General purpose PyNeb's Manual

C. Morisset, V. Luridiana, G. Delgado-Inglada. Updated 15/04/2013

A) Introduction

PyNeb is a package for the analysis of emission lines, which evolved from the IRAF package **nebular** (Shaw and Dufour 1995; Shaw et al 1998). Please note that the name of the code and the tarball are CapitalCase (PyNeb, *PyNeb-<version-number>*), whereas the module and directories are all lowercase (*pyneb*, *pynebcore.py*, etc).

PyNeb is a library rather than a code. This means that to use it, one has to at least write a python script or enter an interactive python session to call PyNeb classes and/or functions. In the following we'll provide some examples on how to use the different classes and functions.

In this manual, python commands are identified by this style:

```
    print('this is a python command')
```

Outputs of python commands will start on the first column:

```
this is a python command
```

The (Linux) shell commands use this style:

```
echo I m on Linux
```

References

Whenever you use PyNeb for calculations that lead to a published paper, you are kindly asked to cite the code as:

PyNeb: a new software for the analysis of emission lines

Luridiana, V., Morisset, C., & Shaw, R. A.

2012, *IAU Symposium*, **283,** 422-423

This paper is accessible HERE.

Discussion group

There is a discussion group where you can post your requests, help other users, share your problems (PyNeb-related only :)) at the following URL:

https://groups.google.com/forum/#!forum/pyneb

The email address to send messages to the group (the best way to get a quick answer to a problem) is pyneb@googlegroups.com

B) Installation

1. Requirements

To run PyNeb, you must have python v. 2.6 or above installed (but ***not*** python v. 3.n, which is a different python branch), plus the *matplotlib* , *numpy* and *scipy* libraries. Numpy minimum version is 1.5.1.

To know which your default python version is, enter from command line:

```
python --version
```

To know whether you have *numpy* and *matplotlib* installed, enter python by typing either *python* or *ipython*, then type:

```
import matplotlib
import numpy
```

You'll access the version number this way:

```
print(matplotlob.__version__)
print(numpy.__version__)
```

2. *Installing the code with pip*

This is the easiest way to install or upgrade PyNeb. You'll need **pip** installed; go here to get it:

http://pypi.python.org/pypi/pip. You can install pip using:

```
sudo easy_install pip
```

or following the instructions on the pip web page. Once pip is installed, enter from the command line:

```
pip install –user pyneb
```

to install PyNeb and:

```
pip install –upgrade –user pyneb
```

for any upgrade. Uninstall PyNeb is easy as well:

```
pip uninstall pyneb
```

3. *Installing the code from tar*

Download the PyNeb package from the pypi site. Unpack the PyNeb tarball in the directory of your choice, enter the directory PyNeb-<version-number>, which is PyNeb's root directory and contains the following:

- *setup.py* (a file)
- *pyneb* (a directory)
- *PKG-INFO* and *README.txt* (information files)

setup.py is the package installer. To install the package, enter:

```
python setup.py install --user
```

Once the package is installed, simply enter your favorite python interface (e.g., *python* or *ipython*) and start using PyNeb by inputting:

```
import pyneb as pn
```

(the "as pn" bit simply establishes a nickname for pyneb). Enjoy!

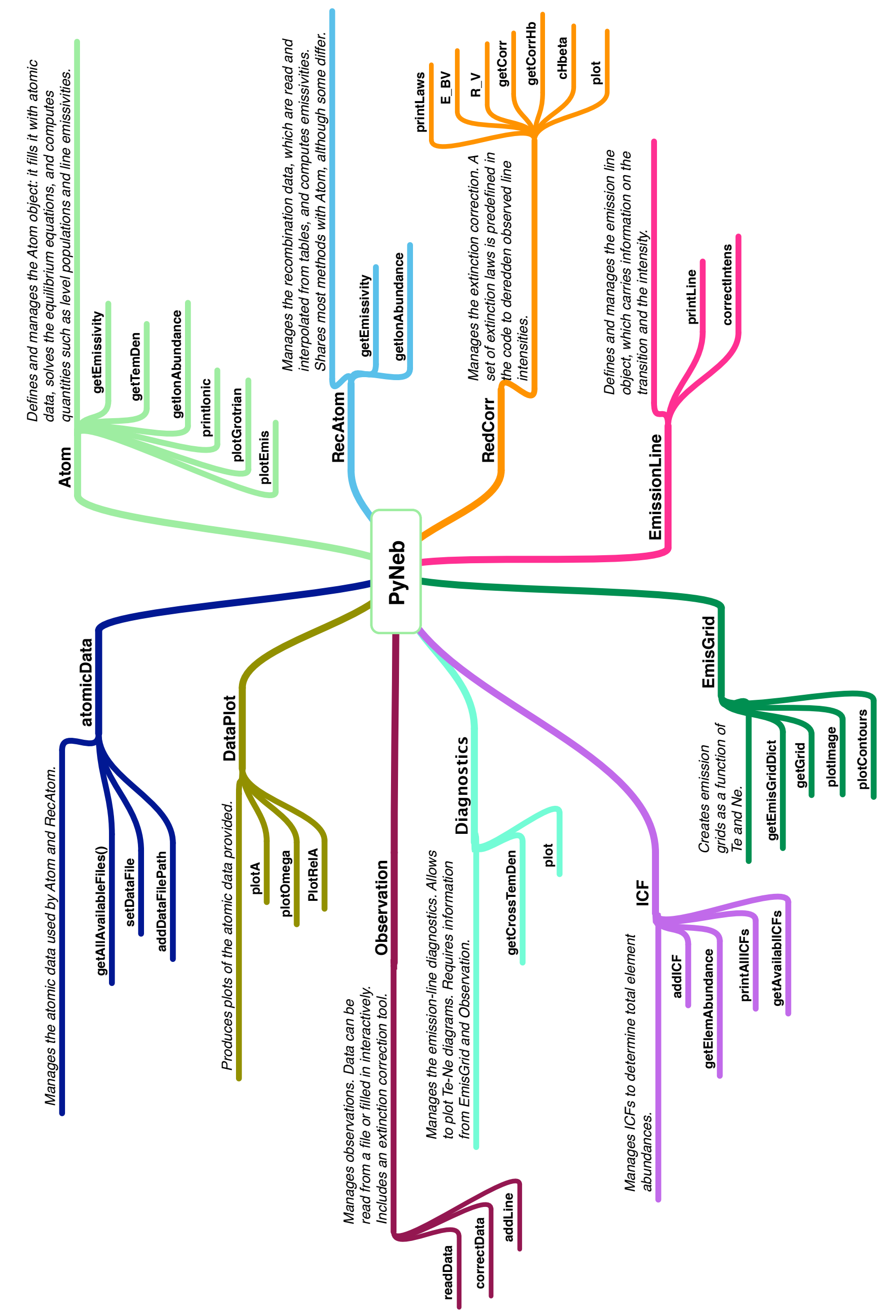
You can know which version of PyNeb you are using by:

```
print(pn.__version__)
```

C) First steps using PyNeb

You can run simple PyNeb commands from the command line, but most of the time, you'll need to import pyNeb as a package imported in python scripts. In the following some examples of the use of PyNeb are given, that can be adapted to your needs.

A mental map of PyNeb can help to understand which class plays which role:


PyNeb
Atom
Defines and manages the Atom object: it fills it with atomic data, solves the equilibrium equations, and computes quantities such as level populations and line emissivities.
getEmissivity
getTemDen
getIonAbundance
printIonic
plotGrotrian
plotEmis
RecAtom
Manages the recombination data, which are read and interpolated from tables, and computes emissivities. Shares most methods with Atom, although some differ.
getEmissivity
getIonAbundance
RedCorr
Manages the extinction correction. A set of extinction laws is predefined in the code to deredden observed line intensities.
printLaws
E_BV
R_V
getCorr
getCorrHb
cHbeta
plot
EmissionLine
Defines and manages the emission line object, which carries information on the transition and the intensity.
printLine
correctIntens
EmisGrid
Creates emission grids as a function of Te and Ne.
getEmisGridDict
getGrid
plotImage
plotContours
ICF
Manages ICFs to determine total element abundances.
addICF
getElemAbundance
printAllICFs
getAvailablICFs
Diagnostics
Manages the emission-line diagnostics. Allows to plot Te-Ne diagrams. Requires information from EmisGrid and Observation.
getCrossTemDen
plot
Observation
Manages observations. Data can be read from a file or filled in interactively. Includes an extinction correction tool.
readData
correctData
addLine
DataPlot
Produces plots of the atomic data provided.
plotA
plotOmega
PlotRelA
atomicData
Manages the atomic data used by Atom and RecAtom.
getAllAvailableFiles()
setDataFile
addDataFilePath

The main classes are Atom, RecAtom, EmisGrid, Observations, and Diagnostics. Some classes use other classes, for example Atom is used in almost all the other classes. In the following a detailed description of each class is given, and the exhaustive list of all the methods of all the classes is available from the reference manual.

PyNeb uses the Object Oriented (OO) method of programming to define most of its features, but the scripts written as PyNeb wrappers need not be OO; they may follow the more classical functional or structured programming style.

D) The Atom class

1. *Description*

The first class you will need to manage is Atom. Atom contains the way atomic data are read and managed to compute for example line emissivities and to determine diagnostics from line ratios, as well as ionic abundances.

In the following, we will always assume that the PyNeb package has been imported under the alias “pn”:

```
import pyneb as pn
```

Instantiation of the Atom class to create an object is done by specifying at least an ion.

```
O3 = pn.Atom('O', 3)
```

This command tells the code to create an O III atom (corresponding to the O++ ion) model by reading the OIII atomic data. As will be explained in this Section, the data are stored in two kinds of fits files, the *atom* files and the *coll* files. You can create (instantiate) as many atoms as you need, or even create atoms for the same ion, but using different atomic data. You can check the feature of a particular Atom object by printing it:

```
print(O3)
```

```
Atom O3 from o_iii_atom_WFD96.fits and o_iii_coll_LB94.fits
```

The Atom object contains methods (functions) to explore it; for example you can display the ion's intrinsic properties:

```
O3.name
O3.spec
O3.gs
```

or the value of the data you are using, such as the energy of the fourth level in eV:

```
O3.getEnergy(4, unit='eV')
```

The complete inventory of features and methods of the Atom class can be displayed by entering:

```
O3.
```

(with the dot) followed by the TAB key(only from an ipython session). The methods can be either public or private and are described in the PyNeb Reference Manual. The private methods have names starting with underscore “_”. They are not supposed to be used by “normal” users.

2. *Print and plot atomic properties*

You can use the following methods to obtained line emissivities:

```
O3.printIonic()
```

Some arguments can be passed to the method to obtain more information:

```
O3.printIonic(tem=10000., den=1e3, printA=True, printPop=True, printCrit=True)
```

the result of the latter being the following:

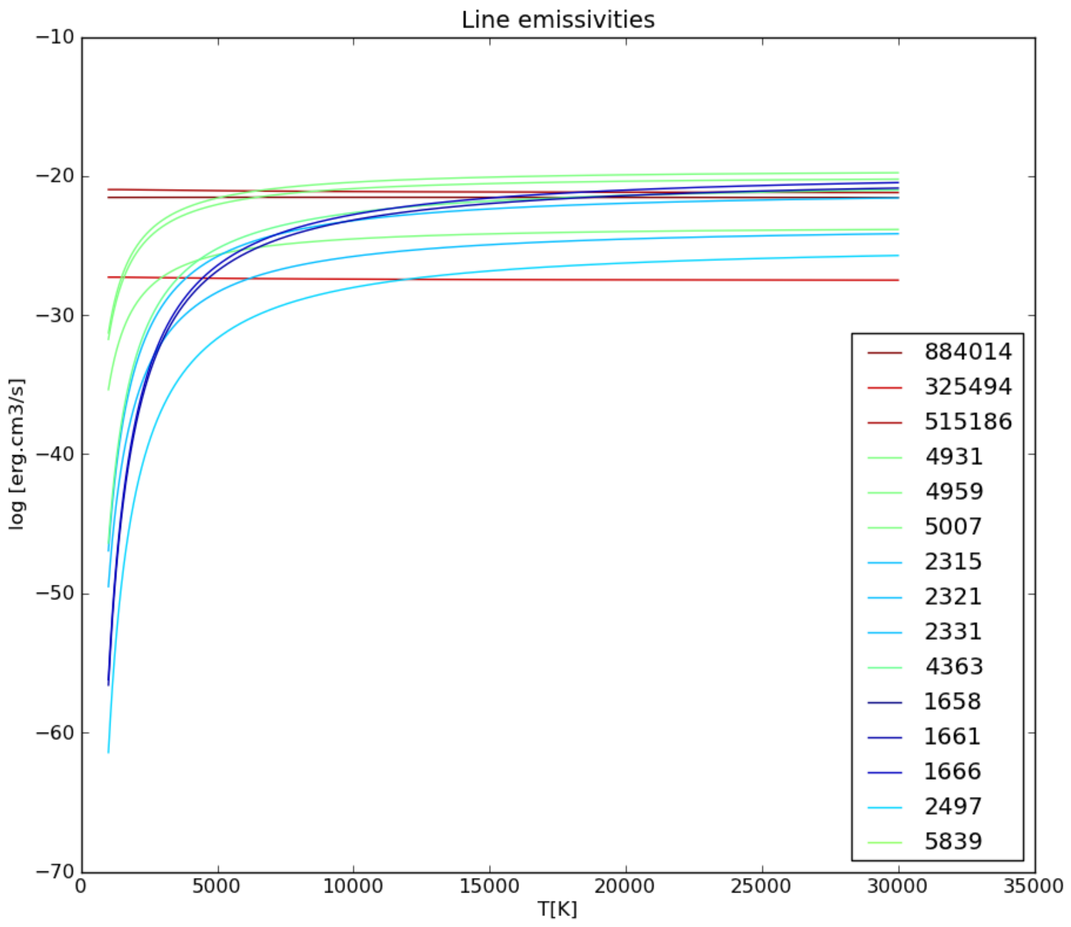
```
atom =  O
```

```
spec = 3
temperature = 10000.0 K
density = 1000.0 cm-3

Level   Populations  Critical densities
Level 1:  3.096E-01  0.000E+00
Level 2:  4.876E-01  5.044E+02
Level 3:  2.028E-01  3.449E+03
Level 4:  4.408E-05  6.884E+05
Level 5:  3.205E-09  2.396E+07
Level 6:  2.584E-12  3.718E+10

2.664E-05
    88.40m
   (2-->1)
  2.919E-22

3.094E-11    9.695E-05
    32.55m       51.52m
   (3-->1)     (3-->2)
  3.829E-28    7.579E-22

1.690E-06    6.995E-03    2.041E-02
  4930.99A     4958.65A     5006.84A
   (4-->1)     (4-->2)     (4-->3)
  3.001E-25    1.235E-21    3.570E-21

0.000E+00    2.268E-01    6.091E-04    1.561E+00
  2314.88A     2320.96A     2331.46A     4363.21A
   (5-->1)     (5-->2)     (5-->3)     (5-->4)
  0.000E+00    6.222E-24    1.663E-26    2.278E-23

0.000E+00    2.200E+02    5.480E+02    4.740E-03    0.000E+00
  1657.66A     1660.77A     1666.14A     2497.12A     5838.63A
   (6-->1)     (6-->2)     (6-->3)     (6-->4)     (6-->5)
  0.000E+00    6.800E-24    1.688E-23    9.744E-29    0.000E+00

# H-beta volume emissivity:
1.237E-25 N(H+) * N(e-)  (erg/s)
```

Each bock corresponds to a transition and it gives the transition probability (in s-1), the wavelength of the corresponding emission line (in Angstrom or microns), the two levels of the transition and the emissivity of the line.

You can plot the line emissivities using the following command:

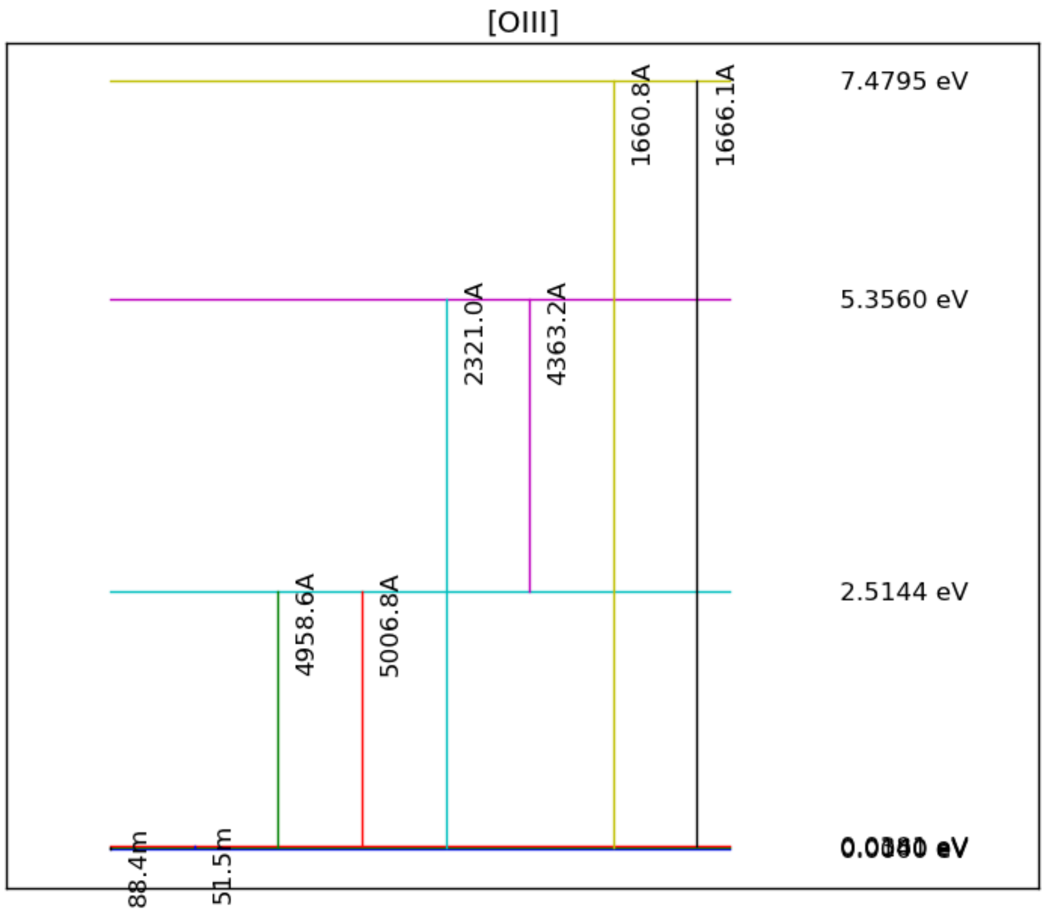
```
O3.plotEmis(tem_min=1000, tem_max=30000, ionic_abund=1.0, den=1e3, style='-', legend_loc=4, temLog=False, plot_total=False, plot_only_total=False, legend=True, total_color='black', total_label='TOTAL')
```



You can use the capacity of matplotlib plotting tools to zoom, crop, move the figure as you need.

A Grotrian diagram of the ion (i.e., a plot of the energy levels and transition wavelengths) can be obtained using:

```
O3.plotGrotrian(tem=1e4, den=1e2, thresh_int=1e-3, unit = 'eV')
```



The atomic data are accessible using the following commands:

```
O3.getA(), O3.getOmegaArray(), O3.getOmega(tem=1e4)
```

As always, use the “?” from within an ipython session, or the __doc__ method in any python session, or consult the User Reference Manual, to gain access to the different options of the methods:

```
O3.getOmega?

print(O3.getOmega.__doc__)
```

3. Use of Atom to compute populations and emissivities

Individual methods are used to access the populations, emissivities, etc:

```
O3.getEnergy(4, unit='eV')

O3.getStatWeight(level=4)

O3.getPopulations(tem=1e4, den=1e2)

O3.getCritDensity(tem=12000)

O3.getEmissivity(tem=1e4, den=1e2, wave=5007)

O3.getIonAbundance(int_ratio=100, tem=1.5e4, den=100., wave=5007)
```

The wavelengths are given with such a precision so that it cannot be confused with another line. In case of any doubt, you can call

```
O3.printTransition(5007)
```

to check which line the program considers. Levels can also be used as in:

```
O3.getIonAbundance(int_ratio=100, tem=1.5e4, den=100., lev_i = 4, lev_j=3)
```

In the case of getEmissivity method, tem and den can be arrays. In such a case, if they have different dimensions N and M, the function will return an array of NxM emissivities corresponding to all tem-den combinations; if both arrays have the same dimension, you can obtain the emissivities of either the NxN array of tem-den combinations as in the previous case, or of the 1D, N-length array obtained pairing tem and den element by element. This is controlled by the “product” parameter:

```
O3.getEmissivity([10000, 12000], [100, 500], 4, 2, product=True)
array([[ 1.22130804e-21,  1.23063754e-21], [ 1.86461478e-21,  1.87609975e-21]])
O3.getEmissivity([10000, 12000], [100, 500], 4, 2, product=False)
array([ 1.22130804e-21,  1.87609975e-21])
```

4. *Physical conditions determined from line ratios*

The Atom object is also able to compute the electron temperature or density given a line ratio:

```
O3.getTemDen(int_ratio=150., den=100., wave1=5007, wave2=4363)
```

The keyword *tem* (or *den*) specifies the supplied value of the temperature (or density). The computed quantity (temperature or density) is determined by the complementary quantity provided to the method: If “den” is given, then “tem” is computed, and viceversa.

If the intensity ratio is the ratio of two transitions, you can:

- either give the wavelengths of the two transitions involved:

```
O3.getTemDen(0.02, den=1.e4,  wave1=4363, wave2=5007)
```

- or give the four levels that define the two transitions, in the following order: (upper level of numerator) (lower level of numerator) (upper level of denominator) (lower level of denominator); e. g.:

```
O3.getTemDen(0.02, den=1.e4, lev_i1=5, lev_j1=4, lev_i2=4, lev_j2=3)
```

In the general case of an intensity ratio formed by any number of transitions, an expression must be supplied as the argument of the keyword *to_eval*:

```
O3.getTemDen(0.02, den=1.e4, to_eval="I(5, 4) / (I(4, 3) + I(4, 2))" )
N2.getTemDen(150., den=100., to_eval = '(L(6584) + L(6548)) / L(5755)')
```

(if you do not know what transition corresponds to a given wavelength, use **O3.getTransition** to find it).

The “to_eval” argument can use I(i, j) or L(wavelength) to identify the transitions involved in the diagnostic. Both can be mixed in the same string.

The parameters tem or den, as well as the line ratio, may be arrays (1D or 2D, as in case of observations obtained from IFUs), in which case the result will have the same shape. Some restrictions can be set to the domain explored by the method when looking for the solution; see the documentation for more details.

```
O3.getTemDen([0.02, 0.04], den=[1.e4, 1.1e4], to_eval="I(5, 4) / (I(4, 3) + I(4, 2))")
O3.getTemDen([0.02, 0.04], den=1.e4, to_eval="I(5, 4) / (I(4, 3) + I(4, 2))")
```

Notice that if you want to simultaneously determine both temperature and density combining two diagnostics (from two different atoms), you need to use the getCrossTemDen method form the Diagnostic object, described here.

5. *Ionic abundance determination*

The ionic abundance is obtained from the intensity of a line over Hbeta.

```
O3.getIonAbundance(int_ratio=100, tem=1.5e4, den=100., wave=5007)
```

See HERE for a determination using the Observation object.

6. *Creating a dictionary of Atom objects*

You can define all the atoms at once and put them in a dictionary by creating each atom at a time through the commands:

```
O3 = pn.Atom('O', '3')
O2 = pn.Atom('O', '2')
N2 = pn.Atom('N', '2')
```

or rather, you use one of the following shortcuts:

```
atoms = pn.getAtomDict() # a method always requires parenthesis, even without argument
atoms = pn.getAtomDict(elem_list=['C', 'N', 'O']) # all the ions from 1 to 6 are created
atoms = pn.getAtomDict(atom_list=['O2', 'O3', 'Ar3'])
```

which create a dictionary whose keys are the conventional atom names <element><spectrum> and the corresponding entries the atoms themselves; e. g.:

```
atoms['N2']                    # N II atom
atoms['N2'].getEmissivity(tem=1e4, den=1e2, wave=6584)   # example of use
```

This can be useful if you need to loop on a list of atoms, to plot atomic data for example. To see what atoms have been created (which is limited by the data includes in the selected atomic data set), enter:

```
atoms.keys()
```

If you want to be able to access them directly rather than through a dictionary, input from the command line:

```
for key in atoms.keys():
   vars()[key]=atoms[key]
```

and then you will be able to do the following:

```
N2.elem
```

7. *The atomic data*

Since for a given ion there can be different calculations of atomic data with slightly (or not so slightly!) different results, more than one version of the data can exist for a given ion. The data are stored in and read from different types of fits files depending on the mechanism of line formation. In the case of collisionally excited lines, the data are contained in the following two fits files: the “atom” file, containing the energies, the statistical weights and the transitions probabilities (the Einstein coefficients Aij), and the “coll” file, containing the temperature-dependent collision strengths. In the case of recombination data, a “rec” file contains the recombination coefficients, in a 2D temperature-density dependent table.

You can print the atomic data used for a given ion by:

```
print(O3.atomFitsFile)
print(O3.collFitsFile)
```

As the files can be found in different directories, you may also find useful to print the directories:

```
print(O3.atomFitsPath)
print(O3.collFitsPath)
```

A set of data files for a given group of ions is called a data set. The default data set can be displayed with the command:

```
pn.atomicData.getDataFile()
```

while data for a particular ion can be displayed by providing an argument to the above, e.g. for [SIV]:

```
pn.atomicData.getDataFile('S4')
```

The complete inventory of data available for a given ion can be displayed with the command:

```
pn.atomicData.getAllAvailableFiles('S3')
```

This method looks for any "s_iii_*.fits" file in a set of paths including the PyNeb package atomic data directory, the current directory (the one from where the python session is running) and any path added previously by:

```
pn.atomicData.addDataFilePath("user_directory")
```

If you want to change the atomic data used for a given ion, use the following command to point to an existing data file:

```
pn.atomicData.setDataFile('s_iii_coll_HRS12.fits')
```

If you want to change several data files, it may be worth defining a dictionary with all your preferred atomic data files within your script:

```
DataFileDict = {'H1': {'rec': 'h_i_rec_SH95.fits'},
                'N1': {'atom': 'n_i_atom_KS86-WFD96.fits', 'coll': 'n_i_coll_PA76-DMR76.fits'},
                'N2': {'atom': 'n_ii_atom_WFD96.fits', 'coll': 'n_ii_coll_LB94.fits'},
                'O2': {'atom': 'o_ii_atom_Z82.fits', 'coll': 'o_ii_coll_P06.fits'},
                'O3': {'atom': 'o_iii_atom_WFD96SZ00.fits', 'coll': 'o_iii_coll_AK99.fits'},
                'Ne3': {'atom': 'ne_iii_atom_M83-KS86.fits', 'coll': 'ne_iii_coll_McLB00.fits'}}
 pn.atomicData.setDataFileDict(DataFileDict)
```

Predefined sets of atomic data

Predefined atomic data dictionaries are provided with PyNeb by:

```
pn.atomicData.getPredefinedDataFileDict().keys()
```

Details are obtained by omitting the .keys():

```
pn.atomicData.getPredefinedDataFileDict()
```

To use a predefined data set, enter for example the following:

```
pn.atomicData.setDataFileDict('IRAF_09')
```

To revert to the default set:

```
pn.atomicData.ResetDataFileDict()
```

You can have a look at the other methods of pn.atomicData in the Reference Manual.

Creating new atomic data files

PyNeb works by creating an n-level atom characterized by the atomic data read at runtime from the fixed format fits files *xxx_atom_XYZ.fits* and *xxx_coll_XYZ.fits*, where xxx represents the ion (e.g., "o_iii" for O III) and XYZ is the acronym of the source(s) paper(s). These data represent the following quantities:

- energy levels
- statistical weights
- transition probabilities
- effective collision strengths

The first three sets of quantities are contained in the *atom* file and the last in the *coll* file. Together they provide a complete set of data for a unique ion. The data themselves are contained in the extension of the corresponding files.

The number of levels is internally called *NLevels* and is a feature of the particular atomic calculation, not an intrinsic feature of the physical atom, so it depends on the particular data used. Since the *atom* and the *coll* files may contain different numbers of levels, *NLevels* is the minimum of them. The energy levels and the statistical weights are *NLevels* arrays (one value per level). The transition probabilities are different from zero only for *(j -> i)* transitions with *j > i*, so the data are ordered in triangular *Nlevels*Nlevels* matrices with zeros on the main diagonal.

The following is an example of a file for a 6-level calculation of O III, containing the energies, the statistical weights, and the matrix of transition probabilities:

```
Energy Stat_Weight Aij

1/Ang         none  1/s   1/s    1/s    1/s    1/s    1/s      1/s
0.                  4  0.00e+00 0.00e+00 0.00e+00 0.00e+00 0.00e+00 0.00e+00 0.00e+00
0.00026818          6  3.06e-05 0.00e+00 0.00e+00 0.00e+00 0.00e+00 0.00e+00 0.00e+00
0.00026838          4  0.000178 1.3e-07  0.00e+00 0.00e+00 0.00e+00 0.00e+00 0.00e+00
0.0004048           4  0.0522    0.0991    0.0534 0.00e+00 0.00e+00 0.00e+00 0.00e+00
0.00040482          2  0.0212    0.0519    0.0867 1.41e-10 0.00e+00 0.00e+00 0.00e+00
0.00119837          6  8.61e+08 0.00e+00 0.00e+00 0.00e+00 0.00e+00 0.00e+00 0.00e+00
0.0012              4  8.65e+08 0.00e+00 0.00e+00 0.00e+00 0.00e+00 0.00e+00 0.00e+00
0.00120082          2  8.67e+08 0.00e+00 0.00e+00 0.00e+00 0.00e+00 0.00e+00 0.00e+00
*** O II energy and Aij data
*** SOURCE1 'Bowen, 1960, ApJ, 132, 1'
*** NOTE1   'Energy values for levels 2 and 3'
*** SOURCE2 'Kaufman & Sugar, 1986, JPCRD, 15, 330'
*** NOTE2   'Energy values for levels 4 and 5'
*** SOURCE3 'Fawcett, 1975, ADNDT, 16, 135'
*** NOTE3   'Energy values for levels 6, 7 and 8'
*** SOURCE4 'Wiese, Fuhr & Deters, 1996, JPCRF, Monograph 7, 396'
*** NOTE4   'A-values from level 5 or lower'
*** SOURCE5 'Unknown'
*** NOTE5   'A-values from level 6 or upper'
```

Since the energies are stored in Angstrom^{-1}, the wavelength of any transition can be simply calculated as the inverse of the energy difference between the corresponding levels. A note of caution: in the older data files inherited from nebular, many energies were simply obtained as the inverse of observed wavelengths, which in most cases were in the air; in such cases, the energy listed are not true energies but rather "energies in 1/Ang in the air”. This might lead to misidentification in the case of closely spaced lines, so please beware of this until this inconsistency is fixed.

The format shown here is the one used to convert ascii data into the PyNeb fits files, using the pn.writeAtom routine. If the data described above are stored in a file named “o_ii_atom_B60-KS86-F75-WFD96-U.dat”, use the following to obtain the “o_ii_atom_B60-KS86-F75-WFD96-U.fits” FITS file:

```
pn.writeAtom("o_ii_atom_B60-KS86-F75-WFD96-U.dat")
```

Notice that the name of the file is relevant to the code, which means that you must use the same format as the fits file, except for the .dat extension (e.g., *xxx_atom_XYZ.dat* and *xxx_coll_XYZ.dat*).

The effective collision strengths are a function of electron temperature (a collective property of the electron distribution) and are obtained as the average over a Maxwellian distribution of the collision strengths, which depend on the energy. (Note: in this document, we will often refer to the effective collision strengths as *collision strengths* for short, although this is not strictly correct.) They are usually published for a handful of T values and must be interpolated to get the collision strength at the desired values. As a result, there is a whole 1-D array of collision strengths, with one element for each tabulated temperature value, for each transition *j -> i* with *j > i*. The existence of this 3rd dimension prevents the data from being simply stored in a matrix as transition probabilities are. Instead, each transition is presented in a line, as in the following example:

```
0  0   3.699       3.778       3.903       4.000       4.204       4.301
1  3   0.54439998  0.5456      0.54799998  0.55000001  0.5564      0.56040001
1  2   0.81660002  0.81840003  0.82200003  0.82499999  0.83459997  0.84060001
1  5   0.13510001  0.1358      0.1371      0.1382      0.14139999  0.1435
1  4   0.2701      0.27149999  0.27410001  0.27649999  0.28290001  0.28690001
2  3   1.22000003  1.20799994  1.18599999  1.16799998  1.12800002  1.11199999
3  5   0.322       0.3238      0.3276      0.3321      0.34760001  0.35839999
3  4   0.47799999  0.4806      0.48640001  0.49309999  0.51599997  0.53200001
2  5   0.34599999  0.3479      0.35210001  0.35690001  0.37349999  0.38510001
2  4   0.85600001  0.86070001  0.87099999  0.88300002  0.92409998  0.95270002
4  5   0.28        0.28200001  0.28400001  0.287       0.29499999  0.30000001
1  6   1.14199996  1.15600002  1.18400002  1.21099997  1.29149997  1.34350002
1  7   0.76130003  0.77069998  0.78930002  0.80729997  0.861       0.89569998
1  8   0.38069999  0.38530001  0.39469999  0.40369999  0.4305      0.44780001
2  6   0.  0.  0.  0.  0.  0.
2  7   0.  0.  0.  0.  0.  0.
2  8   0.  0.  0.  0.  0.  0.
3  6   0.  0.  0.  0.  0.  0.
3  7   0.  0.  0.  0.  0.  0.
3  8   0.  0.  0.  0.  0.  0.
4  6   0.  0.  0.  0.  0.  0.
4  7   0.  0.  0.  0.  0.  0.
4  8   0.  0.  0.  0.  0.  0.
5  6   0.  0.  0.  0.  0.  0.
5  7   0.  0.  0.  0.  0.  0.
5  8   0.  0.  0.  0.  0.  0.
6  7   0.  0.  0.  0.  0.  0.
6  8   0.  0.  0.  0.  0.  0.
7  8   0.  0.  0.  0.  0.  0.
*** OII collision strength data
*** SOURCE1  'Pradhan, 1976, MNRAS, 177, 31'
*** NOTE1    'Collision strengths for transitions 3-2 and 5-4'
*** SOURCE2  'McLaughlin & Bell, 1993, ApJ, 408, 753'
*** NOTE2    'All other collision strengths'
```

The first row of the data (0 0 …) is the list of electron temperatures (in log10(K)).

From experience with **nebular**, it was decided that a suitable interpolating function for collision strengths is the Chebyshev polynomial. By default, each transition is described by a Chebyshev polynomial of order *n* that interpolates the tabulated values. The default value for the Chebyshev order is the number of temperature values, but other order can be specified when calling pn.writeColl.

If, for some reason, a Chebyshev interpolation is not desired, the data can also be interpolated linearly, when the instantiation of the corresponding Atom is done.

The format used in the example above is the one expected by the pn.writeColl routine which generates the fits file in the PyNeb format. For example, if the previous data are stored in a file called “o_ii_coll_P76-McLB93.dat”, the FITS file “o_ii_coll_P76-McLB93.fits” is obtained by:

```
pn.writeColl(“o_ii_coll_P76-McLB93.dat”)
```

If the collision strtengths table is oriented in electron temperatures by rows, use the following keyword:

```
temp_in_cols = False
```

when calling writeColl().

Notice the use of SOURCE and NOTE keyword to store the references of the data. It is very important to document those keywords and to give a name to the file that reflects the sources of the data. This information is available from the Atom objects by:

```
O3.printSources()
```

Plotting atomic data

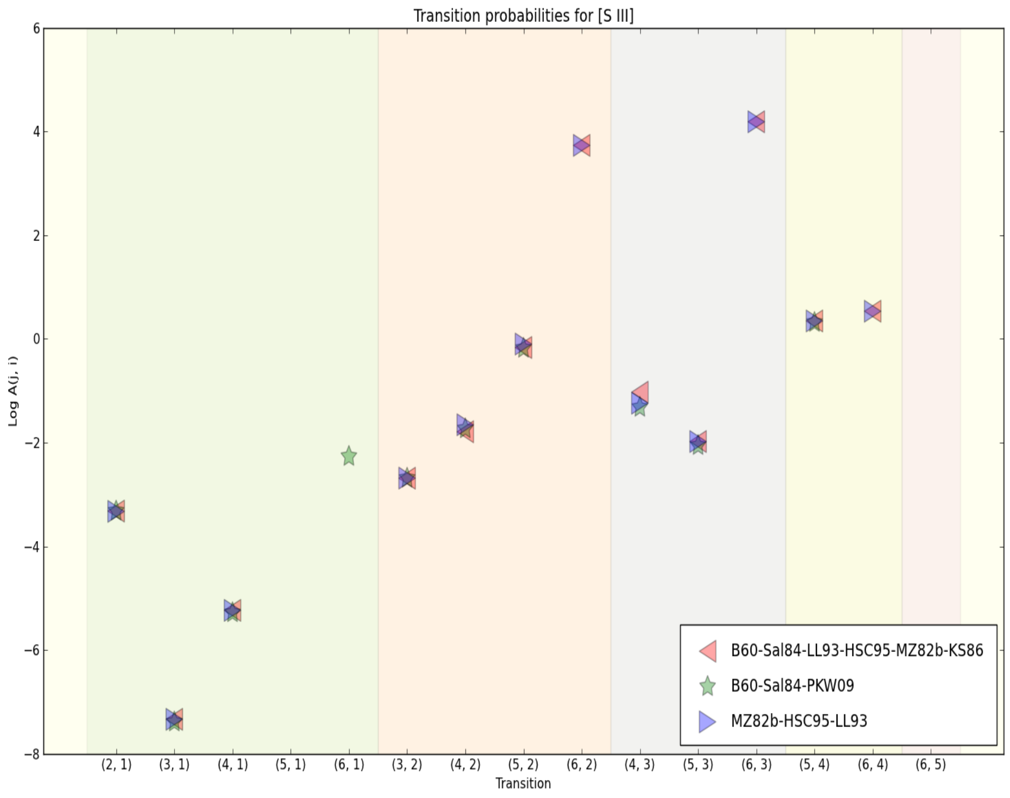
DataPlot is a class to produce plots of the atomic data provided. The module must be initialized by specifying which data sets are to be plotted; it allows plotting the transition probabilities (either as an absolute value or relative to a reference set: **A** and **relA** respectively) and the collision strengths (**Omega**).

You must first create an instantiation of DataPlot for a given atom:

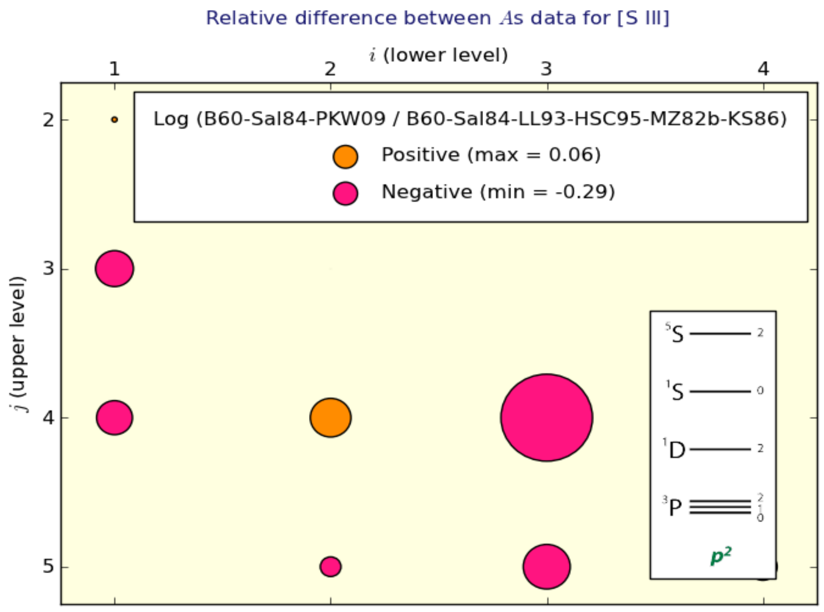
```
dp_O3 = pn.DataPlot('O', 3)
dp_S3 = pn.DataPlot('S', 3)
```

Then you can plot the different As and Omegas :

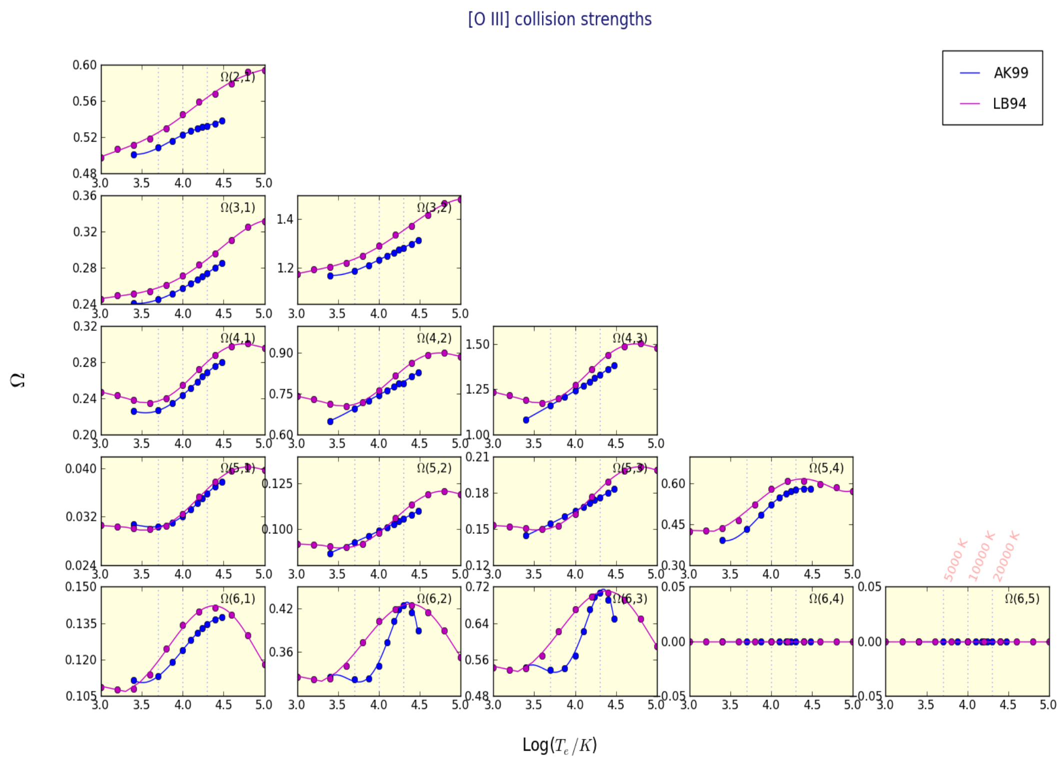
```
    dp_S3.plotA() # transition probabilities plot
```



```
dp_S3.plotRelA() # relative transition probabilities plot
dp_S3.plotRelA(ref_data = 'B60-Sal84-PKW09') # defining relative to which ref
```



```
dp_O3.plotOmega() # collision strength plot
```



E) Recombination lines: the RecAtom class

This object is similar to the Atom one, but some methods differ due to the particularities of the recombination spectrum. It is instantiated like this:

```
H1 = pn.RecAtom('H', 1)
```

The data are read from a fits file defined in the pn.atomicData.getDataFile('H1', 'rec') file and interpolated in tem, den. The ions for which recombination lines are available are listed with the command:

```
pn.atomicData.getDataFile(data_type='rec')
```

You can list all the available data files for a given ion by:

```
pn.atomicData.getAllAvailableFiles('H1')
```

The emissivities are obtained by:

```
Hbeta = H1.getEmissivity(tem=1e4, den=1e3, lev_i=4, lev_j=2)
```

There are two ways in which the lines are identified in the data file, by transition (two levels) or by wavelength. In this case, we use the levels to identify the line. A more compact way is to use the label:

```
Hbeta = H1.getEmissivity(tem=1e4, den=1e3, label="4_2")
```

The label lists the two levels separated by an underscore "_". You can easily generate a 2D table of H alpha/ H Beta depending on Te and Ne:

```
tem = np.linspace(5000, 20000, 100)
den = np.logspace(2, 6, 100)
im_Hab = (H1.getEmissivity(tem, den, label='3_2') /
          H1.getEmissivity(tem, den, label='4_2'))
```

Case A emissivities are available by changing the atomic data file:

```
print(pn.atomicData.getAllAvailableFiles('H1'))
```

```
['h_i_rec_SH95-caseA.fits', 'h_i_rec_SH95.fits']
```

```
pn.atomicData.setDataFile('h_i_rec_SH95-caseA.fits')

H1_A = pn.RecAtom('H', 1)

im_casAonB = (H1_A.getEmissivity(tem, den, label='4_2')/H1.getEmissivity(tem, den, label='4_2'))
```

For the HeI lines, the wavelength is used instead of the line levels.

You can print all the labels doing:

```
He1 = pn.RecAtom('He', 1)

print(He1.labels)
```

```
['2945.0', '3188.0', '3614.0', '3889.0', '3965.0', '4026.0', '4121.0', '4388.0', '4438.0', '4471.0', '4713.0',
'4922.0', '5016.0', '5048.0', '5876.0', '6678.0', '7065.0', '7281.0', '9464.0', '10830.0', '11013.0', '11969.0',
'12527.0', '12756.0', '12785.0', '12790.0', '12846.0', '12968.0', '12985.0', '13412.0', '15084.0', '17003.0',
'18556.0', '18685.0', '18697.0', '19089.0', '19543.0', '20427.0', '20581.0', '20602.0', '21118.0', '21130.0',
'21608.0', '21617.0']
```

The emissivities are obtained by :

```
He1.getEmissivity(1e4, 1e2, wave=4471.0)

He1.getEmissivity(1e4, 1e2, label="4471.0")
```

F) The extinction class: RedCorr()

The class RedCor manages the extinction (reddening) correction. It can compute the logarithmic extinction at Hbeta by comparing an observed ratio to a theoretical one (usually Halpha/Hbeta, but any other ratio can be used). The object is also able to compute the correction to be applied to any intensity, given the wavelength of the line.

Various extinction laws are included in the Class, and any user-defined function can also be used. The available extinction laws can be listed by entering (here no need to instantiate an object):

```
pn.RedCorr().printLaws()
```

Less detailed output is obtained with:

```
pn.RedCorr().getLaws()
```

The object is instantiated using:

```
rc = pn.RedCorr()

rc = pn.RedCorr(E_BV = 1.2, R_V = 3.2, law = 'Fitz 99')
```

Every parameter can be defined after the instantiation:

```
rc.E_BV = 1.34

rc.law = 'S 79 H 83'
```

The relation between cHbeta and E(B-V) is:

E(B-V) = 0.61 * cHbeta + 0.024 * cHbeta ** 2.

Once one of the two parameters is defined, the other is directly obtained by:

```
print(rc.cHbeta)
```

The reddening of a given spectrum is determined by using the ratio of two observed line intensities relative to the theoretical value, for example:

```
rc.setCorr(6.5 / 2.86, 6563., 4861.)
```

Once a law and cHbeta or E(B-V) are defined, the correction for any wavelength is obtained by:

```
correc = rc.getCorr(wave)
```

where wave can be a list or array of wavelengths.

The correction relative to the Hbeta correction is given by:

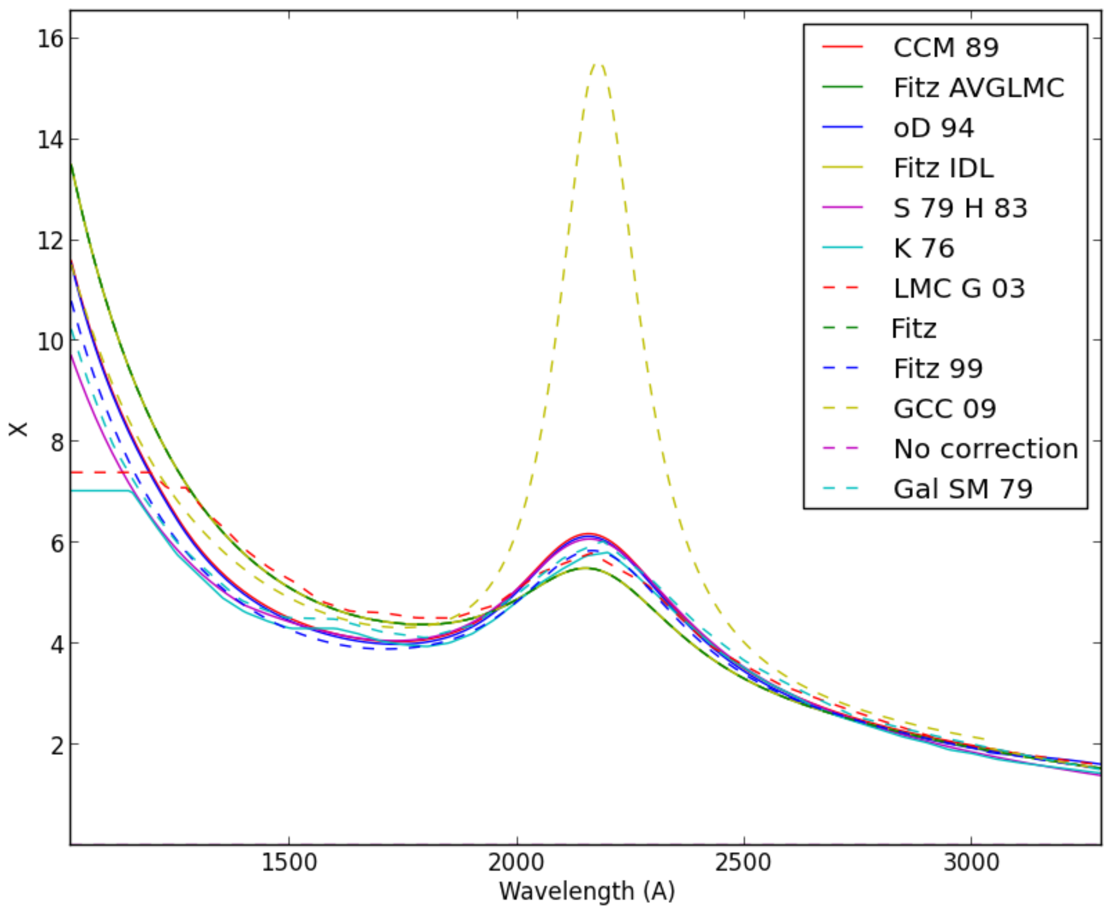
```
correc = rc.getCorrHb(wave)
```

and relative to any other wavelength correction by:

```
correc = rc.getCorr(wave1, wave2)
```

The object contains a plotting tool to have a quick look at the different extinction laws:

```
rc.plot(laws = 'all')
```



A user-defined function can also be used. The user-defined function must accept 2 parameters : wavelength(s) in Angstrom and an optional parameter (which can also be a list). It must returns X(lambda) = A(lambda)/E_BV = RV.A(lambda)/AV. The correction is then: 10**(0.4*E_BV*X)

Here is an example of a user-defined function:

```
    def my_X(wave, params = [5000., 1., 2., 3.]):
        return params[1] * (wave/params[0]) + params[2] * (wave/params[0])**-1
+ params[3] * (wave/params[0])**-2
        rc.UserFunction = my_X
        rc.UserParams = [6000., 0., 0., 1.]
        rc.getCorr(5007)
```

G) The EmissionLine class

This is the class characterizing emission lines. An emission line is identified by an element and a spectrum (which identify the emitting ion), a wavelength in Angstrom, a blend flag, a label in the standard PyNeb format, an observed intensity, a reddening-corrected intensity, an expression describing how the intensity depends on the included wavelengths, an observational error and an error on the corrected intensity. Other programs determine one or more of these values.

To instantiate an Emission Line object, use the following:

```
line = pn.EmissionLine('O', 3, 5007, obsIntens=[1.4, 1.3])
line = pn.EmissionLine(label = 'O3_5007A', obsIntens=320, corrected=True )
```

obsIntens is a value or a list of values (may also be a numpy array) corresponding to the observed intensity of the given emission line.

To know how the label of a given line is exactly spelled, you can print the dictionary pn.LINE_LABEL_LIST:

```
print(pn.LINE_LABEL_LIST['O3'])
```

It is possible to define a line that is not in the pn.LINE_LABEL_LIST. In this case a warning is issued, but the code can still continue to run.

An emission line is instantiated with an observed intensity *line.obsIntens*. But you certainly want to deal with the extinction-corrected intensity. This is stored in the *line.corrIntens* variable, which is set to 0.0 when the line is instantiated, unless the parameter corrected is set to True, in which case the observed value *obsIntens* is copied to the *corrIntens* value (the same applies for *corrError*, which is set to *obsError*).

The *corrIntens* value can also be computed using an instantiation of the pn.RedCorr class (see this Section):

```
redcorr = pn.RedCorr(E_BV = 0.87, law = 'Fitz 99')
line.correctIntens(redcorr) #redcorr is used to compute line.corrIntens
```

The line information is printed using:

```
line.printLine()
```

```
Line O3 O3_5007A evaluated as L(5007.0)
Observed intensity: 320.0
Observed error: 0.0
Corrected intensity: 320.0
Corrected error: 0.0
```

If you need to add data to an existing EmissionLine object, use the addObs() method.

Most of the work on the EmissionLine objects will be performed from the Observation class (read data, extinction correction); see next section.

H) The Observation class: reading and dealing with observations

1. Reading observation from a file

pn.Observation is the class characterizing observation records. An observation record is composed of an object identifier, the observed intensity of one or more emission lines, and, optionally, the dereddened line intensities and the identifier of the extinction law used, and the value of c(Hbeta).

Observations can be initialized by reading data files, which can be organized with different emission lines either in rows or columns (usually, in a survey of many objects with few emission lines emission lines change across columns; and in a high-resolution observation of a small sample of objects lines change across rows).

The following is an example of how to define an observation:

```
obs = pn.Observation()        # define the Observation object 'obs'
obs.readData('smc_24.dat', fileFormat='lines_in_rows', err_default=0.05) # fill obs with data read from smc_24.dat
obs.extinction.law = 'CCM 89'  # define the extinction law from Cardelli et al.
obs.correctData()                  # the dereddened data are computed
```

The data can be read by the *readData* method as above or directly while instantiating the object:

```
obs = pn.Observation('smc_24.dat', fileFormat='lines_in_rows',          corrected=True)
```

The format of the data file from which the emission line intensities are read can be one of two kinds:

- *fileFormat='lines_in_rows'*:

```
LINE         Cn1_5         Hb4        He2_86
O2_3726A     0.71752200    0.17224200 0.12817400
O2_3726Ae    0.05800000    0.09730000 0.07760000
O2_3729A     0.36053100    0.07933200 0.05474200
O2_3729Ae    0.05810000    0.09920000 0.07830000
Ne3_3869A    0.95247500    1.04331400 0.70986900
H1_4861A     1.00         1.00        1.00
```

- *fileFormat='lines_in_cols'* (this is the default):

```
NAME          O2_3726A  O2_3726Ae O2_3729A O2_3729Ae
NGC3132       0.93000   0.05000   0.17224200 0.10
IC418         1.28000   0.05000   0.09920000 0.05
M33           0.03100   0.080     0.03100    0.10
```

The labels “LINE” and “NAME” are mandatory to identify rows or columns that are not line intensities.

The delimiter between the columns is any sequence of spaces or TAB, but it can be changed using the *delimiter* parameter. The line names are defined by a label starting with the name of the atom (‘O2’), followed by an underscore, followed by a wavelength and ending with a unit (‘A’ or ‘m’). The list of all the lines managed by PyNeb is obtained by:

```
for atom in pn.LINE_LABEL_LIST:
  print(atom, pn.LINE_LABEL_LIST[atom] )
```

The presence of a trailing “e” at the end of the label points to the error associated to the line. The error is considered to be relative to the intensity, unless the parameter *errIsRelative* is set to False. A common value for all the errors can be defined by the parameter *err_default=0.10* (0.10 is the default value).

2. *Extinction correction*

Once the data are read, they have to be corrected from extinction. An instantiation of RedCorr() is available inside the Observation object as *obs.extinction*.

If a line label in the data file is "cHbeta" or "E(B-V)", then it is transmitted to the extinction correction object. Otherwise it as to be set by, for example:

```
obs.extinction.E_BV = 0.34
obs.extinction.cHbeta = 1.2
```

An extinction law has also to be specified:

```
obs.extinction.law = 'Fitz 99'
```

To correct all the lines at once:

```
obs.correctData()
```

If you want the corrected line intensities to be normalized to a given wavelength, use the following:

```
obs.correctData(normWave = 4861.)
```

The extinction correction can be determined from the observed values, using the following:

```
obs.def_EBV(label1="H1_6563A", label2="H1_4861A", r_theo=2.85)
```

Once this is done, you will need to call:

```
obs.correctData()
```

The lines (each line being an *EmissionLine* object) are stored in the *Observation.lines* list. They can be printed using:

```
obs.printIntens()
```

By default, the corrected intensities are printed out. The observed intensities are obtained using the *returnObs=True* parameter.

The method getSortedLines return the lines sorted in alphabetical order of atoms, or in wavelengths (using the *crit = 'wave'* parameter):

```
for line in obs.getSortedLines():
   print(line, line.corrIntens)
```

A method that will be useful later is the following, which gives the list of all the atoms implied in the observed emission lines:

```
atomList = obs.getUniqueAtoms()
```

3. *Adding observations and lines*

Once an object Observation is instantiated, you can add a new observation (corresponding to a new object or a new fiber) by using:

```
obs.addObs('test', obs_tab)
```

where 'test' is the name of the new observation. The new observation must be of the same shape than the already present, it means that its length must be of obs.n_lines.

A new emission line can also be added by:

```
obs.addLine(line)
```

4. *Getting line intensities*

You can extract the line intensities from an Observation object by, for example:

```
print(obs.getIntens(obsName='M1_32'))
print(obs.getIntens()['Ar4_7262A'])
```

5. *Using Observation to determine ionic abundances*

Once the electron temperature and density are determined, it is easy to obtain the ionic abundances from a set of emission lines included in an Observation object:

```
# Define a dictionary to hold all the need Atom objects
all_atoms = pn.getAtomDict(atom_list=obs.getUniqueAtoms())
# define a dictionary to store the abundances
 ab_dic = {]
# we  use the following lines to determine the ionic abundances
ab_labels = ['N2_6583A', 'O2_3727A+', 'O3_5007A', 'S2_6716A',
            'S3_9069A', 'Ar3_7136A', 'Ne3_3869A']
for line in obs.getSortedLines():
    if line.label in ab_labels:
        ab = all_atoms[line.atom].getIonAbundance(line.corrIntens, Te, Ne,
                                        to_eval=line.to_eval, Hbeta=100)
        ab_dic[line.atom] = ab
```

I) Exploring line intensities with the EmisGrid class

1. *The EmisGrid class*

Most plots are obtained by operating on emission maps, defined here as grids of emissivities as a function of temperature and density, which can be generated by EmisGrid.

EmisGrid instantiates an atom and computes the emissivities of all the atom lines of this atom for the (tem, den) values of a regularly spaced grid (may be log or linear in the case of the density). Each line is represented in a 2D array (a grid), and there are as many arrays as there are transitions in the atom. The results can be operated on, saved for a later use in a cPickle file, or restored.

The following command instantiates an [O III] atom and computes the emissivity of all its lines in a 30x30 grid:

```
O3_EG = pn.EmisGrid('O', 3, n_tem=30, n_den=30)
O3_EG.save('plot/O3_30by30.pypic')
O3_EG = pn.EmisGrid(restore_file='plot/O3_30by30.pypic') # Restored from a previous 
computation
```

The arguments are described in more details in the Reference Manual. Here is the list:

```
O3_EG = pn.EmisGrid(elem=None, spec=None, n_tem=100, n_den=100, tem_min=5000.,
                tem_max=20000., den_min=10., den_max=1.e8, restore_file=None, atomObj=None)
```

The emissivity grid can be obtained by:

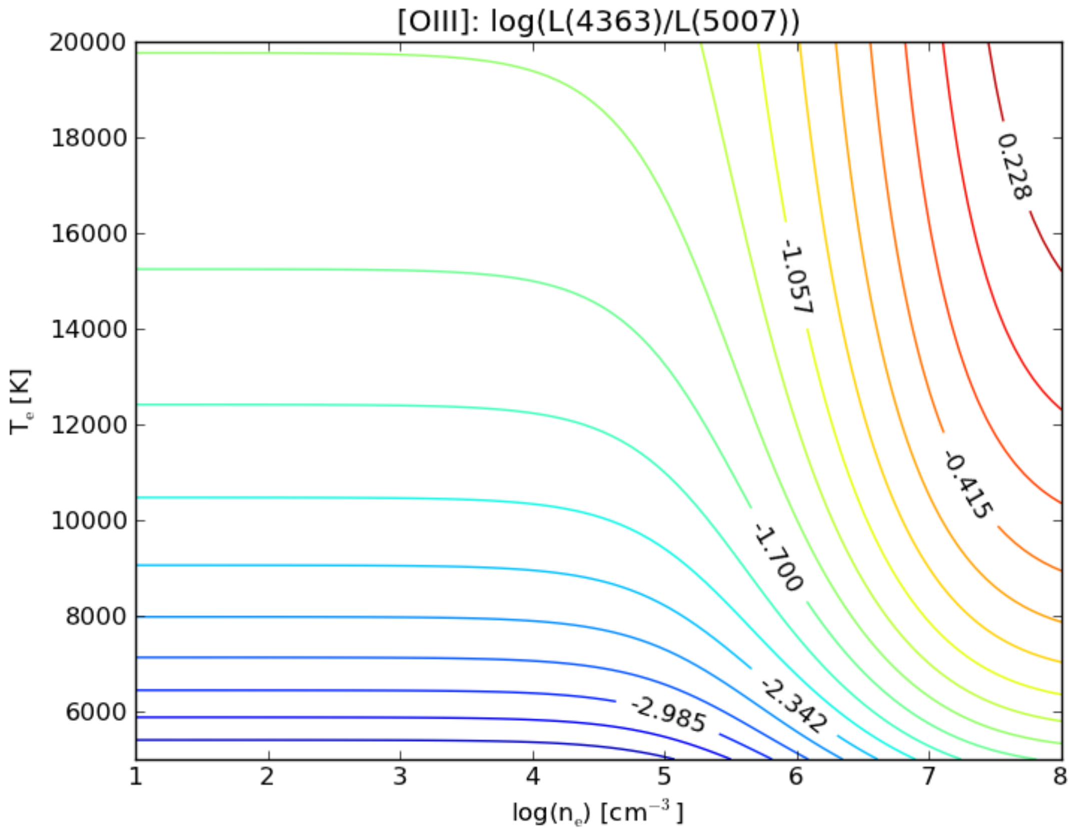
```
O3_5007 = O3_EG.getGrid(wave=5007)
```

Line combinations are available:

```
O3_Te = O3_EG.getGrid(to_eval = 'L(4363)/L(5007)')
```

There are two plotting tools integrated in the EmisGrid object (but see also the Diagnostic Object for combining different atoms in a single plot). See the Reference Manual for more options.

```
O3_EG.plotImage(to_eval = 'L(4363)/L(5007)')
O3_EG.plotContours(to_eval = 'L(4363)/L(5007)')
```



2. *Instantiating various EmisGrid Objects with getEmisGridDict*

It is quite common to have to instantiate various *EmisGrid* objects, especially if you want to make a diagnostic diagram. This can easily be done using the getEmisGridDict method, used for example as follows:

```
emisgrids = pn.getEmisGridDict(['C', 'N', 'O'], [2, 3])
```

The previous command will generate a dictionary of all the combination of ['C', 'N', and 'O'] with ionizations stages 2 and 3, that is 6 atoms. The resulting maps are saved in a directory defined by default when PyNeb is started, in the *pn.config.pypic_path* variable. It first tries to use the $HOME/.pypics directory; if it fails, it tries to use /tmp/pypic; if it fails too, the value is set to None and a user-defined value has to be provided by changing *pn.config.pypic_path* or using the *pypic_path* parameter when calling getEmisGridDict.

If a Diagnostic object is already available (see next Section), it can be used to determine the atoms for which a grid is computed/restored:

```
emisgrids = pn.getEmisGridDict(atomDict=diags.atomDict,
                    den_max=1e6, pypic_path='./pypics')
```

This EmisGrid dictionary will be very useful to plot the diagnostic diagrams with the Diagnostic object, described in the next section.

Two functions are available from the pn.utils.misc domain to manage the stored pypics files, adapting the following examples:

```
from pyneb.utils.misc import cleanPypicFiles, getPypicFiles
print('default directory for the pypics files is {}'.format(pn.config.pypic_path))
getPypicFiles()
cleanPypicFiles('emis_S2.pypic')
cleanPypicFiles(all_ = True)
getPypicFiles()
```

J) The Diagnostics class

Diagnostics is the class used to evaluate temperatures and densities from line ratios. It is also the class that plots the diagnostic Te-Ne diagrams. The object is instantiated like this:

```
diags = pn.Diagnostics()        # instantiate the Diagnostic class
```

An optional parameter *addAll=True* (default is False) lets the object loads all the available diagnostics. Most of the time this option is not used and the diagnostics will be added as they are needed.

1. Determination of temperature and density

The *getCrossTemDen* method cross-converges the temperature and density derived from two sensitive line ratios (not necessary from the same atom), by inputting the quantity derived with one line ratio into the other and then iterating. When the iteration process ends, the two diagnostics give self-consistent results. The first line ratio must be a temperature-sensitive one and the second a density-sensitive one. The temperature and density can be individual numbers as well as arrays (provided they are equal in shape).

```
diags=pn.Diagnostics()
print(diags.getCrossTemDen('[NII] 5755/6548', '[SII] 6731/6716', 0.02, 1.0))
```

The observed ratio can be automatically extracted from an Observation object named *obs*:

```
Te, Ne = diags.getCrossTemDen('[NII] 5755/6548', '[SII] 6731/6716', obs=obs))
```

The complete list of the predefined diagnostics is stored in the *pn.diags* dictionary and can be listed with:

```
for diag in sort(pn.diags_dict.keys()):
   print('"{0}" : {1}'.format(diag, pn.diags_dict[diag]))
```

```
...
"[OIII] 4363/5007" : ('O3', 'L(4363)/L(5007)', 'RMS([E(5007),E(4363)])')
...
```

Each diagnostic is defined by a label and is associated to a tuple containing 3 elements: the atom corresponding to the diagnostic lines, the algebraic definition of the line ratios and the algebraic definition of the error of the diagnostic depending on the error of each line involved. In the present case, the diagnostic is the ratios of two O3 lines, 4363/5007, and the error is the quadratic sum of the relative error of each line (E(lambda)): RMS(a, b) = sqrt(a**2 + b**2).

In the case of more lines, the error is obviously more complex and involves the error and the intensities of the lines, defined by E(lambda) and L(lambda) respectively:

```
        "[SII] 4072+/6720+" : ('S2', '(L(4069)+L(4076))/(L(6716)+L(6731))',
'RMS([E(6716)*L(6716)/(L(6716)+L(6731)),E(6731)*L(6731)/(L(6716)+L(6731)),E(4069)*L(406
9)/(L(4069)+L(4076)),E(4076)*L(4076)/(L(4069)+L(4076))])')
```

The user can also define its own diagnostic, for example using:

```
diags.addDiag('[OIII] 4363/4959', ('O3', 'L(4363)/L(4959)', 'RMS([E(4363),E(4959)])')
```

Notice that the diagnostics are defined so that they increase with the main parameter they trace: [OIII] 4363/5007 increases with the electron temperature.

The diagnostics contained in a *Diagnostic* object are listed by the ***

diags.getDiagLabels() and *diags.getDiags()* methods. Once added to the Diagnostic object, they can be used to compute Te and Ne via *getCrossTemDen*, or to do diagram (see below). A diagnostic can be removed from the list with the *delDiag* method.

2. *Diagnostic diagram combining various atoms*

The diagnostics that will be used in the diagrams are set by using predefined or user-defined specification, as described above. The list of diagnostics can be directly added by hand:

```
diags.addDiag([
        '[NI] 5198/5200',
        '[NII] 5755/6548',
        '[OII] 3726/3729'])
```

The plotting tool included in the Diagnostic class needs an *EmisGrid* dictionary (as returned by *pn.getEmisGridDict*, see the previous section) and an Observation object (see here). Thus the plot is obtained by:

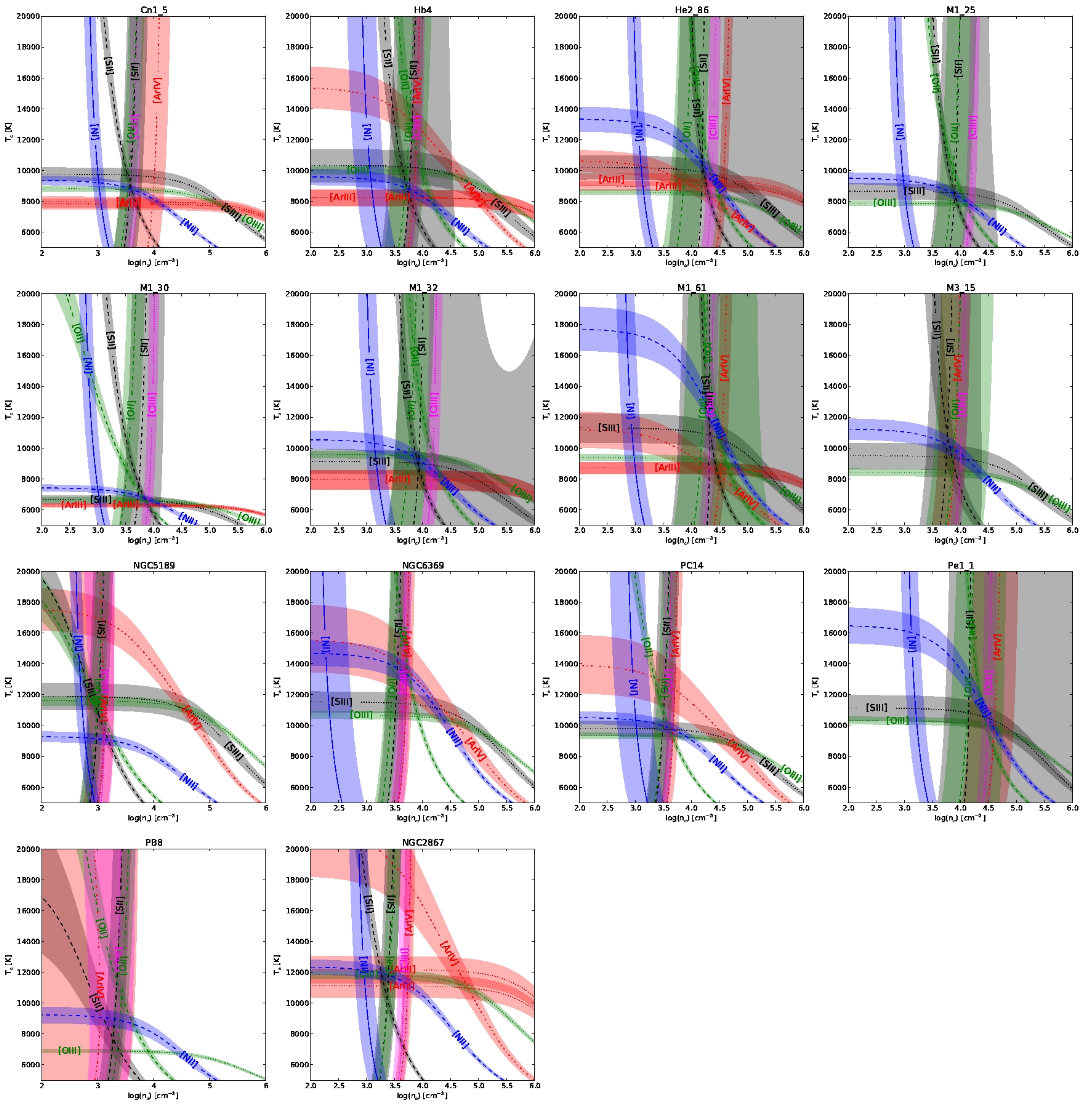
```
diags.plot(emisgrids, obs)
```

If there is more than one spectrum in the *obs* object, the index of the observation to use is given by *i_obs*:

```
diags.plot(emisgrids, obs, i_obs = 4)
```

An example of use with 16 objects:

```
plt.figure(figsize=(30, 30))
for i, obs_name in enumerate(obs.names):
  plt.subplot(4, 4, i+1)
  diags.plot(emisgrids, obs, i_obs=i)
  plt.title(obs_name)
plt.savefig('Diagnostics.pdf')
```



Data and plots from:

Analysis of chemical abundances in planetary nebulae with [WC] central stars. I. Line intensities and physical conditions

García-Rojas, J., Peña, M., Morisset, C., Mesa-Delgado, A., & Ruiz, M. T.

2012, *Astronomy and Astrophysics*, **538,** A54

The label used to identify the emission lines can be changed in the *diags* object, using for example:

```
diags.add_clabel('[OIII] 5007/4363', '[OIII]na')
```

K) Determination of elemental abundances using ICFs

The determination of elemental abundances is complicated by the fact that some ions are not observed. To take this into account when computing the total abundances, it is necessary to use ionization correction factors (ICFs). The ICFs are expressions used to correct the abundance of observed ions for unseen ions to get the total elemental abundance of a given element:

X(elem) = X(ion) * icf,

where X(ion) may be the abundance of one single ion or the sum of several ions of the element. These expressions are generally obtained empirically or semiempirically, based on the results of photoionization models or a comparison between the ionization potentials of different ions. Most expressions have been devised for a specific kind of object (e.g., PNe, HII regions, etc) and should not be applied to objects of a different kind.

In PyNeb, an icf formula is identified by an element ("elem", e.g. "O"), which is the element of which the total abundance is seeked for; the ion or ions whose abundance is corrected ("atom", e.g. "O2+O3"); and the icf proper, which is an expression involving the abundances of other ions, assumed to be known ("icf", e.g. "1 + 0.5 * He3 / (He2 + He3)"). In addition, each icf expressions is identified by a label and holds the original reference and a brief comment specifying its intended field of application. The label is formed by an acronym of the paper and the equation number of the icf within the paper.

The following snippet illustrates how PyNeb manages icfs:

```
        atom_abun = {'O2': 0.001, 'O3': 0.002, 'Ne3': 1.2e-5}

        icf.getAvailableICFs('Ne')   # lists all the available recipes for Ne

        elem_abun = pn.getElemAbundance(atom_abun, icf_list=['TPP85']) # Computes the Ne abundance
with the TPP06 recipe

        elem_abun = pn.getElemAbundance()  # performs all the possible element abundance
computations given the input ionic abundance set and available icf
```

The first line above defines a set of ionic abundances; the second lists the label of all the available recipes for Ne; in the third, a specific one is selected to compute the desired abundance; in the fourth line, all the possible element abundance computations given the present icf set are performed (these may include icfs not suitable for the object under study).

Additionally, PyNeb provides a series of single-line commands to explore the icf collection and its source papers; e.g.:

```
        icf.getAvailableICFs()

        icf.getExpression('KH01_4g')   # returns the analytical expression of the icf identified by the label
KH01_4g

        icf.getReference('KH01_4g')   # returns the bibliographic reference of the source paper

        icf.getURL('KH01_4g')   # returns the ADS URL of the source paper
```

PyNeb provides a large set of icfs compiled from the literature; each icf is stored together with the ref, the URL and a comment with further details. Further expressions will be added in the future, and a special function (addICF) allows to add customized expressions to the collection.

L) The logging facility

When importing PyNeb, a special object is instantiated to hold messages (error, warnings, normal messages and debug tools).
This object is named *pn.log*. The object receives messages, writes them in the standard output (and eventually in a file) and also keeps them in memory. It can also stop the execution and raise an error when error message is sent to it. It also tracks which method in the code sends the message (the calling parameter), and the time at which is was sent.
There is a level of verbosity, which can be changed at any moment (e.g. *pn.log_.level = 2*), that determines the kind of messages that will be printed to the standard output (commonly the screen).

There are 5 types of messages this object can manage:

- debug: using *pn.log_.debug('This is the message', calling = 'Routine1')*, it is printed if *pn.log_level >= 4.*
- message: using *pn.log_.message('This is the message', calling = 'Routine1')*, it is printed if *pn.log_level >= 3.*
- warning: using *pn.log_warn('This is the warning', calling = 'Routine1')*, it is printed if *pn.log_level >= 2.*
- error: using *pn.log_error('This is the error', calling = 'Routine1')*, it is printed if *pn.log_level >= 1.* When error is called, an error is raised using exception *PyNebError*, unless the *exception* argument is given. This default behavior is altered by changing *pn.log_.no_exit* to False, in which case a *SystemExit* exception is raised (and the program stops).
- timer: using *pn.log_.timer('Ending the process', calling = 'Routine1', quiet = False)*. This method prints (if quiet = True) the message with the time spent since the last call of timer.

At any moment in a script file the verbosity can be changed either to see what's happening, or to avoid being annoyed by too many messages.

It is possible to store the messages in a file. The file is open by *pn.log_.open_file('my_log.txt')*, and eventually closed by *pn.log_.close_file()*.
Access to the messages is possible with *pn.log_.print_messages(), pn.log_.print_warnings() and pn.log_.print_errors().*

M) References

Shaw, R. A., Dufour, R. J. 1995, PASP, 107, 896

Shaw, R.A., de La Peña, M. D., Katsanis, R. M., & Williams, R. E. 1998, in ADASS VII, edited by R. Albrecht, R. N. Hook, & H. A. Bushouse, ASP Conf. Ser., 192

N) How it works

1. *getPopulations*

The populations are found by solving the following linear system:

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & \dots \\ n_e\, q_{1,2} & -[n_e(\, q_{2,1}) + n_e(\, q_{2,3} + \dots + \, q_{2,n}) + A_{2,1}] & n_e\, q_{3,2} + A_{3,2} & n_e\, q_{4,2} + A_{4,2} & \dots & \dots \\ n_e\, q_{1,3} & n_e\, q_{2,3} & -[n_e(\, q_{3,1} + \, q_{3,2}) + n_e(\, q_{3,4} + \dots + \, q_{3,n}) + A_{3,1} + A_{3,2}] & n_e\, q_{4,3} + A_{4,3} & n_e\, q_{5,3} + A_{5,3} & \dots \\ n_e\, q_{1,4} & n_e\, q_{2,4} & n_e\, q_{3,4} & -[n_e(\, q_{4,1} + \, q_{4,2} + \, q_{4,3}) + n_e(\, q_{4,5} + \dots + \, q_{4,n}) + A_{4,1} + A_{4,2} + A_{4,3}] & n_e\, q_{5,4} + A_{5,4} & \dots \\ n_e\, q_{1,5} & n_e\, q_{2,5} & n_e\, q_{3,5} & n_e\, q_{4,5} & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \end{pmatrix} \begin{pmatrix} n_1 \\ n_2 \\ n_3 \\ n_4 \\ n_5 \\ \dots \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ \dots \end{pmatrix}$$

The logic underlying the method is that the matrix is filled and the system is solved for the population vector [n1, n2, ...], using the numpy linear algebra method *np.linalg.solve*. The implementation is slightly complicated by the fact that both tem and den can be 1-D arrays rather than individual values, so that an n_tem x n_den array of coefficient matrices is built. This is done as follows (consider that the elements of the coefficient matrix are combinations of the density den, the transition probability A, and the collisional rate q, and that q, but not A, depends on temperature):

- A 1-D vector of length n_tem is created whose elements are equal to 1 (tem_ones = np.ones(n_tem)); the same for den.
- The collisional rates for the whole temperature array are computed and stored in q (which is

therefore defined as a 1-D array of n_tem elements): q = self.getCollRates(tem);

- A 3-D array is created by replicating the 2-D A(i, j) array n_tem times (Atem = np.outer(self.A, tem_ones).reshape(n_level, n_level, n_tem); the reshape is needed because np.outer really makes the outer products of vectors, not arrays, so one dimension is lost and must be restored.

- A 2-D array is built as the outer product of q and den (np.outer(np.swapaxes(q, 0, 1), den); swapaxes is needed to make the 2nd dimension of the fist array equal to the 1st dimension of the 2nd array so that the outer product can be carried out).

- A 2-D array is built as the outer product of A and the unitary vector of size = size(density) (np.outer(np.swapaxes(q, 0, 1), den_ones); swapaxes is to make the 2nd dimension of the fist array equal to the 1st dimension of the 2nd array so that the outer product can be carried out).

- These two are summed up, and then reshaped to be a (n_level, n_level, n_tem, n_den) array (the reshape is needed for the same reason as above: outer always returns a matrix, so the exceeding dimensions are "lost"). The result is stored in coeff_matrix.

- The slice of coeff_matrix corresponding to first index = 0 has been previously set to 1 (see docs/pop_array.pdf; this corresponds to the equation which conserves the total number of electron across levels) (coeff_array[0,:] = 1.).

- We have now one system for each density/temperature combination: a loop over tem and den solves each of these systems.

- The result is squeezed to get rid of any superfluous dimension, in case either tem or den are individual values and not arrays (np.squeeze(pop_result)).

2. *getEmissivity*

returns the emissivity at given temperature and density:

```
O3.getEmissivity(12000, 100, 4, 2)              # (4, 2) transition
O3.getEmissivity(10000, 10000, wave=5007)  # (4, 2) transition
O3.getEmissivity(12000, 100)                      # all transitions
```

This method computes the emissivity for a given transition or the whole transition array. The transition is selected by the argument *wave* (if given); if *wave* is not supplied, it is selected by the upper and lower levels (*lev_i* and *lev_j*); if neither is given, the whole array is computed:

How it works:

Since both *tem* and *den* can be vectors (of *n_tem* and *n_den* elements, respectively) rather than individual numbers, the computation is vectorial:

A 1-D vector of length *n_tem* is created whose elements are equal to 1 (*tem_ones = np.ones(n_tem)*); same for *den*.

The population array *populations* is created (which is in fact a population vector for each of the *n_tem* * *n_den* different combinations of temperature and density).

The emissivity array *resultArray* is initialized (an array of (*Nlevel* * *Nlevel* * *n_tem* * *n_den* different combinations of temperature and density).

A loop over *i* and *j* fills the array: for each *i, j* pair with *i > j* the product *A*DeltaE*n_i* is computed and divided by the density to give it as emissivity per unit densities (the division by density is in fact a division by a 2-D array: one holds the temperature dimension, the other the density dimension and magnitude; the array is reshaped to (*1, 1, n_tem, n_den*) because there is no outer product among arrays and the elements must be redistributed: see also getPopulations).

Finally, the *resultArray* is squeezed to get rid of any extra dimension (i.e., whether *tem* and/or *den* have only one element)