Aurora Documentation

Release 1.0.0

Francesco Sciortino

Dec 11, 2020

CONTENTS

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

CHAPTER

ONE

OVERVIEW

Aurora is a package to simulate heavy-ion transport and radiation in magnetically-confined plasmas. It includes a 1.5D impurity transport forward model which inherits many of the methods from the historical STRAHL code and has been thoroughly benchmarked with it. It also offers routines to analyze neutral states of hydrogen isotopes, both from the edge of fusion plasmas and from neutral beam injection. Aurora's code is mostly written in Python 3 and Fortran 90. A Julia interface has also recently been added. The package enables radiation calculations using ADAS atomic rates, which can easily be applied to the output of Aurora's own forward model, or coupled with other 1D, 2D or 3D transport codes.

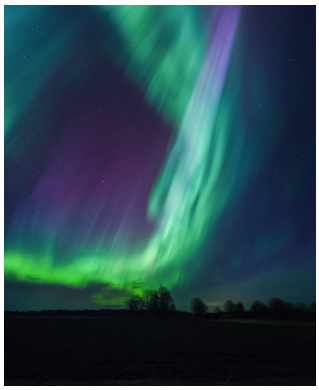

Fig. 1: Inspirational photo of the Aurora Borealis by K.Pikner

This documentation aims at making Aurora usage as clear as possible. Getting started is easy - see the *Installation* section. To learn the basics, head to the *Tutorial* section.

CHAPTER

TWO

WHAT IS AURORA USEFUL FOR?

Aurora is useful for modeling of particle transport, neutrals and radiation in fusion plasmas.

The package includes Python functionality to create inputs and read/plot outputs of impurity transport simulations. It was designed to be as efficient as possible in iterative workflows, where parameters (particularly diffusion and convection coefficients) are run through the forward model and repeatedly modified in order to match some experimental observations. For this reason, Aurora avoids any disk input-output (I/O) during operation. All data is kept in memory.

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 `aurora_sim` class, which allows creation of radial and temporal grids, interpolation of atomic rates, preparation of parallel loss rates at the edge, etc.

The `aurora.atomic` library 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 ColRadPy, using methods in `aurora.radiation`.

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. Interfacing Aurora with several file formats used throughout the fusion community to store kinetic profiles is simple.

Aurora was born as a fast forward model of impurity transport, but it can also be useful for synthetic spectroscopic diagnostics and radiation modeling in fusion plasmas. For example, it may be helpful for parameter scans to explore the performance 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. Background neutrals, either from the edge or from neutral beam injection, can be analyzed using the `aurora.neutrals` and `aurora.nbi_neutrals` libraries.

CHAPTER

THREE

DOCUMENTATION CONTENTS

3.1 Installation

3.1.1 Installing from source

We recommend installing from the latest version of the code, obtained by git-cloning the repository at

https://github.com/fsciortino/aurora

After doing this, you can run:

```
python setup.py install
```

and should do the magic.

Some users may want to have greater control over which compiler is being used for the installation; this can be most easily done by modifying the provided Makefile directly. After changing its top configuration lines, users can do:

```
make clean; make
```

3.1.2 Installing via PyPI or Anaconda

We are working to make the latest stable version of the code available via PyPI and Anaconda, but this process is not yet complete. In the near future, you should be able to do:

```
pip install aurorafusion
```

or from Anaconda Cloud:

```
conda install aurorafusion
```

3.1.3 Running with Julia

Aurora simulations can also be done using a Python-Julia interface; this makes iterative runs even faster!

Assuming that you have Julia already installed on your device, you will want to build a *sysimage* for the Aurora Julia source code. This is useful because whenever you will open a Python session the first run of Aurora using `run_aurora()` will need to pre-compile the Julia source code, which may take a couple of seconds. To create the *sysimage*, you can do:

```
make clean_julia; make julia
```

This may take a couple of minutes, but it only has to be done once.

Once the *sysimage* has been created, Python can directly make use of it and enjoy even greater speed. Note that this is only recommended for "iterative" operation, i.e. when many Aurora simulations are run within the same Python session, since the first run will take much longer than usual. All the following simulations will be faster.

Of course, interfaces to run Aurora completely in Julia are under-development (@ajcav). Interested parties should get in touch!

It may be surprising that Julia can beat good-old Fortran at what it is normally best (speed). Well, we all get used to it after some time :)

3.1.4 What's next?

After installing, see the *Tutorial* section for guidance on how to get started.

3.2 Tutorial

Assuming that you have Aurora already installed on your system, we're now ready to move forward. Some basic Aurora functionality is demonstrated in the *examples* package directory, where users may find a number of useful scripts. Here, we go through some of the same examples and methods.

3.2.1 Running Aurora simulations

If Aurora is correctly installed, you should be able to do:

```
import aurora
```

and then load a default namelist for impurity transport forward modeling:

```
namelist = aurora.load_default_namelist()
```

Note that you can always look at where this function is defined in the package by using, e.g.:

```
aurora.load_default_namelist.__module__
```

Once you have loaded the default namelist, have a look at the *namelist* dictionary. It contains a number of parameters that are needed for Aurora runs. Some of them, like the name of the device, are only important if automatic fetching of the EFIT equilibrium through *MDSplus* is required, or else it can be ignored (leaving it to its default value). Most of the parameter names should be fairly self-descriptive, but a detailed description will be available soon. In the meantime, please refer to docstrings through the code documentation.

Next, read in a magnetic equilibrium. You can find an example from a C-Mod discharge in the *examples* directory:

```
geqdsk = omfit_eqdsk.OMFITgeqdsk('example.gfile')
```

The output *geqdsk* dictionary contains the contents of the EFIT geqdsk file, with additional processing done by the *omfit_eqdsk* package for flux surfaces. Only some of the dictionary fields are used; refer to the *grids_utils* methods for details. The *geqdsk* dictionary is used to create a mapping between the *rhop* grid (square root of normalized poloidal flux) and a *rvol* grid, defined by the normalized volume of each flux surface. Aurora, like STRAHL, runs its simulations on the *rvol* grid.

We next need to read in some kinetic profiles, for example from an *input.gacode* file (available in the *examples* directory):

```
inputgacode = omfit_gapy.OMFITgacode('example.input.gacode')
```

Other file formats (e.g. plasma statefiles, TRANSP outputs, etc.) may also be read with *omfit_gapy* or other OMFIT-distributed packages. It is however not important to Aurora how the users get kinetic profiles: all that matters is that they are stored in the *namelist['kin_prof']* dictionary. To set up time-independent kinetic profiles we can use:

```
kp = namelist['kin_profs']
kp['Te']['rhop'] = kp['ne']['rhop'] = np.sqrt(inputgacode['polflux']/
↪inputgacode['polflux'][-1])
kp['ne']['vals'] = inputgacode['ne']*1e13     # 1e19 m^-3 --> cm^-3
kp['Te']['vals'] = inputgacode['Te']*1e3      # keV --> eV
```

Note that both electron density (*ne*) and temperature (*Te*) must be saved on a *rhop* grid. This grid is internally used by Aurora to map to the *rvol* grid. Also note that, unless otherwise stated, Aurora inputs are always in CGS units, i.e. all spatial quantities are given in cm!! (the extra exclamation mark is there for a good reason...).

Next, we specify the ion species that we want to simulate. We can simply do:

```
imp = namelist['imp'] = 'Ar'
```

and Aurora will internally find ADAS data for that ion (assuming that this is one of the common ones for fusion modeling). The namelist also contains information on what kind of source of impurities we need to simulate; here we are going to select a constant source (starting at t=0) of 10^{24} particles/second.:

```
namelist['source_type'] = 'const'
namelist['Phi0'] = 1e24
```

Time dependent time histories of the impurity source may however be given by selecting *namelist['source_type']="step"* (for a series of step functions), *"synth_LBO"* (for an analytic function

resembling a laser-blow-off (LBO) time history) or *"file"* (to load a detailed function from a file). Refer to the *get_source_time_history()* method for more details.

Assuming that we're happy with all the inputs in the namelist at this point (many more could be changed!), we can now go ahead and set up our Aurora simulation::

```
asim = aurora.aurora_sim(namelist, geqdsk=geqdsk)
```

The *aurora_sim* class creates a Python object with spatial and temporal grids, kinetic profiles, atomic rates and all other inputs to the forward model. Aurora uses a diffusive-convective model for particle fluxes, so we need to specify diffusion (D) and convection (V) coefficients next::

```
D_z = 1e4 * np.ones(len(asim.rvol_grid))  # cm^2/s
V_z = -2e2 * np.ones(len(asim.rvol_grid)) # cm/s
```

Here we have made use of the *rvol_grid* attribute of the *asim* object, whose name is self-explanatory. This grid has a 1-to-1 correspondence with *asim.rhop_grid*. In the lines above we have created flat profiles of $D = 10^4 cm^2/s$ and $V = -2 \times 10^2 cm/s$, defined on our simulation grids. D's and V's could in principle (and, very often, in practice) be defined with more dimensions to represent a time-dependence and also different values for different charge states. Unless specifed otherwise, Aurora assumes all points of the time grid (now stored in *asim.time_grid*) and all charge states to have the same D and V. See the `run_aurora()` method for details on how to speficy further dependencies.

At this point, we are ready to run an Aurora simulation, with:

```
out = asim.run_aurora(D_z, V_z)
```

Blazing fast! Depending on how many time and radial points you have requested (a few hundreds by default), how many charge states you are simulating, etc., a simulation could take as little as <50 ms, which is significantly faster than other code, as far as we know. If you add *use_julia=True* to the `run_aurora()` call the run will be even faster; wear your seatbelt!

You can easily check the quality of particle conservation in the various reservoirs by using:

```
reservoirs = asim.check_conservation()
```

which will show the results in full detail. The *reservoirs* output list contains information about how many particles are in the plasma, in the wall reservoir, in the pump, etc.. Refer to the `run_aurora()` docstring for details.

A plot is worth a thousand words, so let's make one for the charge state densities (on a nice slider!):

```
aurora.slider_plot(asim.rvol_grid, asim.time_out, asim.rad['line_rad'].
↪transpose(1,2,0),
                   xlabel=r'$r_V$ [cm]', ylabel='time [s]', zlabel='Total␣
↪radiation [A.U.]',
                   labels=[str(i) for i in np.arange(0,nz.shape[1])],
                   plot_sum=True, x_line=asim.rvol_lcfs )
```

Use the slider to go over time, as you look at the distributions over radius of all the charge states. It would be really great if you could just save this type of time- and spatially-dependent visualization to a video-format, right? That couldn't be easier, using the *animate_aurora()* function::

```
aurora.animate_aurora(asim.rhop_grid, asim.time_out, nz.transpose(1,0,2),
                      xlabel=r'$\rho_p$', ylabel='t={:.4f} [s]', zlabel=r'$n_z
↪$ [A.U.]',
                      labels=[str(i) for i in np.arange(0,nz.shape[1])],
                      plot_sum=True, save_filename='aurora_anim')
```

After running this, a .mp4 file with the name "aurora_anim.mp4" will be saved locally.

3.2.2 Radiation predictions

Once a set of charge state densities has been obtained, it is simple to compute radiation terms in Aurora. For example, using the results from the Aurora run in *Running Aurora simulations*, one can then run:

```
asim.rad = aurora.compute_rad(imp, nz.transpose(2,1,0), asim.ne, asim.Te,␣
↪prad_flag=True)
```

The documentation on `compute_rad()` gives details on input array dimensions and various flags that may be turned on. In the case above, we simply indicated the ion number (*imp*), and provided charge state densities (with dimensions of time, charge state and space), electron density and temperature (dimensions of time and space). We then explicitely indicated *prad_flag=True*, which means that unfiltered "effective" radiation terms (line radiation and continuum radiation) should be computed. Bremsstrahlung is also estimated using an interpolation formula that is independent of ADAS data and can be found in *asim.rad['brems']*. However, note that bremsstrahlung is already included in *asim.rad['cont_rad']*, which also includes other terms including continuum recombination using ADAS data. It can be useful to compare the bremsstrahlung calculation in *asim.rad['brems']* with *asim.rad['cont_rad']*, but we recommend that users rely on the full continuum prediction for total power estimations.

Other possible flags of the `compute_rad()` function include:

1. *sxr_flag*: if True, compute line and continuum radiation in the SXR range using the ADAS "pls" and "prs" files. Bremsstrahlung is also separately computed using the ADAS "pbs" files.
2. *thermal_cx_rad_flag*: if True, the code checks for inputs *n0* (atomic H/D/T neutral density) and *Ti* (ion temperature) and computes line power due to charge transfer from thermal background neutrals and impurities.
3. *spectral_brem_flag*: if True, use the ADAS "brs" files to compute bremsstrahlung at a wavelength specified by the chosen file.

All of the radiation flags are *False* by default.

ADAS files for all calculations are taken by default from the list of files indicated in `adas_files_dict()` function, but may be replaced by specifying the *adas_files* dictionary argument to `compute_rad()`.

Results from `compute_rad()` are collected in a dictionary (named "rad" above and added as an attribute to the "asim" object, for convenience) with clear keys, described in the function documentation. To get a quick plot of the radiation profiles, e.g. for line radiation from all simulated charge states, one can do:

```
aurora.slider_plot(asim.rvol_grid, asim.time_out, asim.rad['line_rad'].
↪transpose(1,2,0),
                   xlabel=r'$r_V$ [cm]', ylabel='time [s]', zlabel='Total␣
↪radiation [A.U.]',
                   labels=[str(i) for i in np.arange(0,nz.shape[1])],
                   plot_sum=True, x_line=asim.rvol_lcfs)
```

Aurora's radiation modeling capabilities may also be useful when assessing total power radiation for integrated modeling. The *radiation_model()* function allows one to easily obtain the most important radiation terms at a single time slice, both as power densities (units of MW/cm^{-3}) and absolute power (units of MW). To obtain the latter form, we need to integrate over flux surface volumes. We can use the *geqdsk* dictionary obtained via:

```
geqdsk = omfit_eqdsk.OMFITgeqdsk('example.gfile')
```

(or equivalent methods/files) to then extract flux surface volumes (units of m^3) at each value of *rhop*::

```
grhop = np.sqrt(geqdsk['fluxSurfaces']['geo']['psin'])
gvol = geqdsk['fluxSurfaces']['geo']['vol']

# interpolate on our grid
vol = interp1d(grhop, gvol)(rhop)
```

We can now pass the *vol* array to *radiation_model()*, together with the impurity atomic symbol (*imp*), the *rhop* grid array, electron density (*ne_cm3*) and temperature (*Te_eV*) and, optionally, also background neutral densities to include thermal charge exchange::

```
res = aurora.radiation_model(imp,rhop,ne_cm3,Te_eV, vol,
                             n0_cm3=None, frac=0.005, plot=True)
```

Here we specified the impurity densities as a simple fraction of the electron density profile, by specifying the *frac* argument. This is obviously a simplifying assumption, effectively stating that the total impurity density profile should have a maximum amplitude of *frac* (in the case above, set to 0.005) and a profile shape (corresponding t a profile of *V/D*) that is identical to the one of the n_e profile. This may be convenient for parameter scans in the design process of future devices, but is by no means a correct assumption. If we'd rather calculate the total radiated power from a specific set of impurity charge state profiles (e.g. from an Aurora simulation), we can do:

```
res = aurora.radiation_model(imp,rhop,ne_cm3,Te_eV, vol,
                             n0_cm3=None, nz_cm3=nz_cm3, plot=True)
```

where we specified the charge state densities (dimensions of space, charge state) at a single time. Since we specified *plot=True*, a number of useful radiation profiles should be displayed.

Of course, one can also estimate radiation from the main ions. To do this, we first want to estimate the main ion density, using:

```
ni_cm3 = aurora.get_main_ion_dens(ne_cm3, ions)
```

with *ions* being a dictionary of the form:

```
ions = {'C': nC_cm3, 'Ne': nNe_cm3}    # (time,charge state,space)
```

with a number of impurity charge state densities with dimensions of (time,charge state,space). The *get_main_ion_dens()* function subtracts each of these densities (times the Z of each charge state) from the electron density to obtain a main ion density estimate based on quasineutrality. Before we move forward, we need to add a neutral stage density for the main ion species, e.g. using:

```
niz_cm3 = np.vstack((n0_cm3[None,:],ni_cm3)).T
```

such that the *niz_cm3* output is a 2D array of dimensions (charge state, radius).

To estimate main ion radiation we can now do:

```
res_mainion = aurora.radiation_model('H',rhop,ne_cm3,Te_eV, vol, nz_cm3 = niz_
↪cm3, plot=True)
```

(Note that the atomic data does not discriminate between hydrogen isotopes) In the call above, the neutral density has been included in *niz_cm3*, but note that (1) there is no radiation due to charge exchange between deuterium neutrals and deuterium ions, since they are indistinguishable, and (2) we did not attempt to include the effect of charge exchange on deuterium fractional abundances because *n0_cm3* (included in *niz_cm3* already fully specifies fractional abundances for main ions).

3.2.3 Zeff contributions

Following an Aurora run, one may be interested in what is the contribution of the simulated impurity to the total effective charge of the plasma. The *calc_Zeff()* method allows one to quickly compute this by running:

```
asim.calc_Zeff()
```

This makes use of the electron density profiles (as a function of space and time), stored in the "asim" object, and keeps Zeff contributions separate for each charge state. They can of course be plotted with *slider_plot()*::

```
aurora.slider_plot(asim.rvol_grid, asim.time_out, asim.delta_Zeff.transpose(1,
↪0,2),
                   xlabel=r'$r_V$ [cm]', ylabel='time [s]', zlabel=r'$\Delta$
↪$Z_{eff}$',
                   labels=[str(i) for i in np.arange(0,nz.shape[1])],
                   plot_sum=True,x_line=asim.rvol_lcfs)
```

3.2.4 Ionization equilibrium

It may be useful to compare and contrast the charge state distributions obtained from an Aurora run with the distributions predicted by pure ionization equilibium, i.e. by atomic physics only, with no trasport. To do this, we only need some kinetic profiles, which for this example we will load from the sample *input.gacode* file available in the "examples" directory::

```
import omfit_gapy
inputgacode = omfit_gapy.OMFITgacode('example.input.gacode')
```

Recall that Aurora generally uses CGS units, so we need to convert electron densities to cm^{-3} and electron temperatures to eV:

```
rhop = np.sqrt(inputgacode['polflux']/inputgacode['polflux'][-1])
ne_vals = inputgacode['ne']*1e13 # 1e19 m^-3 --> cm^-3
Te_vals = inputgacode['Te']*1e3  # keV --> eV
```

Here we also defined a *rhop* grid from the poloidal flux values in the *inputgacode* dictionary. We can then use the *get_atom_data()* function to read atomic effective ionization ("scd") and recombination ("acd") from the default ADAS files listed in *adas_files_dict()*. In this example, we are going to focus on calcium ions::

```
atom_data = aurora.get_atom_data('Ca',['scd','acd'])
```

In ionization equilibrium, all ionization and recombination processes will be perfectly balanced. This condition corresponds to specific fractions of each charge state at some locations that we define using arrays of electron density and temperature. We can compute fractional abundances and plot results using:

```
logTe, fz, rates = aurora.get_frac_abundances(atom_data, ne_vals, Te_vals,␣
↪rho=rhop, plot=True)
```

The *get_frac_abundances()* function returns the log-10 of the electron temperature on the same grid as the fractional abundances, given by the *fz* parameter (dimensions: space, charge state). This same function can be used to both compute radiation profiles of fractional abundances or to compute fractional abundances as a function of scanned parameters *ne* and/or *Te*. The inverse of the *rates* output correspond to the atomic relaxation time. An additional argument of *ne_tau* (units of $m^{-3} \cdot s$) can be used to approximately model the effect of transport on ionization balance.

3.2.5 Working with neutrals

Aurora includes a number of useful functions for neutral modeling, both from the edge of fusion devices (thermal neutrals) and from neutral beams (fast and halo neutrals).

For thermal neutrals, we make use of atomic data from the *Collrad* collisional-radiative model, part of the DEGAS2 code.

The `erh5_file` class allows one to parse the *erh5.dat* file of DEGAS-2 that contains useful information to assess excited state fractions of neutrals in specific kinetic backgrounds. If the *erh5.dat* file is not available already, Aurora will download it and store it locally within its distribution directory. The data in this file is used for example in the *get_exc_state_ratio()* function, which given a ground state density

of neutrals (*N1*), some ion and electron densities (*ni* and *ne*) and electron temperature (*Te*), will compute the fraction of neutrals in the principal quantum number *m*. Keyword arguments can be passed to this function to plot the results. Note that kinetic inputs may be given as a scalar or as a 1D list/array. The *plot_exc_ratios()* function may also be useful to plot the excited state ratios.

Note that in order to find the photon emissivity coefficient of specific neutral lines, the `read_adf15()` function may be used. For example, to obtain interpolation functions for neutral H Lyman-alpha emissivity, one can use:

```
filename = 'pec96#h_pju#h0.dat' # for D Ly-alpha

# fetch file automatically, locally, from AURORA_ADAS_DIR, or directly from␣
↪the web:
path = aurora.get_adas_file_loc(filename, filetype='adf15')

# plot Lyman-alpha line at 1215.2 A. See available lines with pec_dict.keys()␣
↪after calling without plot_lines argument
pec_dict = aurora.read_adf15(path, plot_lines=[1215.2])
```

This will plot the Lyman-alpha photon emissivity coefficients (both the components due to excitation and recombination) as a function of temperature in eV. Some files (e.g. try *pec96#c_pju#c2.dat*) may also have charge exchange components.

Analysis routines to work with fast and halo neutrals are also provided in Aurora. Atomic rates for charge exchange of impurities with NBI neutrals are taken from Janev & Smith NF 1993 and can be obtained from *js_sigma()*, which wraps a number of functions for specific atomic processes. To compute charge exchange rates between NBI neutrals (fast or thermal) and any ions in the plasma, users need to provide a prediction of neutral densities, likely from an external code like FIDASIM.

Neutral densities for each fast ion population (full-,half- and third-energy), multiple halo generations and a few excited states are expected. Refer to the documentation of *get_neutrals_fsa()* to read about how to provide neutrals on a poloidal cross section so that they may be "flux-surface averaged".

bt_rate_maxwell_average() shows how beam-thermal Maxwell-averaged rates can be obtained; *tt_rate_maxwell_average()* shows the equivalent for thermal-thermal Maxwell-averaged rates.

Finally, *get_NBI_imp_cxr_q()* shows how flux-surface-averaged charge exchnage recombination rates between an impurity ion of charge q with NBI neutrals (all populations, fast and thermal) can be computed for use in Aurora forward modeling. For more details, feel free to contact Francesco Sciortino (sciortino-at-psfc.mit.edu).

3.3 Requirements

3.3.1 Python requirements

Aurora uses the latest Python-3 distribution and requires a modern Fortran compiler, available on most Unix systems. Additionally, the following packages are automatically installed (from PyPI) when installing Aurora::

```
numpy scipy matplotlib xarray omfit-eqdsk omfit-gapy
```

The latter two are part of the OMFIT distribution and will provide lots of capabilities to interact with tokamak modeling tools, with which Aurora can be integrated. Note that *omfit-eqdsk* and *omfit-gapy* will themselves bring a number of automatic requirements which may take some space on disk.

3.3.2 Julia requirements

To run the Julia version of the code, Julia must be installed; see:

```
https://julialang.org/downloads/
```

Everything else should be automatically handled by the Aurora installation (see *Installation*).

3.4 Input parameters

In this page, we describe some of the most important input parameter for Aurora simulations. Since all Aurora inputs are created in Python, rather than in a low-level language, users are encouraged to browse through the module documentation to get a complete picture; here, we only look at some specific features.

3.4.1 Spatio-temporal grids

Aurora's spatial and temporal grids are defined in the same way as in STRAHL. Refer to the STRAHL manual for details. Note that only STRAHL options that have been useful in the authors' experience have been included in Aurora.

In short, the `create_radial_grid()` function produces a radial grid that is equally-spaced on the ρ grid, defined by

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

The corresponding radial step size is given by

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

The radial grid above requires a number of user parameters:

1. The *k* factor in the formulae; large values give finer grids at the plasma edge. A value of 6 is usually appropriate.
2. *dr_0* and *dr_1* give the radial spacing (in *rvol* units) at the center and at the last grid point (in cm).

3. The *r_edge* parameter in the formulae above is given by:

```
r_edge = namelist['rvol_lcfs'] + namelist['bound_sep']
```

where *rvol_lcfs* is the distance from the center to the separatrix and *bound_sep* is the distance between the separatrix and the wall boundary, both given in flux-surface-volume normalized units. The *rvol_lcfs* parameter is automatically computed by the `aurora_sim` class initialization, based on the provided *geqdsk*. *bound_sep* can be estimated via the `estimate_boundary_distance()` function, if an *aeqdsk* file can be accessed via *MDSplus* (alternatively, users may set it to anything they find appropriate). Additionally, since the edge model of Aurora simulates the presence of a limiter somewhere in between the LCFS and the wall boundary, we add a *lim_sep* parameter to specify the distance between the LCFS and the limiter surface.

To demonstrate the creation of a spatial grid, we are going to select some example parameters:

```
namelist={}
namelist['K'] = 6.
namelist['dr_0'] = 1.0  # 1 cm spacing near axis
namelist['dr_1'] = 0.1   # 0.1 cm spacing at the edge
namelist['rvol_lcfs'] = 50.0 # 50cm minor radius (in rvol units)
namelist['bound_sep'] = 5.0  # distance between LCFS and wall boundary
namelist['lim_sep'] = 3.0 # distance between LCFS and limiter

# now create grid and plot it
rvol_grid, pro_grid, qpr_grid, prox_param = create_radial_grid(namelist,
↪plot=True)
```

This will plot the radial spacing over the grid and show the location of the LCFS and the limiter, also specifying the total number of grid points. The larger the number of grid points, the longer simulations will take.

Similarly, to create time grids one needs a dictionary of input parameters, which `aurora_sim` automatically looks for in the dictionary *namelist['timing']*. The contents of this dictionary are

1. *timing['times']*: list of times at which the time grid must change. The first and last time indicate the start and end times of the simulation.
2. *timing['dt_start']*: list of time spacings (dt) at each of the times given by *timing['times']*.
3. *timing['steps_per_cycle']*: number of time steps before adapting the time step size. This defines a "cycle".
4. *timing['dt_increase']*: multiplicative factor by which the time spacing (dt) should change within one "cycle".

Let's test the creation of a grid and plot the result::

```
timing = {}
timing['times'] = [0.,0.5, 1.]
timing['dt_start'] = [1e-4,1e-3, 1e-3]  # last value not actually used,␣
↪except when sawteeth are modelled!
timing['steps_per_cycle'] = [2, 5, 1]   # last value not actually used,␣
↪except when sawteeth are modelled!
```

(continues on next page)

(continued from previous page)

```
timing['dt_increase'] = [1.005, 1.01, 1.0]  # last value not actually used,␣
↪except when sawteeth are modelled!
time, save = aurora.create_time_grid(timing, plot=True)
```

The plot title will show how many time steps are part of the time grid (given by the *time* output). The *save* output is a list of 0's and 1's that is used to indicate which time grid points should be saved to the output.

3.4.2 Recycling

A 1.5D transport model such as Aurora cannot accurately model recycling at walls. Like STRAHL, Aurora uses a number of parameters to approximate the transport of impurities outside of the LCFS; we recommend that users ensure that their core results don't depend sensitively on these parameters:

1. *recycling_flag*: if this is False, no recycling nor communication between the divertor and core plasma particle reservoirs is allowed.
2. *wall_recycling* : if this is 0, particles are allowed to move from the divertor reservoir back into the core plasma, based on the *tau_div_SOL_ms* and *tau_pump_ms* parameters, but no recycling from the wall is enabled. If >0 and <1, recycling of particles hitting the limiter and wall reservoirs is enabled, with a recycling coefficient equal to this value.
3. *tau_div_SOL_ms* : time scale with which particles travel from the divertor into the SOL, entering again the core plasma reservoir. Default is 50 ms.
4. *tau_pump_ms* : time scale with which particles are completely removed from the simulation via a pumping mechanism in the divertor. Default is 500 ms (very long)
5. *tau_rcl_ret_ms* : time scale of recycling retention at the wall. This parameter is not present in STRAHL. It is introduced to reproduce the physical observation that after an ELM recycling impurities may return to the plasma over a finite time scale. Default is 50 ms.
6. *SOL_mach*: Mach number in the SOL. This is used to compute the parallel loss rate, both in the open SOL and in the limiter shadow. Default is 0.1.
7. *divbls* : fraction of user-specified impurity source that is added to the divertor reservoir rather than the core plasma reservoir. These particles can return to the core plasma only if *recycling_flag=True* and *wall_recycling>=0*. This parameter is useful to simulate divertor puffing.

The parallel loss rate in the open SOL and limiter shadow also depends on the local connection length. This is approximated by two parameters: *clen_divertor* and *clen_limiter*, in the open SOL and the limiter shadow, respectively. These connection lengths can be approximated using the edge safety factor and the major radius from the *geqdsk*, making use of the `estimate_clen()` function.

3.5 Atomic data

Almost all of the Aurora functionality depends on having access to Atomic Data and Analysis Structure (ADAS) rates. These are needed to determine effective ionization and recombination rates for all charge states, estimate radiated power, soft X-ray contributions, charge exchange components, etc..

Everything that is needed can be obtained from the OPEN-ADAS website:

https://open.adas.ac.uk/

Aurora attempts to make atomic data usage as simple as possible. The *adas_files_dict()* function gives a dictionary of recommended files that users can adopt (but also easily override, if other files are preferable). See the `get_file_types()` function docstring for a brief description of each relevant file type.

The *adas_data* directory at the base of the Aurora distribution is where ADAS atomic data should be stored, separately for ADF11 (iso-nuclear master files) and ADF15 (photon emissivity coefficients). When running Aurora, the *get_adas_file_loc()* function automatically checks whether the requested ADF11 file is available in *adas_data/adf11/* or in a directory that users may specify by setting an environmental variable *AURORA_ADAS_DIR*. If the requested file is not available here either, Aurora attempts to fetch it automatically from the OPEN-ADAS website. Each ADF11 file is stored in *adas_data* after usage, so downloading over the internet is only done if no other option is available.

Atomic data is also used for radiation predictions, both via ADAS ADF11 files (iso-nuclear master files, giving effective coefficients for combined atomic processes) and via ADF15 files (photon emissivity coefficients - PECs - for specific atomic lines):

(a) A number of functions are available in the *radiation* module to plot effective radiation terms, e.g. total line radiation for an ion, main ion bremsstrahlung, etc.

(b) The `read_adf15()` function allows reading and plotting of ADF15, making it easy to evaluate PECs for specific densities and temperatures by using the returned interpolation functions. PEC components due to excitation, recombination and charge exchange can all be easily loaded and plotted. However, Aurora users may also make use of the coupling to [ColRadPy](https://github.com/johnson-c/ColRadPy) to produce PECs using ADAS ADF04 files and running ColRadPy's collisional-radiative model. This functionality is already available in the `get_pec_prof()` function and will be further developed in the future.

See the tutorial in *Radiation predictions* for more information on these subjects.

3.6 Citing Aurora

While Aurora is released publicly in a selfless effort to support the development of fusion energy, we do appreciate users pushing our numbers by giving a star to the Aurora Github repo

https://github.com/fsciortino/aurora

and by citing the following works:

[1] F. Sciortino et al 2020 Nucl. Fusion 60 126014, https://doi.org/10.1088/1741-4326/abae85

This paper presented the original application of Aurora to infer impurity transport coefficients in Alcator C-Mod plasmas. Here, the code is referred to as *pySTRAHL* (what a terrible name).

[2] R. Dux, 2004, Habilitation Thesis, MPI-IPP. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.830.8834&rep=rep1&type=pdf

The work of R. Dux on STRAHL is at the basis of many of the methods adopted by Aurora. While Aurora's code does not depend on STRAHL in any way, it owes to it for laying much of the groundwork.

3.7 Questions and contributions

For any questions on Aurora, to brainstorm on possible applications or request changes to the code, please contact sciortino-at-psfc.mit.edu.

The code is developed and maintained by F. Sciortino (MIT-PSFC) in collaboration with T. Odstrcil (GA) and A. Cavallaro (MIT), with support from O. Linder (MPI-IPP) and C. Johnson (U. Auburn). The great wisdom (and patience) of S. Smith (GA) has allowed this code to be effectively shared and documented. Finally, the STRAHL documentation provided by R.Dux (MPI-IPP) was extremely helpful to guide code development.

New contributors are more than welcome! Please get in touch via email or open a pull-request via Github.

Generally, we would appreciate if you could work with us to merge your features back into the main Aurora distribution if there is any chance that the changes that you made could be useful to others.

3.8 Aurora modules

3.8.1 Submodules

3.8.2 aurora.core module

This module includes the core class to set up simulations with `aurora`. 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.

calc_Zeff()
Compute Zeff from each charge state density, using the result of an Aurora simulation. The total Zeff change over time and space due to the simulated impurity can be simply obtained by summing over charge states

Results are stored as an attribute of the simulation object instance.

centrifugal_asym(*omega*, *Zeff*, *plot=False*)

Estimate impurity poloidal asymmetry effects from centrifugal forces. See notes the *centrifugal_asym()* function docstring for details.

In this function, we use the average Z of the impurity species in the Aurora simulation result, using only the last time slice to calculate fractional abundances. The CF lambda factor

Parameters

- **omega** – array (nt,nr) or (nr,) [rad/s] Toroidal rotation on Aurora temporal time_grid and radial rhop_grid (or, equivalently, rvol_grid) grids.
- **Zeff** – array (nt,nr), (nr,) or float Effective plasma charge on Aurora temporal time_grid and radial rhop_grid (or, equivalently, rvol_grid) grids. Alternatively, users may give Zeff as a float (taken constant over time and space).

Keyword Arguments **plot** – bool If True, plot asymmetry factor λ vs. radius

Returns

array (nr,) Asymmetry factor, defined as λ in the *centrifugal_asym()* function docstring.

Return type CF_lambda

check_conservation(*plot=True*, *axs=None*, *plot_resolutions=False*)

Check particle conservation for an aurora simulation.

Args :

plot [bool, optional] If True, plot time histories in each particle reservoir and display quality of particle conservation.

axs [matplotlib.Axes instances, optional] Axes to pass to *check_particle_conserv()* These may be the axes returned from a previous call to this function, to overlap results for different runs.

Returns :

out [dict] Dictionary containing density of particles in each reservoir.

axs [matplotlib.Axes instances , only returned if plot=True] New or updated axes returned by *check_particle_conserv()*

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.

plot_resolutions()
Convenience function to show time and spatial resolution in Aurora simulation setup.

run_aurora(*D_z*, *V_z*, *times_DV=None*, *nz_init=None*, *alg_opt=1*, *evolneut=False*, *use_julia=False*, *plot=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

```
aurora.slider_plot(rhop,time, nz.transpose(1,2,0),
                   xlabel=r'$\rho_p$', 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]])
```

Parameters

- **D_z** – arrays, shape of (space,time,nZ) or (space,time) or (space,) 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 D_z and V_z are found to be have only 2 dimensions, it is assumed that all charge states should have the same transport coefficients. If they are only 1-D, it is further assumed that they are time-independent. Note that it is assumed that D_z and V_z profiles are already on the self.rvol_grid radial grid.
- **V_z** – arrays, shape of (space,time,nZ) or (space,time) or (space,) 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 D_z and V_z are found to be have only 2 dimensions, it is assumed that all charge states should have the same transport coefficients. If they are only 1-D, it is further assumed that they are time-independent. Note that it is assumed that D_z and V_z profiles are already on the self.rvol_grid radial grid.

Keyword Arguments

- **times_DV** – 1D array, optional Array of times at which D_z and V_z profiles are given. By Default, this is None, which implies that D_z and V_z are time independent.
- **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.
- **alg_opt** – int, optional If alg_opt=1, use the finite-volume algorithm proposed by Linder et al. NF 2020. If alg_opt=1, use the older finite-differences algorithm in the 2018 version of STRAHL.

- **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. This option is NOT CURRENTLY RECOMMENDED, because this method is still under development/ examination.
- **use_julia** – bool, optional If True, run the Julia pre-compiled version of the code. Run the julia makefile option to set this up. Default is False (still under development)
- **plot** – bool, optional If True, plot density for each charge state using a convenient slides over time and check particle conservation in each particle reservoir.

Returns

array, (nr,nZ,nt) Charge state densities [:math::*cm^{-3}*] over the space and time grids.

N_wall [array (nt,)] Number of particles at the wall reservoir over time.

N_div [array (nt,)] Number of particles in the divertor reservoir over time.

N_pump [array (nt,)] Number of particles in the pump reservoir over time.

N_ret [array (nt,)] Number of particles temporarily held in the wall reservoirs.

N_tsu [array (nt,)] Edge particle loss [:math::*cm^{-3}*]

N_dsu [array (nt,)] Parallel particle loss [:math::*cm^{-3}*]

N_dsul [array (nt,)] Parallel particle loss at the limiter [:math::*cm^{-3}*]

rcld_rate [array (nt,)] Recycling from the divertor [:math::*s^{-1} cm^{-3}*]

rclw_rate [array (nt,)] Recycling from the wall [:math::*s^{-1} cm^{-3}*]

Return type nz

setup_grids()
Method to set up radial and temporal grids given namelist inputs.

setup_kin_profs_depts()
Method to set up Aurora inputs related to the kinetic background from namelist inputs.

3.8.3 aurora.atomic module

Collection of classes and functions for loading, interpolation and processing of atomic data. Refer also to the adas_files.py script.

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 density and temperature grids. Note that such grids vary between files, and the species they refer to may too.

Refer to ADAS documentation for details on each file.

load()

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

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_file_types**()

Obtain a description of each ADAS file type and its meaning in the context of Aurora. For background, refer to:

```
Summers et al., "Ionization state, excited populations and emission of␣
↪impurities
in dynamic finite density plasmas: I. The generalized collisional-
↪radiative model for
light elements", Plasma Physics and Controlled Fusion, 48:2, 2006
```

Returns Dictionary with keys given by the ADAS file types and values giving a description for them.

aurora.atomic.**get_atom_data**(*imp*, *filetypes=['acd', 'scd']*, *filenames=[]*)

Collect atomic data for a given impurity from all types of ADAS files available or for only those requested.

Parameters

- **imp** – str Atomic symbol of impurity ion.
- **filetypes** – list or array-like ADAS file types to be fetched. Default is ["acd","scd"] for effective ionization and recombination rates (excluding CX).
- **filenames** – list or array-like, optional ADAS file names to be used in place of the defaults given by adas_file_dict(). If left empty, such defaults are used. Note that the order of filenames must be the same as the one in the "filetypes" list.

Returns

dict Dictionary containing data for each of the requested files. Each entry of the dictionary gives log-10 of ne, log-10 of Te and log-10 of the data as attributes atom_data[key].logNe, atom_data[key].logT, atom_data[key].data

Return type atom_data

aurora.atomic.**get_cooling_factors**(*atom_data*, *logTe_prof*, *fz*, *plot=True*, *ax=None*)

Calculate cooling coefficients for the given fractional abundances and kinetic profiles.

Parameters

- **atom_data** – dict Dictionary containing atomic data as output by *get_atom_data()* for the atomic processes of interest. "prs","pls","plt" and "prb" are required by this function.
- **logTe_prof** – array (nt,nr) Log-10 of electron temperature profile (in eV)
- **fz** – array (nt,nr) Fractional abundances for all charge states of the ion of "atom_data"
- **plot** – bool If True, plot all radiation components, summed over charge states.
- **ax** – matplotlib.Axes instance If provided, plot results on these axes.

Returns

array (nt,nr) Line radiation in the SXR range for each charge state

prs [array (nt,nr)] Continuum radiation in the SXR range for each charge state

pltt [array (nt,nr)] Line radiation (unfiltered) for each charge state. NB: this corresponds to the ADAS "plt" files. An additional "t" is added to the name to avoid conflict with the common matplotlib.pyplot short form "plt"

prb [array (nt,nr)] Continuum radiation (unfiltered) for each charge state

Return type pls

aurora.atomic.**get_cs_balance_terms**(*atom_data*, *ne_cm3=50000000000000.0*, *Te_eV=None*, *maxTe=10000.0*, *include_cx=True*)

Get S, R and cx on the same logTe grid.

Parameters

- **atom_data** – dictionary of atomic ADAS files (only acd, scd are required; ccd is necessary only if include_cx=True
- **ne_cm3** – float or array Electron density in units of cm^-3
- **Te_eV** – 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

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.

Return type logTe

aurora.atomic.**get_frac_abundances**(*atom_data*, *ne_cm3*, *Te_eV=None*, *n0_by_ne=1e-05*, *include_cx=False*, *ne_tau=inf*, *plot=True*, *ax=None*, *rho=None*, *rho_lbl=None*, *ls='-'*)

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

This method can work with ne,Te and n0_by_ne arrays of arbitrary dimension, but plotting is only supported in 1D (defaults to flattened arrays).

Parameters

- **atom_data** – dictionary of atomic ADAS files (only acd, scd are required; ccd is necessary only if include_cx=True
- **ne_cm3** – float or array Electron density in units of cm^-3
- **Te_eV** – 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.
- **ne_tau** – float, opt Value of electron density in $m^{-3} \cdot s$:math:` imes` particle residence time. This is a scalar value that can be used to model the effect of transport on ionization equilibrium. Setting ne_tau=np.inf (default) corresponds to no effect from transport.
- **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.

Returns

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

Return type logTe

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*)

Approximate bremsstrahlung in units of $mW/nm/sr/m^3 \cdot cm^3$, or equivalently W/m^3 for full spherical emission.

Note that this may not be very useful, since this contribution is already included in the continuum radiation component in ADAS files. The bremsstrahlung estimate in ADAS continuum radiation files is more accurate than the one give by this function, which uses a simpler interpolation of the Gaunt factor with weak ne-dependence. Use with care!

Parameters

- **nz** – array (time,nZ,space) Densities for each charge state [cm^{-3}]
- **ne** – array (time,space) Electron density [cm^{-3}]
- **Te** – array (time,space) Electron temperature [cm^{-3}]

Returns

array (time,nZ,space) Bremsstrahlung for each charge state

Return type brems

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. This function assume that x_prof, y_prof, x,y, table are all base-10 logarithms, and x_prof, y_prof are equally spaced.

Parameters

- **atom_table** – list List with x,y, table = atom_table, containing atomic data from one of the ADAS files.
- **x_prof** – array (nt,nr) Spatio-temporal profiles of the first coordinate of the ADAS file table (usually electron density in cm^-3)
- **y_prof** – array (nt,nr) Spatio-temporal profiles of the second coordinate of the ADAS file table (usually electron temperature in eV)
- **log_val** – bool If True, return natural logarithm of the data
- **x_multiply** – bool If True, multiply output by 10**x_prof.

Returns

array (nt,nion,nr) Interpolated atomic data on time,charge state and spatial grid that correspond to the ion of interest and the spatiotemporal grids of x_prof and y_prof.

Return type interp_vals

aurora.atomic.**null_space**(*A*)

Find null space of matrix A

aurora.atomic.**plot_norm_ion_freq**(*S_z*, *q_prof*, *R_prof*, *imp_A*, *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.

Parameters

- **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)
- **imp_A** – float [amu] Atomic mass number, i.e. number of protons + neutrons (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

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.

Return type nu_ioniz_star

aurora.atomic.**plot_relax_time**(*logTe*, *rate_coeff*, *ax=None*)

Plot relaxation time of the ionization equilibrium corresponding to the inverse of the given rate coefficients.

Parameters

- **logTe** – array (nr,) log-10 of Te [eV], on an arbitrary grid (same as other arguments, but not necessarily radial)

- **rate_coeff** – array (nr,) Rate coefficients from ionization balance. See *get_frac_abundances()* to obtain these via the "compute_rates" argument. N.B.: these rate coefficients will depend also on electron density, which does affect relaxation times.
- **ax** – matplotlib axes instance, optional If provided, plot relaxation times on these axes.

3.8.4 aurora.adas_files module

Functions to provide default ADAS files for Aurora modelling, including capabilities to fetch these files remotely from the OPEN-ADAS website.

aurora.adas_files.**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.

Returns

dict Dictionary with keys equal to the atomic symbols of many of the most common ions of interest in fusion research. For each ion, a sub-dictionary contains recommended file names for the relevant ADAS data types. Not all files types are available for all ions. Files types are usually a subset of 'acd','scd','prb','plt','ccd','prc','pls','prs','fis','brs','pbs',prc' Refer to *get_adas_file_types()* for a description of the meaning of each file.

Return type files

aurora.adas_files.**fetch_adf11_file**(*filename*)

Download ADF11 file from the OPEN-ADAS website and store it in the 'adas_data/adf11' directory.

Parameters **filename** – str Name of ADF11 file to be downloaded, e.g. 'plt89_ar.dat'.

aurora.adas_files.**fetch_adf15_file**(*filename*)

Download ADF15 file from the OPEN-ADAS website and store it in the 'adas_data/adf15' directory.

Parameters **filename** – str Name of ADF15 file to be downloaded, e.g. 'pec96#c_pju#c2.dat'.

aurora.adas_files.**get_adas_file_loc**(*filename*, *filetype='adf11'*)

Find location of requested atomic data file for the indicated ion. The search proceeds with the following attempts, in this order:

1. If the file is available in Aurora/adas_data/*filetype*, with filetype given by the user, always use this data.
2. If the environmental variable "AURORA_ADAS_DIR" is defined, attempt to find the file there and copy it to Aurora/adas_data/*filetype*.
3. Attempt to fetch the file remotely via open.adas.ac.uk and save it in Aurora/adas_data/*filetype*/.

Parameters

- **filename** – str Name of the ADAS file of interest, e.g. 'plt89_ar.dat'
- **filetype** – str ADAS file type. Currently allowed: 'adf11' or 'adf15'

Returns

str Full path to the requested file.

Return type file_loc

3.8.5 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*, *nz*, *ne*, *Te*, *n0=None*, *Ti=None*, *ni=None*, *adas_files_sub={}*, *prad_flag=False*, *sxr_flag=False*, *thermal_cx_rad_flag=False*, *spectral_brem_flag=False*)
Calculate radiation terms corresponding to a simulation result. The nz,ne,Te,n0,Ti,ni arrays are normally assumed to be given as a function of (time,nZ,space), but time and space may be substituted by other coordinates (e.g. R,Z)

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

```
aurora.slider_plot(rhop,time, res['line_rad'].transpose(1,2,0)/1e6,
    xlabel=r'$\rho_p$', ylabel='time [s]',
    zlabel=r'$P_{rad}$ [$MW$]',
    plot_sum=True,
    labels=[f'Ca$^{{{str(i)}}}$' for i in np.arange(res['line_rad'].
↪shape[1])])
```

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

Parameters

- **imp** – str Impurity symbol, e.g. Ca, F, W
- **nz** – array (time, nZ, space) [cm^{-3}] 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.

Keyword Arguments

- **n0** – array(time,space), optional [cm^{-3}] Background neutral density (assumed of hydrogen-isotopes). This is only used if thermal_cx_rad_flag=True.
- **Ti** – array (time,space) [eV] Main ion temperature (assumed of hydrogen-isotopes). This is only used if thermal_cx_rad_flag=True. If not set, Ti is taken equal to Te.
- **adas_files_sub** – dict Dictionary containing ADAS file names for radiation calculations, possibly including keys "plt","prb","prc","pls","prs","pbs","brs"

Any file names that are needed and not provided will be searched in the *adas_files_dict()* dictionary.

- **prad_flag** – bool, optional If True, total radiation is computed (for each charge state and their sum)
- **sxr_flag** – bool, optional If True, soft x-ray radiation is computed (for the given 'pls','prs' ADAS files)
- **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)

Returns

dict Dictionary containing radiation terms, depending on the activated flags. The structure of the "res" dictionary is as follows.

If prad_flag=True,

res['line_rad'] [array (nt,nZ,nr)- from ADAS "plt" files] Excitation-driven line radiation for each impurity charge state.

res['cont_rad'] [array (nt,nZ,nr)- from ADAS "prb" files] Continuum and line power driven by recombination and bremsstrahlung for impurity ions.

res['brems'] [array (nt,nr)- analytic formula.] Bremsstrahlung produced by electron scarrering at fully ionized impurity This is only an approximate calculation and is more accurately accounted for in the 'cont_rad' component.

res['thermal_cx_cont_rad'] [array (nt,nZ,nr)- from ADAS "prc" files] Radiation deriving from charge transfer from thermal neutral hydrogen to impurity ions. Returned only if thermal_cx_rad_flag=True.

res['tot'] [array (nt,nZ,nr)] Total unfilted radiation, summed over all charge states, given by the sum of all known radiation components.

If sxr_flag=True,

res['sxr_line_rad'] [array (nt,nZ,nr)- from ADAS "pls" files] Excitation-driven line radiation for each impurity charge state in the SXR range.

res['sxr_cont_rad'] [array (nt,nZ,nr)- from ADAS "prs" files] Continuum and line power driven by recombination and bremsstrahlung for impurity ions in the SXR range.

res['sxr_brems'] [array (nt,nZ,nr)- from ADAS "pbs" files] Bremsstrahlung produced by electron scarrering at fully ionized impurity in the SXR range.

res['sxr_tot'] [array (nt,nZ,nr)] Total radiation in the SXR range, summed over all charge states, given by the sum of all known radiation components in the SXR range.

If spectral_brem_flag,

res['spectral_brems'] [array (nt,nZ,nr) – from ADAS "brs" files] Bremsstrahlung at a specific wavelength, depending on provided "brs" file.

Return type res

aurora.radiation.**get_colradpy_pec_prof**(*ion*, *cs*, *rhop*, *ne_cm3*, *Te_eV*, *lam_nm=1.8705*, *lam_width_nm=0.002*, *meta_idxs=[0]*, *adf04_repo='/home/docs/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 *read_adf15()* function to read PEC data from an ADAS ADF-15 file and interpolate results on ne,Te grids.

Parameters

- **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 $photons \cdot cm^3/s$
- **phot2energy** – bool If True, results are converted from $photons \cdot cm^3/s$ to $W.cm^3$
- **plot** – bool If True, plot lines profiles and total

Returns

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

Return type pec_tot_prof

aurora.radiation.**get_main_ion_dens**(*ne_cm3*, *ions*, *rhop_plot=None*)

Estimate the main ion density via quasi-neutrality. This requires subtracting from ne the impurity charge state density times Z for each charge state of every impurity present in the plasma in significant amounts.

Parameters

- **ne_cm3** – array (time,space) Electron density in cm^{-3}
- **ions** – dict Dictionary with keys corresponding to the atomic symbol of each impurity under consideration. The values in ions[key] should correspond to the charge state densities for the selected impurity ion in cm^{-3}, with shape (time,nZ,space).
- **rhop_plot** – array (space), optional rhop radial grid on which densities are given. If provided, plot densities of all species at the last time slice over this radial grid.

Returns

array (time,space) Estimated main ion density in cm^{-3}.

Return type ni_cm3

aurora.radiation.**plot_radiation_profs**(*imp*, *nz_prof*, *logne_prof*, *logTe_prof*, *xvar_prof*, *xvar_label=''*, *atom_data=None*)

Compute profiles of predicted radiation, both SXR-filtered and unfiltered. This function offers a simplified interface to radiation calculation with respect to *compute_rad()*, which is more complete.

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

The variable "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.

Parameters

- **imp** – str, optional Impurity ion atomic symbol.
- **nz_prof** – array (TODO for docs: check dimensions) Impurity charge state densities
- **logne_prof** – array (TODO for docs: check dimensions) Electron density profiles in cm^{-3}
- **logTe_prof** – array (TODO for docs: check dimensions) Electron temperature profiles in eV
- **xvar_prof** – array (TODO for docs: check dimensions) Profiles of a variable of interest, on the same grid as kinetic profiles.
- **xvar_label** – str, optional Label for x-axis.
- **atom_data** – dict, optional Dictionary containing atomic data as output by *get_atom_data()* for the atomic processes of interest. "prs","pls","plt" and "prb" are required by this function. If not provided, this function loads these files internally.

Returns

array (TODO for docs: check dimensions) SXR line radiation.

prs [array (TODO for docs: check dimensions)] SXR continuum radiation.

pltt [array (TODO for docs: check dimensions)] Unfiltered line radiation.

prb [array (TODO for docs: check dimensions)] Unfiltered continuum radiation.

Return type pls

aurora.radiation.**radiation_model**(*imp*, *rhop*, *ne_cm3*, *Te_eV*, *vol*, *adas_files_sub={}*, *n0_cm3=None*, *Ti_eV=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.

Parameters

- **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
- **vol** – array (nr,) Volume of each flux surface in m^3. Note the units! We use m^3 here rather than cm^3 because it is more common to work with m^3 for flux surface volumes of fusion devices.

Keyword Arguments

- **adas_files_sub** – dict Dictionary containing ADAS file names for forward modeling and/or radiation calculations. Possibly useful keys include "scd","acd","ccd","plt","prb","prc","pls","prs","pbs","brs" Any file names that are needed and not provided will be searched in the *adas_files_dict()* dictionary.
- **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.
- **Ti_eV** – array (nr,), optional Background ion density (H,D or T). This is only used if CX recombination is requested, i.e. if n0_cm3 is not None. If not given, Ti is set equal to Te.
- **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

dict Dictionary containing results of radiation model.

Return type res

aurora.radiation.**read_adf15**(*path*, *order=1*, *plot_lines=[]*, *ax=None*, *Te_max=None*, *ne_max=None*, *plot_log=False*, *plot_3d=False*, *pec_plot_min=None*, *pec_plot_max=None*)

Read photon emissivity coefficients from an ADAS 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 density and temperature.

Parameters

- **path** – str Path to adf15 file to read.
- **order** – int, opt Parameter to control the order of interpolation.

Keyword Arguments

- **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.
- **ax** – matplotlib axes instance If not None, plot on this set of axes
- **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
- **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

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.

Return type pec_dict

Minimal Working Example (MWE):

```
filename = 'pec96#h_pju#h0.dat' # for D Ly-alpha

# fetch file automatically, locally, from AURORA_ADAS_DIR, or directly␣
↪from the web:
path = aurora.get_adas_file_loc(filename, filetype='adf15')

# plot Lyman-alpha line at 1215.2 A. See available lines with pec_dict.
↪keys() after calling without plot_lines argument
pec_dict = aurora.read_adf15(path, plot_lines=[1215.2])
```

Another example, this time also with charge exchange::

```
filename = 'pec96#c_pju#c2.dat'
path = aurora.get_adas_file_loc(filename, filetype='adf15')
pec_dict = aurora.read_adf15(path, plot_lines=[361.7])
```

Metastable-resolved files will be automatically identified and parsed accordingly, e.g.

filename = 'pec96#he_pjr#he0.dat' path = aurora.get_adas_file_loc(filename, filetype='adf15') pec_dict = aurora.read_adf15(path, plot_lines=[584.4])

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

3.8.6 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.

Parameters

- **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

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.

Return type time

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.

Parameters

- **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

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.

Return type rvol_grid

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

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

Parameters

- **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

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.

Return type time

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

Parameters

- **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

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.

Return type t_vals

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

aurora.grids_utils.**estimate_boundary_distance**(*shot*, *device*, *time_ms*)

Obtain a simple estimate for the distance between the LCFS and the wall boundary. This requires access to the A_EQDSK on the EFIT01 tree on MDS+. Users who may find that this call does not work for their device may try to adapt the OMFITmdsValue TDI string.

Parameters

- **shot** – int Discharge/experiment number
- **device** – str Name of device, e.g. 'C-Mod', 'DIII-D', etc.
- **time_ms** – int or float Time at which results for the outer gap should be taken.

Returns

float Estimate for the distance between the wall boundary and the separatrix [cm]

lim_sep [float] Estimate for the distance between the limiter and the separatrix [cm]. This is (quite arbitrarily) taken to be 2/3 of the bound_sep distance.

Return type bound_sep

aurora.grids_utils.**estimate_clen**(*geqdsk*)

Estimate average connection length in the open SOL and in the limiter shadow NB: these are just rough numbers!

Parameters **geqdsk** – dict EFIT g-EQDSK as processed by the omfit_eqdsk package.

Returns

float Estimate of the connection length to the divertor

clen_limiter [float] Estimate of the connection length to the limiter

Return type clen_divertor

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 rvol grid outside of the LCFS. See the `get_rhopol_rV_mapping()` for an application.

Parameters

- **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 rvol grid should be found. If left to None, an arbitrary grid will be created internally.

Returns

array Major radius [m] on the HFS

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

Return type Rhfs

`aurora.grids_utils.`**`get_rhopol_rvol_mapping`**(*geqdsk*, *rho_pol=None*)

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

rvol 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:

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

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

Parameters

- **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 rvol grid should be found. If left to None, an arbitrary grid will be created internally.

Returns

array Sqrt of normalized poloidal flux grid

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

Return type rho_pol

3.8.7 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. Note that this coordinate conversion is only strictly valid inside of the LCFS.

Parameters

- **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.

Parameters

- **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

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.

Return type quant_vol_avg

3.8.8 aurora.source_utils module

Methods related to impurity source functions.

sciortino, 2020

aurora.source_utils.**get_radial_source**(*namelist*, *rvol_grid*, *pro_grid*, *S_rates*, *Ti_eV=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_rates, the ionization rate of neutral impurities, to be given with S_rates.shape=(len(rvol_grid),len(time_grid))

If namelist['imp_source_energy_eV']<0, the neutrals speed is taken as the thermal speed based on Ti_eV, otherwise the value corresponding to namelist['imp_source_energy_eV'] is used.

Parameters

- **namelist** – dict Aurora namelist. Only elements referring to the spatial distribution and energy of source atoms are accessed.
- **rvol_grid** – array (nr,) Radial grid in volume-normalized coordinates [cm]

- **pro_grid** – array (nr,) Normalized first derivatives of the radial grid in volume-normalized coordinates.
- **S_rates** – array (nr,nt) Ionization rate of neutral impurity over space and time.

Keyword Arguments **Ti_eV** – array, optional (nr,nt) Background ion temperature, only used if source_width_in=source_width_out=0.0 and imp_source_energy_eV<=0, in which case the source impurity neutrals are taken to have energy equal to the local Ti [eV].

Returns

array (nr,nt) Radial profile of the impurity neutral source for each time step.

Return type source_rad_prof

aurora.source_utils.**get_source_time_history**(*namelist*, *Raxis_cm*, *time*)

Load source time history based on current state of the namelist.

There are 4 options to describe the time-dependence of the source:

(1) namelist['source_type'] == 'file': in this case, a simply formatted source file, with one time point and corresponding and source amplitude on each line, is read in. This can describe an arbitrary time dependence, e.g. as measured from an experimental diagnostic.

(2) namelist['source_type'] == 'const': in this case, a constant source (e.g. a gas puff) is simulated. It is recommended to run the simulation for >100ms in order to see self-similar charge state profiles in time.

(3) namelist['source_type'] == 'step': this allows the creation of a source that suddenly appears and suddenly stops, i.e. a rectangular "step". The duration of this step is given by namelist['step_source_duration']. Multiple step times can be given as a list in namelist['src_step_times']; the amplitude of the source at each step is given in namelist['src_step_rates']

(4) namelist['source_type'] == 'synth_LBO': this produces a model source from a LBO injection, given by a convolution of a gaussian and an exponential. The required parameters in this case are inside a namelist['LBO'] dictionary: namelist['LBO']['t_start'], namelist['LBO']['t_rise'], namelist['LBO']['t_fall'], namelist['LBO']['n_particles']. The "n_particles" parameter corresponds to the amplitude of the source (the number of particles corresponding to the integral over the source function.

Parameters

- **namelist** – dict Aurora namelist dictionary.
- **Raxis_cm** – float Major radius at the magnetic axis [cm]. This is needed to normalize the source such that it is treated as toroidally symmetric – a necessary idealization for 1.5D simulations.
- **time** – array (nt,), optional Time array the source should be returned on.

Returns

array (nt,) The source time history on the input time base.

Return type source_time_history

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.

Parameters

- **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

array Times for the source function of each given impurity

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

Return type time_vec

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

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

Parameters **filename** – str Location of the file containing the STRAHL source file.

Returns

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

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

Return type t

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

Write a STRAHL source file.

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

Parameters

- **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

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

Return type contents

3.8.9 aurora.plot_tools module

aurora.plot_tools.**get_color_cycle**()

aurora.plot_tools.**get_line_cycle**()

aurora.plot_tools.**get_ls_cycle**()

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.

Parameters

- **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.

3.8.10 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.

3.8.11 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

The "params" input is broken down into 6 arguments: 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*)
Function 'interp' used in STRAHL for D and V profiles.

aurora.interp.**interp_quad**(*x*, *y*, *d*, *rLCFS*, *r*)
Function 'interp' used for kinetic profiles.

aurora.interp.**interpa_quad**(*x*, *y*, *rLCFS*, *r*)
Function 'interpa' used for kinetic profiles

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

3.8.12 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.

Parameters

- **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.

3.8.13 aurora.particle_conserv module

aurora.particle_conserv.**check_particle_conserv**(*Raxis_cm*, *ds=None*, *filepath=None*, *linestyle='-'*, *plot=True*, *axs=None*)

Check time evolution and particle conservation in Aurora or STRAHL output.

Parameters

- **Raxis_cm** – float Major radius on axis [cm], used for volume integrals.
- **ds** – xarray dataset, optional Dataset containing Aurora results, created using the xarray package. See `check_conservation()` for an illustration on how to use this.
- **filepath** – str, optional If provided, load results from STRAHL output file and check particle particle conservation as for an Aurora run.
- **linestyle** – str, optional matplotlib linestyle, default is '-' (continuous lines). Use this to overplot lines on the same plots using different linestyles, e.g. to check whether some aurora option makes particle conservation better or worse.
- **plot** – bool, optional If True, plot time histories of particle densities in each simulation reservoir.
- **axs** – 2-tuple or array array-like structure containing two matplotlib.Axes instances: the first one for the separate particle time variation in each reservoir, the second for the total particle-conservation check. This can be used to plot results from several aurora runs on the same axes.

Returns

dict Dictionary containing time histories across all reservoirs, useful for the assessment of particle conservation.

axs [2-tuple or array, only returned if plot=True] array-like structure containing two matplotlib.Axes instances, (ax1,ax2). See optional input argument.

Return type out

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

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

- **Raxis_cm** – float Major radius on axis [cm]
- **ds** – xarray dataset Dataset containing Aurora or STRAHL result
- **var** – str Name of the variable in the strahl_result.cdf file
- **rhop_max** – float Maximum normalized poloidal flux for integral. If not provided, integrate over the entire simulation grid.

Returns

array (nt,) Time history of volume integrated variable

Return type var_volint

3.8.14 aurora.neutrals module

Aurora functionality for edge neutral modeling. The ehr5 file from DEGAS2 is used. See https://w3.pppl.gov/degas2/ for details.

aurora.neutrals.**download_ehr5_file**()

Download the ehr5.dat file containing atomic data describing the multi-step ionization and recombination of hydrogen.

See https://w3.pppl.gov/degas2/ for details.

class aurora.neutrals.**ehr5_file**(*filepath=None*)

Bases: object

Read ehr5.dat file from DEGAS2. Returns a dictionary containing

- Ionization rate Seff in $cm^3 s^{-1}$
- Recombination rate Reff in $cm^3 s^{-1}$
- Neutral electron losses $E_{loss}^{(i)}$ in $ergs^{-1}$
- Continuum electron losses $E_{loss}^{(ii)}$ in $ergs^{-1}$
- Neutral "n=2 / n=1", $N_2^{(i)}/N_1$
- Continuum "n=2 / n=1", :math:`N_2^{(ii)}/N_11
- Neutral "n=3 / n=1", $N_3^{(i)}/N_1$
- Continuum "n=3 / n=1", $N_3^{(ii)}/N_1$

... and similarly for n=4 to 9.

Refer to the DEGAS2 manual for details.

load()

plot(*field='Seff'*, *fig=None*, *axes=None*)

aurora.neutrals.**get_exc_state_ratio**(*m*, *N1*, *ni*, *ne*, *Te*, *rad_prof=None*, *rad_label='rmin [cm]'*, *plot=False*)

Compute density of excited states in state *m* (m>1), given the density of ground state atoms.

This function is not l-resolved.

The function returns

$$N_m/N_1 = ($$

rac{N_m^i}{N_1} ight) N_m + left(rac{N_m^{ii}}{n_i} ight) n_i

where N_m is the number of electrons in the excited state m, N_1 is the number in the ground state, and n_i is the density of ions that could recombine. i and ii indicate terms corresponding to coupling to the ground state and to the continuum, respectively.

Ref.: DEGAS2 manual.

Args:

m [int] Principal quantum number of excited state of interest. 2<m<10

N1 [float, list or 1D-array [cm^-3]] Density of ions in the ground state. This must have the same shape as ni!

ni [float, list or 1D-array [cm^-3]] Density of ions corresponding to the atom under consideration. This must have the same shape as N1!

ne [float, list or 1D-array [cm^-3]] Electron density to evaluate atomic rates at.

Te [float, list or 1D-array [eV]] Electron temperature to evaluate atomic rates at.

Keyword Args:

rad_prof [list, 1D array or None] If None, excited state densities are evaluated at all the combinations of ne,Te and zip(Ni,ni). If a 1D array (same length as ne,Te,ni and N1), then this is taken to be a radial coordinate for radial profiles of ne,Te,ni and N1.

rad_label [str] When rad_prof is not None, this is the label for the radial coordinate.

plot [bool] Display the excited state ratio

Returns:

Nm [array of shape [len(ni)=len(N1),len(ne),len(Te)]] Density of electrons in excited state *n* [cm^-3]

`aurora.neutrals.`**`plot_exc_ratios`**(*n_list=[2, 3, 4, 5, 6, 7, 8, 9]*, *ne=10000000000000.0*, *ni=10000000000000.0*, *Te=50*, *N1=1000000000000.0*, *ax=None*, *ls='-'*, *c='r'*, *label=None*)

Plot N_i/N_1, the ratio of hydrogen neutral density in the excited state *i* and the ground state, for given electron density and temperature.

Parameters

- **`n_list`** – list of integers List of excited states (principal quantum numbers) to consider.

- **ne** – float Electron density in cm^-3.
- **ni** – float Ionized hydrogen density [cm^-3]. This may be set equal to ne for a pure plasma.
- **Te** – float Electron temperature in eV.
- **N1** – float Density of ground state hydrogen [cm^-3]. This is needed because the excited state fractions depend on the balance of excitation from the ground state and coupling to the continuum.

Keyword Arguments

- **ax** – matplotlib.axes instance, optional Axes instance on which results should be plotted.
- **ls** – str Line style to use
- **c** – str or other matplotlib color specification Color to use in plots
- **label** – str Label to use in scatter plot.

Returns

list of arrays List of arrays for each of the n-levels requested, each containing excited state densities at the chosen densities and temperatures for the given ground state density.

Return type Ns

3.8.15 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".

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

Parameters

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

- **Ti** – float, 1D or 2D array Target temperature [keV]. Results will be computed for each Ti value in a vectorized manner.
- **E_beam** – float Beam energy [keV]
- **m_bckg** – float Target atomic mass [amu]
- **m_beam** – float Beam atomic mass [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.

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

Return type rate

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.

Parameters

- **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

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

```
rates[beam][f'n={n_level}']['halo']
```

Rates are on a radial grid corresponding to the input neut_fsa['rhop'].

Return type rates

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.).

Parameters

- **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

```
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

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

```
neutrals_ext[beam][f'n={n_level}'][name_idx]
```

Return type neut_fsa

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.

Parameters

- **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
- **2-dimensional.** (*function*) –

Returns

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

Return type rate

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

3.8.16 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.

Parameters

- **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

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

Return type sigma

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*)

3.8.17 aurora.synth_diags module

aurora.synth_diags.**centrifugal_asym**(*rhop*, *Rlfs*, *omega*, *Zeff*, *A_imp*, *Z_imp*, *Te*, *Ti*, *main_ion_A=2*, *plot=False*, *nz=None*, *geqdsk=None*)

Estimate impurity poloidal asymmetry effects from centrifugal forces.

The result of this function is λ, defined such that

ho) (R(r, heta)^2- R_0^2) ight}

See Odstrcil et al. 2018 Plasma Phys. Control. Fusion 60 014003 for details on centrifugal asymmetries. Also see Appendix A of Angioni et al 2014 Nucl. Fusion 54 083028 for details on these should also be accounted for when comparing transport coefficients used in Aurora (on a rvol grid) to coefficients used in codes that use other coordinate systems (e.g. based on rmid).

Args:

rhop [array (nr,)] Sqrt of normalized poloidal flux grid.

Rlfs [array (nr,)] Major radius on the Low Field Side (LFS), at points corresponding to rhop values

omega [array (nt,nr) or (nr,) [rad/s]] Toroidal rotation on Aurora temporal time_grid and radial rhop_grid (or, equivalently, rvol_grid) grids.

Zeff [array (nt,nr), (nr,) or float] Effective plasma charge on Aurora temporal time_grid and radial rhop_grid (or, equivalently, rvol_grid) grids. Alternatively, users may give Zeff as a float (taken constant over time and space).

A_imp [float] Impurity ion atomic mass number (e.g. 40 for Ca)

Z_imp [array (nr,) or int]

Charge state of the impurity of interest. This can be an array, giving the expected charge state at every radial position, or just a float.

Te [array (nr,nt)] Electron temperature (eV)

Ti [array (nr, nt)] Background ion temperature (eV)

main_ion_A [int, optional] Background ion atomic mass number. Default is 2 for D.

Keyword Args:

plot [bool] If True, plot asymmetry factor λ vs. radius and show the predicted 2D impurity density distribution at the last time point.

nz [array (nr,nZ)] Impurity charge state densities (output of Aurora at a specific time slice), only used for 2D plotting.

geqdsk [dict] Dictionary containing the *omfit_eqdsk* reading of the EFIT g-file.

Returns:

CF_lam [array (nr,)] Asymmetry factor, defined as λ in the expression above.

aurora.synth_diags.**line_int_weights**(*R_path*, *Z_path*, *rhop_path*, *dist_path*, *R_axis=None*, *rhop_out=None*, *CF_lam=None*)

Obtain weights for line integration on a rhop grid, given the 3D path of line integration in the (R,Z,Phi) coordinates, as well as the value of sqrt of normalized poloidal flux at each point along the path.

Parameters

- **R_path** – array (np,) Values of the R coordinate [m] along the line integration path.
- **Z_path** – array (np,) Values of the Z coordinate [m] along the line integration path.
- **rhop_path** – array (np,) Values of the rhop coordinate along the line integration path.
- **dist_path** – array (np,) Vector starting from 0 to the maximum distance [m] considered along the line integration.

Keyword Arguments

- **R_axis** – float R value at the magnetic axis [m]. Only used for centrifugal asymmetry effects if CF_lam is not None.
- **rhop_out** – array (nr,) The sqrt of normalized poloidal flux grid on which weights should be computed. If left to None, an equally-spaced grid with 201 points from the magnetic axis to the LCFS is used.
- **CF_lam** – array (nr,) Centrifugal (CF) asymmetry exponential factor, returned by the *centrifugal_asym()* function. If provided, this is taken to be on an input rhop_out grid. If left to None, no CF asymmetry is considered.

3.8.18 Module contents

aurora

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

a

INDEX

G

I

J

L

M

N

P

R

S

T

U

V

W

X