[2]p()*1/2-^ [1]p1-^ TJ 3
10000pt

0.00.5em

0.0.00.5em

0.0.0.00.5em

0.0.0.0.00.5em

0.0.0.0.0.00.5em
fontsize=

aurora Documentation

Release 1.0.0

Francesco Sciortino

Oct 20, 2020

Contents

Github repo: https://github.com/fsciortino/aurora

OverviewOverview *aurora* is a 1.5D modern forward model for radial particle transport in magnetically confined plasmas. It inherets many of its methods from the historical STRAHL code and has been thoroughly benchmarked with it. The core of *aurora* is written in Python3 and Fortran90, with a Julia interface also under development.
aurora includes Python functionality to create inputs and read/plot outputs. OMFIT users may access this functionality via the OMFIT STRAHL module. *aurora* was designed to be as efficient as possible in iterative workflows, where different diffusion and convection coefficients are run through the code in order to match some experimental observations. For this reason, *aurora* keeps all data in memory and avoids any I/O during operation.

A number of standard tests and examples are provided using a real set of Alcator C-Mod kinetic profiles and geometry. In order to interface with EFIT gEQDSK files, *aurora* makes use of the `omfit_eqdsk` package, which offers flexibility to work with data from many devices worldwide. Users may easily substitute this dependence with different magnetic reconstruction packages and/or postprocessing interfaces, if required.

aurora provides convenient interfaces to load a default namelist via *default_nml()*, modify it as required and then pass the resulting namelist dictionary into the simulation setup. This is in the main class of *aurora*, *aurora_sim*, which allows creation of radial and temporal grids, interpolation of atomic rates, preparation of parallel loss rates at the edge, etc.

The library in *atomic()* provides functions to load and interpolate atomic rates from ADAS ADF-11 files, as well as from ADF-15 photon emissivity coefficients (PEC) files. PEC data can alternatively be computed using the collisional-radiative model of CollRadPy, using methods in *radiation()*.

aurora was born as a fast forward model of impurity transport, but it can do much more. For example, it may be helpful for parameter scans in modeling of future devices. The *radiation_model()* method allows one to use ADAS atomic rates and given kinetic profiles to compute line radiation, bremsstrahlung, continuum and soft-x-ray-filtered radiation. Ionization equilibria can also be computed using the *atomic()* methods, thus enabling simple "constant-fraction" models where the total density of an impurity species is fixed to a certain percentage of the electron density.

InstallationInstallation To obtain the latest version of the code, it is recommended to git-clone the repo https://github.com/fsciortino/aurora and run the makefile from the command line.
The latest stable version of the code can also be obtained via:

codes*=
[commandchars=
{}] pip install aurora
of from Anaconda Cloud:

codes*=
[commandchars=
{}] conda install aurora

Demos/ExamplesDemos/Examples A number of demonstrations and examples are available in the `aurora` "examples" directory. These show how to load a default namelist, modify it with specific kinetic profiles, load a magnetic equilibrium, set up an `aurora` object and run. A number of plotting tools and postprocessing scripts to calculate radiation are also presented.

Questions? Suggestions?Questions? Suggestions? Please contact sciortino-at-psfc.mit.edu for any questions. Suggestions and collaborations are more than welcome!

5.1 aurora package

5.1.1 Submodules

5.1.2 aurora.core module

This module includes the core class to set up simulations with `eqtools`. The *`aurora_sim`* takes as input a namelist dictionary and a g-file dictionary (and possibly other optional argument) and allows creation of grids, interpolation of atomic rates and other steps before running the forward model.

class aurora.core.**aurora_sim**(*namelist*, *geqdsk=None*, *nbi_cxr=None*)

Bases: object

Class to setup and run aurora simulations.

get_aurora_kin_profs(*min_T=1.01*, *min_ne=10000000000.0*)

Get kinetic profiles on radial and time grids.

get_par_loss_rate(*trust_SOL_Ti=False*)

Calculate the parallel loss frequency on the radial and temporal grids [1/s].

trust_SOL_Ti should generally be set to False, unless specific Ti measurements are available in the SOL.

get_time_dept_atomic_rates()

Obtain time-dependent ionization and recombination rates for a simulation run. If kinetic profiles are given as time-independent, atomic rates for each time slice will be set to be the same.

interp_kin_prof(*prof*)

Interpolate the given kinetic profile on the radial and temporal grids [units of s]. This function extrapolates in the SOL based on input options using the same methods as in STRAHL.

run_aurora(*times_DV*, *D_z*, *V_z*, *nz_init=None*, *method='old'*, *evolneut=False*)

Run a simulation using inputs in the given dictionary and D,v profiles as a function of space, time and potentially also ionization state. Users may give an initial state of each ion charge state as an input.

Results can be conveniently visualized with time-slider using

codes*=
[commandchars=
{}] aurora.slider_plot(rhop,time, nz.transpose(1,2,0), xlabel=r'ρ_p', ylabel='time [s]', zlabel=r'n_z [cm^{-3}]', plot_sum=**True**, labels=[f'Ca$^{{{/str(i)/}}}$' **for** i **in** np.arange(nz_w.shape[1]])

Args:

times_DV [1D array] Array of times at which D_z and V_z profiles are given. (Note that it is assumed that D and V profiles are already on the self.rvol_grid radial grid).

D_z, V_z: arrays, shape of (space, time,nZ) or (space,time) Diffusion and convection coefficients, in units of cm^2/s and cm/s, respectively. This may be given as a function of (space,time) or (space,nZ, time), where nZ indicates the number of charge states. If inputs are found to be have only 2 dimensions, it is assumed that all charge states should be set to have the same transport coefficients.

nz_init: array, shape of (space, nZ) Impurity charge states at the initial time of the simulation. If left to None, this is internally set to an array of 0's.

method [str, optional] If method='linder', use the Linder algorithm for increased stability and accuracy.

evolneut [bool, optional] If True, evolve neutral impurities based on their D,V coefficients. Default is False, in which case neutrals are only taken as a source and those that are not ionized immediately after injection are neglected.

Returns:

out [list] List containing each particle reservoir of a simulation, i.e. nz, N_wall, N_div, N_pump, N_ret, N_tsu, N_dsu, N_dsul, rcld_rate, rclw_rate = out

run_julia(*times_DV*, *D_z*, *V_z*, *nz_init=None*)
Run a single simulation using the Julia version.

Args:

times_DV [1D array] Array of times at which D_z and V_z profiles are given. (Note that it is assumed that D and V profiles are already on the self.rvol_grid radial grid).

D_z, V_z: arrays, shape of (space, nZ, time) Diffusion and convection coefficients, in units of cm^2/s and cm/s, respectively. This may be given as a function of (space,time) or (space,nZ, time), where nZ indicates the number of charge states. If inputs are found to be have only 2 dimensions, it is assumed that all charge states should be set to have the same transport coefficients.

nz_init: array, shape of (space, nZ) Impurity charge states at the initial time of the simulation. If left to None, this is internally set to an array of 0's.

Returns:

out [list] List containing each particle reservoir of a simulation, i.e. nz, N_wall, N_div, N_pump, N_ret, N_tsu, N_dsu, N_dsul, rcld_rate, rclw_rate = out

5.1.3 aurora.atomic module

class aurora.atomic.**CartesianGrid**(*grids*, *values*)
Bases: object

Linear multivariate Cartesian grid interpolation in arbitrary dimensions This is a regular grid with equal spacing.

class aurora.atomic.**adas_file**(*filepath*)
Bases: object

Read ADAS file in ADF11 format over the given ne, T.

load()

plot(*fig=None*, *axes=None*)

aurora.atomic.**adas_files_dict**()
Selections for ADAS files for Aurora runs and radiation calculations. This function can be called to fetch a set of default files, which can then be modified (e.g. to use a new file for a specific SXR filter) before running a calculation.

aurora.atomic.**balance**(*logTe_val*, *cs*, *n0_by_ne*, *logTe_*, *S*, *R*, *cx*)
Evaluate balance of effective ionization, recombination and charge exchange at a given temperature.

aurora.atomic.**get_adas_ion_rad**(*ion_name*, *n_ion*, *logne_prof*, *logTe_prof*, *sxr=False*)
Get ADAS estimate for total radiation in [M/m^3] for the given ion with the given (log) density and temperature profiles.

If sxr=True, 'prs' files are used instead of 'prb' ones, thus giving SXR-filtered radiation for the SXR filter indicated by the atomic data dictionary.

aurora.atomic.**get_all_atom_data**(*imp*, *files=None*)
Collect atomic data for a given impurity from all types of ADAS files available.

imp: str Atomic symbol of impurity ion.

files [list or array-like] ADAS file names to be fetched.

aurora.atomic.**get_atomdat_info**()
Function to identify location of ADAS atomic data in a generalized fashion and to obtain the list of file_types of interest.

aurora.atomic.**get_cooling_factors**(*atom_data*, *logTe_*, *fz*, *ion_resolved=False*, *plot=True*, *ax=None*)
Calculate cooling coefficients from fz and prs, pls (if these are available). Also plot the inverse of rate coefficients, which gives an estimate for relaxation times.

aurora.atomic.**get_cs_balance_terms**(*atom_data*, *ne=5e+19*, *Te=None*, *maxTe=10000.0*, *include_cx=True*)
Get S, R and cx on the same logTe grid.

Args:

atom_data [dictionary of atomic ADAS files (only acd, scd are required; ccd is] necessary only if include_cx=True

ne [float or array] Electron density in units of m^-3

Te [float or array] Electron temperature in units of eV. If left to None, the Te grid given in the atomic data is used.

maxTe [float] Maximum temperature of interest; only used if Te is left to None.

include_cx [bool] If True, obtain charge exchange terms as well.

Returns:

logTe [array (n_Te)] log10 Te grid on which atomic rates are given

logS, logR (,logcx): arrays (n_ne,n_Te) atomic rates for effective ionization, radiative+dielectronic recombination (+ charge exchange, if requested). After exponentiation, all terms will be in units of s^-1.

aurora.atomic.**get_file_types**()

Returns main types of ADAS atomic data of interest

aurora.atomic.**get_frac_abundances**(*atom_data*, *ne*, *Te=None*, *n0_by_ne=1e-05*, *include_cx=False*, *plot=True*, *ax=None*, *rho=None*, *rho_lbl=None*, *ls='-'*, *compute_rates=False*)

Calculate fractional abundances from ionization and recombination equilibrium. If include_cx=True, radiative recombination and thermal charge exchange are summed.

Args:

atom_data [dictionary of atomic ADAS files (only acd, scd are required; ccd is] necessary only if include_cx=True

ne [float or array] Electron density in units of m^-3

Te [float or array, optional] Electron temperature in units of eV. If left to None, the Te grid given in the atomic data is used.

n0_by_ne: float or array, optional Ratio of background neutral hydrogen to electron density, used if include_cx=True.

include_cx [bool] If True, charge exchange with background thermal neutrals is included.

plot [bool, optional] Show fractional abundances as a function of ne,Te profiles parameterization.

ax [matplotlib.pyplot Axes instance] Axes on which to plot if plot=True. If False, it creates new axes

rho [list or array, optional] Vector of radial coordinates on which ne,Te (and possibly n0_by_ne) are given. This is only used for plotting, if given.

rho_lbl: str, optional Label to be used for rho. If left to None, defaults to a general "rho".

ls [str, optional] Line style for plots. Continuous lines are used by default.

compute_rates [bool] If True, compute rate coefficients for ionization/recombination equilibrium on top of fractional abundances (which should be the same regardless of the method used).

Returns:

logTe [array] log10 of electron temperatures as a function of which the fractional abundances and rate coefficients are given.

fz [array, (space,nZ)] Fractional abundances across the same grid used by the input ne,Te values.

rate_coeff [array, (space, nZ)] Rate coefficients in units of [s^-1].

aurora.atomic.**gff_mean**(*Z*, *Te*)

Total free-free gaunt factor yielding the total radiated bremsstrahlung power when multiplying with the result for gff=1. Data originally from Karzas & Latter, extracted from STRAHL's atomic_data.f.

aurora.atomic.**impurity_brems**(*nz*, *ne*, *Te*)
Impurity bremsstrahlung in units of mW/nm/sr/m^3.cm^3.

This is only approximate and may not be very useful, since this contribution is already included in the continuum in x2.

This estimate does not have the correct ne-dependence of the Gaunt factor... use with care!

aurora.atomic.**interp_atom_prof**(*atom_table*, *x_prof*, *y_prof*, *log_val=False*, *x_multiply=True*)
Fast interpolate atomic data in atom_table onto the x_prof and y_prof profiles. assume that x_prof, y_prof, x,y, table are all decadic logarithms and x_prof, y_prof are equally spaced(always for ADAS data) log_val bool: return natural logarithm of the data x_multiply bool: multiply output by 10**x_prof , it will not not multiplied if x_prof is None

return data interpolated on shape(nt,nion,nr)

aurora.atomic.**main_ion_brems**(*Zi*, *ni*, *ne*, *Te*)
Main-ion bremsstrahlung in units of mW/nm/sr/m^3.cm^3.

It is likely better to calculate this from H/D/T plt files, which will have more accurate Gaunt factors with the correct density dependence.

aurora.atomic.**null_space**(*A*)
Find null space of matrix A

aurora.atomic.**plot_radiation_profs**(*atom_data*, *nz_prof*, *logne_prof*, *logTe_prof*, *xvar*, *imp='F'*, *plot=False*)
Obtain profiles of predicted radiation.

This function can be used to plot radial profiles (setting xvar to a radial grid) or profiles as a function of any variable on which the logne_prof and logTe_prof may depend.

The variable :param:nz_prof may be a full description of impurity charge state densities (e.g. the output of Aurora), or profiles of fractional abundances from ionization equilibrium.

Note that the variables :param:xvar (array) and :param:imp (str) are only needed if plotting is requested.

aurora.atomic.**plot_relax_time**(*logTe*, *rate_coeff*, *ne_mean*, *ax=None*)
Plot relaxation time of the ionization equilibrium corresponding to the inverse of the given rate coefficients

aurora.atomic.**read_adf15**(*path*, *order=1*, *Te_max=None*, *ne_max=None*, *ax=None*, *plot_log=False*, *plot_3d=False*, *recomb=False*, *pec_plot_min=None*, *pec_plot_max=None*, *plot_lines=[]*)
Read photon emissivity coefficients from an ADF15 file.

Returns a dictionary whose keys are the wavelengths of the lines in angstroms. The value is an interp2d instance that will evaluate the PEC at a desired dens, temp.

Args:

path [str] Path to adf15 file to read.

order [int, opt] Parameter to control the order of interpolation.

recomb [bool, opt] If True, fetch recombination contributions to available lines. If False, fetch only ionization contributions.

To plot PEC data:

plot_lines [list] List of lines whose PEC data should be displayed. Lines should be identified by their wavelengths. The list of available wavelengths in a given file can be retrieved by first running this function ones, checking dictionary keys, and then requesting a plot of one (or more) of them.

plot_log [bool] When plotting, set a log scale

plot_3d [bool] Display PEC data as a 3D plot rather than a 2D one.

pec_plot_min [float] Minimum value of PEC to visualize in a plot

pec_plot_max [float] Maximum value of PEC to visualize in a plot

ax [matplotlib axes instance] If not None, plot on this set of axes

Te_max [float] Maximum Te value to plot when len(plot_lines)>1

ne_max [float] Maximum ne value to plot when len(plot_lines)>1

Returns:

pec_dict [dict] Dictionary containing interpolation functions for each of the available lines of the indicated type (ionization or recombination). Each interpolation function takes as arguments the log-10 of ne and Te.

MWE: path='/home/sciortino/atomlib/atomdat_master/adf15/h/pju#h0.dat' pec = read_adf15(path, recomb=False) pec = read_adf15(path, plot_lines=[list(pec.keys())[0]], recomb=False)

This function should work with PEC files produced via adas810 or adas218.

5.1.4 aurora.radiation module

`aurora.radiation.`**`adf04_files`**`()`

Collection of trust-worthy ADAS ADF04 files. This function will be moved and expanded in ColRadPy in the near future.

`aurora.radiation.`**`compute_rad`**(*imp*, *rhop*, *time*, *imp_dens*, *ne*, *Te*, *n0=None*, *nD=None*, *nBckg=None*, *main_ion_AZ=2, 1*, *bckg_imp_AZ=12, 6*, *sxr_pls_file=None*, *sxr_prs_file=None*, *prad_flag=False*, *thermal_cx_rad_flag=False*, *spectral_brem_flag=False*, *sxr_flag=False*, *main_ion_brem_flag=False*)

Calculate radiation terms corresponding to a simulation result. Results are in SI units (NB: inputs are not).

Result can be conveniently plotted with a time-slider using, for example

codes*=
[commandchars=
{}] zmax = imp_dens.shape[1] *# number of charge states (including neutrals)* rad = res['impurity_radiation '][:,:zmax,:] *# no fully-stripped line radiation* aurora.slider_plot(rhop,time, rad.transpose(1,2,0)/1e6, xlabel=r' ρ_p', ylabel='time [s]', zlabel=r'P_{rad} [MW]', plot_sum=**True**, labels=[f'Ca$^{{str(i)}}$' **for** i **in** np.arange(nz_w.shape[1]-1)])

Note that, when sxr_flag=True, SXR radiation will be computed using the default ADAS 'pls' and 'prs' files (line and continuum radiation) given by the atomic.adas_files_dict() unless the sxr_pls_file and sxr_prs_file parameters are provided.

*All radiation outputs are given in W * cm^-3, consistently with units of cm^-3 given for inputs.*

Args:

imp [str] Impurity symbol, e.g. Ca, F, W

rhop [array (space,)] Sqrt of poloidal flux radial grid of simulation output.

time [array (time,)] Time array of simulation output.

imp_dens [array (time, nZ, space)] Dictionary with impurity density result, as given by `run_aurora()` method.

ne [array (time,space) [cm^-3]] Electron density on the output grids.

Te [array (time,space) [eV]] Electron temperature on the output grids.

Optional: n0 : array(time,space), optional [cm^-3]

Background neutral density (assumed of hydrogen-isotopes). This is only used if

nD [array (time,space), optional [cm^-3]] Main ion density. This is only used if main_ion_brem_flag=True. Note that the impurity density of imp_dens times its Z value is internally automatically subtracted from this main ion density.

nBckg [array (time,space), optional [cm^-3]] Background impurity density. This is only used if main_ion_brem_flag=True. Note that this can be of any impurity for which atomic data is available. The atomic symbol of this ion is taken to be 'bckg_imp_name'.

main_ion_AZ [2-tuple, optional] Mass number (number of neutrons+protons in nucleus) and Z for the main ion (background) species. Default is (1,1), corresponding to hydrogen. This is only used if main_ion_brem_flag=sxr_flag=True.

bckg_imp_AZ [2-tuple, optional] Mass number (number of neutrons+protons in nucleus) and Z for the background impurity species. Default is (12,6), corresponding to carbon. This is only used if main_ion_brem_flag=sxr_flag=True. Note that atomic data must be available for this calculation to be possible.

sxr_pls_file [str] ADAS file used for SXR line radiation calculation if sxr_flag=True. If left to None, the default in `adas_files_dict()` is used.

sxr_prs_file [str] ADAS file used for SXR continuum radiation calculation if sxr_flag=True. If left to None, the default in `adas_files_dict()` is used.

Flags: prad_flag : bool, optional

If True, total radiation is computed (for each charge state and their sum)

thermal_cx_rad_flag [bool, optional] If True, thermal charge exchange radiation is computed.

spectral_brem_flag [bool, optional] If True, spectral bremstrahlung is computed (based on available 'brs' ADAS file)

sxr_flag [bool, optional] If True, soft x-ray radiation is computed (for the given 'pls','prs' ADAS files)

main_ion_brem_flag [bool, optional] If True, main ion bremstrahlung (all contributions) is computed.

Returns:

res [dict] Dictionary containing the radiation terms, depending on the activated flags. The structure of this output is intentionally left to be the same as in STRAHL for convenience.
If all flags were on, the dictionary would include {'impurity_radiation','spectral_bremsstrahlung','sxr_radiation'}
Impurity_radiation and sxr_radiation: index 0: total line radiation of neutral impurity index 1: total line radiation of singly ionised impurity .... index n-1: total line radiation of hydrogen-like ion index n: bremsstrahlung due to electron scattering at main ion (if requested) index n+1: total continuum radiation of impurity (bremsstrahlung and recombination continua) index n+2: bremsstrahlung due to electron scattering at impurity index n+3: total radiation of impurity (and main ion, if set in Xx.atomdat)
Spectral_bremsstrahlung: index 0: = 0 index 1: bremsstrahlung due to electron scattering at singly ionised impurity .... index n: bremsstrahlung due to electron scattering at fully ionised impurity index n+1: bremsstrahlung due to electron scattering at main ion index n+2: total bremsstrahlung of impurity (and main ion, if set in Xx.atomdat)

aurora.radiation.**get_pec_prof**(*ion*, *cs*, *rhop*, *ne_cm3*, *Te_eV*, *lam_nm=1.8705*, *lam_width_nm=0.002*, *meta_idxs=[0]*, *adf04_repo='/home/sciortino/adf04_files/ca/ca_adf04_adas/'*, *pec_threshold=1e-20*, *phot2energy=True*, *plot=True*)

Compute radial profile for Photon Emissivity Coefficients (PEC) for lines within the chosen wavelength range using the ColRadPy package. This is an alternative to the option of using the :py:method:atomic.read_adf15() function to read PEC data from an ADAS ADF-15 file and interpolate results on ne,Te grids.

Args:

ion [str] Ion atomic symbol

cs [str] Charge state, given in format like 'Ca18+'

rhop [array (nr,)] Srt of normalized poloidal flux radial array

ne_cm3 [array (nr,)] Electron density in cm^-3 units

Te_eV [array (nr,)] Electron temperature in eV units

lam_nm [float] Center of the wavelength region of interest [nm]

lam_width_nm [float] Width of the wavelength region of interest [nm]

meta_idxs [list of integers] List of levels in ADF04 file to be treated as metastable states.

adf04_repo [str] Location where ADF04 file from :py:method:adf04_files() should be fetched.

prec_threshold [float] Minimum value of PECs to be considered, in photons.cm^3/s

phot2energy [bool] If True, results are converted from photons.cm^3/s to W.cm^3

plot [bool] If True, plot lines profiles and total

Returns:

pec_tot_prof [array (nr,)] Radial profile of PEC intensity, in units of photons.cm^3/s (if phot2energy=False) or W.cm^3 depending (if phot2energy=True).

aurora.radiation.**radiation_model**(*imp*, *rhop*, *ne_cm3*, *Te_eV*, *gfilepath*, *n0_cm3=None*, *nz_cm3=None*, *frac=None*, *plot=False*)

Model radiation from a fixed-impurity-fraction model or from detailed impurity density profiles for the chosen ion. This method acts as a wrapper for :py:method:compute_rad(), calculating radiation terms over the radius and integrated over the plasma cross section.

Args:

imp [str (nr,)] Impurity ion symbol, e.g. W

rhop [array (nr,)] Sqrt of normalized poloidal flux array from the axis outwards

ne_cm3 [array (nr,)] Electron density in cm^-3 units.

Te_eV [array (nr,)] Electron temperature in eV

gfilepath [str] name of gfile to be loaded for equilibrium.

n0_cm3 [array (nr,), optional] Background ion density (H,D or T). If provided, charge exchange (CX) recombination is included in the calculation of charge state fractional abundances.

nz_cm3 [array (nr,nz), optional] Impurity charge state densities in cm^-3 units. Fractional abundancies can alternatively be specified via the :param:frac parameter for a constant-fraction impurity model across the radius. If provided, nz_cm3 is used.

frac [float, optional] Fractional abundance, with respect to ne, of the chosen impurity. The same fraction is assumed across the radial profile. If left to None, nz_cm3 must be given.

plot [bool, optional] If True, plot a number of diagnostic figures.

Returns:

res [dict] Dictionary containing results of radiation model.

5.1.5 aurora.grids_utils module

Methods to create radial and time grids for aurora simulations.

aurora.grids_utils.**create_aurora_time_grid**(*timing*, *plot=False*)

Create time grid for simulations using a Fortran routine for definitions. The same functionality is offered by `create_time_grid()`, which however is written in Python. This method is legacy code; it is recommended to use the other.

Args:

timing [dict] Dictionary containing timing['times'],timing['dt_start'],timing['steps_per_cycle'],timing['dt_increase'] which define the start times to change dt values at, the dt values to start with, the number of time steps before increasing the dt by dt_increase. The last value in each of these arrays is used for sawteeth, whenever these are modelled, or else are ignored. This is the same time grid definition as used in STRAHL.

plot [bool, optional] If True, display the created time grid.

Returns:

time [array] Computational time grid corresponding to *timing* input.

save [array] Array of zeros and ones, where ones indicate that the time step will be stored in memory in aurora simulations. Points corresponding to zeros will not be returned to spare memory.

aurora.grids_utils.**create_radial_grid**(*namelist*, *plot=False*)

Create radial grid for aurora based on K, dr_0, dr_1, rvol_lcfs and bound_sep parameters. The lim_sep parameters is additionally used if plotting is requested.

Radial mesh points are set to be equidistant in the coordinate ρ, with

$$\rho = \frac{r}{\Delta r_{centre}} + \frac{r_{edge}}{k+1}\left(\frac{1}{\Delta r_{edge}} - \frac{1}{\Delta r_{centre}}\right)\left(\frac{r}{r_{edge}}\right)^{k+1}$$

The corresponding radial step size is

$$\Delta r = \left[\frac{1}{\Delta r_{centre}} + \left(\frac{1}{\Delta r_{edge}} - \frac{1}{\Delta r_{centre}}\right)\left(\frac{r}{r_{edge}}\right)^{k}\right]^{-1}$$

See the STRAHL manual for details.

Args:

namelist [dict] Dictionary containing aurora namelist. This function uses the K, dr_0, dr_1, rvol_lcfs and bound_sep parameters. Additionally, lim_sep is used if plotting is requested.

plot [bool, optional] If True, plot the radial grid spacing vs. radial location.

Returns:

rvol_grid [array] Volume-normalized grid used for aurora simulations.

pro [array] Normalized first derivatives of the radial grid, defined as pro = (drho/dr)/(2 d_rho) = rho'/(2 d_rho)

qpr [array] Normalized second derivatives of the radial grid, defined as qpr = (d^2 rho/dr^2)/(2 d_rho) = rho"/(2 d_rho)

prox_param [float] Grid parameter used for perpendicular loss rate at the last radial grid point.

aurora.grids_utils.**create_time_grid**(*timing=None*, *plot=False*)

Create time grid for simulations using the Fortran implementation of the time grid generator.

Args:

timing [dict] Dictionary containing timing elements: 'times', 'dt_start', 'steps_per_cycle','dt_increase' As in STRAHL, the last element in each of these arrays refers to sawtooth events.

plot [bool] If True, plot time grid.

Returns:

time [array] Computational time grid corresponding to :param:timing input.

save [array] Array of zeros and ones, where ones indicate that the time step will be stored in memory in aurora simulations. Points corresponding to zeros will not be returned to spare memory.

aurora.grids_utils.**create_time_grid_new**(*timing*, *verbose=False*, *plot=False*)

Define time base for Aurora based on user inputs This function reproduces the functionality of STRAHL's time_steps.f Refer to the STRAHL manual for definitions of the time grid

Args:

n [int] Number of elements in time definition arrays

t [array] Time vector of the time base changes

dtstart [array] dt value at the start of a cycle

itz [array] cycle length, i.e. number of time steps before increasing dt

tinc : factor by which time steps should be increasing within a cycle

verbose [bool] If Trueprint to terminal a few extra info

Returns:

t_vals [array] Times in the time base [s]

i_save [array] Array of 0,1 values indicating at which times internal arrays should be stored/returned.

~~~~~~~~~~~~ THIS ISN'T FUNCTIONAL YET! ~~~~~~~~~~~~
~~~~~~~~~~~~

aurora.grids_utils.**get_HFS_LFS**(*geqdsk*, *rho_pol=None*)

Get high-field-side (HFS) and low-field-side (LFS) major radii from the g-EQDSK data. This is useful to define the r_V grid outside of the LCFS. See the `get_rhopol_rV_mapping()` for an application.

Args:

geqdsk [dict] Dictionary containing the g-EQDSK file as processed by the *omfit_eqdsk* package.

rho_pol [array, optional] Array corresponding to a grid in sqrt of normalized poloidal flux for which a corresponding r_V grid should be found. If left to None, an arbitrary grid will be created internally.

Returns:

Rhfs [array] Major radius [m] on the HFS

Rlfs [array] Major radius [m] on the LFS

aurora.grids_utils.**get_rhopol_rV_mapping**(*geqdsk*, *rho_pol=None*)

Compute arrays allowing 1-to-1 mapping of rho_pol and r_V, both inside and outside the LCFS.

r_V is defined as $\sqrt{V/(2\pi^2 R_{axis}}$ inside the LCFS. Outside of it, we artificially expand the LCFS to fit true equilibrium at the midplane based on the rho_pol grid (sqrt of normalized poloidal flux).

Method:

$$
\begin{aligned}
r(\rho,\theta) &= r_0(\rho) + (r_{lcfs}(\theta) - r_{0,lcfs}) \times \{ \\
z(\rho,\theta) &= z_0 + (z_{lcfs}(\theta) - z_0) \times \{ \\
\{ &= \frac{r(\rho,\theta=0) - r(\rho,\theta=180)}{r_{lcfs}(\theta=0) - r_{lcfs}(\theta=180)} \\
r_{0,lcfs} &= \frac{1}{2}(r_{lcfs}(\theta=0) + r_{lcfs}(\theta=180)) \\
r_0(\rho) &= \frac{1}{2}(r(\rho,\theta=0) + r(\rho,\theta=180))
\end{aligned}
$$

The mapping between rho_pol and r_V allows one to interpolate inputs on a rho_pol grid onto the r_V grid (in cm) used internally by the code.

Args:

geqdsk [dict] Dictionary containing the g-EQDSK file as processed by the *omfit_eqdsk* package.

rho_pol [array, optional] Array corresponding to a grid in sqrt of normalized poloidal flux for which a corresponding r_V grid should be found. If left to None, an arbitrary grid will be created internally.

Returns:

rho_pol [array] Sqrt of normalized poloidal flux grid

r_V [array] Mapping of rho_pol to a radial grid defined in terms of normalized flux surface volume.

5.1.6 aurora.coords module

aurora.coords.**rV_vol_average**(*quant*, *r_V*)

Calculate a volume average of the given radially-dependent quantity on a r_V grid. This function makes useof the fact that the r_V radial coordinate, defined as r_V = sqrt{ V / (2 pi^2 R_{axis} }, maps shaped volumes onto a circular geometry, making volume averaging a trivial operation via langle Q

angle = Sigma_i Q(r_i) 2 pi Delta r_V where $Delta r_V$ is the spacing between radial points in r_V.

Note that if the input r_V coordinate is extended outside the LCFS, this function will return the effective volume average also in the SOL, since it is agnostic to the presence of the LCFS.

Args:

quant [array, (space, ...)] quantity that one wishes to volume-average. The first dimension must correspond to r_V, but other dimensions may be exist afterwards.

r_V [array, (space,)] Radial r_V coordinate in cm units.

Returns:

quant_vol_avg [array, (space, ...)] Volume average of the quantity given as an input, in the same units as in the input

aurora.coords.**rad_coord_transform**(*x*, *name_in*, *name_out*, *geqdsk*)

Transform from one radial coordinate to another.

Args:

x: array input x coordinate

name_in: str input x coordinate name ('rhon','r_V','rhop','rhov','Rmid','rmid','roa')

name_out: str input x coordinate ('rhon', 'r_V', 'rhop','rhov','Rmid','rmid','roa')

geqdsk: dict gEQDSK dictionary, as obtained from the omfit-eqdsk package.

Returns: Conversion of *x* for the requested radial grid coordinate.

aurora.coords.**vol_average**(*quant*, *rhop*, *method='omfit'*, *geqdsk=None*, *device=None*, *shot=None*, *time=None*, *return_geqdsk=False*)

Calculate the volume average of the given radially-dependent quantity on a rhop grid.

Args:

quant [array, (space, ...)] quantity that one wishes to volume-average. The first dimension must correspond to space, but other dimensions may be exist afterwards.

rhop [array, (space,)] Radial rhop coordinate in cm units.

method [{'omfit','fs'}] Method to evaluate the volume average. The two options correspond to the way to compute volume averages via the OMFIT fluxSurfaces classes and via a simpler cumulative sum in r_V coordinates. The methods only slightly differ in their results. Note that 'omfit' will fail if rhop extends beyond the LCFS, while method 'fs' can estimate volume averages also into the SOL. Default is method='omfit'.

geqdsk [output of the omfit_eqdsk.OMFITgeqdsk class, postprocessing the EFIT geqdsk file] containing the magnetic geometry. If this is left to None, the function internally tries to fetch it using MDS+ and omfit_eqdsk. In this case, device, shot and time to fetch the equilibrium are required.

device [str] Device name. Note that routines for this device must be implemented in omfit_eqdsk for this to work.

shot [int] Shot number of the above device, e.g. 1101014019 for C-Mod.

time [float] Time at which equilibrium should be fetched in units of ms.

return_geqdsk [bool] If True, omfit_eqdsk dictionary is also returned

Returns:

quant_vol_avg [array, (space, ...)] Volume average of the quantity given as an input, in the same units as in the input. If extrapolation beyond the range available from EFIT volume averages over a shorter section of the radial grid will be attempted. This does not affect volume averages within the LCFS.

geqdsk [dict] Only returned if return_geqdsk=True.

5.1.7 aurora.source_utils module

Methods related to impurity source functions.
sciortino, 2020

aurora.source_utils.**get_aurora_source**(*namelist*, *time=None*)

Load source function based on current state of the namelist.

`time' is only needed for time-dependent sources

aurora.source_utils.**get_radial_source**(*namelist*, *radius_grid*, *S*, *pro*, *Ti=None*)

Obtain spatial dependence of source function.

If namelist['source_width_in']==0 and namelist['source_width_out']==0, the source radial profile is defined as an exponential decay due to ionization of neutrals. This requires S, the ionization rate of neutral impurities, to be given with S.shape=(len(radius_grid),)

If axkopt=True, the neutrals speed is taken as the thermal speed based on Ti, otherwise the value corresponding to the namelist['imp_energy'] energy is used.

This funtion reproduces the functionality of neutrals.f of STRAHL.

aurora.source_utils.**lbo_source_function**(*t_start*, *t_rise*, *t_fall*, *n_particles=1.0*, *time_vec=None*)

Model for the expected shape of the time-dependent source function, using a convolution of a gaussian and an exponential decay.

Args:

t_start [float or array-like [ms]] Injection time, beginning of source rise. If multiple values are given, they are used to create multiple source functions.

t_rise [float or array-like [ms]] Time scale of source rise. Similarly to t_start for multiple values.

t_fall [float or array-like [ms]] Time scale of source decay.Similarly to t_start for multiple values.

n_particles [float, opt] Total number of particles in source. Similarly to t_start for multiple values. Defaults to 1.0.

time_vec [array-like] Time vector on which to create source function. If left to None, use a linearly spaced time vector including the main features of the function.

Returns:

time_vec [array] Times for the source function of each given impurity

source [array] Time history of the synthetized source function.

aurora.source_utils.**read_source**(*filename*)

Read a STRAHL source file from {imp}flx{shot}.dat locally.

Args:

filename [str] Location of the file containing the STRAHL source file.

Returns:

t [array of float, (*n*,)] The timebase (in seconds).

s [array of float, (*n*,)] The source function (#/s).

aurora.source_utils.**write_source**(*t*, *s*, *shot*, *imp='Ca'*)

Write a STRAHL source file.

This will overwrite any {imp}flx{shot}.dat locally.

Args:

t [array of float, (*n*,)] The timebase (in seconds).

s [array of float, (*n*,)] The source function (in particles/s).

shot [int] Shot number, only used for saving to a .dat file

imp [str, optional] Impurity species atomic symbol

Returns:

contents [str] Content of the source file written to {imp}flx{shot}.dat

5.1.8 aurora.plot_tools module

aurora.plot_tools.**get_ls_cycle**()

aurora.plot_tools.**plot_norm_ion_freq**(*S_z*, *q_prof*, *R_prof*, *m_imp*, *Ti_prof*, *nz_profs=None*, *rhop=None*, *plot=True*, *eps_prof=None*)

Compare effective ionization rate for each charge state with the characteristic transit time that a non-trapped and trapped impurity ion takes to travel a parallel distance L = q R.

If the normalized ionization rate is less than 1, then flux surface averaging of background asymmetries (e.g. from edge or beam neutrals) can be considered in a "flux-surface-averaged" sense; otherwise, local effects (i.e. not flux-surface-averaged) may be too important to ignore.

This function is inspired by Dux et al. NF 2020. Note that in this paper the ionization rate averaged over all charge state densities is considered. This function avoids the averaging over charge states, unless these are provided as an input.

Args:

S_z [array (r,cs) [s^-1]] Effective ionization rates for each charge state as a function of radius. Note that, for convenience within aurora, cs includes the neutral stage.

q_prof [array (r,)] Radial profile of safety factor

R_prof [array (r,) or float [m]] Radial profile of major radius, either given as an average of HFS and LFS, or also simply as a scalar (major radius on axis)

m_imp [float [amu]] Mass of impurity of interest in amu units (e.g. 2 for D)

Ti_prof [array (r,)] Radial profile of ion temperature [eV]

nz_profs [array (r,cs), optional] Radial profile for each charge state. If provided, calculate average normalized ionization rate over all charge states.

rhop [array (r,), optional] Sqrt of poloidal flux radial grid. This is used only for (optional) plotting.

plot [bool, optional] If True, plot results.

eps_prof [array (r,), optional] Radial profile of inverse aspect ratio, i.e. r/R, only used if plotting is requested.

Returns:

nu_ioniz_star [array (r,cs) or (r,)] Normalized ionization rate. If nz_profs is given as an input, this is an average over all charge state; otherwise, it is given for each charge state.

aurora.plot_tools.**slider_plot**(*x*, *y*, *z*, *xlabel=''*, *ylabel=''*, *zlabel=''*, *labels=None*, *plot_sum=False*, *x_line=None*, *y_line=None*, ***kwargs*)

Make a plot to explore multidimensional data.

Args:

x [array of float, (*M*,)] The abscissa. (in aurora, often this may be rhop)

y [array of float, (*N*,)] The variable to slide over. (in aurora, often this may be time)

z [array of float, (*P*, *M*, *N*)] The variables to plot.

xlabel [str, optional] The label for the abscissa.

ylabel [str, optional] The label for the slider.

zlabel [str, optional] The label for the ordinate.

labels [list of str with length *P*] The labels for each curve in *z*.

plot_sum [bool, optional] If True, will also plot the sum over all *P* cases. Default is False.

x_line [float, optional] x coordinate at which a vertical line will be drawn.

y_line [float, optional] y coordinate at which a horizontal line will be drawn.

5.1.9 aurora.default_nml module

Method to load default namelist. This should be complemented with additional info by each user.
sciortino, July 2020

aurora.default_nml.**load_default_namelist**()

Load default namelist. Users should modify and complement this for a successful run.

5.1.10 aurora.interp module

This script contains a number of functions used for interpolation of kinetic profiles and D,V profiles in STRAHL. Refer to the STRAHL manual for details.

aurora.interp.**exppol0**(*params*, *d*, *rLCFS*, *r*)

aurora.interp.**exppol1**(*params*, *d*, *rLCFS*, *r*)

aurora.interp.**funct**(*params*, *rLCFS*, *r*)

Function 'funct' in STRAHL manual # y0 is core offset # y1 is edge offset # y2 (>y0, >y1) sets the gaussian amplification # p0 sets the width of the inner gaussian # P1 sets the width of the outer gaussian # p2 sets the location of the inner and outer peaks

aurora.interp.**funct2**(*params*, *rLCFS*, *r*)

Function 'funct2' in STRAHL manual.

aurora.interp.**interp**(*x*, *y*, *rLCFS*, *r*)

aurora.interp.**interp_quad**(*x*, *y*, *d*, *rLCFS*, *r*)

aurora.interp.**interpa_quad**(*x*, *y*, *rLCFS*, *r*)

aurora.interp.**ratfun**(*params*, *d*, *rLCFS*, *r*)

5.1.11 aurora.animate module

aurora.animate.**animate_aurora**(*x*, *y*, *z*, *xlabel=''*, *ylabel=''*, *zlabel=''*, *labels=None*, *plot_sum=False*, *uniform_y_spacing=True*, *save_filename=None*)

Produce animation of time- and radially-dependent results from aurora.

Args:

x [array of float, (*M*,)] The abscissa. (in aurora, often this may be rhop)

y [array of float, (*N*,)] The variable to slide over. (in aurora, often this may be time)

z [array of float, (*P*, *M*, *N*)] The variables to plot.

xlabel [str, optional] The label for the abscissa.

ylabel [str, optional] The label for the animated coordinate. This is expected in a format such that ylabel.format(y_val) will display a good moving label, e.g. ylabel='t={:.4f} s'.

zlabel [str, optional] The label for the ordinate.

labels [list of str with length *P*] The labels for each curve in *z*.

plot_sum [bool, optional] If True, will also plot the sum over all *P* cases. Default is False.

uniform_y_spacing [bool, optional] If True, interpolate values in z onto a uniformly-spaced y grid

save_filename [str] If a valid path/filename is provided, the animation will be saved here in mp4 format.

5.1.12 aurora.particle_conserv module

aurora.particle_conserv.**get_particle_nums**(*filepath=None*, *ds=None*, *R_axis=None*)

Check time evolution and particle conservation in (py)STRAHL output. If filepath is None, load data from output STRAHL file, otherwise use xarray Dataset in ds.

R_axis [cm] is needed for volume integrals. If None, it is set to a C-Mod default.

aurora.particle_conserv.**plot_1d**(*filepath=None*, *ds=None*, *linestyle='-'*, *axs=None*, *R_axis=None*)

Check time evolution and particle conservation in (py)STRAHL output.

If filepath is None, load data from output STRAHL file, otherwise use xarray Dataset in ds.

If axs is passed, it is assumed that this is a tuple of 2 set of axes: the first one for the separate particle time variation in each reservoir, the second for the total particle-conservation check.

R_axis [cm] is the major radius, used for volume integrals. If None, it is set to a C-Mod default.

aurora.particle_conserv.**plot_particle_conserv**(*res*, *ax=None*)

Plot number of particles in each Aurora simulation reservoir.

aurora.particle_conserv.**vol_int**(*ds*, *var*, *rhop_max=None*, *R_axis=68.5*)

Perform a volume integral of an input variable. If the variable is f(t,x) then the result is f(t). If the variable is f(t,*,x) then the result is f(t,charge) when "*" represents charge, line index, etc...

Parameters

- **ds** – xarray dataset containing STRAHL result
- **var** – Name of the variable in the strahl_result.cdf file

Params rhop_max Maximum normalized poloidal flux for integral

Params R_axis major radius on axis [cm]

Returns Time history of volume integrated variable

5.1.13 aurora.nbi_neutrals module

Methods for neutral beam analysis, particularly in relation to impurity transport studies. These script collects functions that should be device-agnostic.

aurora.nbi_neutrals.**beam_grid**(*uvw_src*, *axis*, *max_radius=255.0*)

Method to obtain the 3D orientation of a beam with respect to the device. The uvw_src and (normalized) axis arrays may be obtained from the d3d_beams method of fidasim_lib.py in the FIDASIM module in OMFIT.

This is inspired by *beam_grid* in fidasim_lib.py of the FIDASIM module (S. Haskey) in OMFIT.

aurora.nbi_neutrals.**bt_rate_maxwell_average**(*sigma_fun*, *Ti*, *E_beam*, *m_bckg*, *m_beam*, *n_level*)

Calculates Maxwellian reaction rate for a beam with atomic mass *m_beam*, energy *E_beam*, firing into a target with atomic mass *m_bckg* and temperature *T*.

sigma_fun must be a function for a specific charge and n-level of the beam particles. Ref: FIDASIM atomic_tables.f90 bt_maxwellian_n_m.

Args:

sigma_fun: function to compute a specific cross section [cm^2], function of energy/amu ONLY.
Expected call form: sigma_fun(erel/ared)

Ti [target temperature [keV], float, 1D or 2D array] Results will be computed for each Ti value in a vectorized manner.

E_beam : beam energy [keV] m_bckg : target atomic mass [amu] m_beam : beam atomic mass [amu] n_level : n-level of beam. This is used to evaluate the hydrogen ionization potential,

below which an electron is unlikely to charge exchange with surrounding ions.

Returns: rate : output reaction rate in [cm^3/s] units

aurora.nbi_neutrals.**bt_rate_maxwell_average_vec**(*sigma_fun*, *Ti*, *E_beam*, *m_bckg*, *m_beam*, *n_level*)

Calculates Maxwellian reaction rate for a beam with atomic mass *m_beam*, energy *E_beam*, firing into a target with atomic mass *m_bckg* and temperature *T*.

sigma_fun must be a function for a specific charge and n-level of the beam particles. Ref: FIDASIM atomic_tables.f90 bt_maxwellian_n_m.

This version of the function attempts to vectorize the calculation such that we can have Ti being a function of space and time and deal with integrations in vr and vz with no loops.

Args:

sigma_fun: function to compute a specific cross section [cm^2], function of energy/amu ONLY.
Expected call form: sigma_fun(erel/ared)

Ti : target temperature [keV], float, 1D or 2D array E_beam : beam energy [keV] m_bckg : target atomic mass [amu] m_beam : beam atomic mass [amu] n_level : n-level of beam. This is used to evaluate the hydrogen ionization potential,

below which an electron is unlikely to charge exchange with surrounding ions.

Returns: rate : output reaction rate in [cm^3/s] units

~~~~~~~~~~ UNTESTED!~~~~~~~~~~~

aurora.nbi_neutrals.**get_NBI_imp_cxr_q**(*neut_fsa*, *q*, *rhop_Ti*, *times_Ti*, *Ti_prof*, *include_fast=True*, *include_halo=True*, *debug_plots=False*)

Compute flux-surface-averaged (FSA) charge exchange recombination for a given impurity with neutral beam components, applying appropriate Maxwellian averaging of cross sections and obtaining rates in [s^-1] units. This method expects all neutral components to be given in a dictionary with a structure that is independent of NBI model.

Note that while Ti may be time-dependent, with a time base given by times_Ti, the FSA neutrals are expected to be time-independent. Hence, the resulting CXR rates will only have time dependence that reflects changes in Ti, but not the NBI.

Args:

neut_fsa [dict] Dictionary containing FSA neutral densities in the form that is output by *get_neutrals_fsa()*.

q [int or float] Charge of impurity species

rhop_Ti [array-like] Sqrt of poloidal flux radial coordinate for Ti profiles.

times_Ti [array-like] Time base on which Ti_prof is given [s].

Ti_prof [array-like] Ion temperature profile on the rhop_Ti, times_Ti bases.

include_fast [bool, optional] If True, include CXR rates from fast NBI neutrals. Default is True.

include_halo [bool, optional] If True, include CXR rates from themral NBI halo neutrals. Default is True.

debug_plots [bool, optional] If True, plot several plots to assess the quality of the calculation.

Returns:

rates [dict] Dictionary containing CXR rates from NBI neutrals. This dictionary has analogous form to the *get_neutrals_fsa()* function, e.g. we have

codes*=
[commandchars=
{}] rates[beam][f'n={n_level}'][’halo’]
Rates are on a radial grid corresponding to the input neut_fsa[‘rhop’].

For details on inputs and outputs, it is recommendeded to look at the internal plotting functions.

aurora.nbi_neutrals.**get_ls_cycle**()

aurora.nbi_neutrals.**get_neutrals_fsa**(*neutrals*, *geqdsk*, *debug_plots=True*)

Compute charge exchange recombination for a given impurity with neutral beam components, obtaining rates in [s^-1] units. This method expects all neutral components to be given in a dictionary with a structure that is independent of NBI model (i.e. coming from FIDASIM, NUBEAM, pencil calculations, etc.).

Args:

neutrals [dict] Dictionary containing fields {“beams”,”names”,”R”,”Z”, beam1, beam2, etc.} Here beam1,beam2,etc. are the names in neutrals[“beams”]. “names” are the names of each beam component, e.g. ‘fdens’,’hdens’,’halo’, etc., ordered according to “names”. “R”,”Z” are the major radius and vertical coordinates [cm] on which neutral density components are given in elements such as

codes*=
[commandchars=
{}] neutrals[beams[0]]["n=0"][name_idx]
It is currently assumed that n=0,1 and 2 beam components are provided by the user.

geqdsk : gEQDSK post-processed dictionary, as given by the omfit_eqdsk package.

debug_plots [bool, optional] If True, various plots are displayed.

Returns:
~~~~~~~~~~

neut_fsa [dict] Dictionary of flux-surface-averaged (FSA) neutral densities, in the same units as in the input. Similarly to the input "neutrals", this dictionary has a structure like

codes*=
[commandchars=
{}] neutrals_ext[beam][f'n=/n_level/'][name_idx]

aurora.nbi_neutrals.**rotation_matrix**(*alpha*, *beta*, *gamma*)

See the table of all rotation possiblities, on the Tait Bryan side https://en.wikipedia.org/wiki/Euler_angles#Tait.E2.80.93Bryan_angles

aurora.nbi_neutrals.**tt_rate_maxwell_average**(*sigma_fun*, *Ti*, *m_i*, *m_n*, *n_level*)

Calculates Maxwellian reaction rate for an interaction between two thermal populations, assumed to be of neutrals (mass m_n) and background ions (mass m_i).

The 'sigma_fun' argument must be a function for a specific charge and n-level of the neutral particles. This allows evaluation of atomic rates for charge exchange interactions between thermal beam halos and background ions.

Args:

sigma_fun: python function Function to compute a specific cross section [cm^2], function of energy/amu ONLY. Expected call form: sigma_fun(erel/ared)

Ti: float or 1D array background ion and halo temperature [keV]

m_i: float mass of background ions [amu]

m_n: float mass of neutrals [amu]

n_level: int n-level of beam. This is used to evaluate the hydrogen ionization potential, below which an electron is unlikely to charge exchange with surrounding ions.

TODO: add effect of toroidal rotation! This will require making the integration in this function 2-dimensional.

Returns:

rate [float or 1D array] output reaction rate in [cm^3/s] units

aurora.nbi_neutrals.**uvw_xyz**(*u*, *v*, *w*, *origin*, *R*)

Computes array elements by multiplying the rows of the first array by the columns of the second array. The second array must have the same number of rows as the first array has columns. The resulting array has the same number of rows as the first array and the same number of columns as the second array.

See uvw_to_xyz in fidasim.f90

aurora.nbi_neutrals.**xyz_uvw**(*x*, *y*, *z*, *origin*, *R*)

Computes array elements by multiplying the rows of the first array by the columns of the second array. The second array must have the same number of rows as the first array has columns. The resulting array has the same number of rows as the first array and the same number of columns as the second array.

See xyz_to_uvw in fidasim.f90

5.1.14 aurora.janev_smith_rates module

Script collecting rates from Janev & Smith, NF 1993. These are useful in aurora to compute total (n-unresolved) charge exchange rates between heavy ions and neutrals.

sciortino, 2020

aurora.janev_smith_rates.**js_sigma**(*E*, *q*, *n1*, *n2=None*, *type='cx'*)

Cross sections for collisional processes between beam neutrals and highly-charged ions, from Janev & Smith 1993.

Args:

E [float] Normalized beam energy [keV/amu]

q [int] Impurity charge before interaction (interacting ion is A^{q+})

n1 [int] Principal quantum number of beam hydrogen.

n2: int Principal quantum number of excited. This may not be needed for some transitions (if so, leave to None).

type [str] Type of interaction. Possible choices: {'exc','ioniz','cx'} where 'cx' refers to electron capture / charge exchange.

Returns:

sigma [float] Cross section of selected process, in [cm^2] units.

See comments in Janev & Smith 1993 for uncertainty estimates.

aurora.janev_smith_rates.**js_sigma_cx_n1_q1**(*E*)
Electron capture cross section for H^{+} + H(1s) –> H + H^+ Section 2.3.1

aurora.janev_smith_rates.**js_sigma_cx_n1_q2**(*E*)
Electron capture cross section for He^{2+} + H(1s) –> He^+ + H^+ Section 3.3.1

aurora.janev_smith_rates.**js_sigma_cx_n1_q4**(*E*)
Electron capture cross section for Be^{4+} + H(1s) –> Be^{3+} + H^+ Section 4.3.1

aurora.janev_smith_rates.**js_sigma_cx_n1_q5**(*E*)
Electron capture cross section for B^{5+} + H(1s) –> B^{4+} + H^+ Section 4.3.2

aurora.janev_smith_rates.**js_sigma_cx_n1_q6**(*E*)
Electron capture cross section for C^{6+} + H(1s) –> C^{5+} + H^+ Section 4.3.3

aurora.janev_smith_rates.**js_sigma_cx_n1_q8**(*E*)
Electron capture cross section for O^{8+} + H(1s) –> O^{7+} + H^+ Section 4.3.4

aurora.janev_smith_rates.**js_sigma_cx_n1_qg8**(*E*, *q*)
Electron capture cross section for A^{q+} + H(1s) –> A^{(q-1)+} + H^+, q>8 Section 4.3.5, p.172

aurora.janev_smith_rates.**js_sigma_cx_n2_q2**(*E*)
Electron capture cross section for He^{2+} + H(n=2) –> He^+ + H^+ Section 3.3.2

aurora.janev_smith_rates.**js_sigma_cx_ng1_q1**(*E*, *n1*)
Electron capture cross section for H^{+} + H(n) –> H + H^+ , n>1 Section 2.3.2

aurora.janev_smith_rates.**js_sigma_cx_ng1_qg3**(*E*, *n1*, *q*)
Electron capture cross section for A^{q+} + H^*(n) –> A^{(q-1)+}+H^+ , q>3 Section 4.3.6, p.174

aurora.janev_smith_rates.**js_sigma_cx_ng2_q2**(*E*, *n1*)
Electron capture cross section for He^{2+} + H*(n) –> He^+ + H^+ , n>2 Section 3.2.3

aurora.janev_smith_rates.**js_sigma_ioniz_n1_q8**(*E*)
Ionization cross section for O^{8+} + H(1s) –> O^{8+} + H^+ +e^- Section 4.2.4

aurora.janev_smith_rates.**plot_js_sigma**(*q=18*)

5.1.15 Module contents

aurora.**name = 'aurora'**
aurora

Indices and tablesIndices and tables

- genindex
- modindex
- search

Python Module Index

a