ebisim Documentation

Release 0.2.0

Hannes Pahl

May 02, 2023

CONTENT:

The ebisim package is being devleoped to provide a collection of tools for simulating the evolution of the charge state distribution inside an Electron Beam Ion Source / Trap (EBIS/T) using Python.

This documentation contains a few *examples* demonstrating the general features of ebisim. For a detailed description of the included modules please refer to the *API reference*.

CHAPTER

ONE

EXAMPLES

This section contains small examples showcasing some features of ebisim.

1.1 Looking at cross sections

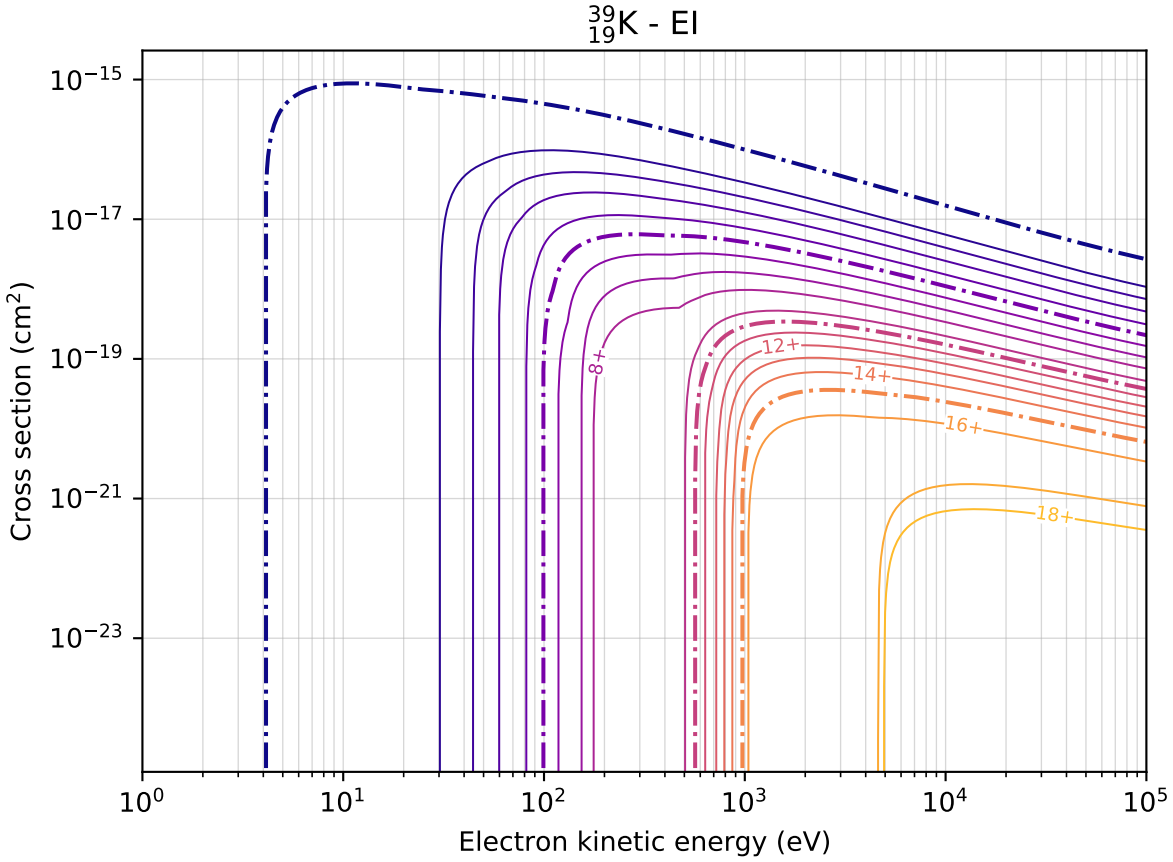
The following code demonstrates how to produce plots showing the cross sections for important ionisation and recombination processes.

```
"""Example: Plotting cross sections"""

from matplotlib.pyplot import show
import ebisim as eb

# The cross section plot commands accept a number of formats for the element parameter
# This example shows the different possibilities

# The first option is to provide an instance of the Element class
potassium = eb.get_element("Potassium")

# This command produces the cross section plot for electron impact ionisation
eb.plot_eixs(element=potassium)

# If no Element instance is provided, the plot command will generate one internally based
# on the provided specifier

# This command produces the cross section plot for radiative recombination
eb.plot_rrxs(element="Potassium")  # Based on name of element

# This command produces the cross section plot for dielectronic recombination
# In addition to the Element the effective line width (eV) has to be specified.
# Typically the natural line width of a DR transition is much smaller than the energy␣
↪spread
# of the electron beam, therefore a gaussian profile with a given line width is assumed␣
↪for
# the transitions.
eb.plot_drxs(element="K", fwhm=15)  # Based on element symbol

# It is also possible to compare all cross sections in a single plot
eb.plot_combined_xs(element=19, fwhm=15, xlim=(2200, 3000))  # Based on proton number
```

(continues on next page)

(continued from previous page)

```
show()
```



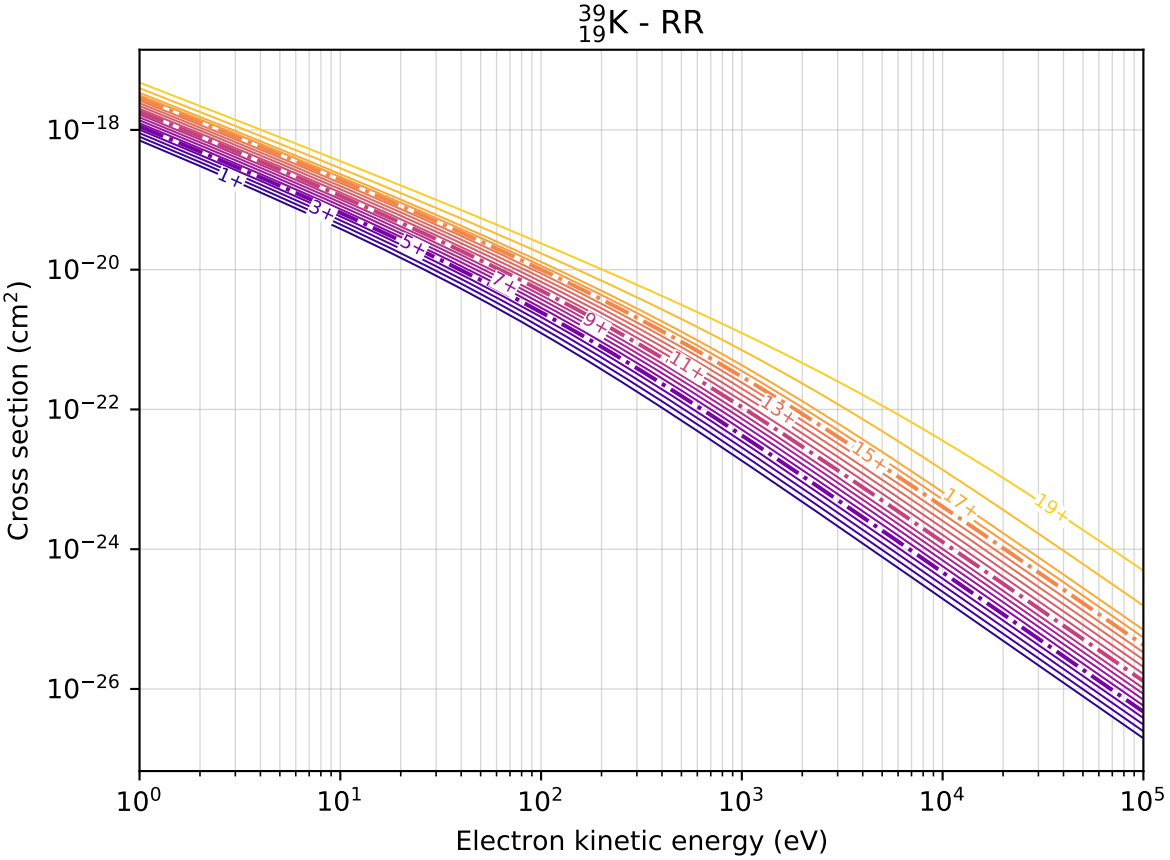
1.2 Running a basic simulation

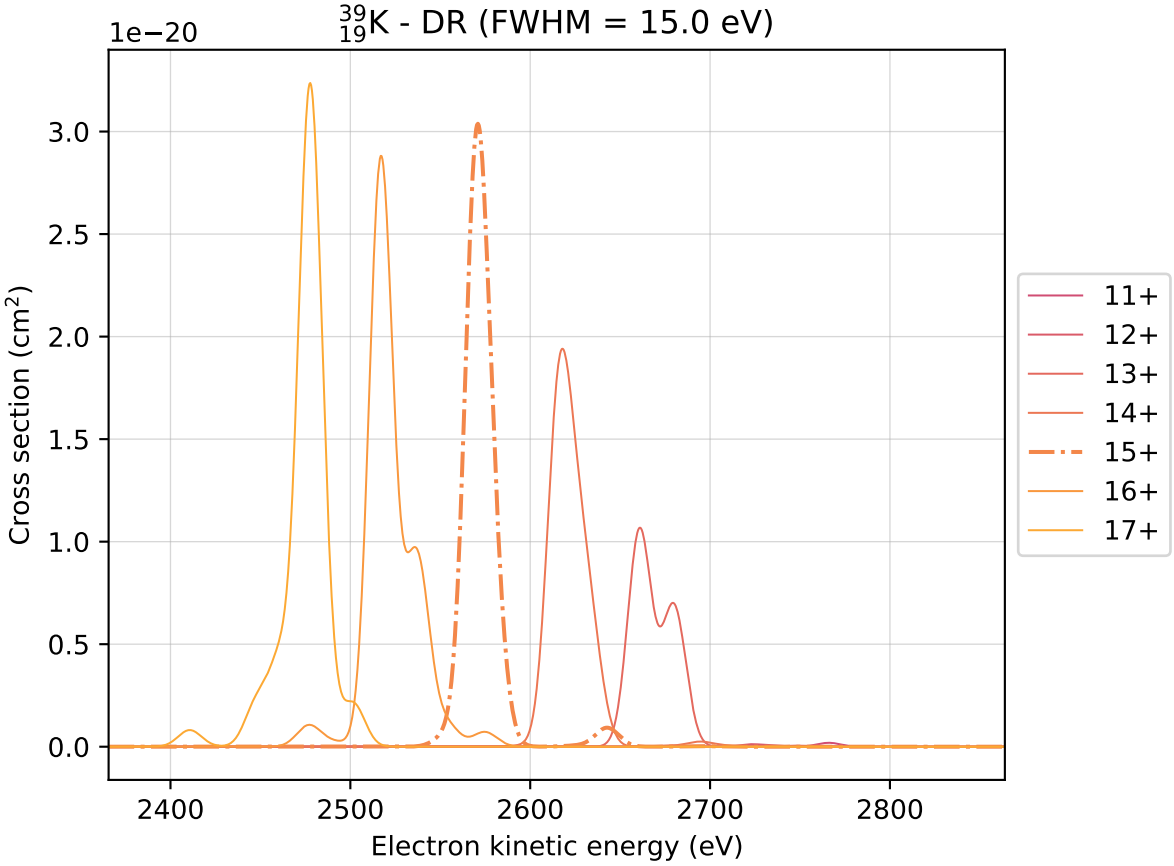
Ebisim offers a basic simulation scenario, which only takes three basic processes into account, namely Electron Ionisation (EI), Radiative Recombination (RR) and Dielectronic Recombination (DR). DR is only included on demand and depends on the availability of corresponding data describing the transitions. Tables with data for KLL type transitions are included in the python package.

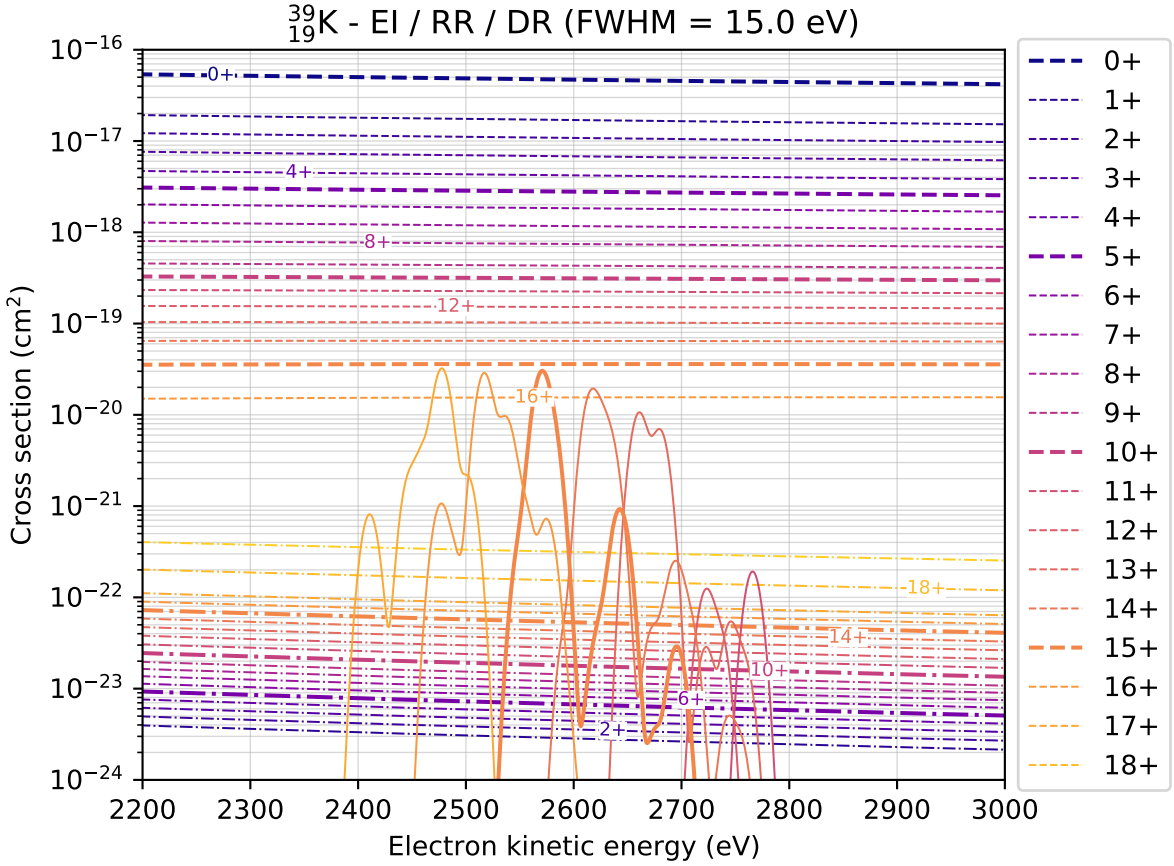
Setting up such a simulation and looking at its results is straightforward.

```
"""Example: Basic simulation with DR"""

from matplotlib.pyplot import show
import ebisim as eb

# For the basic simulation only a small number of parameters needs to be provided

element = eb.get_element("Potassium")  # element that is to be charge bred
```

(continues on next page)

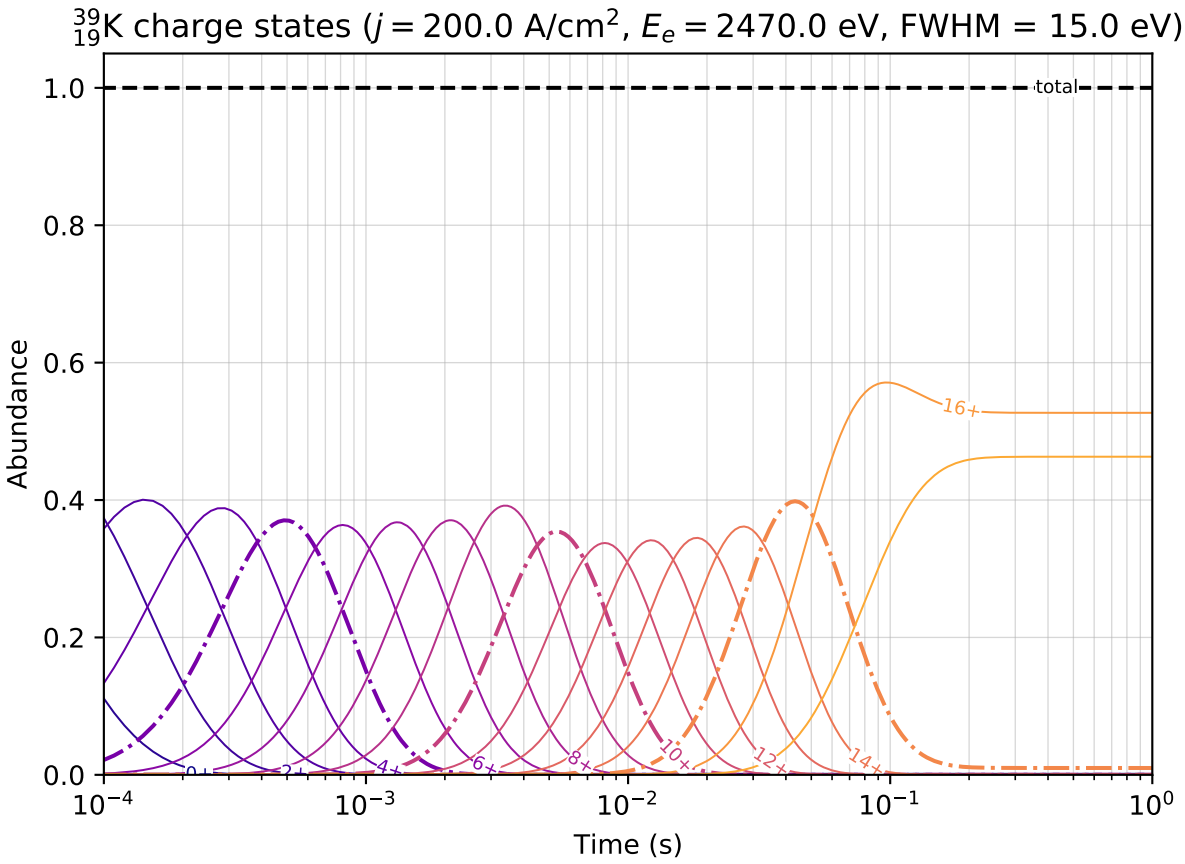

$^{39}_{19}$K - RR
Cross section (cm^2)
10^{-18}
10^{-20}
10^{-22}
10^{-24}
10^{-26}
10^0
10^1
10^2
10^3
10^4
10^5
Electron kinetic energy (eV)
1+
3+
5+
7+
9+
11+
13+
15+
17+
19+


$^{39}_{19}$K - DR (FWHM = 15.0 eV)
1e−20
3.0
2.5
2.0
1.5
1.0
0.5
0.0
Cross section (cm^2)
2400
2500
2600
2700
2800
Electron kinetic energy (eV)
11+
12+
13+
14+
15+
16+
17+


$^{39}_{19}$K - EI / RR / DR (FWHM = 15.0 eV)
Cross section (cm^2)
Electron kinetic energy (eV)
0+
1+
2+
3+
4+
5+
6+
7+
8+
9+
10+
11+
12+
13+
14+
15+
16+
17+
18+

(continued from previous page)

```
j = 200  # current density in A/cm^2
e_kin = 2470  # electron beam energy in eV
t_max = 1  # length of simulation in s
dr_fwhm = 15  # effective energy spread, widening the DR resonances in eV, optional

result = eb.basic_simulation(
    element=element,
    j=j,
    e_kin=e_kin,
    t_max=t_max,
    dr_fwhm=dr_fwhm,
)

# The result object holds the relevant data which could be inspected manually
# It also offers convenience methods to plot the charge state evolution

result.plot_charge_states()

show()
```



The simulation above assumes that all ions start in charge state 1+. One can easily simulate the continuous injection of a neutral gas by activating the flag for Continuous Neutral Injection (CNI).

```
"""Example: Basic simulation with CNI"""

from matplotlib.pyplot import show
import ebisim as eb

element = eb.get_element("Argon")  # element that is to be charge bred
j = 200  # current density in A/cm^2
e_kin = 2470  # electron beam energy in eV
t_max = 1  # length of simulation in s

result = eb.basic_simulation(
    element=element,
    j=j,
    e_kin=e_kin,
    t_max=t_max,
    CNI=True  # activate CNI
)

# Since the number of ions is constantly increasing with CNI,
# it is helpful to only plot the relative distribution at each time step
# This is easily achieved by setting the corresponding flag
result.plot_charge_states(relative=True)

show()
```

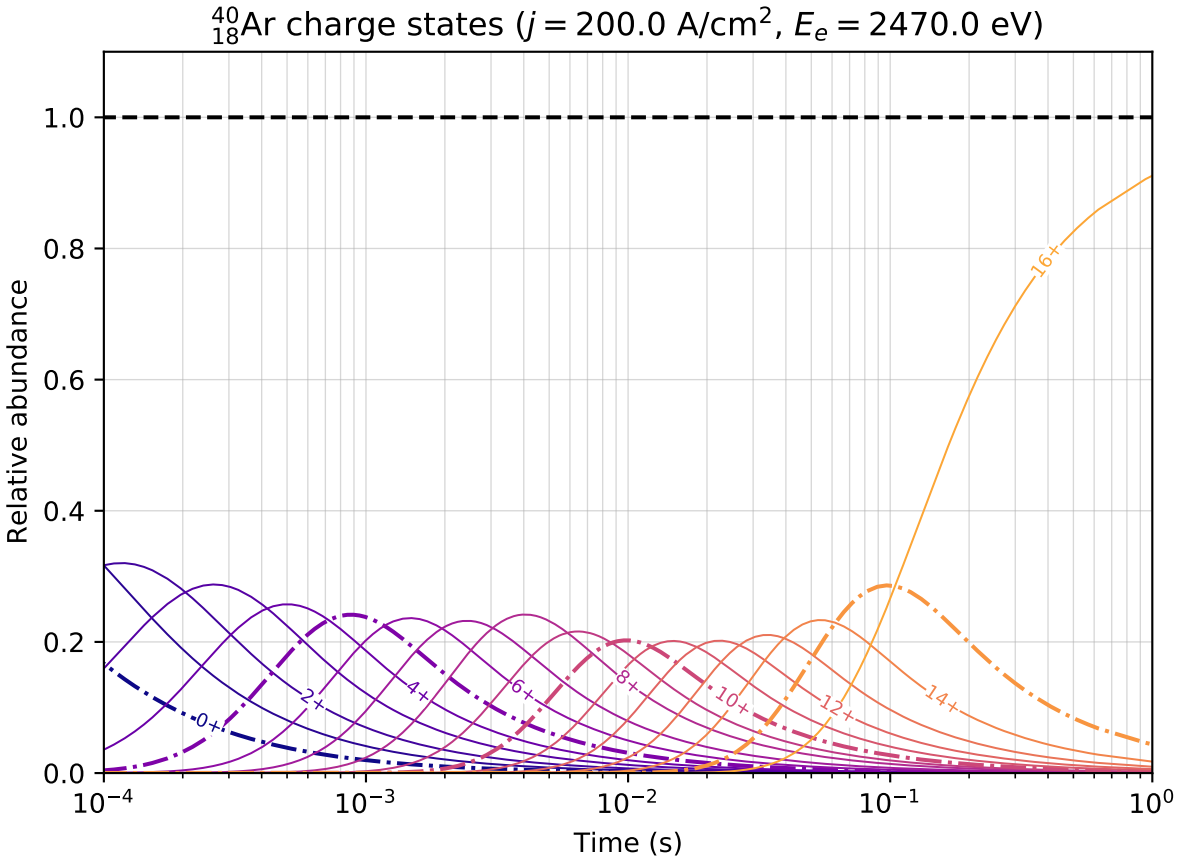
1.3 Energy scans

Occasionally it may be interesting to see, what kind of impact the electron beam energy has on the charge state distribution emerging after a given time. For this purpose, ebisim offers an energy scan feature.

An interesting example may for example be to scan across the ionisation threshold of a certain charge state, e.g. the ionisation of potassium 17+ to 18+ opens up at approximately 4.6 keV.

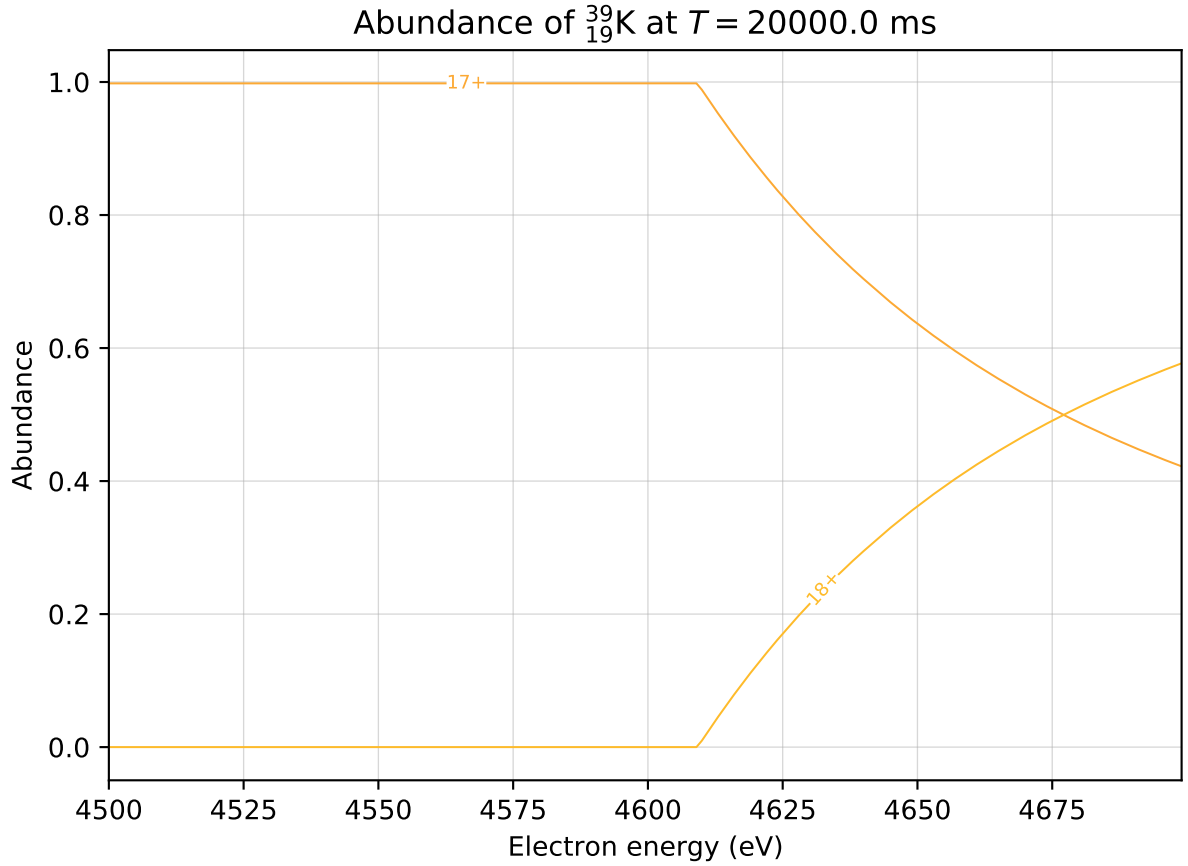
```
"""Example: Energy scan accross ionisation threshold"""

from matplotlib.pyplot import show
import numpy as np
import ebisim as eb

# For energy scans we have to create a python dictionary with the simulation parameters
# excluding the electron beam energy.
# The dictionary keys have to correspond to the argument and keyword argument names of␣
↪the
# simulation function that one wants to use, these names can be found in the API␣
↪Reference

sim_kwargs = dict(
    element=eb.get_element("Potassium"),  # element that is to be charge bred
    j=200,  # current density in A/cm^2
    t_max=100  # length of simulation in s
)
```

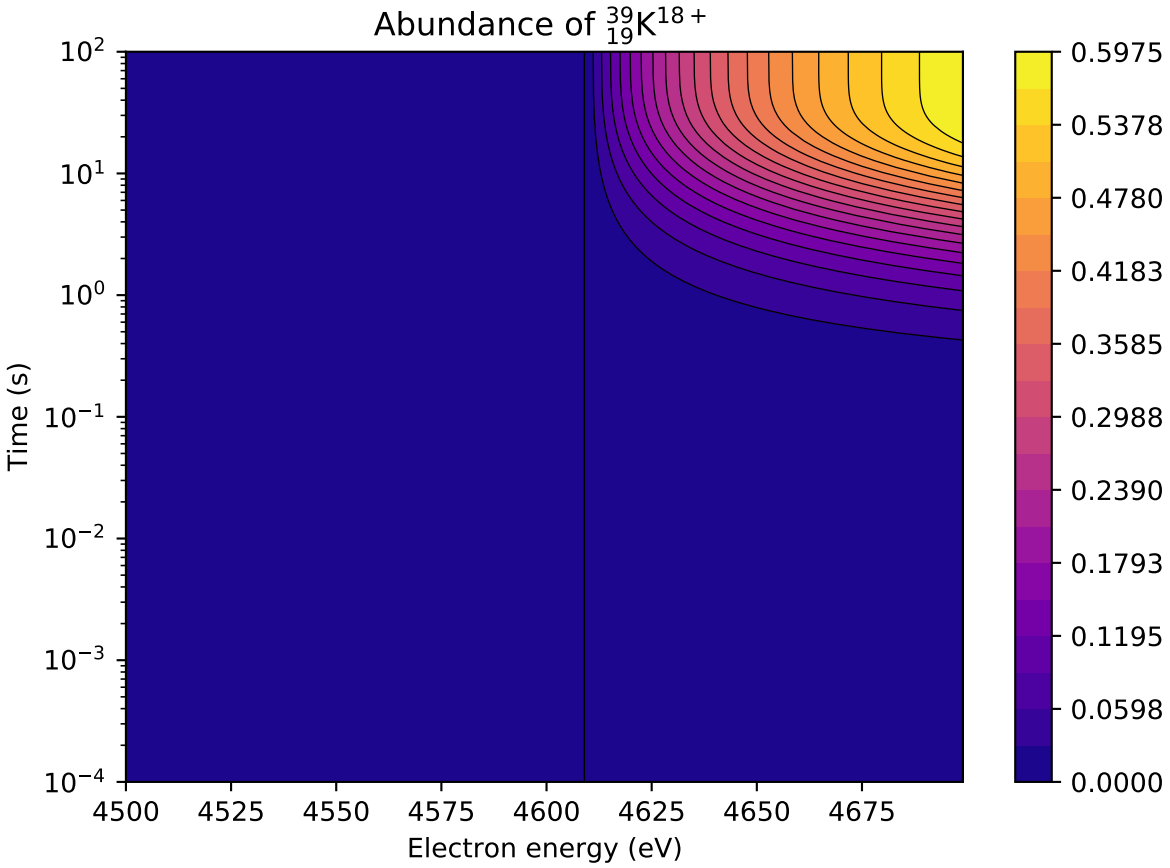
(continues on next page)


$^{40}_{18}$Ar charge states ($j = 200.0$ A/cm^2, $E_e = 2470.0$ eV)
Relative abundance
Time (s)
1.0
0.8
0.6
0.4
0.2
0.0
10^{-4}
10^{-3}
10^{-2}
10^{-1}
10^{0}
0+
2+
4+
6+
8+
10+
12+
14+
16+

(continued from previous page)

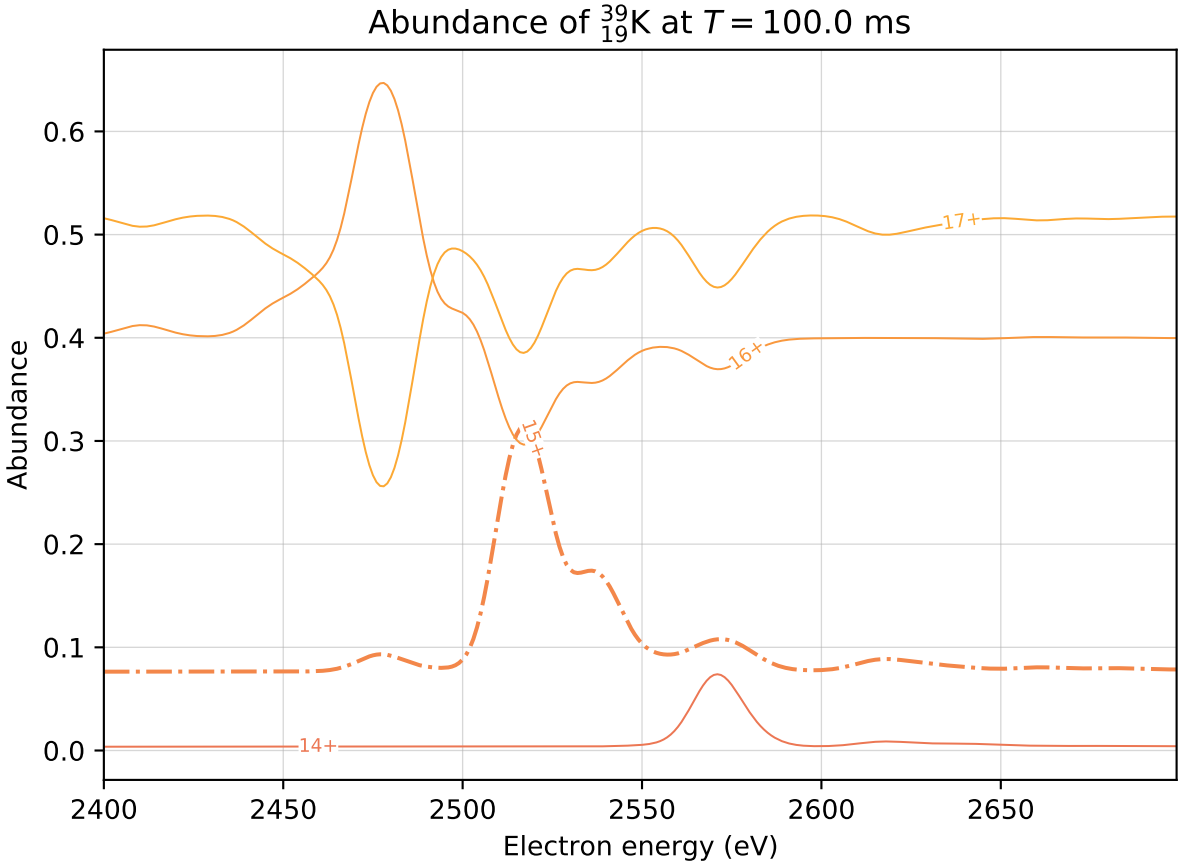
```
scan = eb.energy_scan(
    sim_func=eb.basic_simulation,  # the function handle of the simulation has to be␣
↪provided
    sim_kwargs=sim_kwargs,  # Here the general parameters are injected
    energies=np.arange(4500, 4700),  # The sampling energies
    parallel=True  # Speed up by running simulations in parallel
)

# The scan result object holds the relevant data which could be inspected manually
# It also offers convenience methods for plotting

# One thing that can be done is to plot the abundance of certain charge states
# at a given time, dependent on the energies
# In order not to plot all the different charge states we can supply a filter list
scan.plot_abundance_at_time(t=20, cs=[17, 18])

# Alternatively, one can create a plot to see how a given charge states depends on the␣
↪breeding time
# and the electron beam energy
scan.plot_abundance_of_cs(cs=18)

show()
```

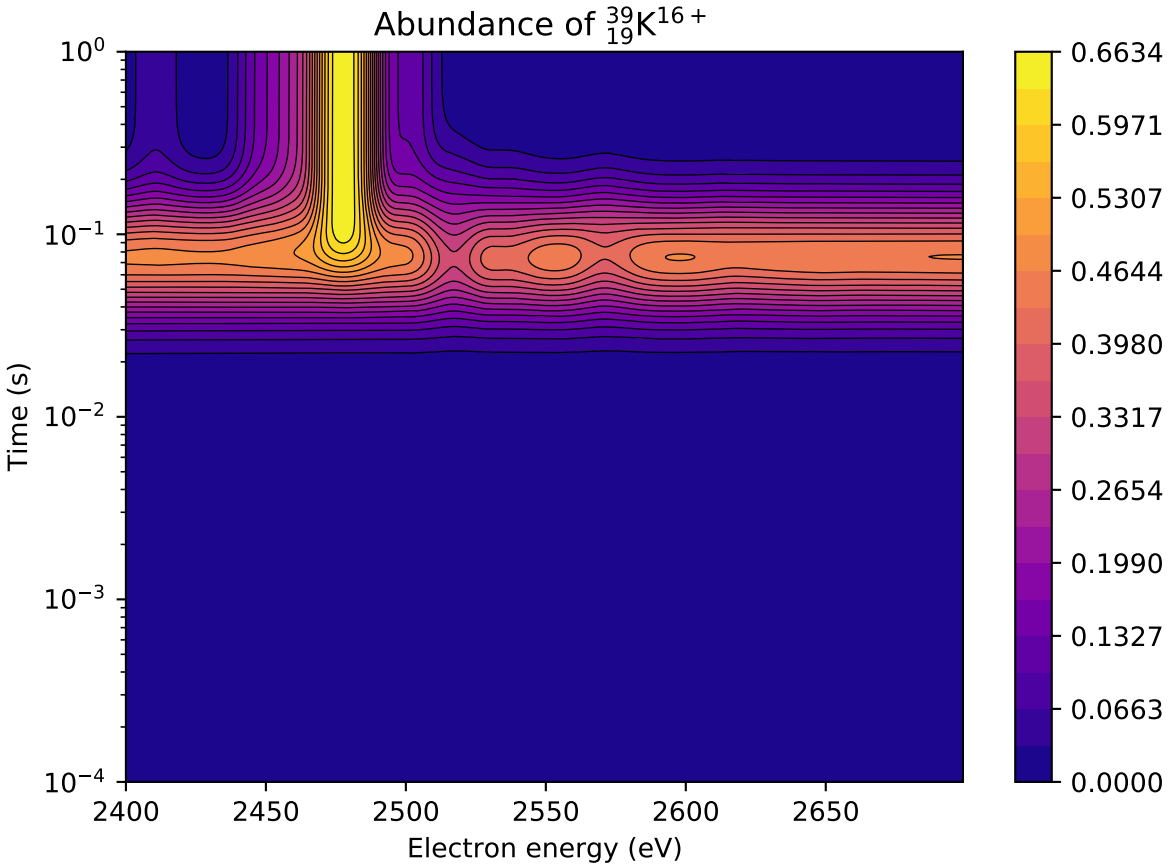



Abundance of $^{39}_{19}K^{18+}$
Time (s)
10^{2}
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
4500
4525
4550
4575
4600
4625
4650
4675
Electron energy (eV)
0.5975
0.5378
0.4780
0.4183
0.3585
0.2988
0.2390
0.1793
0.1195
0.0598
0.0000

Another example, which provides structure that is a bit more interesting, is the sweep over the KLL-type dielectronic recombination resonances of potassium.

```
"""Example: Energy scan accross DR resonances"""

from matplotlib.pyplot import show
import numpy as np
import ebisim as eb

sim_kwargs = dict(
    element=eb.get_element("Potassium"),  # element that is to be charge bred
    j=200,  # current density in A/cm^2
    t_max=1,  # length of simulation in s
    dr_fwhm=15  # This time DR has to be activated by setting an effective line width
)
scan = eb.energy_scan(
    sim_func=eb.basic_simulation,
    sim_kwargs=sim_kwargs,
    energies=np.arange(2400, 2700),  # The sampling energies cover the KLL band
    parallel=True
)

scan.plot_abundance_at_time(t=0.1, cs=[14, 15, 16, 17])
scan.plot_abundance_of_cs(cs=16)

show()
```


Abundance of $^{39}_{19}$K at $T = 100.0$ ms
Abundance
0.0
0.1
0.2
0.3
0.4
0.5
0.6
2400
2450
2500
2550
2600
2650
Electron energy (eV)
17+
16+
15+
14+


Abundance of $^{39}_{19}K^{16+}$
Time (s)
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
2400
2450
2500
2550
2600
2650
Electron energy (eV)
0.6634
0.5971
0.5307
0.4644
0.3980
0.3317
0.2654
0.1990
0.1327
0.0663
0.0000

CHAPTER

TWO

EBISIM PACKAGE

This is the top level module of ebisim.

The ebisim package offers a number of tools to perform and evaluate simulations of the charge breeding process in an Electron Beam Ion Source / Trap.

For easier access a number of submodule members are available at this level, such that they can be referred to as ebisim.[member] without resolving the submodule in the call. The module level documentation lists these members with more granularity.

class ebisim.**AdvancedModel**(*device*, *targets*, *bg_gases*, *options*, *lb*, *ub*, *nq*, *q*, *a*, *eixs*, *rrxs*, *drxs*, *cxxs_bggas*, *cxxs_trgts*)

Bases: tuple

The advanced model class is the base for ebisim.simulation.advanced_simulation. It acts as a fast datacontainer for the underlying rhs function which represents the right hand side of the differential equation system. Since it is jitcompiled using numba, care is required during instatiation.

Parameters

- **device** (*ebisim.simulation.Device*) – Container describing the EBIS/T and specifically the electron beam.
- **targets** (*numba.typed.List[ebisim.simulation.Target]*) – List of ebisim.simulation.Target for which charge breeding is simulated.
- **bg_gases** (*numba.typed.List[ebisim.simulation.BackgroundGas], optional*) – List of ebisim.simulation.BackgroundGas which act as CX partners, by default None.
- **options** (*ebisim.simulation.ModelOptions, optional*) – Switches for effects considered in the simulation, see default values of ebisim.simulation.ModelOptions.
- **lb** (*ndarray*) –
- **ub** (*ndarray*) –
- **nq** (*int*) –
- **q** (*ndarray*) –
- **a** (*ndarray*) –
- **eixs** (*ndarray*) –
- **rrxs** (*ndarray*) –
- **drxs** (*ndarray*) –
- **cxxs_bggas** (*Any*) –
- **cxxs_trgts** (*Any*) –

Create new instance of AdvancedModel(device, targets, bg_gases, options, lb, ub, nq, q, a, eixs, rrxs, drxs, cxxs_bggas, cxxs_trgts)

count(*value*, */*)

Return number of occurrences of value.

classmethod get(*device*, *targets*, *bg_gases=None*, *options=ModelOptions(EI=True, RR=True, CX=True, DR=False, SPITZER_HEATING=True, COLLISIONAL_THERMALISATION=True, ESCAPE_AXIAL=True, ESCAPE_RADIAL=True, RECOMPUTE_CROSS_SECTIONS=False, RADIAL_DYNAMICS=False, IONISATION_HEATING=True, OVERRIDE_FWHM=False, RADIAL_SOLVER_MAX_STEPS=500, RADIAL_SOLVER_REL_DIFF=0.001)*)

Parameters

- **device** (Device) –
- **targets** (*List[Element]*) –
- **bg_gases** (*Optional[List[BackgroundGas]]*) –
- **options** (ModelOptions) –

Return type
AdvancedModel

index(*value*, *start=0*, *stop=9223372036854775807*, */*)

Return first index of value.

Raises ValueError if the value is not present.

property a

Alias for field number 8

property bg_gases

Alias for field number 2

property cxxs_bggas

Alias for field number 12

property cxxs_trgts

Alias for field number 13

property device

Alias for field number 0

property drxs

Alias for field number 11

property eixs

Alias for field number 9

property lb

Alias for field number 4

property nq

Alias for field number 6

property options

Alias for field number 3

property q

Alias for field number 7

property rrxs

Alias for field number 10

property targets

Alias for field number 1

property ub

Alias for field number 5

class ebisim.**AdvancedResult**(*, *t*, *N*, *device*, *target*, *kbT*, *model*, *id_*, *res=None*, *rates=None*)

Bases: *BasicResult*[*Device*]

Instances of this class are containers for the results of ebisim advanced_simulations and contain a variety of convenience methods for simple plot generation etc.

Parameters

- **t** – An array holding the time step coordinates.
- **N** – An array holding the occupancy of each charge state at a given time.
- **device** – If coming from advanced_simulation, this is the machine / electron beam description.
- **target** – If coming from advanced_simulation, this is the target represented by this Result.
- **kbT** – An array holding the temperature of each charge state at a given time.
- **model** – The AdvancedModel instance that was underlying the advanced_simulation
- **id** – The position of this target in the list of all simulated targets.
- **res** – The result object returned by scipy.integrate.solve_ivp. This can contain useful information about the solver performance etc. Refer to scipy documentation for details.
- **rates** – A dictionary containing the different breeding rates in arrays shaped like N.

abundance_at_time(*t*)

Yields the abundance of each charge state at a given time

Parameters
t (*float*) – <s> Point of time to evaluate.

Returns
Abundance of each charge state, array index corresponds to charge state.

Return type
ndarray

plot(*relative=False*, ***kwargs*)

Plot the charge state evolution of this result object.

Parameters

- **relative** (*bool*) – Flags whether the absolute numbers or a relative fraction should be plotted at each timestep, by default False.
- **kwargs** (*Any*) – Keyword arguments are handed down to ebisim.plotting.plot_generic_evolution, cf. documentation thereof. If no arguments are provided, reasonable default values are injected.

Returns
Figure handle of the generated plot.

Raises
ValueError – If the required data (self.t, self.N) is not available, i.e. corresponding attributes of this Result instance have not been set correctly.

Return type
Figure

plot_charge_states(*relative=False*, ***kwargs*)

Plot the charge state evolution of this result object.

Parameters

- **relative** (*bool*) – Flags whether the absolute numbers or a relative fraction should be plotted at each timestep, by default False.
- **kwargs** (*Any*) – Keyword arguments are handed down to ebisim.plotting.plot_generic_evolution, cf. documentation thereof. If no arguments are provided, reasonable default values are injected.

Returns
Figure handle of the generated plot.

Raises
ValueError – If the required data (self.t, self.N) is not available, i.e. corresponding attributes of this Result instance have not been set correctly.

Return type
Figure

plot_energy_density(***kwargs*)

Plot the energy density evolution of this result object.

Parameters
kwargs (*Any*) – Keyword arguments are handed down to ebisim.plotting.plot_generic_evolution, cf. documentation thereof. If no arguments are provided, reasonable default values are injected.

Returns
Figure handle of the generated plot.

Raises
ValueError – If the required data (self.t, self.kbT) is not available, i.e. corresponding attributes of this Result instance have not been set correctly.

Return type
Figure

plot_radial_distribution_at_time(*t*, ***kwargs*)

Plot the radial ion distribution at time t.

Parameters

- **t** (*float*) – <s> Point of time to evaluate.
- **kwargs** (*Any*) – Keyword arguments are handed down to ebisim.plotting.plot_radial_distribution and ebisim.plotting.plot_generic_evolution, cf. documentation thereof. If no arguments are provided, reasonable default values are injected.

Returns
Figure handle of the generated plot.

Return type
Figure

`plot_rate`(*rate_key*, *dens_threshold=0.001*, ***kwargs*)

Plots the requested ionisation- or energy flow rates.

Parameters

- **rate_key** (Rate) – The key identifying the rate to be plotted. See ebisim.simulation.Rate for valid values.
- **dens_threshold** (*float*) – If given temperatures are only plotted where the particle denisty is larger than the threshold value.
- **kwargs** – Keyword arguments are handed down to ebisim.plotting.plot_generic_evolution, cf. documentation thereof. If no arguments are provided, reasonable default values are injected.

Returns
Figure handle of the generated plot

Raises
ValueError – If the required data (self.rates) is not available, or an invalid key is requested.

Return type
Figure

`plot_temperature`(*dens_threshold=0.001*, ***kwargs*)

Plot the temperature evolution of this result object.

Parameters

- **dens_threshold** (*float*) – If given temperatures are only plotted where the particle denisty is larger than the threshold value.
- **kwargs** (*Any*) – Keyword arguments are handed down to ebisim.plotting.plot_generic_evolution, cf. documentation thereof. If no arguments are provided, reasonable default values are injected.

Returns
Figure handle of the generated plot

Raises
ValueError – If the required data (self.t, self.kbT) is not available, i.e. corresponding attributes of this Result instance have not been set correctly.

Return type
Figure

`radial_distribution_at_time`(*t*)

Yields the radial distribution information at time

Parameters
t (*float*) – <s> Point of time to evaluate.

Returns

- *phi* – Radial potential
- *n3d* – On axis 3D density for each charge state.

- *shapes* – The Boltzmann shape factors for each charge state.

Return type
Tuple[*ndarray*, *ndarray*, *ndarray*]

`temperature_at_time`(*t*)

Yields the temperature of each charge state at a given time

Parameters
t (*float*) – <s> Point of time to evaluate.

Returns

- *<eV>*
- *Temperature of each charge state, array index corresponds to charge state.*

Return type
ndarray

`times_of_highest_abundance`()

Yields the point of time with the highest abundance for each charge state

Returns

- *<s>*
- *Array of times.*

Return type
ndarray

`class` ebisim.`BackgroundGas`(*name*, *ip*, *n0*)

Bases: `tuple`

Use the static *get()* factory methods to create instances of this class.

Simple datacontainer for a background gas for advanced simulations. A background gas only acts as a charge exchange partner to the Targets in the simulation.

See also:

ebisim.simulation.BackgroundGas.get

Create new instance of BackgroundGas(name, ip, n0)

Parameters

- **name** (*str*) –
- **ip** (*float*) –
- **n0** (*float*) –

`count`(*value*, */*)

Return number of occurrences of value.

`classmethod get`(*element*, *p*, *T=300.0*)

Factory method for defining a background gas.

Parameters

- **element** (*Union[Element, str, int]*) – An instance of the Element class, or an identifier for the element, i.e. either its name, symbol or proton number.
- **p** (*float*) – <mbar> Gas pressure.

- **T** (*float*) – <K> Gas temperature, by default 300 K (Room temperature)

Returns
ebisim.simulation.BackgroundGas – Ready to use BackgroundGas specification.

Return type
BackgroundGas

index(*value*, *start=0*, *stop=9223372036854775807*, */*)

Return first index of value.

Raises ValueError if the value is not present.

property ip

<eV> Ionisation potential of this Gas.

property n0

<1/m^3> Gas number density.

property name

Name of the element.

class ebisim.**BasicResult**(***, *t*, *N*, *device*, *target*, *res=None*)

Bases: Generic[GenericDevice]

Instances of this class are containers for the results of ebisim basic_simulations and contain a variety of convenience methods for simple plot generation etc.

Parameters

- **t** – An array holding the time step coordinates.
- **N** – An array holding the occupancy of each charge state at a given time.
- **device** – If coming from advanced_simulation, this is the machine / electron beam description.
- **target** – If coming from advanced_simulation, this is the target represented by this Result.
- **res** – The result object returned by scipy.integrate.solve_ivp. This can contain useful information about the solver performance etc. Refer to scipy documentation for details.

abundance_at_time(*t*)

Yields the abundance of each charge state at a given time

Parameters
t (*float*) – <s> Point of time to evaluate.

Returns
Abundance of each charge state, array index corresponds to charge state.

Return type
ndarray

plot(*relative=False*, ***kwargs*)

Alias for plot_charge_states

Parameters

- **relative** (*bool*) –
- **kwargs** (*Any*) –

Return type
Figure

`plot_charge_states`(*relative=False*, ***kwargs*)

Plot the charge state evolution of this result object.

Parameters

- **relative** (*bool*) – Flags whether the absolute numbers or a relative fraction should be plotted at each timestep, by default False.
- **kwargs** (*Any*) – Keyword arguments are handed down to ebisim.plotting.plot_generic_evolution, cf. documentation thereof. If no arguments are provided, reasonable default values are injected.

Returns
Figure handle of the generated plot.

Raises
ValueError – If the required data (self.t, self.N) is not available, i.e. corresponding attributes of this Result instance have not been set correctly.

Return type
Figure

`times_of_highest_abundance`()

Yields the point of time with the highest abundance for each charge state

Returns

- *<s>*
- *Array of times.*

Return type
ndarray

class ebisim.**Device**(*current*, *e_kin*, *r_e*, *length*, *j*, *v_ax*, *v_ax_sc*, *v_ra*, *b_ax*, *r_dt*, *r_dt_bar*, *fwhm*, *rad_grid*, *rad_fd_l*, *rad_fd_d*, *rad_fd_u*, *rad_phi_uncomp*, *rad_phi_ax_barr*, *rad_re_idx*)

Bases: `tuple`

Use the static *get()* factory methods to create instances of this class.

Objects of this class are used to pass important EBIS/T parameters into the simulation.

Create new instance of Device(current, e_kin, r_e, length, j, v_ax, v_ax_sc, v_ra, b_ax, r_dt, r_dt_bar, fwhm, rad_grid, rad_fd_l, rad_fd_d, rad_fd_u, rad_phi_uncomp, rad_phi_ax_barr, rad_re_idx)

Parameters

- **current** (*float*) –
- **e_kin** (*float*) –
- **r_e** (*float*) –
- **length** (*Optional[float]*) –
- **j** (*float*) –
- **v_ax** (*float*) –
- **v_ax_sc** (*float*) –
- **v_ra** (*float*) –

- **b_ax** (*float*) –
- **r_dt** (*float*) –
- **r_dt_bar** (*float*) –
- **fwhm** (*float*) –
- **rad_grid** (*ndarray*) –
- **rad_fd_l** (*ndarray*) –
- **rad_fd_d** (*ndarray*) –
- **rad_fd_u** (*ndarray*) –
- **rad_phi_uncomp** (*ndarray*) –
- **rad_phi_ax_barr** (*ndarray*) –
- **rad_re_idx** (*int*) –

count(*value*, */*)

Return number of occurrences of value.

classmethod get(**, current, e_kin, r_e, v_ax, b_ax, r_dt, length=None, v_ra=None, j=None, fwhm=None, n_grid=400, r_dt_bar=None*)

Factory method for defining a device. All arguments are keyword only to reduce the chance of mixing them up.

Parameters

- **current** (*float*) – <A> Electron beam current.
- **e_kin** (*float*) – <eV> Uncorrected electron beam energy.
- **r_e** (*float*) – <m> Electron beam radius.
- **v_ax** (*float*) – <V> Axial barrier bias.
- **b_ax** (*float*) – <T> Axial magnetic flux density in the trap.
- **r_dt** (*float*) – <m> Drift tube radius.
- **length** (*Optional[float]*) – <m> Trap length -> Currently not used in simulations.
- **v_ra** (*Optional[float]*) – <V> Override for radial trap depth. Only effective if ModelOptions.RADIAL_DYNAMICS=False.
- **j** (*Optional[float]*) – <A/cm^2> Override for current density.
- **fwhm** (*Optional[float]*) – <eV> Override for the electron beam energy spread. Only effective if ModelOptions.RADIAL_DYNAMICS=False.
- **n_grid** (*int*) – Approximate number of nodes for the radial mesh.
- **r_dt_bar** (*Optional[float]*) – Radius of the barrier drift tubes. If not passed, assuming equal radius as trap drift tube.

Returns

ebisim.simulation.Device – The populated device object.

Return type

Device

index(*value*, *start=0*, *stop=9223372036854775807*, */*)

Return first index of value.

Raises ValueError if the value is not present.

property b_ax

<T> Axial magnetic field density.

property current

<A> Beam current.

property e_kin

<eV> Uncorrected beam energy.

property fwhm

<eV> Electron beam energy spread.

property j

<A/cm^2> Current density.

property length

<m> Trap length.

property r_dt

<m> Drift tube radius.

property r_dt_bar

<m> Drift tube radius of the barrier drift tubes.

property r_e

<m> Beam radius.

property rad_fd_d

Diagonal vector of finite difference scheme.

property rad_fd_l

Lower diagonal vector of finite difference scheme.

property rad_fd_u

Upper diagonal vector of finite difference scheme.

property rad_grid

<m> Radial grid for finite difference computations.

property rad_phi_ax_barr

<V> Radial potential of the electron beam in the barrier tube

property rad_phi_uncomp

<V> Radial potential of the electron beam.

property rad_re_idx

Index of the radial grid point closest to r_e

property v_ax

<V> Axial barrier voltage.

property v_ax_sc

<V> Axial barrier space charge correction.

property v_ra

<V> Radial barrier voltage.

class ebisim.**Element**(*z*, *symbol*, *name*, *a*, *ip*, *e_cfg*, *e_bind*, *rr_z_eff*, *rr_n_0_eff*, *dr_cs*, *dr_e_res*, *dr_strength*, *ei_lotz_a*, *ei_lotz_b*, *ei_lotz_c*, *n=None*, *kT=None*, *cx=True*)

Bases: tuple

The Element class is one of the main data structures in ebisim. Virtually any function relies on information provided in this data structure. The leading fields of the underlying tuple contain physcial properties, whereas the *n kT* and *cx* fields are optional and only required for advanced simulations.

Instead of populating the fields manually, the user should choose one of the factory functions that meets their needs best.

For basic simulations and cross sections calculations only the physical / chemical properties of and element are needed. In these cases use the generic *get()* method to create instances of this class.

Advanced simulations require additional information about the initial particle densities, temperature and participation in charge exchange. The user will likely want to chose between the *get_ions()* and *get_gas()* methods, which offer a convenient interface for generating this data based on simple parameters. If these functions are not flexible enough, the *get()* method can be used to populate the required fields manually.

This class is derived from collections.namedtuple which facilitates use with numba-compiled functions.

See also:

ebisim.elements.Element.get, *ebisim.elements.Element.get_ions*, *ebisim.elements.Element.get_gas*

Create new instance of Element(z, symbol, name, a, ip, e_cfg, e_bind, rr_z_eff, rr_n_0_eff, dr_cs, dr_e_res, dr_strength, ei_lotz_a, ei_lotz_b, ei_lotz_c, n, kT, cx)

Parameters

- **z** (*int*) –
- **symbol** (*str*) –
- **name** (*str*) –
- **a** (*float*) –
- **ip** (*float*) –
- **e_cfg** (*ndarray*) –
- **e_bind** (*ndarray*) –
- **rr_z_eff** (*ndarray*) –
- **rr_n_0_eff** (*ndarray*) –
- **dr_cs** (*ndarray*) –
- **dr_e_res** (*ndarray*) –
- **dr_strength** (*ndarray*) –
- **ei_lotz_a** (*ndarray*) –
- **ei_lotz_b** (*ndarray*) –
- **ei_lotz_c** (*ndarray*) –
- **n** (*Optional[ndarray]*) –
- **kT** (*Optional[ndarray]*) –

- **cx** (*bool*) –

classmethod as_element(*element*)

If *element* is already an instance of *Element* it is returned. If *element* is a string or int identyfying an element an appropriate *Element* instance is returned.

Parameters
element (*Union[Element, str, int]*) – An instance of the Element class, or an identifier for the element, i.e. either its name, symbol or proton number.

Returns
An instance of Element reflecting the input value.

Return type
Element

count(*value*, */*)

Return number of occurrences of value.

classmethod get(*element_id*, *a=None*, *n=None*, *kT=None*, *cx=True*)

Factory method to create instances of the Element class.

Parameters

- **element_id** (*Union[str, int]*) – The full name, abbreviated symbol, or proton number of the element of interest.
- **a** (*Optional[float]*) – If provided sets the mass number of the Element object otherwise a reasonable value is chosen automatically.
- **n** (*Optional[ndarray]*) – <1/m> Only needed for advanced simulations! Array holding the initial ion line densities of each charge state. If provided, has to be an array of length Z+1, where Z is the nuclear charge.
- **kT** (*Optional[ndarray]*) – <eV> Only needed for advanced simulations! Array holding the initial ion line densities of each charge state. If provided, has to be an array of length Z+1, where Z is the nuclear charge.
- **cx** (*bool*) – Only needed for advanced simulations! Boolean flag determining whether the neutral particles of this element contribute to charge exchange with ions.

Returns
An instance of Element with the user-supplied and generated data.

Raises

- **ValueError** – If the Element could not be identified or a meaningless mass number is provided.
- **ValueError** – If the passed arrays for *n* or *kT* have the wrong shape.

Return type
Element

classmethod get_gas(*element_id*, *p*, *r_dt*, *T=300.0*, *cx=True*, *a=None*)

Factory method for defining a neutral gas injection target. A gas target is a target with constant density in charge state 0.

Parameters

- **element_id** (*Union[str, int]*) – The full name, abbreviated symbol, or proton number of the element of interest.

- **p** (*float*) – <mbar> Gas pressure.
- **r_dt** (*float*) – <m> Drift tube radius, required to compute linear density from volumetric density.
- **T** (*float*) – <K> Gas temperature, by default 300 K (approx. room temperature)
- **cx** (*bool*) – Boolean flag determining whether the neutral particles of this element contribute to charge exchange with ions.
- **a** (*Optional[float]*) – If provided sets the mass number of the Element object otherwise a reasonable value is chosen automatically.

Returns
Element instance with automatically populated n and kT fields.

Raises
ValueError – If the density resulting from the pressure and temperature is smaller than the internal minimal value.

Return type
Element

classmethod get_ions(*element_id*, *nl*, *kT=10.0*, *q=1*, *cx=True*, *a=None*)

Factory method for defining a pulsed ion injection target. An ion target has a given density in the charge state of choice q.

Parameters

- **element_id** (*Union[str, int]*) – The full name, abbreviated symbol, or proton number of the element of interest.
- **nl** (*float*) – <1/m> Linear density of the initial charge state (ions per unit length).
- **kT** (*float*) – <eV> Temperature / kinetic energy of the injected ions.
- **q** (*int*) – Initial charge state.
- **cx** (*bool*) – Boolean flag determining whether the neutral particles of this element contribute to charge exchange with ions.
- **a** (*Optional[float]*) – If provided sets the mass number of the Element object otherwise a reasonable value is chosen automatically.

Returns
Element instance with automatically populated n and kT fields.

Raises
ValueError – If the requested density is smaller than the internal minimal value.

Return type
Element

index(*value*, *start=0*, *stop=9223372036854775807*, */*)

Return first index of value.

Raises ValueError if the value is not present.

latex_isotope()

Returns the isotope as a LaTeX formatted string.

Returns
str – LaTeX formatted string describing the isotope.

Return type
str

property a

Mass number / approx. mass in proton masses

property cx

Boolean flag determining whether neutral particles of this target are considered as charge exchange partners.

property dr_cs

Numpy array of charge states for DR cross sections.

property dr_e_res

Numpy array of resonance energies for DR cross sections.

property dr_strength

Numpy array of transition strengths for DR cross sections.

property e_bind

Numpy array of binding energies associated with electron subshells. The index of each row corresponds to the charge state. The columns are the subshells sorted as in ('1s', '2s', '2p-', '2p+', '3s', '3p-', '3p+', '3d-', '3d+', '4s', '4p-', '4p+', '4d-', '4d+', '4f-', '4f+', '5s', '5p-', '5p+', '5d-', '5d+', '5f-', '5f+', '6s', '6p-', '6p+', '6d-', '6d+', '7s', '7p-').

property e_cfg

Numpy array of electron configuration in different charge states. The index of each row corresponds to the charge state. The columns are the subshells sorted as in ('1s', '2s', '2p-', '2p+', '3s', '3p-', '3p+', '3d-', '3d+', '4s', '4p-', '4p+', '4d-', '4d+', '4f-', '4f+', '5s', '5p-', '5p+', '5d-', '5d+', '5f-', '5f+', '6s', '6p-', '6p+', '6d-', '6d+', '7s', '7p-').

property ei_lotz_a

Numpy array of precomputed Lotz factor 'a' for each entry of 'e_cfg'.

property ei_lotz_b

Numpy array of precomputed Lotz factor 'b' for each entry of 'e_cfg'.

property ei_lotz_c

Numpy array of precomputed Lotz factor 'c' for each entry of 'e_cfg'.

property ip

Ionisation potential

property kT

<eV> Array holding the initial temperature of each charge state.

property n

<1/m> Array holding the initial linear density of each charge state.

property name

Element name

property rr_n_0_eff

Numpy array of effective valence shell numbers for RR cross sections.

property rr_z_eff

Numpy array of effective nuclear charges for RR cross sections.

property symbol

Element symbol e.g. H, He, Li

property z

Atomic number

class ebisim.EnergyScanResult(*sim_kwargs*, *energies*, *results*)

Bases: object

This class provides a convenient interface to access and evaluate the the results of the ebisim.simulation.energy_scan function. Abundances at arbitrary times are provided by performing linear interpolations of the solutions of the rate equation.

Parameters

- **sim_kwargs** (*dict*) – The sim_kwargs dictionary as provided during the call to ebisim.simulation.energy_scan.
- **energies** (*numpy.array*) – <eV> A sorted array containing the energies at which the energy scan has been evaluated.
- **results** (*list of ebisim.simulation.Result*) – A list of Result objects holding the results of each individual simulation

abundance_at_time(*t*)

Provides information about the charge state distribution at a given time for all energies.

Parameters
t (*float*) – <s> Point of time to evaluate.

Returns

- **energies** (*numpy.array*) – The evaluated energies.
- **abundance** (*numpy.array*) – Contains the abundance of each charge state (rows) for each energy (columns).

Raises
ValueError – If 't' is not part of the simulated time domain

abundance_of_cs(*cs*)

Provides information about the abundance of a single charge states at all simulated times and energies.

Parameters
cs (*int*) – The charge state to evaluate.

Returns

- **energies** (*numpy.array*) – The evaluated energies.
- **times** (*numpy.array*) – The evaluated timesteps.
- **abundance** (*numpy.array*) – Abundance of charge state 'cs' at given times (rows) and energies (columns).

Raises
ValueError – If 'cs' is not a sensible charge state for the performed simulation.

get_result(*e_kin*)

Returns the result object corresponding to the simulation at a given energy

Parameters
e_kin (*float*) – <eV> The energy of the simulation one wishes to retrieve.

Returns
ebisim.simulation.Result – The Result object for the polled scan step.

Raises
ValueError – If the polled energy is not available.

`plot_abundance_at_time`(*t*, *cs=None*, ***kwargs*)

Produces a plot of the charge state abundance for different energies at a given time.

Parameters

- **t** (*float*) – <s> Point of time to evaluate.
- **cs** (*list or None, optional*) – If None, all charge states are plotted. By supplying a list of int it is possible to filter the charge states that should be plotted. By default None.
- ****kwargs** – Keyword arguments are handed down to ebisim.plotting.plot_energy_scan, cf. documentation thereof. If no arguments are provided, reasonable default values are injected.

Returns
matplotlib.Figure – Figure handle of the generated plot.

`plot_abundance_of_cs`(*cs*, ***kwargs*)

Produces a 2D contour plot of the charge state abundance for all simulated energies and times.

Parameters

- **cs** (*int*) – The charge state to plot.
- ****kwargs** – Keyword arguments are handed down to ebisim.plotting.plot_energy_scan, cf. documentation thereof. If no arguments are provided, reasonable default values are injected.

Returns
matplotlib.Figure – Figure handle of the generated plot.

`class ebisim.ModelOptions`(*EI=True*, *RR=True*, *CX=True*, *DR=False*, *SPITZER_HEATING=True*, *COLLISIONAL_THERMALISATION=True*, *ESCAPE_AXIAL=True*, *ESCAPE_RADIAL=True*, *RECOMPUTE_CROSS_SECTIONS=False*, *RADIAL_DYNAMICS=False*, *IONISATION_HEATING=True*, *OVERRIDE_FWHM=False*, *RADIAL_SOLVER_MAX_STEPS=500*, *RADIAL_SOLVER_REL_DIFF=0.001*)

Bases: `tuple`

An instance of ModelOptions can be used to turn on or off certain effects in an advanced simulation.

Create new instance of ModelOptions(EI, RR, CX, DR, SPITZER_HEATING, COLLISIONAL_THERMALISATION, ESCAPE_AXIAL, ESCAPE_RADIAL, RECOMPUTE_CROSS_SECTIONS, RADIAL_DYNAMICS, IONISATION_HEATING, OVERRIDE_FWHM, RADIAL_SOLVER_MAX_STEPS, RADIAL_SOLVER_REL_DIFF)

Parameters

- **EI** (*bool*) –
- **RR** (*bool*) –
- **CX** (*bool*) –
- **DR** (*bool*) –
- **SPITZER_HEATING** (*bool*) –

- **COLLISIONAL_THERMALISATION** (*bool*) –
- **ESCAPE_AXIAL** (*bool*) –
- **ESCAPE_RADIAL** (*bool*) –
- **RECOMPUTE_CROSS_SECTIONS** (*bool*) –
- **RADIAL_DYNAMICS** (*bool*) –
- **IONISATION_HEATING** (*bool*) –
- **OVERRIDE_FWHM** (*bool*) –
- **RADIAL_SOLVER_MAX_STEPS** (*int*) –
- **RADIAL_SOLVER_REL_DIFF** (*float*) –

count(*value*, */*)

Return number of occurrences of value.

index(*value*, *start=0*, *stop=9223372036854775807*, */*)

Return first index of value.

Raises ValueError if the value is not present.

property COLLISIONAL_THERMALISATION

Switch for ion-ion thermalisation, default True.

property CX

Switch for charge exchange, default True.

property DR

Switch for dielectronic recombination, default False.

property EI

Switch for electron impact ionisation, default True.

property ESCAPE_AXIAL

Switch for axial escape from the trap, default True.

property ESCAPE_RADIAL

Switch for radial escape from the trap, default True.

property IONISATION_HEATING

Switch for ionisation heating/recombination cooling

property OVERRIDE_FWHM

If set use FWHM from device definition instead of computed value.

property RADIAL_DYNAMICS

Switch for effects of radial ion cloud extent. May be computationally very intensive

property RADIAL_SOLVER_MAX_STEPS

Alias for field number 12

property RADIAL_SOLVER_REL_DIFF

Alias for field number 13

property RECOMPUTE_CROSS_SECTIONS

Switch deciding whether EI, RR, and DR cross sections are recomputed on each call of the differential equation system. Advisable if electron beam energy changes over time and sharp transitions are expected, e.g. DR or ionisation thresholds for a given shell. Default False.

property RR

Switch for radiative recombination, default True.

property SPITZER_HEATING

Switch for Spitzer- or electron-heating, default True.

class ebisim.Rate(*value*)

Bases: IntEnum

Enum for conveniently identifying rates produced in advanced simulations

AX_CO = 105

CHARGE_EXCHANGE = 104

COLLISIONAL_THERMALISATION = 301

COLLISION_RATE_SELF = 402

COLLISION_RATE_TOTAL = 401

CX = 104

DIELECTRONIC_RECOMBINATION = 103

DR = 103

EI = 101

ELECTRON_IONISATION = 101

E_KIN_FWHM = 532

E_KIN_MEAN = 531

F_EI = 501

IONISATION_HEAT = 303

LOSSES_AXIAL_COLLISIONAL = 105

LOSSES_RADIAL_COLLISIONAL = 107

OVERLAP_FACTORS_EBEAM = 501

RADIATIVE_RECOMBINATION = 102

RA_CO = 107

RR = 102

SPITZER_HEATING = 302

TRAPPING_PARAMETER_AXIAL = 521

TRAPPING_PARAMETER_RADIAL = 522

TRAP_DEPTH_AXIAL = 523

TRAP_DEPTH_RADIAL = 524

T_AX_CO = 205

T_COLLISIONAL_THERMALISATION = 301

T_CT = 301

T_LOSSES_AXIAL_COLLISIONAL = 205

T_LOSSES_RADIAL_COLLISIONAL = 207

T_RA_CO = 207

T_SH = 302

T_SPITZER_HEATING = 302

V_AX = 523

V_RA = 524

W_AX = 521

W_RA = 522

ebisim.**Target**

alias of *Element*

ebisim.**advanced_simulation**(*device*, *targets*, *t_max*, *bg_gases=None*, *options=None*, *rates=False*, *solver_kwargs=None*, *verbose=True*, *n_threads=1*)

Interface for performing advanced charge breeding simulations.

For a list of effects refer to *ebisim.simulation.ModelOptions*.

Parameters

- **device** (Device) – Container describing the EBIS/T and specifically the electron beam.
- **targets** (*Union[Element, List[Element]]*) – Target(s) for which charge breeding is simulated.
- **t_max** (*float*) – <s> Simulated breeding time
- **bg_gases** (*Optional[Union[BackgroundGas, List[BackgroundGas]]]*) – Background gas(es) which act as CX partners.
- **rates** (*bool*) – If true a 'second run' is performed to store the rates, this takes extra time and can create quite a bit of data.
- **options** (*Optional[ModelOptions]*) – Switches for effects considered in the simulation, see default values of ebisim.simulation.ModelOptions.
- **solver_kwargs** (*Optional[Dict[str, Any]]*) – If supplied these keyword arguments are unpacked in the solver call. Refer to the documentation of scipy.integrate.solve_ivp for more information. By default None.
- **verbose** (*bool*) – Print a little progress indicator and some status messages, by default True.

- **n_threads** (*int*) – How many threads to use (mostly for jacbion estimation which can evaluate the RHS in parallel with different inputs.)

Returns

- *An instance of the AdvancedResult class, holding the simulation parameters, timesteps and*
- *charge state distribution including the species temperature.*

Return type
Union[AdvancedResult, *Tuple*[AdvancedResult, ...]]

ebisim.**basic_simulation**(*element*, *j*, *e_kin*, *t_max*, *dr_fwhm=None*, *N_initial=None*, *CNI=False*, *solver_kwargs=None*)

Interface for performing basic charge breeding simulations.

These simulations only include the most important effects, i.e. electron ionisation, radiative recombination and optionally dielectronic recombination (for those transitions whose data is available in the resource directory). All other effects are ignored.

Continuous Neutral Injection (CNI) can be activated on demand.

The results only represent the proportions of different charge states, not actual densities.

Parameters

- **element** (*Union[Element, str, int]*) – An instance of the Element class, or an identifier for the element, i.e. either its name, symbol or proton number.
- **j** (*float*) – <A/cm^2> Current density
- **e_kin** (*float*) – <eV> Electron beam energy
- **t_max** (*float*) – <s> Simulated breeding time
- **dr_fwhm** (*Optional[float]*) – <eV> If a value is given, determines the energy spread of the electron beam (in terms of Full Width Half Max) and hence the effective width of DR resonances. Otherwise DR is excluded from the simulation.
- **N_initial** (*Optional[ndarray]*) – Determines the initial charge state distribution if given, must have Z + 1 entries, where the array index corresponds to the charge state. If no value is given the distribution defaults to 100% of 1+ ions at t = 0 s, or a small amount of neutral atoms in the case of CNI.
- **CNI** (*bool*) – If Continuous Neutral Injection is activated, the neutrals are assumed to form an infinite reservoir. Their absolute number will not change over time and hence they act as a constant source of new singly charged ions. Therefore the absolute amount of ions increases over time.
- **solver_kwargs** (*Optional[Dict[str, Any]]*) – If supplied these keyword arguments are unpacked in the solver call. Refer to the documentation of scipy.integrate.solve_ivp for more information.

Returns

- *An instance of the BasicResult class, holding the simulation parameters, timesteps and*
- *charge state distribution.*

Return type
BasicResult

@numba.jitebisim.**drxs_energyscan**(*element*, *fwhm*, *e_kin=None*, *n=1000*)

Creates an array of DR cross sections for varying electron energies.

Parameters

- **element** (ebisim.Element) – An ebisim.Element object that holds the required physical information for cross section calculations.
- **fwhm** (*float*) – <eV> Energy spread to apply for the resonance smearing, expressed in terms of full width at half maximum.
- **e_kin** (*None or numpy.ndarray, optional*) – <eV> If e_kin is None, the range of sampling energies is chosen based on the binding enrgies of the element and energies are sampled on a logscale. If e_kin is an array with 2 elements, they are interpreted as the minimum and maximum sampling energy. If e_kin is an array with more than two values, the energies are taken as the sampling energies directly, by default None.
- **n** (*int, optional*) – The number of energy sampling points, if the sampling locations are not supplied by the user, by default 1000.

Returns

- **e_samp** (*numpy.ndarray*) – <eV> Array holding the sampling energies
- **xs_scan** (*numpy.ndarray*) – <m^2> Array holding the cross sections, where the row index corresponds to the charge state and the columns correspond to the different sampling energies

See also:

ebisim.xs.eixs_energyscan, *ebisim.xs.rrxs_energyscan*

@numba.jitebisim.**drxs_mat**(*element*, *e_kin*, *fwhm*)

Dielectronic recombination cross section. The cross sections are estimated by weighing the strength of each transition with the profile of a normal Gaussian distribution. This simulates the effective spreading of the resonance peaks due to the energy spread of the electron beam

Parameters

- **element** (ebisim.Element) – An ebisim.Element object that holds the required physical information for cross section calculations.
- **e_kin** (*float*) – <eV> Kinetic energy of the impacting electron.
- **fwhm** (*float*) – <eV> Energy spread to apply for the resonance smearing, expressed in terms of full width at half maximum.

Returns

numpy.array – <m^2> The cross sections for each individual charge state, arranged in a matrix suitable for implementation of a rate equation like dN/dt = j * xs_matrix dot N. out[q, q] = - cross section of q+ ion out[q, q+1] = + cross section of (q+1)+ ion

See also:

ebisim.xs.drxs_vec
Similar method with different output format.

@numba.jitebisim.**drxs_vec**(*element*, *e_kin*, *fwhm*)

Dielectronic recombination cross section. The cross sections are estimated by weighing the strength of each transition with the profile of a normal Gaussian distribution. This simulates the effective spreading of the resonance peaks due to the energy spread of the electron beam

Parameters

- **element** (ebisim.Element) – An ebisim.Element object that holds the required physical information for cross section calculations.

- **e_kin** (*float*) – <eV> Kinetic energy of the impacting electron.
- **fwhm** (*float*) – <eV> Energy spread to apply for the resonance smearing, expressed in terms of full width at half maximum.

Returns

numpy.ndarray – <m^2> The cross sections for each individual charge state, where the array-index corresponds to the charge state, i.e. out[q] ~ cross section of q+ ion.

See also:

ebisim.xs.drxs_mat
Similar method with different output format.

@numba.jitebisim.**eixs_energyscan**(*element*, *e_kin=None*, *n=1000*)

Creates an array of EI cross sections for varying electron energies.

Parameters

- **element** (ebisim.Element) – An ebisim.Element object that holds the required physical information for cross section calculations.
- **e_kin** (*None or numpy.ndarray, optional*) – <eV> If e_kin is None, the range of sampling energies is chosen based on the binding enrgies of the element and energies are sampled on a logscale. If e_kin is an array with 2 elements, they are interpreted as the minimum and maximum sampling energy. If e_kin is an array with more than two values, the energies are taken as the sampling energies directly, by default None.
- **n** (*int, optional*) – The number of energy sampling points, if the sampling locations are not supplied by the user, by default 1000.

Returns

- **e_samp** (*numpy.ndarray*) – <eV> Array holding the sampling energies
- **xs_scan** (*numpy.ndarray*) – <m^2> Array holding the cross sections, where the row index corresponds to the charge state and the columns correspond to the different sampling energies

See also:

ebisim.xs.rrxs_energyscan, *ebisim.xs.drxs_energyscan*

@numba.jitebisim.**eixs_mat**(*element*, *e_kin*)

Electron ionisation cross section.

Parameters

- **element** (ebisim.Element) – An ebisim.Element object that holds the required physical information for cross section calculations.
- **e_kin** (*float*) – <eV> Kinetic energy of the impacting electron.

Returns

numpy.array – <m^2> The cross sections for each individual charge state, arranged in a matrix suitable for implementation of a rate equation like dN/dt = j * xs_matrix dot N. out[q, q] = - cross section of q+ ion out[q+1, q] = + cross section of (q+1)+ ion

See also:

ebisim.xs.eixs_vec
Similar method with different output format.

@numba.jitebisim.**eixs_vec**(*element*, *e_kin*)

Electron ionisation cross section according to a simplified version of the models given in [Lotz1967].

Parameters

- **element** (ebisim.Element) – An ebisim.Element object that holds the required physical information for cross section calculations.
- **e_kin** (*float*) – <eV> Kinetic energy of the impacting electron.

Returns

numpy.ndarray – <m^2> The cross sections for each individual charge state, where the array-index corresponds to the charge state, i.e. out[q] ~ cross section of q+ ion.

References

See also:

ebisim.xs.eixs_mat

Similar method with different output format.

ebisim.**energy_scan**(*sim_func*, *sim_kwargs*, *energies*, *parallel=False*)

This function provides a convenient way to repeat the same simulation for a number of different electron beam energies. This can reveal variations in the charge state balance due to weakly energy dependent ionisation cross sections or even resonant phenomena like dielectronic recombination.

Parameters

- **sim_func** (*callable*) – The function handle for the simulation e.g. ebisim.simulation.basic_simulation.
- **sim_kwargs** (*dict*) – A dictionary containing all the required and optional parameters of the simulations (except for the kinetic electron energy) as key value pairs. This is unpacked in the function call.
- **energies** (*list or numpy.array*) – A list or array of the energies at which the simulation should be performed.
- **parallel** (*bool, optional*) – Determine whether multiple simulations should be run in parallel using pythons multiprocessing.pool. This may accelerate the scan when performing a large number of simulations. By default False.

Returns

ebisim.simulation.EnergyScanResult – An object providing convenient access to the generated scan data.

ebisim.**get_element**(*element_id*, *a=None*)

[LEGACY] Factory function to create instances of the Element class.

Parameters

- **element_id** (*str or int*) – The full name, abbreviated symbol, or proton number of the element of interest.
- **a** (*int or None, optional*) – If provided sets the (isotopic) mass number of the Element object otherwise a reasonable value is chosen automatically, by default None.

Returns

ebisim.elements.Element – An instance of Element with the physical data corresponding to the supplied element_id, and optionally mass number.

Raises
ValueError – If the Element could not be identified or a meaningless mass number is provided.

Return type
Element

ebisim.**plot_combined_xs**(*element*, *fwhm*, ***kwargs*)

Creates a figure showing the electron ionisation, radiative recombination and, dielectronic recombination cross sections of the provided element.

Parameters

- **element** (ebisim.elements.Element *or str or int*) – An instance of the Element class, or an identifier for the element, i.e. either its name, symbol or proton number.
- **fwhm** (*float*) – <eV> Energy spread to apply for the resonance smearing, expressed in terms of full width at half maximum.
- ****kwargs** – Remaining keyword arguments are handed down to ebisim.plotting.decorate_axes, cf. documentation thereof. If no arguments are provided, reasonable default values are injected.

Returns
matplotlib.Figure – Figure handle of the generated plot.

ebisim.**plot_drxs**(*element*, *fwhm*, ***kwargs*)

Creates a figure showing the dielectronic recombination cross sections of the provided element.

Parameters

- **element** (ebisim.elements.Element *or str or int*) – An instance of the Element class, or an identifier for the element, i.e. either its name, symbol or proton number.
- **fwhm** (*float*) – <eV> Energy spread to apply for the resonance smearing, expressed in terms of full width at half maximum.
- ****kwargs** – 'fig' is intercepted and can be used to plot on top of an existing figure. 'ls' is intercepted and can be used to set the linestyle for plotting. Remaining keyword arguments are handed down to ebisim.plotting.decorate_axes, cf. documentation thereof. If no arguments are provided, reasonable default values are injected.

Returns
matplotlib.Figure – Figure handle of the generated plot.

ebisim.**plot_eixs**(*element*, ***kwargs*)

Creates a figure showing the electron ionisation cross sections of the provided element.

Parameters

- **element** (ebisim.elements.Element *or str or int*) – An instance of the Element class, or an identifier for the element, i.e. either its name, symbol or proton number.
- ****kwargs** – 'fig' is intercepted and can be used to plot on top of an existing figure. 'ls' is intercepted and can be used to set the linestyle for plotting. Remaining keyword arguments are handed down to ebisim.plotting.decorate_axes, cf. documentation thereof. If no arguments are provided, reasonable default values are injected.

Returns
matplotlib.Figure – Figure handle of the generated plot.

ebisim.**plot_rrxs**(*element*, ***kwargs*)

Creates a figure showing the radiative recombination cross sections of the provided element.

Parameters

- **element** (ebisim.elements.Element *or str or int*) – An instance of the Element class, or an identifier for the element, i.e. either its name, symbol or proton number.
- ****kwargs** – 'fig' is intercepted and can be used to plot on top of an existing figure. 'ls' is intercepted and can be used to set the linestyle for plotting. Remaining keyword arguments are handed down to ebisim.plotting.decorate_axes, cf. documentation thereof. If no arguments are provided, reasonable default values are injected.

Returns

matplotlib.Figure – Figure handle of the generated plot.

@numba.jitebisim.**rrxs_energyscan**(*element*, *e_kin=None*, *n=1000*)

Creates an array of RR cross sections for varying electron energies.

Parameters

- **element** (ebisim.Element) – An ebisim.Element object that holds the required physical information for cross section calculations.
- **e_kin** (*None or numpy.ndarray, optional*) – <eV> If e_kin is None, the range of sampling energies is chosen based on the binding enrgies of the element and energies are sampled on a logscale. If e_kin is an array with 2 elements, they are interpreted as the minimum and maximum sampling energy. If e_kin is an array with more than two values, the energies are taken as the sampling energies directly, by default None.
- **n** (*int, optional*) – The number of energy sampling points, if the sampling locations are not supplied by the user, by default 1000.

Returns

- **e_samp** (*numpy.ndarray*) – <eV> Array holding the sampling energies
- **xs_scan** (*numpy.ndarray*) – <m^2> Array holding the cross sections, where the row index corresponds to the charge state and the columns correspond to the different sampling energies

See also:

ebisim.xs.eixs_energyscan, *ebisim.xs.drxs_energyscan*

@numba.jitebisim.**rrxs_mat**(*element*, *e_kin*)

Radiative recombination cross section.

Parameters

- **element** (ebisim.Element) – An ebisim.Element object that holds the required physical information for cross section calculations.
- **e_kin** (*float*) – <eV> Kinetic energy of the impacting electron.

Returns

numpy.array – <m^2> The cross sections for each individual charge state, arranged in a matrix suitable for implementation of a rate equation like dN/dt = j * xs_matrix dot N. out[q, q] = - cross section of q+ ion out[q, q+1] = + cross section of (q+1)+ ion

See also:

ebisim.xs.rrxs_vec
Similar method with different output format.

@numba.jitebisim.**rrxs_vec**(*element*, *e_kin*)

Radiative recombination cross section according to [Kim1983].

Parameters

- **element** (ebisim.Element) – An ebisim.Element object that holds the required physical information for cross section calculations.
- **e_kin** (*float*) – <eV> Kinetic energy of the impacting electron.

Returns

numpy.ndarray – <m^2> The cross sections for each individual charge state, where the array-index corresponds to the charge state, i.e. out[q] ~ cross section of q+ ion.

References

See also:

ebisim.xs.rrxs_mat
Similar method with different output format.

2.1 Subpackages

2.1.1 ebisim.simulation package

This subpackage contains functions and classes provide an interface to run simulations and inspect their results.

class ebisim.simulation.**AdvancedModel**(*device*, *targets*, *bg_gases*, *options*, *lb*, *ub*, *nq*, *q*, *a*, *eixs*, *rrxs*, *drxs*, *cxxs_bggas*, *cxxs_trgts*)

Bases: tuple

The advanced model class is the base for ebisim.simulation.advanced_simulation. It acts as a fast datacontainer for the underlying rhs function which represents the right hand side of the differential equation system. Since it is jitcompiled using numba, care is required during instatiation.

Parameters

- **device** (ebisim.simulation.Device) – Container describing the EBIS/T and specifically the electron beam.
- **targets** (*numba.typed.List[ebisim.simulation.Target]*) – List of ebisim.simulation.Target for which charge breeding is simulated.
- **bg_gases** (*numba.typed.List[ebisim.simulation.BackgroundGas], optional*) – List of ebisim.simulation.BackgroundGas which act as CX partners, by default None.
- **options** (ebisim.simulation.ModelOptions, *optional*) – Switches for effects considered in the simulation, see default values of ebisim.simulation.ModelOptions.
- **lb** (*ndarray*) –
- **ub** (*ndarray*) –
- **nq** (*int*) –
- **q** (*ndarray*) –
- **a** (*ndarray*) –

- **eixs** (*ndarray*) –
- **rrxs** (*ndarray*) –
- **drxs** (*ndarray*) –
- **cxxs_bggas** (*Any*) –
- **cxxs_trgts** (*Any*) –

Create new instance of AdvancedModel(device, targets, bg_gases, options, lb, ub, nq, q, a, eixs, rrxs, drxs, cxxs_bggas, cxxs_trgts)

count(*value*, */*)

Return number of occurrences of value.

classmethod get(*device*, *targets*, *bg_gases=None*, *options=ModelOptions(EI=True, RR=True, CX=True, DR=False, SPITZER_HEATING=True, COLLISIONAL_THERMALISATION=True, ESCAPE_AXIAL=True, ESCAPE_RADIAL=True, RECOMPUTE_CROSS_SECTIONS=False, RADIAL_DYNAMICS=False, IONISATION_HEATING=True, OVERRIDE_FWHM=False, RADIAL_SOLVER_MAX_STEPS=500, RADIAL_SOLVER_REL_DIFF=0.001)*)

Parameters

- **device** (Device) –
- **targets** (*List[Element]*) –
- **bg_gases** (*Optional[List[BackgroundGas]]*) –
- **options** (ModelOptions) –

Return type
AdvancedModel

index(*value*, *start=0*, *stop=9223372036854775807*, */*)

Return first index of value.

Raises ValueError if the value is not present.

property a

Alias for field number 8

property bg_gases

Alias for field number 2

property cxxs_bggas

Alias for field number 12

property cxxs_trgts

Alias for field number 13

property device

Alias for field number 0

property drxs

Alias for field number 11

property eixs

Alias for field number 9

property lb

Alias for field number 4

property nq

Alias for field number 6

property options

Alias for field number 3

property q

Alias for field number 7

property rrxs

Alias for field number 10

property targets

Alias for field number 1

property ub

Alias for field number 5

class ebisim.simulation.**AdvancedResult**(*, *t*, *N*, *device*, *target*, *kbT*, *model*, *id_*, *res=None*, *rates=None*)

Bases: *BasicResult*[*Device*]

Instances of this class are containers for the results of ebisim advanced_simulations and contain a variety of convenience methods for simple plot generation etc.

Parameters

- **t** – An array holding the time step coordinates.
- **N** – An array holding the occupancy of each charge state at a given time.
- **device** – If coming from advanced_simulation, this is the machine / electron beam description.
- **target** – If coming from advanced_simulation, this is the target represented by this Result.
- **kbT** – An array holding the temperature of each charge state at a given time.
- **model** – The AdvancedModel instance that was underlying the advanced_simulation
- **id** – The position of this target in the list of all simulated targets.
- **res** – The result object returned by scipy.integrate.solve_ivp. This can contain useful information about the solver performance etc. Refer to scipy documentation for details.
- **rates** – A dictionary containing the different breeding rates in arrays shaped like N.

abundance_at_time(*t*)

Yields the abundance of each charge state at a given time

Parameters

t (*float*) – <s> Point of time to evaluate.

Returns

Abundance of each charge state, array index corresponds to charge state.

Return type

ndarray

plot(*relative=False*, ***kwargs*)

Plot the charge state evolution of this result object.

Parameters

- **relative** (*bool*) – Flags whether the absolute numbers or a relative fraction should be plotted at each timestep, by default False.
- **kwargs** (*Any*) – Keyword arguments are handed down to ebisim.plotting.plot_generic_evolution, cf. documentation thereof. If no arguments are provided, reasonable default values are injected.

Returns

Figure handle of the generated plot.

Raises

ValueError – If the required data (self.t, self.N) is not available, i.e. corresponding attributes of this Result instance have not been set correctly.

Return type

Figure

plot_charge_states(*relative=False*, ***kwargs*)

Plot the charge state evolution of this result object.

Parameters

- **relative** (*bool*) – Flags whether the absolute numbers or a relative fraction should be plotted at each timestep, by default False.
- **kwargs** (*Any*) – Keyword arguments are handed down to ebisim.plotting.plot_generic_evolution, cf. documentation thereof. If no arguments are provided, reasonable default values are injected.

Returns

Figure handle of the generated plot.

Raises

ValueError – If the required data (self.t, self.N) is not available, i.e. corresponding attributes of this Result instance have not been set correctly.

Return type

Figure

plot_energy_density(***kwargs*)

Plot the energy density evolution of this result object.

Parameters

kwargs (*Any*) – Keyword arguments are handed down to ebisim.plotting.plot_generic_evolution, cf. documentation thereof. If no arguments are provided, reasonable default values are injected.

Returns

Figure handle of the generated plot.

Raises

ValueError – If the required data (self.t, self.kbT) is not available, i.e. corresponding attributes of this Result instance have not been set correctly.

Return type

Figure

plot_radial_distribution_at_time(*t*, ***kwargs*)

Plot the radial ion distribution at time t.

Parameters

- **t** (*float*) – <s> Point of time to evaluate.
- **kwargs** (*Any*) – Keyword arguments are handed down to ebisim.plotting.plot_radial_distribution and ebisim.plotting.plot_generic_evolution, cf. documentation thereof. If no arguments are provided, reasonable default values are injected.

Returns

Figure handle of the generated plot.

Return type

Figure

plot_rate(*rate_key*, *dens_threshold=0.001*, ***kwargs*)

Plots the requested ionisation- or energy flow rates.

Parameters

- **rate_key** (Rate) – The key identifying the rate to be plotted. See ebisim.simulation.Rate for valid values.
- **dens_threshold** (*float*) – If given temperatures are only plotted where the particle denisty is larger than the threshold value.
- **kwargs** – Keyword arguments are handed down to ebisim.plotting.plot_generic_evolution, cf. documentation thereof. If no arguments are provided, reasonable default values are injected.

Returns

Figure handle of the generated plot

Raises

ValueError – If the required data (self.rates) is not available, or an invalid key is requested.

Return type

Figure

plot_temperature(*dens_threshold=0.001*, ***kwargs*)

Plot the temperature evolution of this result object.

Parameters

- **dens_threshold** (*float*) – If given temperatures are only plotted where the particle denisty is larger than the threshold value.
- **kwargs** (*Any*) – Keyword arguments are handed down to ebisim.plotting.plot_generic_evolution, cf. documentation thereof. If no arguments are provided, reasonable default values are injected.

Returns

Figure handle of the generated plot

Raises

ValueError – If the required data (self.t, self.kbT) is not available, i.e. corresponding attributes of this Result instance have not been set correctly.

Return type

Figure

radial_distribution_at_time(*t*)

Yields the radial distribution information at time

Parameters
t (*float*) – <s> Point of time to evaluate.

Returns

- *phi* – Radial potential
- *n3d* – On axis 3D density for each charge state.
- *shapes* – The Boltzmann shape factors for each charge state.

Return type
Tuple[*ndarray*, *ndarray*, *ndarray*]

temperature_at_time(*t*)

Yields the temperature of each charge state at a given time

Parameters
t (*float*) – <s> Point of time to evaluate.

Returns

- *<eV>*
- *Temperature of each charge state, array index corresponds to charge state.*

Return type
ndarray

times_of_highest_abundance()

Yields the point of time with the highest abundance for each charge state

Returns

- *<s>*
- *Array of times.*

Return type
ndarray

class ebisim.simulation.**BackgroundGas**(*name*, *ip*, *n0*)

Bases: tuple

Use the static *get()* factory methods to create instances of this class.

Simple datacontainer for a background gas for advanced simulations. A background gas only acts as a charge exchange partner to the Targets in the simulation.

See also:

ebisim.simulation.BackgroundGas.get

Create new instance of BackgroundGas(name, ip, n0)

Parameters

- **name** (*str*) –
- **ip** (*float*) –
- **n0** (*float*) –

count(*value*, /)

Return number of occurrences of value.

classmethod get(*element*, *p*, *T=300.0*)

Factory method for defining a background gas.

Parameters

- **element** (*Union[Element, str, int]*) – An instance of the Element class, or an identifier for the element, i.e. either its name, symbol or proton number.
- **p** (*float*) – <mbar> Gas pressure.
- **T** (*float*) – <K> Gas temperature, by default 300 K (Room temperature)

Returns

ebisim.simulation.BackgroundGas – Ready to use BackgroundGas specification.

Return type

BackgroundGas

index(*value*, *start=0*, *stop=9223372036854775807*, /)

Return first index of value.

Raises ValueError if the value is not present.

property ip

<eV> Ionisation potential of this Gas.

property n0

<1/m^3> Gas number density.

property name

Name of the element.

class ebisim.simulation.**BasicResult**(***, *t*, *N*, *device*, *target*, *res=None*)

Bases: Generic[GenericDevice]

Instances of this class are containers for the results of ebisim basic_simulations and contain a variety of convenience methods for simple plot generation etc.

Parameters

- **t** – An array holding the time step coordinates.
- **N** – An array holding the occupancy of each charge state at a given time.
- **device** – If coming from advanced_simulation, this is the machine / electron beam description.
- **target** – If coming from advanced_simulation, this is the target represented by this Result.
- **res** – The result object returned by scipy.integrate.solve_ivp. This can contain useful information about the solver performance etc. Refer to scipy documentation for details.

abundance_at_time(*t*)

Yields the abundance of each charge state at a given time

Parameters

t (*float*) – <s> Point of time to evaluate.

Returns

Abundance of each charge state, array index corresponds to charge state.

Return type
ndarray

`plot`(*relative=False*, ***kwargs*)

Alias for plot_charge_states

Parameters

- **relative** (*bool*) –
- **kwargs** (*Any*) –

Return type
Figure

`plot_charge_states`(*relative=False*, ***kwargs*)

Plot the charge state evolution of this result object.

Parameters

- **relative** (*bool*) – Flags whether the absolute numbers or a relative fraction should be plotted at each timestep, by default False.
- **kwargs** (*Any*) – Keyword arguments are handed down to ebisim.plotting.plot_generic_evolution, cf. documentation thereof. If no arguments are provided, reasonable default values are injected.

Returns
Figure handle of the generated plot.

Raises
ValueError – If the required data (self.t, self.N) is not available, i.e. corresponding attributes of this Result instance have not been set correctly.

Return type
Figure

`times_of_highest_abundance`()

Yields the point of time with the highest abundance for each charge state

Returns

- <*s*>
- *Array of times.*

Return type
ndarray

`class ebisim.simulation.Device`(*current*, *e_kin*, *r_e*, *length*, *j*, *v_ax*, *v_ax_sc*, *v_ra*, *b_ax*, *r_dt*, *r_dt_bar*, *fwhm*, *rad_grid*, *rad_fd_l*, *rad_fd_d*, *rad_fd_u*, *rad_phi_uncomp*, *rad_phi_ax_barr*, *rad_re_idx*)

Bases: `tuple`

Use the static *get()* factory methods to create instances of this class.

Objects of this class are used to pass important EBIS/T parameters into the simulation.

Create new instance of Device(current, e_kin, r_e, length, j, v_ax, v_ax_sc, v_ra, b_ax, r_dt, r_dt_bar, fwhm, rad_grid, rad_fd_l, rad_fd_d, rad_fd_u, rad_phi_uncomp, rad_phi_ax_barr, rad_re_idx)

Parameters

- **current** (*float*) –

- **e_kin** (*float*) –
- **r_e** (*float*) –
- **length** (*Optional[float]*) –
- **j** (*float*) –
- **v_ax** (*float*) –
- **v_ax_sc** (*float*) –
- **v_ra** (*float*) –
- **b_ax** (*float*) –
- **r_dt** (*float*) –
- **r_dt_bar** (*float*) –
- **fwhm** (*float*) –
- **rad_grid** (*ndarray*) –
- **rad_fd_l** (*ndarray*) –
- **rad_fd_d** (*ndarray*) –
- **rad_fd_u** (*ndarray*) –
- **rad_phi_uncomp** (*ndarray*) –
- **rad_phi_ax_barr** (*ndarray*) –
- **rad_re_idx** (*int*) –

count(*value*, */*)

Return number of occurrences of value.

classmethod get(**, current, e_kin, r_e, v_ax, b_ax, r_dt, length=None, v_ra=None, j=None, fwhm=None, n_grid=400, r_dt_bar=None*)

Factory method for defining a device. All arguments are keyword only to reduce the chance of mixing them up.

Parameters

- **current** (*float*) – <A> Electron beam current.
- **e_kin** (*float*) – <eV> Uncorrected electron beam energy.
- **r_e** (*float*) – <m> Electron beam radius.
- **v_ax** (*float*) – <V> Axial barrier bias.
- **b_ax** (*float*) – <T> Axial magnetic flux density in the trap.
- **r_dt** (*float*) – <m> Drift tube radius.
- **length** (*Optional[float]*) – <m> Trap length -> Currently not used in simulations.
- **v_ra** (*Optional[float]*) – <V> Override for radial trap depth. Only effective if ModelOptions.RADIAL_DYNAMICS=False.
- **j** (*Optional[float]*) – <A/cm^2> Override for current density.
- **fwhm** (*Optional[float]*) – <eV> Override for the electron beam energy spread. Only effective if ModelOptions.RADIAL_DYNAMICS=False.
- **n_grid** (*int*) – Approximate number of nodes for the radial mesh.

- **r_dt_bar** (*Optional[float]*) – Radius of the barrier drift tubes. If not passed, assuming equal radius as trap drift tube.

Returns
ebisim.simulation.Device – The populated device object.

Return type
Device

index(*value*, *start=0*, *stop=9223372036854775807*, */*)

Return first index of value.

Raises ValueError if the value is not present.

property b_ax

<T> Axial magnetic field density.

property current

<A> Beam current.

property e_kin

<eV> Uncorrected beam energy.

property fwhm

<eV> Electron beam energy spread.

property j

<A/cm^2> Current density.

property length

<m> Trap length.

property r_dt

<m> Drift tube radius.

property r_dt_bar

<m> Drift tube radius of the barrier drift tubes.

property r_e

<m> Beam radius.

property rad_fd_d

Diagonal vector of finite difference scheme.

property rad_fd_l

Lower diagonal vector of finite difference scheme.

property rad_fd_u

Upper diagonal vector of finite difference scheme.

property rad_grid

<m> Radial grid for finite difference computations.

property rad_phi_ax_barr

<V> Radial potential of the electron beam in the barrier tube

property rad_phi_uncomp

<V> Radial potential of the electron beam.

property rad_re_idx

Index of the radial grid point closest to r_e

property v_ax

<V> Axial barrier voltage.

property v_ax_sc

<V> Axial barrier space charge correction.

property v_ra

<V> Radial barrier voltage.

class ebisim.simulation.**EnergyScanResult**(*sim_kwargs*, *energies*, *results*)

Bases: object

This class provides a convenient interface to access and evaluate the the results of the ebisim.simulation.energy_scan function. Abundances at arbitrary times are provided by performing linear interpolations of the solutions of the rate equation.

Parameters

- **sim_kwargs** (*dict*) – The sim_kwargs dictionary as provided during the call to ebisim.simulation.energy_scan.
- **energies** (*numpy.array*) – <eV> A sorted array containing the energies at which the energy scan has been evaluated.
- **results** (*list of ebisim.simulation.Result*) – A list of Result objects holding the results of each individual simulation

abundance_at_time(*t*)

Provides information about the charge state distribution at a given time for all energies.

Parameters
t (*float*) – <s> Point of time to evaluate.

Returns

- **energies** (*numpy.array*) – The evaluated energies.
- **abundance** (*numpy.array*) – Contains the abundance of each charge state (rows) for each energy (columns).

Raises
ValueError – If 't' is not part of the simulated time domain

abundance_of_cs(*cs*)

Provides information about the abundance of a single charge states at all simulated times and energies.

Parameters
cs (*int*) – The charge state to evaluate.

Returns

- **energies** (*numpy.array*) – The evaluated energies.
- **times** (*numpy.array*) – The evaluated timesteps.
- **abundance** (*numpy.array*) – Abundance of charge state 'cs' at given times (rows) and energies (columns).

Raises
ValueError – If 'cs' is not a sensible charge state for the performed simulation.

get_result(*e_kin*)

Returns the result object corresponding to the simulation at a given energy

Parameters
e_kin (*float*) – <eV> The energy of the simulation one wishes to retrieve.

Returns
ebisim.simulation.Result – The Result object for the polled scan step.

Raises
ValueError – If the polled energy is not available.

plot_abundance_at_time(*t*, *cs=None*, ***kwargs*)

Produces a plot of the charge state abundance for different energies at a given time.

Parameters

- **t** (*float*) – <s> Point of time to evaluate.
- **cs** (*list or None, optional*) – If None, all charge states are plotted. By supplying a list of int it is possible to filter the charge states that should be plotted. By default None.
- ****kwargs** – Keyword arguments are handed down to ebisim.plotting.plot_energy_scan, cf. documentation thereof. If no arguments are provided, reasonable default values are injected.

Returns
matplotlib.Figure – Figure handle of the generated plot.

plot_abundance_of_cs(*cs*, ***kwargs*)

Produces a 2D contour plot of the charge state abundance for all simulated energies and times.

Parameters

- **cs** (*int*) – The charge state to plot.
- ****kwargs** – Keyword arguments are handed down to ebisim.plotting.plot_energy_scan, cf. documentation thereof. If no arguments are provided, reasonable default values are injected.

Returns
matplotlib.Figure – Figure handle of the generated plot.

class ebisim.simulation.**ModelOptions**(*EI=True*, *RR=True*, *CX=True*, *DR=False*, *SPITZER_HEATING=True*, *COLLISIONAL_THERMALISATION=True*, *ESCAPE_AXIAL=True*, *ESCAPE_RADIAL=True*, *RECOMPUTE_CROSS_SECTIONS=False*, *RADIAL_DYNAMICS=False*, *IONISATION_HEATING=True*, *OVERRIDE_FWHM=False*, *RADIAL_SOLVER_MAX_STEPS=500*, *RADIAL_SOLVER_REL_DIFF=0.001*)

Bases: tuple

An instance of ModelOptions can be used to turn on or off certain effects in an advanced simulation.

Create new instance of ModelOptions(EI, RR, CX, DR, SPITZER_HEATING, COLLISIONAL_THERMALISATION, ESCAPE_AXIAL, ESCAPE_RADIAL, RECOMPUTE_CROSS_SECTIONS, RADIAL_DYNAMICS, IONISATION_HEATING, OVERRIDE_FWHM, RADIAL_SOLVER_MAX_STEPS, RADIAL_SOLVER_REL_DIFF)

Parameters

- **EI** (*bool*) –
- **RR** (*bool*) –
- **CX** (*bool*) –
- **DR** (*bool*) –
- **SPITZER_HEATING** (*bool*) –
- **COLLISIONAL_THERMALISATION** (*bool*) –
- **ESCAPE_AXIAL** (*bool*) –
- **ESCAPE_RADIAL** (*bool*) –
- **RECOMPUTE_CROSS_SECTIONS** (*bool*) –
- **RADIAL_DYNAMICS** (*bool*) –
- **IONISATION_HEATING** (*bool*) –
- **OVERRIDE_FWHM** (*bool*) –
- **RADIAL_SOLVER_MAX_STEPS** (*int*) –
- **RADIAL_SOLVER_REL_DIFF** (*float*) –

count(*value*, */*)

Return number of occurrences of value.

index(*value*, *start=0*, *stop=9223372036854775807*, */*)

Return first index of value.

Raises ValueError if the value is not present.

property COLLISIONAL_THERMALISATION

Switch for ion-ion thermalisation, default True.

property CX

Switch for charge exchange, default True.

property DR

Switch for dielectronic recombination, default False.

property EI

Switch for electron impact ionisation, default True.

property ESCAPE_AXIAL

Switch for axial escape from the trap, default True.

property ESCAPE_RADIAL

Switch for radial escape from the trap, default True.

property IONISATION_HEATING

Switch for ionisation heating/recombination cooling

property OVERRIDE_FWHM

If set use FWHM from device definition instead of computed value.

property RADIAL_DYNAMICS

Switch for effects of radial ion cloud extent. May be computationally very intensive

property RADIAL_SOLVER_MAX_STEPS

Alias for field number 12

property RADIAL_SOLVER_REL_DIFF

Alias for field number 13

property RECOMPUTE_CROSS_SECTIONS

Switch deciding whether EI, RR, and DR cross sections are recomputed on each call of the differential equation system. Advisable if electron beam energy changes over time and sharp transitions are expected, e.g. DR or ionisation thresholds for a given shell. Default False.

property RR

Switch for radiative recombination, default True.

property SPITZER_HEATING

Switch for Spitzer- or electron-heating, default True.

class ebisim.simulation.**Rate**(*value*)

Bases: IntEnum

Enum for conveniently identifying rates produced in advanced simulations

AX_CO = 105

CHARGE_EXCHANGE = 104

COLLISIONAL_THERMALISATION = 301

COLLISION_RATE_SELF = 402

COLLISION_RATE_TOTAL = 401

CX = 104

DIELECTRONIC_RECOMBINATION = 103

DR = 103

EI = 101

ELECTRON_IONISATION = 101

E_KIN_FWHM = 532

E_KIN_MEAN = 531

F_EI = 501

IONISATION_HEAT = 303

LOSSES_AXIAL_COLLISIONAL = 105

LOSSES_RADIAL_COLLISIONAL = 107

OVERLAP_FACTORS_EBEAM = 501

RADIATIVE_RECOMBINATION = 102

RA_CO = 107

RR = 102

SPITZER_HEATING = 302

TRAPPING_PARAMETER_AXIAL = 521

TRAPPING_PARAMETER_RADIAL = 522

TRAP_DEPTH_AXIAL = 523

TRAP_DEPTH_RADIAL = 524

T_AX_CO = 205

T_COLLISIONAL_THERMALISATION = 301

T_CT = 301

T_LOSSES_AXIAL_COLLISIONAL = 205

T_LOSSES_RADIAL_COLLISIONAL = 207

T_RA_CO = 207

T_SH = 302

T_SPITZER_HEATING = 302

V_AX = 523

V_RA = 524

W_AX = 521

W_RA = 522

ebisim.simulation.**advanced_simulation**(*device*, *targets*, *t_max*, *bg_gases=None*, *options=None*, *rates=False*, *solver_kwargs=None*, *verbose=True*, *n_threads=1*)

Interface for performing advanced charge breeding simulations.

For a list of effects refer to *ebisim.simulation.ModelOptions*.

Parameters

- **device** (Device) – Container describing the EBIS/T and specifically the electron beam.
- **targets** (*Union[Element, List[Element]]*) – Target(s) for which charge breeding is simulated.
- **t_max** (*float*) – <s> Simulated breeding time
- **bg_gases** (*Optional[Union[BackgroundGas, List[BackgroundGas]]]*) – Background gas(es) which act as CX partners.
- **rates** (*bool*) – If true a 'second run' is performed to store the rates, this takes extra time and can create quite a bit of data.
- **options** (*Optional[ModelOptions]*) – Switches for effects considered in the simulation, see default values of ebisim.simulation.ModelOptions.
- **solver_kwargs** (*Optional[Dict[str, Any]]*) – If supplied these keyword arguments are unpacked in the solver call. Refer to the documentation of scipy.integrate.solve_ivp for more information. By default None.

- **verbose** (*bool*) – Print a little progress indicator and some status messages, by default True.
- **n_threads** (*int*) – How many threads to use (mostly for jacbion estimation which can evaluate the RHS in parallel with different inputs.)

Returns

- *An instance of the AdvancedResult class, holding the simulation parameters, timesteps and*
- *charge state distribution including the species temperature.*

Return type
Union[AdvancedResult, *Tuple*[AdvancedResult, ...]]

ebisim.simulation.**basic_simulation**(*element*, *j*, *e_kin*, *t_max*, *dr_fwhm=None*, *N_initial=None*, *CNI=False*, *solver_kwargs=None*)

Interface for performing basic charge breeding simulations.

These simulations only include the most important effects, i.e. electron ionisation, radiative recombination and optionally dielectronic recombination (for those transitions whose data is available in the resource directory). All other effects are ignored.

Continuous Neutral Injection (CNI) can be activated on demand.

The results only represent the proportions of different charge states, not actual densities.

Parameters

- **element** (*Union[Element, str, int]*) – An instance of the Element class, or an identifier for the element, i.e. either its name, symbol or proton number.
- **j** (*float*) – <A/cm^2> Current density
- **e_kin** (*float*) – <eV> Electron beam energy
- **t_max** (*float*) – <s> Simulated breeding time
- **dr_fwhm** (*Optional[float]*) – <eV> If a value is given, determines the energy spread of the electron beam (in terms of Full Width Half Max) and hence the effective width of DR resonances. Otherwise DR is excluded from the simulation.
- **N_initial** (*Optional[ndarray]*) – Determines the initial charge state distribution if given, must have Z + 1 entries, where the array index corresponds to the charge state. If no value is given the distribution defaults to 100% of 1+ ions at t = 0 s, or a small amount of neutral atoms in the case of CNI.
- **CNI** (*bool*) – If Continuous Neutral Injection is activated, the neutrals are assumed to form an infinite reservoir. Their absolute number will not change over time and hence they act as a constant source of new singly charged ions. Therefore the absolute amount of ions increases over time.
- **solver_kwargs** (*Optional[Dict[str, Any]]*) – If supplied these keyword arguments are unpacked in the solver call. Refer to the documentation of scipy.integrate.solve_ivp for more information.

Returns

- *An instance of the BasicResult class, holding the simulation parameters, timesteps and*
- *charge state distribution.*

Return type
BasicResult

@numba.jitebisim.simulation.**boltzmann_radial_potential_linear_density**(*r*, *rho_0*, *nl*, *kT*, *q*, *first_guess=None*, *ldu=None*)

Solves the Boltzmann Poisson equation for a static background charge density rho_0 and particles with line number density n, Temperature kT and charge state q.

Below, nRS and nCS are the number of radial sampling points and charge states.

Solution is found through Newton Raphson iterations, cf. [PICNPSb].

Parameters

- **r** (*np.ndarray*) – <m> Radial grid points, with r[0] = 0, r[-1] = r_max.
- **rho_0** (*np.ndarray (nRS,)*) – <C/m^3> Static charge density at r.
- **nl** (*np.ndarray (1, nCS)*) – <1/m> Line number density of Boltzmann distributed particles.
- **kT** (*np.ndarray (1, nCS)*) – <eV> Temperature of Boltzmann distributed particles.
- **q** (*np.ndarray (1, nCS)*) – Charge state of Boltzmann distributed particles.
- **ldu** (*(np.ndarray, np.ndarray, np.ndarray)*) – The lower diagonal, diagonal, and upper diagonal vector describing the finite difference scheme. Can be provided if they have been pre-computed.

Returns

- **phi** (*np.ndarray (nRS,)*) – <V> Potential at r.
- **nax** (*np.ndarray (1, nCS)*) – <1/m^3> On axis number densities.
- **shape** (*np.ndarray (nRS, nCS)*) – Radial shape factor of the particle distributions.

References

@numba.jitebisim.simulation.**boltzmann_radial_potential_linear_density_ebeam**(*r*, *current*, *r_e*, *e_kin*, *nl*, *kT*, *q*, *first_guess=None*, *ldu=None*, *max_step=500*, *rel_diff=0.001*)

Solves the Boltzmann Poisson equation for a static background charge density rho_0 and particles with line number density n, Temperature kT and charge state q. The electron beam charge density is computed from a uniform current density and the iteratively corrected velocity profile of the electron beam.

Below, nRS and nCS are the number of radial sampling points and charge states.

Solution is found through Newton Raphson iterations, cf. [PICNPS].

Parameters

- **r** (*np.ndarray*) – <m> Radial grid points, with r[0] = 0, r[-1] = r_max.
- **current** (*float*) – <A> Electron beam current (positive sign).
- **r_e** (*float*) – <m> Electron beam radius.
- **e_kin** (*float*) – <eV> Uncorrected electron beam energy.
- **nl** (*np.ndarray (1, nCS)*) – <1/m> Line number density of Boltzmann distributed particles.

- **kT** (*np.ndarray (1, nCS)*) – <eV> Temperature of Boltzmann distributed particles.
- **q** (*np.ndarray (1, nCS)*) – Charge state of Boltzmann distributed particles.
- **ldu** (*(np.ndarray, np.ndarray, np.ndarray)*) – The lower diagonal, diagonal, and upper diagonal vector describing the finite difference scheme. Can be provided if they have been pre-computed.

Returns

- **phi** (*np.ndarray (nRS,)*) – <V> Potential at r.
- **nax** (*np.ndarray (1, nCS)*) – <1/m^3> On axis number densities.
- **shape** (*np.ndarray (nRS, nCS)*) – Radial shape factor of the particle distributions.

References

@numba.jitebisim.simulation.**boltzmann_radial_potential_onaxis_density**(*r*, *rho_0*, *n*, *kT*, *q*, *first_guess=None*, *ldu=None*)

Solves the Boltzmann Poisson equation for a static background charge density rho_0 and particles with a fixed on axis density n, Temperature kT and charge state q.

Below, nRS and nCS are the number of radial sampling points and charge states.

Solution is found through Newton iterations, cf. [PICNPSa].

Parameters

- **r** (*np.ndarray*) – <m> Radial grid points, with r[0] = 0, r[-1] = r_max.
- **rho_0** (*np.ndarray (nRS,)*) – <C/m^3> Static charge density at r.
- **n** (*np.ndarray (1, nCS)*) – <1/m> On axis density of Boltzmann distributed particles.
- **kT** (*np.ndarray (1, nCS)*) – <eV> Temperature of Boltzmann distributed particles.
- **q** (*np.ndarray (1, nCS)*) – Charge state of Boltzmann distributed particles.
- **ldu** (*(np.ndarray, np.ndarray, np.ndarray)*) – The lower diagonal, diagonal, and upper diagonal vector describing the finite difference scheme. Can be provided if they have been pre-computed.

Returns

- **phi** (*np.ndarray (nRS,)*) – <V> Potential at r.
- **nax** (*np.ndarray (1, nCS)*) – <1/m^3> On axis number densities.
- **shape** (*np.ndarray (nRS, nCS)*) – Radial shape factor of the particle distributions.

References

ebisim.simulation.**energy_scan**(*sim_func*, *sim_kwargs*, *energies*, *parallel=False*)

This function provides a convenient way to repeat the same simulation for a number of different electron beam energies. This can reveal variations in the charge state balance due to weakly energy dependent ionisation cross sections or even resonant phenomena like dielectronic recombination.

Parameters

- **sim_func** (*callable*) – The function handle for the simulation e.g. ebisim.simulation.basic_simulation.
- **sim_kwargs** (*dict*) – A dictionary containing all the required and optional parameters of the simulations (except for the kinetic electron energy) as key value pairs. This is unpacked in the function call.
- **energies** (*list or numpy.array*) – A list or array of the energies at which the simulation should be performed.
- **parallel** (*bool, optional*) – Determine whether multiple simulations should be run in parallel using pythons multiprocessing.pool. This may accelerate the scan when performing a large number of simulations. By default False.

Returns

ebisim.simulation.EnergyScanResult – An object providing convenient access to the generated scan data.

@numba.jitebisim.simulation.**fd_system_nonuniform_grid**(*r*)

Sets up the three diagonal vectors for a finite Poisson problem with radial symmetry on a nonuniform grid. d phi/dr = 0 at r = 0, and phi = phi0 at r = (n-1) * dr = r_max The finite differences are developed according to [Sundqvist1970].

Parameters

r (*np.ndarray*) – <m> Radial grid points, with r[0] = 0, r[-1] = r_max.

Returns

- **l** (*np.ndarray*) – Lower diagonal vector.
- **d** (*np.ndarray*) – Diagonal vector.
- **u** (*np.ndarray*) – Upper diagonal vector.

References

See also:

ebisim.simulation._radial_dist.fd_system_uniform_grid

@numba.jitebisim.simulation.**fd_system_uniform_grid**(*r*)

Sets up the three diagonal vectors for a finite Poisson problem with radial symmetry on a uniform grid. d phi/dr = 0 at r = 0, and phi = phi0 at r = (n-1) * dr = r_max

Parameters

r (*np.ndarray*) – <m> Radial grid points, must be evenly spaced, with r[0] = 0, r[-1] = r_max.

Returns

- **l** (*np.ndarray*) – Lower diagonal vector.
- **d** (*np.ndarray*) – Diagonal vector.

- **u** (*np.ndarray*) – Upper diagonal vector.

See also:

`ebisim.simulation._radial_dist.fd_system_nonuniform_grid`

`@numba.jit`ebisim.simulation.**heat_capacity**(*r*, *phi*, *q*, *kT*)

Computes the heat capacity of an ion cloud with a given charge state and temperature inside the external potenital phi(r). According to Lu, Currell cloud expansion.

Parameters

- **r** (*np.ndarray*) – <m> Radial grid points, with r[0] = 0, r[-1] = r_max.
- **phi** (*np.ndarray*) – <V> Potential at r.
- **q** (*int*) – Ion charge state.
- **kT** (*float*) – <eV> Ion temperature

Returns

float – <eV/eV> Constant volume heat capacity

`@numba.jit`ebisim.simulation.**radial_potential_nonuniform_grid**(*r*, *rho*)

Solves the radial Poisson equation on a nonuniform grid. Boundary conditions are d phi/dr = 0 at r = 0 and phi(rmax) = 0

Parameters

- **r** (*np.ndarray*) – <m> Radial grid points, with r[0] = 0, r[-1] = r_max.
- **rho** (*np.ndarray*) – <C/m^3> Charge density at r.

Returns

np.ndarray – Potential at r.

`@numba.jit`ebisim.simulation.**radial_potential_uniform_grid**(*r*, *rho*)

Solves the radial Poisson equation on a uniform grid. Boundary conditions are d phi/dr = 0 at r = 0 and phi(rmax) = 0

Parameters

- **r** (*np.ndarray*) – <m> Radial grid points, must be evenly spaced, with r[0] = 0, r[-1] = r_max.
- **rho** (*np.ndarray*) – <C/m^3> Charge density at r.

Returns

np.ndarray – <V> Potential at r.

`@numba.jit`ebisim.simulation.**tridiagonal_matrix_algorithm**(*l*, *d*, *u*, *b*)

Tridiagonal Matrix Algorithm [TDMA]. Solves a system of equations M x = b for x, where M is a tridiagonal matrix.

M = np.diag(d) + np.diag(u[:-1], 1) + np.diag(l[1:], -1)

Parameters

- **l** (*np.ndarray*) – Lower diagonal vector l[i] = M[i, i-1].
- **d** (*np.ndarray*) – Diagonal vector d[i] = M[i, i].
- **u** (*np.ndarray*) – Upper diagonal vector u[i] = M[i, i+1].
- **b** (*np.ndarray*) – Inhomogenety term.

Returns
x (*np.ndarray*) – Solution of the linear system.

References

2.2 Submodules

2.2.1 ebisim.beams module

This module contains tools for computing characteristic quantities of electron beams typically found in an electron beam ion source / trap.

class ebisim.beams.**ElectronBeam**(*cur*, *b_d*, *r_d*, *b_c*, *r_c*, *t_c*)

Bases: object

This class contains logic that allows estimating the space charge corrected energy of an electron beam and the resulting energy spread

Parameters

- **cur** (*float*) – <A> Electron beam current.
- **b_d** (*float*) – <T> Magnetic flux density in the centre (drift tubes) of the EBIS.
- **r_d** (*float*) – <m> Drift tube radius.
- **b_c** (*float*) – <T> Magentic flux density on the cathode surface
- **r_c** (*float*) – <m> Cathode radius.
- **t_c** (*float*) – <K> Cathode temperature.

characteristic_potential(*e_kin*)

Returns the characteristic potential due to spacecharge

Parameters
e_kin (*float*) – <eV> Electron kinetic energy.

Returns
float – <V> Characteristic potential.

herrmann_radius(*e_kin*)

Returns the Hermann radius of an electron beam with the given machine parameters

Parameters
e_kin (*float*) – <eV> Electron kinetic energy.

Returns
float – <m> Hermann radius.

space_charge_correction(*e_kin*, *r=0*)

Returns the space charge correction at a given radius r, current and electron beam energy This is done by iteratively computing the spacecharge correction and the hermann radius until the realtive change in space charge correction is < 1e-6.

Parameters

- **e_kin** (*float*) – <eV> Electron kinetic energy.
- **r** (*float, optional*) – Distance from the axis at which to evaluate the correction, by default 0.

Returns
float – <V> Space charge correction.

Raises
ValueError – If r is larger than the drift tube radius.

property current

Current of the electron beam.

2.2.2 ebisim.elements module

This module most notably implements the Element class, which serves as the main container for physical data going into the ebisim computations.

Besides that, there are some small helper functions to translate certain element properties, which may offer convenience to the user.

class ebisim.elements.**Element**(*z*, *symbol*, *name*, *a*, *ip*, *e_cfg*, *e_bind*, *rr_z_eff*, *rr_n_0_eff*, *dr_cs*, *dr_e_res*, *dr_strength*, *ei_lotz_a*, *ei_lotz_b*, *ei_lotz_c*, *n=None*, *kT=None*, *cx=True*)

Bases: tuple

The Element class is one of the main data structures in ebisim. Virtually any function relies on information provided in this data structure. The leading fields of the underlying tuple contain physcial properties, whereas the *n kT* and *cx* fields are optional and only required for advanced simulations.

Instead of populating the fields manually, the user should choose one of the factory functions that meets their needs best.

For basic simulations and cross sections calculations only the physical / chemical properties of and element are needed. In these cases use the generic *get()* method to create instances of this class.

Advanced simulations require additional information about the initial particle densities, temperature and participation in charge exchange. The user will likely want to chose between the *get_ions()* and *get_gas()* methods, which offer a convenient interface for generating this data based on simple parameters. If these functions are not flexible enough, the *get()* method can be used to populate the required fields manually.

This class is derived from collections.namedtuple which facilitates use with numba-compiled functions.

See also:

ebisim.elements.Element.get, *ebisim.elements.Element.get_ions*, *ebisim.elements.Element.get_gas*

Create new instance of Element(z, symbol, name, a, ip, e_cfg, e_bind, rr_z_eff, rr_n_0_eff, dr_cs, dr_e_res, dr_strength, ei_lotz_a, ei_lotz_b, ei_lotz_c, n, kT, cx)

Parameters

- **z** (*int*) –
- **symbol** (*str*) –
- **name** (*str*) –
- **a** (*float*) –
- **ip** (*float*) –
- **e_cfg** (*ndarray*) –
- **e_bind** (*ndarray*) –
- **rr_z_eff** (*ndarray*) –

- **rr_n_0_eff** (*ndarray*) –
- **dr_cs** (*ndarray*) –
- **dr_e_res** (*ndarray*) –
- **dr_strength** (*ndarray*) –
- **ei_lotz_a** (*ndarray*) –
- **ei_lotz_b** (*ndarray*) –
- **ei_lotz_c** (*ndarray*) –
- **n** (*Optional[ndarray]*) –
- **kT** (*Optional[ndarray]*) –
- **cx** (*bool*) –

classmethod as_element(*element*)

If *element* is already an instance of *Element* it is returned. If *element* is a string or int identyfying an element an appropriate *Element* instance is returned.

Parameters

element (*Union[Element, str, int]*) – An instance of the Element class, or an identifier for the element, i.e. either its name, symbol or proton number.

Returns

An instance of Element reflecting the input value.

Return type

Element

count(*value*, */*)

Return number of occurrences of value.

classmethod get(*element_id*, *a=None*, *n=None*, *kT=None*, *cx=True*)

Factory method to create instances of the Element class.

Parameters

- **element_id** (*Union[str, int]*) – The full name, abbreviated symbol, or proton number of the element of interest.
- **a** (*Optional[float]*) – If provided sets the mass number of the Element object otherwise a reasonable value is chosen automatically.
- **n** (*Optional[ndarray]*) – <1/m> Only needed for advanced simulations! Array holding the initial ion line densities of each charge state. If provided, has to be an array of length Z+1, where Z is the nuclear charge.
- **kT** (*Optional[ndarray]*) – <eV> Only needed for advanced simulations! Array holding the initial ion line densities of each charge state. If provided, has to be an array of length Z+1, where Z is the nuclear charge.
- **cx** (*bool*) – Only needed for advanced simulations! Boolean flag determining whether the neutral particles of this element contribute to charge exchange with ions.

Returns

An instance of Element with the user-supplied and generated data.

Raises

- **ValueError** – If the Element could not be identified or a meaningless mass number is provided.
- **ValueError** – If the passed arrays for n or kT have the wrong shape.

Return type
Element

classmethod get_gas(*element_id*, *p*, *r_dt*, *T=300.0*, *cx=True*, *a=None*)

Factory method for defining a neutral gas injection target. A gas target is a target with constant density in charge state 0.

Parameters

- **element_id** (*Union[str, int]*) – The full name, abbreviated symbol, or proton number of the element of interest.
- **p** (*float*) – <mbar> Gas pressure.
- **r_dt** (*float*) – <m> Drift tube radius, required to compute linear density from volumetric density.
- **T** (*float*) – <K> Gas temperature, by default 300 K (approx. room temperature)
- **cx** (*bool*) – Boolean flag determining whether the neutral particles of this element contribute to charge exchange with ions.
- **a** (*Optional[float]*) – If provided sets the mass number of the Element object otherwise a reasonable value is chosen automatically.

Returns
Element instance with automatically populated n and kT fields.

Raises
ValueError – If the density resulting from the pressure and temperature is smaller than the internal minimal value.

Return type
Element

classmethod get_ions(*element_id*, *nl*, *kT=10.0*, *q=1*, *cx=True*, *a=None*)

Factory method for defining a pulsed ion injection target. An ion target has a given density in the charge state of choice q.

Parameters

- **element_id** (*Union[str, int]*) – The full name, abbreviated symbol, or proton number of the element of interest.
- **nl** (*float*) – <1/m> Linear density of the initial charge state (ions per unit length).
- **kT** (*float*) – <eV> Temperature / kinetic energy of the injected ions.
- **q** (*int*) – Initial charge state.
- **cx** (*bool*) – Boolean flag determining whether the neutral particles of this element contribute to charge exchange with ions.
- **a** (*Optional[float]*) – If provided sets the mass number of the Element object otherwise a reasonable value is chosen automatically.

Returns
Element instance with automatically populated n and kT fields.

Raises
ValueError – If the requested density is smaller than the internal minimal value.

Return type
Element

index(*value*, *start=0*, *stop=9223372036854775807*, */*)

Return first index of value.

Raises ValueError if the value is not present.

latex_isotope()

Returns the isotope as a LaTeX formatted string.

Returns
str – LaTeX formatted string describing the isotope.

Return type
str

property a

Mass number / approx. mass in proton masses

property cx

Boolean flag determining whether neutral particles of this target are considered as charge exchange partners.

property dr_cs

Numpy array of charge states for DR cross sections.

property dr_e_res

Numpy array of resonance energies for DR cross sections.

property dr_strength

Numpy array of transition strengths for DR cross sections.

property e_bind

Numpy array of binding energies associated with electron subshells. The index of each row corresponds to the charge state. The columns are the subshells sorted as in (‘1s’, ‘2s’, ‘2p-’, ‘2p+’, ‘3s’, ‘3p-’, ‘3p+’, ‘3d-’, ‘3d+’, ‘4s’, ‘4p-’, ‘4p+’, ‘4d-’, ‘4d+’, ‘4f-’, ‘4f+’, ‘5s’, ‘5p-’, ‘5p+’, ‘5d-’, ‘5d+’, ‘5f-’, ‘5f+’, ‘6s’, ‘6p-’, ‘6p+’, ‘6d-’, ‘6d+’, ‘7s’, ‘7p-‘).

property e_cfg

Numpy array of electron configuration in different charge states. The index of each row corresponds to the charge state. The columns are the subshells sorted as in (‘1s’, ‘2s’, ‘2p-’, ‘2p+’, ‘3s’, ‘3p-’, ‘3p+’, ‘3d-’, ‘3d+’, ‘4s’, ‘4p-’, ‘4p+’, ‘4d-’, ‘4d+’, ‘4f-’, ‘4f+’, ‘5s’, ‘5p-’, ‘5p+’, ‘5d-’, ‘5d+’, ‘5f-’, ‘5f+’, ‘6s’, ‘6p-’, ‘6p+’, ‘6d-’, ‘6d+’, ‘7s’, ‘7p-‘).

property ei_lotz_a

Numpy array of precomputed Lotz factor ‘a’ for each entry of ‘e_cfg’.

property ei_lotz_b

Numpy array of precomputed Lotz factor ‘b’ for each entry of ‘e_cfg’.

property ei_lotz_c

Numpy array of precomputed Lotz factor ‘c’ for each entry of ‘e_cfg’.

property ip

Ionisation potential

property kT

<eV> Array holding the initial temperature of each charge state.

property n

<1/m> Array holding the initial linear density of each charge state.

property name

Element name

property rr_n_0_eff

Numpy array of effective valence shell numbers for RR cross sections.

property rr_z_eff

Numpy array of effective nuclear charges for RR cross sections.

property symbol

Element symbol e.g. H, He, Li

property z

Atomic number

ebisim.elements.**element_identify**(*element_id*)

Returns the proton number, name, and element symbol relating to the supplied element_id.

Parameters
element_id (*Union[str, int]*) – The proton number, name or symbol of a chemical element.

Returns
(proton number, name, symbol)

Raises
ValueError – If the element_id could not be identified or found in the database.

Return type
Tuple[int, str, str]

ebisim.elements.**element_name**(*element*)

Returns the name of the given element.

Parameters
element (*Union[str, int]*) – The abbreviated symbol or the proton number of the element.

Returns
Element name

Return type
str

ebisim.elements.**element_symbol**(*element*)

Returns the abbreviated symbol of the given element.

Parameters
element (*Union[str, int]*) – The full name or the proton number of the element.

Returns
Element symbol

Return type
str

ebisim.elements.**element_z**(*element*)

Returns the proton number of the given element.

Parameters
element (*str*) – The full name or the abbreviated symbol of the element.

Returns
Proton number

Return type
int

ebisim.elements.**get_element**(*element_id*, *a=None*)

[LEGACY] Factory function to create instances of the Element class.

Parameters

- **element_id** (*str or int*) – The full name, abbreviated symbol, or proton number of the element of interest.
- **a** (*int or None, optional*) – If provided sets the (isotopic) mass number of the Element object otherwise a reasonable value is chosen automatically, by default None.

Returns
ebisim.elements.Element – An instance of Element with the physical data corresponding to the supplied element_id, and optionally mass number.

Raises
ValueError – If the Element could not be identified or a meaningless mass number is provided.

Return type
Element

2.2.3 ebisim.physconst module

Central module for holding physical constants used in the simulation code.

ebisim.physconst.**ALPHA = 0.0072973525693**

Fine structure constant

ebisim.physconst.**COMPT_E_RED = 3.8615926796089057e-13**

<m> Reduced electron compton wavelength

ebisim.physconst.**C_L = 299792458.0**

<m/s> Speed of light in vacuum

ebisim.physconst.**EPS_0 = 8.8541878128e-12**

<F/m> Vacuum permittivity

ebisim.physconst.**HBAR = 1.0545718176461565e-34**

<J*s> Reduced Planck constant

ebisim.physconst.**K_B = 1.380649e-23**

<J/K> Boltzmann constant

ebisim.physconst.**MINIMAL_KBT = 0.001**

<eV> Minimal temperature equivalent

ebisim.physconst.**MINIMAL_N_1D = 1e-06**
<1/m> Minimal particle number density

ebisim.physconst.**MINIMAL_N_3D = 1.0**
<1/m^3> Minimal particle number density

ebisim.physconst.**M_E = 9.1093837015e-31**
<kg> Electron mass

ebisim.physconst.**M_E_EV = 510998.9499961642**
<eV> Electron mass equivalent

ebisim.physconst.**M_P = 1.67262192369e-27**
<kg> Proton mass

ebisim.physconst.**PI = 3.141592653589793**
Pi

ebisim.physconst.**Q_E = 1.602176634e-19**
<C> Elementary charge

ebisim.physconst.**RY_EV = 13.605693122994232**
<eV> Rydberg energy

2.2.4 ebisim.plasma module

This module contains functions for computing collission rates and related plasma parameters.

@numba.vectorizeebisim.plasma.**clog_ei**(*Ni*, *Ne*, *kbTi*, *kbTe*, *Ai*, *qi*)
The coulomb logarithm for ion electron collisions [CLOGEI].

Parameters

- **Ni** (*float or numpy.ndarray*) – <1/m^3> Ion density.
- **Ne** (*float or numpy.ndarray*) – <1/m^3> Electron density.
- **kbTi** (*float or numpy.ndarray*) – <eV> Ion temperature.
- **kbTe** (*float or numpy.ndarray*) – <eV> Electron temperature.
- **Ai** (*float or numpy.ndarray*) – Ion mass number.
- **qi** (*int or numpy.ndarray*) – Ion charge state.

Returns
float or numpy.ndarray – Ion electron coulomb logarithm.

References

Notes

$$\lambda_{ei} = 23 - \ln\left(N_e^{1/2}(T_e)^{-3/2}q_i\right) \quad \text{if } T_i m_e/m_i < T_e < 10q_i^2 \text{ eV}$$

$$\lambda_{ei} = 24 - \ln\left(N_e^{1/2}(T_e)^{-1}\right) \quad \text{if } T_i m_e/m_i < 10q_i^2 \text{ eV} < T_e$$

$$\lambda_{ei} = 16 - \ln\left(N_i^{1/2}(T_i)^{-3/2}q_i^2\mu_i\right) \quad \text{if } T_e < T_i m_e/m_i$$

$$\left[\lambda_{ei} = 24 - \ln\left(N_e^{1/2}(T_e)^{-1/2}\right) \quad \text{else (fallback)}\right]$$

In these formulas N and T are given in cm^-3 and eV respectively. As documented, the function itself expects the density to be given in 1/m^3.

@numba.vectorizeebisim.plasma.**clog_ii**(*Ni*, *Nj*, *kbTi*, *kbTj*, *Ai*, *Aj*, *qi*, *qj*)

The coulomb logarithm for ion ion collisions [CLOGII].

Parameters

- **Ni** (*float or numpy.ndarray*) – <1/m^3> Ion species "i" density.
- **Nj** (*float or numpy.ndarray*) – <1/m^3> Ion species "j" density.
- **kbTi** (*float or numpy.ndarray*) – <eV> Ion species "i" temperature.
- **kbTj** (*float or numpy.ndarray*) – <eV> Ion species "j" temperature.
- **Ai** (*float or numpy.ndarray*) – Ion species "i" mass number.
- **Aj** (*float or numpy.ndarray*) – Ion species "j" mass number.
- **qi** (*int or numpy.ndarray*) – Ion species "i" charge state.
- **qj** (*int or numpy.ndarray*) – Ion species "j" charge state.

Returns

float or numpy.ndarray – Ion ion coulomb logarithm.

References

Notes

$$\lambda_{ij} = \lambda_{ji} = 23 - \ln\left(\frac{q_i q_j(\mu_i + \mu_j)}{\mu_i T_j + \mu_j T_i}\left(\frac{n_i q_i^2}{T_i} + \frac{n_j q_j^2}{T_j}\right)^{1/2}\right)$$

In these formulas N and T are given in cm^-3 and eV respectively. As documented, the function itself expects the density to be given in 1/m^3.

@numba.vectorizeebisim.plasma.**collisional_escape_rate**(*nui*, *w*)

Generic escape rate - to be called by axial and radial escape

Parameters

- **nui** (*float or numpy.ndarray*) – <1/s> Vector of total ion ion collision rates for each charge state.
- **w** (*float or numpy.ndarray*) – <1/s> Vector of trap (loss) frequencies.

Returns

float or numpy.ndarray – <1/s> Vector of ion loss rates for each charge state.

Notes

$$\frac{3}{\sqrt{2}}\nu_i\frac{e^{-\omega_i}}{\omega_i}$$

@numba.vectorizeebisim.plasma.**collisional_thermalisation**(*kbTi*, *kbTj*, *Ai*, *Aj*, *nuij*)

Computes the collisional energy transfer rates for species "i" with respect to species "j".

Parameters

- **kbTi** (*float or numpy.ndarray*) – <eV> Vector of ion species "i" temperatures.
- **kbTj** (*float or numpy.ndarray*) – <eV> Vector of ion species "j" temperatures.
- **Ai** (*float or numpy.ndarray*) – Ion species "i" mass number.
- **Aj** (*float or numpy.ndarray*) – Ion species "j" mass number.
- **nuij** (*float or numpy.ndarray*) – <1/s> Collision rate matrix for the ions (cf. ion_coll_mat).

Returns

float or numpy.ndarray – <eV/s> Vector of temperature change rate for each charge state.

Notes

$$\left(\frac{dk_BT_i}{dt}\right)_j = 2\nu_{ij}\frac{m_i}{m_j}\frac{k_BT_j - k_BT_i}{(1 + m_ik_BT_j/m_jk_BT_i)^{3/2}}$$

@numba.vectorizeebisim.plasma.**coulomb_xs**(*Ni*, *Ne*, *kbTi*, *Ee*, *Ai*, *qi*)

Computes the Coulomb cross section for elastic electron ion collisions

Parameters

- **Ni** (*float or numpy.ndarray*) – <1/m^3> Ion density.
- **Ne** (*float or numpy.ndarray*) – <1/m^3> Electron density.
- **kbTi** (*float or numpy.ndarray*) – <eV> Ion temperature.
- **Ee** (*float or numpy.ndarray*) – <eV> Electron kinetic energy.
- **Ai** (*float or numpy.ndarray*) – Ion mass number.
- **qi** (*int or numpy.ndarray*) – Ion charge state.

Returns

float or numpy.ndarray – <m^2> Coulomb cross section.

Notes

$$\sigma_i = 4\pi\left(\frac{q_iq_e^2}{4\pi\epsilon_0m_e}\right)^2\frac{\ln\Lambda_{ei}}{v_e^4}$$

@numba.jitebisim.plasma.**electron_velocity**(*e_kin*)

Computes the electron velocity corresponding to a kinetic energy.

Parameters
e_kin (*float or numpy.ndarray*) – <eV> Kinetic energy of the electron.

Returns
float or numpy.ndarray – <m/s> Speed of the electron.

Notes

$$v_e = c\sqrt{1 - \left(\frac{m_e c^2}{m_e c^2 + E_e}\right)^2}$$

@numba.vectorizebisim.plasma.**ion_coll_rate**(*Ni*, *Nj*, *kbTi*, *kbTj*, *Ai*, *Aj*, *qi*, *qj*)

Collision rates for ions species "i" and target species "j"

Parameters

- **Ni** (*float or numpy.ndarray*) – <1/m^3> Ion species "i" density.
- **Nj** (*float or numpy.ndarray*) – <1/m^3> Ion species "j" density.
- **kbTi** (*float or numpy.ndarray*) – <eV> Ion species "i" temperature.
- **kbTj** (*float or numpy.ndarray*) – <eV> Ion species "j" temperature.
- **Ai** (*float or numpy.ndarray*) – Ion species "i" mass number.
- **Aj** (*float or numpy.ndarray*) – Ion species "j" mass number.
- **qi** (*int or numpy.ndarray*) – Ion species "i" charge state.
- **qj** (*int or numpy.ndarray*) – Ion species "j" charge state.

Returns
float or numpy.ndarray – <1/s> Ion ion collision rate.

See also:

ebisim.plasma.ion_coll_rate_mat
Similar method for all charge states.

Notes

$$\nu_{ij} = \frac{1}{(4\pi\epsilon_0)^2}\frac{4\sqrt{2\pi}}{3} N_j \left(\frac{q_i q_j q_e^2}{m_i}\right)^2 \left(\frac{m_i}{k_B T_i}\right)^{3/2} \ln\Lambda_{ij}$$

@numba.vectorizebisim.plasma.**spitzer_heating**(*Ni*, *Ne*, *kbTi*, *Ee*, *Ai*, *qi*)

Computes the heating rates due to elastic electron ion collisions ('Spitzer Heating')

Parameters

- **Ni** (*float or numpy.ndarray*) – <1/m^3> Vector of ion densities.
- **Ne** (*float or numpy.ndarray*) – <1/m^3> Electron density.
- **kbTi** (*float or numpy.ndarray*) – <eV> Vector of ion temperatures.
- **Ee** (*float or numpy.ndarray*) – <eV> Electron kinetic energy.

- **Ai** (*float or numpy.ndarray*) – Ion mass number.

Returns

float or numpy.ndarray – <eV/s> Vector of electron heating rate (temperature increase) for each charge state.

Notes

$$\left(\frac{dk_BT_i}{dt}\right)^{\text{Spitzer}} = \frac{2}{3}N_e v_e \sigma_i 2\frac{m_e}{m_i}E_e$$

where sigma_i is the cross section for Coulomb collisions (cf. ebisim.plasma.coulomb_xs).

@numba.vectorizeebisim.plasma.**trapping_strength_axial**(*kbTi*, *qi*, *V*)

Computes the axial trapping strenghts.

Parameters

- **kbTi** (*float or numpy.ndarray*) – <eV> Vector of ion temperatures.
- **qi** (*int or numpy.ndarray*) – Ion species "i" charge state.
- **V** (*float or numpy.ndarray*) – <V> Trap depth.

Returns

float or numpy.ndarray – <1/s> Vector of axial ion trapping strenghts for each charge state.

Notes

$$\omega_i^{ax} = \frac{q_i V_{ax}}{k_B T_i}$$

@numba.vectorizeebisim.plasma.**trapping_strength_radial**(*kbTi*, *qi*, *Ai*, *V*, *B*, *r_dt*)

Radial trapping strenghts.

Parameters

- **kbTi** (*float or numpy.ndarray*) – <eV> Vector of ion temperatures.
- **qi** (*int or numpy.ndarray*) – Ion species "i" charge state.
- **Ai** (*float or numpy.ndarray*) – Ion mass number.
- **V** (*float or numpy.ndarray*) – <V> Trap depth.
- **B** (*float or numpy.ndarray*) – <T> Axial magnetic flux density.
- **r_dt** (*float or numpy.ndarray*) – <m> Drift tube radius.

Returns

float or numpy.ndarray – <1/s> Vector of radial ion trapping strenghts for each charge state.

Notes

$$\omega_i^{rad} = \frac{q_i \left(V_{rad} + Br_{dt} \sqrt{2 k_B T_i / (3 m_i)} \right)}{k_B T_i}$$

2.2.5 ebisim.plotting module

All the plotting logic of ebisim is collected in this module. The functions can be called manually by the user, but are primarily desinged to be called internally by ebisim, thefore the API may lack convenience in some places.

ebisim.plotting.**decorate_axes**(*ax*, *grid=True*, *legend=False*, *label_lines=True*, *tight_layout=True*, ***kwargs*)

This function exists to have a common routine for setting certain figure properties, it is called by all other plotting routines and takes over the majority of the visual polishing.

Parameters

- **ax** (*matplotlib.Axes*) – The axes to be modifed.
- **grid** (*bool, optional*) – Whether or not to lay a grid over the plot, by default True.
- **legend** (*bool, optional*) – Whether or not to put a legend next to the plot, by default False.
- **label_lines** (*bool, optional*) – Whether or not to put labels along the lines in the plot, by default True.
- **tight_layout** (*bool, optional*) – Whether or not to apply matplotlibs tight layout on the parentfigure of ax, by default True.
- ****kwargs** – Are directly applied as axes properties, e.g. xlabel, xscale, title, etc.

ebisim.plotting.**plot_combined_xs**(*element*, *fwhm*, ***kwargs*)

Creates a figure showing the electron ionisation, radiative recombination and, dielectronic recombination cross sections of the provided element.

Parameters

- **element** (ebisim.elements.Element *or str or int*) – An instance of the Element class, or an identifier for the element, i.e. either its name, symbol or proton number.
- **fwhm** (*float*) – <eV> Energy spread to apply for the resonance smearing, expressed in terms of full width at half maximum.
- ****kwargs** – Remaining keyword arguments are handed down to ebisim.plotting.decorate_axes, cf. documentation thereof. If no arguments are provided, reasonable default values are injected.

Returns

matplotlib.Figure – Figure handle of the generated plot.

ebisim.plotting.**plot_drxs**(*element*, *fwhm*, ***kwargs*)

Creates a figure showing the dielectronic recombination cross sections of the provided element.

Parameters

- **element** (ebisim.elements.Element *or str or int*) – An instance of the Element class, or an identifier for the element, i.e. either its name, symbol or proton number.
- **fwhm** (*float*) – <eV> Energy spread to apply for the resonance smearing, expressed in terms of full width at half maximum.

- ****kwargs** – 'fig' is intercepted and can be used to plot on top of an existing figure. 'ls' is intercepted and can be used to set the linestyle for plotting. Remaining keyword arguments are handed down to ebisim.plotting.decorate_axes, cf. documentation thereof. If no arguments are provided, reasonable default values are injected.

Returns
matplotlib.Figure – Figure handle of the generated plot.

ebisim.plotting.**plot_eixs**(*element*, ***kwargs*)

Creates a figure showing the electron ionisation cross sections of the provided element.

Parameters

- **element** (*ebisim.elements.Element or str or int*) – An instance of the Element class, or an identifier for the element, i.e. either its name, symbol or proton number.
- ****kwargs** – 'fig' is intercepted and can be used to plot on top of an existing figure. 'ls' is intercepted and can be used to set the linestyle for plotting. Remaining keyword arguments are handed down to ebisim.plotting.decorate_axes, cf. documentation thereof. If no arguments are provided, reasonable default values are injected.

Returns
matplotlib.Figure – Figure handle of the generated plot.

ebisim.plotting.**plot_energy_scan**(*energies*, *abundance*, *cs=None*, ***kwargs*)

Produces a plot of the charge state abundance for different energies at a given time.

Parameters

- **energies** (*numpy.array*) – <eV> The evaluated energies.
- **abundance** (*numpy.array*) – The abundance of each charge state (rows) for each energy (columns).
- **cs** (*list of int or None, optional*) – If None, all charge states are plotted. By supplying a list of int it is possible to filter the charge states that should be plotted. By default None.
- ****kwargs** – Keyword arguments are handed down to ebisim.plotting.decorate_axes, cf. documentation thereof. If no arguments are provided, reasonable default values are injected.

Returns
matplotlib.Figure – Figure handle of the generated plot.

ebisim.plotting.**plot_energy_time_scan**(*energies*, *times*, *abundance*, ***kwargs*)

Provides information about the abundance of a single charge states at all simulated times and energies.

Parameters

- **energies** (*numpy.array*) – <eV> The evaluated energies.
- **times** (*numpy.array*) – <s> The evaluated timesteps.
- **abundance** (*numpy.array*) – Abundance of charge state 'cs' at given times (rows) and energies (columns).
- ****kwargs** – Keyword arguments are handed down to ebisim.plotting.decorate_axes, cf. documentation thereof. If no arguments are provided, reasonable default values are injected.

Returns
matplotlib.Figure – Figure handle of the generated plot.

ebisim.plotting.**plot_generic_evolution**(*t*, *y*, *plot_total=False*, *ax=None*, *cs=None*, ***kwargs*)

Plots the evolution of a quantity with time

Parameters

- **t** (*numpy.array*) – <s> Values for the time steps.
- **y** (*numpy.array*) – Values of the evoloving quantity to plot as a function of time. Has to be a 2D numpy array where the rows correspond to the different charge states and the columns correspond to the individual timesteps.
- **plot_total** (*bool, optional*) – Indicate whether a black dashed line indicating the total accross all charge states should also be plotted, by default False.
- **ax** (*matplotlib.Axes, optional*) – Provide if you want to add this plot to existing Axes, by default None.
- **cs** (*list[int], optional*) – Specify which charge states should be plotted, plot all if omitted.
- ****kwargs** – Keyword arguments are handed down to ebisim.plotting.decorate_axes, cf. documentation thereof. If no arguments are provided, reasonable default values are injected.

Returns

matplotlib.Figure – Figure handle of the generated plot.

ebisim.plotting.**plot_radial_distribution**(*r*, *dens*, *phi=None*, *r_e=None*, *ax=None*, *ax2=None*, ***kwargs*)

Plots the radial ion distribution, can also plot radial potential and electron beam radius.

Parameters

- **r** (*numpy.ndarray*) – [description]
- **dens** (*numpy.ndarray*) – Array of densities, shaped like 'y' in plot_generic_evolution.
- **phi** (*numpy.ndarray, optional*) – The radial potential, if supplied will be plotted on second y-axis, by default None.
- **r_e** (*numpy.ndarray, optional*) – Electron beam radius, if provided will be marked as vertical likne, by default None.
- **ax** (*numpy.ndarray, optional*) – Axes on which to plot the densities, by default None.
- **ax2** (*numpy.ndarray, optional*) – Axes on which to plot the radial potential, by default None.

Returns

- **ax** (*matplotlib.Axes*) – As above.
- **ax2** (*matplotlib.Axes*) – As above.

ebisim.plotting.**plot_rrxs**(*element*, ***kwargs*)

Creates a figure showing the radiative recombination cross sections of the provided element.

Parameters

- **element** (ebisim.elements.Element *or str or int*) – An instance of the Element class, or an identifier for the element, i.e. either its name, symbol or proton number.
- ****kwargs** – 'fig' is intercepted and can be used to plot on top of an existing figure. 'ls' is intercepted and can be used to set the linestyle for plotting. Remaining keyword arguments are handed down to ebisim.plotting.decorate_axes, cf. documentation thereof. If no arguments are provided, reasonable default values are injected.

Returns
matplotlib.Figure – Figure handle of the generated plot.

ebisim.plotting.**COLORMAP = <matplotlib.colors.ListedColormap object>**

The default colormap used to grade line plots, assigning another colormap to this object will result in an alternative color gradient for line plots

2.2.6 ebisim.utils module

This module contains convenience and management functions not directly related to the simulation code, e.g. loading resources. These functions are meant for internal use only, they have no real use outside this scope.

ebisim.utils.**load_dr_data**()

Loads the avaliable DR transition data from the resource directory

Returns
dict of dicts – A dictiomary with the proton number as dict-keys. Each value is another dictionary with the items "dr_e_res" (resonance energy), "dr_strength" (transitions strength), and "dr_cs" (charge state) The values are linear numpy arrays holding corresponding data on the same rows.

Return type
Dict[int, *Dict*[str, *ndarray*]]

ebisim.utils.**patch_namedtuple_docstrings**(*named_tuple*, *docstrings*)

Add docstrings to the fields of a namedtuple/NamedTuple

Parameters

- **named_tuple** (*Any*) – The class definition inheriting from namedtupe or NamedTuple
- **docstrings** (*Dict[str, str]*) – Dictionary with field names as keys and docstrings as values

Return type
None

ebisim.utils.**validate_namedtuple_field_types**(*instance*)

Checks if the values of a typing.NamedTuple instance agree with the types that were annotated in the class definition.

Values are permitted to be ints if type annotation is float.

Parameters
instance (*Any*) –

Return type
bool

2.2.7 ebisim.xs module

This module contains functions to compute the cross sections for various ionisation and recombination processes.

@numba.jitebisim.xs.**cxxs**(*q*, *ip*)

Single charge exchange cross section according to the Mueller Salzborn formula

Parameters

- **q** (*int*) – Charge state of the colliding ion
- **ip** (*float*) – <eV> Ionisation potential of the collision partner (neutral gas)

Returns
float – <m^2> Charge exchange cross section

@numba.jit ebisim.xs.**drxs_energyscan**(*element*, *fwhm*, *e_kin=None*, *n=1000*)

Creates an array of DR cross sections for varying electron energies.

Parameters

- **element** (ebisim.Element) – An ebisim.Element object that holds the required physical information for cross section calculations.
- **fwhm** (*float*) – <eV> Energy spread to apply for the resonance smearing, expressed in terms of full width at half maximum.
- **e_kin** (*None or numpy.ndarray, optional*) – <eV> If e_kin is None, the range of sampling energies is chosen based on the binding enrgies of the element and energies are sampled on a logscale. If e_kin is an array with 2 elements, they are interpreted as the minimum and maximum sampling energy. If e_kin is an array with more than two values, the energies are taken as the sampling energies directly, by default None.
- **n** (*int, optional*) – The number of energy sampling points, if the sampling locations are not supplied by the user, by default 1000.

Returns

- **e_samp** (*numpy.ndarray*) – <eV> Array holding the sampling energies
- **xs_scan** (*numpy.ndarray*) – <m^2> Array holding the cross sections, where the row index corresponds to the charge state and the columns correspond to the different sampling energies

See also:

ebisim.xs.eixs_energyscan, *ebisim.xs.rrxs_energyscan*

@numba.jit ebisim.xs.**drxs_mat**(*element*, *e_kin*, *fwhm*)

Dielectronic recombination cross section. The cross sections are estimated by weighing the strength of each transition with the profile of a normal Gaussian distribution. This simulates the effective spreading of the resonance peaks due to the energy spread of the electron beam

Parameters

- **element** (ebisim.Element) – An ebisim.Element object that holds the required physical information for cross section calculations.
- **e_kin** (*float*) – <eV> Kinetic energy of the impacting electron.
- **fwhm** (*float*) – <eV> Energy spread to apply for the resonance smearing, expressed in terms of full width at half maximum.

Returns
numpy.array – <m^2> The cross sections for each individual charge state, arranged in a matrix suitable for implementation of a rate equation like dN/dt = j * xs_matrix dot N. out[q, q] = - cross section of q+ ion out[q, q+1] = + cross section of (q+1)+ ion

See also:

ebisim.xs.drxs_vec
Similar method with different output format.

@numba.jitebisim.xs.**drxs_vec**(*element*, *e_kin*, *fwhm*)

Dielectronic recombination cross section. The cross sections are estimated by weighing the strength of each transition with the profile of a normal Gaussian distribution. This simulates the effective spreading of the resonance peaks due to the energy spread of the electron beam

Parameters

- **element** (ebisim.Element) – An ebisim.Element object that holds the required physical information for cross section calculations.
- **e_kin** (*float*) – <eV> Kinetic energy of the impacting electron.
- **fwhm** (*float*) – <eV> Energy spread to apply for the resonance smearing, expressed in terms of full width at half maximum.

Returns

numpy.ndarray – <m^2> The cross sections for each individual charge state, where the array-index corresponds to the charge state, i.e. out[q] ~ cross section of q+ ion.

See also:

ebisim.xs.drxs_mat
Similar method with different output format.

@numba.jitebisim.xs.**eixs_energyscan**(*element*, *e_kin=None*, *n=1000*)

Creates an array of EI cross sections for varying electron energies.

Parameters

- **element** (ebisim.Element) – An ebisim.Element object that holds the required physical information for cross section calculations.
- **e_kin** (*None or numpy.ndarray, optional*) – <eV> If e_kin is None, the range of sampling energies is chosen based on the binding enrgies of the element and energies are sampled on a logscale. If e_kin is an array with 2 elements, they are interpreted as the minimum and maximum sampling energy. If e_kin is an array with more than two values, the energies are taken as the sampling energies directly, by default None.
- **n** (*int, optional*) – The number of energy sampling points, if the sampling locations are not supplied by the user, by default 1000.

Returns

- **e_samp** (*numpy.ndarray*) – <eV> Array holding the sampling energies
- **xs_scan** (*numpy.ndarray*) – <m^2> Array holding the cross sections, where the row index corresponds to the charge state and the columns correspond to the different sampling energies

See also:

ebisim.xs.rrxs_energyscan, *ebisim.xs.drxs_energyscan*

@numba.jitebisim.xs.**eixs_mat**(*element*, *e_kin*)

Electron ionisation cross section.

Parameters

- **element** (ebisim.Element) – An ebisim.Element object that holds the required physical information for cross section calculations.
- **e_kin** (*float*) – <eV> Kinetic energy of the impacting electron.

Returns

numpy.array – <m^2> The cross sections for each individual charge state, arranged in a matrix suitable for implementation of a rate equation like dN/dt = j * xs_matrix dot N. out[q, q] = - cross section of q+ ion out[q+1, q] = + cross section of (q+1)+ ion

See also:

ebisim.xs.eixs_vec
Similar method with different output format.

@numba.jitebisim.xs.**eixs_vec**(*element*, *e_kin*)

Electron ionisation cross section according to a simplified version of the models given in [Lotz1967].

Parameters

- **element** (ebisim.Element) – An ebisim.Element object that holds the required physical information for cross section calculations.
- **e_kin** (*float*) – <eV> Kinetic energy of the impacting electron.

Returns

numpy.ndarray – <m^2> The cross sections for each individual charge state, where the array-index corresponds to the charge state, i.e. out[q] ~ cross section of q+ ion.

References

See also:

ebisim.xs.eixs_mat
Similar method with different output format.

ebisim.xs.**lookup_lotz_factors**(*e_cfg*, *shellorder*)

Analyses the shell structure of each charge state and looks up the correct factors for the Lotz formula.

This function is primarily meant for internal use inside the ebisim.get_element() function and the results are consumed during the Electron Ionisation (EI) cross section computations.

Parameters

- **e_cfg** (*numpy.ndarray*) – Matrix holding the number of electrons in each shell. The row index corresponds to the charge state, the columns to different subshells
- **shellorder** (*numpy.ndarray*) – Tuple containing the names of all shells in the same order as they appear in 'e_cfg'

Returns

- **ei_lotz_a** (*numpy.ndarray*) – Array holding 'Lotz' factor 'a' for each occupied shell in 'e_cfg' up to a certain charge state.
- **ei_lotz_b** (*numpy.ndarray*) – Array holding 'Lotz' factor 'b' for each occupied shell in 'e_cfg' up to a certain charge state.
- **ei_lotz_b** (*numpy.ndarray*) – Array holding 'Lotz' factor 'c' for each occupied shell in 'e_cfg' up to a certain charge state.

See also:

ebisim.xs.eixs_vec, *ebisim.xs.eixs_mat*

@numba.jitebisim.xs.**precompute_rr_quantities**(*e_cfg*, *shell_n*)

Precomputes the effective valence shell and nuclear charge for all charge states, as required for the computation of radiative recombinations cross sections. According to the procedure described in [Kim1983a].

This function is primarily meant for internal use inside the ebisim.get_element() function.

Parameters

- **e_cfg** (*numpy.ndarray*) – Matrix holding the number of electrons in each shell. The row index corresponds to the charge state, the columns to different subshells
- **shell_n** (*numpy.ndarray*) – Array holding the main quantum number n corresponding to each shell listed in e_cfg

Returns

- **rr_z_eff** (*numpy.ndarray*) – Array holding the effective nuclear charge for each charge state, where the array-index corresponds to the charge state.
- **rr_n_0_eff** (*numpy.ndarray*) – Array holding the effective valence shell number for each charge state, where the array-index corresponds to the charge state.

References

See also:

ebisim.xs.rrxs_vec, *ebisim.xs.rrxs_mat*

@numba.jitebisim.xs.**rrxs_energyscan**(*element*, *e_kin=None*, *n=1000*)

Creates an array of RR cross sections for varying electron energies.

Parameters

- **element** (ebisim.Element) – An ebisim.Element object that holds the required physical information for cross section calculations.
- **e_kin** (*None or numpy.ndarray, optional*) – <eV> If e_kin is None, the range of sampling energies is chosen based on the binding enrgies of the element and energies are sampled on a logscale. If e_kin is an array with 2 elements, they are interpreted as the minimum and maximum sampling energy. If e_kin is an array with more than two values, the energies are taken as the sampling energies directly, by default None.
- **n** (*int, optional*) – The number of energy sampling points, if the sampling locations are not supplied by the user, by default 1000.

Returns

- **e_samp** (*numpy.ndarray*) – <eV> Array holding the sampling energies
- **xs_scan** (*numpy.ndarray*) – <m^2> Array holding the cross sections, where the row index corresponds to the charge state and the columns correspond to the different sampling energies

See also:

ebisim.xs.eixs_energyscan, *ebisim.xs.drxs_energyscan*

@numba.jitebisim.xs.**rrxs_mat**(*element*, *e_kin*)

Radiative recombination cross section.

Parameters

- **element** (ebisim.Element) – An ebisim.Element object that holds the required physical information for cross section calculations.

- **e_kin** (*float*) – <eV> Kinetic energy of the impacting electron.

Returns

numpy.array – <m^2> The cross sections for each individual charge state, arranged in a matrix suitable for implementation of a rate equation like dN/dt = j * xs_matrix dot N. out[q, q] = - cross section of q+ ion out[q, q+1] = + cross section of (q+1)+ ion

See also:

ebisim.xs.rrxs_vec
Similar method with different output format.

@numba.jitebisim.xs.**rrxs_vec**(*element*, *e_kin*)

Radiative recombination cross section according to [Kim1983].

Parameters

- **element** (ebisim.Element) – An ebisim.Element object that holds the required physical information for cross section calculations.
- **e_kin** (*float*) – <eV> Kinetic energy of the impacting electron.

Returns

numpy.ndarray – <m^2> The cross sections for each individual charge state, where the array-index corresponds to the charge state, i.e. out[q] ~ cross section of q+ ion.

References

See also:

ebisim.xs.rrxs_mat
Similar method with different output format.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[Lotz1967] "An empirical formula for the electron-impact ionization cross-section", W. Lotz, Zeitschrift Für Physik, 206(2), 205–211 (1967), https://doi.org/10.1007/BF01325928

[Kim1983] "Direct radiative recombination of electrons with atomic ions: Cross sections and rate coefficients", Young Soon Kim and R. H. Pratt, Phys. Rev. A 27, 2913 (1983), https://journals.aps.org/pra/abstract/10.1103/PhysRevA.27.2913

[PICNPSb] "Nonlinear Poisson Solver" https://www.particleincell.com/2012/nonlinear-poisson-solver/

[PICNPS] "Nonlinear Poisson Solver" https://www.particleincell.com/2012/nonlinear-poisson-solver/

[PICNPSa] "Nonlinear Poisson Solver" https://www.particleincell.com/2012/nonlinear-poisson-solver/

[Sundqvist1970] "A simple finite-difference grid with non-constant intervals", Sundqvist, H., & Veronis, G., Tellus, 22(1), 26–31 (1970), https://doi.org/10.3402/tellusa.v22i1.10155

[TDMA] "Tridiagonal matrix algorithm" https://en.wikipedia.org/wiki/Tridiagonal_matrix_algorithm

[CLOGEI] "NRL Plasma Formulary", J. D. Huba, Naval Research Laboratory (2019), https://www.nrl.navy.mil/ppd/sites/www.nrl.navy.mil.ppd/files/pdfs/NRL_Formulary_2019.pdf

[CLOGII] "NRL Plasma Formulary", J. D. Huba, Naval Research Laboratory (2019), https://www.nrl.navy.mil/ppd/sites/www.nrl.navy.mil.ppd/files/pdfs/NRL_Formulary_2019.pdf

[Lotz1967] "An empirical formula for the electron-impact ionization cross-section", W. Lotz, Zeitschrift Für Physik, 206(2), 205–211 (1967), https://doi.org/10.1007/BF01325928

[Kim1983a] "Direct radiative recombination of electrons with atomic ions: Cross sections and rate coefficients", Young Soon Kim and R. H. Pratt, Phys. Rev. A 27, 2913 (1983), https://journals.aps.org/pra/abstract/10.1103/PhysRevA.27.2913

[Kim1983] "Direct radiative recombination of electrons with atomic ions: Cross sections and rate coefficients", Young Soon Kim and R. H. Pratt, Phys. Rev. A 27, 2913 (1983), https://journals.aps.org/pra/abstract/10.1103/PhysRevA.27.2913

PYTHON MODULE INDEX

e

INDEX

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z