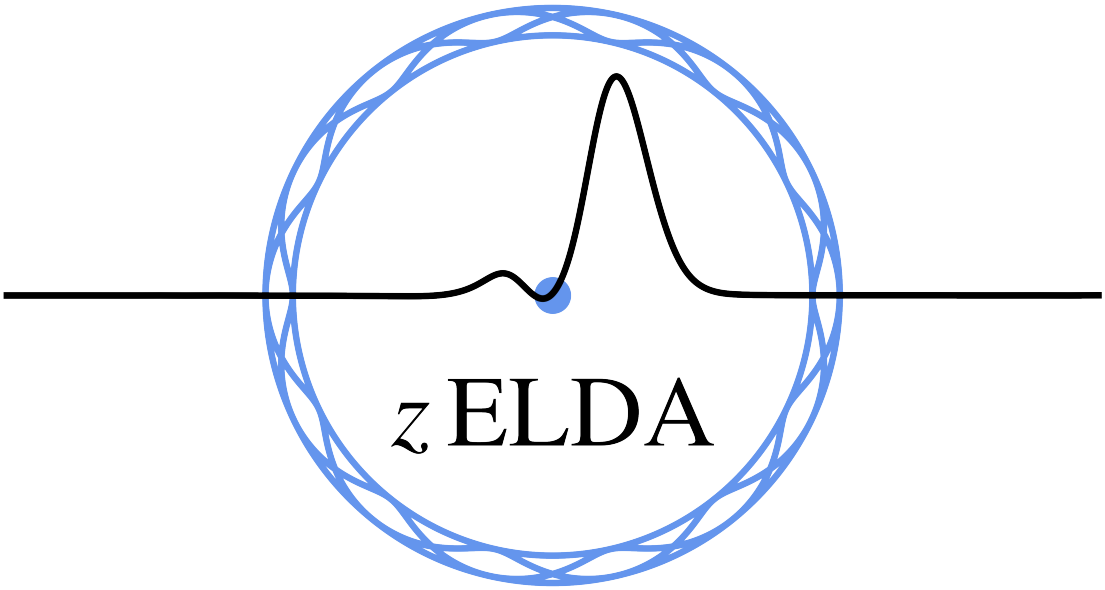
zELDA

Release 0.0.01

Gurung-Lopez, Siddhartha

Oct 27, 2021

CONTENTS:


zELDA

CHAPTER

ONE

INTRODUCTION

zELDA [redshift (z) Estimator using Line profiles of Distant lyman-Alpha emitters], a code to understand Lyman-alpha emission.

1.1 Authors

Siddhartha Gurung Lopez
Max Gronke
Alvaro Orsi
Silvia Bonoli
Shun Saito

1.2 Publication links

zELDA paper:

ADS : https://ui.adsabs.harvard.edu/abs/2021arXiv210901680G/abstract
arXiv : https://arxiv.org/abs/2109.01680

zELDA is based on its previous version, *FLaREON*. Please, if you used *zELDA* in your project, cite also *FLaREON*:

ADS : http://adsabs.harvard.edu/abs/2018arXiv181109630G
arXiv : https://arxiv.org/abs/1811.09630

1.3 Origins and motivation

The main goal of *zELDA* is to provide to the scientific community a common tool to analyze and model Lyman-alpha line profiles.

zELDA is a publicly available *python* package based on a RTMC (Orsi et al. 2012) and *FLaREON* (Gurung-Lopez 2019) able to fit observed Lyman-alpha spectrum and to predict large amounts of Lyman alpha line profiles and escape fractions with high accuracy. We designed this code hoping that it helps researches all over the world to get a better understanding of the Universe. In particular *zELDA* is divided in two main functionalites:

- **Mocking Lyman-alpha line profiles**. Due to the Lyman alpha Radiative Transfer large complexity, the efforts to understand Lyman-alpha emission moved from pure analytic studies to the so-called radiative transfer Monte Carlo (RTMC) codes that simulate Lyman alpha photons in arbitrary gas geometries. These codes provide useful information about the fraction of photons that manage to escape and the resulting Lyman alpha line profiles. The RTMC approach has probed very successful in reproducing the observed properties of Lyman-alpha emitters. *zELDA* contains several data grids of *LyaRT*, the RTMC described in Orsi et al. 2012 (https://github.com/aaorsi/LyaRT), from which Lyman-alpha line profiles are computed using lineal interpolation. This methodology allow us to predict line profiles with a high accuracy at a low computational cost. In fact, the time used by *zELDA* to predict a single line profiles y usually eight orders of magnitud smaller than the full radiative transfer analysis done by *LyaRT*. Additionally, in order to mock observed Lyman-alpha spectrum, *zELDA* also includes routines to mimik the artifacts induced by observations in the line profiles, such a finite spectral resolution or the wavelength binning.
- **Fitting observed Lyman-alpha line profiles**. The main update from *FLaREON* to *zELDA* is the inclusion of several fitting algotirhms to model observed Lyman-alhpa line profiles. On the basics, *zELDA* uses mock Lyman-alpha line profiles to fit observed spectrums in two main phasions :
- **Monte Carlo Markov Chain** : This is the most classic approach taken in the literaute (e.g. Gronke et al. 2017). *zELDA* implementation is powered by the public code *emcee* (https://emcee.readthedocs.io/en/stable/) by Daniel Foreman-Mackey et al. (2013).
- **Deep learning** : *zELDA* is the first open source code that uses machine learning to fit Lyman-alpha line profiles. *zELDA* includes some trained deep neural networks that predicts the best inflow/outflow model and redshift for a given observed line profile. This approach is about 3 orders of magnitud faster than the MCMC analysis and provides similar accuracies. This methodology will prove decesive in the upcoming years when tens of thousands of Lyman-alpha line profiles will be measure by instruments such as the James Webb Space Telescope. The neural network engine powering *zELDA* is *scikitlearn* (https://scikit-learn.org/stable/).

CHAPTER

TWO

INSTALLATION

zELDA, installation is divided in two blocks. First you will need to install the python package containing all the scritps. With this you can already use the Deep Neural Network methodologies to extract information from observed Lyman-alpha line profiles. The second block contains all the grids computed from *LyaRT*. These are necessary in order to compute line profiles and escape fractions for all the outflow geometries. As a consequence, the second block is mandatory to make MCMC analysis.

2.1 Python package

The simplest way of installing *zELDA*'s scripts is via pip:

```
$ pip install Lya_zelda
```

An alternative method to install *zELDA*'scripts is downloading the code from GitHub:

```
$ git clone https://github.com/sidgurun/Lya_zelda.git
$ cd Lya_zelda
$ pip install .
```

Remember that you can also add the tag `--user` , if necessary.

zELDA uses a specific version of *numpy* and *sci-kit-learn*. This means that most likely *pip* will try to change to those versions when you install *zELDA*. If you want to avoid this you can create a virtual environment, which is always useful to tests installations.

2.2 LyaRT data grids

Next, let's download the data grids necessary for generating mock Lyman-alpha line profiles as escape fractions. The data is stored at https://zenodo.org/record/4733518#.YJjw_y_Wf0c . Download the *Grids.zip* file. You can do this in different ways. The recomended method is using the commamnd *wget* or *curl*, which should be more stable. For example, for downloawing it with *curl*, you can do:

```
$ curl --cookie zenodo-cookies.txt "https://zenodo.org/record/4733518/files/Grids.zip?
↪download=1" --output Grids.zip
```

The download might take a while, as it is about 13Gb, so grab your fauvorite snack and be patient =D .

Other way of getting the data is going to the *zenodo* webpage and download it through your internet borwser. As this is a large file, if you brower is a little bit unstable the download might stop in halfway, causing you to restart the download again.

Once you have the *Grids.zip* file, unzip it in the place that you want to keep it.

In order to compute line profiles and escape fraction you will need to indicate *zELDA* the location of grids by doing

```
>>> import Lya_zelda as Lya

>>> your_grids_location = '/This/Folder/Contains/The/Grids/'

>>> Lya.funcs.Data_location = your_grids_location
```

where *your_grids_location* is a *string* with the place where you have stored the grids. If you run the *ls* command you should see something like this:

```
$ ls /This/Folder/Contains/The/Grids/

Dictonary_Bicone_X_Slab_Grid_Lines_In_Bicone_False.npy
.
.
.
GRID_data__V_29_logNH_19_logta_9_EW_20_Wi_31.npy
GRID_data__V_29_logNH_19_logta_9_EW_8_Wi_9.npy
GRID_info__V_29_logNH_19_logta_9_EW_20_Wi_31.npy
GRID_info__V_29_logNH_19_logta_9_EW_8_Wi_9.npy
.
.
.
finalized_model_wind_f_esc_Tree_f_esc.sav
```

You can check if you have set properly the directoy by loading a grid after setting *Lya.funcs.Data_location*, for example:

```
>>> print( Lya.Check_if_DATA_files_are_found() )
```

If the location has been properly set the command should return 1. If the data files are not found, then 0 is return. This function will also tell you the current value of *Lya.funcs.Data_location*. If the funtions returns 0 make sure than running *ls* gives you the expected output (see just above).

2.3 Partial installation for testing

This section is optional and not required for the full installation. If you have done the previous steps you don't need to go through this.

The full *zELDA* (grids+code) is about 13GB of storage. There could be the case in which you might want to test the code but not install it completely. If this is the case, you can download a lighter version of the grid for the Thin Shell geoemtry used to fit observed data. Remember that once you have installed the scripts by pip (above), you can already make the neural network analysis of the line profiles, there is no need of the line profiles grids. However, if you want to plot the line profile given by the predicted outflow propeties you will need the grid of line profiles.

Go to the location where you want to store the test grids. You can download the lighter version of the grids with

```
$ curl -0 --output GRID_data__V_29_logNH_19_logta_9_EW_8_Wi_9.npy  https://zenodo.org/
↪record/4890276/files/GRID_data__V_29_logNH_19_logta_9_EW_8_Wi_9.npy
$ curl -0 --output GRID_info__V_29_logNH_19_logta_9_EW_8_Wi_9.npy  https://zenodo.org/
↪record/4890276/files/GRID_info__V_29_logNH_19_logta_9_EW_8_Wi_9.npy
```

Done! This files should be less than 2GB.

Let's see how you can load them.

```
>>> import Lya_zelda as Lya

>>> your_grids_location = '/This/Folder/Contains/The/Grids/'

>>> Lya.funcs.Data_location = your_grids_location
```

where *your_grids_location* is a *string* with the place where you have stored the grids. If you run the *ls* command you should see something like this:

```
$ ls /This/Folder/Contains/The/Grids/
GRID_data__V_29_logNH_19_logta_9_EW_8_Wi_9.npy
GRID_info__V_29_logNH_19_logta_9_EW_8_Wi_9.npy
```

You can check if you have set properly the directoy by loading a grid after setting *Lya.funcs.Data_location*, for example:

```
>>> Geometry = 'Thin_Shell_Cont'

>>> LyaRT_Grid = Lya.load_Grid_Line( Geometry , MODE='LIGHT' )
```

If this last command worked, then the grids were found correctly and you can start using this line profile grid to test the creation of mock line profiles, for example. However, you won't be able to compute escape fractions and the line profile for the other gas geometries until you install the full package. Also, the grid you have just downlaoded is less heavy because there are fewer bins, which means that the nodes are more spaced. This means that the line profiles computed from this grid will have in general a lower accuracy in comparison with using the full grid. Therefore, for science you sould use the full grid, not this one.

CHAPTER

THREE

ABOUT THE LYART DATA GRIDS

Here we explain a little bit the data grid from *LyaRT*, the Radiative Monte Carlo Code described in Orsi et al. 2012 (https://github.com/aaorsi/LyaRT). These grids are the pillars of *zELDA*, so it is good to familiarized with them.

3.1 Getting started

Let's start by loading *zELDA* and setting the location of the LyaRT grids:

```
>>> import Lya_zelda as Lya

>>> your_grids_location = '/This/Folder/Contains/The/Grids/'

>>> Lya.funcs.Data_location = your_grids_location
```

where */This/Folder/Contains/The/Grids/* is the place where you store the LyaRT data grids, as shown in the *Installation* section.

3.2 Line profile LyaRT data grids

Let's load a data grid to start working. In particular we are going to load the one for the 'Thin_Shell' geometry. This geometry has 3 variables for the outflow configuration: expansion velocity, HI column density and dust optical depth.

```
>>> LyaRT_Grid = Lya.load_Grid_Line( 'Thin_Shell' )
```

LyaRT_Grid is a python dictionary containing all the data necessary for the interpolation. Let's look to the keys

```
>>> print( LyaRT_Grid.keys() )

dict_keys(['logNH_Arr', 'logta_Arr', 'Grid', 'x_Arr', 'V_Arr'])
```

The variables *'V_Arr'*, *'logNH_Arr'* and *'logta_Arr'* are 1-D numpy arrays that contains the values in which the grid is evaluated for the expansion velocity, the logarithmic of the HI column density and the logarithmic of the dust optical depth respectively. If you want to check where the grid is evaluated you can do

```
>>> print( 'The expansion velocity [km/s] is evaluated in : ')
>>> print( LyaRT_Grid['V_Arr'    ] )

>>> print( 'The logarithmic of the HI column density [cm**-2] is evaluated in : ')
>>> print( LyaRT_Grid['logNH_Arr'] )
```

(continues on next page)

(continued from previous page)

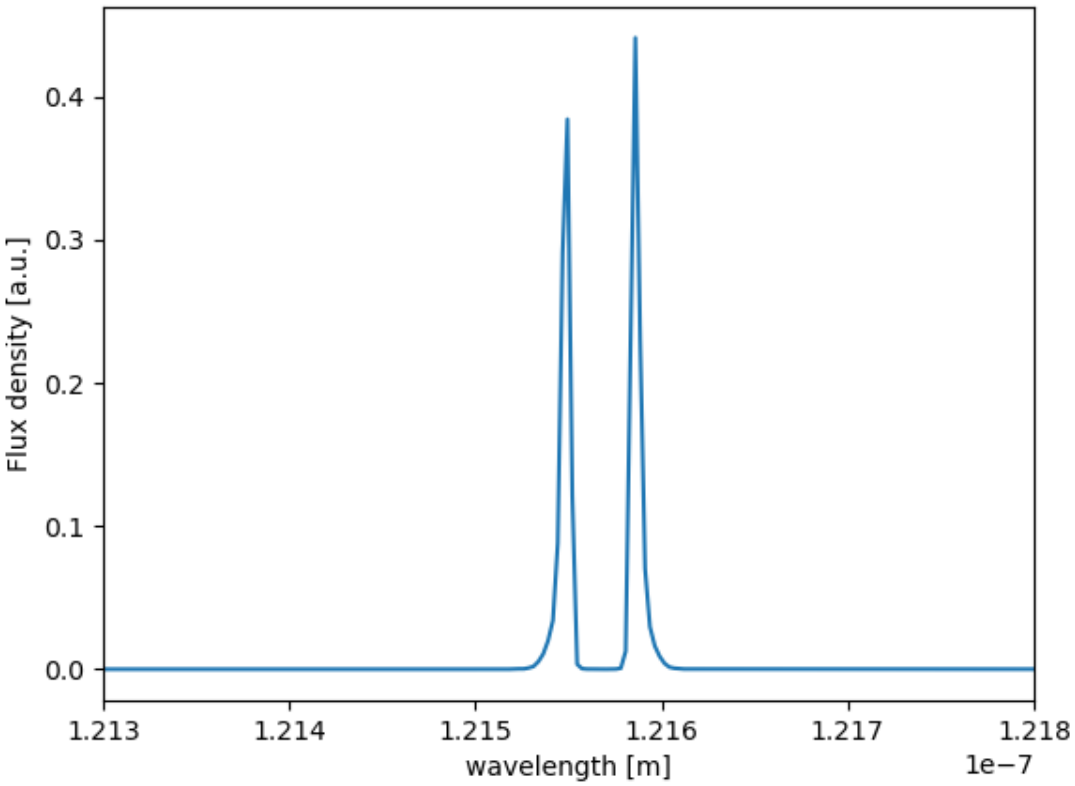
```
>>> print( 'The logarithmic of the dust optical depth is evaluated in : ')
>>> print( LyaRT_Grid['logta_Arr'] )

The expansion velocity [km/s] is evaluated in :
[   0   10   20   30   40   50   60   70   80   90  100  150  200  250
  300  350  400  450  500  550  600  650  700  750  800  850  900  950
 1000]
The logarithmic of the HI column density [cm**-2] is evaluated in :
[17.   17.25 17.5  17.75 18.   18.25 18.5  18.75 19.   19.25 19.5  19.75
 20.   20.25 20.5  20.75 21.   21.25 21.5  21.75 22.  ]
The logarithmic of the dust optical depth is evaluated in :
[-3.75  -3.5   -3.25  -3.    -2.75  -2.5   -2.25  -2.    -1.75  -1.5
 -1.375 -1.25  -1.125 -1.    -0.875 -0.75  -0.625 -0.5   -0.375 -0.25
 -0.125]
```

Then, *LyaRT_Grid['Grid']* are the line profiles in each of the nodes of the 3-D grid. For example, *LyaRT_Grid['Grid'][0,1,2]* is the line profile with *LyaRT_Grid['V_Arr'][0]*, *LyaRT_Grid['logNH_Arr'][1]* and *LyaRT_Grid['logta_Arr'][2]*. This spectrum is evaluated in *LyaRT_Grid['x_Arr']*, that is the frequency in Doppler units. You can convert from frequency in Doppler units to wavelength by doing:

```
>>> w_Arr = Lya.convert_x_into_lamda( LyaRT_Grid['x_Arr'] )
```

w_Arr is a 1-D array with the wavelengths in meters. Let's take a look to the spectrum:

```
>>> import pylab as plt
>>> plt.plot( w_Arr , LyaRT_Grid['Grid'][0,1,2] )
>>> plt.xlim( 1213*1e-10 , 1218*1e-10 )
>>> plt.xlabel( 'wavelength [m]' )
>>> plt.ylabel( 'Flux density [a.u.]' )
>>> plt.show()
```



3.3 Line profile grids with smaller RAM occupation

The data grids for the geometries *'Thin_Shell'*, *'Galactic_Wind'*, *'Bicone_X_Slab_In'* and *'Bicone_X_Slab_Out'* are relatively small and they occupy less than 1GB of RAM. These models have 3 dimensions: expansion velocity, HI column density and dust optical depth. However, the model *'Thin_Shell_Cont'* includes different intrinsic line profiles, which increases the number of dimensions to 5. This increase a lot the data volume, in terms of parameter space and RAM occupation. Indeed, the default *'Thin_Shell_Cont'* line profile grid is about 11GB. This means that when using this mode you would need to have 11GB of RAM or more. In case that you want to do some tests with a smaller grid (but still 5D) we have included a lighter grid, that is about 2GB of size.

You can load the default *'Thin_Shell_Cont'* by doing

```
>>> import Lya_zelda as Lya

>>> your_grids_location = '/This/Folder/Contains/The/Grids/'

>>> Lya.funcs.Data_location = your_grids_location
```

where */This/Folder/Contains/The/Grids/* is the place where you store the LyaRT data grids, as shown in the *Installation* section.

```
>>> LyaRT_Grid_Full = Lya.load_Grid_Line( 'Thin_Shell_Cont' )
```

or

```
>>> LyaRT_Grid_Full = Lya.load_Grid_Line( 'Thin_Shell_Cont' , MODE='FULL' )
```

And you can see where the grid is evaluated by doing

```
>>> print( 'The expansion velocity [km/s] is evaluated in : ')
>>> print( LyaRT_Grid_Full['V_Arr'] )

>>> print( 'The logarithmic of the HI column density [cm**-2] is evaluated in : ')
>>> print( LyaRT_Grid_Full['logNH_Arr'] )

>>> print( 'The logarithmic of the dust optical depth is evaluated in : ')
>>> print( LyaRT_Grid_Full['logta_Arr'] )

>>> print( 'The logarithmic of the intrinsic equivalent width [A] is evaluated in : ')
>>> print( LyaRT_Grid_Full['logEW_Arr'] )

>>> print( 'The logarithmic of the intrinsic line width [A] is evaluated in : ')
>>> print( LyaRT_Grid_Full['Wi_Arr'] )

The expansion velocity [km/s] is evaluated in :
[   0   10   20   30   40   50   60   70   80   90  100  150  200  250
  300  350  400  450  500  550  600  650  700  750  800  850  900  950
 1000]
The logarithmic of the HI column density [cm**-2] is evaluated in :
[17.   17.25 17.5  17.75 18.   18.25 18.5  18.75 19.   19.25 19.5  19.75
 20.   20.25 20.5  20.75 21.   21.25 21.5 ]
The logarithmic of the dust optical depth is evaluated in :
[-4.  -3.5 -3.  -2.5 -2.  -1.5 -1.  -0.5  0. ]
The logarithmic of the intrinsic equivalent width [A] is evaluated in :
[-1.         -0.78947368 -0.57894737 -0.36842105 -0.15789474  0.05263158
  0.26315789  0.47368421  0.68421053  0.89473684  1.10526316  1.31578947
  1.52631579  1.73684211  1.94736842  2.15789474  2.36842105  2.57894737
  2.78947368  3.        ]
The logarithmic of the intrinsic line width [A] is evaluated in :
[0.01 0.05 0.1  0.15 0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9  1.   1.2
 1.4  1.6  1.8  2.   2.2  2.4  2.6  2.8  3.   3.25 3.5  3.75 4.   5.25
 5.5  5.75 6.  ]
```

Now let's load the lighter grid for *'Thin_Shell_Cont'*,

```
>>> LyaRT_Grid_Light = Lya.load_Grid_Line( 'Thin_Shell_Cont' , MODE='LIGHT' )
```

The reduction of the size of the grid is done by reducing the number of bins in *'logEW_Arr'* and *'Wi_Arr'*. You can see the new *'logEW_Arr'* and *'Wi_Arr'* arrays in:

```
>>> print( 'The logarithmic of the intrinsic equivalent width [A] is evaluated in : ')
>>> print( LyaRT_Grid_Light['logEW_Arr'] )

>>> print( 'The logarithmic of the intrinsic line width [A] is evaluated in : ')
>>> print( LyaRT_Grid_Light['Wi_Arr'] )

The logarithmic of the intrinsic equivalent width [A] is evaluated in :
[-1.   0.   0.4  0.8  1.2  1.6  2.   3. ]
```

(continues on next page)

(continued from previous page)

```
The logarithmic of the intrinsic line width [A] is evaluated in :
[0.01 0.05 0.1  0.25 0.5  1.   2.   4.   6.  ]
```

If you want a smaller custom grid, you can build your own data grid by selecting nodes from *LyaRT_Grid_Full*. As long as you keep the format of *LyaRT_Grid_Full*, you will be able to pass your custom grids to the algorithms. Just as a short advice, it would be beneficial in you keep the very extremes in the evaluation arrays (for example, *LyaRT_Grid_Full['V_Arr'][0]* and *LyaRT_Grid_Full['V_Arr'][-1]*) in your new custom grid.

CHAPTER

FOUR

TUTORIAL : COMPUTING IDEAL LINE PROFILES

In this tutorial you will, hopefully, learn how to compute ideal line Lyman-alpha line profiles with *zELDA*. The lines computed in this tutorial are ideal becase they don't suffer from the typical artifacts caused by the fact the instruments are not perfect. These lines are in the rest frame of the galaxy.

4.1 Computing one ideal line profile

Let's start by loading *zELDA* and setting the location of the LyaRT grids:

```
>>> import Lya_zelda as Lya

>>> your_grids_location = '/This/Folder/Contains/The/Grids/'

>>> Lya.funcs.Data_location = your_grids_location
```

where */This/Folder/Contains/The/Grids/* is the place where you store the LyaRT data grids, as shown in the installation section.

Now, let's decide which outflow geometry we want to use. For this tutorial we will use the gas geometry known as Thin Shell in which the intrinsic continuum is a gaussian and a continuum with a give equivalent width.

```
>>> Geometry = 'Thin_Shell_Cont'
```

Let's load the data containing the grid:

```
>>> LyaRT_Grid = Lya.load_Grid_Line( Geometry )
```

This contains all the necessary information to compute the line profiles. To learn more about the grids of line profiles go to *Installation* . Remeber that if you want to use the line profile grid with lower RAM memory occupation you must pass *MODE='LIGHT'* to *Lya.load_Grid_Line*.

Now let's define the parameters of the shell model that we want. these are five:

```
>>> V_Value     = 50.0  # Outflow expansion velocity [km/s]
>>> logNH_Value = 20.   # Logarithmic of the neutral hydrogen column density [cm**-2]
>>> ta_Value    = 0.01  # Dust optical depth
>>> logEW_Value = 1.5   # Logarithmic the intrinsic equivalent width [A]
>>> Wi_Value    = 0.5   # Intrinsic width of the line [A]
```

Now, let's set the wavelength array where we want to put the line in the international system of units (meters). We arbitrarily chose to evaluate the line +-10A around Lyman-alpha:

```
>>> import numpy as np
>>> w_Lya = 1215.68 # Lyman-alpha wavelength in amstrongs
>>> wavelength_Arr = np.linspace( w_Lya-10 , w_Lya+10 , 1000 ) * 1e-10
```

Now he have everything, let's compute the line simply by doing:

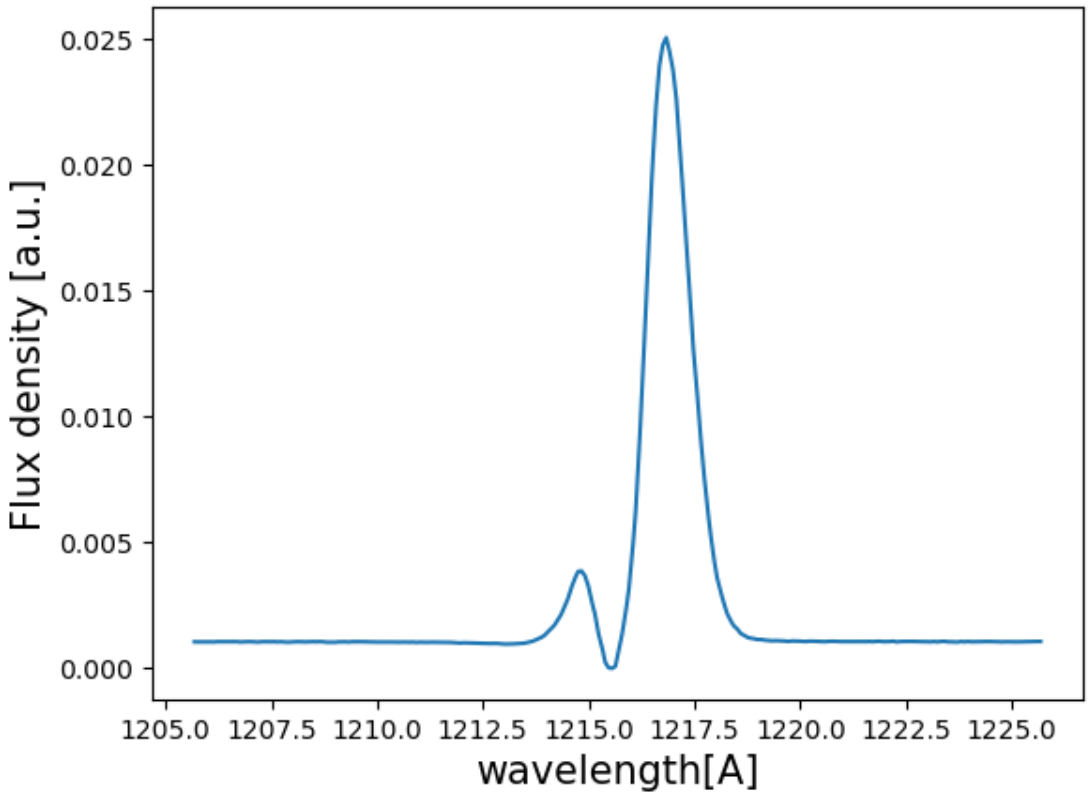
```
>>> Line_Arr = Lya.RT_Line_Profile_MCMC( Geometry , wavelength_Arr , V_Value , logNH_
↪Value , ta_Value , LyaRT_Grid , logEW_Value=logEW_Value , Wi_Value=Wi_Value )
```

And... It's done! *Line_Arr* is a numpy array that contains the line profile evaluated in *wavelength_Arr*.

Let's plot the line by doing

```
>>> import pylab as plt
>>> plt.plot( wavelength_Arr *1e10 , Line_Arr )
>>> plt.xlabel('wavelength[A]' , size=15 )
>>> plt.ylabel('Flux density [a.u.]' , size=15 )
>>> plt.show()
```

This should show something like this



4.2 Computing many ideal line profile

Above we have just seen how to compute one ideal line profile. In the case that you want to compute several *zELDA* has a more compact function.

Let's start like in the case above in which we set the location of the grids:

```
>>> import Lya_zelda as Lya

>>> your_grids_location = '/This/Folder/Contains/The/Grids/'

>>> Lya.funcs.Data_location = your_grids_location
```

where */This/Folder/Contains/The/Grids/* is the place where you store the LyaRT data grids, as shown in the installation section.

Now, let's set the geometry:

```
>>> Geometry = 'Thin_Shell_Cont'
```

And now, instead of loading the grid, let's define the outflow parameters. In this case they will be lists (or numpy arrays) as we want, for example 3 line profile configurations:

```
>>> V_Arr     = [ 50.0 , 100.   , 200.    ] # Outflow expansion velocity [km/s]
>>> logNH_Arr = [ 18.  ,  19.   ,  20.    ] # Logarithmic of the neutral hydrogen column␣
↪density [cm**-2]
>>> ta_Arr    = [  0.1 ,   0.01 ,   0.001 ] # Dust optical depth
>>> logEW_Arr = [  1.  ,   1.5  ,   2.0   ] # Logarithmic the intrinsic equivalent width␣
↪[A]
>>> Wi_Arr    = [  0.1 ,   0.5  ,   1.0   ] # Intrinsic width of the line [A]
```

and the wavelength array

```
>>> import numpy as np
>>> w_Lya = 1215.68 # Lyman-alpha wavelength in amstrongs
>>> wavelength_Arr = np.linspace( w_Lya-10 , w_Lya+10 , 1000 ) * 1e-10
```

Now let's actually compute the lines:

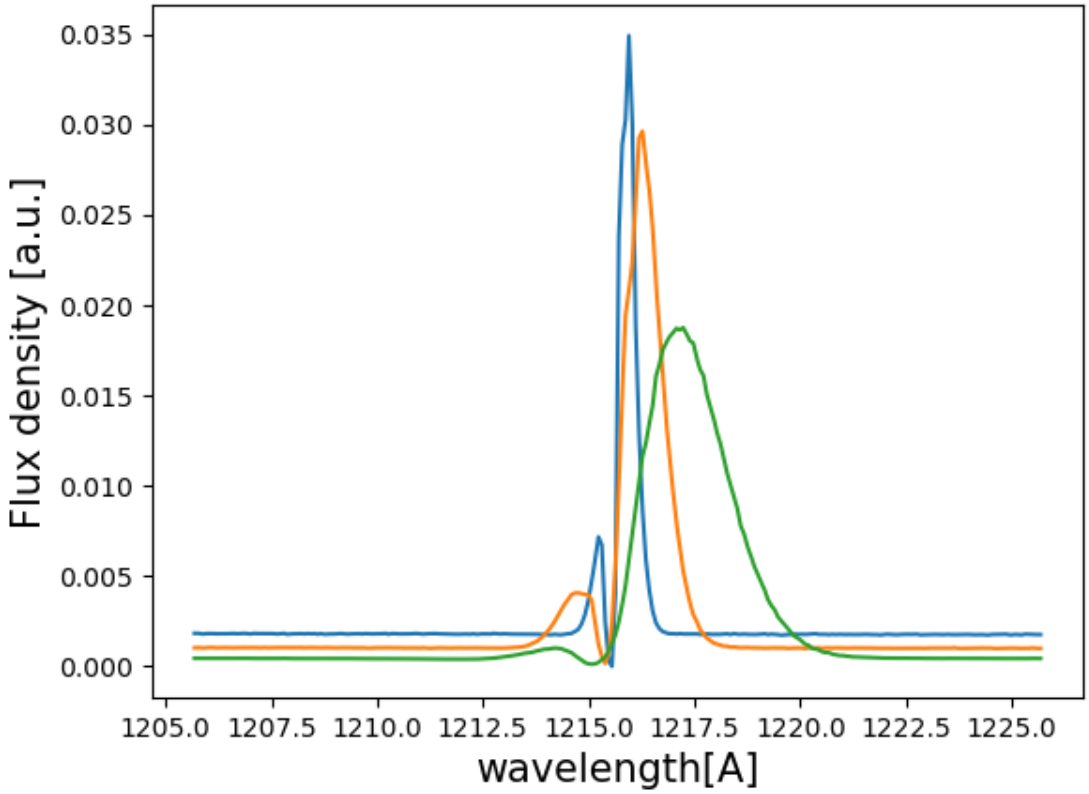
```
>>> Line_Matrix = Lya.RT_Line_Profile( Geometry , wavelength_Arr , V_Arr , logNH_Arr ,␣
↪ta_Arr , logEW_Arr=logEW_Arr , Wi_Arr=Wi_Arr )
```

Line_Matrix is a 2-D numpy array containing the line profiles for the configurations. For example, *Line_Matrix[0]* has outflow velocity *V_Arr[0]*, neutral hydrogen column density *logNH_Arr[0]* and so on.

Let's plot them:

```
>>> import pylab as plt

>>> for i in range( 0 , 3 ) :
>>>     plt.plot( wavelength_Arr *1e10 , Line_Matrix[i] )

>>> plt.xlabel('wavelength[A]'       , size=15 )
>>> plt.ylabel('Flux density [a.u.]' , size=15 )
>>> plt.show()
```

This should show something like this:



Now you know how to get ideal Lyman-alpha line profiles!

CHAPTER

FIVE

TUTORIAL : COMPUTING MOCK LINE PROFILES

In this tutorial you will, hopefully, learn how to compute mock line Lyman-alpha line profiles with *zELDA*. The lines computed in this tutorial suffer from the typical artifacts caused by the fact the instruments are not perfect.

5.1 Mocking Lyman-alpha line profiles

Let's start by loading *zELDA* and setting the location of the LyaRT grids:

```
>>> import Lya_zelda as Lya

>>> your_grids_location = '/This/Folder/Contains/The/Grids/'

>>> Lya.funcs.Data_location = your_grids_location
```

where */This/Folder/Contains/The/Grids/* is the place where you store the LyaRT data grids, as shown in the *Installation* section.

Now, let's decide which outflow geometry we want to use. For this tutorial we will use the gas geometry known as Thin Shell in which the intrinsic continuum is a gaussian and a continuum with a give equivalent width.

```
>>> Geometry = 'Thin_Shell_Cont'
```

Let's load the data containing the grid:

```
>>> LyaRT_Grid = Lya.load_Grid_Line( Geometry )
```

This contains all the necessary information to compute the line profiles. To learn more about the grids of line profiles go to *About the LyaRT data grids* . Remeber that if you want to use the line profile grid with lower RAM memory occupation you must pass *MODE='LIGHT'* to *Lya.load_Grid_Line*.

Now let's define the parameters of the shell model that we want. these are five:

```
>>> z_t      = 0.5   # redshift of the source
>>> V_t      = 50.0  # Outfloe expansion velocity [km/s]
>>> log_N_t  = 20.   # Logarithmic of the neutral hydrogen column density [cm**-2]
>>> t_t      = 0.01  # Dust optical depth
>>> log_EW_t = 1.5   # Logarithmic the intrinsic equivalent width [A]
>>> W_t      = 0.5   # Intrinsic width of the line [A]
>>> F_t      = 1.    # Total flux of the line
```

Now let's set the quality of the line profile:

```
>>> PNR_t  = 10.0 # Signal to noise ratio of the maximum of the line.
>>> FWHM_t = 0.5  # Full width half maximum diluting the line. Mimics finite resolution.␣
↪[A]
>>> PIX_t  = 0.2  # Wavelength binning of the line. [A]
```

Now he have everything, let's compute the line simply by doing:

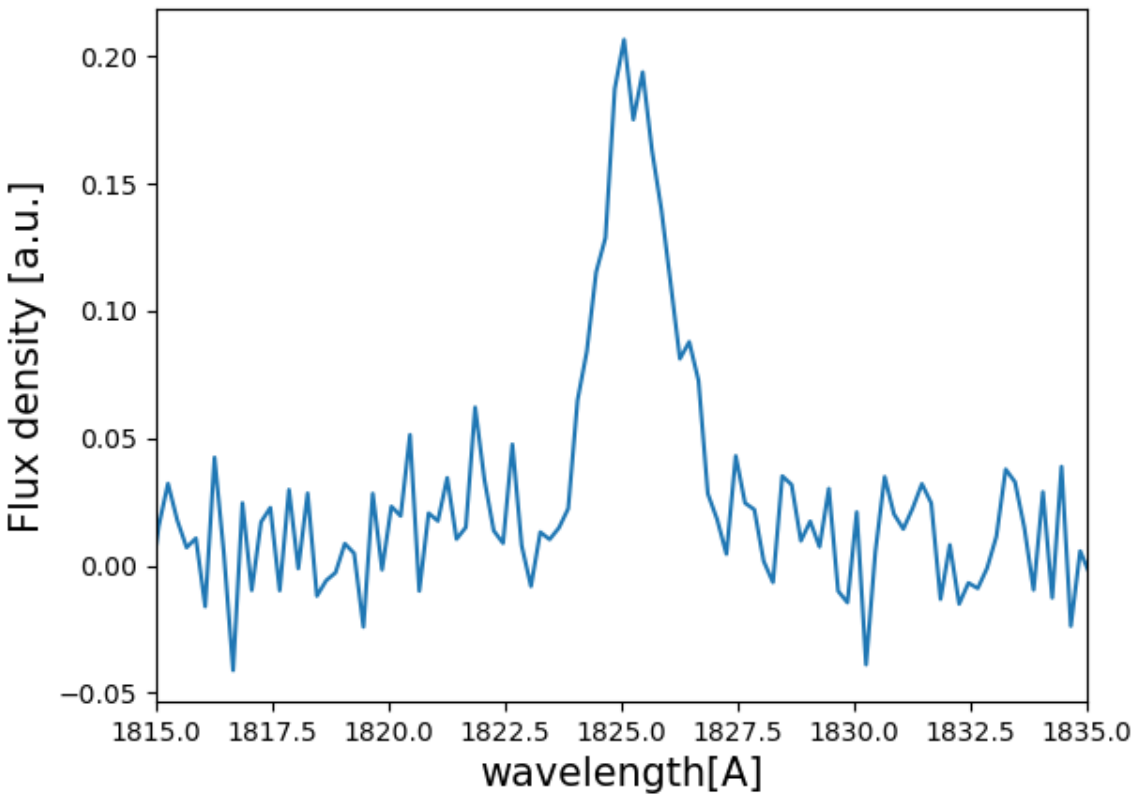
```
>>> w_Arr , f_Arr , _ = Lya.Generate_a_real_line( z_t , V_t, log_N_t, t_t, F_t, log_EW_t,
↪ W_t , PNR_t, FWHM_t, PIX_t, LyaRT_Grid, Geometry )
```

And... It's done! *w_Arr* is a numpy array that contains the wavelength where the line profile is evaluated. Meanwhile, *f_Arr* is the actual line profile.

Let's plot the line by doing

```
>>> import pylab as plt
>>> plt.plot( w_Arr , f_Arr )
>>> plt.xlabel('wavelength[A]' , size=15 )
>>> plt.ylabel('Flux density [a.u.]' , size=15 )
>>> plt.xlim(1815,1835)
>>> plt.show()
```

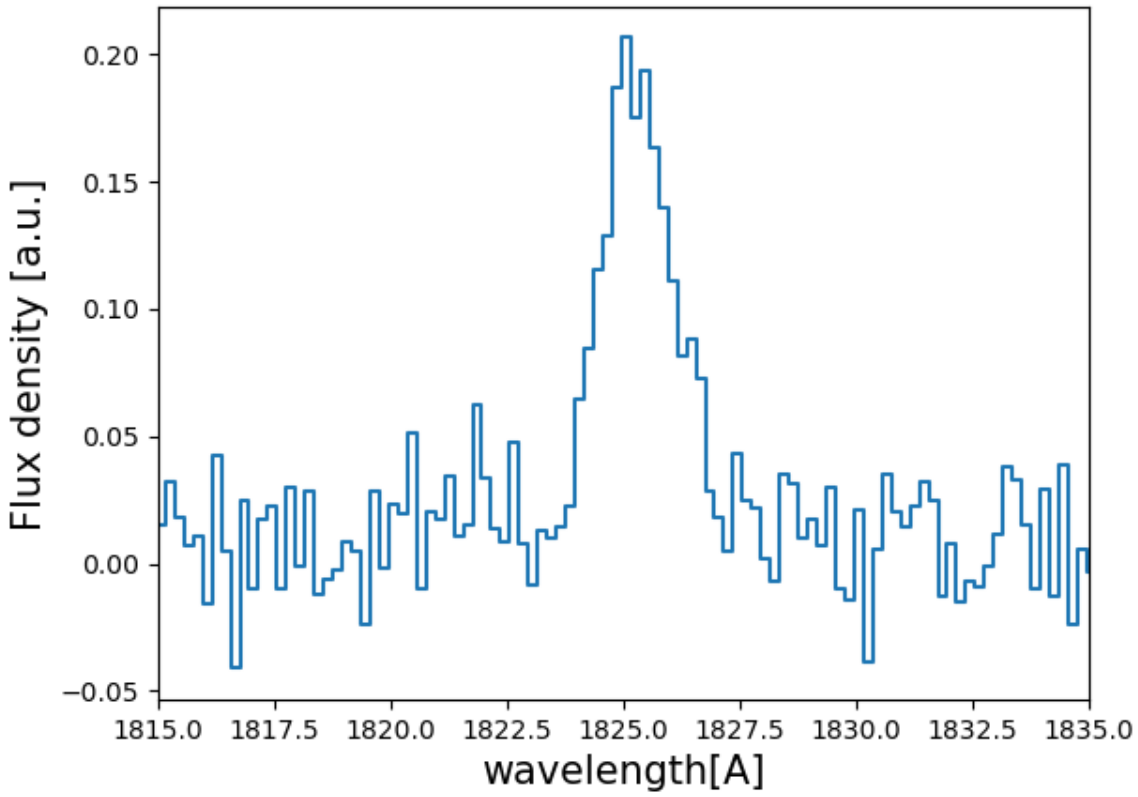
This should show something like this



5.2 Plotting cooler line profiles

If you want a cooler and more 'accurate' plot of the line profile you can use:

```
>>> w_pix_Arr , f_pix_Arr = Lya.plot_a_rebinned_line( w_Arr , f_Arr , PIX_t )
>>> plt.plot( w_pix_Arr , f_pix_Arr )
>>> plt.xlabel('wavelength[A]' , size=15 )
>>> plt.ylabel('Flux density [a.u.]' , size=15 )
>>> plt.xlim(1815,1835)
>>> plt.show()
```



Lya.plot_a_rebinned_line is just a function that returns the line profile and wavelength array in a cool way to plot them. You probably shouldn't use for science the output of *Lya.plot_a_rebinned_line*, just for plotting.

CHAPTER

SIX

TUTORIAL : FITTING A LINE PROFILE USING DEEP LEARNING

In this tutorial you will, hopefully, learn how fit Lyman-alpha line profiles using deep learning with *zELDA*.

6.1 Getting started

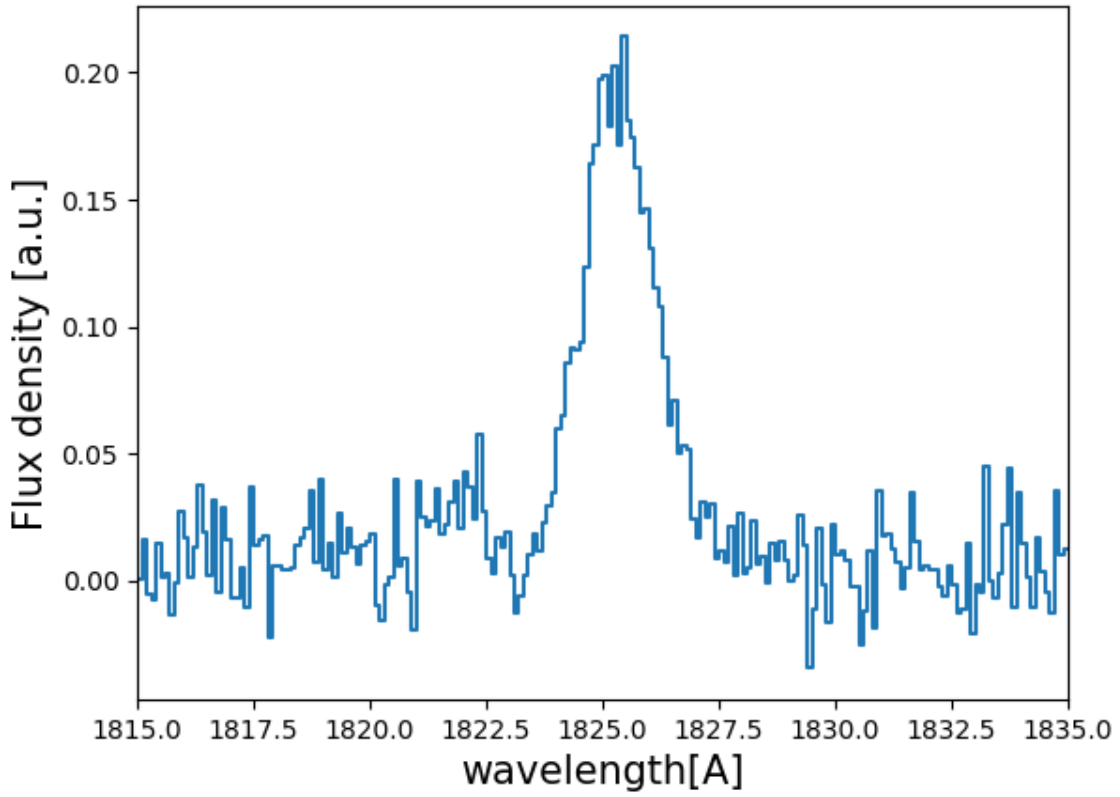
Let's start by loading *zELDA* creating a mock line profile that we will fit later. For more details on how to create a mock line profile go to *Mock line profiles*

```
>>> import Lya_zelda as Lya
>>> your_grids_location = '/This/Folder/Contains/The/Grids/'
>>> Lya.funcs.Data_location = your_grids_location

>>> Geometry = 'Thin_Shell_Cont'
>>> LyaRT_Grid = Lya.load_Grid_Line( Geometry )

>>> # Defining the model parameters:
>>> z_t      = 0.5   # redshift of the source
>>> V_t      = 50.0  # Outflow expansion velocity [km/s]
>>> log_N_t  = 20.   # Logarithmic of the neutral hydrogen column density [cm**-2]
>>> t_t      = 0.01  # Dust optical depth
>>> log_EW_t = 1.5   # Logarithmic the intrinsic equivalent width [A]
>>> W_t      = 0.5   # Intrinsic width of the line [A]
>>> F_t      = 1.    # Total flux of the line

>>> # Defining the quality of the line profile:
>>> PNR_t  = 15.0 # Signal to noise ratio of the maximum of the line.
>>> FWHM_t = 0.2  # Full width half maximum diluting the line. Mimics finite resolution.␣
↪[A]
>>> PIX_t  = 0.1  # Wavelength binning of the line. [A]

>>> w_Arr , f_Arr , s_Arr = Lya.Generate_a_real_line( z_t , V_t, log_N_t, t_t, F_t, log_
↪EW_t, W_t , PNR_t, FWHM_t, PIX_t, LyaRT_Grid, Geometry )
```

where */This/Folder/Contains/The/Grids/* is the place where you store the LyaRT data grids, as shown in the installation section. And... It's done! *w_Arr* is a numpy array that contains the wavelength where the line profile is evaluated. Meanwhile, *f_Arr* is the actual line profile. *s_Arr* is the uncertainty of the flux density. Remeber that if you want to use the line profile grid with lower RAM memory occupation you must pass *MODE='LIGHT'* to *Lya.load_Grid_Line*.

Let's have a look to how the line looks:

```
>>> w_Arr , f_Arr , s_Arr  = Lya.Generate_a_real_line( z_t , V_t, log_N_t, t_t, F_t, log_
↪EW_t, W_t , PNR_t, FWHM_t, PIX_t, LyaRT_Grid, Geometry )

>>> w_pix_Arr , f_pix_Arr = Lya.plot_a_rebinned_line( w_Arr , f_Arr , PIX_t )

>>> import pylab as plt
>>> plt.plot( w_pix_Arr , f_pix_Arr )
>>> plt.xlabel('wavelength[A]' , size=15 )
>>> plt.ylabel('Flux density [a.u.]' , size=15 )
>>> plt.xlim(1815,1835)
>>> plt.show()
```



Now that we have our mock line profile. Let's load the neural network. As we have produce a line profile for an outflow (*V_t*>0) we are going to load the deep neural network for outflows

```
>>> machine_data =  Lya.Load_NN_model( 'Outflow' )
```

In case you want to do the analysis for inflows just call *Lya.Load_NN_model('Inflow')*. *machine_data* is a python dictionary that contains all the necessary data for the deep neural network approach. Let's pick up from it two variables:

```
>>> machine    = machine_data['Machine' ]
>>> w_rest_Arr = machine_data[ 'w_rest' ]
```

machine is an object from skitlearn with the trained neural network and *w_rest_Arr* is the rest frame wavelength where the line profiles used for the training were evaluated. *w_rest_Arr* is important to check that the neural networks is

working in the same wavelength array that the line profiles will be evaluated. In principle you don't have to do anything with *w_rest_Arr*, but we need to pass it to other functions.

6.2 Using the DNN in the un-perturbed line profile

Let's start by simple evaluating the DNN using the mock line profile without perturbing it:

```
>>> Sol , z_sol = Lya.NN_measure( w_Arr , f_Arr , s_Arr , FWHM_t , PIX_t , machine , w_
↪rest_Arr , N_iter=None )
```

Done! . *Sol* is a matrix that contains the prediction by the DNN and *z_sol* is the predicted redshift. You can print the predicted properties doing:

```
>>> print( 'The measured redshift                                                    is
↪' , z_sol    )
>>> print( 'The measured logarithm of the expasion velocity                         is
↪' , Sol[0,1] )
>>> print( 'The measured logarithm of the HI column density                         is
↪' , Sol[0,2] )
>>> print( 'The measured logarithm of the dust optical depth                        is
↪' , Sol[0,3] )
>>> print( 'The measured logarithm of the intrinsic equivalent width               is
↪' , Sol[0,4] )
>>> print( 'The measured logarithm of the intrinsic            width               is
↪' , Sol[0,5] )
>>> print( 'The measured shift of the true Lya wavelgnth from the maximum of the line is
↪' , Sol[0,0] )
```

This should give something like

```
The measured redshift                                                    is 0.
↪49994403239322693
The measured logarithm of the expasion velocity                          is 1.
↪5821419036064905
The measured logarithm of the HI column density                          is 20.
↪149247231711733
The measured logarithm of the dust optical depth                         is -3.
↪310662004999448
The measured logarithm of the intrinsic equivalent width                 is 1.
↪458352960574508
The measured logarithm of the intrinsic            width                 is -0.
↪804093047888869
The measured shift of the true Lya wavelgnth from the maximum of the line is -1.
↪2773994188976223
```

Let's see how this new spectrum compares with the target:

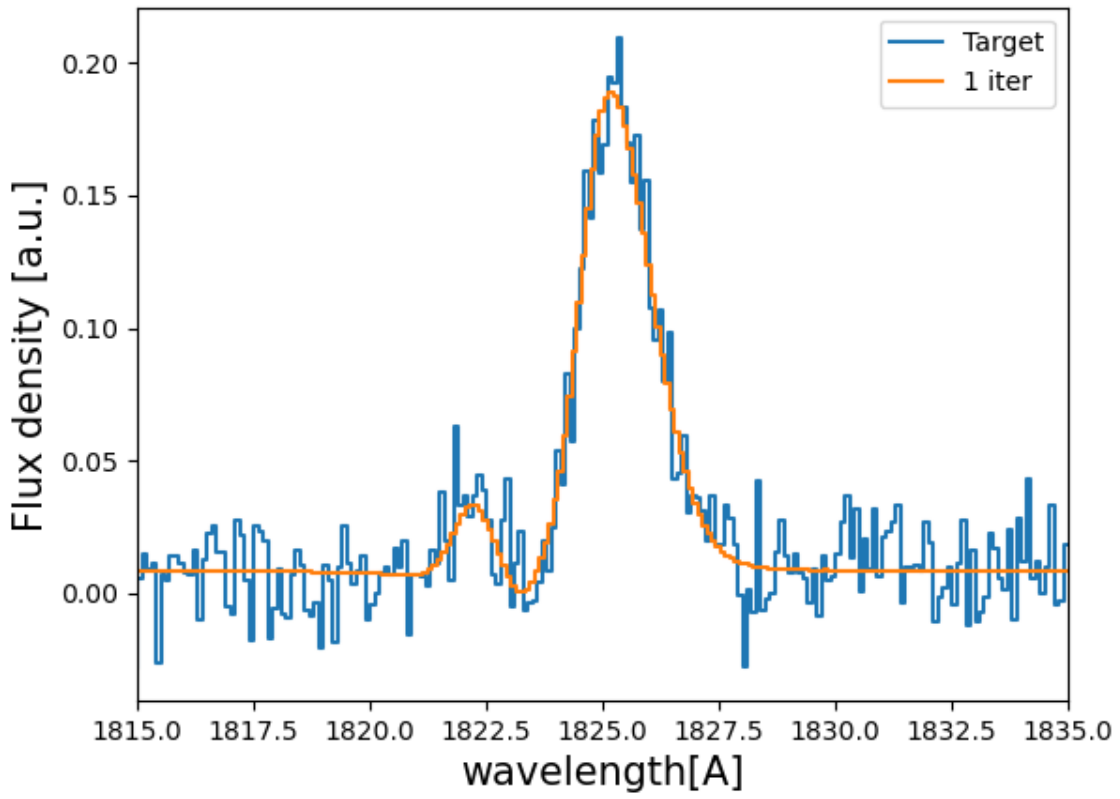
```
>>> PNR = 100000. # let's put infinite signal to noise in the model line

>>> V_sol    = 10**Sol[0,1] # Expansion velocity km/s
>>> logN_sol =    Sol[0,2] # log of HI column density cm**-2
>>> t_sol    = 10**Sol[0,3] # dust optical depth
```

(continues on next page)

(continued from previous page)

```
>>> logE_sol =     Sol[0,4] # log intrinsic EW [A]
>>> W_sol    = 10**Sol[0,5] # intrinsic width [A]

# creates the line

>>> w_One_Arr , f_One_Arr , _  = Lya.Generate_a_real_line( z_sol , V_sol, logN_sol, t_
↪sol, F_t, logE_sol, W_sol, PNR, FWHM_t, PIX_t, LyaRT_Grid, Geometry )

# plot the target and the predicted line

>>> w_pix_One_Arr , f_pix_One_Arr = Lya.plot_a_rebinned_line( w_One_Arr , f_One_Arr ,␣
↪PIX_t )

>>> plt.plot( w_pix_Arr     , f_pix_Arr     , label='Target' )
>>> plt.plot( w_pix_One_Arr , f_pix_One_Arr , label='1 iter' )

>>> plt.legend(loc=0)
>>> plt.xlabel('wavelength[A]' , size=15 )
>>> plt.ylabel('Flux density [a.u.]' , size=15 )
>>> plt.xlim(1815,1835)
>>> plt.show()
```

You should get something like:



6.3 Using the DNN with Monte Carlo perturbations

Normally, it is better to do more than one iteration, as it leads to better results. These iterations basically perturb the flux density *f_Arr* by adding gaussian noise with the amplitude of *s_Arr* in each wavelength bin. Then, this new perturbed spectrum is send to the DNN. For each of these iterations the output of the DNN is stored. For example for 1000 iterations :

```
>>> Sol , z_sol , log_V_Arr , log_N_Arr , log_t_Arr , z_Arr , log_E_Arr , log_W_Arr =␣
↪Lya.NN_measure( w_Arr , f_Arr , s_Arr , FWHM_t , PIX_t , machine , w_rest_Arr , N_
↪iter=1000 )
```

The arrays *log_V_Arr*, *log_N_Arr*, *log_t_Arr*, *z_Arr*, *log_E_Arr* and *log_W_Arr* contain the output of the DNN for the iterations for the logarithms of the expansion velocity, the logarithm of the neutral hydrogen column density, the logarithm of the dust optical depth, the redshift, the logarithm of the intrinsic equivalent width and the logarithm of the intrinsic width of the line. From these arrays we can compute the result from the DNN analysis by taking the 50th percentile. The +-1 sigma uncertainty can be computed as the 16th and 84th percentile.

```
>>> import numpy as np

>>> # Redshitft
>>> z_50     = np.percentile(    z_Arr , 50 )
>>> z_16     = np.percentile(    z_Arr , 16 )
>>> z_84     = np.percentile(    z_Arr , 84 )

>>> # Expansion velocity
>>> V_50     = 10 ** np.percentile( log_V_Arr , 50 )
>>> V_16     = 10 ** np.percentile( log_V_Arr , 16 )
>>> V_84     = 10 ** np.percentile( log_V_Arr , 84 )

>>> # Logarithmic of HI column density
>>> log_N_50 = np.percentile( log_N_Arr , 50 )
>>> log_N_16 = np.percentile( log_N_Arr , 16 )
>>> log_N_84 = np.percentile( log_N_Arr , 84 )

>>> # Dust optical depth
>>> t_50     = 10 ** np.percentile( log_t_Arr , 50 )
>>> t_16     = 10 ** np.percentile( log_t_Arr , 16 )
>>> t_84     = 10 ** np.percentile( log_t_Arr , 84 )

>>> # Logarithmic of intrinsic equivalent width
>>> log_E_50 = np.percentile( log_E_Arr , 50 )
>>> log_E_16 = np.percentile( log_E_Arr , 16 )
>>> log_E_84 = np.percentile( log_E_Arr , 84 )

>>> # Intrinsic width
>>> W_50     = 10 ** np.percentile( log_W_Arr , 50 )
>>> W_16     = 10 ** np.percentile( log_W_Arr , 16 )
>>> W_84     = 10 ** np.percentile( log_W_Arr , 84 )
```

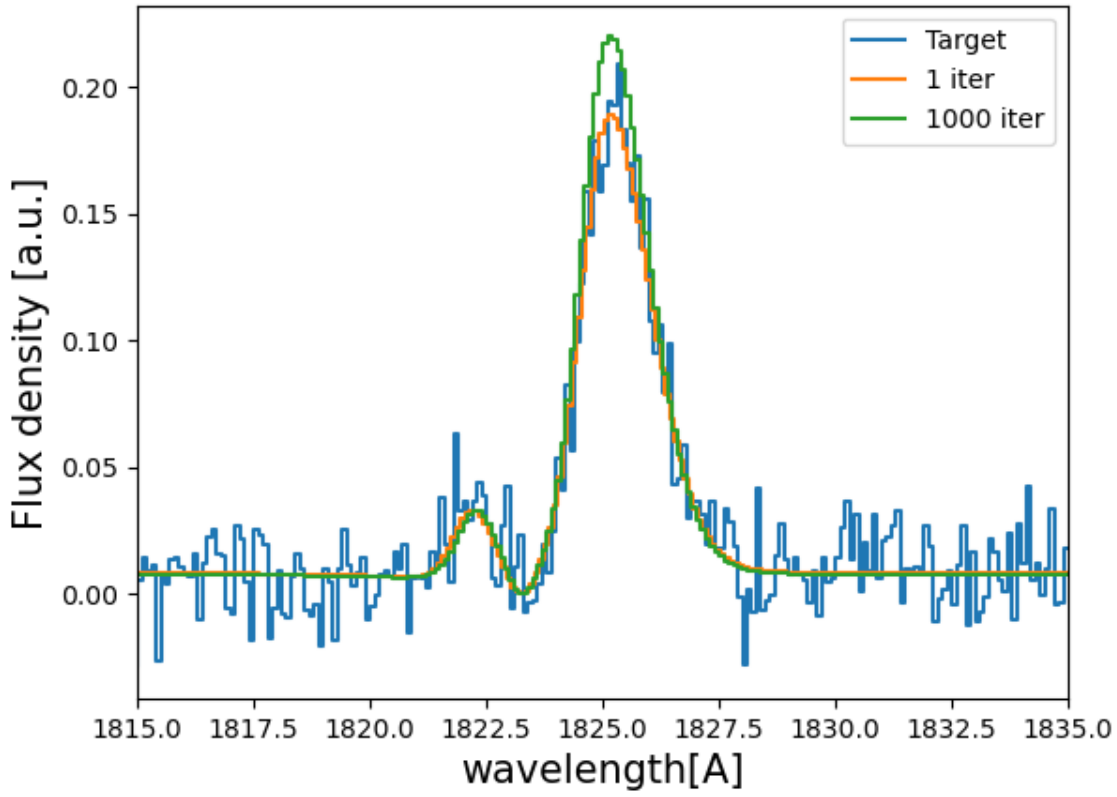
let's see how the line profiles look:

```
>>> # Compute the 100 iterations line profile
>>> w_50th_Arr , f_50th_Arr , _  = Lya.Generate_a_real_line( z_50 , V_50, log_N_50, t_50,
↪ F_t, log_E_50, W_50, PNR, FWHM_t, PIX_t, LyaRT_Grid, Geometry )
```

(continues on next page)

(continued from previous page)

```
>>> # Get cooler profiles
>>> w_pix_50th_Arr , f_pix_50th_Arr = Lya.plot_a_rebinned_line( w_50th_Arr , f_50th_Arr ,
↪ PIX_t )

>>> # Plot
>>> plt.plot( w_pix_Arr      , f_pix_Arr      , label='Target'   )
>>> plt.plot( w_pix_One_Arr  , f_pix_One_Arr  , label='1 iter'   )
>>> plt.plot( w_pix_50th_Arr , f_pix_50th_Arr , label='1000 iter')

>>> plt.legend(loc=0)
>>> plt.xlabel('wavelength[A]' , size=15 )
>>> plt.ylabel('Flux density [a.u.]' , size=15 )
>>> plt.xlim(1815,1835)
>>> plt.show()
```



finally, let's compare the parameters that we got with the input:

```
>>> print( 'The true redshift                is' , z_t      , 'and the predicted is' ,␣
↪z_50     , '(-' , z_50-z_16         , ', +' , z_84-z_50         , ')' )
>>> print( 'The true expansion velocity      is' , V_t      , 'and the predicted is' ,␣
↪V_50     , '(-' , V_50-V_16         , ', +' , V_84-V_50         , ')' )
>>> print( 'The true dust optical depth      is' , t_t      , 'and the predicted is' ,␣
↪t_50     , '(-' , t_50-t_16         , ', +' , t_84-t_50         , ')' )
```

(continues on next page)

(continued from previous page)

```
>>> print( 'The true intrinsic width          is' , W_t      , 'and the predicted is' ,␣
↪W_50     , '(-' , W_50-W_16         , ', +' , W_84-W_50         , ')' )
>>> print( 'The true log of HI column density is' , log_N_t  , 'and the predicted is' ,␣
↪log_N_50 , '(-' , log_N_50-log_N_16 , ', +' , log_N_84-log_N_50 , ')' )
>>> print( 'The true log of equivalent width  is' , log_EW_t , 'and the predicted is' ,␣
↪log_E_50 , '(-' , log_E_50-log_E_16 , ', +' , log_E_84-log_E_50 , ')' )
```

This should give something like:

```
The true redshift                 is 0.5 and the predicted is 0.49999833428137275 (- 0.
↪0001732166523583100 7 , + 0.0003615214512187048 )
The true expansion velocity       is 50.0 and the predicted is 47.070589157142614 (- 16.
↪100374040796254 , + 48.27234502291723 )
The true dust optical depth       is 0.01 and the predicted is 0.00379679848371737 (- 0.
↪003483235501588427 , + 0.049396128990436335 )
The true intrinsic width          is 0.5 and the predicted is 0.280484205908298 (- 0.
↪12228181625600373 , + 0.2150273326940031 )
The true log of HI column density is 20.0 and the predicted is 20.019139948537997 (- 0.
↪5728866241916535 , + 0.207985045834004 )
The true log of equivalent width  is 1.5 and the predicted is 1.5595962407058306 (- 0.
↪09992888862396399 , + 0.16009784914990055 )
```

The particular values that you print will be slightly different when you run it, but more or less it should go in the same direction.

That was fun, hah? Now you know how to use the deep neural network scheme in *zELDA*.

CHAPTER

SEVEN

TUTORIAL : TRAIN YOUR OWN NEURAL NETWORK

In this tutorial you will, hopefully, learn how to train your own deep neural network to predict the properties of outflos/inflows. For this we are going to use the python package *scikitlearn* (https://scikit-learn.org/stable/).

7.1 Generating data sets for the training

Let's start by loading *zELDA* grid of lines:

```
>>> import numpy as np
>>> import Lya_zelda as Lya
>>> import pickle
>>> from sklearn.neural_network import MLPRegressor

>>> your_grids_location = '/This/Folder/Contains/The/Grids/'
>>> Lya.funcs.Data_location = your_grids_location

>>> Geometry = 'Thin_Shell_Cont'

>>> DATA_LyaRT = Lya.load_Grid_Line( Geometry )
```

And let's do it for outflows,

```
>>> MODE = 'Outflow' # 'Inflow' for inflows
```

Let's define the region where we want to generate mock line profiles. You can adjust this to whatever your want. The values presented here are the standard in *zELDA*, but you can change them.

```
>>> # Logarithm of the expansion velocity in [km/s]
>>> log_V_in = [  1.0   ,  3.0   ]

>>> # Logarithm of the HI column density [cm**-2]
>>> log_N_in = [ 17.0   , 21.5   ]

>>> # Logarithm of the dust optical depth
>>> log_t_in = [ -4.0   , 0.0  ]

>>> # Logarithm of the intrinsic equivalent width [A]
>>> log_E_in = [ 0.0    , 2.3  ]

>>> # Logarithm of the intrinsic line width [A]
```

(continues on next page)

(continued from previous page)

```
>>> log_W_in = [ -2.    , 0.7  ]

>>> #Redshift interval
>>> z_in = [ 0.0001 , 4.00 ]

>>> # Logarithm of the full width half maximum convolving the spectrum (resolution) [A]
>>> log_FWHM_in = [ -1.  ,   0.3  ]

>>> # Logarithm of the pixel size [A]
>>> log_PIX_in  = [ -1.3 ,   0.3  ]

>>> # Logarithm of the signal to noise of the peak of the line
>>> log_PNR_in = [ 0.7 , 1.6 ]
```

Each of these lists have 2 elements. For example, *log_V_in[0]* indicates the lower border of the interval and *log_V_in[1]* the upper limit.

Let's set the number of sources that we want in our sample, for example 1000,

```
>>> N_train = 1000
```

Let's generate the properties of each of the training examples:

```
>>> V_Arr , log_N_Arr , log_t_Arr , log_E_Arr , log_W_Arr = Lya.NN_generate_random_
↪outflow_props_5D( N_train , log_V_in , log_N_in , log_t_in , log_E_in , log_W_in ,␣
↪MODE=MODE )

>>> z_Arr = np.random.rand( N_train ) * ( z_in[1] - z_in[0] ) + z_in[0]

>>> log_FWHM_Arr = np.random.rand( N_train ) * ( log_FWHM_in[1] - log_FWHM_in[0] ) + log_
↪FWHM_in[0]
>>> log_PIX_Arr  = np.random.rand( N_train ) * (  log_PIX_in[1] -  log_PIX_in[0] ) + ␣
↪log_PIX_in[0]
>>> log_PNR_Arr  = np.random.rand( N_train ) * (  log_PNR_in[1] -  log_PNR_in[0] ) + ␣
↪log_PNR_in[0]
```

each of these arrays contains random values that will be used in the training, for example, *V_Arr* contains the expansion velocity, etc.

Let's initialize the arrays where we want to store the data that we will need for the training

```
>>> F_t = 1.0

>>> Delta_True_Lya_Arr = np.zeros( N_train )

>>> N_bins = 1000

>>> z_PEAK_Arr = np.zeros( N_train )

>>> LINES_train = np.zeros( N_train * N_bins ).reshape( N_train , N_bins )

>>> N_bins_input = N_bins + 3

>>> INPUT_train = np.zeros( N_train * N_bins_input ).reshape( N_train , N_bins_input )
```

Let's generate the lines using the function *Lya.Generate_a_line_for_training*,

```
>>> print( 'Generating training set' )

>>> cc = 0.0
>>> for i in range( 0, N_train ):

>>>     per = 100. * i / N_train
>>>     if per >= cc :
>>>         print( cc , '%' )
>>>         cc += 1.0

>>>     V_t = V_Arr[i]
>>>     t_t = 10**log_t_Arr[i]
>>>     log_N_t = log_N_Arr[i]
>>>     log_E_t = log_E_Arr[i]
>>>     W_t = 10**log_W_Arr[i]

>>>     z_t = z_Arr[i]

>>>     FWHM_t = 10**log_FWHM_Arr[ i ]
>>>     PIX_t  = 10**log_PIX_Arr[  i ]
>>>     PNR_t = 10**log_PNR_Arr[i]

>>>     rest_w_Arr , train_line , z_max_i , input_i = Lya.Generate_a_line_for_training(␣
↪z_t , V_t, log_N_t, t_t, F_t, log_E_t, W_t , PNR_t, FWHM_t, PIX_t, DATA_LyaRT,␣
↪Geometry)

>>>     z_PEAK_Arr[i] = z_max_i

>>>     Delta_True_Lya_Arr[ i ] = 1215.67 * ( (1+z_t)/(1+z_max_i) - 1. )

>>>     LINES_train[i] = train_line
>>>     INPUT_train[i] = input_i
```

rest_w_Arr is the wavelength array where the profiles are evaluated in the rest frame of the peak of the line. *train_line* is the line profile evaluated in *rest_w_Arr*, *z_max_i* is the redshift of the source if the maximum of the line matches the Lyman-alpha line and *input_i* is the actual input that we will use for the DNN.

Now let's save all the data

```
>>> dic = {}
>>> dic[ 'lines' ] = LINES_train

>>> dic[ 'NN_input' ] = INPUT_train

>>> dic['z_PEAK'         ] = z_PEAK_Arr
>>> dic['z'              ] = z_Arr
>>> dic['Delta_True_Lya'] = Delta_True_Lya_Arr
>>> dic['V'              ] = V_Arr
>>> dic['log_N'          ] = log_N_Arr
>>> dic['log_t'          ] = log_t_Arr
>>> dic['log_PNR'        ] = log_PNR_Arr
>>> dic['log_W'          ] = log_W_Arr
```

(continues on next page)

(continued from previous page)

```
>>> dic['log_E'         ] = log_E_Arr
>>> dic['log_PIX'       ] = log_PIX_Arr
>>> dic['log_FWHM'      ] = log_FWHM_Arr

>>> dic['rest_w'] = rest_w_Arr

>>> np.save( 'data_for_training.npy' , dic )
```

Done, now you have a set of data that can be used as training set. Of course we have done it with only 1000 galaxies. In general you want to use about 100 000 or more. You can divide the data in small data sets for parallelization and then combine them, for example.

7.2 Get your DNN ready!

Let's load the data that we have just saved,

```
>>> Train_data = np.load( 'data_for_training.npy' , allow_pickle=True ).item()
```

Let's get the input that we will use in the training

```
>>> Input_train = Train_data['NN_input']
```

Now let's load the properties that we want to predict,

```
>>> Train_Delta_True_Lya_Arr = Train_data['Delta_True_Lya']

>>> Train_log_V_Arr = np.log10( Train_data[    'V'] )
>>> Train_log_N_Arr =           Train_data['log_N']
>>> Train_log_t_Arr =           Train_data['log_t']
>>> Train_log_E_Arr =           Train_data['log_E']
>>> Train_log_W_Arr =           Train_data['log_W']
```

and let's prepare it for skitlearn,

```
>>> TRAINS_OBSERVED = np.zeros( N_train * 6 ).reshape( N_train , 6 )

>>> TRAINS_OBSERVED[ : , 0 ] = Train_Delta_True_Lya_Arr
>>> TRAINS_OBSERVED[ : , 1 ] = Train_log_V_Arr
>>> TRAINS_OBSERVED[ : , 2 ] = Train_log_N_Arr
>>> TRAINS_OBSERVED[ : , 3 ] = Train_log_t_Arr
>>> TRAINS_OBSERVED[ : , 4 ] = Train_log_E_Arr
>>> TRAINS_OBSERVED[ : , 5 ] = Train_log_W_Arr
```

Now let's actually do the training. For this we have to decide what kind of deep learning configuration we want. For this tutorial let's use 2 hidden layers, each of 100 nodes,

```
>>> hidden_shape = ( 100 , 100 )
```

And train,

```
>>> from sklearn.neural_network import MLPRegressor

>>> est = MLPRegressor( hidden_layer_sizes=hidden_shape , max_iter=1000 )

>>> est.fit( Input_train , TRAINS_OBSERVED )
```

Done! You have now your custom DNN. Let's save it now so that you can use it later

```
>>> dic = {}

>>> dic['Machine'] = est
>>> dic['w_rest' ] = rest_w_Arr

>>> pickle.dump( dic , open( 'my_custom_DNN.sav' , 'wb'))
```

Done! Perfect. In this example we have just saved the skitlearn object and the wavelength array where the input for the DNN is computed. In principle you can put more things inside the dictionary. You can record the dynamical range of the parameters used (e.g. *log_V_in*), etc, etc and you can label them in the dictionary as you wish. However, the fundamental variables that must be saved are *'Machine'* and *'w_rest'*.

7.3 Using your custom DNN

If you want to use you custom DNN you can follow all the steps in *Fitting a line profile using deep learning*. The only difference is that, instead of loading the default DNN with *Lya.Load_NN_model()*, you have to load your DNN, which will also have the *dic['Machine']* and *dic['w_rest']* entries, as well the default one. Let's see an example of how you can load the custom DNN that you have just used:

```
>>> machine_data = pickle.load(open( 'my_custom_DNN.sav' , 'rb'))
```

machine_data is a python dictionary, with two entries: *'Machine'* and *'w_rest'*. These are the ones that you need in *Fitting a line profile using deep learning*.

CHAPTER

EIGHT

TUTORIAL : FITTING A LINE PROFILE USING MONTE CARLO MARKOV CHAINS

In this tutorial you will, hopefully, learn how fit Lyman-alpha line profiles using a Monte Carlo Markov Chain with *zELDA*. The MCMC engine is *emcee* (https://emcee.readthedocs.io/en/stable/).

8.1 Getting started

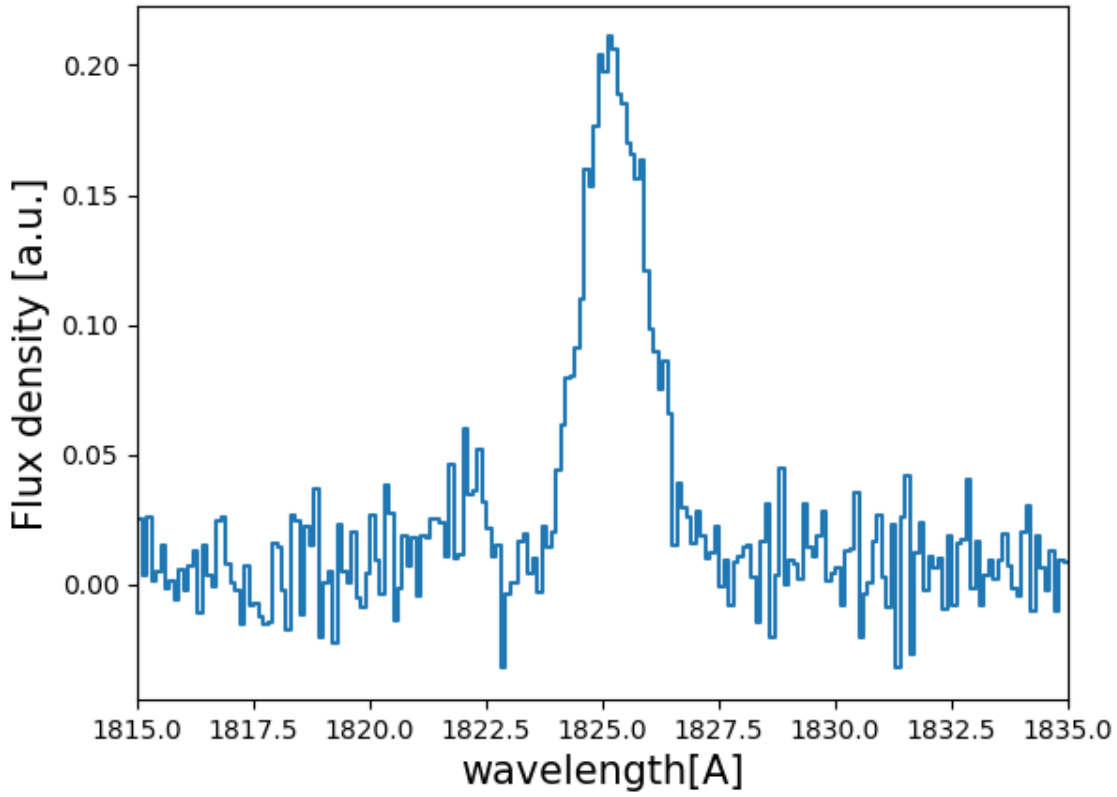
Let's start by loading *zELDA* creating a mock line profile that we will fit later. For more details on how to create a mock line profile go to *Mock line profiles*

```
>>> import Lya_zelda as Lya
>>> your_grids_location = '/This/Folder/Contains/The/Grids/'
>>> Lya.funcs.Data_location = your_grids_location

>>> Geometry = 'Thin_Shell_Cont'
>>> LyaRT_Grid = Lya.load_Grid_Line( Geometry )

>>> # Defining the model parameters:
>>> z_t      = 0.5   # redshift of the source
>>> V_t      = 40.0  # Outflow expansion velocity [km/s]
>>> log_N_t  = 20.   # Logarithmic of the neutral hydrogen column density [cm**-2]
>>> t_t      = 0.01  # Dust optical depth
>>> log_EW_t = 1.5   # Logarithmic the intrinsic equivalent width [A]
>>> W_t      = 0.5   # Intrinsic width of the line [A]
>>> F_t      = 1.    # Total flux of the line

>>> # Defining the quality of the line profile:
>>> PNR_t  = 15.0 # Signal to noise ratio of the maximum of the line.
>>> FWHM_t = 0.2  # Full width half maximum diluting the line. Mimics finite resolution.␣
↪[A]
>>> PIX_t  = 0.1  # Wavelength binning of the line. [A]

>>> w_Arr , f_Arr , s_Arr = Lya.Generate_a_real_line( z_t , V_t, log_N_t, t_t, F_t, log_
↪EW_t, W_t , PNR_t, FWHM_t, PIX_t, LyaRT_Grid, Geometry )
```

where */This/Folder/Contains/The/Grids/* is the place where you store the LyaRT data grids, as shown in the installation section. And... It's done! *w_Arr* is a numpy array that contains the wavelength where the line profile is evaluated. Meanwhile, *f_Arr* is the actual line profile. *s_Arr* is the uncertainty of the flux density. Remeber that if you want to use the line profile grid with lower RAM memory occupation you must pass *MODE='LIGHT'* to *Lya.load_Grid_Line*.

Let's have a look to how the line looks:

```
>>> w_Arr , f_Arr , s_Arr  = Lya.Generate_a_real_line( z_t , V_t, log_N_t, t_t, F_t, log_
↪EW_t, W_t , PNR_t, FWHM_t, PIX_t, LyaRT_Grid, Geometry )

>>> w_pix_Arr , f_pix_Arr = Lya.plot_a_rebinned_line( w_Arr , f_Arr , PIX_t )

>>> import pylab as plt
>>> plt.plot( w_pix_Arr , f_pix_Arr )
>>> plt.xlabel('wavelength[A]' , size=15 )
>>> plt.ylabel('Flux density [a.u.]' , size=15 )
>>> plt.xlim(1815,1835)
>>> plt.show()
```



8.2 The MCMC anlysis

Let's now set the configuration for the MCMC analysis.

```
>>> N_walkers = 200 # Number of walkers
>>> N_burn    = 200 # Number of steps to burn-in
>>> N_steps   = 300 # Number of steps to run after burning-in
```

Now let's choose the method to initialize the walkers. There are basically two methods: using the deep neural network or doing a fast particle swarm optimization (PSO). For this tutorial we will use the deep neural network.

```
>>> MODE = 'DNN'
```

If you want to use instead the PSO you can set *MODE = 'PSO'*.

Now let's get the regions where we want to originally spawn our lovely walkers:

```
>>> log_V_in , log_N_in , log_t_in , log_E_in , W_in , z_in , Best = Lya.MCMC_get_region_
↪6D( MODE , w_Arr , f_Arr , s_Arr , FWHM_t , PIX_t , LyaRT_Grid , Geometry )
```

The variables *log_V_in*, *log_N_in*, *log_t_in*, *log_E_in*, *W_in* and *z_in* are python lists of two elements containing the range where to spawn the walkers for the logarithmic of the bulk velocity, the logarithmic of the HI column density, the logarithmic of the dust optical, the logarithmic of the intrinsic equivalent width, the intrinsic width of the line and the redshift. For example, *z_in[0]* contains the minimum redshift and *z_in[0]* the maximum. Actually this step is not necessary and if you want you can continue without defining these variables or setting them as you please. Also, remember that these list only maker where the walkers are spawned. They might actually get outside this volume if the best fitting region is outside.

Let's now run the MCMC:

```
>>> sampler = Lya.MCMC_Analysis_sampler_5( w_Arr , f_Arr , s_Arr , FWHM_t , N_walkers ,␣
↪N_burn , N_steps , Geometry , LyaRT_Grid , z_in=z_in , log_V_in=log_V_in , log_N_
↪in=log_N_in , log_t_in=log_t_in , log_E_in=log_E_in , W_in=W_in )
```

sampler is an object of the python package *emcee*. Note that there is a way of forcing the redshift to be inside *z_in*. We decided to this with only this property in case you know the redshift of the source before hand. you can do this by passing *FORCE_z=True* to *Lya.MCMC_Analysis_sampler_5*.

Now let's get the actual value of the predicted properties and their 1-sigma uncertainty. For this, in this tutorial we chose as our prediction the percentile 50th o the probability distribution function of the variables. For the +-1-sigma uncertainty we choose the percentiles 16th and 84th.

```
>>> Q_Arr = [ 16 , 50 , 84 ] # You can add more percentiles here, like 95

>>> perc_matrix_sol , flat_samples = Lya.get_solutions_from_sampler( sampler , N_walkers␣
↪, N_burn , N_steps , Q_Arr )
```

flat_samples contains the MCMC chains flatten. *perc_matrix_sol* is a 2-D array with dimensions *6xlen(Q_Arr)* containing the percentiles of the variables. You can extract the values doing something like:

```
>>> # redshift.
>>> z_16     =     perc_matrix_sol[ 3 , 0 ] # corresponds to Q_Arr[0]
>>> z_50     =     perc_matrix_sol[ 3 , 1 ] # corresponds to Q_Arr[1]
>>> z_84     =     perc_matrix_sol[ 3 , 2 ] # corresponds to Q_Arr[2]

>>> # Expansion velocity.
>>> V_16     = 10**perc_matrix_sol[ 0 , 0 ]
>>> V_50     = 10**perc_matrix_sol[ 0 , 1 ]
>>> V_84     = 10**perc_matrix_sol[ 0 , 2 ]

>>> # dust optical depth.
>>> t_16     = 10**perc_matrix_sol[ 2 , 0 ]
>>> t_50     = 10**perc_matrix_sol[ 2 , 1 ]
>>> t_84     = 10**perc_matrix_sol[ 2 , 2 ]

>>> # Intrinsic width.
```

(continues on next page)

(continued from previous page)

```
>>> W_16      =     perc_matrix_sol[ 5 , 0 ]
>>> W_50      =     perc_matrix_sol[ 5 , 1 ]
>>> W_84      =     perc_matrix_sol[ 5 , 2 ]

>>> # Logarithmic of the intrinsic equivalent width.
>>> log_E_16 =     perc_matrix_sol[ 4 , 0 ]
>>> log_E_50 =     perc_matrix_sol[ 4 , 1 ]
>>> log_E_84 =     perc_matrix_sol[ 4 , 2 ]

>>> # Logarithmic of the HI column density.
>>> log_N_16 =     perc_matrix_sol[ 1 , 0 ]
>>> log_N_50 =     perc_matrix_sol[ 1 , 1 ]
>>> log_N_84 =     perc_matrix_sol[ 1 , 2 ]
```

Let's compare the MCMC prediction with the actual input:

```
>>> print( 'The true redshift                 is' , z_t      , 'and the predicted is' ,␣
↪z_50     , '(-' , z_50-z_16         , ', +' , z_84-z_50         , ')' )
>>> print( 'The true expansion velocity       is' , V_t      , 'and the predicted is' ,␣
↪V_50     , '(-' , V_50-V_16         , ', +' , V_84-V_50         , ')' )
>>> print( 'The true dust optical depth       is' , t_t      , 'and the predicted is' ,␣
↪t_50     , '(-' , t_50-t_16         , ', +' , t_84-t_50         , ')' )
>>> print( 'The true intrinsic width          is' , W_t      , 'and the predicted is' ,␣
↪W_50     , '(-' , W_50-W_16         , ', +' , W_84-W_50         , ')' )
>>> print( 'The true log of HI column density is' , log_N_t  , 'and the predicted is' ,␣
↪log_N_50 , '(-' , log_N_50-log_N_16 , ', +' , log_N_84-log_N_50 , ')' )
>>> print( 'The true log of equivalent width  is' , log_EW_t , 'and the predicted is' ,␣
↪log_E_50 , '(-' , log_E_50-log_E_16 , ', +' , log_E_84-log_E_50 , ')' )
```

which should look something like:

```
The true redshift                 is 0.5 and the predicted is 0.49991074547548753 (- 1.
↪9665578543492934e-05 , + 0.0014991528312225944 )
The true expansion velocity       is 40.0 and the predicted is 30.741297629627855 (- 1.
↪097915986182759 , + 244.88872432354253 )
The true dust optical depth       is 0.01 and the predicted is 0.04392859929402969 (- 0.
↪035550939281926146 , + 0.0103076912398413 )
The true intrinsic width          is 0.5 and the predicted is 0.2859470609607235 (- 0.
↪09765211992507192 , + 0.06363668998672473 )
The true log of HI column density is 20.0 and the predicted is 20.215438954615962 (- 2.
↪4584647794744434 , + 0.027551697514507367 )
The true log of equivalent width  is 1.5 and the predicted is 1.7365288817793056 (- 0.
↪29375812799042955 , + 0.033311663274792735 )
```

Now let's plot the lines and see how they compare:

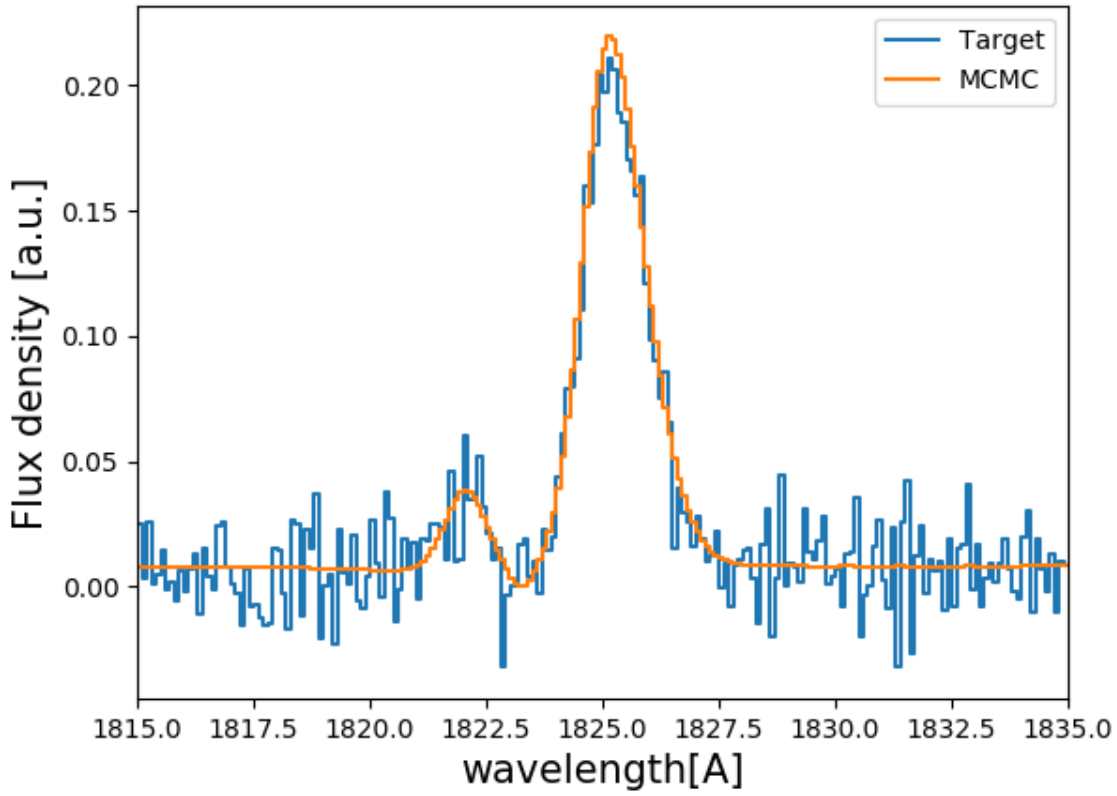
```
>>> # Infinite signal to noise in the model
>>> PNR = 100000.

>>> # Compute line
>>> w_One_Arr , f_One_Arr , _  = Lya.Generate_a_real_line( z_50, V_50, log_N_50, t_50, F_
↪t, log_E_50, W_50, PNR, FWHM_t, PIX_t, LyaRT_Grid, Geometry )
```

(continues on next page)

(continued from previous page)

```
>>> # Make cooler
>>> w_pix_One_Arr , f_pix_One_Arr = Lya.plot_a_rebinned_line( w_One_Arr , f_One_Arr ,␣
↪PIX_t )

>>> # Plot
>>> plt.plot( w_pix_Arr     , f_pix_Arr     , label='Target' )
>>> plt.plot( w_pix_One_Arr , f_pix_One_Arr , label='MCMC'   )
>>>
>>> plt.legend(loc=0)
>>> plt.xlabel('wavelength[A]' , size=15 )
>>> plt.ylabel('Flux density [a.u.]' , size=15 )
>>> plt.xlim(1815,1835)
>>> plt.show()
```

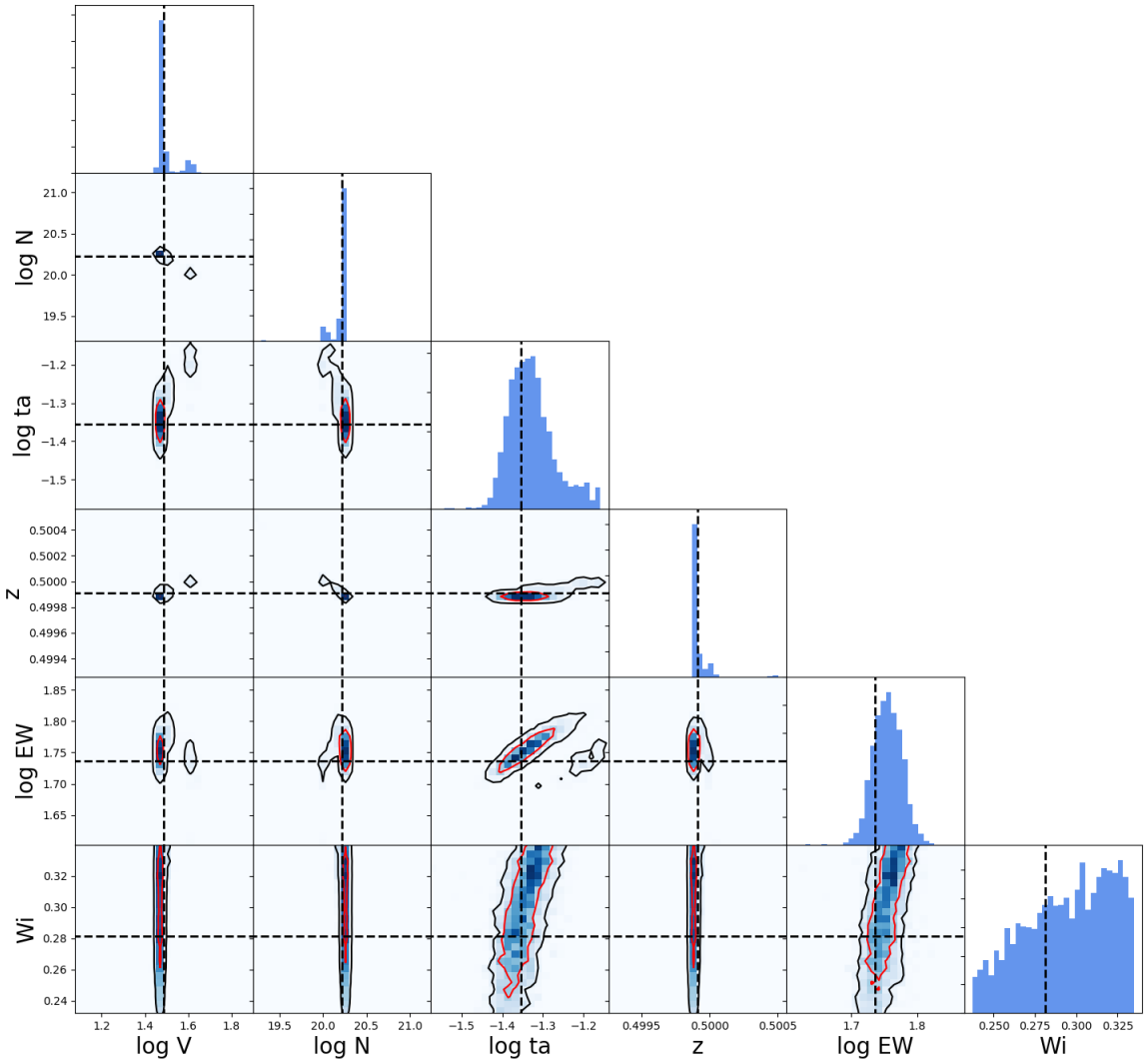
This should give you something like this:



Now let's do a correlation plot to see where the walkers are. For this we will use the function *make_corner_plots* which is define just below in this same page, in *Tool to make corraltion plots* .

```
>>> make_corner_plots( flat_samples )
>>> plt.show()
```

And it should give you something like:



And.. with that it's done. Now you know how to use the MCMC implementation in *zELDA*.

8.3 Tool to make corraltion plots

This is just a code to plot the walkers and the probability distribution funtions of the posteriors of the MCMC analysis.

```
def make_corner_plots( my_chains_matrix ):

    import numpy as np
    import pylab as plt

    N_dim = 6
```

(continues on next page)

(continued from previous page)

```
    ax_list = []

    label_list = [ 'log V' , 'log N' , 'log ta' , 'z' , 'log EW', 'Wi'  ]

    MAIN_VALUE_mean   = np.zeros(N_dim)
    MAIN_VALUE_median = np.zeros(N_dim)
    MAIN_VALUE_MAX    = np.zeros(N_dim)

    for i in range( 0 , N_dim ):

        x_prop = my_chains_matrix[ : , i ]

        x_prop_min = np.percentile( x_prop , 10 )
        x_prop_50  = np.percentile( x_prop , 50 )
        x_prop_max = np.percentile( x_prop , 90 )

        x_min = x_prop_50 - ( x_prop_max - x_prop_min ) * 1.00
        x_max = x_prop_50 + ( x_prop_max - x_prop_min ) * 1.00

        mamamask = ( x_prop > x_min ) * ( x_prop < x_max )

        MAIN_VALUE_mean[  i] = np.mean(       x_prop[ mamamask ] )
        MAIN_VALUE_median[i] = np.percentile( x_prop[ mamamask ] , 50 )

        HH , edges_HH = np.histogram( x_prop[ mamamask ] , 30 , range=[ x_prop_min , x_
↪prop_max ] )

    plt.figure( figsize=(15,15) )

    Q_top = 80
    Q_low = 20

    for i in range( 0 , N_dim ):

        y_prop = my_chains_matrix[ : , i ]

        y_prop_min = np.percentile( y_prop , Q_low )
        y_prop_50  = np.percentile( y_prop , 50 )
        y_prop_max = np.percentile( y_prop , Q_top  )

        mask_y = ( y_prop > y_prop_min ) * ( y_prop < y_prop_max )

        y_min = y_prop_50 - np.std( y_prop[ mask_y ] )
        y_max = y_prop_50 + np.std( y_prop[ mask_y ] )

        for j in range( 0 , N_dim ):

            if i < j : continue

            x_prop = my_chains_matrix[ : , j ]

            x_prop_min = np.percentile( x_prop , Q_low )
```

(continues on next page)

(continued from previous page)

```
            x_prop_50  = np.percentile( x_prop , 50 )
            x_prop_max = np.percentile( x_prop , Q_top )

            mask_x = ( x_prop > x_prop_min ) * ( x_prop < x_prop_max )

            x_min = x_prop_50 - np.std( x_prop[ mask_x ] )
            x_max = x_prop_50 + np.std( x_prop[ mask_x ] )

            ax = plt.subplot2grid( ( N_dim , N_dim ) , (i, j)  )

            ax_list += [ ax ]

            DDX = x_max - x_min
            DDY = y_max - y_min

            if i==j :

                H , edges = np.histogram( x_prop , 30 , range=[x_min,x_max] )

                ax.hist( x_prop , 30 , range=[x_min,x_max] , color='cornflowerblue' )

                ax.plot( [ MAIN_VALUE_median[i] , MAIN_VALUE_median[i] ] , [ 0.0 , 1e10␣
↪] , 'k--' , lw=2 )

                ax.set_ylim( 0 , 1.1 * np.amax(H) )

            else :

                XX_min = x_min - DDX * 0.2
                XX_max = x_max + DDX * 0.2

                YY_min = y_min - DDY * 0.2
                YY_max = y_max + DDY * 0.2

                H , edges_y , edges_x = np.histogram2d( x_prop , y_prop , 30 ,␣
↪range=[[XX_min , XX_max],[YY_min , YY_max]] )

                y_centers = 0.5 * ( edges_y[1:] + edges_y[:-1] )
                x_centers = 0.5 * ( edges_x[1:] + edges_x[:-1] )

                H_min = np.amin( H )
                H_max = np.amax( H )

                N_bins = 10000

                H_Arr = np.linspace( H_min , H_max , N_bins )[::-1]

                fact_up_Arr = np.zeros( N_bins )

                TOTAL_H = np.sum( H )

                for iii in range( 0 , N_bins ):
```

(continues on next page)

(continued from previous page)

```
                    mask = H > H_Arr[iii]

                    fact_up_Arr[iii] = np.sum( H[ mask ] ) / TOTAL_H

                H_value_68 = np.interp( 0.680 , fact_up_Arr , H_Arr )
                H_value_95 = np.interp( 0.950 , fact_up_Arr , H_Arr )

                ax.pcolormesh( edges_y , edges_x , H.T , cmap='Blues' )

                ax.contour( y_centers, x_centers , H.T , colors='k' , levels=[ H_value_
↪95 ] )
                ax.contour( y_centers, x_centers , H.T , colors='r' , levels=[ H_value_
↪68 ] )

                X_VALUE =  MAIN_VALUE_median[j]
                Y_VALUE =  MAIN_VALUE_median[i]

                ax.plot( [ X_VALUE , X_VALUE ] , [    -100 ,     100 ] , 'k--' , lw=2 )
                ax.plot( [    -100 ,     100 ] , [ Y_VALUE , Y_VALUE ] , 'k--' , lw=2 )

                ax.set_ylim( y_min-0.05*DDY , y_max+0.05*DDY )

            ax.set_xlim( x_min-0.05*DDX , x_max+0.05*DDX )

            if i==N_dim-1:
                ax.set_xlabel( label_list[j] , size=20 )

            if j==0 and i!=0 :
                ax.set_ylabel( label_list[i] , size=20 )

            if j!=0:
                plt.setp( ax.get_yticklabels(), visible=False)

            if j==0 and i==0:
                plt.setp( ax.get_yticklabels(), visible=False)

            if i!=len( label_list)-1 :
                plt.setp( ax.get_xticklabels(), visible=False)

    plt.subplots_adjust( left = 0.09 , bottom = 0.15 , right = 0.98 , top = 0.99 ,␣
↪wspace=0., hspace=0.)

    return None
```

CHAPTER

NINE

TUTORIAL : COMPUTING LYMAN-ALPHA ESCAPE FRACTIONS

In this tutorial you will, hopefully, learn how to compute Lyman-alpha escape fractions with *zELDA*. Note that this part of the code compres directly from *FLaREON* (https://github.com/sidgurun/FLaREON , Gurung-lopez et al. 2019b).

9.1 Default computation of escape fractions

Let's move to one of the most powerful products of *FLaREON*: predicting huge amounts of Lyman alpha escape fractions.

However, *zELDA* implements several gas geometries and is optimized to obtain large amount of escape fractions with only one line of code, so let us expand this a little bit more. If we want to compute the escape fraction in a thin shell outflow with the configurations { V , logNH , ta } , { 200 , 19.5 , 0.1 }, { 300 , 20.0 , 0.01 } and { 400 , 20.5 , 0.001 } we could do

```
>>> import Lya_zelda as Lya
>>> your_grids_location = '/This/Folder/Contains/The/Grids/'
>>> Lya.funcs.Data_location = your_grids_location

>>> Geometry = 'Thin_Shell'
>>> # Other options: 'Galactic Wind' or 'Bicone_X_Slab_In' or 'Bicone_X_Slab_Out'

>>> # Expansion velocity array in km/s
>>> V_Arr     = [  200 ,  300 , 400   ]

>>> # Logarithmic of column densities array in cm**-2
>>> logNH_Arr = [ 19.5 , 20.0 , 20.5  ]

>>> # Dust optical depth Array
>>> ta_Arr    = [  0.1 , 0.01 , 0.001 ]
```

Where *Geometry* indicates the gas distribution that is being used. 'Bicone_X_Slab_In' indicates the bicone geometry look through the outflow, while 'Bicone_X_Slab_In' is looking through the optically thick gas. The 'Thin_Shell_Cont' model does not support escape fractions yet.

Now let's compute the escape fraction for this configurations:

```
>>> f_esc_Arr = Lya.RT_f_esc( Geometry , V_Arr , logNH_Arr , ta_Arr )
```

The variable *f_esc_Arr* is an Array of 1 dimension and length 3 that encloses the escape fractions for the configurations. In particular *f_esc_Arr[i]* is computed using *V_Arr[i]* , *logNH_Arr[i]* and *ta_Arr[i]*.

9.2 Deeper options on predicting the escape fraction

There are many algorithms implemented to compute *f_esc_Arr*. By default *FLaREON* uses a machine learning decision tree regressor and a parametric equation for the escape fraction as function of the dust optical depth (Go to the *FLaREON* presentation paper Gurung-Lopez et al. in prep for more information). These settings were chosen as default since they give the best performance. However the user might want to change the computing algorithm so here leave a guide with all the available options.

- *MODE* variable refers to mode in which the escape fraction is computed. There are 3 ways in which *FLaREON* can compute this. i) *'Raw'* Using the raw data from the RTMC (Orsi et al. 2012). ii) *'Parametrization'* Assume a parametric equation between the escape fraction and the dust optical depth that allows to extend calculations outside the grid with the highest accuracy (in *FLaREON*). iii) *'Analytic'* Use of the recalibrated analytic equations presented by Gurung-Lopez et al. 2018. Note that the analytic mode is not enabled in the bicone geometry although it is in the *'Thin_Shel'* and *'Galactic_Wind'*
- *Algorithm* variable determines the technique used. This can be i) *'Intrepolation'*: lineal interpolation is used. ii) *'Machine_Learning'* machine learning is usd. To determine which machine learning algorithm you would like to use please, provide the variable *Machine_Learning_Algorithm*. The machine learning algorithms implemented are Decision tree regressor (*'Tree'*), Random forest regressor (*'Forest'*) and KN regressor (*'KN'*). The machine learning is implemented by *Sci-kit-learn*, please, visit their webside for more information (http://scikit-learn.org/stable/).

Finally, any combination of *MODE* , *Algorithm* and *Machine_Learning_Algorithm* is allowed. However, note that the variable *Machine_Learning_Algorithm* is useles s if *Algorithm='Intrepolation'*.

CHAPTER

TEN

FUNCS MODULE

zELDA es a phantastic code!!

funcs.**Analytic_f_esc_Thin_Shell**(*V_Arr*, *logNH_Arr*, *ta_Arr*)

Return the escape fraction computed analytically for the Thin Shell (Gurung-lopez et al. 2019a)

Input:

V_Arr [1-D sequence of floats] Outflow bulk velocity [km/s]

logNH_Arr [1-D sequence of floats] logarithm of the neutral hydrogen column density [cm**-2]

ta_Arr [1-D sequence of floats] Dust optical depth [no dimensions]

Output:

fesc [1-D sequence of floats] Escape fractions for the input configurations [no dimensions]

funcs.**Analytic_f_esc_Wind**(*V_Arr*, *logNH_Arr*, *ta_Arr*)

Return the escape fraction computed analytically for the Galactic Wind (Gurung-lopez et al. 2019a)

Input:

V_Arr [1-D sequence of floats] Outflow bulk velocity [km/s]

logNH_Ar [1-D sequence of floats] logarithm of the neutral hydrogen column density [cm**-2]

ta_Arr [1-D sequence of floats] Dust optical depth [no dimensions]

Output:

fesc [1-D sequence of floats] Escape fractions for the input configurations [no dimensions]

funcs.**Check_if_DATA_files_are_found**()

This function checks if all the data files are in the set directory.

Input:

None : None

Output:

Bool_1 [Bool] 1 if found. 0 if not found.

funcs.**Compute_Inflow_From_Outflow**(*w_Arr*, *f_out_Arr*)

Computes the line profile of an inflow from the line profiles of an outflow

Input:

w_Arr [1-D sequence of floats] wavelength where the line profile is evaluated.

f_out_Arr [float] Outflow flux density (line profile)

Output:

f_in_Arr [1-D sequence of bool] Inflow flux density (line profile)

funcs.**Define_wavelength_for_NN**(*Delta_min=- 18.5*, *Delta_max=18.5*, *Nbins_tot=1000*, *Denser_Center=True*)

This function defines the wavelength used in for the neural netwroks.

Input

Delta_min [optional float] Defines the minimum rest frame wavelegnth with respecto to Lyman-alpha.

Default = -18.5

Delta_min [optional float] Defines the maximum rest frame wavelegnth with respecto to Lyman-alpha.

Default = +18.5

Nbins_tot [optional int] Total number of wvelgnths bins.

Default = 1000

Denser_Center [optional bool] Populates denser the regions close to Lyman-alpha

Default = True

Output

rest_w_Arr [1-D sequence of float] Wavelgnth array where the line is evaluated in the rest frame.

funcs.**Generate_a_line_for_training**(*z_t*, *V_t*, *log_N_t*, *t_t*, *F_t*, *log_EW_t*, *W_t*, *PNR_t*, *FWHM_t*, *PIX_t*, *DATA_LyaRT*, *Geometry*, *normed=False*, *scaled=True*, *Delta_min=-18.5*, *Delta_max=18.5*, *Denser_Center=True*, *Nbins_tot=1000*, *T_IGM_Arr=None*, *w_IGM_Arr=None*, *RETURN_ALL=False*)

Creates a mock line profile at the desired redshift and returns all the NN products.

Input

z_t [float] Redshift

V_t [float] Outflow expansion velocity [km/s]

log_N_t [float] logarithmic of the neutral hydrogen column density in cm**-2

t_t [float] Dust optical depth

F_t [float] Total flux of the line. You can pass 1.

log_EW_t [optional, float] Logarithmic of the rest frame intrisic equivalent width of the line [A] Requiered if Geometry == 'Thin_Shell_Cont'

W_t [optional, float] Rest frame intrisic width of the Lyman-alpha line [A] Requiered if Geometry == 'Thin_Shell_Cont'

PNR_t [float] Signal to noise ratio of the global maximum of the line profile.

FWHM_t [float] Full width half maximum [A] of the experiment. This dilutes the line profile.

PIX_t [float] Pixel size in wavelgnth [A] of the experiment. This binnes the line profile.

DATA_LyaRT [python dictionary] Contains the grid information.

Geometry [string] Outflow geometry to use.

Delta_min [optional float] Defines the minimum rest frame wavelegnth with respecto to Lyman-alpha.

Default = -18.5

Delta_min [optional float] Defines the maximum rest frame wavelegnth with respecto to Lyman-alpha.

Default = +18.5

Nbins_tot [optional int] Total number of wvelgnths bins.

Default = 1000

Denser_Center [optional bool] Populates denser the regions close to Lyman-alpha

Default = True

normed [optional bool] If True, nomalizes the line profile.

scaled [optinal bool] If True, divides the line profile by its maximum.

Output

rest_w_Arr [1-D sequence of float] Wavelgnth array where the line is evaluated in the rest frame.

train_line [1-D sequence of float] Line profile.

z_max_i [float] Redshift of the source if the global maximum of the spectrum is the Lyman-alpha wavelegth.

INPUT: 1-D secuence of float The actuall input to use in the Neural networks.

funcs.**Generate_a_line_for_training_II**(*z_t*, *V_t*, *log_N_t*, *t_t*, *F_t*, *log_EW_t*, *W_t*, *PNR_t*, *FWHM_t*, *PIX_t*, *DATA_LyaRT*, *Geometry*, *normed=False*, *scaled=True*, *Delta_min=- 10.0*, *Delta_max=10.0*, *Denser_Center=True*, *Nbins_tot=500*, *T_IGM_Arr=None*, *w_IGM_Arr=None*)

Creates a mock line profile at the desired redshift and returns all the NN products.

Input

z_t [float] Redshift

V_t [float] Outflow expansion velocity [km/s]

log_N_t [float] logarithmic of the neutral hydrogen column density in cm**-2

t_t [float] Dust optical depth

F_t [float] Total flux of the line. You can pass 1.

log_EW_t [optional, float] Logarithmic of the rest frame intrisic equivalent width of the line [A] Requiered if Geometry == 'Thin_Shell_Cont'

W_t [optional, float] Rest frame intrisic width of the Lyman-alpha line [A] Requiered if Geometry == 'Thin_Shell_Cont'

PNR_t [float] Signal to noise ratio of the global maximum of the line profile.

FWHM_t [float] Full width half maximum [A] of the experiment. This dilutes the line profile.

PIX_t [float] Pixel size in wavelgnth [A] of the experiment. This binnes the line profile.

DATA_LyaRT [python dictionary] Contains the grid information.

Geometry [string] Outflow geometry to use.

Delta_min [optional float] Defines the minimum rest frame wavelegnth with respecto to Lyman-alpha.

Default = -12.5

Delta_min [optional float] Defines the maximum rest frame wavelegnth with respecto to Lyman-alpha.

Default = +12.5

Nbins_tot [optional int] Total number of wvelgnths bins.

Default = 800

Denser_Center [optional bool] Populates denser the regions close to Lyman-alpha

Default = True

normed [optional bool] If True, nomalizes the line profile.

scaled [optinal bool] If True, divides the line profile by its maximum.

Output

rest_w_Arr [1-D sequence of float] Wavelgnth array where the line is evaluated in the rest frame.

train_line [1-D sequence of float] Line profile.

z_max_i [float] Redshift of the source if the global maximum of the spectrum is the Lyman-alpha wavelegth.

INPUT: 1-D secuence of float The actuall input to use in the Neural networks.

funcs.**Generate_a_real_line**(*z_t*, *V_t*, *log_N_t*, *t_t*, *F_t*, *log_EW_t*, *W_t*, *PNR_t*, *FWHM_t*, *PIX_t*, *DATA_LyaRT*, *Geometry*, *T_IGM_Arr=None*, *w_IGM_Arr=None*, *RETURN_ALL=False*)

Makes a mock line profile for the Thin_Shell_Cont geometry.

Input

z_t [float] Redshift

V_t [float] Outflow expansion velocity [km/s]

log_N_t [float] logarithmic of the neutral hydrogen column density in cm**-2

t_t [float] Dust optical depth

F_t [float] Total flux of the line. You can pass 1.

log_EW_t [optional, float] Logarithmic of the rest frame intrisic equivalent width of the line [A] Requiered if Geometry == 'Thin_Shell_Cont'

W_t [optional, float] Rest frame intrisic width of the Lyman-alpha line [A] Requiered if Geometry == 'Thin_Shell_Cont'

PNR_t [float] Signal to noise ratio of the global maximum of the line profile.

FWHM_t [float] Full width half maximum [A] of the experiment. This dilutes the line profile.

PIX_t [float] Pixel size in wavelgnth [A] of the experiment. This binnes the line profile.

DATA_LyaRT [python dictionary] Contains the grid information.

Geometry [string] Outflow geometry to use.

Output

w_Arr [1-D sequence of float] Wavelgnth array where the line is evaluated in the observed frame.

f_Arr [1-D sequence of float] Line profile flux density in arbitrary units.

noise_Amplitude_Arr [1-D sequence of float] 1-sigma level of the applyed noise.

funcs.**Interpolate_Lines_Arrays_3D_grid**(*V_Arr*, *logNH_Arr*, *logta_Arr*, *x_Arr*, *Grid_Dictionary*)

Computes the escape fraction using the line profiles grids for many configurations

Input:

V_Arr [1-D sequence of floats] Outflow bulk velocity [km/s]

logNH_Ar [1-D sequence of floats] Logarithm of the neutral hydrogen column density [cm**-2]

ta_Arr [1-D sequence of floats] Dust optical depth [no dimensions]

x_Arr [1-D sequence of floats] Frequency in Doppler units where the line prfile will be evaluated.

Grid_Dictionary [python dictionary] All the necessary information for the interpoation. Loaded with load_Grid_Line().

Output:

line_Arr [2-D sequence of floats] Flux density in arbitrary units. The first dimension matches the dimension of the input configurations (e.g. V_Arr). The second dimension matches x_Arr.

funcs.**Interpolate_Lines_Arrays_3D_grid_MCMC**(*V_Value*, *logNH_Value*, *logta_Value*, *x_Arr*, *Grid_Dictionary*)

Computes the escape fraction using the line profiles grids for one configuration. This is usefull for the Thin_Shell, Galactic_Wind and Bicones configurations.

Input:

V_Value [float] Outflow bulk velocity [km/s]

logNH_Value [float] Logarithm of the neutral hydrogen column density [cm**-2]

ta_Value [float] Dust optical depth [no dimensions]

x_Arr [1-D sequence of floats] Frequency in Doppler units where the line prfile will be evaluated.

Grid_Dictionary [python dictionary] All the necessary information for the interpoation. Loaded with load_Grid_Line().

Output:

axu_line_1 [1-D sequence of floats] Flux density in arbitrary units.

funcs.**Interpolate_Lines_Arrays_5D_grid**(*V_Arr*, *logNH_Arr*, *logta_Arr*, *logEW_Arr*, *Wi_Arr*, *x_Arr*, *Grid_Dictionary*)

Computes the escape fraction using the line profiles grids for many configurations. This is usefull for the Thin_Shell_Cont

Input:

V_Arr [1-D sequence of floats] Outflow bulk velocity [km/s]

logNH_Ar [1-D sequence of floats] Logarithm of the neutral hydrogen column density [cm**-2]

ta_Arr [1-D sequence of floats] Dust optical depth [no dimensions]

logEW_Arr [1-D sequence of floats] Logarithm of the rest frame equivalent width [A]

Wi_Arr [1-D sequence of floats] Rest frame instrinc line width [A]

x_Arr [1-D sequence of floats] Frequency in Doppler units where the line prfile will be evaluated.

Grid_Dictionary [python dictionary] All the necessary information for the interpoation. Loaded with load_Grid_Line().

Output:

linew_Arr [2-D sequence of floats] Flux density in arbitrary units. The first dimension matches the dimension of the input configurations (e.g. V_Arr). The second dimension matches x_Arr. Flux density in arbitrary units.

funcs.**Interpolate_Lines_Arrays_5D_grid_MCMC**(*V_Value*, *logNH_Value*, *logta_Value*, *logEW_Value*, *Wi_Value*, *x_Arr*, *Grid_Dictionary*)

Computes the escape fraction using the line profiles grids for many configurations. This is usefull for the Thin_Shell_Cont

Input:

V_Value [float] Outflow bulk velocity [km/s]

logNH_Value [float] Logarithm of the neutral hydrogen column density [cm**-2]

ta_Value [float] Dust optical depth [no dimensions]

logEW_Value [float] Logarithm of the rest frame equivalent width [A]

Wi_Value [float] Rest frame instrinc line width [A]

x_Arr [1-D sequence of floats] Frequency in Doppler units where the line prfile will be evaluated.

Grid_Dictionary [python dictionary] All the necessary information for the interpoation. Loaded with load_Grid_Line().

Output:

axu_line_1 [1-D sequence of floats] Flux density in arbitrary units.

funcs.**Interpolate_f_esc_Arrays_2D_grid**(*V_Arr*, *logNH_Arr*, *ta_Arr*, *Grid_Dictionary*, *Geometry*)
Computes the escape fraction using the escape fraction grids of parameters

Input:

V_Arr [1-D sequence of floats] Outflow bulk velocity [km/s]

logNH_Ar [1-D sequence of floats] logarithm of the neutral hydrogen column density [cm**-2]

ta_Arr [1-D sequence of floats] Dust optical depth [no dimensions]

Grid_Dictionary [python dictionary] Constains the grid to compute the escape fraction. Loaded with load_Grid_fesc().

Geometry [String]

Outflow configuration to use: 'Thin_Shell' , 'Galactic_Wind' , 'Bicone_X_Slab_In' , 'Bicone_X_Slab_Out'

Output:

f_esc_Arr [1-D sequence of floats] Escape fractions for the input configurations [no dimensions]

funcs.**Interpolate_fesc_Arrays_3D_grid**(*V_Arr*, *logNH_Arr*, *ta_Arr*, *Grid_Dictionary*)
Computes the escape fraction using the escape fraction grids of parameters

Input:

V_Arr [1-D sequence of floats] Outflow bulk velocity [km/s]

logNH_Ar [1-D sequence of floats] logarithm of the neutral hydrogen column density [cm**-2]

ta_Arr [1-D sequence of floats] Dust optical depth [no dimensions]

Grid_Dictionary [python dictionary] Constains the grid to compute the escape fraction. Loaded with load_Grid_fesc().

Output:

f_esc_Arr_evaluated [1-D sequence of floats] Escape fractions for the input configurations [no dimensions]

funcs.**Linear_ND_interpolator**(*N_dim*, *Coor_props_Matrix*, *Coor_grid_list*, *Field_in_grid_Matrix*)
Interpolates in an arbitrary dimension space

N_dim [int] Number of dimensions.

Coor_props_Matrix [List of N_dim float values] Coordenates in the N_dim space to evaluate. For example [X , Y , Z]

Coor_grid_list [List of N_dim 1-D sequence of floats] For example, if there is a field evaluated in X_Arr, Y_Arr, Z_Arr [X_Arr , Y_Arr , Z_Arr]

Field_in_grid_Matrix : numpy array with the field to interpolate

Field_at_the_prob_point :

funcs.**Load_NN_model**(*Mode*, *iteration=1*)

Loads the saved parameters of the deep neural networks

Input

Mode [string] 'Inflow' or 'Outflow'

iteration [optional int] Number of the iteration. Currently only 1 Default 1

Output

machine_data [python dictionaty] Contains all the info for the DNN

funcs.**MCMC_Analysis_sampler_5**(*w_target_Arr*, *f_target_Arr*, *s_target_Arr*, *FWHM*, *N_walkers*, *N_burn*, *N_steps*, *Geometry*, *DATA_LyaRT*, *log_V_in=None*, *log_N_in=None*, *log_t_in=None*, *z_in=None*, *log_E_in=None*, *W_in=None*, *progress=True*, *FORCE_z=False*, *Inflow=False*)

Full MCMC anaylsis for the Thin_Shell_Cont

Input

w_tar_Arr [1-D sequence of floats] wavelength where the densit flux is evaluated

f_tar_Arr [1-D sequence of floats] Densit flux is evaluated

s_tar_Arr [1-D sequence of floats] Uncertainty of the densit flux is evaluated

FWHM [float] Full width half maximum [A] of the experiment.

N_walkers [int] Number of walkers

N_dim [int] Number of dimensions (6)

N_burn [int] Number of steps in the burnin-in phase

N_steps [int] Number of steps

Geometry [string] Outflow geometry to use.

DATA_LyaRT [python dictionary] Contains the grid information.

Geometry_Mode [optinal string] Changes from inflow ('Inflow') to outflow ('Outflow') Default: 'Outflow'

log_V_in [optional 1-D sequence of floats.] Range of the logarithm of the bulk velocity log_V_in[0] is the minimum log_V_in[1] is the maximum

log_N_in [optional 1-D sequence of floats.] Range of the logarithm of the neutral hydrogen column density log_N_in[0] is the minimum log_N_in[1] is the maximum

log_t_in [optional 1-D sequence of floats.] Range of the logarithm of the dust optical depth log_t_in[0] is the minimum log_t_in[1] is the maximum

z_in [optional 1-D sequence of floats.] Redshift range to be considered. z_in[0] is the minimum redshift z_in[1] is the maximum redshift

log_E_in [optional 1-D sequence of floats.] Range of the logarithm of the intrinsic equivalent width log_E_in[0] is the minimum log_E_in[1] is the maximum

W_in [optional 1-D sequence of floats.] Instrinsic line width range to be considered. W_in[0] is the minimum redshift W_in[1] is the maximum redshift

progress [optional bool] If True shows the MCMC progress. Default True

FORCE_z [optional bool] If True, force the redshift to be inside z_in

Inflow [optional bool] If True, fits and inflow instead of an outflow. Default False. So by default, it fits outflows.

Output

samples [emcee python packge object.] Contains the information of the MCMC.

funcs.**MCMC_get_region_6D**(*MODE*, *w_tar_Arr*, *f_tar_Arr*, *s_tar_Arr*, *FWHM*, *PIX*, *DATA_LyaRT*, *Geometry*, *Geometry_Mode='Outflow'*)

Computes the region of where the walkers are initialize

Input

MODE [string] Method, DNN or PSO

w_tar_Arr [1-D sequence of floats] wavelength where the densit flux is evaluated

f_tar_Arr [1-D sequence of floats] Densit flux is evaluated

s_tar_Arr [1-D sequence of floats] Uncertainty of the densit flux is evaluated

FWHM [float] Full width half maximum [A] of the experiment.

PIX [float] Pixel size in wavelgnth [A] of the experiment.

w_min [float] minimum wavelength in the observed frame [A] to use. This matches the minimum wavelgnth of wave_pix_Arr (see below).

w_max [float] maximum wavelength in the observed frame [A] to use. This might not be exactly the maximum wavelgnth of wave_pix_Arr (see below) due to pixelization.

DATA_LyaRT [python dictionary] Contains the grid information.

Geometry [string] Outflow geometry to use.

Geometry_Mode [optinal string] Changes from inflow ('Inflow') to outflow ('Outflow') Default: 'Outflow'

Output

log_V_in [1-D sequence of floats.] Range of the logarithm of the bulk velocity log_V_in[0] is the minimum log_V_in[1] is the maximum

log_N_in [1-D sequence of floats.] Range of the logarithm of the neutral hydrogen column density log_N_in[0] is the minimum log_N_in[1] is the maximum

log_t_in [1-D sequence of floats.] Range of the logarithm of the dust optical depth log_t_in[0] is the minimum log_t_in[1] is the maximum

z_in [1-D sequence of floats.] Redshift range to be considered. z_in[0] is the minimum redshift z_in[1] is the maximum redshift

log_E_in [1-D sequence of floats.] Range of the logarithm of the intrinsic equivalent width log_E_in[0] is the minimum log_E_in[1] is the maximum

W_in [1-D sequence of floats.] Instrinsic line width range to be considered. W_in[0] is the minimum redshift W_in[1] is the maximum redshift

funcs.**NN_convert_Obs_Line_to_proxy_rest_line**(*w_obs_Arr*, *f_obs_Arr*, *s_obs_Arr=None*, *normed=False*, *scaled=True*)

Converts an observed line profile to the rest frame of the maximum of the line profile.

Input

w_obs_Arr [1-D sequence of floats] wavelength where the line profile is evaluated.

f_obs_Arr [1-D sequence of floats] Flux density of the observed line profile.

s_obs_Arr [optional 1-D sequence of floats] Uncertainty in the flux density of the observed line profile.

normed [optional bool] If True, nomalizes the line profile.

scaled [optinal bool] If True, divides the line profile by its maximum.

Output

w_rest_Arr [1-D sequence of floats] wavelength where the line profile is evaluated in the rest frame of the global maximum

Delta_rest_Arr [1-D sequence of floats] w_rest_Arr - Lyman-alpha.

f_rest_Arr [1-D sequence of floats] Flux density in the rest frame of the global maximum

z_max [float] Redshift if the global maximum is the Lyaman-alpha wavelength

if s_obs_Arr is not None it also returns: s_rest_Arr : 1-D sequence of floats

Uncertainty of the flux density in the rest frame of the global maximum

funcs.**NN_generate_random_outflow_props**(*N_walkers*, *log_V_in*, *log_N_in*, *log_t_in*, *Allow_Inflows=True*)

Generates random poperties for the Thin_Shell, Galactic_Wind, etc. (Not for Thin_Shell_Cont)

Input

N_walkers [int] Number of walkers

log_V_in [optional 1-D sequence of floats.] Range of the logarithm of the bulk velocity log_V_in[0] is the minimum log_V_in[1] is the maximum

log_N_in [optional 1-D sequence of floats.] Range of the logarithm of the neutral hydrogen column density log_N_in[0] is the minimum log_N_in[1] is the maximum

log_t_in [optional 1-D sequence of floats.] Range of the logarithm of the dust optical depth log_t_in[0] is the minimum log_t_in[1] is the maximum

Allow_Inflows [optional Bool] If True it also return negative values of V in the same range. Default True

Output

init_V_Arr [1-D sequence of floats] Expansion velocity

init_log_N_Arr [1-D sequence of floats] Logarithms of the column density

init_log_t_Arr [1-D sequence of floats] Logarithms of the dust optical depth

funcs.**NN_generate_random_outflow_props_5D**(*N_walkers*, *log_V_in*, *log_N_in*, *log_t_in*, *log_E_in*, *log_W_in*, *MODE='Outflow'*)

Generates random poperties for the Thin_Shell_Cont

Input

N_walkers [int] Number of walkers

log_V_in [optional 1-D sequence of floats.] Range of the logarithm of the bulk velocity log_V_in[0] is the minimum log_V_in[1] is the maximum

log_N_in [optional 1-D sequence of floats.] Range of the logarithm of the neutral hydrogen column density log_N_in[0] is the minimum log_N_in[1] is the maximum

log_t_in [optional 1-D sequence of floats.] Range of the logarithm of the dust optical depth log_t_in[0] is the minimum log_t_in[1] is the maximum

log_E_in [optional 1-D sequence of floats.] Range of the logarithm of the instrinsic equivalent width log_E_in[0] is the minimum log_E_in[1] is the maximum

log_W_in [optional 1-D sequence of floats.] Range of the logarithm of the intrinsic width of the line log_W_in[0] is the minimum log_W_in[1] is the maximum

MODE [optional string] 'Outflow' for outflows 'Inflow' for inflows

Output

init_V_Arr [1-D sequence of floats] Expansion velocity

init_log_N_Arr [1-D sequence of floats] Logarithms of the column density

init_log_t_Arr [1-D sequence of floats] Logarithms of the dust optical depth

init_log_E_Arr [1-D sequence of floats] Logarithms of the instrinsic equivalent width

init_log_W_Arr [1-D sequence of floats] Logarithms of the intrinsic width of the line

`funcs.NN_measure`(*w_tar_Arr*, *f_tar_Arr*, *s_tar_Arr*, *FWHM_tar*, *PIX_tar*, *loaded_model*, *w_rest_Machine_Arr*, *N_iter=None*, *normed=False*, *scaled=True*, *Delta_min=- 18.5*, *Delta_max=18.5*, *Nbins_tot=1000*, *Denser_Center=True*, *Random_z_in=None*)

Generates random poperties for the Thin_Shell_Cont

Input

w_tar_Arr [1-D sequence of floats] wavelength where the densit flux is evaluated

f_tar_Arr [1-D sequence of floats] Densit flux is evaluated

s_tar_Arr [1-D sequence of floats] Uncertainty of the densit flux is evaluated

FWHM [float] Full width half maximum [A] of the experiment.

PIX_tar [float] Pixelization of the line profiles due to the experiment [A]

loaded_model [python dictionaty] Contains all the info for the DNN form Load_NN_model()

w_rest_Machine_Arr [1-D sequence of floats] wavelength used by the INPUT of the DNN

N_iter [optional int] Number of Monte Carlo iterations of the observed espectrum. If None, no iteration is done. Default None

Delta_min [optional float] Defines the minimum rest frame wavelegnth with respecto to Lyman-alpha.

Default = -18.5

Delta_min [optional float] Defines the maximum rest frame wavelegnth with respecto to Lyman-alpha.

Default = +18.5

Nbins_tot [optional int] Total number of wvelgnths bins.

Default = 1000

Denser_Center [optional bool] Populates denser the regions close to Lyman-alpha

Default = True

normed [optional bool] If True, nomalizes the line profile.

scaled [optinal bool] If True, divides the line profile by its maximum.

Random_z_in [optinal list of legnth=2] List with the minimum and maximum redshift for doing Feature importance analysis. For example [0.01,4.0]. This variable will input a random redshift with in the interval as a proxy redshift. This variable should only be used when doing a feature importance analysis. If you are not doing it, leave it as None. Otherwide you will get, probably, bad results. For example [0.01,4.0].

Output

if N_iter is None:

Sol [1-D sequence of float] Array with the solution of the observed spectrum. No Monte Carlo perturbation.

z_Sol [float] Best resdhift

if N_iter is a float: Sol and z_sol and

log_V_sol_2_Arr [1-D sequence] Logarithm of the expasion velocity

log_N_sol_2_Arr [1-D sequence] Logarithm of the neutral hydrogen column density

log_t_sol_2_Arr [1-D sequence] Logarithm of the dust optical depth

z_sol_2_Arr [1-D sequence] redshift

log_E_sol_2_Arr [1-D sequence] Logarithm of the intrinsic equivalent width

log_W_sol_2_Arr [1-D sequence] Logarithm of the instrinsic width of the line

funcs.**NN_measure_II**(*w_tar_Arr*, *f_tar_Arr*, *s_tar_Arr*, *FWHM_tar*, *PIX_tar*, *loaded_model*, *w_rest_Machine_Arr*, *N_iter=None*, *normed=False*, *scaled=True*, *Delta_min=- 10.0*, *Delta_max=10.0*, *Nbins_tot=500*, *Denser_Center=True*, *Random_z_in=None*)

Generates random poperties for the Thin_Shell_Cont

Input

w_tar_Arr [1-D sequence of floats] wavelength where the densit flux is evaluated

f_tar_Arr [1-D sequence of floats] Densit flux is evaluated

s_tar_Arr [1-D sequence of floats] Uncertainty of the densit flux is evaluated

FWHM [float] Full width half maximum [A] of the experiment.

PIX_tar [float] Pixelization of the line profiles due to the experiment [A]

loaded_model [python dictionaty] Contains all the info for the DNN form Load_NN_model()

w_rest_Machine_Arr [1-D sequence of floats] wavelength used by the INPUT of the DNN

N_iter [optional int] Number of Monte Carlo iterations of the observed espectrum. If None, no iteration is done. Default None

Delta_min [optional float] Defines the minimum rest frame wavelegnth with respecto to Lyman-alpha.

Default = -18.5

Delta_min [optional float] Defines the maximum rest frame wavelegnth with respecto to Lyman-alpha.

Default = +18.5

Nbins_tot [optional int] Total number of wvelgnths bins.

Default = 1000

Denser_Center [optional bool] Populates denser the regions close to Lyman-alpha

Default = True

normed [optional bool] If True, nomalizes the line profile.

scaled [optinal bool] If True, divides the line profile by its maximum.

Random_z_in [optinal list of legnth=2] List with the minimum and maximum redshift for doing Feature importance analysis. For example [0.01,4.0]. This variable will input a random redshift with in the interval as a proxy redshift. This variable should only be used when doing a feature importance analysis. If you are not doing it, leave it as None. Otherwide you will get, probably, bad results. For example [0.01,4.0].

Output

if N_iter is None:

Sol [1-D sequence of float] Array with the solution of the observed spectrum. No Monte Carlo perturbation.

z_Sol [float] Best resdhift

if N_iter is a float: Sol and z_sol and

log_V_sol_2_Arr [1-D sequence] Logarithm of the expasion velocity

log_N_sol_2_Arr [1-D sequence] Logarithm of the neutral hydrogen column density

log_t_sol_2_Arr [1-D sequence] Logarithm of the dust optical depth

z_sol_2_Arr [1-D sequence] redshift

log_E_sol_2_Arr [1-D sequence] Logarithm of the intrinsic equivalent width

log_W_sol_2_Arr [1-D sequence] Logarithm of the instrinsic width of the line

funcs.**PSO_Analysis**(*w_tar_Arr*, *f_tar_Arr*, *FWHM*, *PIX*, *DATA_LyaRT*, *Geometry*, *n_particles*, *n_iters*)

Does a PSO analysis to find in a fast way a close good fit

Input

w_tar_Arr [1-D sequence of float] wavelength where the observed density flux is evaluated.

f_tar_Arr [1-D sequence of float] Observed flux density

FWHM [float] Full width half maximum [A] of the experiment.

PIX [float] Pixel size in wavelgnth [A] of the experiment.

DATA_LyaRT [python dictionary] Contains the grid information.

Geometry [string] Outflow geometry to use.

n_particles [int] Number of particles in the PSO

n_iters [int] Number of steps in the PSO

Output

cost [float] Cost of the best configuration

pos [1-D sequence of floats] Position of the best configuration

funcs.**PSO_compute_xi_2_MANY**(*X*, *w_tar_Arr*, *f_tar_Arr*, *FWHM*, *PIX*, *DATA_LyaRT*, *Geometry*)

Compute the chi esquare for the PSO analysis for many configurations

Input

x: 1-D sequence of float

Contains the parameters of the mode: x[0] = logarithim of the expansion velocity x[1] = logarithim of the neutral hydrogen column density x[2] = logarithim of the dust optical depth x[3] = redshift x[4] = logarithm of the intrinsic equivalent width x[5] = intrinsic width

w_tar_Arr [1-D sequence of float] wavelength where the observed density flux is evaluated.

f_tar_Arr [1-D sequence of float] Observed flux density

FWHM [float] Full width half maximum [A] of the experiment.

PIX [float] Pixel size in wavelgnth [A] of the experiment.

DATA_LyaRT [python dictionary] Contains the grid information.

Geometry [string] Outflow geometry to use.

Output

xi_2_Arr [1-D sequence of float] Chi square of the configurations

funcs.**PSO_compute_xi_2_ONE_6D**(*x*, *w_tar_Arr*, *f_tar_Arr*, *FWHM*, *PIX*, *DATA_LyaRT*, *Geometry*, *T_IGM_Arr=None*, *w_IGM_Arr=None*)

Compute the chi esquare for the PSO analysis

Input

x: 1-D sequence of float

Contains the parameters of the mode: x[0] = logarithim of the expansion velocity x[1] = logarithim of the neutral hydrogen column density x[2] = logarithim of the dust optical depth x[3] = redshift x[4] = logarithm of the intrinsic equivalent width x[5] = intrinsic width

w_tar_Arr [1-D sequence of float] wavelength where the observed density flux is evaluated.

f_tar_Arr [1-D sequence of float] Observed flux density

FWHM [float] Full width half maximum [A] of the experiment.

PIX [float] Pixel size in wavelgnth [A] of the experiment.

DATA_LyaRT [python dictionary] Contains the grid information.

Geometry [string] Outflow geometry to use.

Output

xi_2 [float] Chi square of the configuration

w_pso_Arr [1-D sequence of floats] Wavelgnth of the line profile computed by the PSO

my_f_pso_Arr [1-D sequence of floats] Flux density of the line profile computed by the PSO

funcs.**Prior_f**(*theta*)

Decides when a walker from the MCMC is out for the Thin_Shell, Galactic wind and bicones.

Input

theta [1-D sequence of float]

Contains the parameters of the mode: theta[0] = logarithim of the expansion velocity theta[1] = logarithim of the neutral hydrogen column density theta[2] = logarithim of the dust optical depth

Output

True if the walker is inside False if the walker is outside

funcs.**Prior_f_5**(*theta*)

Decides when a walker from the MCMC is out for the Thin_Shell_Cont,

Input

theta [1-D sequence of float]

Contains the parameters of the mode: theta[0] = logarithim of the expansion velocity theta[1] = logarithim of the neutral hydrogen column density theta[2] = logarithim of the dust optical depth theta[3] = redshift theta[4] = logarithm of the intrinsic equivalent width theta[5] = intrinsic width

Output

True if the walker is inside False if the walker is outside

funcs.**RT_Line_Profile**(*Geometry*, *wavelength_Arr*, *V_Arr*, *logNH_Arr*, *ta_Arr*, *logEW_Arr=None*, *Wi_Arr=None*, *MODE_CONT='FULL'*)

Return the Lyman alpha line profile for a given outflow properties.

Input:

Geometry [string] The outflow geometry to use: Options: 'Thins_Shell', 'Galactic_Wind' , 'Bicone_X_Slab', 'Thin_Shell_Cont'

wavelength_Arr [1-D sequence of floats] Array with the wavelength vales where the line profile is computed. The units are meters, i.e., amstrongs * 1.e-10.

V_Arr [1-D sequence of float] Array with the expansion velocity of the outflow. The unit are km/s.

logNH_Arr [1-D sequence of float] Array with the logarithim of the outflow neutral hydrogen column density. The units of the colum density are in c.g.s, i.e, cm**-2.

ta_Arr [1-D sequence of float] Array with the dust optic depth of the outflow.

ta_Value [1-D sequence of bool] Dust optical depth [no dimensions]

logEW_Value [Optional 1-D sequence of bool] Logarithm of rest frame equiavlent width [A] Default = None

Wi_Value [Optional 1-D sequence of bool] Intrinsic width line in the rest frame [A] Default = None

MODE [optinal string.] For the 'Thin_Shell_Cont' ONLY. Defines the grid to be loaded. MODE_CONT='FULL' loads a very dense grid. ~12GB of RAM. MODE_CONT='LIGHT' loads a more sparse grid. ~ 2GB of RAM.

Output:

lines_Arr [2-D sequence of float] The Lyman alpha line profiles. lines_Arr[i] is the line profile computed at the wavelengths wavelength_Arr for wich V_Arr[i] , logNH_Arr[i] , ta_Arr[i] , Inside_Bicone_Arr[i].

funcs.**RT_Line_Profile_MCMC**(*Geometry*, *wavelength_Arr*, *V_Value*, *logNH_Value*, *ta_Value*, *DATA_LyaRT*, *logEW_Value=None*, *Wi_Value=None*)

Return one and only one Lyman alpha line profile for a given outflow properties. This function is especial to run MCMCs or PSO.

Input:

Geometry [string] The outflow geometry to use: Options: 'Thins_Shell', 'Galactic_Wind' , 'Bicone_X_Slab', 'Thin_Shell_Cont'

wavelength_Arr [1-D sequence of floats] Array with the wavelength vales where the line profile is computed. The units are meters, i.e., amstrongs * 1.e-10.

V_Value [float] Value of the expansion velocity of the outflow. The unit are km/s.

logNH_Value [float] Value of the logarithim of the outflow neutral hydrogen column density. The units of the colum density are in c.g.s, i.e, cm**-2.

ta_Value [float] Value of the dust optic depth of the outflow.

DATA_LyaRT [Dictionay] This dictonary have all the information of the grid. This dictionary can be loaded with the function : load_Grid_Line, for example:

DATA_LyaRT = load_Grid_Line('Thin_Shell')

Output:

lines_Arr [1-D sequence of float] The Lyman alpha line profile.

funcs.**RT_f_esc**(*Geometry*, *V_Arr*, *logNH_Arr*, *ta_Arr*, *MODE='Parametrization'*, *Algorithm='Intrepolation'*, *Machine_Learning_Algorithm='Tree'*)

Return the Lyman alpha escape fraction for a given outflow properties.

Input

Geometry [string] The outflow geometry to use: Options: 'Thins_Shell', 'Galactic_Wind' , 'Bicone_X_Slab'.

V_Arr [1-D sequence of float] Array with the expansion velocity of the outflow. The unit are km/s.

logNH_Arr [1-D sequence of float] Array with the logarithim of the outflow neutral hydrogen column density. The units of the colum density are in c.g.s, i.e, cm**-2.

ta_Arr [1-D sequence of float] Array with the dust optic depth of the outflow.

MODE [optional string]

Set the mode in which the escape fraction is computed. It can be:

Analytic : it uses an analytic equation fitted to the output of the RT MC code. Parametrization : it computes the escape fraction using a function that depends on the

dust optical depts as in Neufeld et al. 1990.

Raw : it uses directly the output of the RT MC code.

Default = 'Parametrization'

Algorithm [optional string]

Set how the escape fraction is computed. If MODE='Analytic' then this varialbe is useless. Intrepolation : Direct lineal interpolation. Machine_Learning : uses machine learning algorithms

Default = 'Intrepolation'

Machine_Learning_Algorithm [optial string]

Set the machine learning algorith used. Available: Tree : decision tree Forest : random forest KN : KN

Default = 'Tree'

Output

lines_Arr [1-D sequence of float] The Lyman alpha escape fraction for V_Arr[i] , logNH_Arr[i] , ta_Arr[i] , Inside_Bicone_Arr[i].

funcs.**RT_f_esc_Analytic**(*Geometry*, *V_Arr*, *logNH_Arr*, *ta_Arr*)

Return the escape fraction computed analytically (Gurung-lopez et al. 2019a, 2019b)

Input: Geometry : String

Outflow configuration to use: 'Thin_Shell' , 'Galactic_Wind'

V_Arr [1-D sequence of floats] Outflow bulk velocity [km/s]

logNH_Arr [1-D sequence of floats] logarithm of the neutral hydrogen column density [cm**-2]

ta_Arr [1-D sequence of floats] Dust optical depth [no dimensions]

Output:

fesc [1-D sequence of floats] Escape fractions for the input configurations [no dimensions]

funcs.**RT_f_esc_Interpolation_Parameters**(*Geometry*, *V_Arr*, *logNH_Arr*, *ta_Arr*, *Machine_Learning_Algorithm=None*)

Computes the escape fraction using the escape fraction grids of parameters

Input:

Geometry [String]

Outflow configuration to use: 'Thin_Shell' , 'Galactic_Wind' , 'Bicone_X_Slab_In' , 'Bicone_X_Slab_Out'

V_Arr [1-D sequence of floats] Outflow bulk velocity [km/s]

logNH_Ar [1-D sequence of floats] logarithm of the neutral hydrogen column density [cm**-2]

ta_Arr [1-D sequence of floats] Dust optical depth [no dimensions]

Machine_Learning_Algorithm [String] Kind of algorithm: 'KN', 'Grad', 'Tree' or 'Forest'

Output:

f_esc_Arr [1-D sequence of floats] Escape fractions for the input configurations [no dimensions]

funcs.**RT_f_esc_Interpolation_Values**(*Geometry*, *V_Arr*, *logNH_Arr*, *ta_Arr*, *Machine_Learning_Algorithm=None*)

Computes the escape fraction using the escape fraction grids of values

Input:

Geometry [String]

Outflow configuration to use: 'Thin_Shell' , 'Galactic_Wind' , 'Bicone_X_Slab_In' , 'Bicone_X_Slab_Out'

V_Arr [1-D sequence of floats] Outflow bulk velocity [km/s]

logNH_Ar [1-D sequence of floats] logarithm of the neutral hydrogen column density [cm**-2]

ta_Arr [1-D sequence of floats] Dust optical depth [no dimensions]

Machine_Learning_Algorithm [String] Kind of algorithm: 'KN', 'Grad', 'Tree' or 'Forest'

Output:

f_esc_Arr [1-D sequence of floats] Escape fractions for the input configurations [no dimensions]

funcs.**RT_f_esc_Machine_Parameter**(*Geometry*, *V_Arr*, *logNH_Arr*, *ta_Arr*, *Machine_Learning_Algorithm='Tree'*)

Anallytic expression of the escape fraction as a function of the dust optical depth (Gurung-lopez et al. 2019b). This uses the parametric expression of the escape fraction.

Input:

Geometry [String]

Outflow configuration to use: 'Thin_Shell' , 'Galactic_Wind' , 'Bicone_X_Slab_In' , 'Bicone_X_Slab_Out'

V_Arr [1-D sequence of floats] Outflow bulk velocity [km/s]

logNH_Ar [1-D sequence of floats] logarithm of the neutral hydrogen column density [cm**-2]

ta_Arr [1-D sequence of floats] Dust optical depth [no dimensions]

Machine_Learning_Algorithm [string] Machine learning algorithm: 'KN', 'Grad', 'Tree' or 'Forest'

Output:

f_esc_Arr [1-D sequence of floats] Escape fractions for the input configurations [no dimensions]

funcs.**RT_f_esc_Machine_Values**(*Geometry*, *V_Arr*, *logNH_Arr*, *ta_Arr*, *Machine_Learning_Algorithm='Tree'*)

Anallytic expression of the escape fraction as a function of the dust optical depth (Gurung-lopez et al. 2019b). This uses the directly the escape fraction.

Input:

Geometry [String]

Outflow configuration to use: 'Thin_Shell' , 'Galactic_Wind' , 'Bicone_X_Slab_In' , 'Bicone_X_Slab_Out'

V_Arr [1-D sequence of floats] Outflow bulk velocity [km/s]

logNH_Ar [1-D sequence of floats] logarithm of the neutral hydrogen column density [cm**-2]

ta_Arr [1-D sequence of floats] Dust optical depth [no dimensions]

Machine_Learning_Algorithm [string] Machine learning algorithm: 'KN', 'Grad', 'Tree' or 'Forest'

Output:

f_esc_Arr [1-D sequence of floats] Escape fractions for the input configurations [no dimensions]

funcs.**Signal_to_noise_estimator**(*w_Arr*, *Line_Arr*, *w_line*)
Estimates the signal to noise of a line profile

Input

w_Arr [1-D sequence of float] wavelgnth array

Line_Arr [1-D sequence float] Flux density of the line profile.

w_line [float] wavelgnth of the line

Output

SNR [float] Signal to noise ratio.

funcs.**Test_1**()
Script to test if everything is working fine.

funcs.**Test_2**()
Script to test if everything looks fine.

funcs.**Treat_A_Line_To_NN_Input**(*w_Arr*, *f_Arr*, *PIX*, *FWHM*, *Delta_min=- 18.5*, *Delta_max=18.5*, *Nbins_tot=1000*, *Denser_Center=True*, *normed=False*, *scaled=True*)
Convert a line profile and the usefull information into the INPUT of the NN.

Input

w_Arr [1-D sequence of floats] Wavelgnth of the line profile in the observed frame. [A]

f_Arr [1-D sequence of floats] Flux density of the observed line profile in arbitrary units.

FWHM [float] Full width half maximum [A] of the experiment.

PIX [float] Pixel size in wavelgnth [A] of the experiment.

Delta_min [optional float] Defines the minimum rest frame wavelegnth with respecto to Lyman-alpha.

Default = -18.5

Delta_min [optional float] Defines the maximum rest frame wavelegnth with respecto to Lyman-alpha.

Default = +18.5

Nbins_tot [optional int] Total number of wvelgnths bins.

Default = 1000

Denser_Center [optional bool] Populates denser the regions close to Lyman-alpha

Default = True

normed [optional bool] If True, nomalizes the line profile.

scaled [optinal bool] If True, divides the line profile by its maximum.

Output

rest_w_Arr [1-D sequence of float] Wavelgnth array where the line is evaluated in the rest frame.

NN_line [1-D sequence of float] Line profile evaluated in rest_w_Arr after normalization or scaling.

z_max_i [float] Redshift of the source if the global maximum of the spectrum is the Lyman-alpha wavelegth.

INPUT: 1-D secuence of float The actuall input to use in the Neural networks.

funcs.**Treat_A_Line_To_NN_Input_II**(*w_Arr*, *f_Arr*, *PIX*, *FWHM*, *Delta_min=- 10.0*, *Delta_max=10.0*, *Nbins_tot=500*, *Denser_Center=True*, *normed=False*, *scaled=True*)

Convert a line profile and the usefull information into the INPUT of the NN.

Input

w_Arr [1-D sequence of floats] Wavelgnth of the line profile in the observed frame. [A]

f_Arr [1-D sequence of floats] Flux density of the observed line profile in arbitrary units.

FWHM [float] Full width half maximum [A] of the experiment.

PIX [float] Pixel size in wavelgnth [A] of the experiment.

Delta_min [optional float] Defines the minimum rest frame wavelegnth with respecto to Lyman-alpha.

Default = -18.5

Delta_min [optional float] Defines the maximum rest frame wavelegnth with respecto to Lyman-alpha.

Default = +18.5

Nbins_tot [optional int] Total number of wvelgnths bins.

Default = 1000

Denser_Center [optional bool] Populates denser the regions close to Lyman-alpha

Default = True

normed [optional bool] If True, nomalizes the line profile.

scaled [optinal bool] If True, divides the line profile by its maximum.

Output

rest_w_Arr [1-D sequence of float] Wavelgnth array where the line is evaluated in the rest frame.

NN_line [1-D sequence of float] Line profile evaluated in rest_w_Arr after normalization or scaling.

z_max_i [float] Redshift of the source if the global maximum of the spectrum is the Lyman-alpha wavelegth.

INPUT: 1-D secuence of float The actuall input to use in the Neural networks.

funcs.**bin_one_line**(*wave_Arr_line*, *Line_Prob_Arr*, *new_wave_Arr*, *Bin*, *same_norm=False*)

This functions bins the line profile mimicking the pixelization in a CCD.

Input

wave_Arr_line [1-D sequence of float] Array with the Wavelength where the spectrum is evaluated. Same units as Bin. This has to be sorted.

Line_Prob_Arr [1-D sequence of float] Arrays with the flux of the spectrum.

new_wave_Arr [1-D sequence of float] Array with the nex wavelgnth where the fline profile will be interpolated

Bin [float] Bin size.

same_norm [optional bool.] If true return a line with the same normalization as the input

Output

binned_line [1-D sequence of float] Spectrum after the convolution

funcs.**convert_gaussian_FWHM_to_sigma**(*FWHM_Arr*)
This function computes the sigma of a gaussian from its FWHM.

Input

FWHM_Arr [1-D sequence of float] Array with the Full Width Half Maximum that you want to convert

Output

sigma_Arr [1-D sequence of float] The width of the FWHM_Arr

funcs.**convert_lamda_into_x**(*lamda*, *T4=None*)
This function converts from frequency in Doppler units to wavelength

Input:

lamda [1-D sequence of float] wavelength

T4 [optional float] Temperature in units of 10**4 K: T4 = T[k] / 10**4

Output:

x_Arr [1-D sequence of float] Frequency in Doppler units.

funcs.**convert_x_into_lamda**(*x*, *T4=None*)
This function converts from frequency in Doppler units to wavelength

Input:

x [1-D sequence of float] Frequency in Doppler units.

T4 [optional float] Temperature in units of 10**4 K: T4 = T[k] / 10**4

Output:

w_Arr [1-D sequence of float] wavelength.

funcs.**define_RT_parameters**(*T4=None*)
This function gives the parameters use to compute useful variables when working with radiative transfer.

Input:

T4 [optional float] Temperature in units of 10**4 K: T4 = T[k] / 10**4

Output:

nu0 [float] Lyaman-alpha frequency.

Dv : float

funcs.**dilute_line**(*wave_Arr*, *Spec_Arr*, *FWHM*)
This functions dilutes a given spectrum by convolving with a gaussian filter.

Input

wave_Arr [1-D sequence of float] Array with the Wavelength where the spectrum is evaluated. Same units as FWHM_Arr. This has to be sorted.

Spec_Arr [1-D sequence of float] Arrays with the flux of the spectrum.

FWHM_Arr [1-D sequence of float] Array with the Full width half maximuum of of the gaussian to convolve. If FWHM_Arr is a single value, it uses the same value across the x_Arr range. If FWHM is a 1-D sequence, a different value of width of the gaussian is used. In this case, the length of this array has to be the same as wave_Arr and Spec_Arr.

same_norm [optional bool.] If true return a line with the same normalization as the input

Output

new_Line [1-D sequence of float] Spectrum after the convolution

funcs.**dilute_line_changing_FWHM**(*wave_Arr*, *Spec_Arr*, *FWHM_Arr*, *same_norm=False*)
This functions dilutes a given spectrum by convolving with a gaussian filter.

Input

wave_Arr [1-D sequence of float] Array with the Wavelength where the spectrum is evaluated. Same units as FWHM_Arr. This has to be sorted.

Spec_Arr [1-D sequence of float] Arrays with the flux of the spectrum.

FWHM_Arr [1-D sequence of float] Array with the Full width half maximuum of of the gaussian to convolve. If FWHM_Arr is a single value, it uses the same value across the x_Arr range. If FWHM is a 1-D sequence, a different value of width of the gaussian is used. In this case, the length of this array has to be the same as wave_Arr and Spec_Arr.

same_norm [optional bool.] If true return a line with the same normalization as the input

Output

new_Line [1-D sequence of float] Spectrum after the convolution

funcs.**fesc_of_ta_Bicone**(*ta*, *CCC*, *KKK*, *LLL*)
Anallytic expression of the escape fraction as a function of the dust optical depth (Gurung-lopez et al. 2019b)

Input:

ta [1-D sequence of floats] Dust optical depth [no dimensions]

CCC [float] CCC parameter

KKK [float] KKK parameter

LLL [float] LLL parameter

Output:

fesc [1-D sequence of floats] Escape fractions for the input configurations [no dimensions]

funcs.**fesc_of_ta_Thin_and_Wind**(*ta*, *CCC*, *KKK*)
Anallytic expression of the escape fraction as a function of the dust optical depth (Gurung-lopez et al. 2019a)

Input:

ta [1-D sequence of floats] Dust optical depth [no dimensions]

CCC [float] CCC parameter

KKK [float] KKK parameter

Output:

fesc [1-D sequence of floats] Escape fractions for the input configurations [no dimensions]

funcs.**gaus**(*x_Arr*, *A*, *mu*, *sigma*)
Retruns a gaussian evaluated in x_Arr.

Input

x_Arr [1-D sequence of float] Where the gaussian has to be evaluated.

A [float] Amplitude

mu [float] Mean

sigma [float] width

Output

gauss_Arr [1-D sequence of float] Gaussian

funcs.**generate_a_REAL_line_Noise_w**(*z_f*, *V_f*, *logNH_f*, *ta_f*, *F_line_f*, *logEW_f*, *Wi_f*, *Noise_w_Arr*, *Noise_Arr*, *FWHM_f*, *PIX_f*, *w_min*, *w_max*, *DATA_LyaRT*, *Geometry*, *T_IGM_Arr=None*, *w_IGM_Arr=None*)

Makes a mock line profile for the Thin_Shell_Cont geometry.

Input

z_f [float] Redshift

V_f [float] Outflow expansion velocity [km/s]

logNH_f [float] logarithmic of the neutral hydrogen column density in cm**-2

ta_f [float] Dust optical depth

logEW_f [optional, float] Logarithmic of the rest frame intrisic equivalent width of the line [A] Requiered if Geometry == 'Thin_Shell_Cont'

Wi_f [optional, float] Rest frame intrisic width of the Lyman-alpha line [A] Requiered if Geometry == 'Thin_Shell_Cont'

Noise_w_Arr [1-D sequence of float] wavelength array where the noise pattern is evaluated.

Noise_Arr [1-D sequence of float] Noise pattern. Evolution of the noise as a function of wavelength (Noise_w_Arr)

FWHM_f [float] Full width half maximum [A] of the experiment. This dilutes the line profile.

PIX_f [float] Pixel size in wavelgnth [A] of the experiment. This binnes the line profile.

w_min [float] minimum wavelength in the observed frame [A] to use. This matches the minimum wavelgnth of wave_pix_Arr (see below).

w_max [float] maximum wavelength in the observed frame [A] to use. This might not be exactly the maximum wavelgnth of wave_pix_Arr (see below) due to pixelization.

DATA_LyaRT [python dictionary] Contains the grid information.

Geometry [string] Outflow geometry to use.

Output

wave_pix_Arr [1-D sequence of float] Wavelgnth array where the line is evaluated in the observed frame.

noisy_Line_Arr [1-D sequence of float] Line profile flux density in arbitrary units.

dic [python dictionaty.]

Contains all the meta data used to get the line profiles: 'w_rest' : restframe wavelength of line before reducing quality 'w_obs' : wavelength of line before reducing quality 'Intrinsic' : line profile before quality reduction 'Diluted' : Line profile after the FWHM has been applyed. 'Pixelated' : Line profile after the FWHM and PIX have been applyed 'Noise' : Particular noise patern used.

funcs.**generate_a_obs_line**(*z_f*, *V_f*, *logNH_f*, *ta_f*, *DATA_LyaRT*, *Geometry*, *logEW_f=None*, *Wi_f=None*, *T_IGM_Arr=None*, *w_IGM_Arr=None*, *RETURN_ALL=False*)

Moves in redshift a line profile.

Input

z_f [float] Redshift

V_f [float] Outflow expansion velocity [km/s]

logNH_f [float] logarithmic of the neutral hydrogen column density in cm**-2

ta_f [float] Dust optical depth

DATA_LyaRT [python dictionary] Contains the grid information.

Geometry [string] Outflow geometry to use.

logEW_f [optional, float] Logarithmic of the rest frame intrisic equivalent width of the line [A] Requiered if Geometry == 'Thin_Shell_Cont'

Wi_f [optional, float] Rest frame intrisic width of the Lyman-alpha line [A] Requiered if Geometry == 'Thin_Shell_Cont'

Output

w_rest_Arr [1-D sequence of float] Wavelgnth array where the line is evaluated in the rest frame.

wavelength_Arr [1-D sequence of float] Wavelgnth array where the line is evaluated in the observed frame.

line_Arr [1-D sequence of float] Line profile flux density in arbitrary units.

funcs.**get_solutions_from_flat_chain**(*flat_chains*, *Q_Arr*)
function to get the solution from the emcee sampler sin some given percentiles.

Input

flat_samples [2-D sequence of floats] The MCMC chains but flat.

Q_Arr [1-D list of floats] List of the percentiles that will be computed, for example [16.0 , 50.0 , 84.0]

Output

matrix_sol: 2-D sequence of floats The percentiles of each of the 6 properties.

funcs.**get_solutions_from_sampler**(*sampler*, *N_walkers*, *N_burn*, *N_steps*, *Q_Arr*)
function to get the solution from the emcee sampler sin some given percentiles.

Input

sampler [emcee python packge object.] Contains the information of the MCMC.

N_walkers [int] Number of walkers

N_burn [int] Number of steps in the burnin-in phase

N_steps [int] Number of steps

Q_Arr [1-D list of floats] List of the percentiles that will be computed, for example [16.0 , 50.0 , 84.0]

Output

matrix_sol: 2-D sequence of floats The percentiles of each of the 6 properties.

flat_samples [2-D sequence of floats] The MCMC chains but flat.

funcs.**get_solutions_from_sampler_mean**(*sampler*, *N_walkers*, *N_burn*, *N_steps*)
function to get the solution from the emcee sampler as the mean.

Input

sampler [emcee python packge object.] Contains the information of the MCMC.

N_walkers [int] Number of walkers

N_burn [int] Number of steps in the burnin-in phase

N_steps [int] Number of steps

Output

matrix_sol: 1-D sequence of floats Mean of each of the 6 properties.

flat_samples [2-D sequence of floats] The MCMC chains but flat.

funcs.**get_solutions_from_sampler_peak**(*sampler*, *N_walkers*, *N_burn*, *N_steps*, *N_hist_steps*)
function to get the solution from the emcee sampler as the global maximum of the distribution of the posteriors.

Input

sampler [emcee python packge object.] Contains the information of the MCMC.

N_walkers [int] Number of walkers

N_burn [int] Number of steps in the burnin-in phase

N_steps [int] Number of steps

N_hist_steps [int] Number of bins to sample the PDF of all properties

Output

matrix_sol: 1-D sequence of floats Mean of each of the 6 properties.

flat_samples [2-D sequence of floats] The MCMC chains but flat.

funcs.**init_walkers_5**(*N_walkers*, *N_dim*, *log_V_in*, *log_N_in*, *log_t_in*, *z_in*, *log_E_in*, *W_in*)
Creates the initial position for the walkers

Input

N_walkers [int] Number of walkers

N_dim [int] Number of dimensions (6)

log_V_in [1-D sequence of floats.] Range of the logarithm of the bulk velocity log_V_in[0] is the minimum log_V_in[1] is the maximum

log_N_in [1-D sequence of floats.] Range of the logarithm of the neutral hydrogen column density log_N_in[0] is the minimum log_N_in[1] is the maximum

log_t_in [1-D sequence of floats.] Range of the logarithm of the dust optical depth log_t_in[0] is the minimum log_t_in[1] is the maximum

z_in [1-D sequence of floats.] Redshift range to be considered. z_in[0] is the minimum redshift z_in[1] is the maximum redshift

log_E_in [1-D sequence of floats.] Range of the logarithm of the intrinsic equivalent width log_E_in[0] is the minimum log_E_in[1] is the maximum

W_in [1-D sequence of floats.] Instrinsic line width range to be considered. W_in[0] is the minimum redshift W_in[1] is the maximum redshift

Output

theta_0 [1-D sequence of float]

Contains the parameters of the mode: theta_0[:,0] = logarithim of the expansion velocity theta_0[:,1] = logarithim of the neutral hydrogen column density theta_0[:,2] = logarithim of the dust optical depth theta_0[:,3] = redshift theta_0[:,4] = logarithm of the intrinsic equivalent width theta_0[:,5] = intrinsic width

funcs.**load_Grid_Line**(*Geometry*, *MODE='FULL'*)
Return the dictionary with all the properties of the grid where the lines were run.

Input

Geometry [string] The outflow geometry to use: Options: 'Thins_Shell', 'Galactic_Wind' , 'Bicone_X_Slab_In', 'Bicone_X_Slab_Out', 'Thin_Shell_Cont'.

MODE [optinal string.] For the 'Thin_Shell_Cont' ONLY. Defines the grid to be loaded. MODE='FULL' loads a very dense grid. ~12GB of RAM. MODE='LIGHT' loads a more sparse grid. ~ 2GB of RAM.

Output

loaded_model [Dictionary] This dictonary have all the information of the grid. Entries:

'V_Arr' : Array of velocity expansions used.[km/s] 'logNH_Arr' : Array of logarithm of the column density. [c.g.s.] 'logta_Arr' : Array of logarithm of the dust optical depth. 'x_Arr' : Array of frequency in Doppler units. 'Grid' : Array with the output of the RT MC code LyaRT:

loaded_model['Grid'][i,j,k,:] has the line profile evaluated in loaded_model['x_Arr'] with outflow velocity loaded_model['V_Arr'][i] , logarithm of the neutral hydrogen column density loaded_model['logNH_Arr'][j] and logarithm of dust optical depth loaded_model['logta_Arr'][k]

funcs.**load_Grid_fesc**(*Geometry*, *MODE*)

This functions gives you grids of the escape fraction

Input:

Geometry [String]

Outflow configuration to use: 'Thin_Shell' , 'Galactic_Wind' , 'Bicone_X_Slab_In' , 'Bicone_X_Slab_Out'

MODE [String] Parametrization of the escape fraction. 'Parameters' or 'values'

Output:

loaded_model : file the grid of f_esc parameters/values.

funcs.**load_machine_fesc**(*Machine*, *property_name*, *Geometry*)

This functions gives you the trained model that you want to use.

Input:

Machine [String] Kind of algorithm: 'KN', 'Grad', 'Tree' or 'Forest'

property_name [String] The variable to import: 'KKK' , 'CCC' , 'LLL' or 'f_esc'

Geometry [String]

Outflow configuration to use: 'Thin_Shell' , 'Galactic_Wind' , 'Bicone_X_Slab_In' , 'Bicone_X_Slab_Out'

Output:

loaded_model : file with all the necesary to do machine learning

funcs.**log_likeliehood_of_model_5**(*theta*, *w_obs_Arr*, *f_obs_Arr*, *s_obs_Arr*, *FWHM*, *PIX*, *w_min*, *w_max*, *DATA_LyaRT*, *Geometry*, *z_in*, *FORCE_z=False*, *Inflow=False*, *T_IGM_Arr=None*, *w_IGM_Arr=None*)

Logarithm of the likelihood between an observed spectrum and a model configuration defined in theta

Input

theta [1-D sequence of float]

Contains the parameters of the mode: theta[0] = logarithim of the expansion velocity theta[1] = logarithim of the neutral hydrogen column density theta[2] = logarithim of the dust optical depth theta[3] = redshift theta[4] = logarithm of the intrinsic equivalent width theta[5] = intrinsic width

w_obs_Arr [1-D sequence of float] wavelength where the observed density flux is evaluated.

f_obs_Arr [1-D sequence of float] Observed flux density

s_obs_Arr [1-D sequence of float] Uncertanty in the observed flux density.

FWHM [float] Full width half maximum [A] of the experiment.

PIX [float] Pixel size in wavelgnth [A] of the experiment.

w_min [float] minimum wavelength in the observed frame [A] to use. This matches the minimum wavelgnth of wave_pix_Arr (see below).

w_max [float] maximum wavelength in the observed frame [A] to use. This might not be exactly the maximum wavelgnth of wave_pix_Arr (see below) due to pixelization.

DATA_LyaRT [python dictionary] Contains the grid information.

Geometry [string] Outflow geometry to use.

z_in [1-D sequence of floats.] Redshift range to be considered. In principle the redshift can be outside z_in[0] is the minimum redshift z_in[1] is the maximum redshift

FORCE_z [optional bool] If True, force the redshift to be inside z_in

Inflow [optional bool] If True, fits and inflow instead of an outflow. Default False. So by default, it fits outflows.

Output

log_like [float] Logarithm of the likelihood

funcs.**log_likelihood**(*w_obs_Arr*, *f_obs_Arr*, *s_obs_Arr*, *w_model_Arr*, *f_model_Arr*)

Logarithm of the likelihood between an observed spectrum and a model spectrum.

Input

w_obs_Arr [1-D sequence of float] wavelength where the observed density flux is evaluated.

f_obs_Arr [1-D sequence of float] Observed flux density

s_obs_Arr [1-D sequence of float] Uncertanty in the observed flux density.

w_model_Arr [1-D sequence of float] wavelength where the model density flux is evaluated

f_model_Arr [1-D sequence of float] Model flux density

Output

log_like [float] Logarithm of the likelihood

funcs.**plot_a_rebinned_line**(*new_wave_Arr*, *binned_line*, *Bin*)

This functions is used to plot line profiles. It transforms the line line in a histogram.

Input

new_wave_Arr [1-D sequence of float] Array with the Wavelength where the spectrum is evaluated.

binned_line [1-D sequence of float] Arrays with the flux of the spectrum.

Bin [float] Bin size.

Output

XX_Arr [1-D sequence of float] Wavelength where the new line is evaluated

YY_Arr [1-D sequence of float] Flux density array

funcs.**pre_treatment_Line_profile**(*Geometry*, *V_Arr*, *logNH_Arr*, *ta_Arr*, *logEW_Arr=None*, *Wi_Arr=None*)

Checks the inflow/outflow parameters before doing the proper computation.

Input:

Geometry [String]

Outflow configuration to use: 'Thin_Shell' , 'Galactic_Wind'

, 'Bicone_X_Slab_In' , 'Bicone_X_Slab_Out', 'Thin_Shell_Cont'

V_Value [1-D sequence of bool] Outflow bulk velocity [km/s]

logNH_Value [1-D sequence of bool] logarithm of the neutral hydrogen column density [cm**-2]

ta_Value [1-D sequence of bool] Dust optical depth [no dimensions]

logEW_Value [Optional 1-D sequence of bool] Logarithm of rest frame equiavlent width [A] Default = None

Wi_Value [Optional 1-D sequence of bool] Intrinsic width line in the rest frame [A] Default = None

Output:

Bool_good [1-D sequence of bool] 1 if the parameters are good, 0 if they are bad.

funcs.**pre_treatment_Line_profile_MCMC**(*Geometry*, *V_Value*, *logNH_Value*, *ta_Value*, *logEW_Value=None*, *Wi_Value=None*)

Checks the inflow/outflow parameters before doing the proper computation.

Input:

Geometry [String]

Outflow configuration to use: 'Thin_Shell' , 'Galactic_Wind'

, 'Bicone_X_Slab_In' , 'Bicone_X_Slab_Out' 'Thin_Shell_Cont'

V_Value [float] Outflow bulk velocity [km/s]

logNH_Value [float] logarithm of the neutral hydrogen column density [cm**-2]

ta_Value [float] Dust optical depth [no dimensions]

logEW_Value [Optinal float] Logarithm of rest frame equiavlent width [A] Default = None

Wi_Value [Optinal float] Intrinsic width line in the rest frame [A] Default = None

Output:

Bool_good [1-D sequence of bool] 1 if the parameters are good, 0 if they are bad.

funcs.**pre_treatment_f_esc**(*Geometry*, *V_Arr*, *logNH_Arr*, *ta_Arr*, *MODE*)

Checks the inflow/outflow parameters before doing the proper computation.

Input:

Geometry [String]

Outflow configuration to use: 'Thin_Shell' , 'Galactic_Wind' , 'Bicone_X_Slab_In' , 'Bicone_X_Slab_Out'

V_Arr [1-D sequence of floats] Outflow bulk velocity [km/s]

logNH_Ar [1-D sequence of floats] logarithm of the neutral hydrogen column density [cm**-2]

ta_Arr [1-D sequence of floats] Dust optical depth [no dimensions]

MODE [optional string]

Set the mode in which the escape fraction is computed. It can be: Analytic : it uses an analytic equation fitted to the output of the RT MC code. Parametrization : it computes the escape fraction using a function that depends on the

dust optical depts as in Neufeld et al. 1990.

Raw : it uses directly the output of the RT MC code.

Default = 'Parametrization'

Kind of algorithm: 'KN', 'Grad', 'Tree' or 'Forest'

Output:

mask_good [1-D sequence of bool] 1 if the parameters are good, 0 if they are bad.

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex

PYTHON MODULE INDEX

f

INDEX

P

R

S

T