FRAUNHOFER IST · SIMULATION DIGITAL SERVICES

PICMC / DSMC

Documentation of the DSMC and PIC-MC software package

Next (release 26)

17 November 2026

Working tree — describes what is coming, not a release.

DOCUMENTATION	E. Volkhardt, P. Schulz, A. Pflug, D. Barton
DEVELOPERS	D. Barton, P. Schulz, E. Volkhardt, A. Poudel, T. Melzig
RECENT DEVELOPERS	A. Pflug, M. Siemers, T. Melzig, M. Melzer, C. Schwanke

Contents

Welcome 14
What you find here 14
Where to start 14
Versions 14
Introduction to the DSMC / PIC-MC software package 15
Where this code sits 17
Setup Guide 18
Installation & Setup 18
Installation of the picmc package 18
Python cases without these environment variables 19
Job submission via the SLURM job scheduler 20
General workflow 22
General workflow 22
DSMC gas flow simulation 22
PIC-MC plasma simulation without magnetic field 23
PIC-MC plasma simulation with magnetic field 23
Submitting a run 25
The parameter-file route: submit 25
The Python route: py/pysubmit.py 25
Chaining runs 26
Checkpoints and continuation 27
What is written 27
How often: NDUMPS 27
Continuing a run 27
Writing a checkpoint by hand 28
Python interface (beta) 30
How the two processes cooperate 30
Building and installing 31
Submitting a run 31
Progress while the run is going 32
A minimal case script 32
Methods of Sim 33
Keyword arguments of the classes 34
Histograms on borders 34
Mapping of parameter file keys 35
What the interface does not cover yet 36

Magnetic field solver (beta) 37
How a run is organised 37
A minimal script 38
Keyword arguments 39
What the entries mean 40
The mesh 40
Output 40
Reusing the field on a different plasma mesh 40
Differences to the parameter file route 41
Simulation pipeline 42
Why a chain instead of one code 42
The chain 42
The chain is meant to be run in a loop 43
Stage by stage 43
What PICMC contributes to the chain 44
Examples 46
In-line hot-wire CVD coater 46
Reactive sputtering of TiO_2 with atomistic growth 47
Bandpass filter on a convex lens 47
Concave substrate for a membrane optics project 48
Macroparticles: where does the contamination come from 49
Coupling to CFD 49
What the chain does not do 49
What comes next 49
Where to start with PICMC 50
Modeling 51
Units and conversion factors 51
SCCM - Gas flow rate 51
Conversion of ampere to sccm 51
Pumping Speed 51
Definition 51
Select transmission probability of a pumping surface 52
Example 52
Relation between gas flow, pumping speed and pressure 52
Mean free path 53
Temperature vs. kinetic energy 54
Statistical constraints 55
DSMC parameters with respect to total pressure 55
Scaling behaviour of cell dimension, particle number and time step 55
Numerical example for Argon 56
Footnotes 56

Numerical constraints 57
Limitations to the time step width 57
Cell spacing 57
Collision frequency 57
Electron collision frequency 58
Plasma frequency 58
Debye length 58
Gyrokinetics 59
Limitations to the cell spacing 60
Isotropic vs. anisotropic cell spacing 60
Footnotes 60
Collision routine 62
NTC algorithm 62
How a collision is applied to the partners 63
General NTC implementation with reaction products 63
Handling of frozen species 64
Where this is implemented 64
Footnotes 65
Statistical collision sampling 66
Collisions between species with same weight factor 66
Collisions between species with different weight factors 67
Collisions, where one of the collision partners is frozen 67
Cross sections 69
Important notes on cross sections 69
Energy sampling in cross section array 70
Cross section database 70
Database of the PIC-MC framework 70
Footnotes 71
Species list 72
Electrons 72
Period 1 72
Period 2 72
Period 3 72
Period 4 72
Period 5 73
Period 6 73
Organic compounds 73
Anorganic compounds 73
Including other species 73
Energy partitioning 75
Comparision of different partitioning schemes 75
One takes it all vs. equal share 75
Footnotes 76
Averaged quantities 77

Resampling (beta) 78
What resampling does 78
Switching it on 78
Profiles 78
Tuning 79
What to check afterwards 79
Limitations 80

Parameter file 81

Parameter file review 81
General parameter 81
Species 82
Frozen species 82
Plots 83
Histograms 83
Simulation volume 84
Optimized domain decomposition for DSMC 85
Definition of input files 86
Border definition 86
Additional switches 86
Special model parameters 87
Magnetic and electric field 87
Particle initialization 88
Moving meshes 88
Scaling factor adjustment 89
Resampling (beta) 89
User actions and functions 90
Species and cross sections 90
Debug 91

Parameter file tuning 92
User_Functions 92
Profiling data and load balancing 92
Particles per cell 93
Tuning resampling 93

CAD import (.stp/.igs/.stl) 94
Footnotes 94

External magnetic field 95
Magnetic field support with unstructured data points 95
Magnetic field support with structured data points arranged in a box 96

Wall interaction model 97

Wall border type 97
Footnotes 98

Plain reactions 99
Simple absorption with sticking coefficient 99
Specular reflection 99
Plain reaction with sticking coefficient and reaction products 99

Particle sources 101
Constant sources 101
Sources with inscribed profile 101
Vapor pressure source depending on surface temperature 102
Particle emission 103
Maxwellian distribution with alternative temperature 103
Source of sputtered particles 103
Uniform energy distribution within a given interval 105
Particle emission distribution by user defined histogram 106
Particle emission distribution by user defined functions 106
Specifying energy loss functions 107
Footnotes 108
Wall temperature 109
Secondary electron emission 110
Setting the yield in a case 110
Footnotes 111
Energy/angle dependency 112
Energy dependent reaction probability 112
Energy dependent yield 113
Energy and angular dependent yield 114
Footnotes 115
Surface chemistry 116
Specification of wall materials and initial coverages 118
Surface reaction involving no deposition 119
Surface reaction involving deposition 119
Surface cover deposition 120
Implantation 120
Example for a surface chemistry model for reactive ZnO:Al deposition 121
Footnotes 123
Surface state sources 124
Sputtering sources with given ion flux 125
Thermal desorption of coated material 126
Footnotes 127
Membrane border type 128
Energy/angle sampling 130
Energy histogram 130
Angular histogram 130
Combined histogram 130
Histogram array 131
Footnotes 132
Electric boundary conditions 133
Power regulation 135

DSMC examples 137

Overview 137

3D gas flow simulation 138
Geometric model 138
General considerations about geometries for gas flow simulation 138
Definition of geometric points 139
Definition of straight lines and circle arcs 141
Definition of ruled and plain surfaces 144
Physical surface groups 147
Final step: Saving the mesh file 149
Setting up the parameter file 149
Global control variables 150
Gas species and cell resolution 150
Simulation domain - bounding box 152
Surface functions 153
Footnotes 154
Optimized domain decomposition 155
Tube example with four bounding boxes 156
Definition of multiple bounding boxes in block MQUAD 157
Joining multiple bounding boxes together in block MJOIN 158
Former join_rect function 159
Examples of the join_rect function 160
Sputter transport 162
The three ingredients 162
Order of the reactions on a border 163
Boundary conditions of the bounding box 163
A worked case 163
What to look at afterwards 164
PIC-MC examples 165
Overview 165
1D HF parallel plate 166
Starting the simulation 166
Results 166
Footnotes 166
1D parallel plate results 167
Parallel plate discharge in argon using dual frequency excitation 167
Footnotes 168
2D magnetron discharge 169
Geometric model 169
How To 170
Step 1: Preparation 170
Step 2: Create and edit the parameter file 170
Step 3: Create and edit the magnetic template 172
Step 4: Start magnetic field computation 172
Step 5: Start simulation 173
Results 173
Substrate on ground potential 173
Substrate on floating potential 173
Discharge anomaly(?) with strong magnetron type 174

2D magnetron results (A700V) 176
Magnetron discharge simulation with A700V chamber layout 176
Unbalanced magnetron: Anode shielding with substrate on floating potential 177
Balanced magnetron: Short-circuit with dark space shielding 178
2D dielectric barrier discharge 180
Starting the simulation 180
Results 180
3D linear magnetron 181
Starting the simulation 181
Postprocessing 181
Results 181
3D RF ICP discharge 182
Coil current computation with BEM 182
Setup in parameter file 183
Results and comparison 183
BEM magnetic field 185
Usage 186
Geometry modelling 187
Parameter file settings 188
Maximum model size 191
Validation 191
Validation against measurements 191
Accuracy issues 193
Using invalid (constant) material parameters 193
Using low mesh resolution 194
DSMC/PIC-MC iteration 196
Problem 196
Solution 196
Footnotes 197
Electron movement 198
Problem 198
Solution 198
Use of adapted species and cross section file 198
Modified parameter file including creation of POSITION plots and representative secondary electrons 200
Example 202
Footnotes 203
Postprocessing 204
Available output data 204
POS files 205
Cell structure 207
Node structure 208
Surface logging 209
Particle balance at surfaces 209
Potentials of conductive electrodes 210
Checkpoints 211

Checkpoint file format 212
Common header 212
Particles: particles_<species>.bin 212
Averages: averages_<species>.bin 213
Charges: charges.bin 213
Materials: material_<material>.bin 213
Checking a file before reading it 214
Checkpoint tools 215
Translation 216
Combination 216
Used cross sections 217
Dynamic script files 218
Postprocessing with RVM (ppp) 219
Prerequisites / installation 219
General usage 219
Summary of commands 220
Function reference 220
Functions to import data 221
Simple cut planes 221
General oblique planes and lineprofiles 221
Cylindrical cutplanes 221
Output functions 222
Usage of the module by examples 222
Simple scalar cut-plane 223
Scalar cut-plane with arithmetics 224
Scalar cut-plane with triangulation 224
Scalar cut-plane from multiple input files 224
Time averaging 226
Development over time 226
Generating vector plots as cut plane 227
Revert POS file data to original state 227
General (oblique) interpolation views 228
Additional output formats 229
ASCII export 229
Numerical integration 230
Merging of (position) plots 232
Manipulation of data 232
General concept 232
Creation of new input data 233
Reduction of input data 234
Control of written data 235
Convolution function 236
Footnotes 242
Postprocessing examples 243
Projection of quantities onto cylindrical surfaces 245
Example 1: Magnetic field 245
Example 2: Ion absorption profile 246

Matrix deposition 248
Trajectory object 249
Script syntax 249
Relevant C++ routines 249
Histogram object 250
How to provide matrix histogram files 250
Internal histogram representation 251
Footnotes 252
Postprocessing with GMSH 253
Extracting ion flux onto rotatable targets 253
First step: Create ASCII files of the ion flux in the relevant area 253
Second step: Make a projection of the flux onto the curved target surface 255
Creating animations 259
Footnotes 260
Animations with Blender 261
Step-by-step tutorial 261
Step 0 261
Step 1 261
Step 2 262
Step 3 262
Step 4 262
Step 5 264
Step 6 264
Substrate raytracer 265
Getting started 265
Getting the tool 265
Requirements 265
Building 265
Making the module importable 266
Checking an installation you did not build 266
Running on a cluster 266
Usage 267
Coordinate system 267
The four objects of a scenario 267
Substrate 267
Trajectory 268
Histogram 268
Masks 269
Running the raytracer 270
Reading the result 270
A complete script 271

Fitting with SciPy 272
Basic idea and workflow 272
A simple example 272
The goal function 273
Reusing the raytracer 273
Optimising the fitting process 274
Fitting mask geometries 275
General idea 275
Python script 276
Questions 278
Substrates 279
Explanation 279
Overview 279
Convex substrate types 280
Slanted rectangle 280
Spherical lens 281
Toric lens 282
Disk 283
Aspherical lens 284
Potentially concave substrate types 285
Inverted spherical lens 285
Bent rectangle 286
Parabolic lens 287
Custom substrates 287
Helper module 289
Main usage 289
Functions 289
Loading custom geometries 289
Trajectories from machine parameters 289
Circular trajectory 290
Linear trajectory 290
Algorithm 292
The raytracing algorithm 292
Substrate generation 292
Mask intersections 292
Self-intersections in concave substrates 293
Parallelisation 293
PALADIN 294
Installation 294
Requirements 294
Build 294
Environment 295
Verify the installation 295
Updates 296

Building a scenario 297
Geometry and mesh 298
Simulation volume 298
Walls 298
Fields 299
Species 299
Starting particles 300
Running a case 301
Callbacks 301
Launching 301
Output 302
Controlling a running job 303
Shipped example cases 303
Before a long run 303
Particle model 304
Time integration 304
Fields the particle sees 304
Forces 304
Charging 305
Drag and thermophoresis in two regimes 306
Surfaces: reflection, adhesion, rolling 306
Cohesion and agglomerates 307
Droplets: evaporation and condensation 307
Where the equations are 307
Parameters 308
Run identity and output 308
Domain 308
Physics switches 309
Numerical and termination control 309
Content 310
Species keys 310
Constants and units 312
Units 312
Physical constants 312
Wall types 313
Size distributions 313
Bounding box options 313
Dump formats and modes 313
Quantities a dumper can write 314
Scalars 314
Detail scalars 314
Vectors 315
Detail vectors, one per force term 315
Full state 315

Troubleshooting **316**

Troubleshooting & warnings 316

- Parameter file issue 316
- Geometry issue 316
- Memory issue 316
- Warning messages 316

References 318

1 Welcome

PICMC and DSMC simulate low-pressure gas and plasma transport with particles instead of a continuum: DSMC for the neutral gas flow, PIC-MC for the charged species in a plasma. Both deliver what arrives at a substrate — flux, energy and angle distribution — and from that the deposition or etch result.

This site documents the whole way from an empty directory to a coated part.

What you find here

Section	Content
Setup	Requirements, installation, first run
Workflow	The parts of a simulation case, the Python interface, the simulation pipeline
Modeling	Units, numerical and statistical limits, collision routines, species, cross sections
Parameter file	Every parameter key, tuning, CAD import, external magnetic fields
Wall interaction model	Border types, surface reactions, particle sources, secondary electrons, membranes
Examples	Worked DSMC and PIC-MC cases together with their input files
Postprocessing	Output files, logging, checkpoints, RVM and GMSH, animations, deposition on real parts
Substrate raytracer	Deposition on a real part: shapes, trajectories, masks, and fitting mask geometries with SciPy
PALADIN	Microparticles in plasmas and gas flows: charging, transport, adhesion, and where the contamination comes from
Troubleshooting	Typical warnings and what they mean

Where to start

- **First contact with the code:** Setup, then the 3D gas flow example, which walks through a complete case.
- **An existing case has to be ported or tuned:** Parameter file and the wall interaction model — the two places where most of the behaviour is set.
- **From the simulation to the coated part:** the simulation pipeline shows which stage hands what to the next one, and where matrix deposition or the substrate raytracer take over.

Versions

The version switcher in the navbar selects which release you are reading. **release 25** is the released state; **Next (release 26)** is the working tree and already describes what the next release will bring. What changed between two releases is in the release notes.

2 Introduction to the DSMC / PIC-MC software package

The simulation of transport phenomena is an important method to understand the dynamics in low-pressure deposition reactors. This includes the flow of the process gas as well as the transport of reactive gases and precursors. The most general transport equation in classical mechanics is the Boltzmann transport equation

$$\frac{\partial f}{\partial t} + \vec{v}\,\nabla_x f + \frac{\vec{F}}{m}\,\nabla_v f(\vec{x},\vec{v},t) = \left.\frac{\partial f}{\partial t}\right|_{\text{collision}}$$

It describes the time evolution of a particle distribution function $f(\vec{x},\vec{v},t)$ in a 6-dimensional phase space (space and velocity). Such high-dimensional spaces are usually difficult to handle by numerical simulation, e.g. by the finite element method, because the number of mesh elements would increase with the 6th power of the geometric scale. A perturbation analysis (see e.g. Chapman and Cowling (1970)) shows that for a low value of the mean free path (see section Numerical constraints) in comparison to characteristic geometric dimensions of the flow, the Boltzmann equation can be approximated by the Navier Stokes equations, which merely treat the density, velocity and pressure as flow variables in the three-dimensional space. For the Navier Stokes equations, a large variety of solvers based on Finite Element or Finite Volume Method (FEM, FVM) are available as commercial or open-source codes. A prominent FVM based solver for continuum mechanics is OpenFOAM, whereof different versions are maintained either by the OpenFOAM Foundation or by the ESI Group. Both versions differ slightly with respect to the included solvers and auxiliary libraries.

The validity of the Navier Stokes equations depends on the mean free path λ with respect to characteristic geometric dimensions d, which can be expressed as the so-called Knudsen number

$$\mathrm{Kn} = \frac{\lambda}{d}$$

As shown in the following figure, continuum approaches are only valid for Knudsen numbers well below 0.1. For higher Knudsen values, kinetic simulation codes must be used instead. In case of very large Knudsen numbers $\mathrm{Kn} \gg 1$, the flow can be considered as collisionless; in this case it is possible to use a ray-tracing approach for solving the Boltzmann equation via a particle based approach. An example of a software which solves particle transport under molecular flow conditions is MolFlow developed at CERN. A more detailed discussion on thin film deposition methods and simulation codes is given in Pflug et al. (2016).

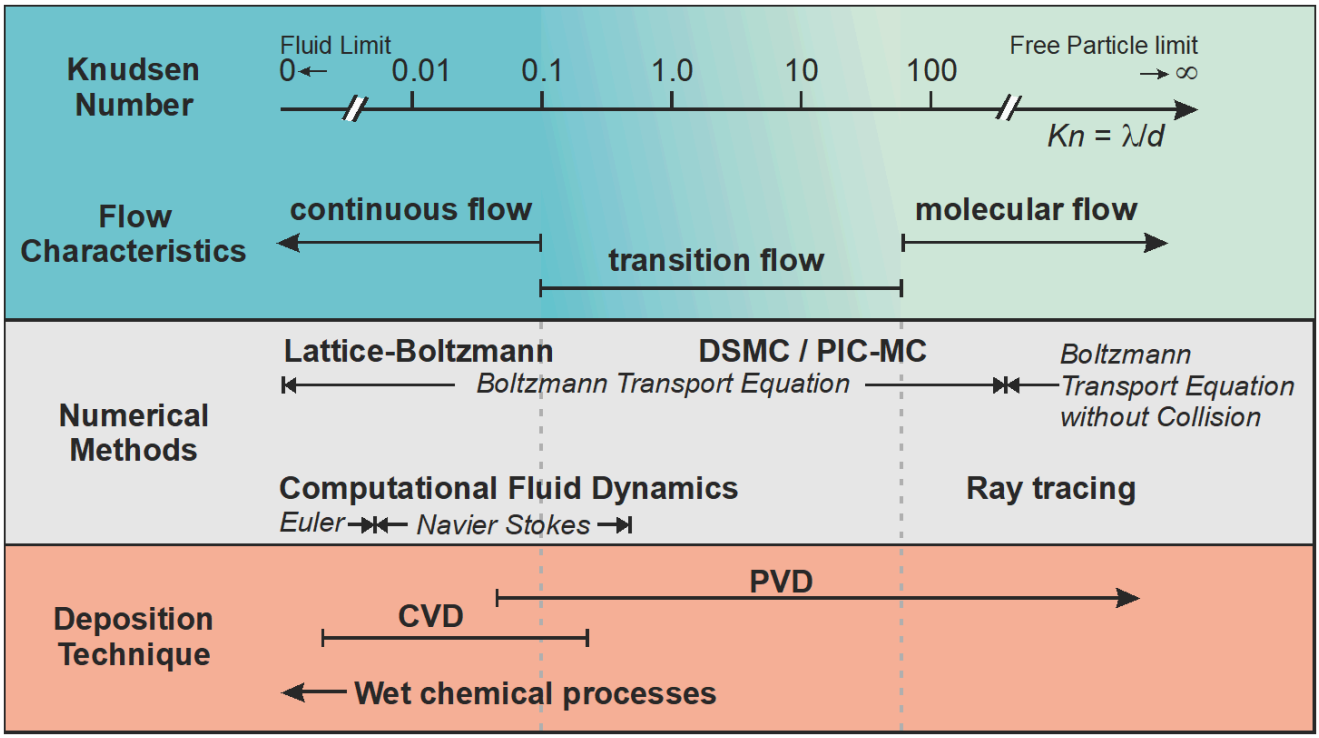


Figure 1.1: Kinetic and continuum solvers for transport phenomena and their validity with respect to the flow characteristic (adapted from Bird (1994))

For an intermediate range $0.1 < \mathrm{Kn} < 100$ of the Knudsen number, neither a continuum approach nor a solver based on collisionless flow is viable. With respect to thin film deposition technology, most Physical Vapor Deposition (PVD) and some of the Chemical Vapor Deposition (CVD) processes fall into this intermediate range with respect to their process conditions. Thus, their proper modeling requires to use a kinetic solver of the Boltzmann equation including collision term.

Other than a fully-fledged FEM or FVM based solution of the Boltzmann equation, a statistical approach called Direct Simulation Monte Carlo (DSMC) - method (see Bird (1994)) is a solution method with feasible computational effort. It is based on following simplifications:

- All flow species are represented by super particles. A super particle has the same mass and properties as real gas molecules of the respective species, but it stands for a number of real particles - its statistical weight. The weight belongs to the individual particle: initially all particles of a species share the value of `NREAL_<species>`, which is optional and defaults to 10. The scaling factor is necessary as otherwise the number of particles to be simulated would be too high to fit into usual high performance computing hardware.
- For collisions between particles, only binary collisions are considered.
- For performance reasons, the particle movement and collision are treated not simultaneously but in two separate routines, which are invoked subsequently for each time cycle.

The 3rd simplification requires that the product $\delta t \times v$ of time step and mean thermal velocity v of all treated species is below the cell spacing of the simulation grid. A typical schedule of the time step in a DSMC solver can be seen in the following figure by following the blue boxes.

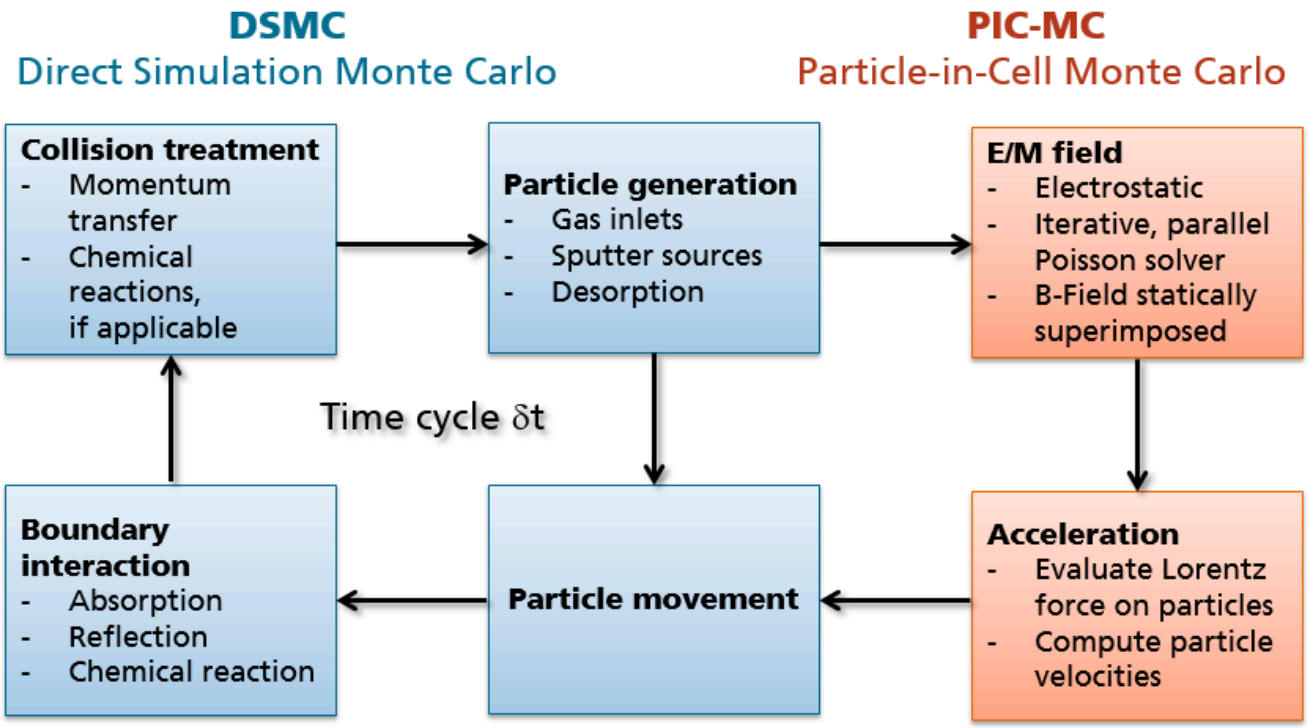


Figure 1.2: Scheme of a time cycle in the Direct Simulation Monte Carlo (DSMC) and Particle-in-Cell Monte Carlo (PIC-MC) methods

The DSMC method only treats the transport of neutral particles and hence is suitable to model rarefied gas flows, evaporation or thermal CVD processes. If the transport of charged particles becomes relevant as in plasma processes, the electric and magnetic fields have to be additionally considered. The approximation used in this software is referred to as Vlasov Poisson equation, which means that the force term in the equation above is replaced by the Lorentz force,

$$\frac{\partial f}{\partial t} + \vec{v}\,\nabla_x f + \frac{q}{m}\left(\vec{E} + \vec{v}\times\vec{B}\right)\nabla_v f(\vec{x},\vec{v},t) = \left.\frac{\partial f}{\partial t}\right|_{\text{collision}}$$

and the electric potential φ is derived from the Poisson equation according to the actual charge density distribution ρ and the relative dielectric permeability ε_r of the background.

$$\Delta\varphi + \frac{\nabla\varepsilon_r\,\nabla\varphi}{\varepsilon_r} = -\frac{\rho}{\varepsilon_0\,\varepsilon_r}$$

In a self-consistent Particle-in-Cell Monte Carlo (PIC-MC) - simulation, the last two equations have to be fulfilled simultaneously, while in the actual iterative implementation they are solved sequentially

- as shown by the large cycle in Figure 1.2. This imposes additional numerical constraints on the simulation parameters such as cell spacing, time step etc. as elaborated further in Numerical constraints. A detailed discussion of the PIC-MC method is given in the monography of Birdsall and Langdon (2005).

Where this code sits

PICMC covers the plasma and gas transport stage of a longer chain that ends at a coated part. The simulation pipeline shows the whole chain, what each stage hands to the next one, and which part of it this documentation describes.

Setup Guide

3 Installation & Setup

Installation of the picmc package

The DSMC / PICMC package is usually being shipped as compressed tar archive, such as `picmc_versionNumber.tar.bz2`. In order to uncompress it, the following command can be used:

```
tar xjvf picmc_versionNumber.tar.bz2
```

This will create an installation folder `picmc_versionNumber/*` where the required binaries, libraries and script files are included. The meaning of the sub folders are as follows:

- `picmc_versionNumber/bin` = the C++ compiled binaries including **picmc** worker task, magnetic field computation module **bem** and the script interpreter **rvm**.
- `picmc_versionNumber/lib` = compiled run time libraries used by the script interpreter **rvm**.
- `picmc_versionNumber/scr` = scripts used by **rvm** for scheduling the simulation run and managing the input and output data as well as internal model parameters (including species and cross sections)
- `picmc_versionNumber/sh` = Linux shell scripts (bash) for submitting simulation runs
- `picmc_versionNumber/data` = Tabulated cross section data for some species

  Python interface. The build produces `lib/PICMC.so` as well, so the Python development files (`python3.X-config`) have to be installed; the interpreter is selected by `PYCONFIGPATH_<goal>` in `makefile.inc`, and the interpreter used at run time should be the same version. Besides the module, the installation receives `lib/pydatabase.py`, `py/pysubmit.py`, its cluster configurations `py/*.json` and the shell script `sh/pysubmit`. See Driving a case from Python.

After creation of the installation folder `picmc_versionNumber/` it shall be moved into an appropriate installation directory, e.g.

```
mv picmc_versionNumber /opt/picmc
```

In the home directories of the users, appropriate environment variable settings need to be included into their `/home/user/.bashrc` files. A possible way is to include the following lines:

```
export PICMCPATH=/opt/picmc/picmc_versionNumber
export LD_LIBRARY_PATH=$PICMCPATH/lib:$LD_LIBRARY_PATH
export RVMPATH=$PICMCPATH/rvmlib
export PATH=$PICMCPATH/bin:$PICMCPATH/sh:$PATH
```

The part after `PICMCPATH=` has to be replaced by the path of your actual installation directory. After changing the .bashrc the users are required to close their command prompt and open a new one in order for the changes to take effect.

Python cases without these environment variables

The exports above are what `rvm`, `rvmmpi` and an interactive `python3` session need - they read `PICMCPATH`, `LD_LIBRARY_PATH` and `PYTHONPATH` from the calling shell. A submitted Python case does not need them, because the submission helper writes them into the batch script:

```
python3 /opt/picmc/picmc_versionNumber/py/pysubmit.py case.py -n 8 -c IST_EPYC
```

`py/pysubmit.py` takes the installation path, the MPI path and the queue defaults from `py/<cluster>.json`, generates `case.slurm` and submits it with `sbatch`. The generated file exports `PICMCPATH`, `LD_LIBRARY_PATH`, `PATH` and `PYTHONPATH` itself, writes `case-out.txt` and `case-err.txt` and appends the error file to the output at the end. One absolute path to the helper is therefore enough - no shell setup, and the case may live in any directory.

Always name the configuration with `-c` (`--config`) - without it the helper falls back to its built-in default `IST_INTEL`, which on every other cluster carries the wrong partition, memory size and MPI path, and the job either queues forever or dies with an MPI error. The installation ships configurations for the known clusters (`IST_EPYC`, `IST_EPYC_TEST`, `IST_INTEL`, `CE_XEON`, `CE_ZEN1`, `CE_ZEN5`). Only `picmc_path` and `mpi_path` normally have to be adapted:

```
{
  "description": "Standard configuration for epyc nodes at IST",
  "default_partition": "epyc",
  "allowed_partitions": ["epyc"],
  "picmc_path": "~/picmc-26-EPYC",
  "memory": 16000,
  "mpi_path": "/opt/openmpi-5",
  "mpi_exec": "mpiexec"
}
```

A copy of the JSON in the working directory takes precedence over the one next to the helper, so a case-specific variant does not have to replace the installed file.

The options of the helper: `-n` number of workers (the Python process itself runs as one additional task), `-b` runs the `bem` binary instead of `picmc`, `-p` partition (it has to be listed in `allowed_partitions`), `-N` number of nodes, `-l` nodelist, `-m` memory per core in MB, `-t` wall time as `hh:mm:ss`, `-s` random seed and `-e` for exclusive node access. What is not given is taken from the configuration file.

The older shell script `sh/pysubmit` is still shipped and does the same for one cluster: it builds its own batch file and takes the flags of the parameter-file route (`-picmc`, `-bem`, `-time`, `-mem`, ...). It needs the exports above, its cluster defaults are fixed in the script, and it cannot read a configuration file - for new cases use `py/pysubmit.py`.

For an interactive session - a case that is to be tried step by step - the exports are still needed, because `import PICMC` finds the module through `PYTHONPATH`. How the two processes cooperate (one `python3 case.py` rank plus the `picmc` workers) is described in Driving a case from Python.

The direct invocation of simulation runs is done via the script `picmc_versionNumber/sh/rvmmpi`. In order for this to work correctly, the correct path of your OpenMPI installation directory has to be specified in this file. The respective line can be found in the upper part of the file and shall look like follows:

```
MPI_PATH=/opt/openmpi-2.1.0
```

Please replace this path with the correct installation directory of OpenMPI for your Linux machine.

Job submission via the SLURM job scheduler

In case of the SLURM job scheduler, we provide a sample submission script which is given here in submit.zip. Unpack this ZIP archive and place the script `submit` into an appropriate folder for executable scripts such as `/home/user/bin` or `/usr/local/bin`. Don't forget to make this script executable via `chmod +x submit`.

In the upper part of the script, there is an editable section which has to be adjusted to your actual SLURM installation:

```
# ======================
# Editable value section
# ======================

picmcpath=$PICMCPATH                          # PICMC installation path
partition="v3"                                # Default partition
memory=1300                                   # Default memory in MB per CPU
walltime="336:00:00"                          # Default job duration in hhh:mm:ss (336 hours -> 14
days),
                                              # specify -time <nn> of command line for different
job duration
MPIEXEC=mpiexec                               # Name of mpiexec executable
MPIPATH=/opt/openmpi-2.1.0                    # Full path to OpenMPI installation
#MAILRECIPIENT=user@institution.com           # Uncomment this to get status emails from SLURM
                                              # (Needs a running mail server installation such as
postfix)

QSUB=sbatch

# ============================
# End of editable value section
# ============================
```

It is important to specify the correct OpenMPI installation path in `MPIPATH` as well as the default SLURM partition under `partition`. Additionally, the approximate maximal memory usage per task and maximal run time of the job are to be specified in `memory` and `walltime`. In the settings shown above, there is a maximal memory allocation of 1.3 GB per task and a maximal run time of 14 days. These settings can be overwritten by command line switches of the submit script. The submit script can be invoked in the following ways:

Command line	Comment
`submit -bem 40 simcase`	Perform magnetic field simulation with 40 cores in the default partition
`submit -picmc 12 simcase`	Perform DSMC / PICMC simulation run with 12 cores in the default partition
`submit -partition special -picmc 60 simcase`	Perform DSMC / PICMC simulation run with 60 cores in a SLURM partition named `special`
`submit -partition special -N 3 -picmc 60 simcase`	Perform DSMC / PICMC simuation run with 60 cores and split the number of cores evenly over 3 compute nodes

Command line	Comment
`submit -mem 4096 -picmc 8 simcase`	Perform DSMC / PICMC simulation run on 8 cores and foresee memory allocation of up to 4 GB per core
`submit -after NNNN -picmc 24 simcase`	Perform DSMC / PICMC simulation run on 24 cores **but do not start until the SLURM job NNNN is completed**. The previous SLURM job may be a magnetic field computation. This way, a picmc simulation can be automatically started when a formerly required magnetic field computation is finished.

All above examples assume that your simulation case is given in the file `simcase.par` in the actual directory. By submitting a SLURM job via `submit` the following additional files will be created:

- **simcase-out.txt**: The screen output in stdout; this file is being refreshed typically every 15-30 seconds.
- **simcase-err.txt**: Error messages (if any) can be found here. If the simulation crashes or behaves in an unexpected way, it is important to look into this file.
- **simcase.slurm**: This is a native SLURM submission script generated by `submit` according to the actual settings as specified in the command line. If the job needs being re-submitted with the same settings, it can be easily done via `sbatch simcase.slurm`

General workflow

4 General workflow

For DSMC-Gasflow simulation, PIC-MC plasma simulation or BEM magnetic field computation, different work flows are required. Depending on the usage of a Grid Scheduler or direct call of mpirun, the invocation commands may differ. In the following, the work flows and invocation commands for the different simulation cases are summarized.

> **Python interface.** The tables below are the RVM and shell invocation. A case can also be started from a Python script: the submit helper is then `py/pysubmit.py`, and the run itself is one MPI job with a single Python rank and the worker program behind it (for a BEM field computation `bem`, selected with `-b`). See Driving a case from Python.

DSMC gas flow simulation

The work flow in a DSMC gas flow simulation can be summarized as follows:

Step	Description	Command via mpirun	Command via LSF batch system
1	Create geometric mesh file via GMSH (see example) and save mesh file as **filename.msh**		
2	Create parameter template **filename.par** out of mesh file **filename.msh**	`initpicmc filename`	
3	Edit parameter template (see example)		
4	Start simulation	`rvmmpi -hostfile <hostfile> -picmc <n> filename`	`rvmlsf -picmc <n> -i filename`
5	Create intermediate results during simulation	`touch filename/plot`	
6	Shutdown simulation and store internal state for later continuation	`touch filename/shutdown`	
7	Restart a simulation after shutdown (equal to step 4)	`rvmmpi -hostfile <hostfile> -picmc <n> filename`	`rvmlsf -picmc <n> -i filename`

The number of parallel **picmc** tasks, which should be used during computation are given in the `<n>` parameter after the `-picmc` switch. Steps 5 to 7 work through files in the case folder: `plot`, `shutdown` and `kill` are read between two time cycles, and a run whose case folder holds `last_shutdown.def` continues from its checkpoint instead of starting over. Both routes are described in Checkpoints and continuation.

Remember that the number of parallel **picmc** tasks may not exceed (but may be less than) the number of segments of the simulation domain. The actual number of allocated CPU cores will be one more than the number of **picmc** tasks, since one more task is required for the master process.

If the simulation is invoked directly via mpirun using the `rvmmpi` script, a *hostfile* must be declared. This hostfile contains the names of the computing nodes together with their available number of CPU slots. An example is given in the following:

```
node1 slots=1
node3 slots=48
node4 slots=48
```

If a batch scheduler such as **LSF** or **Sun Grid Engine (SGE)** is used, no hostfile must be specified since the hostfile will be created dynamically by the scheduler. A graphical visualization of the work flow is given in the following picture.

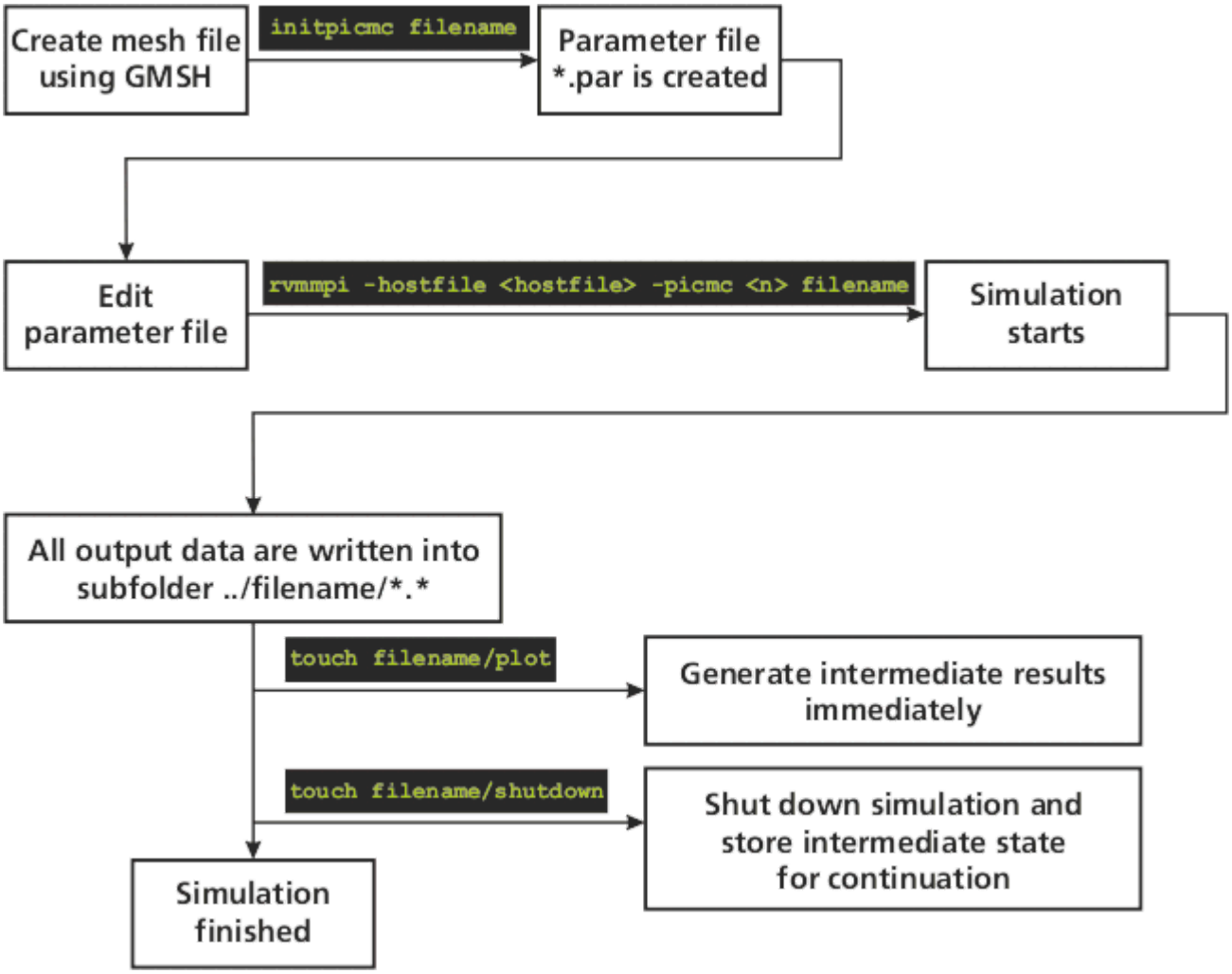


Figure 1: Sketch of the work flow of a DSMC gas flow simulation

PIC-MC plasma simulation without magnetic field

For a PIC-MC plasma simulation without magnetic field (e. g. in a PECVD parallel plate reactor or a glow discharge without magnetic field), the workflow is the same as in the DSMC case.

PIC-MC plasma simulation with magnetic field

For a plasma simulation with an overlying magnetic field - e. g. a discharge of a magnetron sputtering target - additional steps for producing the magnetic field are required prior to starting the picmc simulation. The complete workflow is summarized in the following:

Step	Description	Command via mpirun	Command via LSF batch system
1	Create geometric mesh file of the simulation domain (e. g. **frame.msh**) via GMSH and save mesh file e. g. as **frame.msh**		
2	Create mesh file of the magnetron arrangement (e. g. **magnet.msh**) via GMSH and save mesh file e. g. as **magnet.msh**		
3	Create parameter template **frame.par** out of mesh file **frame.msh**	`initpicmc frame`	
4	Edit parameter template. Fill in the name of the magnetic mesh file in the field `BEMMESH`		
5	Create a template file **frame.bem**, where the relation between surfaces and magnetic domains will be declared	`initbfield frame`	
6	Edit the magnetic template file **frame.bem**		
8	Perform magnetic field computation	`rvmmpi -hostfile <hostfile> -bem <n> frame`	`rvmlsf -bem -i frame`
9	Start PIC-MC simulation	`rvmmpi -hostfile <hostfile> -picmc <n> filename`	`rvmlsf -picmc <n> -i filename`
10	Create intermediate results during simulation	`touch filename/plot`	
11	Shutdown simulation and store internal state for later continuation	`touch filename/shutdown`	
12	Restart a simulation after shutdown	`rvmmpi -hostfile <hostfile> -picmc <n> filename`	`rvmlsf -picmc <n> -i filename`

5 Submitting a run

A production run goes through the queue, and two scripts write the batch file for it: `submit`, the shell script of the parameter-file route, and `py/pysubmit.py`, the Python script introduced with release 26. Both create `<case>.slurm` next to the case, hand it to `sbatch`, write stdout to `<case>-out.txt` and stderr to `<case>-err.txt`, and append the error file to the output file when the job ends. For an interactive run without the queue see Driving a case from Python and the invocation tables of the manual.

The parameter-file route: submit

`submit` sits in `$PICMCPATH/sh` and takes the parameter file; the `.par` suffix may be left off.

```
submit -picmc 8 -time 48:00:00 -seed 4711 case.par
```

option	meaning
`-picmc <n>`	number of `picmc` workers
`-bem <n>`	number of `bem` workers, for a magnetic field computation
`-time <hh:mm:ss>`	wall time, an integer is read as hours
`-mem <MB>`	memory per core
`-partition <name>`	partition, default `epyc`
`-N <n>`	number of nodes
`-exclusive`	exclusive node access
`-seed <n>`	fixed random seed, exported as `PICMC_SEED`
`-collisionModel legacy|SK07`	collision model
`-after <jobid>`	start after that job finished (`--dependency=afterok`)
`-valgrind`	run under valgrind
`-help`	print the usage

The queue defaults - partition, memory, wall time, MPI path - are constants at the top of the script and have to be edited there for a new cluster. `submit` reads `PICMCPATH` from the calling shell, so the exports of the installation have to be in place.

The Python route: py/pysubmit.py

`py/pysubmit.py` submits a Python case and takes the queue defaults from `py/<cluster>.json` instead of from the script:

```
python3 $PICMCPATH/py/pysubmit.py case.py -n 8 -c IST_EPYC -t 48:00:00 -s 4711
```

option	meaning
`-n <n>`	number of workers; the job gets one task more for the Python rank
`-c <cluster>`	configuration file `py/<cluster>.json`, a copy in the working directory wins

option	meaning
-b	run the `bem` binary instead of `picmc`, mandatory for a BEM script
`-t <hh:mm:ss>`	wall time, default `336:00:00`
`-s <n>`	random seed
`-m <MB>`	memory per core
`-p <name>`	partition, has to be listed in `allowed_partitions`
`-N <n>`	number of nodes
`-l <list>`	nodelist
`-e`	exclusive node access
`-a <jobid>`	start after that job finished (`--dependency=afterok`)

Name the configuration with `-c`: without it the helper falls back to its built-in default `IST_INTEL`, which on another cluster carries the wrong partition, memory size and MPI path. What the file holds and how the two processes of a case cooperate is described in Driving a case from Python.

Chaining runs

Both scripts pass a dependency to SLURM, so a chain of runs is queued in one go - the second job waits until the first one has finished successfully:

```
submit -picmc 8 plasma.par
python3 $PICMCPATH/py/pysubmit.py transport.py -n 8 -c IST_EPYC -a <job id printed by the first
submit>
```

A chain that has to survive a wall time limit is cut at a checkpoint instead: the second job continues the first one from its state, described in Checkpoints and continuation.

6 Checkpoints and continuation

A plasma stage of a real coater runs for hours to days. A checkpoint is the state of the run at one time step - particles, averages, charges, surface coverages and the voltage control - so the run can be stopped and continued later, or continued after a change downstream. Both routes write the same kind of files; they differ in who triggers the write and who reads it back.

What is written

file	content	notes
`checkpoint/particles_<species>.bin`	coordinates, velocities and weights of the particles of that species	always
`checkpoint/averages_<species>.bin`	time averaged density, velocity and squared velocity per cell	only with `TAVG`
`checkpoint/charges.bin`	charge per grid node	only for `WORKMODE = 1`
`checkpoint/material_<material>.bin`	coverages and deposition rates per surface cell	parameter file route only
`last_shutdown.def`	iteration step, time step, plot interval and electrode voltage	always

The binary layout of the files is described under checkpoints in postprocessing and checkpoint file format. The parameter file route writes `last_shutdown.def` into the case folder, the Python route into `<case>/checkpoint/` - see the note at the end of this page.

How often: NDUMPS

`NDUMPS` is the number of checkpoints distributed evenly over the run, so one is written every `TSIM/DT/NDUMPS` steps. The default `1` means the checkpoint is written when the run reaches its end. A new checkpoint overwrites the previous one, there is no rotation - raise the value if an unexpected crash would cost days.

```
NDUMPS = 4;   # four checkpoints over the run
```

In Python `NDUMPS` has to be set before `sim.init()`. A value of zero or below turns the dumps off, and `sim.exit()` does not write one either: such a run has no checkpoint at all.

Continuing a run

The parameter file route continues on its own: if `<case>/last_shutdown.def` exists, the master reads the checkpoints and starts at the stored step instead of generating a new particle distribution. To stop a run cleanly, drop a file named `shutdown` into the case folder; the master finishes the remaining `PAR.NTESTCYCLES` (default 200) profiling cycles, writes the checkpoint and ends. A file named `kill` ends the run without a checkpoint.

```
touch case/shutdown         # stop after the profiling cycles, write the checkpoint
submit -picmc 8 case.par    # continue - the same case, nothing to change in the parameter file
```

The Python route decides this in the script:

```
sim.init(reload_if_possible = True)   # continue at the stored step if a state is present
```

With `reload_if_possible = True` and no checkpoint the init prints a note and starts a fresh distribution; with `False` an existing checkpoint is ignored. Python has no `shutdown` file - the run ends at the last step or when the job is stopped, so stopping between two checkpoints loses everything since the last one.

	parameter file	**Python**
stop into a checkpoint	`touch <case>/shutdown`, the run ends by itself	stop at a checkpoint step, nothing else
continue	submit the same case again	`sim.init(reload_if_possible = True)`
write a checkpoint by hand	`S.write_checkpoint(...)`	not offered
coverages and deposition state	written and read back	not written

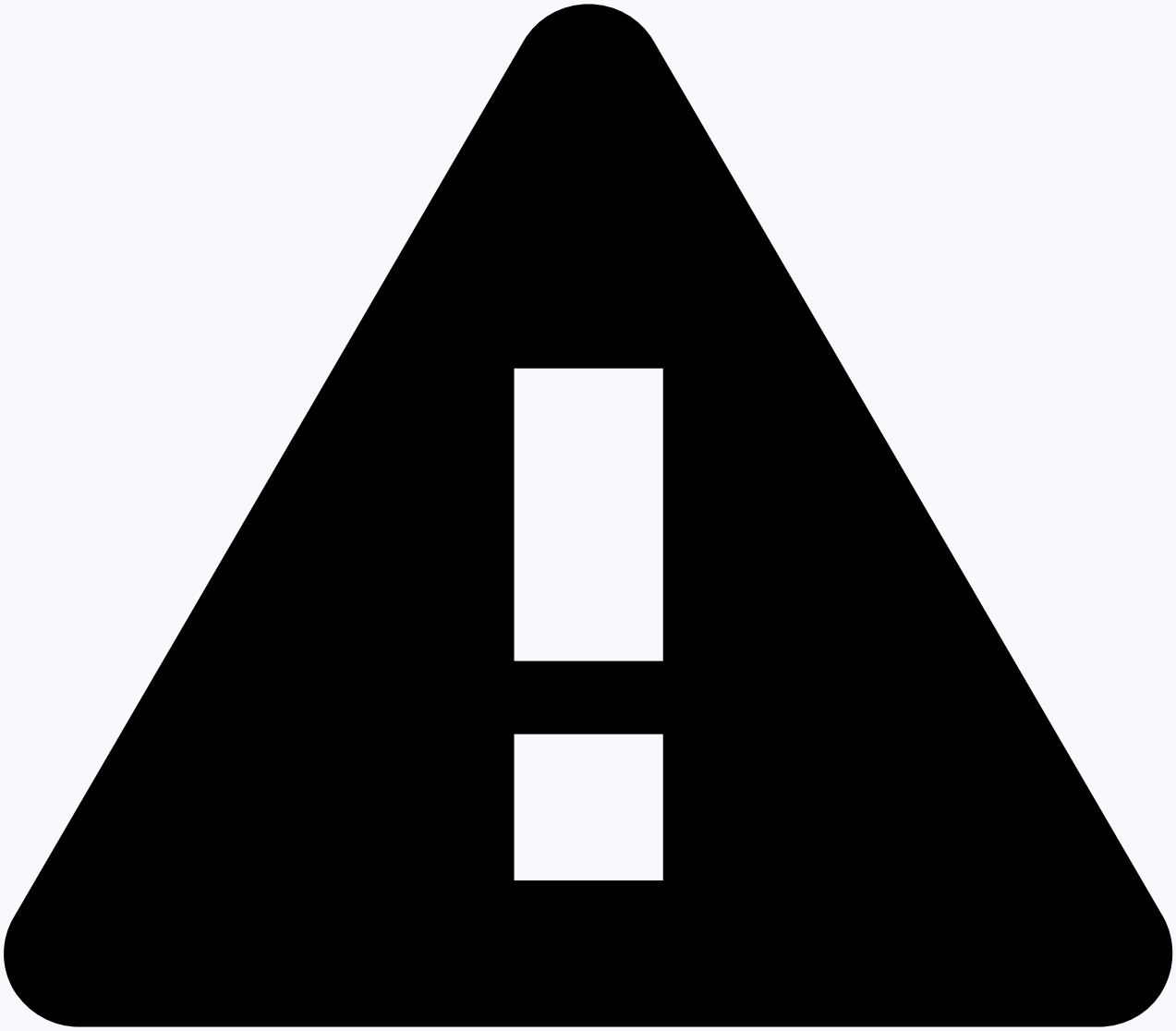
Both routes rescale the step counter with the new time step: the stored step is multiplied by `DT_OLD/DT`, so `DT` may change between two runs of the same case. The geometry may not - a checkpoint carries the grid of the run that wrote it, see troubleshooting.

Writing a checkpoint by hand

The parameter file can place a checkpoint wherever it makes sense, for instance in a user block after the plasma has settled or once a membrane flux has been sampled:

```
S.write_checkpoint("case/checkpoint/particles_Ar.bin",   "particles", "Ar");
S.write_checkpoint("case/checkpoint/averages_Ar.bin",    "averages",  "Ar");
S.write_checkpoint("case/checkpoint/charges.bin",        "charges",   0);
S.write_checkpoint("case/checkpoint/material_TiO2.bin",  "material",  "TiO2");
```

The third argument is the species, the material name, or a placeholder for `charges`. Reading works the same way with `S.read_checkpoint(...)`; as type, `material` and `coverages` are both accepted. The combined DSMC and PIC-MC example uses this to start the plasma from the particle state of a preceding DSMC run.



Do not mix the two routes in one case folder

The parameter file route looks for `<case>/last_shutdown.def`, the Python route for `<case>/checkpoint/last_shutdown.def`, and the Python route writes no coverage state. A case written by one route is therefore not continued by the other - use separate folders when both routes are in play. Checkpoints from release 2.5 and older cannot be continued at all, the particle record changed when the per-particle weights were introduced.

7 Python interface (beta)

> **Status: beta.** The interface ships with release 26 and covers setting up a case, stepping the simulation, sampling histograms on borders, computing or reading a magnetic field, and DSMC cases (`WORKMODE = 0`) including particle sources with an imported emission profile (see Sputter transport). It is still being extended, so keywords and behaviour can change within the 2.x series. The two example scripts in the repository root (`energy_conservation.py`, `pytest_picmc.py`) are marked as provisional by their authors and partly use older keywords - do not copy them unchecked.

Instead of a parameter file, a case can be described by a Python script. `import PICMC` provides the classes `Geometry`, `Border`, `BorderReaction`, `Component`, `Species`, `ParticleReaction`, `Controller` and `Sim`, plus `BEM_Solver` and `BEM_Boundary` for the field solver. The script builds the geometry, the species and the boundary conditions, then starts the simulation and advances it step by step. A case started this way does not read a `.par` file.

The price for the flexibility is that the script has to be written and maintained by hand: the parameter reference on this site documents the RVM keys, not the Python keywords, although both describe the same settings (see mapping).

How the two processes cooperate

The Python script does not simulate anything. A run is a single MPI job with two programs:

- exactly one Python process that executes the case script,
- the `picmc` workers, which hold the mesh and the particles.

Both are started together with MPI's multiple program support:

```
export PICMCPATH=/path/to/picmc-26
export PYTHONPATH=$PICMCPATH/lib
export LD_LIBRARY_PATH=$PICMCPATH/lib:/lib64:$LD_LIBRARY_PATH

mpiexec --bind-to none -np 1 python3 case.py \
  : -np 8 $PICMCPATH/bin/picmc
```

The part after the colon is the worker side. This has three consequences:

- The job has to allocate **one task more** than the number of workers, because the Python process occupies one slot.
- The Python side must run with `-np 1`. Several Python ranks would each initialize the simulation and fight over the workers.
- `picmc` is called without a parameter file - it receives its configuration from the Python process. Running `picmc case.par` and `python3 case.py` side by side is not possible; a case is either RVM based or Python based.

`PYTHONPATH` has to contain `<prefix>/lib`, because that is where the module (`PICMC.so`) and the species database (`pydatabase.py`) are installed. `PICMC_SEED` works as with an RVM case.

Building and installing

`lib/PICMC.so` is part of the default `All` target, so a normal build produces it. The build needs the Python development files; the interpreter is taken from the `PYCONFIGPATH_<goal>` entry in `makefile.inc` (for example `/usr/bin/python3.11-config`). The interpreter that runs a case should be the same version that was used at build time.

The installation receives:

```
lib/PICMC.so            the module itself
lib/pydatabase.py       species and reaction database
py/pysubmit.py          job submission helper
py/<cluster>.json       cluster configurations for pysubmit
sh/pysubmit             shell submit script (builds its own SLURM file)
```

Submitting a run

`py/pysubmit.py` is the counterpart of `rvmslurm`/`rvmmpi`: it builds a SLURM batch file and submits it. The shell script of the parameter-file route and the chaining of several runs are described in Submitting a run.

```
python3 $PICMCPATH/py/pysubmit.py case.py -n 8 -c IST_EPYC -t 48:00:00 -s 4711
```

`-n`/`--ntasks` is the number of workers - the job gets one task more, the driving Python rank. `-c`/`--config` selects the cluster configuration, `-t`/`--time` the wall time and `-s`/`--seed` the seed; `-e`/`--exclusive`, `-N`/`--nodes`, `-m`/`--memory`, `-p`/`--partition` and `-l`/`--nodelist` cover the remaining SLURM options. The generated `<case>.slurm` launches `python3 <case>.py` as one task and the worker program with `-n` tasks, writes `<case>-out.txt` and `<case>-err.txt` and appends the error file to the output file at the end.

One option decides which program the workers are: **`-b`** (`--bem`) starts `bem` instead of `picmc`. Every script that drives the field solver needs it - without it the workers are the wrong program, they ignore the solver's messages, and the run waits forever without writing output (see Computing a magnetic field with the BEM solver).

The wrapper only fills in cluster defaults; the paths come from `py/<cluster>.json`, installed next to `pysubmit.py`:

```
{
  "default_partition": "epyc",
  "allowed_partitions": ["epyc"],
  "picmc_path": "~/picmc-26-EPYC",
  "memory": 16000,
  "mpi_path": "/opt/openmpi-5",
  "mpi_exec": "mpiexec"
}
```

`picmc_path` is the only entry that usually has to be adapted to the local installation.

On a workstation without SLURM, use the `mpiexec` line from above directly.

Progress while the run is going

`pythonscripts/picmc_progress.py` (installed to `<PICMCPATH>/lib` by `make`) prints one line per 0.01 % of the run in the style of `feedback.r` - percent, plot counter, time to the next plot and to the end of the run - plus a `tqdm` bar on a terminal:

```
import picmc_progress

bar = picmc_progress.create(sim, TSIM, NPLOTS)
while sim.running():
  sim.step()
  bar.update()
bar.close()
```

`PICMC_PROGRESS=0` switches it off, `PICMC_PROGRESS=log` drops the bar, `PICMC_PROGRESS_PLAIN=1` leaves every record in the log instead of updating one line for `tail -f`.

A minimal case script

```
import PICMC
import pydatabase

geo = PICMC.Geometry(
  mesh_file = "JustABox.msh",
  mesh_unit = 0.1,
  BB = (
    (-1.0, -0.5, -0.5),   # -x, -y, -z
    (1.5, 0.5, 0.5)       # +x, +y, +z
  ),
  NX = [10] * 10,
  NY = [10] * 4,
  NZ = [10] * 4,
  BOUNDARY_CONDITIONS = (
    ("outlet", "outlet", "outlet"),
    ("outlet", "outlet", "outlet")
  ),
  BORDER_TYPES = {
    "Wall": PICMC.Border(icodec = 1, type = "wall", epsr = 10.0)
  }
)

sim = PICMC.Sim(
  WORKMODE = 1,
  DT = 5e-11,
  TSIM = 1e-4,
  Batchfile = "case",
  Geometry = geo,
  SPECIES = {
    "Arplus": PICMC.Component(NREAL = 1e5, P0 = 1e13),
    "e": PICMC.Component(NREAL = 1e5, P0 = 1e13)
  },
  Materials = [],
  SPECS = pydatabase.SPECIES_LIST,
  Reactions = pydatabase.REACTION_LIST,
  Plots = {
    "density": ["Arplus", "e"],
    "energy": ["Arplus", "e"],
    "FIELD": ["PHI", "E"]
  },
  NPLOTS = 1000,
  TAVG = True
```

```
)

try:
  sim.init()
  while sim.running():
    sim.step()
  sim.exit()
except Exception:
  import traceback
  traceback.print_exc()
  sim.exit()
```

`SPECS` and `Reactions` are what the RVM case takes from `species_definition.r` and the cross-section subroutines: `pydatabase.SPECIES_LIST` holds the known species with mass and charge, `pydatabase.REACTION_LIST` the reaction and scattering parameters. Both can be extended or replaced by own entries.

The `try`/`except` block is not decoration: an exception in the middle of a run leaves the workers waiting, and `sim.exit()` in the handler is what ends them. The shipped example script does the same.

`JustABox.msh` is the small test mesh of the repository example; replace it with a mesh of your own, created as described in the Gmsh page. The physical groups of the mesh have to match the `icodec` values in `BORDER_TYPES`.

Methods of Sim

method	purpose
`init(reload_if_possible = False)`	reads the mesh, distributes the configuration to the workers, harmonizes the weights if `RESAMPLE_SCALE` is set; with `True` it continues an existing checkpoint, see Checkpoints and continuation
`full_run()`	runs until `TSIM` is reached without returning control
`step()`	advances one time cycle
`running()`	`True` while the simulation time is below `TSIM`
`get_time()` / `get_step()`	current simulation time in s / number of finished cycles
`is_species_defined(name)`	whether a species exists in the case
`read_magnetic_field(filename, file_type)`	reads an external magnetic field (same file formats as in the parameter file)
`plot(filename, what, species_name = "")`	forces a plot of one field or species property
`exit()`	tears the run down on the worker side

The loop `while sim.running(): sim.step()` is what makes the interface interesting: between two steps, own code can read the state, log it or change parameters. `full_run()` exists for cases that need none of that.

Keyword arguments of the classes

class	keywords
`Geometry`	required: `mesh_file`, `BB`, `NX`, `NY`, `NZ`; optional: `mesh_unit`, `DIV_X`, `DIV_Y`, `DIV_Z`, `BOUNDARY_CONDITIONS`, `BORDER_TYPES`
`Border`	required: `icodec`, `type`; optional: `T`, `epsr`, `vf_coupling`, `vf`, `reactions`
`BorderReaction`	`species`, `reaction_type` (`plain`, `transmission`, `specular`, `source`), `probability`, `absorption`, `reaction_products`, `product_weights`, `source_value`, `source_unit`; methods `set_emission_Maxwell`, `set_emission_Thompson`, `set_emission_energy`, `set_profile`, `set_probability_function`, `set_temperature_function`, `set_yield_function`, `set_yield2D_function`, `add_energy_histogram`, `add_angular_histogram`, `add_histogram` and their `_array` variants (see histograms)
`Component`	`NREAL`, `P0`, `frozen`
`Species`	`mr`, `charge`, `is_frozen`
`ParticleReaction`	`species_1`, `species_2`, `sigma`, `message`, `delete_1`, `delete_2`, `threshold_eV`, `alpha`, `reaction_products`, `runtime_modification`, `unit_correction_factor`
`Controller`	`PTARGET`, `PSAMPLE`, `RegulationFactor`, `RegulationScale`, `IntegrationTime`, `DifferentiationTime`, `VPMIN`, `VPMAX`, `VPSTART`
`Sim`	required: `WORKMODE`, `DT`, `TSIM`, `Batchfile`, `Geometry`, `SPECIES`, `Materials`; optional: `Reactions`, `SPECS`, `Plots`, `NPLOTS`, `PowerControl`, `TAVG`, `SIGMA_ENERGY`, all `RESAMPLE_*` keys

`Border.vf` accepts a number or a callable object, so a voltage boundary condition can be a Python function instead of a fixed value or a table. The callable is called without arguments at every solve cycle; read the time inside it with `sim.get_time()`. `Sim.PowerControl` takes a `Controller` object and replaces `PSOLL`/`PSAMPLE` plus the power regulation keys of the parameter file; the two input forms differ in their defaults, see power regulation.

Histograms on borders

A border can record what hits it - energy, angle, or both, per species. This is the sampling the parameter file offers for walls and membranes; file names, columns and the handling of the reference direction are the same, see Energy/angle sampling.

The methods sit on the `BorderReaction` of the border and return the object, so calls can be chained:

method	records
`add_energy_histogram(species, E0, E1, nE)`	energy in `nE` bins over `[E0, E1]`
`add_angular_histogram(species, nTheta, nPhi, v)`	polar angle θ in `[0°, 90°]` and azimuth φ in `[0°, 360°]`
`add_histogram(species, nTheta, nPhi, E0, E1, nE, v)`	both, as a correlated distribution
`add_energy_histogram_array(species, nX, nY, x0, x1, x2, E0, E1, nE)`	the energy histogram resolved over an array on the surface
`add_angular_histogram_array(species, nTheta, nPhi, nX, nY, x0, x1, x2)`	the angular histogram resolved over an array
`add_histogram_array(species, nTheta, nPhi, nX, nY, x0, x1, x2, E0, E1, nE)`	both, resolved over an array

`v` is the reference direction of the azimuth, given as a tuple `(vx, vy, vz)`. `x0`, `x1` and `x2` are three corners of the rectangle that the array covers on the surface, each a tuple `(x, y, z)`; `nX` and `nY` are its number of subdivisions. The corners are given in the unit of the mesh file, that is in the unit of `Geometry(mesh_unit = …)`. Wrong units raise no error: the grid then misses the surface and every hit ends up in a single cell. The array only works on a plane surface. In the combined methods `E0`, `E1` and `nE` are optional and fall back to `0` to `1000` eV in a single bin, which is what the two single-purpose methods use for the dimension they do not sample. Mind the total number of bins `nTheta * nPhi * nE` - the same caveat as in the parameter file applies.

```
Arplus_on_membrane = (
  PICMC.BorderReaction(species = "Arplus", reaction_type = "transmission", probability = 1.0)
       .add_histogram(species = "Arplus", nTheta = 18, nPhi = 36,
                      v = (1.0, 0.0, 0.0), E0 = 0.0, E1 = 200.0, nE = 40)
)

membrane = PICMC.Border(icodec = 10, type = "membrane", reactions = [Arplus_on_membrane])
```

The histogram is taken on the physical surface of the border that carries the reaction. Every call creates its own histogram with its own numbered output file, as the file names with the running number suggest - nothing is merged. As in the parameter file, a membrane only samples the side its normal points to.

Mapping of parameter file keys

parameter file	Python
`WORKMODE`, `DT`, `TSIM`	`Sim(WORKMODE = …, DT = …, TSIM = …)`
`Batchfile`	`Sim(Batchfile = "name")`
`BB`, `NX`, `NY`, `NZ`, `DIV_X/Y/Z`	`Geometry(…)`
`BOUNDARY_CONDITIONS`, `BORDER_TYPES`	`Geometry(BOUNDARY_CONDITIONS = …, BORDER_TYPES = {…})`
`SPECIES/NREAL_<sp>/P0_<sp>`	`Sim(SPECIES = {"<sp>": PICMC.Component(…)})`
`species_definition.r`	`Sim(SPECS = pydatabase.SPECIES_LIST)`
cross-section and reaction setup	`Sim(Reactions = pydatabase.REACTION_LIST)`
`NUMBER/DENSITY/ENERGY/FIELD` plots, `NPLOTS`	`Sim(Plots = {...}, NPLOTS = …)`
`POLL/PSOLL/PSAMPLE`, power regulation keys	`Sim(PowerControl = PICMC.Controller(…))`
`RESAMPLE_SCALE` and the other `RESAMPLE_*` keys	`Sim(RESAMPLE_…)`, see Resampling
`add_energy_histogram`, `add_angular_histogram`, `add_histogram`, `add_*_histogram_array`	`BorderReaction.add_*_histogram(…)`, see Histograms on borders
`add_absorption("sp", p)`	`BorderReaction(species = "sp", absorption = True, probability = p)`
`add_source("sp", v, "sccm"/"Pa")`	`BorderReaction(species = "sp", reaction_type = "source", source_value = v, source_unit = "sccm")`
`set_profile("file.pos")`	`BorderReaction.set_profile("file.pos")`, see Sputter transport
`set_emission_thompson/_maxwell/_energy`	`BorderReaction.set_emission_Thompson/_Maxwell/_energy`
`SIGMA_ENERGY`	`Sim(SIGMA_ENERGY = …)`

parameter file	Python
`TAVG`	`Sim(TAVG = …)`

What the interface does not cover yet

- The wall models of the wall interaction model are only partly reachable. `BorderReaction` covers `plain`, `transmission`, `specular` and `source` reactions with absorption, reaction products, the emission profiles `set_emission_Maxwell`, `set_emission_Thompson` and `set_emission_energy`, the spatial `set_profile`, and the histograms of energy and angle. What has no Python counterpart yet are `set_emission_sputter` (Thompson without the ion energy, i.e. `Ub/alpha` only), the temperature dependent vapour source `add_vapor_source`, the formula-based emitters (`set_emission_function_*`), the energy loss functions and the surface chemistry of the RVM side, so those pages show RVM examples only.
- Note the order inside `Border.reactions`: `set_profile` and `set_emission_*` attach to the reaction that was declared last, so a source reaction with an emission profile has to be listed after the reactions of the same border.
- `Materials` is accepted by `Sim` but the code marks it as unfinished.
- There is no Python counterpart of the ppp post-processing module. `Sim.plot()` writes single snapshots during the run; the regular evaluation still runs on the `.pos` files as described in Postprocessing.
- The interface is one Python rank by design, so analysis that needs the particle data of all workers has to happen after the run, not inside the script.

8 Magnetic field solver (beta)

> **Status: beta.** The Python interface of the field solver ships with release 26. Its keywords follow the classes `BEM_Solver` and `BEM_Boundary` and can still change within the 2.x series. The solver itself is the same one the `bem` binary uses.

A plasma case with a fixed magnetic assembly (permanent magnets, iron yoke) needs the static magnetic field of that assembly. It is computed once per geometry by a boundary element method, written to a binary file and then read by the simulation. In the parameter file route the setup is a `.bem` file plus the `bem` binary (see Magnetic fields from a fixed assembly); from Python the same solver is available as `PICMC.BEM_Solver`, and the `.bem` file is replaced by `BEM_Boundary` objects.

Computing the field and simulating the plasma remain two separate runs: the solver writes a field file, and the simulation reads it with `read_magnetic_field` (see Loading an external field).

How a run is organised

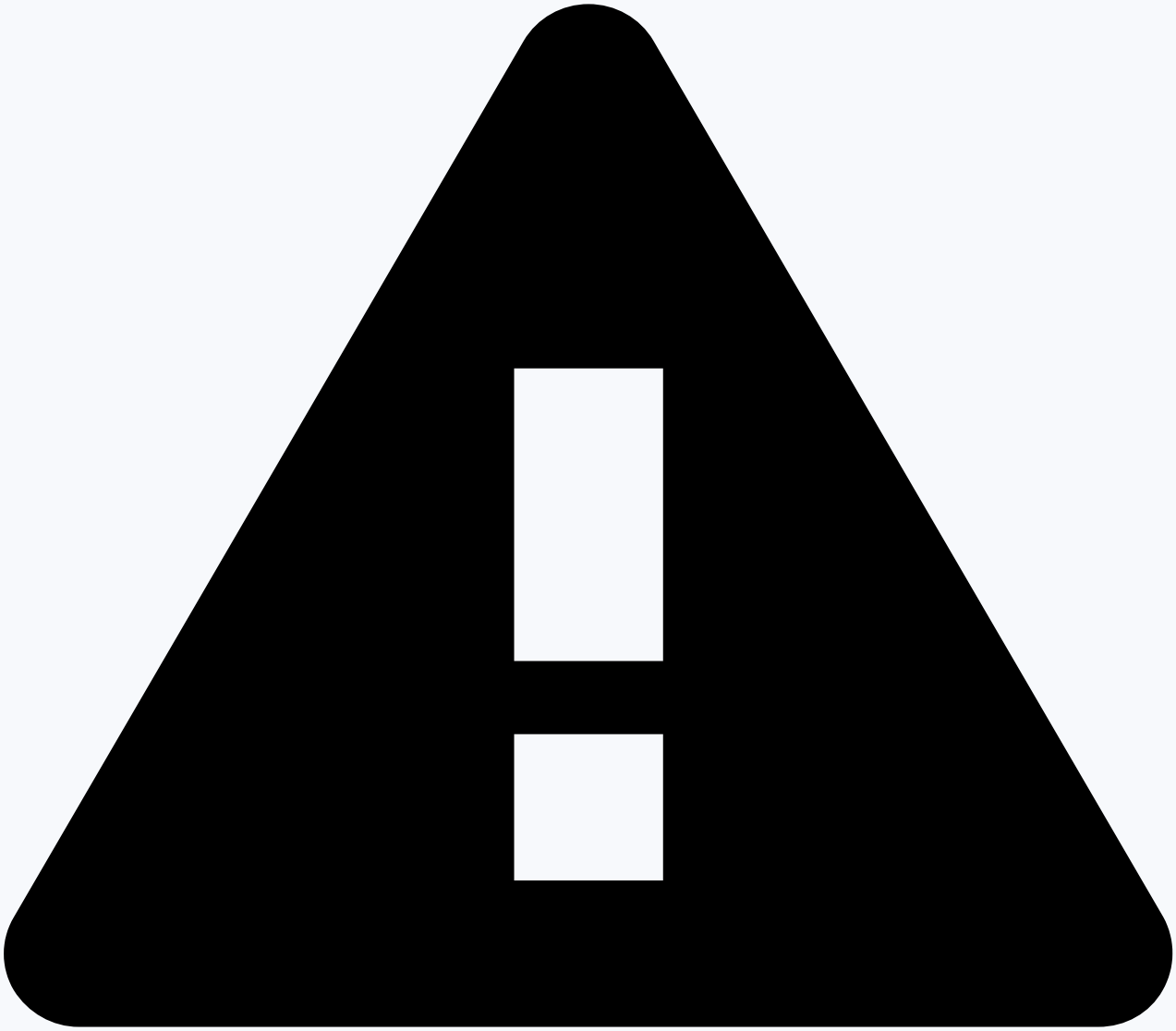
The solver uses the same two-program pattern as `Sim`: exactly one Python rank drives the solver, and the `bem` workers do the computation - including the MPI option `--bind-to none` and the "one task more than workers" rule, see How the two processes cooperate.

```
export PICMCPATH=/path/to/picmc-26
export PYTHONPATH=$PICMCPATH/lib
export LD_LIBRARY_PATH=$PICMCPATH/lib:/lib64:$LD_LIBRARY_PATH

mpiexec --bind-to none -np 1 python3 magnet_bem.py \
  : -np 8 $PICMCPATH/bin/bem
```

On a SLURM cluster the run is submitted with the Python wrapper, and one option selects the worker program: **-b** makes the `-n` workers the `bem` binary instead of `picmc`.

```
python3 $PICMCPATH/py/pysubmit.py magnet_bem.py -n 8 -c IST_EPYC -b -t 1:00:00
```



Do not forget the -b flag

-b is not optional here: without it the wrapper starts `picmc` workers. Those do not answer the solver's messages: the driving rank waits in its first receive, the workers wait for a command that never comes, and `<script>-out.txt` stops after its first two lines. The job then holds its nodes until the time limit without a single progress line.

A minimal script

```
import PICMC

geo = PICMC.Geometry(
  mesh_file = "magbar90.msh",
  mesh_unit = 1e-3,
  BB = (
    (-37.5, 505.0, 180.0),   # -x, -y, -z
```

```
    ( 37.5, 1195.0, 540.0)   # +x, +y, +z
  ),
  NX = [25],
  NY = [23] * 10,
  NZ = [10] * 12
)

R = 1.24   # remanence of the magnets in T
codecs = {
  "M1centBottom": PICMC.BEM_Boundary(icodec = 183, type = "interface",
                                     val = -R, domain = 1, neighbourdomain = 0),
  "M1centTop":    PICMC.BEM_Boundary(icodec = 184, type = "interface",
                                     val =  R, domain = 1, neighbourdomain = 0),
  "M1centSide":   PICMC.BEM_Boundary(icodec = 191, type = "interface",
                                     val =  0, domain = 1, neighbourdomain = 0),
  # ... one entry per physical surface of the magnet mesh
  "M1yoke":       PICMC.BEM_Boundary(icodec = 192, type = "interface",
                                     val =  0, domain = 2, neighbourdomain = 0)
}

solver = PICMC.BEM_Solver(
  Geometry = geo,
  Domains  = [1, 1.05, 1000],
  Codecs   = codecs
)

try:
  solver.init()
  solver.solvepoisson(batchname = "magbar90")
  solver.write_vectorfield("BEM_field.pos")
  solver.write_scalarfield("BEM_potential.pos")
  solver.write_binaryfield("B-Field.bin")
  solver.write_textfield("B-Field.txt")
  solver.exit()
except Exception:
  import traceback
  traceback.print_exc()
  solver.exit()
```

`solver.exit()` in the handler has the same reason as in a case script: an exception leaves the workers waiting.

Keyword arguments

class	keywords
`BEM_Solver`	required: `Geometry`, `Domains`, `Codecs`
`BEM_Boundary`	required: `icodec`, `type`, `val`, `domain`, `neighbourdomain`

`Geometry` is an ordinary `PICMC.Geometry`, but it describes the **magnet** mesh, not the plasma mesh: `mesh_file` and `mesh_unit` of the magnet geometry, plus `BB`, `NX`, `NY` and `NZ` of the box the field is evaluated in. `BORDER_TYPES` is not needed - the boundary description of the field solver is `Codecs`.

`Domains` is the list of relative permeabilities; `Codecs` maps a name to one `BEM_Boundary` per physical surface of the magnet mesh. The names are free, `icodec` is not: it is the tag of the physical surface in the mesh, and it has to match exactly (see the mesh).

What the entries mean

- `type = "interface"` marks a surface between two magnetic materials. `val` is the remanent flux density in T along the surface normal, so the sign gives the magnetisation direction: the pole faces of a bar magnet get `+R` and `-R`, the lateral surface gets `0`.
- `type = "dirichlet"` fixes the magnetic scalar potential to `val`; `type = "neumann"` sets its normal derivative to `val`. A coil described as a surface current uses a `neumann` surface and two `dirichlet` faces at `+1` and `-1` (see Coil current computation with BEM).
- `domain` and `neighbourdomain` are **indices into `Domains`**, not mesh tags. `0` is the surrounding vacuum, `1` the magnet material and `2` the yoke in the example above. A codec with `domain = neighbourdomain` or `neighbourdomain = 0` needs no second material.
- Every physical surface of the magnet mesh needs an entry, including the ones that contribute nothing (`val = 0`). Surfaces left out are not constrained.

The mesh

The magnet assembly gets its own Gmsh mesh, generated like any other (see From STEP to a Gmsh mesh). Only the surfaces of the assembly are meshed and given physical groups: the magnet mesh of the example is nothing but the 20 surfaces of the two bar magnets and their yokes, without an air volume and without an outer boundary. Their tags become the `icodec` values, and they are independent of the numbering of the plasma mesh - it is a different file with its own groups.

Output

method	result
`write_binaryfield(filename)`	the binary field for `read_magnetic_field(filename, file_type = "checkpoint")`
`write_textfield(filename)`	the same field as text, for `read_magnetic_field(filename, file_type = "external_unstructured")`
`write_vectorfield(filename)`	`B` as a `.pos` view, for a check in Gmsh or ParaView
`write_scalarfield(filename)`	the magnetic scalar potential as a `.pos` view
`write_boundaryconditions(qfile, ufile, jfile)`	the Neumann and Dirichlet boundary conditions and the ICP source field as `.pos`; all three names are optional, without a name that file is not written

`solvepoisson(batchname = "name")` takes the name of the run. The argument corresponds to `Batchfile` in the parameter file and is currently only accepted, not evaluated - it does not change the result.

Reusing the field on a different plasma mesh

`write_binaryfield` writes the field quad by quad, in the layout of the solver's `Geometry`. A plasma case that resolves the field differently - other `BB`, other `NX`, `NY`, `NZ` - cannot read that binary. For that case `write_textfield` writes the same field points as plain text, one line per point:

```
x y z Bx By Bz
```

That is the format of the unstructured external field file, so the plasma case loads it with `BFIELDFILE` or `read_magnetic_field(filename, "external_unstructured")` and interpolates it onto its own grid. The two cases are then independent: the field solver only has to cover the volume, not the resolution.

Two conditions come from the reader and not from the solver. The field points have to span the whole simulation volume, because the reader interpolates between them but never extrapolates; and the coordinate columns are compared with the nodes of the plasma mesh in metres, while `write_textfield` divides them by the `mesh_unit` of the solver's geometry - the same scaling the `.pos` views of the run use. Check the numbers when the magnet mesh is not in metres.

In the parameter file route the text file is written together with the binary: `write_binaryfield` in the `.bem` block writes `B-Field.txt` next to `B-Field.bin`.

Differences to the parameter file route

- No `.bem` file. The block `Boundary NAME = {icodec = ...; type = ...; val = ...; domain = ...; neighbourdomain = ...;};` becomes one `BEM_Boundary` with the same five keywords, `Domains` and `R` become the `Domains` list and a plain Python variable.
- `BEMMESH` in the parameter file has no counterpart; the magnet mesh is the `mesh_file` of the solver's `Geometry`.
- The solver is a separate script, not part of the case script. Field computation and plasma run stay two runs, as with `bem` and `picmc`.

9 Simulation pipeline

PICMC does not simulate a coating process on its own. It is one stage in a chain of simulation codes that starts with the geometry of a coater and ends with a predicted or explained coating result: layer thickness on a curved substrate, an optimised mask, the energy distribution of the particles that arrive at the surface. This page shows that chain and what comes out of it, so the single-stage documentation is easier to place.

The rest of this documentation describes PICMC itself. Here the question is the opposite one: which part of a coating problem does PICMC answer, what does it hand to the next stage, and what does it need from the stage before.

Why a chain instead of one code

Each stage of a sputter process lives on its own length and time scale, and each one needs a different numerical method. A plasma needs the motion of individual electrons and ions; the transport of sputtered atoms through metres of chamber is a molecular flow problem; the film that grows on the substrate is a surface process on the scale of nanometres.

- **Make the invisible visible.** The ion and neutral flux that hits the target and the substrate is what determines the result, and it cannot be measured inside a running coater. Simulation gives access to it.
- **Virtual prototyping.** Chamber, shields, masks, targets and substrate motion can be varied before hardware is touched — including geometries that do not exist yet.
- **Pre-optimisation of large lines.** An industrial coater can be designed in-silico long before it is built.
- **Fewer trial runs.** A well-designed parameter study replaces a series of coating runs, which saves time, cost and material.

The chain

The picture below is the pipeline as it is used for optical coatings: first the plasma and the sputter source, then the way back to the substrate.

Read it as two halves. The upper half answers "how much material leaves the target, and where"; the lower half answers "where does it land, and how do I shape the flux so that it lands evenly". PICMC owns the first half up to the erosion profile.

The green-dashed branch is optional and answers a different question: not how thick the layer becomes, but whether particles arrive on the part that should not be there. It reuses the same fields and does not change anything in the rest of the chain — see PALADIN.



Electron density Ar+ ion flux on the target Time-averaged ion flux

The PICMC stage for a source with two cylindrical targets: electron density, Ar^+ ion flux on the target, and the time-averaged flux from which the erosion profile is derived. Source: Pflug, A. et al., Proc. 15th ISSP (2019).

The chain is meant to be run in a loop

The two dashed arrows are the actual point of the framework. A single pass through the chain gives one prediction; a digital twin is what you get when the pass is repeated with changed inputs, and the interesting part is deciding which inputs to change:

- **Inner loop, mask and geometry.** The source stays as it is, only the mask, the shielding or the substrate motion is adapted until the thickness profile on the part matches the target. This loop is cheap because the plasma stage does not have to be repeated, and it is the loop that is automated today (see the release notes for release 26 below).
- **Outer loop, everything upstream.** If the target cannot be reached by shaping the flux, the cause is on the source side: target geometry and erosion, gas pressure and composition, magnetic field, shields, or the material of the target itself. Then the plasma stage is re-run with changed parameters, and the inner loop starts again.

Doing this by hand is a parameter study. Doing it automatically needs a goal function — a rate profile, a spectrum, a uniformity number — and a simulation that is fast and stable enough to be called a few dozen times. Both are what the framework is built for.

Stage by stage

Stage	Done by	Needs	Delivers
Background gas and magnetic field	DSMC, field computation	Geometry, pressures, gas composition, target currents	Gas density and flux field, magnetic field for the plasma stage
Plasma	**PICMC**	Geometry, gas field, magnetic field, species and cross-sections	Electron and ion density, potential, **ion flux on every surface**
Erosion profile	**PICMC** (time-averaged ion flux) and the sputter yield	Ion flux, target material, impact angles	Erosion depth along the racetrack
Sputter transport	DSMC	Erosion profile, gas background	Flux of sputtered atoms towards the substrate, angular and energy resolved

Stage	Done by	Needs	Delivers
Deposition on the substrate	Substrate raytracer or matrix deposition	Atom flux, substrate motion, mask geometry	Local deposition rate on the real part
Mask and geometry	Optimisation loop around the ray tracer	A goal function, e.g. a rate profile that compensates the substrate curvature	Mask shape, substrate motion, shielding
Particle contamination	**PALADIN**, an optional branch	Plasma and gas fields of the same geometry, particle species, wall properties	Contamination on the part, and the surface each particle came from
Verification	Measurement	Coated part	Spectra, thickness profiles, particle contamination

Every stage can also be used on its own. Most questions do not need the whole chain — a plasma question often ends at the ion flux, a uniformity question starts with a measured or estimated erosion profile.

What PICMC contributes to the chain

- **Ion flux on surfaces.** Time-averaged fluxes are written as cell-centred fields and are the input for the erosion estimate. Without the averaging over many RF periods the flux is too noisy to integrate.
- **Angular and energy distributions.** Particles that cross a boundary — a membrane, a monitor substrate — can be recorded as histograms of impact angle and energy. Downstream atomistic growth simulations need exactly that, not a single mean value.
- **Collision and surface statistics.** Where particles collide, where they are absorbed or desorb, how much energy a surface receives: this is what explains a profile rather than just predicting it.
- **Restartable long runs.** A plasma stage for a real coater geometry runs for hours to days; checkpoints make it possible to stop after the plasma has settled and continue with a changed downstream setting.



What is new in release 26

Two things on this page changed with release 26:

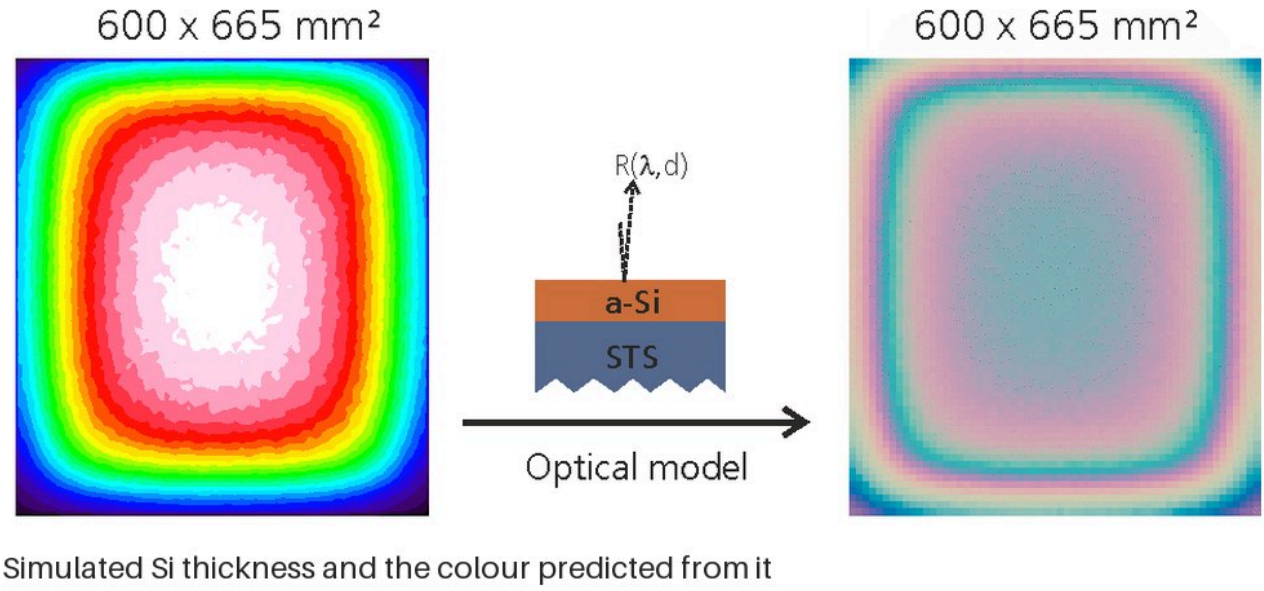
- **Python interface (beta).** A case can be built as a Python script instead of a parameter file, which is what makes automated loops practical: the geometry, the species, the surface reactions and the sampling can be varied from one script and the run submitted from the same place. See Python interface.
- **Substrate raytracer.** The deposition and mask stage can be computed with a separate ray tracer (C++ core with a Python interface) that optimises the mask automatically with SciPy instead of the user iterating on mask shapes. It is an addition, not a replacement: the matrix

> deposition algorithm stays documented and usable, and the two share the same histogram files but not the same code, so numbers are not identical. See Substrate raytracer.

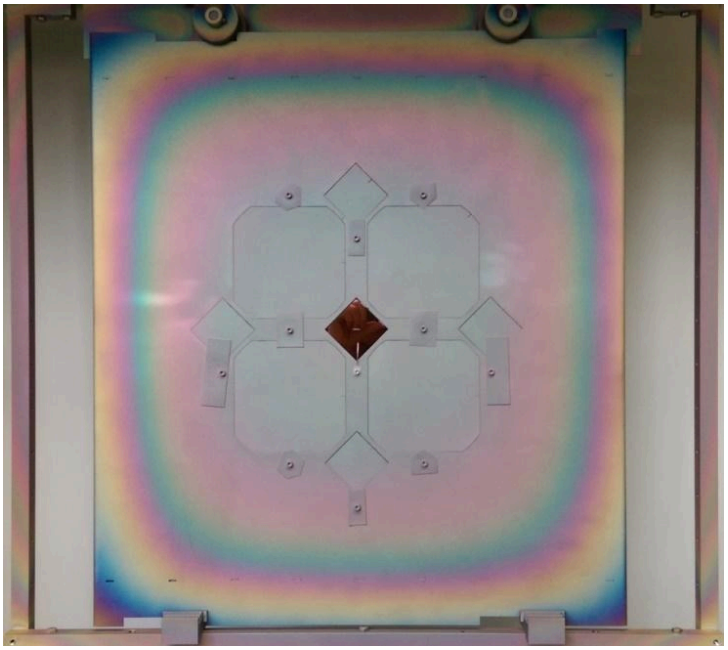
Examples

In-line hot-wire CVD coater

A DSMC model of an in-line HWCVD coater in two and three dimensions produces the silicon thickness profile on the moving carrier. The computed profile was compared with the interference colour of the coated steel carrier — a cheap and very sensitive check of the whole transport model, because a few percent of rate difference changes the colour.



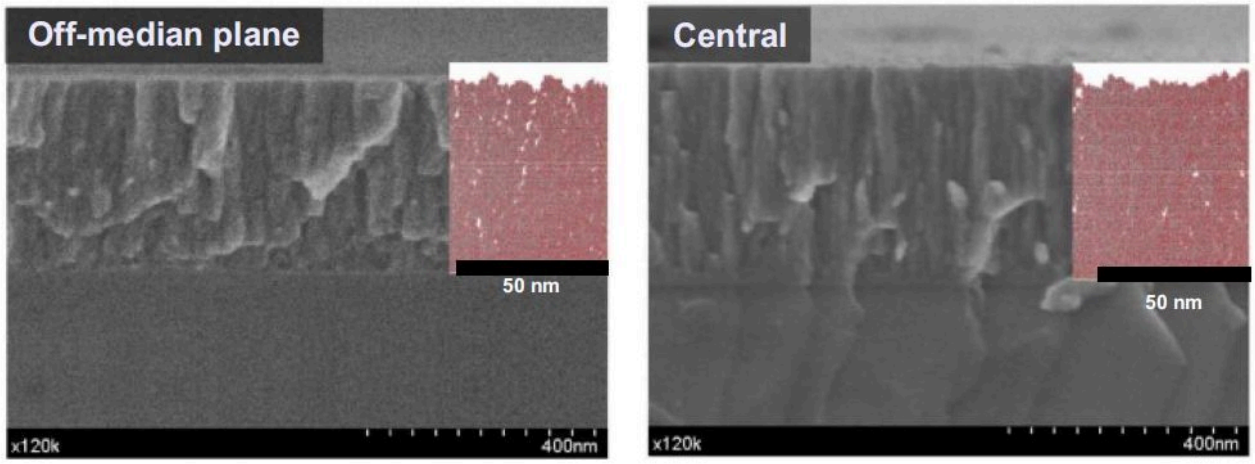
Simulated Si thickness on the 600 × 665 mm² carrier and the interference colour computed from it with an optical model.



The coated steel carrier: the interference colour is the check.

Reactive sputtering of TiO_2 with atomistic growth

A coupled DSMC / PIC-MC simulation of reactive TiO_2 sputtering including ionised species and negative O^- ions, followed by a kinetic Monte Carlo growth simulation. Samples perpendicular and tilted at 70° in the centre and off-centre position. The model reproduces the difference between metallic, oxide and transition mode — the regime where the process is hardest to keep stable.

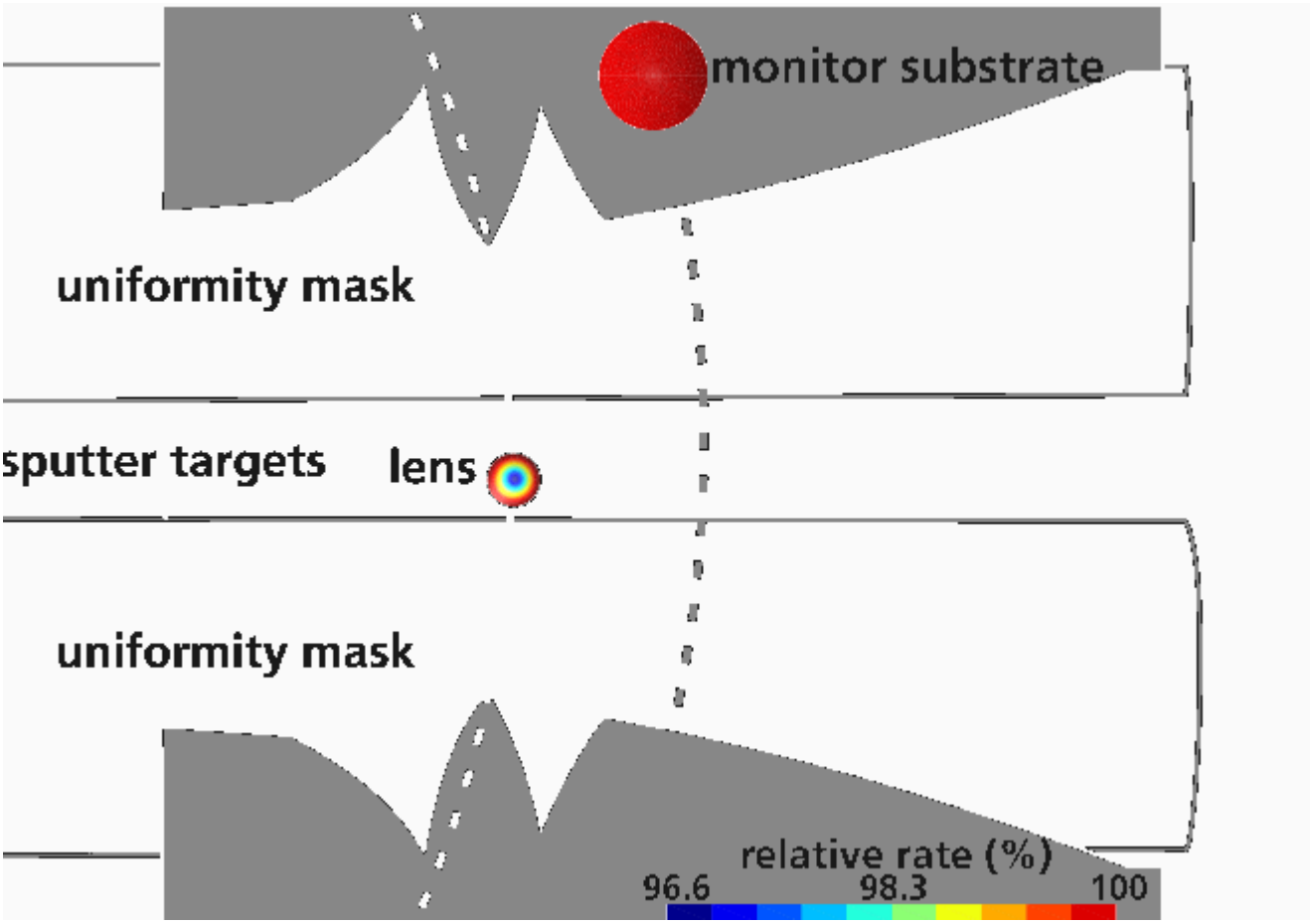


Cross-sections of the grown TiO_2 film, each perpendicular and tilted by 70°, at off-median position and in the centre, with a segmented inset for the column analysis. Source: Tonneau, R. et al., J. Phys. D: Appl. Phys. 54 (2021) 155203.

Bandpass filter on a convex lens

The problem: a bandpass filter deposited on a convex lens shifts towards shorter wavelengths at the edge, because the deposition rate drops with curvature and the angle of incidence grows. At the edge the effective thickness is about 95 % at 16°, and the passband moves out of the specified range.

The chain was run end to end: PICMC for a realistic sputter erosion profile, DSMC for the sputter flux towards the lens, ray tracing for the local rate on the rotating lens for a given mask, and a mask optimisation whose goal function follows the curvature of the lens. With the optimised mask the passband centre stays essentially constant over the lens, below 2 nm shift at 16°.



The geometry the optimisation works on: uniformity mask, sputter targets, lens and monitor substrate, with the resulting relative deposition rate. Source: Pflug, A. et al., Proc. 15th ISSP (2019).



The result: the coated lens of the study.

Concave substrate for a membrane optics project

A rotating concave spherical substrate, argon plasma, silicon sputtering, a membrane at the target position. PICMC delivers the electron density in the plasma, the silicon flux towards the substrate and — as the actual result — the static absorption profile on the substrate. Profiles taken at different arm positions of the rotation agree with the measurement. The same run also produces the angular and

energy histograms of the Ar^+ ions and Si atoms that reach the membrane, which is the input for the atomistic growth simulation.

Macroparticles: where does the contamination come from

Dust in a PVD coater is a separate problem that reuses the same plasma results. **PALADIN** takes the DSMC / PIC-MC output as input and traces macroparticles through the gas, modelling what actually acts on them — its setup, parameters and output are documented in its own section:

- charging and de-charging in the plasma,
- gas friction,
- thermophoresis,
- van der Waals forces at surfaces.

What comes out is not a single trajectory but statistics: a spatially resolved contamination profile, a size distribution, and the distribution of forces along the paths. The most useful output in practice is the backward question — which part of the target or of the chamber is likely to produce a particle that later reaches the substrate. In one case the answer was "particles generated close to the plasma", which identified them as target agglomerates and gave a starting point for countermeasures. Most of these particles do not even adhere to the substrate, but the impact alone can damage the coating.

Coupling to CFD

For processes where the gas flow is not in the free-molecular regime — atmospheric plasma, ventilated chambers, particle transport in air — the background flow comes from a CFD code (OpenFOAM, COMSOL) and only the particle and plasma part is done here. The same macroparticle code is used for aerosols in room air and for particles in liquids, which is how the method travels outside coating.

What the chain does not do

- **Atomistic film growth is external.** Columnar growth, density and stress of the growing film are simulated by a separate kMC/MD code (for example NASCAM). PICMC supplies the flux that goes into it.
- **Optical design is external.** The spectra that a layer stack produces are computed by an optical model; the chain delivers thickness and angle distributions for it.
- **The chain needs calibration.** Cross-sections, sputter yields and surface coefficients are only as good as the measurements behind them. A model that was never compared with a coated sample is a hypothesis, not a prediction.

What comes next

The case studies in the examples above are usually condensed into simple design rules — substrate thickness, pressure range, curvature limit, magnetic angle — because that is what a process engineer can use at the machine. The direction the framework is moving in:

- embedding the chain into recipe development as a routine tool rather than a project,
- surrogate models to make the parameter loops fast enough for large process windows,
- coupling with in-situ diagnostics so that the model is recalibrated during production.

Where to start with PICMC

If you want to use the PICMC stage of this chain:

- Parameter file for the setup, including the surface definitions that receive the ion flux.
- Python interface if the case is generated rather than written by hand.
- Averaged quantities for the time averaging that makes the flux usable by the next stage.
- Particle output for the files that leave a run.

Modeling

10 Units and conversion factors

SCCM - Gas flow rate

Sccm stands for *Standard cubic centimeters per minute*. This rate is proportional to the number of gas molecules per time unit. In this case, the *standard* conditions are defined as follows:

- Pressure p = 1.01325 bar = 1.01325e+5 Pa
- Temperature T = 0 °C = 273.15 K
- Volume V = 1 cm³ = 1e-6 m³

The number N of particles within one cm³ can be obtained from the ideal gas equation via:

$$N = \frac{p\,V}{k_B\,T}$$

With the Boltzmann constant k_B=1.3806504e-23 J/K (see Wikipedia) we obtain N = 2.68675e+19.

Since sccm is the flow rate given per minute but we would like to have the number of particles per second, we have to divide N by 60. The result is:

$$1\,\mathrm{sccm} \approx 4.47796 \cdot 10^{17}\,\mathrm{s}^{-1}\,[\mathrm{Particles/second}]$$

Conversion of ampere to sccm

With the electron charge e = 1.60218e-19 you get that one ampere corresponds to N = 6.24150e+18 particles per second. Divided by the above given value of particles per sccm, you get:

$$1\,\mathrm{A} \approx 13.9397\,\mathrm{sccm}$$

Pumping Speed

Definition

The pumping speed S_p is usually defined as **pumped gas volume / time**. In case of an area A with *ideal* pumping behaviour, the pumping speed is:

$$S_\mathrm{p} = \frac{A\,c}{4}$$

- c = Mean thermal velocity of a gas molecule,
- A = pumping area.

The mean thermal velocity can be obtained from the Maxwellian distribution by:

$$c = \sqrt{\frac{8\,k\,T}{m\,\pi}} = 145.5 \frac{\mathrm{m}}{\mathrm{s}} \sqrt{\frac{T[\mathrm{K}]}{m[\mathrm{u}]}}$$

At a temperature of 300 K, the following velocities are obtained:

Species	Relative mass [u]	Mean thermal velocity c [m/s]
Kr	83.8	275
Ar	39.9	399
O*$_2$*	32.0	446
SiH*$_4$*	32.1	445
N*$_2$*	28.0	476
H*$_2$*	2.02	1770
H	1.01	2510

Select transmission probability of a pumping surface

A surface of a given area *A* can have the maximal pumping speed given above. The pumping speed of a real pump S_p is usually well below. This can be mended to assign a **transmission probability** *f*, which is obtained as follows:

$$f = \frac{4\,S_\mathrm{p}}{A\,c}$$

- S_p = external pumping speed
- *A* = surface area
- *c* = gas velocity

Example

A surface sized 16x75 cm² should have a real pumping speed of 500 l/s (0.5 m³/s) for Argon at *T* = 300 K. The according transmission probability calculates as follows:

$$f = \frac{4 \cdot 0.5}{0.16 \cdot 0.75 \cdot 398.5} \approx 0.0418$$

Relation between gas flow, pumping speed and pressure

The time derivative of the ideal gas equation can be written as:

$$\frac{d}{dt} p\,V = k\,T \frac{d}{dt} N$$

In equilibrium, the left hand side is zero. The change of the particle number becomes the inflow minus the outflow:

$$\frac{d}{dt} N = F_\mathrm{in} f_\mathrm{sccm} - \frac{c}{4} A_\mathrm{p} f_\mathrm{p} n \stackrel{!}{=} 0$$

with the following parameters:

- F_{in} = Inflow in sccm (for the sccm unit, see above)
- f_{sccm} = 4.479 x 10^{17} particles/s (see above)

- A_p = Pumping area
- f_p = Transmission probability of pumping surface
- n = Particle density

The particle density can again be expressed by using the ideal gas equation, yielding:

$$\frac{k\,T\,F_{\mathrm{in}} f_{\mathrm{sccm}}}{p} = \frac{c}{4} A_{\mathrm{p}} f_{\mathrm{p}} = S_{\mathrm{p}}$$

where S_p is the pumping speed in [m³/s]. Thus, we end up with a species-independent factor Φ which determines the relation between pressure, gas flow and pumping speed:

$$p = \Phi \frac{F_{\mathrm{in}}}{S_{\mathrm{p}}} \qquad \Phi = k\,T\,f_{\mathrm{sccm}}$$

Typical values for Φ are listed in following.

Temperature [K]	Φ	Φ with S_p given in liters/second
300	0.001854	1.854
350	0.002163	2.163
400	0.002472	2.472
450	0.002781	2.781
500	0.003091	3.091

Mean free path

In following, the main points from http://en.wikipedia.org/wiki/Mean_free_path are summarized.

The mean free path λ is defined as the mean travelling distance of a particle between two collisions. If a particle ensemble has a Maxwellian velocity distribution, the following relations hold:

$$\lambda = \frac{1}{\sqrt{2}\,n\,\sigma} = \frac{k_{\mathrm{B}}\,T}{\sqrt{2}\pi\,d^2\,p}$$

whereof the variables mean the following:

- n = Density
- p = Pressure
- k_B = Boltzman constant
- d = Diameter of the gas particle
- T = Temperature
- λ = Effective collision cross section

The latter formula shows that λ is inversely proportional to the total pressure. From http://www.kayelaby.npl.co.uk/general_physics/2_2/2_2_4.html the following numerical values are obtained for a few common gases at T=25 °C:

Species	$p \times \lambda$ [Pa mm] Atomic diam	ter [nm]
Ar	6.34	0.382
H_2	11.06	0.288

Species	$p \times \lambda$ [Pa mm] Atomic diam	ter [nm]
N_2	5.96	0.394
O_2	6.41	0.380

Temperature vs. kinetic energy

In many sources the electron temperature T_e is defined via the simple relation:

$E = k_B T_e$.

However for a Maxwellian velocity distribution that is not correct. Here, any degree of freedom contributes to the mean kinetic energy by *1/2 k_B T*. Thus, in a 3D Maxwellian distribution, the following relation holds:

$E = 3/2 k_B T$.

By this formula, a mean energy of 1.0 eV corresponds to a kinetic temperature of 7736.3 K (and not 11604 K as results from the wrong formula mentioned above).

11 Statistical constraints

DSMC parameters with respect to total pressure

Scaling behaviour of cell dimension, particle number and time step

For a gas flow simulation, the set up of the simulation domain and the particle statistics requires to decide values for certain parameters, which are listed in the following. For simplicity we assume that all cells have a cubical shape.

- a = Cell dimension [m]
- V_c = Cell volume = a^3
- N_{REAL} = Quotient between physical number of particles N and number of simulation particles N_S
- δt = Period of one time cycle

The cell dimension should be less than the mean free path λ which itself reciprocally depends on total pressure p[1], which leads to the following considerations:

$$a \sim \lambda \sim p^{-1},$$

$$V_\mathrm{c} \sim a^3 \sim p^{-3}.$$

Since the travelling distance of a particle during one time cycle should be also less than the mean free path, a similar relation holds for the time cycle:

$$\delta t \sim p^{-1}.$$

From the ideal gas equation, we know for the physical number N_c of particles within a cell, that:

$$N_\mathrm{c} = \frac{p\,V_\mathrm{c}}{k\,T} \sim p^{-2},$$

provided that the cell dimension a is adjusted reciprocally with total pressure p as mentioned above. It is possible to keep the number N_S of simulation particles per cell constant by adjusting N_{REAL} according to the pressure. In this case N_{REAL} satisfies the relation:

$$N_\mathrm{s} = \frac{N_\mathrm{c}}{N_\mathrm{REAL}} \stackrel{!}{=} \mathrm{const.} \Rightarrow N_\mathrm{REAL} \sim p^{-2}.$$

Especially at higher pressure, the computational effort is almost completely dominated by the collision routine. In the so-called *No time counter method* [2], the number N_{COLL} of pre-selected collision pairs is determined as follows:

$$N_\mathrm{COLL} = \frac{N_\mathrm{s}^2 N_\mathrm{REAL}\,\delta t < \sigma\,c >_{max}}{V_\mathrm{c}}.$$

If we put in the above described relations, we find for each cell that N_{COLL} is independent from the total pressure p. However, given that the number of cells increases with p^3 and the time cycle δt behaves reciprocal with p the total computational effort for a gas flow system with constant volume and for a constant total real-time period increases with p^4.

Numerical example for Argon

For Argon we know, that for a total pressure of 1 Pa, the mean free path is about 6 mm. Thus, in order to be on the safe side, we chose a cell size of 5 mm at p = 1 Pa.

For the time cycle we consider the mean thermal velocity c of Ar at room temperature, which is approx. 400 m/s. As mean travelling distance at p = 1 Pa, we chose a length of 4 mm which is well below the mean free path resulting in:

$\delta_t(p = 1\ \mathrm{Pa}) = 10^{-5}\ s.$

The physical number of particles within a cell sized 5x5x5 mm^3 at T = 300 K (room temperature) can be estimated from the ideal gas equation as N = 3.02 x 10^{13}. If we would like to maintain a constant number of simulation particles per cell of N_S = 10, the resulting value for N_{REAL} is 3 x 10^{12}.

Based on these starting considerations and on the scaling relations shown above, we obtain the following set of simulation parameters as a function of total pressure:

Pressure [Pa]	Cell dimension a	Time cycle [s]	N_{REAL} for 10 simulation particles per cell
0.1	50 mm	1 x 10^{-4}	3 x 10^{14}
1.0	5 mm	1 x 10^{-5}	3 x 10^{12}
10.0	0.5 mm	1 x 10^{-6}	3 x 10^{10}
100.0	0.05 mm	1 x 10^{-7}	3 x 10^{8}
1000.0	5 µm	1 x 10^{-8}	3 x 10^{6}
10000.0	0.5 µm	1 x 10^{-9}	3 x 10^{4}
100000.0 (~ 1 bar)	0.05 µm	1 x 10^{-10}	3 x 10^{2}

Footnotes

1. see Mean free path ↩
2. Bird, G. A.; *Monte Carlo Simulation in an engineering context*, In: Progr. Astro. Aero. **74** (1981) 239-255. ↩

12 Numerical constraints

Limitations to the time step width

Cell spacing

One limitation for the maximum time step width is given due to the implemented Monte Carlo algorithm. According to their velocities particles must stay within a grid cell for at least on time step. Otherwise the collision probabilities are not evaluated correctly, i.e. a >>cell hopping<< occurs. Hence, the product of time step and particle velocity must not exeed the cell spacing:

$\Delta t \cdot v < \Delta s.$

The velocity of charged particles can be estimated from the maximum voltage in the simulation setup. If neglecting the energy loss due to inelastic collisions, the voltage potential is applied as kinetic energy to the charged particle:

$$\frac{m\,v^2}{2\,q} = E_{\mathrm{kin}}[\mathrm{eV}] = U_{\max}[\mathrm{V}] \quad \Rightarrow \quad v = \sqrt{\frac{2\,U\,q}{m}}\left[\frac{\mathrm{m}}{\mathrm{s}}\right],$$

with q = 1.60211 · 10^{-19} [As]. The table below lists electron velocities for various energies and the accordant distance covered at a time step width of 10^{-11} *[s]*:

Energy [eV]	$v_{Electron}$ [m/s]	$v_{Electron} \cdot \Delta t$ [µm]
100	5.932 · 10^6	59.32
200	8.389 · 10^6	83.89
300	1.027 · 10^7	102.7
400	1.186 · 10^7	118.6
500	1.326 · 10^7	132.6

Collision frequency

According to the Wikipedia[1] the collision frequency Z can be descibed as the rate of collisions between two species in an given volume and time. Regarding both species A and B as hard spheres, Z is expressed as :

$$Z = n_A\, n_B\, \sigma_{AB} \sqrt{\frac{8\,k_B\,T}{\mu_{AB}\,\pi}}\ [m^{-3}\,s^{-1}]$$

where

- $n_A\,[m^{-3}]$ is the number density of species A
- $n_B\,[m^{-3}]$ is the number density of species B
- $\sigma_{AB}\,[m^2]$ is the collision cross section of A and B
- $k_B\,[m^2\,kg\,s^{-2}\,K^{-1}]$ is the Boltzmann's constant
- $T\,[K]$ is the temperature
- $\mu_{AB}\,[kg]$ is the reduced mass of A and B

Since we are interested in the lower bound of the time step, only the highest cross section is of interest which is usually the reaction of the main gas species with itself.

Electron collision frequency

According to http://www.uccs.edu/~tchriste/courses/PHYS549/549lectures/plasma.html the collision frequency for neutrals in an ideal gas can be estimated from:

$$\lambda = N\,\pi\,d^2\sqrt{\frac{8\,k_B\,T}{m\,\pi}}.$$

Whereas *N* is the number density, *d* the effective atom radius, k_B the Boltzmann constant, *T* the gas temperature and m the atom mass. Transfered to the electron collision frequency in a partly ionized plasma one could use the above estimation with:

```
**πd<sup>2</sup> → σ*, the electron-neutral collision cross section with the primary gas species
**T → T<sub>e</sub>*, the electron temperature whereas 1 eV = 11604.5 K
**m → m<sub>e</sub>*, the electron rest mass of 9.1091 · 10<sup>-31</sup> kg
```

Thus, for a H_2-plasma with T_{H2} = 500 K, T_e = 5 eV and an approximated $\sigma = 10^{-19}$ m^2 one gets:

```
*$\lambda=1.0838\cdot 10^{10}\,\textrm{Hz at }P_{H_2} = 500\,\textrm{Pa}$
*$\lambda=2.1677\cdot 10^{10}\,\textrm{Hz at }P_{H_2} = 1000\,\textrm{Pa}$
*$\lambda=4.3355\cdot 10^{10}\,\textrm{Hz at }P_{H_2} = 2000\,\textrm{Pa}$
```

For a valid simulation, time step width must be well below the reciprocal of the collision frequency.

Plasma frequency

The plasma frequency ω_p is the angular frequency of the electron oscillations around the ions. It depends only on electron mass and density:

$$\omega_p = \sqrt{\frac{n\,e^2}{\epsilon_0\,m_\mathrm{e}}}$$

- *n* = Electron density
- *e* = 1.6021 · 10^{-19} As
- ε_0 = 8.8542 · 10^{-12} F/m
- m_e = 9.1091 · 10^{-31} kg

In order to properly resolve the plasma oscillations, the electron time step should be well below the respective oscillation period e.g., dt < 0.1 ω_p, as can be estimated from the following table:

Electron density [m^{-3}]	Plasma frequency [s^{-1}]	Oscillation period [s]
10^{15}	1.78 · 10^{9}	5.605 · 10^{-10}
10^{17}	1.78 · 10^{10}	5.605 · 10^{-11}
10^{19}	1.78 · 10^{11}	5.605 · 10^{-12}

Debye length

The Debye length L_D describes the electrical shielding capability of the plasma. The potential difference induced by a local excess charge will drop to 1/e of the original potential at the distance of L_D. For electrons, the Debye length is given by:

$$L_D = \sqrt{\frac{\epsilon_0 k_B T_e}{n_e e^2}} \simeq 69.01 \cdot \sqrt{\frac{T_e}{n_e}} \simeq 6070 \cdot \sqrt{\frac{E\,[\mathrm{eV}]}{n_e}}.$$

Here, ε_0 is the vacuum permittivity, k_B the Boltzmann constant, T_e the electron temperature, n_e the electron density and e the electron charge.

In order to properly resolve the electric potential distribution in the simulation grid, the cell size has to in the same order of magnitude as the Debye length. From [Langdon70][2] follows, that the cell spacing shall be less than $\pi\, L_D$. In [Tskhakaya2007][3] a constraint of:

$$\delta_x \leq 3.4 L_D$$

is suggested. This is what we also adopted in our constraint check message for the Debye length on plasma simulation startup. If this constraint is broken, artificial numerical oscillations will take over leading to "artificial heating", i.e. more energy will be gained in the plasma than actually put into the system via the electrodes, and the simulation will become unstable.

Our simulation tool cannot predict the plasma density of any given simulation case. Therefore, we give some estimations for typical cases:

- We assume 5 eV for electrons as this is a very typical electron temperature in magnetron sputtering discharges.
- A plasma density of 10^{17}/m³ corresponds to an experimentally realistic density (still with moderate power density, e.g. 50-100 W for a 3 inch cathode).
- A plasma density of less than 10^{15}/m³ is what we often have in simulation (especially 3D) as feasible compromise between realistic power density and computation time.

As an example, for n_e = 10^{15}m^{-3} and T_e=5 eV the constraint 3.4 L_D is computed to 1.459 mm.

If the simulation output shows "Failure" at the Debye length check this is merely an informative message based on the assumed plasma density of 10^{15}/m³ or 10^{17}/m³ and electron temperature of 5 eV. If the actual simulation setup is different from that it may be still numerically valid. If in doubt it helps to check if two simulation runs with slightly changed cell spacing yield the same overall results.

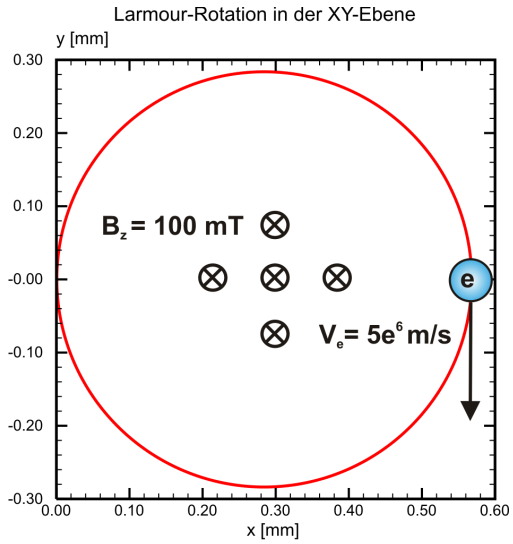
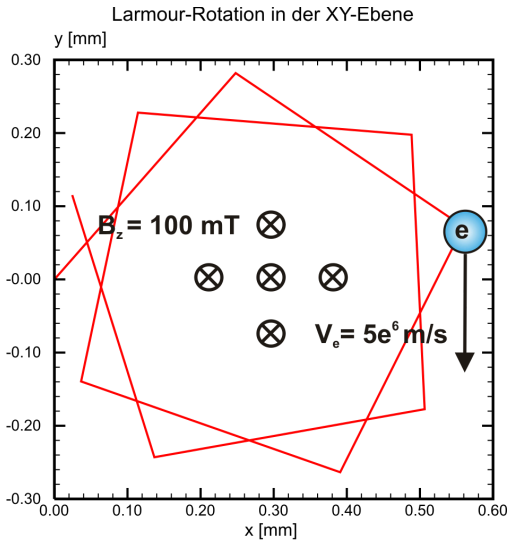
Gyrokinetics

The implemented leap-frog algorithm from Boris is very robust at comparatively small computation costs. Nevertheless if the chosen time step witdh is too large, gyrokinetic trajectories could not be resolved sufficiently as seen in the pictures below. For a proper estimation use the following equation:

$$R_{\mathrm{Larmor}} = \frac{m_{\mathrm{e}}\ |\vec{v}|}{q_{\mathrm{e}}\ \left|\vec{B}\right|}, \text{ with}$$

- m_e = 9.1091 · 10^{-31} kg
- q_e = 1.6021 · 10^{-19} As

This formular yields the path radius of an electron at constant velocity v perpendicular to a homogenous magnetic field B. **We still lack informations on how the time step related granularity is influencing the plasma simulation and the results respectively.** However, for a proper simulation of gyrokinetic plasmas it should not exeed a certain limit. Apparant from the pictures below, at magnetic field strengths around 100 mT and electron velocities around 5 · 10^6 m/s, the time step width for electrons should not exeed 10^{-10} s. These values are commonly valid for magnetron discharges in PVD sputtering processes on which the PICMC simulation environment is focused on.

Larmor-Rotation at Different Time-Step Widths	
	
Time Step Widths = 10^{-11} [s]	Time Step Widths = 10^{-10} [s]

Limitations to the cell spacing

Isotropic vs. anisotropic cell spacing

Within the Particle-in-Cell (PIC) algorithm charged particles are mapped on the cell nodes by a weighting scheme. Then, in a successive computation step the Poisson Equation of the electric field can be solved consistently to the weighted charge distribution. The PIC weighting scheme in general delocalized a point charge to a finite sized space charge with an appropriate shape function. The charge on a certain node is then given by the fraction of the space charge within the cell volume.

In a homogenous grid the particle shape is equivalent to the cell spacing at every point. Thus, for a bilinear (trilinear) weighting scheme, the charge on a certain cell node can be described as the convolution of two square pulses over time:

$$Q_{\mathrm{Node}}(t) = f_{\mathrm{Charge}}(x,t)\, f_{\mathrm{Node}}(x) = \int \Pi_{\Delta x}(x - v_x\, t)\, \Pi_{\Delta x}(x)\, dx.$$

The resulting plot for this mathematical operation can be seen below. Now if the particle's speed v_x is increased **or** the cell width Δx is decreased the variation of Q(x,t) over time increases:

$$\left.\frac{Q(t)}{dt}\right|_{(v_x,\,\Delta x)} < \left.\frac{Q(t)}{dt}\right|_{(2\,v_x,\,\Delta x)} \quad \text{and} \quad \left.\frac{Q(t)}{dt}\right|_{(v_x,\,2\,\Delta x)} < \left.\frac{Q(t)}{dt}\right|_{(v_x,\,\Delta x)}.$$

If the charge on a nodes varies over time so does the electric field resulting from the solution of the poisson equation:

$$\frac{Q(t)}{dt} \approx \frac{E(t)}{dt}.$$

Footnotes

1. https://en.wikipedia.org/wiki/Collision_frequency ↩

2. LANGDON, A. B.: Effects of the Spatial Grid in Simulation Plasmas. In: Journal of Computational Physics 6 (1970), p. 247-267 ↩

3. TSKHAKAYA, D.; MATYASH, K.; SCHNEIDER, R. & TACCOGNA, F.: The Particle-In-Cell Method. In: Contributions to Plasma Physics 47 (2007), No. 8-9, p. 563-594 ↩

13 Collision routine

Rather than analyzing the individual trajectories of any particle, a collision can generally occur for each pair of particles which are within the same sub-cell.

The collision rate within a cell is calculated as follows: The product of the relative velocity and the corresponding cross section $v_\mathrm{r} \cdot \sigma$ multiplied with the time step δt spans a cylindrical volume. The ratio of this volume and the cell volume is the collision probability p:

$$p = \frac{v_\mathrm{r}\,\sigma(E)\,\delta t}{V_\mathrm{cell}}, \qquad E = \frac{1}{2} m_\mathrm{r}\, v_\mathrm{r}^2.$$

Thus, if we assume that the collision cross section σ and all relative velocities v_r are constant, the number N_coll of expected collisions in one cell between two species A and B can be simply calculated by:

$$N_\mathrm{coll} = \begin{cases} \dfrac{N_\mathrm{A}\, N_\mathrm{B}\, N_\mathrm{Real}\, v_\mathrm{r}\, \sigma\, \delta t}{V_\mathrm{Cell}} & \mathrm{A} = \mathrm{B} \\ \quad / & \\ \dfrac{N_\mathrm{A}\, N_\mathrm{B}\, N_\mathrm{Real}\, v_\mathrm{r}\, \sigma\, \delta t}{2\, V_\mathrm{Cell}} & \mathrm{A} = \mathrm{B} \end{cases}.$$

The additional parameters in this equation are:

- N_A, N_B = The number of simulation particles of species A and B within the cell
- N_Real = The scaling factor between real and simulated particle number (which is assumed to be the same for species A and B, here)

The formulas above are written for one common scaling factor. In the code the weight is a property of the individual particle: all particles of a species start with `NREAL_SP`, but resampling may give them different weights, so every pair is evaluated with its own w_A and w_B (see the note on the majorant below).

NTC algorithm

In reality, all relative velocities and corresponding cross sections are different. In this case, the so called "NTC-Method"[1] is applied, which is a variation of the acceptance-reject method of statistics: For each collision type, the maximum product $\max(v_\mathrm{r} \cdot \sigma)$ is determined from the sampled data of $\sigma(E)$. Then, the number of pair selections N_sel is determined as maximum possible collisions in one cell (i. e. assumed that the product $v_\mathrm{r} \cdot \sigma$ is maximal for all possible collision pairs in one cell):

$$N_\mathrm{sel} = \begin{cases} \dfrac{N_\mathrm{A}\, N_\mathrm{B}\, N_\mathrm{Real}\, \max(v_\mathrm{r} \cdot \sigma)\, \delta t}{V_\mathrm{Cell}} & \mathrm{A} = \mathrm{B} \\ \quad / & \\ \dfrac{N_\mathrm{A}\, N_\mathrm{B}\, N_\mathrm{Real}\, \max(v_\mathrm{r} \cdot \sigma)\, \delta t}{2\, V_\mathrm{Cell}} & \mathrm{A} = \mathrm{B} \end{cases}.$$

Because the particles in a cell no longer share one weight, the majorant is formed from the largest weights present: the pre-selection is normalized with $\max(w_\mathrm{A}) \cdot \max(w_\mathrm{B})$ of the cell, and the rate is bounded by the largest $v_\mathrm{r} \cdot \sigma$ found there. A cell is evaluated in one pass, so the majorant costs nothing extra. The fractional part of N_sel is carried over to the next time step per cell instead of being truncated, so no rate is lost to rounding.

In the following, N_sel collision pairs are randomly selected within the sub-cell. The actual relative velocity and energy is calculated from:

$$v_\mathrm{r} = ||\vec{v}_\mathrm{A} - \vec{v}_\mathrm{B}||, \qquad E = \frac{1}{2} m_\mathrm{r}\, v_\mathrm{r}^2, \qquad m_\mathrm{r} = \frac{m_\mathrm{A}\, m_\mathrm{B}}{m_\mathrm{A} + m_\mathrm{B}} \ (\text{reduced mass}).$$

For a selected collision pair, the collision is actually performed, if the following random experiment holds:

$$R < \frac{v_\mathrm{r}\, \sigma(E)}{\max(v_\mathrm{r} \cdot \sigma)},$$

where R is a random number in [0, 1].

How a collision is applied to the partners

A collision is not simply written onto both partners. Because the two partners may represent different numbers of real particles, the post-collision velocity is applied with the merge probability

$$P_k = \frac{\min(w_1, w_2)}{w_k},$$

i. e. fully to the lighter partner ($P = 1$) and only in a fraction $\min / \max$ of the collisions to the heavier one. The new velocity is a weighted blend of the old and the fully scattered velocity, and the perpendicular energy is restored afterwards. The kinetic energy of the pair is therefore preserved in **every individual collision** and not merely in the statistical mean. This matters in cells where the weights differ strongly, for example after resampling has been active.

General NTC implementation with reaction products

We denote $< v_\mathrm{r}\, \sigma >$ as the mean product of relative velocity and cross section of a collision type within one cell. Further, we have species A with weighting factor w_A, species B with weighting factor w_B and reaction product C with weighting factor w_C. It shall be assumed that $w_\mathrm{A} > w_\mathrm{B}$. Further, in the current simulation volume, there may be a scaling factor $S_\mathrm{R} = 1$ for all weighting factors. Under these conditions, the physical number $N_\mathrm{col,phys}$ of collisions within a cell becomes:

$$N_\mathrm{col,phys} = \begin{cases} \dfrac{N_\mathrm{A}\, w_\mathrm{A}\, N_\mathrm{B}\, w_\mathrm{B}\, S_\mathrm{R}^2 < v_\mathrm{r}\, \sigma > \delta t}{V_\mathrm{Cell}} & \mathrm{A} = \mathrm{B} \\ / & \\ \dfrac{N_\mathrm{A}\, w_\mathrm{A}\, N_\mathrm{B}\, w_\mathrm{B}\, S_\mathrm{R}^2 < v_\mathrm{r}\, \sigma > \delta t}{2\, V_\mathrm{Cell}} & \mathrm{A} = \mathrm{B} \end{cases}.$$

To determine the number of collisions occurring for the simulation particles, this quantity is to be divided by their respective weighting factor w_A or w_B and by S_R:

$$N_\mathrm{col,A} = \frac{N_\mathrm{A}\, N_\mathrm{B}\, w_\mathrm{B}\, S_\mathrm{R} < v_\mathrm{r}\, \sigma > \delta t}{V_\mathrm{cell}} = \frac{w_\mathrm{B}}{w_\mathrm{A}} \cdot \frac{N_\mathrm{A}\, N_\mathrm{B}\, w_\mathrm{A}\, S_\mathrm{R} < v_\mathrm{r}\, \sigma > \delta t}{V_\mathrm{cell}}$$

$$N_\mathrm{col,B} = \frac{N_\mathrm{A}\, N_\mathrm{B}\, w_\mathrm{A}\, S_\mathrm{R} < v_\mathrm{r}\, \sigma > \delta t}{V_\mathrm{cell}}.$$

With $w_\mathrm{A} > w_\mathrm{B}$ the heavier weight w_A is the bottleneck of the channel. At setup time the code stores it as `weightPreselect` and the ratio $w_\mathrm{B}/w_\mathrm{A}$ as `PostselectSpecies1`, which is the share of collisions in which the heavier partner participates completely. `PostselectSpecies2` is 1 for the lighter species. At runtime the same ratio is formed per pair from the actual particle weights, so a particle that was reweighted by resampling is treated correctly. If we now have reaction products of species C the number of reaction products created by the collision becomes:

$$N_\mathrm{rp,C} = \frac{1}{w_\mathrm{C}\, S_\mathrm{R}} \cdot \frac{N_\mathrm{A}\, N_\mathrm{B}\, w_\mathrm{A}\, w_\mathrm{B}\, S_\mathrm{R}^2 < v_\mathrm{r}\, \sigma > \delta t}{V_\mathrm{cell}} = \frac{w_\mathrm{B}}{w_\mathrm{C}} \cdot \frac{N_\mathrm{A}\, N_\mathrm{B}\, w_\mathrm{A}\, S_\mathrm{R} < v_\mathrm{r}\, \sigma > \delta t}{V_\mathrm{cell}}.$$

For $w_\mathrm{B} > w_\mathrm{C}$, multiple reaction products are created for each collision, if $w_\mathrm{B} < w_\mathrm{C}$ only a fraction of $\frac{w_\mathrm{B}}{w_\mathrm{C}}$ of the collisions result into a reaction product.

The weights of the reaction products do **not** enter the rate normalization of the channel: the number of candidate pairs is fixed by the weights of the two reactants. A channel that creates very many products per event is therefore a matter of the chosen product weight: a larger w_C reduces the number created per collision, at the price of fewer, heavier product particles.

Generally, the post-collision selection factor for species A/B can be written as:

$$F_{(\mathrm{A,B}),\mathrm{postselect}} = \frac{w_\mathrm{A}\, w_\mathrm{B}}{w_\mathrm{Preselect}} \cdot \frac{1}{w_{(\mathrm{A,B})}}.$$

Handling of frozen species

In some scenarios, species can be *frozen*, i. e. they are loaded into memory and take part within the collision but do not move themselfes. Also, any post-collision velocity is not applied to frozen species. An example is a transport simulation using the species $\mathrm{H}, \mathrm{H}_2, \mathrm{H}_2^+, \mathrm{e}^-$ where $\mathrm{H}_2^+, \mathrm{e}^-$ are frozen. The purpose of such a simulation is to split the time scales in order to speed up the simulation. In an RF parallel plate plasma with the considered species, the population of H_2^+ and e^- becomes stable already after a few microseconds, while the stabilization time of the H population (as reaction product from dissociation of H_2) is larger by many orders of magnitude. Given the species A and B, then for a *frozen* species B the collision numbers as described above become:

$$N_{\mathrm{col,A}} = \frac{N_\mathrm{A}\, N_\mathrm{B}\, w_\mathrm{B}\, S_\mathrm{R} <v_\mathrm{r}\, \sigma> \delta t}{V_\mathrm{cell}} = \frac{w_\mathrm{B}}{w_\mathrm{A}} \cdot \frac{N_\mathrm{A}\, N_\mathrm{B}\, w_\mathrm{A}\, S_\mathrm{R} <v_\mathrm{r}\, \sigma> \delta t}{V_\mathrm{cell}}$$

$$N_{\mathrm{col,B}} \stackrel{!}{=} 0.$$

A frozen species contributes its density and mean velocity from the time-averaged cell data and enters with the species weight `NREAL_SP`. The consequences for the sampling are:

- No weight majorant is needed on the frozen side, and the cell-local $v_\mathrm{r} \cdot \sigma$ majorant is switched off for such a channel: the reservoir has no finite maximum velocity. The global majorant of the cross-section table is used instead.
- The collision partner is a dummy particle generated from the averaged distribution (`generateCollisionTestparticle`), so nothing has to be stored for the frozen species.
- The post-collision velocity is never applied to the frozen partner, which corresponds to $P = 0$ for that side.

The bottleneck of such a channel is the weight of the *moving* species, because that is what limits how many real collisions one test collision stands for. Reaction products of a third species follow the ratio $w_\mathrm{moving}/w_\mathrm{C}$ as above - if a channel produces an implausibly large number of products per event, increase the product weight `NREAL_C`.

Where this is implemented

The whole routine lives in `Quad/Cycle-Functions/Collisioncycle.cxx`:

- `PICMC_Quad::SetCollision(...)` - registers a collision channel: cross-section table, threshold, products, and the weight normalization (`weightPreselect`, `PostselectSpecies1/2`). Called during setup for every channel defined in `sigma.r`.

- `PICMC_Quad::CollisionCycle(...)` - per time step and per collision channel: builds the majorants, selects the candidate pairs and runs the accept test, cell by cell.
- `PICMC_Quad::collision(...)` - executes one accepted pair: threshold handling, counters, merge probabilities.
- `PICMC_Quad::momentumTransfer(...)` - scatters the pair and applies the weighted merge described above.
- `PICMC_Quad::generateCollisionTestparticle(...)` - builds the dummy partner for a frozen species.

Footnotes

1. **N** o **T** ime **C** ounter Method, see e. g. Bird, G.; *Monte Carlo Simulation in an engineering context*, In: Progr. Astro. Aero. **74** (1981) 230-55 ↩

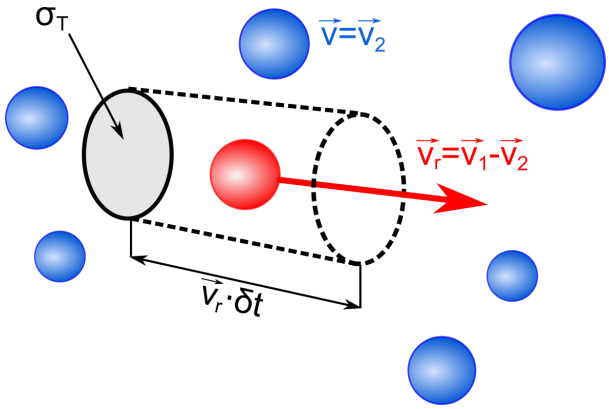
14 Statistical collision sampling

Given are two particle species 1, 2 with their weighting factors N_{R1}, N_{R2}, a cell with volume v_{cell} and particle numbers per cell N_1, N_2.

The derivation uses one weight per species. In the code the weight belongs to the individual particle and may differ within a cell after resampling; the sampling then uses the largest weight present in the cell as the majorant and the actual pair weights in the accept test.

The statistical collision treatment estimates the number of collision between particles, which belong to the same cell with cell volume v_{cell}. It is considered, that - within a time step δt

- two particles with a relative velocity $\vec{v}_r := \vec{v}_1 - \vec{v}_2$ and a total collision cross section σ_T put up a *collision volume* of $\sigma_T ||\vec{v}_r|| \delta t$ (see figure below).



The ratio of this *collision volume* and the cell volume can be considered as collision probability p_{12} between two particles within a cell:

$$p_{12} = \frac{\sigma_T v_r \delta_t}{v_{cell}}, \text{ where } v_r := ||\vec{v}_r||$$

In the NTC method, for pre-selection of the maximum reasonable number of collision pairs, the maximum collision probability $p_{12.max}$ is obtained from

$$p_{12.max} = \frac{(\sigma_T v_r)_{max} \delta_t}{v_{cell}}.$$

Collisions between species with same weight factor

In case, both species have the same weight factor $N_R = N_{R1} = N_{R2}$ the maximum number $N_{col.max}$ of physical collisions is

$$N_{col.max} = N_R N_1 N_R N_2 p_{12.max}$$

However, for the number N_{sel} of *selected* collision pairs, this number has to be divided by N_R since performing a collision with the stored super particles affects in reality N_R physical particles:

$$N_{sel} = N_R N_1 N_2 p_{12.max}.$$

As a side note, in case both species are equal and $N := N_1 = N_2$, the number of pre-selected collision pairs has to be divided by factor two (since in case of equal species the order of collision partners does not count):

$$N_{sel} = \frac{1}{2} N_R N^2 p_{12.max}.$$

For the number N_{per} of actually performed collisions, each selected collision pair has to undergo a random experiment, where the following relation is checked (R# is a random number between]0, 1[):

$$R\# < \frac{p_{12}}{p_{12.max}}.$$

Thus, the mean number $< N_{per} >$ of performed collisions is given as

$$\langle N_{per} \rangle = N_{sel} \times \frac{p_{12}}{p_{12.max}}.$$

Collisions between species with different weight factors

In case, N_{R1}, N_{R2} are different, let's call the low and high statistical weight factor N_{RL}, N_{RH} with $N_{RL} < N_{RH}$. Then, the maximum number of *physical* collision is

$$N_{col.max} = N_{RH} N_{RL} N_1 N_2 p_{12.max}.$$

In the selected collision pairs the collision shall be actually executed each time for the species with low statistical weight and only N_{RL}/N_{RH} times for the other species with the high statistical weight. Under this constraint, the number of selected collision pairs has to be multiplied by N_{RL} to represent the physical number of executed collisions. Therefore, $N_{col.max}$ has to be divided by N_{RL} yielding

$$N_{sel} = N_{RH} N_1 N_2 p_{12.max}.$$

In case, the collision yields **reaction products** of a third species with statistical weight N_{R3} the number N_{rp} of generated reaction products per executed collision is

$$N_{rp} = \frac{N_{RL}}{N_{R3}}.$$

Depending on whether N_{RL}/N_{R3} is bigger or lower than one, either multiple reaction products must be generated per collision or the generation of a single reaction product is determined by random experiment.

If generation of multiple reaction products per time step shall be avoided, a higher number of collision pairs must be pre-selected:

$$N_{sel} = \frac{N_{RH} N_{RL}}{N_{R3}} N_1 N_2 p_{12.max}$$

In this case the number of generated reaction products after collision is one.

Collisions, where one of the collision partners is frozen

In a time-scale splitting scheme, one approach is to consider a certain species distribution as constant, and *freeze* all particles of this species during simulation. Practical examples are e.g.:

- Freeze the Ar background gas distribution and only consider movement of electrons and ions in electron time scale
- Freeze the charge distribution (electrons + ions) in a reactive plasma and only consider creation of radicals in neutral time scale

Let's denote N_{RF} the weight factor of the frozen species and N_{RM} the weight factor of the moving species in a collision. Then, for the number of pre-selected collision pairs follows

$$N_{sel} = N_{RF} N_1 N_2 p_{12.max}.$$

The number of reaction product of a third species with weight factor N_{R3} is then obtained via

$$N_{rp} = \frac{N_{RM}}{N_{R3}}.$$

15 Cross sections

Important notes on cross sections

A good introduction about the topic of (physical) cross sections can be found on http://en.wikipedia.org/wiki/Cross_section_(physics). If extending or using the cross section database of PIC-MC some terminology issues should be taken into account. Here are the most common terms and definitions:

- **Differential scattering cross section**
  - Energy **and** angular dependant elastic or inelastic scattering cross section σ(e,θ).
  - Measuring angular scattering distributions seems very demanding. Hence, accordant data is hard to find on the web.
- **Angular integrated or "total" scattering cross section**
  - Integration of σ(e,θ) over all angles θ ([0,π]) yields σ(e).
  - Using an accordant formular for the energy dependant scattering anisotropy on this cross section yields the momentum transfer cross section.
- **Momentum transfer cross section**
  - Elastic or inelastic momtentum transfer is not the same as elastic or inelastic scattering.
  - You can have a scattering event but the transferred momentum depends on the scattering anisotropy.
  - Momentum transfer cross sections can be obtained from swarm analysis experiments together with the use of Boltzmann equation solvers.
  - The compilations on SIGLO and JILA consists of elastic and inelastic momentum transfer cross sections (not total scattering as I once thought).
- **Effective momentum transfer cross section**
  - This is the sum of all, elastic and inelastic, momentum transfer cross sections.
  - Important: We need the elastic momentum transfer cross sections otherwise we double count the effect of inelastic momentum transfer.
  - Important: Most Boltzmann equation solvers use the effective momentum transfer. Thus, you mainly find effective momentum transfer cross sections, e.g. on JILA and SIGLO.
  - Important: Some compilations, e.g. SIGLO, wrongly labeled effective momentum transfer with elastic momentum transfer. These inconsistencies are sometimes mentioned and sometimes not.
- **Scattering anisotropy**
  - If there are no differential cross sections available, scattering anisotropy and angular distribution, respectively can be approximated.
  - Check Okhrimovskyy [1] on how to approximate anisotropic electron scattering based on a screened Coulomb potential.
  - Important: For the approach of Okhrimovskyy02 et. al. the total elastic scattering cross section and not the elastic momentum transfer cross section must be used.

Energy sampling in cross section array

Previously we used a linear sampling for the energy in the cross section array, i.e. $E(i) = E_0 \cdot i / N$, whereas E0 is the maximum energy in eV and N is the array resolution. However, using a linear sampling scheme we had to distinguish two simulation cases: (i) DSMC gas flow simulation and (ii) PIC-MC plasma simulation. This was because a higher energy resolution in the low energy regime is needed to correctly compute gas flow dynamics and neutral collisions, respectively. To get rid of this limitation we could either increase the array resolution or use a higher order sampling scheme. The latter case was more pratical. Thus, we chose a second order sampling, i.e. $E(i) = E_0(i / N)^2$, which gives a better resolution in the low energy regime. The crucial range hereby is the thermal energy around 25 meV, which is sufficiantly resolved using the second order sampling as seen in the figure below.

Energy sampling in cross section array

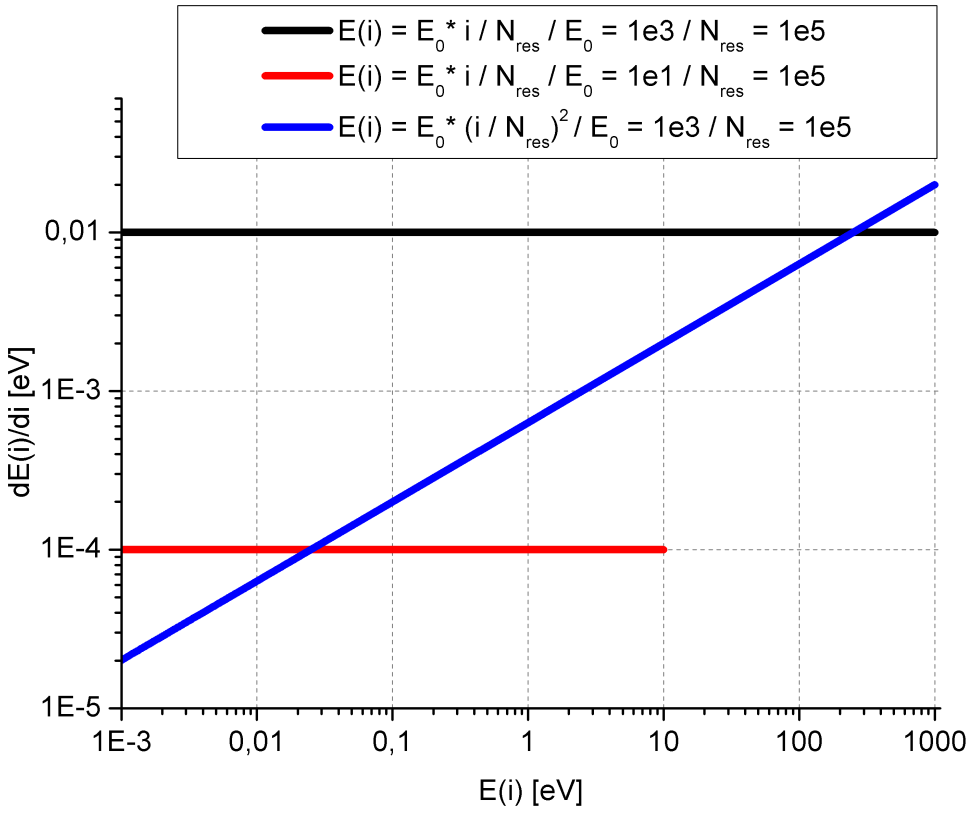


Figure 1: First order sampling for dsmc and pic-mc simulation (black and red curve) and second order sampling (blue curve) to combine dsmc and pic-mc simulation.

Cross section database

Database of the PIC-MC framework

- Electron-Neutral Cross Sections
  - Swarm analysis
  - e + H2
  - e + O2
  - e + N2
  - e + Xe

    - e + Kr
    - e + Ar
    - e + SiH4
- Ion-Neutral Cross Sections
    - Arplus + Ar
- Neutral-Neutral Cross Sections
    - Variable soft sphere model

Footnotes

1. A. Okhrimovskyy: Electron anisotropic scattering in gases: A formula for Monte Carlo simulations, Phys. Rev. E Vol. 65 Nbr. 3, 2002 ↩

16 Species list

In the following all species available for gas flow and plasma simulation are listet depending on their occurance on the periodic table. Keep in mind, that not all cross sections for all species combinations are already available.

Electrons

- e

Period 1

- H, Hplus, Hminus
- H2, H2plus
- H3, H3plus
- He, Heplus

Period 2

- N2, N2plus
- O, Oplus, Ominus
- O2, O2ex, O2plus, O2minus
- Ne, Neplus

Period 3

- Na, Naplus, Naplusplus
- Mg, Mgplus, Mgplusplus
- Al, Alplus, Alplusplus
- Si, Siplus, Siplusplus
- P, Pplus, Pplusplus
- S, Splus, Splusplus
- Cl, Clplus, Clplusplus
- Ar, Arplus

Period 4

- K, Kplus, Kplusplus
- Ca, Caplus, Caplusplus
- Sc, Scplus, Scplusplus
- Ti, Tiplus, Tiplusplus
- V, Vplus, Vplusplus
- Cr, Crplus, Crplusplus
- Mn, Mnplus, Mnplusplus
- Fe, Feplus, Feplusplus
- Co, Coplus, Coplusplus
- Ni, Niplus, Niplusplus
- Cu, Cuplus, Cuplusplus
- Zn, Znplus, Znplusplus
- Ga, Gaplus, Gaplusplus

- Ge, Geplus, Geplusplus
- As, Asplus, Asplusplus
- Se, Seplus, Seplusplus
- Kr, Krplus

Period 5

- Rb
- Sr
- Y
- Zr
- Nb
- Mo
- Tc
- Ru
- Rh
- Pd
- Ag
- Cd
- In
- Sn
- Sb
- Te
- Xe, Xeplus

Period 6

- Ta
- W

Organic compounds

- NH3, NH3ex
- SiH2, SiH3, SiH4, SiH3plus, SiH3minus
- Si2H4plus, Si2H5minus, Si2H5, Si2H6
- Si3H8
- Si4H10
- HMDSO
- Alq3

Anorganic compounds

- ZnO

Including other species

If you like to include other species than listed above, you have to include your own species_definition.r file in your parameter file. To do so, uncomment the following lines from your .par file:

```
SPECIESFILE = "my_species_definition.r";
```

As a basis you can take the original file from your picmc installation at "/your/picmc/path/scr/species_definition.r" and adapt it to your needs. The new file should be placed in the same folder as your parameter file.

17 Energy partitioning

For the electron impact ionization one must decide how to split the remaining collision energy between incident and secondary electron. The most sophisticated numerical expression comes from Opal[1] et. al. who did some substantial measurements on this topic. However, using that expression in our PIC-MC code has seemingly no effect at all on any output results. Hence we checked systematically the two extreme case where the secondary electron is uneffected and the remaining collision energy is divided equally between incident and secondary electron, respectively.

Comparision of different partitioning schemes

One takes it all vs. equal share

As seen in the simulation results below there is no significant difference between the two extremes. Power dissipation, reaction rates and plasma density are pretty much the same whether we use "one takes it all" or the equal share. Thus, we decided to take the less computational expensive one which is the "one takes it all" scheme. Furthermore this is the one which has been proven sufficiently accurate accordant to the comprehensive studies of Yoshida[2] et. al.

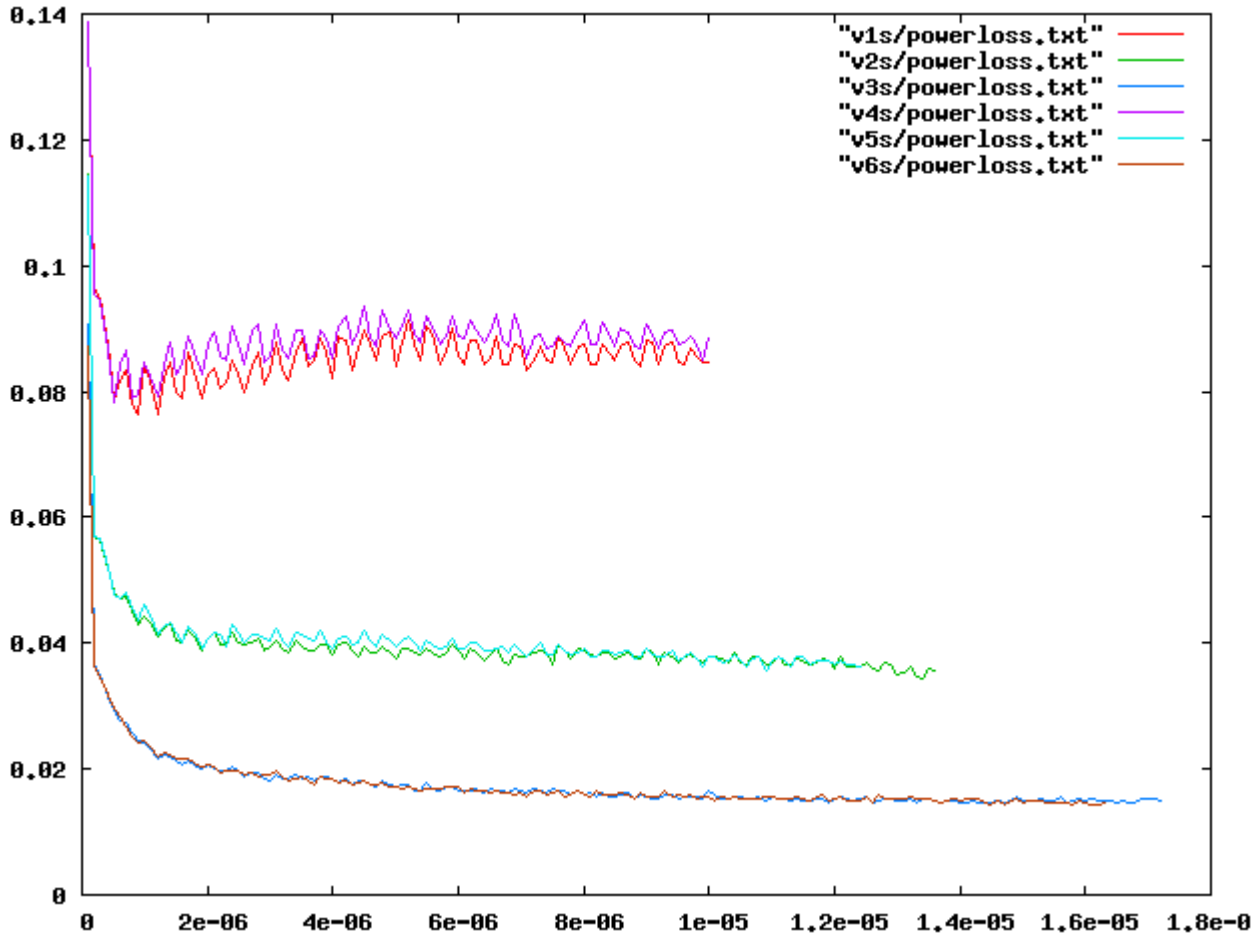


Figure 1: Power dissipation for rf simulations in argon at different E/N conditions using the "one takes it all" and equal share scheme, respectively.

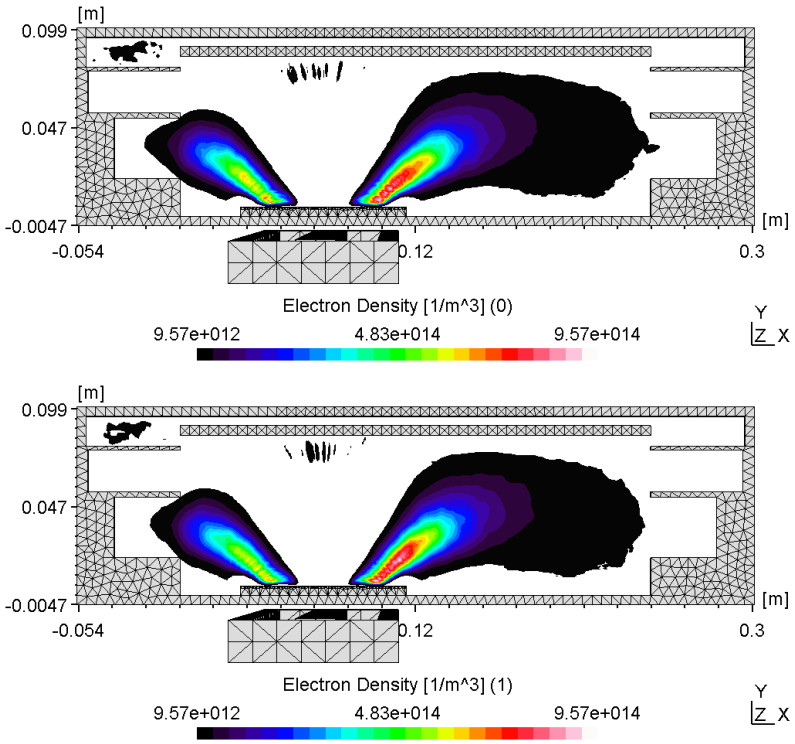


Figure 2: Electron density for magnetron discharge simulations at equal E/N conditions using the "one takes it all" and equal share scheme, respectively.

Footnotes

1. C.B. Opal: Measurements of Secondary-Electron Spectra Produced by Electron Impact Ionization of a Number of Simple Gases, J. of. Chem. Phys. Vol. 55 Nbr. 8, 1971 ↩

2. S. Yoshida, A. V. Phelps, L.C. Pitchford: Effects of electrons produced by ionization on calculated electron-energy distributions, Phys. Rev. Vol. 27 Nbr. 6, 1983 ↩

18 Averaged quantities

During DSMC / PIC-MC runs, averaged values are stored for each simulation cell. The average is taken over the ensemble of particles within the respective cell. If time averageing is enabled (which is usually the case), the ensemble averages are additionally time averaged. The following averaged quantities are continuously recorded during simulation:

Quantity	Formula			
Number	$N = \sum_{i \,\epsilon\, \text{cell}} 1$			
Density	$\langle n \rangle = \frac{N_\text{R}\, N}{V_\text{cell}}$			
Velocity component	$\langle v^{(\alpha)} \rangle = \frac{1}{N} \sum_{i \,\epsilon\, \text{cell}} v_i^{(\alpha)} \qquad \alpha = \{x, y, z\}$			
Square velocity	$\langle v^2 \rangle = \frac{1}{N} \sum_{i \,\epsilon\, \text{cell}} \vec{v}_i^{\,2}$			
Velocity vector	$\langle \vec{v} \rangle = \{\langle v_x \rangle, \langle v_y \rangle, \langle v_z \rangle\}$			
Velocity	$\langle v \rangle = \sqrt{\langle v_x \rangle^2 + \langle v_y \rangle^2 + \langle v_z \rangle^2}$			

The parameter N_R is the weighting factor of the respective species. From these averaged quantities, the following macroscopic results can be obtained:

Quantity	Formula		
Pressure	$p = \frac{1}{3} m \langle n \rangle \{\langle v^2 \rangle - \langle v \rangle^2\}$		
Temperature	$T = \frac{m}{3\, k_\text{B}} \{\langle v^2 \rangle - \langle v \rangle^2\}$		
Energy	$E = \frac{m}{2\, q} \langle v^2 \rangle$		

Here, m is the mass of one particle, k_B is the Boltzmann constant, and q is the elemental charge (in order to yield the averaged energy in [eV]).

19 Resampling (beta)

Resampling is beta. Parameter names, defaults and the numerical behaviour may still change. Before using results in production, compare the case against a run without resampling.

What resampling does

A simulation resolves a region with a fixed number of test particles per real particle, given by the statistical weight (see NREAL). Where the density is low, that wastes particles on nearly empty space; where the density is high, too many particles end up in one cell and the run slows down. Resampling adjusts the number of test particles **per cell** during the run: particles in sparse cells are split, particles in dense cells are merged. The total weight per cell is preserved, so density, current and fluxes stay the same - only the statistical resolution changes.

It is a numerical tool, not an additional physical model. Cross sections, transport and collisions are not affected.

Switching it on

```
RESAMPLE_METHOD = 4;      # 4 = balanced, the default
RESAMPLE_SCALE  = 150;    # target particles per cell at peak density
```

`RESAMPLE_SCALE` is the switch. If the line is missing - or commented out - resampling is off, even if `RESAMPLE_METHOD` is set to something else. Parameter files created with `initpicmc` contain both lines, so freshly created cases resample by default; existing cases keep running unchanged.

In a Python case the same switches are keyword arguments of `Sim`, for example `Sim(RESAMPLE_METHOD = 4, RESAMPLE_SCALE = 150, …)`; leaving `RESAMPLE_SCALE` out disables resampling there as well. See Driving a case from Python.

At startup the weights are harmonised once before the first time step; the log prints `Initial per-particle weight harmonization...`. While resampling is active it is `RESAMPLE_SCALE`, not `NREAL_<species>`, that sets the starting weights: they are derived from `P0_<species>` and the cell volume, and the log warns if `NREAL_<species>` was also given. `P0_<species>` is therefore required for every species in the species lists.

Warning: because the starting weights come from `P0`, a wrong initial pressure or density immediately leads to a wrong particle count per cell. Check the `number/` plots of the first few output times before interpreting anything else.

Profiles

Value	Name	Conserves	Cost
0	off	-	-
1	fast	number only	lowest
2	numberT	number and temperature	low
3	energyT	energy	moderate

Value	Name	Conserves	Cost
4	balanced	charge, momentum and energy in every event	moderate (default)
5	accurate	as 4, plus higher moments	highest

Start with 4. Profile 1 is noticeably cheaper but the temperature develops a visible bias; it is acceptable for a quick look at a density profile, not for energy-sensitive work such as sputter yields or dissociation rates.

Tuning

Key	Default	Meaning
`RESAMPLE_SCALE`	-	target particles per cell at peak density; missing or commented out means off
`RESAMPLE_INTERVAL`	100	time steps between full harmonisations
`RESAMPLE_DENSITY_SCALING`	0	0: same target everywhere; 1: target follows the density, fewer particles in thin regions
`RESAMPLE_DENSITY_THRESHOLD`	1e12	below this density in [1/m³] nothing is resampled
`RESAMPLE_NP_MIN`	4	cells whose average particle number is below this are left alone
`RESAMPLE_GOAL_MIN`	20	lower bound of the target in sparse cells
`RESAMPLE_LOCAL_ELECTRONS`	8	step interval of the cheaper per-species pass, electrons
`RESAMPLE_LOCAL_IONS`	100	same, ions
`RESAMPLE_LOCAL_NEUTRALS`	1000	same, neutrals

`RESAMPLE_SCALE` is read as an integer, so give it as a whole number.

Practical loop: set `RESAMPLE_SCALE`, leave the remaining keys at their defaults, run a few hundred cycles and look at `number/` and `number_goal/`. The particle count should settle in a band around the target. If the band is too wide, reduce `RESAMPLE_INTERVAL`; if the run became slower instead of faster, the target is too high for the available memory.

Cells are only touched when they leave a dead band around the target: split below roughly 0.67 times the target, merged above roughly 1.5 times. Small fluctuations therefore cost nothing.

Warning: `RESAMPLE_GOAL_MIN` must stay above the number of constraints of the selected profile, otherwise dense cells are never merged. The build refuses a configuration that violates this.

Cut cells are handled separately: their target ramps down below one percent cell volume fraction, so thin slivers along curved walls do not attract particles.

What to check afterwards

- Density, current and absorption must agree with a run without resampling within the statistical scatter.
- The total kinetic energy should be conserved. A growing or sinking trend means the target is far too small.

- The particle count per cell should settle, not oscillate. Oscillation means `RESAMPLE_INTERVAL` is too long compared to how fast the density changes.
- The electron energy distribution should keep its tail - compare histograms with a non-resampled run.

Limitations

- Weights differ within a cell after resampling. Anything that averages over single particles must weight by the particle weight instead of counting particles equally.
- Checkpoints contain the weights, so a run can be continued. After a restart the harmonisation runs once, which re-establishes the target particle count immediately; the density stays the same.
- Changing `RESAMPLE_SCALE` between two runs of the same case changes the statistical resolution, not the physics - but it does change the noise level of all plots.
- Beta: not yet validated for every gas mixture. Report suspicious behaviour.

Parameter file

20 Parameter file review

This site explains step by step all parts of the parameter file.

> **Python interface.** The blocks on this page are RVM syntax, and the parameter file is an RVM script. A case can also be described entirely in Python; the keyword names differ from the keys below. See Driving a case from Python for the mapping.

General parameter

```
DT       = 1e-5;
TSIM     = 1.0;
```

`DT` is the time step in [s] of the simulation. For DSMC its usually in the order of 10^{-6}-10^{-5} and all particles are moved in each time step. In PICMC mode its in the order of 10^{-12}-10^{-10} and only electrons are moved in each time step. The ions and neutral atoms are moved appr. each 30^{th} and 300^{th} step depending on the mass ratio related to the electrons. The value should be chosen in a way that the mean travelling distance of particles is less than their mean free path.

Example: If we assume Argon (Ar) as gas species and a mean total pressure of about 0.5 Pa, then the mean free path of the gas atoms will be about 1.3 cm (see mean free path). At 300 K, the mean velocity of an Ar atom is 398 m/s. Within a time step of 10^{-5} s this yields a mean travelling distance per time step of about 4 mm.

`TSIM` gives the total simulation time in [s] and is in the order of 1 for DSMC and 10^{-5} for PICMC.

```
NPLOTS   = 10;
NDUMPS   = 1;
```

`NPLOTS` gives the total number of plot evenly distributed over the total simulation time. If you make a restart of a simulation the number of plots made before is not taken into account.

The value `NDUMPS` is the number of particle dumps / checkpoints made during the simulation also evenly distributed over the total simulation time. If you have a checkpoint you can continue / restart the simulation from that point on. A new checkpoint overwrites the old one. A value of 1 means that the checkpoint is created when the simulation reaches its end. It is recommended to increase this value if the simulation is very time-consuming and you would lose several days if your simulation unexpectedly crashes.

```
WORKMODE = 0;
PSOLL    = 1;
PSAMPLE  = 1e-7;
```

`WORKMODE` accepts three different values, 0 if you want to simulate the gasflow of neutral particles with the DSMC method, 1 if a PICMC plasma simulation is favored or 2 if you are only interested in the electric field of a given setup. In case of a PICMC simulation the next two parameters are likewise evaluated. `PSOLL` is the aimed power dissipation in [W] and `PSAMPLE` is the sampling time for this power setpoint in

[s]. How the regulator works, which versions exist and which keys tune it is described in power regulation.

```
T0       = 300;
NCOUNTS  = 100;
VPMAX    = 1000;
```

The `T0` parameter defines the initial particle and border temperature in [K]. You could also use a non-trivial temperature distribution on a wall. `NCOUNTS` is the number of sampling intervals for particle counters. The value of `VPMAX` is the maximum voltage which is used in the voltage function.

Species

```
SPECIES      = ["Ar", "Arplus", "e"];
NREAL_Ar     = 1e12;
P0_Ar        = 0.5;
NREAL_Arplus = 1e5;
P0_Arplus    = 1e14;
NREAL_e      = 1e5;
P0_e         = 1e14;
```

The selection of gas species is defined within an array `SPECIES`. If multiple species are required, they can be written in a comma separated list such as above. A list of all available species can be found here. For each species `SP`, the parameters `NREAL_SP` and `P0_SP` must be defined with the following meaning:

- `NREAL_SP`: Statistical weight of the species `SP`. A value of e.g. 1e12 means, that one particle in the simulation corresponds to 1e12 particles in the physical reality. The collision statistics are handled according to this weighting factor. It is possible to use different weighting factors for different species; e.g. if we have 1 % oxygen diluted in Argon, we may use `NREAL_Ar = 1e12; NREAL_O2 = 1e10;` in order to get a similar amount of simulation particles for both species.

  The parameter is **optional**: if it is omitted, a weight of 10 is used. The weight may differ between individual particles during a run, see Resampling. If `RESAMPLE_SCALE` is set, the value is overridden - the starting weights are then computed from `P0_SP`, the cell volume and the target particle count, and a warning is written to the log.

- `P0_SP`: If `SP` is a **neutral** species, this defines the initial **pressure** in Pa. If `SP` is a **charged** species (electron / ion), the initial density in [1/m^3] is expected here instead.

In order to define proper values for these parameters, some statistical constraints of your simulation case have to be taken into account first. Please refer to the section statistical constraints for general information.

Frozen species

On demand, some of the species contained in the `SPECIES` array can be treated as *frozen*. In this case, no particles are stored; instead the species are treated as continuous background. Species properties considered are then averaged values of density, velocity components $\langle v_x \rangle$, $\langle v_y \rangle$, $\langle v_z \rangle$ as well as the square velocity $\langle v^2 \rangle$. The averaged values are stored cell-wise. In case of initialization of a frozen species via the `P0_species` parameter in the parameter file, only the averaged density and the square velocity will be non-zero. An alternative is to load the species from a checkpoint of another simulation. In this case, also the flow velocity stored in $\langle v_x \rangle$, $\langle v_y \rangle$, $\langle v_z \rangle$ can be relevant.

Example: In a PIC-MC plasma simulation in Argon, the Argon background should be treated as continuous, homogeneous background. In this case, the species definition section in the parameter file could look as follows.

```
SPECIES = ["Ar", "Arplus", "e"];

NREAL_Ar     = 6e+9;
P0_Ar        = 0.5;
NREAL_e      = 20000;
P0_e         = 1e13;
NREAL_Arplus = 20000;
P0_Arplus    = 1e13;

FROZEN_SPECIES = ["Ar"];
```

Note: If charged species (electrons, ions) are present in a DSMC simulation (`WORKMODE=0`), they are automatically being treated as *frozen*.

Plots

```
NUMBER      = ["Ar", "Arplus", "e"];
DENSITY     = ["Arplus", "e"];
PRESSURE    = [];
TEMPERATURE = [];
ENERGY      = ["Arplus", "e"];
VELOCITY    = [];
COLLISION   = [];
ABSORPTION  = ["Arplus", "e"];
DESORPTION  = [];
MESHDATA    = [];

POSITION    = [];

FIELD       = ["PHI"];

COVERAGE    = [];
DEPOSITION  = [];
```

Here, we have some predefined lists for each output quantity, where the species of interest can be inserted. Please refer to the section Available output data for further details.

Depending on your choice, the respectively quantity will be dumped into a 3D-outputfile for each plot interval (defined by `TSIM` and `NPLOTS`). The 3D-outputfiles have the ending `.pos` and can be displayed with GMSH. They can be further postprocessed via the RIG-VM postprocessing module or the GMSH postprocessing module.

Histograms

The histograms declared in the border reactions (see sampling of particles crossing a surface) are output of their own. They are not listed in the plot section:

- **RVM** writes every histogram of the case in each plot cycle, without any key in the parameter file: `histogram_<n>_<surface>_<species>_<time>.txt` for the angular and the energy histograms and `histogram_array_<n>_<surface>_<species>_<time>.txt` for the ones with a spatial grid, both in the `histogram` folder (or in `plots`, depending on `SINGLE_PLOT_FOLDER`).

- **Python** treats them as plots like any other quantity, and only what is listed is written: `"histogram_generic"` for the angular and energy histograms, `"histogram_array"` for the ones with a spatial grid and `"histogram_diag"` for the diagnostic histogram, each with the species of interest in the value list. A histogram that is defined on a border but missing here produces no file.

The columns of these files are described in sampling of particles crossing a surface.

Simulation volume

```
UNIT = 1.0;
BB   = [x0, y0, z0, x1, y1, z1];
```

The standard treatment of the simulation volume is to put the whole geometric mesh model within a rectangular box, which is referred to as *Bounding box*. The bounding box dimensions are defined within the vector `BB`, where, (x_0, y_0, z_0) and (x_1, y_1, z_1) specify two opposing edges with lower and upper coordinates. **Important:** Before specifying any coordinates, first make sure that the `UNIT` variable correctly represents the scale of your geometric model:

Mesh unit	UNIT variable
m	1.0
mm	0.001
inch	0.0254

When the parameter file is created via `initpicmc meshfile` the upper and lower edge coordinates are automatically set according to the maximum / minimum node coordinates occurring in the mesh file. However it is principally possible to set different bounding box dimensions e.g. in order to perform the simulation in only a fraction of the geometric model.

After you have set the bounding box dimensions, the simulation volume can be divided into multiple segments via the vectors `NX, NY, NZ`, and the number of cells for each segment can be specified. In the 2D plasma example the size of the bounding box is 360 mm in X, 105 mm in Y and 1 mm in Z direction. Assumed the cell dimension should be 1x1x1 mm^3, we would need 360x105x1 cells. A possible segmentation would be:

```
NX = [60, 60, 60, 60, 60, 60];
NY = [35, 35, 35];
NZ = [1];
```

This gives six segments in X direction with 60 cells each, i.e. 360 cells in total, while the total number of cells in Y direction is 3x35 = 105. As it is a 2D model within the XY plane, there is no variation in Z direction expected; therefore the number of cells in Z is 1. The above shown segmentation gives a total of 18 segments, i.e. up to 18 CPUs can be used for parallel execution of the job.

Without further parameters the segmentation is always equi-distant. In order to provide a segmentation with customized distances, the vectors `DIV_X, DIV_Y, DIV_Z` can be used.

```
DIV_X = [lx1, lx2];
DIV_Y = [ly1, ly2];
DIV_Z = [lz1, lz2];
```

The length values of the segments in the respective direction have to be in the scale given by the `UNIT` parameter above. The number of entries in `DIV_X` and `NX` etc. must fit and the sum of the segment lengths (lx1+lx2+...+lxn) has to be equal to the corresponding bounding box length.

```
TYPE_x1 = "outlet";
TYPE_x2 = "outlet";
TYPE_y1 = "outlet";
TYPE_y2 = "outlet";
TYPE_z1 = "outlet";
TYPE_z2 = "outlet";
```

Finally, it must be defined what happens if a particle crosses the outer boundary of the bounding box. The bounding box has six sides labeled `x1, x2, y1, y2, z1, z2` (i.e. lower x side, upper x side etc.). For each side, the type can be one of the following:

- `"outlet"`: Particles hitting this surface disappear immediately.
- `"specular"`: Particles are specularly reflected.
- `"periodic"`: Particles crossing this border are re-inserted at the opposite surface.

Optimized domain decomposition for DSMC

```
block MQUAD: {
  block FirstBox: {
    DIV_X = [50, 50];
    DIV_Y = [100];
    DIV_Z = [20];
    NX = [5, 5];
    NY = [10];
    NZ = [2];
    X0 = 0; Y0 = 0; Z0 = 0;
  };
  block SecondBox: {
    DIV_X = [50, 50];
    DIV_Y = [100];
    DIV_Z = [20];
    NX = [5, 5];
    NY = [10];
    NZ = [2];
    NREAL_SCALE_X = [0.5, 1.0];
  };
};

block MJOIN: {
  FirstBox.join_rect(SecondBox, "x", 0, 0, 100, 20,       0, 0);
};
```

For some geometric models a common rectangular simulation box is not efficient since large parts of the simulation volume might be unused. In such case it is possible to use the *Multiquad* approach, where multiple bounding boxes with individual segmentation and cell spacing can be defined for different parts of the simulation model. The usage of this feature (which is not applicable for plasma simulations) is described in detail in this gasflow simulation example.

Definition of input files

```
MESHFILE = "a700v.msh";
BEMMESH  = "pk750.msh";
```

The geometric information is obtained from mesh files generated by GMSH. The filename is specified in `MESHFILE`. If the parameter file has been created via the `initpicmc` command, the according mesh filename is already automatically inserted. However it is not required that the parameter file and mesh file have the same name, it is e.g. possible to make multiple copies of the parameter file using same mesh file but with a variation of other parameters (e.g. total pressure, gas species composition, power etc.). Also, the mesh file may be replaced by another one provided that the overall geometric dimension and the number and labelling of physical surfaces is the same.

In Python the mesh file is the `mesh_file` argument of `PICMC.Geometry`, together with `BB`, `NX`, `NY` and `NZ`; the remaining arguments of the constructor are optional.

For plasma simulation with magnetic field, it is additionally possible to obtain the magnetic field from a fix-magnetic assembly. In that case, an additional meshfile for the magnetic setup is required in the `BEMMESH` variable (see this example for 2D magnetron sputtering for further details), whose name has to be inserted manually. In Python `BEMMESH` has no counterpart: the field solver is a script of its own and gets its own `PICMC.Geometry` with the magnet mesh, see Computing a magnetic field with the BEM solver.

Border definition

```
block BORDER_TYPES: {
  Border Wall = {
    icodec      =   1;
    type        = "wall";
    T           = PAR.T0;
    #epsr        = 1.0;
    #vf          = "VP*cos(2*PI*13.56e6*TIME)";
    #vf_coupling = "";
    reactions: {
      add_plain_reaction("Arplus", 1, 1, ["Ar", "e"], [1, 0.1]);
      set_emission_energy("e", 0, 5, 1);
      add_absorption("e", 1.0);
      add_energy_histogram("Arplus", 0, 250, 1000);
    };
  };
};
```

In this section, the physical functions of the different *physical surfaces* of the geometric mesh model are specified. After creation of the parameter file with `initpicmc` an empty surface template entry is automatically created for each *physical surface* found in the mesh file. There is a large choice of physical parameters and mechanisms that can be assigned to each surface, please refer to chapter about border types for a detailed guide.

Additional switches

```
NCOUNTS            = 100;
TAVG               =   1;
SINGLE_PLOT_FOLDER =   1;
```

```
ICP_SOURCE         =   1;
```

The number of sampling intervals for the particle counters / averaged surface data can be set in the `NCOUNTS` parameter, more information can be found here.

`TAVG` is the time averaging mode which is activated by default (value of 1). It could also be switched off (value of 0), which means that a plot represents the simulation state at the exact time step of its generation rather than the averaged values since to the last plot. We recommend to not switch it off, because otherwise the plot will be quite noisy.

With `SINGLE_PLOT_FOLDER` set to 1, all post processing files are put directly into the simulation folder instead of sub directories.

`ICP_SOURCE` should be uncommented (and set to 1) if BEM solver is used only to compute an ICP source field. This switch improves solver performance as unnecessary computations are skipped.

Special model parameters

```
ELECTRON_ENERGY_PARTITIONING =     1;
MATERIAL_ACCELERATION        =     1;
NOISE_FILTER_COEFFICIENT     =     0;
ICP_SAMPLING_INTERVAL        =     1;
FIELDSOLVER_THRESHOLD        = 1e-10;
```

The `ELECTRON_ENERGY_PARTITIONING` switch defines how electron energy should be distributed in ionization collisions. With the default behaviour *one-takes-it-all* (= 1) incident electrons keep the remaining energy, i.e. minus the ionization threshold. With *equal-share* (= 0) the remaining energy is equally divided between incident and secondary electron.

`MATERIAL_ACCELERATION` is a factor by which coverage processes can be speed up. This is useful for simulations where surface reactions' convergence is lagging behind that of gas and plasma dynamics (e.g., surface oxidation vs. O_2 partial pressure convergence). The default value of 1 means no acceleration.

`NOISE_FILTER_COEFFICIENT` is still experimental. It enables a dampening algorithm to suppress artificial heating. The goal is to violate plasma frequency related time step constraints by a factor without consequences. The default value (= 0) disables the noise filter.

In ICP simulations setting `ICP_SAMPLING_INTERVAL` to 0 neglects the electron current induced field portion. For ICP simulations at low power and plasma density < 10^{16} m^{-3} this is a valid approximation to speed up performance. A sampling interval of 1 should do in all other cases.

The `FIELDSOLVER_THRESHOLD` parameter sets the accuracy of electric fieldsolver. The default value of 10^{-10} should be sufficient for most of the simulation cases. If you really know what you are doing you can change this value to loosen or tighten the convergence criterion.

Magnetic and electric field

```
BFIELDFILE = "external_field.txt";
```

Path to the external magnetic field file, see External magnetic field support for more information. In Python the file is read with a method of the simulation object, and the second argument selects the

format: `"external_unstructured"` for the file described here, `"external_structured"` for a rastered box and `"checkpoint"` for the binary field the BEM solver writes.

```
ELECTRODES = ["T1", "T2"];
P_SETPOINT = [0.5, 0.6];
I_SETPOINT = [0.005, 0.006];
```

If you have more than one controlled electrode you have to comment in the `ELECTRODES` list. The strings in this list correspond to the names of the borders / geometric boundaries from the `BORDER_TYPES` block. Then you can set individual power (`P_SETPOINT`) or current setpoints (`I_SETPOINT`) for these electrodes. As a result you will have separate controlled voltages, which can be referred to as `VP[0]`, `VP[1]`, etc. in the voltage functions of the borders.

Particle initialization

```
block Init_Particles: {
  S.distribution_xyz("Ar", 0.125, 0.0475, 0, 1, PAR.T0);
};
```

For certain setups you might want to place single particles in the simulation volume. This can be done within the `Init_Particles` block. The syntax of the `distribution_xyz` function is the following:

- Species name
- x, y, and z coordinate
- Number of particles
- Temperature of the particle

Moving meshes

```
NMOVES = 2;
block Move_Mesh: {
  float angle   = 5;
  float sinus   = sin(angle * PI / 180);
  float cosinus = cos(angle * PI / 180);
  # example for rotation of codec 1 around z axis with center point (100, 100, 0) without
translation
  S.move_mesh(1, cosinus, sinus, 0, -sinus, cosinus, 0, 0, 0, 1, 100, 100, 0, 0, 0, 0);
};
```

In DSMC simulations you can move parts of the mesh, normally the substrate. The movements of codecs with particle sources as well as a combination with a restart are not yet possible. The number of moves is set with the `NMOVES` parameter. To preserve the plot integrity you should move the mesh all X plots, while X must divide NPLOTS without remainder. Multiple movements per plot are also possible, but the plots (except the meshdata) may look weird.

The syntax of the `move_mesh` function is the following:

- Number of the codec to move
- The 9 rotation matrix elements: m_{11}, m_{12}, m_{13}, m_{21}, m_{22}, m_{23}, m_{31}, m_{32}, m_{33}
- The location of the point to rotate around
- The translation vector elements t_1, t_2, t_3

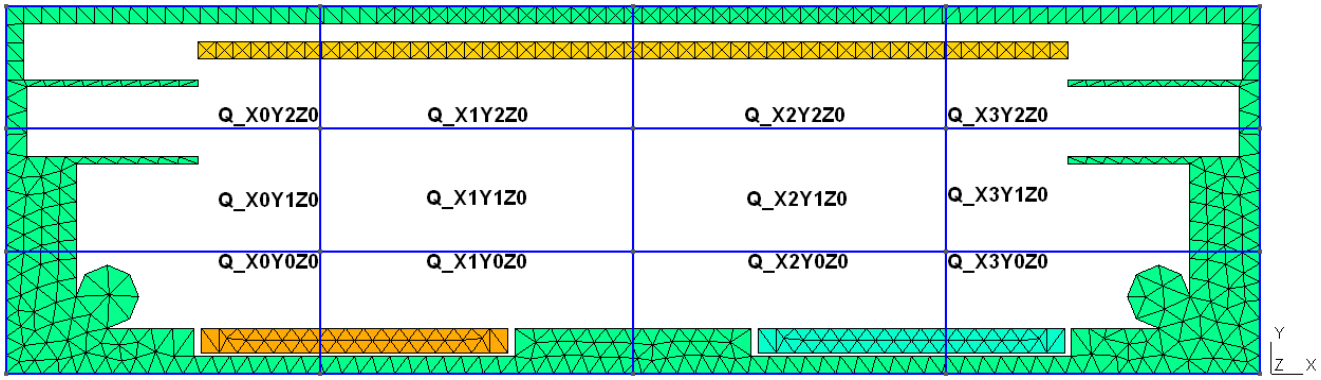
As you see in the above example you can define variables and calculate values inside the block. If you want to move multiple codecs, just use a second function inside the block.

Scaling factor adjustment

```
Q_X8Y1Z0_nreal_scale        = 10;   # single quad, all species
Q_X8Y1Z0_nreal_scale_Ar     = 5;    # additional factor, only for Ar
FirstBox_NREAL_SCALE_x0y11z5 = 10;  # same for a quad of a MQUAD box
```

The particle weighting factor can be adjusted in a single subvolume / quad. The factor is multiplied with `NREAL_SP` (and with the box-level scaling, if any), so `1` leaves the weight unchanged, `10` means ten times fewer test particles in that quad, and `0.5` means twice as many. Without a species suffix the factor applies to all species; a suffix such as `_Ar` adds a species-specific factor on top of it.

The quads have to be called explicitely. The quad numbering follows a simple rule: the quad with the name **Q_X0Y0Z0** is the one in the lower left back corner of the simulation volume. With increasing x, y and z coordinate of the quad the corresponding number in the quad name counts up. To better realize this, it is beneficial to merge the **domain_decomposition.geo** file with the actual mesh file. The following figure illustrates the numbering based on the 2D magnetron discharge example:



For a multiquad box the name is composed of the box name and the segment indices `x<ix>y<iy>z<iz>`, namely `NREAL_SCALE = 1;` for the whole box and `NREAL_SCALE_X = [0.5, 1.0];` (one entry per `DIV_X` segment) for single directions - see the gasflow example. All particles entering or leaving the quad are multiplicated/deleted depending on the weight quotient of the involved quads.

The resulting values are written to `boundingbox.rd` as `nreal = [...]` per quad, which is the quickest way to check a scaling.

Resampling (beta)

```
RESAMPLE_METHOD = 4;      # 0 off, 1 fast, 2 numberT, 3 energyT, 4 balanced, 5 accurate
RESAMPLE_SCALE  = 150;    # target particles per cell at peak density
# RESAMPLE_INTERVAL           = 100;
# RESAMPLE_DENSITY_SCALING    = 0;
# RESAMPLE_DENSITY_THRESHOLD  = 1.0e12;
# RESAMPLE_NP_MIN             = 4;
# RESAMPLE_GOAL_MIN           = 20;
# RESAMPLE_LOCAL_ELECTRONS    = 8;
# RESAMPLE_LOCAL_IONS         = 100;
# RESAMPLE_LOCAL_NEUTRALS     = 1000;
```

Resampling keeps the number of test particles per cell in a band during the run by splitting and merging particles. `RESAMPLE_SCALE` is the switch: if the line is missing or commented out, no resampling takes place. While it is active, the starting weights are derived from `P0_SP` instead of `NREAL_SP`. Full description, defaults and the recommended way to tune it: Resampling.

User actions and functions

```
block User_Block: {
};
```

The `User_Block` is executed once before the time cycles start and can be used for combining DMSC and PIC-MC simulations (*time scale splitting*) as shown in this example

```
block User_Functions: {
};
```

In the `User_Functions` block it is possible to include user defined functions for use during the simulation, more information can be found here.

Python needs no such block: the interface takes a callable wherever the parameter file takes a formula string.

- `PICMC.Border(vf = ...)` accepts a callable without arguments. It is called in every solve cycle, so it reads the time itself (`sim.get_time()`).
- `ParticleReaction(sigma = ...)` accepts a callable of the impact energy in eV instead of a file name.
- `ParticleReaction.runtime_modification` is a callable `(sim, reaction)` that is called during the run to change reaction values on the fly.
- `set_probability_function`, `set_temperature_function`, `set_yield_function` and `set_yield2D_function` take the function together with the grid on which it is sampled (`E0`, `E1`, `nE` or `T0`, `T1`, `nT`); the 2D yield is called with energy and angle. The values are sampled once at start-up, during the run the simulation interpolates the table - a fine grid costs memory, not time per collision.

Species and cross sections

```
SPECIESFILE = "my_species_definition.r";
SIGMAFILE   = "my_sigma.r";
```

If you want to use alternative or additional species not included by default, you have to specify the name of your own file in `SPECIESFILE`. Normally you then also have to specify your own `SIGMAFILE` which includes the cross sections of the new species. The list of all available species can be found here.

```
SIGMA_DEBUG  = 1;
SIGMA_ENERGY = 2000.0;
```

By activating `SIGMA_DEBUG` (setting the value to 1) one text file for each collision cross section in use is written to the simulation folder. This data can be beneficial if you want to show the curves in a presentation. The `SIGMA_ENERGY` parameter can be used to manually set the limit in eV up to which the cross sections are calculated. Since the default value is 1100.0 eV, you may get some warning messages if you have strong electric fields and particle energies are higher than the limit.

Debug

```
DEBUG       = 0;
NTESTCYCLES = 200;
```

The `DEBUG` mode is switched off (= 0) by default, but can be switched on (= 1) if your simulation does not start properly and you want to know exactly what's the problem. The output can be quite long, so consider redirecting it to a file.

The default number of test cycles for the load measurement ahead of every simulation shutdown is 200, which should be appropriate for the most cases. Changing this can be done by setting `NTESTCYCLES` to the desired value. Depending on these measurements a load balancing is carried out, if you restart a simulation and the number of quads is higher than the number of requested CPUs.

21 Parameter file tuning

User_Functions

In this block it is possible to include user defined functions for use during the simulation, e.g. a special voltage function. The syntax is the following:

```
type function_name(variable_name=type) = {
  return calculated_value;
};
```

The type of the function as well as the variables can be "integer" or "float", the names of both are arbitrary but have to start with a letter. The number of variables is not limited and the calculated value can include common mathematical functions. An example of such a user function is the modulo function:

```
float myModulo(dividend=float, divisor=float) = {
  integer quotient = dividend / divisor;
  return dividend - quotient * divisor;
};
```

The use of this user function in the voltage function of an electrode would be e.g.:

```
vf = "-VP*myModulo(TIME, PAR.PERIOD)+DC_BIAS";
```

Profiling data and load balancing

The file **profile.txt** contains information about the computational load of each quad and can be used for load balancing, i.e. the optimized distribution of the quads on the available CPUs. The load balancing is done automatically during restart, if the number of quads is higher than the number of requested CPUs. The load itself is measured at the end or during shutdown of each simulation and the number of timecycles used for this measurement can be set with the `NTESTCYCLES` value in the parameter file.

Additionally to the overall load used for the automatic load balancing, the individual loads for different stages during simulation are measured, this includes:

- the collision of particles,
- the movement of particles,
- the transfer of particles to neighbour quads and
- the calculation of the electric field.

This information can for example be used for fine-tuning of the quad arrangement. If the particle transfer takes a significant amount of time then reducing the overall number of quads might be beneficial. To get a better insight into the computational load not only by staring at the raw numbers, a small script **visualize_load** is provided. It is called from the directory where the parameter file is located and has the folder name of the simulation case as only parameter. The result is the file **load_visualization.pos** which can be viewed with GMSH.

Particles per cell

The statistical weight decides how many test particles represent the gas. For a region with density `n` and cell volume `V` the number of test particles is `n·V/NREAL`, so a target of about 20 particles per cell in the region of interest gives

```
NREAL = n · V / 20
```

`NREAL_<species>` is optional; without it a weight of 10 is used. That is a convenient value for a first test run, but far too fine for production. For argon at process pressure the value typically lies between 1e8 and 1e10, depending strongly on the cell size.

Where the density varies strongly across the mesh, keep the global `NREAL` moderate and scale single quads or boxes instead: `NREAL_SCALE = 10` within a box (or `NREAL_SCALE_X = [1, 1, 5]` per segment) means ten times fewer test particles there. The factors multiply the global value, `1` leaves it unchanged. See Scaling factor adjustment.

Warning: the weight changes the number of test particles, not the physics. Halving `NREAL` and halving `P0` is not the same thing - only `P0` changes the density.

Tuning resampling

With resampling active, `NREAL_<species>` is no longer the quantity to tune: the starting weights are computed from `P0` and `RESAMPLE_SCALE`. Start with the template values (`RESAMPLE_METHOD = 4`, `RESAMPLE_SCALE = 150`), leave the remaining keys commented out and run a few hundred cycles.

- If the particle count per cell oscillates around the target instead of settling, reduce `RESAMPLE_INTERVAL`.
- If the run becomes slower or needs more memory than without resampling, `RESAMPLE_SCALE` is too high for the available hardware.
- If thin regions such as the sheath lose resolution, set `RESAMPLE_DENSITY_SCALING = 1` so the target follows the density, or lower `RESAMPLE_GOAL_MIN` - but keep it above the number of constraints of the profile, otherwise dense cells are never merged.

The target field and the actual particle count are available in `number_goal/` and `number/`. Details: Resampling.

22 CAD import (.stp/.igs/.stl)

In order to use STEP or IGES files from external CAD programs (SolidEdge, Inventor, etc.) in GMSH, the following steps are required:

- Assumed the step file is named `model.stp`, as a first step, an empty *.GEO file (e. g. `model.geo`) has to be created in the same directory. Within the GEO file, one single command for opening the step file has to be typed: `Merge "model.stp";`
- The file `model.geo` can now be opened in GMSH. In order to have a proper mesh size for all points of the geometry, go into the "Mesh" menu and select *Mesh-->Define-->Element size at points*. Type a proper value (e. g. 20 mm) into the number field and press return. Afterwards, you should select every point of the model with the mouse[1] and finally press "e".
- In your `model.geo` file, GMSH has added a line like `Characteristic Length{<point numbers>} = 20;`
- Invoke the meshing algorithm in GMSH by pressing the "2" key and have a close look to the meshed geometry. If there are features, which are not well resolved by the mesh (e. g. small nozzles) it may be required to define a smaller element size at certain points.

Footnotes

1. this is easily obtained by pressing "Ctrl" before the mouse selection, which enables to use a selection frame ↩

23 External magnetic field

There is the possibility (under certain conditions) to load an external magnetic field, instead of calculating it with the included BEM magnetic field solver. There are the following two possibilities:

- Magnetic field with few datapoints and unstructured arrangement
- Magnetic field with many datapoints and structured arrangement in a box

Magnetic field support with unstructured data points

To activate this feature, the following line of the parameter file has to be uncommented:

```
BFIELDFILE = "external_field.txt";
```

The field solver of the same release writes files of this kind: `write_textfield(filename)` gives the computed field as `x y z Bx By Bz`, which is how a field that was computed on one grid becomes usable in a case with a different resolution.

There are two file formats for the unstructured magnetic field supported, text and binary format. The format is determined during runtime, independent of the file extension. The conditions for the **text format** are the following:

- an entry consists of 6 values, three values for the x-y-z-position and three values for the Bx-By-Bz-strength at that point
- only one entry per line
- the separators for the values are either tabulators or spaces

and the conditions for the **binary format** are:

- an entry consists of 6 values, three values for the x-y-z-position and three values for the Bx-By-Bz-strength at that point
- no separators between the values or the entries
- the values have to be double precision floating point numbers

Additionally the following conditions are valid for **both formats**:

- the position values of the external field have to be in the same coordinate system as the mesh in [m]
- the values for the strength of the external field have to be given in tesla [T]
- the volume covered by the external field has to be greater or equal to the simulation volume

The resolution of the external field compared to the simulation resolution does not matter because the magnetic field is interpolated, but not extrapolated (see point above).

Important notes:

- The coordinates have to be given in meter [m] irrespective of the used mesh unit given in the parameter file for the simulation volume.
- The order and structure of the coordinate points can be completely arbitrary, however the interpolation routine scales with the product of coordinate points in the simulation volume and data points in the TXT file. Thus, for large B field datasets the time for reading and interpolating the B field data can become unreasonable.

Magnetic field support with structured data points arranged in a box

The second method of importing a magnetic field is available since version 2.4.3.5 and was implemented for large datasets, where the point coordinates are aligned in a rectangular box. This simplifies the interpolation of the magnetic field data on the simulation grid points and thus, this method is much faster in case of very large B field files.

The import command of "rastered B field files" is implemented as a method of the PICMC object and can be activated via the user block in the parameter file. To use it, leave the parameter `#BFIELDFILE` commented out, uncomment the `User_Block:` at the end of the parameter file and use the following commands:

```
block User_Block: {
  bfieldfile = "large-bfield-file.txt";
  message = "Reading rastered external Bfield from $(bfieldfile)";
  console "** $(message) **\n";
  write_logentry(message);
  S.read_bfield_raster(bfieldfile, -300, -884, 14,   300, 884, 210,   151, 443, 50,   1e-4);
};
```

The syntax of the command is `S.read_bfield_raster(bfieldfile, x0, y0, z0, x1, y1, z1, nx, ny, nz, factor);`

The parameters have the following meaning:

- **bfieldfile** = Filename of the file containing the magnetic field data (path relative to the *.par file)
- **x0, y0, z0** = Lower coordinates of the rectangular box
- **x1, y1, z1** = Upper coordinates of the rectangular box
- **nx, ny, nz** = Number of data points in X, Y, and Z direction
- **factor** = Factor, by which the magnetic field vector shall be multplied (in case if the magnetic field in the TXT file is not given in Tesla units, the above example uses a factor of 10^{-4} for converting from Gauss to Tesla)

Important notes:

- The format of the TXT file is six columns with (X,Y,Z) coordinates in the first three columns and (Bx, By, Bz) field vector in the colums 4-6. The separator between the columns shall be one or more blank spaces.
- The ordering of the coordinate points in the TXT file does not matter. It is only required that all `nx*ny*nz` points are included exactly once, otherwise an error message will be displayed.
- Unlike the older B field import method mentioned above, all coordinates here are sensitive to the mesh unit specified in the simulation volume. This also applies for the coordinate points in the TXT file.
- The interpolation of the B field values onto the simulation grid is performed via trilinear interpolation. It is required that the simulation grid (as specified in the simulation volume) either has the same size as the magnetic field box or fits within.

Wall interaction model

24 Wall border type

The interaction between the gas phase and walls is designed with the aim to provide an interface for setting up arbitrary reaction chains with high flexibility. For meshed geometric boundaries, there is one common "wall" type, where reactions and particle sources can be attached. Additionally, it is possible to declare single-sided mesh surfaces as "membranes". Membranes fulfill special purposes such as virtual particle sources in the simulation space, particle flow barriers or virtual surfaces for sampling of energy and angular flux distributions.

A **wall** is represented in the GMSH mesh file as one physical surface group. In the PIC-MC / DSMC simulation parameter file a wall is represented as a **Border entry** as shown in the following example:

```
block BORDER_TYPES: {
  # ...

  Border Inlet = {
    icodec =   2;                       # Codec number from meshfile
    type = "wall";                      # Alternative type: "membrane"
    T = PAR.T0;                         # Wall temperature
    reactions: {
      add_source("O2", 1, "Pa");
    };
    # epsr = 1.0;                       # Rel. dielectric permittivity (insulator only)
    # vf = "VP*cos(2*PI*13.56e6*TIME)"; # Voltage function (conductor only)
    # vf_coupling = "";                 # Border name of counter electrode for potential coupling
  };

  # ...
};
```

In Python the border is a `Border` object with the same entries as keyword arguments. The objects are collected in the `BORDER_TYPES` dictionary of `Geometry`, keyed by a name of your choice: the surface is found through `icodec`, the key is only used to resolve `vf_coupling`. `vf` takes a number or a callable without arguments there, `reactions` a list of `BorderReaction` objects.

Each **Border entry** has the following common parameters:

- **icodec**: Integer number specifying the physical surface group. This number is automatically extracted from the mesh file and should be only changed in special cases [1].
- **type**: String constant which can be either `"wall"` or `"membrane"`. A `"wall"` must be part of a 3D object of the mesh file, while the `"membrane"` is a virtual, single-sided (i. e. infinitesimal) surface which can be used to block orifices, insert additional particle sources or for sampling purposes.
- **T**: Floating point number specifying the wall temperature in K.
- **reactions: { ... };**: Within this block, wall reactions and particle sources can be specified. A detailed description is given below.
- **epsr**: *Optional* floating point variable specifying the dielectric permittivity of a dielectric insulating object (e. g. a glass susbtrate) in case of a plasma simulation.
- **vf**: *Optional* string variable specifying a time-dependent voltage function.
- **vf_coupling**: *Optional* string variable specifying a counter electrode within systems of floating electrodes with built-in potential difference.

Footnotes

1. e.g. if a modified mesh file with different physical surface numbers shall be used with a given parameter file, this process is error-prone, you need to be careful ↩

25 Plain reactions

A **plain reaction** can be particle absorption or particle transformation with a given contant **sticking coefficient** or **reaction probability**. There is no wall state or surface coverage involved. Reactions which consider the actual surface coverage with different surface materials are described under surface chemistry reactions below. Possible **plain reactions** are listed in the following:

Simple absorption with sticking coefficient

```
add_absorption("species", sticking_coefficient);
```

This simply defines a constant absorption probability of a given **species** by a **sticking coefficient**.

Example:

```
# Define a pumping surface for Ar and H2
add_absorption("Ar", 0.2);
add_absorption("H2", 0.05);
```

Specular reflection

```
add_specular_reflection("species", probability);
```

This declares a specular reflection of a given **species** with a given **probability**.

Plain reaction with sticking coefficient and reaction products

```
add_plain_reaction("species", flag, probability, ["rp1", "rp2", ...], [n1, n2, ...]);
```

This more generalized syntax allows to specify particle absorption and reaction products. The parameters are

- **"species"** => Incoming gas species such as `"Ar"` or `"H2"`
- **flag** => Can be 0 or 1. A value of flag==1 means that the incoming species is absorbed, if flag==0 it is reflected at the surface.
- **probability** => A value between [0..1] specifying the reaction probability.
- **["rp1", "rp", ..]** => A comma-separated list of reaction products (use `[]`, if no reaction products are involved)
- **[n1, n2, ...]** => The amount of reaction products

In Python the same arguments are the keywords `species`, `absorption` (the `flag`), `probability`, `reaction_products` and `product_weights`, with `reaction_type = "plain"`.

Example 1: SiH_4 -> Si (film growth) + 2 H_2 (reaction products given back to the gas volume) with a sticking coefficient of 50%

```
add_plain_reaction("SiH4", 1, 0.5, ["H2"], [2]);
```

Example 2: Secondary electron generation with secondary electron coefficient $\gamma = 0.1$ by ion impact $Ar^+ \rightarrow Ar + \gamma e^-$

```
add_plain_reaction("Arplus", 1, 1, ["Ar", "e"], [1, 0.1]);
```

The emission characteristics of reaction products is by default Maxwellian with a $\cos(\theta)$ like angular distribution. In order to change this behaviour, the same commands for specifying the emission characteristics as in the particle sources can be used.

Example 3: Sputtering with Ar^+

```
add_plain_reaction("Arplus", 1, 1, ["Ar", "e", "Nb", "Ominus"], [1, 0.1, 5, 0.04]);
set_emission_energy("e", 0, 10, 100);     # Electrons are perpendicularly emitted with energies
between 0..10 eV
set_emission_sputter("Nb", 7.39, 1.5);    # Nb is sputtered with Ub=7.39 eV and alpha=1.5
set_emission_sputter("Ominus", 4.5, 1.5); # Creation of negative oxygen ions with a sputter
distribution
```

The methods for the emission characteristics return the reaction, so they can be chained. `set_emission_sputter` itself has no Python form: give the Thompson distribution the kinetic energy and the mass of the impinging ions explicitly, as above - those are the two arguments the parameter file derives from the impacting particle. Also missing in Python are the formula-based emitters (`set_emission_function_*`) and the energy loss functions (`set_energy_loss_*`).

26 Particle sources

Constant sources

Particle sources can either have Maxwellian or Sputter type energy emission, or a given emission energy interval can be specified. The declaration syntax is `add_source("species", amount, "unit");`

Example:

```
add_source("O2", 50, "sccm");
add_source("Al", 10, "Pa");
```

Both commands declare a particle source. The first argument is the **gas species** to be created, the second argument the **amount**. The **unit** argument can be either `"sccm"` or `"Pa"` specifying a mass flow or pressure controlled source, respectively.

Sources with inscribed profile

For the source surface it may be required to have a non-uniform profile over the source surface.

An example is magnetron sputtering where the sputtering emission distribution from the cathode depends on the Ar^+ ion bombardment profile. The Ar^+ ion bombardment profile can be stored in an absorption plot by putting `Arplus` into the `ABSORPTION` plot list.

In a subsequent DSMC neutral transport simulation, e.g. in order to evaluate the film thickness profile of the sputtered material, there is now an easy way to use the Ar^+ ion bombardment profile for defining the sputtering source:

```
add_source("Ti", 50, "sccm");                       # particle flux of the source
set_emission_thompson("Ti", 4.89, 1.0, 500, 40);    # Thompson distribution, 4.89 eV binding energy,
                                                    # 500 eV Ar+ bombardment
set_profile("Arplus-absorption.pos");                # erosion profile, taken from the Arplus
                                                     # absorption plot of the preceding PIC-MC run
```

`set_emission_thompson` is the recommended form: the binding energy, the cosine exponent and the energy and mass of the bombarding ion define the recoil cutoff of the distribution, so the energy spectrum follows the real bombardment. The older `set_emission_sputter("Ti", 4.89, 1.0)` is still accepted and applies the same distribution, but it assumes argon ions of 40 u and a cutoff of 100 times the binding energy. Both are described in particle emission.

Two things to watch with a profile source. The profile is only a **distribution**: it is normalized over the source points of the border, so the absolute flux comes from `add_source`, not from the numbers in the file. And `set_profile` and `set_emission_*` attach to the wall reaction that was declared last, so the source reaction has to be the last one in the reaction list of that border.

For the file `Arplus-absorption.pos` has to be copied from one of the Ar^+ absorption volume and mesh plots, respectively from PIC-MC plasma simulation. Alternatively, a time-averaged file can be made from several absorption plots via post processing. In the `set_profile` command, the filename has to be given as relative path with respect to the location of the parameter file (in the example above, `Arplus-absorption.pos` must be in the same directory as the parameter file).

A more accurate way is to model the sputtering process within the PIC-MC plasma simulation by taking into account the angular and energy dependency of the sputtering yield. In this case, the `desorption_XXX.pos` files or their mesh plot equivalents can be taken as profile for the subsequent DSMC simulation. However the assumption that the local sputtering rate is proportional to the ion flux is a good starting approximation in most cases.

Vapor pressure source depending on surface temperature

In some cases, a source surface might have a temperature profile. This is especially relevant in connection with imported wall temperature distributions. In case of a pressure source - e.g. for an evaporation surface - the vapor pressure is temperature dependent. Most temperature-dependent vapor-pressure curves can be fitted with a simple exponential expression:

$p(T) = exp(-a/T + b)$

The syntax of the temperature-dependent pressure source is:

```
add_vapor_source("species", a, b);
```

The parameters a and b are material dependent. For Aluminum, suitable values are `a=36070` and `b=24.469`. Thus, a temperature-dependent Al evaporation source can be defined by:

```
add_vapor_source("Al", 36070, 24.469);
```

In Python there is no counterpart: `BorderReaction` accepts `"Pa"` and `"sccm"` as source unit, a temperature dependent vapour pressure curve does not exist yet. A constant equivalent is `BorderReaction(species = "Al", reaction_type = "source", source_value = 1.0, source_unit = "Pa")`, which keeps the flux fixed instead of following the wall temperature.

27 Particle emission

Particles are emitted from walls for two possible reasons. Either a Particle source is attached to the wall or a reaction involving reaction products takes place.

In both cases, the **default behaviour** is to emit particles with a Maxwellian velocity according to the given wall temperature and a *cos(θ)* like angular distribution. A Maxwellian distribution in 3D space is characterized by following distribution function in the velocity v,

$$f(v)dv = \sqrt{\frac{2m^3}{\pi k^3 T^3}} v^2 \exp\left(-\frac{mv^2}{2kT}\right),$$

where T is the temperature, k the Boltzmann constant, and m the molecular mass. When sitting at a wall surface, the velocity distribution function has to be multplied with the particle velocity v_z normal to the surface, since the arrival of fast particles is statistically emphasized. In the software implementation, the particle velocities of diffusely reflected or generated particles with Maxwellian distribution are generated as follows in cylindric coordinates (r, ϕ, z) with the *z* coordinate aligned parallel to the surface normal:

$$v_z = \sqrt{-\frac{2kT}{m}\ln(R\#)}\, v_r \quad = \sqrt{-\frac{2kT}{m}\ln(R\#)}\, \phi = 2\pi R\# \quad \vec{v}_{\text{Emitted}} \quad = v_z\hat{n} + v_r\cos(\phi)\hat{t_1} + v_r\sin(\phi)\hat{t_2},$$

with *R#* being a random number between $[0, 1[$ and $\hat{n}, \hat{t_1}, \hat{t_2}$ being the surface normal and normalized tangential vectors. If this algorithm is performed for reflected particles in a closed gas volume, the gas will be everywhere in thermal equilibrium with the wall and will have a Maxwellian distribution according to the wall temperature.

It is possible to modify the behaviour of generated particles by supplying an additional command just after the last wall command involving reaction products (either after the last source or reaction).

Maxwellian distribution with alternative temperature

If required it is possible emitting particles with Maxwellian distribution with a different temperature than specified in the wall properties. For this purpose, there is a function `set_emission_maxwell` which takes the desired temperature as argument:

```
set_emission_maxwell("species", temperature);
```

Source of sputtered particles

A sputter process results from random ballistic collisions between atoms inside the target material. The distribution of sputtered species is non-thermal. A thorough discussion on that topic is given in the book "Sputtering by Particle Bombardment" by Behrisch/Eckstein [1]. The distribution function of sputtered particles has been discussed by Thompson[2] yielding following relation:

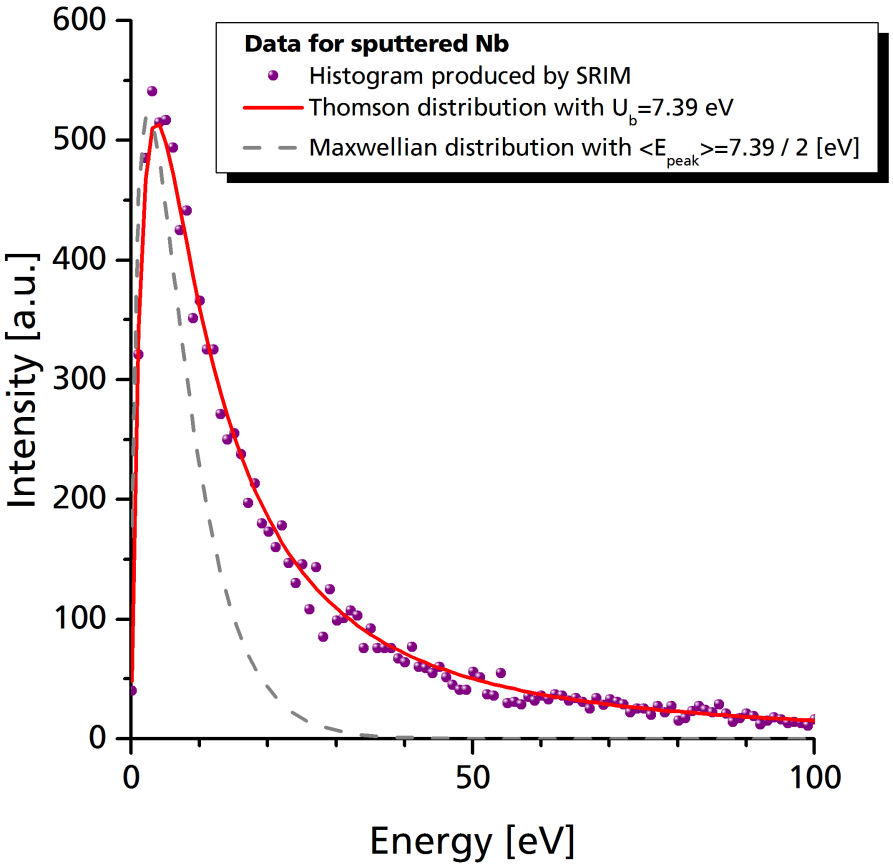
$$f(E,\theta) \propto \frac{\cos(\theta)}{E^{2-2m}\left(1 + U_b/E\right)^{3-2m}}.$$

Here, *m* is a parameter describing the interatomic potential between target atoms. Assumption of the hard-sphere approximation leads to *m*-->0. The software implementation uses this simplification within the function `set_emission_thompson` together with an angular distribution function $f_\alpha(\theta)$ depending on a parameter α,

$$f(E,\theta) \propto \frac{E}{(E+U_b)^3} \times f_\alpha(\theta)\ f_\alpha(\theta) \quad = \begin{cases} \cos^\alpha(\theta); & \alpha \geq 0 \\ \cos(\theta)\left(1+\alpha\cos^2(\theta)\right); & \alpha < 0 \end{cases}$$

The angular distribution for negative values of α is an empirical formula for off-axis sputtering characteristics taken from Yamamura, 1991[3]. It could be relevant for polycrystalline target materials and in case of low ion energies[4]. For performing ballistic simulation of the sputter process in targets, there is a free software called SRIM, which can produce sputter yields, velocity and angle distribution of sputtered particles as well as implantation profiles for arbitrary target compositions and ion impact. The software implementation includes both formula for the angular distribution function, which are selected according to the parameter α.

As can be seen in the following graph [5], the distribution of sputtered particles decays much slower with energy than e.g. a Maxwellian distribution with the same peak energy (The peak energy in case of a sputter distribution is located at $E_{\text{peak}} = U_b/2$).



One conceptual problem of the above distribution function by Thompson is that there is no upper limitation regarding the kinetic energy of sputtered species. Generation of particles with arbitrarily high kinetic energy is problematic since they potentially violate numerical constraints with respect to time step and cell spacing.

A practical solution is based on the maximum fraction λ of energy which can be transferred from an impinging ion with mass m_i to a target atom with mass m_t by perpendicular collision[6],

$$\lambda = \frac{4 m_i m_t}{(m_i + m_t)^2}.$$

With E_i as the energy of impinging ions it is reasonable to assume that the energy of sputtered atoms can be no higher than $\lambda \times E_i$. Thus, in the DSMC / PICMC software implementation, $\lambda \times E_i$ is used as upper limit of the kinetic energy of sputtered particles. The accordant function for the Thompson distribution is

```
set_emission_thompson("species", Ub, alpha, E_ion, m_ion);
```

This is the recommended form for sputtered particles: the cutoff follows the real bombardment, whereas the legacy variant described at the end of this section assumes it. The parameters are as follows:

- `"species"` --> the name of the sputtered species
- `Ub` --> The binding energy
- `alpha` --> The parameter α determining the angular distribution
- `E_ion` --> The kinetic energy of impinging ions
- `m_ion` --> The relative molecular mass of impinging ions

The latter two parameters are used to compute the above mentioned recoil factor λ. The simplified Thomson distribution can be analytically integrated, thus in the practical implementation the energy is taken via

$$E = \frac{U_b}{1 - \sqrt{R\#}} - U_b$$

Again, *R#* is a random number within[0,1[. The emission angle is created via

$$\theta = \begin{cases} \arccos\left((1 - R\#)^{\frac{1}{\alpha+1}}\right); & \alpha \geq 0 \\ \arccos\left(\sqrt{\frac{\sqrt{1+\alpha(2+\alpha)(1-R\#)}-1}{\alpha}}\right); & \alpha < 0 \end{cases}$$

If the random number yields $E > \lambda E_i$ then the value is dropped and the calculation is repeated for another random number *R#*.

Note: In former versions (<2.4.2.8) the Thompson distibution function was generated via the legacy command

```
set_emission_sputter("species", Ub, alpha);
```

Here, the cut-off energy was internally hard-wired as $E_{\text{cutoff}} = 100 \times U_b$. This function still works but we recommend using the newer `set_emission_thompson` instead - which is also the only form the Python interface offers, where the cut-off energy can be freely specified. Especially in case of sputtered species deposited onto a substrate it turned out that unrealistically high values of the cut-off energy can lead to significant deviations in their mean energy.

Uniform energy distribution within a given interval

For a uniform distribution within an energy interval between E_0..E_1, the following function can be used:

```
set_emission_energy("species", E0, E1, alpha);
```

The angular distribution as a function of the parameter α is the same as in case of sputtered particles (see previous section).

Example: Niobium sputter source

```
add_source("Nb", 50, "sccm");                    # constant rate, equivalent to 50 sccm particle
flux
set_emission_thompson("Nb", 7.39, 1.5, 500, 40);  # recommended: 7.39 eV binding energy, 500 eV Ar+
# set_emission_sputter("Nb", 7.39, 1.5);          # legacy: same distribution, cutoff 100 x Ub
```

Particle emission distribution by user defined histogram

If the built-in particle emission profiles are not sufficient, there is also the possibility of implementing a user-defined distribution function via histogram data files. The histogram data files must have the same format as the files resulting from sampling energy and angular distribution, i.e. four columns comprising

- Energy [eV]
- Theta [°]
- Phi [°]
- Scalar intensity

There are the following four options:

1. **Energy only distribution function** `set_emission_histogram_E("species", alpha, hist_filename);` Here, only the first and fourth column of the histogram file (energy and intensity) will be taken into account. For the angular distribution function, the parameter `alpha` has the same meaning as shown above.
2. **Angular only distribution function** `set_emission_histogram_A("species", temperature, hist_filename, ex, ey, ez);` Here, the first column of the histogram file is ignored; relevant are columns 2-4 (theta and phi angle as well as intensity). The angles are to be given in degree. Additionally, a unit vector `ex, ey, ez` is to be specified, this vector defines the direction, where the azimuth angle is zero.
3. **Separated energy and angular distribution functions** `set_emission_histogram_EA("species", hist_E_filename, hist_A_filename, ex, ey, ez);` Here, energy and angular distribution function are both to be specified and treated independently.
4. **Correlated energy and angular distribution function** `set_emission_histogram_Corr("species", hist_filename, ex, ey, ez);` This function uses only one histogram file, where energy and angular columns are both included, i.e. energy and angular distributions are correlated.

Particle emission distribution by user defined functions

Instead of providing emission distribution functions as histogram files, they can be also applied as analytic formula. There are the following options:

1. **Energy distribution function** `set_emission_function_E("species", alpha, eMax, ne, formula);` The energy distribution function for the emitted `"species"` is defined on an energy interval `0...eMax` with `ne` being the number of slots. The energy unit used to specify the range is eV. The parameter `alpha` specifies the angular emission profile as specified here. The analytic function is specified via parameter `formula`, which can contain an analytical formula with `E` being the energy in eV.
2. **Angular distribution function** `set_emission_function_A("species", T0, nTheta, nPhi, formula, ex, ey, ez);` This specifies an angular distribution function while assuming an

Maxwellian energy distribution function according to the temperature `T0`. The resolution of the angular distribution function is defined by the number of slots, `nTheta` for the polar angle and `nPhi` for the azimuth angle. The analytical function in parameter `formula` may contain the variables `THETA` (range: 0-90° in degree), `theta` (range: $0 \dots \pi$), `PHI` (range: 0-360° in degree) or `phi` (range: $0 \dots 2\pi$). The parameters `ex, ey, ez` represent a vector which defines the direction where $\phi = 0$.

3. **Independent energy and angular distribution functions**
   `set_emission_function_EA("species", eMax, ne, formula_E, nTheta, nPhi, formula_theta_phi, ex, ey, ez);` This function - as a combination of the two previous functins - allows to set independent energy and angular distributions. The energy distribution function for the emitted `"species"` is defined on an energy interval `0...eMax` with `ne` being the number of slots. The energy unit used to specify the range is eV, and the analytical formula as a function of `E` is specified via parameter `formula_E`. The resolution of the angular distribution function is defined by the number of slots, `nTheta` for the polar angle and `nPhi` for the azimuth angle. The analytical function parameter `formula_theta_phi` may contain the variables `THETA` (range: 0-90° in degree), `theta` (range: $0 \dots \pi$), `PHI` (range: 0-360° in degree) or `phi` (range: $0 \dots 2\pi$). The parameters `ex, ey, ez` represent a vector which defines the direction where $\phi = 0$.
4. **Correlated energy and angular distribution function**
   `set_emission_function_Corr("species", eMax, ne, nTheta, nPhi, formula, ex, ey, ez);` This function allows to specify a *correlated* energy and angular distribution function. The energy distribution function for the emitted `"species"` is defined on an energy interval `0...eMax` with `ne` being the number of slots and the energy unit in eV. The resolution of the angular distribution function is defined by the number of slots, `nTheta` for the polar angle and `nPhi` for the azimuth angle. The distribution function specified in `formula` can contain all variables `E, THETA, PHI, theta, phi` as specified above.

Specifying energy loss functions

In case of e.g. reflected neutrals from ion impact or reflected high energetic electrons, where a certain *energy loss* occurred in the material, the energy distribution of the reflected species is not given as a function of absolute energy. Instead, there is a distribution function for either the absolute or the relative energy loss. Such cases can be specially treated:

1. **Relative energy loss function** `set_energy_loss_rel("species", formula);` A relative loss function occurs e.g. for fast reflected neutrals resulting from ion impact on a target. Here, the ratio between impact and emitted energy depends e.g. on a mass ration between ion and target atom and thus, the loss energy distribution is defined on a relative scale `E=0...1`. This function uses a resolution of 1000 slots of the relative energy scale by default. The parameter `formula` may depend on the variable E, which - in this case - is not the absolute energy but a relative one (E=0...1), where E=1 corresponds to the impact energy.
2. **Absolute energy loss function** `set_energy_loss_abs("species", eMax, ne, formula);` This realizes an absolute loss function. One scenario may be a high energetic electron impinging on a surface thereby inducing some Auger excitation or other energy loss mechanism in the material which would substract an absolute loss energy from the electron. In this case, the energy range is E=0...eMax with the unit in eV, and the number of slots is specified in the `nE` parameter.

In these energy loss functions, the direction of the reflected particle is currently set specularly with respect to the impinging particle (even if e.g. the species changed from ion to neutral). In future versions this might be modified e.g. by specifying an angular cone for the reflected particle.

Example: In plasma simulations, the creation of fast reflected neutrals by ion impact can be realized as follows:

```
add_plain_reaction("Arplus", 1, 1, ["Ar", "e"], [1, 0.1]);
set_emission_function_E("e", 1.0, 10, 100, exp(-((E-5)^2)/2));
set_energy_loss_rel("Ar", E>0.35);
```

In this example, an `Arplus` ion impinging onto a surface results in an electron with a secondary electron yield of 0.1 (10%) and a neutral Ar atom[7] with a yield of 100%. From the reflected neutral (Ar) the relative loss energy distribution is set to `E>0.35`. In the relative scale E=0...1 this evaluates to 1 of E>0.35 and 0 if E<=0.35. It means that the energy of the reflected Ar is reduced by 35% or more in comparison with the impact energy of the ion. In other words: The energy loss is equally distrbuted between 35% and 100% (The energy distribution of the secondary electron is set as Gaussian function centered around 5 eV, but this is not important, here).

Footnotes

1. R. Behrisch, W. Eckstein: Sputtering by Particle Bombardment, Springer Science & Business Media, page 135, Table 1, https://books.google.de/books?id=K2Oh8QQ1Kf0C&pg=PA135 ↩

2. M.W. Thompson, Physical Mechanisms of Sputtering, *Physics Reports* **69** (1981) 335-371 --> see equation (12). ↩

3. Yamamura, Y.; Takiguchi, T.; Ishida, M.: Energy and angular distributions of sputtered atoms at normal incidence, In: Radiation Effects and Defects in Solids 118 No. 3 (1991) p. 237-261; see equation (13) on page 245. ↩

4. G.K. Wehner, D. Rosenberg, Angular Distribution of Sputtered Material, *Journal of Applied Physics* **31/1** (1960) 177-179 ↩

5. the graph is adapted from A. Pflug, M. Siemers, T. Melzig, D. Rademacher, T. Zickenrott, M. Vergöhl, Numerical optimization of baffles for sputtering optical precision filters, *Surface and Coatings Technology* **241** (2014) 45-49. ↩

6. M.W. Thompson, The energy spectrum of ejected atoms during the hig energy sputtering of gold, *Philosophical Magazine* **18** (1968) 377-414. ↩

7. If the neutral density of Ar is very high with respect to ions and electrons, it might make sense to define a special species such as `ArFast` in order to distinguish the fast reflected neutrals from the background gas ↩

28 Wall temperature

A non-trivial temperature distribution on a wall is especially used in combination with a vapor pressure dependent particle source in order to get a more realistic behavior of e.g. large evaporation sources.

In order to define a temperature distribution rather than a single fixed temperature, the following command has to be used:

```
add_temperature("temperature_distribution.txt");
```

The temperature distribution file format is currently according to Comsol-generated temperature output files. The format is as follows:

- First line: Title (will be ignored)
- Next lines: Seven columns of numbers separated by TAB or whitespace with following meaning: `<Facet> <Point> <X> <Y> <Z> <Element> <Temperature [K]>` Only the (X, Y, Z) coordinates and the temperature will be read in, the other columns will be ignored.

The temperature can be defined at arbitrary points within the simulation volume, there is no specific ordering of the points required. Upon initialization, for each mesh element the nearest point of the temperature distribution is chosen and the respective temperature is put into the mesh element.

There is no Python counterpart of `add_temperature`: the file-based temperature distribution is only read by the parameter file. `BorderReaction.set_temperature_function(T0, T1, nT, function)` is a different mechanism - it scales the probability of a reaction with the local wall temperature, sampled in `[T0, T1]`, and does not assign temperatures to a surface. Until that exists, a Python case keeps the single value from `Border(T = …)`.

29 Secondary electron emission

According to Zalm84 [1] there ary two mechanisms for electron ejection upon ion bombardment:

- **Potential Emission:** If the ion neutralization energy exceeds twice the electron work function of the material, electron emission is possible. The secondary electron emission yield resulting from this mechanism is energy dependent and applies for rather low ion incident energies up to a few 100 eV.
- **Kinetic Emission:** At higher energies, electron emission by direct ionization of metal atoms by the ion projectile and/or by energetic recoils becomes important. This mechanism becomes relevant for ion incident energies beyond 1 keV.

Given the relevant energy range, only the first mechanism seems relevant for magnetron sputtering. The secondary electron emission coefficient related to *potential emission* can be calculated as:

$$\gamma_{\mathrm{PEE}} \approx 0.2 \cdot \frac{0.8\,E_{\mathrm{i}} - 2\,\phi}{\epsilon_{\mathrm{i}}}$$

- E_i = Ionization energy of the ion (= 15.8 eV for Ar^+)
- ϕ = Electron work function of the metal
- ϵ_i = Fermi energy of the electrons within the metal

In the following, representative numerical values are given. Most of the values are taken from the following sources:

- Table of fermi energies from http://hyperphysics.phy-astr.gsu.edu/hbase/tables/fermi.html#c1 from the Hyperphysics project (University of Georgia)
- Wikipedia article about work function

Metal	Work function [eV]	Fermi energy [eV]	ISEE for Ar^+ (15.8 eV)	ISEE for O_2^+ (12.06 eV)
Ag	4.63	5.49	0.123	0.014
Al	4.16[2]	11.7	0.073	0.023
Cu	4.53-5.10	7.00	0.070-0.1022	0.0 - 0.017
Ti	4.33	4.556[^3]	0.175	0.043

- Bhokare, V. V.; Yussouff, M.: Electronic Structures of Beryllium and Titanium by Green's Function Method, In: Il Nuovo Cimento **19 B, 2** (1974), pp. 149-160, section 3.2 on p. 154.

Setting the yield in a case

The yield is not a property of the border but of the reaction that emits the electrons: the reaction products carry it, and it can be made energy dependent with a yield function.

```
add_plain_reaction("Arplus", 1, 1, ["Ar", "e"], [1, 0.1]);    # gamma = 0.1
set_yield_function("e", 0, 1000, 1000, (E>20) ? 0.1*sqrt(E/20) : 0);
```

`set_yield_function` takes the species after `E0`, `E1` and `nE` in Python, and the formula string becomes a callable of the impact energy. The angular dependence is available as `set_yield2D_function`, see Energy dependent yield.

Footnotes

1. Zalm, P. C.; Beckers, L. J.: Ion-induced secondary electron emission from copper and zinc, In: Surface Science **152/153** (1985), pp. 135-141. ↩

2. average value ↩

30 Energy/angle dependency

Energy dependent reaction probability

If reactions have an energy-dependent reaction probability, this can be defined via the function

```
set_probability_function(E0, E1, nE, function);
```

This function is to be put just after the last reaction definition, and the parameters are as follows:

- **E0, E1**: Defines the energy interval [E0, E1]; the unit is eV.
- **nE**: The number of sub-intervals within [E0, E1];
- **function**: Functional expression, where the energy can be used as variable **E**

If the actual particle energy E is outside of the [E0, E1] interval, the probability values at E0, E1 will be taken for E<E0, E>E1, respectively.

Example: In *McRae1976*[1], a detailed measurement of the electron reflection behaviour at Copper surfaces is shown in the following graph:

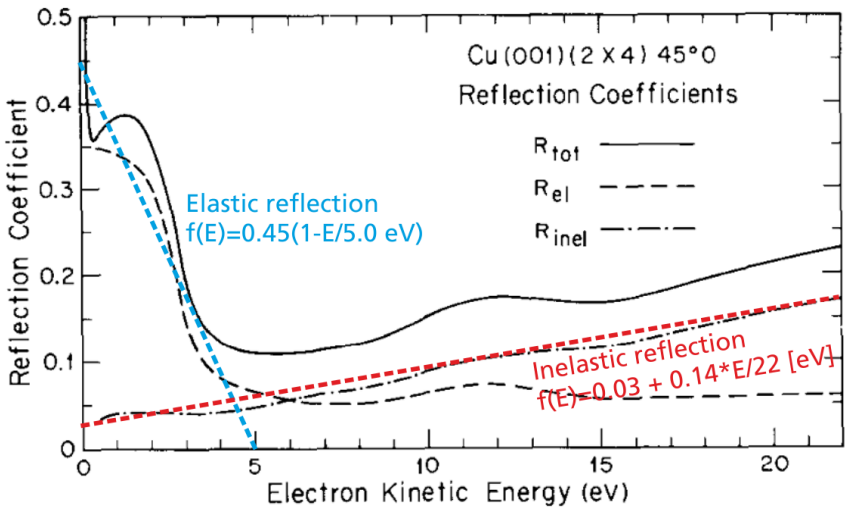


Fig. 9. Observed total, elastic and inelastic reflection coefficients for oxygen-covered Cu(001) surface.

Fig. 6: Elastic and inelastic electron reflection at Copper surfaces due to Fig. 9 of *McRae1976*

As can be seen, there is an elastic and an inelastic reflection probability curve; both curves can be approximated in first order by a linear function. In order to use these behaviour in PIC-MC simulations, the following plasma-wall interaction definition would be appropriate:

```
add_plain_reaction("Arplus", 1, 1, ["Ar", "e"], [1, 0.1]);
set_emission_energy("e", 0, 10, 100);

add_specular_reflection("e", 1.0);
set_probability_function(0, 5, 20, 0.45*(1-E/5));
add_absorption("e", 1.0);
set_probability_function(0, 22, 100, 1-(0.03+0.14*E/22));
```

The upper part of the code defines the emission of secondary electrons upon impact of positive ions. The reflection behaviour of low-energy electrons is defined in the lower four lines. First, a specular reflection

is added with 100% probability. Afterwards, the energy-dependent probability function is applied. The resulting reaction probability will be the probability given in the `add_specular_reflection` function *multiplied with* the probability function (see also blue line in Fig. 6) given in the next command.

Since inelastic reflection is the default behaviour of all surfaces unless other reactions are specified, there is no `set_inelastic_reflection` command implemented. Instead the inverse process, i. e. the electron absorption at the wall is specified. Consequently, the probability function of the absorption is the inverse probability of the inelastic reflection probability (shown as red line in Fig. 6).

Energy dependent yield

If we have an energy-dependent *yield* of reaction products, it is better to use a yield function instead of the reaction probability. Its syntax is

```
set_yield_function("species", E0, E1, nE, function);
```

In Python the species comes after `E0`, `E1` and `nE`, and the formula string is a callable of the impact energy.

The parameters are as follows:

- `species`: Name of reaction product, whereof the yield energy dependency should be specified
- `E0, E1`: Energy interval of interest
- `nE`: Number of points within energy interval
- `function`: Arithmetic expression, where the variable `E` can be used for energy

The yield function will be multiplied to the yield given in the reaction product vector. All energy values are to be specified in (eV).

An example is the sputtering yield of Ti by Ar^+ impact, whereof an empirical yield formula can be constructed from the data in [2] as follows:

$$Y(E) = 0.01085(E/(1eV) - 118)^{2/3}(E \geq 118eV)$$

For this case, there are now two seemingly equivalent possible definitions:

Yield function

```
add_plain_reaction("Arplus", 1, 1,
                   ["Ar", "e", "Ti"], [1, 0.1, 1.0]);
set_emission_sputter("Ti", 4.89, 1.5);
set_yield_function("Ti", 0, 1000, 1000,
                   (E>118) ? 0.01085*(E-118)^(2/3) ! 0);
```

Probability function

```
add_plain_reaction("Arplus", 1, 1,
                   ["Ar", "e", "Ti"], [1, 0.1, 1.0]);
set_emission_sputter("Ti", 4.89, 1.5);
set_probability_function(0, 1000, 1000,
                         (E>118) ? 0.01085*(E-118)^(2/3) ! 0);
```

However there are two important differences:

- The probability function is applied to the whole sputtering process including creation of secondary electrons and Ar neutrals. This is not appropriate since e. g. the energy dependency of secondary electron generation should be different.
- With a probability function, the whole reaction either takes place or not. With a high maximal yield but a low average probability, poor numerical statistics will result (there will be Ti emission only in rare cases but then, a large bunch of Ti particles will be emitted). Instead, a constant reaction probability combined with an energy-dependent yield function will provide a smooth statistical process.

Energy and angular dependent yield

In some cases, e.g. in sputtering processes, the number of emitted particles also depends on the angle of the impinging ion. In such cases, the yield of reaction products depends on both, the impinging particle energy and its angle. In such cases, the following function can be used to describe the energy and angular yield dependency:

```
set_yield_function_2D("species", E0, E1, nE, nTheta, function);
```

The parameters are as follows:

- `species`: Name of reaction product, whereof the yield energy dependency should be specified
- `E0, E1`: Energy interval of interest
- `nE`: Number of points within energy interval
- `nTheta`: Number of points for interval of polar angle θ
- `function`: Arithmetic expression, where the variables `E`, `THETA` can be used for the energy and polar angle, respectively. The energy unit is [eV], the angular unit is radian in the range $[0, \ldots, \pi/2[$.

A simplified interface for typical cases is

```
set_yield("species", function);
```

The shortcut has no Python form; the same table is set up with `set_yield2D_function` and the ranges given explicitly.

Here, the energy range is predetermined to $[0, \ldots, 1000]$ eV and the angular range is divided into 20 intervals. These settings should be appropriate e.g. for most sputtering processes.

An alternative yield function is given in [3]. For an example calculaton (Ar ions on Ti) you can use the following script:

```
float qq      = 4.8957;
float epsilon = 8.56428*10^4;
float E_th    = 25.019;
float mu      = 1.8291;
float lambda  = 0.3152;

float E       = 200;

out "$(qq/2*ln(1+1.2288*E/epsilon)/(E/epsilon+0.1728*sqrt(E/epsilon)+0.008*(E/epsilon)^(0.1504))*
      (E/E_th-1)^mu/(lambda+(E/E_th-1)^mu))\n";
```

The parameter at the beginning have to be set according to the species involved. The script can then be executed with:

```
rvm sputter_yield.r
```

Footnotes

1. McRae, E. G.; Caldwell, C. W.: Very low energy electron reflection at Cu(OO1) surfaces, In: Surface Science 57 (1976), pp. 77-92. ↩

2. Ranjan, R.; Allain, J. P.; Hendricks, M. R. & Ruzic, D. N.: Absolute sputtering yield of Ti/TiN by Ar+ / N+ at 400-700 eV, In: Journal of Vacuum Science & Technology A 19 (2001), pp. 1004-1007 ↩

3. R. Behrisch, W. Eckstein: Sputtering by Particle Bombardment, Springer Science & Business Media, page 38, https://books.google.de/books?id=K2Oh8QQ1Kf0C&pg=PA38 ↩

31 Surface chemistry

With plain reactions, particle absorptions and transformations are defined without considering the state of the surface. A more advanced approach allows to define similar reactions **in dependency of the actual surface material coverage**. Fig. 1 below shows an example of a surface coated with two different material components, which is exposed to the gas and/or plasma.

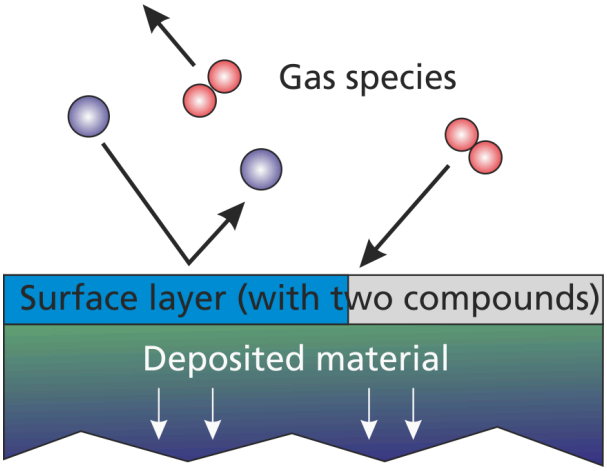


Fig. 1: Reactive plasma-wall interaction model.

According to this representation, the advanced surface chemistry model considers three different regions:

1. The gas phase and/or plasma
2. The first atomic layer of the surface
3. The deeper atomic layers which are in total representing the deposited layer material.

As already indicated, the surface layer and the deposited layer consists of multiple material compounds. A reasonable reactive model should comprise at least two different surface materials e. g. `"Al"` as metallic aluminum and `"AlOx"` as metal-oxide in a reactive deposition model.



note

The surface chemistry model has no Python counterpart yet. `Sim` accepts a `Materials` argument, but the code marks it as unfinished, and the commands of this page (`add_material`, `set_initial_coverage`, `add_surface_reaction`, `add_surface_deposition`, `add_cover_deposition`, `add_implantation`) exist only in the parameter file. A reactive deposition case runs from the `.par` route.

Specification of wall materials and initial coverages

If we stick to the reactive AlOx deposition model, the first step is to declare two different names for the metal and metal-oxide:

```
add_material("Al", 1e19);
add_material("AlOx", 1e19);
set_initial_coverage("AlOx", 0.5);
```

The involved commands are

- Declare material name with bond density `add_material("material_name", bond_density);`
- Declare initial surface coverage of a material `set_initial_coverage("material_name", initial_coverage);`

The **bond_density** parameter represents the surface density of bond sites of the material in [1/m^2]. For realistic molecular distances, the bond density typically is in the 10^{19} [1/m^2] range. However it is possible to chose a smaller value in the simulation (e. g. 10^{17}) in order to speed things up. If the wall is in equilibrium with the gas/plasma, the actual bond density has no influence on the result - it only influences the time required to reach the equilibrium state.

Note: In the current version, different bond density values for different materials are not yet supported and lead to undefined behaviour.

In order to get the deposition in the right unit you need to calculate the bond density depending on your material. The example calculation for AlOx is the following:

$\rho_{Al_2O_3} = 3.94\ g\,cm^{-3}$

$m_{Al_2O_3} = 102\ u = 1.7 \cdot 10^{-22}\ g$

$\rho'_{Al_2O_3} = \rho / m = 2.3 \cdot 10^{22}\ cm^{-3} = 2.3 \cdot 10^{28}\ m^{-3}$

$\rho''_{Al_2O_3} = \rho'^{2/3} = 8 \cdot 10^{18}\ m^{-2}$

$\rho''_{AlOx} = 2 \cdot \rho''_{Al_2O_3} = 1.6 \cdot 10^{19}\ m^{-2}$

In the example given above it is sufficient to specify the initial coverage of only one material, e. g. AlOx. The coverage of Al will be automatically adjusted in a way that the sum of all coverages is 1.0. This works also for more than two materials.

Example:

```
add_material("Ti", 1e19);
add_material("TiO", 1e19);
add_material("TiO2", 1e19);
set_initial_coverage("TiO2", 0.9);
```

Here, the surface consists by 90% of `"TiO2"` (i. e. it is almost fully oxidized). The remaining 10% of the surface coverage is equally distributed amongst `"TiO"` and `"Ti"`. If we write

```
add_material("Ti", 1e19);
add_material("TiO", 1e19);
add_material("TiO2", 1e19);
set_initial_coverage("TiO2", 0.9);
```

```
set_initial_coverage("TiO", 0.1);
```

then the initial coverage of `"Ti"` will be zero.

Surface reaction involving no deposition

```
add_surface_reaction("species", flag, amount, probability, "material_before", "material_after",
["rp1", "rp2", ...], [n1, n2, ...]);
```

This is the syntax for a surface reaction without deposition. The parameters are

- **species** => String specifying the gas species
- **flag** => Specifies, if gas particle is absorbed (flag==1) or not (flag==0)
- **amount** => Number of gas particles required to accomplish one reaction (can be non-integer)
- **probability** => The reaction probability
- **material_before, material_after**: Strings specifying the required initial surface material and resulting surface material.
- **`["rp1", "rp2", ...], [n1, n2, ...]`**: Reaction products and their amounts. As always their emission characteristics is Maxwellian by default and can be modified via `set_emission_sputter`, `set_emission_energy` etc.

Still sticking to the reactive AlOx example, the oxidation of Aluminum can be described by Al + 0.75 O_2->AlO_x. The number 0.75 comes from the chemical formula Al_2O_3 of aluminum oxide [1]. Translated to the command syntax and assuming a reaction probability of 50%, this yields

```
add_surface_reaction("O2", 1, 0.75, 0.5, "Al", "AlOx", [], []);
```

Surface reaction involving deposition

While the surface oxidation of Al described above leads to a change in the surface material coverage, it does not increase the deposited film thickness. In contrast, if metallic Al is absorbed on the surface this leads to film growth. For that purpose, a modified syntax is required:

```
add_surface_deposition("species", amount, probability, "material_before", "material_after",
"material_deposited", amount_deposited, ["rp1", ...], [n1, ...]);
```

Here, an additional parameter **material_deposited** describes the material component which contributes to the deposition zone. With the parameter **amount_deposited** the amount of surface sites per reaction can be controlled. There is no `flag` for the incoming gas species, since it is assumed that for a deposition process it will be always absorbed. Thus, the complete reactive AlOx example would look like follows (for Al with a deposition probability of 100%):

```
add_surface_reaction("O2", 1, 0.75, 0.5, "Al", "AlOx", [], []);
add_surface_deposition("Al", 1, 1.0, "Al", "Al", "Al", 1, [], []);
add_surface_deposition("Al", 1, 1.0, "AlOx", "Al", "AlOx", 1, [], []);
```

Please note that surface oxidation by `"O2"` only works if there is a non-zero metal fraction of `"Al"` present at the surface. `"Al"` deposition works for both possible surface materials.

Surface cover deposition

Many deposition reactions such as deposition of sputtered metal work equally on all possible surface materials. In all of these cases, the deposited metal "covers" the actual surface layer component and replaces it by itself. The surface layer component will then contribute to the deposited film. A visualization of this process is given in Fig. 2.

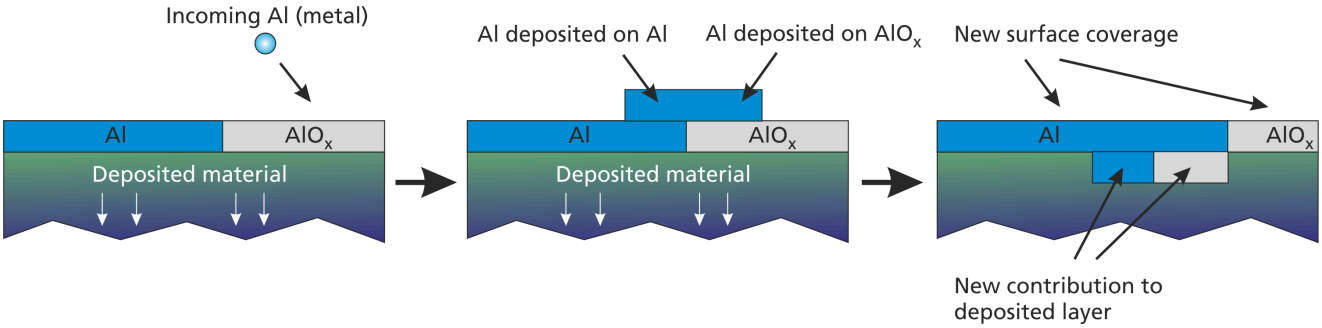


Fig. 2: Deposition sequence in the view of the wall reaction model.

For this "covering" deposition, a special command can be used: `add_cover_deposition("species", amount, probability, "deposition_material", ["rp1", "rp2", ...], [n1, n2, ...]);` Here, the **deposition_material** parameter specifies, which material is deposited on top of all possible surface materials. In the reactive AlOx example, the following definitions are equivalent:

```
add_surface_deposition("Al", 1, 1.0, "Al", "Al", "Al", 1, [], []);
add_surface_deposition("Al", 1, 1.0, "AlOx", "Al", "AlOx", 1, [], []);
```

... and ...

```
add_cover_deposition("Al", 1, 1.0, "Al", [], []);
```

Implantation

A third deposition mechanism performs deposition by implantation, i. e. without modifying the state of the top surface layer:

```
add_implantation("species", amount, probability, "deposition_material", ["rp1", "rp2", ...], [n1,
n2, ...]);
```

Here, the **deposited_material** is implanted into the layer *through* the top surface layer and without altering the surface coverage configuration. A visualization of such process in given in Fig. 3.

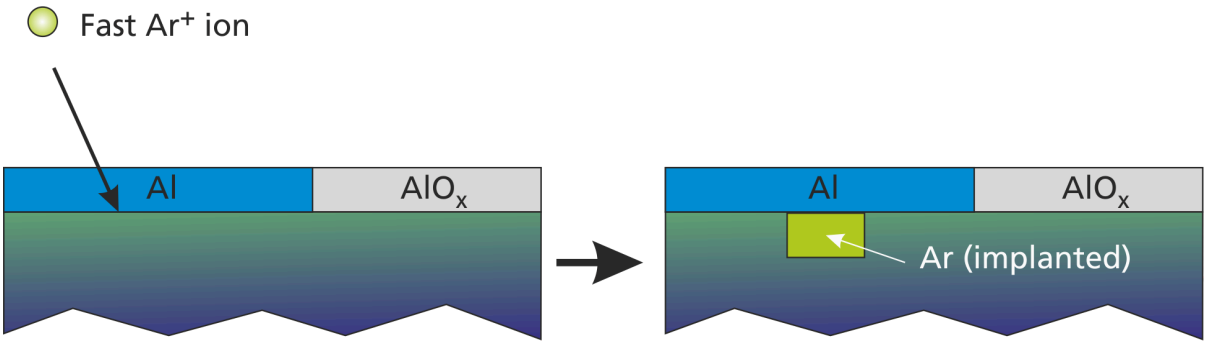


Fig. 3: Implantation in the view of the reactive wall model

The accordant command for introducing Ar implantation by fast Ar+ bombardment with 20% probability is

```
add_implantation("Arplus", 1, 0.2, "Ar", ["e"], [0.05]);
set_emission_energy("e", 0, 10, 1.0);
```

Here, additionally a low-energy secondary electron (E=0..10 eV) can be emitted with 5% probability during such process.

Example for a surface chemistry model for reactive ZnO:Al deposition

In this example, the reactive deposition of doped ZnO:Al layers is analyzed[2]. Especially the doping efficiency of the Al dopant is investigated. For this purpose, a simplified DSMC reactor model consisting of two Zn:Al sputter sources, an Ar gas inlet, an O2 gas inlet as well as a substrate is considered (see Fig. 4).

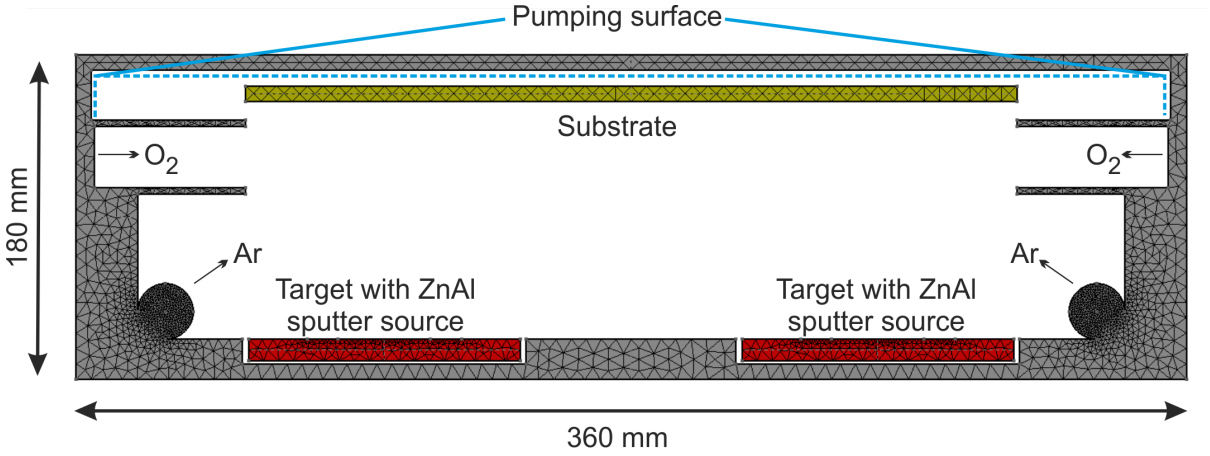


Fig. 4: Sketch of a 2D reactor model for reactive deposition of ZnO:Al

For the inclusion of Al into the ZnO, the following hypothetical model is constructed (see Fig. 5):

- On the one hand, metallic Zn coated on the substrate has a low binding energy (in fact, metallic Zn has a high vapor pressure). Thus, it should be rather easy for an incoming Al atom to replace one Zn atom on the surface thereby creating a Zn:Al mixed site.
- We assume that oxidation of a Zn:Al mixed site leads to a ZnO:AlO site, where the Al atom can act as dopant
- If the substrate is covered by ZnO on the other hand, the binding energy is rather high. Thus, an incoming Al atom is not able to replace a Zn atom and is rather being deposited as a separate Al site.
- The oxidation of separate Al sites leads to formation of separated, insulating AlOx clusters, which do not contribute to the doping process.

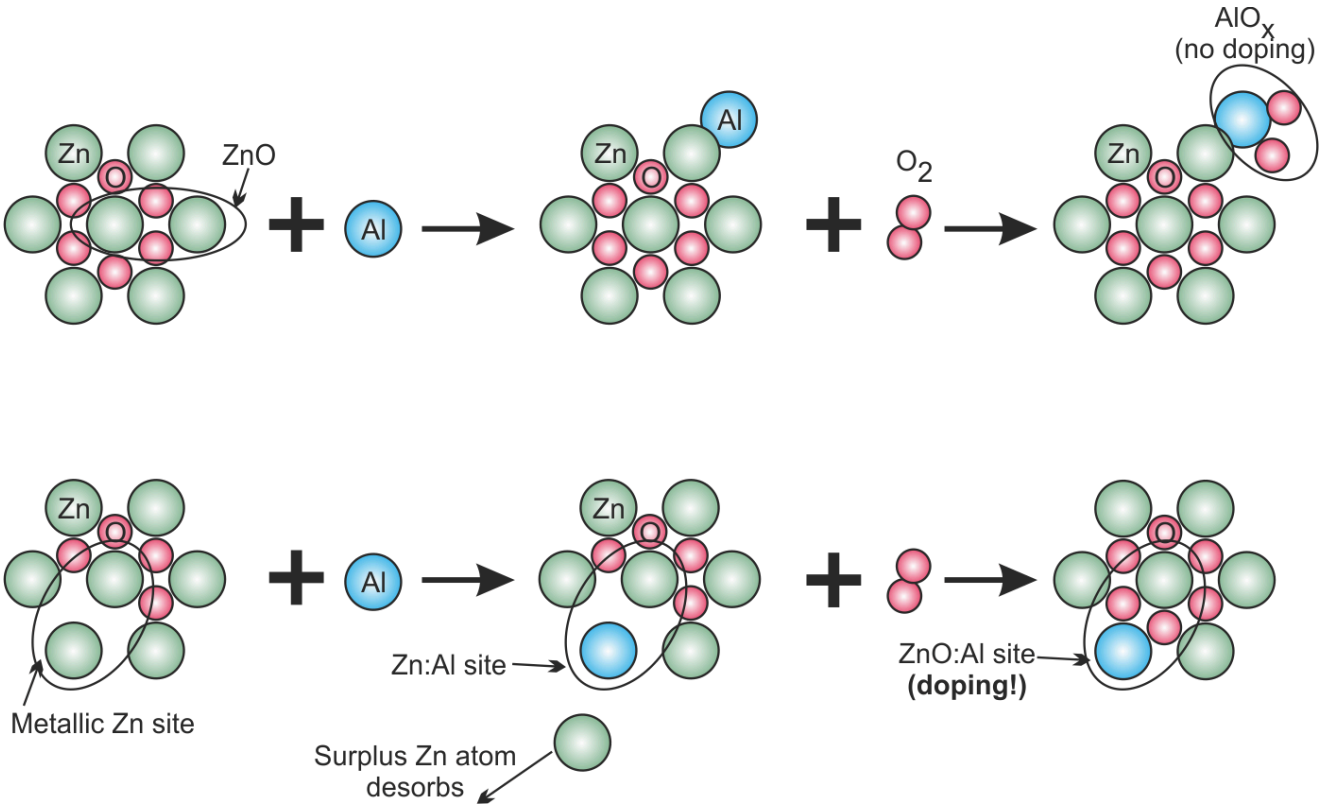


Fig. 5: Hypothetical model for the reactive deposition of doped ZnO:Al layers.

With four gas species `"Ar", "O2", "Zn", "Al"` the above described growth model can be implemented via the wall reaction model as shown in the following:

```
# Initialize six possible surface material compounds:
add_material("Zn2",      2e17);
add_material("Al2",      2e17);
add_material("Zn2O2",    2e17);
add_material("ZnAl",     2e17);
add_material("ZnOAlO",   2e17);
add_material("Al2O3",    2e17);
set_initial_coverage("Zn2",      0.5);
set_initial_coverage("Zn2O2",    0.5);

# Oxidation of Zn (gettering coefficient 50%)
add_surface_reaction("O2",  1, 1, 0.5, "Zn2", "Zn2O2", [], []);

# Exchange reaction Al (gas) + Zn2 (solid) --> ZnAl (solid) + Zn (gas)
add_surface_reaction("Al",  1, 1, 1.0, "Zn2", "ZnAl", ["Zn"], [1]);

# Exchange reaction Zn (gas) + Al2 (solid) --> ZnAl (solid) + Al (gas)
add_surface_reaction("Zn",  1, 1, 1.0, "Al2", "ZnAl", ["Al"], [1]);

# This is the insulating  species (gettering coefficient 70%)
add_surface_reaction("O2", 1, 1.5, 0.7, "Al2", "Al2O3", [], []);

# This is the doped oxide species (gettering coefficient 70%)
add_surface_reaction("O2", 1, 1, 0.7, "ZnAl", "ZnOAlO", [], []);

# Film growth by Zn or Al covering deposition
add_cover_deposition("Zn", 2, 1, "Zn2", [], []);
add_cover_deposition("Al", 2, 1, "Al2", [], []);
```

Footnotes

1. i. e. "AlOx" stands for a "half Al_2O_3 molecule" ↩

2. Pflug, A.; Siemers, M.; Melzig, T.; Sittinger, V.; Schäfer, L.: Heuristic modelling of the doping efficiency in sputtered TCO layers, In: Surface and Coatings Technology **267** (2015) 81-89. ↩

32 Surface state sources

Particle sources may change their behavior depending on their surface coverage. Examples, which are implemented in the DSMC algorithm are

- Sputtering sources with a given ion flux
- Thermal evaporation of coated material with high evaporation pressure



note

Both source types read the surface coverage and therefore need the material model of the surface chemistry, which has no Python counterpart yet: `add_surface_emission`, `add_desorption` and `add_desorption_r` are parameter file only.

Sputtering sources with given ion flux

A sputtering source with a given ion flux is given by the command

```
add_surface_emission("material", ion_flux, ["rp_1", "rp_2", ...], [yield_1, yield_2, ...]);
```

Parameters are as in the following:

- **material** -> Material phase in the surface layer. For different surface materials different sputtering processes can be defined.
- **ion_flux** -> The effective ion flux is given in sccm.
- **["rp_1", "rp_2", ...]** -> list of gaseous reaction products
- **[yield_1, yield_2, ...]** -> list of partial yield of the gaseous reaction products

The sputtering yield of a gasous reaction product is the product of the reaction **probability** and its partial **yield**. As an example, the surface chemistry and sputtering source for the reactive TiOx process is given:

```
# Specification of surface material phases and initial coverage:
add_material("Ti_metallic", 9e18);
add_material("TiO2",        9e18);
set_initial_coverage("TiO2", 0.9);

# Surface reactions:
add_surface_reaction("O2", 1, 1, 1.0, "Ti_metallic", "TiO2", [], []);
add_cover_deposition("Ti", 1, 1, "Ti_metallic", [], []);

# Sputtering source:
add_surface_emission("Ti_metallic", ion_flux_sccm, ["Ti"], [0.5]);
set_emission_sputter("Ti", 4.89, 1.5);
add_surface_emission("TiO2", ion_flux_sccm, ["Ti", "O2"], [0.05, 0.05]);
set_emission_sputter("Ti", 6.0, 1.5);
set_emission_sputter("O2", 6.0, 1.5);
```

In the above example, sputtering of metallic Ti and oxidized TiO_2 is handled separately. The sputtering yield of TiO_2 is by factor of 10 lower compared to metallic Ti. The energy distributions of the sputtered gaseous species are to be specified afterwards for each case; the recommended form is `set_emission_thompson("Ti", Ub, alpha, E_ion, m_ion)`, described in particle emission. The older `set_emission_sputter` used above still works, but it assumes argon ions and a cutoff of 100 times the binding energy.

Note: When sputtering the surface layer, it is being replaced by the material beneath the surface layer.

- If there is deposition on the sputtering surface, the replaced surface fractions are according to the stoichiometry of the material in the deposition layer.
- If there is no deposited material beneath the surface layer, the replacement material will be the material which is firstly defined in the deposition chemistry (i.e. Ti_metallic in the above example).

Thermal desorption of coated material

If surfaces are coated with materials with high vapour pressure, there is the possibility of thermal **desorption** of that material. For this purpose, a **desorption source** is implemented. It can be declared via the syntax

```
add_desorption("species", pressure, "material", amount);
```

The parameters are as follows:

- **species**: The species, which is to be emitted from the wall into the gas phase
- **pressure**: The vapor pressure in Pa. The vapor pressure generally depends on the wall temperature. For many materials, the vapor pressure curve as a function of temperature is available in literature[1].
- **material**: The associated wall material.
- **amount**: The number of gas molecules needed to form a bond site of the wall material

If the surface material does not completely desorb but leaves behind some remainder, modified command is available:

```
add_desorption_r("species", pressure, "material", amount_gas_species, "remainder",
amount_remainder);
```

Here, we have the following parameters:

- **species**: The species, which is to be emitted from the wall into the gas phase
- **pressure**: The vapor pressure in Pa.
- **material**: The associated wall material.
- **amount_gas_species**: The number of gas molecules needed to form a bond site of the wall material.
- **remainder**: The wall material which is left behind by the desorption process.
- **amount_remainder**: The number of wall sites of the remaining material created by the desorption process.

Example: If we have a metallic `ZnAl` mixed site and assume that only Zn can desorp and leave behind a half `Al2` site, the command would look like follows:

```
add_desorption_r("Zn", vapor_pressure, "ZnAl", 1, "Al2", 0.5);
```

Footnotes

1. For metals see e. g. Alcock et al.; Vapour pressure equations for the metallic elements: 298-2500 K, *Canadian Metallurgy Quaterly*, **1984**, 23, 309-313 ↩

33 Membrane border type

Membranes are declared by setting the border type to `"membrane"`. They consist of single-sided separate mesh surfaces which do not belong to a closed 3D object. In that sense, they are *virtual* surfaces that can be used for sampling the energy distribution function in free-space. Membranes can act as particle sources and can perform all plain reactions. However surface chemistry reactions are excluded for membranes, since virtual surfaces are not supposed to have a deposition material coverage. There are further possible functions for a membrane, which are listed in the following.

For simulations with many species and a membrane which should be fully transparent for all of these, you can use the following command:

```
reactions: {
  Transparent_Membrane;
};
```

Internally a loop over all used species is applied for which and the transmission is set to 1.

- **Fully transparent membrane (for sampling purposes, for more details please refer to energy and angular sampling):**

```
    Border Sensor = {

      icodec =   2;                       # Codec number from meshfile
      type = "membrane";
      T = PAR.T0;                         # Wall temperature
      reactions: {
        add_transmission("Ar", 1.0, [], []);
        add_transmission("O2", 1.0, [], []);
        add_energy_histogram("Ar", 0, 1, 1000);
        add_angular_histogram("O2", 9, 36,     1, 0, 0);
      };
    };
```

- **Membrane acting as reflector**

```
    Border Barrier = {
        icodec =   2;                       # Codec number from meshfile
        type = "membrane";
        T = PAR.T0;                         # Wall temperature
        reactions: {
          # The default behaviour of membranes is 100% diffuse reflection for all species
        };
      };
```

- **Transparent membrane which transforms 30% of NH3 into NH3ex**

Important note: For membranes it is implicitely assumed that the sum of reaction products masses is the same as the mass of the incoming species. Thus, for maintaining momentum conservation the reaction products are inserted with the same velocity vector as the incoming gas particle. This behaviour is unlike a regular ""wall"" where the reaction products get a Maxwellian velocity distribution by default.

```
    Border Activator = {
        icodec =   2;                       # Codec number from meshfile
        type = "membrane";
        T = PAR.T0;                         # Wall temperature
```

```
        reactions: {
          # Provide 100% transmission for all other species:
          add_transmission("SiH4",  1, [], []);
          add_transmission("NH3ex", 1, [], []);
          add_transmission("H2",    1, [], []);

          # 30% probability for conversion of NH3 to NH3ex
          add_plain_reaction("NH3", 1, 0.3, ["NH3ex"], [1]);

          # 100% transmission for the non-converted NH3
          add_transmission("NH3", 1);
        };
      };
```

- **Membrane acting as barrier and absorbing Zn with 100% sticking coefficient**

```
    Border Barrier = {
        icodec =   2;                       # Codec number from meshfile
        type = "membrane";
        T = PAR.T0;                         # Wall temperature
        reactions: {
          add_absorption("Zn", 1);
        };
      };
```

Two membranes should be separated by a small value, so that the distance between them is not zero. Which means that you can not use the same *Line Loop* from GMSH only with different signs for two membranes aiming to sample particles from both directions. For the distance a millionth of the cell width should be sufficient.

34 Energy/angle sampling

Each particle hitting a surface can be recorded within an energy or angular histogram vector. This functionality is applicable for "walls" as well as for "membranes".

All histograms are taken per physical surface. **Important:** For membranes, the histogram is only taken for particles hitting the side, where the normal vector points out. This is useful if e. g. multiple histograms shall be taken for a large physical surface at different areas: In this case it is possible to use several transparent membranes placed close to the surface at the desired locations. Since particles hitting the surface can be diffusely reflected, it would be misleading if the reflected particles are also sampled by the membranes.

The involved commands for generating histograms must be written within the wall description blocks (see wall reaction model) and are listed in the following.

Energy histogram

```
add_energy_histogram("species", E0, E1, nE);
```

Parameters are:

- **species**: The gas species which shall be recorded
- **E0, E1**: Energy interval of interest
- **nE**: Number of histogram points within [E0, E1]

Angular histogram

```
add_angular_histogram("species", nTheta, nPhi, vx, vy, vz);
```

Parameters are:

- **species**: The gas species which shall be recorded
- **nTheta**: The number of polar angle intervals ($\theta \in [0,90]$)
- **nPhi**: The number of azimuth angle intervals over $\varphi \in [-180°, 180°)$
- **vx, vy, vz**: A vector defining the direction where the azimuthal angle φ shall be defined as zero[1]

Combined histogram

Since energy and angular distribution are generally correlated, it is also possible to generate a **combined energy and angular distribution:**

```
add_histogram("species", nTheta, nPhi, E0, E1, nE, vx, vy, vz);
```

The parameters have the same meaning as in the single functions described above. The two single-purpose commands are wrappers around this one and switch off the dimension they do not sample, which is why a one-dimensional histogram still carries the other two axes with a single bin. In Python `E0`, `E1` and `nE` are optional and default to `0` to `1000` eV in one bin, `v` defaults to `(1, 0, 0)`.

For a combined histogram please take care of the total number of histogram points (`nTheta * nPhi * nE`): if that number is very large, a very long time averaging interval will be needed to obtain useful statistics.

The resulting files in the output directory have the filenames **histogram__.txt**, e. g. **histogram_1_Substrate_Ti_60.00ms.txt**[2].

The file has one line per (energy, polar angle, azimuth) bin, that is `nE * nTheta * nPhi` lines, and four columns with numbers:

- Energy [eV]
- Theta [°]
- Phi [°]
- Sampled histogram intensity

A dimension that is not sampled is still present as a single bin, so an angular histogram with `nTheta = 18` and `nPhi = 36` writes 18 * 36 = 648 lines, all with the energy value of the single energy bin.

When making polar contour plots of the angular distribution, remember to adjust the intensity:

```
Intensity / sin(θ*π/180)
```

The reason for this is the integration in spherical coordinates, where the differential element is: $\sin(\theta) d\theta d\phi$.

Histogram array

It is possible to divide a physical surface into an array of histograms. This only works for single plane surfaces by using the following command:

```
add_histogram_array("species", nTheta, nPhi, E0, E1, nE, nX, nY,
                    xA, yA, zA, xB, yB, zB, xC, yC, zC);
```

Parameters are:

- **species**: The gas species which shall be recorded
- **nTheta**: The number of polar angle intervals ($\theta \in [0,90]$)
- **nPhi**: The number of azimuth angle intervals over $\varphi \in [-180°, 180°)$
- **E0, E1**: Energy interval of interest
- **nE**: Number of histogram points within [E0, E1]
- **nX, nY**: Number of histograms in x- and y-direction
- **xA, yA, zA**: First anchor point A of the histogram plane
- **xB, yB, zB**: Second anchor point B, defines together with A the x-direction
- **xC, yC, zC**: Third anchor point C, defines together with A the y-direction

The point coordinates are given in the unit of the mesh file (`UNIT` in the parameter file, `mesh_unit` in Python); with `UNIT = 0.001` they are millimetres. Coordinates in the wrong unit raise no error - the grid then lies far outside the geometry and every hit of the surface ends up in one cell. A histogram array that covers a single cell is almost always a unit mistake.

Two more traps in the Python form: the arguments are ordered differently (`nX`, `nY` and the three corners before the energy range), and the three corners are passed as tuples `x0`, `x1`, `x2`. Both sides take plain numbers, so a positional translation of the line above swaps them without a complaint.

In the resulting file instead of a single intensity column, there are $n_X n_Y$ columns with the order

$I_{00}, I_{10}, I_{20}, \ldots I_{n_X-1\,nY-1}$. If i_x, i_y are the indices of the X and Y partition and i_c is the column number, the conversion rules are

$$i_c = \quad i_x + n_X i_y \in [0, \ldots (n_X n_Y) - 1]\; i_x = \quad i_c \bmod n_X \in [0, \ldots n_X - 1]\; i_y = \quad \lfloor i_c / n_X \rfloor \in [0, \ldots n_Y - 1]$$

Warning: Please take care not to use too many histogram slots / histogram segments as this could easily lead to memory overflow. A bin costs about 32 bytes and every quad keeps its own copy of the array: 20 energy bins times 9 polar angles times 18 azimuth angles on a 31 x 48 grid is 4.8 million bins, some 154 MB per plane and quad. If a simulation run with large histogram arrays is crashing on startup, please check the memory consumption on the compute nodes and consider using a fewer number of slots / histogram segments.

Besides the general function there is an energy (`add_energy_histogram_array`) and an angular (`add_angular_histogram_array`) only function. Their parameters are a subset of the general command; the dimension that is not sampled is fixed to a single bin (`0` to `1000` eV for the angular variant, one angle bin for the energy variant).

```
add_energy_histogram_array("species", E0, E1, nE, nX, nY,
                           xA, yA, zA, xB, yB, zB, xC, yC, zC);
add_angular_histogram_array("species", nTheta, nPhi, nX, nY,
                            xA, yA, zA, xB, yB, zB, xC, yC, zC);
```

Footnotes

1. This is mandatory, since otherwise the tangential vectors needed to compute the azimuthal angle are undefined ↩

2. The 'number' is required since different histogram types are possible for the same species and surface, which would lead to the same filename otherwise ↩

35 Electric boundary conditions

In PIC-MC plasma simulations, mesh objects act as electrodes which are either conductive or dielectric (= insulating). Conductive electrodes can be either put on a fixed potential, they can be floating or they can have a built-in potential *relative* to another electrode. This behaviour can be specified by the parameters **epsr, vf, vf_coupling** as summarized in the following table.

Note 1: Electrodes need to be closed 3D objects in the mesh geometry. If you need particle sources which are part of an electrode, this is currently not supported (in DSMC simulation, it is possible to have source surfaces which are only part of a complete 3D object). A workaround is to use membranes as particle sources, since membranes do not act as electrodes in plasma simulation.

Case	epsr	vf	vf_coupling
Dielectric floating potential (insulator domain)	Dielectric constant	commented out	commented out
Conductive floating potential (metal domain)	commented out	commented out	commented out
Absolute voltage (electrode, ground)	ignored	Voltage function (string)	commented out
Single voltage plus floating potential (single electrode capacitively coupled to ground potential)	ignored	Voltage function (string)	Name of current domain (string)
Dual voltage plus floating potential (dual electrode capacitively coupled to ground potential)	ignored	Voltage function (string)	Name of second domain (string)

A couple of pre-defined parameters and functions can be used in the voltage function string **vf**. RVM comes with a set of mathematical functions like sin, cos, atan,...,etc. Additionally the keywords TIME, VP and PI can be used within in the mathematical expression:

- **TIME** - The current simulation time, i.e. time step x current iteration.
- **PI** - You guessed it, pi (3.14159265).
- **VP** - Internally controlled voltage to meet aimed power dissipation 'PSOLL', which could be a time-dependent function or absolute an value.

In Python `epsr`, `vf` and `vf_coupling` are keywords of `Border`. `vf` takes a callable without arguments there instead of a formula string; it is evaluated at every time step and returns the voltage. `TIME` is available as `sim.get_time()`, `PI` as `math.pi` and `VP` as `controller.VP`, and the lambda has to close over `sim` and `controller`, so both must be in scope when the run starts. `vf_coupling` takes the key of the counter electrode in `BORDER_TYPES`, see the wall border type.

Example 1: System of three walls, "Target", "Chamber" and dielectric "Substrate"

```
block BORDER_TYPES: {
  Border Target: {
    icodec =  1;
    type   = "wall";
    T      = PAR.T0;
    reactions: {
      add_plain_reaction("Arplus", 1, 1, ["Ar", "e"], [1, PAR.SECONDARY_ELECTRON_YIELD]);
      set_emission_energy("e", 0, 5, 1);
      add_absorption("e", 0.5);   # 50% electron reflection
    };
    vf = "-VP";  # Negative voltage amplitude for DC operation
  };
```

```
  Border Chamber: {
    icodec =  2;
    type   = "wall";
    T      = PAR.T0;
    reactions: {
      add_absorption("Arplus", 1);  # We neglect secondary electrons at chamber and substrate
surface
      add_absorption("e", 0.5);
    };
    vf = "0";   # Chamber wall grounded and on anode potential
  };

  Border Substrate: {
    icodec =  3;
    type   = "wall";
    T      = PAR.T0;
    reactions: {
      add_absorption("Arplus", 1);
      add_absorption("e", 0.5);
    };
    epsr = 6.0;   # Dielectric constant of glass
    # vf = "";  vf must be commented out or deleted for a dielectric material!
  };
};
```

Example 2: The same system with floating conductive "Substrate" and with radio-frequency sputtering

```
block BORDER_TYPES: {
  Border Target: {
    icodec =  1;
    type   = "wall";
    T      = PAR.T0;
    reactions: {
      add_plain_reaction("Arplus", 1, 1, ["Ar", "e"], [1, PAR.SECONDARY_ELECTRON_YIELD]);
      set_emission_energy("e", 0, 5, 1);
      add_absorption("e", 0.5);   # 50% electron reflection
    };
    vf = "VP*cos(2*PI*13.56e6*TIME)";  # RF (13.56 MHz) voltage
    vf_coupling = "Target"; # Indicates that this electrode is floating (capacitively coupled)
against ground
  };

  Border Chamber: {
    icodec =  2;
    type   = "wall";
    T      = PAR.T0;
    reactions: {
      add_absorption("Arplus", 1);  # We neglect secondary electrons at chamber and substrate
surface
      add_absorption("e", 0.5);
    };
    vf = "0";   # Chamber wall grounded and on anode potential
  };

  Border Substrate: {
    icodec =  3;
    type   = "wall";
    T      = PAR.T0;
    reactions: {
      add_absorption("Arplus", 1);
      add_absorption("e", 0.5);
    };
    # Both, epsr and vf, must be commented out or deleted for a floating conductor
  };
```

```
};
```

Example 3: Two targets "Target_1", "Target_2" acting as bipolar electrodes with 100 kHZ

```
block BORDER_TYPES: {
  Border Target_1: {
    icodec =  1;
    type   = "wall";
    T      = PAR.T0;
    reactions: {
      add_plain_reaction("Arplus", 1, 1, ["Ar", "e"], [1, PAR.SECONDARY_ELECTRON_YIELD]);
      set_emission_energy("e", 0, 5, 1);
      add_absorption("e", 0.5);   # 50% electron reflection
    };
    # Target_1 is treated as floating conductor, i.e. no vf or epsr parameter declared!
  };

  Border Target_2: {
    icodec =  4;
    type   = "wall";
    T      = PAR.T0;
    reactions: {
      add_plain_reaction("Arplus", 1, 1, ["Ar", "e"], [1, PAR.SECONDARY_ELECTRON_YIELD]);
      set_emission_energy("e", 0, 5, 1);
      add_absorption("e", 0.5);   # 50% electron reflection
    };
    vf = "VP*cos(2*PI*100e3*TIME)";  # Oscillating voltage with 100 kHz frequency
    vf_coupling = "Target_1";        # Thus, both targets together are a floating electrode
                                     # system with built-in voltage difference
  };

  Border Chamber: {
    icodec =  2;
    type   = "wall";
    T      = PAR.T0;
    reactions: {
      add_absorption("Arplus", 1);  # We neglect secondary electrons at chamber and substrate
surface
      add_absorption("e", 0.5);
    };
    vf = "0";   # Chamber wall grounded and on anode potential
  };

  Border Substrate: {
    icodec =  3;
    type   = "wall";
    T      = PAR.T0;
    reactions: {
      add_absorption("Arplus", 1);
      add_absorption("e", 0.5);
    };
    # Both, epsr and vf, must be commented out or deleted for a floating conductor
  };
};
```

Power regulation

The keyword `VP` in a `vf` string is the voltage the regulator computes. It changes the electrode voltage until the power dissipated in the plasma reaches the setpoint, `PSOLL` in the parameter file and `PTARGET` in Python. The regulation is sampled every `PSAMPLE` seconds, `PSOLL` may be a time-dependent formula.

Two regulator versions exist in the parameter file, selected by `POWER_REGULATION_VERSION`:

value	behaviour
`0`	step limited regulator: moves `VP` by a bounded fraction per sampling interval, clamped to `VPMIN` and `VPMAX`. This is what runs when the key is missing, and the run log warns about it.
`1`	PID regulator with anti-windup.

The PID keys of version 1:

key	default	meaning
`POWER_REGULATION_FACTOR`	`1.5`	regulation strength, dimensionless
`POWER_REGULATION_SCALE`	`1e9`	superparticle weight times particle count; a value `<= 0` takes the current number of charged particles of the run
`POWER_REGULATION_INTEGRATION_TIME`	`1e-7` s	time constant of the I term
`POWER_REGULATION_DIFFERENTIATION_TIME`	`1e-6` s	time constant of the D term

With version 0 the keys `POWER_REGULATION_FACTOR` and `POWER_REGULATION_SCALE` are ignored; the log says so when they are set anyway.

The voltage limits apply to both versions: `VPMAX` (default 1000 V), `VPMIN` (default 1 V) and `VPSTART` (start voltage, a negative value means `VPMAX/2`).

```
WORKMODE = 1;
PSOLL = 10;
PSAMPLE = 1e-7;
POWER_REGULATION_VERSION = 1;
VPMAX = 1000;
```

The Python controller has no version switch: it always computes the PID form, but with different defaults. `RegulationFactor` is `1.0`, `IntegrationTime` is infinite and `DifferentiationTime` is `0`, so a controller created with `PTARGET` alone is a purely proportional regulator. The values in the example above reproduce the parameter file defaults of version 1. `Controller.VP` reads the voltage the regulator currently applies.

Both forms append the control loop to `powerloss.txt` next to the case: time, dissipated power and applied voltage at every sampling step. The Python interface adds the setpoint, the particle count and the P, I and D contributions as further columns.

DSMC examples

36 Overview

Type	Description	Setup files	Guide
3D - Argon gas flow simulation	Basic example with simple tube geometry	sample_v24.zip	Detailed setup guide
3D - Argon gas flow simulation	Simple tube geometry with optimized domain decomposition	tube-multiquad.zip	Guide for optimized domain decomposition
3D - Sputter transport	Ta target in a process module, flux from an `Arplus` absorption profile	—	Sputter transport

37 3D gas flow simulation

Geometric model

In this example, we analyze the gas flow through a cylindrical tube. The geometric model as shown in Fig. 1 contains the following elements:

- The cylindrical tube
- A buffer chamber at the left side of the tube for simulation of the gas inflow
- A buffer chamber at the right side of the tube for simulation of the gas outflow
- A gas inlet surface within the left buffer chamber
- A pumping surface within the right buffer chamber
- A box surrounding all above mentioned elements

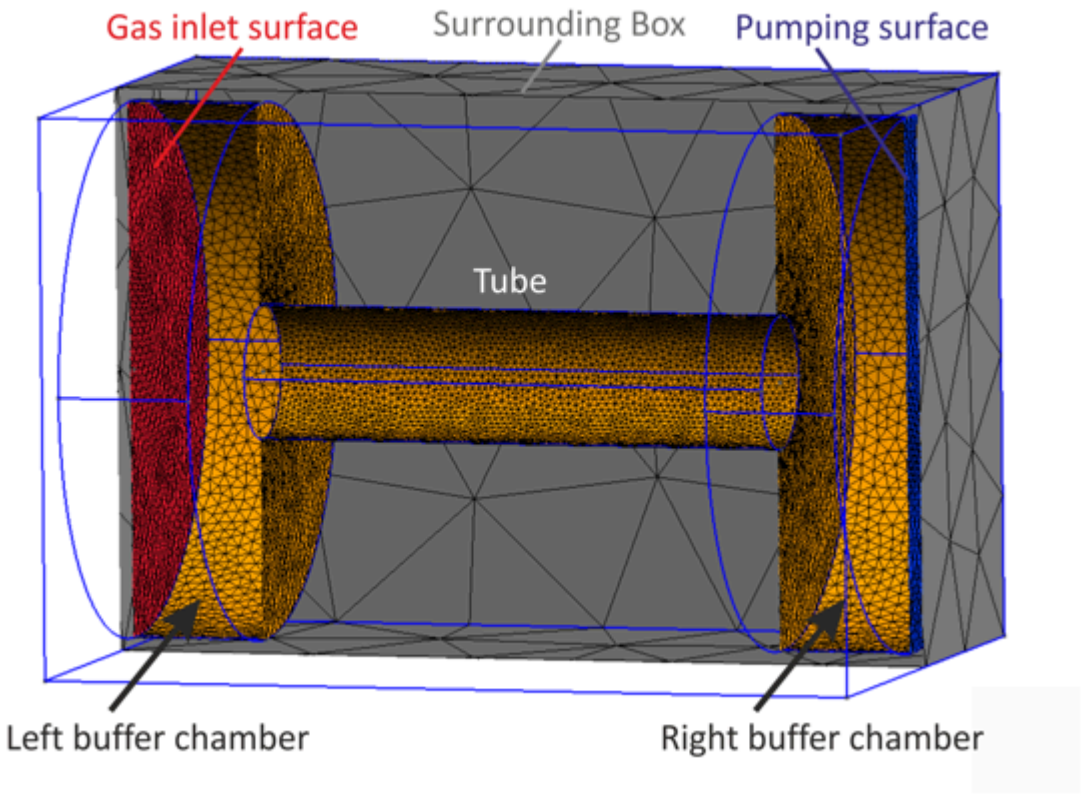


Figure 1: 3D Model (with cross-sectional cut) of a cylindrical tube with buffer chambers and surrounding box

General considerations about geometries for gas flow simulation

In order to create such a model, we use the open source meshing and postprocessing tool GMSH[1]. There are two possible ways of creating geometric model with GMSH: Either the model can be created manually by GMSH's own interactive mode and scripting language, or an IGS file (or STEP or other formats) originating from a CAD system can be imported. For our example, we use the manual way in order to point out the most relevant parts of a geometric model definition. Before starting to create the model, the following points are to be considered:

- At first, it is important to mention that we will create a **surface mesh** rather than a volume mesh. Within the DSMC/PIC-MC code, the surface mesh will be used to track collisions between particles

and chamber walls, while collisions between the particles will be treated within a cartesian volume grid, which is created independently from the surface mesh model.

- Second, each wall has to be represented by at least two surfaces (for front and back side). This is important for determining, whether a given point lies within a wall or within the free space. The algorithm which decides about that is pointed out in Fig. 2: From a given point in space, a virtual line is drawn along X axis[2], and the number of intersections between this line and any mesh surface is counted. For an even number of intersections, the point is considered to be within the gas flow volume, while for an uneven number of intersections, the point is considered to be within a solid wall.

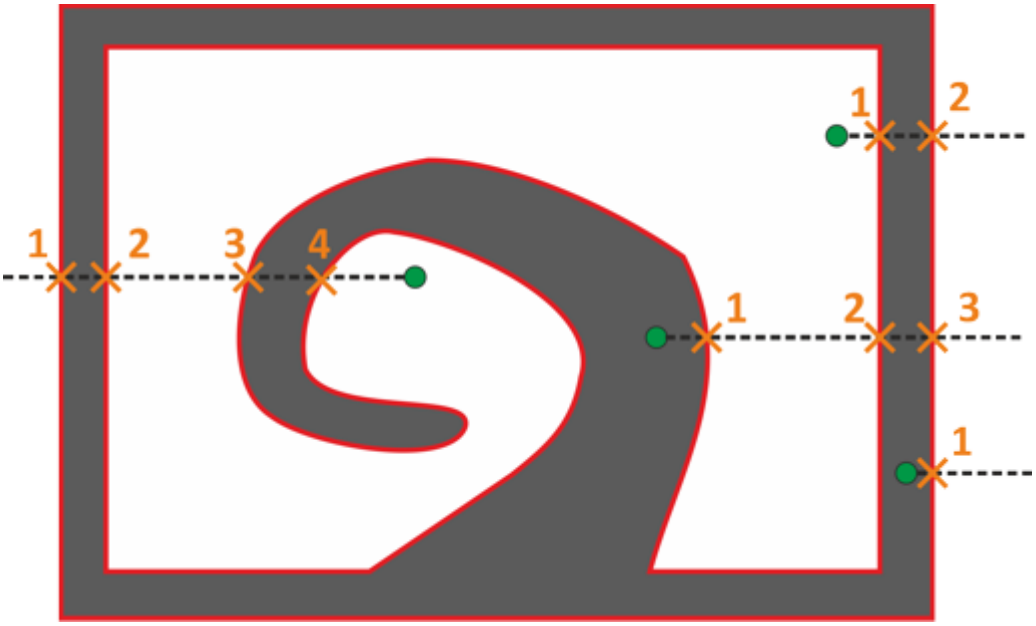


Figure 2: Algorithm for determining whether a point lies within a wall or within the free space

Definition of geometric points

Before creating the model, we need to think about some conventions about the coordinates, orientation, units etc. In this example, we stick to the following conventions:

- All length units are in mm
- The tube is oriented along X axis
- The coordinate origin (0,0,0) is located at the left end of the tube, centralized within the circular cross section

A geometric model always consists of points, lines, and surfaces. First, the geometric points have to be defined, second, the points are connected with lines and finally, surfaces are defined by line loops. For definition of the geometric points, we use the following GMSH script (click name to download):

```
LC = 10;  // characterictic mesh size

L_TUBE = 400;
R_TUBE =  50;

L_BUFFER = 100;
R_BUFFER = 200;

WALL = 10;

// Tube orientation along x axis
```

```
// Center point at left tube edge
Point(101) = {0, 0, 0, LC};

// Four points on tube radius separated by 90°
Point(102) = {0,  R_TUBE, 0,       LC};
Point(103) = {0,  0,      R_TUBE,  LC};
Point(104) = {0, -R_TUBE, 0,       LC};
Point(105) = {0,  0,      -R_TUBE, LC};

// The same is repeated for the right tube edge
Point(106) = {L_TUBE,  0,      0,       LC};
Point(107) = {L_TUBE,  R_TUBE, 0,       LC};
Point(108) = {L_TUBE,  0,      R_TUBE,  LC};
Point(109) = {L_TUBE, -R_TUBE, 0,       LC};
Point(110) = {L_TUBE,  0,      -R_TUBE, LC};

// Additionally, we need buffer chambers at left and right side of tube
// Left buffer chamber:
Point(201) = {-L_BUFFER,  0,         0, LC};
Point(202) = {-L_BUFFER,  R_BUFFER,  0, LC};
Point(203) = {-L_BUFFER,  0,         R_BUFFER, LC};
Point(204) = {-L_BUFFER, -R_BUFFER,  0, LC};
Point(205) = {-L_BUFFER,  0,        -R_BUFFER, LC};
Point(206) = {0,  R_BUFFER,  0, LC};
Point(207) = {0,  0,         R_BUFFER, LC};
Point(208) = {0, -R_BUFFER,  0, LC};
Point(209) = {0,  0,        -R_BUFFER, LC};

// Right buffer chamber:
Point(302) = {L_TUBE,           R_BUFFER,  0, LC};
Point(303) = {L_TUBE,           0,         R_BUFFER, LC};
Point(304) = {L_TUBE,          -R_BUFFER,  0, LC};
Point(305) = {L_TUBE,           0,        -R_BUFFER, LC};
Point(301) = {L_TUBE+L_BUFFER,  0,         0, LC};
Point(306) = {L_TUBE+L_BUFFER,  R_BUFFER,  0, LC};
Point(307) = {L_TUBE+L_BUFFER,  0,         R_BUFFER, LC};
Point(308) = {L_TUBE+L_BUFFER, -R_BUFFER,  0, LC};
Point(309) = {L_TUBE+L_BUFFER,  0,        -R_BUFFER, LC};

// Bounding box around the whole setup

x0 = -L_BUFFER-WALL;
x1 =  L_BUFFER+L_TUBE+WALL;
y0 = -R_BUFFER-WALL;
y1 = R_BUFFER+WALL;
z0 = -R_BUFFER-WALL;
z1 = R_BUFFER+WALL;

Point(401) = {x0, y0, z0, LC*5};
Point(402) = {x1, y0, z0, LC*5};
Point(403) = {x0, y1, z0, LC*5};
Point(404) = {x1, y1, z0, LC*5};
Point(405) = {x0, y0, z1, LC*5};
Point(406) = {x1, y0, z1, LC*5};
Point(407) = {x0, y1, z1, LC*5};
Point(408) = {x1, y1, z1, LC*5};
```

A few comments on this code are given in the following:

- In the first line, a variable `LC` is defined, which is subsequently used as fourth argument for each `Point`. This is a **characteristic length** and defines the average mesh size around the points.

- Further characteristics dimensions such as the tube length (`L_TUBE`), its radius etc. are also defined as variables, which are subsequently being used in the point coordinates. This simplifies to create variations of the geometric model with different tube lengths etc. later on.
- The syntax of a point definition in GMSH is `Point(number)={x, y, z, lc};` The point number must be unique within the whole GMSH model. It is not required that the set of numbers is continuous. It is convenient to assign the same major number to points which belong to a certain part of the geometry, e. g. 1XX for the tube, 2XX for the left buffer chamber etc.
- Wherever a coordinate is required in GMSH it is possible to make use of a built-in formula parser.
- If you copy the above script into an ASCII file (named e. g. 'tube3d.geo') and open this file with GMSH, you will get a picture similar as in Fig. 3.

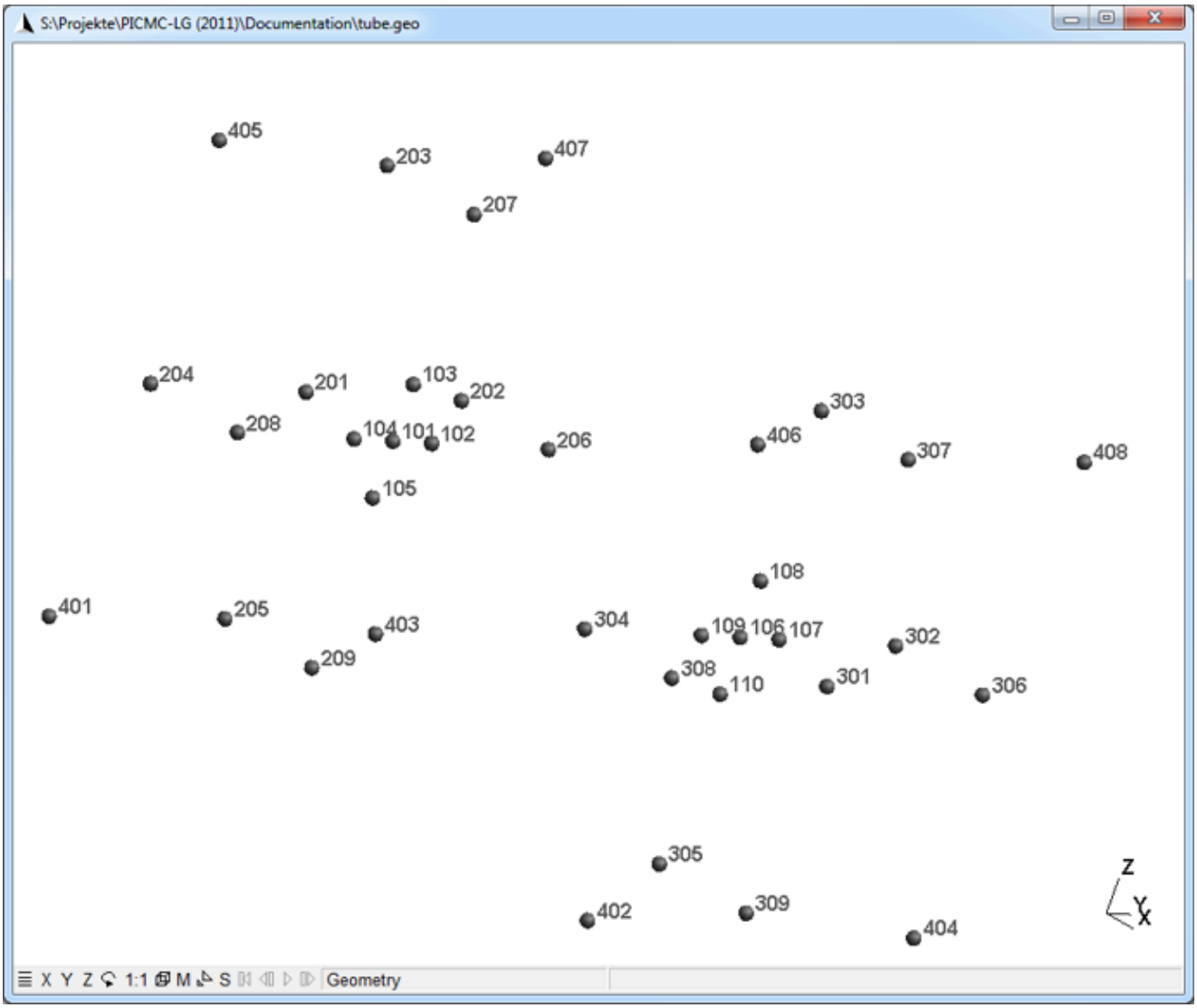


Figure 3: GMSH view of the points defined in the above script (with Point numbers turned on)

3

Definition of straight lines and circle arcs

The next step is to connect all points with straight or circular lines. This can be either done by further script commands or by using GMSH interactively. We will follow the interactive way. First, use the GMSH menu sequence shown in Fig. 4 in order to select "circle arcs".

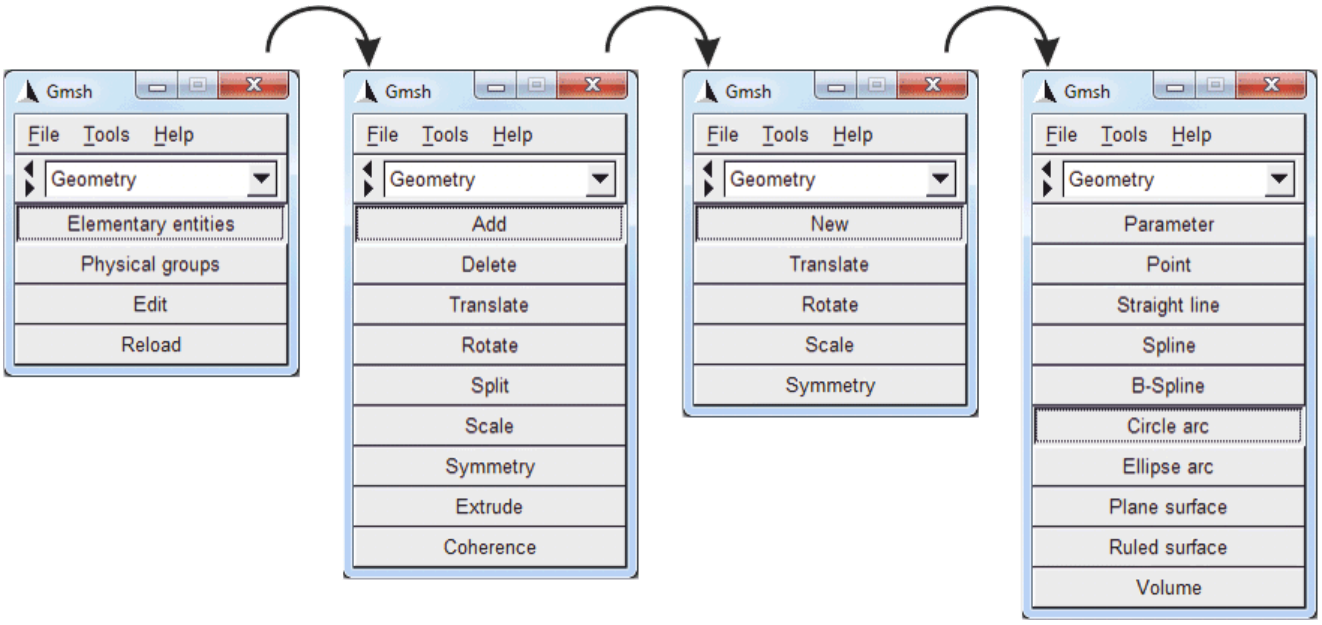


Figure 4: GMSH menu sequence for selecting circle arcs

A circle arc is now created by clicking on three points subsequently: (i) A point on the circle as start point, (ii) the center of the circle and (iii) another point on the circle as end point. The left circular entrance of the tube is e. g. defined by clicking on points no. 102-->101-->103; 103-->101-->104; 104-->101-->105; 105-->101-->102. The right circular entrance of the tube is defined analogously (107-->106-->108; 108-->106-->109; 109-->106-->110; 110-->106-->107).

After having defined both circular surfaces, please click on "Straight line" in the GMSH menu and connect the following points: 102-->107; 103-->108; 104-->109; 105-->110. The interactive GMSH graph should now look as in Fig. 5.

While interactively defining lines between the points, GMSH automatically appends accordant script lines to the geometry script. If you look at the ASCII file `tube2d.geo`, you will find the following lines automatically appended at its end[4].

```
Circle(1) = {102, 101, 103};
Circle(2) = {103, 101, 104};
Circle(3) = {104, 101, 105};
Circle(4) = {105, 101, 102};
Circle(5) = {108, 106, 109};
Circle(6) = {109, 106, 110};
Circle(7) = {110, 106, 107};
Circle(8) = {107, 106, 108};
Line(9) = {102, 107};
Line(10) = {103, 108};
Line(11) = {104, 109};
Line(12) = {105, 110};
```

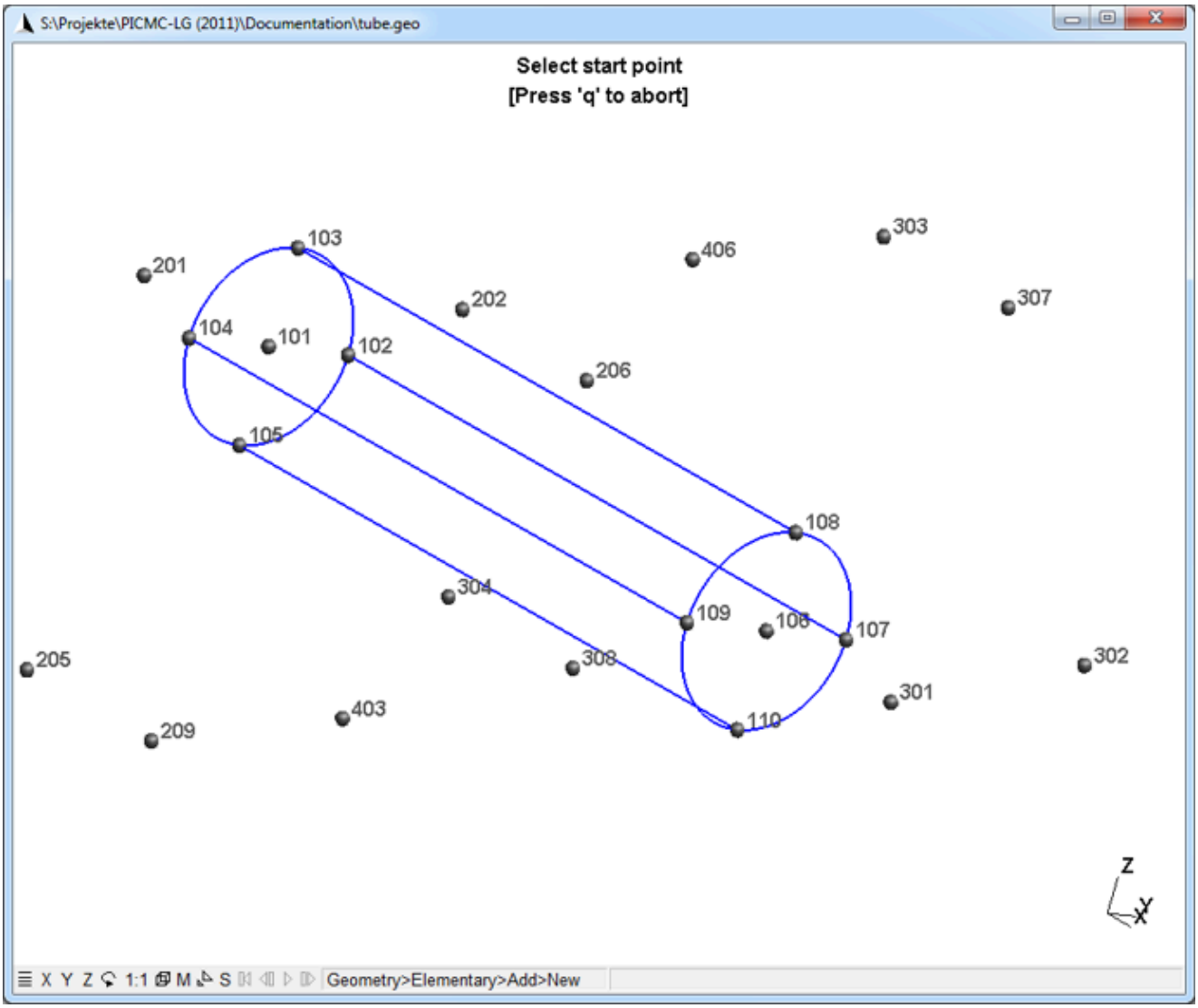


Figure 5: GMSH view of the points and the lines / circle arcs related to the inner tube

By continuing in the same manner, all other circle arcs and straight lines are to be defined. At the end, each point will be part of at least one line or circle arc, and the geometry will look like in Fig. 6. At this stage, please press the "q" button in order to leave the line selection mode. For your convenience, a zipped file of this stage (i. e. all points and lines but no surfaces) is provided for download.

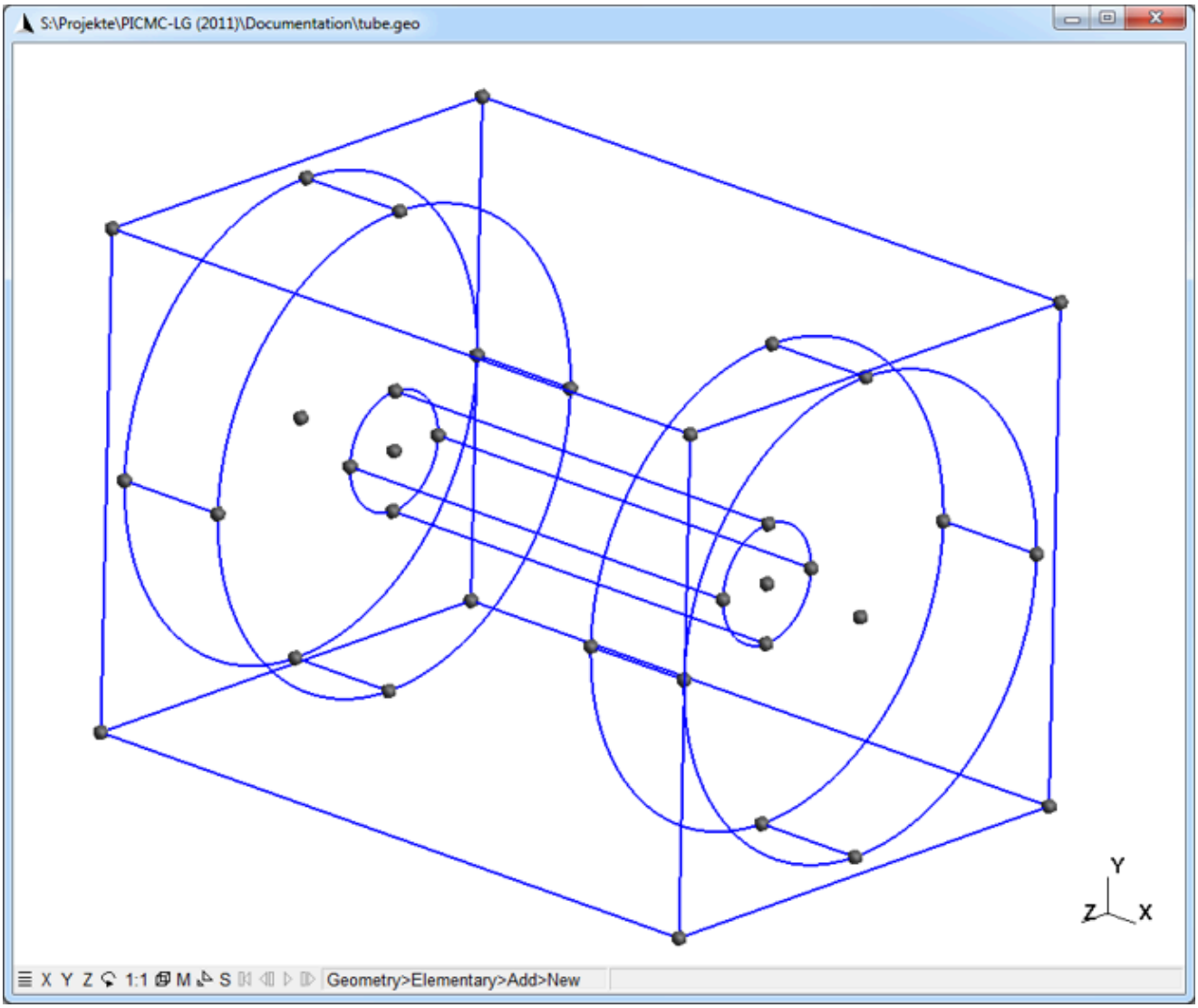


Figure 6: GMSH view of the tube model with all points and lines / circle arcs.

Definition of ruled and plain surfaces

The next step will be to define the surfaces. In this case, we have to distinguish between

- Plane surfaces and
- Ruled surfaces.

Ruled surfaces are required for the cylindrical walls of the tube and buffer chambers. By the straight lines, the cyclindrical wall of the tube is divided into four parts. Each part has to be selected, individually.

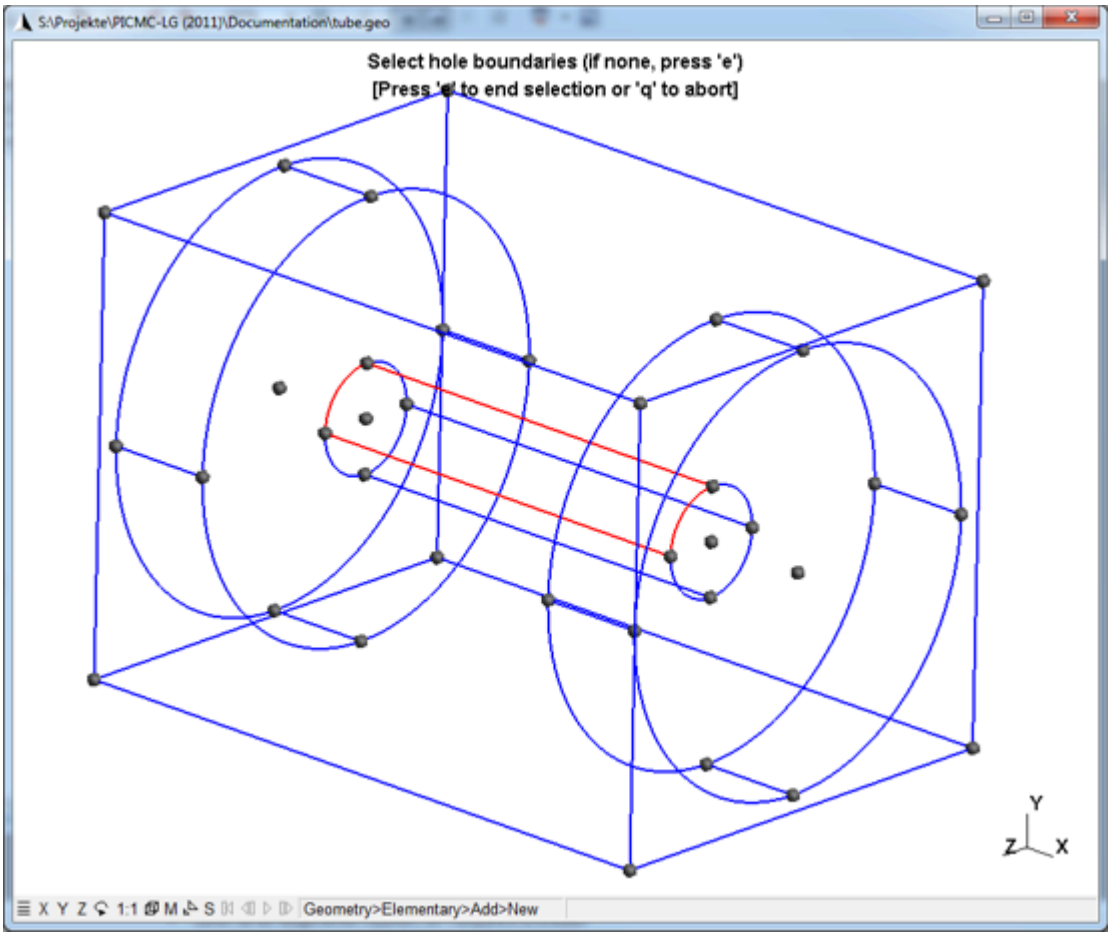


Figure 7: GMSH view of the tube model, where one of the four ruled surfaces of the tube is being selected.

In order to start building surfaces, please select the entry "Ruled Surface" within the GMSH menu shown in Fig. 4. Subsequently, select one of the four possible line loops on the cylindrical wall as shown in Fig. 7[5]. If you mis-clicked a line, pressing on "u" will undo the last selection. When all appropriate lines are highlighted as shown in Fig. 7, press "e" in order to confirm the selection.

This process has to be repeated for all four partial surfaces of the tube as well as of both buffer chambers. In total, twelfe "Ruled Surfaces" are required. After having defined all ruled surfaces, please press "2" in order to obtain a two-dimensional surface mesh, which will yield a picture similar as in Fig. 8.

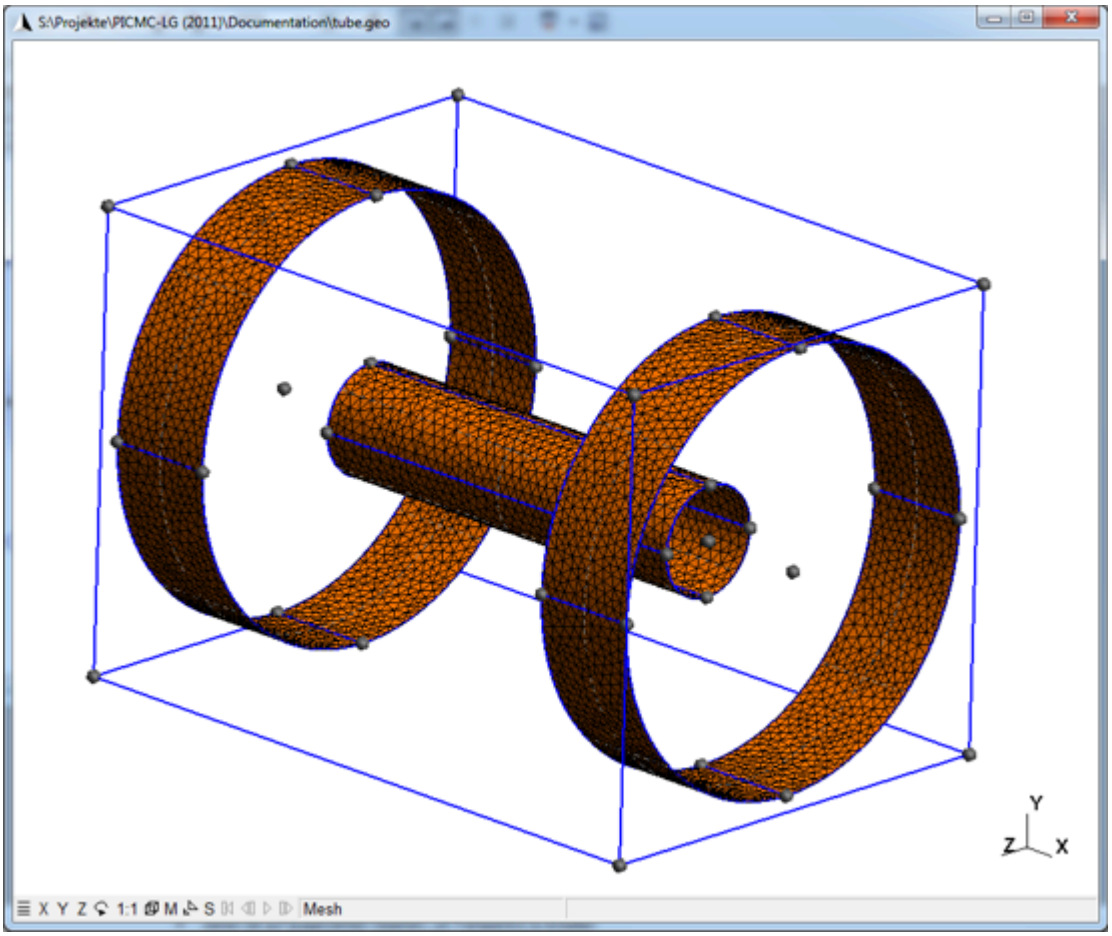


Figure 8: GMSH view of the tube model, where all ruled surfaces are defined and meshed.

Finally, the remaining plane surfaces should be defined. These are the plane surfaces of the surrounding box as well as front and back surfaces of the buffer chambers. **Important:** There should be no surfaces defined for both tube entrances, since otherwise the gas flow will be blocked. Instead, the adjacent surface of the buffer chamber is to be defined **with the tube entrance as hole boundary**. This is accomplished by first selecting the outer line loop of the large buffer chamber and then selecting the inner line loop of the tube entrance and finally pressing on "e". Fig. 9 shows a possible way of doing this. Please proceed with all remaining plain surfaces in order to come to the final step of defining physical surface groups. The attached ZIP file contains the geometry model with all surfaces but without any physical surface group.

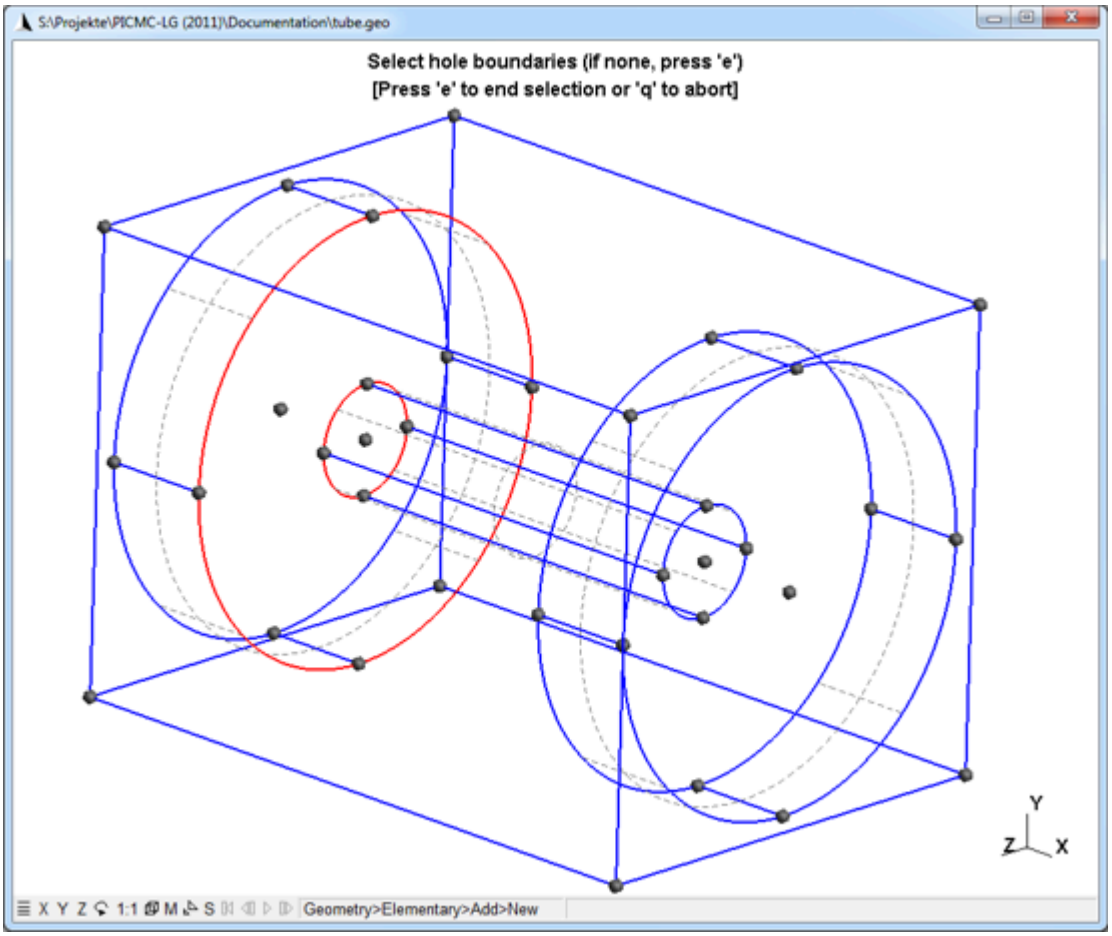


Figure 9: Selection of a surface with a "hole" in GMSH.

Physical surface groups

The surfaces of our tube model are supposed to have different physical functions. While most of the surfaces are just acting as walls, where particles are being diffusively reflected, there is also an inlet and a pumping surface. Thus, we expect to define three different physical surface groups:

- Wall --> All surfaces except from inlet and outlet surfaces
- Inlet --> The left surface of the left buffer chamber
- Pump --> The right surface of the right buffer chamber

In order to select physical surface groups in GMSH, please use the menu sequence as shown in Fig. 10.

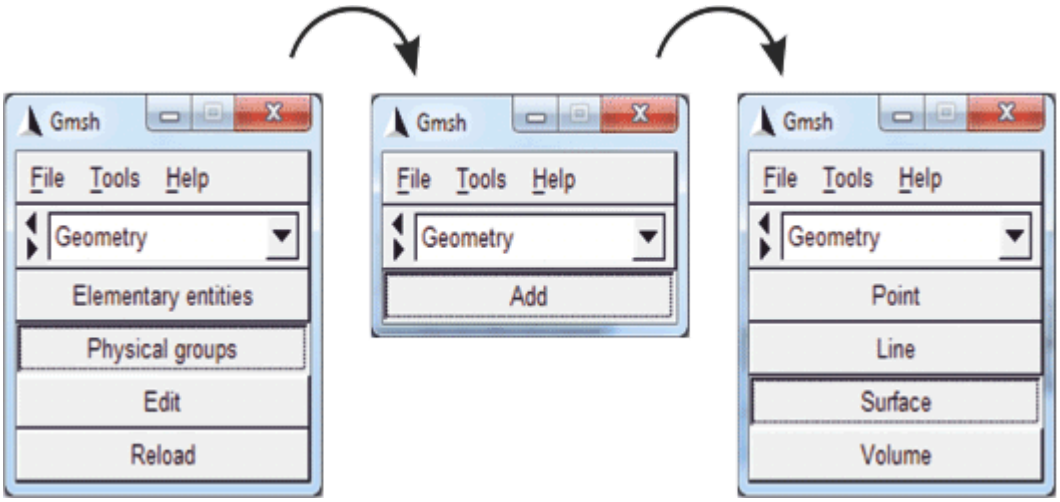


Figure 10: Menu sequence in GMSH for definition of physical surface groups.

An appropriate strategy is to first select **all** surfaces in the model in order to form the physical surface group for "Wall". Afterwards, the "Inlet" and the "Pumping" surface should be individually selected. In order to select **all** surfaces, the following steps are required:

- With the center mousebutton, select an appropriate zoom in the GMSH window, where the whole geometry becomes visible.
- Hold down the "Ctrl" key and click once at a location in the upper left from the geometry.
- Release the "Ctrl" key and move the mouse cursor to the lower right. A dotted frame will appear, when this frame encloses the whole geometry, click on the mouse button the second time (see also Fig. 11 below).
- All surfaces will appear in red now, if this is the case, press "e" to confirm the selection.

Afterwards, please select the left plane surface of the left buffer chamber as well as the right surface of the right buffer chamber individually. After having completed the physical surface selection, GMSH has added three more lines to the geometry script, which look similar as follows:

```
Physical Surface(95) = {60, 92, 62, 52, 54, 72, 77, 86, 50, 58, 64, 56, 70, 75, 84, 88, 80, 66, 68,
82, 90, 94};
Physical Surface(96) = {77};
Physical Surface(97) = {82};
```

Here, we have to apply some changes manually. First, we would rather use strings than numbers in order to identify the physical surfaces. This can be done by replacing the numbers in round brackets by quoted strings as follows:

```
Physical Surface("Wall")  = {60, 92, 62, 52, 54, 72, 77, 86, 50, 58, 64, 56, 70, 75, 84, 88, 80,
66, 68, 82, 90, 94};
Physical Surface("Inlet") = {77};
Physical Surface("Pump")  = {82};
```

Since, we have selected **all** surfaces for the first physical surface group "Wall", the surfaces number 77 and 82 are now selected twice. These ambiguities must be removed; thus, the duplicate surface numbers (in our case 77 and 82) should be removed from the first physical surface group "Wall". As a final result, the last three lines of the GMSH file should now look as in the following:

```
Physical Surface("Wall")  = {60, 92, 62, 52, 54, 72, 86, 50, 58, 64, 56, 70, 75, 84, 88, 80, 66,
68, 90, 94};
Physical Surface("Inlet") = {77};
```

```
Physical Surface("Pump")  = {82};
```

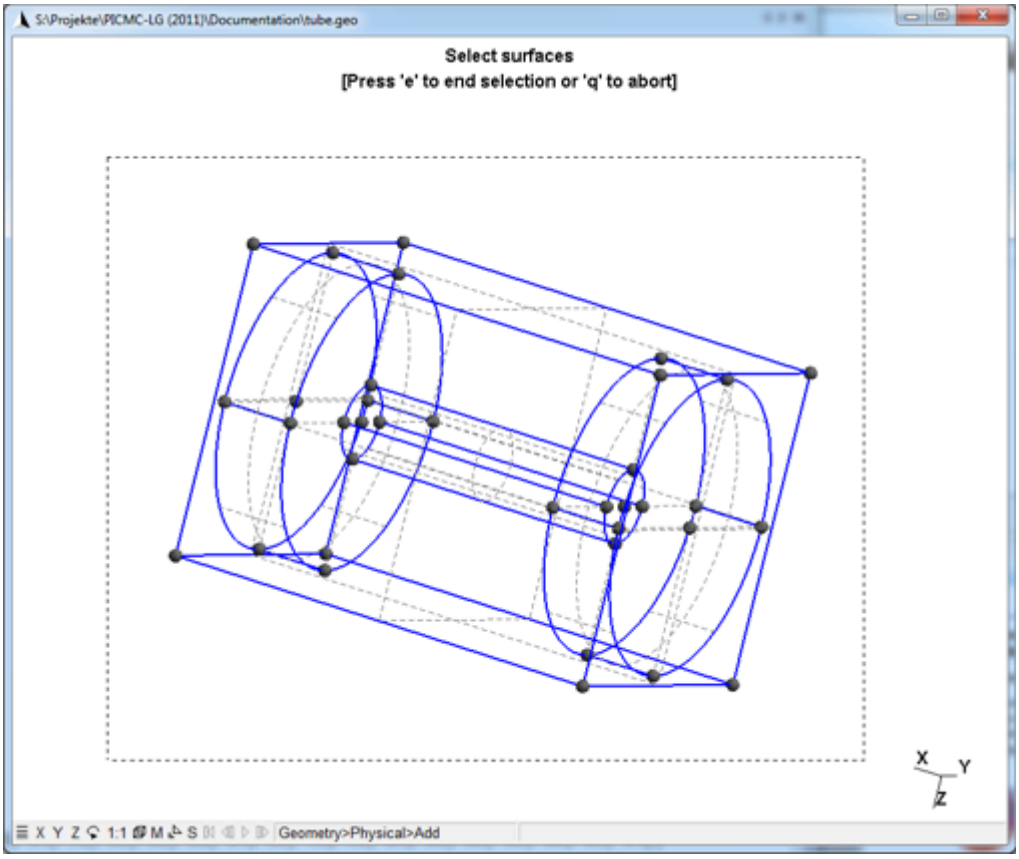


Figure 11: GMSH interactive window while selecting all surfaces as physical group with a lasso.

Final step: Saving the mesh file

As a final step, a 2D surface mesh is created by pressing the "2" button. Afterwards, the mesh file can be saved by clicking on the "save" button of the GMSH main dialog. A complete version of the tube model can be found as "tube.geo" within this ZIP file.

Setting up the parameter file

After having created the GMSH mesh file, which will be named `tube.msh` here, the next step is to copy this file to the Linux cluster. In this example, it is copied into `/home/ap/sim/Tube`. In order to create an appropriate parameter file, use the command `initpicmc tube`. The following will appear on the terminal:

```
ap@sim037:~/sim/Tube$ initpicmc tube
***** scr/init.r, 15642c0 *****
***** tube, 15642c0 *****
** Assuming NON-STEPFILE mode
** The mesh has 9978 nodes and 3 codecs
** The mesh dimensions are {-110, -210, -210} --> {510, 210, 210}
** Accordant to the mesh geomtry a parameter file tube.par has been created.
** Please edit the parameter file and invoke: rvmmpi -picmc [NumTasks] tube.par
Physical strings:
  1 <--> 'Wall'
  2 <--> 'Inlet'
  3 <--> 'Pump'
***** tube, 15642c0 *****
```

```
ap@sim037:~/sim/Tube$ _
```

The output comprises the following informations:

- The initpicmc script recognized, that all mesh elements are contained within a box between (-110,-210,-210)-->(510,210,210). This volume will be set as simulation volume ("bounding box") within the parameter file.
- Three physical surface groups are identified.
- A parameter file "tube.par" has been created.

The parameter file is parsed by "RIG-VM", which is a script interpreter developed at Fraunhofer IST. This allows to use variables and formula for definition of numerical values. Comments can be placed within the parameter file by using the "#" character. The different sections of the parameter file are discussed in detail here.

Global control variables

The first section of the parameter file comprises some global variables for the simulation run and its runtime behavior. It should be edited like follows:

```
DT       = 1e-5;     # Time step width
TSIM     = 1.0;      # Total simulation time in seconds
TAVG     = 1;        # Time averaging mode (0 = off, 1 = on)
NPLOTS   = 10;       # Number of plots
NDUMPS   = 1;        # Number of particle data dumps
WORKMODE = 0;        # Work mode (0 = DSCM gasflow simuation, 1 = PICMC plasma simulation, 2 =
electric field solver)
T0       = 300;      # Initial particle and border temperature [K]
PSOLL    = 1;        # Aimed power dissipation in Watt (plasma simulation only)
PSAMPLE  = 1e-7;     # Sampling time for power dissipation (plasma simulation only)
#VPMAX    = 1000;    # Maximum voltage for power control (default = 1000 volt)
```

Gas species and cell resolution

The next lines within the parameter file deal with the selection of gas species, their initial distribution and their statistical weight. In our test case, we will define a gas inlet operating with Argon at a pressure of 1.0 Pa and a pumping surface. This means, that the pressure will reach 1 Pa close to the gas inlet but will be significantly lower in the major part of the simulation volume. For Argon with a pressure of 1 Pa, the mean free path is approx. 6 mm, thus a cell dimension of about 0.5 mm would be required close to the gas inlet. Since we assume that the pressure in the tube will be significantly lower, we will start with a homogeneous pressure distribution of 0.5 Pa and use 1.0 cm as appropriate cell dimension.

In section statistical constraints, for Argon with a pressure of 1.0 Pa, a cell dimension of 5 mm and a value of `NREAL_Ar=3e12;` is suggested, which leads to an average of 10 simulation particles per cell. To transfer this to the case with a pressure 0.5 Pa with 10 mm cell dimension, we need to keep in mind that `NREAL_Ar` scales reciprocally with the square of the pressure[6]. Accordingly, the resulting value for `NREAL_Ar` is 1.2 x 10^{13}. Thus, the lines within the parameter file should be edited as follows:

```
SPECIES  = ["Ar"];   # List of species strings
NREAL_Ar = 1.2e13;   # Default scale factor for Ar
P0_Ar    = 0.5;      # Default pressure of initial Ar distribution [Pa]
```

In the next lines of the parameter file, the desired output of the simulation run is specified:

```
NUMBER      = ["Ar"];
PRESSURE    = ["Ar"];
VELOCITY    = ["Ar"];
FIELD       = ["PHI"];
```

The next section of the parameter file deals with the **bounding box**, i.e. the simulation volume and its cell resolution. First, we have to define the length unit, which is used within the mesh file. Our geometry model uses **mm** as length unit, while the default setting in the parameter file is **m**. Thus, it is required to change the length `UNIT` to 0.001.

Next, the bounding box coordinates are shown within the array `BB`. It is possible to edit the `BB` array in order to restrict the simulation volume to a fraction of the mesh geometry, but we will use the full volume, here. The six numbers define two points in 3D with lower and upper coordinates. In our case, the bounding box spans between (-110,-210,-210)-->(510,210,210). Accordingly, the length of the bounding box is 62 cm in X direction and 42 cm in Y and Z direction. With the cell dimension of 10 mm as discussed above, we would like to have 62 cells along X direction and 42 cells in Y and Z direction, leading to a total of 109368 cells.

The cell resolution and segmentation of the simulation volume is defined in the arrays `NX, NY, NZ`. The segmentation is important with respect to parallel processing: Each segment of the simulation volume can be assigned to an indidividual CPU core. By this way, the number of segments limits the maximal number of parallel processes[7]. In the default settings, the simulation volume is divided into two segments along each axis, and each segment has a resolution of 10 grid cells. Given our model dimension, this seems not appropriate. For our purpose, we should change these lines to:

```
UNIT = 0.001;          # Unit of mesh geometry (1.0 [m], 0.001 [mm], 0.0254 [inch], ...)
BB   = [-110, -210, -210, 510, 210, 210]; # Bounding box dimensions
NX   = [62];     # Cell resolution of subvolumes in x-direction
NY   = [14, 14, 14];     # Cell resolution of subvolumes in y-direction
NZ   = [14, 14, 14];     # Cell resolution of subvolumes in z-direction
```

By these settings, we divide the volume into three segments in Y and Z direction, while only one segment is used in X direction (see Fig. 12). The total number of segments is 3x3 = 9; thus, a maximal number of 9 parallel PIC-MC processes can be used for this simulation model. The numbers given for each segment denote the number of cells. In Y and Z direction we have the 42 cells divided into three segments, i.e. 14 cells per segment, while in X direction we use 62 cells for one segment.

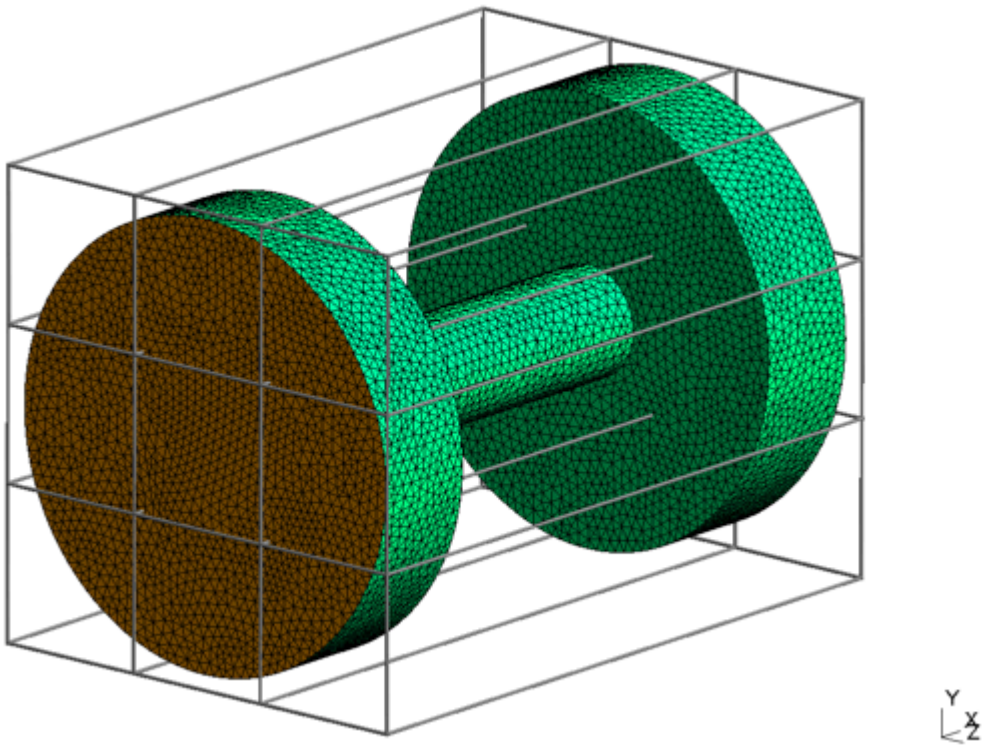


Figure 12: Possible segmentation of the gas flow model into 9 segments.

The selection of species, their statistical weights and the definition of the cell resolution is the most crucial part in the parameter file. The remaining steps are comparably easy.

Simulation domain - bounding box

The simulation domain has the shape of a cuboid, which may be divided into segments and sub-cells. It is required to specify the boundary conditions in case, if a simulation particle reaches a cuboid surface. Possible boundary conditions are **outlet**,**specular** and **periodic**. We have to distinguish between the following cases:

- In 3D models, usually the gas volume is contained into a closed mesh geometry. In this case, no particle should reach the cuboid boundary. If the mesh geometry is not rectangular, there may be particles originating from an initial homogeneous distribution, which are *outside* the gas volume. In all such cases the most appropriate boundary condition is **outlet**, i. e. a particle reaching the cuboid boundary will be deleted. The respective lines in the parameter file should look like follows:

```
TYPE_x1 = "outlet";
    TYPE_x2 = "outlet";
    TYPE_y1 = "outlet";
    TYPE_y2 = "outlet";
    TYPE_z1 = "outlet";
    TYPE_z2 = "outlet";
```

- In some cases, there may be 3D models with a mirror symmetry plane. In such cases it is possible to reduce computation time by factor of two by placing one of the cuboid boundaries at the plane of symmetry. The appropriate boundary condition for such plane is **specular**, i. e. a particle hitting this surface will be specularly reflected. An example, where the cuboid boundary at lower Y coordinates defines a symmetry plane, is given as follows:

```
TYPE_x1 = "outlet";
    TYPE_x2 = "outlet";
    TYPE_y1 = "specular";
    TYPE_y2 = "outlet";
    TYPE_z1 = "outlet";
    TYPE_z2 = "outlet";
```

- In 2D models, there are only mesh geometry elements for boundaries within e. g. the XY plane, while the geometry remains open in Z direction. In this case, both cuboid boundaries perpendicular to the Z axis should be either **specular** or **periodic**[8].
- In 1D models, four of the six boundaries must be set on **periodic** or **specular**. This may look like follows:

```
TYPE_x1 = "periodic";
    TYPE_x2 = "periodic";
    TYPE_y1 = "periodic";
    TYPE_y2 = "periodic";
    TYPE_z1 = "outlet";
    TYPE_z2 = "outlet";
```

Surface functions

Finally, the geometric mesh file is to be specified, and the functions of the respective physical surfaces must be defined. The specification of the mesh file's filename follows just after the boundary conditions of the bounding box:

```
# ================================================
# Specify mesh files and boundary conditions
# Available border types are: "wall" and  "membrane"
# ================================================
MESHFILE = "sample.msh"; # Name of mesh file for PICMC particle simulation
BEMMESH  = ""; # Name of mesh file for BEM b-Field solver
```

Here, only for `MESHFILE` a filename must be specified; `BEMMESH` is only used for computation of the magnetic field in PVD plasma simulation setups and can be left empty, here.

In the following parameter file section, the functions of the respective physical surfaces can be defined. A detailed description about the possibilities here is given in section Wall reaction model. In our case we just need a particle source at surface `Inlet` and an absorption coefficient at surface `Pump`. The surface `Wall` is by default diffusely reflective at a temperature `T0` and can be left unchanged. With these two modifications, the wall function definition block would look like follows:

```
block BORDER_TYPES: {
  Border Wall = {
    icodec =   1;                       # Codec number from meshfile
    type = "wall";                      # Alternative type: "membrane"
    T = PAR.T0;                         # Wall temperature
    #epsr        = 1.0;                         # Rel. dielectric permittivity (insulator only)
    #vf          = "VP*cos(2*PI*13.56e6*TIME)"; # Voltage function (conductor only)
    #vf_coupling = "";                          # Border name of counter electrode for potential
coupling
    reactions: {
      # Add reactions here - please check SIM-Wiki for available reaction schemes!
  };

  Border Inlet = {
    icodec =   2;                       # Codec number from meshfile
```

```
    type = "wall";                      # Alternative type: "membrane"
    T = PAR.T0;                         # Wall temperature
    # epsr = 1.0;                       # Rel. dielectric permittivity (insulator only)
    # vf = "VP*cos(2*PI*13.56e6*TIME)"; # Voltage function (conductor only)
    # vf_coupling = "";                 # Border name of counter electrode for potential coupling
    reactions: {
      add_source("Ar", 1.0, "Pa");
    };
  };

  Border Pump = {
    icodec =   3;                       # Codec number from meshfile
    type = "wall";                      # Alternative type: "membrane"
    T = PAR.T0;                         # Wall temperature
    # epsr = 1.0;                       # Rel. dielectric permittivity (insulator only)
    # vf = "VP*cos(2*PI*13.56e6*TIME)"; # Voltage function (conductor only)
    # vf_coupling = "";                 # Border name of counter electrode for potential coupling
    reactions: {
      add_absorption("Ar", 0.25);
    };
  };
};
```

The completed simulation case including GEO and PAR file can be downloaded here.

Footnotes

1. For more information about GMSH See this link ↩
2. Depending on the position of the point with respect to the edges of the simulation volume, the line is either drawn in positive or negative X direction. ↩
3. To do so, open the Tools->Options dialog, click on "Geometry" at the left side and chose the "Visibility" ribbon ↩
4. If the file is still open in an editor, you may need to use a refresh function e.g. by pressing F5 in order to see the appended lines. It is **not** recommended to use the Windows notepad editor, instead a more functional editor such as Programmer's notepad is recommended, which will automatically notice any changes externally applied to the text file ↩
5. This is accomplished by clicking on the lines rather than on points ↩
6. Provided that the cell dimension is scaled reciprocally with pressure. ↩
7. However it is always possible to have less parallel processes than segments. In this case, some of the PIC-MC processes have to handle multiple segments ↩
8. This makes no difference for DSMC gas flow simulation; for PIC-MC plasma simulation we recommend to use rather **periodic** ↩

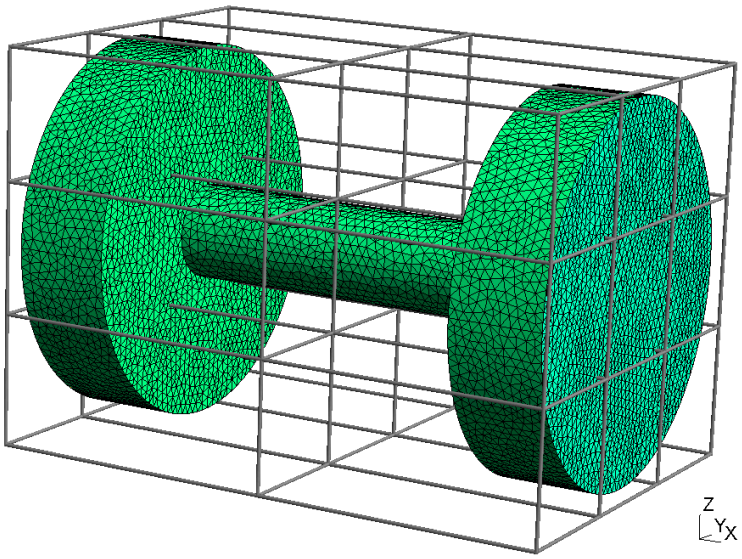
38 Optimized domain decomposition

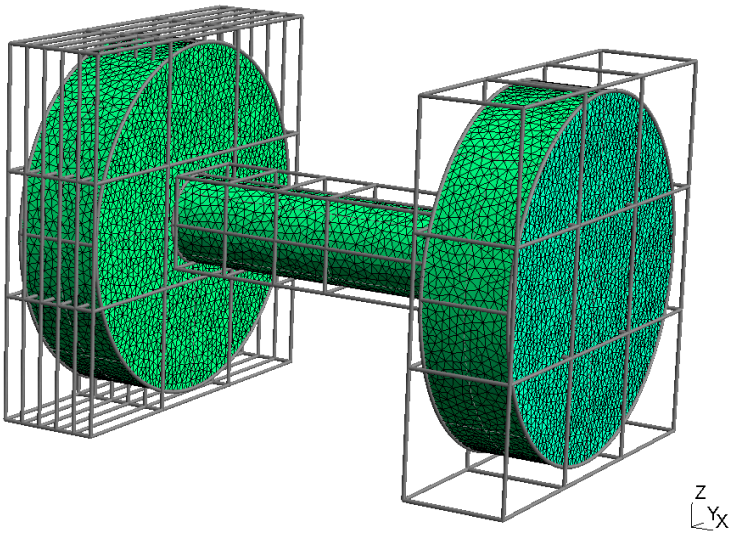
While the concept of embedding the whole mesh geometry into a single "bounding box" is simple and user-friendly, this method is not always efficient. In some cases, only a small fraction of the simulation volume will be actually used; thus the simulation model will consists of a majority of empty cells. In other cases, there may be big pressure differences within the simulation volume, which makes the definition of strongly varying cell dimensions necessary. With one single bounding box, this is not always possible in a flexible manner.

For this purpose, it is possible since picmc version 2.3.4 to use multiple joined rectangular bounding boxes as simulation volume:

- The bounding boxes can be defined in the parameter file with ''block MQUAD: { ... }; ''
- The connections between the bounding boxes are defined within ''block MJOIN: { ... }; ''
- If these blocks are omitted (or commented out), picmc will use the previous method of a large single bounding box.
- Due to limitations of the electric field solver with respect to multi-gridding schemes, the MQUAD / MJOIN feature is currently **only available for DSMC simulation** (WORKMODE=0 in the parameter file).

An example of the usage of multiple bounding boxes is shown in the following picture; it is based on the same tube geometry as in the DSMC example.





Single box: Most of the simulation volume is wasted.	Multiple boxes: Efficient usage of simulation volume.

In the following, it will be shown how to realize the multiple boxes arrangement in the parameter file.

Tube example with four bounding boxes

In the tube example of the previous section, we have an inlet and an outlet chamber which are connected by a long tube. With one single bounding box, the simulation volume has a much larger cross section than the tube diameter resulting in a large number of wasted simulation cells. Thus, we would like to divide the simulation volume into the following bounding boxes,

- one for the inlet region,
- one for the left side of the tube with higher pressure,
- one for the right side of the tube with lower pressure,
- one for the outlet region.

The GEO and PAR file of the tube example with multiquad approach can be downloaded here as ZIP file.

Definition of multiple bounding boxes in block MQUAD

In order to define that in the parameter file, a `block MQUAD` must be defined. Within this block, all desired bounding boxes have to be defined as shown in the following:

```
# ... geometry definition within parameter file

UNIT = 0.001; # Unit of mesh geometry (1.0 [m], 0.001 [mm], 0.0254 [inch], ...)

block MQUAD: {

  block Left: {
    DIV_X = [23, 23, 23, 23, 23]; NX = [4, 4, 4, 4, 4];
    DIV_Y = [140, 140, 140];      NY = [28, 28, 28];
    DIV_Z = [140, 140, 140];      NZ = [28, 28, 28];
    X0 = -110; Y0 = -210; Z0 = -210;
    NREAL_SCALE=0.5;
  };

  block Tube1: {
    DIV_X = [65, 65, 65]; NX = [12, 10, 8];
    DIV_Y = [105];        NY = [20];
    DIV_Z = [105];        NZ = [20];
    NREAL_SCALE_X = [0.5, 0.5, 1];
  };

  block Tube2: {
    DIV_X = [195]; NX = [18];
    DIV_Y = [105]; NY = [10];
    DIV_Z = [105]; NZ = [10];
  };

  block Right: {
    NX=[12]; DIV_X=[115];
    NY=[8, 8, 8]; DIV_Y=[140, 140, 140];
    NZ=[8, 8, 8]; DIV_Z=[140, 140, 140];
  };
};
```

For each bounding box, the following information is to be specified:

- For each bounding box, a block has to be specified with the following syntax:

```
block boundingbox_name: {
    # ... all parameters of the
    # ... bounding box go here
    };
```

- The geometric dimension in X, Y, Z direction is specified in the arrays `DIV_X`, `DIV_Y`, `DIV_Z`. If e. g. a box extends over 195 mm in X direction and should have three segments whereof each extends 65 mm, the specification reads: `DIV_X = [65, 65, 65];` If only one segment in X direction is needed, the line reads: `DIV_X = [195];`
- The number of cells for each sub-segment in X, Y and Z direction is specified in the arrays `NX`, `NY`, `NZ`. The number of entries in e. g. `NX` must be the same as the number of entries in `DIV_X`.

- Further, it is now possible to specify a different `NREAL` particle weighting factor in each bounding box. This is performed by specifying: `NREAL_SCALE = 0.5;` which means, that the particle weighting factor within this bounding is 50% of the global particle weighting factor.
- In case of a sub-segmentation It is also possible to specify different weighting factors for each segment along **one** of the directions X, Y, Z. If we have e. g. three sub-segments along X direction it is possible to specify three different weighting factors by: `NREAL_SCALE_X = [0.25, 0.5, 1];`. The arrays for Y and Z are called `NREAL_SCALE_Y` and `NREAL_SCALE_Z`; each must have exactly as many entries as `DIV_X`, `DIV_Y` or `DIV_Z`.
- A single quad inside a box can be scaled on its own. The name is composed of the box name and the segment indices, and a species suffix restricts the factor to one species:

  `FirstBox_NREAL_SCALE_x0y11z5 = 10;` `FirstBox_NREAL_SCALE_x0y11z5_Ar = 10;`
- The factors multiply the global `NREAL_<species>`: `1` leaves it unchanged, `10` means ten times fewer test particles in that quad, `0.5` twice as many. The values actually used are written to `boundingbox.rd` as `nreal = [...]` per quad, which is the quickest way to check a scaling.
- **Exactly one** of the bounding boxes needs absolute coordinates, these are to be specifed in the variables `X0`, `Y0`, `Z0`. The line: `X0 = -110; Y0 = -210; Z0 = -210;` means, that the lower coordinate edge of the bounding box is located at (-110, -210, -210). The unit of the coordinates is given in the `UNIT` variable in the parameter file. The absolute coordinates of the other bounding boxes will be automatically determined, when all boxes are **joined** together (see below).

Joining multiple bounding boxes together in block MJOIN

After specifying the dimension, segmentation and cell numbers of all bounding boxes, they have to be joined together in order to form a contiguous simulation volume. For this purpose, the block `MJOIN` is used in the parameter file as shown in the following lines.

```
block MJOIN: {
  Left.join(Tube1,  "x", -160, -160);
  Tube1.join(Tube2, "x",    0,    0);
  Tube2.join(Right, "x",  160,  160);
};
```

As can be seen, adjacent boxes are connected via the `join` command, whereof the syntax is `box1.join(box2, "side", da, db);`. The meaning of the parameters is as follows:

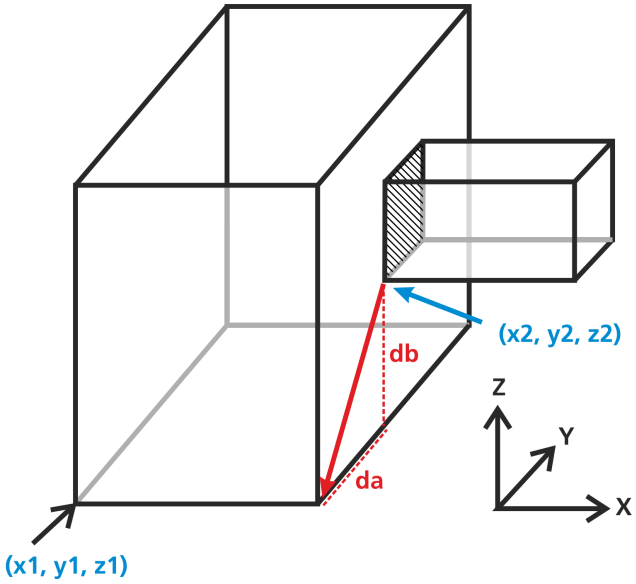
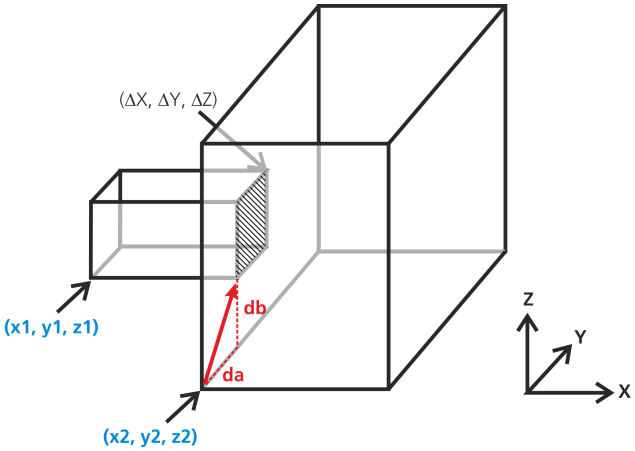
- `box1, box2` = The names of the boxes which are to be joined.
- `side` = The direction of the junction, can be either `"x"`, `"y"` or `"z"`.
- `da, db` = If the junction areas of both boxes are different, a coordinate offset is needed to specify the lateral coordinate offset between both boxes.

For the offsets, the coordinates `a` and `b` are the remaining axes which are vertical to the junction direction. The offsets are obtained by *drawing a vector from the lower edge of the second box to the lower edge of the first box*. Example: If the coordinates of the lower edges are (x1, y1, z1) for the first box and (x2, y2, z2) for the second box, we can obtain the offsets as follows:

Join direction	Direction of a	Direction of b	Offset da	Offset db
x	y	z	y1-y2	z1-z2

Join direction	Direction of a	Direction of b	Offset da	Offset db
y	x	z	x1-x2	z1-z2
z	x	y	x1-x2	y1-y2

Generally, if the lower edge of the second box is positioned within the surface of the first box, the offset is negative. Otherwise it is positive. Examples are given in the following figure:

Second box fits within first box surface	First box fits within second box surface
	

In case if all boxes are joined central-symmetrically, there is no need to calculate the offsets manually. Instead, the command `box1.join_center(box2, "side");` can be used, where the offset parameters are omitted. Our tube example has central-symmetric junctions; thus, the `MJOIN` block can be simplified to

```
block MJOIN: {
  Left.join_center(Tube1,  "x");
  Tube1.join_center(Tube2, "x");
  Tube2.join_center(Right, "x");
};
```

Former join_rect function

In picmc versions prior 2.4.0.5 only the `join_rect` function was available within the block `MJOIN`. For backwards compatibility reasons, this function can still be used. If you are new to picmc you should skip this paragraph and stick to the `join` and `join_center` command ;-).

With the `join_rect` command, our `MJOIN` block would look like follows:

```
block MJOIN: {
  Tube1.join_rect(Tube2, "x", 0, 0, 105, 105,     0, 0);
  Tube2.join_rect(Right, "x", 0, 0, 105, 105,     160, 160);
  Left.join_rect(Tube1,  "x", 160, 160, 260, 260, 0, 0);
};
```

Generally, the syntax of the `join_rect` command between two boxes - "FirstBox" and "SecondBox" - is as follows:

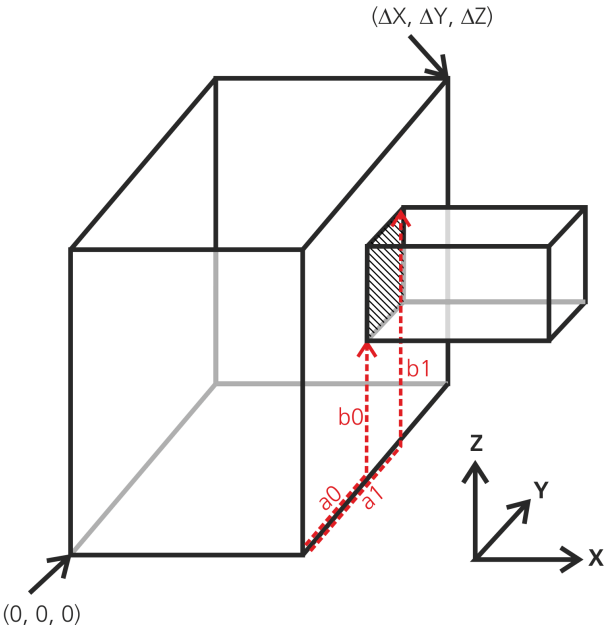
```
FirstBox.join_rect(SecondBox, "direction", a0, b0, a1, b1, delta_a, delta_b);
```

with the following parameters:

- `FirstBox`, `SecondBox` = The names of the bounding boxes as defined in block `MQUAD`. The box with the lower coordinates (with respect to the direction of the junction) should always be specified first as `FirstBox`, the other box with higher coordinates will be the `SecondBox`.
- For `"direction"` chose either `"x", "y", "z"`. This is the direction of the junction.
- The parameters a0,b0,a1,b1 specify the position of the rectangular orifice in relative coordinates with respect to the first box.
- The parameters delta_a, delta_b specify the offset of the rectangular orifice in relative coordinates with respect to the second box.

Examples of the join_rect function

Let's see how it works using some easy examples. In the first example, a big bounding box is joined to a smaller bounding box along X direction:



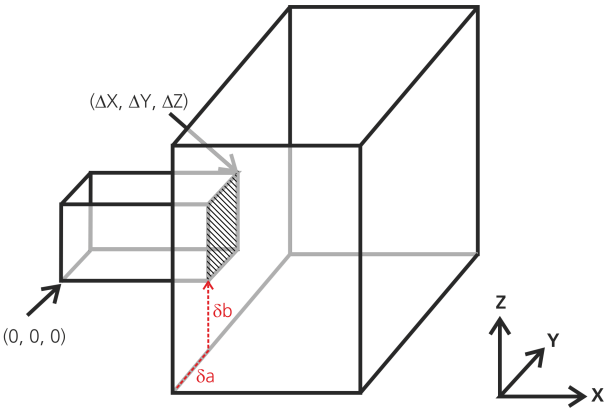
Here, the first box is joined to the second box via the hatched rectangular area. Since the hatched area is smaller than the full right side of the first box, the position of the rectangular area within the right side can be freely chosen. It spans between the coordinates (a0, b0)-->(a1, b1). In the given coordinate system, the connection is performed along X direction; then the a coordinate is parallel to Y, and the b coordinate is parallel to Z. The coordinate conventions are shown in the table in the previous section.

It has to be mentioned again, that the (a0, b0)-->(a1, b1) coordinates are **relative** with respect to the first box, i. e.

$0 \leq a0, a1 \leq \Delta Y$ and $0 \leq b0, b1 \leq \Delta Z$.

Since the hatched junction surface is identical with the left surface of the smaller right box, the offset (delta_a, delta_b) is (0, 0) in this case.

The opposite situation occurs, if a smaller bounding box is joined to a bigger one:



Here, the hatched junction surface completely fills out the right surface of the first bounding box. Thus, the surface coordinates are

$(a0, b0) = (0, 0)$

$(a1, b1) = (\Delta Y, \Delta Z)$.

Now, the position of the junction surface within the left surface of the second box can be freely adjusted via the offset coordinates (delta_a, delta_b). Thus, the full command would read like

```
FirstBox.join_rect(SecondBox, "x", 0, 0, Delta_X, Delta_Y,   delta_a, delta_b);
```

39 Sputter transport

A magnetron sputter process is a two-scale problem: the erosion of the target is decided by the plasma (ions, some 10 eV to 1 keV, time scale ns to µs), the transport of the sputtered atoms to the substrate by the neutral gas flow (0.1 to 10 Pa, time scale ms to s). PICMC covers both, but not in the same run - the time steps differ by four orders of magnitude. The usual way is therefore a PIC-MC plasma simulation that produces the ion bombardment profile, followed by a DSMC simulation that uses this profile as the sputter source.

This page walks through the DSMC part and shows every step in both input forms. The pieces themselves are described in Particle sources, Particle emission and DSMC / PIC-MC iteration.

The physics is the same in both forms:

- the background gas is a DSMC species with collisions from the species and cross-section database,
- the target emits the film species with a Thompson energy distribution,
- the local emission flux follows an imported profile, usually the time averaged `Arplus` absorption of the preceding plasma run,
- the film species sticks on the walls (absorption probability 1), the working gas is reflected.

The three ingredients

1. A case in `WORKMODE = 0`. The work mode is the same keyword as in the parameter file. `WORKMODE = 0` runs DSMC, so no electric field is solved and no charged species is required. `FIELD` plots are skipped with a message in this mode - the relevant output are `number`, `density`, `pressure`, `temperature`, `velocity` for the gas, and `absorption` plus `meshdata` for the film species. `meshdata` writes the absorbed flux per mesh element (a GMSH view on the surface mesh), which is the direct input for a thickness profile.

```
WORKMODE = 0;
DT       = 1e-6;
TSIM     = 1e-3;
```

2. An absorption profile from the plasma run. Put `Arplus` into the `ABSORPTION` plot list of the PIC-MC case, time average the resulting `.pos` files over the RF periods of interest, and copy one file next to the case. `set_profile` reads a GMSH POS view and expects the coordinates in mesh units, the value in the fourth component of each record:

```
add_source("Ta", 20, "sccm");
set_profile("absorption_Arplus.pos");
set_emission_thompson("Ta", 8.1, 1.0, 500, 40);
```

The profile is only a **distribution**: it is binned onto the volume cells touching the border and normalized over the source points of that border. The absolute flux therefore comes from `source_value`, not from the numbers in the profile file. This is convenient (a profile normalized to percent works) but also the most common mistake - a profile from a low density test run then still gives the flux that `source_value` asks for.

`Ub` is the surface binding energy of the emitted species (`8.1` eV for Ta, `4.7` eV for Si), `Eion` and `mion` are energy and mass of the bombarding ion, here 500 eV Ar^+. From these the code computes the recoil cutoff energy of the Thompson distribution itself, so a wrong `mion` silently changes the energy spectrum. This is the recommended form; the older command `set_emission_sputter` still works and

gives the same distribution, but it assumes the bombarding ions (Ar, 40 u) and a cutoff of 100 times the binding energy instead of taking the real bombardment into account.

3. Reactions for the DSMC. In the parameter file the elastic cross sections come from the species definition file `SCR/species_definition.r` (`SPECIESFILE` selects another one), in Python from the `pydatabase`:

```
sim = PICMC.Sim(Specs     = pydatabase.SPECIES_LIST,
                Reactions = pydatabase.REACTION_LIST)
```

Leave the reaction data out and the sputtered atoms fly collisionless, whatever the pressure says.

Order of the reactions on a border

The profile and the emission profile do not belong to the border, they belong to **one wall reaction** of it - the one that was added last. Both input forms attach `set_profile` and `set_emission_*` to the most recently declared reaction, so the list has to be ordered:

```
# film species sticks, the source reaction comes last
add_absorption("Ta", 1);
add_source("Ta", 20, "sccm");
set_profile("absorption_Arplus.pos");
set_emission_thompson("Ta", 8.1, 1.0, 500, 40);
```

Reversing the two puts the emission parameters on the absorption reaction, where they are accepted without complaint but never used, and the target then emits a Maxwellian at wall temperature.

Boundary conditions of the bounding box

For a deposition profile one wants a defined background pressure. Two settings are sensible:

`BOUNDARY_CONDITIONS`	behaviour
all `"specular"`	closed box, the initial filling `P0` is kept, static gas at rest, no pumping. Cheap and consistent with the homogeneous background a PIC-MC case assumes.
all `"outlet"`	every particle leaving the box is deleted. Combined with gas inlets this gives a flow field, but the box sides then act as a very large pump: a few 100 sccm in a 0.1 m^3 volume settle at 1e-3 Pa, not at the process pressure.

The second variant only makes sense with a pump surface of realistic conductance or with gas inlets sized for the pressure that is wanted. For the film thickness this is not a detail - at 1e-3 Pa the mean free path exceeds the target-substrate distance and the profile is purely geometric. In Python the setting is `Geometry(BOUNDARY_CONDITIONS = "specular")`, per side as a dict of the six boundaries.

A worked case

`AP4/DSMC_TA_C32.py` in the EOSS600 project ports the old RVM matrix case (`Si-5e-3-Plasma.par`) to the FHR process module C32 with a Ta target. The mapping:

RVM matrix case	Python case
`WORKMODE = 0`, `DT = 1e-6`, `TSIM = 5.0`	same mode and step, `TSIM = 0.1` (the Ta inventory is at steady state after ~1 ms, the old run needed ~79 h)
`SPECIES = ["Ar", "Si"]`	`{"Ar": …, "Ta": …}`
`NREAL_Ar = 6e12`, `P0_Ar = 0.6`	adapted to the process pressure 0.35 Pa and the cell size
`add_source("Si", 20, "sccm")`	`BorderReaction(reaction_type = "source", source_value = 20, source_unit = "sccm")`
`set_emission_thompson("Si", 4.7, 1, 500, 40)`	`set_emission_Thompson(species = "Ta", Ub = 8.1, alpha = 1.0, Eion = 500, mion = 40)`
`set_profile("…-T1-t102.5-t250.pos")` per target	one profile containing both targets, each border picks its own cells
`abs=1` on all walls	`BorderReaction(absorption = True, probability = 1.0)`

The grid is chosen from the mean free path, not from the plasma: 4.24 mm cells at 0.35 Pa (λ ≈ 19 mm for argon) give 40 x 120 x 204 cells in 288 segments; `NREAL_Ar = 3e11` puts about 20 simulation particles into each cell. Both numbers are far coarser than in a PIC-MC case.

Submit it like any Python case:

```
python3 $PICMCPATH/py/pysubmit.py DSMC_TA_C32.py -n 48 -c IST_EPYC
```

with `-n` below the number of segments (288 here); additional segments are distributed over the available ranks. The parameter file case starts the same way through `rvmslurm`, with the `.par` file instead of the script.

What to look at afterwards

- `absorption` and `meshdata` for the film species: flux and thickness profile. `meshdata` is a surface view, so it can be opened in GMSH on the same mesh as the case.
- `pressure` of the working gas: it should stay near `P0` in a closed box. A pressure that keeps dropping means the box is a pump, not a chamber wall.
- The particle counters in `<case>-out.txt`: the number of sputtered atoms per second should agree with `source_value` (1 sccm ≈ 4.48e17 particles/s per species).
- A histogram array on a plane between target and substrate gives the angular and energy distribution the raytracer works with.

PIC-MC examples

40 Overview

Type	Description	Setup files	How-to
1D - RF parallel plate discharge	Accuracy benchmark for PIC-MC codes	turner1d.zip	Guide and results
3D - magnetic field computation	Basic example with planar magnetron geometry	a700v.zip	Detailed setup guide
Feature info	Description on how to load magnetic field files from external sources		Guide
Feature info	Description on how to iterate subsequently between DSMC and PIC-MC simulations		Guide
2D - magnetron discharge	Basic 2D plasma example with planar magnetron	a700v.zip	Detailed setup guide
2D - magnetron discharge	Visualization of electron movement in an existing plasma	put ZIP file here	How to visualize electron movement
2D - dielectric barrier discharge	Atmospheric pressure discharge in N2	dbd.zip	Detailed setup guide
3D - magnetron discharge	Linear setup to investigate ionization waves a.k.a. 'spokes'	plasma_wave_analysis.zip	Guide and results
3D - RF ICP discharge	Inductively coupled plasma discharge simulation of radio frequency ion thruster	urit-1.zip	Guide and results

41 1D HF parallel plate

A simple 1D setup in Argon with one grounded and one HF excitated parallel plate. This setup has been motivated by Turner et al. (2013)[1] knowing how unreliable PIC-MC modeling results can be without comprehensive testing of the underlying implementation.

Starting the simulation

The setup file contains four parameter files P1.par, P2.par, P3.par and P4.par which can be started with up to 8 CPUs each. Start the simulation (e.g. P1.par) with:

```
$ rvmmpi -picmc 8 P1.par
```

Results

The simulation plots, among others, electron and Ar+ density profiles along domain axis which can be compared against accordant benchmark results: Turner13 - supplementary material. The comparison against these benchmark results can also be seen in the figures below. As a side node: There is a deviation for the low pressure case P1 when using the "wrong" electron energy partitioning model for ionization collisions. We prefer the simpler model 'one-takes-it-all' where the incident electron keeps the remaining energy whereas Turner et. al. rely on 'equal-share' in which the energy is equally divided between incident and secondary electron. The truth should be lying somewhere in between. However, we implemented a switch for each partitioning model so that the user can choose.

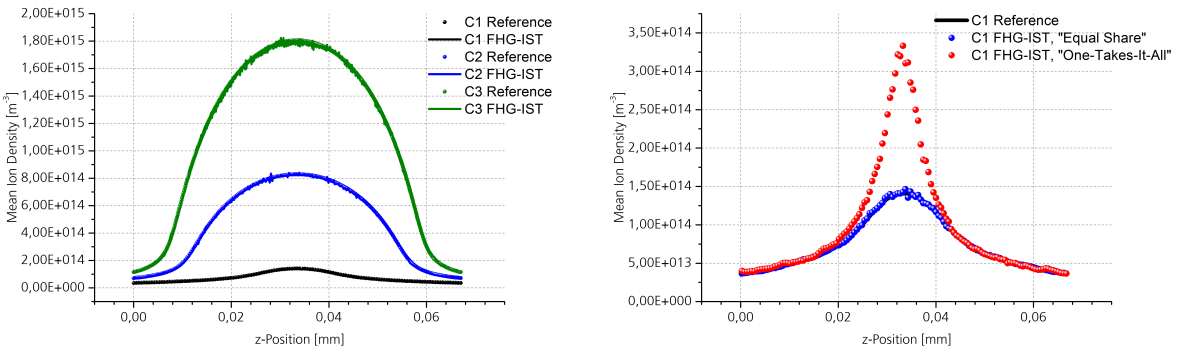


Figure 1: Comparison against benchmark results with only minor deviation (left). Different electron energy partitioning models for ionization events yield deviations in the low pressure case P1 (right).

Footnotes

1. Turner, M.M.; Derzsi, A.; Donkó, Z.; Eremin, D.; Kelly, S.J.; Lafleur, T.; Mussenbrock, T.: Simulation benchmarks for low-pressure plasmas: Capacitive discharges, Physics of Plasmas 20 (2013) 013507 ↩

42 1D parallel plate results

Parallel plate discharge in argon using dual frequency excitation

We validated our PIC-MC implementation with that from Donko08[1] by comparing our results for a 1d3v simulated parallel plate discharge in argon. Hereby a dual frequency excitation is applied, i.e. a 13.56 MHz and a 27.12 MHz cosine signal is superposed with an amplitude of 315 volts.

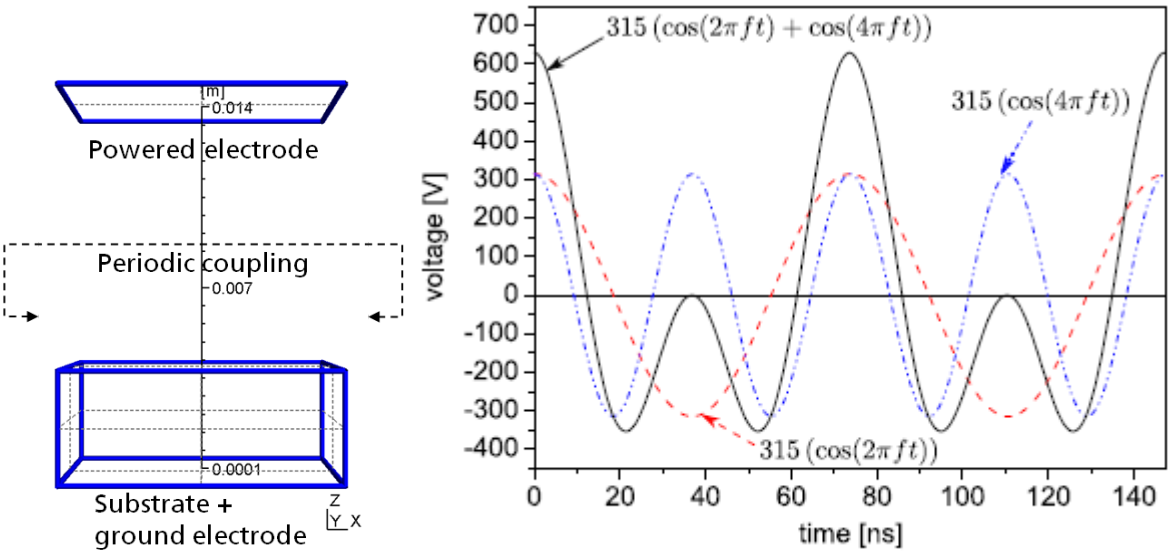


Figure 1: Left: GMSH geometry model for the 1D simulation setup. Right: Applied voltage on powered (top) electrode.

Parameter	Setting	Unit
Argon pressure	0.2	[Torr]
Amplitude	315	[V]
Frequency	13.56 + 27.12	[MHz]
Electrode distance	6.5	[cm]
Electron time step	$1e^{-11}$	[s]
Cell spacing	1 x 1 x 1	[mm^3]
Charge weighting	$1e^6$	[-]
Secondary electron yield	0.2 (> 0.1 to approximate electron reflection)	[-]
Electron reflection on wall	None	[-]
Electron energy partitioning	one-takes-it-all	[-]

Table 1: PIC-MC settings and simulation parameters used for the dual frequency discharge.

The results are nearly identical for both PIC-MC implementations, like seen for example in the ar+ hit histogram on the powered and grounded electrode, respectively. However, we had some discrepancies at first as we used a more simplified ar+ collision model as in Donko08. Thus, we adjusted our model to that of Donko08 using the more sophisticated ar+ collision model from Phelps.

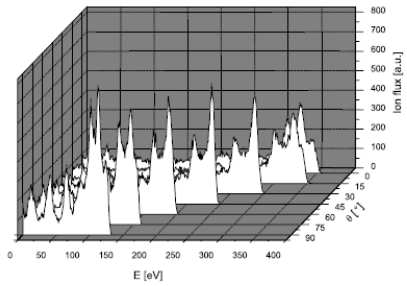
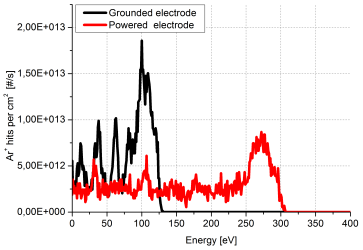
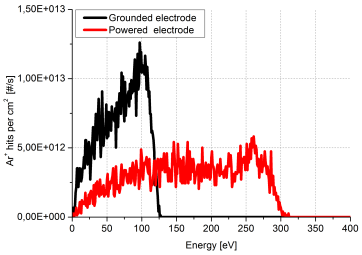
Ar+ histogram from Donko08	Ar+ histogram from IST	Ar+ histogram from IST using simplified ar+ collision scheme
		

Figure 2: Ar+ hit histogram for different PIC-MC implementations and ar+ collision models, respectively.

Footnotes

1. Z. Donkó, PIC simulations of the separate control of ion flux and energy in CCRF discharges via the electrical asymmetry effec, 2008 ↩

43 2D magnetron discharge

This Example is based on the understanding of the DSMC example, so look at this one first.

Geometric model

In this example, we simulate the plasma flow through an A700V sputtering chamber provided with a PK750 magnetron. We have a 2D-model of the chamber and a 3D-model of the magnetron (see Fig. 1).

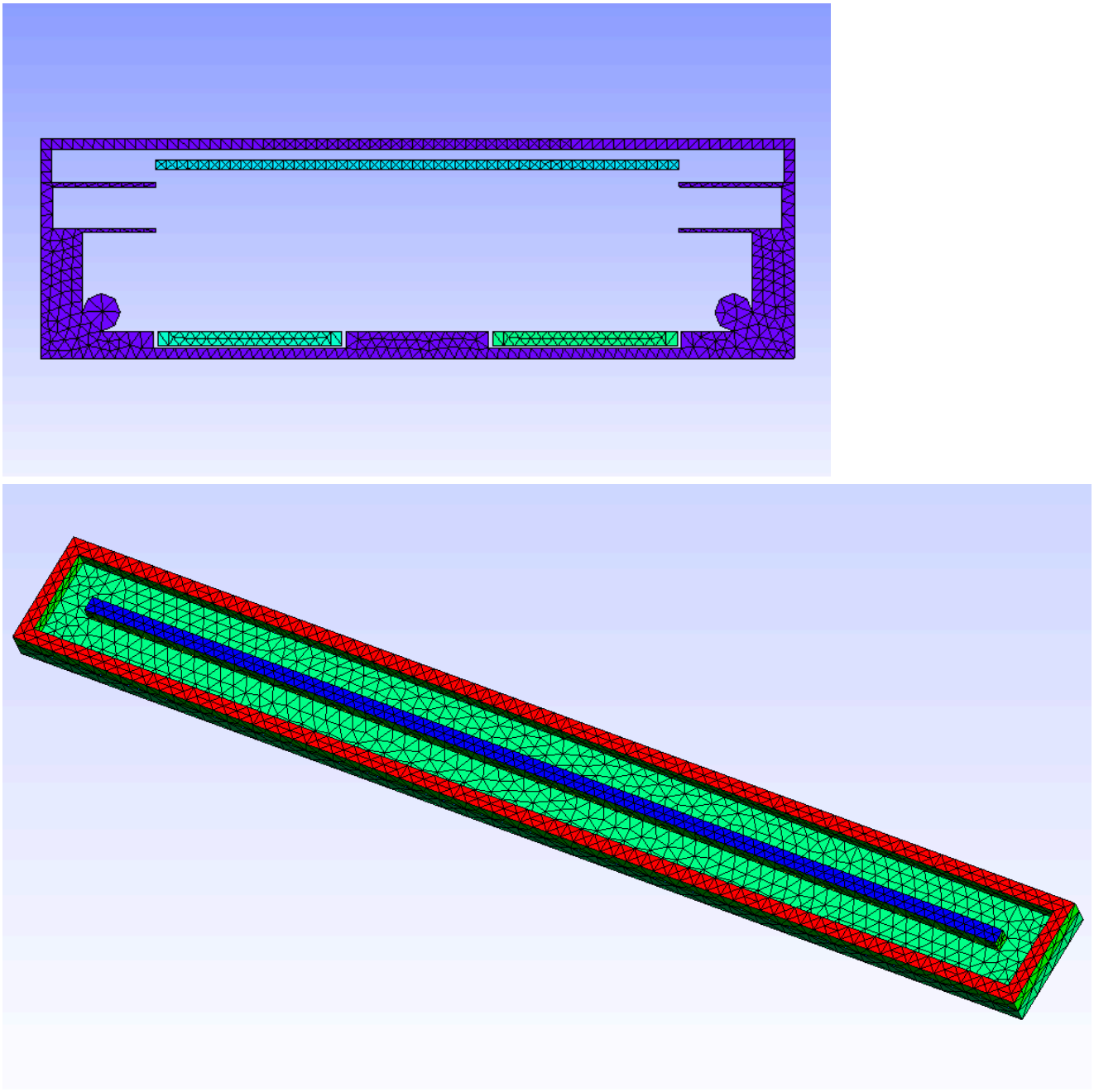


Figure 1: 2D-model of an A700V sputtering chamber & 3D-model of a PK750 magnetron

How To

Step 1: Preparation

Unzip A700V_PK750.zip in the same directory. The file contains the GMSH models for this example.

Step 2: Create and edit the parameter file

Create a parameter file:

```
$ initpicmc a700v
```

Edit the "a700v.par"-file.

First we need a good time resolution for the plasma simulation:

```
DT       = 5e-11;
TSIM     = 10e-6;
```

Set the workmode to plasma simulation:

```
WORKMODE = 1;
```

We have to adjust the power dissipation:

```
PSOLL = 10/750;
```

This corresponds to 10 W distributed over 750 mm target length. We simulate only a 1mm-slice of the target, so the power dissipation has to be adjusted accordingly. Since we want to simulate a plasma, we need some particles:

```
SPECIES = ["Ar", "Arplus", "e"];
```

The "NREAL"-parameter is the factor between the number of simulated and real particles and the "P0"-parameter is the pressure (for neutral particles like Ar) or the initial density (for charged Particles like Ar+ and e-). The appropriate "NREAL_Ar"-parameter can be calculated (formula further down, because it depends on other parameters). Possible values for the other parameters are:

```
NREAL_Ar     = 5e9;
P0_Ar        = 0.5;
NREAL_Arplus = 1e5;
P0_Arplus    = 1e13;
NREAL_e      = 1e5;
P0_e         = 1e13;
```

Depending on what plots you are interested in, insert the species into the desired category:

```
NUMBER   = ["Arplus", "e"];
DENSITY  = ["Arplus", "e"];
PRESSURE = ["Ar", "Arplus", "e"];
VELOCITY = ["Arplus", "e"];
```

The unit of the mesh geometry depend on the person who generated the mesh file. In our case it is in [m], so the 1.0 is ok. The extent of the bounding box is calculated automatically from the a700v.msh-file, but we have to adjust the cell spacing. The bounding box has a volume of 360*105*1 mm³. For a simulation with a reasonable plasma density, a cell dimension of 1x1x1 mm³ is appropriate. We also segment the total volume into subvolumes, to speed up the simulation, because the calculation can be parallelized. A possible cell resolution with 12 subvolumes (4x3x1) looks like:

```
NX   = [90, 90, 90, 90];
NY   = [35, 35, 35];
NZ   = [1];
```

Since we have a 2D-xy-model of the sputtering chamber, we have to change the border types in z-direction:

```
TYPE_z1 = "periodic";
TYPE_z2 = "periodic";
```

We also want so simulate the magnetic field, so the mesh file for the PK750 magnetron is needed:

```
BEMMESH  = "pk750-single_standard_unbalanced.msh";
```

Now we need to specify the reactions wich occour on our surfaces. Additionally the wall of the sputtering chamber should be set to ground potential:

```
Border Wall = {
    ⋮
  vf = "0";
  reactions: {
    add_absorption("Arplus", 1.0);
    add_absorption("e",      1.0);
  };
};
```

If the simulated substrate should be an insulator instead of a metal, e.g. a glass:

```
Border Substrate = {
    ⋮
  epsr = 6.0;
  reactions: {
    add_absorption("Arplus", 1.0);
    add_absorption("e",      1.0);
  };
};
```

The voltage function of the target T1 should be set to "-VP", which is an internal variable. In the actual example the second target is not powered, so it gets the same parameters as the chamber wall:

```
Border T1 = {
    ⋮
  vf = "-VP";
  reactions: {
    add_plain_reaction("Arplus", 1, 1, ["Ar", "e"], [1, 0.1]);
    set_emission_energy("e", 0, 5, 1);
    add_absorption("e",      1.0);
  };
};
```

```
Border T2 = {
    ⋮
  vf = "0";
  reactions: {
    add_absorption("Arplus", 1.0);
    add_absorption("e",      1.0);
  };
};
```

As mentioned above, the "NREAL_Ar"-parameter has to be calculated, depending on some other parameters. The number of particles in each cell, has a big impact on the simulation time. A good benchmark are ca. 20 simulated particles/cell. Depending on the Ar pressure (P0_Ar = 0.5 Pa), temperature (T0 = 300 K) and the cell dimension (1x1x1 mm^3) you can calculate the number of real particles per cell and then adjust the "NREAL-Ar":

$$N = \frac{p\,V}{k\,T} = 1.21 \cdot 10^{11} \qquad \Rightarrow \qquad \mathrm{NREAL_Ar} = 5 \cdot 10^{9} \qquad \Rightarrow \qquad 24\ \mathrm{particles/cell}$$

Step 3: Create and edit the magnetic template

Create a template for the magnetic fieldsolver:

```
$ initbfield a700v
```

Edit the "a700v.bem"-file. The relative permeabilities of the three domains (vacuum, magnets and yoke) can stay the same:

```
Domains = [1, 1.05, 1000];
```

Then you have to define and correlate all physical surfaces. The "val"-parameter is the magnetic remanence value. The "domain"- and "neighbourdomain"-parameter correspond to the above defined domains. The bottom of the inner and outer magnet are connected to the yoke and all other surfaces are connected to vacuum:

```
Boundary Outer_Magnet_Top    = {icodec = 1; type = "interface"; val = -1.40;  domain = 1;
neighbourdomain = 0;};
Boundary Outer_Magnet_Bottom = {icodec = 2; type = "interface"; val =  1.40;  domain = 1;
neighbourdomain = 2;};
Boundary Inner_Magnet_Top    = {icodec = 3; type = "interface"; val =  1.40;  domain = 1;
neighbourdomain = 0;};
Boundary Inner_Magnet_Bottom = {icodec = 4; type = "interface"; val = -1.40;  domain = 1;
neighbourdomain = 2;};
Boundary Magnet_Skin         = {icodec = 5; type = "interface"; val =     0;  domain = 1;
neighbourdomain = 0;};
Boundary Yoke                = {icodec = 6; type = "interface"; val =     0;  domain = 2;
neighbourdomain = 0;};
```

A complete version of the model including both parameter files can be found here.

Step 4: Start magnetic field computation

Start the computation with n processes. In our case with n = 12 just enter:

```
$ rvmmpi -bem 12 a700v
```

In the output look for the following lines:

```
MASTER <- {BEM 10}:     [MESSAGE]        domain:   2 checksum:  1.0319e-13
MASTER <- {BEM  6}:     [MESSAGE]        domain:   1 checksum: -5.0333e-13
MASTER <- {BEM  1}:     [MESSAGE]        domain:   0 checksum:  1.2566e+01
```

For all areas the sum should be exactly $4\pi = 12.5664$ or 0 (round-off errors like 1e-12 are ok).

Step 5: Start simulation

Start the simulation with n processes (n may not exceed the number of subvolumes). In our case with n = 12 just enter:

```
$ rvmmpi -picmc 12 a700v
```

Results

Substrate on ground potential

The plasma in a 2D set-up is not coupled and thus, more sensitive to the anode configuration. With the substrate on ground potentialthe right plasma is as bright as the left one (see figure 1).

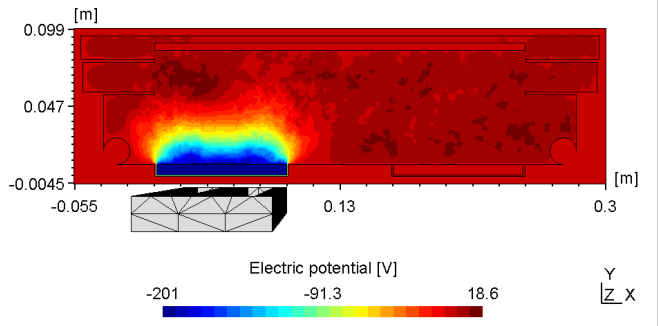
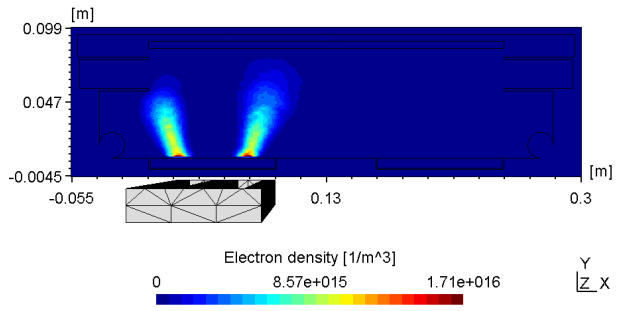
Substrate on ground potential	
	

Figure 1: Electric potential (left) and electron density (right) during simulation with substrate on ground potential (0 V).

Substrate on floating potential

Here the left plasma is conductively connected to the ajacent chamber wall, the right one however is connected with the top of the chamber wall. Thus, with the substrate on floating potential some of the anodic surface is shielded causing the right plasma to fade (see figure 2). Due to the uncoupled electron drift current in a 2d setup this effect is even more pronounced. Thus, if using constant power the voltage can drop below ignition point causing the right plasma to diminish.

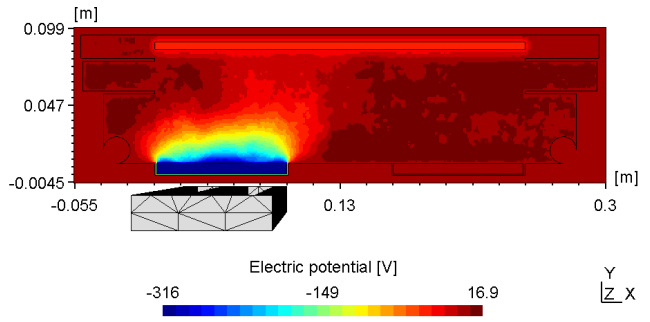
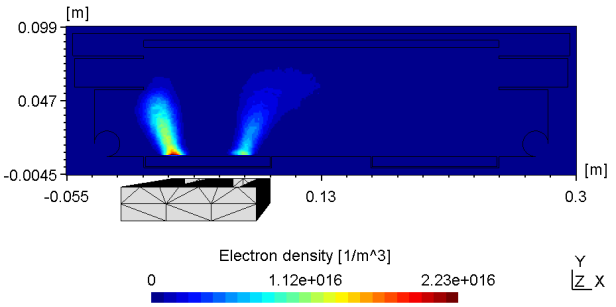
Substrate on floating potential	
	

Figure 2: Electric potential (left) and electron density (right) during simulation with substrate on floating potential.

Discharge anomaly(?) with strong magnetron type

As seen in the pictures below we get some anomalies if using our coater geomtry settings together with a balanced magnetron. A spacing of 1 mm between target and dark space shielding causes an distortion of the electric field. Thus, we get an extensive electric field strength parallel to the target surface. Thus, electrons are shifted sideways in the magnetic confinement and can escape more easily. Therefore excessively high voltages are needed to get a self sustaining plasma which is even so quite unstable.

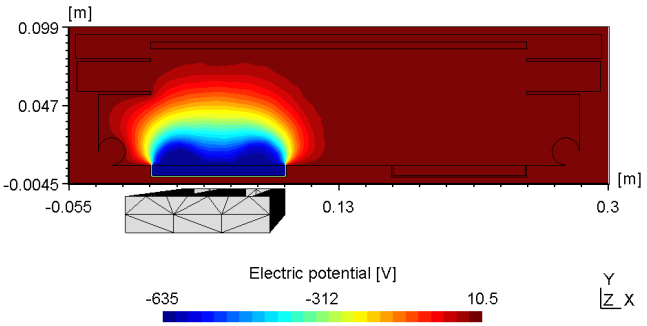
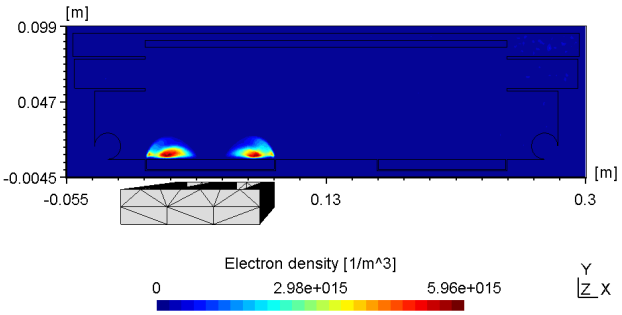
Short-circuit with the edge of the dark space shielding	
	

Figure 1: Electric potential (left) and electron density (right) during simulation with balanced magnetron.

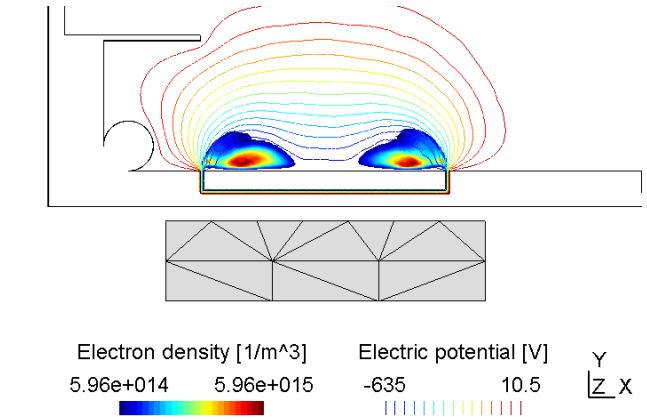
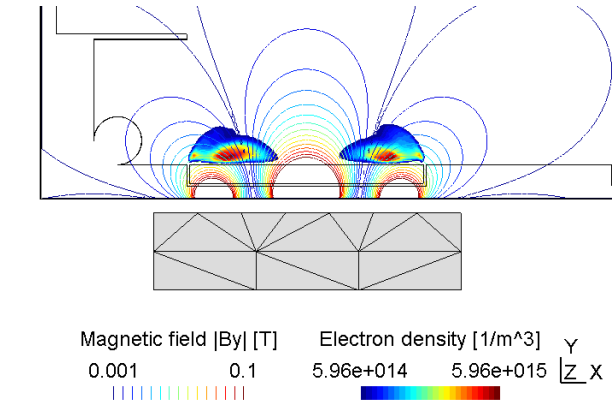
Short-circuit with the edge of the dark space shielding	
	

Figure 2: Electron density overlayed with electric potential (left) and magnetic field By (right) isolines.

44 2D magnetron results (A700V)

Magnetron discharge simulation with A700V chamber layout

We performed a series of quasi two-dimensional magnetron discharge simulations in argon. Hereby we used a simplified chamber layout of our A700V coater to define the simulation domain. The results were a little bit puzzling, cause they do not match with oberservations and/or measurements on our real coater. It seemed there is anything wrong or missing in the simulation setup or in our software. Hence, we tried to figure out the problem using a well defined parameter set. The simulation parameters can be found in the table below. If not mentioned otherwise these are settings the following simulation results are based on.

Parameter	Setting	Unit
Argon pressure	0.5	[Pascal]
Applied power	100	[W/m]
Chamber wall potential	0 (grounded)	[V]
Dark space shield spacing	1	[mm]
Electron time step	$2.5e^{-11}$	[s]
Cell spacing	1 x 1 x 1	[mm^3]
Charge weighting	$5e^5$	[-]
Secondary electron yield	0.088 (argon on copper)	[-]
Electron scattering	isotropic	[-]
Electron energy partitioning	one-takes-it-all	[-]
Electron reflection on wall	None	[-]

Table 1: PIC-MC settings and simulation parameters used for the DC magnetron discharge simulations.

We tried out two kinds of magnetron configurations for the general PK750 magnetron design, i.e. balanced and unbalanced, respectively. The balanced magnetron configuration uses the same size and strength for the inner and outer magnets on the yoke. For the unbalanced configuration however, size and/or strength can differ either for the inner or outer magnets, whether one wants more or less ion bombardment on the substrate. In the figures below you see the magnetic isolines for the balanced and unbalanced magnetron configuration, respectively.

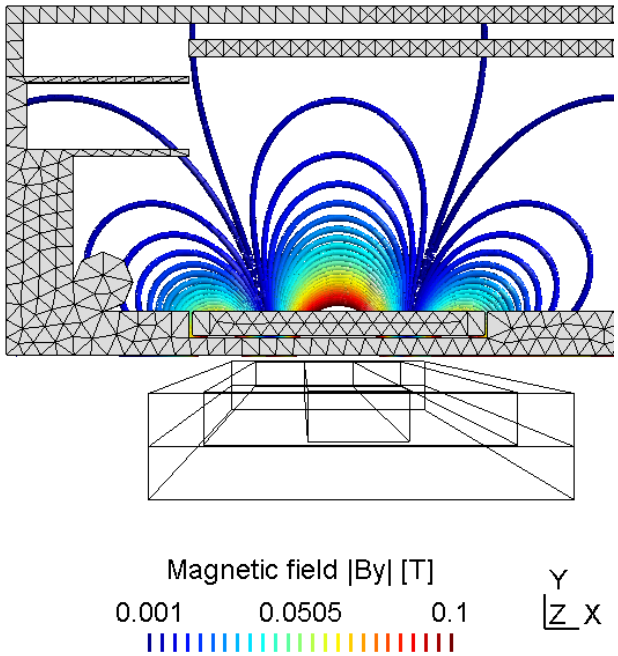
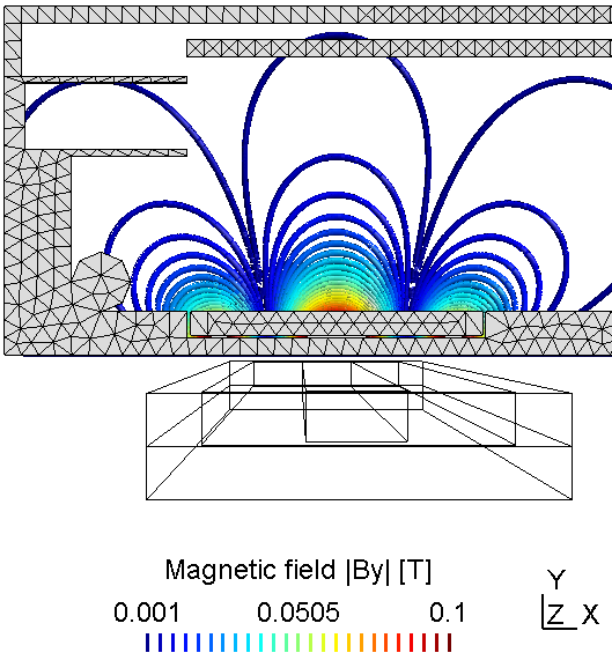
Inner remanence = outer remanence = 1.35	Inner remanence = 0.85, outer remanence = 1.35
	

Figure 1: Isolines of the magnetic field strength in y-direction for the balanced (left) and unbalanced (right) magnetic configuration.

Unbalanced magnetron: Anode shielding with substrate on floating potential

In this simulation we used the unbalanced magnetron configuration. With an unbalanced magnetron however, it seems the plasma is more sensitive to the anode configuration. Whereas the left plasma is conductively connected to the ajacent chamber wall, the right one is connected with the top of the chamber wall. Thus, with the substrate on floating potential some of the anodic surface is shielded causing the right plasma to fade (fig. 1). With the substrate on ground potential however,the right plasma is as bright as the left one (fig. 2) due to an anodic surface to connect with.

Due to the uncoupled electron drift current in a 2d setup this effect is even more pronounced. Thus, if using constant power the voltage can drop below ignition point causing the right plasma to diminish.

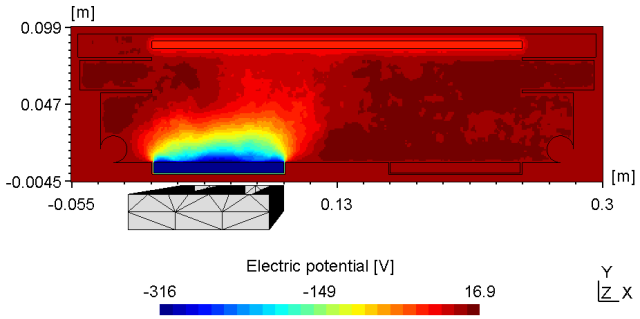
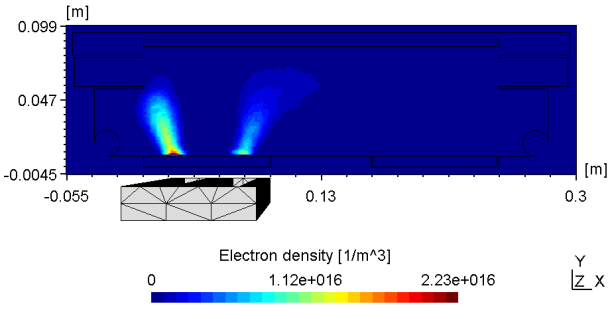
Substrate on floating potential	
	

Figure 1: Electric potential (left) and electron density (right) during simulation with substrate on floating potential.

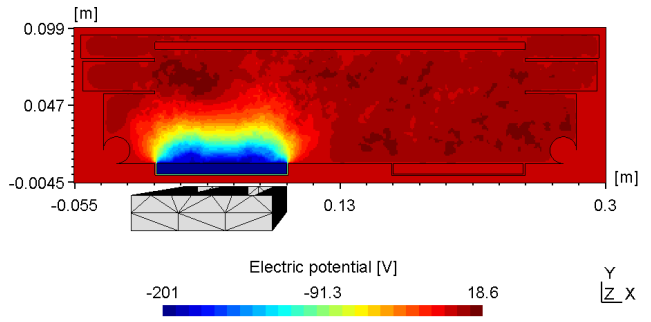
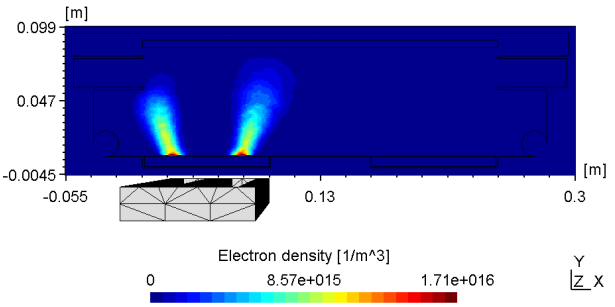
Substrate on ground potential	
	

Figure 2: Electric potential (left) and electron density (right) during simulation with substrate on ground potential (0 V).

Balanced magnetron: Short-circuit with dark space shielding

As seen in the pictures below we get some anomalies if using our coater geomtry settings together with a balanced magnetron. A spacing of 1 mm between target and dark space shielding causes an distortion of the electric field. Thus, we get an extensive electric field strength parallel to the target surface. Thus, electrons are shifted sideways in the magnetic confinement and can escape more easily. Therefore excessively high voltages are needed to get a self sustaining plasma which is even so quite unstable.

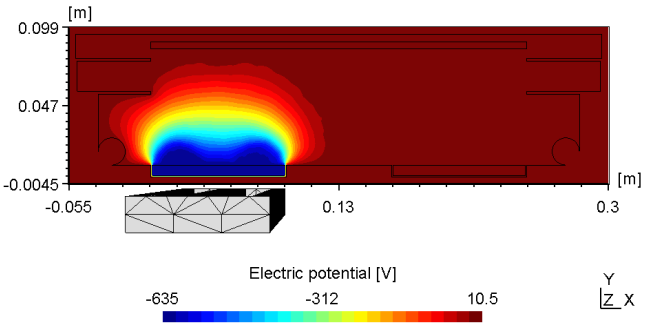
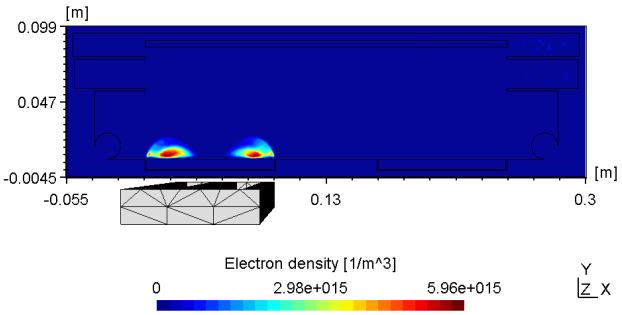
Short-circuit with the edge of the dark space shielding	
	

Figure 1: Electric potential (left) and electron density (right) during simulation with balanced magnetron.

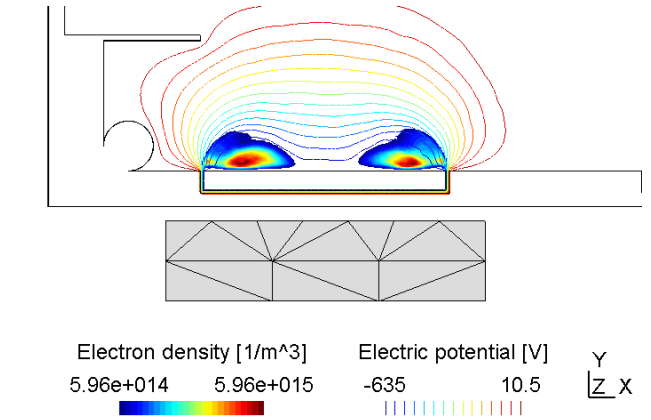
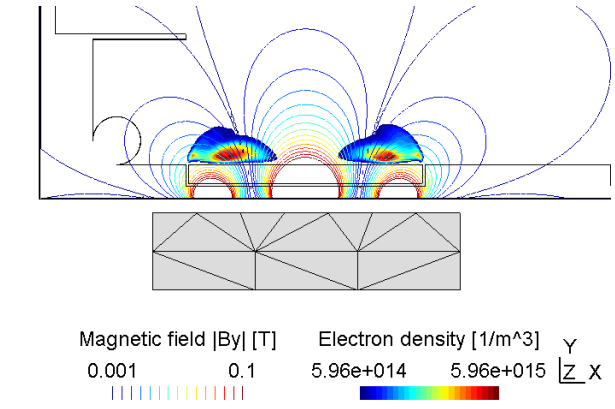
Short-circuit with the edge of the dark space shielding	
	

Figure 2: Electron density overlayed with electric potential (left) and magnetic field By (right) isolines.

45 2D dielectric barrier discharge

A two-dimensional dielectric barrier discharge with N2 gas at atmospheric pressure. Even though discharge conditions - high pressure, high plasma density - are not favourable for a PIC-MC approach it is applicable here due to a very small geometry and time scale, respectively.

Starting the simulation

The zip archive contains mesh and two parameter files for the plasma simulation. To start case one with negative electrode potential (-1000 V) and 16 processes maximum enter:

```
$ rvmmpi -picmc 16 v3_highRes_minus.par
```

To start case two with positve electrode potential (-1000 V) and 16 processes maximim enter accordantly:

```
$ rvmmpi -picmc 16 v3_highRes_plus.par
```

Results

The videos below show the resulting electron density and potential evolution during 40 nanoseconds(!) of the simulated discharge.

MP4 Video	MP4 Video

46 3D linear magnetron

A three-dimensional, linear magnetron discharge setup in Argon with perodic boundary conditions and a homogenous field distribution. This setup was used to investigate the so called 'spokes' or 'ionization wave' phaenomenon as seen in the results section.

Starting the simulation

The zip archive contains mesh and parameter files for magnetic field computation and plasma simulation. To start the magnetic field computation process e.g., with 40 CPUs, enter:

```
$ rvmmpi -bem 40 v0.par
```

The simulation domain is divided into 32 sub-quads and thus, can be executed with a maximum of 32 CPUs - or less of course. To start plasma simulation enter:

```
$ rvmmpi -picmc 32 v0.par
```

Postprocessing

There is a script included to generate cut-planes from the 3D data plots. As the script parser 'RigVM' is intalled together with the DSMC / PIC-MC software package just enter:

```
$ rvm ppp_v0.rvm
```

The script contains only five lines of code so check it out and adjust it accordant to your needs.

Results

The video below was realized with the resulting Ar+ and electron density plots. It shows a spontaniously formed plasma density aggregation moving against electron drift direction. An extensive plasma physical description of this effect is given by Anders2014.

MP4 video

47 3D RF ICP discharge

In this example we show the required steps to set up an inductively coupled plasma (ICP) simulation. The simulation model is a so called µRIT - a small scaled, RF excitated ion thruster using Xenon as propellent. The example setup is similar to the model of an µRIT as shown in Robert Henrich et. al. at IEPC 2015 ((Henrich, R.; Becker, M.; Gärtner, W.; Holste, K.; Heiliger, C.: Overview of RIT Simulation Areas in Giessen with Focus on Plasma Simulation, IEPC-2015-91/ISTS-2015-b-91 conference paper (2015))).

Coil current computation with BEM

The zip-archive for this example contains two mesh files 'RIT-1.msh' and 'RIT-1_coils.msh' as seen in **figure 1**. The latter is derived from the former by commenting out all physical domains beside the coil's skin surface and both facesides from corresponding 'RIT-1.geo' geometry file.

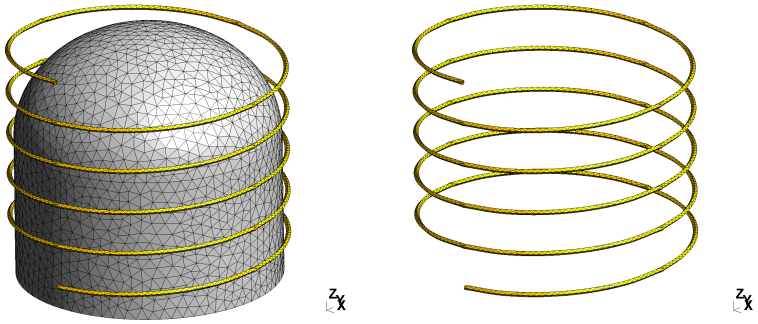


Figure 1: Geometry mesh with coil for PIC-MC simulation (left) and separated coil mesh for BEM computation (right).

Due to the skin effect the coil current can be described as surface current. We can use the BEM solver to compute such a field by applying Laplace boundary conditions to the coil's physical surfaces. The accordant parameter file 'RIT-1_0p83P_0p08W.bem' looks like this:

```
Domains = [1];

Boundary coils_i = {icodec = 1; type = "neumann";   val = 0; domain = 0; neighbourdomain = 0;};
Boundary coils_p = {icodec = 2; type = "dirichlet"; val =+1; domain = 0; neighbourdomain = 0;};
Boundary coils_n = {icodec = 3; type = "dirichlet"; val =-1; domain = 0; neighbourdomain = 0;};
```

There is only one domain and no interfaces defined. Hence, 'domain' and 'neighbourdomain' have no relevance here and can stay zero. You start the BEM solving process on 20 CPU cores by entering:

```
rvmmpi -bem 20 RIT-1_0p83P_0p08W.par
```

Since version 2.4.2.2. there is a switch in the parameter file's 'Additional switch' section:

```
ICP_SOURCE = 1;
```

Please uncomment it to improve performance of BEM solver when used for ICP source computation only. After the solving process a file 'BEM_icp_source_field.pos' is available in the project folder (see **figure 2**).

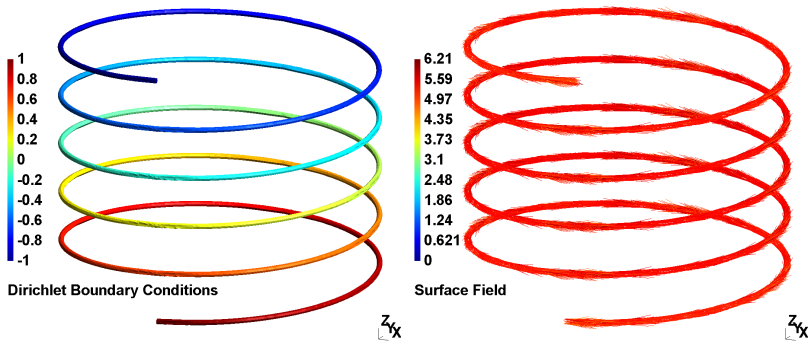


Figure 2: Resulting potential distribution 'BEM_Dirichlet.pos' (left) and derived gradient field 'BEM_icp_source_field.pos' (right).

Setup in parameter file

The mesh for PIC-MC simulation must include the coil on which the vector field from previous BEM computation is applied. This is done by the reaction entry 'add_icp_source("filename", frequency)'. Move the file 'BEM_icp_source_field.pos' to where the parameter file resides, otherwise 'filename' must include the project folder e.g., 'RIT-1_0p83P_0p8W/BEM_icp_source_field.pos'. The second parameter 'freqency' defines the excitation frequency. The coil has been defined as fully transparent membrane here only because it is outside the discharge chamber and thus, coil-particle interactions must not be considered:

```
Border coils_i = {
  icodec      =   4;                          # Codec number from meshfile
  type        = "membrane";                   # Membrane for convenience
  T           = PAR.T0;                       # Wall temperature
  reactions: {
    Transparent_Membrane;                   # Fully transparent membrane
    # Load vector field from file with excitation frequency of 5 MHz
    add_icp_source("BEM_icp_source_field.pos", 5e6);
  };
};
```

As for the other parameters e.g., applied power, time step width or particle weighting, they follow the same rules as with CCP simulation. Other then the aforementioned procedure there are no additional switches or settings to be taken into account. The simulation is started with the command:

```
rvmmpi -picmc 10 RIT-1_0p83P_0p08W.par
```

Results and comparison

Using the discharge parameters found in the IEPC paper (0.085 Watt @ 5 MHz) the ICP simulation converged after approximately 30 RF cycles. As seen in **Figure 3**, results from the Fraunhofer IST PIC-MC software are in relatively good agreement with the reference from IEPC proceedings paper. However, the coil slope related plasma density asymmetry is much less pronounced. This might be related to differing geometry and / or parameter settings.

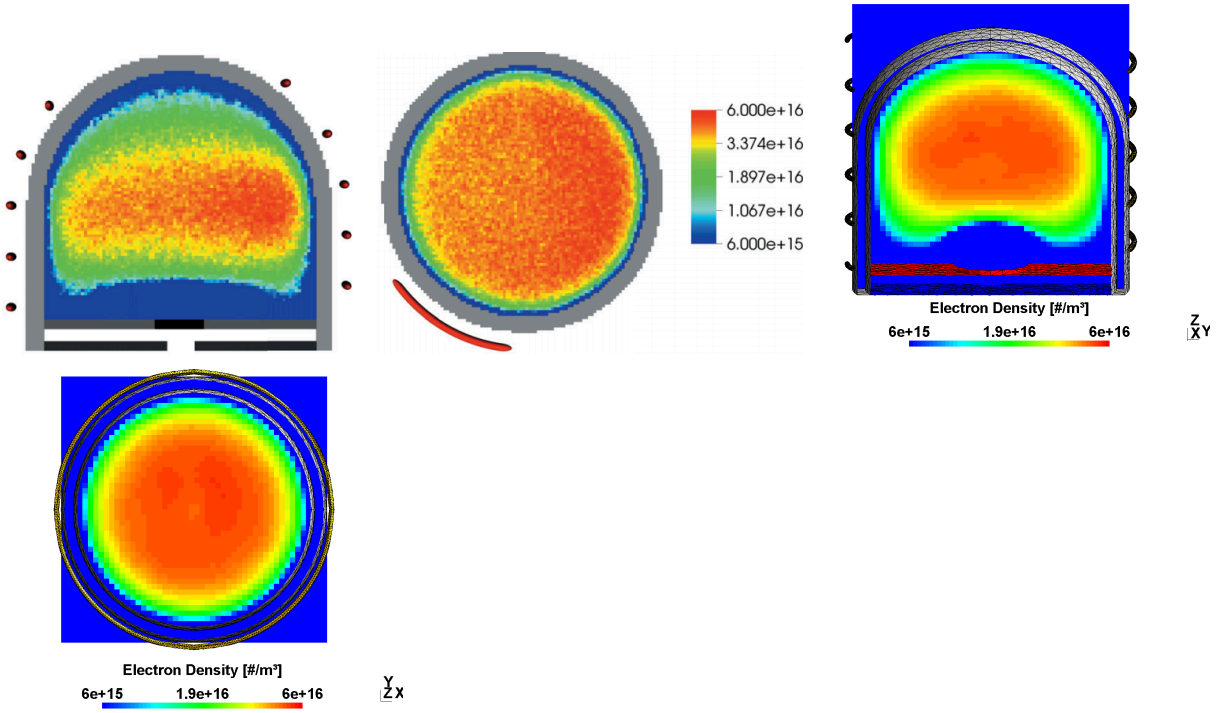


Figure 3: Electron density in $1/m^3$ at y-z plane and x-y plane from Gießen PIC-MC software (top) and accordant results from Fraunhofer IST PIC-MC software (bottom).

48 BEM magnetic field

The field problem of a permanent magnet arrangement, according to [Jac83] ((Jackson, JD:. Classical Electrodynamics, Walter de Gruyter, p. 223-227 (1983))) as the solution of the Poisson equation for a magnetostatic scalar potential to be formulated. The magnetization of the permanent magnets is assumed in good approximation as homogeneous and replaced by an equivalent magnetic surface charge. The distribution of surface charges and magnetic permeability in the computational domain is the source term of the Poisson equation. With an appropriate definition of boundary conditions, the magnetostatic field problem is described so fully.

For the numerical calculation of the Poisson equation the boundary element method (BEM) is used. This method does not need to be discretized as finite differences or finite element method, the entire computational volume, but only the surface of the computational domain. The method of boundary elements is therefore particularly suitable for field problems with open boundary conditions in an infinite computational domain. Another advantage of this method is the high accuracy over the above-mentioned volume methods, but are paid for with increased memory and computational effort has to.

Usage



note

This page describes the parameter file route. The same solver is available from a Python case script as `PICMC.BEM_Solver`, where the `.bem` file is replaced by `BEM_Boundary` objects and the `bem` binary is started from the script - see Magnetic field solver.

The boundary element method (BEM), how we implemented it, can be used also for electrostatic field problems without space charges. However it has been primarily developed for magnetostatic field

problems, i.e. magnetic fields from permanent magnet arrangements. Thus, we exemplarily show the application of the BEM field solver for a simple rectangular magnetron setup in this section.

First we have to define the geometric model using GMSH. Hereby some modeling constraints have to be taken into account slightly different to that of the DSMC and PIC-MC simulation respectively. From the meshed surface model a parameter file is created automatically which has to be adjusted manually. Finally the computation can be executed via a console command. In the following subsections these subsequent steps are described in more details.

Geometry modelling

We exemplarily take an simple planar magnetron setup and create a geometry model using GMSH. The end section of planar magnetron is seen in figure 1.

Exemplary PK750 magnetron layout (unbalanced)

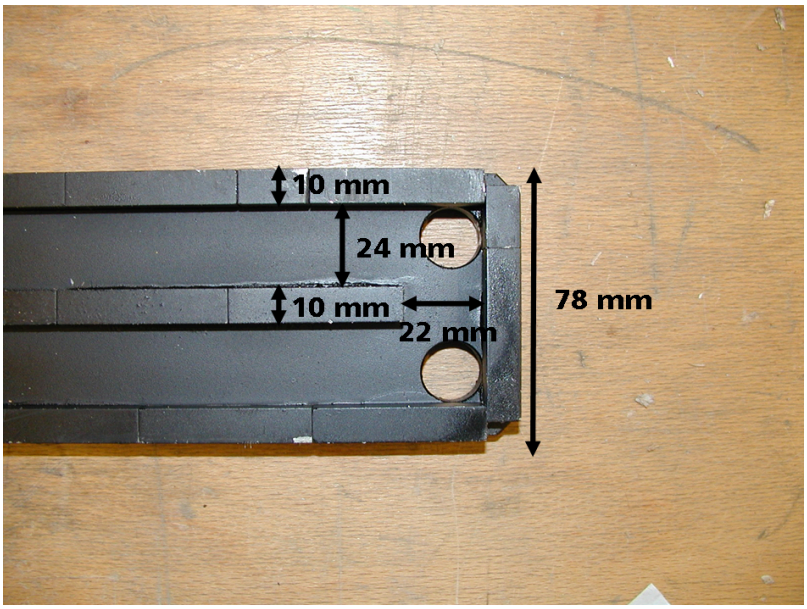


Figure 1: Empirical magnetron layout from which we create a geometry and mesh model, respectively.

The first thing we have to do is to distinguish the number of domains, i.e. number of self-contained material areas. In the current case we have (i) vacuum, (ii) magnets and (iii) yoke. Unless you have magnets with different permeabilities, all magnets belong to the same domain. For each domain there is an interface to at least one other domain. Additionally, for the magnets we need an interface for the top surface, the bottom surface and the skin surface. The top and bottom surfaces will later on be assigned the north and south pole, so they have to be separated from the skin surfaces.

In our case we have six interfaces: (1) from the yoke to the vacuum, (2) from the magnet skin surfaces to the vacuum, (3) from the outer magnet top surface to the vacuum, (4) from the outer magnet bottom surface to the yoke, (5) from the inner magnet top surface to the vacuum and (6) from the inner magnet bottom surface to the yoke.

Now when we create the geometry model with GMSH we define **one and only one** geometrical surface for each interface. Thus, in our case the upper surface of the yoke has two "holes" for the bottom surfaces of the inner and the outer magnet as seen in figure 2.

Step 1: Create or load a geomtric surface model with GMSH.

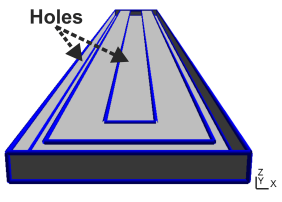
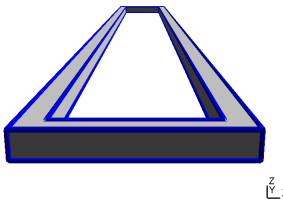
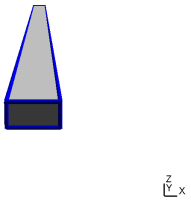
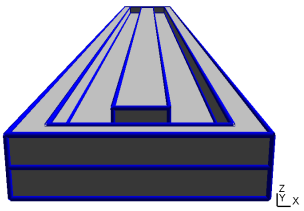
Yoke	Outer magnet	Inner magnet	Complete Magnetron
			

Figure 2: GMSH geometry models of planar magnetron.

When all geometric surface are defined they must be grouped accordantly and assigned to a string tag. Read the GMSH instructions on how to define physical surfaces from geometric surfaces. After the physical surface definition the geometric model can be meshed and saved. In short, there are the following general rules for setting up a magnetron geometry:

- Magnets should always be constructed as a closed body.
- If a magnet directly touches a yoke, there should be **only one** surface between the magnet and the yoke.

Parameter file settings

If a magnetic field computation is performed, there are always **two different meshfiles** required:

- One meshfile for the magnetic arrangement itself, let's call it **magnet.msh**
- Another meshfile, where the geometric **region of interest** is defined. This is the region, where the magnetic field vectors should be computed and where later on a plasma simulation will be performed. Let's call the second meshfile **chamber.msh**.

Step 1: Create a parameter file. To generate a parameter file for the BEM field solver first execute the command on the mesh-model for plasma simulation:

```
initpicmc chamber.msh
```

whereas "chamber.msh" is the accordant mesh file to define the computation domain for plasma simulation and electric field computation, respectively (i.e. the second meshfile mentioned above). This command generates a parameter file named "chamber.par" in the same directory. All definitions regarding computation domain, cell spacing, cell resolution, number of subquads, etc., are defined within this parameter file. The region of interest for magnetic field should be the same computation domain and cartesian grid, respectively as for plasma simulation. Thus, by deriving parameter file for magnetic field computation from primary parameter file, twofold definition is avoided. Hence, you have to find the section where mesh files are defined and insert a chamberfor the magnetic mesh-model you want to use. The section looks like this:

```
MESHFILE = "chamber.msh";   # Name of mesh file for PICMC particle simulation
BEMMESH  = "";              # Name of mesh file for BEM b-Field solver
```

Insert the proper filename, e.g. **magnet.msh** in the empty quotes next to the BEMMESH tag. Remember that mesh files must be in same directory as parameter files. Now execute the command:

```
initbfield chamber.par
```

This command creates a template for the magnetic fieldsolver named **chamber.bem** in the same directory.

Step 2.0: Adjust parameter file.

The only parameters you have to adjust in **chamber.bem** are:

1. The relative permeability of every domain, e.g. 1.05 for the magnets or 1000 for the yoke. These are the entries in the "Domains" array.
2. The definitions for all surfaces defined in the magnetic mesh-model. This is the tricky part, because you have to connect interfaces surfaces manually by using the prober domain numbers. Also you have to define the magnetic remanence, if any, on the accordant surfaces. All surface definitions take place within the record entry "Codecs" in the parameter file.

Step 2.1: Set number of domains and domains' relative permeability, respectively.

The magnetron example from above section consists of three domains:

1. The vacuum with a relative permeability of 1.0.
2. The magnets with a relative permeability of 1.05 (found in the datasheet for the magnets).
3. The yoke of ST37 steel with high relative permeability, i.e. we set it to 1000 which is sufficient.

These values are entered in the "Domains" array:

```
# Number of domains and relative permeability respectively
#          vacuum magnets yoke
#          0      1       2
Domains = [1,     1.05,   1000];
```

Every physical surface and codec defined in mesh-model, respectively is defined in the codecs section of the generated parameter file. For our example of a planar magnetron it looks like this:

```
Boundary Outer_Magnet_Top    = {icodec = 1; type = "interface"; val =     0;  domain = 0;
neighbourdomain = 0;};
Boundary Outer_Magnet_Bottom = {icodec = 2; type = "interface"; val =     0;  domain = 0;
neighbourdomain = 0;};
Boundary Inner_Magnet_Top    = {icodec = 3; type = "interface"; val =     0;  domain = 0;
neighbourdomain = 0;};
Boundary Inner_Magnet_Bottom = {icodec = 4; type = "interface"; val =     0;  domain = 0;
neighbourdomain = 0;};
Boundary Magnet_Skin         = {icodec = 5; type = "interface"; val =     0;  domain = 0;
neighbourdomain = 0;};
Boundary Yoke                = {icodec = 6; type = "interface"; val =     0;  domain = 0;
neighbourdomain = 0;};
```

By default every surface is an interface type which is valid for the magnetostatic field computation. Now step by step. Permanent magnets have a homogenous magnetization, resulting in a magnetic positive and negative pole, respectively. In our parameter file we define those poles by inserting either a positve or negative magnetic remanence value to accordant placeholder.

Step 2.2: Set "val"-parameter of accordant surfaces to signed magnetic remanence value, i.e. +1.41 and -1.41 in our case.

```
Boundary Outer_Magnet_Top    = {icodec = 1; type = "interface"; val = -1.41;  domain = 0;
neighbourdomain = 0;};
```

```
Boundary Outer_Magnet_Bottom = {icodec = 2; type = "interface"; val =  1.41;  domain = 0;
neighbourdomain = 0;};
Boundary Inner_Magnet_Top    = {icodec = 3; type = "interface"; val =  1.41;  domain = 0;
neighbourdomain = 0;};
Boundary Inner_Magnet_Bottom = {icodec = 4; type = "interface"; val = -1.41;  domain = 0;
neighbourdomain = 0;};
Boundary Magnet_Skin         = {icodec = 5; type = "interface"; val =     0;  domain = 0;
neighbourdomain = 0;};
Boundary Yoke                = {icodec = 6; type = "interface"; val =     0;  domain = 0;
neighbourdomain = 0;};
```

The next and final step is to define the surface dependencies of the mesh-model, i.e. to which domain a surface belongs and which domain adjoins it. For this you have to know, that the counting start with 0 and not with 1. Take for example the surfaces of the inner magnet bar. The inner magnet surfaces belong to domain 1, i.e. the second entry in "Domains"-array. However, the bottom surface adjoins the yoke (domain 2) whereas the top and the skin surfaces adjoin the vacuum (domain 0).

Step 2.3: Set "domain"- and "neighbourdomain"-parameters to define accordant dependencies.

```
Boundary Outer_Magnet_Top    = {icodec = 1; type = "interface"; val = -1.41;  domain = 1;
neighbourdomain = 0;};
Boundary Outer_Magnet_Bottom = {icodec = 2; type = "interface"; val =  1.41;  domain = 1;
neighbourdomain = 2;};
Boundary Inner_Magnet_Top    = {icodec = 3; type = "interface"; val =  1.41;  domain = 1;
neighbourdomain = 0;};
Boundary Inner_Magnet_Bottom = {icodec = 4; type = "interface"; val = -1.41;  domain = 1;
neighbourdomain = 2;};
Boundary Magnet_Skin         = {icodec = 5; type = "interface"; val =     0;  domain = 1;
neighbourdomain = 0;};
Boundary Yoke                = {icodec = 6; type = "interface"; val =     0;  domain = 2;
neighbourdomain = 0;};
```

Step 3: Start computation. Now the mesh-model is fully defined for computation. To start the magnetic field computation with one process just enter:

```
rvmmpi -bem 1 chamber.par
```

Enter the accordant number of processes you want to use for compuation behind "-bem" tag. Computation becomes significantly faster if you use more than one process.

The magnetic field computation is performed in two steps. First, the Poisson equation is solved within the magnetic arrangement. Second, the actual magnetic field vectors are computed for each grid node within the region of interest. Both steps can be parallelized and thus considerably sped up by invoking multiple instances of **bem**.

Very important: During the first step, there is some output written which may look like follows:

```
 ******************************
 *** BEM B-Field Calculation ***
 ******************************
MASTER <- {BEM  1}:     [MESSAGE]        Matrix generation...
MASTER <- {BEM  1}:     [MESSAGE]        Number of elements: 4944
MASTER <- {BEM  1}:     [MESSAGE]        domain:   0 permeability:     1 elements:  4258
MASTER <- {BEM  1}:     [MESSAGE]        domain:   1 permeability:  1.05 elements:  3316
MASTER <- {BEM  1}:     [MESSAGE]        domain:   2 permeability:  1000 elements:  2314
MASTER <- {BEM 10}:     [MESSAGE]        domain:   2 checksum:  1.0319e-13
MASTER <- {BEM  6}:     [MESSAGE]        domain:   1 checksum: -5.0333e-13
MASTER <- {BEM  1}:     [MESSAGE]        domain:   0 checksum:  1.2566e+01
MASTER <- {BEM  1}:     [MESSAGE]        Matrix diagonalization...
```

Very important (continued): Please look carefully at the numbers given after **checksum**: These are angular sums of all domains and should equal either **1.2566e+01** (=4π, for the vacuum domain) or 0 (for all other domains). If there is a finite value between 0 and 4π instead, **this is a certain sign, that there are some errors within the magnetic mesh file or the magnetic template definition**. In this case, please check the direction of all normal vectors and the completeness of all surfaces with GMSH and check further if the magnetic template definition in **chamber.bem** is consistent. If the numbers of **checksum** are not either 4π or 0 (values in the 10^{-10} region due to round-off errors are ok), the result of the magnetic field computation will be erroneous.

Maximum model size

In principle the size of the magnetic field you can compute is only limited by your computing resources, more precisely your main memory (RAM). As a rule of thumb the memory needed is:

```
50 * (number of mesh elements)² bytes
```

e.g. for a model with 50000 elements you need about 125 GB of RAM. The simulation work and thus the required memory can be distributed to multiple computing nodes.

Validation

Validation against measurements

A PK750 magnetron in its strong and standard version has been measured by Von Ardenne Anlagentechnik (VAAT). The BEM compuation is compared with these measurements. The magnets used for the strong magnetron are **VACODYM 510 HR** from VAC Vakuumschmelze with a remanence of 1.40 T. Additonally we use different permeability values for the yoke material to check the effect of magnetic saturation on the results.

Parameter	Setting	Unit
Remanence (strong)	1.40	[T]
μ_r-Vacuum	1	[-]
μ_r-Magnets	1.05	[-]
μ_r-Yoke (ST 37 steel)	1000	[-]

Table 1: Parameter setting for the balanced magnetron setup with strong magnets.

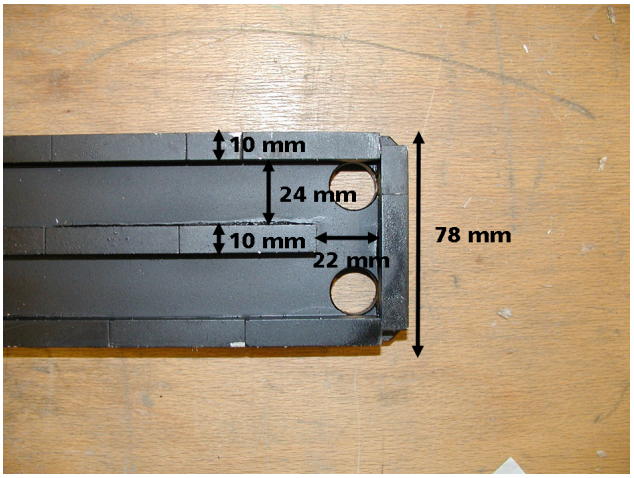
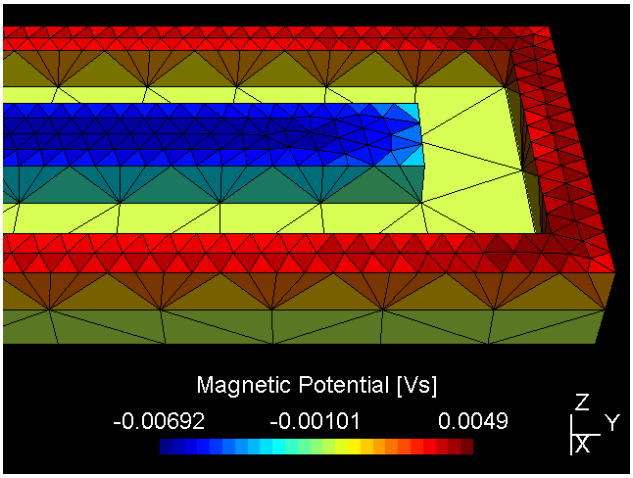
Exemplary PK750 magnetron layout (unbalanced)	FE mesh (balanced) for numerical field computation
	

Figure 1: Setup of an unbalanced PK750 magnetron. The values have been assigned to the FE mesh for the BEM computation.

The results are in good aggreement with the measurements as seen in the pictures below. Discrepancies should rather be related to remanence deviations of the real magnets.

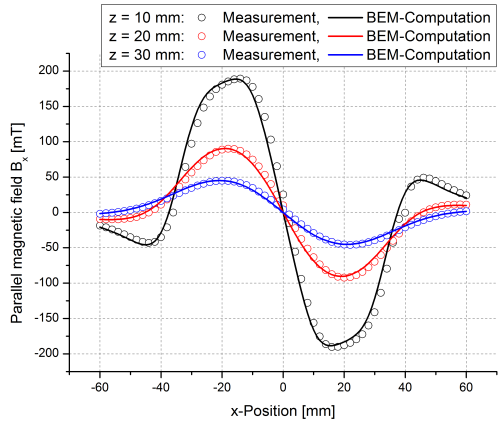
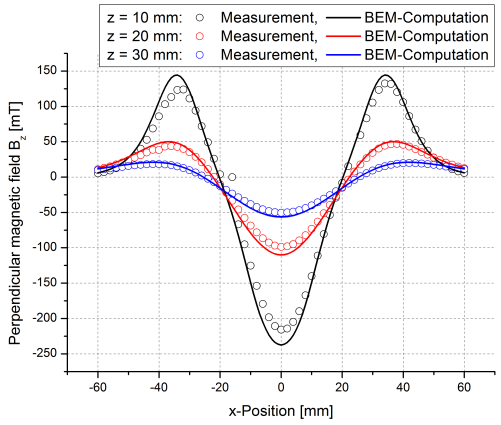
Bx-field at z = 10, 20 and 30 mm and y = 400 mm	Bz-field at z = 10, 20 and 30 mm and y = 400 mm
	

Figure 2: Comparison of the BEM computation results with the VAAT measurements of a strong, balanced PK750 magnetron.

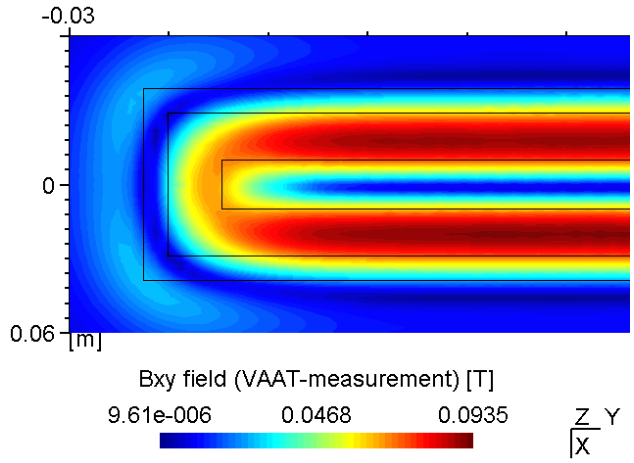
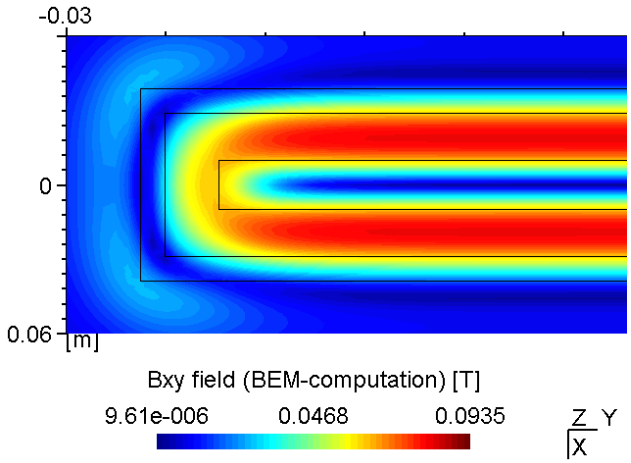
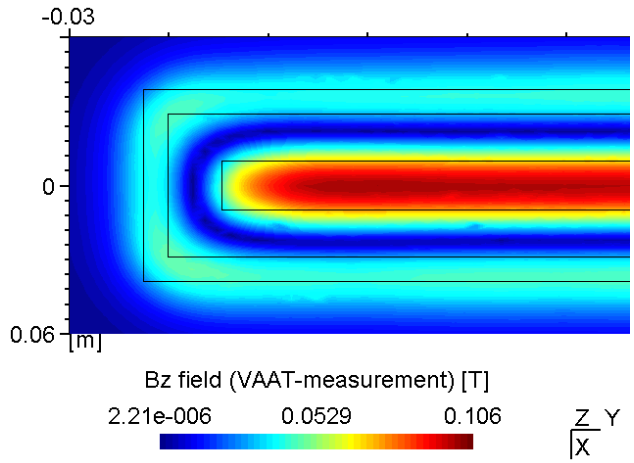
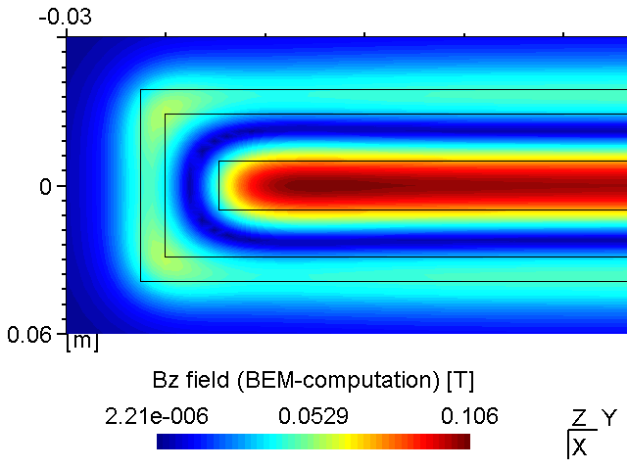
Measured bxy-field at z = 20 mm	BEM-computed bxy-field at z = 20 mm
	
Measured bz-field at z = 20 mm	BEM-computed bz-field at z = 20 mm
	

Figure 3: Comparison of the BEM computation results with the VAAT measurements of a strong, balanced PK750 magnetron.

Accuracy issues

Using invalid (constant) material parameters

For the materials' permeability we use constant values, e.g. μ_r = 1000 for ST 37 steel. This approximation is valid as long as the material is beyond magnetic saturation. If this is not the case the effective permeability is somewhere below of the unmagnetized state. This is still an uncertainty in our BEM calculations and should be resolved soon.

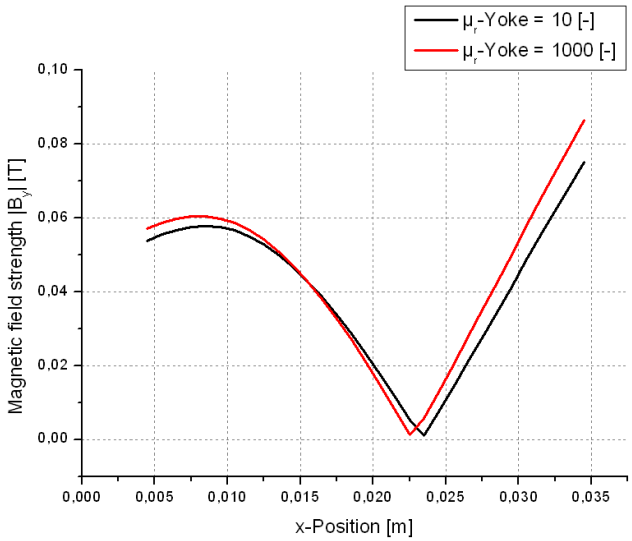
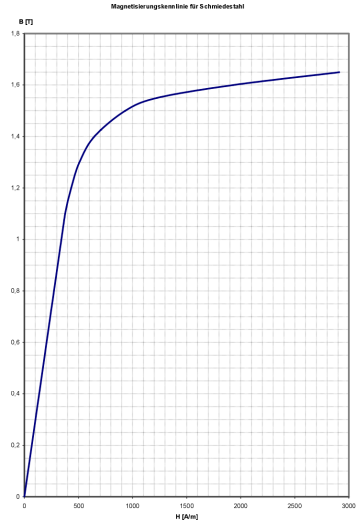
By-field in racetrack zone with μr-Yoke = 10 and 1000, respectively	Magnetization curve of ST37 steel
	

Figure 1: BEM results for the balanced planar magnetron example with different permeability values for the yoke material.

Using low mesh resolution

If the region of interest is far away from the mesh, the mesh resolution itself is not crucial for the accuracy of the magnetic field solution. However, as seen below in certain cases the error may be small but not negligible.

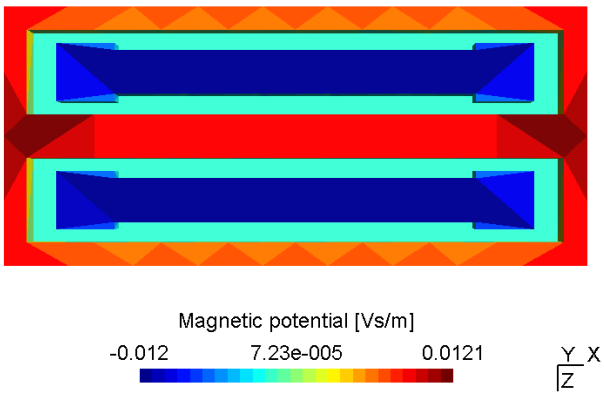
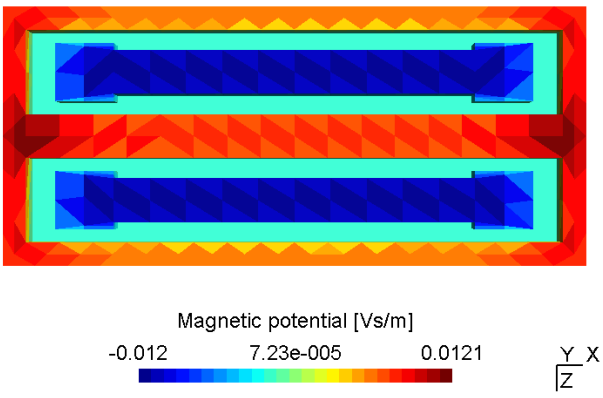
Low mesh resolution	High(er) mesh resolution
	

Figure 1: Magnetic potential and Dirichlet boundary conditions, respectively for different mesh resolutions.

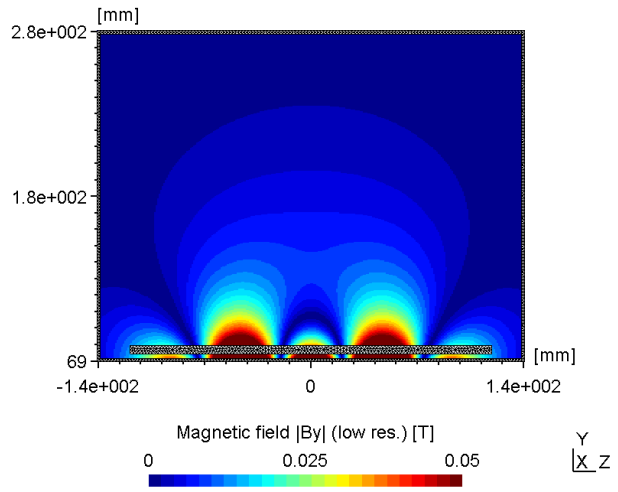
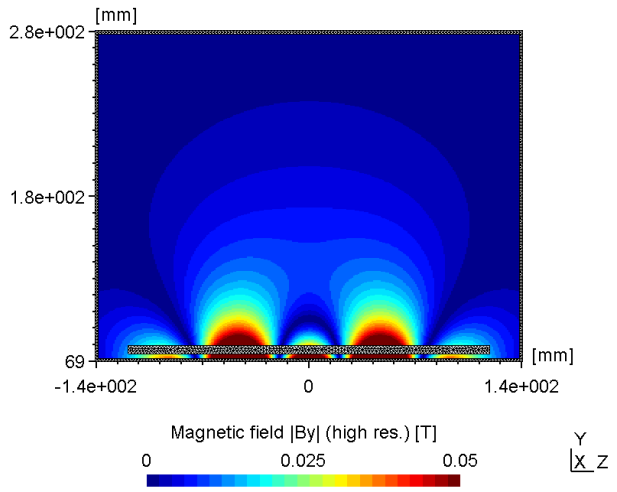
Low mesh resolution	High(er) mesh resolution
	

Figure 2: Magnetic field strength in y-direction resulting from the BEM for different mesh resolutions.

49 DSMC/PIC-MC iteration

Problem

While for PIC-MC plasma simulations usually a homogeneous background gas pressure is assumed, it may be desired to use a "real" background gas distribution. However there are a few issues related to that task:

- The DSMC simulation uses time steps of typically 10^{-5}...10^{-6} s and may reach total simulation times in the range of seconds on the one hand.
- On the other hand, typical time steps in PIC-MC are less than $5 \cdot 10^{-11}$ s. Under these conditions the gas flow can not be simultaneously simulated: It would take ages until some noticeable development in the background gas pressure distribution shows up.
- Another problem is related to the wall boundary conditions: In PIC-MC e. g. a conductive chamber wall has to be defined as a single physical surface in order to be on the same potential everywhere. However DSMC related features such as gas inlets or pumping surfaces have different physical IDs - this would confuse the electric field solver in a PIC-MC simulation.

Solution

The way to circumvent the above mentioned issues is to perform *time scale splitting*. For this purpose we need the following:

- One mesh file `simcase-DSMC.msh` of the geometry **with physical surfaces defined according to DSMC gas flow simulation**
- One mesh file `simcase-PICMC.msh` of the geometry **with physical surfaces defined according to PIC-MC plasma simulation**
- Two separated parameter files for DSMC and for PIC-MC simulation. Let us assume, these files are called `simcase-DSMC.par` and `simcase-PICMC.par`. The `WORKMODE` must be set to 0 in case of the DSMC file and to 1 in case of the PIC-MC file.

The solution is to perform the DSMC simulation first. After completion of the DSMC simulation, the neutral pressure distribution can be loaded into the PICMC simulation case. In this first stage, only the relevant neutral species are required, while the charged species come into play in the subsequent PICMC simulation[1].

It is recommended to use a rather low `NREAL_xxx` factor for the neutrals[2]: With higher number of simulation particles the statistics gets improved. Since the neutral particles will act as ionization sources in the plasma simulation, a well-distributed statistics is important in order to avoid artificial inhomogeneities in the ion generation profile.

For the PICMC plasma simulation case it is important to consider that all "physical surfaces" must form closed three-dimensional objects. If source surfaces (e.g. gas inlets, sputter sources etc.) are still needed they must be defined as separate membranes. Finally, we need a way of including the DSMC computed gas distribution into the plasma simulation. This is done via the optional block `User_Block`, which can be specified in the parameter file. In the following, a working example is given (assuming that the PICMC case is in `simcase-picmc.par` and the DSMC case in `simcase-dsmc.par` and both are in the same folder).

```
block User_Block: {
```

```
  if INITMODE==0 then {
    out "** Reading neutral gas distribution **\n";
    S.read_checkpoint("simcase-dsmc/checkpoint/particles_Ar.bin", "particles", "Ar");
    S.read_checkpoint("simcase-dsmc/checkpoint/particles_O2.bin", "particles", "O2");
    S.read_checkpoint("simcase-dsmc/checkpoint/averages_Ar.bin",  "averages",  "Ar");
    S.read_checkpoint("simcase-dsmc/checkpoint/averages_O2.bin",  "averages",  "O2");
  } else {
    out "** Skipping User_Block since this is a restart **\n";
  }
};
```

A few explanatory notes follow:

- The variable `INITMODE` is zero, if the simulation case is started for the first time. The `if` statement assures that the particle initialization procedure is performed only once and will be ignored on restarts.
- In the initialization routine, first the neutral gas distribution - Ar and O2 in the example - is read from the case `simcase-dsmc`.
- The initial charge distribution can be defined in the species definition of the PICMC parameter file. Since only the neutral gas checkpoint files are read, the initialization of the charge distribution is untouched.

In case you want to use the surface chemistry defined in your DSMC case during your PICMC calculations, you also have to read in the checkpoints for the materials with:

```
S.read_checkpoint("simcase-dsmc/checkpoint/material_TiO2.bin",   "material", "TiO2"  );
S.read_checkpoint("simcase-dsmc/checkpoint/material_Ti_met.bin", "material", "Ti_met");
```

If you iterate many times between DSMC and PICMC you maybe also want to reuse a previously simulated charge distribution:

```
S.read_checkpoint("simcase-picmc/checkpoint/charges.bin",        "charges",  0       );
```

Footnotes

1. This was different in PICMC version <=2.3.7.x, where the particle information of all species went into a single dump-file. ↩

2. ...at least during a couple of time steps at the end of the DSMC simulation ↩

50 Electron movement

Problem

The electron movement can be visualized via the 'POSITION' plots. In order to properly resolve the movement trajectory of typical electrons in magnetron plasma discharges, the time interval of the POSITION plot should be set to 0.1 ns.

The problem is that there are usually too many simulation particles and thus, the POSITION plots are difficult to visualize.

One possibility is to put only a few single electrons in an existing magnetic field configuration, however these electrons are not representative for electrons within a magnetron plasma discharge.

Solution

Use of adapted species and cross section file

The solution is to start from an existing plasma simulation and show only a small fraction of the electrons as "representatives". Additional "representative" electron species are to be created in the which are to be plotted in the POSITION plot. For that purpose, the file `species_definition.r` needs to be copied into the local folder and extended in the following way:

```
block GenSPEC : { float mr, charge, is_frozen = 0; };

Record SPECS = {
  Record e          = GenSPEC + { mr = 1/1823;   charge = -1.0; };
  Record eSec       = GenSPEC + { mr = 1/1823;   charge = -1.0; };
  Record eGas       = GenSPEC + { mr = 1/1823;   charge = -1.0; };

  ...
```

Here, the original electron species `e` is copied to the identical species `eGas` and `eSec`. The idea is to modify the gas phase reactions in a way that in a small percentage of ionizations, `eGas` is created instead of `e`. Analogously, the surface reactions are modified in a way that in a small percentage of cases, `eSec` are generated as secondary electrons instead of `e`.

Obviously, the cross section file `sigma.r` must be modified to include the additional species `eGas` and `eSec` which should behave the same way as regular electrons `e`:

```
...
  # ========================================================
  # Plasma interactions of electrons and argon [A.V. Phelps]
  # ========================================================
  block e_Ar -> {
    block list[] -> [
      CG.FCTeV_Entry + {
        MSG="$('e' %11s) + $('Ar' %11s) -> e + Ar (Elastic Momentum Transfer)";
        sigma-> (eV>0) ?   (abs(6/(1+(eV/0.1)+(eV/0.6)^2)^3.3-1.1*eV^1.4/(1+(eV/15)^1.2)/(1+
(eV/5.5)^2.5+(eV/60)^4.1)^0.5)
                        + 0.05/(1+eV/10)^2 + 0.01*eV^3/(1+(eV/12)^6))*1.0E-20 ! MIN_SIGMA;
      },
      CG.FCTeV_Entry + {
        MSG="$('e' %11s) + $('Ar' %11s) -> e + Ar* (Total Excitation)";
```

```
        EV_THRESH = 11.5;
        flag_1 = 0; flag_2 = 1;
        if (defined(Properties.Arex)) then rp_vec = ["Arex"];
        sigma-> (eV>EV_THRESH) ?   0.85*(4E-22*(eV-11.5)^1.1*(1+(eV/15)^2.8)/(1+(eV/23)^5.5)
                                 + 2.7E-22*(eV-11.5)/(1+(eV/80))^1.9) ! MIN_SIGMA;
      },
      # Splitting: 5% probability of generating eGas
      CG.FCTeV_Entry + {
        MSG="$('e' %11s) + $('Ar' %11s) -> 2e + Arplus (Ionization)";
        EV_THRESH = 15.8;
        flag_1 = 0; flag_2 = 1;
        rp_vec = ["e", "Arplus"];
        sigma-> (eV>EV_THRESH) ? 0.98*(9.7E-18*(eV-15.8)/(70+eV)^2 + 6E-22*(eV-15.8)^2*exp(-eV/9))
! MIN_SIGMA;
      },
      CG.FCTeV_Entry + {
        MSG="$('e' %11s) + $('Ar' %11s) -> e + eGas + Arplus (Ionization)";
        EV_THRESH = 15.8;
        flag_1 = 0; flag_2 = 1;
        rp_vec = ["eGas", "Arplus"];
        sigma-> (eV>EV_THRESH) ? 0.02*(9.7E-18*(eV-15.8)/(70+eV)^2 + 6E-22*(eV-15.8)^2*exp(-eV/9))
! MIN_SIGMA;
      }
    ];
  };

  block eSec_Ar -> {
    block list[] -> [
      CG.FCTeV_Entry + {
        MSG="$('eSec' %11s) + $('Ar' %11s) -> eSec + Ar (Elastic Momentum Transfer)";
        sigma-> (eV>0) ?   (abs(6/(1+(eV/0.1)+(eV/0.6)^2)^3.3-1.1*eV^1.4/(1+(eV/15)^1.2)/(1+
(eV/5.5)^2.5+(eV/60)^4.1)^0.5)
                         + 0.05/(1+eV/10)^2 + 0.01*eV^3/(1+(eV/12)^6))*1.0E-20 ! MIN_SIGMA;
      },
      CG.FCTeV_Entry + {
        MSG="$('eSec' %11s) + $('Ar' %11s) -> eSec + Ar* (Total Excitation)";
        EV_THRESH = 11.5;
        flag_1 = 0; flag_2 = 1;
        if (defined(Properties.Arex)) then rp_vec = ["Arex"];
        sigma-> (eV>EV_THRESH) ?   0.85*(4E-22*(eV-11.5)^1.1*(1+(eV/15)^2.8)/(1+(eV/23)^5.5)
                                 + 2.7E-22*(eV-11.5)/(1+(eV/80))^1.9) ! MIN_SIGMA;
      },
      CG.FCTeV_Entry + {
        MSG="$('eSec' %11s) + $('Ar' %11s) -> e + eSec + Arplus (Ionization)";
        EV_THRESH = 15.8;
        flag_1 = 0; flag_2 = 1;
        rp_vec = ["e", "Arplus"];
        sigma-> (eV>EV_THRESH) ? 9.7E-18*(eV-15.8)/(70+eV)^2 + 6E-22*(eV-15.8)^2*exp(-eV/9) !
MIN_SIGMA;
      }
    ];
  };

  block eGas_Ar -> {
    block list[] -> [
      CG.FCTeV_Entry + {
        MSG="$('eGas' %11s) + $('Ar' %11s) -> eGas + Ar (Elastic Momentum Transfer)";
        sigma-> (eV>0) ?   (abs(6/(1+(eV/0.1)+(eV/0.6)^2)^3.3-1.1*eV^1.4/(1+(eV/15)^1.2)/(1+
(eV/5.5)^2.5+(eV/60)^4.1)^0.5)
                         + 0.05/(1+eV/10)^2 + 0.01*eV^3/(1+(eV/12)^6))*1.0E-20 ! MIN_SIGMA;
      },
      CG.FCTeV_Entry + {
        MSG="$('eGas' %11s) + $('Ar' %11s) -> eGas + Ar* (Total Excitation)";
        EV_THRESH = 11.5;
        flag_1 = 0; flag_2 = 1;
        if (defined(Properties.Arex)) then rp_vec = ["Arex"];
```

```
        sigma-> (eV>EV_THRESH) ?   0.85*(4E-22*(eV-11.5)^1.1*(1+(eV/15)^2.8)/(1+(eV/23)^5.5)
                                 + 2.7E-22*(eV-11.5)/(1+(eV/80))^1.9) ! MIN_SIGMA;
      },
      CG.FCTeV_Entry + {
        MSG="$('eGas' %11s) + $('Ar' %11s) -> e + eGas + Arplus (Ionization)";
        EV_THRESH = 15.8;
        flag_1 = 0; flag_2 = 1;
        rp_vec = ["e", "Arplus"];
        sigma-> (eV>EV_THRESH) ? 9.7E-18*(eV-15.8)/(70+eV)^2 + 6E-22*(eV-15.8)^2*exp(-eV/9) !
MIN_SIGMA;
      }
    ];
  };
  ...
```

In this modified `my_sigma.r` the reaction path between electrons and Argon are copied also for `eGas` and `eSec`, because the latter should behave the same way as regular electrons `e`. Additionally, in the reaction `e_Ar` the ionization is split into two reactions with 98% and 2% probability, respectively. In the first ionization reaction, regular electrons are created as reaction product, however the alternative ionization branch creates `eGas` particles.

Modified parameter file including creation of POSITION plots and representative secondary electrons

The modified parameter file shall produce POSITION plots in a high plot frequency and also include the production of "representative secondary electrons" `eSec` in the surface reactions.

To start from an existing simulation the following files should be copied into the folder of the new simulation case:

1. The whole checkpoint/* folder
2. The file `last_shutdown.txt`

The global variables of the PAR file should be modified to include the small plot interval of 0.1 ns:

```
# Parameter file for the PICMC gas discharge and DSCM gas flow simulation

# ============================================
# Global variables
# ============================================
DT       = 1e-11;    # Time step width
TSIM     = 2e-5;     # Total simulation time in seconds
TAVG     = 1;        # Time averaging mode (0 = off, 1 = on)
NPLOTS   = 200000;   # 0.1 ns interval
NDUMPS   = 1;        # Number of particle data dumps
WORKMODE = 1;        # Work mode (0 = DSCM gasflow simuation, 1 = PICMC plasma simulation, 2 =
electric field solver)
T0       = 300;      # Initial particle and border temperature [K]
PSOLL    = 2;        # Aimed power dissipation in Watt (plasma simulation only)
PSAMPLE  = 1e-7;     # Sampling time for power dissipation (plasma simulation only)
VPMAX    = 700;      # Maximum voltage for power control (default = 1000 volt)

# ============================================
# Specify species list
# Check SIM-Wiki for list of available species and reactions
# ============================================
SPECIES  = ["Ar", "Arplus", "e", "eSec", "eGas"];     # List of species strings

NREAL_Ar     = 5e9;   # Default scale factor for Ar
```

```
P0_Ar        = 0.8;    #
SEY          = 0.12;   # Nb Secondary electron yield

# Comment/uncomment accordant to defined species:
NREAL_Arplus = 5e4;    # Default scale factor for Arplus
P0_Arplus    = 1e13;   # Default density of initial Arplus distribution [Number/m^3]

NREAL_e      = 5e4;    # Default scale factor for e
P0_e         = 1e13;   # Default density of initial e distribution [Number/m^3]

NREAL_eSec   = 5e4;    # Default scale factor for e
P0_eSec      = 0;   # Default density of initial e distribution [Number/m^3]
NREAL_eGas   = 5e4;    # Default scale factor for e
P0_eGas      = 0;   # Default density of initial e distribution [Number/m^3]
# ================================================
# 3D data extraction
# Examples: NUMBER = ["Ar", "e"]; FIELD = ["PHI", "E"];
# ================================================
NUMBER      = [];
DENSITY     = [];
PRESSURE    = [];
TEMPERATURE = [];
ENERGY      = [];
VELOCITY    = [];
ABSORPTION  = [];
# POSITION plots can get very large, so handle with care!
POSITION    = ["eSec", "eGas"];
# "PHI" = electric potential, "PHI_MEAN" = time averaged electric potential, "E" = electric field
FIELD       = [];

# For COVERAGE and DEPOSITION fill in required surface materials:
COVERAGE    = [];
DEPOSITION  = [];
...
```

First, there are 200000 plots for a physical time interval of 20 µs yielding a plot interval of 0.1 ns.

Second, the species list is extended by the representative species `eGas` and `eSec`. They get the same NREAL values as regular electrons and a starting density of zero (because they should be created by the gas and surface reactions in order to represent real electrons in the given plasma!).

In the plot section all regular plots are disabled and only the POSITION plots for `eSec` and `eGas` are enabled. This is important due to the high plot frequency - otherwise your harddisk space will be quickly flooded!

In the surface reactions, we have to add a branch with a low probability to create "representative" secondary electrons `eSec`:

```
...
block Ar_Plasma_ext: {
  show_reaction_message("Simple Ar plasma-wall model (legacy version); SEY = $(PAR.SEY)");

  add_plain_reaction("Arplus", 1, 0.8, ["Ar", "e"], [1, PAR.SEY]);
  set_emission_energy("e", 0, 10, 100);
  add_absorption("e", 1.0);

  add_plain_reaction("Arplus", 1, 1, ["Ar", "eSec"], [1, PAR.SEY]);
  set_emission_energy("eSec", 0, 10, 100);
  add_absorption("eSec", 1.0);

  add_absorption("eGas", 1.0);
};
...
```

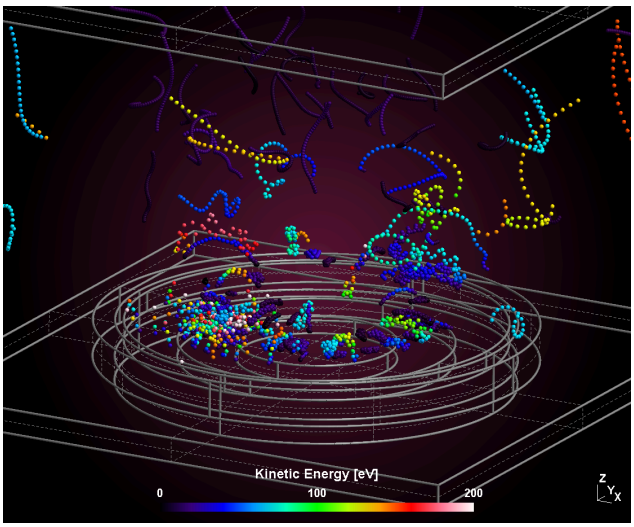
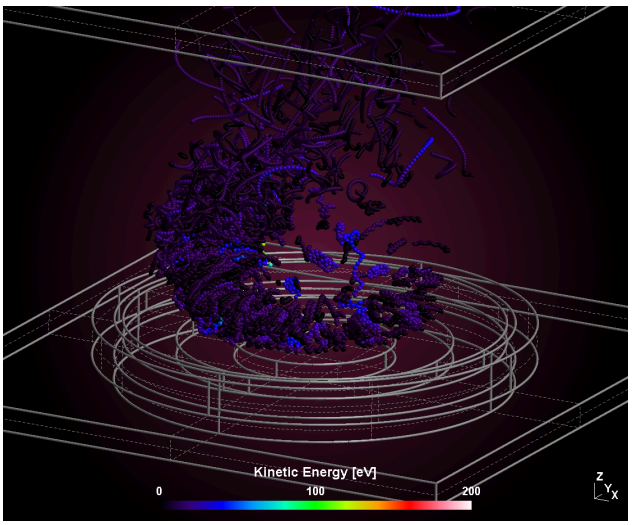
This looks like a regular Ar plasma chemistry, where `Arplus` ions hitting the surface are neutralized and create secondary electrons. However the reaction branch for creating secondary electrons is doubled. In the first reactionm a reaction probability of 80% is given, while the second reaction branch has 100% probability[1].

Depending on the simulation case and electron density etc. it might be necessary to adapt the reaction branching probabilities in order to create the desired amount of "representative" electrons.

Example

The example below shows trajectories of secondary electrons created at the target surface (left) and bulk electrons created by gas phase electrons (right). As can be seen, both groups of electrons differ strongly in their mean kinetic energy. Especially secondary electrons, which escape the magnetic confinement can gain the full target voltage as kinetic energy, while bulk electrons are equilibrated by numerous gas collisions and have some mean kinetic energy in the order of 5 eV.

The example is taken from a 3D model of a circular sputtering target. The trajectories are shown for a time interval of 10 ns with a plot interval of 0.1 ns (100 positions), respectively.

Secondary electrons	Gas electrons
	

Footnotes

1. Please note that multiple surface reactions are checked from top-down, i.e. if the first reaction is executed with 80% probability there is only 20% probability that the second reaction applies. Thus, the 80% - 100% probabilities are effectively translated to 80% - 20% reaction probability of both branches. This is unlike cross section lists, where multiple reaction entries are constantly shuffled during a simulation run in order to avoid such conditional probability issues. ↩

Postprocessing

51 Available output data

Each simulation case requires at least a geometric mesh file and a parameter file (e.g. `simcase.par`), where the simulation conditions, species, plasma-wall interaction etc. is contained. When the simulation is started, all resulting output data is written into a folder `simcase/` which has the same name as the parameter file. The folder is automatically being created if it does not exist.

The output data generated during simulation can be divided into following categories:

Type of data	Resolution	File format	How often
Information on gaseous species	cell resolution	GMSH POS file	`NPLOTS` times during simulation run
Information on wall materials	cell resolution	GMSH POS file	`NPLOTS` times during simulation run
Absorption and Desorption on surfaces	per mesh element	GMSH POS file	`NPLOTS` times during simulation run
Electromagnetic field related	node resolution	GMSH POS file	`NPLOTS` times during simulation run
Collision statistics	quad resolution	GMSH POS file	`NPLOTS` times during simulation run
Time-dependent logging of average surface data	Integrated over physical surface	ASCII TXT file	`NCOUNTS` log entries
Time-dependent logging of potentials	Taken for each conductive electrode	ASCII TXT file	`NCOUNTS` log entries
Visualization of cell spacing and domain decomposition	-	GMSH GEO file to be merged with geometric MSH file	At the beginning
Information on used cross sections	-	Postscript / ASCII	At the beginning
Information required for continuation of a simulation run	-	binary files and script files	At the end or shutdown of a simulation run
Dynamic script files	-	ASCII (RIG-VM script format)	At the beginning

52 POS files

Most of the data volume is given as time- and particle averaged values within the cell structure. They are provided as *POS files* which can be viewed and further processed within GMSH.

These data can comprise information on gaseous species as well as on particle absorption and status of walls. Additionally, some data on particle absorption can be also given in the resolution of the surface mesh.

The time-averaging feature is activated by setting `TAVG=1;` at top of the parameter file. By setting `TAVG=0;` momentary snapshots are displayed instead, however in most cases they will be quite noisy or a very large number of particles has to be simulated to get reasonable statistics. Therefore, `TAVG=1;` is the default setting.

The number of time averages within the whole physical simulation time interval is defined in the variable `NPLOTS` at top of the parameter file. The selection of output data by species is defined via lists in the upper part of the parameter file.

The following lists for cell-oriented information are available:

- **Information on gaseous species**
  - `NUMBER` = Number of simulation particles per cell
  - `DENSITY` = Density of simulation particles in physical units [1/m^3]
  - `PRESSURE` = Partial pressure in [Pa]
  - `TEMPERATURE` = Translational kinetic temperature in [K]
  - `ENERGY` = Mean particle energy in [eV]
  - `VELOCITY` = Mean particle velocity in [m/s]. This is a vector plot for three components v_x, v_y, v_z.
- **Information on wall absorption and wall state**
  - `ABSORPTION` = Particle absorption on walls in [1/m^{2}s]
  - `DESORPTION` = Particle desorption from walls in [1/m^{2}s]
  - `COVERAGE` = Surface coverage fraction of a wall material (between 0 and 1)
  - `DEPOSITION` = Deposition rate of a wall material [nm/s]
- **Information on fields**
  - `FIELD` = Quantities concerning electromagnetic field computation

All of these data go into sub folders of the simulation data folder. This behaviour can be suppressed by setting `SINGLE_PLOT_FOLDER=1;` (which can be found in the bottom part of the parameter file).

If resampling is active, the additional folder `number_goal/` contains the target particle number per cell that the run tries to maintain (see Resampling). Comparing it with `number/` shows where the target is not reached.

The filename structure is `<type>_<species>_<time>ms.pos`, e.g. the Ti absorption after 50 ms will be stored in a file `absorption/absorption_Ti_50.00ms.pos`. These files can be opened and further processes in GMSH.

The list `COLLISION` enables the plotting of collision statistics on cell resolution for every used cross section defined in the `sigma.r` file. By putting a species name in that list all reactions involving this species are written in a separate file including the energy threshold to identify the corresponding cross section. The files are written to the `collisions/` folder; cells in which no collision took place are

included as well. The sum over all cells for a specific collision matches the total collision rate written in the `collision.txt` files present in the same folder.

Additionally, with the list `MESHDATA` particle absorption information is given in the resolution of the surface mesh. The filename for mesh absorption data for Ti after 50 ms will be e.g. `meshdata/meshdata_Ti_50.00ms.pos`. These data files contain multiple plots:

- Absorption
- Desorption
- Mean energy of absorbed particles
- Mean angle of incidence α
- Mean cos(α)

The `POSITION` list has to be handled with care. If you put a species in there, for all particles of that species the position and energy is plotted. This plot is used normally with a low number of particles, e.g. a high NREAL or individually distributed particles with the `Init_Particles` block. An example would be the movement of single electrons in the electromagnetic field.

A typical setting of these lists may look as follows:

```
# ================================================== #
# 3D data extraction                                 #
# Examples: NUMBER = ["Ar", "e"]; FIELD = ["PHI", "E"]; #
# ================================================== #
NUMBER      = [];
DENSITY     = [];
PRESSURE    = ["Ar", "O2", "Ti"];
TEMPERATURE = [];
ENERGY      = [];
VELOCITY    = ["Ar", "O2", "Ti"];
COLLISION   = [];
ABSORPTION  = ["Ti"];
DESORPTION  = ["Ti"];
MESHDATA    = ["Ti"];
# POSITION plots can get very large, so handle with care!
POSITION    = [];
# "PHI" = electric potential, "PHI_MEAN" = time averaged electric potential, "E" = electric field
# "JICP" = source current density, "EICP" = current induced electric field, "BICP" = current
induced magnetic field
FIELD       = [];

# For COVERAGE and DEPOSITION fill in required surface materials:
COVERAGE    = ["TiO2"];
DEPOSITION  = ["Ti_metal", "TiO2"];
```

With these settings, the pressure and velocity are plotted for three species, Ar, O2 and Ti, respectively. For Ti, additionally the absorption plot in cell resolution and absorption data in mesh resolution are generated. Finally, for wall materials `TiO2` and `Ti_metal` the deposition rate is plotted together with the surface coverage fraction for `TiO2`.

53 Cell structure

Especially for complex geometries with non trivial domain decomposition, a visualization of cell volumes, cell surfaces and the domain decomposition can be helpful for debugging purposes.

Therefore the following files are regularly created upon each start of a simulation case:

- `domain_decomposition.pos` - Visualization of the volume segment structure. In GMSH this can be best viewed when merged together with the MSH file of the geometric model. An example for a domain decomposition plot is given in 3D gas flow - optimized domain decomposition.
- `cell_volume.pos` - Effective cell volume per cell. Cells within solid walls should have zero volume. Cells within the free gas volume should have a volume according to their dimension (length, width, height, $Vol = L \times W \times H$). Cells partially intersecting with a solid wall should have a smaller volume depending on the intersection fraction.
- `cell_surfaces.pos` - Effective surface of intersecting walls. This should be only non-zero for cells, which are partially intersecting with a wall. For walls which are located *exactly* between two cells, at least one of both cells should show the effective wall surface.

If e.g. the `cell_volumes.pos` plot shows non-zero volume, where a solid wall is expected, this gives a hint about possible mistakes in the geometric model such as a missing or doubly selected single surface.

54 Node structure

According to the "leap frog" algorithm used for the PIC-MC computation, the electric and magnetic field / potentials are computed per nodes rather than per cell. A cell "node" is one of the corner points of the cell, i.e. a rectangular cell has 8 nodes. The nodes are shared with the neighboring cells.

The file format is GMSH POS. However the node oriented data have one element more in each direction. This may cause issues in postprocessing together with cell oriented data. For an example, in the RIG-VM postprocessing routines cell oriented and node oriented data shall be processed in separate postprocessing objects.

The choice of electric / magnetic field and potential output can be done within the list `FIELD` in the parameter file. Possible choices are

- **Capacitively coupled plasma (CCP)**
  - `"PHI"` - Electric potential (momentary snapshot)
  - `"PHI_MEAN"` - Electric potential, time averaged over the last time averaging period
  - `"E"` - Electric field as gradient of PHI (momentary snapshot)
- **Inductively coupled plasma (ICP)**
  - `"JC"` - Primary current, i.e. coil surface current density
  - `"JE"` - Secondary current, i.e. electron current (convection current) in plasma
  - `"EC"` - Primary current induced electric field
  - `"EE"` - Secondary current induced electric field
  - `"BC"` - Magnetic field as curl of EC
  - `"BE"` - Magnetic field as curl of EE

In the following example, the time averaged electric potential and the electric field are plotted:

```
FIELD = ["PHI_MEAN", "E"];
```

55 Surface logging

Particle balance at surfaces

During simulation, particles are absorbed and desorbed from surfaces and/or flowing through transparent membranes. In order to better track the current state of the simulation run, these processes are time averaged over physical surfaces and put into ASCII logfiles. The following log files are automatically created during simulation:

- `absorption_log.txt` - Particle absorption on surfaces
- `desorption_log.txt` - Particle desorption from surfaces
- `transmission_log.txt` - Particle flow through transparent membranes
- `pressure_log.txt` - Kinetic pressure due to particle impingement on surfaces

By default, 100 log entries are created per simulation run. This number can be modified with the global variable `NCOUNTS` in the parameter file, e.g.

```
NCOUNTS = 1000;
```

Each of the log files has the physical time in the first column. The other columns contain particle fluxes / pressures according to the headlines given in the file. The choice of columns is performed automatically according to the absorption / desorption mechanisms defined within the wall reactions.

Depending on if the particles are charged/neutral, absorption, desorption and transmission are given in [A] / [sccm]. Both units are proportional to the total number of particles per second:

- $1\,\mathrm{sccm} = 4.47796 \times 10^{17}\,\mathrm{particles/s}$
- $1\,\mathrm{A} = 1/\mathrm{q} = 6.2415 \times 10^{18}\,\mathrm{particles/s} = 13.9382\,\mathrm{sccm}$

For the kinetic pressure p_{kin}, the average momentum transfer on surfaces by particle collision is accumulated:

$p_{kin} = \frac{1}{A \times \delta t} \sum_i 2 m_i v_{\perp i}$

The variables are

- A = Total surface area of physical surface
- δt = Sampling time interval
- m_i = mass of i'th particle
- $v_{\perp i}$ = normal velocity of i'th particle

This only works correctly, if the whole area A of the physical surface is in contact with the gas volume. If there is additional area which is not in contact with the gas, the pressure number in the log file is normalized with respect to the whole area and will therefore be lower than the actual pressure in the gas volume.

For transparent membranes pressure logging currently is omitted because the contribution of the net flow *through* a membrane has to be handled separately, which is not yet working properly.

Potentials of conductive electrodes

The electric potentials for conductive electrodes are being logged into a TXT file `potentials_log.txt` as a function of time. The time resolution is the same as for the particle-balance related log files such as `absorption_log.txt` etc. and can be controlled via the `NOUNTS` parameter. There are the following cases of electrodes:

Dielectric insulating electrodes Since the electric potential on dielectrics is not unique, such surfaces are ignored. **Electrodes with fixed voltage** The time-dependent evaluation of the `vf` parameter is logged. **Single floating electrodes** The floating potential is logged. **Double floating electrodes with relative bias** The absolute potentials of both electrodes are logged.

RF electrode with self bias	**Only the self bias is logged.**

56 Checkpoints

If a simulation run is finished or shut down via the `touch shutdown` command, the following data is created for restart and continuation of the simulation case later:

- `checkpoint/particles_<species>.bin` - Coordinates and velocities of all simulation particles
- `checkpoint/averages_<species>.bin` - Averaged data (density, mean velocity, mean square velocity) in cell resolution.
- `checkpoint/material_<wall_species>.bin` - Averaged data on surface coverage and deposition rate on walls.
- `checkpoint/charges.bin` - Charge per grid node, written for `WORKMODE = 1`.
- `last_shutdown.def` - ASCII file containing the last iteration step number, the last time step, the length of the current plot interval and the voltage amplitude(s) in case of a plasma simulation. The master reads it back to start a continued run at the stored step.
- `profile.txt` - ASCII file containing the information on the CPU load of each rectangular segment of the simulation volume. Upon restart this information is used for optimizing the load balance of the total CPU load amongst the individual processes. This file can be deleted if the load balancing step shall be omitted.

How the files are written and how a run is continued from them is described in Checkpoints and continuation, the binary layout in Checkpoint file format. Two tools for translating and combining a particle checkpoint are described under Checkpoint tools.

57 Checkpoint file format

This page describes the binary layout for reading or writing the checkpoint files outside PICMC — for your own postprocessing script, a converter, or for checking what a file contains. Which files exist, what they are for and how a run continues from them is described on Checkpoints and Checkpoints and continuation.

The layout below is the one of release 26. The files carry no magic number and no version field, so a checkpoint written by release 25 or older has to be read with the layout of that release — it cannot be parsed with the description on this page.

Conventions. All values are little endian. The records are the C structs of the code, written as a whole on x86-64: `int` is 4 bytes, `float` 4 bytes, `double` 8 bytes, and the padding bytes are part of the file.

Common header

Every checkpoint file starts with

- 1 `int` — number of quads in the file

followed by one block per quad, in the order the quads appear in the domain decomposition. A quad is a rectangular part of the simulation volume with its own cell grid; a reader does not need the mesh, every block can be read on its own.

Particles: particles_<species>.bin

Per quad, a header of 64 bytes:

offset	type	meaning
0	6 × `double`	corner `x0`, `y0`, `z0` of the quad, then its lengths `lx`, `ly`, `lz`
48	3 × `int`	number of cells `nx`, `ny`, `nz`
60	1 `int`	number of particles `n` in this quad

followed by `n` particle records of 64 bytes each:

offset	type	meaning
0	3 × `double`	position relative to the quad corner `x0`, `y0`, `z0`
24	3 × `double`	velocity in m/s
48	1 `float`	`timeleft`, time left in the current time step, in s
52	4 bytes	padding, content undefined
56	1 `double`	`weight`, the number of real particles this superparticle stands for

Until release 25 the record was 56 bytes and the number of real particles was stored once per quad, as an additional `double` between the cell numbers and the particle count. Release 26 moved it into every particle record.

Averages: averages_<species>.bin

Per quad, a header of 60 bytes: the same 6 `double` corner and lengths, then the 3 `int` cell numbers — without a particle count. The number of records is `nx*ny*nz`, one per cell, with x running fastest:

offset	type	meaning
0	1 `int`	`count`, number of samples taken
4	4 bytes	padding, content undefined
8	1 `double`	`weight`, summed statistical weight of the samples
16	1 `double`	sum of the x velocity over all samples
24	1 `double`	sum of the y velocity
32	1 `double`	sum of the z velocity
40	1 `double`	sum of the squared velocity

These are raw sums, not the means that end up in the plots: the mean velocity of a cell is the sum divided by `weight`. The sums of the squared components (`vx²`, `vy²`, `vz²`) are not stored, so those terms start from zero after a restart.

Charges: charges.bin

Written for `WORKMODE = 1`. Per quad:

- 1 `int` — number of charged nodes that follow

then that many records of 32 bytes:

offset	type	meaning
0	3 × `double`	absolute node coordinates `x`, `y`, `z`
24	1 `double`	charge on that node, in C

Only nodes that carry a charge are written, so the record count differs from quad to quad. Nodes on the boundary of a quad are shifted by twice the grid tolerance towards the outside — a node shared by two quads therefore does not appear at exactly the same coordinate in both blocks.

Materials: material_<material>.bin

One file per material. Per quad, a header of 60 bytes like the averages file, then `nx*ny*nz` records of 16 bytes, one per surface cell:

offset	type	meaning
0	1 `double`	`coverage` of the material in that cell
8	1 `double`	`deposition` in that cell

Checking a file before reading it

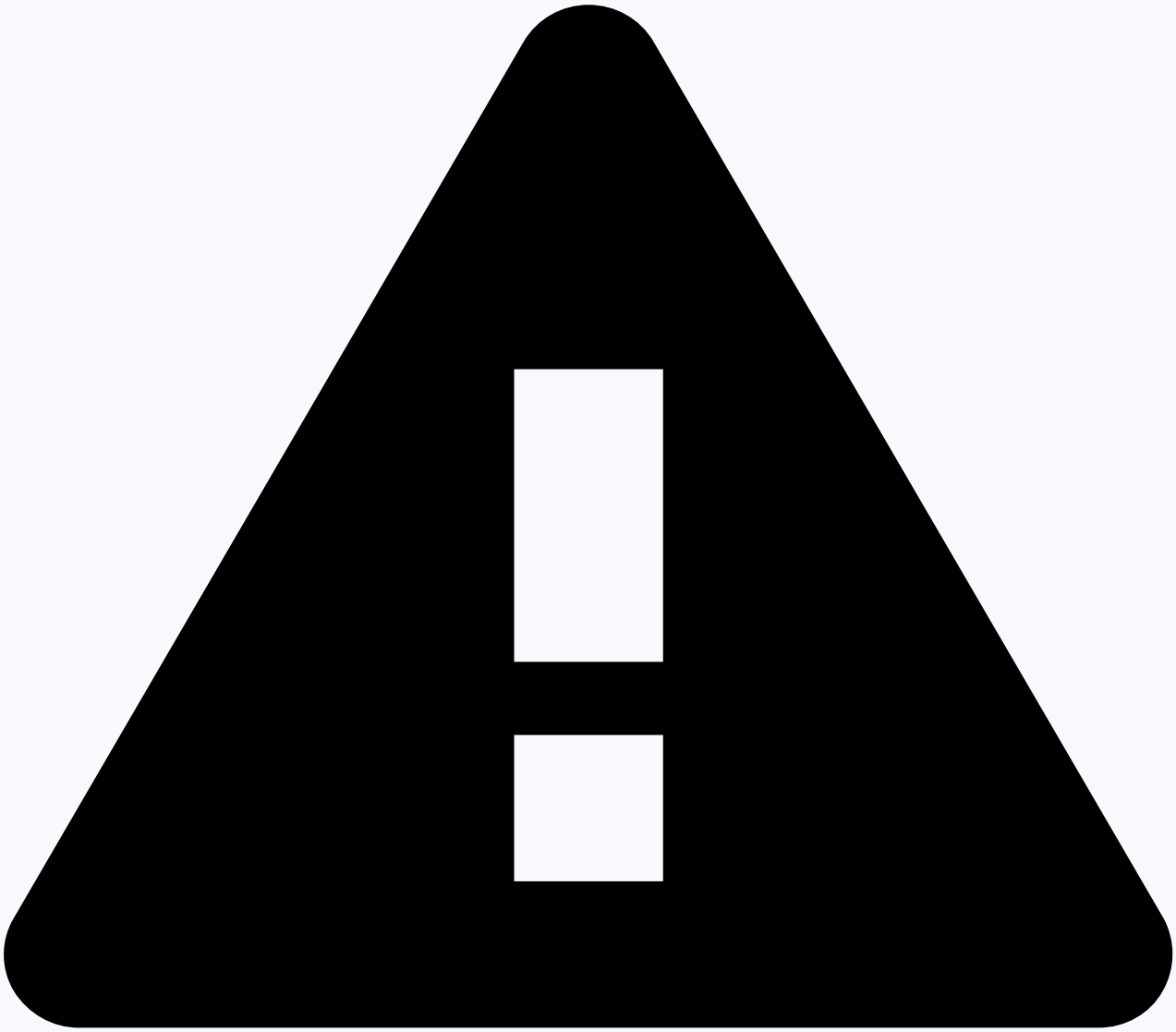
Release 26 measures the file against the sum of its blocks and refuses a file that does not fit:

```
particles:  4 + Σ (64 + n·64)
averages:   4 + Σ (60 + nx·ny·nz·48)
charges:    4 + Σ (4 + m·32)
materials:  4 + Σ (60 + nx·ny·nz·16)
```

A mismatch ends the read with `Checkpoint format validation failed for …` in the log and the checkpoint is not used. This catches a file of an older release, which is why a run started on such a checkpoint stops instead of continuing with particles at wrong positions. A file whose size happens to match is not detected — the check compares sizes, it is not a checksum.

58 Checkpoint tools

Two small command line programs translate or combine an existing particle checkpoint. Both are precompiled binaries in checkpoint_executables.zip.



These binaries fit the checkpoint layout up to release 25

They were built in February 2017 and read particle records of 56 bytes, with the number of real particles stored once per quad. Release 26 writes 64-byte records with a weight per particle (see Checkpoint file format), so on a checkpoint of release 26 they read at the wrong offsets. The source code of the two programs is not part of the repository, so they cannot be rebuilt — for a release 26 checkpoint, write the conversion yourself from the format page, or take the particle state over with `S.read_checkpoint(...)` and shift it in the parameter file.

Translation

```
./checkpoint_translate infilename outfilename X Y Z
```

- infilename — particle checkpoint file to read
- outfilename — checkpoint file to write
- X Y Z — coordinates of the translation vector in m

Combination

```
./checkpoint_combine infilename1 infilename2 outfilename
```

- infilename1 — first checkpoint file to read
- infilename2 — second checkpoint file to read
- outfilename — checkpoint file to write

Two files can be combined at a time; the result can be combined with a third file and so on.

The quads of the files are appended as they are, so overlapping quads increase the particle density in the overlapping region. Translate a checkpoint by its full width first and combine the two afterwards, so that they do not overlap.

59 Used cross sections

At the beginning of each simulation run, the collision cross sections between the selected species are computed and sent to the PICMC worker tasks as tabulated data. Additionally, a postscript file `cs.ps` is generated, where each cross section curve is plotted into a graph.

The postscript files can be opened in Linux using the tool `gv` or `okular` (KDE desktop environment), under Windows ghostscript and gsview are required to open that file.

A selection of cross section sources is given in the section cross sections.

60 Dynamic script files

Upon start of a simulation case, a few dynamic script files with file ending `*.rd` are generated. The contain the following information:

- `boundingbox.rd` - The names and detailed information of each volume segment actually used in the simulation run.
- `surfaces.rd` - Information on the interconnection and boundary conditions of each volume segment.
- `species.rd` - Molecular mass, charge and statistical weighting factor of each gaseous species actually being used in the simulation run.
- `load_balance.rd` - Assignment of each volume segment to a CPU process.

61 Postprocessing with RVM (ppp)

The 3D postprocessing files (`*.pos`) produced in simulation runs can be opened and further processed with GMSH; however, the handling of many files and the performance of post processing operations such as time-averaging, cut-planes, line profiles etc. can be cumbersome this way. For that purpose, a RIG-VM module **module_ppp.so**[1]) is provided. This module offers the following:

- Importing binary post-processing (`*.pos`) - files generated during DSMC / PIC-MC runs
- Further arithmetic processing of imported data
- Perform time averaging
- Creation of user-specified reduced views such as cut-planes, line profiles etc.
- ASCII export of processed data

Python interface. Post-processing is done with the RVM module `module_ppp.so`; the Python interface has no counterpart yet and only offers `Sim.plot()` during the run, see Driving a case from Python.

Prerequisites / installation

The file **module_ppp.so** is a module of the scripting language RIG-VM which is part of the PIC-MC software package. Within the installation directory, the module file is located in the `.../lib/*` directory, while the RIG-VM script interpreter is located in `.../bin/*`. The module comes with the PIC-MC distribution and should be located under `/.../path/to/picmc/lib`. To use it within RIG-VM from command line, make sure that the following environment variables are set as follows:

- **LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/path/to/picmc/lib**
- **PATH=$PATH:/path/to/picmc/bin**

If PIC-MC is installed, it is very likely that the environment variables are already correctly set.

General usage

The postprocessing module is used within RIG-VM scripts (ASCII files, which typically have the file extension `*.r`). These scripts should in general be in the following format:

```
compiletime load "module_ppp.so";   # Import post processing module
case = BATCH.file;                  # Get filename of respective *.par file
PPP obj = {};                       # Initiate post processing object 'obj'
# do some postprocessing stuff with obj.
# ...
```

The acronym **PPP** stands for PIC-MC Postprocessing.

Assumed that the script is named **extract.r**, its invocation from the command line works via

```
rvm -b extract.r simulationcase.par
```

It is also possible to use the same script on multiple `*.par` files in batch mode:

```
rvm -b extract.r *.par
```

The following sections give a summary of all commands integrated in the postprocessing module as well as a general description of the concept. Afterwards, the functionality of the respective functions is shown in examples.

Summary of commands

In the following all postprocessing commands are summarized:

```
obj.read("input.pos", "variable_name");
obj.merge("input.pos", "variable_name");
obj.append("input.pos", "variable_name");

obj.xy_plane(z0, delta_z, "method");
obj.xz_plane(y0, delta_y, "method");
obj.yz_plane(x0, delta_x, "method");
obj.plane(x0, y0, z0,   x1, y1, z1,   x2, y2, z2,   delta, N, M, "method");
obj.line(x0, y0, z0,   x1, y1, z1,   delta, N, "method");

obj.cylinder(x0, y0, z0,   x1, y1, z1,  r_cyl,  delta,  nl, nphi, "method");

obj.write_pos_scalar("output.pos", "View title", expression);
obj.write_pos_vector("output.pos", "View title", expression_x, expression_y, expression_z);
obj.write_pos_scalar_binary("output.pos", "View title", expression);
obj.write_pos_vector_binary("output.pos", "View title", expression_x, expression_y, expression_z);
obj.write_pos_triangulate("output.pos", "View title", expression);
obj.write_tec_scalar("output.dat", "View title", expression);
obj.write_tec_vector("output.dat", "View title", expression_x, expression_y, expression_z);
obj.write_vtk_scalar("output.vtp", "View title", expression);
obj.write_vtk_vector("output.vtp", "View title", expression_x, expression_y, expression_z);
obj.write_txt("output.txt", ["col1", "col2", ...], [exp1, exp2, ...]);
obj.write_csv("output.csv", ["col1", "col2", ...], [exp1, exp2, ...]);

obj.write_pos_scalar_parametric("output.pos", "View title", x, y, z, expression);
obj.write_pos_scalar_binary_parametric("output.pos", "View title", x, y, z, expression);
obj.write_pos_vector_parametric("output.pos", "View title", x, y, z, expression_x, expression_y,
expression_z);
obj.write_pos_vector_binary_parametric("output.pos", "View title", x, y, z, expression_x,
expression_y, expression_z);

obj.write_pos_time_steps_scalar("output.pos", "View title", "variable_name");
obj.write_pos_time_steps_scalar_binary("output.pos", "View title", "variable_name");
obj.write_pos_time_steps_vector("output.pos", "View title", "variable_name");
obj.write_pos_time_steps_vector_binary("output.pos", "View title", "variable_name");

obj.create_scalar("variable_name", expression);
obj.create_vector("variable_name", expression_x, expression_y, expression_z);

result = obj.integrate(expression);
obj.identity();
obj.reduce();
obj.convolution(vx, vy, vz, T, interpolation);
obj.convolution_circular();
```

Function reference

After having initiated a post processing object `obj` (any other name is possible, too) as described in section General usage, its internal methods can be invoked as member functions.

Functions to import data

There are three main functions for importing data:

- `obj.read("input.pos", "name");` Reads an input file in GMSH postprocessing format `*.pos` and stores it under the internal name *name*. If multiple files are loaded under the same name, they are automatically internally averaged. This feature is mostly being used for time-averaging over multiple data snapshots taken at different times.
- `obj.merge("input.pos", "name");` This command is used for combining multiple postprocessing files with different data coordinates into a single dataset. It is mostly being used for combining multiple files with e.g. individual particle positions taken at different timesteps into a trajectory plot.
- `obj.append("input.pos", "name");` The append command is used to append multiple postprocessing files into a time series.

Simple cut planes

A simple functionality, which is most frequently used, is to make a 2D cutplane from a 3D dataset, which is parallel to the coordinate axes. There are the following three options for that:

- `obj.xy_plane(z0, delta_z, "method");` Cutplane parallel to the X-Y axes at z coordinate *z0* within a range for data points of $z0 \pm \delta z$
- `obj.xz_plane(y0, delta_y, "method");` Cutplane parallel to the X-Z axes at y coordinate *y0* within a range for data points of $y0 \pm \delta y$
- `obj.yz_plane(x0, delta_x, "method");` Cutplane parallel to the Y-Z axes at x coordinate *x0* within a range for data points of $x0 \pm \delta x$

The `"method"` parameter describes how to project data points onto the cut plane. There are three options:

- `"nearest_neighbour"` The nearest datapoint located in perpendicular direction within the search range is projected onto the cutplane.
- `"average"` All data points located in perpendicular direction within the search range are averaged, and the average value is taken at the position of the cutplane.
- `"interpolate"` The value is calculated by linear interpolation between two data points located on both sides of the cutplane within the search radius

General oblique planes and lineprofiles

Oblique cutplanes can be created with the more general methods

```
obj.plane(x0, y0, z0,   x1, y1, z1,   x2, y2, z2,   delta, N, M, "method");
obj.line( x0, y0, z0,   x1, y1, z1,   delta, N, "method");
```

For a detailed description, please refer to this example.

Cylindrical cutplanes

2D cutplanes along a cylindrical surface can be obtained via

```
obj.cylinder(x0, y0, z0,   x1, y1, z1,   r_cyl, delta, nl, nphi, "method");
```

Please refer to the example on cylindrical cutplanes for further explanation.

Output functions

The postprocessing module can create output files either in GMSH postprocessing format `*.pos`, Techplot format or VTK format. Depending on the type of data, the outputs can be either in scalar or vector format. The available GMSH output commands are listed in the summary of commands above.

The commands `write_pos_scalar` and `write_pos_vector` produce GMSH `*.pos` files in ASCII format, which contain either scalar or vector data. The commands `write_pos_scalar_binary` and `write_pos_vector_binary` do the same but produce files in binary `*.pos` format, which is somewhat compressed in comparison to the ASCII format. The command `write_pos_triangulate` works only after a 2D cutplane operation and produces a triangulated view of scalar data, which means that they are represented as a triangulated surface.

Each command takes the name of the outputfile and the caption of the view ("View title") as arguments, followed by one or more arithmetic expressions. The expression parameters contain an arithmetic expression, which should be evaluated at the respective datapoints of the last operation (cutplane, lineprofile etc.). They could use the variables *x*, *y*, *z*, which represent the actual coordinates, and in case of cylindrical cutplanes also the variable *phi*, which represents the azimuth angle. Additionally, it can contain any of the variable names specified under *name* in the import functions as well as their maximum and minimum value via *name*`_max` and *name*`_min`. If the imported quantity is a vector dataset, the individual components are accessible via *name*`_x`, *name*`_y`, *name*`_z`, and thereof, the maxima / minima are again accessible via *name*`_x_max`, *name*`_x_min` etc.

There is an additional category of output functions (`write_pos_scalar_parametric`, `write_pos_scalar_binary_parametric`, `write_pos_vector_parametric`, `write_pos_vector_binary_parametric`), where also the coordinates of the output data can be explicitely manipulated. These functions work in the same way as the functions described above, but take the coordinates *x*, *y*, *z* as additional input. The purpose is to transform the coordinates of the output data in order to e.g. give them a coordinate offset or to project them onto a cylindrical surface.

Output functions of scalar and vector data in Techplot (`write_tec_*`) and VTK (`write_vtk_*`) format work in the same way; their syntax is given in the summary of commands and they are described in the section Additional output formats.

For export of data in ASCII files (either `*.txt` or `*.csv`) the commands `write_txt` and `write_csv` are available. Besides the name of the output file, these functions expect a list of column names and a list of expressions for the columns (obviously, both lists have to be of the same length). The expressions in the expression list can use the same variable names as described above.

Finally, there are output functions for time series (animations), the `write_pos_time_steps_*` commands. In contrast to the previous output functions, they work not (yet) with arithmetic expressions but only with variable names, as were used in the "name" argument of the append command. If needed, additional input data sets can be computed via the commands `create_scalar`, `create_vector` as described in the data manipulation section.

Usage of the module by examples

The module is to be used within RIG-VM scripts. We demonstrate the postprocessing capabilities on the DSMC gas flow example from this tutorial.

Simple scalar cut-plane

One of the easiest and most frequent tasks is to create a quick cut-plane aligned parallel to XY, XZ or YZ plane. Assumed, we have the file **pressure_Ar_60.00ms.pos** from a DSMC run of the tube example as output file, and would like to have a scalar pressure plot in the XY plane. The according script - let's call it **extract.r** looks like follows:

```
compiletime load "module_ppp.so";
PPP obj = {};

obj.read("pressure_Ar_60.00ms.pos", "pAr");
obj.xy_plane(0, 10, "nearest_neighbour");
obj.write_pos_scalar("cutplane_xy.pos", "Ar pressure [Pa]", pAr);
```

The script can be executed with **rvm** as shown in the following console output.

```
ap@sim037:~/sim/Tube/sample$ rvm extract.r
Created View {pAr}
File pressure_Ar_60.00ms.pos has 109368 scalar and 0 vector points
XY-Plane extract, z0 = 0 +/- 10, method = 'nearest_neighbour'
ap@sim037:~/sim/Tube/sample$
```

As result, the output file **cutplane_xy.pos** is created and can be viewed in GMSH.

The meaning of the commands in the last three lines is as follows:

- `obj.read("input.pos", "variable_name");`
  - The first parameter is the filename of the binary POS file
  - The second parameter is a *variable name* used for referencing these data in a later step
- `obj.xy_plane(z0, delta_z, "method");`
  - `z0` is the center coordinate of the XY plane
  - `delta_z` is the range, where data points will be considered
  - `method` is the interpolation method with following options:
    - `"nearest_neighbour"`: Within the given range **delta_z**, the points which are closest to **z0** are projected onto the cut plane
    - `"average"`: Within the given range **z0 +/- delta_z**, all points belonging to the same (X,Y) coordinate are projected onto the cut-plane and **averaged**.
    - `"interpolate"`: Points from the given range **z0 +/- delta_z** are interpolated on the cut-plane. The interpolation weighting function is 1.0 at **z==z0** and decreases linearly to 0.0 for **z-->z0 +/l delta_z**.
    - `"sum"`: Values from the given coordinate range **z0 +/- delta_z** are summed up.
- `obj.write_pos_scalar("output.pos", "plot_title", expression);`
  - This writes a *scalar* output file named `"output.pos"`.
  - The `"plot_title"` will be shown as view title in GMSH.
  - With `expression` the scalar output value can be specified. In the above example, it is just the pressure value named `pAr`.

Besides of `obj.xy_plane(z0, delta_z, "method")` there are also `obj.xz_plane(y0, delta_y, "method");` and `obj.yz_plane(x0, delta_x, "method");` available.

Scalar cut-plane with arithmetics

Within the `expression` parameter it is possible to do some arithmetics. If e.g. the pressure should be plotted in mTorr instead of Pa (1 mTorr = 133 mPa), the following script will do it:

```
compiletime load "module_ppp.so";
PPP obj = {};

obj.read("pressure_Ar_60.00ms.pos", "pAr");
obj.xy_plane(0, 10, "nearest_neighbour");
obj.write_pos_scalar("cutplane_xy.pos", "Ar pressure [mTorr]", pAr*1000/133);
```

Scalar cut-plane with triangulation

By replacing the command `write_pos_scalar` by `write_pos_triangulate` it is possible to create a triangulated scalar view:

```
compiletime load "module_ppp.so";
PPP obj = {};

obj.read("pressure_Ar_60.00ms.pos", "pAr");
obj.xy_plane(0, 10, "nearest_neighbour");
obj.write_pos_triangulate("cutplane_xy.pos", "Ar pressure [mTorr]", pAr*1000/133);
```

Scalar cut-plane from multiple input files

Let's assume we would like to plot the flux (=density*velocity) in positive X direction but have forgotten to activate the *density* plot in the parameter file. Instead we have only the *number* and the *velocity* plots available.

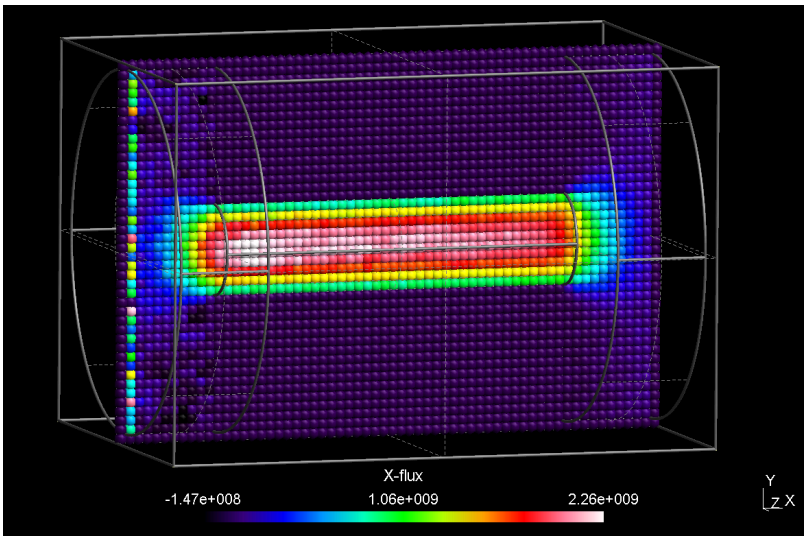
The postprocessing module allows us to plot the density and the flux anyway as shown in the following:

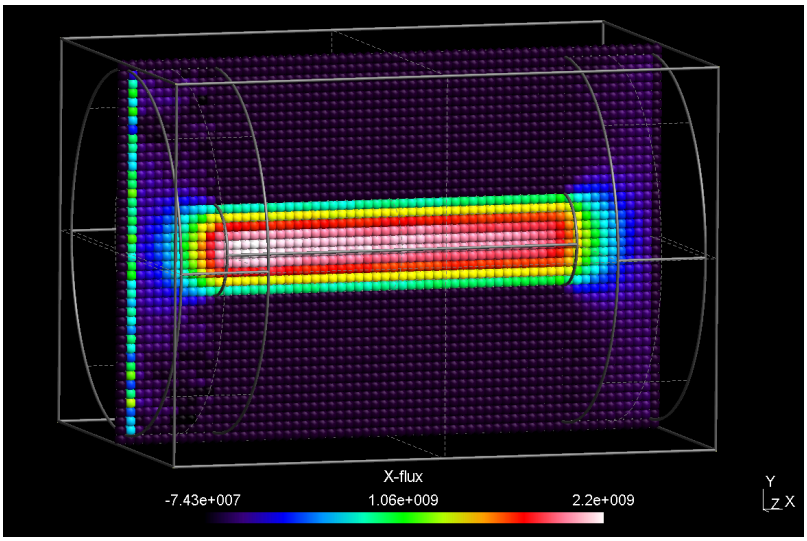
```
compiletime load "module_ppp.so";
PPP obj = {};

obj.read("number_Ar_60.00ms.pos", "N");
obj.read("cell_volumes.pos", "vol");
obj.read("velocity_Ar_60.00ms.pos", "v");
obj.xy_plane(0, 20, "nearest_neighbour");
obj.write_pos_scalar("cutplane_xy.pos", "X-flux", vol>0 ? N/vol*v_x ! 0);
```

Important: Please note, that the velocity is a *vector* plot. In the `read` command we have assigned the variable name `v` to the velocity. In the expression below, the three components `v_x`, `v_y` and `v_z` must be used instead. The appendixes `_x`, `_y` and `_z` are attached automatically to each variable name in case of a vector plot!.

Depending on the statistics, the cut-plane of the flux plot may look a bit noisy. In this case, it helps to use the `"interpolate"` option rather than `"nearest_neighbour"`:





The expression `vol>0 ? N/vol*v_x ! 0` contains an implicit if statement: The computation density = number/volume is not possible for all cells, since in some cases the volume may be zero. Thus, first it is checked, if `vol>0`. If that condition evaluates to *true*, the expression behind the `?` applies, otherwise the expression branches behind the `!` symbol.

It may look a bit ugly to have these arithmetics in one line. Another possibility is to define the density within the PPP object:

```
compiletime load "module_ppp.so";

PPP obj = {
  float NREAL=1e11;
  float density: vol>0 ? NREAL*N/vol ! 0;
};

obj.read("number_Ar_60.00ms.pos", "N");
obj.read("cell_volumes.pos", "vol");
obj.read("velocity_Ar_60.00ms.pos", "v");
obj.xy_plane(0, 20, "interpolate");
obj.write_pos_scalar("cutplane_xy.pos", "X-flux", density*v_x);
```

Since the variables `N` and `vol` are not known during definition of `density` within the PPP object, it is important to use the *alias* method (i. e. the ":") instead of the *assign* method (i. e. "=") for defininig

`density`. The result of this modified script is the same as for the script above but now, it looks more clearly arranged.

Time averaging

If you would like to perform averaging of the same quantity over several time steps, it is sufficient to call `obj.read(...)` several times but with the same variable name. An example is given in the following:

```
compiletime load "module_ppp.so";

case = "sample2";

PPP obj={};

for (time=100; time<=200; time=time+20) {
  obj.read("$case/density_Ar_$(time).00ms.pos", "d");
  obj.read("$case/velocity_Ar_$(time).00ms.pos", "v");
}

obj.xy_plane(0, 10, "nearest_neighbour");
obj.write_pos_vector_binary("flux-cutplane.pos", "Ar flux [1e18/(m²s)]", 1e-18*d*v_x, 1e-18*d*v_y,
1e-18*d*v_z);
```

The `read` command of the PPP object automatically recognizes if one variable is used several times and performs an averaging of the data in this case.

Development over time

If you would like to illustrate the development of the same quantity over several time steps, you need to call `obj.append(...)` several times but with the same variable name and then one of the `write_pos_time_steps` functions. An example is given in the following:

```
compiletime load "module_ppp.so";
compiletime load "module_system_linux.so";

# defining parameters
case   = "sample2";
start  = 100;
end    = 200;
dtime  = 20;
unit   = "us";
z      = 0;
folder = "density";
files  = "density_e";

# create ppp subfolder if it does not exist
System FS = { rootpath = "$case"; };
if (FS.folder_exists("ppp") == 0) then {
  out "Creating folder $case/ppp\n";
  FS.create_folder("ppp", 493)
};

# applying a cut plane to each time step
for (time=$start; time<=$end+0.001; time=time+dtime) {
  PPP obj1 = {};

  obj1.read("$case/$folder/$(files)_$(time% 4,2f)$(unit).pos", "X");
  obj1.xy_plane($z, 10, "nearest_neighbour");
  obj1.write_pos_scalar_binary("$case/ppp/cut_$(files)_$(time% 4,2f)$(unit).pos", "$files", X);
}
```

```
# appending the cut planes to each other
PPP obj2 = {};
for (time=$start; time<=$end+0.001; time=time+dtime) {
  obj2.append("$case/ppp/cut_$(files)_$(time% 4,2f)$(unit).pos", "Y");
}
obj2.write_pos_time_steps_scalar_binary("$case/timestep_cut_$(z)mm_$(files)_$(start)-$(end)$(unit).
pos", "$files", "Y");
```

First of all `module_system_linux.so` is loaded, which is needed for creating the ppp subfolder for storing the intermediate cut planes. Next some parameters are set including the time interval, the z position of the cut plane and the name of the plot. Then the ppp subfolder is created, where the cut planes generated in the next part are stored. In the last part these cut planes are appeded to each other and written to the output file.

The order of first creating the cut planes and then appending them to each other is important, because it is (at the moment) not possible to make cut planes of variables with multiple time steps. Additionally the `write_pos_time_steps` functions can only be called on the variable name and not on an expression, so that calculations have to be done before.

Generating vector plots as cut plane

Finally we would like to plot the flux as vector plot. All we have to do is to replace the `obj.write_pos_scalar(...)` by `obj.write_pos_vector(...)`. Since a vector plot has three components, we have three instead of one expression arguments:

```
compiletime load "module_ppp.so";

PPP obj = {
  float NREAL=1e11;
  float density: vol>0 ? NREAL*N/vol ! 0;
};

obj.read("number_Ar_60.00ms.pos", "N");
obj.read("cell_volumes.pos", "vol");
obj.read("velocity_Ar_60.00ms.pos", "v");
obj.xy_plane(0, 20, "interpolate");
obj.write_pos_vector("cutplane_xy.pos", "X-flux", density*v_x, density*v_y, density*v_z);
```

Revert POS file data to original state

In case you have applied cut planes to your plots and you want to revert them to their original state, you can use the `identity` function. This is shown in the following example, where the complete density distribution is created after a cut plane of the number plot was written:

```
compiletime load "module_ppp.so";

PPP obj = {};

obj.read("number_Ar_60.00ms.pos", "N");
obj.xy_plane(0, 10, "nearest_neighbour");
obj.write_pos_scalar("cutplane_xy.pos", "Ar number [#]", N);
obj.identity();
obj.read("cell_volumes.pos", "vol");
obj.write_pos_scalar("density_Ar.pos", "Ar density [#/m³]", vol>0 ? N/vol ! 0);
```

General (oblique) interpolation views

It is possible to generate cut planes in a more general way. For this purpose, a general description of a parametric plane looks as follows:

$\vec{r}(s,t) = \vec{x}_0 + s\,(\vec{x}_1 - \vec{x}_0) + t\,(\vec{x}_2 - \vec{x}_0)$.

The accordant post processing command is written as:

```
obj.plane(x0, y0, z0,   x1, y1, z1,    x2, y2, z2,  delta, N, M, "method");
```

First, the three points of the above equation are given coordinate-wise (i.e. the first 9 parameters). The next parameter `delta` is the vertical distance to the plane, wherin points are considered for interpolation. The integer numbers `N, M` give the resolution of the cut plane in direction of the first and second vector. **Update:** It is possible to specify `0, 0` for `N, M`; in this case the actual resolution of the imported POS file will be taken.

An example of an *oblique plane* with respect to our tube DSMC case is shown in the following:

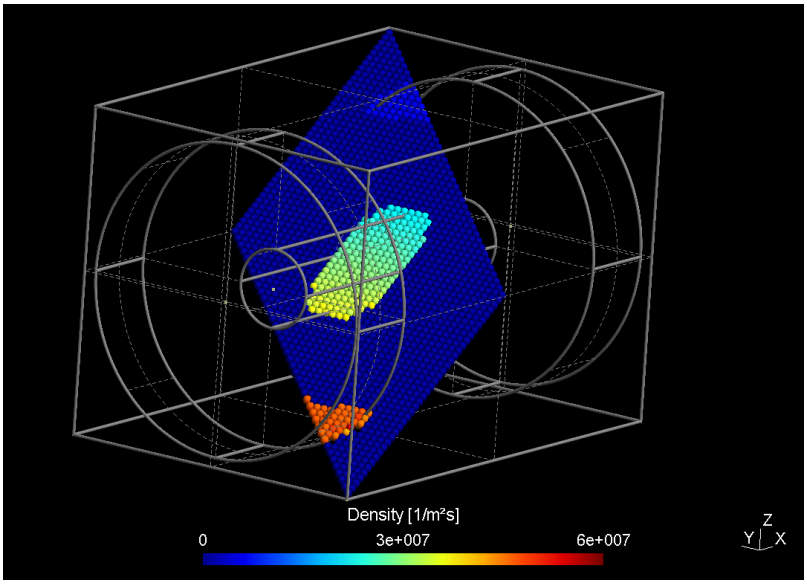
```
compiletime load "module_ppp.so";

PPP obj={
  float density: vol>0 ? N/vol ! 0;
};

obj.read("number_Ar_60.00ms.pos", "N");
obj.read("cell_volumes.pos", "vol");
obj.read("velocity_Ar_60.00ms.pos", "v");

obj.plane(-110, -210, -210,   200, 210, 0,    200, -210, 0,    15,   40, 40, "interpolate");

obj.write_pos_scalar("obliqueplane.pos", "Density [1/m²s]", density);
```



Analogously, it is possible to extract data along a straight line:

$\vec{r}(s) = \vec{x}_0 + s\,(\vec{x}_1 - \vec{x}_0)$

with the postprocessing command:

```
obj.line(x0, y0, z0,   x1, y1, z1,    delta, N, "method");
```

Additional output formats

Visualization Toolkit (VTK)

Besides the POS output format of GMSH, it is possible to write the 3D data in the Visualization Toolkit format. The two functions for scalar and vector data are:

```
obj.write_vtk_scalar("output.vtp", "View title", expression);
obj.write_vtk_vector("output.vtp", "View title", expression_x, expression_y, expression_z);
```

The syntax is identical to that of the POS format.

Tecplot

Also the Tecplot format is supported for both scalar and vector data:

```
obj.write_tec_scalar("output.dat", "View title", expression);
obj.write_tec_vector("output.dat", "View title", expression_x, expression_y, expression_z);
```

Again the syntax is identical to that of the POS format.

ASCII export

TXT output

It is convenient to directly export certain results as ASCII TXT file, which can be imported e.g. in Origin or other graph presentation software. For that purpose, the following method is implemented:

```
obj.write_txt("file_name.txt", ["col1", "col2", ...], [exp1, exp2, ...]);
```

The arguments have the following meaning:

- The first argument is the name of the TXT file, which should be created
- The second argument is a **list** containing all column title strings. They will appear in the first line of the ASCII file.
- The third argument is a list of expressions, which should be computed for each column.
    - Within the expressions, all variable names originating from imported POS files as well as the variables "x, y, z" for the respective coordinates can be used.
    - Additionally, the maximum and minimum values of all variables are computed and can be accessed via `variablename_max`, `variablename_min`.

We demonstrate this in a short example:

```
compiletime load "module_ppp.so";

PPP obj1 = {};

obj1.read("absorption.pos", "A");
obj1.xy_plane(-141.75, 10, "nearest_neighbour");
obj1.write_pos_scalar_binary("absorption-2D.pos", "SiH4 absorption [1/m²s]", A);
```

```
PPP obj2 = {};

obj2.read("absorption-2D.pos", "A");
obj2.line(420, -380, -141.75,   607.5, -380, -141.75,  40, 19, "average");
obj2.write_txt("profile_avg.txt", ["r [mm]", "abs [1/m²s]", "abs [%]"], [x-420, A, 100*A/A_max]);
```

This script reads a 3D posfile "absorption.pos" of an absorption plot into **obj1**, creates a 2D cutplane of it and stores the cutplane into "absorption-2D.pos". This created file is then read into a second PPP object **obj2**. Here, the absorption profile is averaged along a line in X direction. The averaged profile is finally dumped into an ASCII file. The ASCII file finally looks like follows:

```
         r [mm]   abs [1/m^2s]        abs [%]

        4.93421    3.80416e+20        99.9454
        14.8026    3.77636e+20        99.2152
        24.6711    3.80623e+20            100
        34.5395     3.7374e+20        98.1916
        44.4079    3.75455e+20         98.642
        54.2763    3.73143e+20        98.0347
        64.1447    3.68571e+20        96.8336
        74.0132    3.63662e+20        95.5439
        83.8816    3.58961e+20        94.3087
          93.75    3.54935e+20         93.251
        103.618    3.47403e+20         91.272
        113.487    3.43792e+20        90.3235
        123.355    3.31429e+20        87.0752
        133.224    3.24909e+20        85.3624
        143.092    3.18416e+20        83.6563
        152.961    3.03792e+20        79.8144
        162.829    2.88857e+20        75.8905
        172.697    2.66338e+20        69.9741
        182.566    2.29174e+20        60.2102
```

CSV output

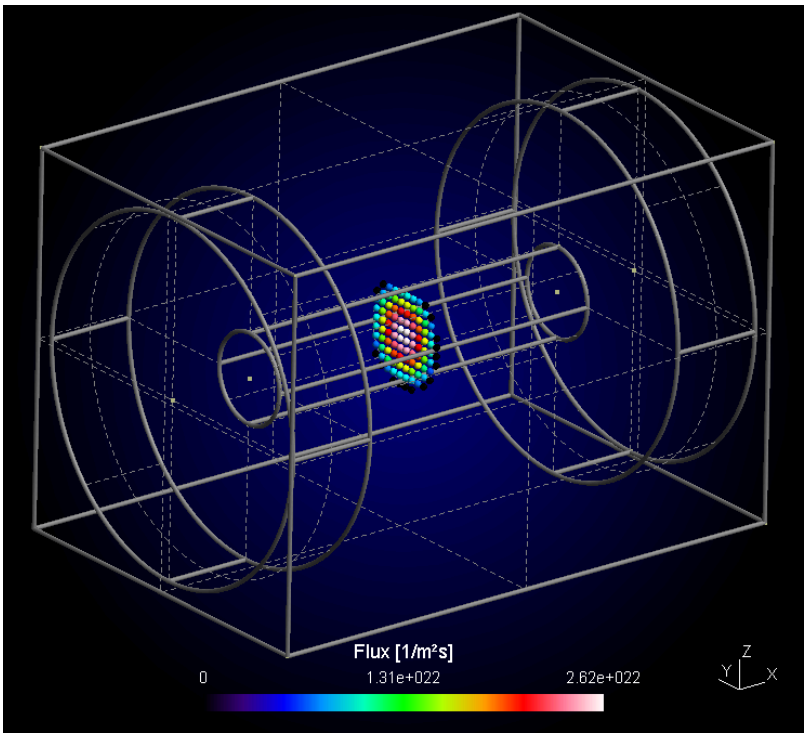
Instead of TXT files, writing the output in the CSV format is also possible. The syntax of `write_csv` is identical to that of `write_txt`:

```
obj.write_csv("file_name.csv", ["col1", "col2", ...], [exp1, exp2, ...]);
```

The column separator is a *colon*, which is explicitely mentioned in the file and the decimal separator is a *point*.

Numerical integration

The numerical integration feature is currently available for cut-planes only. In our tube example we might want to integrate the particle flux through the tube. This can be accomplished as follows:



Tube model with cross section of interest

```
compiletime load "module_ppp.so";

mesh_unit=1e-3;
Nsccm = 4.478e17;

file "integration.txt" <<
for (time=20; time<=400; time=time+20) {
  PPP obj={};

  obj.read("density_Ar_$(time).00ms.pos", "density");
  obj.read("velocity_Ar_$(time).00ms.pos", "v");
  obj.plane(200, -60, -60,   200, 60, -60,   200, -60, 60,  10, 0, 0, "interpolate");

  flow = obj.integrate(density*v_x)*mesh_unit^2/Nsccm;
  out "$(time)  $(flow)\n";
}
```

Script for obtaining integrated flow through cross section

In the script shown above, the density and velocity of Argon are loaded for a complete time series t=20...400 ms. For each time step, a PPP object is created, and the flux is generated via the expression `density*v_x` within the cut plane. With the `integrate` function, the total flux through the tube cross section can be computed.

The numerical integration is internally performed over triangles of adjacent points. For each triangle, the average of the three point values is multiplied with the area of the triangle element. If the mesh unit is not 1 m, the integration result should be multiplied with `mesh_unit^2` in order to get the correct unit.

In the example script, the result is additionally divided by the particle number per second for 1 sccm in order to get the flow in sccm.

Merging of (position) plots

In the normal output files (e.g. density) each point corresponds to a single cell with its position and its value. A second density plot has the same cells at the same position but with different values. Therefore it makes no sense to merge these files into a single one, this is only the case for position plots, where the positions of particles are stored independent of the cell resolution. By merging multiple of these files the illustration of the particle movement is possible:

```
compiletime load "module_ppp.so";

PPP obj = {};

for (float time=0; time<=100; time=time+0.5) {
  obj.merge("position/position_e_$(time %,2f)ns.pos", "pos");
 }
obj.write_pos_scalar_binary("position_e_0-100ns.pos", "e position", pos);
```

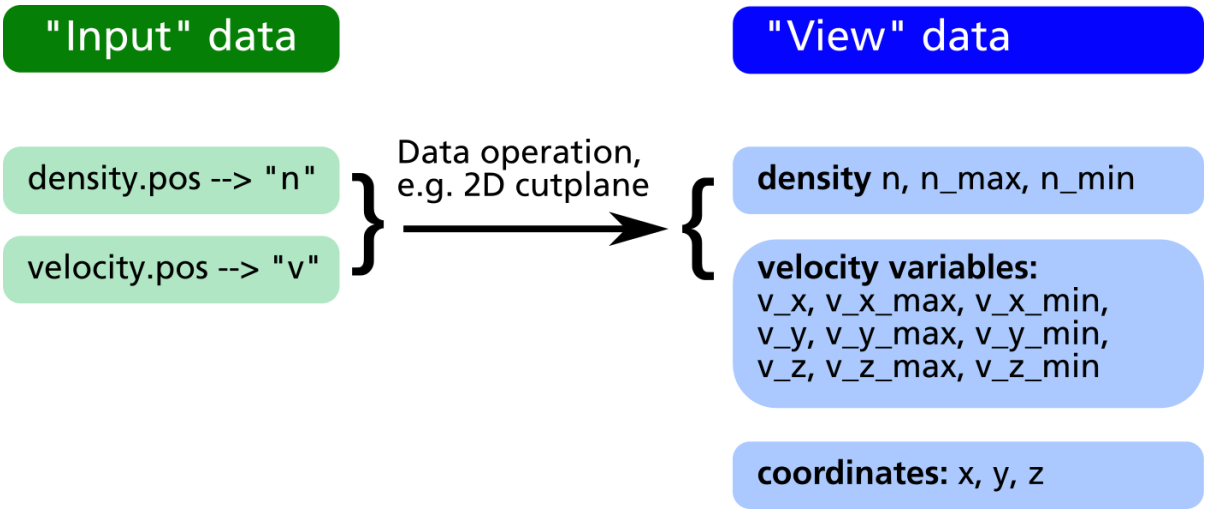
In this example a position plot each 0.5 ns for a time interval between 0 and 100 ns is merged and written to a single file. The syntax of the merge function is identical to that of the read function.

Manipulation of data

General concept

The RVM postprocessing module generally uses two different data sets, namely the "input" and the "view" data as shown in the figure below.

- The "input" data consists of the raw data obtained from the simulation output files (GMSH POS format). With every `read` command, a new entry in the input data is created. Each input data can be assigned a short name which will be later used as variable name (e.g. "n" for density in the figure below).
- The "view" data are obtained from the "input" data by some geometric transformation, e.g. a 2D cut plane or a line scan. They have the same names as the input data with some exceptions:
    - If the input data are vectors (e.g. the velocity field "v" in the example below) three different "view" variables `v_x, v_y, v_z` are generated in order to allow a component-wise access to the vector field.
    - Additional variables such as the coordinates `x, y, z` as well as maximum and minimum values for each "view" variable (`n_min, n_max, v_x_min, v_x_max etc.`) are automatically generated and appended to the "view" dataset.
- If another geometrical operation is applied, the former "view" data is being deleted and replaced by data generated from the "input" in connection with the geometrical operation.
- If no geometrical operation is applied at all, the "view" data will be a copy of all input data (in former versions, the command `obj.identity();` had to be explicitely invoked for that purpose, nowadays this is automatically done internally).



After applying the geometrical operations, in the `obj.write_XXX` commands all variable names in the "view" data can be used for forming mathematical expressions defining what to extract into the output files. For the above example it is e.g. possible to extract the particle *flux*, i.e. the product of density and velocity, in direction of the Z coordinate as well as the *relative* density distribution in %:

```
compiletime load "module_ppp.so";
PPP obj = {};

obj.read("density.pos",  "n");
obj.read("velocity.pos", "v");
obj.xy_plane(0, 10, "nearest_neighbour");

obj.write_pos_scalar_binary("flux-Z-2D.pos", "Vertical flux [1/m^2 s]", n*v_z);
obj.write_pos_scalar_binary("density-relative.pos", "Relative density [1/m^2 s]",
                            100*n/n_max);
```

Creation of new input data

Usually, the input data is obtained directly via the command `obj.read("filename", "label");`. In some cases it may be useful to create additional input data via an algebraic formula from the existing data. For this purpose, the following commands can be used:

- `obj.create_scalar("new_label", expression);` --> Create new scalar input data entry
- `obj.create_vector("new_label", exp_x, exp_y, exp_z);` --> Create new vector input data entry

In the example above, it is possible to plot the *relative* density distribution in % because of the existence of the view variable `n_max`. However it is not clear how to plot e.g. the relative flux distribution, since there is no obvious way to obtain the maximum value of `n*v_z`. By creation of a "flux" variable as new input entry, this problem can be solved (see example listing and data structure diagram below).

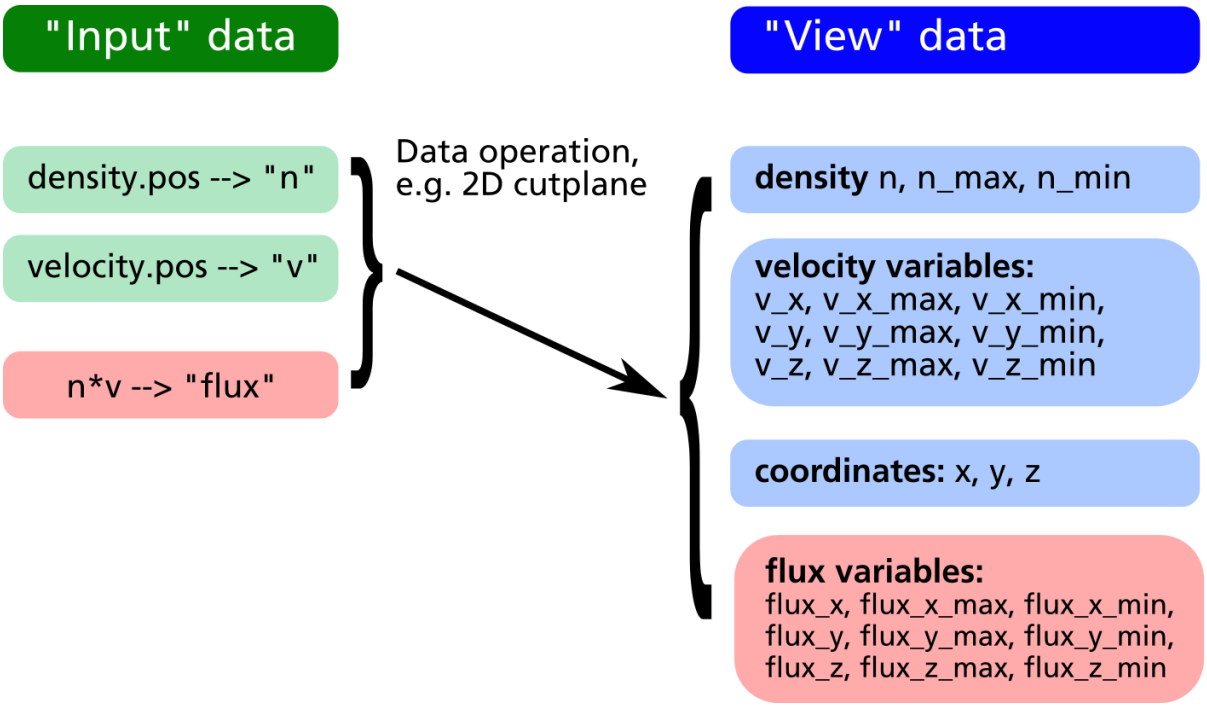
```
compiletime load "module_ppp.so";
PPP obj = {};

obj.read("density.pos",  "n");
obj.read("velocity.pos", "v");

obj.create_vector("flux", n*v_x, n*v_y, n*v_z);

obj.xy_plane(0, 10, "nearest_neighbour");
```

```
obj.write_pos_scalar_binary("flux-Z.pos", "Vertical flux [1/m^2 s]", flux_z);
obj.write_pos_scalar_binary("flux-Z-relative.pos", "Relative Vertical flux [%]",
                            100*flux_z/flux_z_max);
```



Reduction of input data

Let us assume we have 3D data of the density and velocity of evaporated atoms. In the XY-plane at Z=100 there is a substrate where the evaporated atoms can condensate.

First, in order to get the thickness profile on the substrate, the flux has to be extracted by multiplying density and z component of velocity in a cut-plane just below the substrate surface.

Second, let us assume the substrate is moving into Y direction, i.e. we have a dynamic deposition. In such cases, only the linescan of the profile in X direction averaged over Y is of relevance.

In order to do this, in prior versions of the post processing module it was required to make the 2D cut plane first, save it into a temporary POS file, load this POS file into a second post processing object and finally extract the line profile from there.

From version 2.4.3 on, there is the possibility of reducing the input data according to the last executed geometric operation, the command syntax is:

- `obj.reduce();` --> Reduction of input data space according to current arrangement of view data.

This allows for obtaining a line profile from 3D data in a much easier way:

```
compiletime load "module_ppp.so";
PPP obj = {};

aperture_width = 100;   # Aperture width on substrate in transport direction

obj.read("density.pos", "n");
obj.read("velocity.pos", "v");

# Define vertical flux component
```

```
obj.create_scalar("Fz", n*v_z);

# Make cut plane of data just below the substrate surface
obj.xy_plane(99.5, 10, "nearest_neighbour");

# Reduce input data to cut-plane (this is the important command)
obj.reduce();

# Make another cut plane averaging over the aperture_width
# in order to get the line profile
obj.xz_plane(0, aperture_width/2, "average");

# Write line profile into TXT file
obj.write_txt("absorption_line.txt",
              ["x", "avg_absorption", "rel_absorption"],
              [x, Fz, 100*Fz/Fz_max]);
```

Control of written data

Regarding the format of written data, there are the following possibilities:

- The coordinates, where the written data get plotted in POS files, can be explicitely specified
- It is possible to set a flag deciding whether to write data at a given coordinate point or not

The first option involves the following output methods:

- `obj.write_pos_scalar_parametric("filename.pos", "Legend", x, y, z, expression);`
- `obj.write_pos_scalar_binary_parametric("filename.pos", "Legend", x, y, z, expression);`
- `obj.write_pos_vector_parametric("filename.pos", "Legend", x, y, z, exp_x, exp_y, exp_z);`
- `obj.write_pos_vector_binary_parametric("filename.pos", "Legend", x, y, z, exp_x, exp_y, exp_z);`

The first two commands are for parametrized scalar plots (in ASCII and binary format, respectively), the second two commands are for vector plots. An example is given in the following listing.

```
compiletime load "module_ppp.so";

PPP obj = {};
obj.read("density.pos",  "n");
obj.read("velocity.pos", "v");
obj.xy_plane(100, 10, "nearest_neighbour");

# Plot density and flux side by side. A coordinate offset allows for
# loading both pos files into GMSH and have the cut planes non-overlapping

obj.write_pos_scalar_binary_parametric("density-2D.pos", "Density [1/m^3]", x, y, z, n);
obj.write_pos_scalar_binary_parametric("flux-2D.pos", "Flux [1/m^3]", x+200, y, z, n*v_z);
```

In order to control explicitely, which data point should be written and which should be omitted, there is the possibility to define a `write_flag` variable within the postprocessing object. This is demonstrated in the following:

```
compiletime load "module_ppp.so";

PPP obj = {
  write_flag: A>0;
```

```
};

obj.read("absorption.pos", "A");
obj.write_pos_scalar_binary("absorption-noZeros.pos", "Absorption [1/m^2 s]", A);
```

In a large 3D absorption file, non-zero values only occur at the intersection between volume grid and mesh elements of walls. Thus a large fraction of the POS file consists of zeros, which is a waste in memory consumption. With the above script, the `write_flag` variable forces the write routine to only output those values with non-zero absorption, i.e. `A>0`.

It is important to define `write_flag` with a colon and not to write `write_flag = (A>0);`. In the rvm scripting language, the colon means that the value of `write_flag` will be only computed on request according to the currently defined parameters on right side. With the regular "=" assignment operator, the value of the variable `A` will be required at the moment of the definition of `write_flag`, which is not the case in the above script.

It is also possible to combine multiple conditions e.g. in order to specify a certain coordinate range for output:

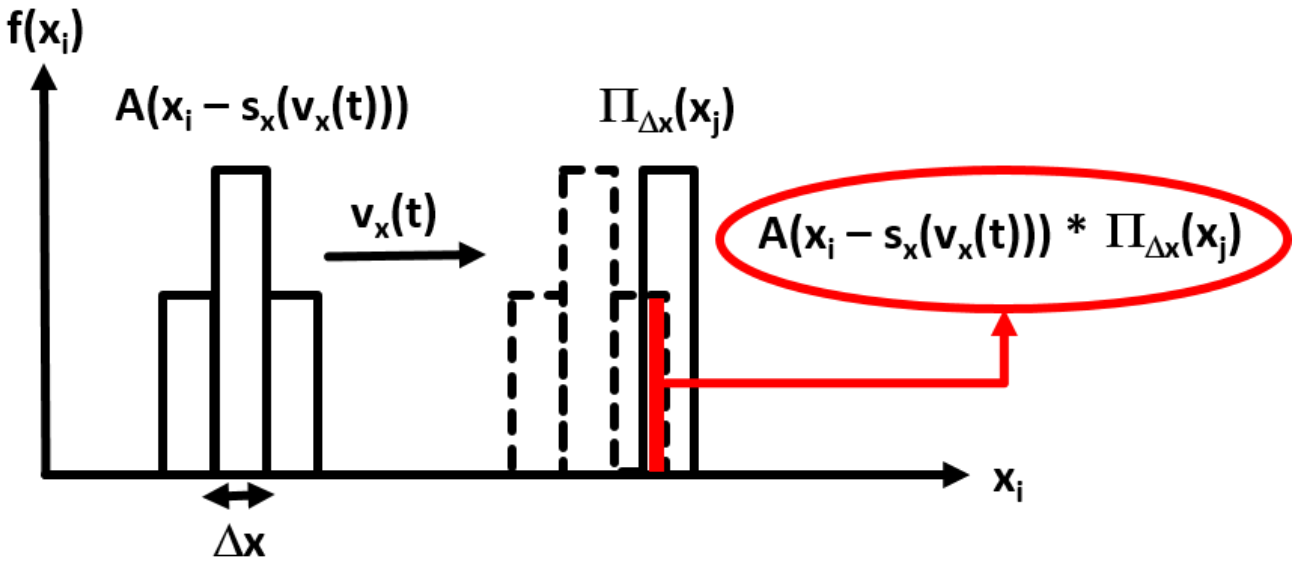
```
compiletime load "module_ppp.so";

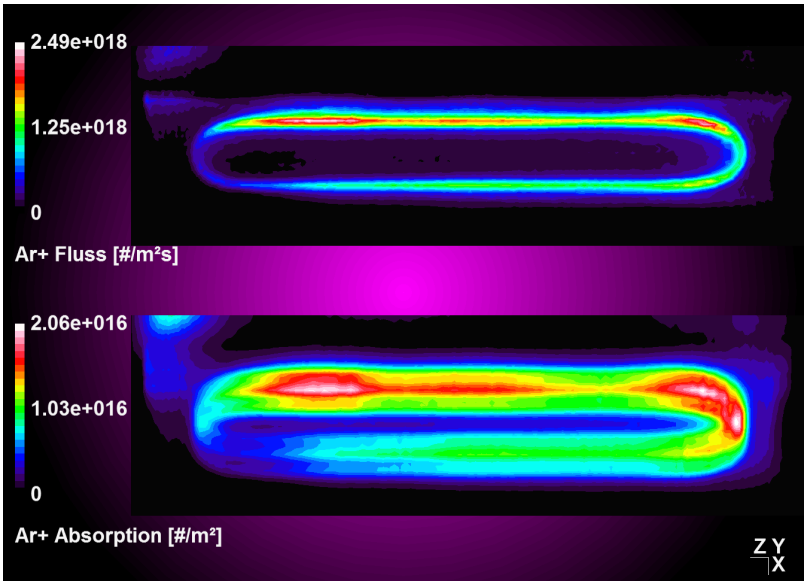
PPP obj = {
  write_flag: (abs(x)<100) and (y>-50) and (y<150);
};

# ...
```

Convolution function

A convolution function has been implemented to operate on target absorptions plots in an attempt to model erosion profiles in MoveMag aplications. The primary approach here is to assume plasma and ion bombardment, respectively do not change with magnetron position which should be valid for small displacements vectors. Thus, we simply integrate over time the convolution of a time deviant, i.e. moving absorption plot with a stationary step function.





Convolution and integration over time of moving source data with stationary step function.	Absorption plot (top) and convolution result (bottom) for a periodic oscillation in y.

```
compiletime load "module_ppp.so";

PPP obj1 = {};

#Define velocity vector and integration time
float vx[4], vy[4], vz[4]; #[m/s]
float T = 4; #[s]

#Oscillation starting from center position
#Move 1s to left with 1 m/s, then 2 times to right accordantly, then left again.
vx[0] = -1;
vx[1] = +1;
vx[2] = +1;
vx[3] = -1;

obj1.read("absorption_Arplus_100.00us.pos", "val");

obj1.xz_plane(0.0088, 1, "nearest_neighbour");
```

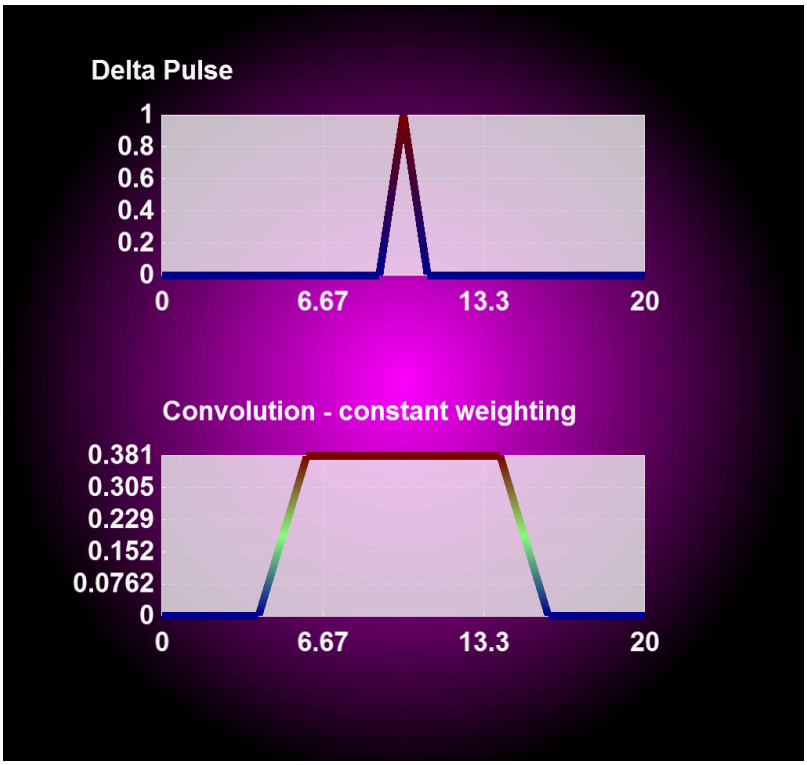
```
obj1.reduce; #Operate on cut plane (2D) data set for better performance!

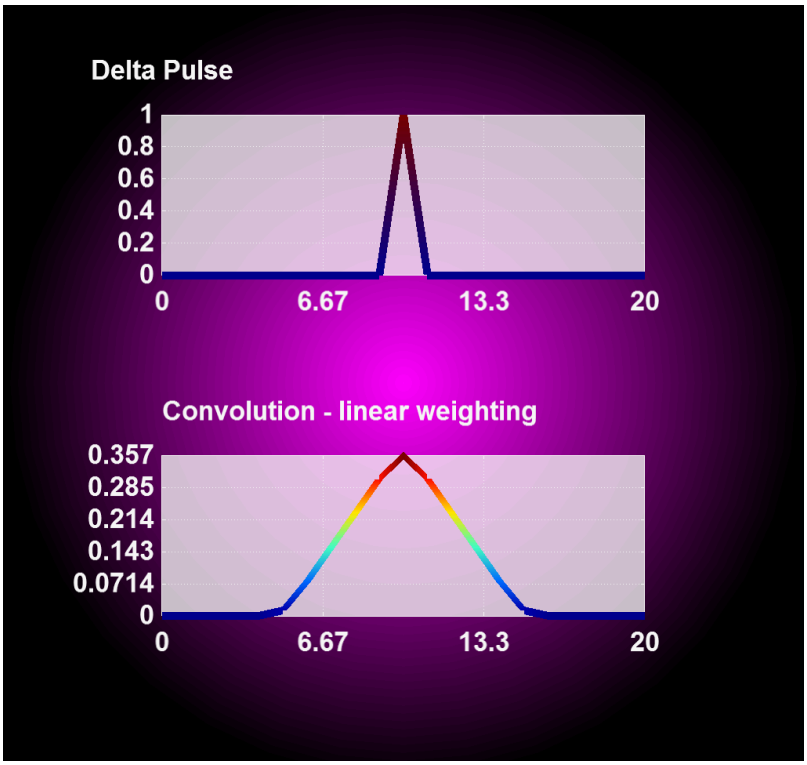
obj1.convolution(vx, vy, vz, T, 0);

obj1.write_pos_scalar_binary("absorption_Arplus_100.00us_xz-plane_convolution.pos", val);
```

Convolution with linear weighting

Entering '1' for the weighting flag enables a linear weighting scheme during convolution operation from 1 (no displacement) to 0 (maximum displacment). With this you can realize an interpolation between two or more states, i.e. magnetron positions, by merging the results from corresponding convolution operations together.





Convolution with constant weighting.	Convolution with linear weighting.

Circular Convolution

For a 360° convolution over the plot center use the function 'convolution_circular()' as seen below. This function has been implemented to adress modeling of absorption profiles on rotating substrates.

```
compiletime load "module_ppp.so";

PPP obj1 = {};
```

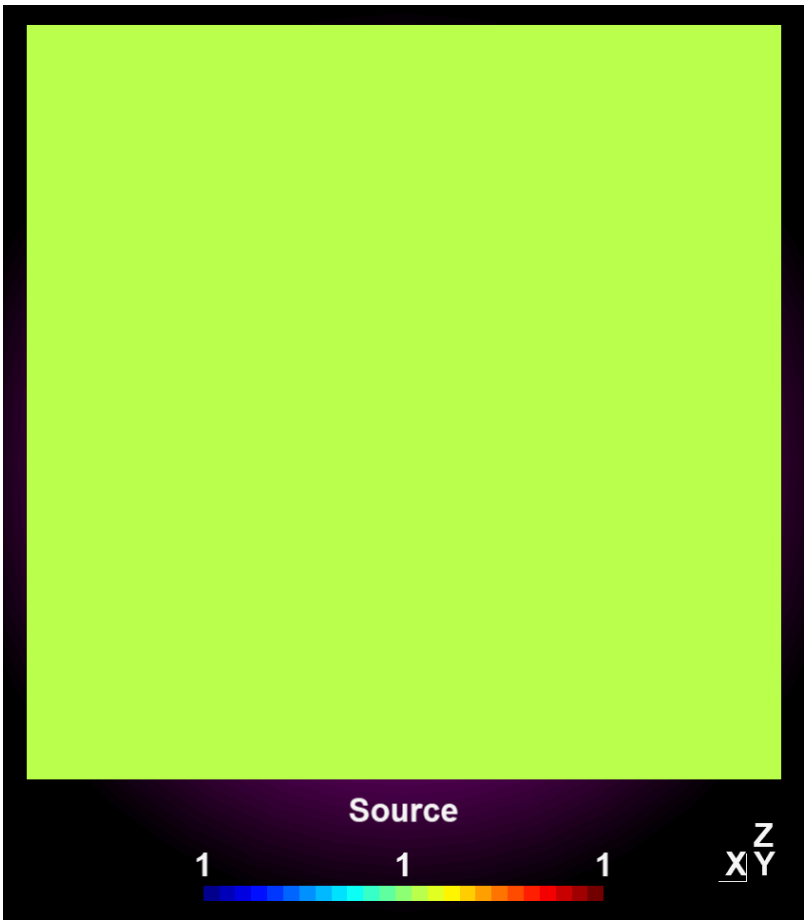
```
obj1.read("absorption_Arplus_100.00us.pos", "val");

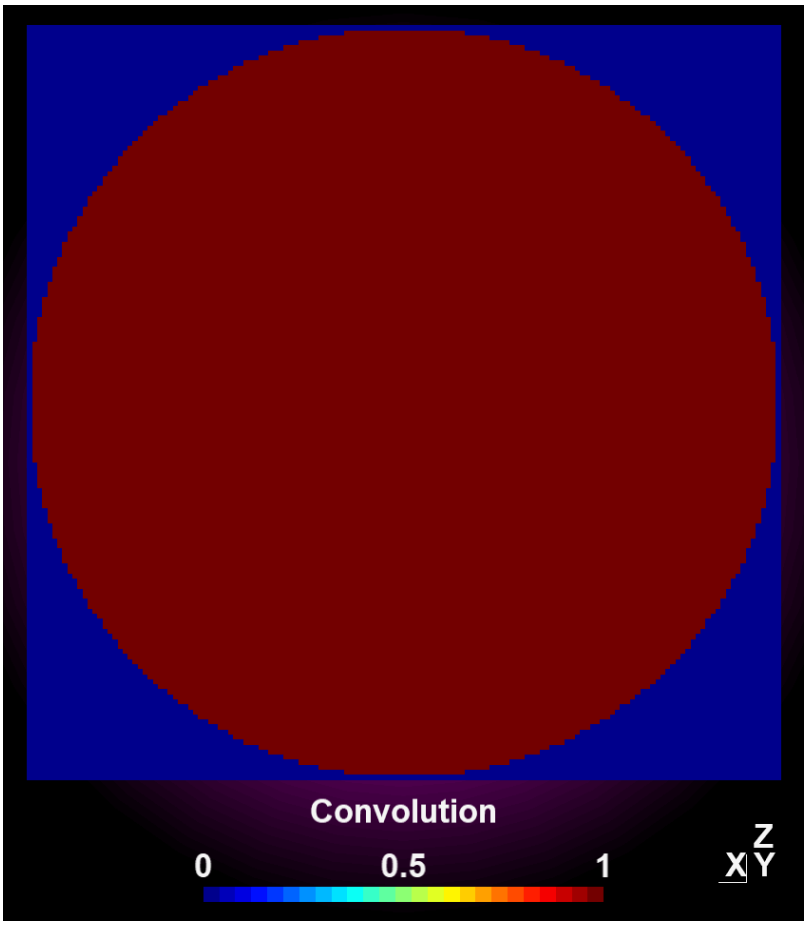
obj1.xz_plane(0.0088, 1, "nearest_neighbour");

obj1.reduce; #Operate on cut plane (2D) data set for better performance!

obj1.convolution_circular();

obj1.write_pos_scalar_binary("absorption_Arplus_100.00us_xz-plane_convolution.pos", val);
```





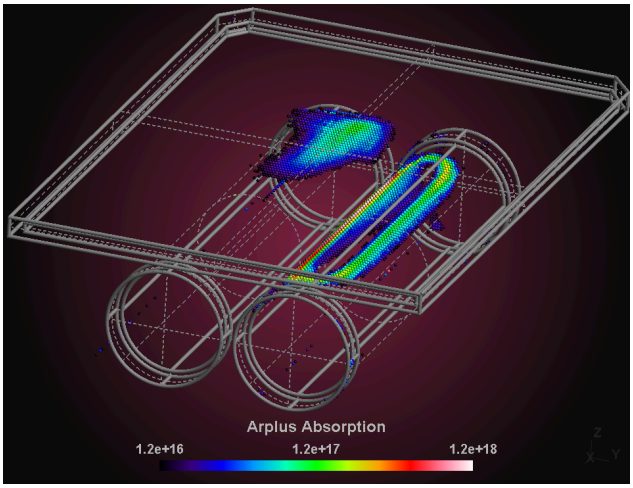
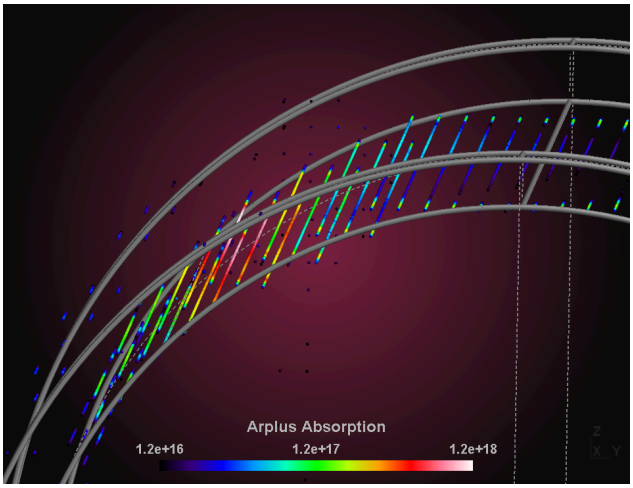
Source data.	After circular convolution.

Footnotes

1. The acronym *PPP* stands for **P** IC-MC **P** ost **p** rocessing ;- ↩

62 Postprocessing examples

An important result in PIC-MC plasma simulations of sputter cathodes is the ion current absorption profile on the target, because this information can be subsequently used for determining the sputter erosion profile. With planar targets the extraction of the absorption data in a cut plane on the target surface is rather simple. However things become more complicated in case of curved substrates as e.g. in case of cylindrical sputter targets:

Ar^+ absorption plot in whole model	Ar^+ absorption cells at cylindrical target
	

The absorption plot comprises the absorbed particle flux per second and per unit area and is stored as 3D volume plot within the cartesian volume grid structure. It can have only non-zero values in cells which intersect with meshed walls. For area normalization, the intersecting cell surfaces are used. These are always computed together with the effective cut-size cell volumina during initialization of a simulation run and stored in the file `simulationcase/cell_surfaces.pos`.

The right graph in the figure above shows an enlarged section of the absorption profile close to the target surface thereby illustrating the main issue in obtaining accumulated absorption profiles on curved substrates: The cartesian volume grid structure does not fit well to the cylindrical shape of the surface. Therefore, the cells intersecting with the cylinder form a step-like structure, and partially multiple intersecting cells are aligned on top of each other.

The solution is to consider the product *S A* between cell surface *S* and normalized absorption *A* and to take the following average along vertical direction through all cells intersecting with the cylindrical surface:

$$< A >= \frac{\sum\limits_{i \in \text{cells}} S_i A_i}{\sum\limits_{i \in \text{cells}} S_i}$$

This way, the *total* absorption cumulated over all intersecting cells in vertical direction is divided by the *total* intersecting cell surface, which is the proper way of averaging the absorption profile on curved surfaces. With the RIG VM postprocessing tools this can be accomplished within following script.

```
compiletime load "module_ppp.so";
case = BATCH.file;

PPP obj = {
  # Create forward reference for the maximal
```

```
  # absorption at left and right side from center line:
  float AMAX : 0.5*(A_left_max + A_right_max + abs(A_left_max - A_right_max));
};

T0 = 15;     # Start time for time averaging
T1 = 20;     # End time for time averaging
DT = 2.5;    # Plot interval
radius = 75; # Target radius
tilt   = 10; # Magnet tilt angle

# Compute resulting center coordinate of absorption profile
pi = 4*atan(1);
Y_center = 92.5 - radius*sin(tilt*pi/180);

# Make center coordinate also visible in obj
obj.Y_center = Y_center;

# Read averaged absorption and cell surfaces
for(time=T0; time<=T1+1e-6; time=time+2.5) {
  obj.read("$case/absorption/absorption_Arplus_$(time%,2f)us.pos", "A");
}
obj.read("$case/cell_surfaces.pos", "S_All");

# Consider only surfaces, where Arplus absorption is non-zero
obj.create_scalar("S", S_All*A>0 ? S_All ! 0);

# Create product A*S
obj.create_scalar("AS_product", S*A);

# Make horizontal cutplane at Target T2 averaging across upper half surface
obj.plane(-298, 0, 205,   298, 0, 205,    -298, 170, 205,   40, 0, 0, "average");
obj.reduce();

# Create averaged absorption sum A*S / sum S (if sum S > 0)
obj.create_scalar("A_avg", S>0 ? AS_product/S ! 0);

# Save plane view with projected absorption for illustration
obj.write_pos_scalar_binary("extract/$case-Arplus2D-T2.pos",
                            "Averaged Arplus absorption %",
                            A_avg/A_avg_max*100);

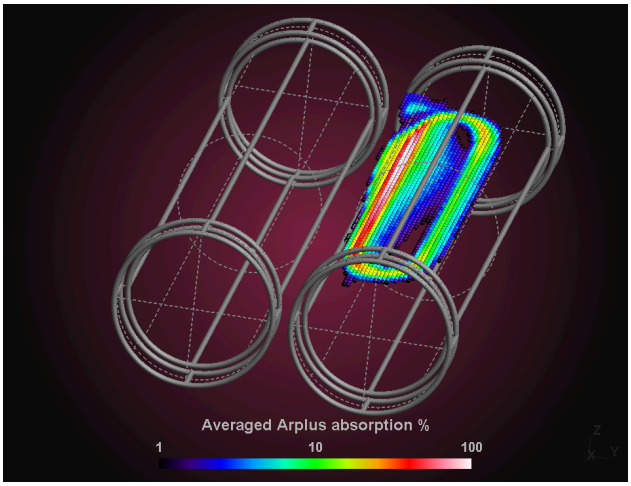
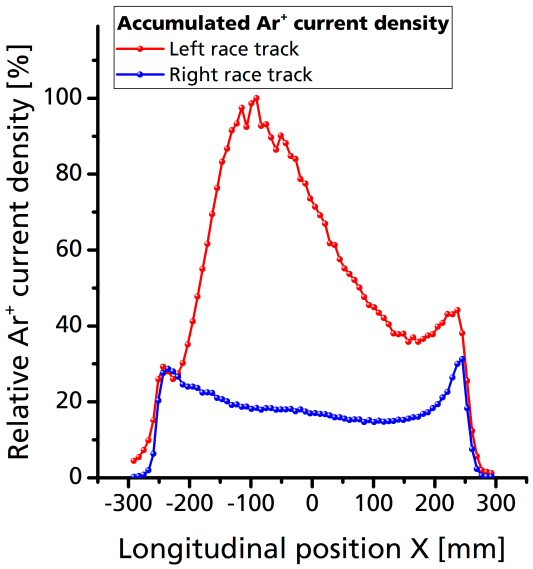
# Make cumulated line profiles

# Create scalars for distinguishing absorption on left and right side from center
obj.create_scalar("A_left",  y<Y_center ? A_avg ! 0);
obj.create_scalar("A_right", y>Y_center ? A_avg ! 0);

# Convert plane profile into accumulated linescan
obj.xz_plane(Y_center, radius, "sum");

# Write relative line profiles into TXT file
obj.write_txt("extract/$case-T2-lineprofiles.txt",
             ["x", "A_left_%", "A_right_%"],
             [x, 100*A_left/AMAX, 100*A_right/AMAX]);
```

The pictures below show the accumulated Ar^+ flux distribution projected onto the 2D cutplane (left graph) as well as the cumulated line profiles of the racetracks on both sides (right graph).

Ar^+ current density projected onto plane	Cumulated line profiles of both race tracks
	

Projection of quantities onto cylindrical surfaces

Example 1: Magnetic field

For a cylindrical surface and a given magnetic field, it is often interesting to plot the horizontal and vertical magnetic field component. This can be obtained via the cylindric cutplane function embedded in the RIG-VM postprocessing toolkit. In the following example, a cylindrical target has the following dimensions:

- Length: 520 mm from X=115 ... 625
- Center: Y = 167.5, Z = 247
- Radius: 70 mm

A postprocessing script for extracting the magnetic field on the cylindrical target surface is given as follows.

```
compiletime load "module_ppp.so";
case = BATCH.file;

X0 = 115;   X1 = 635;    # Cylinder elongation in X direction
Y0 = 167.5; Z0 = 247;    # Center Coordinates
R0 = 70;                 # Cylinder radius
Delta = 5;               # Search radius

NL   = 60;               # Point resolution in longitudinal direction
NPHI = 75;               # Point resolution in angular direction

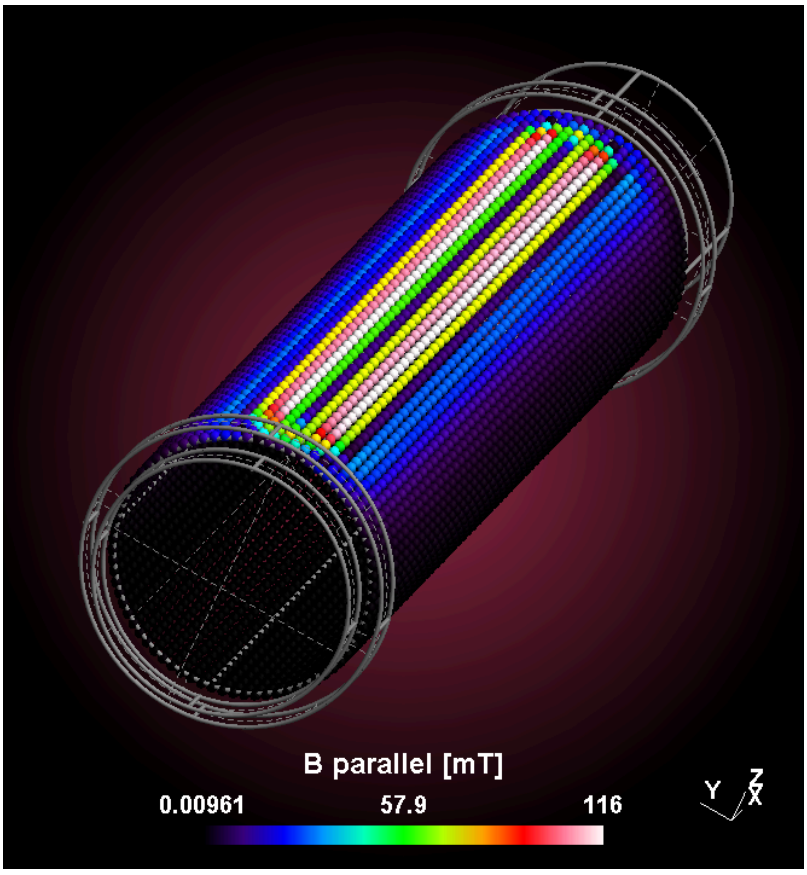
PPP obj = { };
obj.read("$case/field/B_0.00us.pos", "B");

obj.cylinder(X0, Y0, Z0,   X1, Y0, Z0,    R0, Delta,  NL, NPHI, "interpolate");

obj.write_pos_scalar_binary("extract/$case-B-radial.pos", "B radial [mT]", B_r*1000);
```

```
obj.write_pos_scalar_binary("extract/$case-B-parallel.pos", "B parallel [mT]", sqrt(B_x^2 +
B_phi^2)*1000);
```

The resulting POS file can be visualized in GMSH as follows:



The point resolutions `NL, NPHI` in longitudinal and angular direction shall be in the same order as the resolution of the simulation output data. There may be some trials with different settings required for getting a proper visualization.

Example 2: Ion absorption profile

While some quantities such as the magnetic field exist continuously in the 3D space, other quantities such as the "absorption" exists only at the boundary between open volume and walls. For the absorption data stored in cell resolution this means, that only cells which intersect with a meshed wall can have an absorption > 0.

Therefore, taking a simple average of the absorption across several cells in the vicinity of the surface is not useful, because some of the cells will contribute zero absorption.

A workaround is to additionally load the cell surfaces *S* and take an average of the absorption *A* weighted by the cell surface

$$< A >= \frac{\sum_{cells} A \times S}{\sum_{cells} S}$$

This can be applied in a postprocessing script as shown below.

```
compiletime load "module_ppp.so";
case = BATCH.file;

X0 = 115;   X1 = 635;    # Cylinder elongation in X direction
Y0 = 167.5; Z0 = 247;    # Center Coordinates
R0 = 70;                 # Cylinder radius
Delta = 5;               # Search radius

NL   = 60;               # Point resolution in longitudinal direction
NPHI = 75;               # Point resolution in angular direction

PPP obj = { };
obj.read("$case/absorption/absorption_Arplus_10.00us.pos", "A");
obj.read("$case/cell_surfaces.pos", "S");

obj.create_scalar("AxS", A*S);
obj.cylinder(X0, Y0, Z0,   X1, Y0, Z0,    R0, Delta,  NL, NPHI, "average");
obj.write_pos_scalar_binary("extract/$case-abs-cyl-Arplus.pos",
                            "Ar+ absorption [1/m^2 s]",
                            (S>0) ? AxS / S  ! 0);
```

63 Matrix deposition



info

The matrix deposition is one of two ways to compute the deposition. Release 26 adds the substrate raytracer, a separate program that reads the same histogram files but works with real part geometries, moving masks and an automatic mask optimisation. This page stays valid for both releases.

Trajectory object

This object controls the trajectory of the substrate. At the same time, it determines the coordinate system of the setup. The trajectory is to be defined within an `MPROJECTOR` object.

Within the object, an internal coordinate system is used, whereof the axes have the following meaning:

- X axis - This is the radial coordinate pointing from the center of the turntable towards outside (into the direction, where PHI=0°)
- Y axis - This is the tangential coordinate which points into the move coordinate (into the direction, where PHI=90°)
- Z axis - This is the vertical distance from the substrate holder

The external coordinate system of the simulation setup may be different. By definition of appropriate unit vectors in the Trajectory object, the correct orientation of the coordinate system within this module can be chosen.

Script syntax

The definition syntax is as follows:

```
Trajectory_type="circular";
trajectory: {
  PHI_0 = -19; PHI_1 = 19; NPHI = 381;
  x0    = -232;   y0 = 260; z0 = z_front;
  R     = 540;

  nx = 0; ny = 0; nz = -1;
  tx = 1; ty = 0; tz = 0;
};
```

There are the following input parameters for a “circular” trajectory.

- `PHI_0, PHI_1` = The interval of the turntable rotation.
- `NPHI` = Number of turntable rotation steps
- `x0, y0, z0` = The point (x0, y0, z0) corresponds to the center of the turntable in lateral coordinates and to the central point of the substrate in vertical coordinates.
- `R` = Radius between center of turntable and center of substrate
- `nx, ny, nz` = Unit vector pointing from turntable towards the histogram plane. This is parallel to the Z axis mentioned above.
- `tx, ty, tz` = Unit vector pointing in radial direction from center to outside; which is identical to the X axis mentioned above.

Relevant C++ routines

- `Trajectory_UTX::set_plot_position()`: Put the substrate into a central position for plotting
- `int Trajectory_UTX::get_nsteps()`: Return the number of steps of the move trajectory
- `Trajectory_UTX::get_coordinates(double &_xt, double &_yt, double &zt, double &PHI_t)`: Retrieve the current coordinates of substrate center. The angle `PHI_t` is used for the relative rotation of the substrate with respect to the external coordinate system.
- `Trajectory_UTX::get_origin(double &_x0, double &_y0, double &_z0)`: Get coordinates of center of turntable. If the turntable is placed within XY plane, the Z coordinate `_z0` is at the same vertical distance from the turntable as the center surface of the substrate.

- `Trajectory_UTX::set_index(int &_index)`: Go to the step `_index` of the move trajectory. The variable `_index` has to be between 0 and `Trajectory_UTX::get_nsteps()-1`.

Histogram object

How to provide matrix histogram files

A matrix histogram is an array of `iX*iY` histograms arranged in a rectangular array. It can be extracted from a DSMC simulation via the histogram array technique. The syntax to use within a `reactions: { … }` block is

```
add_histogram_array("species", nTheta, nPhi, E0, E1, nE, nx, ny, x0, y0, z0, x0+dx1, y0+dy1,
z0+dz1, x0+dx2, y0+dy2, z0+dz2);
```

The histogram array is applied on a single rectangular membrane with the following coordinates:

$$\begin{aligned}\vec{x} &= (x_0, y_0, z_0)\\ \vec{x}+\vec{\delta x_1} &= (x_0+\delta x_1, y_0+\delta y_1, z_0+\delta z_1)\\ \vec{x}+\vec{\delta x_2} &= (x_0+\delta x_2, y_0+\delta y_2, z_0+\delta z_2)\\ \vec{x}+\vec{\delta x_1}+\vec{\delta x_2} &= (x_0+\delta x_1+\delta x_2, y_0+\delta y_1+\delta y_2, z_0+\delta z_1+\delta z_2)\end{aligned}$$

In case of a sputter coater with turntable, please use the following conventions:

- $\vec{\delta x_1}$ should be the **radial coordinate** and point from the turntable center towards outside (i.e. the "X" direction mentioned above)
- $\vec{\delta x_2}$ should be the **tangential coordinate** which is identical with the "Y" direction mentioned above.

In the GMSH mesh geometry, make sure that the normal vectors from the Matrix mesh point *towards the arriving particles*.

Example: in the model of the EOSS coater, there is a histogram sampling plane spanning between the coordinates (225, 120, 380) → (525, 400, 380). The turntable is located in XY plane, and its lower Z coordinate is Z0=382. The upper coordinate of the uniformity masks is Z=380; this is identical with the Z coordinate of the histogram membrane. The radial direction is identical to the positive X axis direction, and the tangential direction corresponds to Y. Thus, an appropriate way to define the histogram array would be

```
add_angular_histogram_array("Ta", 9, 36,   30, 28,   225, 120, 380,   525, 120, 380,   225, 400,
380);
```

The first two parameters correspond to the number of angular slots for θ, ϕ - corresponding to a 10° resolution. The second two parameters (30, 28) are the number of histogram slabs in the first and second direction. With $\vec{\delta x_1} = (300, 0, 0)$ and $\vec{\delta x_2} = (0, 280, 0)$ this corresponds to a spatial resolution of 10 mm in both directions.

During a DSMC simulation, histogram files are produced in the `histogram/*` subdirectory of the simulation run folder. These histogram files are plain ASCII. Their first three columns consist of (i) energy in eV, (ii) polar angle and (iii) azimuthal angle. The following `nx*ny` columns consist of the sampled particle counts from all of the histogram slabs. The column number (starting with 0) depends on the indices ix, iy in the following way:

$$i_c = i_c + n_c i_c \ \in [0, \ldots, n_x n_y - 1]$$

On the other hand, the indices `ix, iy` can be retrieved from the column number via

$$i_x = i_c \bmod n_x \ \in [0, \ldots, n_x - 1]$$
$$i_y = \lfloor i_c / n_x \rfloor \in [0, \ldots n_y - 1]$$

The histogram files are generated for each plot snapshot. For better statistics please use the script contained in this ZIP file for time averaging. This is done via following steps:

- The script should be placed in the same directory as the *.PAR files
- Create a subdirectory via

```
mkdir histogram-matrix
```

- Edit the headlines of the script. You should adjust the time averaging interval as well as specify the species for each of the matrix histograms attached to the sampling membrane:

```
T0 = 1000;          # Time averaging start
T1 = 3000;          # Time averaging end
DT = 50;            # Time between two snapshots
TUNIT  = "ms";      # Time unit (as written in the filenames)
sensor = "Matrix";  # Physical name of matrix membrane
EXPORT = "plot";    # use "plot" for use in projector algorithm

var species[] -> ["Ta", "Ta", "Ta"];
```

In the example, time averaging is performed from 1000 to 3000 ms with a time step of 50 ms between snapshots. There are three different histograms attached to the sampling membrane, each of them samples arriving Tantalum (“Ta”) species.

- After having adjusted the file `histogram-matrix.r`, type the command:

```
rvm -b histogram-matrix.r simcase.par
```

Internal histogram representation

Within `module_mprojector`, histograms are represented by the C++ class `Histogram`. It has the following relevant member functions:

- `Histogram::alloc(int _ntheta, int _nphi)`: Allocate memory for internal variable `double *intensity` to store count values for `_ntheta*_nhi` angle combinations
- `Histogram::restore()`: Fill all intensity entries with 0.
- `Histogram::read_file(char *_fullpath, int *_debugflag=0)`: Read histogram from single histogram file. *This function is deprecated and will be removed.*
- `double Histogram::get_intensity(int _itheta, int _iphi)`: Get counts for `_itheta, _iphi` combination.
- `Histogram::set_intensity(int _itheta, int _iphi, double _intensity)`: Set counts for `_itheta, _iphi` combination.
- `Histogram::add_intensity(int _itheta, int _iphi, double _intensity)`: Add counts for `_itheta, _iphi` combination.

In the object `MProjector_UTX`, histograms are stored in the component `Histogram *MProjector_UTX::histograms`. In previous versions, single histograms could be read in via many separated histogram slabs, however this method is too cumbersome and has therefore been removed.

In the current version, all histograms are obtained from the single histogram matrix file described above. The corresponding input section in the script file looks like follows:

```
Hist_NTHETA  = 9;
Hist_NPHI    = 36;
Hist_DELTAPHI= 90;
Hist_Z       = 380;

Hist_X0      = 225;     Hist_NX = 30;    Hist_DX = 10;
Hist_Y0      = 120;     Hist_NY = 28;    Hist_DY = 10;

Hist_Filename= "histogram-matrix/$(histlabel)_histogram_3_Matrix_Ta.txt";
```

The coordinates of the histogram slabs are now passed via the parameters `Hist_X0, Hist_NX, Hist_DX, Hist_Y0, Hist_NY, Hist_DY`. The coordinates `Hist_X0, Hist_Y0` correspond to the lower edge coordinates of the sampling membrane in the GMSH file[1]. The third coordinate is `Hist_Z` which represents the vertical position of the histogram plane[2].

Footnotes

1. internally, the coordinates will be shifted by `+Hist_DX/2, +Hist_DY/2`. Thus, that they correspond to the center points of the membrane slabs rather than to the edge coordinate; this is required by the bilinear interpolation algorithm. **Note:** In the previous version, the first center coordinates rather than the edge coordinates were required in the input file which caused some confusion. ↩

2. Note: In case if the external coordinate system is differently rotated, please keep in mind that `Hist_X0` always represents the lower coordinate of the first directional vector, `Hist_Y0` the one of the second directional vector, and `Hist_Z` the vertical coordinate of the plane. ↩

64 Postprocessing with GMSH

Extracting ion flux onto rotatable targets

This section corresponds to 3D PIC-MC simulations of cylindrical sputter targets. In order to predict the sputter erosion profile, the normal component of the ion flux with respect to the target surface is required. The ion flux is defined as the product of ion density and velocity.

For a planar target, where the sputter direction is e.g. parallel to the Z axis, it is very trivial to obtain the averaged flux by following steps:

- Merge the `density_Arplus_xxxx.pos` file into GMSH (--> View 0)
- Merge the `velocity_Arplus_xxxx.pos` fils into GMSH (--> View 1)
- Use the **MathEval** plugin of GMSH in order to create the product density and vz
- Use the **CutPlane** plugin of GMSH in order to extract the plane just above the target surface

Alternatively, the absorption plot `absorption_Arplus_xxxx.pos` can be used in this case.

For cylindrical rotating targets, the absorption plot might be not sufficient, since - due to the use of a cartesian grid in PIC-MC - the cells adjacent to the target surface form a step-like structure rather than a smooth cylindrical surface.

Thus, the scripts introduced here have the following purposes:

- Make a smooth projection of the ion flux onto the cylindrical surface
- Perform a time average over the last N snapshots
- Create POS files of the projected absorption on the curved target surface, which can be triangulated
- Finally performing circular integration of the ion flux in order to predict the effective erosion profile.

First step: Create ASCII files of the ion flux in the relevant area

In order to work properly, the extraction script needs the ion fluxes as ASCII text file with following restrictions:

- The data should be restricted to the region adjacent to the target surface of interest
- The data points should have **equal spacing** in X, Y, and Z direction

In order to prepare the ASCII files, go into the folder of your PAR files and create the following directories:

- `./extract`
- `./extract/flux`

Within the `./extract` directory, create a file `make-flux.geo` with a content like follows:

```
time = 0;
For time In {50 : 60 : 1.0}

    Merge Sprintf("../frame-dual-3D/velocity_Arplus_%.2fus.pos", time);
    Merge Sprintf("../frame-dual-3D/density_Arplus_%.2fus.pos",  time);
```

```
    Plugin(MathEval).Expression0= "v0*w0";
    Plugin(MathEval).Expression1= "v0*w1";
    Plugin(MathEval).Expression2= "v0*w2";
    Plugin(MathEval).Expression3= "";
    Plugin(MathEval).Expression4= "";
    Plugin(MathEval).Expression5= "";
    Plugin(MathEval).Expression6= "";
    Plugin(MathEval).Expression7= "";
    Plugin(MathEval).Expression8= "";
    Plugin(MathEval).TimeStep=-1;
    Plugin(MathEval).View=1;
    Plugin(MathEval).OtherTimeStep=-1;
    Plugin(MathEval).OtherView=0;
    Plugin(MathEval).Run;

    Plugin(CutPlane).A=0;
    Plugin(CutPlane).B=0;
    Plugin(CutPlane).C=1;
    Plugin(CutPlane).D=-265;
    Plugin(CutPlane).ExtractVolume=1;
    Plugin(CutPlane).RecurLevel=4;
    Plugin(CutPlane).TargetError=0;
    Plugin(CutPlane).View=2;
    Plugin(CutPlane).Run;

    Plugin(CutPlane).A=0;
    Plugin(CutPlane).B=0;
    Plugin(CutPlane).C=1;
    Plugin(CutPlane).D=-340;
    Plugin(CutPlane).ExtractVolume=-1;
    Plugin(CutPlane).RecurLevel=4;
    Plugin(CutPlane).TargetError=0;
    Plugin(CutPlane).View=3;
    Plugin(CutPlane).Run;

    Plugin(CutPlane).A=0;
    Plugin(CutPlane).B=1;
    Plugin(CutPlane).C=0;
    Plugin(CutPlane).D=-260;
    Plugin(CutPlane).ExtractVolume=-1;
    Plugin(CutPlane).RecurLevel=4;
    Plugin(CutPlane).TargetError=0;
    Plugin(CutPlane).View=4;
    Plugin(CutPlane).Run;

    Plugin(CutPlane).A=0;
    Plugin(CutPlane).B=1;
    Plugin(CutPlane).C=0;
    Plugin(CutPlane).D=70;
    Plugin(CutPlane).ExtractVolume=1;
    Plugin(CutPlane).RecurLevel=4;
    Plugin(CutPlane).TargetError=0;
    Plugin(CutPlane).View=5;
    Plugin(CutPlane).Run;

    // Creating POS files is optional and can be omitted
    Save View[6] Sprintf("flux/flux_Arplus_%.2fus.pos", time);
    Save View[6] Sprintf("flux/flux_Arplus_%.2fus.txt", time);

    Delete View[6];
    Delete View[5];
    Delete View[4];
    Delete View[3];
    Delete View[2];
    Delete View[1];
```

```
    Delete View[0];
EndFor
```

This script loads the density and the velocity plot of a given time step and performs the **CutPlane** plugin four times in order to restrict the resulting vector plot to the region around the target surface. **It is recommended to perform these steps once in GMSH manually and to transfer the CutPlane parameters to the script above**. Afterwards the script can be invoked with the Linux binary of GMSH in batch mode:

```
gmsh make-flux.geo -
```

After running the GMSH script, the flux files created under `./extract/flux` should look like follows[1]:

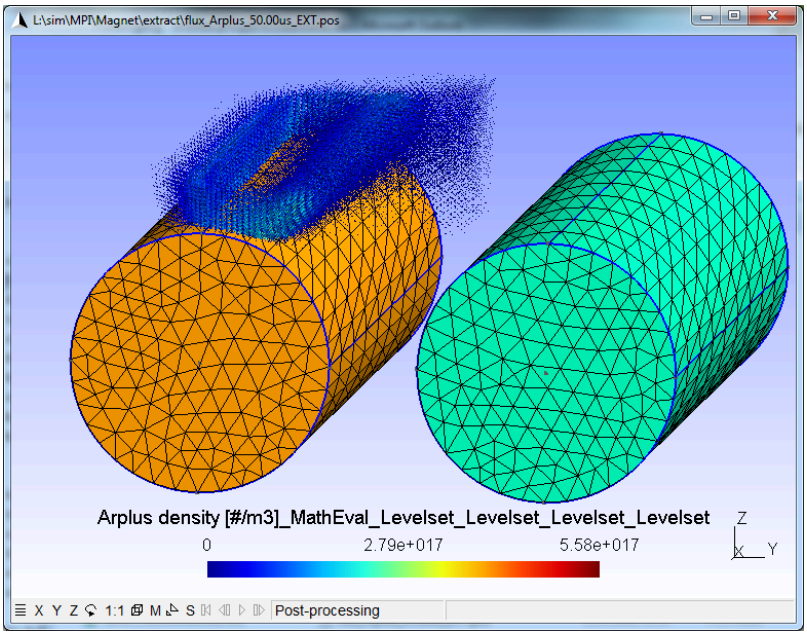


Fig. 1: Cylindrical target geometry with extracted flux in region of interest

Second step: Make a projection of the flux onto the curved target surface

The following RIG-VM script assumes, that the target axis is parallel to the Z axis. If that is not the case in the actual geometry, there is the possibility to perform an internal coordinate rotation within the script.

Before working with the RIG-VM script, you have to find out the number of cells as well as the lowest and highest coordinate in each direction (X, Y, Z), which can be done by viewing one of the ASCII flux files within a text editor.

All of the required adjustments are to be made in the upper part of the script, as summarized in the following:

- **BASENAME**: Template for input filenames. Change this e. g. if your ion species is different from `Arplus`
- **ColX, ColY, ColZ**: Column offsets for the X, Y, and Z coordinate, should be either (0, 1, 2), (1, 2, 0), or (2, 0, 1). If the cylinder is already oriented along z-axis chose the first one (i. e. ColZ=2). If the cylinder is oriented along X axis in the GEO file, chose the combination, where ColZ=0. If it is oriented along Y axis chose ColZ=1.

- **IZ0, IZ1**: The lowest and highest **cell index** along target axis, where the flux evaluation should be performed. Make sure, that the full racetrack is comprised within that range.
- **NX, NY, NZ**: Number of cells in each coordinate direction. **Important:** If the coordinate system is rotated, that also applies for these values. Remember that the Z-axis is always parallel to the target axis...
- **X0, X1, Y0, Y1, Z0, Z1**: The lowest and highest coordinate in each direction.
- **Alpha0, Alpha1, dAlpha**: The lowest and highest angular value as well as the angular step in order to define the relevant circle arc on the target surface. Make sure that this region is fully comprised within the flux field.
- **XM, YM**: The center of the cylinder in the XY plane.
- **R0**: The radius of the cylindrical target
- **DR**: A safety margin in the order of one half cell spacing in order to prevent, that empty cells are considered within the evaluation.

If all of these values are properly adjusted, the script, which is shown in the following, can be run with

```
rvm extract_flux.r
```

```
# extract_flux.r
# 3D flux files should be present under ./flux/flux_Arplus_*.txt

integer IZ0 = 13,   IZ1 = 86;   # Mininum and maximum Z coordinate index
float   XM  = 167.5, YM = 247;  # Center coordinate of cylinder in XY plane
string BASENAME : "flux_Arplus_$(time%,2f)us";  # Scheme of input filenames

integer ColX=1, ColY=2, ColZ=0; # Column offsets for rotating the coordinate system
                                # If the cylindricat target in the geo file is already
                                # oriented in Z direction, use ColX=0, ColY=1, ColZ=2

# Number of entries in X, Y, Z
# Lowest and highest coordinate in X, Y, Z, respectively

integer NX = 70;  float X0 = 47.03214285714284, X1 = 258.4678571428572, DX = (X1-X0)/(NX-1);
integer NY = 37;  float Y0 = 267,               Y1 = 339,               DY = (Y1-Y0)/(NY-1);
integer NZ = 100; float Z0 = 3.75,              Z1 = 746.25,            DZ = (Z1-Z0)/(NZ-1);

float Alpha0 = -30, Alpha1=50, dAlpha=1;  # Extracted angular range
float R0     = 69;                        # Target diameter plus safety margin
float DR     = 2;                         # Safety margin in the order of at least one half cell
spacing

#
================================================================================================
====
# No more adjustments needed from this point on
#
================================================================================================
====

string str12(x=float) = return "$("$(x)"%12s)";
integer dim_fn = (IZ1-IZ0+1)*((Alpha1-Alpha0)/dAlpha+1);
float FN_Sum[dim_fn];

float realX(x=float, y=float, z=float) = return ColX==0 ? x ! ColY==0 ? y ! z;
float realY(x=float, y=float, z=float) = return ColX==1 ? x ! ColY==1 ? y ! z;
float realZ(x=float, y=float, z=float) = return ColX==2 ? x ! ColY==2 ? y ! z;

block extract_flux(BASENAME=string, sum->var) : {
  float bilinear(f00=float, f10=float, f01=float, f11=float, x=float, y=float, x0=float, y0=float,
x1=float, y1=float) = {
```

```
  float F = 1/((x1-x0)*(y1-y0));
  return F*(f00*(x1-x)*(y1-y) + f10*(x-x0)*(y1-y) +
        f01*(x1-x)*(y-y0) + f11*(x-x0)*(y-y0));
};

float index(x=float, y=float, z=float) =
  return floor((x-X0)/DX) + NX*floor((y-Y0)/DY) + NX*NY*floor((z-Z0)/DZ);

float simeq(a=float, b=float) = return abs(a-b)<1e-6 ? 1 ! 0;

console "** Reading $(BASENAME).txt **\n";
SHEET M={
  filename="flux/$(BASENAME).txt";
  startline=1; separator=" ";
};

integer iz;
float   zplane -> Z0 + DZ*iz;

console "** Generating flux fields **\n";
float FX[NX*NY*NZ], FY[NX*NY*NZ], FZ[NX*NY*NZ];

integer i, ind;
for(i=0; i<M.n_rows; i=i+1) {
  float x = floor((M.data[4+ColX][i]-X0)/DX+0.5)*DX+X0+DX/10,
        y = floor((M.data[4+ColY][i]-Y0)/DY+0.5)*DY+Y0+DY/10,
        z = floor((M.data[4+ColZ][i]-Z0)/DZ+0.5)*DZ+Z0+DZ/10;

  float fx= M.data[7+ColX][i],
    fy= M.data[7+ColY][i],
    fz= M.data[7+ColZ][i];

  ind = index(x, y, z);
  FX[ind] = fx; FY[ind] = fy; FZ[ind] = fz;
}

for(iz=IZ0; iz<=IZ1; iz=iz+1) {
  console "** Processing plane ix=$(iz) **\n";

  if(iz==IZ0) then file "$(BASENAME)_EXT.pos" << out "View\"$(BASENAME)\" {\n";

  append "$(BASENAME)_EXT.pos" << {
    float alpha, pi = 4*atan(1), x, y, x0, y0, z0, nx, ny, tx, ty;
    integer ind;

    for(alpha=Alpha0; alpha<=Alpha1+1e-6; alpha=alpha+dAlpha) {
  x =  XM+(R0+DR)*sin(alpha*pi/180);
  y =  YM+(R0+DR)*cos(alpha*pi/180);

  x0 = floor((x-X0)/DX)*DX + X0;
  y0 = floor((y-Y0)/DY)*DY + Y0;

  ind = index(x, y, zplane+1e-3);

      fx = bilinear(FX[ind], FX[ind+1], FX[ind+NX], FX[ind+NX+1], x, y, x0, y0, x0+DX, y0+DY);
      fy = bilinear(FY[ind], FY[ind+1], FY[ind+NX], FY[ind+NX+1], x, y, x0, y0, x0+DX, y0+DY);
      fz = bilinear(FZ[ind], FZ[ind+1], FZ[ind+NX], FZ[ind+NX+1], x, y, x0, y0, x0+DX, y0+DY);

  # Normal unit vector
      nx = (x-XM)/sqrt((x-XM)^2+(y-YM)^2);
      ny = (y-YM)/sqrt((x-XM)^2+(y-YM)^2);

  # Tangential unit vector
  tx = -ny;
  ty = nx;
```

```
    fn = fx*nx + fy*ny;    # Normal flux component
    ft = fx*tx + fy*ty;    # Tangential flux component

    out "SP($(realX(x, y, zplane)), $(realY(x, y, zplane)), $(realZ(x, y, zplane))){$(-fn)};\n";

        # fn auf externes Array addieren
        integer isum = (alpha-Alpha0)/dAlpha + ((Alpha1-Alpha0)/dAlpha+1)*(iz-IZ0);
        sum[isum] = sum[isum] + fn;
      }
    };

    if(iz==IZ1) then append "$(BASENAME)_EXT.pos" << out "};\n";
  }

  return 0;
};

integer count;
for(time=50.0; time<=60.0+1e-6; time=time+1) {
  extract_flux(BASENAME, FN_Sum);
  count=count+1;
}

for(i=0; i<FN_Sum.len; i=i+1) FN_Sum[i] = -(FN_Sum[i])/count;  # Flux unit is 1/m^2s

# Output of averaged flux as POS and TXT
file "AVG_flux_Arplus.pos" << {
  file "AVG_flux_Arplus.txt" << out "$("alpha"%12s)  $("z"%12s)  $("fn"%12s)\n";
  out "View\"AVG_Flux_Arplus\" {\n";
  integer iz;
  float   zplane -> Z0 + DZ*iz;
  for(iz=IZ0; iz<=IZ1; iz=iz+1) {
    float alpha, pi = 4*atan(1), x, y, x0, y0, z0, nx, ny, tx, ty;
    for(alpha=Alpha0; alpha<=Alpha1+1e-6; alpha=alpha+dAlpha) {
      x =  XM+(R0+DR)*sin(alpha*pi/180);
      y =  YM+(R0+DR)*cos(alpha*pi/180);
      integer isum = (alpha-Alpha0)/dAlpha + ((Alpha1-Alpha0)/dAlpha+1)*(iz-IZ0);
      out "SP($(realX(x, y, zplane)), $(realY(x, y, zplane)), $(realZ(x, y, zplane)))
{$FN_Sum[isum]};\n";
      append "AVG_flux_Arplus.txt" << out "$(str12(alpha))  $(str12(zplane))
$(str12(FN_Sum[isum]))\n";
    }
  }
  out "};\n";
};
```

After running the script, the normal fluxes extracted for the target surface for each time step can be found as POS file `flux_Arplus_XX.XXus_EXT.pos` within the `./extract` directory. Furthermore, two files, `AVG_flux_Arplus.pos` and `AVG_flux_Arplus.txt`, contain the projected flux averaged over the given time interval. The resulting representation in GMSH may look as in the following Fig. 2. Since the points are in regular arrangement, they can be triangulated in GMSH even if they are not located within a plane surface.

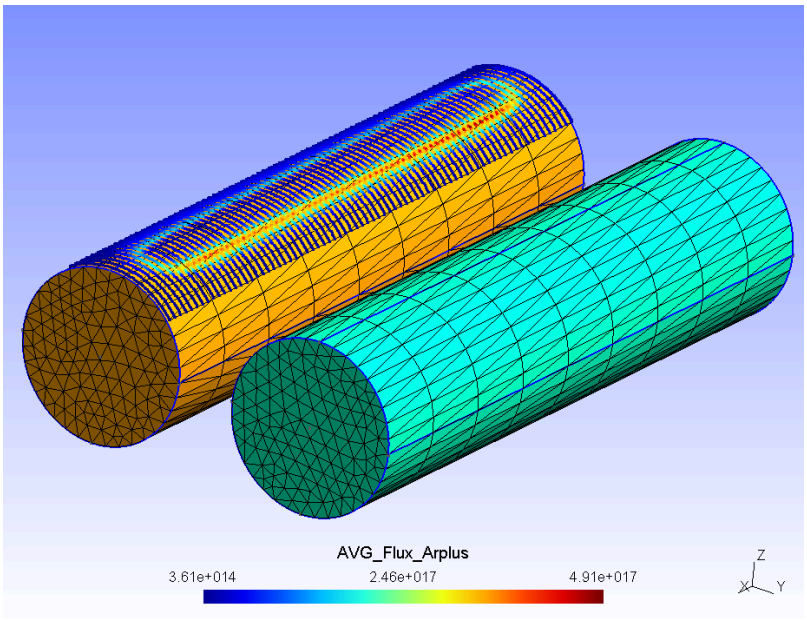


Fig. 2: Averaged projected flux on cylindrical target surface

Creating animations

As first step, multiple successive views have to be loaded into gmsh and combined into a sequence with the command `Combine Timesteps->From All Views` (see Fig. 3). The command `From Visible Views` would only consider the views with an active checkbox in front. Another possibility to combine plots would be to use one of the `write_pos_time_steps()` functions from the RVM postprocessing module.

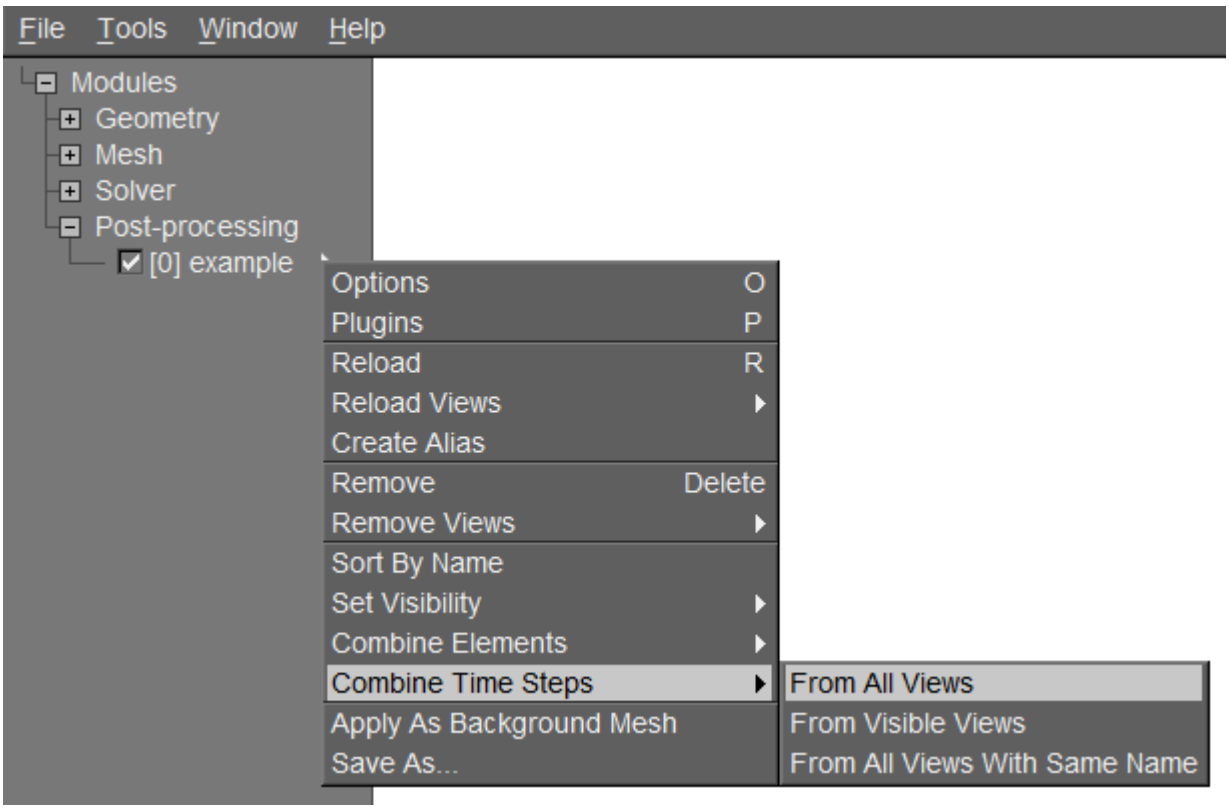


Figure 3: GMSH menu entry for creating image sequences

The next step is to create a GMSH script which subsequently saves each image of the sequence as PNG file. A possible example of such a script is given in the following:

```
Print.Background       =  1; // Use this, if the background color should be included
General.GraphicsWidth  =  1280;
General.GraphicsHeight =  720;

Steps = View[0].NbTimeStep; // Automatic determination of the number of time steps

For Num In {0:Steps-1}
  View[0].TimeStep = Num;
  Draw;
  Print Sprintf("ani-%02g.png", Num);
EndFor
```

With *Windows Movie Maker* these PNG files can be combined into a WMV film sequence. Another possibility is the use of *Blender* which is described step-by-step in this tutorial. For creating animated GIFs, the linux tool `convert` from the ImageMagick package (maybe not installed by default on your cluster) can be used as follows:

```
convert -delay 20 -loop 0 ani*.png output.gif
```

With the `-delay` switch, the time delay between frames can be specified in 1/100 seconds. The switch `-loop 0` creates an endless loop.

Footnotes

1. The POS files are not required in the following operations and can be deleted ↩

65 Animations with Blender

With the free and open-source 3D computer graphics software program Blender it is easily possible to create short animations from images extracted from GMSH. Although the program is much more powerful it is suitable for our purpose.

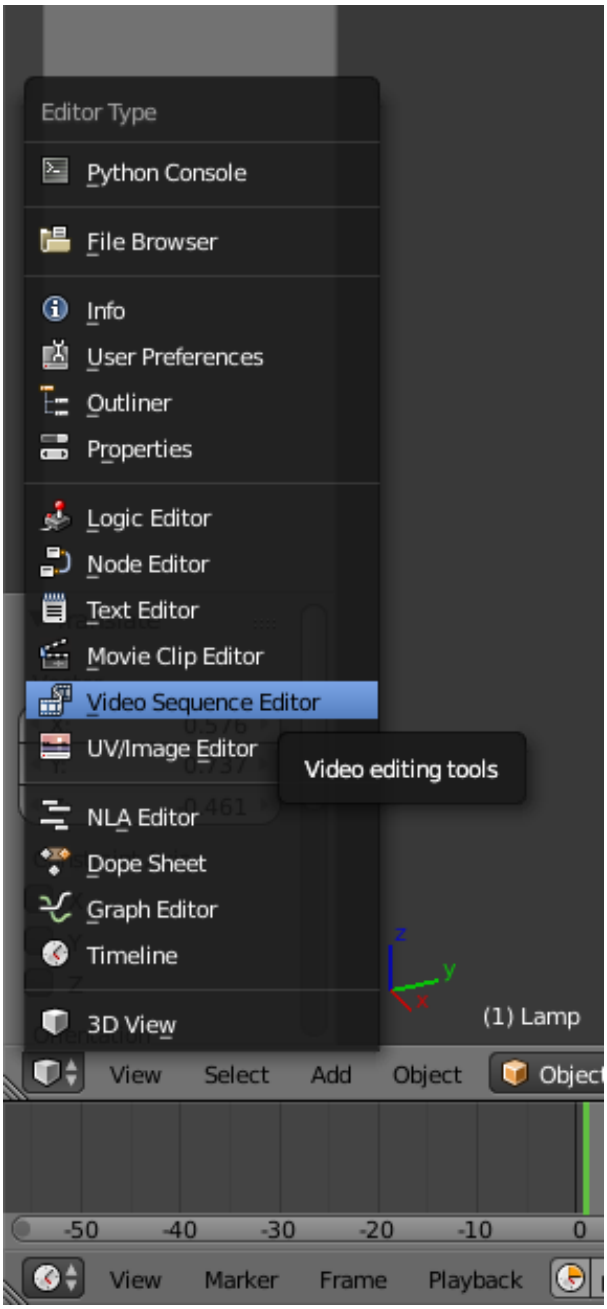
Step-by-step tutorial

Step 0

Download, install and open the program.

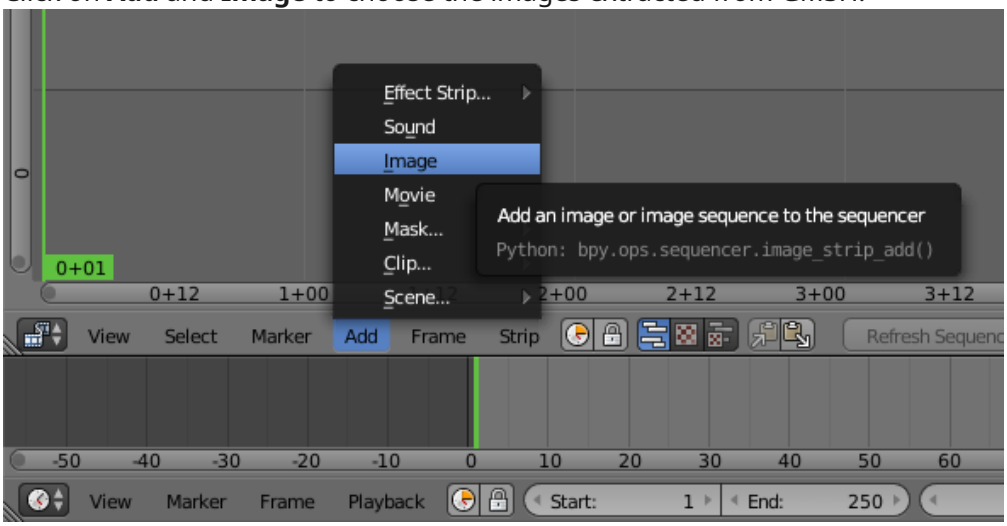
Step 1

On the left bottom side click on the cube icon and open the **Video Sequence Editor**.



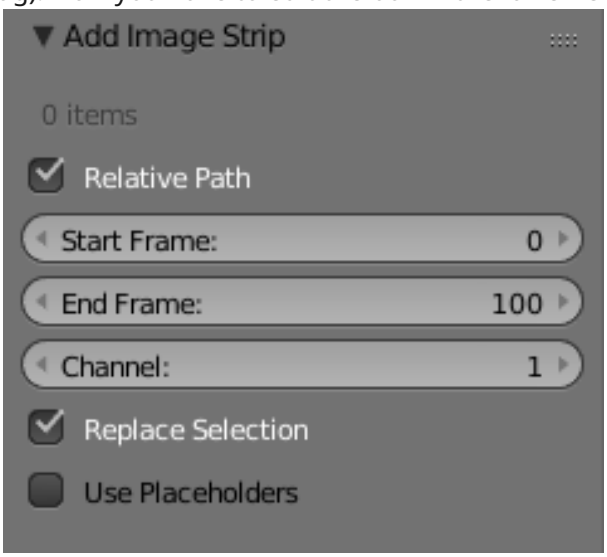
Step 2

Click on **Add** and **Image** to choose the images extracted from GMSH.



Step 3

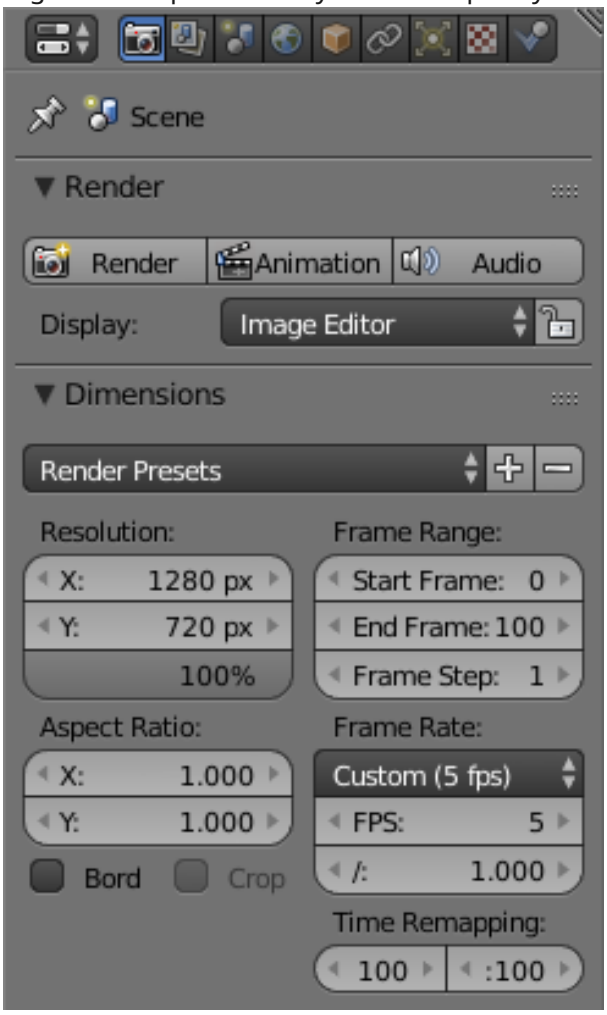
Navigate to the folder where the images are located and select them simply by making multiple selection boxes (hold left mouse button and drag). Now you have to edit the box in the lower left corner and adjust the start and end frame to your case.



Step 4

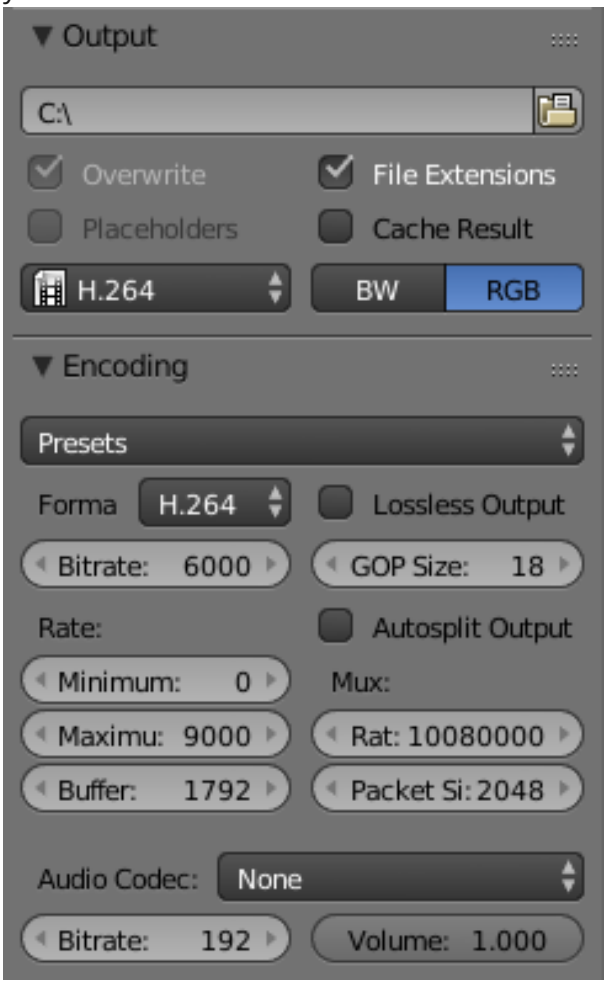
Then you have to open the render menu on the right by clicking on the camera symbol. Now the dimensions of the animation have to be set. Depending on the resolution of your images edit the **X** and **Y** value and set the percentage to 100. In the frame range you should use the same values as in step 3.

The frame rate namely the images shown per second you can adapt to your means, nevertheless a value of 5 is a good starting point.



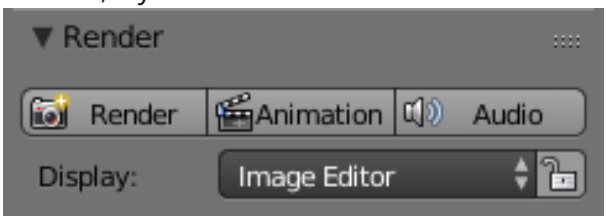
Step 5

Now the output path and the file format have to be set. For the file format H.264 is a good choice and you should select **RGB** for a coloured animation. In the encoding menu also select H.264 as the format.



Step 6

That's it, if you now click on **Animation** in the render menu your video is created.



Substrate raytracer

66 Getting started

Getting the tool

The raytracer is not part of the PICMC installation and is not published for download. Fraunhofer IST provides it as a source archive or as a ready installation for the project environment; the version described here is 0.2.0.

If you are not sure which files you have, look for the directory containing the `Makefile` (source archive) or for an installed tree with `build/libs/SRT.so` next to the helper scripts `init_venv` and `run_parallel` (installation).

Requirements

Building from the source archive needs:

- Linux. The build scripts are Bash; there is no Windows build out of the box.
- Python 3 with development headers and `python3-config`.
- `g++` with C++17 support.
- GNU Make.
- Optionally TBB (`libtbb-dev`). Recommended: without it the run is noticeably slower, but the results are identical.

Building

```
cd substrate-ray-tracer       # directory of the unpacked archive
make
```

The build produces `build/libs/SRT.so` and `build/libs/SRTHelper.py`. Then install into a directory of your choice:

```
INSTALLDIR=~/raytracer make install
```

`make install` also copies the two helper scripts `init_venv` and `run_parallel`.

If the build stops early, the two flags that resolve most cases:

```
# TBB not available — build without parallel execution (slower, same results)
NO_PARALLEL=1 make

# Python headers not found — point make at the right python-config
PYTHON_CONFIG_PATH=/usr/bin/python3.11-config make
```

For a fitting workflow you also need SciPy in the Python environment that runs the script. `init_venv` creates a virtual environment and installs it:

```
~/raytracer/init_venv ~/rtenv
```

Making the module importable

The script has to find `SRT.so`. Either place it in a subdirectory of the working directory and import it by path, or put it on `PYTHONPATH` and import it by name:

```
# SRT.so sits in path/to/, relative to the script
import path.to.SRT as SRT

# SRT.so is in the working directory or on PYTHONPATH
import SRT as SRT
import SRTHelper as SRTHelper
```

Without one of these lines nothing in the raytracer is available, and the error message is a plain `ModuleNotFoundError`, not a raytracer message.

A minimal script that only checks that the module is alive:

```
import SRT as SRT

rt = SRT.Raytracer()
s  = rt.get_substrate()
t  = rt.get_trajectory()

print("Raytracer version:", rt.get_version())
```

Checking an installation you did not build

Ask for the version before trusting a feature that is described here:

```
import SRT

print("Raytracer version:", SRT.Raytracer().get_version())
```

It has to read 0.2.0 or newer. An older number although a new installation was announced usually means that another `SRT.so` sits earlier in the load path (`PYTHONPATH`, working directory). If loading fails with `failed to map segment from shared object`, the library lies on a partition mounted `noexec`; move it to a normal filesystem.

Running on a cluster

`run_parallel` writes a SLURM batch file and submits it. Only the path to the installation, the Python environment and the number of processes are needed:

```
~/raytracer/run_parallel -raytracerpath ~/raytracer -env ~/rtenv -n 32 myscript.py
```

The job writes its output files, a log and an error file into the directory it was submitted from. Nothing else has to be configured — the raytracer itself parallelises over substrate patches, so the process count is the only tuning knob on that side.

67 Usage

Coordinate system

The z axis is the sputtering axis, x and y complete a right-handed system, and all vectors are given as `(x, y, z)`. There is no unit system — every length in the script has to use the same unit, and it has to be the unit of the histogram coordinates. Angles in the API are in degrees.

Spherical coordinates follow the ISO physics convention: the azimuthal angle is zero at the x axis and increases counter-clockwise when looking from positive z down onto the xy plane.

The four objects of a scenario

```
import SRT as SRT
import SRTHelper as SRTHelper

rt = SRT.Raytracer()      # the simulation
s  = rt.get_substrate()   # the part
t  = rt.get_trajectory()  # how the part moves
```

These objects carry the whole scenario. Create them once, at the top of the script, and reset them between runs instead of building new ones — especially for concave substrates, where the precomputation is expensive. At the start, neither the substrate nor the trajectory has any content; that is what the following sections fill in.

Substrate

The substrate is a set of quadrilateral patches generated from a parametric shape:

```
s.generate_geometry("spherical_lens", {
  "n_r": 25,          # patches in radial direction
  "n_phi": 60,        # patches in angular direction
  "r_curvature": 120.0,
  "r_base": 40.0
})

# Concave shapes need a precomputed self-intersection table
if s.is_concave():
  rt.precompute_substrate_intersections()

# Coat from below instead of from above
s.set_sputter_up()
```

The available shapes are disk, slanted rectangle, spherical lens, aspherical lens, toric lens, inverted spherical lens, bent rectangle and parabolic lens, plus custom triangle meshes. Every shape with its parameters and its exact parametrisation is listed on the Substrates page.

`precompute_substrate_intersections()` takes some time but prevents rays from passing through another part of a concave substrate. You can omit it for a quick test run, but not for a production result.

`s.set_sputter_up()` flips the geometry around the trajectory in z direction; everything else keeps its coordinates.

Trajectory

A straight line, a circular arc, or a list built by hand:

```
# Straight line
t.generate_line(11, SRT.PhysVector(100.0, 100.0, 100.0),
                    SRT.PhysVector(200.0, 100.0, 100.0), 90.0)

# Circular arc: samples, radius, start angle, end angle, centre, substrate rotation
t.generate_circle_arc(381, 540.0, -19.0, 19.0, SRT.PhysVector(-232.0, 260.0, 385.0), -19.0)
```

Both take the number of samples including the start and end point, and the initial substrate orientation in degrees. On a circular arc the substrate keeps its orientation towards the centre of the circle.

For anything else, build the list yourself — each entry is a position plus a rotation:

```
t.load_from_list([
  (SRT.PhysVector(100.0, 100.0, 100.0), 0.0),
  (SRT.PhysVector(150.0, 150.0, 100.0), 45.0),
  (SRT.PhysVector(200.0, 200.0, 100.0), 90.0),
])
```

Lists can be generated by normal Python code, so there is no need to write them out by hand. If the trajectory is defined by machine parameters instead of points — turntable RPM, carrier speed — use the helper module, which produces exactly such a list.

Check an unfamiliar trajectory in a viewer before running the raytracer:

```
t.dump_gmsh("trajectory.pos")
```

The time resolution is the parameter to check first: too coarse and the deposition shows stripes, too fine and the run takes hours. Halve it until the profile stops changing.

Histogram

The flux source is the histogram matrix written by DSMC or PICMC. Each line of such a file is one angle combination:

```
energy_eV  theta_deg  phi_deg  flux_1 .. flux_nx*ny
```

The first three columns are the energy in eV, the polar angle and the azimuthal angle in degrees. The following `nx*ny` columns are the sampled flux for each slab of the sampling plane, ordered with the x index running fastest:

```
i_column = i_x + n_x * i_y
```

A histogram batch is one source. Several histograms inside a batch are the same source sampled at different z coordinates — the raytracer uses the one closest to the patch it is looking at, which is what you want for strongly curved parts:

```
batch = rt.add_histogram_batch("Main")
batch.add_histogram_from_file(
  "histogram-matrix/histogram_3_Matrix_Ta.txt",
```

```
  SRT.PhysVector(225.0, 120.0, 380.0),   # lower corner of the sampling plane
  SRT.PhysVector(525.0, 400.0, 380.0),   # upper corner, same z
  30, 28,                                # n_x, n_y — must match the file
  90.0,                                  # rotate the plane around z if needed
  False                                  # swap x and y if the DSMC axes are swapped
)
```

`d_phi` and `swap_x_y` exist because the histogram files carry no orientation of their own. Check the orientation once with a large flat disk and no mask before trusting a curved result.

Several batches are independent sources and all of them are used. `batch.set_inactive()` switches one off without deleting it, `batch.set_active()` brings it back; a new batch is active by default.

Masks

A mask is a closed polygon in a plane of constant z, built from nodes connected by circular arcs, plus a thickness in sputtering direction. Rays crossing it are discarded.

```
m = rt.add_mask("Uniformity")
m.set_thickness(4.0)

n0 = m.add_node(SRT.PhysVector(100.0, 0.0, 380.0))
n1 = m.add_node(SRT.PhysVector(0.0, 100.0, 380.0))
n2 = m.add_node(SRT.PhysVector(-100.0, 0.0, 380.0))

m.add_circle_arc(n0, n1, 100.0)     # radius of the arc between the two nodes
m.add_circle_arc(n1, n2, 100.0)
m.add_circle_arc(n2, n0, 100.0)
```

Rules that the code enforces or that silently ruin a result:

- The mask must be closed, otherwise the inside/outside test is meaningless.
- The arc radius has to be at least half the distance between its two nodes.
- A negative radius curves the arc the other way round.
- A very large radius (e.g. `10000.0`) approximates a straight line.
- All nodes of one mask need the same z coordinate.

A mask can be mirrored at a point and an axis, which is how symmetric setups with two targets are built:

```
m2 = rt.mirror_mask(
  template_name = "Uniformity", new_name = "UniformityMirror",
  mirror_point = SRT.PhysVector(-232.0, 260.0, 380.0),
  mirror_axis = SRT.PhysVector(1.0, 0.0, 0.0)
)
```

Mirrored masks do not follow later changes of the original, and they do not inherit its trajectory.

A mask can move on its own trajectory, which is how rotating or oscillating shutters are modelled:

```
m.get_trajectory().generate_line(381, SRT.PhysVector(0.0, 0.0, 380.0),
                                      SRT.PhysVector(20.0, 0.0, 380.0), 0.0)

# Pivot point of the rotation, if it is not where the mask sits on the trajectory
m.set_rotation_center(SRT.PhysVector(0.0, 1.0, 0.0))
```

The mask trajectory should have the same number of samples as the substrate trajectory. Check a hand-built mask for crossings of its own connections before running anything:

```
print("self intersections:", m.count_self_intersections())
```

`m.dump_gmsh_geo("mask.geo", offset_points = 1000, offset_lines = 1000)` writes the polygon as a GMSH geometry file for a quick look in a viewer. The offsets keep several masks from reusing the same point numbers in one file.

Running the raytracer

Once substrate, trajectory and histogram are defined, and optionally masks, the run is a loop:

```
while not rt.finished():
  rt.step()
```

`rt.get_time_step()` gives the current step (counting from 0) inside the loop — useful for conditionals and for filenames. The order of operations has to be kept, but anything can be put around the step, and the loop can live inside a function:

```
def run_scenario():
  while not rt.finished():
    # ... before the step
    rt.step()
    # ... after the step
  return s.dump_list(offset = SRT.PhysVector(0.0, 0.0, 0.0), normalize_to = 1.0)
```

That pattern is the basis for fitting, where the same function is called once per optimisation step.

Reading the result

The main result is the intensity on each substrate patch:

```
# Intensities per patch, as a list of (v0, v1, v2, v3, intensity)
values = s.dump_list(offset = SRT.PhysVector(0.0, 0.0, 0.0), normalize_to = 1.0)

# The same as a POS file for GMSH, and as CSV for everything else
s.dump_gmsh("profile.pos", offset = SRT.PhysVector(0.0, 0.0, 0.0), normalize_to = 1.0)
s.dump_csv("profile.csv", offset = SRT.PhysVector(0.0, 0.0, 0.0), normalize_to = 1.0)
```

Each list entry holds the four points spanning the patch and, as the last element, the intensity:

```
(SRT.PhysVector, SRT.PhysVector, SRT.PhysVector, SRT.PhysVector, float)
```

- `normalize_to = 0.0` gives absolute values, `normalize_to = 1.0` scales the maximum to one.
- `use_temp_values = True` returns only the last time step (a snapshot), `False` the accumulated sum.
- `offset` and `rotation_deg` place the part where it sits at a given step — useful for snapshots: `t.get_point_at(step)` and `t.get_rotation_deg_at(step)` give the values to pass in.

- `s.average_temp_angular_values()` and `s.average_full_angular_values()` average over the angular patches; the second one is permanent. Both only work if `n_phi` was a parameter of the geometry.
- All three dumping routines can be called inside the loop (after the step) or after it, depending on whether you want a snapshot or the accumulated result. For one file per step, build the filename from the step number: `"intensity_" + str(time_step) + ".pos"`.

A complete script

Circular trajectory over a disk, one circular mask, output as a POS file. Everything above appears once, in the order it has to be called.

```
import SRT as SRT
import SRTHelper as SRTHelper

rt = SRT.Raytracer()
s = rt.get_substrate()
t = rt.get_trajectory()

# Substrate
s.generate_geometry("disk", {"n_r": 10, "n_phi": 10, "radius": 100.0})
if s.is_concave():
  rt.precompute_substrate_intersections()
s.set_sputter_up()

# Trajectory: 11 samples on a 540 mm circle around (-232, 260, 385)
traj_radius = 540.0
t.generate_circle_arc(11, traj_radius, -19.0, 19.0, SRT.PhysVector(-232.0, 260.0, 385.0), -19.0)

# Histogram
batch = rt.add_histogram_batch("Main")
batch.add_histogram_from_file(
  "histogram.txt",
  SRT.PhysVector(0.0, 0.0, 100.0),
  SRT.PhysVector(100.0, 100.0, 100.0),
  20, 10,
  0.0,
  False
)

# Mask
m = rt.add_mask("Circle")
m.set_thickness(1.0)
n0 = m.add_node(SRT.PhysVector(100.0, 0.0, 100.0))
n1 = m.add_node(SRT.PhysVector(0.0, 100.0, 100.0))
n2 = m.add_node(SRT.PhysVector(-100.0, 0.0, 100.0))
n3 = m.add_node(SRT.PhysVector(0.0, -100.0, 100.0))
m.add_circle_arc(n0, n1, 100.0)
m.add_circle_arc(n1, n2, 100.0)
m.add_circle_arc(n2, n3, 100.0)
m.add_circle_arc(n3, n0, 100.0)

# Run
while not rt.finished():
  rt.step()

# Output
s.dump_gmsh("intensities.pos", offset = SRT.PhysVector(0.0, 0.0, 0.0), normalize_to = 1.0)
```

68 Fitting with SciPy

Basic idea and workflow

The raytracer can calculate the particle flux intensity reaching a substrate in a given scenario. In many cases this routine can be used for optimising coating processes by adjusting mask geometries or trajectories, for example.

This can be automated by using fitting routines. The algorithm is designed in a way that makes fitting relatively simple: a function has to be written that takes a vector of parameters, calls the raytracing routine and then calculates a number based on its output, which it then returns. Using the numerical optimisation methods provided by SciPy, the vector of parameters is iteratively adjusted to either minimise or maximise that number.

A simple example

`scipy.optimize.least_squares` minimises a given function. A minimal illustration:

```
# Import the least_squares routine from SciPy (make sure to have SciPy installed)
from scipy.optimize import least_squares

# Required for some methods in this example
import math

# Define a function that takes a parameter vector and returns a number
def fit_example(params):
  # Take the first value of the parameter vector
  x = params[0]

  # Some example function that has its minimum at x = 10
  return 1.0 - math.exp(-(x - 10)**2)

# Let's assume we don't know where the minimum is and guess the initial value 5 for x
# We also restrict x to the range between 0 and 20
final_result = least_squares(fit_example, [5], bounds = ([0], [20]))

# Print the result
print(final_result)
```

The result will look something like this:

```
     message: `gtol` termination condition is satisfied.
     success: True
      status: 1
         fun: [ 0.000e+00]
           x: [ 1.000e+01]
        cost: 0.0
         jac: [[ 1.490e-07]]
        grad: [ 0.000e+00]
  optimality: 0.0
 active_mask: [0]
        nfev: 2
        njev: 2
```

The important part is that `x` is 10, as expected. The approach extends to more than one variable trivially, simply by extending the vectors.

The goal function

Any scenario becomes fittable by wrapping it in a function that returns a single number. Uniformity, as the helper module defines it: zero for a perfectly uniform coating, one for no coating at all.

```
def uniformity_of(s):
  values = s.dump_list(offset = SRT.PhysVector(0.0, 0.0, 0.0), normalize_to = 1.0)
  return sum((1.0 - patch[-1]) ** 2 for patch in values) / len(values)
```

`SRTHelper.get_uniformity_of_substrate(s)` computes the square root of that, and `SRTHelper.get_manhattan_uniformity_of_substrate(s)` the linear version. Total and maximum intensity are available as `get_total_intensity_of_substrate(s)` and `get_max_intensity_of_substrate(s)`.

The linear version is the sum of the differences between 1 and the relative intensity of each patch, divided by the number of patches:

```
def uniformity(params):
  # ... (Do something with the params vector here)

  while not rt.finished():
    rt.step()

  intensity_list = s.dump_list(offset = SRT.PhysVector(0, 0, 0), normalize_to = 1.0)

  # Division by the number of patches is not strictly necessary, but normalises the
  # result to the range 0 ... 1, which helps the optimiser
  return (len(intensity_list) - sum([patch[-1] for patch in intensity_list])) / len(intensity_list)
```

Maximising the total intensity works the same way, by inverting the sum:

```
def maximum_intensity(params):
  # ... (Do something with the params vector here)

  while not rt.finished():
    rt.step()

  intensity_list = s.dump_list(offset = SRT.PhysVector(0, 0, 0), normalize_to = 0.0)

  return 1.0 / sum([patch[-1] for patch in intensity_list])
```

Any other target works the same way: a thickness profile that compensates the curvature of a lens, a ratio between centre and edge, a fit against a measured spectrum.

A comparison against a *radial* target profile is not built in — the helper that would do it is commented out in the source because it does not handle the substrate curvature yet. It is a few lines in the style above: loop over the radial patches, take the target value at that radius and sum the squared differences.

Reusing the raytracer

Inside the optimisation the raytracer is reset, not rebuilt. Creating a new instance per evaluation throws away the precomputed self-intersections and costs most of the run time for concave substrates.

```
def fit_function(params):
  s.reset_intensities()         # throw away the old intensities
```

```
    rt.reset_time_step()          # start the trajectory again
    m.reset_nodes_and_elements()  # empty the mask, keep its name

    # rebuild the mask from the parameters
    n0 = m.add_node(SRT.PhysVector(185.0, 170.0, 380.0))
    n1 = m.add_node(SRT.PhysVector(303.1, 184.9 + params[0] - 100.0, 380.0))  # free node
    m.add_circle_arc(n0, n1, -117.5)

    # mirrored masks have to be rebuilt as well
    rt.delete_mask(name = "UniformityMirror", has_to_exist = False)
    rt.mirror_mask(template_name = "Uniformity", new_name = "UniformityMirror",
                   mirror_point = SRT.PhysVector(-232.0, 260.0, 380.0),
                   mirror_axis = SRT.PhysVector(1.0, 0.0, 0.0))

    while not rt.finished():
      rt.step()

    return uniformity_of(s)
```

What can be reset and what cannot:

Call	Effect
`s.reset_intensities()`	All accumulated intensities back to zero
`rt.reset_time_step()`	Time step back to 0, trajectory starts again
`s.generate_geometry(...)`	New geometry and new ray intersections, intensities are kept
`m.reset_nodes_and_elements()`	Mask contents emptied, name and thickness kept
`t.generate_...()` / `t.load_from_list(...)`	Trajectory replaced
`rt.delete_mask(name, has_to_exist)`	Mask removed; `has_to_exist = False` makes it a no-op if it is gone

Mirrored masks keep their geometry when the template changes, so they have to be deleted and re-created. The same holds for their trajectory, which is not inherited.

Optimising the fitting process

Since the raytracing routine is called at every optimisation step, a few tricks are needed to get results with sufficient accuracy:

- **No convergence guarantee.** There is no guarantee that least squares converges, let alone onto the optimal parameter set. Evaluate a few sample values of the function first to get a feeling for where the optimum lies; the optimiser will then more reliably reach a local extremum.
- **Bounds.** Good bounds improve both stability and reliability. They also keep the geometry valid — see the next section.
- **Tolerances.** `xtol`, `ftol` and `diff_step` trade speed against accuracy. This matters more here than in most fits: the default function tolerance of `1e-8` is far below the noise of an accumulated intensity field.
- **Cheap first pass.** Limit the number of parameters, the number of substrate patches and the number of time steps. A simple planar substrate (disk or rectangle) already gives a rough estimate for the optimal trajectory or mask position, which can be refined afterwards with a detailed run.

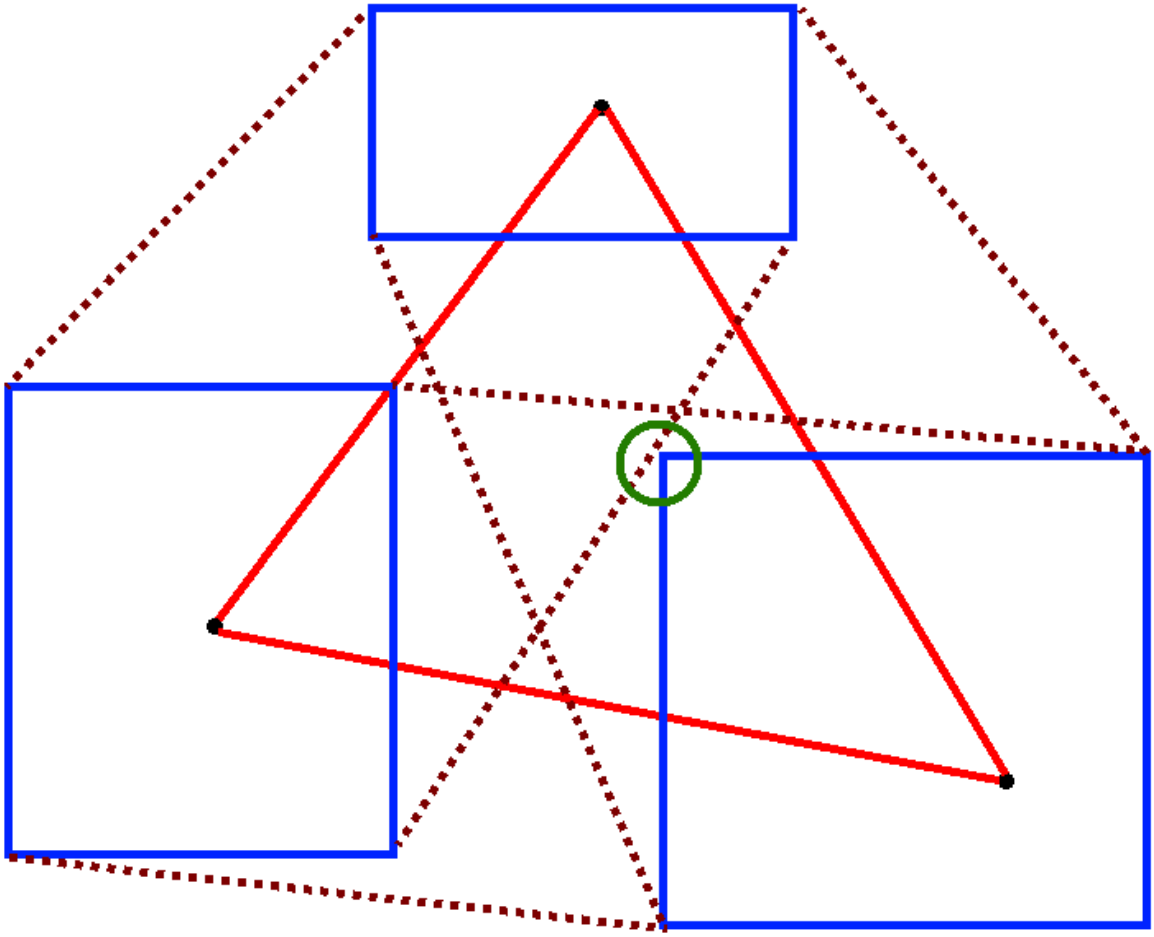
Fitting mask geometries

General idea

A typical use case is to optimise a mask geometry in order to maximise the uniformity of a coating. Technically this can be done by using all mask node positions as free parameters, but that leads to long computation times and potentially invalid mask geometries, with connections crossing each other.

So do not go into this without a plan. Ideally you consider as few mask nodes as possible and keep the others fixed — mainly those with the largest impact on the uniformity.

Furthermore, proper bounds have to be passed for the node positions, because two connections of the mask must never cross. This can be checked on paper, as in the following figure.



Triangular mask with three nodes (black) and their connections (red). The blue rectangles are the bounds chosen for each node. Drawing lines between the corners of the bounds of connected nodes (dark red, dashed) spans the area in which the connections can move; if a bound of a third node intersects that area, crossings become possible. The green circle marks a case that is only just safe.

It is your job to prevent these situations by choosing appropriate bounds. Trial and error is a completely valid method, but drawing the mask and the bounds on paper also does the job.

Keep in mind that using low radius values for the connections can produce geometrically nonsensical connections, preventing the raytracer from working. Rather use high radii or straight lines for a first guess.

Python script

Please read the previous sections before simply copying this script. It optimises a rectangular mask for uniformity, based on a given histogram (which you need to provide); see Usage for the individual lines and the scenario.

```
import path.to.SRT as SRT
from scipy.optimize import least_squares

rt = SRT.Raytracer()
s = rt.get_substrate()
t = rt.get_trajectory()

# Substrate
substrate_geometry = "disk"
substrate_params = {
  "n_r": 10,
  "n_phi": 10,
  "radius": 100.0
}
s.generate_geometry(substrate_geometry, substrate_params)
if s.is_concave():
  rt.precompute_substrate_intersections()
s.set_sputter_up()

# Trajectory
traj_samples = 11
traj_radius = 540.0
traj_center = SRT.PhysVector(100.0, 100.0, 100.0)
traj_angle_start = -10
traj_angle_end = 10
traj_substrate_rotation = 90
t.generate_circle_arc(traj_samples, traj_radius, traj_angle_start, traj_angle_end,
                      traj_center, traj_substrate_rotation)

# Histogram
hist_batch = rt.add_histogram_batch("Test")
hist_filename = "histogram.txt"
hist_min_coords = SRT.PhysVector(0.0, 0.0, 100.0)
hist_max_coords = SRT.PhysVector(100.0, 100.0, 100.0)
hist_size_x = 20
hist_size_y = 10
hist_d_phi = 0
hist_swap_x_y = False
hist_batch.add_histogram_from_file(hist_filename, hist_min_coords, hist_max_coords,
                                   hist_size_x, hist_size_y, hist_d_phi, hist_swap_x_y)

# Mask
mask = rt.add_mask("Rectangle")
mask.set_thickness(1.0)

def my_function(parameters):
  s.reset_intensities()
  rt.reset_time_step()

  # Reset the contents of the mask
  mask.reset_nodes_and_elements()

  # Add all nodes and connections again
  node_0 = mask.add_node(SRT.PhysVector(parameters[0], parameters[1], 100.0))
  node_1 = mask.add_node(SRT.PhysVector(parameters[2], parameters[3], 100.0))
  node_2 = mask.add_node(SRT.PhysVector(parameters[4], parameters[5], 100.0))
  node_3 = mask.add_node(SRT.PhysVector(parameters[6], parameters[7], 100.0))
```

```
  # High values to model straight lines (until they're implemented properly)
  r_0_1 = 10000.0
  r_1_2 = 10000.0
  r_2_3 = 10000.0
  r_3_0 = 10000.0

  mask.add_circle_arc(node_0, node_1, r_0_1)
  mask.add_circle_arc(node_1, node_2, r_1_2)
  mask.add_circle_arc(node_2, node_3, r_2_3)
  mask.add_circle_arc(node_3, node_0, r_3_0)

  while not rt.finished():
    rt.step()

  intensity_list = s.dump_list(offset = SRT.PhysVector(0, 0, 0), normalize_to = 1.0)
  result = (len(intensity_list) - sum([patch[-1] for patch in intensity_list])) /
len(intensity_list)

  # Give us some information about what's happening
  print(parameters, " -> ", result)

  return result

# Our starting values in pairs of xy-coordinates
fit_start_values = [100, 0, 0, 100, -100, 0, 0, -100]

# Minimum and maximum values, also in pairs of xy-coordinates
fit_min_values = [60, -40, -40, 60, -140, -40, -40, -140]
fit_max_values = [140, 40, 40, 140, -60, 40, 40, -60]

# Limit our required parameter accuracy to 1.0
parameter_tolerance = 1.0

# Limit our required function result accuracy to 0.001
function_tolerance = 0.001

# Only use steps of about 1% for the parameters
fit_steps = 0.01

# Start the fitting
final_result = least_squares(my_function, fit_start_values,
                             bounds = (fit_min_values, fit_max_values),
                             xtol = parameter_tolerance, ftol = function_tolerance,
                             diff_step = fit_steps)

# Print the final result
print(final_result)

# One more run with the optimum, for the output files
my_function(final_result.x)
```

What matters in practice:

- **Few parameters.** Every parameter costs a full raytracer run per iteration. Move one or two nodes, keep the rest fixed.
- **Bounds, and a drawing.** Every parameter box must be chosen so that no two connections of the mask can cross — a crossing mask produces nonsense, not an error.
- **Tolerances.** The default `ftol` of SciPy is far below the noise of an accumulated intensity field. `ftol` around `1e-3` and `xtol` around `0.1` are realistic.
- **Cheap first pass.** Fit on a flat disk with few patches and few time steps, then verify the result with the real geometry and a finer trajectory.

- **Large value ranges.** If one parameter is in millimetres and another one is a ratio, fit normalised values in `[0, 1]` and convert them back inside the function.

Questions

How can I apply tolerances to vastly different parameter values?

If you want to fit two parameters with entirely different value ranges, a single `xtol` or `ftol` value for both is difficult to choose. In that case use normalised values as function parameters (e.g. from 0 to 1) and calculate them back into their actual values inside the function.

69 Substrates

Explanation

All parametrisations either follow 2D Cartesian coordinates (x', y') or 2D polar coordinates (r, φ), depending on the type of substrate, and transform these into 3D Cartesian coordinates (x, y, z).

Each of the 2D coordinates is determined from the index of the respective substrate patch, transforming them into the 3D coordinates of the patch vertices. The point at $\underline{x} = 0$ determines the position of the substrate on the trajectory.

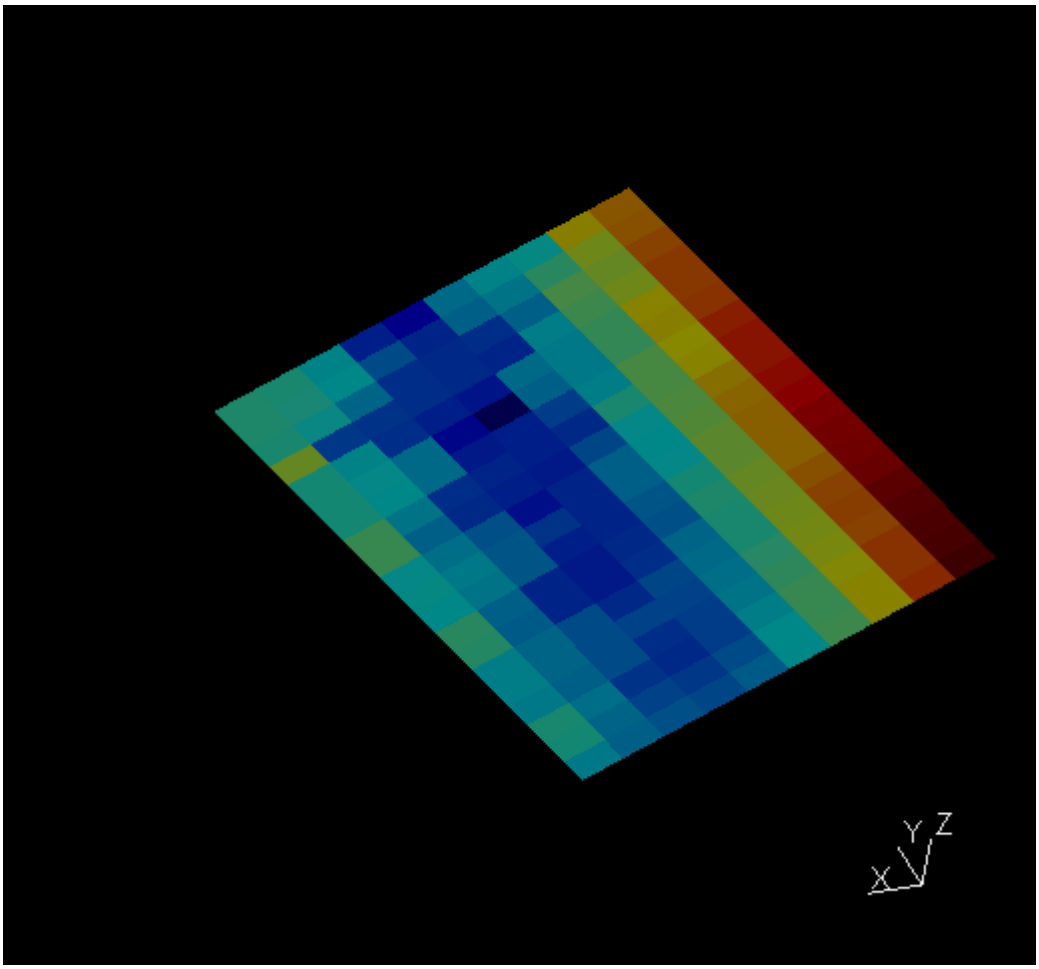
The following depictions are always for a sputter-up scenario.

Overview

Shape	Concave	Python name	Parameters
Disk	no	`disk`	`n_r`, `n_phi`, `radius`
Slanted rectangle	no	`slanted_rectangle`	`n_x`, `n_y`, `size_x`, `size_y`, `tilt_x`, `tilt_y`
Spherical lens	no	`spherical_lens`	`n_r`, `n_phi`, `r_curvature`, `r_base`
Aspherical lens	no	`aspherical_lens`	`n_r`, `n_phi`, `r_curvature`, `r_base`, `asph_k`, `asph_alpha_*`
Toric lens	no	`toric_lens`	`n_r`, `n_phi`, `r_curvature_x`, `r_curvature_y`, `height`
Inverted spherical lens	always	`inverted_spherical_lens`	`n_r`, `n_phi`, `r_curvature`, `r_base`
Bent rectangle	if `height > 0`	`bent_rectangle`	`n_x`, `n_y`, `size_x`, `size_y`, `height`
Parabolic lens	if `coefficient > 0`	`parabolic`	`radius`, `coefficient`

Convex substrate types

Slanted rectangle



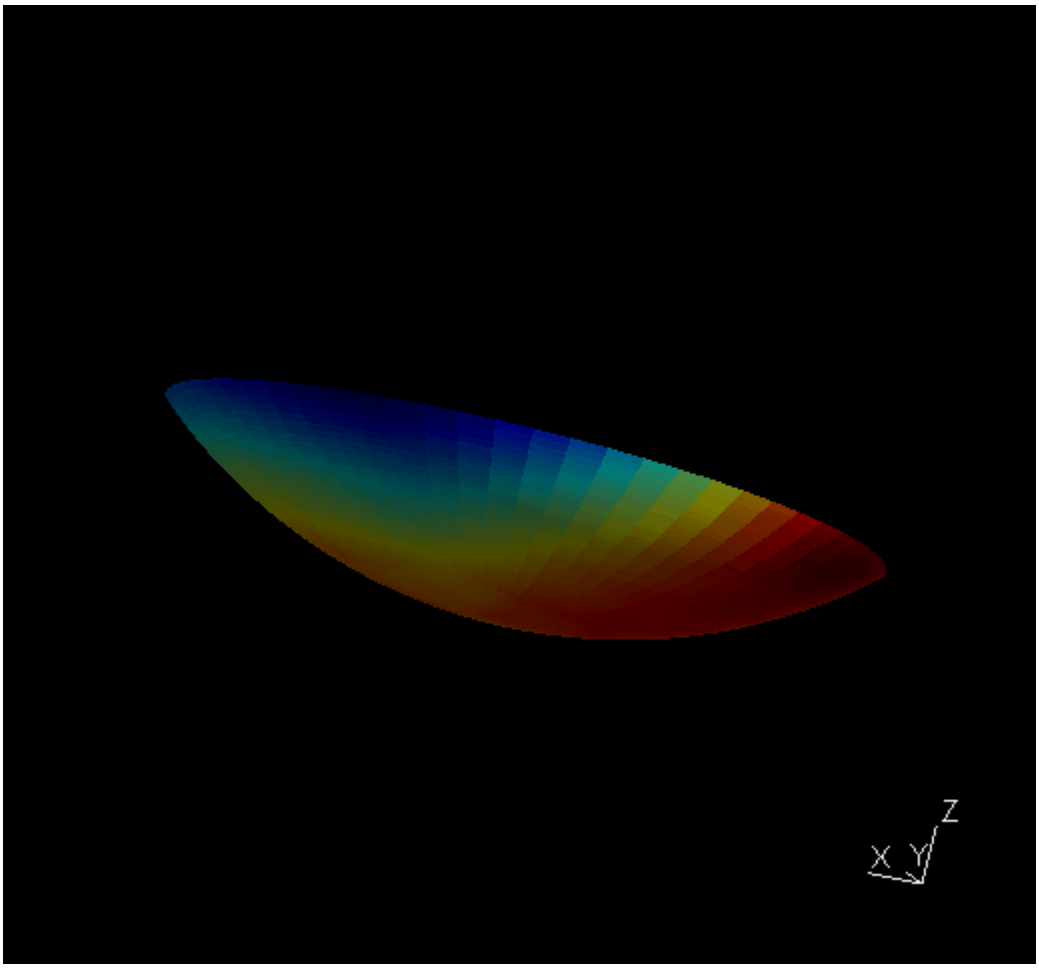
Slanted rectangle: sizes, patch counts and the tilt in x and y direction

`slanted_rectangle`

- `n_x`: number N_x of patches in x-direction
- `n_y`: number N_y of patches in y-direction
- `size_x`: size S_x of the rectangle in x-direction
- `size_y`: size S_y of the rectangle in y-direction
- `tilt_x`: total z-tilt Δ_x of the rectangle in x-direction, can be 0
- `tilt_y`: total z-tilt Δ_y of the rectangle in y-direction, can be 0

$$x(x', y') = -\frac{1}{2}S_x + \frac{S_x}{N_x}x'$$
$$y(x', y') = -\frac{1}{2}S_y + \frac{S_y}{N_y}y'$$
$$z(x', y') = \frac{\Delta_x}{S_x}x + \frac{\Delta_y}{S_y}y$$

Spherical lens



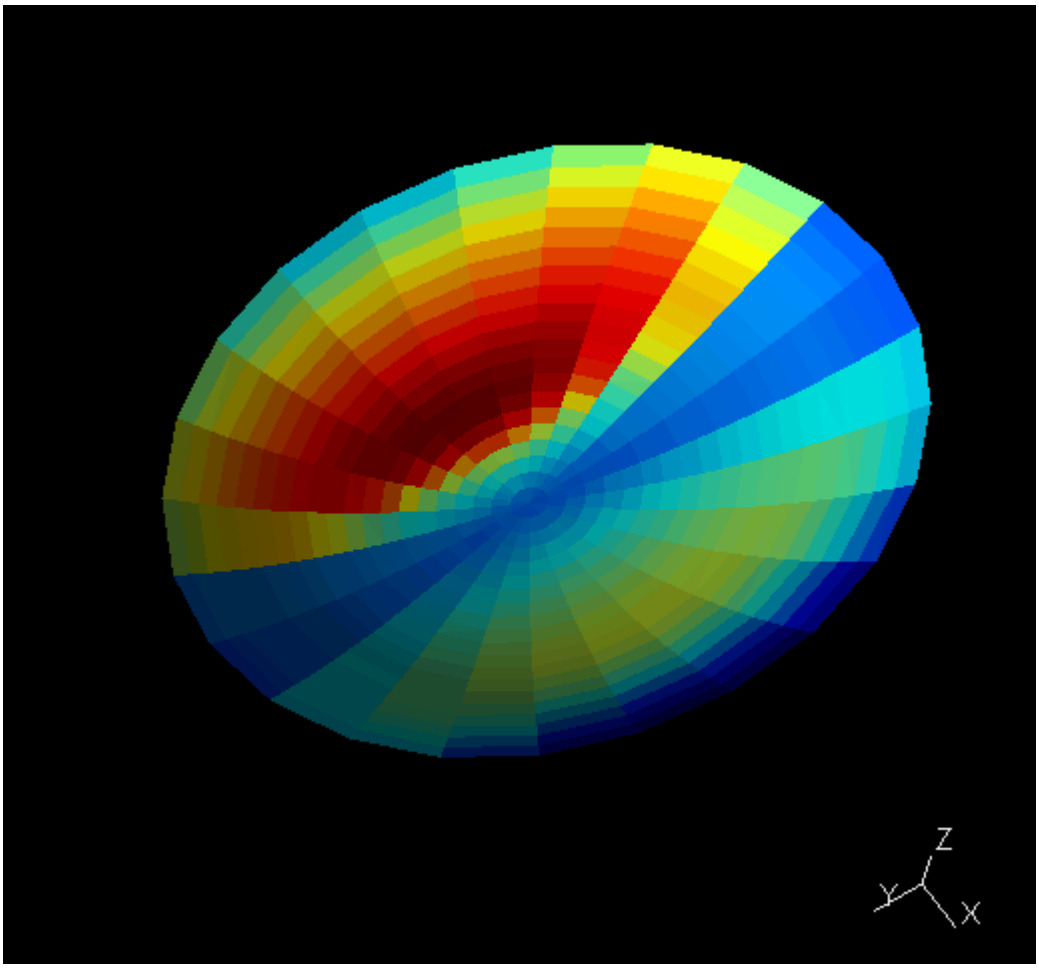
Spherical lens: curvature radius R_c and base radius

`spherical_lens`

- `n_r`: number of patches in radial direction
- `n_phi`: number of patches in angular direction
- `r_curvature`: curvature radius R_c of the lens (if it would be a full sphere)
- `r_base`: radius of the lens at its base

$$
\begin{aligned}
x(r,\varphi) &= r\cos(\varphi) \\
y(r,\varphi) &= r\sin(\varphi) \\
z(r,\varphi) &= \sqrt{R_\mathrm{c}^2 - r^2} - R_\mathrm{c}
\end{aligned}
$$

Toric lens



Toric lens: two different curvature radii, one per axis

`toric_lens`

- `n_r`: number of patches in radial direction
- `n_phi`: number of patches in angular direction
- `r_curvature_x`: curvature radius $R_{\mathrm{c},x}$ of the lens in x-direction (if it would be a full torus)
- `r_curvature_y`: curvature radius $R_{\mathrm{c},y}$ of the lens in y-direction (if it would be a full torus)
- `height`: height of the lens (between base and tip)

This parametrisation differs from the others, since the curvature radii are used ordered by their size, defined as $R_{\mathrm{c,min}}$ and $R_{\mathrm{c,max}}$.

Some additional functions used are:

$$z_0(r) = R_{\mathrm{c,max}} - R_{\mathrm{c,min}} + z_{\mathrm{base}}(r)$$

Also, the z position at the base of the lens is used:

$$z_{\mathrm{base}}(r) = \sqrt{R_{\mathrm{c,min}}^2 - r^2}$$

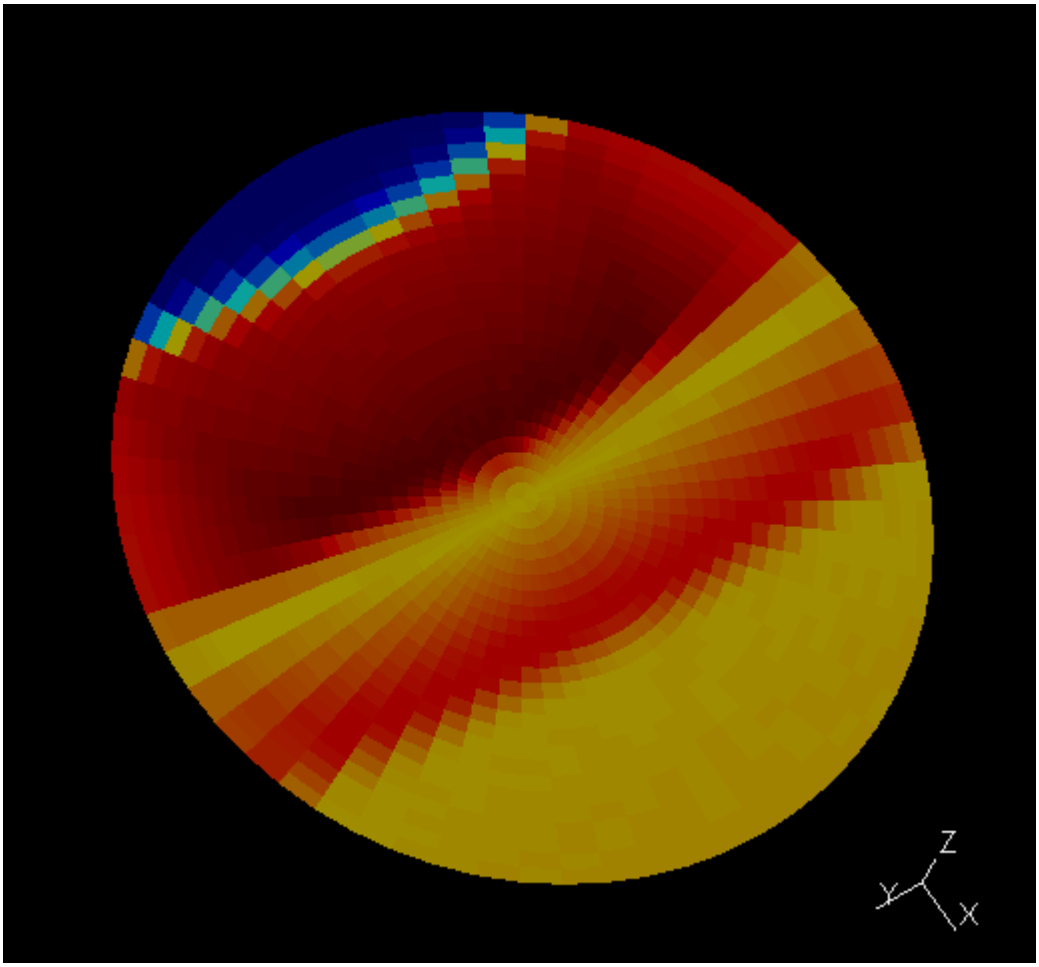
Finally, the semiaxes of the elliptic cuts at given radii are (in case of the y curvature radius being the larger one):

$$S_x(r) = \sqrt{R_{\mathrm{c},x}^2 - z_{\mathrm{base}}(r)}$$
$$S_y(r) = \sqrt{R_{\mathrm{c},y}^2 - z_0(r)}$$

If on the other hand the x-axis has the larger curvature radius, the roles of x and y have to be switched. Put together we obtain:

$$x(r,\varphi) = S_x(r)\cos(\varphi)$$
$$y(r,\varphi) = S_y(r)\sin(\varphi)$$
$$z(r,\varphi) = z_0(r) - R_{\mathrm{c,max}}$$

Disk



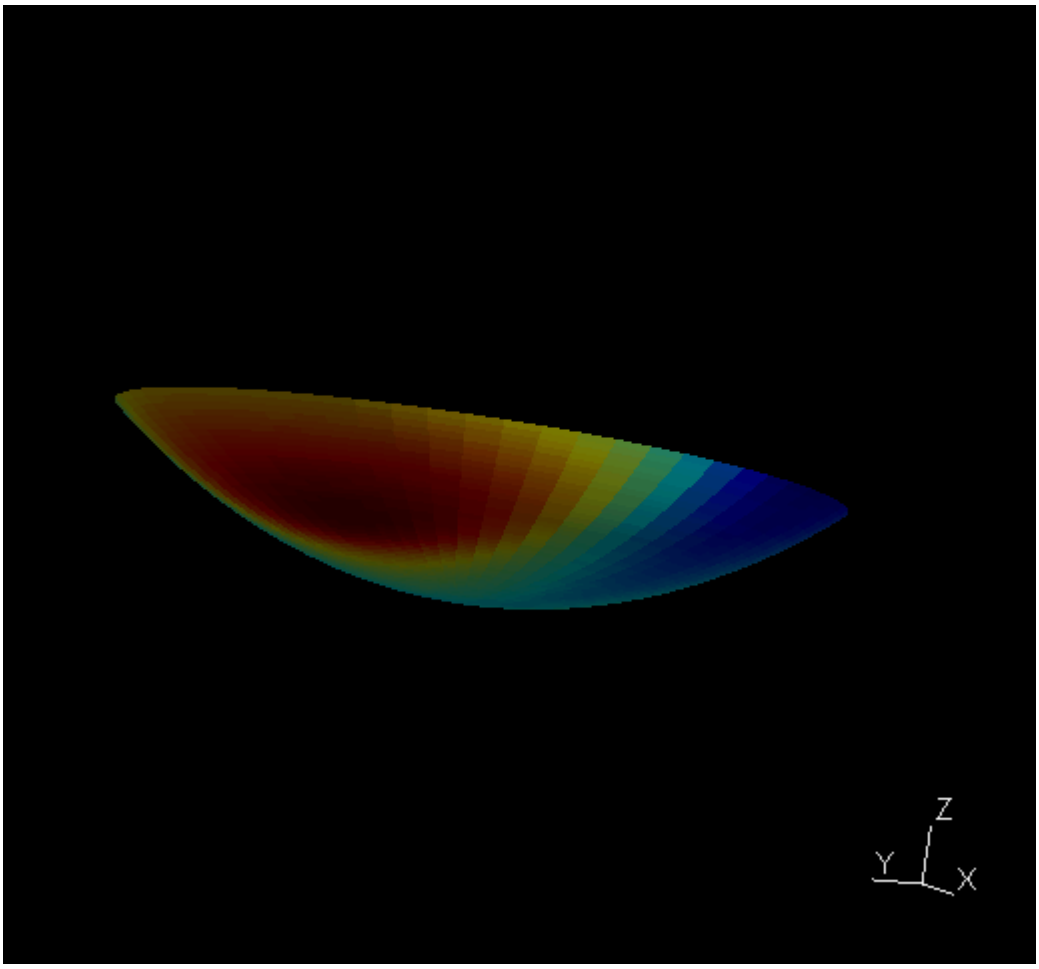
Disk: radius and the number of radial and angular patches

`disk`

- `n_r`: number of patches in radial direction
- `n_phi`: number of patches in angular direction
- `radius`: radius of the disk

$$x(r,\varphi) = r\cos(\varphi)$$
$$y(r,\varphi) = r\sin(\varphi)$$
$$z(r,\varphi) = 0$$

Aspherical lens



Aspherical lens: spherical base shape plus a conic factor and alpha coefficients

`aspherical_lens`

- `n_r`: number of patches in radial direction
- `n_phi`: number of patches in angular direction
- `r_curvature`: curvature radius R_c of the lens (if it would be a full sphere)
- `r_base`: radius of the lens at its base
- `asph_k`: conic shape factor κ

Additionally, aspherical lenses require an arbitrary number of alpha coefficients α_j, named `asph_alpha_4`, `asph_alpha_6`, `asph_alpha_8` and so on. The number in the parameter name describes the exponent of the coefficient (has to be even and at least 4) and its value is the coefficient itself. Any unspecified coefficient is set to 0 automatically.

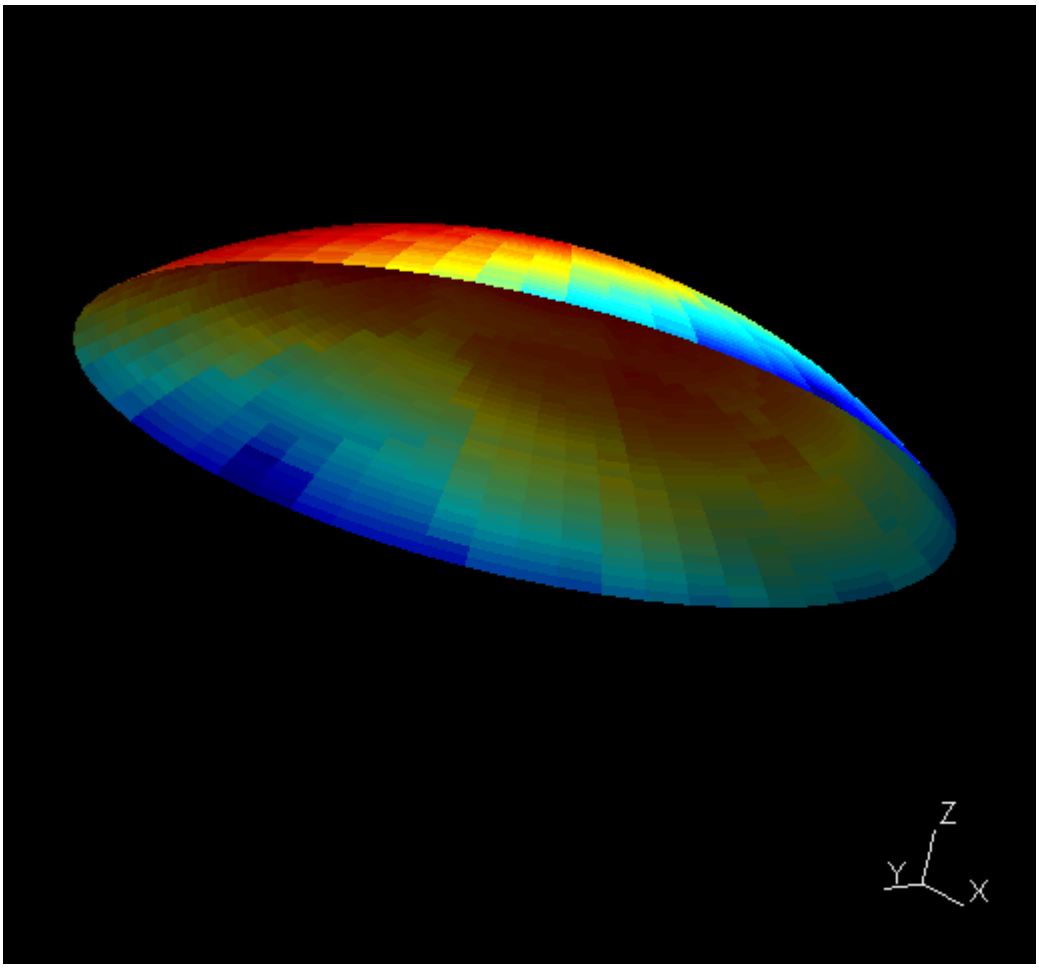
The function resulting from the coefficients defines the height of the surface:

$$z_{\mathrm{asph}}(r) = \frac{r^2}{R_{\mathrm{c}}\left(1+\sqrt{1-(1+\kappa)\frac{r^2}{R_{\mathrm{c}}^2}}\right)} + \sum_{j=2} \alpha_{2j} r^{2j}$$

$$x(r,\varphi) = r\cos(\varphi)$$
$$y(r,\varphi) = r\sin(\varphi)$$
$$z(r,\varphi) = z_{\mathrm{asph}}(r)$$

Potentially concave substrate types

Inverted spherical lens



Inverted spherical lens: the same shape as the spherical lens, with the curvature turned inwards

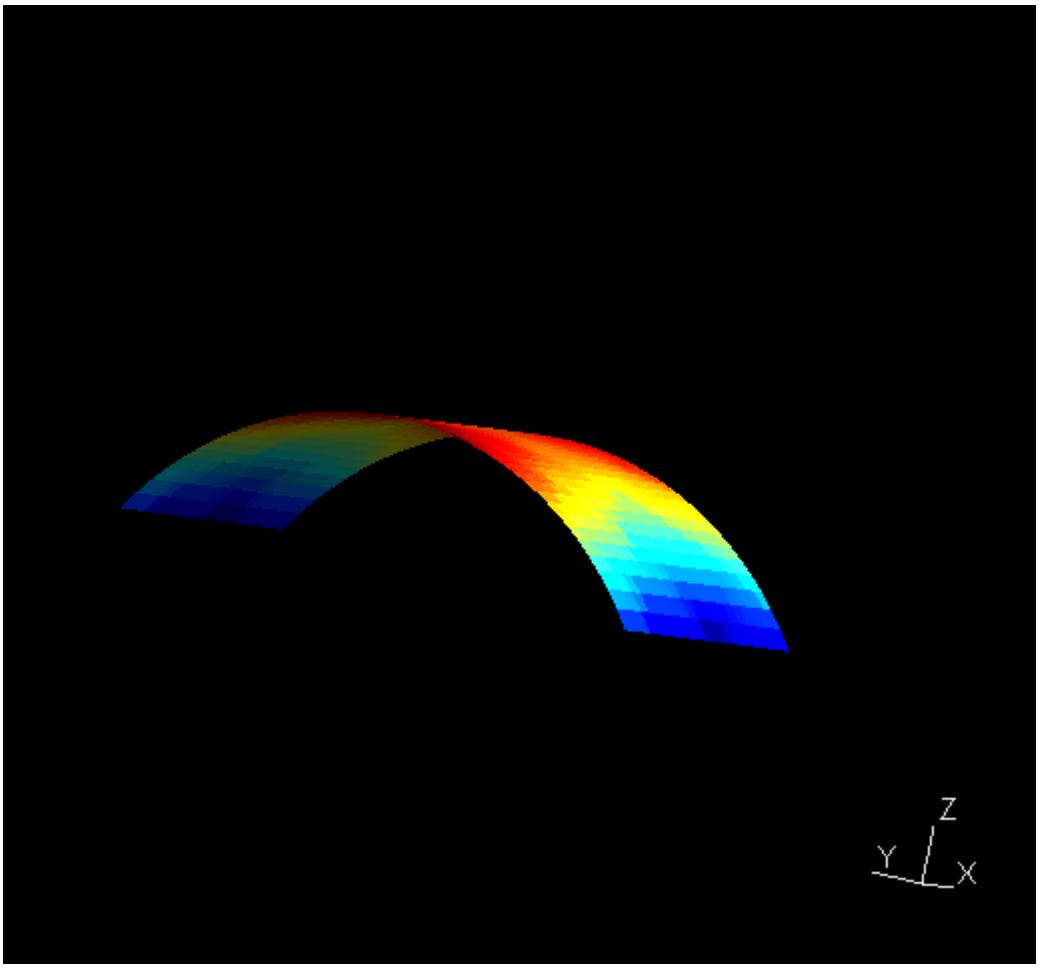
`inverted_spherical_lens`

- `n_r`: number of patches in radial direction
- `n_phi`: number of patches in angular direction
- `r_curvature`: curvature radius R_{c} of the lens (if it would be a full sphere)
- `r_base`: radius of the lens at its base

Inverted spherical lenses are always concave.

$$
\begin{aligned}
x(r, \varphi) &= r\cos(\varphi) \\
y(r, \varphi) &= r\sin(\varphi) \\
z(r, \varphi) &= -\sqrt{R_\mathrm{c}^2 - r^2} + R_\mathrm{c}
\end{aligned}
$$

Bent rectangle



Bent rectangle: sizes, patch counts and the height difference that bends it

`bent_rectangle`

- `n_x`: number N_x of patches in x-direction
- `n_y`: number N_y of patches in y-direction
- `size_x`: size S_x of the rectangle in x-direction
- `size_y`: size S_y of the rectangle in y-direction
- `height`: height difference h between base and tip, can be negative

Bent rectangles are only concave if `height` is set to a value greater than 0.

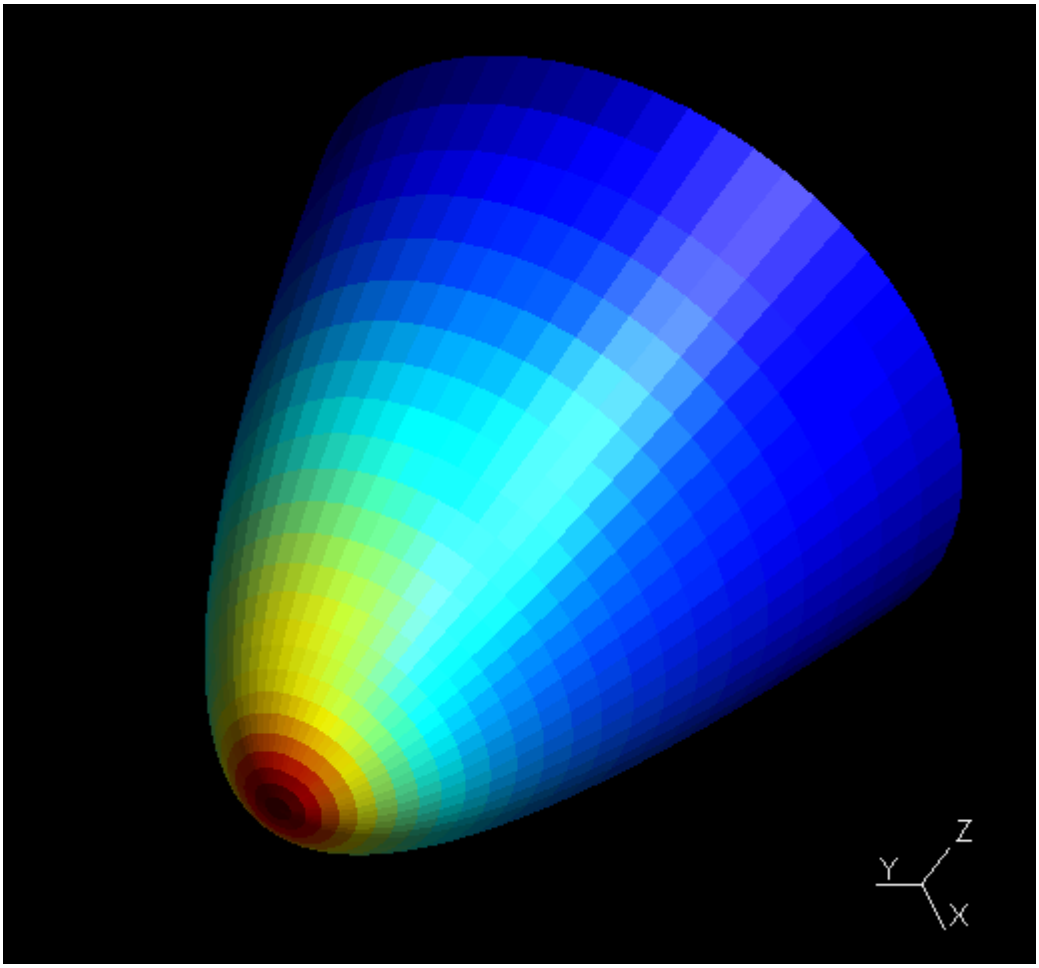
The curvature radius of the rectangle is obtained using:

$$
R_\mathrm{c} = \frac{1}{2h}\left(h^2 + \frac{S_y^4}{4}\right)
$$

The parametrisation is then:

$$x(x', y') = -\frac{1}{2}S_x + \frac{S_x}{N_x}x'$$
$$y(x', y') = -\frac{1}{2}S_y + \frac{S_y}{N_y}y'$$
$$z(x', y') = R_\mathrm{c} - \frac{h}{|h|}\sqrt{R_\mathrm{c}^2 - y^2}$$

Parabolic lens



Parabolic lens: base radius and the coefficient of the parabola

`parabolic`

- `radius`: radius at the base of the lens
- `coefficient`: coefficient α of the parabolic shape function

Parabolic lenses are only concave if `coefficient` is set to a value greater than 0.

$$x(r, \varphi) = r\cos(\varphi)$$
$$y(r, \varphi) = r\sin(\varphi)$$
$$z(r, \varphi) = \alpha r^2$$

Custom substrates

An entirely custom geometry is defined as a list of triangles:

```
list_of_triangles = [
```

```
  (SRT.PhysVector(0, 0, 0), SRT.PhysVector(-2, -2, 2), SRT.PhysVector(2, -2, 2)),
  (SRT.PhysVector(0, 0, 0), SRT.PhysVector(2, -2, 2), SRT.PhysVector(2, 2, 2))
]

s.generate_geometry_from_mesh_triangles(list_of_triangles, invert_normal = False)
```

`invert_normal` flips all triangle normals in case they are not oriented towards the histogram. Individual normals cannot be flipped, so define the mesh consistently.

Larger meshes in the GMSH `.msh` format can be loaded through the helper module:

```
import SRTHelper as SRTHelper

list_of_triangles = SRTHelper.load_mesh_from_file("custom.msh")

s.generate_geometry_from_mesh_triangles(list_of_triangles, invert_normal = False)
```

The mesh loader needs the Python package `gmsh`. The coordinate system of the mesh is taken as it is, so the coordinate origin of the mesh is fixed to the current trajectory point.

70 Helper module

Main usage

`SRTHelper` contains the routines that would otherwise be repeated in every script. The helper library is installed into `build/libs` by the build, and it is imported like the raytracer itself:

```
# Installed tree, library on PYTHONPATH
import SRTHelper as SRTHelper

# Working directly in the build directory
import build.libs.SRTHelper as SRTHelper
```

Functions

Function	Purpose
`create_circular_trajectory_from_rpm(...)`	Circle from turntable and carrier RPM, time resolution in seconds
`create_linear_trajectory_from_speed(...)`	Straight pass from speed, time resolution in seconds
`get_uniformity_of_substrate(s)`	RMS deviation from a uniform coating
`get_manhattan_uniformity_of_substrate(s)`	Linear deviation from a uniform coating
`get_total_intensity_of_substrate(s)`	Sum over all patches
`get_max_intensity_of_substrate(s)`	Maximum patch intensity
`load_mesh_from_file(filename)`	Triangle list from a GMSH mesh, for custom substrates

The uniformity and intensity functions read the accumulated intensities of a substrate, so they are meant to be called after the raytracer has finished — see Fitting with SciPy for how they are used as goal functions.

Loading custom geometries

```
list_of_triangles = SRTHelper.load_mesh_from_file("custom.msh")

s.generate_geometry_from_mesh_triangles(list_of_triangles, invert_normal = False)
```

Details on the mesh orientation are on the Substrates page. The loader needs the Python package `gmsh`.

Trajectories from machine parameters

Trajectories can be defined by more practical parameters than a list of points — the velocity of a carrier or the RPM of a turntable, for example. The helper functions produce the list that `t.load_from_list(...)` expects.

It is important to set the time resolution as small as the problem needs, but not smaller: the total number of samples grows with it, and the run time with the number of samples. For testing, the number of samples of a generated list can be obtained with `len(traj_list)`. In general, do multiple test runs

and either decrease the time resolution until the results stop changing significantly or increase it until they change too much.

Circular trajectory

```
# Center of the circle
traj_center = SRT.PhysVector(0.0, 0.0, 100.0)

# Start and final angle in degrees
traj_angle_start = -30.0
traj_angle_final = 30.0

# Radius of the trajectory
traj_radius = 200.0

# Initial orientation of the substrate (here equal to the start angle)
traj_substrate_rotation = traj_angle_start

# RPM rate of the turntable
traj_rpm = 10.0

# RPM rate of the substrate relative to its carrier
traj_substrate_rpm = 0.0

# Time resolution in seconds
traj_time_resolution = 0.1

# Generate the trajectory
traj_list = SRTHelper.create_circular_trajectory_from_rpm(
  traj_time_resolution, traj_center, traj_radius, traj_rpm,
  traj_angle_start, traj_angle_final, traj_substrate_rpm,
  traj_substrate_rotation, add_alignment_rotation = True
)

# Load it into the trajectory
t.load_from_list(traj_list)
```

Linear trajectory

A straight pass along the y axis, driven by the speed of the carrier:

```
# Start and end point of the line
traj_point_start = SRT.PhysVector(0.0, -50.0, 100.0)
traj_point_end = SRT.PhysVector(540.0, 50.0, 100.0)

# Initial rotation of the substrate
traj_substrate_rotation = 0.0

# Speed of the substrate motion per second, in the unit used for all other coordinates
traj_speed = 1.0

# Substrate rotation RPM rate relative to its carrier
traj_substrate_rpm = 0.0

# Time resolution of the trajectory in seconds
traj_time_resolution = 0.5

# Generate the trajectory
traj_list = SRTHelper.create_linear_trajectory_from_speed(
  traj_time_resolution, traj_speed, traj_point_start, traj_point_end,
  traj_substrate_rpm, traj_substrate_rotation
)
```

```
# Load it into the trajectory
t.load_from_list(traj_list)
```

71 Algorithm

This page describes what the raytracer computes and why — the physics and the numerical idea, not the code. It is meant for judging whether a result can be trusted and for understanding where the run time goes.

The raytracing algorithm

The basic idea is that each patch of the substrate tries to get as much data from each histogram as possible. The approach goes back to the deposition model developed by Dr. Andreas Pflug and was extended later.

The program is split into time steps that discretise the trajectory of the substrate. In each time step the substrate is placed at the respective point of the trajectory and optionally rotated to keep its orientation towards a fixed point. Then each patch of the substrate geometry is iterated over by casting a number of rays between the centre of the patch and each histogram source, with each angular combination of the respective histogram defining the direction of one ray.

Any ray with the wrong direction, or hitting an obstacle — a concave part of the substrate or a mask — is discarded, leaving only rays that are physically able to reach the histogram.

For each remaining ray the intersection point with the histogram is obtained and the flux at that point is interpolated bilinearly from the four neighbouring histogram samples. All fluxes reaching the patch are summed and weighted by their normalised scalar product with the patch normal, since an oblique ray transfers geometrically less material to a patch than one hitting it perpendicularly.

Over the whole trajectory the intensities of each patch are summed up again. The final values can then be normalised or averaged, depending on what the evaluation needs.

Substrate generation

The predefined shapes are assembled from connected quadrilateral patches. Each patch is defined by four vertices produced from a 2D parametrisation of the 3D geometry — a sphere, for example, is parametrised in spherical coordinates at a constant radius. The exact formulas are on the Substrates page.

The geometry is sampled over a number of patches in two dimensions: a rectangle by a number of patches in x- and y-direction, a disk by its radial and angular direction.

From the vertices, the centre of each patch is the mean of its four corners. The normal at the centre follows from the cross product of the two diagonals. Two normals are always possible, so a sign convention decides which one points outwards — for that, the vertices have to be ordered consistently in every patch.

Mask intersections

A mask is a series of vertices connected by circular arcs — straight lines are approximated by a very large radius — plus a finite thickness in sputtering direction. Two independent tests decide whether a ray is blocked.

First the mask base is tested. The question whether a ray crosses the base reduces to whether the point of the ray at the z coordinate of the mask lies inside the 2D mask shape. This is answered by counting intersections: a point known to be outside the mask is connected to the ray point by a straight line, and this line is tested against every circular arc of the mask with a line-circle intersection. Whenever a circle is hit, the angle of incidence decides whether the intersection lies on the actual arc or only on the full circle, in which case it is discarded. An even number of true intersections means the ray point is outside the mask, an odd number means it is inside and the ray is blocked.

If the ray survives the base test, the mask walls are tested: the arcs extruded by the mask thickness act as cylinders, and a ray-wall intersection is a cylinder-line intersection with the same additional angle check. The top face of the mask needs no test at all — a straight ray that crossed neither the base nor a wall cannot hit the top.

Self-intersections in concave substrates

In a concave substrate a ray cast between the substrate and a histogram can hit the substrate again at a different point. Ignoring this would produce unphysical results: flux would appear at a patch that the geometry shadows.

Because the algorithm is structured as it is, there is only a finite, deterministic number of possible rays between the substrate and all histograms — the patch centres and the angular combinations of the histograms are fixed. This allows all of these intersections to be computed before the run starts, which costs some startup time and saves a large amount of run time.

The procedure: collect all angles from all histograms, cast a ray from every substrate patch along each of these directions into infinity, and test that ray against every patch of the substrate. Each patch is split into triangles for a line-triangle test. The closest intersection for each patch and angle combination is stored.

During the run, each ray looks up this table for its originating patch; if an intersection lies between the histogram and the substrate, the ray is discarded. This is why `rt.precompute_substrate_intersections()` has to be called for concave geometries — without it the table is missing and shadowing is ignored.

Parallelisation

The self-intersection table and the raytracing loop are parallelised over substrate patches, which are independent of each other. The results do not depend on the number of threads: the same scenario produces the same numbers, only the run time changes.

PALADIN

72 Installation

PALADIN is distributed by Fraunhofer IST. A site that received the source builds it on the machine or cluster it will run on; there is no package manager entry and no container image.

Requirements

Component	Version / note
Linux	64-bit, x86-64
C++ compiler	C++14, used through `mpic++`
MPI	OpenMPI 5 by default, any MPI with `mpic++`, `mpiexec` and MPMD support otherwise
GNU Make	for the build targets below
Python	3.x with development headers (`Python.h`) and a matching `python3-config`

MPMD support matters: a run starts one MPI rank as the Python server process and the remaining ranks as `start_client`. Slurm handles that natively; a bare `mpiexec` needs the colon syntax shown on the scenario page.

Note: the Python that runs the scenario is not the Python that was linked at build time. The extension is loaded at run time from `libs/`, so the interpreter only has to match the ABI of the headers used for the build.

Build

All settings are Make variables, so nothing has to be edited for a normal build:

```
make MPIPATH=/opt/openmpi-5 PYTHON_CONFIG_PATH=/usr/bin/python3-config -j
```

Variable	Default	Meaning
`MPIPATH`	`/opt/openmpi-5`	Root of the MPI installation; `mpic++` is taken from `$MPIPATH/bin`
`PYTHON_CONFIG_PATH`	`/usr/bin/python3.11-config`	Script that supplies the Python include flags
`DESTINATION`	`.`	Install root. With a path, the tree is created there and the Python part is copied into it
`DEBUG`	unset	`DEBUG=1` builds with `-O0 -g` instead of `-O3`

Targets:

Target	Result
`make`	Libraries, `start_client`, `signal_helper`, and the Python part in the install root

Target	Result
`make full`	The same plus the example cases
`make clean`	Removes objects, libraries and executables
`make clean_examples`	Removes the example directory of a `full` build
`make clean_full`	Both of the above

What lands in the install root:

Path	Content
`libs/pyPALADIN.so`	The Python extension module: all classes, all constants
`start_client`	The MPI client binary that advances the particles
`signal_helper`	Sends control commands (pause, unpause, dump, shutdown, kill) to a running job
`python_scripts/`	`pyPALADINServer.py` (the server class), `constants.py` (units and physical constants), `console_operations.py`
`submit_py`	Slurm submit wrapper
`examples/`	The four example cases, only after `make full`

Environment

Three variables have to be set in the shell, or inside the job script. `PALADINPATH` is the install root, the directory that contains `libs/` and `python_scripts/`:

```
export PALADINPATH=/path/to/paladin
export PYTHONPATH=$PALADINPATH/libs:$PALADINPATH/python_scripts:$PYTHONPATH
export LD_LIBRARY_PATH=$PALADINPATH/libs:$LD_LIBRARY_PATH
```

`PYTHONPATH` makes `import pyPALADIN` and `import pyPALADINServer` work from anywhere; `LD_LIBRARY_PATH` is needed by `start_client` for the shared libraries in `libs/`. On a cluster, `submit_py` sets all three itself for the batch job from `PALADINPATH` and `PYTHONPATH`, so the interactive shell does not have to be prepared.

Warning: `submit_py` has the MPI root compiled in as `/opt/openmpi-5` and starts `mpiexec` from there. If your MPI lives elsewhere, edit the `MPIPATH` line at the top of the script — the environment variable does not override it.

Verify the installation

```
python3 -c "import pyPALADIN, pyPALADINServer; print('import ok')"
```

If the import fails with a message about a missing shared object, `LD_LIBRARY_PATH` is not set. If it fails with an undefined symbol from MPI, the extension was built against a different MPI than the one on the path at run time.

A full check is one of the example cases: run it, and look at the header of the run output — it prints the PALADIN version and the run name, and the run directory appears next to the script.

Updates

There is no in-place update mechanism. A new version replaces `libs/`, `start_client`, `python_scripts/` and `submit_py`; scenarios that only use documented parameters keep working, but check the parameters page for renamed or added keys before reusing an old script.

73 Building a scenario

A PALADIN case is a Python script. The script describes the geometry, the fields, the species and the output, starts the server object and lets it run. Everything else — moving the particles, charging them, resolving wall collisions — happens in the C++ part, which is distributed over the MPI client ranks.

A script therefore has three parts, always in the same order:

1. a `Parameters` object with everything that is fixed for the run,
2. a `pyPALADINServer` object built from it,
3. the particle initialisation and `run()`.

The smallest complete case, a capacitor with a constant field and ten dust particles:

```
from pyPALADINServer import *

pars = Parameters(
    NAME = "Capacitor",
    DT = 0.001*SECOND,
    TSIM = 2.5*SECOND,
    MESH_FILE = "capacitor.msh",

    UNIT = METER,

    BB = (-1.1, -0.6, -0.05, 1.1, 0.6, 0.0),
    BB_OPTIONS = (BB_SOLID, BB_SOLID, BB_PERIODIC),
    MESH_DIVISION = (10, 5, 1),

    BORDER_TYPES = {
        "Inside":  pyPALADIN.Border(icodec = 1, type = WALL_TYPE_SPECULAR),
        "Outside": pyPALADIN.Border(icodec = 2, type = WALL_TYPE_SPECULAR)
    },

    GRAVITY = pyPALADIN.Field(vector = (0, -GRAVITATIONAL_ACCELERATION, 0)),
    ELECTRIC = pyPALADIN.Field(vector = (1*VOLT*PER(METER), 0, 0)),

    PLASMA_SPECIES_LIST = {},

    SPECIES_LIST = {
        "Dust": pyPALADIN.Species(
            size_distribution_type = DISTRIBUTION_CONSTANT,
            size_distribution_param_1 = 1*MICRO*METER,
            mass_density = 1000*KILOGRAM*PER(CUBIC(METER))
        )
    },

    DUMPER_INSTANCES = {
        "Tracer": pyPALADIN.DumperInstance(
            file_template = "tracer",
            total_number = round(TSIM/DT),
            mode = DUMP_ALL_IN_ONE,
            values = [DUMP_SCALAR_TIME],
            value_units = [SECOND],
            format = FORMAT_GMSH_POS,
            additional_dump_at_start = True
        )
    },

    SEED_FOR_FIRST_CLIENT = 1234
)

server = pyPALADINServer(pars)
```

```
server.spawn_particles_on_surface(10, border_name = "Inside", species_name = "Dust")
server.run()
```

The `Parameters` object is checked before anything is started: a wrong type, a missing entry or a time step larger than the simulated time stops the script with a message instead of a crash in the middle of a run.

Geometry and mesh

Key	Meaning
`MESH_FILE`	Gmsh `.msh` file with the volume mesh and physical groups
`UNIT`	Length unit of the mesh file, in metres. A mesh in millimetres uses `MILLI*METER`

The physical groups of the mesh are what connects the geometry to the physics: each of them is addressed by its `icodec`, the number given in the mesh. Which wall is the target, which is the substrate, which one is an inlet is decided in the script, not in the mesh file.

The mesh has to resolve the geometry, not the particle. Particles are points with a radius; all fields are interpolated from the cell centres of the mesh, so the cell size sets the resolution of the field a particle sees.

Simulation volume

The volume the particles live in is described independently of the mesh:

Key	Meaning
`BB`	Bounding box as `(x_min, y_min, z_min, x_max, y_max, z_max)` in `UNIT`
`BB_OPTIONS`	One entry per direction: `BB_SOLID`, `BB_PERIODIC`, `BB_OUT_MIN`, `BB_OUT_MAX` or `BB_OUT`
`MESH_DIVISION`	Resolution of the internal element lookup grid, as `(nx, ny, nz)`

`BB_SOLID` closes the box with a wall, `BB_PERIODIC` wraps it, and the `BB_OUT*` variants let a particle leave.

`MESH_DIVISION` is not about the parallelisation: it subdivides the bounding box into a lookup grid in which every mesh triangle is registered, so that a wall collision only tests the triangles of the neighbourhood instead of all of them. The product `nx · ny · nz` is the number of grid cells and therefore the memory the lookup costs; a grid that is too coarse leaves many triangles per cell and costs collision tests, a grid that is too fine costs memory for the cells. The tube example uses `(100, 100, 100)` for a mesh that covers the whole tube.

The clients split the particles, not the volume: each client owns a share of the particles and carries the whole geometry, so the number of clients is a free choice.

Walls

`BORDER_TYPES` is a dictionary of named borders. Each entry is a `Border` with the `icodec` of a physical group from the mesh:

Field	Meaning
`icodec`	Physical group number in the mesh
`type`	`WALL_TYPE_SPECULAR`, `WALL_TYPE_DIFFUSE` or `WALL_TYPE_MEMBRANE`
`elasticity`	Restitution of the impact, 1.0 = elastic, 0.0 = fully absorbing in the normal direction
`destruction`	The particle is removed when it hits this border
`tracking`	The border records impacts and origins, which is what a backtrace dump needs
`ejection_speed`	Speed normal to the surface that particles get when they are spawned on this border

The wall types differ in what happens to the tangential motion: a specular wall mirrors the particle, a diffuse wall re-emits it with a randomised direction at unchanged speed, and a membrane leaves the velocity untouched so the particle passes through the surface. What adheres and what bounces is decided by the particle model — the adhesion section has the details.

Note: `WALL_TYPE_SPECULAR` is the name in the Python interface. The corresponding type in the C++ code is called `WALL_TYPE_WALL`; both mean the same thing.

Fields

A field is always a `pyPALADIN.Field`. It can be a constant, a file, or a file with a constant fallback:

Argument	Meaning
`scalar`	Constant scalar value
`vector`	Constant vector, e.g. `(0, 0, -GRAVITATIONAL_ACCELERATION)`
`field`	Path to a `.pos` file with the field, scalar or vector
`potential`	Path to a `.pos` file with a scalar potential; the field is the negative gradient
`unit`	Unit of the values in the file
`default_if_zero`	Use the constant instead of a cell whose file value is zero
`none`	Explicitly no field

The three fields a case can have are fixed slots:

- `ELECTRIC` — either from a potential (PICMC writes the potential, so this is the normal route) or as a constant vector.
- `MAGNETIC` — from a file or constant; only needed if the particles are charged and the field is strong enough to bend their path.
- `GRAVITY` — almost always the constant `(0, 0, -GRAVITATIONAL_ACCELERATION)`.

`default_if_zero` exists for a practical reason: a plasma field from PICMC is zero outside the plasma region, and without the flag the particle would see a temperature of 0 K there instead of the neutral gas temperature.

Species

Two lists, and the distinction matters:

- `PLASMA_SPECIES_LIST` — the charged species of the plasma (electrons, ions, sometimes negative ions). They do not move; they provide the currents that charge the particles and the gas properties for drag and thermophoresis.
- `SPECIES_LIST` — the particle species that are actually traced. This is what `spawn_particles*` refers to.

A `PlasmaSpecies` in practice needs `mass`, `density` and `temperature`; `charge` is the charge number (−1 for electrons, +1 for singly charged ions). Optional entries are `velocity`, `pressure`, `temperature_gradient`, `pressure_gradient`, `heat_flux`, `cross_section`, `dynamic_viscosity`, `heat_conductivity` and `relative_humidity`. Each of them is a `Field`, so every one can come from a file.

Warning: the defaults of these entries are placeholders — a mass of 1 kg, a viscosity of 1 Pa·s and a relative humidity of 1.0. Set the properties of your gas explicitly; a humidity of 1.0 means saturated vapour, which makes every droplet in the case grow instead of evaporate.

A `Species` (the traced particle) has:

Field	Meaning
`size_distribution_type`	`DISTRIBUTION_CONSTANT`, `DISTRIBUTION_UNIFORM`, `DISTRIBUTION_LOG_NORMAL` or `DISTRIBUTION_WEIBULL`
`size_distribution_param_1`, `..._2`	Radius parameters in metres; for the uniform distribution the lower and upper bound
`mass_density`	Density of the particle material
`dry_mass_density`	Density of the non-volatile core, for droplets
`initial_charge_density`	Charge per volume at the start, in C/m³
`relative_permittivity`	For the polarisation force
`hamaker_constant`	For the van der Waals adhesion
`minimum_wall_distance`	Distance at which the adhesion is evaluated
vapour properties	`saturation_pressure`, `latent_heat`, `diffusivity`, `molar_fraction`, `specific_heat_p`, `vapor_gas_constant` and their temperature derivatives

An empty `PLASMA_SPECIES_LIST` is legitimate: without plasma species no current reaches the particle, so it keeps the charge it was given and only the neutral-gas forces act on it.

Starting particles

```
server.spawn_particles_on_surface(100, border_name = "T1", species_name = "Sample")
server.spawn_particles(50, species_name = "Sample",
                       position = (0.1, 0.0, 0.02),
                       velocity = (0.0, 0.0, 1.0))
```

- `spawn_particles_on_surface` distributes particles over the triangles of a border and gives them the border's `ejection_speed` along the surface normal. This is the usual start for a contamination study: particles are released where they are produced.
- `spawn_particles` places particles at one point with one velocity. Useful for a single test particle.
- `inject_test_particle` / `inject_test_particles` add particles during a run, for example from an `each_tick` callback.

Both are called before `run()`; the initial dump is taken after spawning.

Running a case

`pyPALADINServer(pars)` creates the run directory, initialises the mesh, distributes the volume over the clients and prepares the dumpers. `run()` then advances the simulation until `TSIM` is reached, writes the dumps at their intervals and prints the progress block.

The run directory is created next to the script and is named after `NAME`. All dumps and log files go there, unless `DUMP_PATH` or `LOGFILE_PATH` point to a subdirectory.

Two attributes of the server object change the console behaviour:

```
server.show_interface = True        # progress block, False for batch jobs
server.interface_update_interval = 100   # time steps between two updates
```

Callbacks

The server object can call a function during the run:

```
def each_tick(server):
    if server.get_time() > 0.05:
        server.reload_field("ELECTRIC")

server.each_tick = each_tick
```

`each_tick` runs once per time step, `at_time(value)` and `every_n_steps(n)` are helpers to test against the current time. This is the hook for time-dependent fields: PICMC fields are written per time slice, and `reload_field` reads the next one. `reload_field_unit` changes the unit of a field file, `reload_trait` and `reload_trait_unit` do the same for the entries of a plasma species, and `push_field_to_cache` / `reload_field_from_cache` keep a loaded field in memory instead of reading the file again.

Launching

The Python script is one MPI rank; the clients are the others. On a machine without a batch system, the whole run is one `mpiexec` line with two MPMD parts:

```
export PALADINPATH=/path/to/paladin
export PYTHONPATH=$PALADINPATH/libs:$PALADINPATH/python_scripts:$PYTHONPATH
export LD_LIBRARY_PATH=$PALADINPATH/libs:$LD_LIBRARY_PATH

mpiexec -np 1 --bind-to none python3 case.py : -np 8 $PALADINPATH/start_client
```

The number of clients is free: each of them receives its share of the particles. On a Slurm cluster, `submit_py` writes the batch script and submits it:

```
./submit_py -n 8 -partition epyc -time 24:00:00 case.py
```

Option	Meaning
`-n <n>`	Number of client ranks

Option	Meaning
`-N <n>`	Number of nodes
`-partition <p>`, `-time <hh:mm:ss>`, `-mem <MB>`, `-exclude <list>`, `-exclusive`	Slurm settings
`-dummy`	Only write the batch file, do not submit
`-test`	Write the batch file and submit it as a test job
`-valgrind <log>`, `-callgrind <log>`, `-massif <log>`	Run server and clients under Valgrind

`-dummy` is the fastest way to see what the job would look like: it produces `<script>.slurm` next to the case.

Output

A dumper instance is one output file family. Its options:

Option	Meaning
`file_template`	Name fragment; the file becomes `<NAME>_<file_template>`
`values`	List of quantities to write, see constants and units
`value_units`	One entry per quantity, in which unit it is written
`total_number`	Number of dumps over the whole run
`mode`	`DUMP_ALL_IN_ONE` (one file, one block per step) or `DUMP_SEPARATE` (one file per step)
`format`	`FORMAT_GMSH_POS`, `FORMAT_TXT`, `FORMAT_GMSH_MSH` or `FORMAT_EMDI_DMP`
`additional_path`	Subdirectory inside the run directory
`particle_species`	Restrict the dump to one species
`collision_dumper`	Write only on collisions
`loss_dumper`	Write only particles that left the volume
`additional_dump_at_start`	Write the initial state as well
`add_counter`	Append a running number to the file name
`file_float_precision`, `binary`	Formatting

Typical set for a contamination study: one dumper with positions over time (`FORMAT_GMSH_POS`, small `total_number`), one collision dumper with velocities and kinetic energy, one loss dumper, and one final dump with the backtrace quantities `DUMP_SCALAR_STARTING_ELEMENT_ID` and `DUMP_SCALAR_LAST_SURFACE_TAG` — that pair is what makes "particles came from the target" a checkable statement.

For a backtrace, the relevant borders need `tracking = True`; without it the origin is not recorded.

Warning: `total_number` is spread over the run, not a time interval. A value of `round(TSIM/DT)` writes every step and fills the disk quickly with many particles; the magnetron example deliberately uses `round(TSIM/DT/10)` for the trajectory dump.

Controlling a running job

`signal_helper` sends commands to a running job by process name:

```
$PALADINPATH/signal_helper -s d server        # dump now
$PALADINPATH/signal_helper -s p server        # pause
$PALADINPATH/signal_helper -s u server        # continue
$PALADINPATH/signal_helper -s s server        # shut down in an orderly way
$PALADINPATH/signal_helper -s k server        # kill
```

The letters can be combined, and `c[...]` sends a custom command. `-h` prints the help.

Shipped example cases

The examples that come with a `make full` build are small on purpose: each one isolates a different part of the workflow.

Case	What it is	What it teaches
`Example_Capacitor`	Two walls, a constant electric field, no plasma, ten 1 µm particles, 2.5 s	The minimal case: field plus gravity, one dumper, nothing else
`Example_Charging`	Uniform electron and argon ion density of 10^{10} m^{-3} (10^4 cm^{-3}) at 10^4 K, five test particles, a strong magnetic field	Charging of a particle in a homogeneous plasma; dumps of charge and potential over time
`Example_Magnetron`	PICMC fields of the EOSS® geometry read from `.pos` files, 1–2 µm particles, 100 released from each target	The realistic workflow: fields from files, backtrace dumpers, collision dumpers, a size distribution
`Example_Tube`	Gas density, pressure, temperature, velocity and heat flux from a flow solution, mesh in millimetres, walls, inlet and pump	A gas-flow case without a plasma: aerosol transport in a tube, large `MESH_DIVISION`

Warning: the example scripts reference their mesh and field files as `../…`, because the working directory of a run is meant to be a subdirectory of the case. Running the script directly in the example folder fails to find the files. Copy the script into a subdirectory of the case, start it there, and the run directory appears next to it.

Before a long run

- Check `MESH_DIVISION`: a grid that is too coarse makes wall collisions slow, one that is too fine costs memory before the run even starts.
- Check the unit of the mesh. A mesh in millimetres with `UNIT = METER` puts every particle outside the volume, and the run ends immediately with everything lost.
- Check that every physical group that particles can hit is in `BORDER_TYPES`.
- Start with a small number of particles and `additional_dump_at_start = True`; if the initial state already looks wrong in ParaView, the fields or the geometry are wrong, not the physics.
- `SEED_FOR_FIRST_CLIENT` makes a run reproducible. Change it deliberately, not accidentally.

74 Particle model

PALADIN is a Lagrangian code: every particle is followed individually, and all quantities that act on it are evaluated at its current position from the fields that were read at start-up. This page describes what those quantities are and which parameter switches them on. The derivation — moments of the Maxwell-Boltzmann distribution, the resulting force and current expressions, and their validation — is in the dissertation and is not repeated here.

Time integration

The particle is advanced with a fixed time step `DT`, which is a physical time in seconds. The step has to be short enough for two things: the particle must not cross a cell per step (the fields are interpolated per cell, so a large step means the particle sees an averaged field), and the charging must not overshoot — a particle that gains its whole equilibrium charge within one step oscillates.

Note: `MINOR_TIME_STEP_COUNT` subdivides `DT` internally. It helps with the stability of the integration, but it does not change the output interval and it does not make a coarse spatial resolution finer.

Charging and force evaluation are per particle and per step; there is no particle-particle interaction included in the force sum except the cohesion of particles that are in contact.

Fields the particle sees

All fields are cell-centred and are interpolated at the particle position. A scalar, a vector, its square norm, its gradient and its Hessian are all available, which is what the different force terms need — the polarisation force needs a field gradient, the electric force only the field itself.

If a field was read from a file, a cell value of zero can be replaced by the constant that was given alongside it (`default_if_zero`). That is how a plasma field that only covers part of the volume keeps a sensible neutral-gas temperature outside.

Forces

Every force has its own switch in the parameter object, and the dump can write the contribution of each one separately — which is the practical way to find out what dominates a trajectory.

Switch	Force	What it needs
`USE_GRAVITY`	Weight	`GRAVITY`
`USE_ELECTRIC_FORCE`	Coulomb force `qE`	`ELECTRIC`
`USE_MAGNETIC_FORCE`	Lorentz force `qv×B`	`MAGNETIC`
`USE_POLARIZATION_FORCE`	Force on the induced dipole in a field gradient	`ELECTRIC`, `relative_permittivity`
`USE_DEBYE_POLARIZATION_FORCE`	Force from the polarised ion cloud around a charged particle	plasma species
`USE_COLLISIONAL_DRAG`	Friction with the neutral gas	gas species, pressure, temperature

Switch	Force	What it needs
`USE_COULOMB_DRAG`	Ion drag from Coulomb collisions with the plasma	plasma species, charge
`USE_COLLISIONAL_THERMOPHORESIS`	Force along a temperature gradient in the neutral gas	gas temperature gradient
`USE_COULOMB_THERMOPHORESIS`	The same for the plasma species	plasma temperature gradient

Default state: gravity, electric force, magnetic force, both drag terms and both thermophoresis terms on; polarisation and Debye polarisation off. That is the set for a micrometre particle in a low-pressure plasma: the neutral drag and the electric force dominate, the magnetic force is negligible for a slow, heavy particle unless the field is strong, and the polarisation terms only matter for a dielectric particle in a strong field gradient.

The two thermophoresis and two drag terms are deliberately separate: a particle that is charged interacts with the plasma through Coulomb collisions, and a particle that is neutral only through hard collisions with gas molecules. Whether both are relevant depends on the ionisation degree.

Charging

The charge of a particle is a state variable and is written in units of the elementary charge. In a plasma it reaches thousands to tens of thousands of elementary charges for a micrometre particle, and it follows the local plasma rather than the distance to a wall.

Charging is computed from the currents that each plasma species delivers to the particle surface. For every species the code evaluates the flux of ions or electrons onto the sphere in the presence of the particle's own charge and of the drift of the species through the gas, with a shielding length that combines the Debye length with a term from the directed motion. The result is a current per species, and the sum over species moves the charge one step forward.

Switch	Effect
`USE_PARTICLE_CURRENTS`	Turn the current-based charging on at all. Without it the charge stays at its initial value
`MORE_ACCURATE_COULOMB_FORCE_CALCULATION`	Use the more expensive evaluation of the Coulomb interaction
`DELETE_PARTICLES_IF_INVALID_CHARGE`	Remove a particle whose charge became unphysical instead of carrying the value through the run
`MINIMUM_PARTICLE_SIZE`	Lower limit for the radius of an evaporating droplet, see below

Warning: the charging needs at least one charged plasma species with a density and a temperature. The species lists are not validated for physical consistency — an electron density without ions charges the particle to a potential that no plasma would produce, and the run looks perfectly healthy while doing so.

What the charge is used for is everything downstream: the Coulomb force, the ion drag, the Coulomb thermophoresis, the Debye polarisation force, and the currents again in the next step. A particle that is charged in a region with a field is therefore also accelerated there, which is the mechanism behind the EOSS® observation on the overview page.

Drag and thermophoresis in two regimes

Both the neutral drag and the neutral thermophoresis depend on the Knudsen number — the ratio of the mean free path to the particle radius. At low pressure the particle is in the free molecular regime and the momentum transfer is a sum over single molecule impacts; at higher pressure the gas behaves as a continuum around the particle and the force follows the Stokes form with a slip correction.

`LOW_KNUDSEN_NEUTRAL_GAS_CALCULATION` selects the continuum form for the neutral gas terms. It is off by default, which is right for a sputter chamber at 10^{-3} to 10^{-1} mbar and wrong for an atmospheric case.

`REFLECTION_COEFFICIENT` belongs to the same family of terms: it describes how a gas molecule leaves the particle surface and therefore how much tangential momentum it takes away. The default of 1.0 means specular reflection; a value of 0 corresponds to full accommodation. If the literature values for your gas-particle combination are available, use them here — the drag term scales with this coefficient directly.

`REFERENCE_TEMPERATURE` and `REFERENCE_PRESSURE` are the state at which the gas properties (viscosity, thermal conductivity) are evaluated before they are scaled to the local temperature. They do not change the physical setup of the case, only the property model.

Surfaces: reflection, adhesion, rolling

A wall collision is decided in two steps. First the particle bounces with the restitution given by the border's `elasticity`. Then an energy balance is evaluated for the normal component: the kinetic energy after the impact against the van der Waals well of the surface, which is built from the Hamaker constant of the particle species and the minimum wall distance.

- If the remaining energy is above a threshold, the particle is reflected with the reduced normal velocity.
- Otherwise it sticks. From then on the time since sticking is recorded, and the particle can roll along the surface under the tangential force balance.

That is why `elasticity` and `hamaker_constant` together decide whether a surface collects particles or not. A perfectly elastic wall (`elasticity = 1.0`) has no energy loss, so the sticking criterion is reached only by the van der Waals term — with the default Hamaker constant of 10^{-19} J a micrometre particle usually bounces, a sub-micrometre particle usually sticks.

Switch	Effect
`CHECK_FOR_WALLS`	Test whether a particle is inside a wall volume, not only whether it crossed a surface
`CHECK_ALL_SURFACE_ANGLES_FOR_ROLLING_PARTICLES`	For a rolling particle, check all surface normals of the element instead of only the current one
`WALL_MODULUS`	Parity setting for the inside-the-wall test; flip it for a mesh whose surface orientation is inverted

`WALL_MODULUS` exists because the inside/outside test counts how often a ray from the particle to the boundary crosses the surface. For a mesh with correctly oriented normals the default of 1 is right; if a particle that is clearly outside is treated as being inside a wall, the mesh orientation is the cause.

Cohesion and agglomerates

`CALCULATE_MESH_CONNECTIONS` (on by default) builds a connectivity graph of all particles that touch each other, so that a cluster moves as one body instead of as independent spheres. The graph is computed once and cached in a `connection_caches` directory inside the run directory.

For a contamination study this is the difference between a cloud of single particles and a realistic agglomerate that falls, hits the substrate and stays there.

Droplets: evaporation and condensation

A particle species can carry vapour properties: saturation pressure, latent heat, diffusivity, molar fraction and specific heat of the condensable component, plus the relative humidity of the surrounding gas species. With those, the radius of the particle changes along its path — it evaporates in dry gas and takes up vapour in a saturated one — and the heat of condensation changes its temperature.

The radius is changed by the mass flux to or from the surface, evaluated with a Nusselt number for mass transfer; the temperature follows from the heat flux with the Nusselt number for heat transfer and the latent heat term. Two knobs control the numerical side:

- `TEMPERATURE_DIFFERENCE_FOR_INSTANT_EQUILIBRIUM` — if the computed temperature change per step exceeds this value, the particle is set to the gas temperature instead of being integrated. This avoids an oscillation of the temperature in a fast-changing environment.
- `MINIMUM_PARTICLE_SIZE` — the radius at which the droplet is set back to its initial radius and temperature. With the default of 0 the limit is the radius of the non-volatile core, so a droplet ends as its dry core and starts over as a fresh droplet.

This part of the model is why the same code is used for aerosol problems outside coating, where the carrier phase is air with a humidity and the particles are droplets.

Where the equations are

The force and current expressions, the shielding length, the collision model and the validation against the EOSS® coater are in the dissertation cited on the overview page. The switches on this page are named exactly as in the source, so the mapping from a parameter to a term in the thesis is direct.

75 Parameters

Everything a case needs is passed as keyword arguments to `Parameters(...)`. This page lists all keys, grouped by what they do, with the value that applies when the key is left out.

Two rules apply to all of them: the object is validated at start-up (a wrong type or format stops the script), and unknown keywords are a Python error, not a warning — a misspelled key does not silently fall back to its default.

Run identity and output

Key	Type	Default	Meaning
`NAME`	str	required	Name of the run. The run directory is created with this name next to the script
`DT`	float	required	Time step in seconds. Must not be larger than `TSIM`
`TSIM`	float	required	Simulated time in seconds
`MESH_FILE`	str	required	Gmsh mesh file
`UNIT`	float	`1.0`	Length unit of the mesh, in metres
`DUMP_PATH`	str	`"dumps"`	Subdirectory of the run directory for all dumps
`LOGFILE_PATH`	str	`"logfiles"`	Subdirectory of the run directory for the log files
`LOGFILE_SERVER`	str	`"log_server"`	Base name of the server log; `.txt` is appended
`LOGFILE_CLIENT`	str	`"log_client"`	Base name of the client logs, one per rank
`SEED_FOR_FIRST_CLIENT`	int	`-1`	Seed of the random number generator. A negative value leaves the seed to the system
`VERBOSE_SERVER`	int	`VERBOSE_NONE`	Verbosity of the server, as a combination of `VERBOSE_*` flags
`VERBOSE_CLIENT`	int	`VERBOSE_NONE`	Verbosity of the clients

Dump files are named `<NAME>_<file_template>` inside `DUMP_PATH`. `additional_path` of a dumper instance adds another level below it.

Domain

Key	Type	Default	Meaning
`BB`	6 floats	required	Bounding box `(x_min, y_min, z_min, x_max, y_max, z_max)` in `UNIT`
`BB_OPTIONS`	3 ints	required	Behaviour of the box per direction: `BB_SOLID`, `BB_PERIODIC`, `BB_OUT_MIN`, `BB_OUT_MAX`, `BB_OUT`
`MESH_DIVISION`	3 ints	required	Resolution of the internal element lookup grid per direction; the product is the number of grid cells
`BORDER_TYPES`	dict	required	Named `Border` objects, each with the `icodec` of a physical group

A physical group that is not listed in `BORDER_TYPES` is not a wall for the particles: a particle that reaches it crosses it as if it were open space.

Physics switches

The switches are integers, and any non-zero value means on.

Key	Default	Meaning
`USE_GRAVITY`	`1`	Weight, from the `GRAVITY` field
`USE_ELECTRIC_FORCE`	`1`	Coulomb force `qE`
`USE_MAGNETIC_FORCE`	`1`	Lorentz force `qv×B`
`USE_POLARIZATION_FORCE`	`0`	Force on the induced dipole in a field gradient
`USE_COLLISIONAL_DRAG`	`1`	Friction with the neutral gas
`USE_COULOMB_DRAG`	`1`	Ion drag from Coulomb collisions
`USE_COLLISIONAL_THERMOPHORESIS`	`1`	Thermophoresis in the neutral gas
`USE_COULOMB_THERMOPHORESIS`	`1`	Thermophoresis in the plasma
`USE_PARTICLE_CURRENTS`	`1`	Charge the particles from the plasma species
`USE_DEBYE_POLARIZATION_FORCE`	`0`	Force from the polarised ion cloud
`REFLECTION_COEFFICIENT`	`1.0`	Gas-surface scattering: 1.0 specular, 0.0 full accommodation
`LOW_KNUDSEN_NEUTRAL_GAS_CALCULATION`	`0`	Use the continuum (Stokes) form of the neutral gas terms instead of the free-molecular one
`MORE_ACCURATE_COULOMB_FORCE_CALCULATION`	`0`	More expensive evaluation of the Coulomb interaction
`CALCULATE_MESH_CONNECTIONS`	`1`	Build the connectivity graph of touching particles (agglomerates)
`REFERENCE_TEMPERATURE`	`300.0`	Reference temperature for the gas property model, in K
`REFERENCE_PRESSURE`	`101325.0`	Reference pressure for the gas property model, in Pa

Numerical and termination control

Key	Default	Meaning
`MINOR_TIME_STEP_COUNT`	`1`	Sub-steps per `DT`; a negative value is treated as 1
`WALL_MODULUS`	`1`	Parity of the inside-the-wall test; flip it for an inverted mesh orientation
`CHECK_FOR_WALLS`	`0`	Test whether a particle is inside a wall volume, not only on a crossing
`CHECK_ALL_SURFACE_ANGLES_FOR_ROLLING_PARTICLES`	`0`	Check all normals of an element for a rolling particle
`TEMPERATURE_DIFFERENCE_FOR_INSTANT_EQUILIBRIUM`	`10.0`	Temperature change per step above which a droplet is set to the gas temperature
`MINIMUM_PARTICLE_SIZE`	`0.0`	Radius at which a droplet is reset; 0 means the radius of the dry core

Key	Default	Meaning
`DELETE_PARTICLES_IF_INVALID_CHARGE`	`0`	Remove a particle whose charge became unphysical
`TERMINATION_LEVEL_IF_ALL_PARTICLES_VANISHED`	`NO_TERMINATION`	What happens when no particle is left: `NO_TERMINATION`, `TERMINATION_LEVEL_1`, `TERMINATION_LEVEL_2`

`TERMINATION_LEVEL_IF_ALL_PARTICLES_VANISHED` is the switch to keep a long run from burning wall-clock time on an empty box: a case whose particles have all left or stuck produces nothing after that moment.

Content

Key	Type	Meaning
`SPECIES_LIST`	dict of `Species`	The traced particle species, by name
`PLASMA_SPECIES_LIST`	dict of `PlasmaSpecies`	The plasma species that charge the particles; may be empty
`BORDER_TYPES`	dict of `Border`	The walls, by name
`DUMPER_INSTANCES`	dict of `DumperInstance`	The output files, by name
`ELECTRIC`	`Field`	Electric field, constant or from a potential file
`MAGNETIC`	`Field`	Magnetic field
`GRAVITY`	`Field`	Gravity

The three field slots may be `None`; a slot that is `None` and a switch that is on is not an error, the field simply contributes nothing.

Species keys

`Species` describes the traced particle:

Key	Default	Meaning
`size_distribution_type`	`DISTRIBUTION_CONSTANT`	`DISTRIBUTION_CONSTANT`, `DISTRIBUTION_UNIFORM`, `DISTRIBUTION_LOG_NORMAL`, `DISTRIBUTION_WEIBULL`
`size_distribution_param_1`	`1.0`	First radius parameter in metres (bound, mean or scale depending on the distribution)
`size_distribution_param_2`	`0.0`	Second radius parameter (upper bound or width)
`size_distribution_param_int`	`0`	Integer parameter of the distribution
`mass_density`	`1.0`	Density of the particle material in kg/m³
`dry_mass_density`	`1.0`	Density of the non-volatile core
`initial_charge_density`	`0.0`	Initial charge per volume in C/m³
`relative_permittivity`	`1.0`	For the polarisation force
`hamaker_constant`	`1.0e-19`	For the van der Waals adhesion, in J
`minimum_wall_distance`	`4.0e-10`	Distance at which the adhesion is evaluated, in m

Key	Default	Meaning
`vapor_gas_constant`	`1.0`	Vapour property of the condensable component
`latent_heat`	`1.0`	Latent heat of evaporation
`dT_latent_heat`	`0.0`	Temperature dependence of the latent heat
`dT_mass_density`	`0.0`	Temperature dependence of the density
`saturation_pressure`	`1.0`	Saturation pressure of the vapour
`dT_saturation_pressure`	`0.0`	Temperature dependence of the saturation pressure
`diffusivity`	`1.0`	Diffusion coefficient of the vapour in the gas
`diffusivity_exponent`	`1.0`	Temperature exponent of the diffusivity
`molar_fraction`	`1.0`	Molar fraction of the condensable component
`specific_heat_p`	`1.0`	Specific heat at constant pressure

Warning: the defaults are placeholders, not physical values. In particular `mass_density = 1.0` and `hamaker_constant = 1e-19` are only sensible for a rough test; a real case sets at least the density, the size distribution and, if adhesion matters, the Hamaker constant of the material pair.

`PlasmaSpecies` describes a species of the background plasma or gas:

Key	Default	Meaning
`mass`	`1.0`	Particle mass in kg, e.g. `40*AMU`
`charge`	`0`	Charge number, −1 for electrons, +1 for singly charged ions
`density`	`Field`	Number density
`temperature`	`Field`	Temperature
`velocity`	`Field`	Drift velocity, vector
`pressure`	`Field`	Pressure
`energy`	`Field`	Energy
`temperature_gradient`	`Field`	For thermophoresis
`pressure_gradient`	`Field`	Pressure gradient
`heat_flux`	`Field`	Heat flux
`cross_section`	`3.0e-19`	Collision cross-section in m^2, for the Knudsen number
`specific_heat_p`	`1.0`	Specific heat at constant pressure
`heat_conductivity`, `dT_heat_conductivity`	`1.0`, `0.0`	Thermal conductivity and its temperature dependence
`dynamic_viscosity`, `dT_dynamic_viscosity`	`1.0`, `0.0`	Dynamic viscosity and its temperature dependence
`relative_humidity`	`1.0`	Humidity of the gas, for droplet evaporation

`Border` and `DumperInstance` are documented with the fields they set on the scenario page and in the output section.

76 Constants and units

`from pyPALADINServer import *` brings in three things besides the classes: the unit constants, the physical constants, and the value constants that are passed to a `Border` or a `DumperInstance`. All of them are plain numbers, so they can be combined in the usual way.

Units

Every unit constant is 1.0, because PALADIN works in SI units throughout. The names exist to make a script readable and to make unit errors visible:

```
size = 1*MICRO*METER            # 1e-6 m
density = 1000*KILOGRAM*PER(CUBIC(METER))
field = 1*VOLT*PER(METER)
```

Quantity	Constants
Length	`METER`, `METRE`, `ANGSTROM`, `MICRO`, `NANO`, `MILLI`, `CENTI`, `DECI`, `KILO`, …
Time	`SECOND`, `MINUTE`, `HOUR`, `DAY`, `WEEK`, `MONTH`, `YEAR`
Mass	`KILOGRAM`, `GRAM`, `TON`
Pressure	`PASCAL`, `BAR`, `TORR`, `ATM`
Temperature	`KELVIN`
Electrical	`AMPERE`, `COULOMB`, `VOLT`, `TESLA`, `HENRY`, `ELECTRON_VOLT`
Energy and power	`JOULE`, `WATT`, `NEWTON`

The SI prefixes run from `YOCTO` (1e-24) to `YOTTA` (1e24): `DECA`, `HECTO`, `KILO`, `MEGA`, `GIGA`, `TERA`, `PETA`, `EXA`, `ZETTA` upward and `DECI`, `CENTI`, `MILLI`, `MICRO`, `NANO`, `PICO`, `FEMTO`, `ATTO`, `ZEPTO` downward.

Three helpers exist for compound units: `SQUARE(x)`, `CUBIC(x)` and `PER(x)`, plus `LOG_SCALE(x)` for a factor of `x` decades.

Warning: `ATM` is defined as `1.01325e3` Pa, which is 1013.25 Pa — a factor of 100 below a standard atmosphere. Use `PASCAL`, `BAR` or `TORR` for a pressure, or write `1.01325e5*PASCAL` explicitly.

Physical constants

Constant	Value	Unit
`ELEMENTARY_CHARGE`	1.6021766208e-19	C
`AMU`	1.660539040e-27	kg
`ELECTRON_MASS`	9.10938356e-31	kg
`BOLTZMANN_CONSTANT`	1.38064852e-23	J/K
`LIGHT_SPEED`	299792458.0	m/s
`PLANCK_CONSTANT`	6.626070040e-34	J·s
`GRAVITATIONAL_ACCELERATION`	9.81	m/s^2, always positive

`GRAVITATIONAL_ACCELERATION` carries no direction: the sign comes from the vector in the `GRAVITY` field, e.g. `(0, 0, -GRAVITATIONAL_ACCELERATION)` for a z-up geometry.

Wall types

Constant	Behaviour on impact
`WALL_TYPE_SPECULAR`	Mirror reflection; the restitution is the border's `elasticity`
`WALL_TYPE_DIFFUSE`	Re-emitted in a random direction at unchanged speed
`WALL_TYPE_MEMBRANE`	No change of velocity, the particle passes through

Whether the particle adheres after the impact is decided by the adhesion model, not by the wall type.

Size distributions

Constant	Parameters
`DISTRIBUTION_CONSTANT`	`size_distribution_param_1` is the radius
`DISTRIBUTION_UNIFORM`	`param_1` and `param_2` are the lower and upper radius
`DISTRIBUTION_LOG_NORMAL`	`param_1` and `param_2` describe the underlying normal distribution of the logarithm
`DISTRIBUTION_WEIBULL`	`param_1` and `param_2` are scale and shape of the Weibull distribution

Note: the meaning of `param_1` and `param_2` beyond the constant and the uniform case follows the distribution formula and is not spelled out in the source. For a quantitative size distribution, check the resulting radii in the initial dump before running a long case.

Bounding box options

Constant	Meaning
`BB_SOLID`	Closed by a wall
`BB_PERIODIC`	Periodic; a particle that leaves enters on the opposite side
`BB_OUT_MIN`	Open at the lower coordinate
`BB_OUT_MAX`	Open at the upper coordinate
`BB_OUT`	Open on both sides

A particle that leaves through an open boundary is lost and can be written by a `loss_dumper`.

Dump formats and modes

Constant	Meaning
`FORMAT_GMSH_POS`	Gmsh `.pos` view file, openable in ParaView and Gmsh
`FORMAT_GMSH_MSH`	Gmsh mesh file
`FORMAT_TXT`	Plain text table, one row per particle and dump

Constant	Meaning
`FORMAT_EMDI_DMP`	Binary dump in PALADIN's own format
`DUMP_ALL_IN_ONE`	One file, one block per dump step
`DUMP_SEPARATE`	One file per dump step

`FORMAT_GMSH_POS` is the right choice for anything that is looked at; `FORMAT_TXT` for anything that is evaluated with a script, because it needs no reader.

Quantities a dumper can write

The `values` list of a dumper instance takes any combination of the constants below, and `value_units` takes one entry per value.

Scalars

Constant	Quantity
`DUMP_SCALAR_ID`	Particle index
`DUMP_SCALAR_TIME`	Simulation time
`DUMP_SCALAR_RADIUS`	Radius
`DUMP_SCALAR_MASS`	Mass
`DUMP_SCALAR_CHARGE`	Charge in elementary charges
`DUMP_SCALAR_CHARGE_MASS_RATIO`	Charge-to-mass ratio
`DUMP_SCALAR_KINETIC_ENERGY`	Kinetic energy
`DUMP_SCALAR_VELOCITY_SQUARE`	Squared velocity
`DUMP_SCALAR_POTENTIAL`	Potential at the particle position
`DUMP_SCALAR_COLLISION_TYPE`	Kind of the last collision (reflection, sticking, …)
`DUMP_SCALAR_LAST_SURFACE_TAG`	Physical group the particle last touched
`DUMP_SCALAR_STARTING_ELEMENT_ID`	Element the particle started from, the backtrace quantity
`DUMP_SCALAR_TIME_SINCE_STICKING`	Time since the particle stuck
`DUMP_SCALAR_MAX_STICKING_TIME`	Longest sticking time so far
`DUMP_SCALAR_GENERAL_STICKING_TIME`	Accumulated sticking time
`DUMP_SCALAR_FLAG_CRITICAL_DENSITY`	Set when the particle is in a region with a plasma density of $2 \cdot 10^{12}$ m^{-3} or more
`DUMP_SCALAR_ONE`	Constant 1, for a pure position dump

Detail scalars

Constant	Quantity
`DUMP_DETAIL_SCALAR_MAX_CURRENT`	Largest charging current the particle saw
`DUMP_DETAIL_SCALAR_SHIELDING_LENGTH`	Shielding length used in the charging

Constant	Quantity
`DUMP_DETAIL_SCALAR_SHIELD_STRENGTH`	Strength of the shielding
`DUMP_DETAIL_SCALAR_TEMPERATURE`	Particle temperature
`DUMP_DETAIL_SCALAR_DENSITY`	Local gas density
`DUMP_DETAIL_SCALAR_CHI`, `DUMP_DETAIL_SCALAR_XI`	Normalised charge and drift parameter of the charging model
`DUMP_DETAIL_SCALAR_COULOMB_LOG`	Coulomb logarithm used for the collision terms

Vectors

Constant	Quantity
`DUMP_VECTOR_POSITION`	Position
`DUMP_VECTOR_VELOCITY`	Velocity
`DUMP_VECTOR_ACCELERATION`	Acceleration

Detail vectors, one per force term

Constant	Quantity
`DUMP_DETAIL_VECTOR_FORCE_GRAVITY`	Weight
`DUMP_DETAIL_VECTOR_FORCE_ELECTRIC`	Coulomb force
`DUMP_DETAIL_VECTOR_FORCE_MAGNETIC`	Lorentz force
`DUMP_DETAIL_VECTOR_FORCE_POLARIZATION`	Polarisation force
`DUMP_DETAIL_VECTOR_DEBYE_POLARIZATION`	Debye polarisation force
`DUMP_DETAIL_VECTOR_DRAG_COLLISIONAL`	Neutral drag
`DUMP_DETAIL_VECTOR_DRAG_ORBITAL`	Ion drag (Coulomb collisions)
`DUMP_DETAIL_VECTOR_THERMO_COLLISIONAL`	Neutral thermophoresis
`DUMP_DETAIL_VECTOR_THERMO_ORBITAL`	Plasma thermophoresis
`DUMP_DETAIL_VECTOR_HEAT_FLUX`	Local heat flux
`DUMP_DETAIL_VECTOR_GRAD_TEMPERATURE`	Local temperature gradient

The force breakdown is the most useful diagnostic in the code: a trajectory that goes where it should not go can be explained by looking at which of these terms dominates where.

Full state

`DUMP_SERIALIZED` writes the complete state of a particle as one flat list of values — species, index, inverse mass, charge, radius, temperature, initial radius, position, velocity, previous acceleration, flags and the collision state. It is the format the particle serialisation uses, and it is larger than any other dump.

Troubleshooting

77 Troubleshooting & warnings

In case your simulation does not start properly, this site gives some hints how to find possible issues. Many potential problems are intercepted and (hopefully) comprehensible error messages are printed. If this is not the case, there are different kinds of errors:

- Not even the information about the picmc version is printed, most probable this is a Parameter file issue.
- The simulation does not reach the point: **Starting time cycles**, maybe this is a Geometry issue.
- Terminating simulations at any time could be caused by Memory issues.

Parameter file issue

A common troublemaker are missing semicolon at the line ends. The first approach would be to assure, that every line in the parameter file is terminated by a semicolon (except those ending with an open curly bracket). Comments, which start with a hash mark, are perfectly fine after a semicolon.

Geometry issue

One of the first steps to find possible problems is to have a look at the `cell_volumes.pos` plot. In this plot you can see if the inside/outside algorithm correctly determined your simulation volume or if due to some double defined or missing surfaces the particles are placed inside the chamber walls. Also you should check your mesh file by cycling through the individual *Physical Surfaces* with the *Visibility* tool of GMSH and verifying that there are no peculiarities.

Memory issue

If you have a big simulation volume and the program terminates at the first time cycles or shortly before, maybe the available memory of your computer is not enough. On your own cluster have a look at the cluster report otherwise talk to your admin. To solve this, try to reduce the simulation volume or the particle numbers (as long as the constraints are satisfied). If that is not possible get more memory.

Warning messages

Below some warning messages are listed and more information about the origin and the solution (if any) is given.

- **N particles of type M are located inside the mesh volume!**

  If you read a checkpoint file into a simulation with different mesh (compared to the mesh with which the checkpoint was created), some of the particles may now be positioned inside the mesh. These particles then will be deleted. As long as the number of particles is not too high you don't have to worry. This message may also occur if your mesh resolution is too coarse, so that particles can "tunnel" into the mesh.

- **DSMC_Emitter, yield function: Energy/Angular range violated**
- **No surface found at hit position -> SP(X, Y, Z){N};**
- **Check the orientation of the reference vector of the histogram with Codec N, it looks as if some mesh elements (M %) are orthogonal to it!**

  For angular histograms a reference direction where the azimuth angle is zero has to be defined. If this reference vector is orthogonal to the mesh element plain, the azimuth angle is undefined. For oblique surfaces the projection of the reference vector onto its plain is taken.

- **Elements with codecNumber N are ignored...OK?**

  You can delete whole codec/border blocks from your parameter file if you don't need them while using an existing mesh file. In this case the simulation ignores all mesh elements which belong to the deleted codec. This warning is just for your convenience to prevent accidental codec deletions.

- **Collision cross-section table too short: channel N-M has X collisions above its upper energy limit of Y eV**

  For each collision channel a cross-section table is generated up to a maximum energy, by default 1100 eV. A collision above that limit is not an error and does not stop the run: the cross section is held constant at the last table entry, which usually over-estimates the rate. The message reports how many collisions were affected, the highest energy that occurred and which share of the total reaction rate lies above the table - that share is an upper bound of the model error. If it is not negligible, extend the table with SIGMA_ENERGY or extend the cross-section data of that channel.

  At startup the same situation is reported before the run starts, e.g. `SetCollision: s1=... s2=... cross-section table ends at ... eV while the nominal pair energy reaches ... eV`.

- **For the charged species N no absorption is defined in the border with icodec M!**

  This is a convenience message if you forgot to define an absorption of a charged species on a surface. Usually all charged species should react somehow with all surfaces. In the rare case where a surface is not reachable for particles you can ignore this warning.

- **TEX solver divergence: iteration = xxx, residuum = xxx, lastResiduum = xxx, secondLastResiduum = xxx, ConvergenceThreshold = xxx**

78 References

Alcock, C. B., V. P. Itkin, and M. K. Horrigan. 1984. "Vapour Pressure Equations for the Metallic Elements: 298-2500 K." *Canadian Metallurgical Quarterly* 23 (3): 309–13.

Behrisch, R., and W. Eckstein. 2007. *Sputtering by Particle Bombardment*. Edited by R. Behrisch and W. Eckstein. Springer-Verlag Berlin Heidelberg.

Bhokare, V. V., and M. Yussouff. 1974. "Electronic Structures of Beryllium and Titanium by Green's Function Method." *Il Nuovo Cimento* 19B (2): 149–60.

Bird, G. A. 1994. *Molecular Gas Dynamics and the Direct Simulation of Gas Flows*. Vol. 42. Oxford Engineering Science Series.

Birdsall, C. K., and A. B. Langdon. 2005. *Plasma Physics via Computer Simulation*. Taylor & Francis Group.

Chapman, S., and T. G. Cowling. 1970. *The Mathematical Theory of Non-Uniform Gases* - Third Edition.

Geuzaine, C., and J.-F. Remacle. 2009. "Gmsh: A Three-Dimensional Finite Element Mesh Generator with Built-in Pre- and Post-Processing Facilities." *International Journal for Numerical Methods in Engineering* 79 (11): 1309–31. https://doi.org/10.1002/nme.2579.

Jackson, John D. 1999. *Classical Electrodynamics*. John Wiley & Sons.

Langdon, A. B. 1970. "Effects of the Spatial Grid in Simulation Plasmas." *Journal of Computational Physics* 6: 247–67.

McRae, E. G., and C. W. Caldwell. 1976. "Very Low Energy Electron Reflection at Cu(001) Surfaces." *Surface Science* 57: 77–92.

Pflug, A., M. Siemers, T. Melzig, M. Keunecke, L. Schäfer, and G. Bräuer. 2016. "Thin-Film Deposition Processes." In *Handbook of Software Solutions for ICME*, edited by G. J. Schmitz and U. Prahl, 157–89. Wiley Online Library.

Pflug, A., M. Siemers, T. Melzig, D. Rademacher, T. Zickenrott, and M. Vergöhl. 2014. "Numerical Optimization of Baffles for Sputtering Optical Precision Filters." *Surface and Coatings Technology* 241: 45–49. https://doi.org/10.1016/j.surfcoat.2013.11.008.

Pflug, A., M. Siemers, T. Melzig, V. Sittinger, and L. Schäfer. 2015. "Heuristic Modeling of the Doping Efficiency in Sputtered TCO Layers." *Surface and Coatings Technology* 267: 81–89. https://doi.org/10.1016/j.surfcoat.2014.11.046.

Ranjan, R., J. P. Allain, M. R. Hendricks, and D. N. Ruzic. 2001. "Absolute Sputtering Yield of Ti/TiN by Ar+ / n+ at 400-700 eV." *Journal of Vacuum Science & Technology, A: Vacuum, Surfaces, and Films* 19: 1004–7. https://doi.org/10.1116/1.1362678.

Thompson, Michael Warwick. 1981. "Physical Mechanisms of Sputtering." *Physics Reports* 69 (4): 335–71.

Tskhakaya, D., K. Matyash, R. Schneider, and F. Taccogna. 2007. "The Particle-in-Cell Method." *Contributions to Plasma Physics* 47 (8-9): 563–94. https://doi.org/10.1002/ctpp.200710072.

Turner, M. M., A. Derzsi, Z. Donko, D. Eremin, S. J. Kelly, T. Lafleur, and T. Mussenbrock. 2013. "Simulation Benchmarks for Low-Pressure Plasmas: Capacitive Discharges." *Physics of Plasmas* 20: 013507 (12pp).

Wehner, G. K., and D. Rosenberg. 1960. "Angular Distribution of Sputtered Material." *Journal of Applied Physics* 31 / 1: 177–79.

Yamamura, Y., T. Takiguchi, and M. Ishida. 1991. "Energy and Angular Distributions of Sputtered Atoms at Normal Incidence." *Radiation Effects and Defects in Solids* 118 / 3: 237–61.

Zalm, P. C., and L. J. Beckers. 1985. "Ion-Induced Secondary Electron Emission from Copper and Zinc." *Surface Science* 152/153: 135–41.