eduPIC: an introductory particle based code for radio-frequency plasma simulation MANUAL

Zoltán Donkó, Aranka Derzsi, Máté Vass, Benedek Horváth,
Sebastian Wilczek, Botond Hartmann, Peter Hartmann

June 10, 2021

1 Introduction

eduPIC is an open source "starting tool" that intends to assist the education and research of those interested in particle based plasma simulations. Our intention has been to keep this code as simple and as transparent as possible; optimization and further development is left for the interested readers. Therefore, the code that we provide has a length of ~ 1000 lines, in a single file. Besides the "simulation core" the code includes some discharge diagnostics features, which can be further expanded.

The code focuses on Capacitively Coupled Plasmas (CCPs) and is based on the Particle-in-Cell/Monte Carlo Collisions (PIC/MCC) technique – an introduction to this technique and the description of the physical basis of the eduPIC code has been provided in [1] and is not repeated in this Manual. For a complete understanding of the operation of the code the description of the code parts provided in this Manual should be simultaneously studied with the respective parts of Ref. [1].

The code is of the type "1d3v", meaning that the physical setting assumes a 1-dimensional spatial variation of the discharge characteristics (within a bounded space), while the particles are traced in the 3-dimensional velocity space. The code includes a basic cross section set for argon (Ar) gas, see [1], discharges in other gases can be simulated with a proper change of the cross section set.

2 Usage

2.1 Compilation

The eduPIC C code has been tested on an Ubuntu Linux computing cluster on nodes equipped with x86-64 based Intel® Xeon and AMD® Threadripper CPUs. The compilation of the C code can be performed with both `icpc` Intel C++ compiler (ver. 2021.1.2, part of the freely available Intel oneAPI Base Toolkit) and `g++` from the GNU Compiler Collection (ver. 9.3.0). Best performance was achieved on Intel architecture using:

```
icpc -fast -o edupic eduPIC.cc
```

2.2 Program execution and files created

The code can be invoked in a terminal window as

```
./edupic arg1
```

To start a new simulation, running an initialization cycle is required by setting `arg1` to 0, i.e. executing

```
./edupic 0
```

In this case, seeding of a number of initial particles (1000 of both species, as default), the simulation of a single radio-frequency (RF) cycle and saving of the state of the system (to a binary file `picdata.bin`) is executed.

The code can be run, subsequently, for any number of additional RF cycles specified by `arg1`, e.g., for 500 cycles, as

```
./edupic 500
```

In this run, the previously saved state of the system is restored, the given number of RF cycles is simulated, and at the end of the run, the state of the system is saved. This procedure can be repeated any number of times, the simulation always continues from the previously saved state of the system. During these runs, the time evolution of the number of superparticles is stored in a file named `conv.dat`, but no other data (results) are saved. The content of this file is illustrated in figure 1 for the reference conditions listed in table 1. The code simulates a CCP driven by a single-frequency excitation voltage waveform.

Table 1: Set of "default" parameters for the reference simulation run. (The parameters have these values in the code that can be downloaded.)

Quantity	Symbol	In the code	Value
Driving voltage amplitude	V_0	`VOLTAGE`	250 V
Driving frequency	f	`FREQUENCY`	13.56 MHz
Superparticle weight	W	`WEIGHT`	7×10^4
Electrode gap	L	`L`	25 mm
Ar pressure	p	`PRESSURE`	10 Pa
Gas temperature	T_g	`TEMPERATURE`	350 K
Number of grid points	N_g	`N_G`	400
Number of time steps / RF cycle	N_t	`N_T`	4000

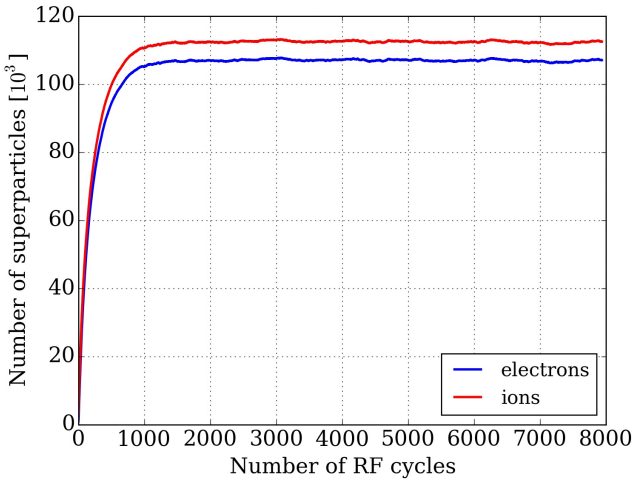

Figure 1: The number of superparticles as a function of the number of RF cycles (file: `conv.dat`). Discharge conditions: argon $p = 10$ Pa, $L = 25$ mm, $T_g = 350$ K, $V = 250$ V and $f = 13.56$ MHz [1].

As one can observe in figure 1, the number of superparticles grows in the simulation from the initial value of 1000 to $\approx 110\,000$ during ~ 1500 RF cycles. The number of RF periods to reach convergence depends strongly on the discharge conditions (e.g., pressure, electrode gap).

Measurements on the system should be executed only after reaching convergence. The measurement mode can be activated by specifying a second command line argument `m` when the code is invoked, e.g., a measurement for 1000 RF cycles is carried out as

```
./edupic 1000 m
```

Here, all built-in diagnostics are turned on. The number of RF cycles for which measurements are run affects the quality of the statistics of the results. Therefore, we recommend using at least ~ 1000 RF cycles to obtain results with good signal to noise ratio. When measurements on the system are taken, the code saves the data into the files listed in table 2.

`density.dat`	Time-averaged density distributions of electrons (second column) and ions (third column) as a function of the position (first column)
`eepf.dat`	Time-averaged EEPF in the central 10% spatial domain of the discharge (second column) as a function of the energy (first column). The data are normalized corresponding to (the discretized form of) $\int f(\varepsilon)\sqrt{\varepsilon}\mathrm{d}\varepsilon = 1$
`ifed.dat`	Time-averaged IFEDF at the powered (second column) and grounded (third column) electrode as a function of the energy (first column). The data are normalized corresponding to (the discretized form of) $\int F(\varepsilon)\mathrm{d}\varepsilon = 1$
`pot_xt.dat`	Spatio-temporal distribution of the potential
`efield_xt.dat`	Spatio-temporal distribution of the electric field
`ne_xt.dat`	Spatio-temporal distribution of the electron density
`ni_xt.dat`	Spatio-temporal distribution of the ion density
`je_xt.dat`	Spatio-temporal distribution of the electron current density
`ji_xt.dat`	Spatio-temporal distribution of the ion current density
`powere_xt.dat`	Spatio-temporal distribution of the power absorption by the electrons
`poweri_xt.dat`	Spatio-temporal distribution of the power absorption by the ions
`meanee_xt.dat`	Spatio-temporal distribution of the mean electron energy
`meanei_xt.dat`	Spatio-temporal distribution of the mean ion energy
`ioniz_xt.dat`	Spatio-temporal distribution of the ionization rate

Table 2: Data files created by the eduPIC code in "measurement" mode.

The "xt files" that store the spatio-temporal variation of the given quantities contain a number of rows equal to the number of spatial grid points `N_G`. The number of columns represents the temporal variation of these quantities. In order to improve the signal to noise ratio of the data, the number of time steps per RF cycle, `N_T`, is binned to a lower number, `N_XT`. The number of time steps binned for the xt files can be set by the variable `N_BIN` in the code. (At this point, care should be taken to ensure that `N_T` is an integer multiple of `N_BIN`.) All the output data are in SI units, except for the mean electron energy, as well as for the EEPF and the IFEDF, for which eV units are used.

Following a run in the "measurement" mode, the code saves a file named `info.txt`. This file contains the operation parameters and simulation settings as a record of the run. The file also displays information about the stability and accuracy conditions. First, the conditions concerning the relation of the grid spacing to the Debye length, the relation of the time step to the electron plasma frequency, and the collision probabilities during a time step.

Whenever any of these checks fail, an error message is issued and no further diagnostics data are saved. It is very important to realise that the evaluation of the stability and accuracy criteria is meaningful only when measurements are taken over the converged state. The initial setting of the simulation parameters (like the grid size and the number of the time steps) is normally based on an educated guess, these parameters can be refined later according to the information contained in `info.txt`. In case of need of modifications of the parameter settings, the code has to be re-compiled, the simulation has to be re-converged, and the stability and accuracy criteria have to be checked again.

In case the above conditions are met, the diagnostics data listed above are saved to the respective files, and the maximum electron energy for which the Courant–Friedrichs–Lewy condition still holds at the actual values of `N_G` and `N_T`, is also displayed in `info.txt`. To make sure that this condition holds for the *vast majority* of the electrons, one has to confirm that the EEPF decays to a small value at this energy. This is not done automatically in the code, the procedure is left to the user. We advise to observe the EEPF obtained in the center of the plasma, and to use a threshold value of $f(\varepsilon) \sim 10^{-6}$ eV$^{-3/2}$. For a more rigorous check, the complete space- and time-resolved EEPF should be considered.

Additional information about the particle characteristics at the electrodes and about the power absorption by the electrons and ions is also saved to `info.txt` at the end of the simulation. The latter is computed as the spatial and temporal average of the product $j(x,t)E(x,t)$. This product is also saved to the files `powere_xt.dat` and `poweri_xt.dat` (corresponding to the power absorption rate by the electrons and ions, respectively) with spatial and temporal resolution.

Without any change to the code (including the parameter settings) the simulation should result the same data as plotted in figures 10–13 of [1]. Such a "test run" is recommended in order to get acquainted with the compilation and the running of the code to the converged state, and to carry out measurements.

3 Description of the code

Below, we provide a detailed description of the individual parts of the eduPIC code. No additional explanation is given in the cases where variables or operations are explained by comments in the code lines.

Inclusion of the header files required.

```
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <cstdbool>
#include <cmath>
#include <ctime>
#include <random>
```

Defining the default function name space

```
using namespace::std;
```

Declaration of mathematical and physical constants.

```
const double      PI              = 3.141592653589793;          // mathematical constant Pi
const double      TWO_PI          = 2.0 * PI;                   // two times Pi
const double      E_CHARGE        = 1.60217662e-19;             // electron charge [C]
const double      EV_TO_J         = E_CHARGE;                   // eV <-> Joule conversion factor
const double      E_MASS          = 9.10938356e-31;             // mass of electron [kg]
const double      AR_MASS         = 6.63352090e-26;             // mass of argon atom [kg]
const double      MU_ARAR         = AR_MASS / 2.0;              // reduced mass of two argon atoms [kg]
const double      K_BOLTZMANN     = 1.38064852e-23;             // Boltzmann's constant [J/K]
const double      EPSILON0        = 8.85418781e-12;             // permittivity of free space [F/m]
```

Declaration of constants that control the execution of the simulation. These include the discharge conditions (frequency, voltage, pressure, electrode gap and temperature), as well as the resolution of the temporal and spatial grids, the superparticle weight, the electrode area assumed in the simulation as well as the number of electron and ion superparticles that are seeded within the discharge gap upon the initialization of the simulation. The electrode area is needed only to connect the number of superparticles and the density of the real particles, see [1]. The number specified here (1 cm^2, in line 9) should be kept unchanged.

```
const int         N_G             = 400;                        // number of grid points
const int         N_T             = 4000;                       // time steps within an RF period
const double      FREQUENCY       = 13.56e6;                    // driving frequency [Hz]
const double      VOLTAGE         = 250.0;                      // voltage amplitude [V]
const double      L               = 0.025;                      // electrode gap [m]
const double      PRESSURE        = 10.0;                       // gas pressure [Pa]
const double      TEMPERATURE     = 350.0;                      // background gas temperature [K]
const double      WEIGHT          = 7.0e4;                      // weight of superparticles
const double      ELECTRODE_AREA  = 1.0e-4;                     // (fictive) electrode area [m^2]
const int         N_INIT          = 1000;                       // number of initial electrons and ions
```

Additional constants derived from the parameters specified above. `DT_E` and `DT_I` are the time steps of electrons and ions. `DT_E` is also the basic time step of for the PIC/MCC cycle (shown in figure 2). The two time steps are related via the subcycling parameter `N_SUB`: ions are treated only in every `N_SUB`-th time step, which is possible due to their higher mass with respect to the electrons. Subcycling accelerates the computation considerably. The same time step for electrons and ions is recovered at `N_SUB`=1. Too large values for `N_SUB` are to be avoided as the accuracy of the calculations can degrade when ions are not treated frequently enough (see the stability and accuracy criteria discussed in [1]).

`DX` is the division of the (uniform) computational grid. The inverse of this quantity `INV_DX` is also defined here as in many cases division by `DX` is required in the computations. In these cases, the computationally more efficient operation of multiplication by `INV_DX` is carried out.

```
const double      PERIOD          = 1.0 / FREQUENCY;                    // RF period length [s]
const double      DT_E            = PERIOD / (double)(N_T);             // electron time step [s]
```

```
const int         N_SUB           = 20;                                   // ions move only in these cycles (subcycling)
const double      DT_I            = N_SUB * DT_E;                         // ion time step [s]
const double      DX              = L / (double)(N_G - 1);                // spatial grid division [m]
const double      INV_DX          = 1.0 / DX;                             // inverse of spatial grid size [1/m]
const double      GAS_DENSITY     = PRESSURE / (K_BOLTZMANN * TEMPERATURE);   // background gas density [1/m^3]
const double      OMEGA           = TWO_PI * FREQUENCY;                   // angular frequency [rad/s]
```

Some important constants related to the cross sections of elementary processes and declaration of arrays for the cross section data. Lines 1–6 define the number of elementary processes considered (in this case 5) and the identifiers of these processes. Lines 7–8 specify the energy thresholds for the inelastic electron - Ar atom collision processes, while in lines 9–14 the data structures for the storage of cross section data are declared.

```
const int         N_CS            = 5;                                    // total number of processes / cross sections
const int         E_ELA           = 0;                                    // process identifier: electron/elastic
const int         E_EXC           = 1;                                    // process identifier: electron/excitation
const int         E_ION           = 2;                                    // process identifier: electron/ionization
const int         I_ISO           = 3;                                    // process identifier: ion/elastic/isotropic
const int         I_BACK          = 4;                                    // process identifier: ion/elastic/backscattering
const double      E_EXC_TH        = 11.5;                                 // electron impact excitation threshold [eV]
const double      E_ION_TH        = 15.8;                                 // electron impact ionization threshold [eV]
const int         CS_RANGES       = 1000000;                              // number of entries in cross section arrays
const double      DE_CS           = 0.001;                                // energy division in cross section arrays [eV]
typedef float     cross_section[CS_RANGES];                               // cross section array
cross_section     sigma[N_CS];                                            // set of cross section arrays
cross_section     sigma_tot_e;                                            // total macroscopic cross section of electrons
cross_section     sigma_tot_i;                                            // total macroscopic cross section of ions
```

Declaration of the `particle_vector` type for arrays that store the space and velocity coordinates of the various particles. These arrays have a size `MAX_NP`, this is the maximum number of superparticles. `N_e` and `N_i` are the actual numbers of the electron and ion superparticles (initialized with zero values). These values should never exceed `MAX_NP` during the simulation runs.

```
const int         MAX_N_P = 1000000;                                      // maximum number of particles (electrons / ions)
typedef double    particle_vector[MAX_N_P];                               // array for particle properties
int               N_e = 0;                                                // number of electrons
int               N_i = 0;                                                // number of ions
particle_vector   x_e, vx_e, vy_e, vz_e;                                  // coordinates of electrons (one spatial, three velocity components)
particle_vector   x_i, vx_i, vy_i, vz_i;                                  // coordinates of ions (one spatial, three velocity components)
```

Declaration of the spatial grid and the quantities calculated at the points of this grid. `efield`, `potential` `e_density`, and `i_density` store the values of the electric field, the potential, the electron density, and the ion density, respectively, in the given time step. `cumul_e_density` and `cumul_i_density` accumulate the electron and the ion density during the simulation run and are are used for the computation of the time averaged values of these quantities (via dividing them by the number of time steps of the whole simulation) at the saving of these data to the file `density.dat` by the function `save_density()` (see later).

```
typedef double    xvector[N_G];                                           // array for quantities defined at gird points
xvector           efield,pot;                                             // electric field and potential
xvector           e_density,i_density;                                    // electron and ion densities
xvector           cumul_e_density,cumul_i_density;                        // cumulative densities
```

Counters for the electrons and ions that reach the powered and grounded electrodes during the simulation run. These quantities can be used for the calculation of particle fluxes to the electrodes.

```
typedef unsigned long long int Ullong;                                    // compact name for 64 bit unsigned integer
Ullong        N_e_abs_pow  = 0;                                           // counter for electrons absorbed at the powered electrode
Ullong        N_e_abs_gnd  = 0;                                           // counter for electrons absorbed at the grounded electrode
Ullong        N_i_abs_pow  = 0;                                           // counter for ions absorbed at the powered electrode
Ullong        N_i_abs_gnd  = 0;                                           // counter for ions absorbed at the grounded electrode
```

Declaration of the quantities needed for the calculation of the electron energy probability function (EEPF). Data for this function are accumulated in the array `eepf`, which is initialized with zero values, accumulates data

during the simulation run, and is normalized properly upon saving of the data by the function `save_eepf()` (see later).

```
const int    N_EEPF  = 2000;                                    // number of energy bins in Electron Energy
    Probability Function (EEPF)
const double DE_EEPF = 0.05;                                    // resolution of EEPF [eV]
typedef double eepf_vector[N_EEPF];                             // array for EEPF
eepf_vector eepf     = {0.0};                                   // time integrated EEPF in the center of the
    plasma
```

Declaration of the quantities needed for the calculation of the ion flux-energy distribution (IFED) function. Data for this function are accumulated in two arrays corresponding the two electrodes (`ifed_pow` and `idef_gnd`), which are normalized properly upon saving of the data. The values of the mean energy of the ions reaching both electrodes are also derived from these functions in the function `save_ifed()` (see later).

```
const int    N_IFED   = 200;                                    // number of energy bins in Ion Flux-Energy
    Distributions (IFEDs)
const double DE_IFED  = 1.0;                                    // resolution of IFEDs [eV]
typedef int  ifed_vector[N_IFED];                               // array for IFEDs
ifed_vector  ifed_pow = {0};                                    // IFED at the powered electrode
ifed_vector  ifed_gnd = {0};                                    // IFED at the grounded electrode
double       mean_i_energy_pow;                                 // mean ion energy at the powered electrode
double       mean_i_energy_gnd;                                 // mean ion energy at the grounded electrode
```

Declarations related to the data for the spatio-temporal maps, which are computed for several discharge characteristics. These data reside in two-dimensional arrays, which are initialized with zero values. The number of grid points in space is the same as for the main simulation grid (`N_G`), the number of grid points in time is reduced with respect to the number of time steps (`N_T`) by a factor of `N_BIN`. This binning can optimise the trade-off between time-resolution and signal to noise ratio of the data.

```
const int N_BIN                    = 20;                        // number of time steps binned for the XT
    distributions
const int N_XT                     = N_T / N_BIN;               // number of spatial bins for the XT
    distributions
typedef double xt_distr[N_G][N_XT];                             // array for XT distributions (decimal numbers)
xt_distr pot_xt                    = {0.0};                     // XT distribution of the potential
xt_distr efield_xt                 = {0.0};                     // XT distribution of the electric field
xt_distr ne_xt                     = {0.0};                     // XT distribution of the electron density
xt_distr ni_xt                     = {0.0};                     // XT distribution of the ion density
xt_distr ue_xt                     = {0.0};                     // XT distribution of the mean electron
    velocity
xt_distr ui_xt                     = {0.0};                     // XT distribution of the mean ion velocity
xt_distr je_xt                     = {0.0};                     // XT distribution of the electron current
    density
xt_distr ji_xt                     = {0.0};                     // XT distribution of the ion current density
xt_distr powere_xt                 = {0.0};                     // XT distribution of the electron powering (
    power absorption) rate
xt_distr poweri_xt                 = {0.0};                     // XT distribution of the ion powering (power
    absorption) rate
xt_distr meanee_xt                 = {0.0};                     // XT distribution of the mean electron energy
xt_distr meanei_xt                 = {0.0};                     // XT distribution of the mean ion energy
xt_distr counter_e_xt              = {0.0};                     // XT counter for electron properties
xt_distr counter_i_xt              = {0.0};                     // XT counter for ion properties
xt_distr ioniz_rate_xt             = {0.0};                     // XT distribution of the ionisation rate
```

Declaration of additional variables for diagnostics and program control purposes. Lines 1–2 contain variables that allow computation of the mean energy of the electrons in the centre of the discharge gap. `N_e_coll` and `N_i_coll` count the total number of collision of electron and ion superparticles, these are used in the computation of the collision frequencies of the two species. The variables in line 6 are used in the `main()` function of the code. The Boolean flag `measurement_mode` allows data collection (measurements on the system) when it is set to `true`. Measurements should be carried out only after reaching the converged state of the simulation.

```
double   mean_energy_accu_center    = 0;                        // mean electron energy accumulator in the
    center of the gap
Ullong   mean_energy_counter_center = 0;                        // mean electron energy counter in the center
    of the gap
Ullong   N_e_coll                   = 0;                        // counter for electron collisions
Ullong   N_i_coll                   = 0;                        // counter for ion collisions
double   Time;                                                  // total simulated time (from the beginning of
    the simulation)
int      cycle,no_of_cycles,cycles_done;                        // current cycle and total cycles in the run,
    cycles completed
int      arg1;                                                  // used for reading command line arguments
char     st0[80];                                               // used for reading command line arguments
FILE     *datafile;                                             // used for saving data
bool     measurement_mode;                                      // flag that controls measurements and data
    saving
```

The code utilizes the Mersenne Twister (MT) 19937 random number generator (RNG), included in the C++ standard library (beginning with version C++ 11). The same RNG is used to generate random samples (i) from the uniform distribution on the [0,1) interval for several purposes, and (ii) from the normal distribution for assigning velocity components of background gas atoms in thermal equilibrium (Maxwell-Boltzmann distribution), when potential collision partners have to be selected in the case of ion-atom collisions. The MT generator is initialized with the standard C++ random device.

```
//------------------------------------------------------------------------//
// C++ Mersenne Twister 19937 generator                                   //
// R01(MTgen) will genarate uniform distribution over [0,1) interval      //
// RMB(MTgen) will generate Maxwell-Boltzmann distribution (of gas atoms) //
//------------------------------------------------------------------------//

std::random_device rd{};
std::mt19937 MTgen(rd());
std::uniform_real_distribution<> R01(0.0, 1.0);
std::normal_distribution<> RMB(0.0,sqrt(K_BOLTZMANN * TEMPERATURE / AR_MASS));
```

Pre-calculation and filling of the cross section arrays for the electrons. The calculation is based on the analytic forms provided by Phelps and Petrović [2]. This set includes the elastic momentum transfer cross section, one (lumped) excitation cross section that represents the sum of all excitation cross sections, and the ionization cross section. The data are stored in the arrays in units of m^2, as a function of the energy of the electron in the laboratory frame. The energy resolution of the data is `DE_CS`.

Via storing the data in pre-calculated arrays one can avoid the need for evaluating very frequently the quite complicated analytic forms, this way improving computational efficiency.

```
//------------------------------------------------------------------------//
//  electron cross sections: A V Phelps & Z Lj Petrovic, PSST 8 R21 (1999) //
//------------------------------------------------------------------------//

void set_electron_cross_sections_ar(void){
    int    i;
    double en,qmel,qexc,qion;

    printf(">> eduPIC: Setting e- / Ar cross sections\n");
    for(i=0; i<CS_RANGES; i++){
        if (i == 0) {en = DE_CS;} else {en = DE_CS * i;}                          // electron energy
        qmel = fabs(6.0 / pow(1.0 + (en/0.1) + pow(en/0.6,2.0), 3.3)
                    - 1.1 * pow(en, 1.4) / (1.0 + pow(en/15.0, 1.2)) / sqrt(1.0 + pow(en/5.5, 2.5) + pow(en
    /60.0, 4.1)))
        + 0.05 / pow(1.0 + en/10.0, 2.0) + 0.01 * pow(en, 3.0) / (1.0 + pow(en/12.0, 6.0));
        if (en > E_EXC_TH)
            qexc = 0.034 * pow(en-11.5, 1.1) * (1.0 + pow(en/15.0, 2.8)) / (1.0 + pow(en/23.0, 5.5))
            + 0.023 * (en-11.5) / pow(1.0 + en/80.0, 1.9);
        else
            qexc = 0;
        if (en > E_ION_TH)
            qion = 970.0 * (en-15.8) / pow(70.0 + en, 2.0) + 0.06 * pow(en-15.8, 2.0) * exp(-en/9);
        else
            qion = 0;
        sigma[E_ELA][i] = qmel * 1.0e-20;       // cross section for e- / Ar elastic collision
        sigma[E_EXC][i] = qexc * 1.0e-20;       // cross section for e- / Ar excitation
        sigma[E_ION][i] = qion * 1.0e-20;       // cross section for e- / Ar ionization
    }
}
```

Pre-calculation and filling of the cross section arrays for the argon ions. The calculation is based on the analytic forms provided by A. V. Phelps [3]. The data are stored as a function of the energy in the centre-of-mass reference frame, in units of m^2. The energy resolution of the data is `DE_CS`.

```
//------------------------------------------------------------------------//
//  ion cross sections: A. V. Phelps, J. Appl. Phys. 76, 747 (1994)       //
//------------------------------------------------------------------------//

void set_ion_cross_sections_ar(void){
    int    i;
    double e_com,e_lab,qmom,qback,qiso;

    printf(">> eduPIC: Setting Ar+ / Ar cross sections\n");
    for(i=0; i<CS_RANGES; i++){
        if (i == 0) {e_com = DE_CS;} else {e_com = DE_CS * i;}            // ion energy in the center of
    mass frame of reference
        e_lab = 2.0 * e_com;                                              // ion energy in the laboratory
    frame of reference
        qmom  = 1.15e-18 * pow(e_lab,-0.1) * pow(1.0 + 0.015 / e_lab, 0.6);
        qiso  = 2e-19 * pow(e_lab,-0.5) / (1.0 + e_lab) + 3e-19 * e_lab / pow(1.0 + e_lab / 3.0, 2.0);
        qback = (qmom-qiso) / 2.0;
        sigma[I_ISO][i]  = qiso;            // cross section for Ar+ / Ar isotropic part of elastic
    scattering
        sigma[I_BACK][i] = qback;           // cross section for Ar+ / Ar backward elastic scattering
```

```
    }
}
```

Calculation of the total macroscopic cross sections (which are the sum of all cross sections for a given species, multiplied with the gas density) for both species: upon the computation of the collision probabilities of the particles these total macroscopic cross sections are used and it is computationally more efficient to pre-calculate these arrays at the beginning of the simulation run.

```
//------------------------------------------------------------------------//
//  calculation of total cross sections for electrons and ions            //
//------------------------------------------------------------------------//

void calc_total_cross_sections(void){
    int i;

    for(i=0; i<CS_RANGES; i++){
        sigma_tot_e[i] = (sigma[E_ELA][i] + sigma[E_EXC][i] + sigma[E_ION][i]) * GAS_DENSITY;   // total
     macroscopic cross section of electrons
        sigma_tot_i[i] = (sigma[I_ISO][i] + sigma[I_BACK][i]) * GAS_DENSITY;                    // total
     macroscopic cross section of ions
    }
}
```

Test of the cross sections: the use of this function is optional, it writes all the different cross sections to a data file (`cross_sections.dat`), from which the energy dependent cross sections can be plotted and inspected. It is advisable to use this feature of the code after changing the cross section set.

```
//------------------------------------------------------------------------//
//  test of cross sections for electrons and ions                         //
//------------------------------------------------------------------------//

void test_cross_sections(void){
    FILE  * f;
    int   i,j;

    f = fopen("cross_sections.dat","w");        // cross sections saved in data file: cross_sections.dat
    for(i=0; i<CS_RANGES; i++){
        fprintf(f,"%12.4f ",i*DE_CS);
        for(j=0; j<N_CS; j++) fprintf(f,"%14e ",sigma[j][i]);
        fprintf(f,"\n");
    }
    fclose(f);
}
```

Finding of the maximum of the collision frequencies for the electrons and for the ions, based on their total collision cross sections. These quantities are needed for checking the maximum collision probability per time step of the respective species. These probabilities have to be kept low in order to minimise the chance of more than one collision per time step (see the section on stability and accuracy in [1]). The values corresponding to the actual simulation settings appear in the file `info.txt`.

```
//------------------------------------------------------------------------//
// find upper limit of collision frequencies                              //
//------------------------------------------------------------------------//

double max_electron_coll_freq (void){
    int i;
    double e,v,nu,nu_max;
    nu_max = 0;
    for(i=0; i<CS_RANGES; i++){
        e  = i * DE_CS;
        v  = sqrt(2.0 * e * EV_TO_J / E_MASS);
        nu = v * sigma_tot_e[i];
        if (nu > nu_max) {nu_max = nu;}
    }
    return nu_max;
}

double max_ion_coll_freq (void){
    int i;
    double e,g,nu,nu_max;
    nu_max = 0;
    for(i=0; i<CS_RANGES; i++){
        e  = i * DE_CS;
        g  = sqrt(2.0 * e * EV_TO_J / MU_ARAR);
        nu = g * sigma_tot_i[i];
        if (nu > nu_max) nu_max = nu;
    }
    return nu_max;
}
```

Upon the initialization of the simulation this function seeds an equal number (`nseed`) of electron and ion superparticles at random positions within the electrode gap, with zero initial velocity.

```
//------------------------------------------------------------------------//
// initialization of the simulation by placing a given number of          //
// electrons and ions at random positions between the electrodes          //
//------------------------------------------------------------------------//

void init(int nseed){
    int i;

    for (i=0; i<nseed; i++){
        x_e[i]  = L * R01(MTgen);               // initial random position of the electron
        vx_e[i] = 0; vy_e[i] = 0; vz_e[i] = 0;  // initial velocity components of the electron
        x_i[i]  = L * R01(MTgen);               // initial random position of the ion
        vx_i[i] = 0; vy_i[i] = 0; vz_i[i] = 0;  // initial velocity components of the ion
    }
    N_e = nseed;    // initial number of electrons
    N_i = nseed;    // initial number of ions
}
```

Execution of an electron – Ar atom collision. The arguments of the function contain the position and the velocity components of the colliding electron, as well an index related to its energy, which can be used for retrieving the values of the cross sections of the processes in which the given particle can participate. In lines 14-20, the relative velocity (based on the cold gas approximation, i.e., assuming stationary target atoms) and the velocity of the center of mass reference frame (`w`) are computed. This is followed in lines 24-28 by the determination of the Euler angles of the projectile. In the following part of the function the type of the process is chosen in a stochastic manner and depending on the type of the collision to take place (elastic, excitation, or ionization) the scattering and azimuth angles are set. In the case of excitation and ionization, the threshold energy is subtracted from the energy in the center of mass frame (lines 48 / 54). In the case of ionization, the sharing of the remaining kinetic energy between the ejected and scattered electrons is defined in lines 55 and 56. Subsequently, the scattering and azimuth angles for the ejected electron are generated and this electron is scattered. Finally, the new (ejected) electron and the new ion are added to the list of electrons and ions, respectively (lines 70-79). In the final, common branch of the function (lines 82-99) the incoming electron is scattered (in the case of all processes).

```
//------------------------------------------------------------------------//
// e / Ar collision  (cold gas approximation)                             //
//------------------------------------------------------------------------//

void collision_electron (double xe, double *vxe, double *vye, double *vze, int eindex){
    const double F1 = E_MASS  / (E_MASS + AR_MASS);
    const double F2 = AR_MASS / (E_MASS + AR_MASS);
    double t0,t1,t2,rnd;
    double g,g2,gx,gy,gz,wx,wy,wz,theta,phi;
    double chi,eta,chi2,eta2,sc,cc,se,ce,st,ct,sp,cp,energy,e_sc,e_ej;

    // calculate relative velocity before collision & velocity of the centre of mass

    gx = (*vxe);
    gy = (*vye);
    gz = (*vze);
    g  = sqrt(gx * gx + gy * gy + gz * gz);
    wx = F1 * (*vxe);
    wy = F1 * (*vye);
    wz = F1 * (*vze);

    // find Euler angles

    if (gx == 0) {theta = 0.5 * PI;}
    else {theta = atan2(sqrt(gy * gy + gz * gz),gx);}
    if (gy == 0) {
        if (gz > 0){phi = 0.5 * PI;} else {phi = - 0.5 * PI;}
    } else {phi = atan2(gz, gy);}
    st  = sin(theta);
    ct  = cos(theta);
    sp  = sin(phi);
    cp  = cos(phi);

    // choose the type of collision based on the cross sections
    // take into account energy loss in inelastic collisions
    // generate scattering and azimuth angles
    // in case of ionization handle the 'new' electron

    t0   =     sigma[E_ELA][eindex];
    t1   = t0 +sigma[E_EXC][eindex];
    t2   = t1 +sigma[E_ION][eindex];
    rnd  = R01(MTgen);
    if (rnd < (t0/t2)){                             // elastic scattering
        chi = acos(1.0 - 2.0 * R01(MTgen));         // isotropic scattering
        eta = TWO_PI * R01(MTgen);                  // azimuthal angle
    } else if (rnd < (t1/t2)){                      // excitation
```

```
47          energy = 0.5 * E_MASS * g * g;                // electron energy
48          energy = fabs(energy - E_EXC_TH * EV_TO_J);   // subtract energy loss for excitation
49          g   = sqrt(2.0 * energy / E_MASS);            // relative velocity after energy loss
50          chi = acos(1.0 - 2.0 * R01(MTgen));           // isotropic scattering
51          eta = TWO_PI * R01(MTgen);                    // azimuthal angle
52      } else {                                          // ionization
53          energy = 0.5 * E_MASS * g * g;                // electron energy
54          energy = fabs(energy - E_ION_TH * EV_TO_J);   // subtract energy loss of ionization
55          e_ej  = 10.0 * tan(R01(MTgen) * atan(energy/EV_TO_J / 20.0)) * EV_TO_J; // energy of the ejected
     electron
56          e_sc = fabs(energy - e_ej);                   // energy of scattered electron after the collision
57          g    = sqrt(2.0 * e_sc / E_MASS);             // relative velocity of scattered electron
58          g2   = sqrt(2.0 * e_ej / E_MASS);             // relative velocity of ejected electron
59          chi  = acos(sqrt(e_sc / energy));             // scattering angle for scattered electron
60          chi2 = acos(sqrt(e_ej / energy));             // scattering angle for ejected electrons
61          eta  = TWO_PI * R01(MTgen);                   // azimuthal angle for scattered electron
62          eta2 = eta + PI;                              // azimuthal angle for ejected electron
63          sc  = sin(chi2);
64          cc  = cos(chi2);
65          se  = sin(eta2);
66          ce  = cos(eta2);
67          gx  = g2 * (ct * cc - st * sc * ce);
68          gy  = g2 * (st * cp * cc + ct * cp * sc * ce - sp * sc * se);
69          gz  = g2 * (st * sp * cc + ct * sp * sc * ce + cp * sc * se);
70          x_e[N_e]  = xe;                               // add new electron
71          vx_e[N_e] = wx + F2 * gx;
72          vy_e[N_e] = wy + F2 * gy;
73          vz_e[N_e] = wz + F2 * gz;
74          N_e++;
75          x_i[N_i]  = xe;                               // add new ion
76          vx_i[N_i] = RMB(MTgen);                       // velocity is sampled from background thermal
     distribution
77          vy_i[N_i] = RMB(MTgen);
78          vz_i[N_i] = RMB(MTgen);
79          N_i++;
80      }
81
82      // scatter the primary electron
83
84      sc = sin(chi);
85      cc = cos(chi);
86      se = sin(eta);
87      ce = cos(eta);
88
89      // compute new relative velocity:
90
91      gx = g * (ct * cc - st * sc * ce);
92      gy = g * (st * cp * cc + ct * cp * sc * ce - sp * sc * se);
93      gz = g * (st * sp * cc + ct * sp * sc * ce + cp * sc * se);
94
95      // post-collision velocity of the colliding electron
96
97      (*vxe) = wx + F2 * gx;
98      (*vye) = wy + F2 * gy;
99      (*vze) = wz + F2 * gz;
100 }
```

Execution of an Ar^+ – argon collision. The arguments of the function contain the velocity components of both the projectile ion and the target atom. In lines 13-19, the relative velocity between these two particles and the velocity of the center of mass frame are calculated. In lines 21-26, the Euler angles of the incoming particle are determined, and subsequently, in lines 30-37 the type of scattering (isotropic / backscattering) is determined based on the cross sections and a random number. The scattering angle is chosen accordingly in lines 34/36, while the azimuth angle is set in line 38. Finally, the ion is scattered and the components of its new velocity vector are calculated and are returned by the function.

```
1  //-------------------------------------------------------------------------//
2  // Ar+ / Ar collision                                                      //
3  //-------------------------------------------------------------------------//
4
5  void collision_ion (double *vx_1, double *vy_1, double *vz_1,
6                      double *vx_2, double *vy_2, double *vz_2, int e_index){
7      double   g,gx,gy,gz,wx,wy,wz,rnd;
8      double   theta,phi,chi,eta,st,ct,sp,cp,sc,cc,se,ce,t1,t2;
9
10     // calculate relative velocity before collision
11     // random Maxwellian target atom already selected (vx_2,vy_2,vz_2 velocity components of target atom
     come with the call)
12
13     gx = (*vx_1)-(*vx_2);
14     gy = (*vy_1)-(*vy_2);
15     gz = (*vz_1)-(*vz_2);
16     g  = sqrt(gx * gx + gy * gy + gz * gz);
17     wx = 0.5 * ((*vx_1) + (*vx_2));
18     wy = 0.5 * ((*vy_1) + (*vy_2));
19     wz = 0.5 * ((*vz_1) + (*vz_2));
20
```

```
21      // find Euler angles
22
23      if (gx == 0) {theta = 0.5 * PI;} else {theta = atan2(sqrt(gy * gy + gz * gz),gx);}
24      if (gy == 0) {
25          if (gz > 0){phi = 0.5 * PI;} else {phi = - 0.5 * PI;}
26      } else {phi = atan2(gz, gy);}
27
28      // determine the type of collision based on cross sections and generate scattering angle
29
30      t1  =      sigma[I_ISO][e_index];
31      t2  = t1 + sigma[I_BACK][e_index];
32      rnd = R01(MTgen);
33      if  (rnd < (t1 /t2)){                       // isotropic scattering
34          chi = acos(1.0 - 2.0 * R01(MTgen));     // scattering angle
35      } else {                                    // backward scattering
36          chi = PI;                               // scattering angle
37      }
38      eta = TWO_PI * R01(MTgen);                  // azimuthal angle
39      sc  = sin(chi);
40      cc  = cos(chi);
41      se  = sin(eta);
42      ce  = cos(eta);
43      st  = sin(theta);
44      ct  = cos(theta);
45      sp  = sin(phi);
46      cp  = cos(phi);
47
48      // compute new relative velocity
49
50      gx = g * (ct * cc - st * sc * ce);
51      gy = g * (st * cp * cc + ct * cp * sc * ce - sp * sc * se);
52      gz = g * (st * sp * cc + ct * sp * sc * ce + cp * sc * se);
53
54      // post-collision velocity of the ion
55
56      (*vx_1) = wx + 0.5 * gx;
57      (*vy_1) = wy + 0.5 * gy;
58      (*vz_1) = wz + 0.5 * gz;
59  }
```

Solving the Poisson equation. The spatial distribution of the potential and the electric field are computed based on the spatial distribution of the space charge density (`rho1`) and the potential applied to the (powered) electrode situated at the grid point 0. By default, a simple single-frequency (cosine) excitation waveform is used (line 16). The potential of the other (grounded) electrode, situated at the grid point `N_G-1` is set to zero (line 17). Lines 21-31 implement the calculation of the potential based on the Thomas-algorithm. Subsequently, the electric field is derived in lines 35-37.

```
1   //-----------------------------------------------------------------//
2   // solve Poisson equation (Thomas algorithm)                       //
3   //-----------------------------------------------------------------//
4
5   void solve_Poisson (xvector rho1, double tt){
6       const double A =  1.0;
7       const double B = -2.0;
8       const double C =  1.0;
9       const double S = 1.0 / (2.0 * DX);
10      const double ALPHA = -DX * DX / EPSILON0;
11      xvector      g, w, f;
12      int          i;
13
14      // apply potential to the electrodes - boundary conditions
15
16      pot[0]     = VOLTAGE * cos(OMEGA * tt);         // potential at the powered electrode
17      pot[N_G-1] = 0.0;                               // potential at the grounded electrode
18
19      // solve Poisson equation
20
21      for(i=1; i<=N_G-2; i++) f[i] = ALPHA * rho1[i];
22      f[1] -= pot[0];
23      f[N_G-2] -= pot[N_G-1];
24      w[1] = C/B;
25      g[1] = f[1]/B;
26      for(i=2; i<=N_G-2; i++){
27          w[i] = C / (B - A * w[i-1]);
28          g[i] = (f[i] - A * g[i-1]) / (B - A * w[i-1]);
29      }
30      pot[N_G-2] = g[N_G-2];
31      for (i=N_G-3; i>0; i--) pot[i] = g[i] - w[i] * pot[i+1];            // potential at the grid points
        between the electrodes
32
33      // compute electric field
34
35      for(i=1; i<=N_G-2; i++) efield[i] = (pot[i-1] - pot[i+1]) * S;      // electric field at the grid points
          between the electrodes
36      efield[0]     = (pot[0]     - pot[1])     * INV_DX - rho1[0]     * DX / (2.0 * EPSILON0);   // powered
        electrode
37      efield[N_G-1] = (pot[N_G-2] - pot[N_G-1]) * INV_DX + rho1[N_G-1] * DX / (2.0 * EPSILON0);   // grounded
```

```
    electrode
}
```

Main simulation cycle that accomplishes the steps that are executed in every time step, according to figure 2. The main loop (`for` cycle in line 19) covers the time steps in the [0,... N_T-1] range.

Step 1 is the computation of the charged particle densities at the grid points. This is accomplished in lines 25-33 for the electrons. The densities at the outermost grid points are multiplied by a factor of two, since data for these points are collected from spatial domains that are of half size as compared to the rest of the grid points (lines 32-33). The computed density distribution is used for updating the cumulative distribution in line 34. The same calculations are carried out for the positive ions in lines 36-46, but only in every `N_SUB`-th time step (ion subcycling.)

Step 2 (lines 51-52) computes the charge density distribution at the grid points and calls the function that solves the Poisson equation.

Steps 3 and 4 of the PIC-MCC cycle are merged in the forthcoming part of the function. The calculation of the electric field present at the location of the electrons (lines 57-61) the consequent acceleration and displacement of the electrons is computed here (lines 94-95). The latter corresponds to the leapfrog integration scheme. The measurements of various discharge characteristics (that depend directly on the characteristics of the particles, e.g. their velocity and energy) are also integrated into this part of the code. For the electrons, the mean velocity (`ue_xt`), the mean energy (`meane_xt`), and the ionization rate (`ioniz_rate_xt`) are measured with spatial and temporal resolution (line 67-80). Data for each electron contribute to two entries of the specific arrays (that correspond to the grid points that surround the given electron, similarly to the calculation of the density). Data for the EEPF (Electron Energy Probability Function) in the central 10% domain of the discharge are also collected here (lines 84-88). The same calculation is conducted for the positive ions in lines 98-126, with the difference that no ionization rate for the ions and no energy distribution function in the centre of the discharge are computed.

Step 5 is contained in lines 130-172. Here, the particles, which left the computational domain (after being moved in the previous step) are identified. The electrons that arrive to the powered/grounded electrode are counted using the variables `N_e_abs_pow` and `N_e_abs_gnd`, and are subsequently removed from the arrays of coordinates (lines 136-139). In the second part of this code section, the same procedure is executed for the positive ions. For the ions, additionally, the flux-energy distribution function is accumulated here (lines 151-154 / 159-162): the energy of the arriving ions is calculated and an element of the array `ifed_pow` or `ifed_gnd` is incremented accordingly.

Step 6 (check and execution of collisions) is contained in lines 176-187 for electrons and in lines 189-208 for ions. The calculation of collision probability requires the determination (i) of the energy and the velocity of the particles in the case of the electrons and (ii) of the energy in the center of mass reference frame and the relative velocity in the case of ions. In the latter case, this requires sampling a random background gas atom, with Maxwell-Boltzmann velocity distribution; the velocity components are generated in lines 191-193. Whenever the comparison between a random number with uniform distribution (`RO1(MTgen)`) and the collision probability (`p_coll`) indicates the occurrence of a collision, the corresponding function is called in line 184 (for electrons) and in line 204 (for ions).

At the end of the function (lines 210-220), data for the spatio-temporal distributions of some discharge characteristics are collected. In line 222, runtime information is printed to the terminal screen about the actual RF cycle number and the superparticle numbers, in line 224 the same data are saved to the `datafile = conv.dat`.

```
//-------------------------------------------------------------------------//
// simulation of one radiofrequency cycle                                  //
//-------------------------------------------------------------------------//

void do_one_cycle (void){
    const double DV       = ELECTRODE_AREA * DX;
    const double FACTOR_W = WEIGHT / DV;
    const double FACTOR_E = DT_E / E_MASS * E_CHARGE;
    const double FACTOR_I = DT_I / AR_MASS * E_CHARGE;
    const double MIN_X    = 0.45 * L;                    // min. position for EEPF collection
    const double MAX_X    = 0.55 * L;                    // max. position for EEPF collection
    int      k, t, p, energy_index;
    double   g, g_sqr, gx, gy, gz, vx_a, vy_a, vz_a, e_x, energy, nu, p_coll, v_sqr, velocity;
    double   mean_v, c0, c1, c2, rate;
    bool     out;
    xvector  rho;
    int      t_index;

```

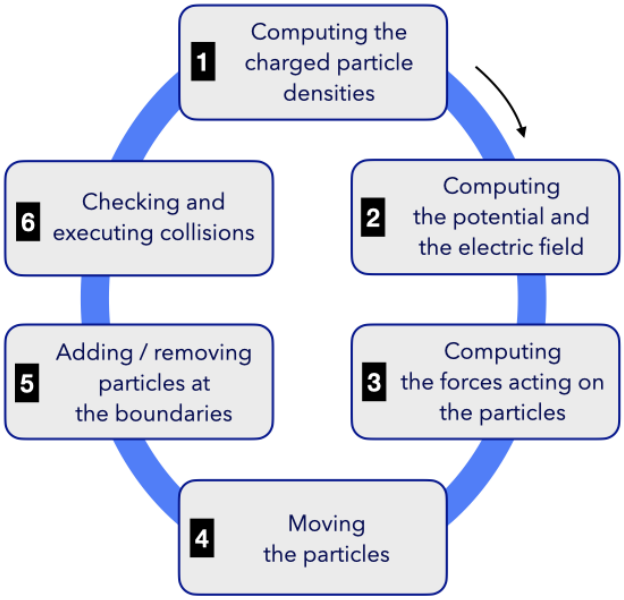

Figure 2: Steps of the basic PIC/MCC cycle [1]

.

```
    for (t=0; t<N_T; t++){          // the RF period is divided into N_T equal time intervals (time step
    DT_E)
        Time += DT_E;               // update of the total simulated time
        t_index = t / N_BIN;        // index for XT distributions

        // step 1: compute densities at grid points

        for(p=0; p<N_G; p++) e_density[p] = 0;                              // electron density - computed in
     every time step
        for(k=0; k<N_e; k++){
            c0 = x_e[k] * INV_DX;
            p  = int(c0);
            e_density[p]   += (p + 1 - c0) * FACTOR_W;
            e_density[p+1] += (c0 - p) * FACTOR_W;
        }
        e_density[0]     *= 2.0;
        e_density[N_G-1] *= 2.0;
        for(p=0; p<N_G; p++) cumul_e_density[p] += e_density[p];

        if ((t % N_SUB) == 0) {                                            // ion density - computed in
    every N_SUB-th time steps (subcycling)
            for(p=0; p<N_G; p++) i_density[p] = 0;
            for(k=0; k<N_i; k++){
                c0 = x_i[k] * INV_DX;
                p  = int(c0);
                i_density[p]   += (p + 1 - c0) * FACTOR_W;
                i_density[p+1] += (c0 - p) * FACTOR_W;
            }
            i_density[0]     *= 2.0;
            i_density[N_G-1] *= 2.0;
        }
        for(p=0; p<N_G; p++) cumul_i_density[p] += i_density[p];

        // step 2: solve Poisson equation

        for(p=0; p<N_G; p++) rho[p] = E_CHARGE * (i_density[p] - e_density[p]);  // get charge density
        solve_Poisson(rho,Time);                                                 // compute potential and
    electric field

        // steps 3 & 4: move particles according to electric field interpolated to particle positions

        for(k=0; k<N_e; k++){                       // move all electrons in every time step
            c0  = x_e[k] * INV_DX;
            p   = int(c0);
            c1  = p + 1.0 - c0;
            c2  = c0 - p;
            e_x = c1 * efield[p] + c2 * efield[p+1];

            if (measurement_mode) {

                // measurements: 'x' and 'v' are needed at the same time, i.e. old 'x' and mean 'v'

                mean_v = vx_e[k] - 0.5 * e_x * FACTOR_E;
```

```
                counter_e_xt[p][t_index]   += c1;
                counter_e_xt[p+1][t_index] += c2;
                ue_xt[p][t_index]   += c1 * mean_v;
                ue_xt[p+1][t_index] += c2 * mean_v;
                v_sqr  = mean_v * mean_v + vy_e[k] * vy_e[k] + vz_e[k] * vz_e[k];
                energy = 0.5 * E_MASS * v_sqr / EV_TO_J;
                meanee_xt[p][t_index]   += c1 * energy;
                meanee_xt[p+1][t_index] += c2 * energy;
                energy_index = min( int(energy / DE_CS + 0.5), CS_RANGES-1);
                velocity = sqrt(v_sqr);
                rate = sigma[E_ION][energy_index] * velocity * DT_E * GAS_DENSITY;
                ioniz_rate_xt[p][t_index]   += c1 * rate;
                ioniz_rate_xt[p+1][t_index] += c2 * rate;

                // measure EEPF in the center

                if ((MIN_X < x_e[k]) && (x_e[k] < MAX_X)){
                    energy_index = (int)(energy / DE_EEPF);
                    if (energy_index < N_EEPF) {eepf[energy_index] += 1.0;}
                    mean_energy_accu_center += energy;
                    mean_energy_counter_center++;
                }
            }

            // update velocity and position

            vx_e[k] -= e_x * FACTOR_E;
            x_e[k]  += vx_e[k] * DT_E;
        }

        if ((t % N_SUB) == 0) {                       // move all ions in every N_SUB-th time steps (subcycling)
            for(k=0; k<N_i; k++){
                c0  = x_i[k] * INV_DX;
                p   = int(c0);
                c1  = p + 1 - c0;
                c2  = c0 - p;
                e_x = c1 * efield[p] + c2 * efield[p+1];

                if (measurement_mode) {

                    // measurements: 'x' and 'v' are needed at the same time, i.e. old 'x' and mean 'v'

                    mean_v = vx_i[k] + 0.5 * e_x * FACTOR_I;
                    counter_i_xt[p][t_index]   += c1;
                    counter_i_xt[p+1][t_index] += c2;
                    ui_xt[p][t_index]   += c1 * mean_v;
                    ui_xt[p+1][t_index] += c2 * mean_v;
                    v_sqr  = mean_v * mean_v + vy_i[k] * vy_i[k] + vz_i[k] * vz_i[k];
                    energy = 0.5 * AR_MASS * v_sqr / EV_TO_J;
                    meanei_xt[p][t_index]   += c1 * energy;
                    meanei_xt[p+1][t_index] += c2 * energy;
                }

                // update velocity and position and accumulate absorbed energy

                vx_i[k] += e_x * FACTOR_I;
                x_i[k]  += vx_i[k] * DT_I;
            }
        }

        // step 5: check boundaries

        k = 0;
        while(k < N_e) {    // check boundaries for all electrons in every time step
            out = false;
            if (x_e[k] < 0) {N_e_abs_pow++; out = true;}    // the electron is out at the powered electrode
            if (x_e[k] > L) {N_e_abs_gnd++; out = true;}    // the electron is out at the grounded electrode
            if (out) {                                      // remove the electron, if out
                x_e [k] = x_e [N_e-1];
                vx_e[k] = vx_e[N_e-1];
                vy_e[k] = vy_e[N_e-1];
                vz_e[k] = vz_e[N_e-1];
                N_e--;
            } else k++;
        }

        if ((t % N_SUB) == 0) {   // check boundaries for all ions in every N_SUB-th time steps (subcycling)
            k = 0;
            while(k < N_i) {
                out = false;
                if (x_i[k] < 0) {       // the ion is out at the powered electrode
                    N_i_abs_pow++;
                    out    = true;
                    v_sqr  = vx_i[k] * vx_i[k] + vy_i[k] * vy_i[k] + vz_i[k] * vz_i[k];
                    energy = 0.5 * AR_MASS *  v_sqr/ EV_TO_J;
                    energy_index = (int)(energy / DE_IFED);
                    if (energy_index < N_IFED) {ifed_pow[energy_index]++;}       // save IFED at the powered electrode
                }
                if (x_i[k] > L) {       // the ion is out at the grounded electrode
```

```
157                     N_i_abs_gnd++;
158                     out     = true;
159                     v_sqr   = vx_i[k] * vx_i[k] + vy_i[k] * vy_i[k] + vz_i[k] * vz_i[k];
160                     energy = 0.5 * AR_MASS * v_sqr / EV_TO_J;
161                     energy_index = (int)(energy / DE_IFED);
162                     if (energy_index < N_IFED) {ifed_gnd[energy_index]++;}        // save IFED at the
    grounded electrode
163                 }
164                 if (out) {  // delete the ion, if out
165                     x_i [k] = x_i [N_i-1];
166                     vx_i[k] = vx_i[N_i-1];
167                     vy_i[k] = vy_i[N_i-1];
168                     vz_i[k] = vz_i[N_i-1];
169                     N_i--;
170                 } else k++;
171             }
172         }
173 
174         // step 6: collisions
175 
176         for (k=0; k<N_e; k++){                              // checking for occurrence of a collision for
    all electrons in every time step
177             v_sqr = vx_e[k] * vx_e[k] + vy_e[k] * vy_e[k] + vz_e[k] * vz_e[k];
178             velocity = sqrt(v_sqr);
179             energy   = 0.5 * E_MASS * v_sqr / EV_TO_J;
180             energy_index = min( int(energy / DE_CS + 0.5), CS_RANGES-1);
181             nu = sigma_tot_e[energy_index] * velocity;
182             p_coll = 1 - exp(- nu * DT_E);                  // collision probability for electrons
183             if (R01(MTgen) < p_coll) {                      // electron collision takes place
184                 collision_electron(x_e[k], &vx_e[k], &vy_e[k], &vz_e[k], energy_index);
185                 N_e_coll++;
186             }
187         }
188 
189         if ((t % N_SUB) == 0) {                             // checking for occurrence of a collision for
    all ions in every N_SUB-th time steps (subcycling)
190             for (k=0; k<N_i; k++){
191                 vx_a = RMB(MTgen);                          // pick velocity components of a random target
    gas atom
192                 vy_a = RMB(MTgen);
193                 vz_a = RMB(MTgen);
194                 gx   = vx_i[k] - vx_a;                       // compute the relative velocity of the
    collision partners
195                 gy   = vy_i[k] - vy_a;
196                 gz   = vz_i[k] - vz_a;
197                 g_sqr = gx * gx + gy * gy + gz * gz;
198                 g = sqrt(g_sqr);
199                 energy = 0.5 * MU_ARAR * g_sqr / EV_TO_J;
200                 energy_index = min( int(energy / DE_CS + 0.5), CS_RANGES-1);
201                 nu = sigma_tot_i[energy_index] * g;
202                 p_coll = 1 - exp(- nu * DT_I);              // collision probability for ions
203                 if (R01(MTgen)< p_coll) {                   // ion collision takes place
204                     collision_ion (&vx_i[k], &vy_i[k], &vz_i[k], &vx_a, &vy_a, &vz_a, energy_index);
205                     N_i_coll++;
206                 }
207             }
208         }
209 
210         if (measurement_mode) {
211 
212             // collect 'xt' data from the grid
213 
214             for (p=0; p<N_G; p++) {
215                 pot_xt   [p][t_index] += pot[p];
216                 efield_xt[p][t_index] += efield[p];
217                 ne_xt    [p][t_index] += e_density[p];
218                 ni_xt    [p][t_index] += i_density[p];
219             }
220         }
221 
222         if ((t % 1000) == 0) printf(" c = %8d  t = %8d  #e = %8d  #i = %8d\n", cycle,t,N_e,N_i);
223     }
224     fprintf(datafile,"%8d  %8d  %8d\n",cycle,N_e,N_i);
225 }
```

Saving (lines 5-29) and loading (lines 35-60) of the "state of the system". At the end of the simulation of a specified number of RF cycles, all the data that will allow continuing the simulation of the same setting is saved to the binary file `picdata.bin`. These include the time, the number of the RF cycles completed, the number of electron and ion superparticles, as well as all the coordinates of all the particles. Loading these data allows a smooth continuation of the simulation for the next cycles.

```
1 //-----------------------------------------------------------------------//
2 // save particle coordinates                                            //
3 //-----------------------------------------------------------------------//
4 
5 void save_particle_data(){
6     double   d;
```

```
7       FILE   * f;
8       char fname[80];
9
10      strcpy(fname,"picdata.bin");
11      f = fopen(fname,"wb");
12      fwrite(&Time,sizeof(double),1,f);
13      d = (double)(cycles_done);
14      fwrite(&d,sizeof(double),1,f);
15      d = (double)(N_e);
16      fwrite(&d,sizeof(double),1,f);
17      fwrite(x_e, sizeof(double),N_e,f);
18      fwrite(vx_e,sizeof(double),N_e,f);
19      fwrite(vy_e,sizeof(double),N_e,f);
20      fwrite(vz_e,sizeof(double),N_e,f);
21      d = (double)(N_i);
22      fwrite(&d,sizeof(double),1,f);
23      fwrite(x_i, sizeof(double),N_i,f);
24      fwrite(vx_i,sizeof(double),N_i,f);
25      fwrite(vy_i,sizeof(double),N_i,f);
26      fwrite(vz_i,sizeof(double),N_i,f);
27      fclose(f);
28      printf(">> eduPIC: data saved : %d electrons %d ions, %d cycles completed, time is %e [s]\n",N_e,N_i,
         cycles_done,Time);
29  }
30
31  //-------------------------------------------------------------------------//
32  // load particle coordinates                                               //
33  //-------------------------------------------------------------------------//
34
35  void load_particle_data(){
36      double   d;
37      FILE   * f;
38      char fname[80];
39
40      strcpy(fname,"picdata.bin");
41      f = fopen(fname,"rb");
42      if (f==NULL) {printf(">> eduPIC: ERROR: No particle data file found, try running initial cycle using
         argument '0'\n"); exit(0); }
43      fread(&Time,sizeof(double),1,f);
44      fread(&d,sizeof(double),1,f);
45      cycles_done = int(d);
46      fread(&d,sizeof(double),1,f);
47      N_e = int(d);
48      fread(x_e, sizeof(double),N_e,f);
49      fread(vx_e,sizeof(double),N_e,f);
50      fread(vy_e,sizeof(double),N_e,f);
51      fread(vz_e,sizeof(double),N_e,f);
52      fread(&d,sizeof(double),1,f);
53      N_i = int(d);
54      fread(x_i, sizeof(double),N_i,f);
55      fread(vx_i,sizeof(double),N_i,f);
56      fread(vy_i,sizeof(double),N_i,f);
57      fread(vz_i,sizeof(double),N_i,f);
58      fclose(f);
59      printf(">> eduPIC: data loaded : %d electrons %d ions, %d cycles completed before, time is %e [s]\n",N_e
         ,N_i,cycles_done,Time);
60  }
```

Saving of the time-averaged electron and ion density. The function makes use of the cumulative electron density (`cumul_e_density`) and cumulative ion density (`cumul_i_density`) data accumulated in each time step of the simulation. Upon writing to the data file, these data are multiplied by a factor that accounts for the number of accumulation times during the whole simulation (computed in line 11).

```
1   //-------------------------------------------------------------------------//
2   // save density data                                                       //
3   //-------------------------------------------------------------------------//
4
5   void save_density(void){
6       FILE *f;
7       double c;
8       int m;
9
10      f = fopen("density.dat","w");
11      c = 1.0 / (double)(no_of_cycles) / (double)(N_T);
12      for(m=0; m<N_G; m++){
13          fprintf(f,"%8.5f  %12e  %12e\n",m * DX, cumul_e_density[m] * c, cumul_i_density[m] * c);
14      }
15      fclose(f);
16  }
```

Saving of the EEPF (Electron Energy Probability Function) to the data file `eepf.dat`. The data are normalized corresponding to (the discretized form of) $\int f(\varepsilon)\sqrt{\varepsilon}\mathrm{d}\varepsilon = 1$. Recall that these data represent the central 10% spatial region of the discharge.

```
1   //-------------------------------------------------------------------------//
```

```
// save EEPF data                                                          //
//-------------------------------------------------------------------------//

void save_eepf(void) {
    FILE   *f;
    int    i;
    double h,energy;

    h = 0.0;
    for (i=0; i<N_EEPF; i++) {h += eepf[i];}
    h *= DE_EEPF;
    f = fopen("eepf.dat","w");
    for (i=0; i<N_EEPF; i++) {
        energy = (i + 0.5) * DE_EEPF;
        fprintf(f,"%e  %e\n", energy, eepf[i] / h / sqrt(energy));
    }
    fclose(f);
}
```

Saving of the time-averaged IFEDF (Ion Flux Energy Distribution Function) at the powered (second column) and grounded (third column) electrode as a function of the energy (first column) to the data file `ifed.dat`. The functions are normalized as $\int F(\varepsilon)\mathrm{d}\varepsilon = 1$.

```
//-------------------------------------------------------------------------//
// save IFED data                                                          //
//-------------------------------------------------------------------------//

void save_ifed(void) {
    FILE   *f;
    int    i;
    double h_pow,h_gnd,energy;

    h_pow = 0.0;
    h_gnd = 0.0;
    for (i=0; i<N_IFED; i++) {h_pow += ifed_pow[i]; h_gnd += ifed_gnd[i];}
    h_pow *= DE_IFED;
    h_gnd *= DE_IFED;
    mean_i_energy_pow = 0.0;
    mean_i_energy_gnd = 0.0;
    f = fopen("ifed.dat","w");
    for (i=0; i<N_IFED; i++) {
        energy = (i + 0.5) * DE_IFED;
        fprintf(f,"%6.2f %10.6f %10.6f\n", energy, (double)(ifed_pow[i])/h_pow, (double)(ifed_gnd[i])/h_gnd)
    ;
        mean_i_energy_pow += energy * (double)(ifed_pow[i]) / h_pow;
        mean_i_energy_gnd += energy * (double)(ifed_gnd[i]) / h_gnd;
    }
    fclose(f);
}
```

Saving of the spatio-temporal maps of several discharge characteristics. The list of these was provided in table 2. The raw data accumulated for the various quantities needs specific normalization that is accomplished by the function `norm_all_xt()` in lines 19-58. The normalized distributions are saved by the function `save_xt_1()`, which is called for each of the quantities in the function `save_all_xt()`. These distributions have `N_G` points in space and `N_XT` points in time, where `N_XT = N_T / N_BIN`, see earlier.

```
//-------------------------------------------------------------------------//
// save XT data                                                            //
//-------------------------------------------------------------------------//

void save_xt_1(xt_distr distr, char *fname) {
    FILE   *f;
    int    i, j;

    f = fopen(fname,"w");
    for (i=0; i<N_G; i++){
        for (j=0; j<N_XT; j++){
            fprintf(f,"%e  ", distr[i][j]);
        }
        fprintf(f,"\n");
    }
    fclose(f);
}

void norm_all_xt(void){
    double f1, f2;
    int    i, j;

    // normalize all XT data

    f1 = (double)(N_XT) / (double)(no_of_cycles * N_T);
    f2 = WEIGHT / (ELECTRODE_AREA * DX) / (no_of_cycles * (PERIOD / (double)(N_XT)));

    for (i=0; i<N_G; i++){
        for (j=0; j<N_XT; j++){
```

```
30              pot_xt[i][j]    *= f1;
31              efield_xt[i][j] *= f1;
32              ne_xt[i][j]     *= f1;
33              ni_xt[i][j]     *= f1;
34              if (counter_e_xt[i][j] > 0) {
35                  ue_xt[i][j]     =  ue_xt[i][j] / counter_e_xt[i][j];
36                  je_xt[i][j]     = -ue_xt[i][j] * ne_xt[i][j] * E_CHARGE;
37                  meanee_xt[i][j] =  meanee_xt[i][j] / counter_e_xt[i][j];
38                  ioniz_rate_xt[i][j] *= f2;
39               } else {
40                  ue_xt[i][j]         = 0.0;
41                  je_xt[i][j]         = 0.0;
42                  meanee_xt[i][j]     = 0.0;
43                  ioniz_rate_xt[i][j] = 0.0;
44              }
45              if (counter_i_xt[i][j] > 0) {
46                  ui_xt[i][j]     = ui_xt[i][j] / counter_i_xt[i][j];
47                  ji_xt[i][j]     = ui_xt[i][j] * ni_xt[i][j] * E_CHARGE;
48                  meanei_xt[i][j] = meanei_xt[i][j] / counter_i_xt[i][j];
49              } else {
50                  ui_xt[i][j]     = 0.0;
51                  ji_xt[i][j]     = 0.0;
52                  meanei_xt[i][j] = 0.0;
53              }
54              powere_xt[i][j] = je_xt[i][j] * efield_xt[i][j];
55              poweri_xt[i][j] = ji_xt[i][j] * efield_xt[i][j];
56          }
57      }
58  }
59
60  void save_all_xt(void){
61      char fname[80];
62
63      strcpy(fname,"pot_xt.dat");      save_xt_1(pot_xt, fname);
64      strcpy(fname,"efield_xt.dat");   save_xt_1(efield_xt, fname);
65      strcpy(fname,"ne_xt.dat");       save_xt_1(ne_xt, fname);
66      strcpy(fname,"ni_xt.dat");       save_xt_1(ni_xt, fname);
67      strcpy(fname,"je_xt.dat");       save_xt_1(je_xt, fname);
68      strcpy(fname,"ji_xt.dat");       save_xt_1(ji_xt, fname);
69      strcpy(fname,"powere_xt.dat");   save_xt_1(powere_xt, fname);
70      strcpy(fname,"poweri_xt.dat");   save_xt_1(poweri_xt, fname);
71      strcpy(fname,"meanee_xt.dat");   save_xt_1(meanee_xt, fname);
72      strcpy(fname,"meanei_xt.dat");   save_xt_1(meanei_xt, fname);
73      strcpy(fname,"ioniz_xt.dat");    save_xt_1(ioniz_rate_xt, fname);
74  }
```

The following function generates a simulation report, which is saved to the file `info.dat` and controls the saving of all data for the plasma characteristics. This report includes information about the basic simulation parameters specified by the user (lines 22-32) as well as about some of the computed primary plasma parameters (lines 34-39).

Subsequently, some of the stability and accuracy criteria of the simulation are evaluated according to the prescriptions given in section 2.2 of [1]. These are the conditions concerning the relation of the grid spacing to the Debye length, the relation of the time step to the electron plasma frequency, and the collision probabilities during a time step.

In the case when any of these conditions is violated the function writes an error message to the file `info.dat`. When no violation of the conditions is found, the function continues to save further data to files (including the density and energy distributions, and spatio-temporal maps of a number of plasma characteristics (lines 77-81)) and calculates and writes additional diagnostics data to the file `info.dat` (including the fluxes of electrons and ions at both electrodes and the mean ion energy at both electrodes (lines 82-88)), as well as the power density absorbed by both charged species (lines 93-106)).

```
1  //---------------------------------------------------------------------//
2  // simulation report including stability and accuracy conditions       //
3  //---------------------------------------------------------------------//
4
5  void check_and_save_info(void){
6      FILE     *f;
7      double   plas_freq, meane, kT, debye_length, density, ecoll_freq, icoll_freq, sim_time, e_max, v_max,
       power_e, power_i, c;
8      int      i,j;
9      bool     conditions_OK;
10
11     density    = cumul_e_density[N_G / 2] / (double)(no_of_cycles) / (double)(N_T);  // e density @ center
12     plas_freq  = E_CHARGE * sqrt(density / EPSILON0 / E_MASS);                       // e plasma frequency @
         center
13     meane      = mean_energy_accu_center / (double)(mean_energy_counter_center);     // e mean energy @
       center
14     kT         = 2.0 * meane * EV_TO_J / 3.0;                                        // k T_e @ center (
       approximate)
15     sim_time   = (double)(no_of_cycles) / FREQUENCY;                                 // simulated time
```

```
    ecoll_freq = (double)(N_e_coll) / sim_time / (double)(N_e);                    // e collision
     frequency
    icoll_freq = (double)(N_i_coll) / sim_time / (double)(N_i);                    // ion collision
     frequency
    debye_length = sqrt(EPSILON0 * kT / density) / E_CHARGE;                       // e Debye length @
     center

    f = fopen("info.txt","w");
    fprintf(f,"########################## eduPIC simulation report ##########################\n");
    fprintf(f,"Simulation parameters:\n");
    fprintf(f,"Gap distance                          = %12.3e [m]\n",  L);
    fprintf(f,"# of grid divisions                   = %12d\n",      N_G);
    fprintf(f,"Frequency                             = %12.3e [Hz]\n", FREQUENCY);
    fprintf(f,"# of time steps / period              = %12d\n",      N_T);
    fprintf(f,"# of electron / ion time steps        = %12d\n",      N_SUB);
    fprintf(f,"Voltage amplitude                     = %12.3e [V]\n",  VOLTAGE);
    fprintf(f,"Pressure (Ar)                         = %12.3e [Pa]\n", PRESSURE);
    fprintf(f,"Temperature                           = %12.3e [K]\n",  TEMPERATURE);
    fprintf(f,"Superparticle weight                  = %12.3e\n",      WEIGHT);
    fprintf(f,"# of simulation cycles in this run    = %12d\n",      no_of_cycles);
    fprintf(f,"------------------------------------------------------------------------------\n");
    fprintf(f,"Plasma characteristics:\n");
    fprintf(f,"Electron density @ center             = %12.3e [m^{-3}]\n", density);
    fprintf(f,"Plasma frequency @ center             = %12.3e [rad/s]\n",  plas_freq);
    fprintf(f,"Debye length @ center                 = %12.3e [m]\n",      debye_length);
    fprintf(f,"Electron collision frequency          = %12.3e [1/s]\n",    ecoll_freq);
    fprintf(f,"Ion collision frequency               = %12.3e [1/s]\n",    icoll_freq);
    fprintf(f,"------------------------------------------------------------------------------\n");
    fprintf(f,"Stability and accuracy conditions:\n");
    conditions_OK = true;
    c = plas_freq * DT_E;
    fprintf(f,"Plasma frequency @ center * DT_E      = %12.3f (OK if less than 0.20)\n", c);
    if (c > 0.2) {conditions_OK = false;}
    c = DX / debye_length;
    fprintf(f,"DX / Debye length @ center            = %12.3f (OK if less than 1.00)\n", c);
    if (c > 1.0) {conditions_OK = false;}
    c = max_electron_coll_freq() * DT_E;
    fprintf(f,"Max. electron coll. frequency * DT_E  = %12.3f (OK if less than 0.05)\n", c);
    if (c > 0.05) {conditions_OK = false;}
    c = max_ion_coll_freq() * DT_I;
    fprintf(f,"Max. ion coll. frequency * DT_I       = %12.3f (OK if less than 0.05)\n", c);
    if (c > 0.05) {conditions_OK = false;}
    if (conditions_OK == false){
        fprintf(f,"------------------------------------------------------------------------------\n");
        fprintf(f,"** STABILITY AND ACCURACY CONDITION(S) VIOLATED - REFINE SIMULATION SETTINGS! **\n");
        fprintf(f,"------------------------------------------------------------------------------\n");
        fclose(f);
        printf(">> eduPIC: ERROR: STABILITY AND ACCURACY CONDITION(S) VIOLATED!\n");
        printf(">> eduPIC: for details see 'info.txt' and refine simulation settings!\n");
    }
    else
    {
        // calculate maximum energy for which the Courant-Friedrichs-Levy condition holds:

        v_max = DX / DT_E;
        e_max = 0.5 * E_MASS * v_max * v_max / EV_TO_J;
        fprintf(f,"Max e- energy for CFL condition       = %12.3f [eV]\n", e_max);
        fprintf(f,"Check EEPF to ensure that CFL is fulfilled for the majority of the electrons!\n");
        fprintf(f,"------------------------------------------------------------------------------\n");

        // saving of the following data is done here as some of the further lines need data
        // that are computed / normalized in these functions

        printf(">> eduPIC: saving diagnostics data\n");
        save_density();
        save_eepf();
        save_ifed();
        norm_all_xt();
        save_all_xt();
        fprintf(f,"Particle characteristics at the electrodes:\n");
        fprintf(f,"Ion flux at powered electrode         = %12.3e [m^{-2} s^{-1}]\n", N_i_abs_pow * WEIGHT /
     ELECTRODE_AREA / (no_of_cycles * PERIOD));
        fprintf(f,"Ion flux at grounded electrode        = %12.3e [m^{-2} s^{-1}]\n", N_i_abs_gnd * WEIGHT /
     ELECTRODE_AREA / (no_of_cycles * PERIOD));
        fprintf(f,"Mean ion energy at powered electrode  = %12.3e [eV]\n", mean_i_energy_pow);
        fprintf(f,"Mean ion energy at grounded electrode = %12.3e [eV]\n", mean_i_energy_gnd);
        fprintf(f,"Electron flux at powered electrode    = %12.3e [m^{-2} s^{-1}]\n", N_e_abs_pow * WEIGHT /
     ELECTRODE_AREA / (no_of_cycles * PERIOD));
        fprintf(f,"Electron flux at grounded electrode   = %12.3e [m^{-2} s^{-1}]\n", N_e_abs_gnd * WEIGHT /
     ELECTRODE_AREA / (no_of_cycles * PERIOD));
        fprintf(f,"------------------------------------------------------------------------------\n");

        // calculate spatially and temporally averaged power absorption by the electrons and ions

        power_e = 0.0;
        power_i = 0.0;
        for (i=0; i<N_G; i++){
            for (j=0; j<N_XT; j++){
                power_e += powere_xt[i][j];
                power_i += poweri_xt[i][j];
            }
```

```
100         }
101         power_e /= (double)(N_XT * N_G);
102         power_i /= (double)(N_XT * N_G);
103         fprintf(f,"Absorbed power calculated as <j*E>:\n");
104         fprintf(f,"Electron power density (average)      = %12.3e [W m^{-3}]\n", power_e);
105         fprintf(f,"Ion power density (average)          = %12.3e [W m^{-3}]\n", power_i);
106         fprintf(f,"Total power density(average)         = %12.3e [W m^{-3}]\n", power_e + power_i);
107         fprintf(f,"-------------------------------------------------------------------------------\n");
108         fclose(f);
109     }
110 }
```

The `main()` function controls code execution. The command line arguments are evaluated in lines 17-35. The first argument gives the number of RF cycles to simulate. A zero value of this parameter has a special meaning: in this case a new simulation can be initiated. An accidental initialization and overwriting of previous simulation results is prevented in lines 41-47; here, if a `picdata.bin` file originating from a previous simulation is found in the current folder a warning message is issued in the terminal window and the code is stopped. In case no such file is present in the current folder, the code seeds a given number (`N_INIT`) of electron and ion superparticles within the electrode gap, executes the simulation of a single RF cycle, and saves the state of the system in the function `save_particle_data()` to the file `picdata.bin`.

In case the first command line argument is greater than zero, the code will load the state of the system (saved previously) using the call to `load_particle_data()`. Subsequently, it executes the simulation of the specified number of RF cycles, and saves the state of the system as explained above.

```
1  //-------------------------------------------------------------------------------------------//
2  // main                                                                                      //
3  // command line arguments:                                                                   //
4  // [1]: number of cycles (0 for init)                                                        //
5  // [2]: "m" turns on data collection and saving                                              //
6  //-------------------------------------------------------------------------------------------//
7
8  int main (int argc, char *argv[]){
9      printf(">> eduPIC: starting...\n");
10     printf(">> eduPIC: *********************************************************************\n");
11     printf(">> eduPIC: Copyright (C) 2021 Z. Donko et al.\n");
12     printf(">> eduPIC: This program comes with ABSOLUTELY NO WARRANTY\n");
13     printf(">> eduPIC: This is free software, you are welcome to use, modify and redistribute it\n");
14     printf(">> eduPIC: according to the GNU General Public License, https://www.gnu.org/licenses/\n");
15     printf(">> eduPIC: *********************************************************************\n");
16
17     if (argc == 1) {
18         printf(">> eduPIC: error = need starting_cycle argument\n");
19         return 1;
20     } else {
21         strcpy(st0,argv[1]);
22         arg1 = atol(st0);
23         if (argc > 2) {
24             if (strcmp (argv[2],"m") == 0){
25                 measurement_mode = true;                  // measurements will be done
26             } else {
27                 measurement_mode = false;
28             }
29         }
30     }
31     if (measurement_mode) {
32         printf(">> eduPIC: measurement mode: on\n");
33     } else {
34         printf(">> eduPIC: measurement mode: off\n");
35     }
36     set_electron_cross_sections_ar();
37     set_ion_cross_sections_ar();
38     calc_total_cross_sections();
39     //test_cross_sections(); return 1;
40     datafile = fopen("conv.dat","a");
41     if (arg1 == 0) {
42         if (FILE *file = fopen("picdata.bin", "r")) { fclose(file);
43             printf(">> eduPIC: Warning: Data from previous calculation are detected.\n");
44             printf("           To start a new simulation from the beginning, please delete all output files
    before running ./eduPIC 0\n");
45             printf("           To continue the existing calculation, please specify the number of cycles to
    run, e.g. ./eduPIC 100\n");
46             exit(0);
47         }
48         no_of_cycles = 1;
49         cycle = 1;                                        // init cycle
50         init(N_INIT);                                     // seed initial electrons & ions
51         printf(">> eduPIC: running initializing cycle\n");
52         Time = 0;
53         do_one_cycle();
54         cycles_done = 1;
55     } else {
56         no_of_cycles = arg1;                              // run number of cycles specified in command line
57         load_particle_data();                             // read previous configuration from file
```

```
58          printf(">> eduPIC: running %d cycle(s)\n",no_of_cycles);
59          for (cycle=cycles_done+1;cycle<=cycles_done+no_of_cycles;cycle++) {do_one_cycle();}
60          cycles_done += no_of_cycles;
61      }
62      fclose(datafile);
63      save_particle_data();
64      if (measurement_mode) {
65          check_and_save_info();
66      }
67      printf(">> eduPIC: simulation of %d cycle(s) is completed.\n",no_of_cycles);
68  }
```

4 Licence / Disclaimer

References

[1] Donkó Z, Derzsi A, Vass M, Horváth B, Wilczek S, Hartmann B and Hartmann P 2021 *arXiv preprint arXiv:2103.09642*

[2] Phelps A and Petrovic Z L 1999 *Plasma Sources Science and Technology* **8** R21

[3] Phelps A V 1994 *Journal of applied physics* **76** 747–753