pythonradex Documentation

Release 0.1

Gianni Cataldi

December 22, 2017

CHAPTER ONE

INSTALLATION

`pythonradex` requires you to have python 3 installed on your system. The easiest way to install `pythonradex` is by using pip. In a terminal, type

```
pip install pythonradex
```

On can also install `pythonradex` by downloading the package from github (link??????). Then, in a terminal, cd into the directory containing the setup.py file. Install by typing

```
python setup.py install
```

After the installation finished, you may test that everything works by typing

```
python setup.py test
```

This will run the unit tests of the package.

CHAPTER TWO

SUMMARY

`pythonradex` is a python re-implementation of the RADEX (http://home.strw.leidenuniv.nl/~moldata/radex.html) code *[vanderTak07]*. It solves the radiative transfer for a uniform medium in non-LTE using the Accelerated Lambda Iteration (ALI) method in combination with an escape probability formalism. The code can be used to quickly estimate the emission from an astrophysical gas given input parameters such as the density and kinetic temperature of the collision partners, the column density of the gas and width of the emission lines.

`pyhonradex` also provides a convenient method to read files from the LAMDA (http://home.strw.leidenuniv.nl/~moldata/) database.

Note that `pythonradex` uses SI units.

CHAPTER

THREE

EXAMPLE

Radiative transfer

Let us consider a typical example of how pyhonradex is used. Note that all input should be in SI units. Assume you want to compute the emission of a CO cloud. First, let's import the necessary modules:

```
>>> from pythonradex import nebula, helpers
>>> from scipy import constants
>>> import numpy as np
```

pyhonradex needs a file containing atomic data. Download the file for CO from the LAMDA database. We need to tell pyhonradex where the file is located:

```
>>> data_filepath = 'path/to/file/co.dat'
```

We need to define the geometry of the nebula. Let's consider a uniform sphere:

```
>>> geometry = 'uniform sphere'
```

We need to set the kinetic temperature, total colum density, line profile type and width of the emission lines in velocity space:

```
>>> Tkin = 150
>>> Ntot = 1e16/constants.centi**2
>>> line_profile = 'square'
>>> width_v = 2*constants.kilo
```

Next, we need to tell pyhonradex the density of the collision partner(s). This is implemented as a dictionary. For example, let's assume that the density of ortho- and para-H_2 is 100 cm^{-3} and 250 cm^{-3} respecitvely:

```
>>> coll_partner_densities = {'para-H2':100/constants.centi**3,
                              'ortho-H2':250/constants.centi**3}
```

Finally, we need to define the background radiation field. The CMB is already defined in the helpers module, so one could simply do:

```
>>> ext_background = helpers.CMB_background
```

But a custom background field is also possible. For example, let's assume we want to add the radiation from a star that is 100 au from the cloud, has an effective temperature of 6000 K and the same radius as the Sun. We would then define:

```
>>> R_Sun = 6.957e8
>>> T_Sun = 6000
>>> cloud_star_distance = 100*constants.au
>>> star_solid_angle = R_Sun**2*np.pi/cloud_star_distance**2

>>> def star_and_CMB_background(nu):
        I_star = helpers.B_nu(nu=nu,T=T_Sun)*star_solid_angle/(4*np.pi)
        return I_star + helpers.CMB_background(nu)
```

```
>>> ext_background = star_and_CMB_background
```

where the Planck function defined in the helpers module is used. For no background, a custom function is defined in the helpers module:

```
>>> ext_backbround = helpers.zero_background
```

Now we can initialise the object that is used to solve the radiative transfer:

```
>>> example_nebula = nebula.Nebula(
                    data_filepath=data_filepath,geometry=geometry,
                    ext_background=ext_background,Tkin=Tkin,
                    coll_partner_densities=coll_partner_densities,
                    Ntot=Ntot,line_profile=line_profile,width_v=width_v)
```

To solve, simply do:

```
>>> example_nebula.solve_radiative_transfer()
```

To print out the results to the terminal, you can do:

```
>>> example_nebula.print_results()
    up   low      nu [GHz]    T_ex [K]      poplow         popup         tau_nu0
    1    0      115.271202     14.89      0.240349      0.497337      0.563936
    2    1      230.536998      8.15      0.497337      0.213311       1.86095
    3    2      345.798390      8.15      0.213311      0.038966       0.84055
    4    3      461.040389     11.53      0.038966    0.00734724      0.143254
    5    4      576.265992     17.68    0.00734724    0.00187956     0.0242841
    6    5      691.475202     24.03    0.00187956   0.000558413    0.00576143
    7    6      806.650034     29.21   0.000558413   0.000171229    0.00165335
    ...
```

Here, ‘up’ and ‘low’ are the indices of the upper and lower level of the transition respectively (0 for the lowest level), ‘nu’ is the frequency, ‘T_ex’ the excitation temperature, ‘poplow’ and ‘popup’ the fracitional populations of the lower and upper level respectively, and ‘tau_nu0’ the optical depth at the line centre.

The nebula object has now a number of attributes that contain the result of the calculation. For example, to access the excitation temperature of the third transition (as listed in the LAMDA datafile; note that the first index is 0), you can do:

```
>>> example_nebula.Tex[2]
    8.1489880206102789
```

Similarly, for the fractional population of the 4th level, do:

```
>>> example_nebula.level_pop[3]
    0.038965991285387587
```

And for the optical depth of the lowest transition:

```
>>> example_nebula.tau_nu0[0]
    0.56393648003569496
```

Now we want to calculate the flux recorded by the telescope. Define the distance of the cloud and its surface:

```
>>> d_observer = 20*constants.parsec
>>> source_radius = 3*constants.au
>>> source_surface = 4/3*source_radius**3*np.pi
```

Then calculate the observed fluxes:

```
>>> obs_fluxes = example_nebula.observed_fluxes(
                    source_surface=source_surface,d_observer=d_observer)
```

This returns a list with the flux for each line in W/m^2. To get the flux of the second transition:

```
>>> obs_fluxes[1]
    1.6189817656980213e-11
```

Reading a file from the LAMDA database

pythonradex also provides a useful function to read data files from the LAMDA (http://home.strw.leidenuniv.nl/~moldata/) database. Let's see how it can be used:

```
>>> from pythonradex import LAMDA_file
>>> data_filepath = 'path/to/datafile/co.dat'
>>> data = LAMDA_file.read(data_filepath)
```

The data is stored in a dictionary containing all levels, radiative transitions and collisional transitions.:

```
>>> levels = data['levels']
>>> rad_transitions = data['radiative transitions']
>>> coll_transitions = data['collisional transitions']
```

Lets first look at the levels. This is a list containing all atomic energy levels (as instances of the `Level` class, see :ref:'rad_trans_doc') listed in the file. It is ordered the same way as in the file. Let's access the statistical weight, energy, and number of the 3rd level as an example (note that the index is 0 based):

```
>>> levels[2].g
5.0
>>> levels[2].E
2.2913493542995655e-22
>>> levels[2].number
2
```

Similarly, the radiative transitions are stored in a list, also ordered as they appear in the file. Each element of the list is an instance of the `RadiativeTransition` class (see *Reading LAMDA files*). Let's see how many radiative transitions there are:

```
>>> len(rad_transitions)
40
```

Let's look at a random transition:

```
>>> rad_trans = rad_transitions[10]
```

We can access the upper and lower level of the transition. These are instance of the `Level` class:

```
>>> rad_trans.up.g
23.0
>>> rad_trans.low.E
4.1994278867414716e-21
```

Let's look at some of the other attributes of this transition such as frequency, energy difference and Einstein coefficients:

```
>>> rad_trans.nu0
1267014531042.1921
>>> rad_trans.Delta_E
8.3953270243833185e-22
>>> rad_trans.A21
0.0001339
```

For a list of all attributes available, see *Reading LAMDA files*. We can also compute the excitation temperature of the transition for given fractional populations of the lower and upper level:

```
>>> rad_trans.Tex(x1=0.3,x2=0.1)
array(51.11629261333541)
```

One can also give numpy arrays as input:

```
>>> import numpy as np
>>> x1 = np.array((0.1,0.5,0.15))
>>> x2 = np.array((0.05,0.1,0.07))
>>> rad_trans.Tex(x1=x1,x2=x2)
array([ 77.54834464,  35.76028035,  71.27685386])
```

Finally, let's have a look at the collisional transitions. This is a dictionary containing the transitions for each collision partner. Let's see which collision partners are present:

```
>>> coll_transitions.keys()
dict_keys(['ortho-H2', 'para-H2'])
```

Let's look at collisions with ortho-H_2. This is a list with instances of the `CollisionalTransition` class (see *Reading LAMDA files*). How many collisional transitions are there? Let's see:

```
>>> len(coll_transitions['ortho-H2'])
820
```

Similarly to the radiative transition, there are a number of attributes we can access:

```
>>> coll_trans = coll_transitions['ortho-H2'][99]
>>> coll_trans.up.number
14
>>> coll_trans.low.g
17.0
>>> coll_trans.Delta_E
5.26548816268121e-21
>>> coll_trans.name
'14-8'
```

Again, see *Reading LAMDA files* to get all attributes. Like for radiative transitions, one can calculate the excitation temperature. In addition, one can get the collisional transition rates. The LAMDA data file provides these rates at specific temperatures. Here we can request an interpolated rate at any temperature within the limits defined in the file:

```
>>> coll_trans.coeffs(Tkin=100.5)
{'K21': 6.4715447026880032e-18, 'K12': 2.4825283934065823e-19}
```

Numpy arrays are also allowed as input:

```
>>> Tkin = np.array((52.3,70.4,100.2,150.4))
>>> coll_trans.coeffs(Tkin=Tkin)
{'K21': array([5.93161938e-18, 6.13652817e-18, 6.46702134e-18, 7.11359510e-18]),
 'K12': array([6.88961220e-21, 4.64695547e-20, 2.45276671e-19, 9.61109445e-19])}
```

CHAPTER

FOUR

RADIATIVE TRANSFER THEORY

Basics

We briefly discuss the basics theory of radiative transfer that is relevant for `pyhonradex`. A more detailed discussion can for example be found in *[Rybicki04]*.

The radiation field in every point of space can be described by the specific intensity I_ν, defined as the energy radiated per unit of time, surface, frequency and solid angle, i.e., I_ν has units of W/m^2/sr/Hz. The differential equation describing the change of the specific intensity along a spatial coordinate s is given by

$$\frac{\mathrm{d}I_\nu}{\mathrm{d}s} = -\alpha_\nu I_\nu + j_\nu$$

Here, α_ν is the absorption coefficient in m^{-1}. It describes how much energy is removed from the beam per unit length. On the other hand, the emission coefficient j_ν is the energy emitted per unit time per unit solid angle per unit volume. The subscript ν reminds the reader that the quantities are given per unit of frequency. Defining the optical depth as $\mathrm{d}\tau_\nu = \alpha_\nu \mathrm{d}s$, we can rewrite the equation as

$$\frac{\mathrm{d}I_\nu}{\mathrm{d}\tau_\nu} = -I_\nu + S_\nu$$

with the source function $S_\nu = \frac{j_\nu}{\alpha_\nu}$. In general, the goal of radiative transfer is to solve this equation. Fro example, for a uniform medium (the emission and absorption coefficients are the same everywhere) as assumed for `pythonradex`, the solution reads $I_\nu = I_\nu(0)e^{-\tau_\nu} + S_\nu(1 - e^{-\tau_\nu})$.

Gas emission

Now let's consider radiation from a gas. An atom can spontaneously emit a photon when it transits from an upper to a lower energy level. The transition probability per unit time is given by the Einstein coefficient for spontaneous emission, A_{21}, in units of s^{-1}. Thus, we can write the emission coefficient of the gas as.

$$j_\nu = \frac{h\nu_0}{4\pi} n_2 A_{21} \phi_\nu$$

where h is the Planck constant, ν_0 is the central frequency of the transition, n_2 is the number density of atoms in the upper level of the transition and ϕ_ν is the normalised line profile (i.e. $\int \phi_\nu \mathrm{d}\nu = 1$ and ϕ_ν describes how the energy is distributed over frequency), for example a Gaussian. Photons can also be absorbed with a transition from a lower to an upper energy level. This process is parametrised by the Einstein B_{12} coefficient. Defining the mean intensity J_ν as the follwing mean over solid angle:

$$J_\nu = \frac{1}{4\pi} \int I_\nu \mathrm{d}\Omega$$

and $\bar{J} = \int J_\nu \phi_\nu \mathrm{d}\nu$, the transition probability per unit time for absorption is $B_{12}\bar{J}$, i.e. the transition rate is proportional to the flux of incoming photons. There is a third process, called stimulated emission, that results in the *emission* of a photon (and a transition from an upper to a lower level) when the atom interacts with an incoming photon. The probability per unit time for stimulated emission $B_{21}\bar{J}$. Thus, we can write the absorption coefficient as

$$\alpha_\nu = \frac{h\nu_0}{4\pi}(n_1B_{12} - n_2B_{21})\phi_\nu$$

with n_1 the number density of atoms in the lower level. Thus, stimulated emission is treated as 'negative absorption'.

Therefore, in order to compute the emission and absorption coefficients and solve the radiative transfer equation, we need to know the level population, i.e. the fraction of atoms occupying the different energy levels. Actually, using relations between the Einstein coefficients *[Rybicki04]*, it can easily be shown that the source function can be written as

$$S_\nu = B_\nu(T_\mathrm{ex})$$

where $B_\nu(T)$ is the blackbody radiation field (Planck's law) and T_ex is the excitation temperature, defined as

$$\frac{n_2}{n_1} = \frac{g_2}{g_1}\exp\left(-\frac{h\nu_0}{kT_\mathrm{ex}}\right)$$

with k the Boltzmann constant and g_1 and g_2 the statistical weights of the lower and upper level respectively. The excitation temperature is thus the temperatue we need to plug into the Boltzmann equation to get the observed level ratio. In LTE, the levels are indeed populated according to the Boltzmann distribution. But in non-LTE, this is not true, and the excitation temperature is not equal to the kinetic temperature.

In summary, we need to know the level populations (i.e. the excitation temperature) in order to compute the radiation field. To do this, we first need to consider another process that can populate and depopulate the energy levels of an atom: collisions, for example with hydrogen or electrons. The rate of collision-induced transitions between two levels i and j is given by $C_{ij} = K_{ij}(T_\mathrm{kin})n_\mathrm{col}$ where $K_{ij}(T_\mathrm{kin})$ is the collision rate coefficient in m^{3}s^{-1} and n_col is the number density of the collision partner. The collision rate coefficient in general depends on the kinetic temperature of the collision partner. If several collision partners are present, the total rate is simply the sum of the individual rates.

We can now write down the equations of statistical equilibrium (SE) that determine the level population. In SE, we assume that processes that populate a level are balanced by processed depopulate it. Thus, for every level i, we write

$$\frac{\mathrm{d}x_i}{\mathrm{d}t} = \sum_{j>i}(x_jA_{ji} + (x_jB_{ji} - x_iB_{ij})\bar{J}_{ji}) - \sum_{j<i}(x_iA_{ij} + (x_iB_{ij} - x_jB_{ji})\bar{J}_{ij}) + \sum_{j\neq i}(x_jC_{ji} - x_iC_{ij}) = 0$$

where $x_k = \frac{n_k}{n}$ is the fractional population of level k. In the above equation, the positive terms populate the level, while the negative terms depopulate the level.

The level populations can be computed by solving this linear system of equations. But there is a problem: we see that to solve for the level populations, we need to know the radiation field $\bar{J}$. This is a fundamental issue in radiative transfer: to compute the radiation field, we need to know the level population. But in order to compute the radiation field, we need to know the level populations.

Escape probability

One way to solve this problem is to use an escape probability method to decouple the computation of the level population from the computation of the radiation field. We consider the probability β of a newly created photon to escape the cloud. This probability depends on the geometry of the cloud and the optical depth. In a completely opaque case, we expect the radiation field to equal the source function. Thus, we write $J_\nu = (1-\beta(\tau_\nu))S_\nu = (1-\beta(\tau_\nu))B_\nu(T_\mathrm{ex})$. If we plug the corresponding expression for $\bar{J}$ into the SE equations, they become independent of the radiation field and can be solved, because τ_ν and T_ex only depend on the level population.

In practice, an iterative approach is used to solve the SE equations: one makes a first guess of the level populations and computes the corresponding escape probability, which is used to compute a new solution of the SE equations. This is repeated until convergence. Finally, the converged level population is used to compute the emitted flux and the radiative transfer problem is solved.

An external radiation field I_{ext} can also contribute to the excitation of the atoms. This is easily incorporated in the calculation by adding a term βI_{ext} to J_ν.

Let's consider the example of a uniform sphere. The escape probability for this geometry is calculated in *[Osterbrock74]* and given by

$$\beta(\tau_\nu) = \frac{3}{2\tau_\nu}\left(1 - \frac{2}{\tau_\nu^2} + \left(\frac{2}{\tau_\nu} + \frac{2}{\tau_\nu^2}\right)e^{-\tau_\nu}\right)$$

where τ_ν is the optical depth of the diameter of the sphere. The flux at the surface of the sphere in [W/m^2/Hz] is given by (see again *Osterbrock74*)

$$F = 2\pi\frac{B_\nu(T_{\text{ex}})}{\tau_\nu^2}\left(\frac{\tau_\nu^2}{2} - 1 + (\tau_\nu + 1)\,e^{-\tau_\nu}\right)$$

Accelerated Lamda Iteration (ALI)

The task of computing the mean radiation field knowing the source function is generally represented with a 'Lambda Operator' like this:

$$J_\nu = \Lambda(S_\nu)$$

We mentioned earlier that the SE equations are solved iteratively. This approach falls into the class of Lamda Iteration methods. A first guess of the level population (which determines S_ν) is made. The Λ operator allows to compute J_ν. From this, an updated S_ν can be determined. And so on, until convergence is reached. In our case, the Lambda operator is given by

$$J_\nu = \Lambda(S_\nu) = \beta(S_\nu)I_{\text{ext}} + (1 - \beta(S_\nu))S_\nu$$

where I_{ext} is the external field. However, the Lambda Iteration method is known to converge extremely slowly at large optical depth. In fact, one can easily be fooled to think that convergence is reached, while in reality one is far from convergence. Without going into detail, the basic reason is that the number of iterations corresponds to the number scattering events that are treated. For large optical depth, a photon is scattered many times before it exits the cloud. Thus, many iterations are necessary.

There is a method to circumvent this problem known as Accelerated Lambda Iteration (ALI). Details can be found in *Rybicki91* and the lectures notes of Dullemond (http://www.ita.uni-heidelberg.de/~dullemond/lectures/radtrans_2012/), sections 4.4 and 7.8–7.10. The basic idea is to decompose the Lambda opertor like this:

$$\Lambda = \Lambda^* + (\Lambda - \Lambda^*)$$

and then iterate using the $(\Lambda - \Lambda^*)$, while keeping the Λ^* out of the iteration scheme. The simplest choice for Λ^* is the *local* part, of the operator. In 3D, Λ can be represented as a matrix, and the local part would be the diagonal, corresponding to the self-coupling of each grid cell. Loosly speaking, by splitting out the local contribution, photons that scatter within the same grid cell are not considered in the iteration, resulting in considerably faster convergence. This is reasonable since such we are not interested in what photons do at sub-cell resolution.

In our case, the ALI scheme is found by inserting the expression for J_ν into the SE equations. By expressing the source function in terms of the Einstein coefficients, one finds

$$\frac{\mathrm{d}x_i}{\mathrm{d}t} = \sum_{j>i}(x_jA_{ji}\beta + (x_jB_{ji} - x_iB_{ij})\beta I_{\text{ext}}) - \sum_{j<i}(x_iA_{ij}\beta + (x_iB_{ij} - x_jB_{ji})\beta I_{\text{ext}}) + \sum_{j\neq i}(x_jC_{ji} - x_iC_{ij}) = 0$$

These are the equations that `pythonradex` iteratively solves.

Difference between `pythonradex` and `RADEX`

There is a difference between the outputs of `RADEX` and `pythonradex`. The `RADEX` output T_R (or the corresponding flux outputs) is intended to be directly compared to telescope data. To be more specifc, from the computed optical depth and excitation temperature, `RADEX` first computes $I_{\mathrm{tot}} = B_\nu(T_{\mathrm{ex}})(1-e^{-\tau}) + I_{\mathrm{bg}}e^{-\tau}$, i.e. the total intensity at the line centre that is recorded at the telescope, where I_{bg} is the background radiation. This is the sum of the radiation from the gas (first term) and the background radiation attenuated by the gas (second term). From this, the observer will subtract the background (or, in other words, the continuum), giving $I_{\mathrm{measured}} = I_{\mathrm{tot}} - I_{\mathrm{bg}} = (B_\nu(T_{\mathrm{ex}}) - I_{\mathrm{bg}})(1-e^{-\tau})$. The `RADEX` output T_R is the Rayleigh-Jeans temperature corresponding to I_{measured}. On the other hand, `pythonradex` computes the line flux directly, i.e. the output corresponds simply to the photons coming from the gas alone. In general, one can directly compare the amount of observed 'gas photons' to I_{measured} as long as $B_\nu(T_{\mathrm{ex}}) \gg I_{\mathrm{bg}}$.

CHAPTER

FIVE

DETAILED DOCUMENTATION OF `PYHONRADEX`

Radiative transfer

The core of `pyhonradex` is the Nebula class which is used to solve the radiative transfer.

class `pythonradex.nebula.`**`Nebula`**(*data_filepath*, *geometry*, *ext_background*, *Tkin*, *coll_partner_densities*, *Ntot*, *line_profile*, *width_v*, *verbose=False*)

Represents an emitting gas cloud

- **emitting_molecule: EmittingMolecule** An object containing atomic data and line profile information
- **geometry: str** geometry of the gas cloud
- **ext_background: func** function returning the external background radiation field for given frequency
- **Tkin: float** kinetic temperature of colliders
- **coll_partner_densities: dict** densities of the collision partners
- **Ntot:** total column density
- **rate_equations: RateEquations** object used to set up and solve the equations of statistical equilibrium
- **verbose: bool** if True, additional information is printed out

The following attributes are available after the radiative transfer has been solved:

- **tau_nu0: numpy.ndarray** optical depth of each transition at the central frequency.
- **level_pop: numpy.ndarray** fractional population of levels.
- **Tex: numpy.ndarray** excitation temperature of each transition.

`__init__`(*data_filepath*, *geometry*, *ext_background*, *Tkin*, *coll_partner_densities*, *Ntot*, *line_profile*, *width_v*, *verbose=False*)

data_filepath: str path to the LAMDA data file that contains the atomic data

geometry: str geometry of the gas cloud. Currently available are “uniform sphere”and “uniform sphere RADEX”. The latter uses the forumla for a uniform sphere for the escape probability and the formula for a uniform slab to calculate the flux, as in RADEX.

ext_background: func The function should take the frequency in Hz as input and return the background radiation field in [W/m2/Hz/sr]

Tkin: float kinetic temperature of the colliders

coll_partner_densities: dict number densities of the collision partners in [1/m3]. Following keys are recognised: “H2”, “para-H2”, “ortho-H2”, “e”, “H”, “He”, “H+”

Ntot: float total column density in [1/m2]

line_profile: str type of line profile. Available are “Gaussian” and “square”.

width_v: float width of the line in [m/s]. For Gaussian, this is the FWHM.

verbose: bool if True, additional information is printed out

`observed_fluxes`(*source_surface*, *d_observer*)
Compute the flux recorded at the telescope. Can only be called if the radiative transfer has been solved.

source_surface: float the surface of the emitting cloud in [m2]

d_observer: the distance between observer and emitting cloud in [m]

numpy.ndarray the flux in W/m2 seen by the observer, for each radiative transition

`print_results`()
print out the results from the radiative transfer computation. Can only be called if the radiative transfer has been solved.

`solve_radiative_transfer`()
Solves the radiative transfer by iterating and initialises new attributes that contain the solution.

Reading LAMDA files

`pyhonradex` provides a convenient function in the `LAMDA_file` module to read files from the LAMDA database:

`pythonradex.LAMDA_file.read`(*datafilepath*)
Read a LAMDA data file.

Reads a LAMDA data file and returns the data in the form of a dictionary. The LAMDA database can be found at http://home.strw.leidenuniv.nl/~moldata/molformat.html

datafilepath [str] path to the file

dict Dictionary containing the data read from the file. The dictionary has the following keys:

- ‘levels’: list of levels (instances of the Level class)
- ‘radiative transitions’: list of radiative transitions (instances of RadiativeTransition class)
- ‘collisional transitions’: dict, containing lists of instances of the CollisionalTransition class for each collision partner appearing in the file

The elements of these lists are in the order they appear in the file

The data is returned using the following classes:

class `pythonradex.atomic_transition.Level`(*g*, *E*, *number*)
Represents an atomic/molecular energy level

•**g: float** statistical weight

•**E: float** energy in [J]

•**number: int** the level number (0 for the lowest level)

class `pythonradex.atomic_transition.RadiativeTransition`(*up*, *low*, *A21*)
Represents the radiative transition between two energy levels

•**up: Level** upper level

•**low: Level** lower level

•**Delta_E: float** energy difference between upper and lower level

•**name: str** transition name, for example ‘3-2’ for the transition between the fourth and the third level

•**A21: float** Einstein A21 coefficient

- **nu0: float** central frequency of the transition
- **B21: float** Einstein B21 coefficient
- **B12: float** Einstein B12 coefficient

Tex (*x1*, *x2*)
Excitation temperature

Computes the excitation temperature from the fractional population

x1: array_like fractional population of the lower level

x2: array_like fractional population of the upper level

numpy.ndarray excitation temperature in K

class pythonradex.atomic_transition.**CollisionalTransition** (*up*, *low*, *K21_data*, *Tkin_data*)
Represent the collisional transtion between two energy levels

- **up: Level** upper level
- **low: Level** lower level
- **Delta_E: float** energy difference between upper and lower level
- **name: str** transition name, for example '3-2' for the transition between the fourth and the third level
- **K21_data: numpy.ndarray** value of the collision rate coefficient K21 at different temperatures
- **log_K21_data: numpy.ndarray** the logarithm of K21_data
- **Tkin_data: numpy.ndarray** the temperature values corresponding to the K21 values
- **log_Tkin_data: numpy.ndarray** the logarithm of the temperature values
- **Tmax: float** the maximum temperature value
- **Tmin: float** the minimum temperature value

Tex (*x1*, *x2*)
Excitation temperature

Computes the excitation temperature from the fractional population

x1: array_like fractional population of the lower level

x2: array_like fractional population of the upper level

numpy.ndarray excitation temperature in K

coeffs (*Tkin*)
collisional transition rates

computes the collisional transition rate coefficients by interpolation.

Tkin: array_like kinetic temperature in K. Must be within the interpolation range.

dict The keys "K12" and "K21" of the dict are the requested collision coefficients

helpers module

The `helpers` module provides a number of convenience functions, some of which might be of interest to the user.

`pythonradex.helpers.`**`B_nu`**`(`*nu*, *T*`)`

Planck function

Return the value of the Planck function (black body) in [W/m2/Hz/sr].

nu [float or numpy.ndarray] frequency in Hz

T [float or numpy.ndarray] temperature in K

numpy.ndarray Value of Planck function in [W/m2/Hz/sr]

`pythonradex.helpers.`**`CMB_background`**`(`*nu*`)`

CMB background

Computes the CMB background radiation

nu: float or numpy.ndarray frequency in Hz

numpy.ndarray CMB background radiation intensity in [W/m2/Hz/sr]

`pythonradex.helpers.`**`zero_background`**`(`*nu*`)`

Zero intensity radiation field

Returns zero intensity for any frequency

nu: array_like frequency in Hz

numpy.ndarray Zero at all requested frequencies

`pythonradex.helpers.`**`FWHM2sigma`**`(`*FWHM*`)`

Convert FWHM of a Gaussian to standard deviation.

FWHM: float or numpy.ndarray FWHM of the Gaussian

float or numpy.ndarray the standard deviation of the Gaussian

CHAPTER

SIX

ACKNOWLEDGEMENT:

I would like to thank Simon Bruderer for his helpful clarifications about the ALI method.

[Osterbrock74] Osterbrock, D. E. 1974, *Astrophysics of Gaseous Nebulae*, ISBN 0-716-70348-3, W. H. Freeman

[Rybicki91] Rybicki, G. B., & Hummer, D. G. 1991, Astronomy and Astrophysics, 245, 171

[Rybicki04] Rybicki, G. B., & Lightman, A.P. 2004, *Radiative Processes in Astrophysics*, ISBN 0-471-82759-2, Wiley-VCH

[vanderTak07] van der Tak, F. F. S., Black, J. H., Schöier, F. L., Jansen, D. J., & van Dishoeck, E. F. 2007, Astronomy and Astrophysics, 468, 627