Aurora Documentation

Release 2.1.2

Francesco Sciortino

Aug 22, 2023

CONTENTS

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

Paper/presentation in Plasma Physics & Fusion Energy and on the arXiv.

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. The package includes a public release of the Fast and Accurate Collisional Impurity Transport (FACIT) model for the calculation of neoclassical diffusion and convection coefficients in tokamak plasmas. 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.

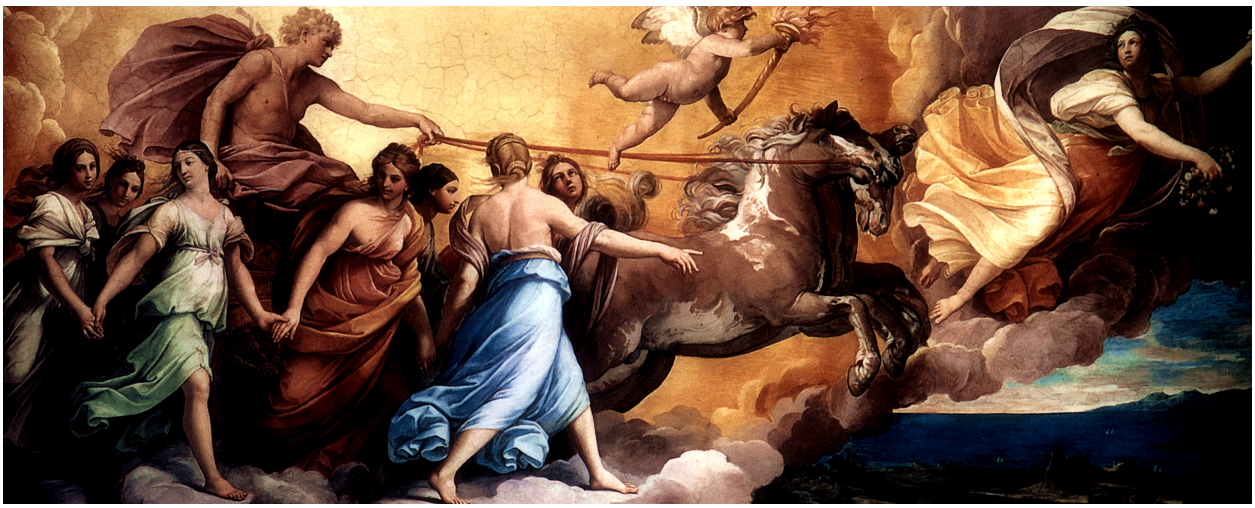

Fig. 1: Aurora fresco, by Guido Reni (circa 1612-1614)

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, impurities, 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*.

Aurora also includes a Python version of the FACIT code, described in the Tutorials section of this documentation, which allows users to rapidly estimate neoclassical impurity transport coefficients. This capability is particularly useful in integrated transport modeling, as well as in experimental inference of impurity transport coefficients.

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
```

which 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

The latest release is now available via PyPI using:

```
pip install aurorafusion
```

Installing via conda is also possible using

conda install -c conda-forge 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 Core impurity transport 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. Please refer to docstrings throughout the code documentation.

Aurora leverages the *omfit_classes* package to interface with MDS+, EFIT, and a number of other codes. Thanks to this, users can focus more on their applications of interest, and less on re-inventing the wheel to write code that the OMFIT Team has kindly made public! To follow the rest of this tutorial, do:

```
from omfit_classes import omfit_eqdsk, omfit_gapy
```

Next, we 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_classes* 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['source_rate'] = 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 `aurora.core.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 `aurora.core.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 `aurora.core.run_aurora()` docstring for details.

A plot is worth a thousand words, so let's make one for the charge state densities::

```
nz = out[0]  # charge state densities are stored first in the output of the run_
↪aurora method
aurora.slider_plot(asim.rvol_grid, asim.time_out, nz.t.transpose(1,0,2),
                   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 )
```

You should get a slider showing a result like the following:

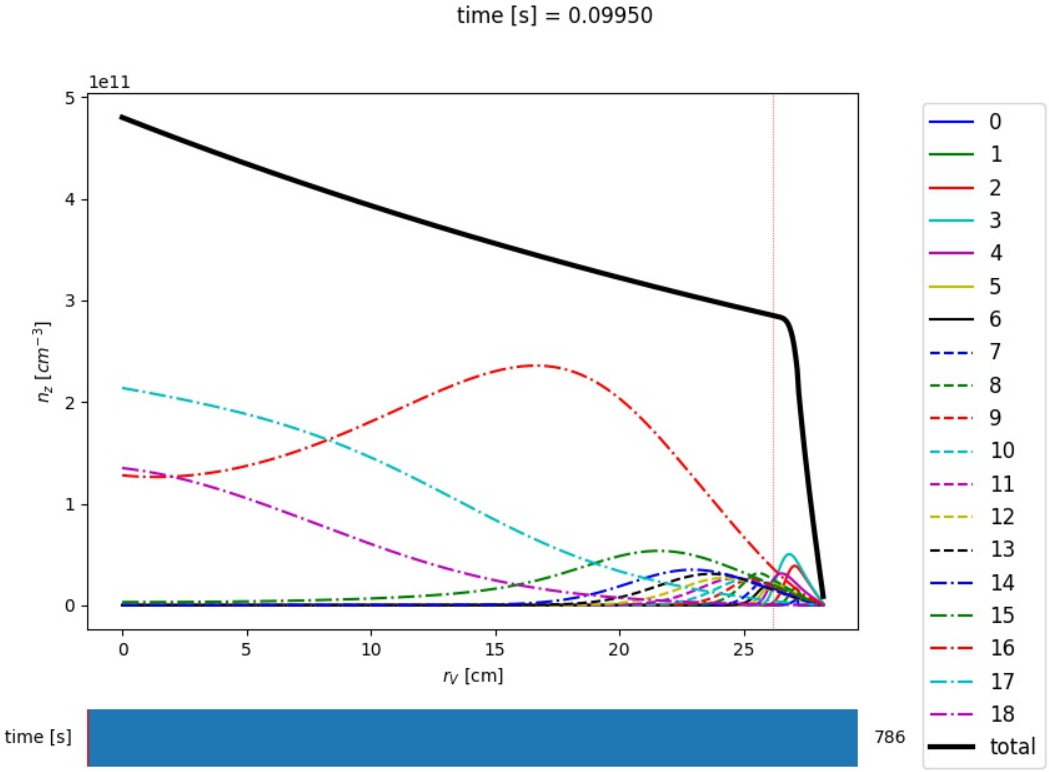

Fig. 1: Example of charge state density profiles at the end of an Aurora Ar simulation

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)
```

This will give you a slider again, showing figures like this:

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

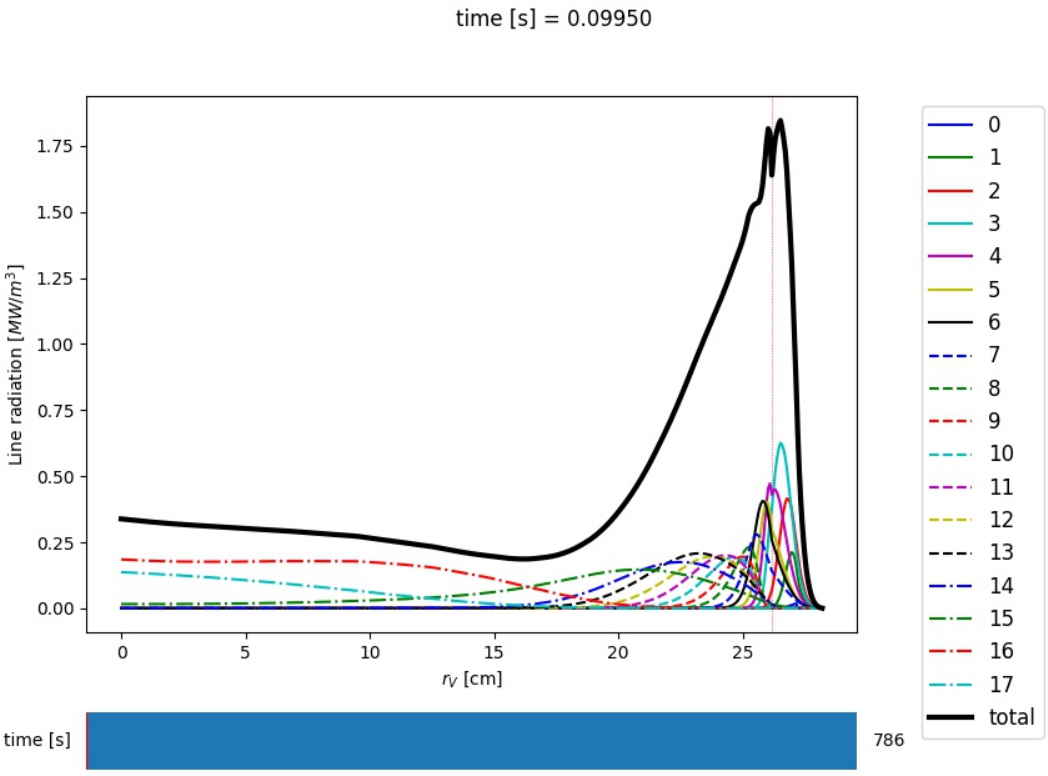

Fig. 2: Example of line radiation at the end of an Aurora Ar simulation

MW). To obtain the latter form, we need to integrate over flux surface volumes. To do so, we make use of the *geqdsk* dictionary obtained via:

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

We then pass that 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, geqdsk,
                             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 to 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, geqdsk,
                             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 states, radii).

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)
```

You should get something that looks like this:

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:

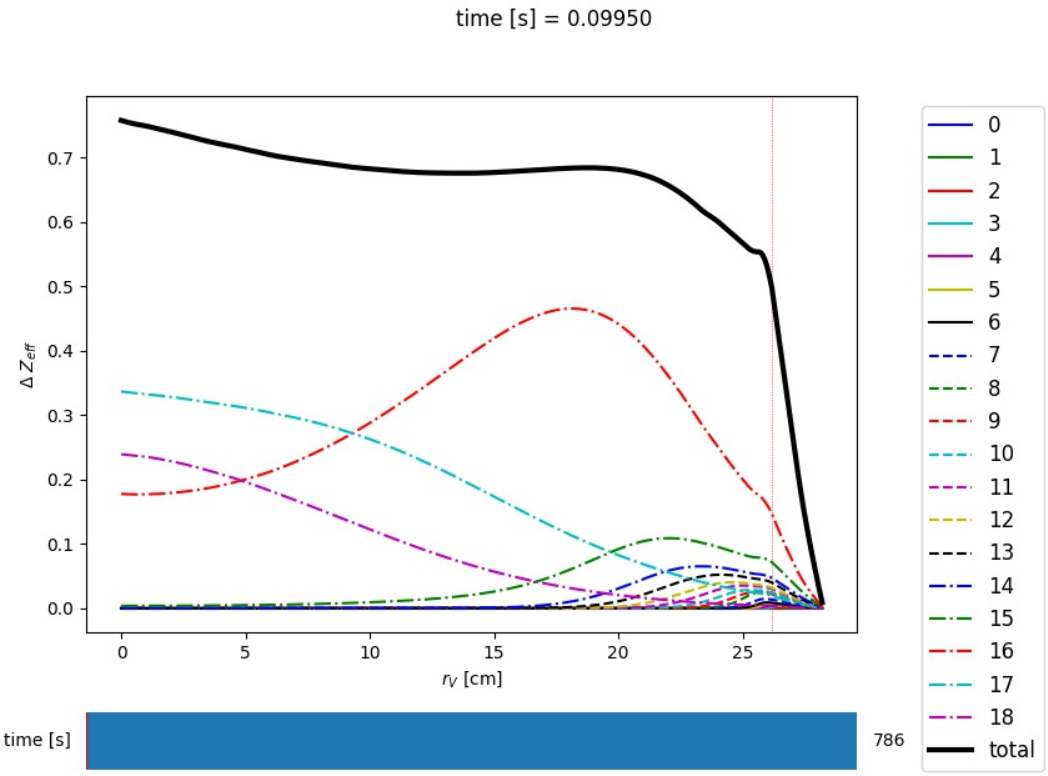

Fig. 3: Example of Z-effective contributions at the end of an Aurora Ar simulation

```
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:

```
Te, fz = 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*.

Additionally, the function *get_atomic_relax_time()* allows one to obtain the relaxation time of the ionization equilibrium for a given species, which permits an assessment of the time scales for atomic vs transport effects. The effect of transport can be mimicked by passing a *tau_s* parameter, representing a typical (global) particle residence time. This capability can be used, for example, to explore how sensitive the ionization balance is to transport, but it can of course only provide a very approximate assessment.

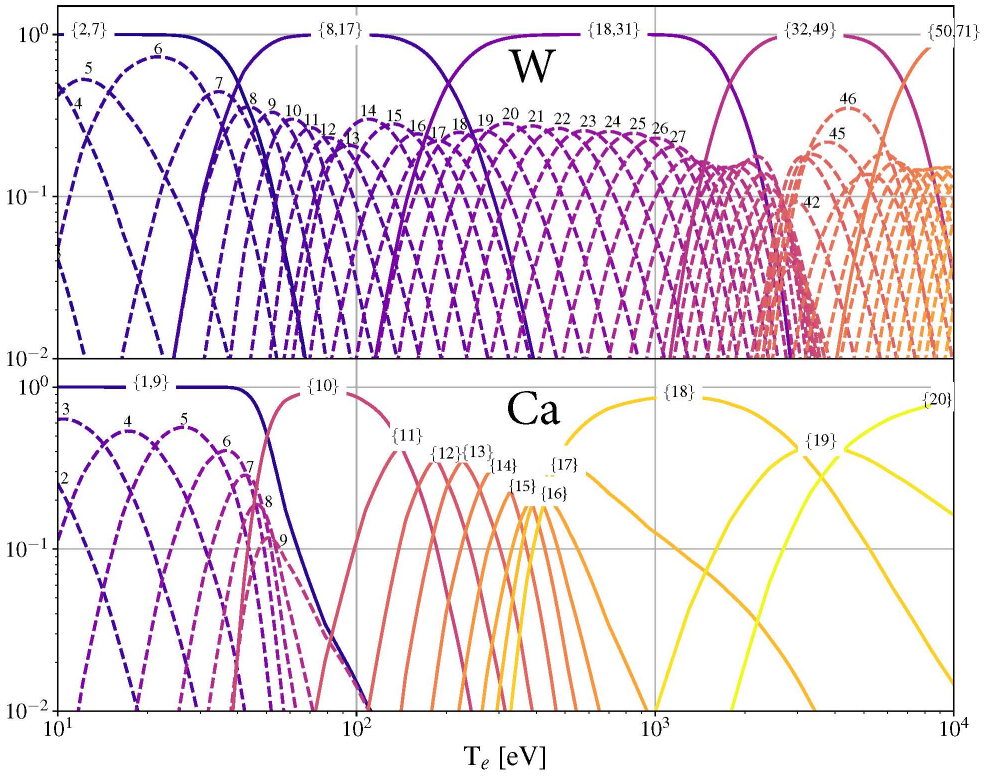

Fig. 4: Ionization equilibria of W and Ca (dashed lines), also showing some choices of charge state bundling (superstaging) for both species.

The figure above shows examples of ionization equilibria for W and Ca as a function of electron temperature. Dashed lines here show the complete/standard result, whereas the continuous lines show examples of charge

state bundling (superstaging), using arbitrarily-chosen partitions. Superstaging is an extremely useful and interesting technique to reduce the computational complexity of medium- and high-Z ions, since often the cost of simulations scales linearly (as in Aurora), or worse, with the number of charge states (Z). You can read more about superstaging in the paper F Sciortino et al 2021 Plasma Phys. Control. Fusion 63 112001.

3.2.5 Atomic spectra

If you have atomic data files containing photon emissivity coefficients (PECs) in ADF15 format, you can use Aurora to combine them and see what the overall spectrum might look like. Let's say you want to look at the K_α spectrum of Ca at a specific electron density of 10^{14} cm^{-3} and temperature of 1 keV. Let's begin with a single ADF15 file located at the following path:

```
filepath_he='~/pec#ca18.dat'
```

The simplest way to check what the spectrum may look like is to weigh contributions from different charge states according to their fractional abundances at ionization equilibrium. Aurora allows you to get the fractional abundances with just a couple of lines:

```
ion = 'Ca'
ne_cm3 = 1e14
Te_eV = 1e3
atom_data = aurora.get_atom_data(ion,['scd','acd'])
Te, fz = aurora.get_frac_abundances(atom_data, np.array([ne_cm3,]), np.array([Te_
↪eV,]), plot=False)
```

You can now use the *aurora.get_local_spectrum* function to read all the lines in each ADF15 file and broaden them according to some ion temperature (which could be dictated by broadening mechanisms other than Doppler effects, in principle). For our example, one can do:

```
# He-like state
out= aurora.get_local_spectrum(filepath_he, ion, ne_cm3, Te_eV, n0_cm3=0.0,
                               ion_exc_rec_dens=[fz[0,-4], fz[0,-3], fz[0,-2]], #␣
↪Li-like, He-like, H-like
                               dlam_A = 0.0, plot_spec_tot=False, no_leg=True,
                               plot_all_lines=True, ax=None)
wave_final_he, spec_ion_he, spec_exc_he, spec_rec_he, spec_dr_he, spec_cx_he, ax␣
↪= out
```

By changing the *dlam_A* parameter, you can also add a wavelength shift (e.g. from the Doppler effect). The *ion_exc_rec_dens* parameter allows specification of fractional abundances for the charge stages of interest. To be quite general, in the lines above we have included contributions to the spectrum from ionizing, excited and recombining PEC components. By passing an *ax* argument one can also specify which matplotlib axes are used for plotting.

By repeating the same operations using several ADF15 files, one can overplot contributions to the spectrum from several charge states. As an example, the figure below shows the K-alpha spectrum of Ca, with contributions from Li-like, He-like and H-like Ca, evaluated at 1 keV and 10^{20} cm^{-3}.

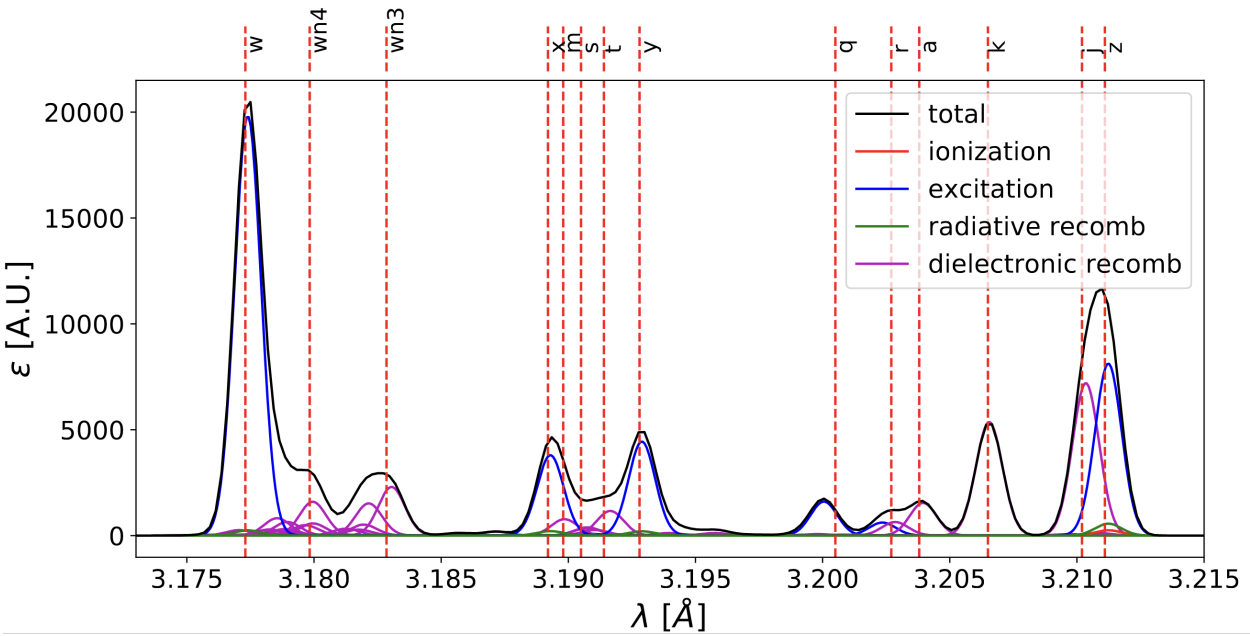

Fig. 5: Example of high-resolution K-alpha spectrum of Ca

If you just want to plot where atomic lines are expected to be and how intense their PECs are at specific plasma conditions, you can also use the simpler *aurora.adf15_line_identification* function. This can be called as:

```
aurora.adf15_line_identification(pec_files, Te_eV=Te_eV, ne_cm3=ne_cm3,␣
↪mult=mult)
```

and can be used to plot something like this:

3.2.6 A&M data from AMJUEL/HYDHEL

Aurora allows one to load and process atomic and molecular (A&M) rates collected in the form of polynomial fits in the AMJUEL and HYDHEL databases, publicly released as part of the EIRENE code. These rates are partly redundant and partly complement those provided by ADAS.

The following code illustrates how one may evaluate A&M components of the Balmer lines of deuterium:

```
import os, copy
import numpy as np
import matplotlib.pylab as plt

plt.ion()
import sys

# Make sure that package home is added to sys.path
sys.path.append("../")
import aurora
```

(continues on next page)

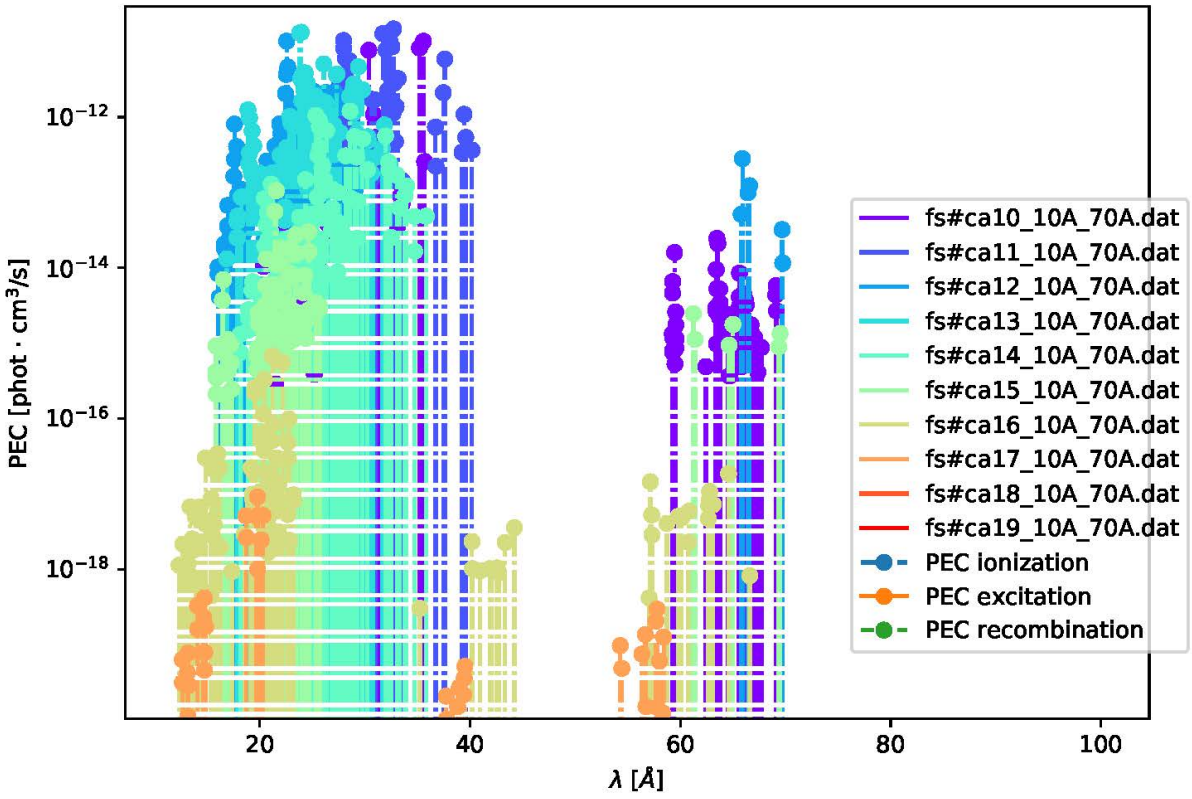

Fig. 6: Example of Ca spectrum overview combining several PEC files

(continued from previous page)

```
rdb = aurora.amdata.reactions_database()


RR_alpha = [
    "AMJUEL,12,2_1_5a",  # H(3)/H
    "AMJUEL,12,2_1_8a",  # H(3)/H+
    "AMJUEL,12,2_2_5a",  # H(3)/H2
    "AMJUEL,12,2_2_14a",  # H(3)/H2+
    "AMJUEL,12,2_0c",  # H2+/H2
    "AMJUEL,12,7_2a",  # H(3)/H-
    "AMJUEL,11,7_0a",  # H-/H2
    "AMJUEL,12,2_2_15a",  # H(3)/Hll3+    # not used
    "AMJUEL,11,4_0a",  # H3+/H2/H2+/ne  # not used
]

# Lyman series spontaneous emission coeffs for n=2 to 1, 3 to 1, ... 16 to 1
A_lyman = [
    4.699e8,
    5.575e7,
    1.278e7,
    4.125e6,
    1.644e6,
    7.568e5,
    3.869e5,
    2.143e5,
    1.263e5,
    7.834e4,
    5.066e4,
    3.393e4,
    2.341e4,
    1.657e4,
    1.200e4,
]

# Balmer series spontaneous emission coeffs for n=3 to 2, 4 to 2, ... 17 to 2
A_balmer = [
    4.41e7,
    8.42e6,
    2.53e6,
    9.732e5,
    4.389e5,
    2.215e5,
    1.216e5,
    7.122e4,
    4.397e4,
```

(continues on next page)

(continued from previous page)

```
    2.83e4,
    18288.8,
    12249.1,
    8451.26,
    5981.95,
    4332.13,
]

fig, ax = plt.subplots(2, 1, sharex=True)
ls_list = ["-", "--", "-.", ":"]

nt = 200
lines = ["alpha", "beta", "gamma", "delta"]
rates = np.zeros((len(RR_alpha), nt, len(lines)))

for ii, choice in enumerate(lines):

    ls = ls_list[ii]

    # allow one to read various Balmer-series lines by changing final reaction␣
↪letter
    sub = {"alpha": "a", "beta": "c", "gamma": "d", "delta": "e"}
    RR = copy.deepcopy(RR_alpha)
    for i in [0, 1, 2, 3, 5, 7]:  # only cross sections specific to Halpha
        RR[i] = "AMJUEL" + RR_alpha[i].split("AMJUEL")[1].replace("a",␣
↪sub[choice])

    # select Einstein Aki coefficients
    aa = {"alpha": 0, "beta": 1, "gamma": 2, "delta": 3}
    a0 = A_balmer[aa[choice]]

    te = np.linspace(0.2, 10.0, nt)
    ne = np.ones(nt) * 1e19
    ni = np.ones(nt) * 1e19
    nh = np.ones(nt) * 1e19
    nh2 = np.ones(nt) * 1e19

    for j, R in enumerate(RR):
        rdb.select_reaction(R)
        rates[j, :, ii] = rdb.reaction(ne, te)

    c1 = a0 * rates[0, :, ii] * nh
    c2 = a0 * rates[1, :, ii] * ni
    c3 = a0 * rates[2, :, ii] * nh2
    c4 = a0 * rates[3, :, ii] * rates[4, :, ii] * nh2
    c5 = a0 * rates[5, :, ii] * rates[6, :, ii] * nh2
```

(continues on next page)

(continued from previous page)

```
    ct = c1 + c2 + c3 + c4 + c5  # np.ones_like(c1)

    ax[0].plot(te, ct, c="k", ls=ls)
    ax[1].plot(te, c1 / ct, c="b", ls=ls)
    ax[1].plot(te, c2 / ct, c="c", ls=ls)
    ax[1].plot(te, c3 / ct, c="g", ls=ls)
    ax[1].plot(te, c4 / ct, c="r", ls=ls)
    ax[1].plot(te, c5 / ct, c="m", ls=ls)

ax[0].plot([], [], c="k", ls="-", label="alpha")
ax[0].plot([], [], c="k", ls="--", label="beta")
ax[0].plot([], [], c="k", ls="-.", label="gamma")
ax[0].plot([], [], c="k", ls=":", label="delta")
ax[0].legend(loc="best").set_draggable(True)
ax[0].set_yscale("log")
ax[0].set_ylabel("$\epsilon_{tot}$ [ph/m$^{-3}$]")

ax[1].plot([], [], c="b", label="H")
ax[1].plot([], [], c="c", label="H+")
ax[1].plot([], [], c="g", label="H2")
ax[1].plot([], [], c="r", label="H2+")
ax[1].plot([], [], c="m", label="H2-")
ax[1].legend(loc="best").set_draggable(True)
ax[1].set_yscale("log")
ax[1].set_ylim((1e-3, 1e0))
ax[1].set_xlabel("$T_e$ [eV]")
ax[1].set_ylabel("$\epsilon_i/\epsilon_{tot}$")
```

3.2.7 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')

# load all the transitions in the chosen ADF15 file -- returns a pandas.DataFrame
trs = aurora.read_adf15(path)

# select and plot the Lyman-alpha line at 1215.2 A
tr = trs.loc[(trs['lambda [A]']==1215.2) & (trs['type']=='excit')]
aurora.plot_pec(tr)
```

This will plot the Lyman-alpha photon emissivity coefficients (both the components due to excitation and recombination) as a function of temperature in eV, as shown in the figures below.

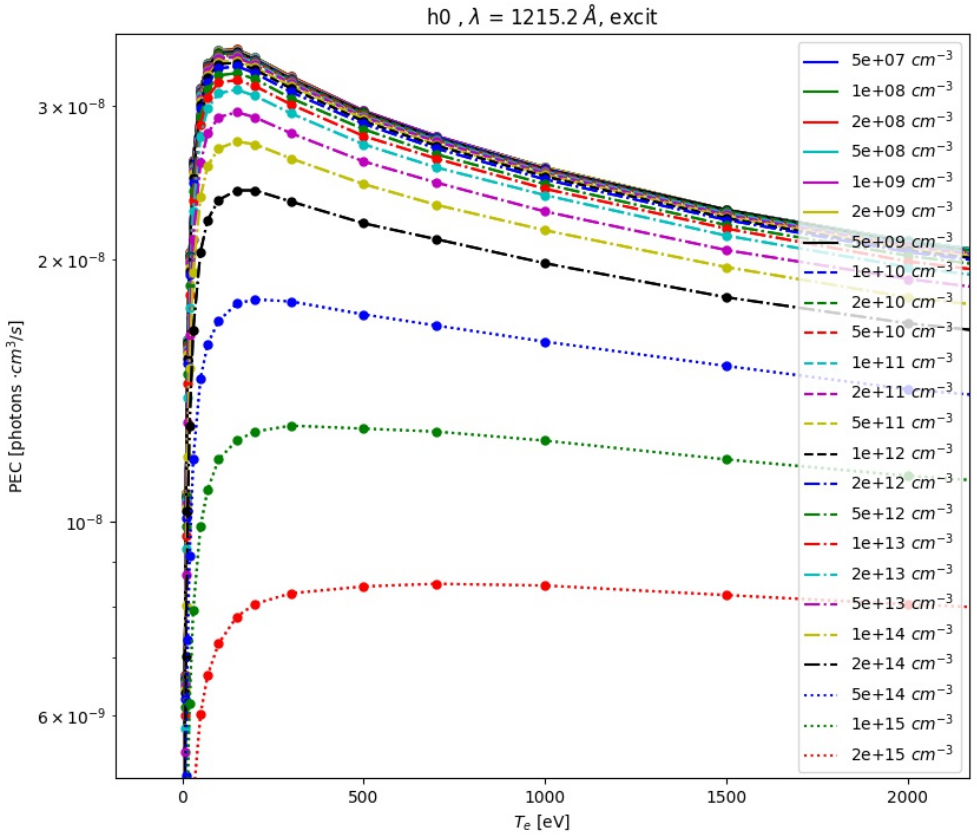

Fig. 7: ADAS photon emissivity coefficients for the excitation contribution to the H Ly_α transition.

Some files (e.g. try *pec96#c_pju#c2.dat*) may also have charge exchange components. Note that both the inputs and outputs of the `read_adf15()` function act on log-10 values, i.e. interpolants should be called on log-10 values of n_e and T_e, and the result of interpolation will only be in units of $photons \cdot cm^3/s$ after one

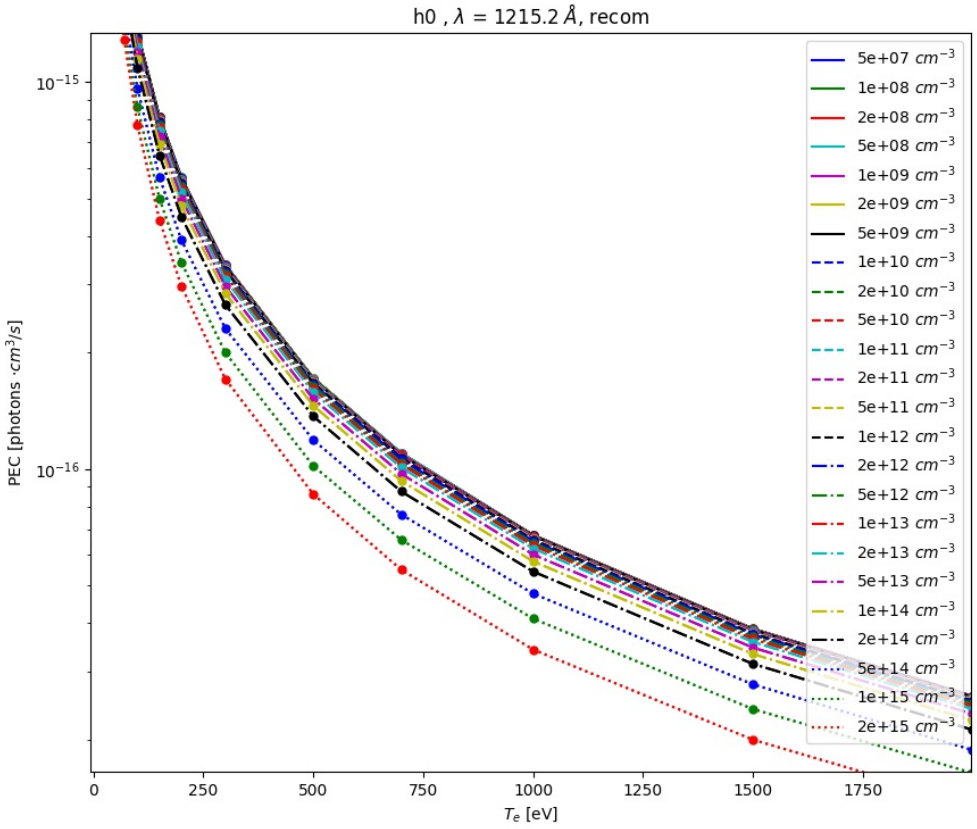

Fig. 8: ADAS photon emissivity coefficients for the recombination contribution to the H Ly_α transition.

takes the power of 10 of it.

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).

3.2.8 Interfacing with SOLPS-ITER

While running SOLPS-ITER is a complex task, reading and processing its results does't need to be. Aurora offers a convenient Python interface to rapidly load results, set them to convenient data arrays, plot on 1D or 2D grids, etc.

Here's an example of how you could load a SOLPS-ITER run and create some useful plots:

```
"""
Script to demonstrate capabilities to load and post-process SOLPS results.
Note that SOLPS output is not distributed with Aurora; so, in order to run this␣
↪script,
either you are able to access the default AUG SOLPS MDS+ server, or you need
to appropriately modify the script to point to your own SOLPS results.
"""
import numpy as np
import matplotlib.pyplot as plt

plt.ion()
from omfit_classes import omfit_eqdsk
import sys, os
import aurora

# if one wants to load a SOLPS case from MDS+ (defaults for AUG):
so = aurora.solps_case(solps_id=141349)

# alternatively, one may want to load SOLPS results from files on disk:
so2 = aurora.solps_case(
    b2fstate_path="/afs/ipp/home/s/sciof/SOLPS/141349/b2fstate",
```

(continues on next page)

(continued from previous page)

```
    b2fgmtry_path="/afs/ipp/home/s/sciof/SOLPS/141349/b2fgmtry",
)

# plot some important fields
fig, axs = plt.subplots(1, 2, figsize=(10, 6), sharex=True)
ax = axs.flatten()
so.plot2d_b2(so.data("ne"), ax=ax[0], scale="log", label=r"$n_e$ [$m^{-3}$]")
so.plot2d_b2(so.data("te"), ax=ax[1], scale="linear", label=r"$T_e$ [eV]")


if hasattr(so, "fort46") and hasattr(so, "fort44"):
    # if EIRENE data files (e.g. fort.44, .46, etc.) are available,
    # one can plot EIRENE results on the original EIRENE grid.
    # SOLPS results also include EIRENE outputs on B2 grid
    fig, axs = plt.subplots(1, 2, figsize=(10, 6), sharex=True)
    so.plot2d_eirene(
        so.fort46["pdena"][:, 0] * 1e6,
        scale="log",
        label=r"$n_n$ [$m^{-3}$]",
        ax=axs[0],
    )
    so.plot2d_b2(so.fort44["dab2"][:, :, 0].T, label=r"$n_n$ [$m^{-3}$]",␣
↪ax=axs[1])
    plt.tight_layout()
```

In this example, we are first loading a SOLPS-ITER case from MDS+, using the default server and tree which are set for Asdex-Upgrade. These MDS+ settings can be easily changed by looking at the docstring for *solps_case*. The alternative of loading SOLPS output from files on disk (*b2fstate* and *b2fgmry*, in this case) is also shown.

The instantiation of a *solps_case* object enables a large number of operations based on the SOLPS output. The second part of the script above shows how one can plot 2D data on the B2 grid. The last section demonstrates how the EIRENE output can be displayed both on the B2 grid, on which it is interpolated by SOLPS, or on the native EIRENE grid.

Warning: EIRENE results can only be displayed on the EIRENE mesh if EIRENE (*fort.**) output files are provided. At present, SOLPS output saved to MDS+ trees only provides EIRENE results interpolated on the B2 mesh.

In the original Plasma Physics & Fusion Energy paper on Aurora, an example of processing SOLPS-ITER output for the ITER baseline scenario was described. Some example figures produced via the methods shown above are displayed here below.

Note: Note that no SOLPS results are not distributed with Aurora. You must have the output of a SOLPS-

ITER run available to you in order to try out these Aurora capabilities.

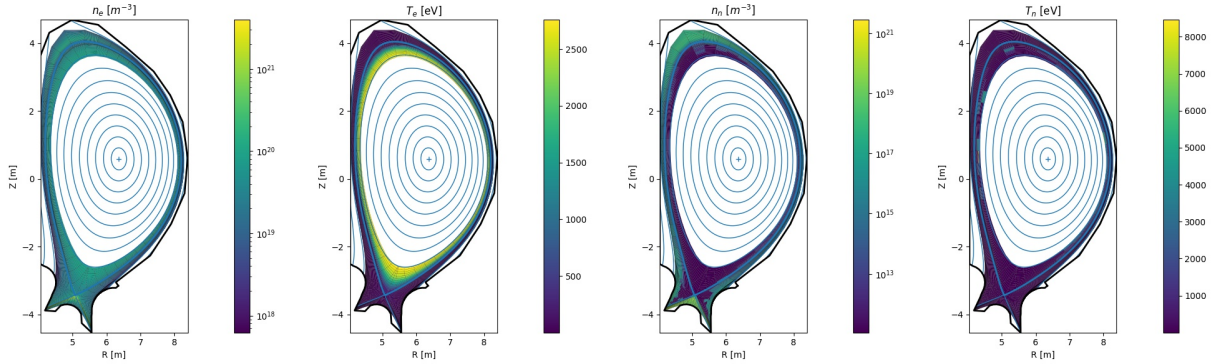

Fig. 9: Example of electron density and temperature + atomic D/T neutral density and temperature from a SOLPS-ITER simulation of ITER

Aurora capabilities to post-process SOLPS results can be useful, for example, to assess synthetic diagnostics for the edge of a fusion device. For this purpose, the *eval_LOS()* method can help to extract a specific data field from the loaded SOLPS case, interpolating resuls along a line-of-sight (LOS) that goes between two spatial (3D) points. This, combined with Aurora's capability to examine and simulate atomic spectra, reduces the technical barrier to investigate edge physics.

3.2.9 Interfacing with OEDGE

OEDGE is a simulation package developed around DIVIMP, a Monte Carlo impurity transport code for the plasma edge. OEDGE allows one to load plasma backgrounds from either EDGE2D-EIRENE or B2-EIRENE, or to create approximate backgrounds based on Onion-Skin Modeling (OSM) coupled with EIRENE. Use of the OSM-EIRENE scheme is itself an interesting application of OEDGE.

Aurora includes tools to read OEDGE input files and read/postprocess its results. The following code illustrates how one may run an OEDGE simulation, load and postprocess its results in order to evaluate Balmer line A&M components, using the AMJUEL rates from the EIRENE package:

```
import aurora
import matplotlib.pyplot as plt

plt.ion()

shot = 38996
time_s = 3.5
label = "osm_test"
casename = f"oedge_AUG_{shot}_{int(time_s*1e3)}_{label}"

# set up an OEDGE case
osm = aurora.oedge_case(shot, (t0 + t1) / 2.0, label=label)
```

(continues on next page)

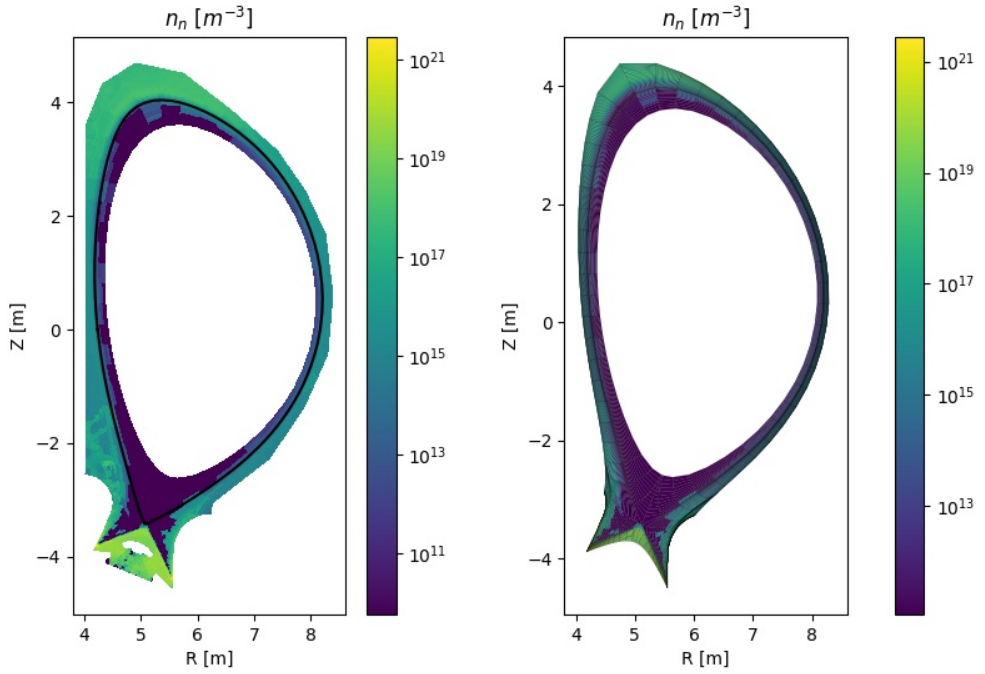

Fig. 10: Comparison of D/T atomic neutral density on the B2 and EIRENE grids.

(continued from previous page)

```
# load a specific input file
osm.load_input_file(filepath="/path/to/my/file.d6i")

# possibly modify specific parts of the input file:
osm.inputs["+P01"]["data"] = 22

# update input file
osm.write_input_file(filepath="/path/to/my/file.d6i")

# now run simulation
osm.run(grid_loc="/path/to/my/grid/file.sno")

# once simulation is run, load the output
osm.load_output()

# now, initialize emission calculation
am = aurora.h_am_pecs()

# extract relevant fields from the OEDGE run
ne_m3 = osm.output.read_data_2d("KNBS")
Te_eV = osm.output.read_data_2d("KTEBS")
```

(continues on next page)

(continued from previous page)

```
Ti_eV = osm.output.read_data_2d("KTIBS")
n_H2_m3 = osm.output.read_data_2d("PINMOL")
n_H_m3 = osm.output.read_data_2d("PINATO")

# load all contributions predicted by AMJUEL
cH, cHp, cH2, cH2p, cH2m = am.load_pec(
    ne_m3,
    Te_eV,
    ne_m3,  # ni=ne
    n_H_m3,
    n_H2_m3,
    series="balmer",
    choice="alpha",
    plot=False,
)

fig, axs = plt.subplots(1, 5, figsize=(25, 6), sharex=True)
osm.output.plot_2d(cH, ax=axs[0])
axs[0].set_title("cH")
osm.output.plot_2d(cHp, ax=axs[1])
axs[1].set_title("cHp")
osm.output.plot_2d(cH2, ax=axs[2])
axs[2].set_title("cH2")
osm.output.plot_2d(cH2p, ax=axs[3])
axs[3].set_title("cH2p")
osm.output.plot_2d(cH2m, ax=axs[4])
axs[4].set_title("cH2m")
plt.tight_layout()
```

3.2.10 Neoclassical transport with FACIT

The FACIT model can be used in Aurora to calculate charge-dependent collisional transport coefficients analytically for the impurity species of interest. FACIT takes kinetic profiles and some magnetic geometry quantities (which shall be described shortly) as inputs, and outputs the collisional diffusion coefficient D_z $[m^2/s]$ and convective velocity V_z $[m/s]$ which can then be given as an input for *run_aurora()*.

An example of a standalone call to FACIT is provided in *aurora/facit.py*, from which profiles of the transport coefficients are obtained:

A complete description of the inputs and outputs of the model is provided in the documentation of the `FACIT` class.

Note that FACIT provides the individual Pfirsch-Schüter, Banana-Plateau and classical collisional flux components, facilitating additional analysis of the physical processes involved in the transport.

An important feature of FACIT is the description of the effects of rotation on neoclassical transport across collisionality regimes, particularly relevant when heavy impurities like tungsten are analyzed. Rotation is

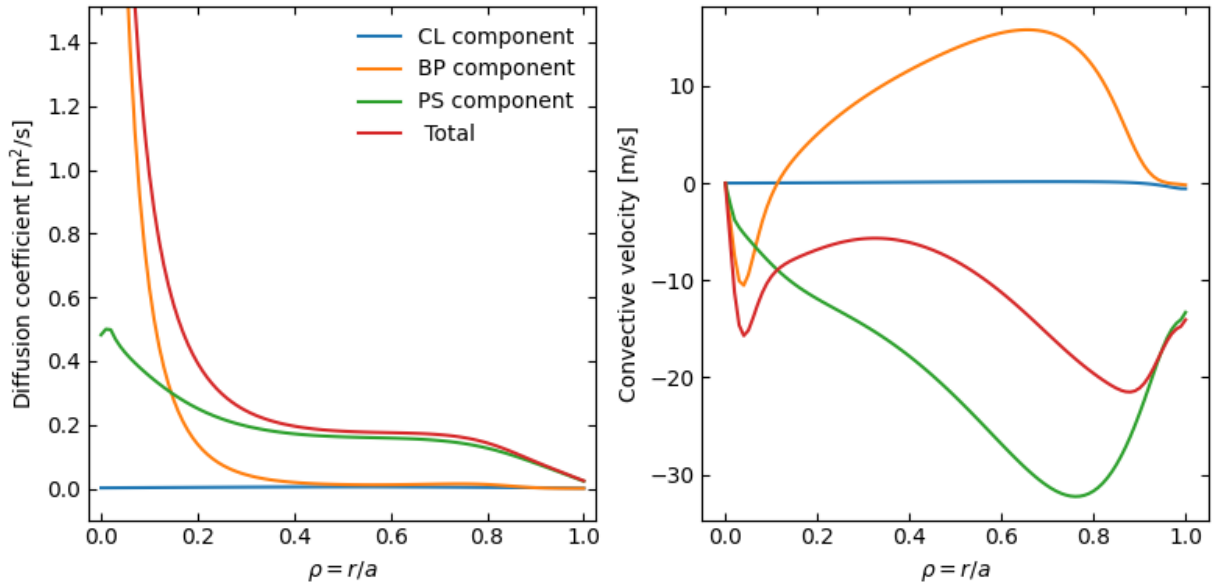

Fig. 11: Example of neoclassical W transport coefficients calculated with FACIT.

described by the main ion Mach number $M_i(r) = v_\varphi/v_{ti} = \Omega R_0/\sqrt{2T_i/m_i}$ and the *rotation_model* flag. These effects can typically be ignored for light impurities (from experience, the impact of rotation on Argon is small but potentially non-negligible).

Warning: If *rotation_model=2*, then the flux surface contours $R(r,\theta)$, $Z(r,\theta)$ that are inputs of FACIT should have a radial discretization equal to the *rho* coordinate in which FACIT will be evaluated. If they are not given as inputs, circular geometry will be assumed internally.

A full example on how to run FACIT in Aurora is given *examples/facit_basic.py*. The following code shows the initialization of the transport coefficients, the magnetic geometry and the main call to FACIT looped over each charge state of the impurity species::

```
# note that to be able to give charge-dependent Dz and Vz,
# one has to give also time dependence (see documentation of aurora.core.run_
↪aurora())

times_DV = np.array([0])
# note that to be able to give charge-dependent Dz and Vz,
# one has to give also time dependence (see documentation of aurora.core.run_
↪aurora()),
# so a dummy time dimension with only one element is introduced here
nz_init  = np.zeros((asim.rvol_grid.size, asim.Z_imp+1))

D_z = np.zeros((asim.rvol_grid.size, times_DV.size, asim.Z_imp+1)) # space, time,
↪ nZ
```

(continues on next page)

(continued from previous page)

```
V_z = np.zeros(D_z.shape)

# flux surface contours (when rotation_model = 2)
RV, ZV = aurora.rhoTheta2RZ(geqdsk, rhop, theta, coord_in='rhop', n_line=201)
RV, ZV = RV.T, ZV.T

# call to FACIT for each charge state
for j, tj in enumerate(times_DV):

    for i, zi in enumerate(range(asim.Z_imp + 1)):

        if zi != 0:
            Nz     = nz_init[:idxsep+1,i]*1e6 # in 1/m**3
            gradNz = np.gradient(Nz, roa*amin)

            fct = aurora.FACIT(roa,
                               zi, asim.A_imp,
                               asim.main_ion_Z, asim.main_ion_A,
                               Ti, Ni, Nz, Machi, Zeff,
                               gradTi, gradNi, gradNz,
                               amin/R0, B0, R0, qmag,
                               rotation_model = rotation_model, Te_Ti = TeovTi,
                               RV = RV, ZV = ZV)

            D_z[:idxsep+1,j,i] = fct.Dz*100**2 # convert to cm**2/s
            V_z[:idxsep+1,j,i] = fct.Vconv*100 # convert to cm/s
```

Warning: FACIT uses lengths in $[m]$, NOT $[cm]$. In particular, input densities should be given in $[m^{-3}]$ and the output transport coefficients *Dz* and *Vconv* should be converted back from $[m^2/s]$ and $[m/s]$ to $[cm^2/s]$ and $[cm/s]$, respectively, before passing them to *run_aurora()*.

In addition to the collisional transport coefficients calculated with FACIT, we can add a turbulent component to the total diffusion and convection, imposed by hand as in previous sections of this tutorial. In this example, we fix an approximate position for a pedestal top at $\rho_{pol} \approx 0.9$, and assume that turbulence is suppresed inside the pedestal::

```
Dz_an = np.zeros(D_z.shape) # space, time, nZ
Vz_an = np.zeros(D_z.shape)

# estimate pedestal top position and find radial index of separatrix
rped   = 0.90
idxped = np.argmin(np.abs(rped - asim.rhop_grid))
idxsep = np.argmin(np.abs(1.0 - asim.rhop_grid))
```

(continues on next page)

(continued from previous page)

```
# core turbulent transport coefficients
Dz_an[:idxped,:,:] = 1e4  # cm^2/s
Vz_an[:idxped,:,:] = -1e2 # cm/s

# SOL transport coefficients
Dz_an[idxsep:,:,:] = 1e4  # cm^2
Vz_an[idxsep:,:,:] = -1e2 # cm/s

D_z += Dz_an
V_z += Vz_an
```

Note: FACIT is distributed within Aurora. The following papers contain the derivation of the model: the self-consistent calculation of the Pfirsch-Schlüter flux and the poloidal asymmetries of heavy impurities at high collisionality is given in Maget et al 2022 Plasma Phys. Control. Fusion 64 069501, while the extension to arbitrary collisionality and inclusion of the Banana-Plateau flux in the poloidally-symmetric (non-rotating) limit is obtained in Fajardo et al 2022 Plasma Phys. Control. Fusion 64 055017, and finally the description of the effects of rotation across collisionality regimes is presented in Fajardo et al 2023 Plasma Phys. Control. Fusion 65 035021. Please cite these papers when using FACIT.

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_classes

The latter is part of the OMFIT distribution and will provide lots of capabilities to interact with tokamak modeling tools, with which Aurora can be easily integrated (indeed, Aurora is automatically installed as part of any OMFIT installation).

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 Namelist for ion transport simulations

The table below describes the main input parameters to Aurora's forward model of ion transport. Refer to the following sections for details on spatio-temporal grids, recycling and kinetic profiles specifications.

Parameter	Default	Description
imp	Ca	Atomic symbol of the simulated ion.
main_element	D	Background ion species, usually hydrogen isotopes.
source_rate	1e+21	Flux [particles/s] of simulated ions.
source_type	const	Type of ion source, one of ['file','const','step','synth_LBO'], see `get_source_time_history()`.
explicit_source_vals	*None*	2D array for sources on *explicit_source_time* and *explicit_source_rhop* grids
explicit_source_time	*None*	Time grid for explicit source
explicit_source_rhop	*None*	ρ_p grid for explicit source
source_width_in	0.0	Inner Gaussian source width, only used if >0. See `get_radial_source()`.
source_width_out	0.0	Outer Gaussian source width, only used if >0. See `get_radial_source()`.
imp_source_energy_eV	0	Energy of neutral ion source, only used if *source_width_in=source_width_out=0*, see `get_radial_source()`.
prompt_redep_flag	False	If True, a simple prompt redeposition model is activated, see `get_radial_source()`.
source_file	None	Location of source file, using STRAHL format, only used if *source_type="file"*, see `get_source_time_history()`.
source_cm_out_lcfs	1.0	Source distance in cm from LCFS
LBO["n_particles"]	1e+18	Number of particles in LBO synthetic source, only used if *source_type="synth_LBO"*
LBO["t_fall"]	0.3	Decay time of LBO synthetic source, only used if *source_type="synth_LBO"*
LBO["t_rise"]	0.05	Rise time of LBO synthetic source, only used if *source_type="synth_LBO"*
LBO["t_start"]	0.0	Start time of LBO synthetic source, only used if *source_type="synth_LBO"*
timing["dt_increase"]	[1.005 1.]	*dt* multipliers at every time step change. See detailed description.

continues on next page

Table 1 – continued from previous page

Parameter	Default	Description
timing["dt_start"]	[1.e-05 1.e-03]	*dt* values at the beginning of each interval/cycle. See detailed description.
timing["steps_per_cycle"]	[1 1]	Number of steps before *dt* is multiplied by a *dt_increase* value. See detailed description.
timing["times"]	[0. 0.1]	Times at which intervals/cycles change.
bound_sep	2.0	Distance between wall boundary and plasma separatrix [cm].
lim_sep	1.0	Distance between nearest limiter and plasma separatrix [cm].
clen_divertor	17.0	Connection length to the divertor [cm].
clen_limiter	0.5	Connection length to the nearest limiter [cm]
dr_0	0.3	Radial grid spacing on axis. See detailed description.
dr_1	0.05	Radial grid spacing near the wall. See detailed description.
K	6.0	Exponential grid resolution factor. See detailed description.
SOL_decay	0.05	Decay length at the wall bounday, numerical parameter for the last grid point.
saw_model["saw_flag"]	False	If True, activate sawtooth phenomenological model.
saw_model["rmix"]	1000.0	Mixing radius of sawtooth model. Each charge state density is flattened inside of this.
saw_model["times"]	[1.0]	Times at which sawteeth occur.
saw_model["crash_width"]	10	Smoothing width of sawtooth crash [cm].
recycling_flag	False	If True, particles may return to main chamber, either via flows from the SOL or proper recycling.
wall_recycling	0.0	If True, recycling is activated: particles from the wall and divertor may return to main chamber.
source_div_time	None	(Optional) Time base for any particle sources going into the divertor reservoir [s].
source_div_vals	None	(Optional) Particle sources going into the divertor reservoir [particles/s/cm].
tau_div_SOL_ms	50.0	Time scale for transport between the divertor and the open SOL [ms].
tau_pump_ms	500.0	Time scale for pumping out of divertor reservoir [ms].
tau_rcl_ret_ms	50.0	Time scale for retention at the wall [ms] before recycling may occur.
SOL_mach	0.1	Mach number in the SOL, determining parallel loss rates.
kin_profs["ne"]	{'fun': 'interpa', 'times': [1.0]}	Specification of electron density [cm^{-3}]. *fun="interpa"* interpolates data also in the SOL.
kin_profs["Te"]	{'fun': 'interp', 'times': [1.0], 'decay': [1.0]}	Specification of electron temperature [eV]. *fun="interp"* sets decay over *decay* length in the SOL.
kin_profs["Ti"]	{'fun': 'interp', 'times': [1.0], 'decay': [1.0]}	Specification of ion temperature [eV]. Only used for charge exchange rates.
kin_profs["n0"]	{'fun': 'interpa', 'times': [1.0]}	Specification of background (H-isotope) neutral density [cm^{-3}].

continues on next page

Table 1 – continued from previous page

Parameter	Default	Description
nbi_cxr	{'rhop': None, 'vals': None}	Radial profiles of charge exchange rates from NBI neutrals (fast+thermal) for each simulated charge state.
cxr_flag	False	If True, activate charge exchange recombination with background thermal neutrals. Requires *kin_profs["n0"]*.
nbi_cxr_flag	False	If True, activate charge exchange recombination with NBI neutrals (to be specified in `aurora_sim`).
device	CMOD	Name of experimental device, only used by MDS+ if device database can be read via omfit_eqdsk.
shot	99999	Shot number, only used in combination with *device* to connect to MDS+ databases.
time	1250	Time [ms] used to read magnetic equilibrium, if this is fetched via MDS+.
acd	None	ADAS ADF11 ACD file (recombination rates). If left to None, uses defaults in `adas_files_dict()` for the chosen ion species.
scd	None	ADAS ADF11 SCD file (ionization rates). If left to None, uses defaults in `adas_files_dict()` for the chosen ion species.
ccd	None	ADAS ADF11 CCD file (nl-unresolved charge exchange rates). If left to None, uses defaults in `adas_files_dict()` for the chosen ion species.

3.4.2 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␣
```

(continues on next page)

(continued from previous page)

```
↪when sawteeth are modelled!
timing['steps_per_cycle'] = [2, 5, 1]   # last value not actually used, except␣
↪when sawteeth are modelled!
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.

3.4.3 Particle sources

Core sources of particles can be specified in a number of ways. A time- and radially-dependent source can be set by setting *namelist['source_type'] = 'arbitrary_2d_source'* and then providing the parameters

1. *explicit_source_rhop* : radial grid (in square root of normalized poloidal flux)
2. *explicit_source_time* : time grid (in seconds)
3. *explicit_source_vals* : values of source flux (particles/s)

Alternatively, if time and radial dependences of core sources can be effectively separated, source time histories and radial profiles can be described in other ways. The time history of core sources can be created using the *get_source_time_history()* function, whereas radial profiles of core sources can be defined by specifying parameters for the *get_radial_source()* function. Please refer to the documentation of these functions for explanations of how to call these.

Particle sources can also be specified such that they enter the simulation from the divertor reservoir. This parameter can be useful to simulate divertor puffing. Note that it can only have an effect if *recycling_flag* = True and *wall_recycling* is >=0, so that particles from the divertor are allowed to flow to the main chamber plasma. In order to specify a source into the divertor, one needs to specify 2 parameters:

1. *source_div_time* : time base for the particle source into the divertor;
2. *source_div_vals* : values of the particle source into the divertor.

Note that while core sources (e.g. in *explicit_source_vals*) are in units of $particles/cm^3$, sources going into the divertor have different units of $particles/cm/s$ since they are going into a 0D edge model.

3.4.4 Edge parameters

A 1.5D transport model such as Aurora cannot accurately model edge transport. 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.

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.

3.4.5 Kinetic profiles

In this section, we add a few more details on the specification of kinetic profiles in the Aurora namelist for 1.5D simulations of ion transport. We reproduce here the rows of the previous table that are relevant to this.

Table 2: Kinetic profiles specification

Parameter	Default	Description
kin_profs["ne"]	{'fun': 'interpa', 'times': [1.0]}	Specification of electron density [cm^{-3}]. *fun=interpa* interpolates data also in the SOL.
kin_profs["Te"]	{'fun': 'interp', 'times': [1.0], 'decay': [1.0]}	Specification of electron temperature [eV]. *fun=interp* sets decay over *decay* length in the SOL.
kin_profs["Ti"]	{'fun': 'interp', 'times': [1.0], 'decay': [1.0]}	Specification of ion temperature [eV]. Only used for charge exchange rates.
kin_profs["n0"]	{'fun': 'interpa', 'times': [1.0]}	Specification of background (H-isotope) neutral density [cm^{-3}].

Simulations that don't include charge exchange will only need electron density (*ne*) and temperature (*Te*). If charge exchange is added, then an ion temperature *Ti* and background H-isotope neutral density must be specified. Note that *Ti* should strictly be $T_{red} = (m_H T_n + m_{imp} T_i)/(T_n + T_i)$, where *m_H* is the background species mass and *T_n* is the background neutral temperature, since only the effective ("reduced") energy of the neutral-impurity interaction enters the evaluation of charge exchange rates. *Ti* is also used to compute parallel loss rates in the SOL; if not provided by users, it is substituted by *Te*.

Each field of *kin_profs* requires specification of *fun*, *times*, *rhop* and *vals*.

1. *fun* corresponds to a specification of interpolation functions in Aurora. Users should choose whether to interpolate data as given also in the SOL (*fun=interp*) or if SOL profiles should be substituted by an exponential decay. In the latter case, a decay scale length (in cm units) should also be provided as *decay*.
2. *times* is a 1D array of times, in seconds, at which time-dependent profiles are given. If only a single value is given, whatever it may be, profiles are taken to be time independent.
3. *rhop* is a 1D array of radial grid values, given as square-root of normalized poloidal flux.

4. *vals* is a 2D array of values of the given kinetic quantity. The first dimension is expected to be time, the second radial coordinate.

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 to produce PECs using ADAS ADF04 files and running ColRadPy's collisional-radiative model. This functionality is already available in the *get_colradpy_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

Aurora is released under the MIT License, one of the most common, permissive, open-source software licenses. This licensing option aims at making the package as useful and widely-applicable as possibe, in an effort to support the development of fusion energy. In the spirit of an open-source, collabWe 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 2021, "Modeling of particle transport, neutrals and radiation in magnetically-confined plasmas with Aurora", Plasma Phys. Control. Fusion 63 112001, https://doi.org/10.1088/1361-6587/ac2890

> This paper introduces Aurora and describes the general development philosophy, structure of the forward model for impurity transport, use of atomic data, the theory behind superstaging, and a few example applications for fusion simulation and data analysis.

[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, it owes to it for laying much of the groundwork.

If you use Aurora, you will likely need atomic rates of some sort. Aurora offers a simple interface to Atomic Data and Analysis Structure (ADAS) files - if you make use of this, please make sure to cite appropriately. Information about publicly available ADAS data can be found at https://open.adas.ac.uk/ . To cite ADAS, it would be reasonable to use

[3] H.P. Summers et al, 2001, "The ADAS manual" version ** [see the version number at https://www.adas.ac.uk/manual.php]

[4] H.P. Summers et al, 2006 Plasma Phys. Control. Fusion 48 263. https://iopscience.iop.org/article/10.1088/0741-3335/48/2/007

If you know the origin of the ADAS data that you are using, it would also be good to cite the specific works that resulted in the ADAS-distributed results. Other references for specific methods/data used in Aurora are shown in some docstrings.

If you use the FACIT neoclassical transport model in Aurora, please cite the following papers: Maget et al 2022 Plasma Phys. Control. Fusion 64 069501, Fajardo et al 2022 Plasma Phys. Control. Fusion 64 055017, Fajardo et al 2023 Plasma Phys. Control. Fusion 65 035021.

3.7 Questions and contributions

For any questions on Aurora, to brainstorm on possible applications or request changes to the code, please contact francesco.sciortino-at-ipp.mpg.de.

The code is developed and maintained by F. Sciortino (MPI-IPP) in collaboration with T. Odstrcil (GA) and contributions by D. Fajardo (MPI-IPP), A. Zito (MPI-IPP), A. Cavallaro (MIT), C. Johnson (ORNL), O. Linder (MPI-IPP), R. Reksoatmodjo (W&M), A. Rosenthal (MIT), D. Vezinet (CEA). For an updated list of contributors, please see the Github contributors list.

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*)

Bases: object

Setup the input dictionary for an Aurora ion transport simulation from the given namelist.

Parameters

- **namelist** (*dict*) – Dictionary containing aurora inputs. See default_nml.py for some defaults, which users should modify for their runs.
- **geqdsk** (*dict, optional*) – EFIT gfile as returned after postprocessing by the omfit_classes.omfit_eqdsk package (OMFITgeqdsk class). If left to None (default), the minor and major radius must be indicated in the namelist in order to create a radial grid.

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_asymmetry()* 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).
- **plot** (*bool*) – If True, plot asymmetry factor λ vs. radius

Returns

CF_lambda – Asymmetry factor, defined as λ in the *centrifugal_asymmetry()* function docstring.

Return type

array (nr,)

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

Check particle conservation for an aurora simulation.

Parameters

- **plot** (*bool, optional*) – If True, plot time histories in each particle reservoir and display quality of particle conservation.
- **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

- **out** (*dict*) – Dictionary containing density of particles in each reservoir.
- **axs** (*matplotlib.Axes instances, only returned if plot=True*) – Array-like structure containing two matplotlib.Axes instances, (ax1,ax2). See optional input argument.

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].

Parameters

trust_SOL_Ti (*bool*) – If True, the input Ti is trusted also in the SOL to calculate

a parallel loss rate. Often, Ti measurements in the SOL are unrealiable, so this parameter is set to False by default.

Returns

dv – Parallel loss rates in s^{-1} units. Values are zero in the core region and non-zero in the SOL.

Return type

array (space,time)

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.

load(*filename*)

Load *aurora_sim* object.

load_dict(*aurora_dict*)

plot_resolutions()

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

reload_namelist(*namelist=None*)

(Re-)load namelist to update scalar variables.

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

Run a simulation using the provided diffusion and convection profiles as a function of space, time and potentially also ionization state. Users can 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** (*array, shape of (space,time,nZ) or (space,time) or (space,)*) – Diffusion coefficients, in units of cm^2/s. This may be given as a function of space only, (space,time) or (space,nZ, time), where nZ indicates the number of charge states. If given with 1 or 2 dimensions, it is assumed that all charge states should have the same diffusion coefficients. If given as 1D, it is further assumed that diffusion is time-independent. Note that it is assumed that radial profiles are already on the self.rvol_grid radial grid.
- **V_z** (*array, shape of (space,time,nZ) or (space,time) or (space,)*) – Convection coefficients, in units of cm/s. This may be given as a function of space only, (space,time) or (space,nZ, time), where nZ indicates

the number of charge states. If given with 1 or 2 dimensions, it is assumed that all charge states should have the same convection coefficients. If given as 1D, it is further assumed that convection is time-independent. Note that it is assumed that radial profiles are already on the self.rvol_grid radial grid.

- **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.
- **unstage** (*bool, optional*) – If superstages are indicated in the namelist, this parameter sets whether the output should be "unstaged" by multiplying by the appropriate fractional abundances of all charge states at ionization equilibrium. Note that this unstaging process cannot account for transport and is therefore only an approximation, to be used carefully.
- **alg_opt** (*int, optional*) – If *alg_opt=1*, use the finite-volume algorithm proposed by Linder et al. NF 2020. If *alg_opt=0*, 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. NB: It is recommended to only use this with explicit 2D sources, otherwise
- **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

- **nz** (*array, (nr,nZ,nt)*) – Charge state densities [cm^{-3}] over the space and time grids. If a number of superstages are indicated in the input, only charge state densities for these are returned.
- **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 [cm^{-3}]
- **N_dsu** (*array (nt,)*) – Parallel particle loss [cm^{-3}]
- **N_dsul** (*array (nt,)*) – Parallel particle loss at the limiter [cm^{-3}]

- **rcld_rate** (*array (nt,)*) – Recycling from the divertor [$s^{-1}cm^{-3}$]
- **rclw_rate** (*array (nt,)*) – Recycling from the wall [$s^{-1}cm^{-3}$]

run_aurora_steady(*D_z*, *V_z*, *nz_init=None*, *unstage=False*, *alg_opt=1*, *evolneut=False*, *use_julia=False*, *tolerance=0.01*, *max_sim_time=100*, *dt=0.0001*, *dt_increase=1.05*, *n_steps=100*, *plot=False*)

Run an Aurora simulation until reaching steady state profiles. This method calls `run_aurora()` checking at every iteration whether profile shapes are still changing within a given fractional tolerance. Note that this method differs from `run_aurora_steady_analytic()` in that it runs time-dependent simulations until reaching steady-state, rather than using an analytic solution for steady-state profiles.

Parameters

- **D_z** (*array, shape of (space,nZ) or (space,)*) – Diffusion coefficients, in units of cm^2/s. This may be given as a function of space only or (space,nZ). No time dependence is allowed in this function. Here, nZ indicates the number of charge states. Note that it is assumed that radial profiles are already on the self.rvol_grid radial grid.
- **V_z** (*array, shape of (space,nZ) or (space,)*) – Convection coefficients, in units of cm/s. This may be given as a function of space only or (space,nZ). No time dependence is allowed in this function. Here, nZ indicates the number of charge states.
- **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.
- **unstage** (*bool, optional*) – If a list of superstages are provided in the namelist, this parameter sets whether the output should be "unstaged". See docs for `run_aurora()` for details.
- **alg_opt** (*int, optional*) – If *alg_opt=1*, use the finite-volume algorithm proposed by Linder et al. NF 2020. If *alg_opt=0*, 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. See docs for `run_aurora()` for details.
- **use_julia** (*bool, optional*) – If True, run the Julia pre-compiled version of the code. See docs for `run_aurora()` for details.
- **tolerance** (*float*) – Fractional tolerance in charge state profile shapes. This method reports charge state density profiles obtained when the discrepancy between normalized profiles at adjacent time steps varies by less than this tolerance fraction.
- **max_sim_time** (*float*) – Maximum time in units of seconds for which simulations should be run if a steady state is not found.

- **dt** (*float*) – Initial time step to apply, in units of seconds. This can be increased by a multiplier given by **:param:`dt_increase`** after each time step.
- **dt_increase** (*float*) – Multiplier for time steps.
- **n_steps** (*int*) – Number of time steps (>2) before convergence is checked.
- **plot** (*bool*) – If True, plot time evolution of charge state density profiles to show convergence.

run_aurora_steady_analytic(*D_z*, *V_z*)

Evaluate the analytic steady state solution of the transport equation. Small differences in absolute densities from the full time-dependent, multi-reservoir Aurora solutions can be caused by recycling and divertor models, which are not included here.

Parameters

- **D_z** (*array, shape of (space,nZ) or (space,)*) – Diffusion coefficients, in units of cm^2/s. This may be given as a function of space only or (space,nZ). No time dependence is allowed in this function. Here, nZ indicates the number of charge states. Note that it is assumed that radial profiles are already on the self.rvol_grid radial grid.
- **V_z** (*array, shape of (space,nZ) or (space,)*) – Convection coefficients, in units of cm/s. This may be given as a function of space only or (space,nZ). No time dependence is allowed in this function. Here, nZ indicates the number of charge states.

save(*filename*)

Save state of *aurora_sim* object.

save_dict()

set_time_dept_atomic_rates(*superstages=[]*, *metastables=False*)

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.

Parameters

- **superstages** (*list or 1D array*) – Indices of charge states that should be kept as superstages. The default is to have this as an empty list, in which case all charge states are kept.
- **metastables** (*bool*) – Load metastable resolved atomic data and rates. Default is False.

Sne_rates

Effective ionization rates [s]. If superstages were indicated, these are the rates of superstages.

Type

array (space, nZ(-super), time)

Rne_rates

Effective recombination rates [s]. If superstages were indicated, these are the rates of superstages.

Type
array (space, nZ(-super), time)

`Qne_rates`

Cross-coupling coefficients, only computed if :param:`metastables`=True.

Type
array (space, nZ(-super), time)

`Xne_rates`

Parent cross-coupling coefficients, only computed if :param:`metastables`=True.

Type
array (space, nZ(-super), time)

`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.

`superstage_DV(`*D_z*, *V_z*, *times_DV=None*, *opt=1*`)`

Reduce the dimensionality of D and V time-dependent profiles for the case in which superstaging is applied.

Three options are currently available:

1. opt=1 gives a simple selection of D_z and V_z fields corresponding to each superstage index.
2. opt=2 averages D_z and V_z over the charge states that are part of each superstage.

#. opt=3 weights D_z and V_z corresponding to each superstage by the fractional abundances at ionization equilibrium. This is mostly untested – use with care!

Parameters

- **D_z** (*array, shape of (space,time,nZ)*) – Diffusion coefficients, in units of cm^2/s.
- **V_z** (*array, shape of (space,time,nZ)*) – Convection coefficients, in units of cm/s.

Returns

- **Dzf** (*array, shape of (space,time,nZ-superstages)*) – Diffusion coefficients of superstages, in units of cm^2/s.
- **Vzf** (*array, shape of (space,time,nZ-superstages)*) – Convection coefficients of superstages, in units of cm/s.

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

Fast linear interpolation for 1D and 2D vector data on equally spaced grids. This offers optimal speed in Python for interpolation of atomic data tables such as the ADAS ones.

Parameters

- **grids** (*list of arrays, N=len(grids), N=1 or N=2*) – List of 1D arrays with equally spaced grid values for each dimension
- **values** (*N+1 dimensional array of values used for interpolation*) – Values to interpolate. The first dimension typically refers to different ion stages, for which data is provided on the input grids. Other dimensions refer to values on the density and temperature grids.

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.

Parameters

filepath (*str*) – Path to location where ADAS file is located.

load()

ADF11 format description: https://www.adas.ac.uk/man/appxa-11.pdf

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

Plot data from input ADAS file. If provided, the arguments allow users to overplot and compare data from multiple files.

Parameters

- **fig** (*matplotlib Figure object*) – If provided, add specification as to which ADAS file is being plotted.
- **axes** (*matplotlib Axes object (or equivalent)*) – If provided, plot on these axes. Note that this typically needs to be a set of axes for each plotted charge state. Users may want to call this function once first to get some axes, and then pass those same axes to a second call for another file to compare with.

aurora.atomic.**get_adas_file_types**()

Obtain a description of each ADAS file type and its meaning in the context of Aurora.

Returns

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

Return type
dict

Notes

For background on ADAS generalized collisional-radiative modeling and data formats, refer to [1]_.

References

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

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.
- **files** (*list or dict*) – ADAS file types to be fetched. Default is ["acd","scd"] for effective ionization and recombination rates (excluding CX) using default files, listed in `adas_files_adas_files_dict()`. If users prefer to use specific files, they may pass a dictionary instead, of the form

```
{'acd': 'acd89_ar.dat', 'scd': 'scd89_ar.dat'}
```

or

```
{'acd': 'acd89_ar.dat', 'scd': None}
```

if only some of the files need specifications and others (given as None) should be taken from the default files.

Returns
atom_data – 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 res.logNe, res.logT, res.logdata, res.meta_ind, res.metastables

Return type
dict

aurora.atomic.**get_atomic_relax_time**(*atom_data*, *ne_cm3*, *Te_eV=None*, *Ti_eV=None*, *n0_by_ne=0.0*, *superstages=[]*, *tau_s=inf*, *plot=True*, *ax=None*, *ls='-'*)

Obtain the relaxation time of the ionization equilibrium for a given atomic species.

If n0_by_ne is not 0, thermal charge exchange is added to radiative and dielectronic recombination. This function can work with ne,Te and n0_by_ne arrays of arbitrary dimension. It uses a matrix SVD approach in order to find the relaxation rates, as opposed to the simpler approach of `get_frac_abundances()`, but fractional abundances produced by the two methods should always be the same.

This function also allows use of superstages as well as specification of a τ value representing the effect of transport on the ionization equilibrium. NB: this is only a rough metric to characterize complex physics.

Parameters

- **atom_data** (*dictionary of atomic ADAS files (only acd, scd are required; ccd is*) – necessary only if n0_by_ne is not 0).
- **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.
- **Ti_eV** (*float or array*) – Bulk ion temperature in units of eV, only needed for CX. If left to None, Ti is set equal to Te.
- **n0_by_ne** (*float or array, optional*) – Ratio of background neutral hydrogen to electron density. If set to 0, CX is not considered.
- **superstages** (*list or 1D array*) – Indices of charge states of chosen ion that should be included. If left empty, all ion stages are included. If only some indices are given, these are modeled as "superstages".
- **tau_s** (*float, opt*) – Value of the particle residence time [s]. This is a scalar value that can be used to model the effect of transport on ionization equilibrium. Setting tau=np.inf (default) corresponds to no effect from transport.
- **plot** (*bool, optional*) – If True, the atomic relaxation time is plotted as a function of Te. Default is True.
- **ax** (*matplotlib.pyplot Axes instance*) – Axes on which to plot if plot=True. If False, new axes are created.
- **ls** (*str, optional*) – Line style for plots. Continuous lines are used by default.

Returns

- **Te** (*array*) – 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}].

Examples

To visualize relaxation times for a given species: >>> atom_data = aurora.atomic.get_atom_data('N', ["scd", "acd"]) >>> aurora.get_atomic_relax_time(atom_data, [1e14], Te_eV=np.linspace(0.1,200,1000), plot=True)

To compare ionization balance with different values of ne*tau: >>> Te0, fz0, r0 = aurora.get_atomic_relax_time(atom_data, [1e14], Te_eV=np.linspace(0.1,200,1000), tau_s=1e-3, plot=False) >>> Te1, fz1, r1 = aurora.get_atomic_relax_time(atom_data, [1e14], Te_eV=np.linspace(0.1,200,1000), tau_s=1e-2, plot=False) >>> Te2, fz2, r2 = aurora.get_atomic_relax_time(atom_data, [1e14], Te_eV=np.linspace(0.1,200,1000), tau_s=1e-1, plot=False) >>> >>> plt.figure() >>> for cs in np.arange(fz0.shape[1]): >>> l = plt.plot(Te0, fz0[:,cs], ls='-') >>> plt.plot(Te1, fz1[:,cs], c=l[0].get_color(), ls='–') >>> plt.plot(Te2, fz2[:,cs], c=l[0].get_color(), ls='-.')

aurora.atomic.**get_cs_balance_terms**(*atom_data*, *ne_cm3=50000000000000.0*, *Te_eV=None*, *Ti_eV=None*, *include_cx=True*, *metastables=False*)

Get S*ne, R*ne and cx*ne rates 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.
- **Ti_eV** (*float or array*) – Bulk ion temperature in units of eV, only needed for CX. If left to None, Ti is set equal to Te.
- **include_cx** (*bool*) – If True, obtain charge exchange terms as well.

Returns

- **Te** (*array (n_Te)*) – Te grid on which atomic rates are given
- **Sne, Rne (,cxne, Qne, Xne)** (*arrays (n_ne,n_Te)*) – atomic rates for effective ionization, radiative+dielectronic recombination (+ charge exchange, + crosscoupling if requested). All terms will be in units of s^{-1}.

Notes

The nomenclature with 'ne' at the end of rate names indicates that these are rates in units of $m^3 \cdot s^{-1}$ multiplied by the electron density 'ne' (hence, final units of s^{-1}).

aurora.atomic.**get_frac_abundances**(*atom_data*, *ne_cm3*, *Te_eV=None*, *Ti_eV=None*, *n0_by_ne=0.0*, *superstages=[]*, *plot=True*, *ax=None*, *rho=None*, *rho_lbl=None*)

Calculate fractional abundances from ionization and recombination equilibrium. If n0_by_ne is not 0, 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.
- **Ti_eV** (*float or array, optional*) – Bulk ion temperature in units of eV. If left to None, Ti is set to be equal to Te
- **n0_by_ne** (*float or array, optional*) – Ratio of background neutral hydrogen to electron density. If not 0, CX is considered.
- **superstages** (*list or 1D array*) – Indices of charge states of chosen ion that should be included. If left empty, all ion stages are included. If only some indices are given, these are modeled as "superstages".
- **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 "x".

Returns

- **Te** (*array*) – 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.

aurora.atomic.**get_natural_partition**(*ion*, *plot=True*)

Identify natural partition of charge states by plotting the variation of ionization energy as a function of charge for a given ion, using the ADAS metric $2(I_{z+1} - I_z)/(I_{z+1} + I_z)$.

Parameters

- **ion** (*str*) – Atomic symbol of species of interest.
- **plot** (*bool*) – If True, plot the variation of ionization energy.

Returns

q – Metric to identify natural partition.

Return type
1D array

Notes

A ColRadPy installation must be available for this function to work.

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*, *freq='all'*, *cutoff=0.1*)

Approximate impurity bremsstrahlung in :math:`W/m^3`for a given range of frequencies or a specific frequency.

We apply here a cutoff for Bremsstrahlung at h*c/lambda = cutoff*Te, where *cutoff* is an input parameter, conventionally set to 0.1 (default).

Gaunt factors from *gff_mean()* are applied. NB: recombination is not included.

Formulation based on Hutchinson's Principles of Plasma Diagnostics, p. 196, Eq. (5.3.40).

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}]
- **freq** (*float, 1D array, or str*) – If a float, calculate bremsstrahlung from all charge states at this frequency. If a 1D array, evaluate bremsstrahlung at these wavelengths. If set to *all*, then bremsstrahlung is integrated over the whole range from plasma frequency to cutoff Frequencies are expected in units of s^{-1}.
- **cutoff** (*float*) – Fraction of Te below which bremsstrahlung is set to 0. A value of 0.1 is commonly set and is the default.

Returns
Bremsstrahlung for each charge state at the given frequency or, if multiple frequences are given (or if *freq='all'*), integrated over frequencies. Units of W/cm^3.

Return type
array (time,nZ,space)

aurora.atomic.**interp_atom_prof**(*atom_table*, *xprof*, *yprof*, *log_val=False*, *x_multiply=True*)

Fast interpolate atomic data in atom_table onto the xprof and yprof profiles. This function assume that xprof, yprof, x, y, table are all base-10 logarithms, and xprof, yprof are equally spaced.

Parameters

- **atom_table** (*list*) – object atom_data, containing atomic data from one of the ADAS files.

- **xprof** (*array (nt,nr)*) – Spatio-temporal profiles of the first coordinate of the ADAS file table (usually electron density in cm^{-3})
- **yprof** (*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^{xprof}.

Returns

interp_vals – Interpolated atomic data on time,charge state and spatial grid that correspond to the ion of interest and the spatiotemporal grids of xprof and yprof.

Return type

array (nt,nion,nr)

Notes

This function uses *np.log10* and exponential operations to optimize speed, since it has been observed that base-e operations are faster than base-10 operations in numpy.

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 passing and trapped impurity ions take to travel a parallel distance $L = qR$, defining

$$\nu^*_{ion} \equiv \nu_{ion}\tau_t = \nu_{ion}\frac{qR}{v_{th}} = \frac{\sum_z n_z \nu_z^{ion}}{\sum_z n_z} qR\sqrt{\frac{m_{imp}}{2k_B T_i}}$$

following Ref.[1]_. If the normalized ionization rate (ν^*_{ion}) is less than 1, then flux surface averaging of background asymmetries (e.g. from edge or beam neutrals) may be taken as a good approximation of reality; in this case, 1.5D simulations of impurity transport are expected to be valid. If, on the other hand, $\nu^*_{ion} > 1$ then local effects may be too important to ignore.

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

nu_ioniz_star – 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

array (r,cs) or (r,)

References

aurora.atomic.**read_adf12**(*filename*, *block*, *Ebeam*, *ne_cm3*, *Ti_eV*, *zeff*)

Read charge exchange effective emission coefficients from ADAS ADF12 files.

Files may be automatically downloaded using **:py:fun:`~aurora.adas_files.get_adas_file_loc`**.

Parameters

- **filename** (*str*) – adf12 file name/path
- **block** (*int*) – Source block selected
- **Ebeam** (*float*) – Energy of the neutral beam population of interest, in units of eV/amu.
- **ne_cm3** (*float or 1D array*) – Electron densities at which to evaluate coefficients, in units of cm^{-3}.
- **Ti_eV** (*float or 1D array*) – Bulk ion temperature at which to evaluate coefficients, in units of eV.
- **Zeff** (*float or 1D array*) – Effective background charge.

Returns

Interpolated coefficients, units of cm^3/s.

Return type

float or 1D array

aurora.atomic.**read_adf21**(*filename*, *Ebeam*, *ne_cm3*, *Te_eV*)

Read ADAS ADF21 or ADF22 files.

ADF21 files contain effective beam stopping/excitation coefficients. ADF22 contain effective beam emission/population coefficients.

Files may be automatically downloaded using **:py:fun:`~aurora.adas_files.get_adas_file_loc`**.

Parameters

- **filename** (*str*) – adf21 or adf22 file name/path
- **Ebeam** (*float*) – Energy of the neutral beam, in units of eV/amu.
- **ne_cm3** (*float or 1D array*) – Electron densities at which to evaluate coefficients.
- **Te_eV** (*float or 1D array*) – Electron temperature at which to evaluate coefficients.

Returns

Interpolated coefficients. For ADF21 files, these have units of :math:`cm^3/s` for ADF21 files. For ADF22, they correspond to n=2 fractional abundances.

Return type

float or 1D array

aurora.atomic.**read_filter_response**(*filepath*, *adas_format=True*, *plot=False*)

Read a filter response function over energy.

This function attempts to read the data checking for the following formats (in this order):

1. The ADAS format. Typically, this data is from obtained from http://xray.uu.se and produced via ADAS routines.
2. The format returned by the Center for X-Ray Optics website .

Note that filter response functions are typically a combination of a filter transmissivity and a detector absorption.

Parameters

- **filepath** (*str*) – Path to filter file of interest.
- **plot** (*bool*) – If True, the filter response function is plotted.

aurora.atomic.**superstage_rates**(*R*, *S*, *superstages*, *save_time=None*)

Compute rate for a set of ion superstages. Input and output rates are log-values in arbitrary base.

Parameters

- **R** (*array (time,nZ,space)*) – Array containing the effective recombination rates for all ion stages, These are typically combinations of radiative and dielectronic recombination, possibly also of charge exchange recombination.
- **S** (*array (time,nZ,space)*) – Array containing the effective ionization rates for all ion stages.
- **superstages** (*list or 1D array*) – Indices of charge states of chosen ion that should be included.
- **save_time** (*list or 1D array of bools*) – Indices of the timeslcies which are actually returned by AURORA

Returns

- **superstages** (*array*) – Set of superstages including 0,1 and final stages if these were missing in the input.
- **R_rates_super** (*array (time,nZ-super,space)*) – effective recombination rates for superstages
- **S_rates_super** (*array (time,nZ-super,space)*) – effective ionization rates for superstages
- **fz_upstage** (*array (space, nZ, time_*) – fractional abundances of stages within superstages

3.8.4 aurora.adas_files module

Functions to provide default ADAS files for Aurora modelling, including capabilities to fetch derived data 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

files – 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

dict

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

Find location of requested atomic data file for the indicated ion. Accepts all ADAS "derived" data file types (adf11, adf12, adf13, adf15, adf21, adf22). 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 input filename is actually a full path to the file on the local system, use this file and copy it to Aurora/aurora/adas_data/actual_filename, where actual_filename is the file name rather than the full path.
3. If the environmental variable "AURORA_ADAS_DIR" is defined, attempt to find the file there, under adf11/ and adf15/ directories. AURORA_ADAS_DIR may for example be the path to a central repository of ADAS files for Aurora on a cluster where not everyone may have write-permissions. For this option, files are not copied at all.
4. Attempt to fetch the file remotely via open.adas.ac.uk and save it in Aurora/aurora/adas_data/*filetype*/.

Parameters

- **filename** (*str*) – Name of the ADAS file of interest, e.g. 'plt89_ar.dat'.
- **filetype** (*str*) – ADAS file type. Only derived data types are allowed, i.e. one of "adf11", "adf12", "adf13", "adf15", "adf21", "adf22"

Returns

file_loc – Full path to the requested file.

Return type

str

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.**adf15_line_identification**(*pec_files*, *lines=None*, *Te_eV=1000.0*, *ne_cm3=50000000000000.0*, *mult=[]*)

Display all photon emissivity coefficients from the given list of ADF15 files and (optionally) compare to a set of chosen wavelengths, given in units of Angstrom.

Parameters

- **pec_files** (*str or list of str*) – Path to a single ADF15 file or a list of files.
- **lines** (*dict, list or 1D array*) – Lines to overplot with the loaded PECs to consider overlap within spectrum. This argument may be a dictionary, with keys corresponding to line names and values corresponding to wavelengths (in units of Angstrom). If provided as a list or array, this is assumed to contain only wavelengths in Angstrom.
- **Te_eV** (*float*) – Single value of electron temperature at which PECs should be evaluated [eV].
- **ne_cm3** (*float*) – Single value of electron density at which PECs should be evaluated [cm^{-3}].
- **mult** (*list or array*) – Multiplier to apply to lines from each PEC file. This could be used for example to rescale the results of multiple ADF15 files by the expected fractional abundance or density of each element/charge state.

Notes

To attempt identification of spectral lines, one can load a set of ADF15 files, calculate approximate fractional abundances at equilibrium and overplot expected emissivities in a few steps:

```
>>> pec_files = ['mypecs1','mypecs2','mypecs3']
>>> Te_eV=500; ne_cm3=5e13; ion='Ar'   # examples
>>> atom_data = aurora.atomic.get_atom_data(ion,['scd','acd'])
>>> _Te, fz = aurora.atomic.get_frac_abundances(atom_data, ne_cm3, Te_eV,␣
↪plot=False)
>>> mult = [fz[0,10], fz[0,11], fz[0,12]] # to select charge states 11+,␣
↪12+ and 13+, for example
>>> aurora.adf15_line_identification(pec_files, Te_eV=Te_eV, ne_cm3=ne_cm3,␣
↪mult=mult)
```

3.8.5.1 Minimal Working Example

```
>>> Te_eV=500; ne_cm3=5e13 # eV and cm^{-3}
>>> filepath = aurora.get_adas_file_loc('pec96#he_pju#he0.dat', filetype=
↪'adf15')
>>> aurora.adf15_line_identification(filepath, Te_eV=Te_eV, ne_cm3=ne_cm3)
```

aurora.radiation.**compute_rad**(*imp*, *nz*, *ne*, *Te*, *n0=None*, *Ti=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 Wcm^{-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.
- **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
res – Dictionary containing radiation terms, depending on the activated flags.

Return type
dict

Notes

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.

aurora.radiation.**get_colradpy_pec_prof**(*ion*, *cs*, *rhop*, *ne_cm3*, *Te_eV*, *lam_nm*, *lam_width_nm*, *adf04_loc*, *meta_idxs=[0]*, *pec_threshold=1e-20*, *pec_units=2*, *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 '17', indicating total charge of ion (e.g. '17' would be for Li-like Ca)
- **rhop** (*array (nr,)*) – Sqrt 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]
- **adf04_loc** (*str*) – Location from which ADF04 files listed in *adf04_files()* should be fetched.

- **meta_idxs** (*list of integers*) – List of levels in ADF04 file to be treated as metastable states. Default is [0] (only ground state).
- **prec_threshold** (*float*) – Minimum value of PECs to be considered, in $photons \cdot cm^3/s$
- **pec_units** (*int*) – If 1, results are given in $photons \cdot cm^3/s$; if 2, they are given in $W.cm^3$. Default is 2.
- **plot** (*bool*) – If True, plot lines profiles and total.

Returns

pec_tot_prof – Radial profile of PEC intensity, in units of $photons \cdot cm^3/s$ (if pec_units=1) or $W \cdot cm^3$ (if pec_units=2).

Return type

array (nr,)

aurora.radiation.**get_cooling_factors**(*imp*, *ne_cm3*, *Te_eV*, *n0_cm3=0.0*, *ion_resolved=False*, *superstages=[]*, *line_rad_file=None*, *cont_rad_file=None*, *sxr=False*, *plot=True*, *ax=None*)

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

Parameters

- **imp** (*str*) – Atomic symbol of ion of interest
- **ne_cm3** (*1D array*) – Electron density $[cm^{-3}]$, used to find charge state fractions at ionization equilibrium.
- **Te_eV** (*1D array*) – Electron temperature $[eV]$ at which cooling factors should be obtained.
- **n0_cm3** (*1D array or float*) – Background H/D/T neutral density $[cm^{-3}]$ used to account for charge exchange when calculating ionization equilibrium. If left to 0, charge exchange effects are not included.
- **ion_resolved** (*bool*) – If True, cooling factors are returned for each charge state. If False, they are summed over charge states. The latter option is useful for modeling where charge states are assumed to be in ionization equilibrium (no transport). Default is False.
- **superstages** (*list or 1D array*) – List of superstages to consider. An empty list (default) corresponds to the inclusion of all charge states. Note that when ion_resolved=False, cooling coefficients are independent of whether superstages are being used or not.
- **line_rad_file** (*str or None*) – Location of ADAS ADF11 file containing line radiation data. This can be a PLT (unfiltered) or PLS (filtered) file. If left to None, the default file given in *adas_files_dict()* will be used.
- **cont_rad_file** (*str or None*) – Location of ADAS ADF11 file containing recombination and bremsstrahlung radiation data. This can be a PRB (unfiltered) or PRS (filtered) file. If left to None, the default file given in *adas_files_dict()* will be used.

- **sxr** (*bool*) – If True, line radiation, recombination and bremsstrahlung radiation are taken to be from SXR-filtered ADAS ADF11 files, rather than from unfiltered files.
- **plot** (*bool*) – If True, plot all radiation components, summed over charge states.
- **ax** (*matplotlib.Axes instance*) – If provided, plot results on these axes.

Returns

- **line_rad_tot** (*1D array*) – Cooling coefficient from line radiation [$W \cdot m^3$]. Depending on whether :param:`sxr`=True or False, this indicates filtered or unfiltered radiation, respectively.
- **cont_rad_tot** (*1D array*) – Cooling coefficient from continuum radiation [$W \cdot m^3$]. Depending on whether `sxr`=True or False, this indicates filtered or unfiltered radiation, respectively.

aurora.radiation.**get_local_spectrum**(*adf15_file*, *ne_cm3*, *Te_eV*, *ion_exc_rec_dens=[0, 1, 0]*, *Ti_eV=None*, *n0_cm3=0.0*, *dlam_A=0.0*, *plot_spec_tot=True*, *plot_all_lines=False*, *no_leg=False*, *ax=None*)

Plot spectrum based on the lines contained in an ADAS ADF15 file at specific values of electron density and temperature. Charge state densities can be given explicitely, or alternatively charge state fractions will be automatically computed from ionization equilibrium (no transport).

Parameters

- **adf15_file** (*str or dict*) – Path on disk to the ADAS ADF15 file of interest or dictionary returned when calling the *read_adf15()* with this file path as an argument. All wavelengths and radiating components in the file or dictionary will be read/processed.
- **ne_cm3** (*float*) – Local value of electron density, in units of cm^{-3}.
- **Te_eV** (*float*) – Local value of electron temperature, in units of eV. This is used to evaluate local values of photon emissivity coefficients.
- **ion_exc_rec_dens** (*list of 3 floats*) – Density of ionizing, excited and recombining charge states that may contribute to emission from the given ADF15 file. In the absence of charge state densities from particle transport modeling, these scalars may be taken from the output of *aurora.atomic.get_frac_abundances()*. Default is [0,1,0], which means that only excitation components are considered.
- **Ti_eV** (*float*) – Local value of ion temperature, in units of eV. This is used to represent the effect of Doppler broadening. If left to None, it is internally set equal to *Te_eV*.
- **n0_cm3** (*float, optional*) – Local density of atomic neutral hydrogen isotopes. This is only used if the provided ADF15 file contains charge exchange contributions.

- **dlam_A** (*float or 1D array*) – Doppler shift in A. This can either be a scalar or an array of the same shape as the output wavelength array. For the latter option, it is recommended to call this function twice to find the wave_final_A array first.
- **plot_spec_tot** (*bool*) – If True, plot total spectrum (sum over all components) from given ADF15 file.
- **plot_all_lines** (*bool*) – If True, plot all individual lines, rather than just the profiles due to different atomic processes. If more than 50 lines are included, a down-selection is automatically made to avoid excessive memory consumption.
- **no_leg** (*bool*) – If True, no plot legend is shown. Default is False, i.e. show legend.
- **ax** (*matplotlib Axes instance*) – Axes to plot on if plot=True. If left to None, a new figure is created.

Returns

- **wave_final_A** (*1D array*) – Array of wavelengths in units of on which the total spectrum is returned.
- **spec_ion** (*1D array*) – Spectrum from ionizing components of the input ADF15 file as a function of wave_final_A.
- **spec_exc** (*1D array*) – Spectrum from excitation components of the input ADF15 file as a function of wave_final_A.
- **spec_rr** (*1D array*) – Spectrum from radiative recombination components of the input ADF15 file as a function of wave_final_A.
- **spec_dr** (*1D array*) – Spectrum from dielectronic recombination components of the input ADF15 file as a function of wave_final_A.
- **spec_cx** (*1D array*) – Spectrum from charge exchange recombination components of the input ADF15 file as a function of wave_final_A.
- **ax** (*matplotlib Axes instance*) – Axes on which the plot is returned.

Notes

Including ionizing, excited and recombining charge states allows for a complete description of spectral lines that may derive from various atomic processes in a plasma.

Doppler broadening depends on the local ion temperature and mass of the emitting species. It is modeled here using

$$\theta(\nu) = \frac{1}{\sqrt{\pi}\Delta\nu_D} e^{-\left(\frac{\nu-\nu_0}{\Delta\nu_D}\right)^2}$$

with the Doppler profile half-width being

$$\Delta\nu_D = \frac{1}{\nu_0}\sqrt{\frac{2T_i}{m}}$$

The Doppler shift dlam_A can be calculated from

$$\Delta\lambda_v = \lambda \cdot \left(1 - \frac{v \cdot \cos(\alpha)}{c}\right)$$

where v is the plasma velocity and α is the angle between the line-of-sight and the direction of plasma rotation.

Refs: S. Loch's and C. Johnson's PhD theses.

3.8.5.2 Minimal Working Example

Plot spectrum in one of the neutral H ADF15 files >>> filepath = aurora.get_adas_file_loc('pec96#h_pju#h0.dat', filetype='adf15') >>> trs = aurora.read_adf15(filepath) >>> Te_eV = 80.; ne_cm3 = 1e14 # local ne [cm^{-3}]and Te [eV] >>> out = aurora.get_local_spectrum(filepath, 'H', ne_cm3, Te_eV) # defaults to excitation-only

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

ni_cm3 – Estimated main ion density in cm^{-3}.

Return type

array (time,space)

aurora.radiation.**get_photon_emissivity**(*adf15*, *lam*, *ne_cm3*, *Te_eV*, *imp_density*, *n0_cm3=0*, *meta_ind=None*)

Evaluate PEC coefficients and return all components of intensity saved in an ADF15 file in units of ph/s.

Parameters

- **adf15** (*PandasFrame*) – ADF15 PEC coefficients
- **ne_cm3** (*array (nr)*) – Profile of electron density, in units of cm^{-3}.
- **Te_eV** (*array (nr)*) – Profile of electron temperature, in units of eV.

- **imp_density** (*list or 2D array (nstate, nr)*) – Density of all impurity states. These which are not needed can be set to None
- **n0_cm3** (*array, optional*) – Local density of atomic neutral hydrogen isotopes. This is only used if the provided ADF15 file contains charge exchange contributions.
- **meta_ind** (*list (nstate), optional*) – indexes of metastates corresponding to imp_density

Returns
intensity – arrays of line emissivity in ph/s units

Return type
dict of arrays (nr)

aurora.radiation.**parse_adf15_configs**(*path*)

Parse ADF15 file to identify electron configurations and energies used to produce Photon Emissivity Coefficients (PECs) in the same file.

Parameters
path (*str*) – Path to adf15 file to read.

Returns
DataFrame containing information about all electron configurations used for PEC calculations.

Return type
pandas.DataFrame

Notes

The format of ADAS ADF15 files has varied over time; consequently, it is surprisingly difficult to figure out parsing methods that work for all files. If this function fails on one of your files, please report this via a Github issue and one of the Aurora core developers will help adapting the parser.

aurora.radiation.**parse_adf15_spec**(*lines*, *num_lines*)

Parse full description of provided rates from an ADF15 file.

Parameters

- **lines** (*list or None*) – List of lines read from ADF15 files.
- **num_lines** (*int*) – Number of spectral lines in the processed ADF15 file.

Returns
Dictionary containing information about all loaded transitions.

Return type
dict

aurora.radiation.**plot_pec**(*transition*, *ax=None*, *plot_3d=False*)

Plot a single Photon Emissivity Coefficient dependency on electron density and temperature.

Parameters

- **transition** (*pandas array with the PEC data*) –
- **ax** (*matplotlib axes instance*) – If not None, plot on this set of axes.
- **plot_3d** (*bool*) – Display PEC data as 3D plots rather than 2D ones.

Examples

```
>>> filename = 'pec96#h_pju#h0.dat'
Fetch file automatically, locally, from AURORA_ADAS_DIR, or directly from␣
↪the web:
>>> path = aurora.get_adas_file_loc(filename, filetype='adf15')
Load all transitions provided in the chosen ADF15 file:
>>> trs = aurora.read_adf15(path)
Select the Lyman-alpha transition and plot its rates:
>>> tr = trs[(trs['lambda [A]']==1215.2) & (trs['type']=='excit')]
>>> aurora.plot_pec(tr)
Alternatively, to select a line using the ADAS "ISEL" indices:
>>> aurora.plot_pec(trs.loc[(trs['isel']==1)])
or
>>> aurora.plot_pec(trs.iloc[0])
```

aurora.radiation.**radiation_model**(*imp*, *rhop*, *ne_cm3*, *Te_eV*, *geqdsk*, *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 `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
- **geqdsk** (*dict, optional*) – EFIT gfile as returned after postprocessing by the `omfit_classes.omfit_eqdsk` package (OMFITgeqdsk class).
- **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

res – Dictionary containing results of radiation model.

Return type

dict

aurora.radiation.**read_adf15**(*path*, *order=1*)

Read photon emissivity coefficients (PECs) from an ADAS ADF15 file.

Returns a *pandas.DataFrame* object describing all transitions available within the chosen ADF15 file.

PECs are provided in the form of an interpolant that will evaluate the log10 of the PEC at a desired electron density and temperature. The power-10 exponentiation of this PEC has units of $photons \cdot cm^3/s$.

Units for interpolation: cm^{-3} for density; eV for temperature.

Parameters

- **path** (*str*) – Path to adf15 file to read.
- **order** (*int*) – Parameter to control the order of interpolation. Default is 1 (linear interpolation).

Returns

Parsed data from the given ADF15 file, including an interpolation function for the log-10 of the PEC of each spectral line. See the examples below for advice on using this form of output for plotting and further analysis.

Return type

pandas.DataFrame

Examples

To plot the Lyman-alpha photon emissivity coefficients for H (or its isotopes), one can use:

```
>>> import aurora
>>> filename = 'pec96#h_pju#h0.dat'
>>> # fetch file automatically, locally, from AURORA_ADAS_DIR, or directly␣
↪from the web:
>>> path = aurora.get_adas_file_loc(filename, filetype='adf15')
>>> # load all transitions provided in the chosen ADF15 file:
>>> trs = aurora.read_adf15(path)
>>> # select the excitation-driven component of the Lyman-alpha transition:
>>> tr = trs[(trs['lambda [A]']==1215.2) & (trs['type']=='excit')]
>>> # now plot the rates:
>>> aurora.plot_pec(tr)
```

Note that excitation-, recombination- and charge-exchange driven components can be fetched in the same way by specifying the *type*, e.g. using >>> trs['type']=='excit' # 'excit', 'recom' or 'chexc'

Since the output of *aurora.radiation.read_adf15* is a pandas DataFrame, its contents can be indexed in a variety of ways, e.g. one can select a line based on the ADAS "ISEL" indices and plot it using >>> aurora.plot_pec(trs.loc[(trs['isel']==1)]) or simply >>> aurora.plot_pec(trs.iloc[0]) Spectral lines are ordered by ISEL numbers -1 (Python indexing!), so using the *iloc* method of a pandas DataFrame allows effective indexing by ISEL numbers.

The log-10 of the PEC interpolation function can be used as >>> tr['log10 PEC fun'].iloc[0].ev(np.log10(ne_cm3), np.log10(Te_eV)) where the interpolant was evaluated (via the *ev* method) at specific points of *n_e* (units of $[cm^{-3}]$) and *T_e* (units of eV). Note that the log-10 of *n_e* and *T_e* is needed, not *n_e* and *T_e* themselves!

Metastable-resolved files are automatically identified based on the file nomenclature and parsed accordingly, e.g.:

```
>>> filename = 'pec96#he_pjr#he0.dat'
>>> path = aurora.get_adas_file_loc(filename, filetype='adf15')
>>> trs = aurora.read_adf15(path)
>>> aurora.plot_pec(trs[trs['lambda [A]']==584.4])
```

Notes

This function expects the format of PEC files produced via the ADAS adas810 or adas218 routines.

aurora.radiation.**sync_rad**(*B_T*, *ne_cm3*, *Te_eV*, *r_min*, *R_maj*)

Calculate synchrotron radiation following Trubnikov's formula [1]_. We make use of a simplified formulation as given by Zohm[2].

Parameters

[2] Zohm et al., Journal of Fusion Energy (2019) 38:3-10

- **B_T** (*float or 1D array*) – Magnetic field amplitude [T].
- **ne_cm3** (*float or 1D array*) – Electron density [cm^{-3}]
- **Te_eV** (*float or 1D array*) – Electron temperature [eV]
- **r_min** (*float*) – Minor radius [m].
- **R_maj** (*float*) – Major radius [m].
- **Returns** –
- **array** – Rate of synchrotron radiation [W/cm^3]

References

3.8.6 aurora.grids_utils module

Methods to create radial and time grids for aurora simulations.

exception aurora.grids_utils.**MissingAuroraBuild**

Bases: UserWarning

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

- **time** (*array*) – Computational time grid corresponding to *timing* input.
- **save** (*array*) – Array of zeros and ones, where ones indicate that the time step will be stored in memory in aurora simulations. Points corresponding to zeros will not be returned to spare memory.

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

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

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

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

The corresponding radial step size is

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

See the STRAHL manual for details.

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

- **rvol_grid** (*array*) – Volume-normalized grid used for Aurora simulations.
- **pro** (*array*) – Normalized first derivatives of the radial grid, defined as pro = (drho/dr)/(2 d_rho) = rho'/(2 d_rho)
- **qpr** (*array*) – Normalized second derivatives of the radial grid, defined as qpr = (d^2 rho/dr^2)/(2 d_rho) = rho"/(2 d_rho)
- **prox_param** (*float*) – Grid parameter used for perpendicular loss rate at the last radial grid point.

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

This interfaces the package subroutine to create the radial grid exactly as STRAHL does it. Refer to the STRAHL manual for details.

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

- **time** (*array*) – Computational time grid corresponding to :param:timing input.
- **save** (*array*) – Array of zeros and ones, where ones indicate that the time step will be stored in memory in Aurora simulations. Points corresponding to zeros will not be returned to spare memory.

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

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

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

- **t_vals** (*array*) – Times in the time base [s]
- **i_save** (*array*) – Array of 0,1 values indicating at which times internal arrays should be stored/returned.

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

aurora.grids_utils.**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

- **bound_sep** (*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.

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 *omfit_classes.omfit_eqdsk*.

Returns

- **clen_divertor** (*float*) – Estimate of the connection length to the divertor [m]
- **clen_limiter** (*float*) – Estimate of the connection length to the limiter [m]
~~~~~~~~~~~~

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_rvol_mapping()* for an application.

Parameters

- **geqdsk** (*dict*) – Dictionary containing the g-EQDSK file as processed by the *omfit_classes.omfit_eqdsk*.
- **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

- **Rhfs** (*array*) – Major radius [m] on the HFS
- **Rlfs** (*array*) – Major radius [m] on the LFS

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:

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

The mapping between rho_pol and 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 *omfit_classes.omfit_eqdsk*.
- **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

- **rho_pol** (*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.

aurora.grids_utils.**vol_int**(*var*, *rvol_grid*, *pro_grid*, *Raxis_cm*, *rvol_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

- **var** (*2D+ array (time, ..., radius)*) – Data array for which a volume integral should be evaluated. The last dimension must be radial, other dimensions are arbitrary.
- **rvol_grid** (*1D array*) – Volume-normalized radial grid.
- **pro_grid** – Normalized first derivative of the radial grid, see *create_radial_grid()*.
- **Raxis_cm** (*float*) – Major radius on axis [cm]
- **rvol_max** (*float*) – Maximum volume-normalized radius for integral. If not provided, integrate over the entire simulation radial grid.

Returns

var_volint – Time history of volume integrated variable

Return type

array (nt,)

3.8.7 aurora.coords module

aurora.coords.**get_rhop_RZ**(*R*, *Z*, *geqdsk*)

Find rhop at every R,Z [m] based on the equilibrium in the geqdsk dictionary.

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 :math:`langle Q

angle = Sigma_i Q(r_i) 2 pi Delta r_V`

where Δr_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.

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.

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. A number of common coordinate nomenclatures are accepted, but it is recommended to use one of the coordinate names indicated in the input descriptions below.

Parameters

- **x** (*array or float*) – input x coordinate
- **name_in** (*str*) – input x coordinate name ('rhon','rvol','rhop','rhov','Rmid','rmid','r/a')
- **name_out** (*str*) – input x coordinate ('rhon','psin','rvol', 'rhop','rhov','Rmid','rmid','r/a')
- **geqdsk** (*dict*) – gEQDSK dictionary, as obtained from the omfit-eqdsk package.

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

Return type
array

aurora.coords.**rhoTheta2RZ**(*geqdsk*, *rho*, *theta*, *coord_in='rhop'*, *n_line=201*)

Convert values of rho,theta into R,Z coordinates from a geqdsk dictionary.

Parameters

- **geqdsk** (output of the omfit_classes.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_classes.omfit_eqdsk*. In this case, device, shot and time to fetch the equilibrium
- **rho** (*np.ndarray*) – Values of normalized radial coordinate to consider.
- **theta** (*np.ndarray*) – Values of poloidal angle coordinate to consider.
- **coord_in** (*str*) – Label describing the nature of the radial coordinate in use.
- **n_line** (*int*) – Number of points to discretize flux surface.
- **Results** –
- **-------** –
- **R** (*np.array, (ntheta, nrho)*) – Values of the major radius along flux surfaces.
- **Z** (*np.array, (ntheta, nrho)*) – Values of the vertical coordinate along flux surfaces.

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_classes.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_classes.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_classes.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_classes.omfit_eqdsk* dictionary is also returned

Returns

- **quant_vol_avg** (*array, (space, ...)*) – Volume average of the quantity given as an input, in the same units as in the input. If extrapolation beyond the range available from EFIT volume averages over a shorter section of the radial grid will be attempted. This does not affect volume averages within the LCFS.
- **geqdsk** (*dict*) – Only returned if return_geqdsk=True.

3.8.8 aurora.source_utils module

Methods related to impurity source functions.

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.
- **Ti_eV** (*array, optional (nt,nr)*) – 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

source_rad_prof – Radial profile of the impurity neutral source for each time step.

Return type

array (nr,nt)

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

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

Parameters

- **namelist** (*dict*) – Aurora namelist dictionary. The field namelist['source_type'] specifies how the source function is being specified – see the notes below.
- **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

source_time_history – The source time history on the input time base.

Return type
array (nt,)

Notes

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

#. 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.

#. namelist['source_type'] == 'interp': the time history for the source is provided by the user within the 'explicit_source_time' and 'explicit_source_vals' fields of the namelist dictionary and this data is simply interpolated.

#. 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.

#. 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']

#. 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.

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

- **time_vec** (*array*) – Times for the source function of each given impurity
- **source** (*array*) – Time history of the synthetized source function.

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

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

Parameters

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

Returns

- **t** (array of float, (*n*,)) – The timebase (in seconds).
- **s** (array of float, (*n*,)) – The source function (#/s).

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

Write a STRAHL source file.

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

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

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

Return type

str

3.8.9 aurora.plot_tools module

class aurora.plot_tools.**DraggableColorbar**(*cbar*, *mapimage*)

Bases: object

Create a draggable colorbar for matplotlib plots to enable quick changes in color scale.

Example::

```
fig,ax = plt.subplots()
cntr = ax.contourf(R, Z, vals)
cbar = plt.colorbar(cntr, format='%.3g', ax=ax)
```

(continues on next page)

(continued from previous page)

```
cbar = DraggableColorbar(cbar,cntr)
cbar.connect()
```

connect()

Matplotlib connection for button and key pressing, release, and motion.

disconnect()

key_press(*event*)

Key pressing event

on_motion(*event*)

Move if the mouse is over the colorbar.

on_press(*event*)

Button pressing; check if mouse is over colorbar.

on_release(*event*)

Upon release, reset press data

aurora.plot_tools.**get_color_cycle**(*num=None*, *map='plasma'*)

Get an iterable to select different colors in a loop. Efficiently splits a chosen color map into as many (*num*) parts as needed.

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

aurora.default_nml.**load_default_namelist**()

Load default namelist. Users should modify and complement this for a successful forward-model 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

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.**Lya_to_neut_dens**(*emiss_prof*, *ne*, *Te*, *ni=None*, *plot=True*, *rhop=None*, *rates_source='adas'*, *axs=None*)

Estimate ground state neutral density from measured emissivity profiles. This ignores possible molecular dynamics and effects that may be captured via forward modeling of neutral transport.

Parameters

- **emiss_prof** (*1D array*) – Emissivity profile, units of W/cm
- **ne** (*1D array*) – Electron density, units of cm^{-3}
- **Te** (*1D array*) – Electron temperature, units of eV
- **ni** (*1D array*) – Main ion (H/D/T) density, units of cm^{-3}. If left to None, this is internally set to ni=ne.
- **plot** (*bool*) – If True, plot some of the key density profiles.
- **rhop** (*1D array*) – Sqrt of normalized poloidal flux radial coordinate. Used only for plotting.
- **rates_source** (*str*) – Source of atomic rates. Possible choices are 'adas' or 'colrad'
- **axs** (*Axes instance*) – If given, plot on these axes.

Returns

N1 – Radial profile of estimated ground state atomic neutral density on the same grid as the input arrays. Units of cm^{-3}.

Return type

1D array

Examples

```
>>> N2_colrad,axs = Lya_to_neut_dens_basic(emiss_prof, ne, Te, ni,
>>>                                        plot=True, rhop=rhop, rates_source=
↪'colrad')
```

```
>>> N2_adas,axs = Lya_to_neut_dens_basic(emiss_prof, ne, Te, ni, plot=True,␣
↪rhop=rhop,
>>>                         rates_source='adas',axs=axs)
```

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

$$N_m/N_1 = \left(\frac{N_m^i}{N_1}\right) N_m + \left(\frac{N_m^{ii}}{n_i}\right) n_i$$

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.

Parameters

- **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.
- **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 – Density of electrons in excited state *n* [cm^{-3}]

Return type

array of shape [len(ni)=len(N1),len(ne),len(Te)]

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.
- **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

Ns – 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

list of arrays

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_keV*, *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_keV** (*float, 1D or 2D array*) – Target temperature [keV]. Results will be computed for each Ti_keV 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

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

Return type

float, 1D or 2D array

aurora.nbi_neutrals.**get_NBI_imp_cxr_q**(*neut_fsa*, *q*, *rhop_kp*, *times_kp*, *Ti_eV*, *ne_cm3*, *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 and ne may be time-dependent, with a time base given by times_kp, the FSA neutrals are expected to be time-independent. Hence, the resulting CXR rates will only have time dependence that reflects changes in Ti and ne, 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_kp** (*array-like*) – Sqrt of poloidal flux radial coordinate for Ti profiles.
- **times_kp** (*array-like*) – Time base on which Ti_eV is given [s].
- **Ti_eV** (*array-like*) – Ion temperature profile on the rhop_kp, times_kp bases, in units of eV.
- **ne_cm3** (*array-like*) – Electron density profile on the rhop_kp, times_kp bases, in units of cm^{-3}.
- **include_fast** (*bool, optional*) – If True, include CXR rates from fast NBI neutrals. Default is True.
- **include_halo** (*bool, optional*) – If True, include CXR rates from themral NBI halo neutrals. Default is True.
- **debug_plots** (*bool, optional*) – If True, plot several plots to assess the quality of the calculation.

Returns

rates – 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
dict

Notes

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

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** (dictionary output of *omfit_classes.omfit_eqdsk.OMFITgeqdsk* class) – gEQDSK post-processed dictionary, as given by *omfit_classes.omfit_eqdsk*.

- **debug_plots** (*bool, optional*) – If True, various plots are displayed.

Returns

neut_fsa – 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
dict

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_keV*, *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_keV** (*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.

Returns

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

Return type

float or 1D array

Notes

This does not currently account for the effect of rotation! Doing so will require making the integration in this function 2-dimensional.

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.

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

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

Return type

float

Notes

See comments in Janev & Smith 1993 for uncertainty estimates.

aurora.janev_smith_rates.**js_sigma_cx_n1_q1**(*E*)

Electron capture cross section for

$$H^+ + H(1s) - - > H + H^+$$

Notes

Section 2.3.1 of Janev & Smith, NF 1993.

aurora.janev_smith_rates.**js_sigma_cx_n1_q2**(*E*)

Electron capture cross section for

$$He^{2+} + H(1s) - - > He^+ + H^+$$

Notes

Section 3.3.1 of Janev & Smith, NF 1993.

aurora.janev_smith_rates.**js_sigma_cx_n1_q4**(E)

Electron capture cross section for

$$Be^{4+} + H(1s) - - > Be^{3+} + H^{+}$$

Notes

Section 4.3.1 of Janev & Smith, NF 1993.

aurora.janev_smith_rates.**js_sigma_cx_n1_q5**(E)

Electron capture cross section for

$$B^{5+} + H(1s) - - > B^{4+} + H^{+}$$

Notes

Section 4.3.2 of Janev & Smith, NF 1993.

aurora.janev_smith_rates.**js_sigma_cx_n1_q6**(E)

Electron capture cross section for

$$C^{6+} + H(1s) - - > C^{5+} + H^{+}$$

Notes

Section 4.3.3 of Janev & Smith, NF 1993.

aurora.janev_smith_rates.**js_sigma_cx_n1_q8**(E)

Electron capture cross section for

$$O^{8+} + H(1s) - - > O^{7+} + H^{+}$$

Notes

Section 4.3.4 of Janev & Smith, NF 1993.

aurora.janev_smith_rates.**js_sigma_cx_n1_qg8**(E, q)

Electron capture cross section for

$$A^{q+} + H(1s) - - > A^{(q-1)+} + H^{+}, q > 8$$

Notes

Section 4.3.5, p.172, of Janev & Smith, NF 1993.

aurora.janev_smith_rates.**js_sigma_cx_n2_q2**(*E*)

Electron capture cross section for

$$He^{2+} + H(n=2) - - > He^{+} + H^{+}$$

Notes

Section 3.3.2 of Janev & Smith, NF 1993.

aurora.janev_smith_rates.**js_sigma_cx_ng1_q1**(*E*, *n1*)

Electron capture cross section for

$$H^{+} + H(n) - - > H + H^{+}, n > 1$$

Notes

Section 2.3.2 of Janev & Smith, NF 1993.

aurora.janev_smith_rates.**js_sigma_cx_ng1_qg3**(*E*, *n1*, *q*)

Electron capture cross section for

$$A^{q+} + H^{*}(n) - - > A^{(q-1)+} + H^{+}, q > 3$$

Notes

Section 4.3.6, p.174, of Janev & Smith, NF 1993.

aurora.janev_smith_rates.**js_sigma_cx_ng2_q2**(*E*, *n1*)

Electron capture cross section for

$$He^{2+} + H*(n) - - > He^{+} + H^{+}, n > 2$$

Notes

Section 3.2.3 of Janev & Smith, NF 1993.

aurora.janev_smith_rates.**js_sigma_ioniz_n1_q8**(*E*)

Ionization cross section for

$$O^{8+} + H(1s) - - > O^{8+} + H^{+} + e^{-}$$

Notes

Section 4.2.4 of Janev & Smith, NF 1993.

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

Plot/check sensitivity of JS cross sections to beam energy. NB: cross section is taken to only depend on partially-screened ion charge

3.8.17 aurora.synth_diags module

aurora.synth_diags.**centrifugal_asymmetry**(*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

$$n(r,\theta) = n_0(r) \times \exp\left(\lambda(\rho)(R(r,\theta)^2 - R_0^2)\right)$$

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).

Parameters

- **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.
- **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_classes.omfit_eqdsk* reading of the EFIT g-file.

Returns

CF_lam – Asymmetry factor, defined as λ in the expression above.

Return type

array (nr,)

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.
- **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 aurora.kn1d module

Aurora functionality to set up and run KN1D to extract atomic and neutral background densities at the edge.

KN1D is a 1D kinetic neutral code originally developed by B.LaBombard (MIT). For information, refer to the KN1D manual .

Note that this Aurora module is merely a wrapper of KN1D. Users require an IDL license on the computer where this module is called in order to be able to run KN1D. The IDL (and Fortran) code themselves are automatically downloaded and compiled by this module.

aurora.kn1d.**plot_emiss**(*res*, *check_collrad=True*)

Plot profiles of Ly-a and D-alpha emissivity from the KN1D output. KN1D internally computes Ly-a and D-alpha emission using the Johnson-Hinnov coefficients; here we check the result of that calculation and compare it to the prediction from atomic data from the COLLRAD collisional-radiative model included in DEGAS2.

Parameters

- **res** (*dict*) – Output dictionary from function *run_kn1d()*.
- **check_collrad** (*bool*) – If True, compare KN1D prediction of Ly-a and D-a emission using Johnson-Hinnov rates using rates from COLLRAD.

aurora.kn1d.**plot_exc_states**(*res*)

Plot excited state fractions of atomic neutral density from a KN1D run.

Parameters

res (*dict*) – Output dictionary from function *run_kn1d()*.

aurora.kn1d.**plot_input_kin_prof**(*rmid_to_wall_m*, *ne_cm3*, *Te_eV*, *Ti_eV*, *innermost_rmid_cm*, *bound_sep_cm*, *lim_sep_cm*)

Plot extent of kinetic profiles entering KN1D calculation

aurora.kn1d.**plot_overview**(*res*)

Plot an overview of a KN1D run, showing both kinetic profile inputs and a small selection of the outputs.

Parameters

res (*dict*) – Output dictionary from function *run_kn1d()*.

aurora.kn1d.**plot_transport**(*res*)

Make a simple set of plots of gradient scale lengths and effective diffusion coefficients from the KN1D output.

Parameters

res (*dict*) – Output dictionary from function *run_kn1d()*.

aurora.kn1d.**run_kn1d**(*rhop*, *ne_cm3*, *Te_eV*, *Ti_eV*, *geqdsk*, *p_H2_mTorr*, *clen_divertor_cm*, *clen_limiter_cm*, *bound_sep_cm*, *lim_sep_cm*, *innermost_rmid_cm=5.0*, *mu=2.0*, *pipe_diag_cm=0.0*, *vx=0.0*, *collisions={}*, *kin_prof_exp_decay_SOL=False*, *kin_prof_exp_decay_LS=False*, *ne_decay_len_cm=[1.0, 1.0]*, *Te_decay_len_cm=[1.0, 1.0]*, *Ti_decay_len_cm=[1.0, 1.0]*, *ne_min_cm3=1000000000000.0*, *Te_min_eV=1.0*, *Ti_min_eV=1.0*, *plot_kin_profs=False*)

Run KN1D for the given parameters. Refer to the KN1D manual for details.

Depending on the provided options, kinetic profiles are extended beyond the Last Closed Flux Surface (LCxFS) and the Limiter Shadow (LS) via exponential decays with specified decay lengths. It is assumed that the given kinetic profiles extend from the core until at least the LCFS. All inputs are taken to be time-independent.

This function automatically checks if a KN1D repository is available; if it is not, it obtains it from the web and compiles the necessary code.

Note that an IDL license must be available. Aurora does not currently include a Python translation of KN1D – it only acts as a wrapper.

Parameters

- **rhop** (*1D array*) – Sqrt of poloidal flux grid on which ne_cm3, Te_eV and Ti_eV are given.
- **ne_cm3** (*1D array*) – Electron density on rhop grid [cm^{-3}].
- **Te_eV** (*1D array*) – Electron temperature on rhop grid [eV].
- **Ti_eV** (*1D array*) – Main ion temperature on rhop grid [eV].
- **geqdsk** (*omfit_classes.omfit_eqdsk.OMFITgeqdsk* class instance) – gEQDSK file as processed by the *omfit_classes.omfit_eqdsk.OMFITgeqdsk* class.
- **p_H2_mTorr** (*float*) – Pressure of molecular hydrogen-isotopes measured at the wall. This may be estimated from experimental pressure gauges. This variable effectively sets the amplitude of the neutral source at the edge. Units of $mTorr$.
- **clen_divertor_cm** (*float*) – Connection length from the midplane to the divertor [cm].
- **clen_limiter_cm** (*float*) – Connection length from the midplane to the limiter [cm].
- **bound_sep_cm** (*float*) – Distance between the wall/boundary and the separatrix [cm].
- **lim_sep_cm** (*float*) – Distance between the limiter and the separatrix [cm].
- **innermost_rmid_cm** (*float*) – Distance from the wall to solve for. Default is 5 cm.
- **mu** (*float*) – Atomic mass number of simulated species. Default is 2.0 (D).
- **pipe_diag_cm** (*float*) – Diameter of the pipe through which H2 pressure is measured (see *p_H2_mTorr* variable). If left to 0, this diameter is effectively set to infinity. Default is 0.
- **vx** (*float*) – Radial velocity imposed on neutrals. This only has a weak effect usually. Default is 0 [cm/s].
- **collisions** (*dict*) – Collision terms flags. Set each to True or False. If any of the flags are not given, all collision terms are internally set to be active. Possible flags are 'H2_H2_EL','H2_P_EL','H2_H_EL','H2_HP_CX','H_H_EL','H_P_CX','H_P_EL','Simple_CX'
- **kin_prof_exp_decay_SOL** (*bool*) – If True, kinetic profiles are set to exponentially decay over the SOL region.
- **kin_prof_exp_decay_LS** (*bool*) – If True, kinetic profiles are set to exponentially decay over the LS region.
- **ne_decay_len_cm** (*list of 2 float*) – Exponential decay lengths of electron density in the SOL and LS regions. Default is [1,1] cm.

- **Te_decay_len_cm** (*float*) – Exponential decay lengths of electron temperature in the SOL and LS regions. Default is [1,1] cm.
- **Ti_decay_len_cm** (*float*) – Exponential decay lengths of main ion temperature in the SOL and LS regions. Default is [1,1] cm.
- **ne_min_cm3** (*float*) – Minimum electron density across profile. Default is $10^{12} cm^{-3}$.
- **Te_min_eV** (*float*) – Minimum electron temperaure across profile. Default is eV.
- **Ti_min_eV** (*float*) – Minimum main ion temperaure across profile. Default is eV.
- **plot_kin_profs** (*bool*) – If True, kinetic profiles input to KN1D are plotted.

Returns
KN1D results and inputs, all collected into a dictionary. See example script for an illustration of using this.

Return type
dict

Notes

For an example application, see the examples/aurora_kn1d.py script.

3.8.19 aurora.solps module

Aurora tools to read SOLPS results and extract atomic neutral density from EIRENE output. These enable examination of charge exchange recombination and radiation due to the interaction of heavy ions and thermal neutrals.

aurora.solps.**apply_mask**(*triang*, *geqdsk*, *max_mask_len=0.4*, *mask_up=False*, *mask_down=False*)

Function to apply basic masking to a matplolib triangulation. This type of masking is useful to avoid having triangulation edges going outside of the true simulation grid.

Parameters

- **triang** (*instance of matplotlib.tri.triangulation.Triangulation*) – Matplotlib triangulation object for the (R,Z) grid.
- **geqdsk** (*dict*) – Dictionary containing gEQDSK file values as processed by *omfit_classes.omfit_eqdsk*.
- **max_mask_len** (*float*) – Maximum length [m] of segments within the triangulation. Segments longer than this value will not be plotted. This helps avoiding triangulation over regions where no data should be plotted, beyond the actual simulation grid.
- **mask_up** (*bool*) – If True, values in the upper vertical half of the mesh are masked. Default is False.

- **mask_down** (*bool*) – If True, values in the lower vertical half of the mesh are masked. Default is False.

Returns
triang – Masked instance of the input matplotlib triangulation object.

Return type
instance of matplotlib.tri.triangulation.Triangulation

aurora.solps.**get_fort44_info**(*NDX*, *NDY*, *NATM*, *NMOL*, *NION*, *NSTRA*, *NCL*, *NPLS*, *NSTS*, *NLIM*)

Collection of labels and dimensions for all fort.44 variables, as collected in the SOLPS-ITER 2020 manual.

aurora.solps.**get_fort46_info**(*NTRII*, *NATM*, *NMOL*, *NION*)

Collection of labels and dimensions for all fort.46 variables, as collected in the SOLPS-ITER 2020 manual.

aurora.solps.**get_mdsmap**()

Load dictionary allowing a mapping of chosen variables with SOLPS variable names on MDS+.

class aurora.solps.**solps_case**(**args*, ***kwargs*)

Bases: object

Read SOLPS output, either from

1. output files on disk
2. an MDS+ tree

If arguments are provided with no keys, it is assumed that paths to b2fstate andd b2fgmtry are being provided (option #1). If keyword arguments are provided instead, we first check whether *b2fstate_path* and *b2fgmtry_path* are being given (option #1) or *solps_id* is given to load results from MDS+ (option #2). Other keyword arguments can be provided to specify the MDS+ server and tree to use (defaults are for AUG), and to provide a gEQDSK file.

Parameters

- **b2fstate_path** (*str (option #1)*) – Path to SOLPS b2fstate output file.
- **b2fgmtry_path** (*str (option #1)*) – Path to SOLPS b2fgmtry output file.
- **solps_id** (*str or int (option #2)*) – Integer identifying the SOLPS run/shot to load from MDS+.
- **server** (*str, optional (option #2)*) – MDS+ server to load SOLPS results. Default is 'solps-mdsplus.aug.ipp.mpg.de:8001'.
- **tree** (*str (option #2)*) – Name of MDS+ tree to load data from. Default is 'solps'.
- **geqdsk** (*str, optional (option #1 and #2)*) – Path to the geqdsk to load from disk. Users may also directly provide an instance of the *omfit_classes.omfit_geqdsk.OMFITgeqdsk* class that contains the processed gEQDSK file. If not provided, the code tries to reconstruct a geqdsk based on the experiment

name and time of analysis stored in the SOLPS output. If set to None, no geqdsk is loaded and functionality related to the geqdsk is not used.

Notes

The b2fstate and b2fgmtry parser expects the filenames to be simply named "b2fstate" and "b2fgmtry", or else it may not recognize fields appropriately.

3.8.19.1 Minimal Working Example

import aurora so = aurora.solps_case(solps_id=141349) # load case 141349 from AUG MDS+; must have access to the server! so.plot2d_b2(so.data('ne'))

data(*varname*)

Fetch data either from files or MDS+ tree.

Parameters

varname (*str*) – Name of SOLPS variable, e.g. ne,te,ti,dab2,etc.

eval_LOS(*pnt1*, *pnt2*, *vals*, *npt=501*, *method='linear'*, *plot=False*, *ax=None*, *label=None*)

Evaluate the SOLPS output *field* along the line-of-sight (LOS) given by the segment going from point *pnt1* to point *pnt2* in 3D geometry.

Parameters

- **pnt1** (*array (3,)*) – Cartesian coordinates x,y,z of first extremum of LOS.
- **pnt2** (*array (3,)*) – Cartesian coordinates x,y,z of second extremum of LOS.
- **vals** (*array (self.data('ny'), self.data('nx'))*) – Data array for a variable of interest.
- **npt** (*int, optional*) – Number of points to use for the path discretization.
- **method** (*{'linear','nearest','cubic'}, optional*) – Method of interpolation.
- **plot** (*bool, optional*) – If True, display variation of the *field* quantity as a function of the LOS path length from point *pnt1*.
- **ax** (*matplotlib axes, optional*) – Instance of figure axes to use for plotting. If None, a new figure is created.
- **label** (*str, optional*) – Text identifying the LOS under examination.

Returns

array – Values of requested SOLPS output along the LOS

Return type

(npt,)

find_gfile()

Identify the name of the gEQDSK file from the directory where b2fgmtry is also located.

find_xpoint()

Find location of the x-point in (R,Z) coordinates by using the fact that it is the only vertex shared by 8 cells.

get_3d_path(*pnt1*, *pnt2*, *npt=501*, *plot=False*, *ax=None*)

Given 2 points in 3D Cartesian coordinates, returns discretized R,Z coordinates along the segment that connects them.

Parameters

- **pnt1** (*array (3,)*) – Cartesian coordinates x,y,z of first path extremum (expected in [m]).
- **pnt2** (*array (3,)*) – Cartesian coordinates x,y,z of second path extremum (expected in [m]).
- **npt** (*int, optional*) – Number of points to use for the path discretization.
- **plot** (*bool, optional*) – If True, display displaying result. Default is False.
- **ax** (*matplotlib axes instance, optional*) – If provided, draw 3d path on these axes. Default is to create a new figure and also draw the B2.5 grid polygons.

Returns

- **pathR** (*array (npt,)*) – R [m] points along the path.
- **pathZ** (*array (npt,)*) – Z [m] points along the path.
- **pathL** (*array (npt,)*) – Length [m] discretization along the path.

get_b2_patches()

Get polygons describing B2 grid as a mp.collections.PatchCollection object.

get_poloidal_prof(*vals*, *plot=False*, *label=''*, *rhop=1.0*, *topology='LSN'*, *ax=None*)

Extract poloidal profile of a quantity "quant" from the SOLPS run. This function returns a profile of the specified quantity at the designated radial coordinate (rhop=1 by default) as a function of the poloidal angle, theta.

Note that double nulls ('DN') are not yet handled.

Parameters

- **vals** (*array (self.data('ny'), self.data('nx'))*) – Data array for a variable of interest.
- **plot** (*bool*) – If True, plot poloidal profile.
- **label** (*string*) – Label for plot
- **rhop** (*float*) – Radial coordinate, in rho_p, at which to take poloidal surface. Default is 1 (LCFS)
- **ax** (*matplotlib axes instance*) – Axes on which poloidal profile should be plotted. If not given, a new set of axes is created. This is useful to possibly overplot profiles at different radii.

Returns

- **theta_rhop** (*1D array*) – Poloidal grid measured in degrees from LFS midplane on which prof_rhop is given
- **prof_rhop** (*1D array*) – Mean poloidal profile at rhop on theta_rhop grid.
- **prof_rhop_std** (*1D array*) – Standard deviation of poloidal profile at rhop on the theta_rhop grid, based on variations within +/-*dr_mm*/2 millimeters from the surface at rhop.

`get_radial_prof`(*vals*, *dz_mm=5*, *theta=0*, *label=''*, *plot=False*)

Extract radial profiles of a quantity "quant" from the SOLPS run. This function returns profiles on the low- (LFS) and high-field-side (HFS) midplane, as well as flux surface averaged (FSA) ones.

Parameters

- **vals** (*array (self.data('ny'), self.data('nx'))*) – Data array for a variable of interest.
- **dz_mm** (*float*) – Vertical range [mm] over which quantity should be averaged near the midplane. Mean and standard deviation of profiles on the LFS and HFS will be returned based on variations of atomic neutral density within this vertical span. Note that this does not apply to the FSA calculation. Default is 5 mm.
- **theta** (*float (0-360)*) – Poloidal angle [degrees] at which to take radial profile, measured from 0 degrees at Outer Midplane. Default is 0 degrees
- **label** (*string*) – Optional string label for plot and legend. Default is empty ('')
- **plot** (*bool*) – If True, plot radial profiles.

Returns

- **rhop_fsa** (*1D array*) – Sqrt of poloidal flux grid on which FSA profiles are given.
- **prof_fsa** (*1D array*) – FSA profile on rhop_fsa grid.
- **rhop_LFS** (*1D array*) – Sqrt of poloidal flux grid on which LFS profile (prof_LFS) is given.
- **prof_LFS** (*1D array*) – Mean LFS midpane profile on rhop_LFS grid.
- **prof_LFS_std** (*1D array*) – Standard deviation of LFS midplane profile on the rhop_LFS grid, based on variations within +/-*dz_mm*/2 millimeters from the midplane.
- **rhop_HFS** (*1D array*) – Sqrt of poloidal flux grid on which the midplane HFS profile (prof_HFS) is given.
- **prof_HFS** (*1D array*) – Mean HFS midplane profile on rhop_HFS grid.
- **prof_HFS_std** (*1D array*) – Standard deviation of HFS midplane profile on rhop_HFS grid, based on variations within +/-*dz_mm*/2 millimeters from the midplane.

load_data(*fields=None*, *P_idxs=None*, *R_idxs=None*, *Rmin=None*, *Rmax=None*, *Pmin=None*, *Pmax=None*)

Load SOLPS output for each of the needed quantities

Parameters

- **fields** (*list or array*) – List of fields to fetch from SOLPS output. If left to None, by default uses ['ne','Te','nn','Tn','nm','Tm','Ti']
- **P_idxs** (*list or array*) – Poloidal indices to load.
- **R_idxs** (*list or array*) – Radial indices to load.
- **Rmin** (*int or None.*) – Minimum major radius index to load, if R_idxs is not given
- **Rmax** (*int or None*) – Maximum major radius index to load, if R_idxs is not given
- **Pmin** (*int or None*) – Minimum poloidal index to load, if P_idxs is not given
- **Pmax** (*int or None*) – Maximum poloidal index to load, if P_idxs is not given

load_eirene_mesh()

Load EIRENE nodes from the fort.33 file and triangulation from the fort.34 file

load_fort44()

Load result from one of the fort.44 file with EIRENE output on B2 grid.

Returns
out – Dictionary containing a subdictionary with keys for each loaded field.

Return type
dict

load_fort46()

Load result from fort.46 file with EIRENE output on EIRENE grid.

Returns
out – Dictionary for each loaded file containing a subdictionary with keys for each loaded field from each file.

Return type
dict

load_mesh_extra()

Load the mesh.extra file.

plot2d_b2(*vals*, *ax=None*, *scale='log'*, *label=''*, *lb=None*, *ub=None*, ***kwargs*)

Method to plot 2D fields on B2 grids. Colorbars are set to be manually adjustable, allowing variable image saturation.

Parameters

- **vals** (*array (self.data('ny'), self.data('nx'))*) – Data array for a variable of interest.

- **ax** (*matplotlib Axes instance*) – Axes on which to plot. If left to None, a new figure is created.
- **scale** (*str*) – Choice of 'linear','log' and 'symlog' for matplotlib.colors.
- **label** (*str*) – Label to set on the colorbar. No label by default.
- **lb** (*float*) – Lower bound for colorbar. If left to None, the minimum value in *vals* is used.
- **ub** (*float*) – Upper bound for colorbar. If left to None, the maximum value in *vals* is used.
- **kwargs** – Additional keyword arguments passed to the *PatchCollection* class.

plot2d_eirene(*vals*, *ax=None*, *scale='log'*, *label=''*, *lb=None*, *ub=None*, *replace_zero=True*, ***kwargs*)

Method to plot 2D fields from EIRENE.

Parameters

- **vals** (*array (self.triangles)*) – Data array for an EIRENE variable of interest.
- **ax** (*matplotlib Axes instance*) – Axes on which to plot. If left to None, a new figure is created.
- **scale** (*str*) – Choice of 'linear','log' and 'symlog' for matplotlib.colors.
- **label** (*str*) – Label to set on the colorbar. No label by default.
- **lb** (*float*) – Lower bound for colorbar. If left to None, the minimum value in *vals* is used.
- **ub** (*float*) – Upper bound for colorbar. If left to None, the maximum value in *vals* is used.
- **replace_zero** (*boolean*) – If True (default), replace all zeros in 'vals' with minimum value in 'vals'
- **kwargs** – Additional keyword arguments passed to the *tripcolor* function.

plot_radial_summary(*ls='o-b'*)

Plot a summary of radial profiles (ne, Te, Ti, nn, nm), at the inner and outer targets, as well as the outer midplane.

Parameters

ls (*str*) –

plot_solps_2d_overview()

Display 2D views of most important SOLPS outputs.

plot_wall_geometry()

Method to plot vessel wall segment geometry from wall_geometry field in fort.44 file

species_id()

Identify species included in SOLPS run, both for B2 and EIRENE quantities. This is only designed to work in the "full" data format.

3.8.20 Module contents

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

a

INDEX

D

E

F

G

I

Q

R

S

T

U

V

W

X