and “**m**” indices are given in [Table 3.1](#). and the “**e_step**” index (energy step) is in the range **1** to **n_lev**. The meanings of “**valley**” and “**types**” indices are given in the Table 3.4. Note: in case of a fully self-consistent solution of the transport properties such as current-voltage characteristics, the “*section-index*” must be included in the scattering table.

FW: Forward process. **BW:** Backward process

| Scattering mechanism | valley index | | types index | |
|-----------------------------------|--------------|-------------|-------------|-------------------------|
| | Value | Description | Value | Description |
| Acoustic phonon (intra-valley) | 1 | At valley 1 | 1 | Scatterings at valley 1 |
| | 2 | At valley 2 | 1 | Scatterings at valley 2 |
| | 3 | At valley 3 | 1 | Scatterings at valley 3 |

| | | | | |
|---|---|-----------------------------|---|--------------------------------|
| Non-polar Optical Phonon Scattering (Inter-valley) | 1 | Scattering from valley 1 | 2 | To valley 2 with f-Absorption |
| | | | 3 | To valley 2 with f-Emission |
| | | | 4 | To valley 3 with f-Absorption |
| | | | 5 | To valley 3 with f-Emission |
| | | | 6 | To valley 1' with g-Absorption |
| | | | 7 | To valley 1' with g-Emission |
| | 2 | Scattering from valley 2 | 2 | To valley 1 with f-Absorption |
| | | | 3 | To valley 1 with f-Emission |
| | | | 4 | To valley 3 with f-Absorption |
| | | | 5 | To valley 3 with f-Emission |
| | | | 6 | To valley 2' with g-Absorption |
| | | | 7 | To valley 2' with g-Emission |
| | 3 | Scattering from valley 3 | 2 | To valley 1 with f-Absorption |
| | | | 3 | To valley 1 with f-Emission |
| | | | 4 | To valley 2 with f-Absorption |
| | | | 5 | To valley 2 with f-Emission |
| | | | 6 | To valley 3' with g-Absorption |
| | | | 7 | To valley 3' with g-Emission |

| | | | | |
|--|---|---------------------------|----|-----------------------------|
| Surface Roughness Scattering (Intra-valley) | 1 | Scattering at valley 1 | 8 | At Top interface with FW |
| | | | 9 | At Top interface with BW |
| | | | 10 | At Bottom interface with FW |
| | | | 11 | At Bottom interface with BW |
| | | | 12 | At Right interface with FW |
| | | | 13 | At Right interface with BW |
| | | | 14 | At Left interface with FW |
| | | | 15 | At Left interface with BW |
| | 2 | Scattering at valley 2 | 8 | At Top interface with FW |
| | | | 9 | At Top interface with BW |
| | | | 10 | At Bottom interface with FW |
| | | | 11 | At Bottom interface with BW |
| | | | 12 | At Right interface with FW |
| | | | 13 | At Right interface with BW |
| | | | 14 | At Left interface with FW |
| | | | 15 | At Left interface with BW |
| | 3 | Scattering at valley 3 | 8 | At Top interface with FW |
| | | | 9 | At Top interface with BW |

| | | | | |
|--|--|--|----|-----------------------------|
| | | | 10 | At Bottom interface with FW |
| | | | 11 | At Bottom interface with BW |
| | | | 12 | At Right interface with FW |
| | | | 13 | At Right interface with BW |
| | | | 14 | At Left interface with FW |
| | | | 15 | At Left interface with BW |

Table 3.4 Meanings of “valley” and “types” indices for scattering table

There are three important variables for making scattering table and deciding the state after scattering.

- **flag_mech[types][valley]:** The “types” index is indicated in the Table 3.4 and the “valley” index is the valley where carrier resides before scattering (**valley** = 1 or 2 or 3). The description of **flag_mech[types][valley]** is shown in Table 3.5

| | Value | Meaning |
|--------------------------------------|-------|----------------------|
| If flag_mech[types][valley] = | 1 | Isotropic scattering |
| | 2 | SRS FW |
| | 3 | SRS BW |

Table 3.5 The description for **flag_mech[types][valley]**

- **i_valley[types][valley]:** The meanings of “types” and “valley” indices are the same as described for **flag_mech[types][valley]**. The **i_valley[types][valley]** will store value of valley after scattering, thus **i_valley[types][valley]** = 1 or 2 or 3.

- **enerAdded[types][valley]**: The meanings of “**types**” and “**valley**” indices are the same as described for **flag_mech[types][valley]**. The energy gain or lost after scattering depend on the scattering mechanism, as shown in Table 3.6.

| | Value | If |
|-----------------------------------|----------------------|---------------------------------|
| enerAdded[types][valley] = | 0.0 | Elastic scattering |
| | $\pm \hbar \omega_f$ | Inelastic with f-process |
| | $\pm \hbar \omega_g$ | Inelastic with g-process |

Table 3.6 The description for **enerAdded[types][valley]**

The size of scattering table depends on how many scattering mechanisms included in the model. The maximum value of “**types**” index is defined by the **NscatTypes** variable. **NscatTypes** is equal to **1** (Only acoustic phonon), **7** (Acoustic and Non-polar optical phonon) and **15** (Acoustic phonon, Non-polar Optical phonon and SRS scatterings).

NOTE: If the spectra of the four interfaces are uncorrelated, then the square scattering matrix elements can be given by

$$\left| H_{i,i'}^{SRS(TOTAL)}(\mathbf{k}_x, \mathbf{k}'_x) \right|^2 = \left| H_{i,i'}^{SRS(TOP)}(\mathbf{k}_x, \mathbf{k}'_x) \right|^2 + \left| H_{i,i'}^{SRS(BOTTOM)}(\mathbf{k}_x, \mathbf{k}'_x) \right|^2 + \left| H_{i,i'}^{SRS(RIGHT)}(\mathbf{k}_x, \mathbf{k}'_x) \right|^2 + \left| H_{i,i'}^{SRS(LEFT)}(\mathbf{k}_x, \mathbf{k}'_x) \right|^2 \quad (3.2)$$

where $\left| H_{i,i'}^{SRS(TOP)}(\mathbf{k}_x, \mathbf{k}'_x) \right|^2$ is given by the [Eq. \(3.23\) \(Tuan's thesis\)](#).

3.4.1. The *AC_scattering_table()* function. See [AC_scattering_table.c](#)

This function is to calculate acoustic scattering table. An example code is shown in Figure 3.2. The reader who is familiar with C code can skip this section.


```

1. #include "mpi.h"
2. #include <stdio.h>
3. #include <math.h>
4. #include "nrutil.h"
5. #include "constants.h"
6. #include "functions.h"
7.
8. void AC_scattering_table(){
9.
10.  double *****Get_scat_table(), *****Get_form_factor(), ***Get_eig();
11.  double *****scat_table = Get_scat_table();
12.  double *****form_factor = Get_form_factor();
13.  double ***eig = Get_eig();
14.
15.  double **Get_enerAdded();
16.  double **enerAdded = Get_enerAdded();
17.  int **Get_flag_mech(),**Get_i_valley();
18.  int **flag_mech = Get_flag_mech();
19.  int **i_valley = Get_i_valley();
20.
21.  double Get_ml(),Get_mt(),Get_emax();
22.  double ml = Get_ml();
23.  double mt = Get_mt();
24.  double emax = Get_emax();
25.
26.  int Get_NSELECT(), Get_n_valley(),Get_NScatTypes(),Get_n_lev();
27.  int NSELECT = Get_NSELECT();
28.  int n_valley = Get_n_valley();
29.  int NScatTypes = Get_NScatTypes();
30.  int n_lev = Get_n_lev();
31.
32.  double Get_nonparabolicity_factor(), Get_constant_acoustic();
33.  double af = Get_nonparabolicity_factor();
34.  double constant_acoustic = Get_constant_acoustic();
35.
36.  int valley=0,types=0,n=0,m=0,e_step=0;
37.  int ValleyBefore=0, ValleyAfter=0;
38.
39.  double de=emax/(double)(n_lev); // energy interval
40.  double *E;
41.  E=dvector(1,n_lev); // range of energy level
42.  for(e_step=1; e_step<=n_lev; e_step++)
43.  {
44.      E[e_step] = e_step*de;
45.  }
46.  double Ef = 0.0;
47.  double theta = 0.0;
48.  double DOS_temp = 0.0;
49.

```

```

50. double const_DOS_ml = 0.0, const_DOS_mt=0.0; // First term of
    Eq.(A.26) - Tuan's thesis
51. const_DOS_ml = sqrt(ml*m0)/(sqrt(2.0)*pi*hbar*sqrt(q));
52. const_DOS_mt = sqrt(mt*m0)/(sqrt(2.0)*pi*hbar*sqrt(q));
53.
54. types = 1; // acoustic. See Table 3.4 - Handbook for MSMC
55.
56. // valley pair 1, m* = ml. See Table 2.1 - Tuan's thesis
57. ValleyBefore = 1;
58. ValleyAfter = ValleyBefore;
59. for(n=1; n<=NSELECT; n++){ // Scattering from subband n-th
60.     for(m=1; m<=NSELECT; m++){ // to subband m-th
61.         for(e_step=1; e_step<=n_lev; e_step++){
62.             Ef = eig[0][ValleyBefore][n] - eig[0][ValleyAfter][m] +
                E[e_step]; // [eV] for Mobility tai section 0
63.
64.             if(Ef > 0.0) { // step function -> Ef<0: theta=0; Ef>0: theta=1
65.                 theta = 1.0;
66.                 DOS_temp =
                    const_DOS_ml*theta*((1+2.0*af*Ef)/sqrt(Ef*(1+af*Ef)));
67.                 scat_table[ValleyBefore][n][m][e_step][types] =
                    constant_acoustic*form_factor[0][n][m][1]*DOS_temp;
68.             }
69.             else { // Ef <=0.0 then // theta =0.0; DOS_temp = 0.0
70.                 scat_table[ValleyBefore][n][m][e_step][types] = 0.0;
71.             }
72.         }
73.     }
74. }
75.
76. flag_mech[types][ValleyBefore] = 1; // Isotropic
77. i_valley[types][ValleyBefore] = ValleyAfter;
78. enerAdded[types][ValleyBefore] = 0.0; // Elastic
79.
80. //m* = mt: valley pair 2 and 3. See Table 2.1 - Tuan thesis
81. for(valley=2; valley<=3; valley++){
82.     ValleyBefore = valley; // 2 or 3
83.     ValleyAfter = ValleyBefore;
84.     for(n=1; n<=NSELECT; n++){
85.         for(m=1; m<=NSELECT; m++){
86.             for(e_step=1; e_step<=n_lev; e_step++){
87.                 Ef = eig[0][ValleyBefore][n] - eig[0][ValleyAfter][m] +
                    E[e_step]; // [eV]
88.
89.                 if(Ef > 0.0){
90.                     theta = 1.0;
91.                     DOS_temp =
                        const_DOS_mt*theta*((1+2.0*af*Ef)/sqrt(Ef*(1+af*Ef)));

```

```

92.         scat_table[ValleyBefore][n][m][e_step][types] =
           constant_acoustic*form_factor[0][n][m][valley]*DOS_temp;
93.     }
94.     else{
95.         scat_table[ValleyBefore][n][m][e_step][types] = 0.0;
96.     }
97. }
98. }
99. }
100.
101.     flag_mech[types][ValleyBefore] = 1;
102.     i_valley[types][ValleyBefore] = ValleyAfter;
103.     enerAdded[types][ValleyBefore] = 0.0;
104. }
105.
106.     free_dvector(E,1,n_lev); // Free local variables
107.     return;
108. }// End of void AC_scattering_table()

```

Figure 3.2 An example source code for making acoustic phonon scattering table

- For Figure 3.2.
- Line 1 to line 6: The standard library header files such as **stdio.h** and **math.h** and the user-defined header files such as **nrutil.h** and **constants.h** are declared.
- Line 10 to line 19: To get arrays that will be used in this function such as **scat_table** and **form_factor**.
- Line 21 to line 34: To get values of parameters and constants.
- Line 39 to line 45: To define energy range for making scattering table.
- Line 50 to line 53: The first term in [Eq. \(A.26\) – Tuan’s thesis](#) for valley pair 1 with effective mass (m_l) and valley pair 2 and 3 with effective masses (m_t) are calculated.
NOTE: For low-field mobility calculation, the scattering table is calculated only one time, however, for a fully self-consistent transport calculation, the scattering table is calculated many times. Since these parameters are not changed in the self-consistent calculation, it is better to calculate them outside this function.
- Line 54: The “**types**” index is equal to **1**. Because we are making acoustic scattering table. See Table 3.4.
- Line 57 to line 78: Making acoustic scattering table for valley pair 1
 - Line 57: **ValleyBefore** = 1;
ValleyBefore is the valley in which an electron is resided before scattering.

- Line 58: **ValleyAfter = ValleyBefore;**
ValleyAfter is the valley in which the electron is resided after scattering.
 Acoustic phonon scattering is intra-valley scattering.
- Line 62: To calculate E_f (the kinetic energy of the electron after scattering). [See Eq. \(A.27\) – Tuan’s thesis.](#)
- Line 64 to line 68: If ($E_f > 0$): calculating the density of state (DOS) and then scattering rate. [See Eq. \(A.26\) and \(A.28\) – Tuan’s thesis](#)
 NOTE: For clarify, I have set **theta = 1.0**; but it does not need any more.
- Line 76 to 78: To set values for **flag_mech**, **i_valley** and **enerAdded** as discussed above in [Section 3.4.](#)
- Line 81 to line 104: Calculating the acoustic scattering table for valley 2 and 3. It is the same way as described above.

3.4.2 The *OP_scattering_table()* function. See [OP_scattering_table.c](#)

The method for making non-polar optical phonon scattering table is similar to the acoustic scattering table as described in [Section 3.4.1.](#)

```

1. // Scattering from valley 1 to valley 2
2. ValleyBefore = 1;
3. ValleyAfter = 2;
4. // f-process ABSORPTION
5. types = 2; // See Table 3.4
6. for(n=1; n<=NSELECT; n++){
7.   for(m=1; m<=NSELECT; m++){
8.     for(e_step=1; e_step<=n_lev; e_step++){
9.       Ef = eig[0][ValleyBefore][n] - eig[0][ValleyAfter][m] + E[e_step] +
         hw0f_phonon;
10.      if(Ef > 0.0) {
11.        theta = 1.0;
12.        DOS_temp = const_DOS_mt*theta*((1+2.0*af*Ef)/sqrt(Ef*(1+af*Ef)));
13.        scat_table[ValleyBefore][n][m][e_step][types] =
          constant_optical_f_a*form_factor[0][n][m][4]*DOS_temp;
14.      }
15.      else {
16.        scat_table[ValleyBefore][n][m][e_step][types] = 0.0;
17.      }
18.    }
19.  }
20. }
21. flag_mech[types][ValleyBefore] = 1;

```

```

22. i_valley[types][ValleyBefore] = ValleyAfter;
23. enerAdded[types][ValleyBefore] = hw0f_phonon;

```

Figure 3.3 An example source code for making optical phonon scattering table

The Figure 3.3 shows an example code for making optical phonon scattering table (from valley 1 to valley 2 for f-process with absorption).

- For Figure 3.3.
- Line 2 and line 3: Meaning that the scattering is from valley 1 to valley 2.
- Line 5: **types=2**. Because the **types** index corresponding to this mechanism is equal to 2. See the Table 3.4.
- Line 9: Calculating E_f (the kinetic energy of the electron after scattering). [See Eq. \(A.35\) – Tuan’s thesis](#).
- Line 13: The last index for **form_factor** is equal to 4. See the Table 3.2.
- Line 23: in this case, it is f-process with absorption so the **enerAdded** has a value **hw0f_phonon**.

NOTE: The way to make other types of this scattering mechanism is trivial and straightforward.

3.4.3. The *SRS_scattering_table()* function. See [SRS_scattering_table.c](#)

The SRS scattering rate is given by the [Eq. \(3.25\) – Tuan’s thesis](#). We write it here for clarity.

$$\begin{aligned}
\Gamma_{ii'}^{SRS}(k_x, \pm) &= \frac{4\sqrt{2}\pi e^2}{\hbar} \frac{\Delta^2 \Lambda}{2 + (q_x^\pm)^2 \Lambda^2} |F_{ii'}^{SRS}|^2 D_{1D} \Theta(E_f) \\
&= \frac{2\sqrt{m^*} e^2}{\hbar^2} \frac{\Delta^2 \Lambda}{2 + (q_x^\pm)^2 \Lambda^2} |F_{ii'}^{SRS}|^2 \frac{(1 + 2\alpha E_f)}{\sqrt{E_f (1 + \alpha E_f)}} \Theta(E_f)
\end{aligned} \tag{3.3}$$

where

$$(q_x^\pm)^2 = (k_x \pm k'_x)^2 = \frac{2m^*}{\hbar^2} \left(\sqrt{E_f (1 + \alpha E_f)} \pm \sqrt{E_i (1 + \alpha E_i)} \right)^2 \tag{3.4}$$

where E_i and E_f are the kinetic energy of the electron before and after scattering, the upper and lower signs are for backward and forward processes, respectively.

```

1. // At valley 1, Top Interface, Forward process
2. ValleyBefore = 1;
3. ValleyAfter = ValleyBefore;
4. types = 8;
5. for(n=1; n<=NSELECT; n++){
6.   for(m=1; m<=NSELECT; m++){
7.     for(e_step=1; e_step<=n_lev; e_step++){
8.       Ef = eig[0][ValleyBefore][n] - eig[0][ValleyAfter][m] + E[e_step];
9.       if(Ef > 0.0) {
10.        theta = 1.0;
11.        DOS_temp = theta*((1+2.0*af*Ef)/sqrt(Ef*(1+af*Ef)));
12.        SumFW = sqrt(Ef*(1.0+af*Ef)) -
sqrt(E[e_step]*(1.0+af*E[e_step]));
13.        SquareqxFW = 2.0*q*(m1*m0)/(hbar*hbar)*SumFW*SumFW;
14.        scat_table[ValleyBefore][n][m][e_step][types] =
SRS_constant_m1/(2.0+SquareqxFW*Square_correlation_length)*DOS_temp*SRS_
form_factor[0][n][m][1];
15.      }
16. else {
17.       scat_table[ValleyBefore][n][m][e_step][types] = 0.0;
18.     }
19.   }
20. }
21. }
22. flag_mech[types][ValleyBefore] = 2;
23. i_valley[types][ValleyBefore] = ValleyAfter;
24. enerAdded[types][ValleyBefore] = 0.0;
25.
26. // In valley 1, at Top interface with Backward process
27. types = 9;
28. for(n=1; n<=NSELECT; n++){
29.   for(m=1; m<=NSELECT; m++){
30.     for(e_step=1; e_step<=n_lev; e_step++){
31.       Ef = eig[0][ValleyBefore][n] - eig[0][ValleyAfter][m] + E[e_step];
32.       if(Ef > 0.0) {
33.        theta = 1.0;
34.        DOS_temp = theta*((1+2.0*af*Ef)/sqrt(Ef*(1+af*Ef)));
35.        SumBW = sqrt(Ef*(1.0+af*Ef)) +
sqrt(E[e_step]*(1.0+af*E[e_step]));
36.        SquareqxBW = 2.0*q*(m1*m0)/(hbar*hbar)*SumBW*SumBW;
37.        scat_table[ValleyBefore][n][m][e_step][types] =
SRS_constant_m1/(2.0+SquareqxBW*Square_correlation_length)*DOS_temp*SRS_
form_factor[0][n][m][1];

```

```

38.     }
39.     else {
40.         scat_table[ValleyBefore][n][m][e_step][types] = 0.0;
41.     }
42. }
43. }
44. }
45. flag_mech[types][ValleyBefore] = 3;
46. i_valley[types][ValleyBefore] = ValleyAfter;
47. enerAdded[types][ValleyBefore] = 0.0;

```

Figure 3.4 An example source code for making SRS scattering table

Figure 3.4 shows an example source code for making SRS scattering table (at valley 1 and top interface with forward and backward processes).

- For Figure 3.4.
- Line 2 and line 3: The SRS scattering is treated as an intra-valley scattering mechanism.
- Line 4 to line 24: Making SRS scattering table for top interface and at valley 1 with forward process.
 - Line 4: **types=8**. In SRS scattering table, the “**types**” index for scattering at valley 1 and at Top interface with forward process is equal 8. See the Table 3.4.
 - Line 12 and 13: Calculating the term $(q_x^-)^2$. See Equation (3.4)
 - Line 14: Calculating the SRS scattering rate. See Equation (3.3)
 - ◆ The last index of **SRS_form_factor** array is equal to 1. See Table 3.3.
 - Line 22: To set **flag_mech** is equal to 2 (forward process). See the Table 3.5.
 - Line 24: To set **enerAdded** is equal to 0.0, because we treat SRS scattering as an elastic scattering mechanism.
- Line 27 to line 47: Making SRS scattering table for top interface and at valley 1 with backward process. It is the same way as described above (from line 4 to line 24).

3.5. The *normalize_table()* function. See [normalize_table.c](#)

- **max_gm[valley][n]** is to store the maximum of total scattering rate for each valley and subband ($\Gamma_{v,i}$)
 - “**valley**” index is the valleys. Its range is from **1** to **n_valley**.

- “n” index is the subbands. Its range is from 1 to NSELECT.

```

1. void normalize_table(){
2.     int node;// node: rank
3.     MPI_Comm_rank(MPI_COMM_WORLD,&node);

4.     double *****Get_scat_table();
5.     double *****scat_table = Get_scat_table();
6.     int Get_NSELECT(), Get_n_valley(),Get_NScatTypes(),Get_n_lev();
7.     int NSELECT = Get_NSELECT();
8.     int n_valley = Get_n_valley();
9.     int NScatTypes = Get_NScatTypes();
10.    int n_lev = Get_n_lev();
11.    int valley,n,m,e_step, types;

12.    double **Get_max_gm();
13.    double **max_gm = Get_max_gm();
14.    for(valley=1; valley<=n_valley; valley++){
15.        for(n=1; n<=NSELECT; n++){
16.            max_gm[valley][n] = 0.0;// Reset process
17.        }
18.    }

19.    double ***gm = d3matrix(1,n_valley,1,NSELECT,1,n_lev);
20.    for(valley =1; valley<=n_valley; valley++){
21.        for(n=1; n<=NSELECT; n++){
22.            for(e_step=1; e_step<=n_lev; e_step++){
23.                gm[valley][n][e_step] = 0.0;
24.            }
25.        }
26.    }

27.    for(valley =1; valley<=n_valley; valley++){
28.        for(e_step=1; e_step<=n_lev; e_step++){
29.            for(n=1; n<=NSELECT; n++){
30.                for(types =1; types<=NScatTypes; types++){
31.                    for(m=1; m<=NSELECT; m++){
32.                        gm[valley][n][e_step] += scat_table[valley][n][m][e_step][types];
33.                    }
34.                }
35.            }
36.        }
37.    }

38.    for(valley =1; valley<=n_valley; valley++){
39.        for(n=1; n<=NSELECT; n++){
40.            max_gm[valley][n] = gm[valley][n][n_lev];
41.            for(e_step=1; e_step<=n_lev-1; e_step++){
42.                if(max_gm[valley][n]< gm[valley][n][e_step])
43.                {
44.                    max_gm[valley][n] = gm[valley][n][e_step];
45.                }
46.            }
47.            if(node==0)
48.            {
49.                printf("\n Maximum scatering rate at valley %d subband %d

```



```

50.         is %le",valley,n,max_gm[valley][n]);
51.     }
52. }

53.     double sum = 0.0;
54.     for(valley =1; valley<=n_valley; valley++){
55.         for(e_step=1; e_step<=n_lev; e_step++){
56.             for(n=1; n<=NSELECT; n++){
57.                 sum = 0.0;
58.                 types = 0;
59.                 do {
60.                     types = types + 1;
61.                     scat_table[valley][n][1][e_step][types] += sum;// at m=1
62.                     for(m=2; m<=NSELECT; m++){
63.                         scat_table[valley][n][m][e_step][types] += scat_table[valley][n][m-
64.                         1][e_step][types];
65.                     }
66.                     sum = scat_table[valley][n][NSELECT][e_step][types];
67.                 } while(types < NScatTypes);
68.             }
69.         }

70.     // Normalized scattering table with max_gm[valley][n]
71.     for(valley =1; valley<=n_valley; valley++){
72.         for(e_step=1; e_step<=n_lev; e_step++){
73.             for(n=1; n<=NSELECT; n++){
74.                 for(types =1; types <=NScatTypes; types++){
75.                     for(m=1; m<=NSELECT; m++){
76.                         scat_table[valley][n][m][e_step][types] /= max_gm[valley][n];
77.                     }
78.                 }
79.             }
80.         }
81.     }
82.     free_d3matrix(gm,1,n_valley,1,NSELECT,1,n_lev);
83.     return;
84. }

```

Figure 3.5 An example code for normalization of scattering table

Figure 3.5 shows an example code for normalization of scattering table.

- For Figure 3.5.
- Line 4 to line 10: To get **scat_table**, **NSELECT**, **n_valley**, **NScatTypes** and **n_lev**.
- Line 12 to line 18: The **max_gm[valley][n]** array is reset to zero.
- Line 19 to line 26: To declare a local variable **gm[valley][n][e_step]**, which is to store the summation of scattering rates for each valley, subband and energy step.
- Line 27 to 37: Summation of scattering rates for each valley, subband and energy step and then store in the array **gm[valley][n][e_step]**.

- Line 38 to line 52: To find maximum of **gm[valley][n][e_step]** at each valley and subband and then store it in **max_gm[valley][n]**. We also display the values of **max_gm[valley][n]** at node 0.
- Line 53 to line 69: To sum the scattering rates and store in tabulated scattering table. Figure 3.6 shows an example of successive summations of scattering rates at **valley=1** (ValleyBefore = 1), **e_step=2** (EnerStepBefore = 2*de), **n=2** (SubbandBefore = 2) and **types=1**. See also Figure 5.6, 5.8 and 5.10 in [Section 5.1.2](#) for more intuitively.
- Line 70 to line 81: To normalize the scattering rate with **max_gm[valley][n]**.


| | | |
|---|--|--|
| $\Gamma_1+\Gamma_2+\Gamma_3+..\Gamma_N+\Gamma_{\text{self-scattering}}$ | Self-scattering |  |
| $\Gamma_1+\Gamma_2+\Gamma_3+...+\Gamma_N$ | Nth scattering mechanism (subband n=2 to subband m=NSELECT) | |
| ... | ... | |
| $\Gamma_1+\Gamma_2+\Gamma_3$ | 3 (subband n=2 to subband m=3) | |
| $\Gamma_1+\Gamma_2$ | 2 (subband n=2 to subband m=2) | |
| Γ_1 | 1 (subband n=2 to subband m=1) | |
| valley = 1, e_step = 2, n = 2 and types=1 | | |

Figure 3.6. Example of successive summation of scattering rates

4. Initial Condition module

This module includes the following functions

[electrons_initialization_for_Mobility\(\);](#)

Figure 4.1 List of functions used in the initial condition module

4.1. The *electrons_initialization_for_Mobility()* function. See [electrons_initialization_for_Mobility.c](#)

To initialize parameters for each electron including energy, momentum, position, valley and subband where the particle is residing for the first time.

The parameters of an electron are given by the following variables.

- **p[ne][i]**
 - The “ne” index: **ne = 1** to **Number_of_Electrons**
 - ◆ In serial computing, **Number_of_Electrons** is the total number of electrons used in the simulator.
 - ◆ In parallel computing, **Number_of_Electrons** is the number of particle in each node.
 - The “i” index: **i = 1** to **4**. The meaning of **p[ne][i]** is given in Table 4.1.

| | |
|------------------------------------|--|
| p[ne][1] = kx | Wave vector in x-direction |
| <i>p[ne][2] = x_position</i> | <i>The position of electron in the x-direction. NOT used in low-field mobility calculation</i> |
| p[ne][3] = free_flight_time | The time for drift process |
| <i>p[ne][4] = i_region</i> | <i>The particle is in Source, Channel or Drain region. NOT used for low-field mobility calculation</i> |

Table 4.1 The meanings of 2D array p[ne][i]

- **valley[ne]**: To store valley index in which the ne-th particle is residing.
- **subband[ne]**: To store subband index in which the ne-th particle is residing.
- **energy[ne]**: To store energy of the ne-th particle.

The “ne” index in **valley**, **subband** and **energy** variables has the same meaning as it is used in **p[ne][i]** variable.

```

1. int Get_inum();
2. int inum = Get_inum(); // Number of particles
3. for(ne=1; ne<=inum; ne++){ // Run for each particle
4.     electron_energy = -(0.5*Vt)*log(rr); // [eV] electron energy in 1D
5.     sum = 0.0;
6.
7.     // For valley 1
8.     for(j=1; j<=NSELECT; j++){
9.         if(rr <= sum + PercentagePopulation[1][j]){
10.            iv = 1; // valley index is and corresponding m*=m1
11.            k_momen =
                sqrt(2.0*m0*m1)*sqrt(q)/hbar*sqrt(electron_energy*(1+af*electron_energy)
            );
12.
13.            double RandomNumber = random2(&idum);
14.            if(RandomNumber<=0.5){
15.                kx = k_momen; // forward process
16.            }
17.            else {
18.                kx = - k_momen; // chon backward process
19.            }
20.            subband_index = j;
21.            goto NEXT;
22.        } // End of if(rr <= PercentagePopulation[1][j]){
23.        sum = sum + PercentagePopulation[1][j];
24.    }
25.
26.    // For valley 2
27.    for(j=1; j<=NSELECT; j++){
28.        if(rr <= (sum+PercentagePopulation[2][j])){// 2 la valley 2. NOTE:
            sum+
29.            iv = 2; // m*=mt
30.            k_momen =
                sqrt(2.0*m0*mt)*sqrt(q)/hbar*sqrt(electron_energy*(1+af*electron_energy)
            );
31.
32.            double RandomNumber = random2(&idum);
33.            if(RandomNumber<=0.5){
34.                kx = k_momen;

```

```

35.     }
36.     else {
37.         kx = - k_momen;
38.     }
39.     subband_index = j;
40.     goto NEXT;
41. }
42. sum = sum + PercentagePopulation[2][j];
43. }
44.
45. // For valley 3
46. for(j=1; j<=NSELECT; j++){
47.     if(rr <= (sum+PercentagePopulation[3][j])){//NOTE: sum+
48.         iv=3; // m*=mt
49.         k_momen =
            sqrt(2.0*m0*mt)*sqrt(q)/hbar*sqrt(electron_energy*(1+af*electron_energy)
            );
50.
51.         double RandomNumber = random2(&idum);
52.         if(RandomNumber<=0.5){
53.             kx = k_momen;
54.         }
55.         else {
56.             kx = - k_momen;
57.         }
58.
59.         subband_index = j;
60.         goto NEXT;
61.     }
62.     sum = sum + PercentagePopulation[3][j];
63. }
64.
65. NEXT:
66. sum = 0.0;
67.
68. //x_position = 0.0;
69. init_free_flight = -log(random2(&idum))/max_gm[iv][subband_index];
70.
71. // Mapping particle attributes
72. p[ne][1] = kx;
73. //p[ne][2] = x_position;
74. p[ne][3] = init_free_flight; // ts or tc phai sau khi tinh bang
    scattering
75. p[ne][4] = 0;
76. valley[ne] = iv;
77. subband[ne] = subband_index;
78. energy[ne] = electron_energy;// [eV] electron energy of particle neth
79.
80. }// End of for(ne=1; ne<=inum; ne++){

```

```

81. return;
82. }

```

Figure 4.2 An example code for electron initialization

Figure 4.2 shows an example code for electron initialization. See [section 4.3.5-Tuan's thesis](#) for more detail. By using random numbers, the number of electrons in each subband is initialized based on the percentage population ($P_{v,i}$) as described in Equation (2.1). See Figure 4.3 for more intuitively.

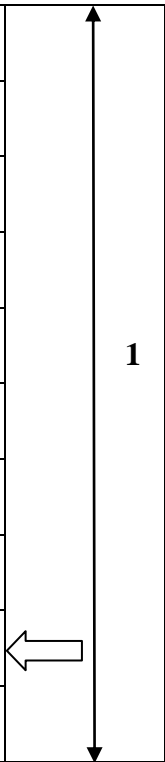
| | | |
|----------|---|--|
| Valley 3 | $P_{1,1} + \dots + P_{1,NSELECT} + P_{2,1} + \dots + P_{2,NSELECT} + P_{3,1} + \dots + P_{3,NSELECT}$ |  |
| | ... | |
| | $P_{1,1} + \dots + P_{1,NSELECT} + P_{2,1} + \dots + P_{2,NSELECT} + P_{3,1}$ | |
| Valley 2 | $P_{1,1} + \dots + P_{1,NSELECT} + P_{2,1} + \dots + P_{2,NSELECT}$ | |
| | ... | |
| | $P_{1,1} + \dots + P_{1,NSELECT} + P_{2,1}$ | |
| Valley 1 | $P_{1,1} + P_{1,2} + \dots + P_{1,NSELECT}$ | |
| | ... | |
| | $P_{1,1} + P_{1,2}$ | |
| | $P_{1,1}$ | |

Figure 4.3 An illustration of how to initialize electron based on the percentage population

- Line 3 to line 80: To initialize parameters for each particle.
 - Line 4: To initialize electron energy. See [Equation \(4.14\)-Tuan's thesis](#).
 - Line 7 to line 24: If an electron is assigned to valley 1 (The random number has a value in the range of valley 1 – See the arrow in Figure 4.3).
 - ◆ Line 10: The valley index is equal to 1.
 - ◆ Line 11: The magnitude of wave vector is calculated. See [Equation \(4.15\)-Tuan's thesis](#).
 - ◆ Line 13 to line 19: Choosing the direction of wave vector.

- ◆ Line 20: The subband index is determined. (E.g. in Figure 4.3, subband index is equal to 2.
- Line 26 to line 63: For valley 2 or 3.
- Line 66: To reset the “sum” variable for the next electron.
- Line 69: Free-flight time is initialized. See [Equation \(4.16\)-Tuan’s thesis.](#)
- Line 72 to line 78: To map particle attributes as clearly indicated in the code.

4.2. The *save_electron_distribution()* function. See [save_electron_distribution.c](#)

Based on the parameters of all electrons, we can know the electron distribution in each valley and subband, and electron energy distribution.

Even though this function is placed on the file [electrons_initilization_for_Mobility.c](#), it can be used at any time (usually, it is used for the initial step and final step).

The function [save_electron_parameters\(\)](#) also can be used at any time, however, it is usually used in checking process.

5. Multi-subband Monte Carlo transport for electrons

Two versions of MSMC method have been developed, one is the Single particle Monte Carlo (SMC) and the other is the Ensemble Monte Carlo (EMC). Depending on the problems we need to solve, we can choose SMC or EMC.

- For low-field mobility calculation, we can choose SMC or alternatively EMC, however, the implementation of SMC is easier.
- For fully self-consistent solution of the transport properties such as current-voltage characteristics, the EMC method must be used.

Below the implementation of the two models is introduced.

5.1. Single particle Monte Carlo

This part includes the following functions.

```
emcd Mobility();  
drift();  
scattering();  
isotropic Acoustic();  
isotropic();  
SRS Forward or Backward();  
occupation electron();  
gather data from all nodes();
```

Figure 5.1 List of functions used in the Single particle Monte Carlo

The important variables used in SMC code are given in the Table 5.1.

| Variable | Range of index | Calculating for | Description |
|----------------|----------------------------|--------------------|--|
| *VeloAllTime | [0] | All particles | “ Velo ” and “ Ener ” stand for velocity and energy, respectively. “ AllTime ” meaning that we |
| *VeloTransTime | [0,NumTransientPoints – 1] | All particles | |

| | | | |
|----------------------------|----------------------------|---------------|--|
| *EnerAllTime | [0] | All particles | calculate the velocity and energy in all time (including the transient time). “ TransTime ” meaning that |
| *EnerTransTime | [0,NumTransientPoints – 1] | All particles | we calculate parameters only after transient time. |
| *ElecOccupancyV1 | [0,NSELECT-1] | All particles | Electron occupancy in Valley 1, 2 and 3, respectively. Index “0” is for the first subband in the considered valley. |
| *ElecOccupancyV2 | [0,NSELECT-1] | All particles | |
| *ElecOccupancyV3 | [0,NSELECT-1] | All particles | |
| *VeloAllTimeSMC | [0] | One particle | <p>The meanings of the parameters can be referred to the above variables with a note that here we consider for only one particle.</p> <p>“SMC” appeared in the name of variables meaning that we use SMC method</p> |
| *VeloTransTimeSMC | [0,NumTransientPoints – 1] | One particle | |
| *EnerAllTimeSMC | [0] | One particle | |
| *EnerTransTimeSMC | [0,NumTransientPoints – 1] | One particle | |
| *ElecOccupancyV1SMC | [0,NSELECT-1] | One particle | |
| *ElecOccupancyV2SMC | [0,NSELECT-1] | One particle | |
| *ElecOccupancyV3SMC | [0,NSELECT-1] | One particle | |

Table 5.1 The meaning of some important variables in the SMC code

```

1. void emcd_Mobility(){ // Infact Single Particle Monte Carlo for Mobility
   and run for "inum" particles
2.
3.     void Set_kx(double k_momen_x),Set_dtau(double
   flight_time),Set_x_position(double position_in_x);
4.     void Set_electron_energy(double ener),Set_iv(int
   valley_pair_index),Set_subb(int subband_index);
5.
6.     double Get_kx(),Get_dtau(),Get_x_position(),Get_electron_energy();
7.     int Get_iv(),Get_subb();
8.
9.     void drift(const double tau, const double time);
10.    void scattering(const double time );
11.    void occupation_electron(const int i, const double tau, const double
   time);// "i" i-th particle
12.
13.    double **Get_p(),*Get_energy();
14.    double **p = Get_p();
15.    double *energy = Get_energy();
16.
17.    int *Get_valley(),*Get_subband();
18.    int *valley = Get_valley();
19.    int *subband = Get_subband();
20.
21.    float random2(long *idum);
22.
23.    double **Get_max_gm();// max_gm[valley][subband]
24.    double **max_gm = Get_max_gm();
25.
26.    int Get_inum();
27.    int inum = Get_inum();// Number of particle to simulate
28.
29.    double Get_tot_time();
30.    double tot_time = Get_tot_time();
31.
32.    int Get_NSELECT();
33.    int NSELECT = Get_NSELECT();
34.
35.    int Get_NumTransientPoints();
36.    int NumTransientPoints = Get_NumTransientPoints();// Da yeu cau it
   nhath la 1
37.
38.    double
   *Get_VeloAllTime(),*Get_VeloTransTime(),*Get_EnerAllTime(),*Get_EnerTran
   sTime();// CONG DON CHO TAT CA CAC HAT
39.    double
   *Get_ElecOccupancyV1(),*Get_ElecOccupancyV2(),*Get_ElecOccupancyV3();
40.    double *VeloAllTime = Get_VeloAllTime();
41.    double *VeloTransTime = Get_VeloTransTime();
42.    double *EnerAllTime = Get_EnerAllTime();

```

```

43.  double *EnerTransTime = Get_EnerTransTime();
44.  double *ElecOccupancyV1 = Get_ElecOccupancyV1();
45.  double *ElecOccupancyV2 = Get_ElecOccupancyV2();
46.  double *ElecOccupancyV3 = Get_ElecOccupancyV3();
47.
48.  double *Get_VeloAllTimeSMC(), *Get_VeloTransTimeSMC(),
    *Get_EnerAllTimeSMC(), *Get_EnerTransTimeSMC();//
49.
50.  double *Get_ElecOccupancyV1SMC(), *Get_ElecOccupancyV2SMC(),
    *Get_ElecOccupancyV3SMC();
51.  double *VeloAllTimeSMC = Get_VeloAllTimeSMC();
52.  double *VeloTransTimeSMC = Get_VeloTransTimeSMC();
53.  double *EnerAllTimeSMC = Get_EnerAllTimeSMC();
54.  double *EnerTransTimeSMC = Get_EnerTransTimeSMC();
55.  double *ElecOccupancyV1SMC = Get_ElecOccupancyV1SMC();
56.  double *ElecOccupancyV2SMC = Get_ElecOccupancyV2SMC();
57.  double *ElecOccupancyV3SMC = Get_ElecOccupancyV3SMC();
58.
59.  int i, j;
60.  int ValleyReside =0, SubbandReside =0;
61.  double time = 0.0, tau =0.0, rr = 0.0;
62.
63.  for(i=1; i<=inum; i++){ // Calculate for each particle
64.
65.      // RESET for each particle.
66.      time = 0.0;
67.
68.      for(j=0; j<=0; j++)//Chu y j
69.      {
70.          VeloAllTimeSMC[j] = 0.0;
71.          EnerAllTimeSMC[j] = 0.0;
72.      }
73.      for(j=0; j<=NumTransientPoints - 1; j++)
74.      {
75.          VeloTransTimeSMC[j] = 0.0;
76.          EnerTransTimeSMC[j] = 0.0;
77.      }
78.      for(j=0; j<=NSELECT-1; j++)
79.      {
80.          ElecOccupancyV1SMC[j] = 0.0;
81.          ElecOccupancyV2SMC[j] = 0.0;
82.          ElecOccupancyV3SMC[j] = 0.0;
83.      }
84. // En of RESET for each particle
85.
86.      while(time < tot_time)
87.      {
88.          // Inverse mapping of particle attributes.

```

```

89.      Set_kx(p[i][1]); // kx = p[i][1].
90.      //Set_dtau(p[i][3]); //dtau=p[i][3]. Do not need for SMC version
91.      Set_iv(valley[i]); // iv = valley[i]
92.      Set_electron_energy(energy[i]); // electron_energy = energy[i]
93.      Set_subb(subband[i]); // subb= subband[i]
94.
95.      ValleyReside = Get_iv();
96.      SubbandReside = Get_subb();
97.
98.      tau = -log(rr)/max_gm[ValleyReside][SubbandReside]; // tau
99.
100.     drift( tau, time);
101.
102.     scattering( time );
103.
104.     occupation_electron( i, tau, time);
105.
106.     time += tau;
107.
108.     // Map particle attributes
109.     p[i][1] = Get_kx(); //p[i][2] = Get_x_position();
110.     //p[i][3] = Get_dtau(); //Do not need for SMC version
111.     valley[i] = Get_iv();
112.     energy[i] = Get_electron_energy();
113.     subband[i] = Get_subb();
114.
115. } // while(time < tot_time)
116.
117.
118. VeloAllTime[0] += VeloAllTimesMC[0];
119. EnerAllTime[0] += EnerAllTimesMC[0];
120.
121. for(j=0; j< NumTransientPoints; j++)
122. {
123.     VeloTransTime[j] += VeloTransTimesMC[j];
124.     EnerTransTime[j] += EnerTransTimesMC[j];
125. }
126.
127. for(j=1; j<=NSELECT; j++)
128. {
129.     ElecOccupancyV1[j - 1] += ElecOccupancyV1SMC[j - 1];
130.     ElecOccupancyV2[j - 1] += ElecOccupancyV2SMC[j - 1];
131.     ElecOccupancyV3[j - 1] += ElecOccupancyV3SMC[j - 1];
132. }
133.
134. } // End of for(i=1; i<=inum; i++){ // Calculate for each particle
135. return;
136.} // End of void emcd_Mobility()

```

Figure 5.2 An example code for Single particle Monte Carlo

Figure 5.2 shows an example code for Single particle Monte Carlo method.

- For Figure 5.2.
- Line 3 to 7: Declaring the **Set** and **Get** functions. Function **Set** will set a new value for a variable, and function **Get** will get a value of a variable.
- Line 9 to 11: **drift()**, **scattering()** and **occupation_electron()** functions. See [Section 5.1.1](#), [Section 5.1.2](#) and [Section 5.1.3](#), respectively.
- Line 38: “Velo” and “Ener” mean velocity and energy, respectively. “AllTime” means that we calculate the velocity and energy in all time (including the transient time). “TransTime” means that we calculate these parameters only after transient time.
- Line 39: The electron occupancy in valley 1, 2 and 3 are calculated for all particles only after the transient time. See [Section 5.1.3](#).
 - **ElecOccupancyV1, ElecOccupancyV2 and ElecOccupancyV3** are 1D array `dvector(0,NSELECT-1);`
- Line 40 to 43: **VeloAllTime, VeloTransTime, EnerAllTime and EnerTransTime** are the sum for all particles in each node. These variables will be collected in the function [gather data from all nodes.c](#).
- Line 48 to 57: “**SMC**” means that for a single particle.
- Line 63 to line 135: The simulator runs for each particle. Say: the particle “ i^{th} ” is simulated
 - Line 66 to 83: To reset variables to simulate “ i^{th} ” particle.
 - Line 86 to line 115: To simulate “ i^{th} ” particle if the time evolution is still smaller than the pre-defined total time (**tot_time**).
 - ◆ Line 89 to 93: Inverse mapping of particle attributes. For the first time, these parameters are given by the [Initial Condition module](#), for other times these parameters are getting from the drift and scattering processes.
 - ◆ Line 95 and 96: To know which valley and subband the “ i^{th} ” particle is resided in.
 - ◆ Line 98: To generate free-flight time (“tau”) for the “ i^{th} ” particle to drift. See [Equation 4.16-Tuan’s thesis](#).
 - ◆ Line 100: The “ i^{th} ” particle is drifted during time “tau”

- ◆ Line 102: To check the scattering processes. The “ith” particle may experience an acoustic, non-polar optical phonon, SRS scatterings or “self-scattering”.
- ◆ Line 104: Electron occupancy of “ith” particle will be calculated if the evolution time is bigger than pre-defined transient time.
- ◆ Line 106: The evolution time will be increased by “tau”
- ◆ Line 109 to 113: The parameters of “ith” particle after drift and scattering processes are updated.
- Line 118 to 132: The summation of velocity and energy for all particles is taken place.

5.1.1. The *drift()* function. See [drift.c](#)

```

1.  #define qHBAR (q/hbar)
2.  #define hbarSquare (hbar*hbar)
3.  #define TwoM0 (2.0*m0)
4.
5.  void drift(const double tau, const double time){
6.
7.      int Get_flag_using_PARABOLIC_Band();
8.
9.      void Set_electron_energy(double ener), Set_kx(double k_momen_x);
10.
11.     double Get_electron_energy();
12.     double ei = Get_electron_energy();
13.
14.     double Get_kx();
15.     double kx = Get_kx();
16.
17.     double Get_Field_using();
18.     double Fx = Get_Field_using();
19.
20.     double dkx = (qHBAR)*Fx*tau;
21.     double kxNEW = kx+dkx; // Update momentum.[1/m]
22.
23.     int Get_iv();
24.     int v = Get_iv(); // v-th valley
25.
26.     // Update energy
27.     double E_parab = 0.0;
28.     switch (v)
29.     {
30.     case 1: // v=1: valley pair 1 m* = m1
31.         E_parab = hbarSquare*kxNEW*kxNEW/(TwoM0*m1); // [J]
32.         break;
33.     case 2: // v=2: valley pair 2 thi m* = mt
34.         E_parab = hbarSquare*kxNEW*kxNEW/(TwoM0*mt); // [J]
35.         break;
36.     case 3: // v=3: valley pair 3 thi m* = mt
37.         E_parab = hbarSquare*kxNEW*kxNEW/(TwoM0*mt); // [J]
38.         break;

```

```

38.     default:
39.     {
40.         printf("\n Wrong due to assign valley index! CHECK drift.c");
41.         exit(1);
42.     }
43. } // End of switch (v)
44.
45. E_parab = E_parab/q; // [J] -> [eV]
46.
47. double E_Nonparab = 0.0, ef = 0.0;
48.
49. if(Get_flag_using_PARABOLIC_Band() == 0) //Use NON-PARABOLIC
50. {
51.     E_Nonparab = (sqrt(1+4.0*af*E_parab)-1.0) / (2.0*af); // [eV]
52.     Set_electron_energy(E_Nonparab);
53.     Set_kx(kxNEW);
54.     ef = E_Nonparab;
55. }
56. else // Parabolic
57. {
58.     Set_electron_energy(E_parab);
59.     Set_kx(kxNEW);
60.     ef = E_parab;
61. }
62.
63. // All Time
64. VeloAllTimeSMC[0] += (ef - ei);
65. EnerAllTimeSMC[0] += (ef + ei)*tau;
66.
67. // Transient time
68. double Get_transient_time(); // from Input.d
69. double transient_time = Get_transient_time();
70. int Get_NumTransientPoints();
71. int NumTransientPoints = Get_NumTransientPoints();
72. double Get_LengthTransPoint();
73. double LengthTransPoint = Get_LengthTransPoint();
74.
75. double TotalLengTransPoint = (double)(NumTransientPoints)*LengthTransPoint;
76. int j, MAX = 0;
77. if( time > transient_time)
78. {
79.     if( time < transient_time + TotalLengTransPoint)
80.     {
81.         for(j=1; j<=NumTransientPoints; j++)
82.         {
83.             if(time < transient_time + (double)(j)*LengthTransPoint)
84.             {
85.                 MAX = j-1;
86.                 break;
87.             }
88.         }
89.
90.         for(j=0; j<=MAX; j++)
91.         {
92.             VeloTransTimeSMC[j] += (ef - ei);
93.             EnerTransTimeSMC[j] += (ef + ei)*tau;
94.         }
95.     }
96.
97.     else
98.     {

```

```

99.         for(j=0; j<=NumTransientPoints-1; j++)
100.         {
101.             VeloTransTimeSMC[j] += (ef - ei);
102.             EnerTransTimeSMC[j] += (ef + ei)*tau;
103.         }
104.     }
105.
106.     } // End of if( time > transient_time)
107.     return;
108. } // End of void drift(double tau)

```

Figure 5. 3 An example code for drift() function

Figure 5.3 shows an example code for *drift()* function. It is noted that during the drift process, the particle does not change its valley and subband.

- For Figure 5.3.
- Line 1 to 3: Some parameters are defined. The *drift()* function is used iteratively thus the more optimization on the code the better.
- Line 5: “**tau**” is the free-flight time. “**time**” is the evolution time, which is used to check whether the evolution time is over the pre-defined transient time or not
- Line 7: In the transport direction, we can use parabolic or non-parabolic band. If we use parabolic band, the non-parabolic factor is set to zero.
- Line 10 to 14: To know the kinetic energy (**ei**) and momentum (**kx**) of the particle before drifting.
- Line 16 and 17: To know the lateral field (**Fx**) in the transport direction (Pre-defined in Inputdeck)
- Line 19: The change in wave vector is calculated. [See Eq \(4.17\) – Tuan’s thesis.](#)
- Line 20: The wave vector is updated.
- Line 22 and 23: To know which valley the particle is residing. (NOTE: in the case, the effective mass for each subband in the same valley is different, we also need to know which subband the particle is residing in by using the code (See [section 6](#) for more detail):

```

int Get_subb();
int SubbandReside = Get_subb();

```

- Line 25 to 43: Updating electron energy using the parabolic band. [See Eq \(4.18\) – Tuan’s thesis.](#)

- Line 49 to 55: If the non-parabolic band for transport direction is used, the energy is calculated using [See Eq \(4.19\) – Tuan’s thesis](#), and then the energy and momentum after drift process are updated.
- Line 56 to 60: Using parabolic band, the energy and momentum after drift process are updated.
- Line 64 to 65: Cumulative summations of velocity and energy for all time are implemented. See [Eq. \(4.23\) to \(4.26\) – Tuan’s thesis](#). For clarity, they are given as below

$$\langle v_x \rangle_T = \frac{1}{T} \sum \langle v_x \rangle_\tau \tau = \frac{1}{T} \sum -\frac{(E_f - E_i)}{qF_x} = -\frac{1}{TqF_x} \sum (E_f - E_i) \quad (5.1)$$

$$\langle E \rangle_T = \frac{1}{T} \sum \langle E \rangle_\tau \tau = \frac{1}{T} \sum \frac{E_i + E_f}{2} \tau = \frac{1}{2T} \sum (E_i + E_f) \tau \quad (5.2)$$

In the above equations, since the first terms are constant, we will include them in final calculation (See [gather data from all nodes.c](#)).

- Line 68 to 104: Cumulative summations of velocity and energy after transient time are implemented.

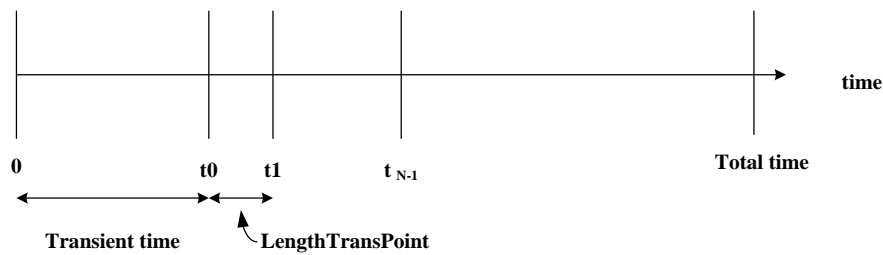


Figure 5.4 The position of transient points are indicated

Figure 5.4 shows the position of transient points. In this figure, $N = \text{NumTransientPoints}$.

And we assumed that the distance between two adjacent points is equal and it is $(t1-t0) = \text{LengthTransPoint}$.

- Line 68 and 69: **transient_time = t0**

- Line 70 and 71: **NumTransientPoints** = **N**. The variable **NumTransientPoints** is at least equal to 1.
- Line 72 and 73: **LengthTransPoint** = $t_1 - t_0$
- Line 75: **TotalLengTransPoint** = $t_{N-1} - t_0$
- Line 77 to 104: The velocity and energy will be calculated after t_0, t_1, \dots, t_{N-1} and stored in array with indices **0, 1, ..., N-1**, respectively.

5.1.2. The *scattering()* function. See [scattering_Mobility.c](#)

The **scattering()** function is to:

- select a scattering mechanism by which the particles are scattered
- calculate the states of particles after scattering (**kx**, **valley**, **subband** and **energy**)

This function will call the following functions: [isotropic Acoustic\(\)](#), [isotropic\(\)](#), and

[SRS Forward or Backward\(\)](#).

| Variable | Range of index | Scattering events (calculated after transient time) for | Index Description |
|--------------|----------------|---|---|
| *ACNumScat | [0,1] | Acoustic | Calculating the number of acoustic scattering events. [0]: Intra-subband; [1]: Inter-subband |
| *OPNumScat | [0,1] | Optical | [0]: f-type; [1]: g-type |
| *SRSNumScat | [0,1] | SRS | [0]: Intra-subband; [1]: Inter-subband |
| *SelfScatNum | [0,1] | Self-scattering | [0]: self-scattering; [1]: Not used now |

Table 5.2 Variables to calculate number of scattering events for each type of scattering mechanisms

The Table 5.2 shows the variables to calculate number of scattering events for each type of scattering mechanisms. These calculations are optional. Usually, they are very helpful in the checking process.

```

1. void scattering(const double time) {
2.     double de = e_max/(double)(n_lev); //energy_interval
3.     double Get_electron_energy();
4.     double EnergyBefore = Get_electron_energy
5.     int EnerStepBefore = (int)(EnergyBefore/de+0.5);
6.     int Get_iv();
7.     int ValleyBefore = Get_iv();
8.     int Get_subb();
9.     int SubbandBefore = Get_subb();
10.    int types = 0, typeSCAT = 0; // for last index of
    scat_table[valley][n][m][ener_step][types]; Note: types=0:Self-scattering
11.    int select_mech=0; //Isotropic or SRS_FW and BW process
12.    int ValleyAfter = 0, SubbandAfter = 0, m = 0; // valley, subband after

```

```

scattering
13.     double EnergyAfter = 0.0; // energy after scattering
14.     double AddEner = 0.0;

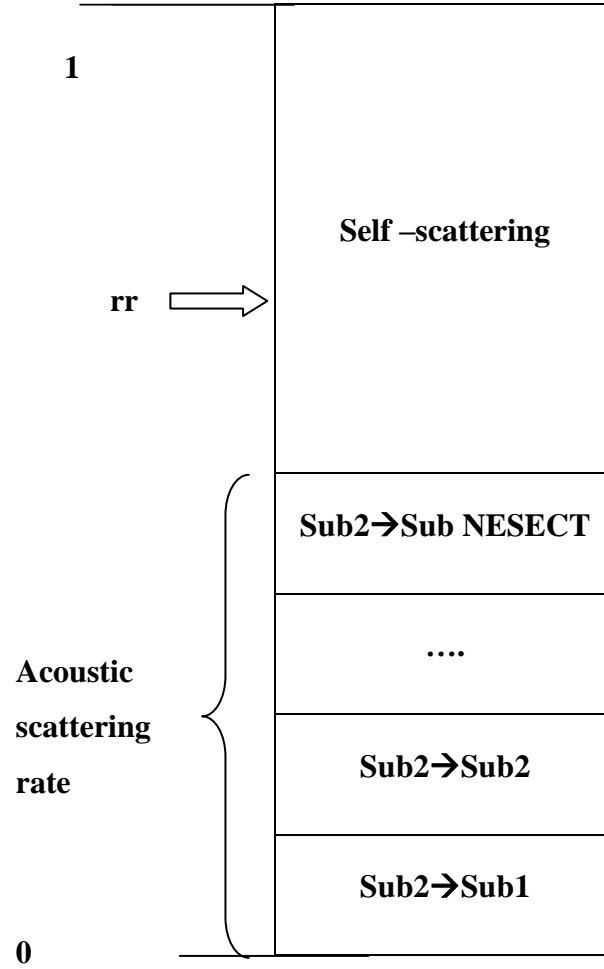
15.     double rr=0.0, bound_lower = 0.0, bound_upper = 0.0;
16.     // PART I. ONLY ACOUSTIC PHONON SCATTERING
17.     if ((flag_acoustic==1)&&(flag_zero_order_optical==0)&&(flag_Surface==0))
18.     {
19.         rr = random2(&idum);
20.         types = 1;
21.         if(rr >=
scat_table[ValleyBefore][SubbandBefore][NSELECT][EnerStepBefore][types])
22.             { // self scattering.
23.                 if(time > transient_time )
24.                 {
25.                     SelfScatNum[0] += 1.0;
26.                 }
27.                 goto Lend; // End of void scattering();
28.             }
29.             if(rr <
scat_table[ValleyBefore][SubbandBefore][1][EnerStepBefore][types]) // Note at 1
30.             {
31.                 SubbandAfter = 1; // di tu SubbandBefore den subband 1
32.                 ValleyAfter = ValleyBefore;
33.                 EnergyAfter = eig[0][ValleyBefore][SubbandBefore] -
eig[0][ValleyAfter][SubbandAfter] + EnergyBefore; // [eV]
34.                 if(EnergyAfter <=0)
35.                 {
36.                     goto Lend;
37.                 }
38.                 isotropic_Acoustic(time,
SubbandBefore, ValleyAfter, SubbandAfter, EnergyAfter);
39.                 goto Lend;
40.             }
41.             for(m=1; m<=NSELECT-1; m++) // Note: NSELECT-1
42.             {
43.                 bound_lower =
scat_table[ValleyBefore][SubbandBefore][m][EnerStepBefore][types]; // Note at m
44.                 bound_upper =
scat_table[ValleyBefore][SubbandBefore][m+1][EnerStepBefore][types]; // Note at
m+1
45.                 if((rr >= bound_lower)&&(rr< bound_upper)){
46.                     SubbandAfter = m+1; // Note m+1
47.                     ValleyAfter = ValleyBefore;
48.                     EnergyAfter = eig[0][ValleyBefore][SubbandBefore] -
eig[0][ValleyAfter][SubbandAfter] + EnergyBefore; // [eV]
49.                     if(EnergyAfter <=0.0){
50.                         goto Lend;
51.                     }
52.                     isotropic_Acoustic(time,
SubbandBefore, ValleyAfter, SubbandAfter, EnergyAfter);
53.                     goto Lend;
54.                 } // End of if((rr >= bound_lower)&&(rr< bound_upper))
55.             } // End of for(m=1; m<=NSELECT; m++)
56.         } // End of if Only Acoustic
57.     } // End of PART I. ONLY ACOUSTIC PHONON SCATTERING

```

Figure 5.5. The scattering() function. Part 1: if only Acoustic scattering is included

Figure 5.5 shows a part of **scattering()** function in which only acoustic scattering is included.

- For Figure 5.5.
- Please see the Figure 5.6 for understanding the source code more easily. In the figure, we assumed **EnerStepBefore = 2*de**, **ValleyBefore = 1** and **SubbandBefore = 2**.
- Line 2 to line 9: To get the electron energy, energy step, valley and subband of the “ith” particle from drift process. “**Before**” means that the parameters are for an electron before it is undergone a scattering event.
- Line 17: Only Acoustic scattering is included in the model.
- Line 20: **types = 1**. Because we consider acoustic phonon scattering. See Table 3.4.
- Line 21 to 28: If random number is bigger than the real-value of scattering rate, we choose self-scattering. **It is strongly recommended to check self-scattering before any types of real scattering to make the simulator run much faster.**
 - Line 23 to 26: We calculate self-scattering events only if the evolution time is over pre-defined transient time (t_0).
 - Line 27: The type of scattering mechanism has been chosen thus goto the end of **scattering()** function. Note: For self-scattering, there is no change on the particle parameters.
- Line 29 to 40: To check the scattering mechanism from **SubbandBefore** to subband 1.
 - Line 29: If random number is smaller than the normalized scattering rate from **SubbandBefore** to subband 1, this type of scattering is chosen.
 - Line 31: **SubbandAfter = 1**, because the subband after scattering is equal to 1.
 - Line 32: **ValleyAfter = ValleyBefore**, because acoustic is an intra-valley scattering.
 - Line 33: Energy after scattering is calculated. See [Equation \(A.27\) – Tuan’s thesis](#).
 - Line 34 to 38: Depending on the value of energy after scattering, this kind of scattering mechanism will be chosen or not. If it is chosen, the function [isotropic_Acoustic\(\)](#) is called. See [section 5.1.2.1](#).
- Line 41 to 55: Checking other scattering mechanisms
 - The **bound_lower** and **bound_upper** are used to decide which kind of scattering mechanisms is chosen.
 - The step is the same as described for scattering mechanism from **SubbandBefore** to Subband 1)



EnerStepBefore = 2*de, ValleyBefore = 1 and SubbandBefore = 2

Figure 5. 6 Illustration of normalized scattering table for only Acoustic phonon

```

1  // PART II.ACOUSTIC + ZERO-ORDER NON-POLAR OPTICAL PHONON SCATTERING
2  else if((flag_acoustic==1)&&(flag_zero_order_optical==1)&&(flag_Surface==0)){
3      rr = random2(&idum);
4      if(rr >=
5      scat_table[ValleyBefore][SubbandBefore][NSELECT][EnerStepBefore][NScatTypes])
6          { // self scattering.
7              if(time > transient_time )
8              {
9                  SelfScatNum[0] += 1.0;
10             }
11             goto Lend;
12         }
13     for(types = 1; types <=NScatTypes; types++){ // NScatTypes = 7 if AC+OP included
14         if(rr < scat_table[ValleyBefore][SubbandBefore][1][EnerStepBefore][types])
15         {
16             typeSCAT = types;
17             SubbandAfter = 1; // SubbandBefore to subband 1

```

```

16     ValleyAfter = i_valley[types][ValleyBefore];
17     AddEner = enerAdded[types][ValleyBefore];
18     select_mech = flag_mech[types][ValleyBefore];
19     goto NEXT;
20 }
21 for(m=1; m<=NSELECT-1; m++)// Note: NSELECT-1
22 {
23     bound_lower =
24     scat_table[ValleyBefore][SubbandBefore][m][EnerStepBefore][types];// Note: m
25     bound_upper =
26     scat_table[ValleyBefore][SubbandBefore][m+1][EnerStepBefore][types];//Note: m+1
27     if((rr >= bound_lower)&&(rr< bound_upper)){
28         typeSCAT = types;
29         SubbandAfter = m+1;//
30         ValleyAfter = i_valley[types][ValleyBefore];
31         AddEner = enerAdded[types][ValleyBefore];
32         select_mech = flag_mech[types][ValleyBefore];
33         goto NEXT;
34     }
35 }
36 NEXT:
37 if(select_mech == 1){
38     isotropic(time, ValleyBefore,SubbandBefore,EnergyBefore,AddEner,
39     ValleyAfter,SubbandAfter,typeSCAT);
40     goto Lend;
41 }
42 else{
43     printf("\n Here is only Isotropic mechanisms. ERROR at scattering.c");
44     exit(1);
45 }

```

Figure 5. 7 The scattering() function. Part II: if Acoustic and Non-polar optical phonon scatterings are included

Figure 5.7 shows a part of **scattering()** function in which acoustic and non-polar optical phonon scatterings are included. For more easily understanding the code, see Figure 5.8, which illustrates the normalized scattering table for acoustic and non-polar optical phonon. (In this figure, we assumed **EnerStepBefore** = 2*de, **ValleyBefore** = 1 and **SubbandBefore** = 2). For the meaning of “types”, see Table 3.4.

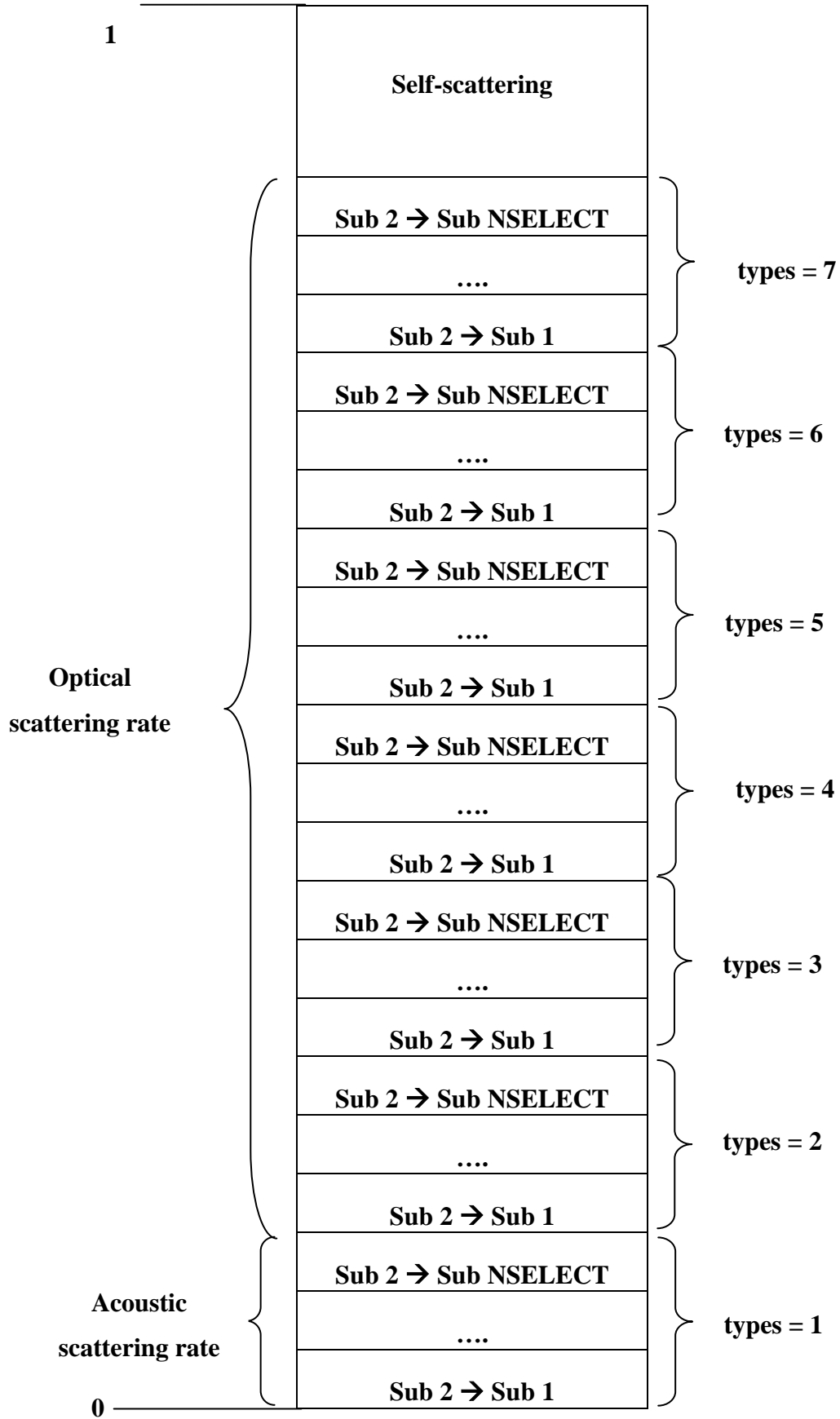


Figure 5. 8 Illustration of normalized scattering table for Acoustic and Non-polar Optical phonon.

- For Figure 5.7.
- Line 2: To indicate that the acoustic and non-polar optical phonon scatterings are included in the model.
- Line 3 to line 10: To check “self-scattering”. As mentioned above, it is advisable to firstly check “self-scattering”. If only acoustic and optical phonon included in the model, the **NScatTypes** = 7, (as mentioned in [Section 3.4](#)).
- Line 11 to line 34: To run for each “types” of scattering.

For each “types” of scattering, the way to select scattering mechanism is the same as for acoustic phonon as shown in Figure 5.5 and 5.6.

- For example, if the scattering mechanism for a particular “types” from **SubbandBefore** to subband 1 is selected ← Line 14 to 19

- ◆ Line 14: To get “types” (the type of the chosen scattering mechanism).
- ◆ Line 15: To get **SubbandAfter** (the subband after scattering).
- ◆ Line 16: To get **ValleyAfter** (the valley after scattering). This information is received from array **i_valley[types][ValleyBefore]** as described in [Section 3.4](#).
- ◆ Line 17: To get **AddEner** (the energy gain or lost after scattering mechanism). This information is received from array **enerAdded[types][ValleyBefore]** as described in [Section 3.4](#).
- ◆ Line 18: To get **select_mech** (the scattering mechanism is isotropic or anisotropic). This information is received from array **flag_mech[types][ValleyBefore]** as described in [Section 3.4](#).
- ◆ Line 19: go out the “loop” and then go to the function depending on the value of **select_mech**.
- Line 36 to 38: if the **select_mech=1**, the function *isotropic()* is called. See [section 5.1.2.2](#).

```

1.  // PART III. ACOUSTIC + ZERO-ORDER NON-POLAR OPTICAL + SRS SCATTERING *
2.  else if((flag_acoustic==1)&&(flag_zero_order_optical==1)&&(flag_Surface==1)){
3.      rr = random2(&idum);
4.      if(rr >=
scat_table[ValleyBefore][SubbandBefore][NSELECT][EnerStepBefore][NScatTypes])
5.          { // self scattering.
6.              if(time > transient_time )
7.              {
8.                  SelfScatNum[0] += 1.0;
9.              }
10.             goto Lend;

```

```

11.     }
12.     for(types = 1; types <= NScatTypes; types++){
13.         if(rr < scat_table[ValleyBefore][SubbandBefore][1][EnerStepBefore][types])
14.             {
15.                 typeSCAT = types;
16.                 SubbandAfter = 1;
17.                 ValleyAfter = i_valley[types][ValleyBefore];
18.                 AddEner = enerAdded[types][ValleyBefore];
19.                 select_mech = flag_mech[types][ValleyBefore];
20.                 goto Choice;
21.             }
22.         for(m=1; m<=NSELECT-1; m++)// chu y NSELECT-1
23.             {
24.                 bound_lower =
25.                 scat_table[ValleyBefore][SubbandBefore][m][EnerStepBefore][types];// chu y ta m
26.                 bound_upper =
27.                 scat_table[ValleyBefore][SubbandBefore][m+1][EnerStepBefore][types];//chu y tai
28.                 m+1
29.                 if((rr >= bound_lower)&&(rr< bound_upper)){
30.                     typeSCAT = types;
31.                     SubbandAfter = m+1;// chu y
32.                     ValleyAfter = i_valley[types][ValleyBefore];
33.                     AddEner = enerAdded[types][ValleyBefore];
34.                     select_mech = flag_mech[types][ValleyBefore];
35.                     goto Choice;
36.                 }
37.             }// End of for(m=1; m<=NSELECT; m++)
38.         }// End of for(types = 1; types <= NScatTypes; types++)
39.         Choice:
40.         if(select_mech == 1){// Acoustic, Zero order
41.             isotropic(time, ValleyBefore, SubbandBefore, EnergyBefore, AddEner,
42.             ValleyAfter, SubbandAfter, typeSCAT);
43.             goto Lend;
44.         }
45.         else if(select_mech == 2){// SRS FW
46.             SRS_Forward_or_Backward(time, ValleyBefore, SubbandBefore, EnergyBefore,
47.             AddEner, ValleyAfter, SubbandAfter, typeSCAT, select_mech );
48.             goto Lend;
49.         }
50.         else if(select_mech == 3){//SRS BW
51.             SRS_Forward_or_Backward(time, ValleyBefore, SubbandBefore, EnergyBefore,
52.             AddEner, ValleyAfter, SubbandAfter, typeSCAT, select_mech );
53.             goto Lend;
54.         }
55.         else{
56.             printf("\n Here is only Isotropic or SRS FW or SRS BW mechanisms.
57.             ERROR at scattering.c");
58.             exit(1);
59.         }
60.     }// End of else if((flag_acoustic==1)&&(flag_zero_order_optical==1)&&(flag_Surface==1))
61.     else{ // Flag cua cac loai scattering khong phu hop
62.         printf("\n Check Scattering list at Input.d. ERROR at scattering.c");
63.         exit(1);
64.     }
65.     Lend:
66.     return;
67. }

```

Figure 5. 9 The scattering() function. Part III: if Acoustic, Non-polar optical phonon and SRS scatterings are included

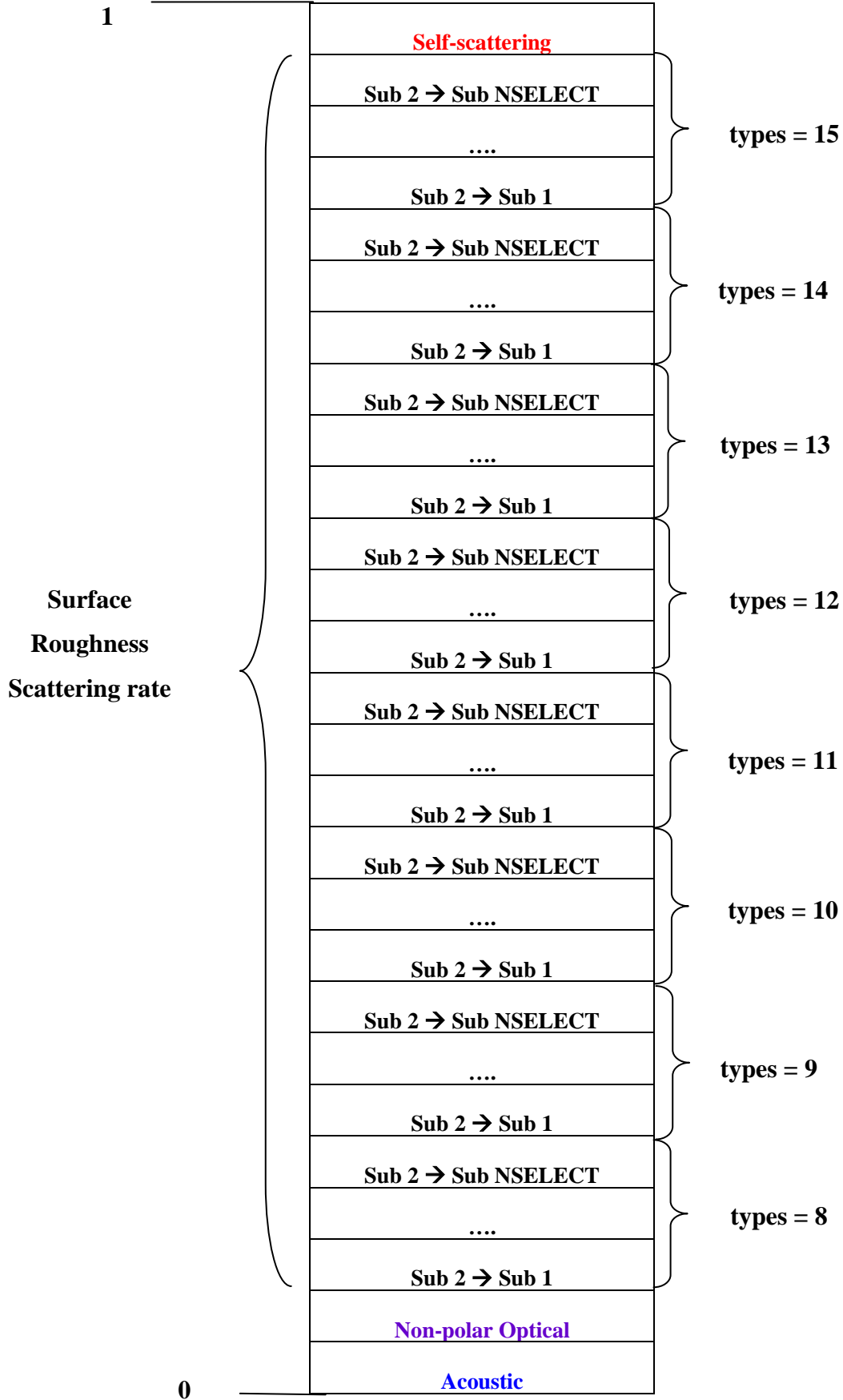


Figure 5. 10 Illustration of normalized scattering table for Acoustic, Non-polar Optical phonon and SRS scatterings.

Figure 5.9 shows a part of **scattering()** function in which acoustic, non-polar optical phonon and surface roughness scatterings are included. See Figure 5.10 (for more easily understanding the code), which illustrates normalized scattering table for acoustic, non-polar optical phonon and surface roughness. (In this figure, we assumed **EnerStepBefore = 2*de**, **ValleyBefore = 1** and **SubbandBefore = 2**).

- For Figure 5.9.
- Line 2: If acoustic, non-polar optical phonon and surface roughness scatterings mechanisms are included in the model.
- Line 3 to 35: For choosing the scattering mechanism. It is the same step as described in Figure 5.7. If acoustic and optical phonon and SRS are included in the model, the **NScatTypes = 15**, (as mentioned in [Section 3.4](#)).
- Line 36 to 52: Depending on the value of **select_mech**, the following functions should to be called.
 - If **select_mech =1**: The *isotropic()* function is called. See [section 5.1.2.2](#).
 - If **select_mech=2**: The *SRS_Forward_or_Backward()* function for forward process is called. See [section 5.1.2.3](#).
 - If **select_mech=3**: The *SRS_Forward_or_Backward()* function for backward process is called. See [section 5.1.2.3](#).

5.1.2.1. The *isotropic_Acoustic()* function. See [isotropic_Acoustic.c](#)

```

1. void isotropic_Acoustic(const double time, const int SubbandBefore, const int
   ValleyAfter, const int SubbandAfter, const double EnergyAfter){
2.     void Set_iv(int valley_pair_index);
3.     void Set_subb(int subband_index);
4.     void Set_kx(double k_momen_x);
5.     void Set_electron_energy(double ener);

6.     float random2(long *idum);

7.     double Get_transient_time();
8.     double transient_time = Get_transient_time();

9.     double *Get_ACNumScat();
10.    double *ACNumScat = Get_ACNumScat();

11.    double Get_nonparabolicity_factor(), Get_ml(), Get_mt();
12.    double af = Get_nonparabolicity_factor();
13.    double ml = Get_ml();
14.    double mt = Get_mt();

15.    double MassAfter = 0.0;
16.    if(ValleyAfter==1)
17.    {

```

```

18.     MassAfter = m1*m0;
19. }
20. else
21. {
22.     MassAfter = mt*m0;
23. }

24. double kxAfter = 0.0, k_update = 0.0, rr = 0.0;
25. k_update=sqrt(2.0*MassAfter*q)*sqrt( EnergyAfter*(1.0+af* EnergyAfter))/hbar;
26. rr = random2(&idum);
27. if(rr < 0.5)
28. {
29.     kxAfter = k_update; // forward process
30. }
31. else
32. { // random_number > 0.5
33.     kxAfter = -k_update; // backward process
34. }

35. // Update after SCATTERING
36. Set_iv(ValleyAfter);           // valley
37. Set_subb(SubbandAfter);       // subband
38. Set_kx(kxAfter);              // momentum
39. Set_electron_energy(EnergyAfter); // energy

40. if(time > transient_time )
41. {
42.     if(SubbandBefore == SubbandAfter) // Intra subband scat
43.     {
44.         ACNumScat[0] += 1.0; // nho la += nhe
45.     }
46.     else // Inter subband
47.     {
48.         ACNumScat[1] += 1.0;
49.     }
50. }
51. return;
52. } // End of void isotropic_Acoustic

```

Figure 5. 11 An example code for isotropic_Acoustic() function

Figure 5.11 shows an example code for **isotropic_Acoustic()** function.

- For Figure 5.11.
- Line 1: The *isotropic_Acoustic()* function. All input arguments have been described previously.
- Line 15 to line 23: The transport effective mass after scattering (**MassAfter**) depends on the valley after scattering. See [Table 2.1 – Tuan’s thesis](#)
 - If **ValleyAfter** = 1: **MassAfter** = $m_l \cdot m_0$.
 - If **ValleyAfter** = 2 or 3: **MassAfter** = $m_t \cdot m_0$.
- Line 24 and 25: The momentum after scattering is calculated. See [Equation \(4.15\) – Tuan’s thesis](#)

- Line 26 to line 34: A random number is generated to decide this kind of scattering mechanism is forward or backward process.
- Line 35 to line 39: To update the state after scattering including valley, subband, momentum and energy. These parameters are used as the inputs for drift process.
- Line 40 to line 50: If the evolution time is bigger than the pre-defined transient time, we calculate the number of intra- and inter-subband scattering events for acoustic scattering mechanism.
 - If **SubbandBefore** = **SubbandAfter**, it is intra-subband scattering mechanism; otherwise it is inter-subband scattering mechanism.

5.1.2.2. The *isotropic()* function. See [isotropic.c](#)

```

1. void isotropic(const double time, const int ValleyBefore, const int
   SubbandBefore, const double EnergyBefore, const double AddEner,
2.     const int ValleyAfter, const int SubbandAfter, const int typeSCAT){

3.     void Set_iv(int valley_pair_index);
4.     void Set_subb(int subband_index);
5.     void Set_kx(double k_momen_x);
6.     void Set_electron_energy(double ener);
7.     float random2(long *idum);
8.     double Get_transient_time();
9.     double transient_time = Get_transient_time();
10.    double *Get_ACNumScat(), *Get_OPNumScat();
11.    double *ACNumScat = Get_ACNumScat();
12.    double *OPNumScat = Get_OPNumScat();
13.    double ***Get_eig();
14.    double ***eig = Get_eig();
15.    double Get_nonparabolicity_factor(), Get_m1(), Get_mt();
16.    double af = Get_nonparabolicity_factor();
17.    double m1 = Get_m1();
18.    double mt = Get_mt();
19.    double MassAfter = 0.0;

20.    if(ValleyAfter==1)
21.    {
22.        MassAfter = m1*m0;
23.    }
24.    else
25.    {
26.        MassAfter = mt*m0;
27.    }

28.    double EnergyAfter;
29.    EnergyAfter = eig[0][ValleyBefore][SubbandBefore] -
   eig[0][ValleyAfter][SubbandAfter] + EnergyBefore + AddEner;
30.    if(EnergyAfter <=0.0)
31.    {
32.        return;
33.    }

34.    // Update carrier wavevector
35.    double kxAfter = 0.0, k_update = 0.0, rr = 0.0;

```

```

36. k_update=sqrt(2.0*MassAfter*q)*sqrt( EnergyAfter*(1.0+af* EnergyAfter))/hbar;
37. rr = random2(&idum);
38. if(rr < 0.5)
39. {
40.     kxAfter = k_update; // forward process
41. }
42. else
43. { // random_number > 0.5
44.     kxAfter = -k_update; // backward process
45. }

46. // Update after SCATTERING
47. Set_iv(ValleyAfter);           // valley
48. Set_subb(SubbandAfter);       // subband
49. Set_kx(kxAfter);              // momentum
50. Set_electron_energy(EnergyAfter); // energy

51. if(time > transient_time )
52. {
53.     if(typeSCAT == 1)// kieu scat la Acoustic
54.     {
55.         if(SubbandBefore == SubbandAfter)// AC Intrsub
56.         {
57.             ACNumScat[0] += 1.0;
58.         }
59.         else//AC Intersub
60.         {
61.             ACNumScat[1] += 1.0;
62.         }
63.     }
64.     else if ((typeSCAT==2)|| (typeSCAT==3)|| (typeSCAT==4)|| (typeSCAT==5))// OP
f-process
65.     {
66.         OPNumScat[0] += 1.0;
67.     }
68.     else if ((typeSCAT==6)|| (typeSCAT==7)) // OP g-process
69.     {
70.         OPNumScat[1] += 1.0;
71.     }
72. }
73. return;
74. }// End of void isotropic(

```

Figure 5.12 An example code for isotropic() function

Figure 5.12 shows an example code for **isotropic()** function.

- For Figure 5.12.
 - The code is very similar to the code used in *isotropic_Acoustic()* function (as shown in Figure 5.11).
- Line 20 to line 27: To know the transport effective mass after scattering.
- Line 28 to 29: To know energy after scattering (**EnergyAfter**). See [Equation \(A.35\) –Tuan’s thesis](#). **AddEner** is received in *scattering()* function.
 - **AddEner = 0.0** for Acoustic phonon. See Table 3.6.

- **AddEner** = $\pm\hbar\omega_f$ or $\pm\hbar\omega_g$ for optical phonon. See Table 3.6.
- Line 30 to 33: If **EnergyAfter** < **0.0**, the scattering is not occurred.
- Line 34 to line 72: If the scattering is occurred.
 - Line 34 to line 50: To update the wave vector, valley, subband and energy after scattering. As described in *isotropic_Acoustic()* function.
- Line 51 to line 71: To calculate the number of scattering events for acoustic and non-polar optical phonon scattering mechanisms. Basing on the **typeSCAT** value, we know the type of chosen scattering mechanism.

5.1.2.3. The *SRS_Forward_or_Backward()* function. See [SRS_Forward_or_Backward.c](#)

```

1. void SRS_Forward_or_Backward(const double time, const int ValleyBefore, const int
   SubbandBefore, const double EnergyBefore, const double AddEner,
2.     const int ValleyAfter, const int SubbandAfter, const int
   typeSCAT, const int scattering_mechanism ){

3.     void Set_iv(int valley_pair_index);
4.     void Set_subb(int subband_index);
5.     void Set_kx(double k_momen_x);
6.     void Set_electron_energy(double ener);
7.     double Get_transient_time();
8.     double transient_time = Get_transient_time();
9.     double *Get_SRSNumScat();
10.    double *SRSNumScat = Get_SRSNumScat();
11.    double ***Get_eig();
12.    double ***eig = Get_eig();
13.    double Get_nonparabolicity_factor(), Get_ml(), Get_mt();
14.    double af = Get_nonparabolicity_factor();
15.    double ml = Get_ml();
16.    double mt = Get_mt();

17.    double MassAfter = 0.0;
18.    if(ValleyAfter==1)
19.    {
20.        MassAfter = ml*m0;
21.    }
22.    else
23.    {
24.        MassAfter = mt*m0;
25.    }

26.    double EnergyAfter;
27.    EnergyAfter = eig[0][ValleyBefore][SubbandBefore] -
   eig[0][ValleyAfter][SubbandAfter] + EnergyBefore + AddEner;
28.    if(EnergyAfter <=0.0)
29.    {
30.        return;
31.    }

32.    double Get_kx();
33.    double kxBefore = Get_kx();
34.    // Update carrier wavevector
35.    double kxAfter = 0.0, k_update = 0.0;

```



```

36. k_update=sqrt(2.0*MassAfter*q)*sqrt( EnergyAfter*(1.0+af* EnergyAfter))/hbar;
37. if (scattering_mechanism == 2)//SRS FW
38. {
39.     if( kxBefore >=0.0)
40.     {
41.         kxAfter = k_update;
42.     }
43.     else //kxBefore < 0.0
44.     {
45.         kxAfter = -k_update;
46.     }
47. }
48. else if(scattering_mechanism == 3)// SRS BW
49. {
50.     if( kxBefore >=0.0)
51.     {
52.         kxAfter = -k_update;
53.     }
54.     else //kxBefore < 0.0
55.     {
56.         kxAfter = k_update;
57.     }
58. }
59. else{
60.     printf("\n Input for scattering_mechanism in SRS_Forward_or_Backward.c should
be 2 or 3. ERROR at SRS_Forward_or_Backward.c");
61.     exit(1);
62. }

63. // Update after SCATTERING
64. Set_iv(ValleyAfter);           // valley
65. Set_subb(SubbandAfter);       // subband
66. Set_kx(kxAfter);              // momentum
67. Set_electron_energy(EnergyAfter);// energy

68. if(time > transient_time )
69. {
70.     if(SubbandBefore == SubbandAfter)// SRS Intrsub
71.     {
72.         SRSNumScat[0] += 1.0;
73.     }
74.     else//SRS Intersub
75.     {
76.         SRSNumScat[1] += 1.0;
77.     }
78. }
79. return;
80. }// End of void SRS_Forward_or_Backward

```

Figure 5.13 An example code for *SRS_Forward_or_Backward()* function

Figure 5.13 shows an example code for *SRS_Forward_or_Backward()* function

- For Figure 5.13.
- Line 17 to line 25: To know the transport effective mass after scattering. It is the same as the code for *isotropic()* function as shown in Figure 5.10.

- Line 26 to line 31: To know energy after scattering (**EnergyAfter**). It is the same as the code for *isotropic()* function. In this case, due to SRS is elastic scattering, the **addEner** = **0.0**. See Table 3.6.
- Line 32 to 36: Calculating the magnitude of wave vector after scattering (**k_update**). The direction of wave vector after scattering is decided based on the **scattering_mechanism**, which is obtained from **flag_mech[types][valley]** (See Table 3.5).
 - If **scattering_mechanism** = **2**, it is a forward process (see Table 3.5). Thus if **kxBefore** is positive, **kxAfter** is positive and vice versa.
 - If **scattering_mechanism** = **3**, it is a backward process (see Table 3.5). Thus if **kxBefore** is positive, **kxAfter** is negative and vice versa.
- Line 63 to 67: Update the wave vector, valley, subband and energy after scattering. It is same as code described in *isotropic()* function.
- Line 68 to 78: Calculating number of scattering events for SRS scattering mechanisms. If (**SubbandBefore** == **SubbandAfter**), it is an intra-subband scattering, otherwise it is an inter-subband scattering.

5.1.3. The *occupation_electron()* function. See [occupation_electron.c](#)

```

1.  void occupation_electron(const int i, const double tau, const double time){
2.      double *Get_ElecOccupancyV1SMC(), *Get_ElecOccupancyV2SMC(),
      *Get_ElecOccupancyV3SMC();
3.      double *ElecOccupancyV1SMC = Get_ElecOccupancyV1SMC();
4.      double *ElecOccupancyV2SMC = Get_ElecOccupancyV2SMC();
5.      double *ElecOccupancyV3SMC = Get_ElecOccupancyV3SMC();

6.      int *Get_valley();
7.      int *valley = Get_valley();
8.      int *Get_subband();
9.      int *subband = Get_subband();
10.     int Get_NSELECT();
11.     int NSELECT = Get_NSELECT();
12.     double Get_transient_time();
13.     double transient_time = Get_transient_time();

14.     if(time > transient_time){
15.         int SubIndex;
16.         if(valley[i]==1){// Valley 1
17.             for(SubIndex=1; SubIndex<=NSELECT; SubIndex++){
18.                 if(subband[i]==SubIndex){
19.                     ElecOccupancyV1SMC[SubIndex - 1] += tau;
20.                     goto Lend;
21.                 }
22.             }
23.         }// end of if(valley[i]==1)
24.         else if (valley[i]==2){// Valley 2

```

```

25.     for(SubIndex=1; SubIndex<=NSELECT; SubIndex++){
26.         if(subband[i]==SubIndex){
27.             ElecOccupancyV2SMC[SubIndex - 1] += tau; // +=
28.             goto Lend;
29.         }
30.     }
31. }
32. else if (valley[i]==3){// Valley 3
33.     for(SubIndex=1; SubIndex<=NSELECT; SubIndex++){
34.         if(subband[i]==SubIndex){
35.             ElecOccupancyV3SMC[SubIndex - 1] += tau; // +=
36.             goto Lend;
37.         }
38.     }
39. }
40. else{
41.     printf("\n Something WRONG at occupation_electron.c due to assigned valley
index");
42.     exit(1);
43. }
44. }// End of if(time > transient_time){
45. Lend:
46.     return;
47. }

```

Figure 5. 14 An example code for *occupation_electron()* function

Figure 5.14 shows an example code for *occupation_electron()* function

- For Figure 5.14.
- Line 1: Input argument “i” is to indicate i^{th} particle. The “tau” and “time” arguments are the free-flight time and evolution time of the “ i^{th} ” particle.
- Line 14 to line 44: We calculate electron occupancy only if the evolution time is over the pre-defined transient time.
 - The **valley[i]** gives us the valley index in which the “ i^{th} ” particle is residing. For example, **valley[i] = 2** means that the “ i^{th} ” particle is in valley 2.
 - ◆ On the given valley (valley 2 in this example), based on **subband[i]**, we know which subband the particle is residing in. For instance, **subband[i] = 3** means that the “ i^{th} ” particle is in subband 3.
 - ◆ Hence the value of **ElecOccupancyV2SMC[SubIndex-1]** increase to “tau”, meaning that **ElecOccupancyV2SMC[SubIndex-1] += tau**.

5.1.4. The `gather_data_from_all_nodes()` function. See [gather_data_from_all_nodes.c](#)

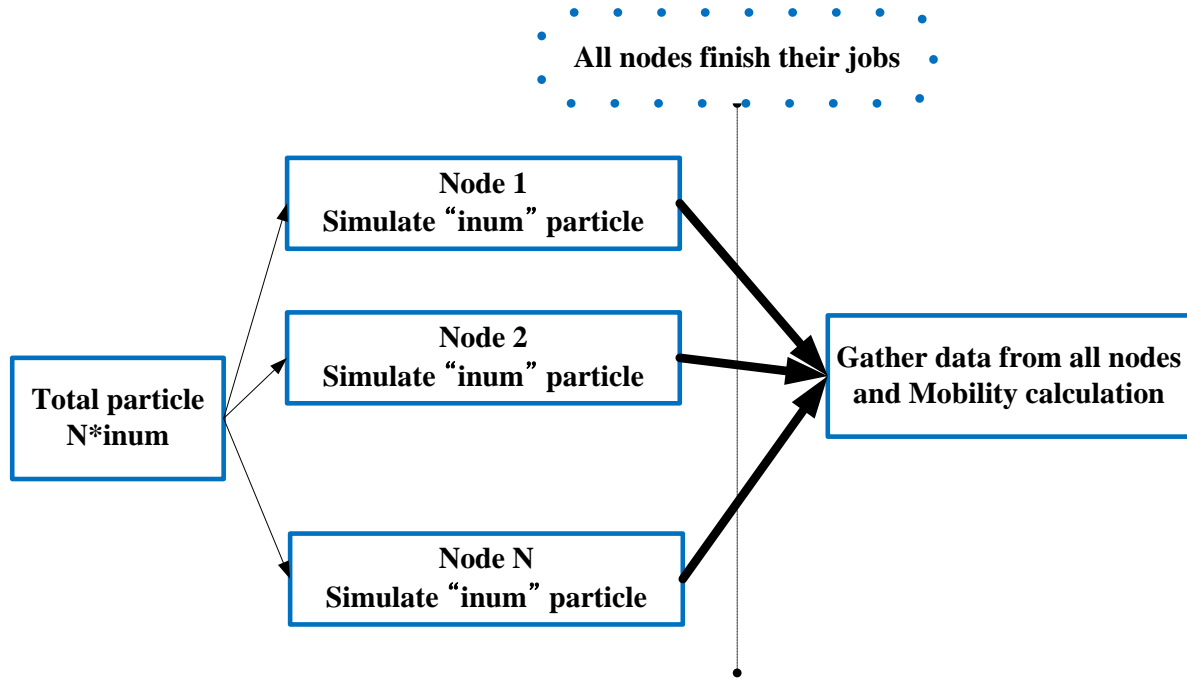


Figure 5. 15 Schematic of making SMC parallel code for mobility calculation.

Figure 5.15 shows the schematic of making SMC parallel code for mobility calculation.

- For Figure 5.15.
- Assuming that we want to use **N** nodes and assign number of particles “**inum**” for each node thus the total of particle is equal to **N*inum**.
- Each node simulates “**inum**” particles. Each particle is initialized differently and has a random number with different “seed”. Thus we “mimic” to simulate “**inum**” ensemble particles.
- After all the nodes finish their jobs, we gather data from all nodes and then calculate mobility.

```

1. void gather_data_from_all_nodes(char *fnAllTimes, char *fnTransientTimes, char
   *fnTransientTimesAverage, char *fnElectronOccupancy, char *fnScatEvent, const
   int DataAtGateInput){
2.     int numprocs = 1;
3.     int node = 0;
4.     MPI_Comm_rank(MPI_COMM_WORLD, &node);
5.     MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
6.     double Get_Field_using();
7.     double Fx = Get_Field_using()/1.0e+2; // [V/cm]
  
```

```

8.     int Get_inum();
9.     int inum = Get_inum();

10.    int inumAllNode = inum*numprocs;

11.    int Get_NSELECT();
12.    int NSELECT = Get_NSELECT();
13.    int Get_NumTransientPoints();
14.    int NumTransientPoints = Get_NumTransientPoints();// Da you cau it nhat la 1
15.    int i;
16.    // PART 1. For VELOCITY and ENERGY
17.    double
    *Get_VeloAllTime(),*Get_VeloTransTime(),*Get_EnerAllTime(),*Get_EnerTransTime();
18.    double *VeloAllTime = Get_VeloAllTime();
19.    double *VeloTransTime = Get_VeloTransTime();
20.    double *EnerAllTime = Get_EnerAllTime();
21.    double *EnerTransTime = Get_EnerTransTime();

22.    double *VeloAllTime_AllNode,*VeloTransTime_AllNode;//Sum Velo,Ener for AllNode
23.    double *EnerAllTime_AllNode, *EnerTransTime_AllNode;
24.    VeloAllTime_AllNode = dvector(0,0);
25.    EnerAllTime_AllNode = dvector(0,0);
26.    VeloTransTime_AllNode = dvector(0,NumTransientPoints - 1);
27.    EnerTransTime_AllNode = dvector(0,NumTransientPoints - 1);

28.    int rank = 0;
29.    int count = 1;
30.    MPI_Reduce( VeloAllTime, VeloAllTime_AllNode, count, MPI_DOUBLE, MPI_SUM, rank,
MPI_COMM_WORLD );
31.    MPI_Reduce( EnerAllTime, EnerAllTime_AllNode, count, MPI_DOUBLE, MPI_SUM, rank,
MPI_COMM_WORLD );

32.    rank = 0;
33.    count = NumTransientPoints;
34.    MPI_Reduce( VeloTransTime, VeloTransTime_AllNode, count, MPI_DOUBLE, MPI_SUM,
rank, MPI_COMM_WORLD );
35.    MPI_Reduce( EnerTransTime, EnerTransTime_AllNode, count, MPI_DOUBLE, MPI_SUM,
rank, MPI_COMM_WORLD );
36.    // END OF PART 1. For VELOCITY and ENERGY

37.    if(node ==0) // At NODE 0
38.    {
39.        double Vg = (double)(DataAtGateInput)/10.0;//trick at read_data_for_Mobility.c
40.        double Get_ChargeDensity();
41.        double ChargeDensity = Get_ChargeDensity(); // [1/m]
42.        double Get_TsiInput(), Get_WsiInput();
43.        double TsiInput = Get_TsiInput();
44.        double WsiInput = Get_WsiInput();
45.        double Nsurface = 1.0e+5*ChargeDensity/(2.0*TsiInput+2.0*WsiInput);// [1/cm2]
46.        double Get_tot_time(), Get_transient_time();
47.        double tot_time = Get_tot_time();
48.        double transient_time = Get_transient_time();// first t0 point

49.        // PART 1. For VELOCITY and ENERGY
50.        // PART 1.1. All Time
51.        int Get_flag_calculate_VeloAllTimeSteps();
52.        int flag_calculate_VeloAllTimeSteps =
Get_flag_calculate_VeloAllTimeSteps();
53.        if(flag_calculate_VeloAllTimeSteps == 1)
54.        {
55.            FILE *fAllTime;

```

```

56.         fAllTime = fopen(fnAllTimes,"a");
57.         if(fAllTime == NULL)
58.         {
59.             printf("\n Cannot open file at gather_velo_ener_from_all_nodes.c");
60.             exit(1);
61.         }
62.         double VeloAverage = VeloAllTime_AllNode[0]/
(Fx*tot_time*(double)(inumAllNode));
63.         double Mobility = VeloAverage / Fx; //[cm2/V.s].
64.         double EnerAverage = EnerAllTime_AllNode[0]/
(2.0*tot_time*(double)(inumAllNode));
65.         fprintf(fAllTime,"%f %f %le %le %f %le %le \n",
66.             Vg ,
67.             Fx/1.0e+3,
68.             ChargeDensity,
69.             Nsurface,
70.             Mobility,
71.             VeloAverage,
72.             EnerAverage
73.         );
74.         fclose(fAllTime);
75.     } // End of if(flag_calculate_VeloAllTimeSteps == 1)
76. // End of PART 1.1. ALL Time

77. // PART 1.2. AFTER TRANSIENT TIME
78. double t0 = transient_time;
79. double Get_LengthTransPoint();
80. double LengthTransPoint = Get_LengthTransPoint();
81. double *transient_time_Array, *Mobility_Array; // store t0, t1 ,,...
82. transient_time_Array = dvector(0,NumTransientPoints - 1);
83. Mobility_Array = dvector(0,NumTransientPoints - 1);
84. for( i=0; i<=NumTransientPoints -1; i++)
85. {
86.     transient_time_Array[i] = tot_time -( t0 +
(double)(i)*LengthTransPoint);
87. }
88. double MobiAverage = 0.0, VeloTransAverage = 0.0, EnerTransAverage = 0.0;
89. for(i=0; i<=NumTransientPoints - 1; i++)
90. {
91.     VeloTransTime_AllNode[i] = VeloTransTime_AllNode[i] /
(Fx*transient_time_Array[i]*(double)(inumAllNode));
92.     VeloTransAverage += VeloTransTime_AllNode[i];
93.     Mobility_Array[i] = VeloTransTime_AllNode[i] / Fx; //[cm2/V.s]. Mobility
= Velo/Fx;
94.     MobiAverage += Mobility_Array[i]; // +=
95.     EnerTransTime_AllNode[i] = EnerTransTime_AllNode[i]/
(2.0*transient_time_Array[i]*(double)(inumAllNode)); // CHU Y co 2
96.     EnerTransAverage += EnerTransTime_AllNode[i];
97. }
98. MobiAverage = MobiAverage/((double)(NumTransientPoints));
99. VeloTransAverage = VeloTransAverage/((double)(NumTransientPoints));
100. EnerTransAverage = EnerTransAverage/((double)(NumTransientPoints));
101. FILE *fTransient, *fTransientAverage;
102. fTransient = fopen(fnTransientTimes,"a"); // Chu y kieu mo la "a"
103. fTransientAverage = fopen(fnTransientTimesAverage,"a");
104. if((fTransient==NULL)|| (fTransientAverage==NULL))
105. {
106.     printf("\n Cannot open file at gather_velo_ener_from_all_nodes.c");
107.     exit(1);
108. }
109. for(i=0; i<=NumTransientPoints - 1; i++)

```

```

110.     {
111.         fprintf(fTransient,"%7d  %18.6le  %18.6f  %18.6le  %18.6f\n",
112.             i, // Transient point thu may
113.             (tot_time - transient_time_Array[i])/1.0e-12,// [ps].
114.             Mobility_Array[i],
115.             VeloTransTime_AllNode[i], // Velocity cm/s
116.             EnerTransTime_AllNode[i] // eV
117.         );
118.     }
119.     fclose(fTransient);
120.     fprintf(fTransientAverage,"%f %f %le %le  %f  %le  %f  \n",
121.         Vg ,
122.         Fx/1.0e+3,
123.         ChargeDensity,
124.         Nsurface,
125.         MobiAverage,
126.         VeloTransAverage,
127.         EnerTransAverage
128.     );
129.     fclose(fTransientAverage);
130.     // END OF PART 1. For VELOCITY and ENERGY
131.     free_dvector(transient_time_Array, 0,NumTransientPoints - 1); //free local
variable
132.     free_dvector(Mobility_Array,0,NumTransientPoints - 1);
133. } // End of if(node ==0)

134. MPI_Barrier(MPI_COMM_WORLD);
135. free_dvector(VeloAllTime_AllNode,0,0); // free local variable
136. free_dvector(EnerAllTime_AllNode,0,0);
137. free_dvector(VeloTransTime_AllNode,0,NumTransientPoints - 1);
138. free_dvector(EnerTransTime_AllNode,0,NumTransientPoints - 1);

139. //Reset prprocess (to be used for the next Vg and Vd)
140. for(i=0; i<=0; i++)
141. {
142.     VeloAllTime[i] = 0.0;
143.     EnerAllTime[i] = 0.0;
144. }
145. for(i=0; i<=NumTransientPoints - 1; i++)
146. {
147.     VeloTransTime[i] = 0.0;
148.     EnerTransTime[i] = 0.0;
149. }
150. return;
151. }

```

Figure 5. 16 An example code for gather_data_from_all_nodes() function. Part 1: Velocity and Energy

Figure 5.16 shows an example code for **gather_data_from_all_nodes()** function (Part 1: Velocity and Energy).

- For Figure 5.16.
- Line 2 to line 4: To get size (**numprocs**) and rank (**node**).
- Line 6 and 7: To get lateral electric field (F_x).
- Line 8 and 9: To get number of particles (“**inum**”) for each node.

- Line 10: To have total number of particles for all nodes (“**inumAllNode**”). It is the total number of particles to be used in the simulator.
- Line 11 to 14: To get **NSELECT** and **NumTransientPoints**.
- Line 17 to 21: To obtain **VeloAllTime**, **VeloTransTime**, **EnerAllTime** and **EnerTransTime** in each node.
- Line 22 to 27: The variables **VeloAllTime_AllNode**, **VeloTransTime_AllNode**, **EnerAllTime_AllNode** and **EnerTransTime_AllNode** are used to store the sum of **VeloAllTime**, **VeloTransTime**, **EnerAllTime** and **EnerTransTime**, respectively. Thus the size of these corresponding variables should be the same. For example **VeloAllTime_AllNode** and **VeloAllTime** should have the same size.
- Line 28 to line 31: To sum **VeloAllTime** at all nodes and then store in **VeloAllTime_AllNode**. Similarly for **EnerAllTime** and **EnerAllTime_AllNode**.
- Please see the **MPI_REDUCE** routine in any MPI book for more detail. Here, we summarize this routine.
 - **MPI_REDUCE** enables you to:
 - ◆ Collect data from each processor.
 - ◆ Reduce the data to a single value (such as sum or max). In our case we use “sum”.
 - ◆ And store the reduced result on the root processor (node 0).
 - E.g. **MPI_Reduce(VeloTransTime, VeloTransTime_AllNode, count, MPI_DOUBLE, MPI_SUM, rank, MPI_COMM_WORLD);**
 - ◆ **VeloTransTime** is the send_buffer
 - ◆ **VeloTransTime_AllNode** is the receive_buffer
 - ◆ **count** is the size of array to be send and receive
 - ◆ **MPI_DOUBLE** is the data_type
 - ◆ **MPI_SUM** is the reduction_operation. Here we use summation.
 - ◆ **rank** is the rank_of_receiving_process.
 - ◆ **MPI_COMM_WORLD** is the communicator.
- Line 32 to line 35: Similarly, we use **MPI_Reduce** for **VeloTransTime** and **EnerTransTime**.
- Line 37 to line 133: The simulator run only on node 0.
 - Line 39: The Vg input as a trick is calculated. See Table 1.1.
 - Line 40 and 41: To get inversion charge density (N_{inv})

- Line 42 to line 45: In the case of Gate-all-Around, the surface electron density is calculated by using N_{inv} divided by the perimeter.
- Line 46 to line 48: To get total and transient times.
- Line 51 to line 75: Calculating the velocity, energy and then the mobility for all time (including the transient time).
 - ◆ Line 62: Velocity is calculated as an average value for all time and all particles. The F_x is appeared in the denominator ← See Equation (5.1)
 - ◆ Line 63: Low-field mobility is calculated.
 - ◆ Line 64: Energy is calculated ← See equation (5.2)
 - ◆ Line 65 to line 74: Open a file to save the result.
- Line 78 to line 129: Calculating the velocity, energy and then the mobility only after the transient time. Please see Figure 5.4.
 - Line 78 to line 80: To get t_0 and **LengthTransPoint**.
 - Line 81 to line 87:
 - ◆ ***transient_time_Array** is to store the “remain” time after transient points. For example **transient_time_Array[0] = tot_time – t_0** and **transient_time_Array[1] = tot_time – t_1** and so on.
 - ◆ * **Mobility_Array** is to store the mobility calculation from point t_0 , t_1 and so on.
 - Line 88 to line 100: To calculate the ensemble average of velocity, energy and mobility from point t_0 , t_1 and so on. And also to calculate the average of velocity, energy and mobility for all transient points.
 - Line 101 to line 129: Save the results into files.
- Line 134: Using **MPI_Barrier (MPI_COMM_WORLD)**. All nodes are needed to synchronize before running for the next data obtained from particular V_g and F_x . Please NOTE that only insert barriers when they are needed.
- Line 135 to line 138: Free local variables if they do not need to be used.
- Line 139 to line 149: Reset process for **VeloAllTime**, **EnerAllTime**, **VeloTransTime** and **EnerTransTime**.

Using the same method as described above (for velocity and energy), we can obtain electron occupancy and scattering events very easily. Please see the source code.

5.2 Ensemble Monte Carlo

For low-field mobility calculation, the Single particle Monte Carlo (SMC) has been used, however, at the first stage of development, an Ensemble Monte Carlo (EMC) code has been also implemented. In term of final results, the two models produce the “same” results. In term of implementation, however, SMC model has a little bit simpler than EMC model. Especially, in term of using parallel computation, the “speed-up” in SMC is almost linearity with the increase of the number of processors, whereas the “speed-up” in EMC is about 4.5 times at the maximum, as shown in Figure 5.17. It is due to the inherent limitation in the EMC model for making parallel code as explained later in this section. (The reference data in Figure 5.17 is taken from [Saraniti, Goodnick, et al. “Parallel Approaches for Particle-Based Simulation of Charge Transport in Semiconductors– 2002”](#)).

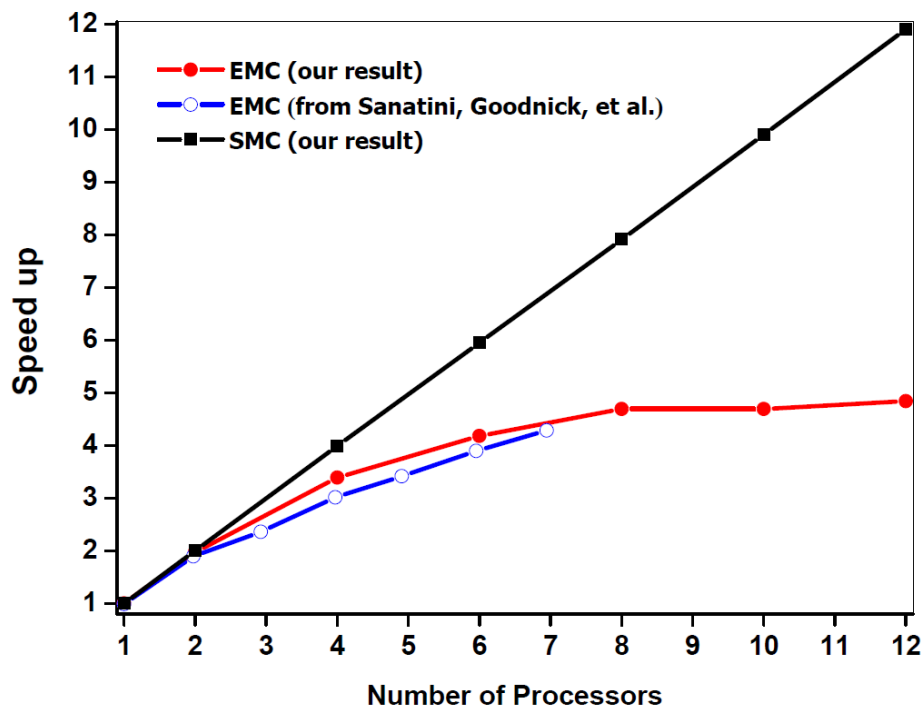


Figure 5. 17 Parallel performance of the SMC and EMC code

This section is to describe how to make EMC code for low-field mobility calculation. It may be useful for future developers, who will make the MSMC code for “fully self-consistent solution of the transport properties”.

This part includes the following functions.

```
emcd_Mobility();// For Ensemble Monte Carlo  
drift();// For Ensemble Monte Carlo  
velocity_energy_cumulative_Mobility();  
emcd_Mobility_parallel();// For Ensemble Monte Carlo
```

Figure 5.18 List of functions used for EMC code

5.2.1 Ensemble Monte Carlo (Serial code). See *emcd_Mobility()* function in [EMC emcd_Mobility.c](#)

Please see [Figure 4.2 – Tuan’s thesis](#) “the time evolution of an Ensemble Monte Carlo”. In EMC, a time step, Δt , is no longer defined as a single free flight, τ , terminated by a scattering event as for SMC method. The time step, Δt , is defined as a fixed time period. Within a time step, each particle may have many free-fights terminated by scattering events as is required to fill the time step. In the final free-flight, the carrier is only propagated for the time remaining in the current time step and not for the full free-flight. This ensures that all particles are propagated for the full time step, Δt , only. Figure 5.19 shows a flowchart for a particle experiences drift and scattering processes within one time step (This figure is taken from Prof. Vasileska’s document).

Figure 5.20 shows an example EMC code. Since it is a serial version, it can run only on one node (one processor).

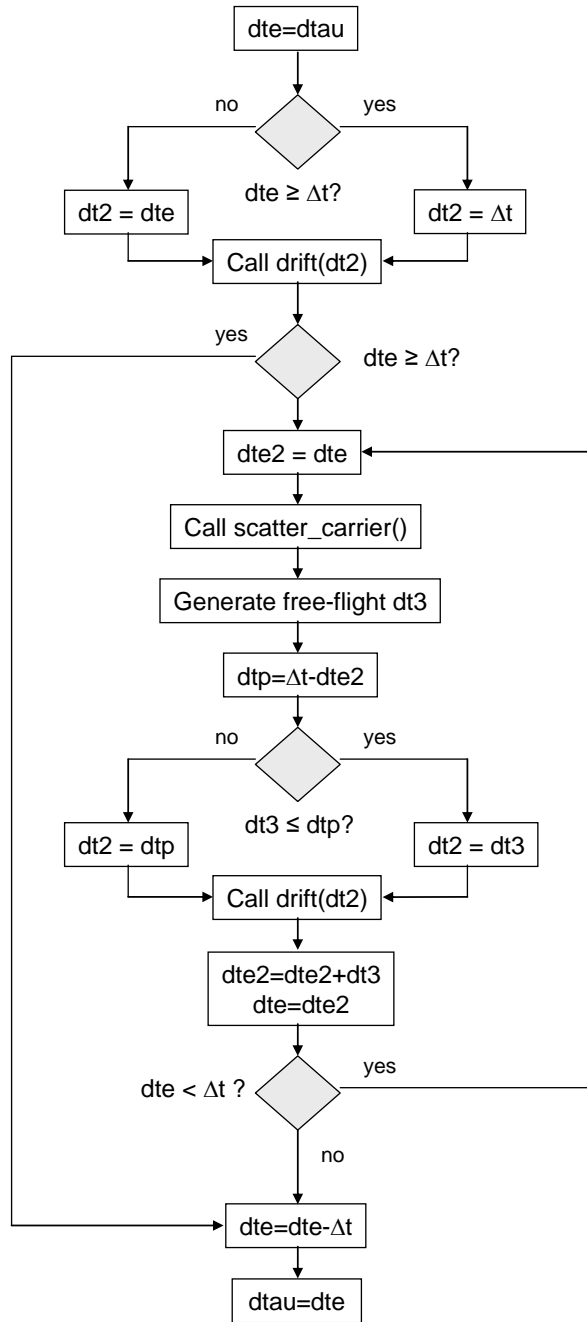


Figure 5.19 Flowchart for a particle experiences drift and scattering processes within one time step

```

1.  //***** EMC SERIAL VERSION *****
2.  void emcd_Mobility(){//EMC for Mobility in one observation time dt

3.      void Set_kx(double k_momen_x),Set_dtau(double flight_time),
        Set_x_position(double position_in_x);
4.      void Set_electron_energy(double ener),Set_iv(int
        valley_pair_index),Set_subb(int subband_index);
5.      double Get_kx(),Get_dtau(),Get_x_position(),Get_electron_energy();
6.      int Get_iv(),Get_subb();
7.      void drift(const double tau);
8.      void scattering();
9.      double **Get_p(),*Get_energy();
10.     double **p = Get_p();

```

```

11. double *energy = Get_energy();
12. int *Get_valley(),*Get_subband();
13. int *valley = Get_valley();
14. int *subband = Get_subband();
15. float random2(long *idum);
16. double **Get_max_gm();// max_gm[valley][subband]
17. double **max_gm = Get_max_gm();
18. double Get_dt();
19. double dt = Get_dt();
20. int Get_flag_ballistic_transport();
21. int flag_ballistic_transport = Get_flag_ballistic_transport();//=1 la BALLISTIC
22. int i,j, ValleyAfter, SubbandAfter;
23. double dte=0.0,dt2=0.0,dte2=0.0,rr=0.0,dt3=0.0,ntp=0.0,tau=0.0;

24. int Get_inum();
25. int inum = Get_inum();
26. int Get_n_valley();
27. int n_valley = Get_n_valley();
28. int Get_NScatTypes();
29. int NScatTypes = Get_NScatTypes();
30. int Get_NSELECT();
31. int NSELECT = Get_NSELECT();
32.
33. for(i=1; i<=inum; i++){ // Calculate for each particle
34.     // Inverse mapping of particle attributes
35.     Set_kx(p[i][1]);           // kx = p[i][1]
36.     //Set_x_position(p[i][2]); // x_position = p[i][2]
37.     Set_dtau(p[i][3]);         // dtau = p[i][3]
38.     Set_iv(valley[i]);         // iv = valley[i]
39.     Set_electron_energy(energy[i]); // electron_energy = energy[i]
40.     Set_subb(subband[i]);      // subb= subband[i]

41.     // BALLISTIC TRANSPORT or DIFFUSIVE TRANSPORT ?
42.     if( flag_ballistic_transport==1)//ballistic
43.     {
44.         drift(dt);// ballistic so just classical drift in time dt
45.         goto L403;// to update paramters of particle after drift process
46.     }

47.     // DIFFUSIVE TRANSPORT case
48.     tau = Get_dtau();
49.     dte = tau;
50.     if(dte >= dt)
51.     {
52.         dt2=dt;
53.     }
54.     else
55.     { // dte < dt
56.         dt2=dte;
57.     }
58.     // -> dt2 is a drift time
59.     drift(dt2); // Call drift() function during dt2
60.     while(dte <= dt)
61.     {
62.         // Free-flight and scatter part
63.         dte2 = dte;
64.         scattering();
65.         ValleyAfter = Get_iv();
66.         SubbandAfter = Get_subb();

67.         do

```

```

68.         {
69.             rr=random2(&idum);
70.         }
71.         while ((rr<=1.0e-6)|| (rr>=1.0));

72.         dt3 = -log(rr)/max_gm[ValleyAfter][SubbandAfter]; // tau
73.         dtp = dt - dte2;      // remaining time to scatter in dt-interval

74.         if(dt3 <= dtp)
75.         {
76.             dt2 = dt3;
77.         }
78.         else
79.         {
80.             dt2 = dtp;
81.         }

82.         drift(dt2);

83.         // Update times
84.         dte2 = dte2 + dt3;
85.         dte = dte2;
86.     } // End of the while(dte <= dt){

87.     // Meaning dte >= dt // after "while" loops
88.     dte = dte - dt;
89.     tau = dte;
90.     Set_dtau(tau); // tau

91. L403:
92.     // Map particle attributes
93.     p[i][1] = Get_kx();
94.     //p[i][2] = Get_x_position();
95.     p[i][3] = Get_dtau();
96.     valley[i] = Get_iv();
97.     energy[i] = Get_electron_energy();
98.     subband[i] = Get_subb();
99. } // End of for(i=1; i<=inum; i++)
100. return;
101. } // End of void emcd_Mobility()

```

Figure 5.20 An example EMC code (serial version)

- For Figure 5.20.
- Line 3 to line 17: All parameters and functions are described in the previous sections, except the function **void drift (const double tau)** has a little bit change in the code and will be described in the [section 5.2.2](#).
- Line 18 and line 19: To get observation time Δt , which is the pre-defined parameter in InputDeck.
- Line 20 and line 21: To get the **flag_ballistic_transport**. If its value is equal to **1**, particles are simulated in classical ballistic case; otherwise they are simulated in diffusive case.

- Line 23: The meanings of variables are exactly the same as indicated in Figure 5.19. Except that we used “**tau**” instead of “**dtau**”.
- Line 24 to line 31: The meanings of the parameters are described in the previous sections.
- Line 33 to line 99: “For each particle running in one time step, Δt ”, *i.e.*, if the first particle has propagated for the full time step, Δt , the second particle will be simulated and so on.
 - Line 34 to line 40: As described in Figure 5.2.
 - Line 41 to line 46: If choosing to simulate particles in classical ballistic case, the particles just drift in Δt .
 - Line 47: Indicating that after this point we use diffusive case. From line 48 to line 90: we exactly follow the steps as shown in Figure 5.19.
 - Line 48: (in this line) at the first time, **tau** is taken from initial condition, however, at other times, **tau** is taken from the remaining time for last free-flight of the previous time step (Δt). See line 89 and line 90.
 - Line 59: A particle is drifted during time dt_2 , which is determined depending on the conditions as indicate from line 50 to line 57.
 - Line 64: After drifting, we check scattering events by calling **scattering()** function. The **scattering()** function is the same as the scattering function described for SMC. See [section 5.1.2](#).
 - Line 65 and line 66: After having a scattering event, the particle may change the valley and subband thus we need to know this information.
 - Line 72: The new “**tau**” is initialized based on a random number and $\Gamma_{v,i}$. See [Equation 4.16 – Tuan’s thesis](#).
 - Line 88 to line 90: After the particle propagates a full time step Δt , the remaining time is stored in the **Set_tau()** function and will be used for the next time step.
 - Line 92 to line 98: As described in the Figure 5.2 of previous section ([Section 5.1](#)).

5.2.2 The *drift()* function. See [EMC drift.c](#)

```
1.  #define qHBAR (q/hbar)
2.  #define hbarSquare (hbar*hbar)
3.  #define TwoM0 (2.0*m0)
4.  void drift(const double tau){
5.      int Get_flag_using_PARABOLIC_Band();
6.      void Set_electron_energy(double ener), Set_kx(double k_momen_x);
7.      double Get_m1(), Get_mt(), Get_nonparabolicity_factor();
8.      double m1 = Get_m1();
9.      double mt = Get_mt();
10.     double af = Get_nonparabolicity_factor();

11.     double Get_kx();
12.     double kx = Get_kx();
13.     double Get_Field_using();
14.     double Fx = Get_Field_using();

15.     double dkx = (qHBAR)*Fx*tau;
16.     double kxNEW = kx+dkx; // Update momentum.[1/m]

17.     int Get_iv();
18.     int v = Get_iv(); // v-th valley
19.     // Update energy
20.     double E_parab = 0.0; // energy tính theo parabolic
21.     switch (v)
22.     {
23.     case 1: // v=1: valley pair 1: m* = m1
24.         E_parab = hbarSquare*kxNEW*kxNEW/(TwoM0*m1);
25.         break;
26.     case 2: // v=2: valley pair 2: m* = mt
27.         E_parab = hbarSquare*kxNEW*kxNEW/(TwoM0*mt);
28.         break;
29.     case 3: // v=3: valley pair 3: m* = mt
30.         E_parab = hbarSquare*kxNEW*kxNEW/(TwoM0*mt);
31.         break;
32.     default:
33.     {
34.         printf("\n Wrong due to assign valley index! CHECK drift()");
35.         exit(1);
36.     }
37.     } // End of switch (v)

38.     E_parab = E_parab/q; // [J] --> [eV]
39.     if(Get_flag_using_PARABOLIC_Band() == 0) // Use NON-PARABOLIC, NOT use Parabolic.
40.     {
41.         double E_Nonparab = (sqrt(1+4.0*af*E_parab)-1.0)/(2.0*af); // [eV]
42.         Set_electron_energy(E_Nonparab); // [eV]
43.         Set_kx(kxNEW);
44.     }
45.     else // Parabolic
46.     {
47.         Set_electron_energy(E_parab);
48.         Set_kx(kxNEW);
49.     }
50.     return;
51. } // End of void drift(double tau)
```

Figure 5. 21 An example code for *drift()* function used in EMC

Figure 5.21 shows an example code for **drift()** function used in EMC method.

- For Figure 5.21
- It is almost the same as Figure 5.3 (from line 1 to line 62).

5.2.3 The *velocity_energy_cumulative_Mobility()* function. See [EMC velocity_energy_cumulative_Mobility](#)

| Variables | Description |
|---------------------------------------|--|
| dt | Time step, Δt , (see Figure 4.2-Tuan's thesis). It is defined in the Inputdeck. |
| velocity_x_sum | Sum of velocity for all time (including the transient time). In fact, at the initial stage, we intended to make “fully-self-consistent calculation”, hence velocity_x_sum[s] , is to store the velocity in the x-direction (transport direction). (“s” is section index). Here, we calculate low-field mobility thus we do not need section index. |
| velocity_x_sum_after_transient | Sum of velocity after the transient time |
| energy_sum | Sum of energy for all time (including the transient time). |
| energy_sum_after_transient | Sum of energy after the transient time |
| iteration_reference | Used as an input argument (See line 1 in Figure 5.22). If evolution time of all particles “approach” total time iteration_reference=0 and we save the results; otherwise iteration_reference!=0 (different with 0), we calculate velocity, energy cumulatively. |

Table 5. 3 Some variables used in *velocity_energy_cumulative_Mobility()* function

The velocity of a particle in valley v is given by

$$velo_v = \frac{\hbar k_x}{m_v^*} \frac{1}{1 + 2\alpha E_{nonparabolic}} \quad (5.3)$$

where α is the non-parabolic factor, v is valley index.

The ensemble average velocity and energy at time step i are given by

$$\begin{aligned} \langle v \rangle_N^i &= \frac{\sum_{v=1}^{n_valley} V_v}{\sum_{v=1}^{n_valley} N_v} \\ \langle e \rangle_N^i &= \frac{\sum_{j=1}^{n_valley} E_v}{\sum_{v=1}^{n_valley} N_v} \end{aligned} \quad (5.4)$$

where V_v , E_v , and N_v represent sum of velocity, energy for all particles in valley v and total number of particles in that valley, respectively. And n_valley is number of valleys.

After calculating is above ensemble average for n time steps, the time average velocity and energy are given by

$$\begin{aligned} \langle \bar{v} \rangle_t &= \frac{1}{n} \sum_{i=1}^n \langle v \rangle_N^i \\ \langle \bar{e} \rangle_t &= \frac{1}{n} \sum_{i=1}^n \langle e \rangle_N^i \end{aligned} \quad (5.5)$$

```

1. void velocity_energy_cumulative_Mobility(char *fnAllTimes, char
   *fnTransientTimes, char *fnVeloEnervsTime, const double time, const int
   iteration_reference, const int DataAtGateInput ){

2.     double Fx = Get_Field_using()/1.0e+2; // [V/cm]
3.     int inum = Get_inum();
4.     int *valley = Get_valley();
5.     double **p = Get_p();
6.     double *energy= Get_energy();
7.     double af = Get_nonparabolicity_factor();//[1/eV]
8.     double ml = Get_ml();
9.     double mt = Get_mt();

10.    double dt = Get_dt(); // Observation time step
11.    double tot_time = Get_tot_time();
12.    double transient_time = Get_transient_time();

```

```

13. void Set_velocity_x_sum( double velo), Set_energy_sum(double ener);
14. void Set_velocity_x_sum_after_transient(double velo),
Set_energy_sum_after_transient(double ener);
15. double Get_velocity_x_sum(),Get_energy_sum();
16. double VelocityXSum =0.0, EnergySum = 0.0;
17. double Get_velocity_x_sum_after_transient(),Get_energy_sum_after_transient();
18. double VelocityXSumAfterTransient = 0.0, EnergySumAfterTransient = 0.0;

19. int total_time_steps = (int)(tot_time/dt + 0.5);
20. int n_time_steps_after_transient = (int)((tot_time-transient_time)/dt + 0.5);
21. int n_valley = Get_n_valley();
22. int j;

23. // 1. CALCULATE THE AVERAGE VALUE IF iteration_reference==0
24. if( iteration_reference == 0){

25. // Save for all time steps
26. double Vg = (double)(DataAtGateInput)/10.0; // trick at read_data_for_Mobility.c
27. double ChargeDensity = Get_ChargeDensity();
28. FILE *f;
29. f=fopen(fnAllTimes,"a");
30. //fprintf(f,"\n #Vg[V] Fx[kV/cm] ChargeDensity[1/m] Mobility[cm2/V.s]
velocity_x[cm/s] energy[eV] \n");
31. VelocityXSum = Get_velocity_x_sum();
32. double VeloAll = VelocityXSum*1.0e+2/((double)(total_time_steps)); // [cm/s]
33. double MobiAll = (VeloAll)/Fx; //[cm2/V.s]. "All" means all time steps
34. EnergySum = Get_energy_sum();
35. fprintf(f,"%f %f %le %f %le %le \n", Vg , Fx/1.0e+3,ChargeDensity,
MobiAll,
36. VelocityXSum*1.0e+2/((double)(total_time_steps)), // [cm/s]
37. EnergySum/((double)(total_time_steps)) // [eV]
38. );
39. fclose(f);
40. Set_velocity_x_sum(0.0);//Reset prrocess for next data of (Vg and Vd)
41. Set_energy_sum(0.0);

42. // SAVE data calculated for all time step after transient time
43. FILE *fTransient;
44. fTransient = fopen(fnTransientTimes,"a");
45. //fprintf(f,"\n #Vg[V] Fx[kV/cm] ChargeDensity[1/m] Mobility[cm2/V.s]
velocity_x[cm/s] energy[eV] \n");
46. VelocityXSumAfterTransient = Get_velocity_x_sum_after_transient();
47. double Velo =
VelocityXSumAfterTransient*1.0e+2/((double)(n_time_steps_after_transient)); // [cm/s]
48. double Mobility = Velo/Fx; //[cm2/V.s]
49. EnergySumAfterTransient = Get_energy_sum_after_transient();
50. fprintf(fTransient,"%f %f %le %f %le %le \n",Vg, Fx/1.0e+3,ChargeDensity,
Mobility,
51. VelocityXSumAfterTransient*1.0e+2/((double)(n_time_steps_after_transient)),
// [cm/s]
52. EnergySumAfterTransient/((double)(n_time_steps_after_transient)) // [eV]
53. );
54. fclose(fTransient);
55. Set_velocity_x_sum_after_transient(0.0);//Reset prrocess for next data of (Vg and Vd)
56. Set_energy_sum_after_transient(0.0);
57. }// End of if(iter_reference==0)
58. //**** End of 1. CALCULATE THE AVERAGE VALUE IF iteration_reference==0

59. //2. CALCULATE of velocity, energy, if iter_reference !=0
60. int n = 0, valley_index = 0;

```

```

61.     double ee = 0.0, velx = 0.0, kx = 0.0;
62.     // 2.1 For all time steps*****
63.     double *velx_sum,*sum_ener,*nvaly;
64.     velx_sum = dvector(1,n_valley );
65.     sum_ener = dvector(1,n_valley );
66.     nvaly = dvector(1,n_valley );
67.     for(j=1; j<=n_valley; j++){
68.         velx_sum[j]=0.0;
69.         sum_ener[j]=0.0;
70.         nvaly[j] = 0.0;
71.     }
72.     for(n=1; n<=inum; n++){
73.         valley_index = valley[n]; // 1, 2 or 3
74.         ee = energy[n]; // [ev] energy
75.         kx = p[n][1]; // [1/m]lay momentum
76.         switch (valley_index)
77.         {
78.             case 1: // valley_index=1
79.                 velx = ((hbar*kx)/(m1*m0)) / (1.0+2.0*af*ee);//[m/s]
80.                 break;
81.             case 2: // valley_index=2
82.                 velx = ((hbar*kx)/(mt*m0)) / (1.0+2.0*af*ee);//[m/s]
83.                 break;
84.             case 3: // valley_index=3
85.                 velx = ((hbar*kx)/(mt*m0)) / (1.0+2.0*af*ee);//[m/s]
86.                 break;
87.             default:
88.                 printf("\n Something Wrong! CHECK velocity_energy_cumulative.c");
89.                 } // End of switch (valley_index)
90.         velx_sum[valley_index] += velx;
91.         sum_ener[valley_index] += ee;
92.         nvaly[valley_index] += 1.0;
93.     } // End of for(int n=1; n<=inum; n++)
94.     double temp_velo, temp_ener;
95.     temp_velo=0.0; // Reset
96.     temp_ener=0.0;
97.     double NumParticle = 0.0;
98.     for(j=1; j<=n_valley; j++)
99.     {
100.         temp_velo  += velx_sum[j]; // Sum over all valleys
101.         temp_ener  += sum_ener[j];
102.         NumParticle += nvaly[j];
103.     }
104.     temp_velo = temp_velo / NumParticle;
105.     temp_ener = temp_ener / NumParticle;

106.     // Save file VeloEner vs. Times
107.     FILE *fVeloTime;
108.     fVeloTime = fopen(fnVeloEnervsTime,"a");
109.     //fprintf(fVeloTime,"\n #Time[ps]    velocity_x[cm/s]  energy[eV]  \n");
110.     fprintf(fVeloTime ,"%f %le %le \n",time/1.0e-12,
temp_velo*1.0e+2,temp_ener );
111.     fclose(fVeloTime);
112.     VelocityXSum = Get_velocity_x_sum();
113.     VelocityXSum += temp_velo;
114.     Set_velocity_x_sum(VelocityXSum);
115.     EnergySum = Get_energy_sum();
116.     EnergySum += temp_ener;
117.     Set_energy_sum(EnergySum);
118.     // End of 2.1 For all time steps *****

```

```

119. // 2.2. For time steps after transient time*****
120. velx = 0.0;
121. if(time > transient_time){
122.     double *velx_sum_after_transient,*sum_ener_after_transient,*nvaly_after_transient;
123.     velx_sum_after_transient = dvector(1,n_valley);
124.     sum_ener_after_transient = dvector(1,n_valley );
125.     nvaly_after_transient = dvector(1,n_valley );
126.     for(j=1; j<=n_valley; j++){
127.         velx_sum_after_transient[j]=0.0;
128.         sum_ener_after_transient[j]=0.0;
129.         nvaly_after_transient[j]= 0.0;
130.     }
131.     for(n=1; n<=inum; n++){
132.         valley_index = valley[n];
133.         ee = energy[n];
134.         kx = p[n][1];
135.         switch (valley_index)
136.         {
137.             case 1: // valley_index=1
138.                 velx = ((hbar*kx)/(m1*m0)) / (1.0+2.0*af*ee);//[m/s]
139.                 break;
140.             case 2: // valley_index=2
141.                 velx = ((hbar*kx)/(mt*m0)) / (1.0+2.0*af*ee);//[m/s]
142.                 break;
143.             case 3: // valley_index=3
144.                 velx = ((hbar*kx)/(mt*m0)) / (1.0+2.0*af*ee);//[m/s]
145.                 break;
146.             default:
147.                 printf("\n Something Wrong!CHECK velocity_energy_cumulative Transient.c");
148.             } // End of switch (valley_index)
149.             velx_sum_after_transient[valley_index] += velx;
150.             sum_ener_after_transient[valley_index] += ee;
151.             nvaly_after_transient[valley_index] += 1;
152.         } // End of for(int n=1;n<=inum;n++)

153.     temp_velo=0.0;
154.     temp_ener=0.0;
155.     double temp_nvaley = 0.0;
156.     for(j=1; j<=n_valley; j++)
157.     {
158.         temp_velo  += velx_sum_after_transient[j];
159.         temp_ener  += sum_ener_after_transient[j];
160.         temp_nvaley += nvaly_after_transient[j];
161.     }
162.     temp_velo = temp_velo/temp_nvaley;
163.     temp_ener = temp_ener/temp_nvaley;
164.     // cumulative
165.     VelocityXSumAfterTransient = Get_velocity_x_sum_after_transient();
166.     VelocityXSumAfterTransient += temp_velo;
167.     Set_velocity_x_sum_after_transient(VelocityXSumAfterTransient);

168.     EnergySumAfterTransient = Get_energy_sum_after_transient();
169.     EnergySumAfterTransient += temp_ener;
170.     Set_energy_sum_after_transient(EnergySumAfterTransient);
171. }// End of if(time > transient_time){
172. // End of 2.2. For time steps after transient time*****
173. return;
174. }

```

Figure 5.22 An example code for velocity_energy_cumulative_Mobility() function

Figure 5.22 shows an example code for **velocity_energy_cumulative_Mobility()** function.

- For Figure 5.22.
- Line 19: Total time steps (including the transient time) is calculated
- Line 20: Number of time step after transient time is calculated.
- Line 24 to line 57: To save results into files (**iteration_reference = 0**)
 - Line 31 to line 33: Mobility is calculated (“All” here meaning for all time steps).
 - Line 36 and line 37: Average velocity and energy for all time steps are calculated. See Equation (5.5).
 - Line 40 and line 41: To reset the **velocity_x_sum** and **energy_sum** variables.
 - Line 42 to line 56: To calculate the average velocity, average energy and mobility after transient time. The procedure is the same as for “all time steps”.
- From line 60 to line 172: Cumulative calculation of velocity and energy
- Line 63 to line 117: Cumulative calculation of velocity and energy for all time steps.
 - Line 63 to line 71: To declare local variables to store the velocity, energy, and number of electrons in each valley.
 - Line 72 to line 93: To calculate the parameters for all particles (from first particle to the last particle).
 - ◆ Line 73: Getting valley index of the n^{th} particle
 - ◆ Line 74: Getting energy the n^{th} particle
 - ◆ Line 75: Getting wave vector of the n^{th} particle
 - ◆ Line 76 to line 89: Depending on the valley index, the velocity of n^{th} particle is calculated. See Equation (5.3).
 - ◆ Line 90 to line 92: Velocity, energy and number of particles are summed cumulatively for each valley. Meaning that we calculate V_v , E_v , and N_v as described in Equation (5.4)).
 - Line 98 to line 105: To calculate ensemble average velocity and energy at a given time step i . See Equation (5.4).

NOTE: In fact, for low-field mobility calculation, the number of particles does not change with the time thus we do not need to calculate it. Because after go out the line 103, we have

$$\mathbf{NumParticle} = \sum_{v=1}^{n_valley} N_v = inum$$

where *inum* getting from Inputdeck.

For “fully self-consistent calculation”, however, number of particles may change with evolution time, hence the variable **nvaly[valley_index]** and **NumParticle** should be used.

- Line 106 to line 110: To save *velocity and energy versus time* in a file. For example, see [Figure 4.6 – Tuan’s thesis](#). However, the simulator will run more slowly. Thus it is recommended to make an option and only save these results in the checking process.
- Line 112 to line 117: The terms $\sum_{i=1}^n \langle v \rangle_N^i$ and $\sum_{i=1}^n \langle e \rangle_N^i$ in Equation (5.5) are calculated.
- Line 119 to line 171: Cumulative calculation of velocity and energy after transient time. The same procedure as described for “all time steps”, except here we calculate the quantities of interest after transient time.

5.2.5 Ensemble Monte Carlo (Parallel code). See *emcd_Mobility_parallel()* function in [EMC emcd Mobility.c](#)

It is worth to note that there are some available methods to make parallel code for EMC transport kernel.

- The first method is based on spatial division of the device into subgrids with each of processors assigned to a particular subgrid. Each processor simulate the motion of particles in its subgrid thus it has a responsibility for simulation of particles in a certain area of the device. See references such as:
[M. Goodnick et al. "PMC-3D A Parallel Three-Dimensional Monte Carlo Semiconductor Device Simulator" IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, VOL. 13, NO. 6, JUNE 1994.](#)
[Kepke and Ravaoli "3-D Parallel Monte Carlo Simulation of Sub-0.1 Micron MOSFETs on a Cluster Based Supercomputer" Journal of Computational Electronics 1 171–174, 2002.](#)
- In the second method, for each time step Δt , the ensemble of particles is divided into sub-ensembles, and each processor is responsible for the sub-ensemble of particles.

See reference [Saraniti, Goodnick, et al. “Parallel Approaches for Particle-Based Simulation of Charge Transport in Semiconductors– 2002”](#)).

In this part, the implementation for the second method is introduced. An example code is only shown for calculating the quantities of interest for “all time steps”. For “time after transient time”, the same procedure is used.

```

1.  //***** EMC PARALELL VERSION *****
2.  void emcd_Mobility_paralell(const int time_step_count,const double time,char
   *fnVeloEnervsTime )
3.  {
4.      double *velx_sum,*sum_ener,*nvaly;
5.      velx_sum = dvector(1,n_valley );
6.      sum_ener = dvector(1,n_valley );
7.      nvaly = dvector(1,n_valley );
8.      double temp_velo_All = 0.0, temp_ener_All = 0.0;// for All time steps
9.      int np = 0, npMAX = 0;
10.     int numprocs = 1;
11.     int node = 0;
12.     MPI_Comm_rank(MPI_COMM_WORLD,&node);//node: from 0 to numprocs-1
13.     MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
14.     if (inum % numprocs == 0)
15.     {
16.         np = (int)(inum/numprocs);
17.         npMAX = np;
18.     }
19.     else
20.     {
21.         np = (int)(inum / (numprocs-1));
22.         npMAX = np;
23.         if( node == (numprocs-1))
24.         {
25.             np = inum - np* (numprocs-1);
26.         }
27.     }
28.     MPI_Barrier(MPI_COMM_WORLD); // Blocks until all process have reached this routine
29.     //parallel code
30.     for(i = 1 + node*npMAX; i<=(1+node)*npMAX; i++)
31.     {
32.         // The assign for node 0: i from 1 to npMAX, for node 1: i from npMAX + 1 den 2*npMAX, etc.
33.         if (i > inum)// Checking process
34.         {
35.             break;
36.         }
37.         // Same as EMC serial code thus not shown source code for this part
38.         // Calculate Sub-ensemble average
39.         // Step 2.1 For all time steps.
40.         valley_index = valley[i];
41.         ee = energy[i];
42.         kx = p[i][1];
43.         switch (valley_index)
44.         {
45.             case 1:
46.                 velx = ((hbar*kx)/(m1*m0)) / (1.0+2.0*af*ee);
47.                 break;

```



```

47.         case 2:
48.             velx = ((hbar*kx)/(mt*m0)) / (1.0+2.0*af*ee);//[m/s]
49.             break;
50.         case 3:
51.             velx = ((hbar*kx)/(mt*m0)) / (1.0+2.0*af*ee);//[m/s]
52.             break;
53.         default:
54.             printf("\n Something Wrong! CHECK emcd_Mobility_parallel.c");
55.         }
56.         velx_sum[valley_index] += velx; // +=
57.         sum_ener[valley_index] += ee; // +=
58.         nvaly[valley_index] += 1.0;
59.         // End of Step 2.1 For all time steps
60.     } // End of for(i = 1 + node*npMAX; i<=(1+node)*npMAX; i++)
61.     MPI_Barrier(MPI_COMM_WORLD); // Blocks until all process have reached this routine
62.     //Continue Step 2.1 For all time steps.
63.     double VeloAllV1 = velx_sum[1]; // 'All' : all times
64.     double VeloAllV2 = velx_sum[2];
65.     double VeloAllV3 = velx_sum[3];
66.     double EnerAllV1 = sum_ener[1];
67.     double EnerAllV2 = sum_ener[2];
68.     double EnerAllV3 = sum_ener[3];
69.     double NumParAllV1 = nvaly[1]; //'Par': particle
70.     double NumParAllV2 = nvaly[2];
71.     double NumParAllV3 = nvaly[3];
72.     double TotalVeloAllV1 = 0.0, TotalVeloAllV2 = 0.0, TotalVeloAllV3 = 0.0; //
    "Total:sum from all nodes
73.     double TotalEnerAllV1 = 0.0, TotalEnerAllV2 = 0.0, TotalEnerAllV3 = 0.0;
74.     double TotalNumParAllV1 = 0.0, TotalNumParAllV2 = 0.0, TotalNumParAllV3 = 0.0;
75.     int count = 1;
76.     int rank = 0;
77.     MPI_Reduce(&VeloAllV1, &TotalVeloAllV1, count, MPI_DOUBLE, MPI_SUM, rank, MPI_COMM_WORLD);
78.     MPI_Reduce(&VeloAllV2, &TotalVeloAllV2, count, MPI_DOUBLE, MPI_SUM, rank, MPI_COMM_WORLD);
79.     MPI_Reduce(&VeloAllV3, &TotalVeloAllV3, count, MPI_DOUBLE, MPI_SUM, rank, MPI_COMM_WORLD);
80.     MPI_Reduce(&EnerAllV1, &TotalEnerAllV1, count, MPI_DOUBLE, MPI_SUM, rank, MPI_COMM_WORLD);
81.     MPI_Reduce(&EnerAllV2, &TotalEnerAllV2, count, MPI_DOUBLE, MPI_SUM, rank, MPI_COMM_WORLD);
82.     MPI_Reduce(&EnerAllV3, &TotalEnerAllV3, count, MPI_DOUBLE, MPI_SUM, rank, MPI_COMM_WORLD);
83.     MPI_Reduce(&NumParAllV1, &TotalNumParAllV1, count, MPI_DOUBLE, MPI_SUM, rank, MPI_COMM_WORLD);
84.     MPI_Reduce(&NumParAllV2, &TotalNumParAllV2, count, MPI_DOUBLE, MPI_SUM, rank, MPI_COMM_WORLD);
85.     MPI_Reduce(&NumParAllV3, &TotalNumParAllV3, count, MPI_DOUBLE, MPI_SUM, rank, MPI_COMM_WORLD);
86.     //End of Continue step 2.1 For all time steps
87.     MPI_Barrier(MPI_COMM_WORLD);
88.     if(node==0) //// For all times
89.     {
90.         temp_velo_All = 0.0; // Reset
91.         temp_ener_All = 0.0;
92.         temp_velo_All = TotalVeloAllV1 + TotalVeloAllV2 + TotalVeloAllV3; //3 valley
93.         temp_ener_All = TotalEnerAllV1 + TotalEnerAllV2 + TotalEnerAllV3;
94.         temp_velo_All = temp_velo_All / (TotalNumParAllV1 + TotalNumParAllV2 + TotalNumParAllV3);
95.         temp_ener_All = temp_ener_All / (TotalNumParAllV1 + TotalNumParAllV2 + TotalNumParAllV3);
96.         int Get_flag_write_Velocity_vs_Time(); //Write VelocityEnergy vs Time in a file YES or NO
97.         int flag_write_Velocity_vs_Time = Get_flag_write_Velocity_vs_Time(); // =1
    Write, 0: Not write
98.         if(flag_write_Velocity_vs_Time == 1)
99.         {
100.             FILE *fVeloTime;
101.             fVeloTime = fopen(fnVeloEnervsTime, "a");
102.             if(fVeloTime==NULL){
103.                 printf("\n Cannot open file at emcd_Mobility_parallel.c");
104.                 exit(1);
105.             }

```

```

106.      //fprintf(fVeloTime,"\n  #Time[ps]    velocity_x[cm/s]  energy[eV]  \n");
107.      fprintf(fVeloTime ,"%f %le  %le \n",time/1.0e-12, temp_velo_All*1.0e+2,
temp_ener_All);
108.      fclose(fVeloTime);
109.  }
110.  // All time steps (cumulative)
111.  double Get_velocity_x_sum(),Get_energy_sum();
112.  void Set_velocity_x_sum( double velo), Set_energy_sum(double ener);
113.  double VelocityXSum = Get_velocity_x_sum();
114.  VelocityXSum += temp_velo_All;
115.  Set_velocity_x_sum(VelocityXSum);
116.  double EnergySum = Get_energy_sum();
117.  EnergySum += temp_ener_All;
118.  Set_energy_sum(EnergySum);
119.  }// End of if(node==0) All times
120. return;
121. } // End of void emcd_Mobility_parallel()

```

Figure 5.23 An example code for EMC (parallel code)

Figure 5.23 shows an example code for EMC (parallel code)

- For Figure 5.23.
- Line 4 to line 7: Local variables to store velocity, energy and total number of particles for each valley.
- Line 12 and line 13: To get number of processors (**numprocs**) and the rank (**node**) for each processor in the communicator MPI_COMM_WORLD.
- Line 14 to line 27: To decide number of particles for each processor.
 - For low-field mobility calculation, we always choose the total number of particles (**inum**) are multiple of number of processors (**numprocs**), thus the line 14 to line 18 always occur.
 - For fully self-consistent solution, since the total number of particles can be changed during the time, thus the line 19 to 26 may be occurred. In that case, the last node will simulate the residual number of particles obtained from the (**inum** / (**numprocs-1**)).
- Line 30 to line 60: In each processor, the sub-ensemble of particles is simulated.
 - Line 36: The code is the same as serial version of EMC used in each processor. See line 35 to line 98 in Figure 5.20 of [section 5.2.1](#).
 - Line 37 to line 59: Calculating the sum of velocity and energy of the sub-ensemble. See line 73 to line 92 in Figure 5.22.
- Line 61 to line 87: To collect data from all nodes and store in node 0. For MPI_Reduce, see the description for Figure 5.16. NOTE: since it is the first parallel

version, I have ever made, I sent data as a single value. It is not convenient anymore. For sending data as an array, please see the line 32 to line 35 in Figure 5.16.

- Line 88 to line 119: Working only in node 0
 - Line 90 to line 95: The average velocity and energy for all particles are calculated.
 - Line 96 to line 109: A file *velocity and energy versus time* can be saved.
 - Line 110 to line 118: To sum cumulatively velocity and energy for each time step.

In the EMC parallel code, the ensemble average velocity and energy (for all particles) for each time step are calculated in the *emcd_Mobility_parallel()* function, thus the time average of velocity and energy are calculated in the *velocity_energy_cumulative_Mobility_parallel()* function. See this function in the [EMC_velocity_energy_cumulative_Mobility.c](#).

In the EMC parallel code as shown in Figure 5.23, after each time step Δt , all processes have synchronized to gather data therefore the speed-up of this method is only about 4.5 times at the maximum as shown in Figure 5.17. (It is also worth to note that for fully self-consistent solution of the transport properties, after each time step Δt , the profile of carrier density is calculated from the particle distribution and then it is fed into Poisson's solver for potential calculation. See [section 4.1.3-Tuan's thesis](#)).

6. Multi-subband Monte Carlo transport for holes

This section is to describe *the MSMC code for hole mobility calculation using an approximate scheme* as shown in [Section 6.2-Tuan's thesis](#). In this approach, the E-k band diagram for hole consists of perfect parabolic subbands as in the case for electron subbands, thus the MSMC transport kernel for electrons can be straightforwardly applied for hole counterpart with a few changes.

- For valance band, there is only one valley (**n_valley = 1**) thus it does not need to consider the valley index.
- Each subband has its own effective mass thus we need to make an array to store effective mass for subbands.

In the MSMC code for hole mobility calculation, some variables have been changed so as not to have a subband index, the other are kept the same as in the MSMC code for electrons by setting the valley index is equal to **1**.

Table 6.1 shows variables that have been changed so as not to have subband index.

| Variables | Description |
|--------------------------------|---|
| eig[i] | See Table 2.10 for the meanings of “i”, “j” and “k” indices |
| wave[i][j][k] | |
| LineDensity[i] | |
| PercentagePopulation[i] | |
| MassSub[i] | This array is to store the effective mass for each subband |
| form_factor[n][m] | See Table 3.1 for the meanings of “n” and “m” indices |

| | |
|---|--|
| SRS_form_factor[n][m] | <ul style="list-style-type: none"> ● See Table 3.1 for the meanings of “n” and “m” indices ● See section 6.1.3 to understand why we don’t need to consider SRS form factor for “Top”, “Bottom”, “Left” and “Right” interfaces separately. |
| scat_table [n][m][e_step][types] | <ul style="list-style-type: none"> ● See Table 3.1 for the meanings of “n” and “m” indices. ● The “e_step” is energy step. Its range is from 1 to n_level. ● The meanings “types” index <ul style="list-style-type: none"> ■ types = 1 for acoustic phonon. See section 6.1.1. ■ types = 2 for optical phonon with absorption process. See section 6.1.2. ■ types = 3 for optical phonon with emission process. See section 6.1.2. ■ types = 4 for SRS scattering with forward process. See section 6.1.3. ■ types = 5 for SRS scattering with backward process. See section 6.1.3. |

Table 6.1 Some variables that have been changed so as not to have a subband index

In this case, **NscatTypes** is equal to **1** (If only acoustic phonon), **3** (If acoustic and non-polar optical phonon; or if only non-polar optical phonon) and **5** (Acoustic phonon, Non-polar Optical phonon and SRS scatterings are included in the model). See [read_scattering_save_list.c](#).

6.1. Scattering mechanisms

Here we concentrate on **the scattering mechanisms for holes using an approximate scheme** as described in [Section 6.2-Tuan’s thesis](#).

6.1.1 Acoustic phonon scattering

The expression of intra-valley acoustic phonon scattering for holes is the same as that for electron counterpart. See [Equation \(A.28\)-Tuan’s thesis](#).

6.1.2 Optical phonon scattering

The **intra-valley** optical phonon scattering rate for holes takes an expression very similar to the electron case (See [Equation \(A.34\)-Tuan's thesis](#)) and it is given by

$$\Gamma_{n,m}^{v,v'}(k_x) = \frac{\pi \Xi_{iv0}^2}{\rho \omega_0} \left(n_{\mathbf{q}} + \frac{1}{2} \mp \frac{1}{2} \right) F_{n,m}^{v,v'} D_{1D} \Theta(E_f), \quad (6.1)$$

where E_f is given by

$$E_f = E_n - E_m + \frac{\sqrt{1 + 4\alpha\gamma(\mathbf{k}_x)} - 1}{2\alpha} \pm \hbar\omega_0. \quad (6.2)$$

The upper and lower notations in Equation (6.2) are for absorption and emission processes, respectively.

It is worth to note that the optical phonon scattering is an intra-valley scattering mechanism thus $v = v'$. For this reason, the value of “**types**” index in the scattering table is equal to **2** and **3** for absorption and emission processes, respectively. See Table 6.1.

6.1.3 SRS scattering

The SRS scattering rate is given by Equation (3.3) in [section 3.4.3](#), however, the SRS form factor need to be modified for hole inversions in silicon nanowire. The derivation of SRS form factor for hole is shown in [Appendix B-Tuan's thesis](#).

As mentioned in [Section 3.4](#) (Equation 3.2), if the spectra of the four interfaces are uncorrelated, the square matrix element can be summed. Furthermore, if we use the same root mean square (Δ) and correlation length (Λ) parameters for all interfaces, the square of the SRS form factor can be also summed for all interfaces, as below:

$$\left| F_{n,m}^{SRS(Total)} \right|^2 = \left| F_{n,m}^{SRS(Top)} \right|^2 + \left| F_{n,m}^{SRS(Bottom)} \right|^2 + \left| F_{n,m}^{SRS(Left)} \right|^2 + \left| F_{n,m}^{SRS(Right)} \right|^2 \quad (6.3)$$

As a result, for SRS form factor we don't need to consider "Top", "Bottom", "Left" and "Right" interfaces separately. It will make the scattering table a little bit simpler, as shown in Table 6.1. Consequently, we need only two values of "types" indices for SRS scattering rate (**types=4** for forward process and **types=5** for backward process).

(Please remember that in Figure 5.10 for electron case, due to considering the SRS form factor for four interfaces separately, we need eight values of "types" indices ((**types=8** and **9** for **Top** interface), (**types=10** and **11** for **Bottom** interface), (**types=12** and **13** for **Right** interface), (**types=14** and **15** for **Left** interface)).

6.1.4. The scattering parameters used for hole mobility calculation

| | Symbol | Value | Unit |
|---------------------------------------|-------------|-----------------------|--------|
| <i>Acoustic deformation potential</i> | Ξ_{ac} | 5.6 | [eV] |
| <i>Optical deformation potential</i> | Ξ_{opt} | 11.5×10^{10} | [eV/m] |
| <i>Optical phonon energy</i> | E_p | 0.0612 | [eV] |
| <i>SRS r.m.s height</i> | Δ | 0.55 | [nm] |
| <i>SRS correlation length</i> | Λ | 2.6 | [nm] |

Table 6.2 Scattering parameters used for hole mobility calculation

Table 6.2 shows the scattering parameters used for hole mobility calculation. They are taken from the reference [De Michielis, et al., "A Semianalytical Description of the Hole Band Structure in Inversion Layers for the Physically Based Modeling of pMOS Transistors," Electron Devices, IEEE Transactions on , vol.54, no.9, pp.2164-2173, Sept. 2007.](#)

6.2 Some changes in MSMC code

Basically, the MSMC code for hole is almost the same as that for electron counterpart. Infact it is simpler than MSMC code for electrons, because:

- We need to consider only one valley.

- The phonon form factor and especially the SRS form factor have been calculated by Professor Mincheol Shin's simulator.

6.2.1 The *read_data_for_Mobility()* function.

This function is to get data from 2D Schrodinger and Poisson solver written by Professor Mincheol Shin, they are included:

- The eigen energies
- The effective mass for each subband
- The line charge density
- The percentage population
- The phonon form factor: Thus we do not need to calculate it.
- The SRS form factor: Thus we do not need to calculate it.
- The inversion charge density: Thus we do not need to calculate it.

6.2.2 For other functions

We just need to pay attention to two important factors, as below

- Only one valley (**n_valley = 1**) needs to be considered.
- We need to use appropriate effective mass for each subband. Meaning that the **MassSub[i]** for **i-th** subband is used instead of m_l for valley 1 and m_t for valley 2 and 3 as in the electron case.

7. A fully self-consistent MSMC method for the modeling of Silicon Nanowire MOSFETs

As mentioned in the previous sections, our initial purpose is to make a fully self-consistent MSMC method for device simulation such as modeling of Silicon nanowire MOSFETs. In order to save time for future developers, this section is devoted to describe the main steps for this model. However, please note the following points, before moving forward.

- This part is written to the best of my knowledge however **there are no results for current-voltage characteristics obtained from our in-house code**. Therefore this part may be used as a suggestion and **it is up to a future developer if he or she wants to read this part or not**.
- The future developers need to fill in the details. (In this part, if needed we only show the pseudo-code).
- A brief review of semiclassical Monte Carlo device simulation for bulk MOSFETs is given in [Section 4.1.3-Tuan's thesis](#), and a more detail can be referred from the references therein. The source code of semiclassical Monte Carlo for Single Gate (SG) and Double Gate MOSFETs (DG MOSFETs) can be found [here](#). (It is just a “test model” for me to be familiar with Monte Carlo method).
- Our model is an extension of the Multi-subband Monte Carlo method for quasi-2DEG systems (See [section 4.2-Tuan's thesis](#) and the references therein).
- Some efforts are needed to complete “a detail picture”, especially how to enhance the performance of the model. (Because MSMC transport kernel is computationally expensive).

7.1. Overall Description of the Model

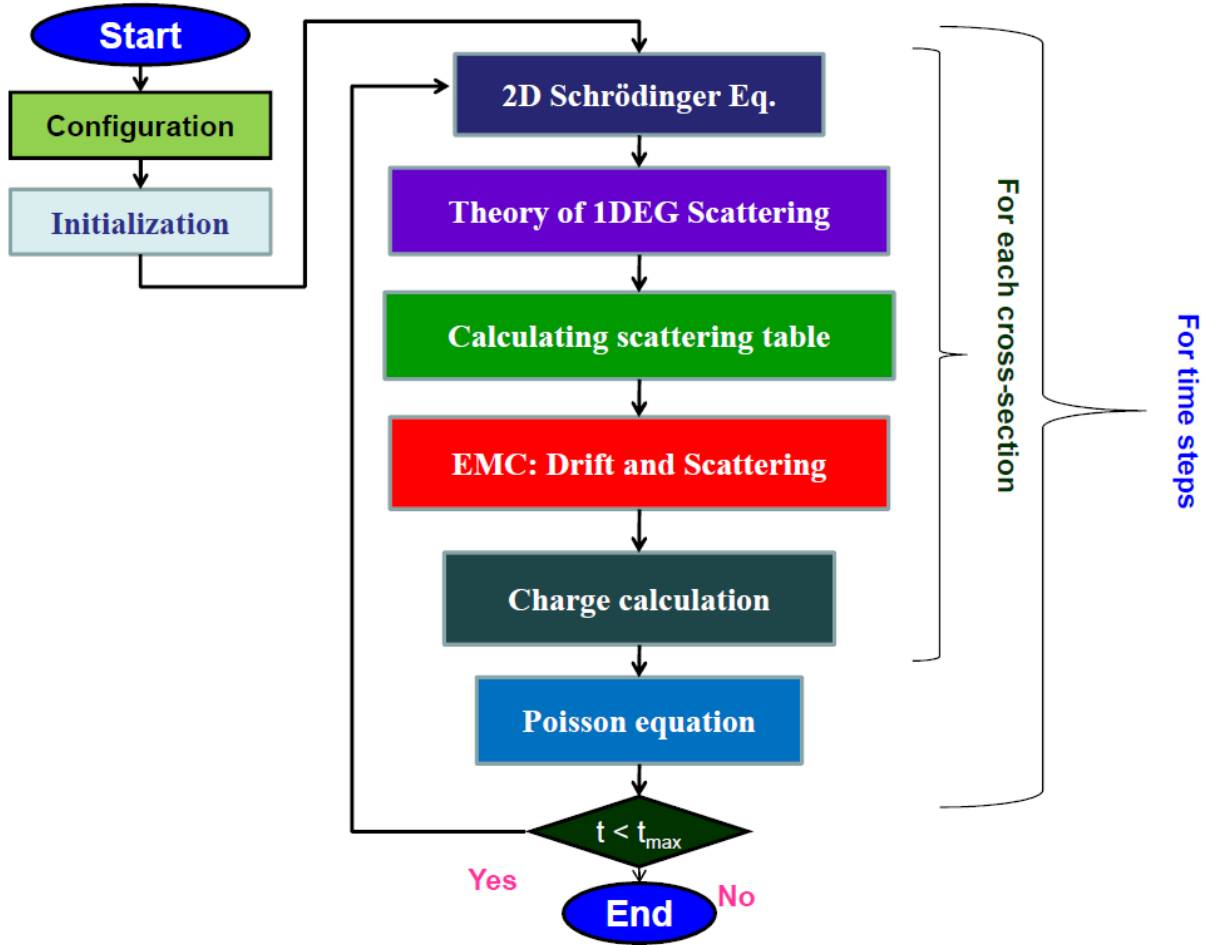


Figure 7. 1 Flowchart of the typical Multi-subband Monte Carlo for quasi-1DEG systems

Figure 7.1 shows a flowchart of the typical Multi-subband Monte Carlo for quasi-1DEG systems. The description below is very similar with the [section 4.2-Tuan's thesis](#).

The 3D device, for example silicon nanowire MOSFETs, is sliced along channel direction (x-direction). For each cross-section, the 2D Schrödinger equation is solved in the confinement plane (y-z plane). The thus-obtained eigen energy, $E_{s,v,i}$, and wave function, $\Psi_{s,v,i}(y,z)$, where s, v and i are the section, valley and subband indices, respectively, are used to calculate the 1DEG scattering rates for each section. In the MSMC transport kernel, the carrier is transport through sections due to the electric field, $F_{s,v,i}$, which is given by the x-derivative of eigen energy.

$$F_{s,v,i} = -\frac{dE_{s,v,i}}{dx} = -\frac{E_{s+1,v,i} - E_{s,v,i}}{x_{s+1} - x_s}. \quad (7.1)$$

The particle is drifted and scattered in the 1D electron gas EMC routine. Once the stationary is reached, the electron density for each section can be calculated, based on the electron population in each section, valley and subband, $N_{s,v,i}$, as

$$n_s(y, z) = \sum_v \sum_i \frac{N_{s,v,i}}{N_s} |\Psi_{s,v,i}(y, z)|^2, \text{ where } N_s = \sum_v \sum_i N_{s,v,i}. \quad (7.2)$$

The charge density from Eq. (7.2) is fed into 3D Poisson's equation to get a new potential, $V(x, y, z)$. This potential is used as an input for the 2D Schrödinger equation thus returning to a new MSMC iteration. The loop is iterated until the convergence is reached. By this approach, 2D Schrödinger, 3D Poisson and 1D BTE equations are solved self-consistently.

7.2. The Configuration module

This module is very similar to that described in the [section 2](#). (See Figure 2.1 from *material_param()* to *trapezoidal_weights()* functions). In this part, we need to consider all the parameters in the transport direction. Figure 7.2 shows the device co-ordinate system for silicon nanowire MOSFETs.

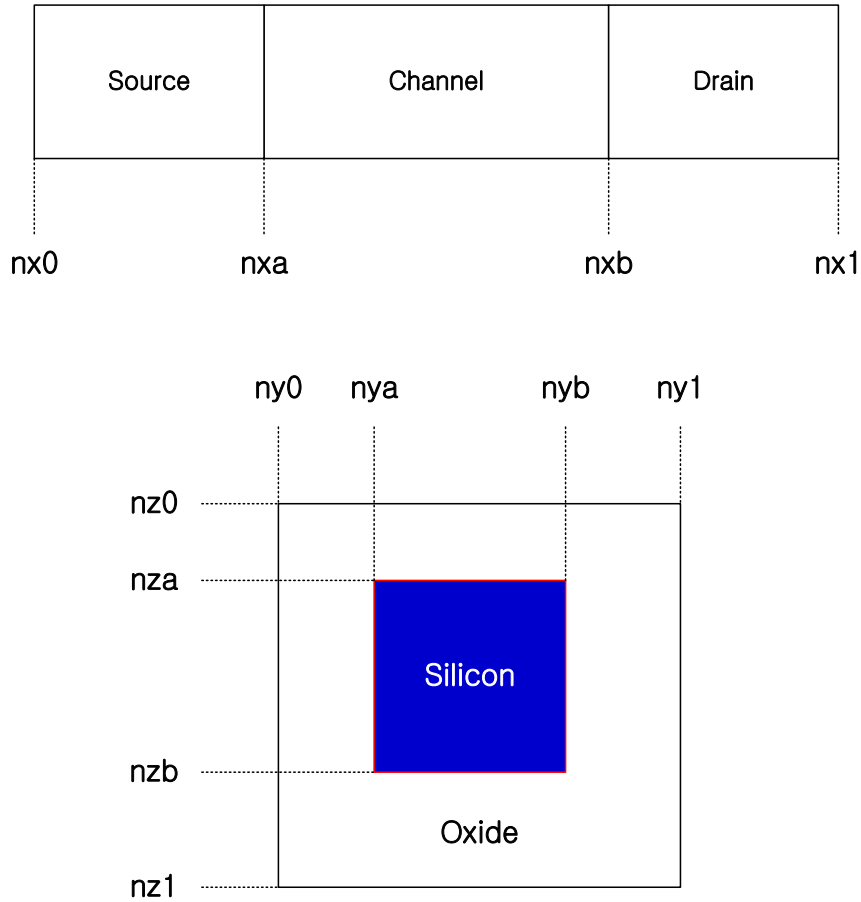


Figure 7. 2 The device co-ordinate system

In device simulation, it is needed to maintain the charge neutrality in the vicinity of the source and drain ohmic contacts, meaning that:

- Keeping the number of particles constant in the cells that constitute the Source and Drain contacts, as indicated in Figure 7.3.

Before each new time step, this condition is checked, and if needed an appropriate number of particles is injected (created) or deleted. Therefore three new functions may be needed to add to the source code: *source_drain_carrier_number()*, *delete_particles()* and *check_source_drain_contacts()*.

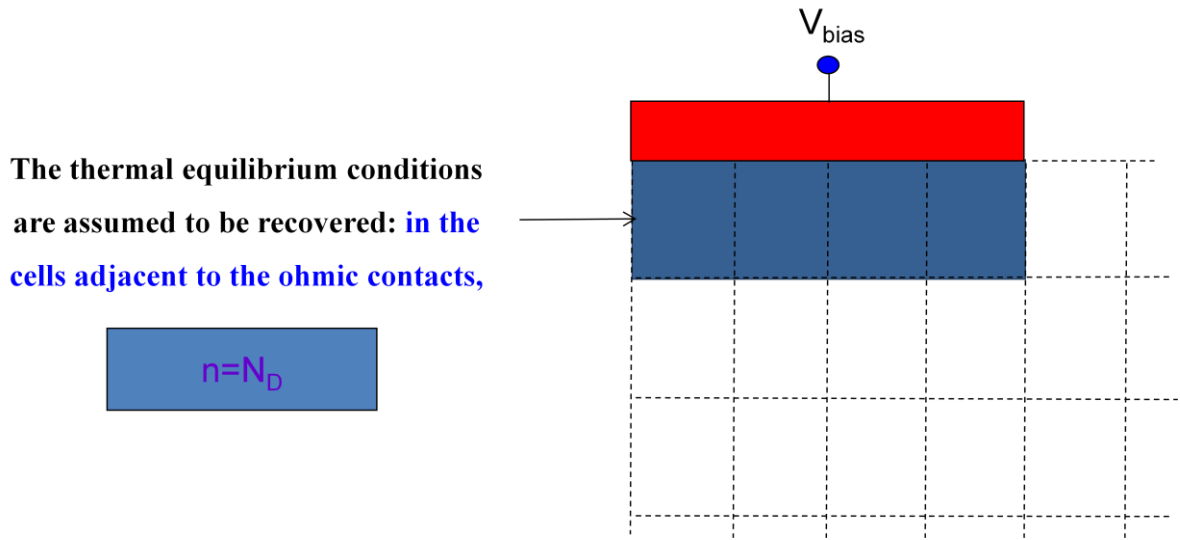


Figure 7.3 Schematic of mesh layout near the source contact. The shaded area is the charge neutral region.

The *source_drain_carrier_number()* function is to calculate the number of particles at source and drain contacts. The information is needed to keep the charge neutrality at the contact regions. The *delete_particles()* and *check_source_drain_contacts()* functions will be introduced later.

7.3. The Initialization module

In this module:

- To calculate the number of particles used in this simulator for the first time (**n_used**) and after that we know which “cell” the particle is resided in.
- And then the parameter of each particle is initialized in k-space and real-space
- Guess potential

7.3.1. Initialization in k-space

Firstly, electron energy is initialized based on Boltzmann distribution and then wave vector can be obtained. The electron distribution in each valley and subband can be initialized based on the percentage population of electron getting from mobility calculation. See [section 4](#) Figure 4.2).

7.3.2 Initialization in real-space

To initialize the position of a particle in real space.

Assuming that the particle is in “sth” section, the position of particle in the transport direction (**x_position**) can be defined based on grid location, as below.

```
if((s!=nx0) && (s!= nx1))
{
    x_position = ((double) (s)+random(&idum)-0.5)*mesh_size_x;
}
else if(s==nx0)
{
    x_position = 0.5*random(&idum)*mesh_size_x;
}
else if(s==nx_max)
{
    x_position = device_length-0.5*random(&idum)*mesh_size_x;
}
```

where **random(&idum)** is a random number, and **device_length** is the length of the device (including source, channel and drain).

7.3.3 Guess potential

The guess potential is used to solve 2D Schrodinger equation for the first time.

7.4. The 2D Schrodinger solver

For each sth cross-section, the 2D Schrodinger equation is solved in the quantization plane (y-z plane) to obtain subband structure.

It is worth to note that, in this case, the wave function and eigen energy should have the section-index. (Please see [section 2.11](#)).

7.5. The scattering table and normalization module

Please see [section 3](#).

Since we calculate the scattering rate for each cross-section, the “sth” section index needs to be taken into account. For example,

scat_table[s][valley][n][m][e_step][types],

flag_mech[s][types][valley],

i_valley[s][types][valley],

```
enerAdded[s][types][valley],
max_gm[s][valley][n].
```

The method to calculate the scattering rate and normalize table is very similar to that described in [section 3](#). Specifically, we calculate scattering table and normalize table for each cross-section.

For the first time, the free-flight time of a particle in “sth” section and valley “v” and subband “n” is initialized as follow

```
init_free_flight = -log(random(&idum))/max_gm[s][v][n];
```

7.6 Ensemble Monte Carlo

Basically, it is very similar to the EMC code that is described in [section 5.2](#). There are some variables that are needed to record at every time step.

- The number of particles goes out of the device at the source and drain contacts (**iss_out** and **idd_out**, respectively). See *check_boundary()* function.
- The number of particles is created at the source and drain contacts for keeping the charge neutrality at the ohmic contacts (**iss_cre** and **idd_cre**, respectively). See *check_source_drain_contacts()* function.
- The number of particles is eliminated at the source and drain contacts for keeping the charge neutrality at the ohmic contacts (**iss_eli** and **idd_eli**, respectively). See *check_source_drain_contacts()* function.

7.6.1. Drift process

It is very similar to the *drift()* function which is described in Figure 5.21 of the [section 5.2.2](#). However there are some notes as below:

- The electric field using to accelerate the particle (in each section) on each subband is given by the Equation (7.1).
- The position of electron, after drifting during “tau” time, should be updated

$$\begin{aligned}
x_position &= x_position + \frac{\hbar}{m^*} \tau \frac{k_x + 0.5\Delta k_x}{\sqrt{1 + 4\alpha E_{parabolic}}} \\
&= x_position + \frac{\hbar}{m^*} \tau \frac{k_x + 0.5\Delta k_x}{1 + 2\alpha E_{Non-parabolic}}
\end{aligned} \tag{7.3}$$

where Δk_x is the change of wave vector, α is non-parabolic factor and m^* is effective mass of the subband.

- After updating the **x_position**, we check boundary by calling the **check_boundary()** function.
- Inside the **check_boundary()** function

```

if (x_position < nx0) // The particle "n-th" is outside of the device at Source contact
{
    iss_out = iss_out + 1;
    valley[n] = 9; // Indicating that this particle will be deleted
}
if (x_position > nx1) // The particle "n-th" is outside of the device at Drain contact
{
    idd_out = idd_out + 1;
    valley[n] = 9; // Indicating that this particle will be deleted
}

```

It is worth to note that for electron the **valley[n] = 1, 2, or 3** if the n-th particle is reside in valley 1, 2 or 3, respectively. Therefore, in order to indicate that the n-th particle will be inactivated, we set **valley[n] = 9**.

7.6.2. Scattering process

It is very similar to the **scattering()** function described in [section 5.1.2](#). It is note that we should use the appropriate scattering table based on the section that the particle is reside in.

7.7. The **check_source_drain_contacts()** function

After running Ensemble Monte Carlo for each time step, we need to check the number of particle at the ohmic contacts. Assuming that the number of particles at source and drain contacts for charg neutrality are **n_source** and **n_drain**, respectively. And also assuming that after one time step, the number of particles at source and drain contact are **n_source_NEW** and **n_drain_NEW**.


```

// Part 1. Delete extra carriers at the source and drain contacts
if (n_source < n_source_NEW)
{
    // The particles n_source+1, n_source+2,...,n_source_NEW need to be
    // deleted hence set their valley index is equal to 9 (valley[] = 9)
    iss_eli = iss_eli + (n_source_NEW-n_source);
}
if (n_drain < n_drain_NEW)
{
    // The particles n_drain+1, n_drain+2,...,n_drain_NEW need to be
    // deleted hence set their valley index is equal to 9 (valley[] = 9)
    idd_eli = idd_eli + (n_drain_NEW-n_drain);
}

// Part 2. Create carriers at the source and drain contacts
if (n_source > n_source_NEW)
{
    // We need create (n_source - n_source_NEW) particles at the Source
    // contact thus we need to initialize their parameters in real space and k-
    // space
    iss_cre = iss_cre + (n_source-n_source_NEW);
}
if (n_drain > n_drain_NEW)
{
    // We need create (n_drain - n_drain_NEW) particles at the Drain
    // contact thus we need to initialize their parameters in real space and k-
    // space
    idd_cre = idd_cre + (n_drain-n_drain_NEW);
}

```

7.8 The *delete_particles()* function

The particles, which have **valley[] = 9**, need to be deleted. One simple way to delete particles is shown in Figure 7.4 and described as below.

- Step 1: Assuming that we have 10 particles (**n_used = 10**) in which the 5th and 8th particles have **valley index = 9**.
- Step 2 and step 3: Moving the parameters of 10th particle to that of the 5th particle and then inactivate 10th particle by setting its valley index is equal to 9.
- Step 4 and step 5: Moving the parameters of 9th particle to that of the 8th particle and then inactivate 9th particle by setting its valley index is equal to 9.
- At the end we have **n_used = 8** (Because only 8 particles have valley indices that are different with 9). The **n_used** value will be used for the Ensemble Monte Carlo for the next time step.

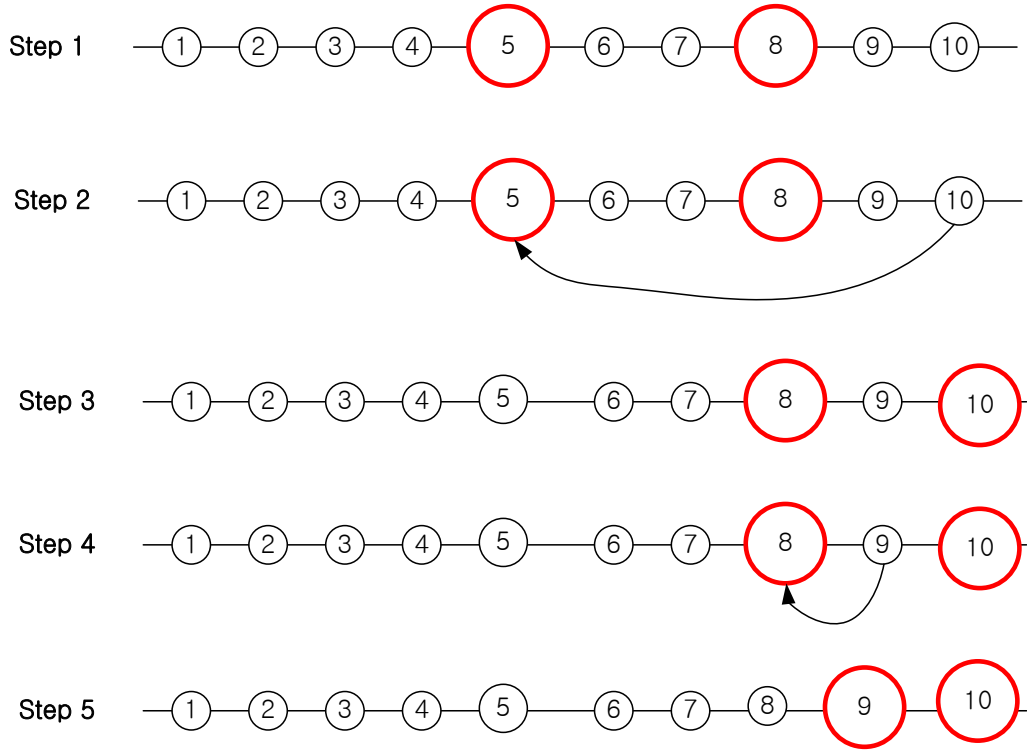


Figure 7. 4 A simple method to delete particles

7.9. Charge calculation

The charge density is calculated by using Equation (7.2).

7.10. Poisson Equation

The 3D Poisson's equation can be solved by using our in-house Poisson solver which is written by Professor Mincheol Shin.

7.11. Current calculation

By keeping track of the charges entering and exiting each terminal/contact, the net number of charges over a period of the simulation can be used to calculate the terminal current.

For example, the net charge entering and exiting at the source and drain contacts in one time step is given by

```
source_factor = (double) (iss_out+iss_eli-iss_cre);  
drain_factor  = (double) (idd_out+idd_eli-idd_cre);
```

7.12. Enhancement of the model

Because MSMC approach for modeling Silicon nanowire MOSFETs is computationally expensive, some techniques should be used to relieve this difficulty.

As mentioned in the reference [“Efficient multi sub-band Monte Carlo simulation of nano-scaled Double Gate MOSFETs - Saint-Martin – 2006”](#), to avoid computational burden, the Schrodinger equation is solved every N time steps (e.g. N = 100) and for every other step, the eigen energies is updated using a perturbative approach at the first order (they found that there is no need to correct the wave functions). Since the scattering table is updated whenever the new Schrodinger solutions are obtained, the table is also re-constructed at the same intervals.