FEBID Simulator

Release 0.8

mrcheatak

Mar 12, 2023

CONTENTS

CHAPTER

ONE

INTRODUCTION

Simulation of the FEBID process written in Python. FEBID stands for Focused Electron Beam Induced Deposition, a variation of a CVD (chemical vapor deposition) process. It uses volatile organometallic molecules (precursor) as material and a controlled electron beam to selectively deposit material on a substrate.

Simulation base features:

1. Continuum model
2. Enabled diffusion
3. Enabled temperature effects
4. Electron-matter interaction via Monte Carlo simulation
5. No gas dynamics implications

CHAPTER

TWO

INSTALLATION

The simulation requires Python 3.7 or later.

Package is available via PyPi: `pip install febid`

Alternatively, it can be installed directly from GitHub via pip, although that will require compilation of some modules:

`pip install git+https://github.com/MrCheatak/FEBID_py`

Tip: Linux user may need to manually install Tkinter, as it is not always shipped with the default installation of Python.

CHAPTER

THREE

RUNNING THE FIRST SIMULATION

In order to run the first simulation, *Parameters.yml* and *Me3PtCpMe.yml* parameter files in the Examples folder are needed. As the installation finishes, run `python -m febid`, which will show the main control panel:

There are three main setup steps that are essential for the simulation: space, pattern and parameters of the beam and deposition material. Further instructions will configure a simulation on a 200x200 nm substrate with a stationary 5s exposure of a 30keV 0.15nA beam deposition using the Me3PtCpMe precursor.

Space:
Firstly, a simulation volume domain has to be created. The simplest case is a clean substrate. Choose *Parameters* option and specify 200x200x200 nm dimensions with 5 nm cell size and 10 nm substrate. This will define a cubic simulation domain divided into voxels with 5 nm edge length. Finally, a 20 nm layer of substrate material (Au) is laid at the bottom.

Pattern:
Next, pattern has to be defined for the controlled beam. Choose *Simple patterns* and select *Point* from the drop-down menu. This option will fix the beam at a point specified by X and Y parameters, set both of them to 100 to position the beam at the center of the substrate. The time that the beam will spend at that point is defined by *dwell time* and *repeats* parameters. Lets set a 5 s exposure by setting them to 1000 and 5000 correspondingly. A beam is configured now to stay at the (100,100) point for 5 s.

Beam and precursor:
Finally, open *Parameters.yml* for *Settings* and *Me3PtCpMe.yml* for *Precursor parameters*. The first one specifies the beam parameters and precursor flux, the second provides precursor material properties.

Lastly, uncheck all the saving options and enable *Show the process* to watch the growth in real time and hit **Start**.

A new window is then shown with a scene containing the substrate. The scene can be rotated and zoomed to get a better view angle.

Important: Besides the graphical representation, a console is will display simulation info. It is important to keep an eye on it as the deposition progress, execution speed and warnings and errors, if any occur, are output to the console.

CHAPTER

FOUR

MANUAL

4.1 Contents:

4.1.1 Interface

Control panel:

Here is the list of all settings available on the control panel.

Load last session – initially unchecked. Checking it will create a session file at the location, from where the Python command was executed. Lunching from the same location again will load settings used previously. This file can be as well be manually edited to change the settings preset, i.e for a series of simulation runs.

Simulation volume:

- VTK file – allows specifying a VTK-type file (.vtk) that contains a predefined 3D structure to be used in the simulation.
- Parameters – create a fresh simulation volume with specified dimensions and voxel(cell) size with a substrate at the bottom.
- Auto – to be used only when using a stream-file. The dimensions of the simulation volume will be defined automatically to encapsulate the printing path with a sufficient margin.
- Width, length, height – simulation volume dimensions, nm.
- Cell size – edge length of a cubic cell or voxel that is the simulation volume is divided into. The smallest volume fraction of the simulation volume
- Substrate height – the thickness of a substrate layer at the bottom of the simulation volume. By default, it has properties of gold. It should be a multiple of *Cell size*.

  Volume dimensions have to be set only if *Parameters* is chosen. When *VTK file* is chosen, they are set automatically from the file, as well as *Substrate height* and *Cell size*. For the *Auto* option, only *Cell size* and *Substrate height* have to be specified.

Pattern:

- Simple pattern – allows generation of a path with one of the available simple shapes: Available shapes: *point, line, square, rectangle and circle.*
- x, y – parameters of the selected shape. Position for a point, *length* for a line, *edge length* for a square and rectangle, *radius* for a circle. Except for the point, all shapes are placed in the center. Keep in mind, that the printing path should be inside the borders of the simulation volume.
- Pitch – the shape contour is divided into discrete points, which a beam visits in a sequence. This parameter defines the distance between two consequent positions of the beam along it's path.

- Dwell time – the amount of time the beam sits or dwells at a single position.
- Repeats – the number of times the pattern defined by shape, dwell time and pitch has to be repeated.
- Stream file – allows specifying a special stream-file, that defines a more complex printing path. It represents a sequence of beam positions with dwell times. This option requires *Auto* to be chosen in the *Simulation volume* section.
- HFW – Half Field Width sets the scale of the structure. Because pattern files are resolved in pixels, they have to be related to the actual distance units. This relation is provided by the magnification or HFW.

Beam and precursor:

- Settings – a YAML (.yml) file with beam parameters and precursor flux to be specified here.
- Precursor parameters – a YAML (.yml) file with precursor(printing material) and deponat(printed material) properties.
- Temperature tracking – check to enable calculation of the temperature profile and temperature dependency of the precursor coverage.

Warning: Corresponding precursor parameters have to be included in the parameter file in order for the temperature tracking to work.

Note: If a loaded 3D structure does not have temperature profile data, it will be added automatically.

Save file:

- Save simulation data – check to regularly save statistical data of the simulation including time passed, deposition time passed and volume filled. The save interval is specified in the next field.
- Save structure snapshots – check to regularly save the state of the deposition process. The save interval is specified in the next field.

VTK file option:
Read the volume with a structure from a .vtk file. The file can be a regular .vtk file with a structure in it or it can be a file produced by the simulation (by checking Save structure snapshots). If an arbitrary .vtk file is specified, it has to be a UniformGrid, have a cubic cell (equal spacings) and have a single cell array.

Graphical:
When 'Show the process' is checked to view the simulation process in real-time, a window with a 3D scene will open. Refresh rate is set to 0.5 s, thus it may be slow to interact with. The scene is interactive, meaning it can be zoomed by scrolling, rotated with a mouse, moved around (with Shift pressed) and focused at the cursor by pressing 'F'. The coloring and the corresponding scale represents the concentration of the precursor at the surface. Thus, the 3D object displayed is not the solid structure itself, but it's whole surface, that follows the shape of the solid 3D object.

Saving simulation results:

When any of the ‚Save…' options are checked a new folder for the current simulation is created. The intervals of statistics records and snapshots saving refer to the deposition time.

Save simulation data creates an .xlsx Excel file and records simulation setup information and statistical data. Simulation setup is recorded before the simulation start and includes Precursor/deposit properties, Beam/precursor flux settings and Simulation volume attributes, which are saved on separate sheets. Statistical data is then recorded repeatedly during the simulation and includes the following default columns:

- Precise time of record (real)
- Time passed (real), s
- Time passed (deposition/experiment), s
- Current lowest precursor coverage 1/nm ^{2}
- Temperature, K
- Deposited volume, nm ^{3}
- Growth rate

Note: The data collected can be extended via Statistics class by adding columns at the simulation initialization and then providing data for timely records in the monitoring function.

Hint: While real time refers to the real-world time, simulation/experiment refers to the time defined by the beam pattern.

Save structure snapshots enables regular dumping of the current state of structure. The data is saved in .vtk format, and includes 3D arrays that define:

- Grown structure
- Surface deposit
- Surface precursor coverage
- Temperature
- Surface cells
- Semi-surface cells
- Ghost cells

  Additionally, current time, time passed, deposition time passed and beam position are saved.

The files saved via this option can be then viewed as 3D models by the included show_file.py and show_animation.py scripts or in ParaView®.

Warning: 3D structure file (.vtk) may reach 500 Mb for finer grids and, coupled with regular saving with short intervals, may occupy significant disc space. If only the end-result is needed, input an interval that is larger than the total deposition time.

Important: Currently, patterning information is not included in the saved simulation setup info and has to be managed manually.

Viewing simulation results:

There are three options to inspect a 3D structure deposited by FEBID simulation.

The first one is viewing a specific snapshot with all the corresponding data layers (precursor coverage, temperature etc.).

```
python -m febid show_file
```

The second option is to view the process based on a series of structure snapshots. Unlike viewing a single file, only one data layer can be 'animated'.

```
python -m febid show_animation
```

Surface deposit, precursor coverage and temperature profile data are currently supported, it can be set up inside the script.

The third option is to use Paraview®. Examples folder contains a process file, that has all presets for each dataset included in the 3D structure file to render the same result as the *show_file* script.

4.1.2 Experimental settings

An example of a settings file can be found in the Examples folder of the repository.

Beam:

Experiment beam settings:

- **beam_energy** – energy of the electron beam, keV
- **beam_current** – current of the electron beam, A

Modulation of the beam profile:

- **gauss_dev** – standard deviation of a Gaussian beam shape function in nm
- **n** – order of the Gaussian function (see super or higher order gaussian distribution)

Electron trajectory settings:

- **minimum_energy** – energy at which electron trajectory following concludes, keV

Other:

- **precursor_flux** – precursor flux at the surface, 1/(nm^2*s)
- **substrate_element** – material of the substrate, i.e. 'Au'
- **deposition_scaling** – multiplier for deposited volume for artificial speed up of the simulation
- **emission_fraction** – fraction of the total energy lost by primary electrons that is converted to secondary electron emission

4.1.3 Precursor parameters file

An example of a precursor parameters file can be found in the Examples folder of the repository.

Precursor parameters list:

Base parameters:

- **name** – a common name of the selected precursor
- **formula** - a chemical formula of the precursor ,i,e 'Me3PtCpMe'
- **molar_mass_precursor** – molecular mass of the precursor molecule, g/mol
- **max_density** - maximum site density of the precursor, 1/nm^2
- **dissociated_volume** – deposited material volume resulting from dissociation of s single molecule, nm^3
- **sticking_coefficient** – a probability that a precursor molecule adheres to the surface upon collision
- **P_vap**: precursor vapor pressure in the chamber, Pa

Dissociation:

- **cross_section** – precursor molecule integral dissociation cross-section, nm^2

Diffusion:

- **diffusion_coefficient** – surface diffusion coefficient , nm^2/s
- **diffusion_activation_energy*** – activation energy of the diffusion in its Arrhenius equation, eV
- **diffusion_prefactor*** – prefactor in diffusion Arrhenius equation, nm^2/s

Desorption:

- **residence_time** – a mean time a precursor molecule stays on the surface, μs
- **adsorption_activation_energy*** – activation energy of the adsorption in the residence time Arrhenius equation, eV
- **desorption_attempt_frequency*** – a frequency, at which a molecule attempts to desorb from the surface, Hz

Deposit parameters list:

- **deposit** – chemical formula reflecting resulting deposit composition
- **molar_mass_deposit** – molecular mass of the given formula, g/mol
- **SE_emission_activation_energy** – energy required to emit a secondary electron, eV
- **SE_mean_free_path** – secondary electron mean free path nm
- **average_element_number** – average or effective atomic number of the given formula
- **average_element_mol_mass** – average molecular mass of the given formula g/mol
- **average_density** – deposit mass density, g/cm^3
- **thermal_conductivity** – thermal conductivity of the bulk deposit, W/nm/K

* – parameters required for temperature tracking

4.1.4 Setting up a series of simulations

Optimisation of pattern files, simulation input parameters or simulation of several structures may require running a significant number of simulations. The package offers some simple automation features for such tasks. Setting up a simulation series requires composing a Python script.

The first feature allows executing a sequence of simulations arising from consequently changing a single parameter. A series of such simulations is regarded as a *scan*. Such scan can be carried out on any parameter from the Precursor or Settings file.

```
# Initially, a session configuration file has to be specified.
# This file, along settings and precursor parameters files specified in it, is to be␣
↪modified
# and then used to run a simulation. This routine is repeated until the desired parameter
# has taken a given number of values.
# The routine only changes a single parameter. All other parameters have to be preset␣
↪forehand.
session_file = '/home/kuprava/simulations/last_session.yml'

# The first parameter change or scan modifies the Gaussian deviation parameter of the␣
↪beam.
# The file that will be modified in this case is the settings file.
# Set up a folder (it will be created automatically) for simulation save files
directory = '/home/kuprava/simulations/gauss_dev_scan/'
write_param(session_file, 'save_directory', directory)
# Specify parameter name
param = 'gauss_dev'
# Specify values that the parameter will take during consequent simulations
vals = [2, 3, 4, 5, 6, 7, 8]
# Launch the scan
scan_settings(session_file, param, vals, 'hs')
# Files that are saved during the simulation are named after the specified common name␣
↪(here i.e. 'hs')
# and the parameter name.``
```

It is also possible to run a 2D scan, meaning another parameter is scanned for each value of the first parameter.

The second option is to run simulations by using a collection of pattern files. This mode requires that all the desired pattern files are collected in a single folder, that has to be provided to the script.

```
# Again, specify a desired location for simulation save files
directory = '/home/kuprava/simulations/longs/'
# Optionally, an initial structure can be specified. This will 'continue' deposition
# onto a structure obtained in one of the earlier simulations.
# It can be used i.e. when all planned structures share a same initial feature such as a␣
↪pillar.
# Keep in mind that it can be used only for patterning files with the same patterning␣
↪area.
# To that, the patterning area must correspond to one that is defined by the simulation␣
↪for the current
# pattern including margins.
initial_structure = '/home/kuprava/simulations/hockey_stick_therm_050_5_01_15:12:31.vtk'
write_param(session_file, 'structure_source', 'vtk')
write_param(session_file, 'vtk_filename', initial_structure)
write_param(session_file, 'save_directory', directory)
```

(continues on next page)

(continued from previous page)

```
# Specifying a folder with patterning files
stream_files = '/home/kuprava/simulations/steam_files_long_s'
# Launching the series
scan_stream_files(session_file, stream_files)
```

Note: Scanning only modifies the selected parameter(s). Thus, all other parameters as well as saving options and output directory have to be preset.

CHAPTER

FIVE

HOW IT WORKS

This section will explain how various modules work, what solutions are applied and how some of the input parameters are estimated

5.1 Contents:

5.1.1 Monte Carlo module

The Monte Carlo module realises electron beam – matter interaction. There are two results, that are eventually transferred to the deposition module. The first one is secondary electron flux profile, the second is distribution of the volumetric heat sources in the solid or beam heating power.

There are a total of 5 stages that the simulation consists of:

1. Primary electron scattering
2. Secondary electron emission
3. Surface electron flux estimation
4. Primary electron energy deposition
5. Secondary electron energy deposition

1. Primary electron scattering

At this step, scattering of the primary electrons is simulated, resulting in a collection of electron trajectories coupled with energy losses along the trajectory.

Initially, a number of electrons are generated around the beam position according to the Gaussian distribution:

The scattering process occurs in a simulation volume domain of a predefined material.

Each electron initially has the energy of the beam E_0, that is continuously lost as the electron propagates through the solid. The trajectory of an electron consists of a number of consequent scattering points, that are characterised by the electron position and energy. Together, a number of trajectories represent the spacial scattering of the emitted electrons.

At each scattering point, based on the electron energy, the scattering angle and the free path length are calculated based on *random values* from a normal distribution.

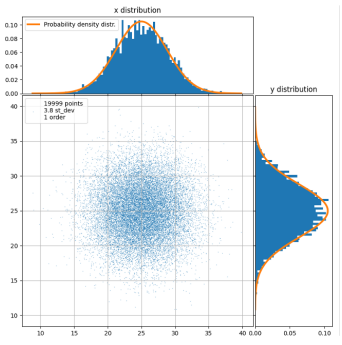


Fig. 1: A total of 20000 generated electrons on a 50x50 grid with 3.8 standard deviation. Histograms reflect equatorial distributions.

After this, the trajectory is extended by an additional segment. The trajectory proceeds likewise until an electron reaches a cut-off energy or escapes the simulation volume domain:

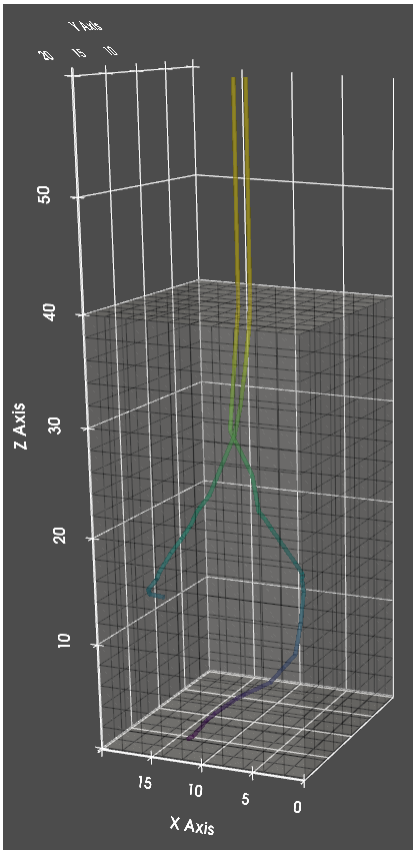


Fig. 3: Two electron trajectories in the simulation volume. Coloring corresponds to electron energy.

2. Electron trajectory discretisation

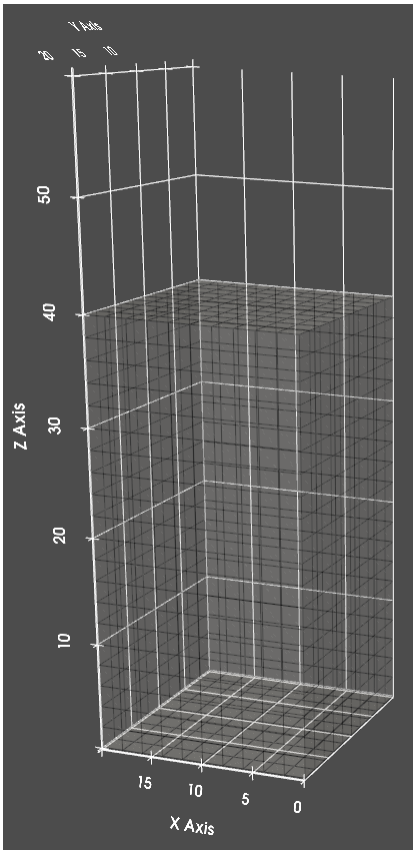
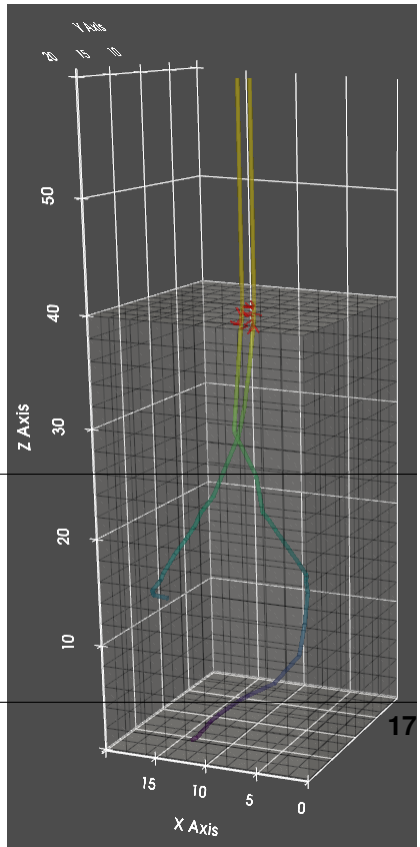


Fig. 2: Simulation volume domain subdivided into cubic cells.

Here, the generated trajectories are subdivided and secondary electrons are emitted based on the energy loss on those subsegments.

Firstly, the trajectories of primary electrons are finely (less then a nm) subdivided into subsegments. Each subsegment corresponds to the energy lost by an electron at this distance E. Based on that energy, the number of emitted secondary electrons is calculated. Electrons are emitted from from the beginning of the subsegment and the emission direction is random. The free path that secondary electrons may travel is fixed for a given material, thus all the emission vectors as assigned the same length. The result at this step is a collection of secondary electron vectors, stemming from primary electron trajectories.

At this stage, those vectors are as well filtered. All SEs that cannot reach the surface due to being buried too deep in the solid are separated from those that have their emission sources in the vicinity of the surface.



3. Surface electron flux estimation

Now, the secondary electron vectors are converted into surface secondary electron flux.

Each vector may or may not reach the surface depending on its position and direction. To test each vector for crossing with the surface, they are followed along and each cell that they traverse through is checked. If a traversed cell appears to be a surface cell, the number of emitted secondary electron that the vector 'carries' is added to that surface cell. Performing such routine on all the vectors results in accumulation of secondary electrons in the surface cells and yields a surface secondary electron flux.

---

4. Beam heating power estimation

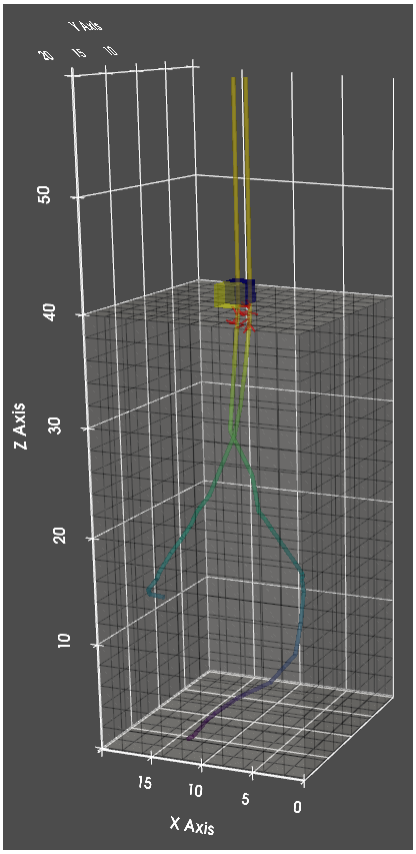
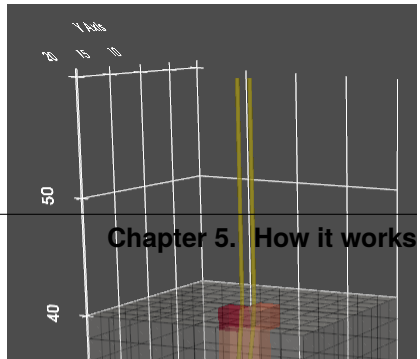


Fig. 5: Surface SE flux, lighter color corresponds to higher flux rate.

Finally, the power of the beam heating is calculated at this step.

The energy of primary electrons is spent as well on Joule heating. Each electron, as it travels through the solid, deposits a fraction of the lost energy into the solid, resulting in heating of the solid medium. Due to the fact that the solid is discretised into cubic cells, the heating power is a collective of cells



traversed by primary electrons with energy deposited in them or a collection of volumetric heat sources.

Each trajectory is followed along to determine the distance traveled inside the cells they traverse. Traversed cells are then added the energy lost by that electron proportional to that distance. This results in a spatially resolved volumetric heat sources distribution, that follow electron trajectories.

In the end, the resulting distribution is added all the secondary electrons, that were buried too deep. Those electrons are considered scattered and contribute to the heating process.

Secondary electron emission energy (ε):

It is the energy required to launch a cascade of secondary electrons. While these values are tabulated for most of the elements in [Lin2005], compound energies shall be averaged volumetrically, i.e. AB compound(amorphous):

$$\overline{\varepsilon} = V_A \cdot \varepsilon_A + V_B \cdot \varepsilon_B,$$

where

V_A and V_B are volume fractions of the phases

ε_A and ε_B are emission activation energies

5.1.2 Diffusion

Surface diffusion plays an important role in precursor coverage replenishment at the beam interaction region (BIR).

In the discretised simulation volume, diffusion occurs on a monolayer of cells that separates solid and empty cell domains. It is the same cell layer that contains information about surface precursor coverage.

Solution of the diffusion equation is a subroutine of the reaction-diffusion equation solution. Each time the solution occurs, it outputs a profile of local precursor changes induced by the diffusion process and then added to the precursor coverage profile.

Ghost cells

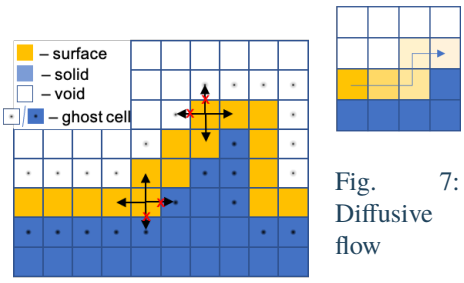


Fig. 7: Diffusive flow

In order to enable diffusion exclusively along the surface a so-called ‘ghost cell’ method is used. The thin layer of surface cells is ultimately contained between the void and solid cells. The solid and void cells that neighbor a surface cell are marked as ghost cells, encapsulating the whole surface. During the application of the stencil operator, the ghost cells mimic the value in the current cell. This artificially creates a condition of zero concentration difference and consequently zero diffusive flux.

Diffusion is described via a parabolic PDE equation and thus requires a special numerical solution.

Characteristic time of the diffusion makes it feasible to use the simplest approach – the FTCS scheme.

Numerical solution

The diffusion equation:

$\frac{\partial T}{\partial t} = D\nabla^2 T,$

where:

D is surface diffusion coefficient $\left[\frac{nm^2}{s}\right]$

T is temperature [K]

which is resolved in 3D space:

$\frac{\partial T}{\partial t} = D\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2}\right)$

The solution occurs in a discretized volume domain, where the finest spacial step along each axis is $\Delta x, \Delta y, \Delta z$. Temperature in each cell is addressed by it's index along each axis *i,j,k*. The FTCS scheme can then be described as follows:

$\frac{\partial T}{\partial t} = D\left(\frac{T_{i-1,j,k}-2T_{i,j,k}+T_{i+1,j,k}}{\Delta x^2} + \frac{T_{i,j-1,k}-2T_{i,j,k}+T_{i,j+1,k}}{\Delta y^2} + \frac{T_{i,j,k-1}-2T_{i,j,k}+T_{i,j,k+1}}{\Delta z^2}\right)$

Partial derivatives are averaged from the current and neighboring cells along a single axis.

For a case with a cubic cell, where $\Delta x = \Delta y = \Delta z$, the expression can be simplified:

$\partial T = D\partial t\left(\frac{T_{i-1,j,k}+T_{i+1,j,k}+T_{i,j-1,k}+T_{i,j+1,k}+T_{i,j,k-1}+T_{i,j,k+1}-6T_{i,j,k}}{\Delta x^2}\right)$

where:

D is surface diffusion coefficient $\left[\frac{nm^2}{s}\right]$

$T_{i,j,k}$ is temperature in the given cell [K]

∂t is the time step or time resolution, [s]

Using the derived expression, the state of the system at the next time step can be derived from the current state.

From analysing the expression, it is evident that it sums every neighbor and subtracts the value of the central cell. Such operation, that is applied to all cells in the same manner is called a *stencil*.

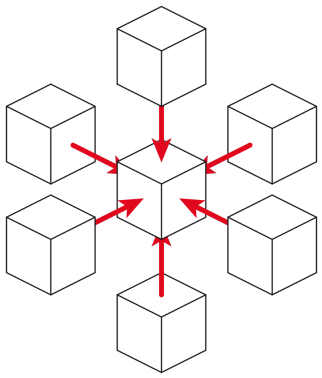


Fig. 8: 3D Stencil operator

The Fourier stability criterion for the FTCS scheme is:

$F = \frac{2D\Delta t}{\Delta x^2},$

for 3D space $F < \frac{1}{6}$ yielding maximum stable time step:

$\Delta t = \frac{\Delta x^2}{6D},$

while semi-implicit Crank-Nicholson and implicit Euler methods are unconditionally stable.

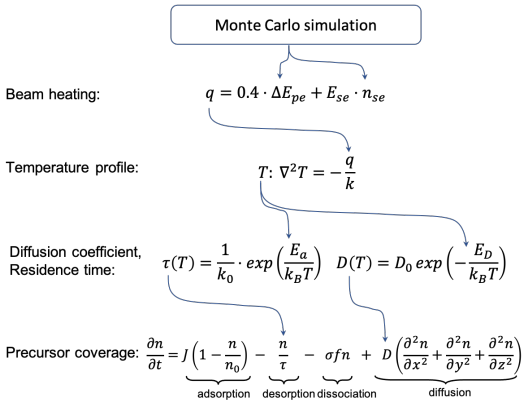
> Nevertheless Crank-Nicholson is unconditionally stable, meaning it works for any time step, it may suffer oscillations. At the same time, implicit Euler is immune to oscillations, but has only 1st-order accuracy in time.

5.1.3 Thermal effects

This page covers the model of beam heating influence on the deposit shape and presents utilised solution for the heat equation.

Temperature dependence

The influence of the beam heating effect through temperature increase is multifold.



Initially, the heating power of the beam q is generated by the Monte Carlo module.

After that, a temperature profile is derived based on q and thermal conductivity of the deposit k

The temperature profile is then used to calculate surface profiles of residence time and diffusion coefficient.

Finally, those profiles are used for the calculation of the precursor coverage profile. Precursor coverage then directly affects the amount of the deposited material.

Heat equation

Both heat distribution and diffusion are described via a parabolic PDE equation and thus require a numerical solution.

Although, the processes are similar in nature, they occur at characteristic time steps differing by orders of magnitude. This fact implies usage of different numerical solution for th heat transfer problem.

In the actual version of the package, the default for heat transfer is SOR.

The heat equation:

$c_p \rho \frac{\partial T}{\partial t} = k \nabla^2 T + q,$

which is resolved in 3D space:

$c_p \rho \frac{\partial T}{\partial t} = k \left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2} \right) + q$

where:

c_p is the heat capacity of the solid medium $\left[\frac{J}{kg \cdot K}\right]$

ρ is the density $\left[\frac{kg}{nm^3}\right]$

k is thermal conductance $\left[\frac{W}{nm \cdot K}\right]$

q is the heating source originating from electron beam heating $\left[\frac{J}{nm^3}\right]$

T is temperature [K]

Due to the fact, that heat transfer characteristic time step is orders of magnitude shorter than one of mass transport (diffusion), the solution of heat equation requires an accordingly shorter time step. Such fine time discretization would make the simulation orders of magnitude slower.

Although, the same feature of the heat transfer means that evolution of an equilibrium or *steady state* occurs almost instantly [Mutunga2019]. It means that time discretization is neglected and the problem simplifies to a calculation of a steady state:

$k\nabla^2 T = -q$

The problem of deriving a steady state is called a relaxation problem and is solved by a family of relaxation methods. Here it is solved via a *Simultaeous Over-Relaxation* (SOR) method. Generally, it represents an FTCS scheme, ultimately applied with the maximum stable time step. The main prerequisition for the SOR method is convergence of the solution. The convergence is evaluated based on a norm of the difference between current and previous iterations. When the norm diminishes below a certain value that is called *solution accuracy* the convergence is reached.

Due to the slow rise of temperature caused by beam heating, a steady state profile can be derived at a significantly lower rate than the diffusion equation is solved.

Effectively, re-calculation of the steady state temperature profile is necessary approximately 10 times per deposition time second for the PtC deposit.

CHAPTER

SIX

FEBID

FEBID Simulator package

febid.Process	Deposition process code
febid.Statistics	Module for continuous process data recording
febid.Structure	Main internal data framework
febid.diffusion	Diffusion module Solution for diffusion equation via FTCS method
febid.febid_core	Control core of the simulation
febid.heat_transfer	Heat transfer module
febid.libraries	Extension modules.
febid.monte_carlo	Monte Carlo electron beam – matter interaction simulation subpackage
febid.simple_patterns	Stream-file reader and pattern generator
febid.start	Scripting template for running series of simulations

6.1 febid.Process

Deposition process code

The Process class implements the methods necessary to support the deposition process.

Functions

restrict	Prevent simultaneous call of the decorated methods

6.1.1 febid.Process.restrict

restrict(*func*)

Prevent simultaneous call of the decorated methods

Classes

Process	Class representing the core deposition process.

6.1.2 febid.Process.Process

class Process(*structure*, *equation_values*, *timings*, *deposition_scaling=1*, *temp_tracking=True*, *name=None*)

Bases: object

Class representing the core deposition process. It contains all necessary arrays, variables, parameters and methods to construct a continuous deposition process.

Methods

check_cells_filled	Check if any deposit cells are fully filled
deposition	Calculate an increment of a deposited volume for all irradiated cells over a time step
diffusion_coefficient	Calculate surface diffusion coefficient for every surface cell.
diffusion_coefficient_expression	Calculate surface diffusion coefficient at a specified temperature.
equilibrate	Bring precursor coverage to a steady state with a given accuracy
get_dt	
heat_transfer	Define heating effect on the process
precursor_density	Calculate an increment of the precursor density for every surface cell
residence_time	Calculate residence time for every surface cell.
residence_time_expression	Calculate residence time at the given temperature :type temp: :param temp: temperature, K :return:
update_helper_arrays	Define new views to data arrays, create axillary indexes and flatten beam_matrix array
update_surface	Updates all data arrays after a cell is filled.
view_dt	

Attributes

deposited_vol	
kd	
kr	
max_temperature	
nd	Calculate depleted precursor coverage
nr	Calculate replenished precursor coverage
precursor_min	

check_cells_filled()

Check if any deposit cells are fully filled

Returns
bool

deposition()

Calculate an increment of a deposited volume for all irradiated cells over a time step

Returns

diffusion_coefficient()

Calculate surface diffusion coefficient for every surface cell.

Returns

diffusion_coefficient_expression(*temp=294*)

Calculate surface diffusion coefficient at a specified temperature.

Parameters
temp – temperature, K

Returns

equilibrate(*eps=0.0001*, *max_it=10000*)

Bring precursor coverage to a steady state with a given accuracy

It is advised to run this method after updating the surface in order to determine a more accurate precursor density value for newly acquired cells

Parameters
eps – desired accuracy

heat_transfer(*heating*)

Define heating effect on the process

Parameters
heating – volumetric heat sources distribution

Returns

property nd

Calculate depleted precursor coverage

Returns

property nr

Calculate replenished precursor coverage

Returns

precursor_density()

Calculate an increment of the precursor density for every surface cell

Returns

residence_time()

Calculate residence time for every surface cell.

Returns

residence_time_expression(*temp=294*)

Calculate residence time at the given temperature :type temp: :param temp: temperature, K :return:

update_helper_arrays()

Define new views to data arrays, create axillary indexes and flatten beam_matrix array

Returns

update_surface()

Updates all data arrays after a cell is filled.

Returns

6.2 febid.Statistics

Module for continuous process data recording

Classes

Statistics	Class implementing statistics gathering and saving(to excel).

6.2.1 febid.Statistics.Statistics

class Statistics(*filename='run_id447920'*)

Bases: object

Class implementing statistics gathering and saving(to excel).

Report contains following columns:

Time, Time passed, Simulation time, Simulation speed, N of cells(filled), Volume, Min.precursor coverage, Growth rate

It is possible to automatically include graphs into Excel files Additionally, initial simulation parameters are added to 3 separate sheets

Methods

add_plot	
add_plots	Add scatter plots to the Excel-file.
add_stat	Add a new statistic to the table.
append	Add a new record to the statistics.
get_growth_rate	
get_params	Collect initial parameters and save them to Excel-file
plot	['Time', 'Sim.time', 'Sim.speed', 'Volume', Min.precursor coverage', 'Growth rate'] :type x: :param x: :type y: :param y: :return:
save_to_file	Write collected statistics to an Excel file.

Attributes

shape

add_plots(**args*, *position='J1'*)

Add scatter plots to the Excel-file.

Args is a list of tuples of column names to be plotted: [(x1, y1), (x2, y2)] Position is a list of cells where to put the graphs (by the upper-left corner)

add_stat(*name*, *first_value=0*)

Add a new statistic to the table. It is recorded in monitoring function and how it is collected is up to the user.

append(**stats*)

Add a new record to the statistics. The number of stats must include manually added ones

Parameters

stats – current simulation time, current number of deposited cells and manually added columns

Returns

get_params(*arg*, *name*)

Collect initial parameters and save them to Excel-file

Parameters

- **arg** (dict) – a dictionary of parameters
- **name** (str) – a name for the provided parameters

Returns

plot(*x*, *y*)

['Time', 'Sim.time', 'Sim.speed', 'Volume', Min.precursor coverage', 'Growth rate'] :type x: :param x: :type y: :param y: :return:

save_to_file(*force=False*)

Write collected statistics to an Excel file. The gathered statistics are appended to the end of the table every couple of seconds Caution: the session keeps the file open until it finishes.

6.3 febid.Structure

Main internal data framework

Classes

Structure	Represents the discretized space of the simulation volume and keeps the current state of the structure.

6.3.1 febid.Structure.Structure

class Structure(*precursor_empty=0.0*, *precursor_full=1.0*, *deposit_empty=0.0*, *deposit_full_substrate=-2.0*, *deposit_full_deponat=-1.0*)

Bases: object

Represents the discretized space of the simulation volume and keeps the current state of the structure.

Set values to mark empty and full cells for precursor and deposit arrays.

Parameters

- **precursor_empty** –
- **precursor_full** –
- **deposit_empty** –
- **deposit_full_substrate** –
- **deposit_full_deponat** –

Methods

create_from_parameters	Frame initializer.
define_ghosts	Determining ghost shell wrapping the surface This is crucial for the diffusion to work if rolling method is used.
define_semi_surface	Determining semi-surface of the initial structure
define_surface	Determining surface of the initial structure
define_surface_neighbors	Find solid cells that are n-closest neighbors to the surface cells.
fill_surface	Covers surface of the deposit with initial precursor density
flush_structure	Resets and prepares initial state of the grid.
load_from_vtk	Frame initializer.
max_z	Get the height of the structure.
resize_structure	Resize the data framework.
save_to_vtk	
update_shape	Read the shape of the data and set shape and absolute shape of the class.

create_from_parameters(*cell_dim=5*, *width=50*, *length=50*, *height=100*, *substrate_height=4*, *nr=0*)

Frame initializer. Create a discretized simulation volume framework from parameters.

Parameters

- **cell_dim** – size of a cell, nm
- **width** – width of the simulation chamber (along X-axis), number of cells
- **length** – length of the simulation chamber (along Y-axis), number of cells
- **height** – height of the simulation chamber (along Z-axis), number of cells
- **substrate_height** – thickness of the substrate along Z-axis, number of cells
- **nr** – initial precursor density, normalized

Returns

define_ghosts()

Determining ghost shell wrapping the surface This is crucial for the diffusion to work if rolling method is used.

Returns

define_semi_surface()

Determining semi-surface of the initial structure

Semi-surface cell is a concept that enables diffusion on the steps of the structure. These cells take part neither in the deposition process, nor in adsorption/desorption.

If semi-surface cell turns into a regular surface cell, the precursor density in it is preserved. :return:

define_surface()

Determining surface of the initial structure

Returns

define_surface_neighbors(*n=0*, *deposit=None*, *surface=None*, *neighbors=None*)

Find solid cells that are n-closest neighbors to the surface cells. If deposit, surface amd neighbors are provided, nearest neighbors are defined for them.

Parameters
n – order of nearest neighbor, if 0, then index all the solid cells

Returns

fill_surface(*nr*)

Covers surface of the deposit with initial precursor density

Parameters
nr (float) – initial precursor density

Returns

flush_structure()

Resets and prepares initial state of the grid.

Parameters

- **substrate** – 3D precursor density array
- **deposit** – 3D deposit array
- **init_density** – initial precursor density on the surface
- **init_deposit** – initial deposit on the surface, can be a 2D array with the same size as deposit array along 0 and 1 dimensions
- **volume_prefill** – initial deposit in the volume, can be a predefined structure in an 3D array same size as deposit array (constant value is virtual and used for code development)

Returns

load_from_vtk(*vtk_obj*, *add_substrate=4*)

Frame initializer. Load structure from a .vtk file.

A vtk object can either represent only a single solid structure array or a result of a deposition process with the full set of arrays.

Important requirement: vtk data must be a UniformGrid with 'spacing' attribute.

Parameters

- **vtk_obj** (DataSet) – a vtk object from file
- **add_substrate** – if a value is specified, a substrate with such height will be created for simple vtk files. 0 or False otherwise. If the value is not a multiple of the 'spacing' attribute, it will be rounded down.

max_z()

Get the height of the structure. :return: 0-axis index of the highest cell

resize_structure(*delta_z=0*, *delta_y=0*, *delta_x=0*)

Resize the data framework. The specified lengths are attached along the axes.

If any of the data is referenced, only a warning is shown and data is resized anyway.

Changing dimensions along y and x axes should be done mindful, because if these require extension in the negative direction, that data has to be centered after the resizing.

Parameters

- **delta_z** – increment from the z-axis, nm
- **delta_y** – increment for the y-axis, nm
- **delta_x** – increment fro the x-axis, nm

Returns

update_shape()

Read the shape of the data and set shape and absolute shape of the class. :return:

6.4 febid.diffusion

Diffusion module Solution for diffusion equation via FTCS method

Functions

diffusion_ftcs	Calculate diffusion term for the surface cells using stencil approach
get_diffusion_stability_time	Get max stable time step for FTCS solution.
laplace_term_stencil	Apply stencil operator to the selected cells in the grid.
prepare_surface_index	Get a multiindex from the surface array.
stencil_debug	

6.4.1 febid.diffusion.diffusion_ftcs

diffusion_ftcs(*grid*, *surface*, *D*, *dt*, *cell_dim*, *surface_index=None*, *flat=True*, *add=0*)

Calculate diffusion term for the surface cells using stencil approach

Nevertheless the 'surface_index' is an optional argument, it is highly recommended to handle index from the caller function

Parameters

- **grid** – 3D precursor density array, normalized
- **surface** – 3D boolean surface array
- **D** – diffusion coefficient, nm^2/s
- **dt** – time interval over which diffusion term is calculated, s
- **cell_dim** – grid space step, nm
- **surface_index** – a tuple of indices of surface cells for the 3 dimensions
- **flat** – if True, returns a flat array of surface cells. Otherwise, returns a 3d array with the same shape as grid.
- **add** – Runge-Kutta intermediate member

Returns

3d or 1d ndarray

6.4.2 febid.diffusion.get_diffusion_stability_time

`get_diffusion_stability_time`(*D*, *dx*)

Get max stable time step for FTCS solution.

Parameters

- **D** – diffusion coefficient, nm/nm^2
- **dx** – grid spacing, nm

Returns
time step, s

6.4.3 febid.diffusion.laplace_term_stencil

`laplace_term_stencil`(*grid*, *surface_index*)

Apply stencil operator to the selected cells in the grid.

Parameters

- **grid** – operated grid
- **surface_index** – selected cell index [z, y, x]

Returns

6.4.4 febid.diffusion.prepare_surface_index

`prepare_surface_index`(*surface*)

Get a multiindex from the surface array.

Parameters
surface (ndarray) – boolean array defining surface cells position in space

Returns
tuple of 1d ndarrays

6.4.5 febid.diffusion.stencil_debug

`stencil_debug`(*grid_out*, *grid*, *z_index*, *y_index*, *x_index*)

6.5 febid.febid_core

Control core of the simulation

Functions

buffer_constants	Calculate necessary constants and prepare parameters for modules
dump_structure	
initialize_framework	Open simulation configuration files and prepare data framework
monitoring	A daemon process function to manage statistics gathering and graphics update.
print_all	Main event loop, that iterates through consequent points in a stream-file.
print_step	Sub-loop, that iterates through the dwell time by a time step
run_febid	Create necessary objects and start the FEBID process.
run_febid_interface	
update_graphical	Update the visual representation of the current process state

6.5.1 febid.febid_core.buffer_constants

buffer_constants(*precursor*, *settings*, *cell_dimension*)

Calculate necessary constants and prepare parameters for modules

Parameters

- **precursor** (dict) – precursor properties
- **settings** (dict) – simulation conditions
- **cell_dimension** (int) – side length of a square cell, nm

Returns

6.5.2 febid.febid_core.dump_structure

dump_structure(*structure*, *sim_t=None*, *t=None*, *beam_position=None*, *filename='FEBID_result'*)

6.5.3 febid.febid_core.initialize_framework

initialize_framework(*from_file=False*, *precursor=None*, *settings=None*, *sim_params=None*, *vtk_file=None*, *geom_params=None*)

Open simulation configuration files and prepare data framework

Parameters

- **from_file** – True to load structure from vtk file
- **precursor** – path to a file with precursor properties
- **settings** – path to a file with beam parameters and settings

- **sim_params** – path to a file with simulation volume parameters
- **vtk_file** – if from_file is True, path to a vtk file to get structure from
- **geom_params** – a list of predetermined simulation volume parameters

Returns

6.5.4 febid.febid_core.monitoring

monitoring(*pr*, *stats=None*, *location=None*, *stats_rate=60*, *dump_rate=60*, *render=False*, *frame_rate=1*, *refresh_rate=0.5*, *displayed_data='precursor'*)

A daemon process function to manage statistics gathering and graphics update.

Parameters

- **pr** (*Process*) – object of the core deposition process
- **stats** (Optional[*Statistics*]) – object for gathering monitoring data
- **location** – file saving directory
- **stats_rate** – statistics recording interval in seconds, None disables statistics recording
- **dump_rate** – dumping interval in seconds, None disables structure dumping
- **render** – True will enable graphical monitoring of the process
- **frame_rate** – redrawing delay
- **refresh_rate** – sleep time

Returns

6.5.5 febid.febid_core.print_all

print_all(*path*, *process_obj*, *sim*)

Main event loop, that iterates through consequent points in a stream-file.

Parameters

- **path** – patterning path from a stream file
- **process_obj** – Process class instance
- **sim** – Monte Carlo simulation object

Returns

6.5.6 febid.febid_core.print_step

print_step(*y*, *x*, *dwell_time*, *pr*, *sim*, *t*)

Sub-loop, that iterates through the dwell time by a time step

Parameters

- **y** – spot y-coordinate
- **x** – spot x-coordinate
- **dwell_time** – time of the exposure

- **pr** (*Process*) – Process object
- **sim** – MC simulation object
- **t** – tqdm progress bar

Returns

6.5.7 febid.febid_core.run_febid

run_febid(*structure*, *precursor_params*, *settings*, *sim_params*, *path*, *temperature_tracking*, *gather_stats=False*, *monitor_kwargs=None*)

Create necessary objects and start the FEBID process.

Parameters

- **structure** – structure object
- **precursor_params** – precursor properties
- **settings** – beam and precursor flux settings
- **sim_params** – simulation volume properties
- **path** – printing path
- **gather_stats** – True enables statistics gathering
- **monitor_kwargs** – settings for the monitoring function

Returns

6.5.8 febid.febid_core.run_febid_interface

run_febid_interface(*structure*, *precursor_params*, *settings*, *sim_params*, *path*, *temperature_tracking*, *saving_params*, *rendering*)

6.5.9 febid.febid_core.update_graphical

update_graphical(*rn*, *pr*, *time_spent*, *displayed_data='precursor'*, *update=True*)

Update the visual representation of the current process state

Parameters

- **rn** (*Render*) – visual scene object
- **pr** (*Process*) – process object
- **time_step** –
- **time_spent** –

Returns

6.6 febid.heat_transfer

Heat transfer module

Functions

fit_exponential	Fit data to an exponential equation y = a*exp(b*x)
fragmentise	Collect columns along each axis that do not contain zero cells
get_heat_transfer_stability_time	Get the largest stable time step for the FTCS scheme.
heat_transfer_BE	Calculate temperature distribution after the specified time step by solving the parabollic heat equation.
heat_transfer_steady_sor	Find steady-state solution to the heat equation with the given accuracy
prepare_solid_index	
subdivide_list	Extract start and end indexes of the non-zero sections in the array.
temperature_stencil	Calculates diffusion term for the surface cells using stencil operator

6.6.1 febid.heat_transfer.fit_exponential

fit_exponential(*x0*, *y0*)

Fit data to an exponential equation y = a*exp(b*x)

Parameters

- **x0** – x coordinates
- **y0** – y coordinates

Returns

ln(a), b

6.6.2 febid.heat_transfer.fragmentise

fragmentise(*grid*)

Collect columns along each axis that do not contain zero cells

Parameters

grid – 3d array

Returns

array of index triples

6.6.3 febid.heat_transfer.get_heat_transfer_stability_time

get_heat_transfer_stability_time(*k*, *rho*, *cp*, *dx*)

Get the largest stable time step for the FTCS scheme.

Parameters

- **k** – thermal conductivity, [W/m/K]
- **rho** – density, [g/cm^3]
- **cp** – heat capacity, [J/kg/K]
- **dx** – grid step (cell size), nm

Returns

time step in seconds

6.6.4 febid.heat_transfer.heat_transfer_BE

heat_transfer_BE(*grid*, *conditions*, *k*, *cp*, *rho*, *dt*, *dl*, *heat_source=0*, *substrate_T=294*)

Calculate temperature distribution after the specified time step by solving the parabollic heat equation.

The heat equation with the heat source term is solved by backward Euler scheme.

Fractional step method is used to numerically solve the PDE in 3D space.

There are two options for boundary conditions:

'isolated': the structure is isolated from both void and substrate

"heatsink': the structure disipates heat through the substarete that has a constant temperature.

6.6.5 febid.heat_transfer.heat_transfer_steady_sor

heat_transfer_steady_sor(*grid*, *k*, *dl*, *heat_source*, *eps*, *solid_index=None*)

Find steady-state solution to the heat equation with the given accuracy

Parameters

- **grid** – 3D temperature array
- **k** – thermal conductivity, [W/K/m]
- **dl** – grid spacing (cell size), nm
- **heat_source** – 3D volumetric heating source array, [W/nm^3]
- **eps** – desired accuracy
- **solid_index** – indexes of solid cells

Returns

3D temperature array

6.6.6 febid.heat_transfer.prepare_solid_index

prepare_solid_index(*grid*)

6.6.7 febid.heat_transfer.subdivide_list

subdivide_list(*grid*, *i=0*, *j=0*, *axis=2*)

Extract start and end indexes of the non-zero sections in the array.

This function virtually prevents zeros from appearing in a solution matrix by extracting the 'solid' cells along the slice.

Parameters

- **grid** – 1D array
- **i** – first index of the current slice
- **j** – second index of the current slice
- **axis** – axis along which the slice was taken

Returns

6.6.8 febid.heat_transfer.temperature_stencil

temperature_stencil(*grid*, *k*, *cp*, *rho*, *dt*, *dl*, *heat_source=0*, *solid_index=None*, *substrate_T=294*, *flat=False*, *add=0*)

Calculates diffusion term for the surface cells using stencil operator

Nevertheless, 'solid_index' is an optional argument, it is highly recommended to handle index from the caller function.

Parameters

- **grid** – 3D temperature array
- **k** – thermal conductivity, [W/K/m]
- **cp** – heat capacity, [J/kg/K]
- **rho** – density, [g/cm^3]
- **dt** – time interval over which diffusion term is calculated, s
- **dl** – grid spacing (cell size), nm
- **heat_source** – 3D volumetric heating source array, [W/nm^3]
- **solid_index** – indexes of solid cells
- **substrate_T** – temperature of the substrate
- **flat** – if True, returns a flat array of surface cells. Otherwise, returns a 3d array with the same shape as grid.
- **add** –

Returns

3d or 1d ndarray

6.7 febid.libraries

Extension modules. Contains electron ray-tracing, stencil and visualisation modules

febid.libraries.pde	

febid.libraries.ray_traversal	Extension modules.
febid.libraries.rolling	Extension modules.
febid.libraries.vtk_rendering	Visualization utilities via Pyvista

6.7.1 febid.libraries.pde

febid.libraries.pde.tridiag	Tridiagonal parallel matrix solver

febid.libraries.pde.tridiag

Tridiagonal parallel matrix solver

Functions

adi_3d	Solve a PDE in 3D uniform domain using ADI method with backward Euler scheme.
adi_3d_indexing	Solve a PDE in 3D domain using ADI method.
tridiag_1d	Tridiagonal matrix solver

febid.libraries.pde.tridiag.adi_3d

adi_3d()

Solve a PDE in 3D uniform domain using ADI method with backward Euler scheme.

Parameters

- **d** – right hand side vector
- **x** – vector to be solved
- **a** – equation coefficient
- **boundaries** – type of boundary conditions: 0 for 0 at boundaries, 1 for fixed boundaries, 2 for no flow through boundaries

febid.libraries.pde.tridiag.adi_3d_indexing

adi_3d_indexing()

Solve a PDE in 3D domain using ADI method. Use provided slices to solve for certain regions.

Parameters

- **d** – right hand side vector
- **x** – vector to be solved
- **s1** – index triples for x-axis, that define a 1d slice
- **s2** – index triples for y-axis, that define a 1d slice
- **s3** – index triples for z-axis, that define a 1d slice
- **a** – equation coefficient, proportional to diffusivity
- **boundaries** – type of boundary conditions: 0 for 0 at boundaries, 1 for fixed boundaries, 2 for no flow through boundaries

febid.libraries.pde.tridiag.tridiag_1d

tridiag_1d()

Tridiagonal matrix solver

The solver uses Thomas algorithm.

Parameters

- **d** – right hand side vector
- **x** – output vector
- **b** – main diagonal value
- **c** – upper and lower diagonal value
- **b0** – boundary value for main diagonal
- **c0** – boundary value for upper and lower diagonals

Returns

6.7.2 febid.libraries.ray_traversal

Extension modules. Contains electron ray-tracing, stencil and visualisation modules

febid.libraries.ray_traversal.traversal

febid.libraries.ray_traversal.traversal

Functions

det_1d	Calculate the length of a vector :param vector: array with 3 elements :return:
det_2d	Calculate the length of vectors in an array
divide_segments	
generate_flux	Wrapper for Cython function.
get_Eloss	
get_alpha_and_lambda	
get_direction	
get_solid_crossing	
get_surface_crossing	
get_surface_solid_crossing	
traverse_segment	Wrapper for Cython function.

febid.libraries.ray_traversal.traversal.det_1d

det_1d(*double[:] vector*) → double

Calculate the length of a vector :param vector: array with 3 elements :return:

febid.libraries.ray_traversal.traversal.det_2d

det_2d(*double[:, :] arr_of_vectors*, *double[:] out*) → void

Calculate the length of vectors in an array

Parameters

- **arr_of_vectors** – array of vectors listed along 0 axis
- **out** – output array, has to be the same length as input's 0 axis

Returns

febid.libraries.ray_traversal.traversal.divide_segments

divide_segments(*double[:] dEs*, *double[:, :] coords*, *int[:] num*, *double[:, :] delta*, *double[:, :] pieces*, *double[:] energies*) → void

febid.libraries.ray_traversal.traversal.generate_flux

generate_flux(*double[:, :, :] flux*, *unsigned char[:, :, :] surface*, *int cell_dim*, *double[:, :] p0*, *double[:, :] pn*, *double[:, :] direction*, *signed char[:, :] index_corr*, *double[:, :] t*, *double[:, :] step_t*, *double[:] n_se*, *int max_count*) → double

Wrapper for Cython function. Generate surface SE flux.

Parameters

- **flux** – array to accumulate SEs
- **surface** – array describing surface
- **cell_dim** – size of a grid cell
- **p0** – starting points
- **pn** – end-points
- **direction** – pointing directions(vectors)
- **t** – arbitrary values to detect crossing
- **step_t** – increments of t value
- **n_se** – number of SEs emitted
- **max_count** – maximum number of crossing events per emission

Returns

total SE yield

febid.libraries.ray_traversal.traversal.get_Eloss

get_Eloss(*double E*, *int Z*, *double rho*, *double A*, *double J*, *double step*) → double

febid.libraries.ray_traversal.traversal.get_alpha_and_lambda

get_alpha_and_lambda(*double E*, *int Z*, *double rho*, *double A) -> (float*, *float*)

febid.libraries.ray_traversal.traversal.get_direction

get_direction(*double ctheta*, *double stheta*, *double psi*, *double cz*, *double cy*, *double cx) -> (float*, *float*, *float*)

febid.libraries.ray_traversal.traversal.get_solid_crossing

get_solid_crossing(*double[:, :, :] grid*, *int cell_dim*, *double[:] p0*, *double[:] direction*, *double[:] t*, *double[:] step_t*, *signed char[:] sign*, *double[:] coord*) → unsigned char

febid.libraries.ray_traversal.traversal.get_surface_crossing

get_surface_crossing(*unsigned char[:, :, :] surface*, *int cell_dim*, *double[:] p0*, *double[:] pn*, *double[:] direction*, *double[:] t*, *double[:] step_t*, *signed char[:] sign*, *double[:] coord*) → void

febid.libraries.ray_traversal.traversal.get_surface_solid_crossing

get_surface_solid_crossing(*unsigned char[:, :, :] surface*, *double[:, :, :] grid*, *int cell_dim*, *double[:] p0*, *double[:] pn*, *double[:] direction*, *double[:] t*, *double[:] step_t*, *signed char[:] sign*, *double[:] coord*, *double[:] coord1*) → unsigned char

febid.libraries.ray_traversal.traversal.traverse_segment

traverse_segment(*double[:, :, :] energies*, *double[:, :, :] grid*, *int cell_dim*, *double[:, :] p0*, *double[:, :] pn*, *double[:, :] direction*, *double[:, :] t*, *double[:, :] step_t*, *double[:] dEs*, *int max_count*) → double

Wrapper for Cython function. Deposits energies to the structure based on the energy losses.

Parameters

- **L** – distances between segment points
- **cell_dim** – size of a cell
- **dEs** – energies lost on segments
- **direction** – segment pointing direction
- **energies** – structured array of deposited energies
- **grid** – surface array
- **p0** – starting points of segments
- **pn** – c of segments
- **step_t** – increments of t value
- **t** – arbitrary values to detect crossing
- **N** – number of segments

Returns

total deposited energy

6.7.3 febid.libraries.rolling

Extension modules. Contains electron ray-tracing, stencil and visualisation modules

febid.libraries.rolling.roll

febid.libraries.rolling.roll

Functions

rolling_1d	
rolling_2d	Analog of the np.roll for 2d arrays :param arr: array to add to :param brr: addition
rolling_3d	Analog of the np.roll for 3d arrays :param arr: array to add to :param brr: addition
stencil	Stencil operator.
stencil_gs	Stencil operator.
stencil_sor	Stencil operator.
surface_temp_av	Define temperature of the surface cells by averaging temperature of the neighboring solid cells

febid.libraries.rolling.roll.rolling_1d

rolling_1d()

febid.libraries.rolling.roll.rolling_2d

rolling_2d()

Analog of the np.roll for 2d arrays :param arr: array to add to :param brr: addition

Returns

febid.libraries.rolling.roll.rolling_3d

rolling_3d()

Analog of the np.roll for 3d arrays :param arr: array to add to :param brr: addition

Returns

febid.libraries.rolling.roll.stencil

stencil()

Stencil operator. Sums all the neighbors to the current cell. If a neighbor is 0 or out of the bounds, then adds cell's current value to itself. Arrays must have the same shape. :param grid_out: operated array :param grid: source array :param z: first array index :param y: second array index :param x: third array index :return:

febid.libraries.rolling.roll.stencil_gs

stencil_gs()

Stencil operator. Sums all the neighbors to the current cell. If a neighbor is 0 or out of the bounds, then adds cell's current value to itself. Arrays must have the same shape. :param grid: operated array :param s: power source array :param w: over-relaxation parameter :param z_index: first array index :param y_index: second array index :param x_index: third array index :return:

febid.libraries.rolling.roll.stencil_sor

stencil_sor()

Stencil operator. Sums all the neighbors to the current cell. If a neighbor is 0 or out of the bounds, then adds cell's current value to itself. Arrays must have the same shape. :param grid: operated array :param s: power source array :param w: over-relaxation parameter :param z_index: first array index :param y_index: second array index :param x_index: third array index :return:

febid.libraries.rolling.roll.surface_temp_av

surface_temp_av()

Define temperature of the surface cells by averaging temperature of the neighboring solid cells

Parameters

- **surface_temp** – surface temperature array
- **temp** – solid temperature array
- **z** – first array index
- **y** – second array index
- **x** – third array index

Returns

6.7.4 febid.libraries.vtk_rendering

Visualization utilities via Pyvista

febid.libraries.vtk_rendering.VTK_Rendering	Core visualization module
febid.libraries.vtk_rendering.show_animation_new	View series of consequent 3D-Structure files as an animated process.
febid.libraries.vtk_rendering.show_file	View the 3D-structure files produced by the simulation.

febid.libraries.vtk_rendering.VTK_Rendering

Core visualization module

Functions

export_obj	Export deposited structure as an .obj file
numpy_to_vtk	Convert numpy array to a VTK-datastructure (UniformGrid or UnstructuredGrid).
read_field_data	Read run time, simulation time and beam position from vtk-file.
save_deposited_structure	Save current deposition result to a vtk file.

febid.libraries.vtk_rendering.VTK_Rendering.export_obj

export_obj(*structure*, *filename=None*)

Export deposited structure as an .obj file

Parameters

- **structure** – Structure class instance, must have 'deposit' array and 'cell_dimension' value
- **filename** – full path with file name

Returns

febid.libraries.vtk_rendering.VTK_Rendering.numpy_to_vtk

numpy_to_vtk(*arr*, *cell_dim*, *data_name='scalar'*, *grid=None*, *unstructured=False*)

Convert numpy array to a VTK-datastructure (UniformGrid or UnstructuredGrid). If grid is provided, add new dataset to that grid.

Parameters

- **arr** – numpy array
- **cell_dim** – array cell (cubic) edge length
- **data_name** – name of data
- **grid** – existing UniformGrid
- **unstructured** – if True, return an UnstructuredGrid

Returns

febid.libraries.vtk_rendering.VTK_Rendering.read_field_data

read_field_data(*vtk_obj*)

Read run time, simulation time and beam position from vtk-file.

Parameters
vtk_obj – VTK-object (UniformGrid)

Returns

febid.libraries.vtk_rendering.VTK_Rendering.save_deposited_structure

save_deposited_structure(*structure*, *sim_t=None*, *t=None*, *beam_position=None*, *filename=None*)

Save current deposition result to a vtk file. If filename does not contain path, saves to the current directory.

Parameters

- **structure** – an instance of the current state of the process
- **sim_t** – simulation time, s
- **t** – run time
- **beam_position** – (x,y) current position of the beam
- **filename** – full file name

Returns

Classes

Render	Class implementing rendering utilities for visualizing of Numpy data using Pyvista

febid.libraries.vtk_rendering.VTK_Rendering.Render

class Render(*cell_dim*, *font=12*, *button_size=25*)

Bases: object

Class implementing rendering utilities for visualizing of Numpy data using Pyvista

Parameters

- **cell_dim** (int) – cell data spacing for VTK objects
- **font** – button caption font size
- **button_size** – size of the show on/off button

Methods

save_3Darray	Dump a Numpy array to a vtk file with a specified name and creation date
show	Shows plotting scene
show_full_structure	Render and plot all the structure components
show_mc_result	
update	Update the plot
update_mask	

class SetVisibilityCallback(*actor*)

Bases: object

Helper callback to keep a reference to the actor being modified. This helps button show and hide plot elements

__call__(*state*)

Call self as a function.

save_3Darray(*filename*, *arr*, *data_name='scalar'*)

Dump a Numpy array to a vtk file with a specified name and creation date

Parameters

- **filename** – distinct name of the file
- **arr** – array to save
- **data_name** – name of the data to include in the vtk dataset

Returns

show(*screenshot=False*, *show_grid=True*, *keep_plot=False*, *interactive_update=False*, *cam_pos=None*)

Shows plotting scene

Parameters

- **screenshot** – if True, a screenshot of the scene will be saved upon showing
- **show_grid** – indicates axes and scales
- **keep_plot** – if True, creates a copy of current Plotter before showing
- **interactive_update** – if True, code execution does not stop while scene window is opened
- **cam_pos** – camera view

Returns

current camera view

show_full_structure(*structure*, *struct=True*, *deposit=True*, *precursor=True*, *surface=True*, *semi_surface=True*, *temperature=True*, *ghosts=True*, *t=None*, *sim_time=None*, *beam=None*, *cam_pos=None*)

Render and plot all the structure components

Parameters

- **structure** (*Structure*) – data object

- **struct** – if True, plot solid structure
- **deposit** – if True, plot deposit on the surface
- **precursor** – if True, plot precursor surface density
- **surface** – if True, color all surface cells
- **semi_surface** – if True, color all semi_surface cells
- **ghosts** – if True, color ghost cells

Returns

update(*time=1*, *force_redraw=False*)

Update the plot

Parameters

- **time** – minimum time before each subsequent update
- **force_redraw** – redraw the plot immediately

Returns

febid.libraries.vtk_rendering.show_animation_new

View series of consequent 3D-Structure files as an animated process.

Functions

open_file	Gather files and timestamps sorted in the order of creation
show_animation	Show animated process from series of vtk files.

febid.libraries.vtk_rendering.show_animation_new.open_file

open_file(*directory=''*)

Gather files and timestamps sorted in the order of creation

Parameters

directory – folder with vtk files

Returns

filenames and timestamps

febid.libraries.vtk_rendering.show_animation_new.show_animation

show_animation(*directory=''*, *show='precursor'*)

Show animated process from series of vtk files. Files must have consequent creation dates to align correctly

Parameters

- **directory** – folder with vtk files
- **show** – which dataset to use for imaging. Accepts 'precursor' for surface precursor density or 'deposit' for surface deposit filling.

Returns

febid.libraries.vtk_rendering.show_file

View the 3D-structure files produced by the simulation.

Functions

show_structure

febid.libraries.vtk_rendering.show_file.show_structure

show_structure(*filenames*, *solid=True*, *deposit=True*, *precursor=True*, *surface=True*, *semi_surface=True*, *ghost=True*)

6.8 febid.monte_carlo

Monte Carlo electron beam – matter interaction simulation subpackage

febid.monte_carlo.compiled

febid.monte_carlo.etraj3d	Monte Carlo simulation main module
febid.monte_carlo.etrajectory	Primary electron trajectory simulator
febid.monte_carlo.etrajmap3d	Electron-matter interaction simulator
febid.monte_carlo.mc_base	Monte Carlo simulator utility module

6.8.1 febid.monte_carlo.compiled

febid.monte_carlo.compiled.etrajectory_c

febid.monte_carlo.compiled.etrajectory_c

Functions

get_materials
start_sim

febid.monte_carlo.compiled.etrajectory_c.get_materials

get_materials()

febid.monte_carlo.compiled.etrajectory_c.start_sim

start_sim()

Classes

BuffVector
Electron
SimulationVolume

febid.monte_carlo.compiled.etrajectory_c.BuffVector

class BuffVector
Bases: object

Methods

febid.monte_carlo.compiled.etrajectory_c.Electron

class Electron
Bases: object

Methods

febid.monte_carlo.compiled.etrajectory_c.SimulationVolume

class SimulationVolume
Bases: object

Methods

6.8.2 febid.monte_carlo.etraj3d

Monte Carlo simulation main module

Functions

run_mc_simulation	Create necessary objects and run the MC simulation

febid.monte_carlo.etraj3d.run_mc_simulation

run_mc_simulation(*structure*, *E0=20*, *sigma=5*, *n=1*, *N=100*, *pos='center'*, *precursor='Au'*, *Emin=0.1*, *emission_fraction=0.6*, *heating=False*, *params={}*, *cam_pos=None*)

Create necessary objects and run the MC simulation

Parameters

- **structure** –
- **E0** –
- **sigma** –
- **N** –
- **pos** –
- **precursor** –
- **Emin** –

Returns

Classes

MC_Simulation	Monte Carlo simulation main class

febid.monte_carlo.etraj3d.MC_Simulation

class MC_Simulation(*structure*, *mc_params*)

Bases: *MC_Sim_Base*

Monte Carlo simulation main class

Methods

plot	Show the structure with surface electron flux and electron trajectories
plot_flux_2d	
run_simulation	Run MC simulation with the beam coordinates
update_structure	Renew memory addresses of the arrays

Attributes

shape
shape_abs

plot(*primary_e=True*, *secondary_flux=True*, *secondary_e=False*, *heat_total=False*, *heat_pe=False*, *heat_se=False*, *timings=(None, None, None)*, *cam_pos=None*)

Show the structure with surface electron flux and electron trajectories

Returns

run_simulation(*y0*, *x0*, *heat*, *N=None*)

Run MC simulation with the beam coordinates

Parameters

- **y0** – spot y-coordinate
- **x0** – spot x-coordinate
- **heat** – if True, calculate beam heating

Returns
SE surface flux

update_structure(*structure*)

Renew memory addresses of the arrays

Parameters
structure –

Returns

6.8.3 febid.monte_carlo.etrajectory

Primary electron trajectory simulator

Classes

ETrajectory	A class responsible for the generation and scattering of electron trajectories
Electron	A class representing a single electron with its properties and methods to define its scattering vector.

febid.monte_carlo.etrajectory.ETrajectory

class ETrajectory

Bases: *MC_Sim_Base*

A class responsible for the generation and scattering of electron trajectories

Methods

get_crossing_point	
get_next_crossing	Get next two crossing points and a flag showing if volume boundaries are met
get_norm_factor	Calculate norming factor with the given number of generated trajectories
map_trajectory	Simulate trajectory of the electrons with a specified starting position.
map_trajectory_verbose	Simulate trajectory of the electrons with a specified starting position.
map_wrapper	Create normally distributed electron positions and run trajectory mapping
map_wrapper_cy	Create normally distributed electron positions and run trajectory mapping in Cython
plot_distribution	Plot a scatter plot of the (x,y) points with 2D histograms depicting axial distribution
rnd_gauss_xy	Generate a specified number of points according to a Gaussian distribution.
rnd_super_gauss	Generate a specified number of points according to a Super Gaussian distribution.
save_passes	Save passes to a text file or by pickling
setParameters	Initialise the instance and set all the necessary parameters

Attributes

shape
shape_abs

get_next_crossing(*coords*)

Get next two crossing points and a flag showing if volume boundaries are met

Parameters
coords –

Returns

get_norm_factor(*N=None*)

Calculate norming factor with the given number of generated trajectories

Parameters
N – number of trajectories

Returns

map_trajectory(*x0*, *y0*)

Simulate trajectory of the electrons with a specified starting position.

Parameters

- **x0** – x-positions of the electrons
- **y0** – y-positions of the electrons

Returns

map_trajectory_verbose(*x0*, *y0*)

Simulate trajectory of the electrons with a specified starting position. Version with step-by-step output to console.

Parameters

- **x0** – x-positions of the electrons
- **y0** – y-positions of the electrons

Returns

map_wrapper(*y0*, *x0*, *N=0*)

Create normally distributed electron positions and run trajectory mapping

Parameters

- **y0** – y-position of the beam, nm
- **x0** – x-position of the beam, nm
- **N** – number of electrons to create

Returns

map_wrapper_cy(*y0*, *x0*, *N=0*)

Create normally distributed electron positions and run trajectory mapping in Cython

Parameters

- **y0** – y-position of the beam, nm
- **x0** – x-position of the beam, nm
- **N** – number of electrons to create

Returns

plot_distribution(*x*, *y*, *func=None*)

Plot a scatter plot of the (x,y) points with 2D histograms depicting axial distribution

Parameters

- **x** – array of x-coordinates
- **y** – array of y-coordinates
- **func** – 2D probability density function

Returns

rnd_gauss_xy(*x0*, *y0*, *N*)

Generate a specified number of points according to a Gaussian distribution. Standard deviation and order of the super gaussian are class properties.

Parameters

- **x0** – mean along X-axis
- **y0** – mean along Y-axis
- **N** – number of points to generate

Returns
two arrays of N-length with x and y positions

rnd_super_gauss(*x0*, *y0*, *N*)

Generate a specified number of points according to a Super Gaussian distribution. Standard deviation and order of the super gaussian are class properties.

Parameters

- **x0** – mean along X-axis
- **y0** – mean along Y-axis
- **N** – number of points to generate

Returns
two arrays of N-length with x and y positions

save_passes(*fname*, *type*)

Save passes to a text file or by pickling

Parameters

- **fname** – name of the file
- **type** – saving type: accepts ‘pickle’ or ‘text’

Returns

setParameters(*structure*, *params*, *stat=1000*)

Initialise the instance and set all the necessary parameters

Parameters

- **structure** – solid structure representation
- **params** – contains all input parameters for the simulation
- **stat** – number of simulated trajectories

febid.monte_carlo.etrajectory.Electron

class Electron(*x*, *y*, *parent*)

Bases: object

A class representing a single electron with its properties and methods to define its scattering vector.

Methods

check_boundaries	Check if the given (z,y,x) position is inside the simulation chamber.
get_direction	
get_next_point	
index_corr	Corrects indices according to the direction if coordinates are on the cell wall

Attributes

coordinates	Current coordinates (z, y, x)
coordinates_prev	Previous coordinates (z, y, x)
direction	
indices	Gets indices of a cell in an array according to its position in the space

check_boundaries(*z=0*, *y=0*, *x=0*)

Check if the given (z,y,x) position is inside the simulation chamber. If bounds are crossed, return corrected position

Parameters

- **z** –
- **y** –
- **x** –

Returns

property coordinates

Current coordinates (z, y, x)

Returns
tuple

property coordinates_prev

Previous coordinates (z, y, x)

Returns
tuple

index_corr()

Corrects indices according to the direction if coordinates are on the cell wall

Returns

property indices

Gets indices of a cell in an array according to its position in the space

Returns
i(z), j(y), k(x)

6.8.4 febid.monte_carlo.etrajmap3d

Electron-matter interaction simulator

Functions

process_trajectories	Convert raw trajectories into a collection of segments

febid.monte_carlo.etrajmap3d.process_trajectories

process_trajectories(*points*, *energies*, *mask*)

Convert raw trajectories into a collection of segments

Parameters

- **points** – consequent scattering points
- **energies** – remaining energy at each point
- **mask** – marks segments that lie outside of solid

Returns
an array of start- and end-points of segments, energy loss at segment

Classes

ETrajMap3d	Implements energy deposition and surface secondary electron flux calculation.

febid.monte_carlo.etrajmap3d.ETrajMap3d

class ETrajMap3d

Bases: *MC_Sim_Base*

Implements energy deposition and surface secondary electron flux calculation.

Create an empty ETrajMap3d instance

Methods

extract_se_heat	Calculate energy loss by scattered secondary electrons per cell.
follow_segment	Calculate total energy deposited by primary electrons per cell.
generate_se	Estimate surface secondary electron flux.
joule_heating	Get total energy loss from primary and secondary electrons peel
map_follow	Get surface secondary electron flux and volumetric heat source distribution
prep_se_emission	Subdivide trajectory segments and energy losses
setParametrs	Initialise the instance and set all the necessary parameters
traverse_cells	AABB Ray-Voxel traversal algorithm.

Attributes

shape
shape_abs

extract_se_heat()

Calculate energy loss by scattered secondary electrons per cell.

Returns

follow_segment(*points*, *dEs*)

Calculate total energy deposited by primary electrons per cell.

Parameters

- **points** – array of (z, y, x) points representing a trajectory from MC simulation
- **dEs** – list of energies losses between consecutive points. dEs[0] corresponds to a loss between p[0] and p[1]

Returns

generate_se()

Estimate surface secondary electron flux.

Returns

joule_heating()

Get total energy loss from primary and secondary electrons peel

map_follow(*passes*, *heating=False*)

Get surface secondary electron flux and volumetric heat source distribution
from primary electron trajectories.

Parameters

- **passes** – a collection of trajectories
- **heating** – True will calculate collective heat effect from PEs and SEs

Returns

prep_se_emission(*points*, *dEs*, *ends*)

Subdivide trajectory segments and energy losses

Parameters

- **points** – segment start- and end-points
- **dEs** – energy loss
- **ends** – trajectory end positions, check comments

Returns

setParametrs(*structure*, *params*, *segment_min_length=0.3*)

Initialise the instance and set all the necessary parameters

Parameters

- **structure** – solid structure representation
- **params** – contains all input parameters for the simulation
- **segment_min_length** – segment subdivision length

traverse_cells(*p0*, *pn*, *direction*, *t*, *step_t*)

AABB Ray-Voxel traversal algorithm. Gets coordinates, where ray crosses voxel walls

Parameters

- **p0** – ray origin
- **pn** – ray endpoint
- **direction** – direction of the ray
- **t** – first t-value
- **step_t** – step of the t-value

Returns

6.8.5 febid.monte_carlo.mc_base

Monte Carlo simulator utility module

Classes

Element	Represents a solid material.
MC_Sim_Base	

febid.monte_carlo.mc_base.Element

class Element(*name='noname'*, *Z=1*, *A=1.0*, *rho=1.0*, *e=50*, *lambda_escape=1.0*, *mark=1*)

Bases: object

Represents a solid material. Contains properties necessary for electron beam-matter interaction.

Methods

febid.monte_carlo.mc_base.MC_Sim_Base

class MC_Sim_Base(**args*)

Bases: ABC

Methods

Attributes

shape
shape_abs

6.9 febid.simple_patterns

Stream-file reader and pattern generator

Functions

analyze_pattern	Parse stream-file and split it into stages
generate_circle	
generate_line	
generate_pattern	Generate a stream-file for a simple figure.
generate_point	
generate_square	
open_stream_file	Open stream file, convert to nm and define enclosing volume dimensions.

6.9.1 febid.simple_patterns.analyze_pattern

analyze_pattern(*file*, *unit_pitch*)

Parse stream-file and split it into stages

6.9.2 febid.simple_patterns.generate_circle

generate_circle(*loops*, *dwell_time*, *x*, *y*, *diameter*, *_*, *step=1*)

6.9.3 febid.simple_patterns.generate_line

generate_line(*loops*, *dwell_time*, *x*, *y*, *line*, *_*, *step=1*)

6.9.4 febid.simple_patterns.generate_pattern

generate_pattern(*pattern*, *loops*, *dwell_time*, *x*, *y*, *params*, *step=1*)

Generate a stream-file for a simple figure.

Parameters

- **pattern** – name of a shape: point, line, square, rectangle, circle
- **loops** – amount of passes
- **dwell_time** – time spent on each point, s
- **x** – center x position of the figure, nm
- **y** – center y position of the figure, nm
- **params** – figure parameters, nm; (length) for line, (diameter) for circle, (edge length) for cube
- **step** – distance between each point, nm

Returns

array(x positions[nm], y positions[nm], dwell time[s])

6.9.5 febid.simple_patterns.generate_point

generate_point(*loops*, *dwell_time*, *x*, *y*)

6.9.6 febid.simple_patterns.generate_square

generate_square(*loops*, *dwell_time*, *x*, *y*, *side_a*, *side_b=None*, *step=1*)

6.9.7 febid.simple_patterns.open_stream_file

open_stream_file(*file=None*, *offset=200*, *collapse=False*, *unit_pitch=0.13*)

Open stream file, convert to nm and define enclosing volume dimensions.
A valid stream-file should consist of 3 columns and start with 's16' line.

Parameters

- **file** – path to the stream-file
- **offset** – determines a margin around the printing path
- **collapse** – if True, summ dwell time of consecutive instructions with identical coordinates

Returns
normalized directives in nm and s, dimensions of the enclosing volume in nm

6.10 febid.start

Scripting template for running series of simulations

Functions

atoi	
extr_number	
read_param	Read a parameter value from a configuration file.
scan_settings	Launch a series of simulations by changing a single parameter
scan_stream_files	Launch a series of simulations using multiple patterning files
start_default	
start_no_ui	
start_ui	
write_param	Write a value to a parameter in a configuration file.

6.10.1 febid.start.atoi

atoi(*text*)

6.10.2 febid.start.extr_number

extr_number(*text*)

6.10.3 febid.start.read_param

read_param(*file*, *param_name*)

Read a parameter value from a configuration file.

Parameters

- **file** – path to configuration file
- **param_name** – name of the parameter

Returns

value of the parameter

6.10.4 febid.start.scan_settings

scan_settings(*session_file*, *param_name*, *scan*, *base_name=''*)

Launch a series of simulations by changing a single parameter

Parameters

- **session_file** – YAML file with session configuration
- **param_name** – the name of the parameter, refer to settings and precursor parameters
- **scan** – a collection of values to use in consequent runs
- **base_name** – a common name for simulation files

Returns

6.10.5 febid.start.scan_stream_files

scan_stream_files(*session_file*, *directory*)

Launch a series of simulations using multiple patterning files

The files are named after the patterning file :type session_file: :param session_file: YAML file with session configuration :type directory: :param directory: folder with stream files :return:

6.10.6 febid.start.start_default

start_default(*config_f=None*)

6.10.7 febid.start.start_no_ui

start_no_ui(*config_f=None*)

6.10.8 febid.start.start_ui

start_ui(*config_f=None*)

6.10.9 febid.start.write_param

write_param(*file*, *param_name*, *val*)

Write a value to a parameter in a configuration file.

Parameters

- **file** – path to configuration file
- **param_name** – name of the parameter
- **val** – value to write

Returns

CHAPTER

SEVEN

WELCOME TO FEBID SIMULATION DOCUMENTATION!

The package is virtual representation of the direct-write nanofabrication technique called FEBID driven by an electron beam that typically takes place in a SEM. The simulation takes in a handful of parameters and allows prediction of the deposit shape expected from an experiment. It features a live visual process representation, periodical save of the current state of the 3D deposited structure and recording of the process parameters like time and growth rate. Additionally, the package features an electron beam - matter simulator, that can be run separately using a previously saved 3D structure to reveal beam related details of the process.

The saved 3D structure files can then be interactively viewed or compiled into a animated series depicting the process.

The Getting started section will let you quickly install the package and run an example simulation. A more detailed interface manual, input parameter files explanation and features list will give a full understanding on how to use the simulation.

For more in-deep understanding of the simulation design and code details check the API section.

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[Lin2005] Lin Y., Joy D., A new examination of secondary electron yield data, Surf. Interface Anal. 2005, 37, 895–900

[Mutunga2019] Mutunga E., Winkler R., Sattelkow J. et al., Impact of Electron-Beam Heating during 3D Nanoprinting, ACS Nano 2019, 13, 5198-5213

PYTHON MODULE INDEX

f

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

W