Sculptor

Release 1.1.0

Jan-Torge Schindler

Sep 11, 2023

GETTING STARTED:

Version: 1.1.0

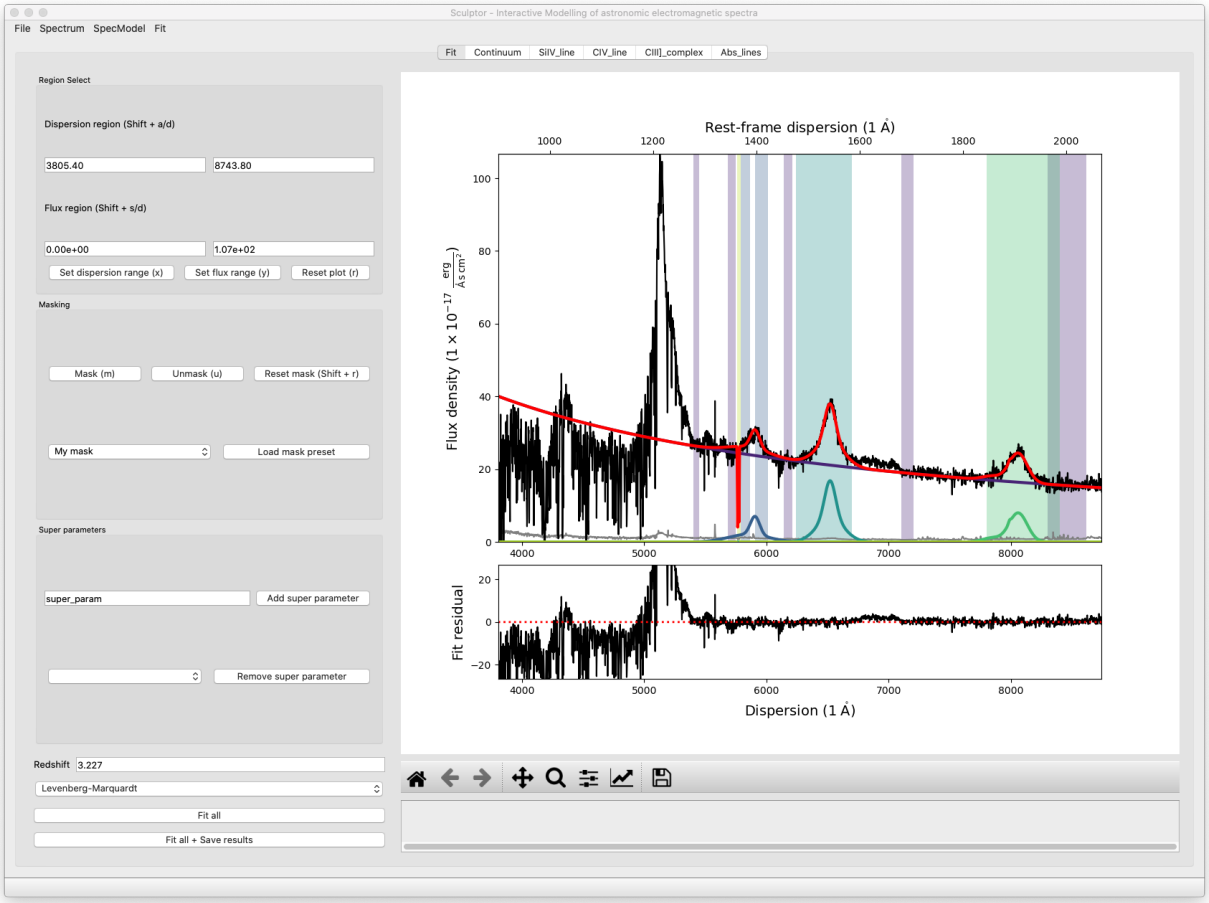
Sculptor is a high level API and Graphical User Interface around LMFIT tailored specifically for the analysis of astronomical spectra. This package is designed to facilitate reproducible scientific results and easy to inspect model fits in an open source framework. For this purpose the *Sculptor* package introduces four core python modules and a Graphical User Interface for interactive control:

1. **SpecOneD**: The *SpecOneD* module introduces the SpecOneD class, which is designed to store and manipulate 1D astronomical spectral data.

2. **SpecFit**: The core module of *Sculptor* introducing the *SpecFit* class, which holds complex models to fit to the 1D astronomical spectrum.

3. **SpecModel**: A helper class, which holds one complex spectral model, which can consist of multiple pre-defined or user-defined model functions to be fit to the 1D spectrum.

4. **SpecAnalysis**: A module focused on the analysis of continuum models of models of emission/absorption features. It interfaces with the SpecFit class and streamlines the process of analyzing the fitted spectral models.



At the heart of the *Sculptor* package is the Graphical User Interface, which offers interactive control to set up and combine multiple spectral models to fully fit the astronomical spectrum of choice. This includes masking of spectral features, defining fit regions, and setting of fit parameter boundaries. The framework allows to add interdependent fit parameters (e.g., to couple the FWHM of multiple emission/absorption lines).

If you are interested in being involved with this project, please contact Jan-Torge Schindler via github.

Disclaimer: Version 1.0.0 is the first stable release version of Sculptor. Be advised that all future 1.x.x versions will adhere to the same API. However API changes might occur between major releases.

CHAPTER

ONE

INSTALLATION

1.1 1. Clone the github repository

This document describes how to install Sculptor and its dependencies. For now the project has not been published on PyPi, yet. Therefore, the first step is to clone the Sculptor repository from github.

To do this simply clone the repository to your folder of choice.

```
git clone https://github.com/jtschindler/sculptor.git
```

The current version of Sculptor is designed to work with python >= 3.9.

1.2 2. Installing Sculptor and its requirements

Navigate to the main folder of sculptor. It should contain the *setup.py* file as well as *requirements.txt*, *conda_requirements.yml*, and *environment.yml*.

1.2.1 2.1 Installing Sculptor via conda (Recommended)

While some may prefer a pure pip installation, I have run into issues with installing PyQt5 and pytables via pip on new Mac OSX M1/M2 machines. Therefore, I recommend to use the provided environment.yml file instead.

It automatically creates the *sculptor* environment installing all necessary dependencies with the following command:

```
conda env create --file environment.yml
```

The environment is crearted using python 3.10. If a different python version is needed, please modify your version of the environment.yml file. Following the creating of the environment activate it via

```
conda activate sculptor
```

and then install Sculptor with

```
pip install -e .
```

This has been tested on a Mac OSX with a M2 chip, recently.

1.2.2 2.2 Installing Sculptor (using setup.py via pip)

One can attempt a pure pip installation. Due to issues with pytables and PyQt5 installations via pip on Mac OSX M1/M2 machines, one needs to first install these packages independently:

```
pip install PyQt5
pip install tables
```

Then you need to navigate into the main package folder with the setup.py file and execute:

```
pip install -e .
```

This will install all of the remaining dependencies and the Sculptor package itself.

1.2.3 2.3 Installing Sculptor (using requirements.txt via pip)

In the sculptor github repository you will find a 'requirements.txt', which allows you to install the necessary requirements using pip from the main sculptor directory:

```
pip install -r requirements.txt
```

If you are managing your python installation with Anaconda, this can work as well as long as you have pip installed in your Anaconda working environment. However, it may lead to issues if a pip version and an anaconda version of the same package (e.g., astropy) is installed.

In the same folder you then execute:

```
pip install -e .
```

This will install the Sculptor package.

1.3 3. Open up the sculptor GUI

To test whether the installation was successful, open your terminal and simply type

```
run_sculptor
```

If this opens up the Sculptor GUI, the installation was a success!

To test Sculptor further one can also load the example spectrum via

```
run_sculptor --ex=True
```

CHAPTER

TWO

THE SCULPTOR GUI

To carefully analyze small samples of astronomic spectra the Sculptor GUI offers an interactive way to put together complex spectral models. Therefore, we will start with an introduction to the GUI and its capabilities.

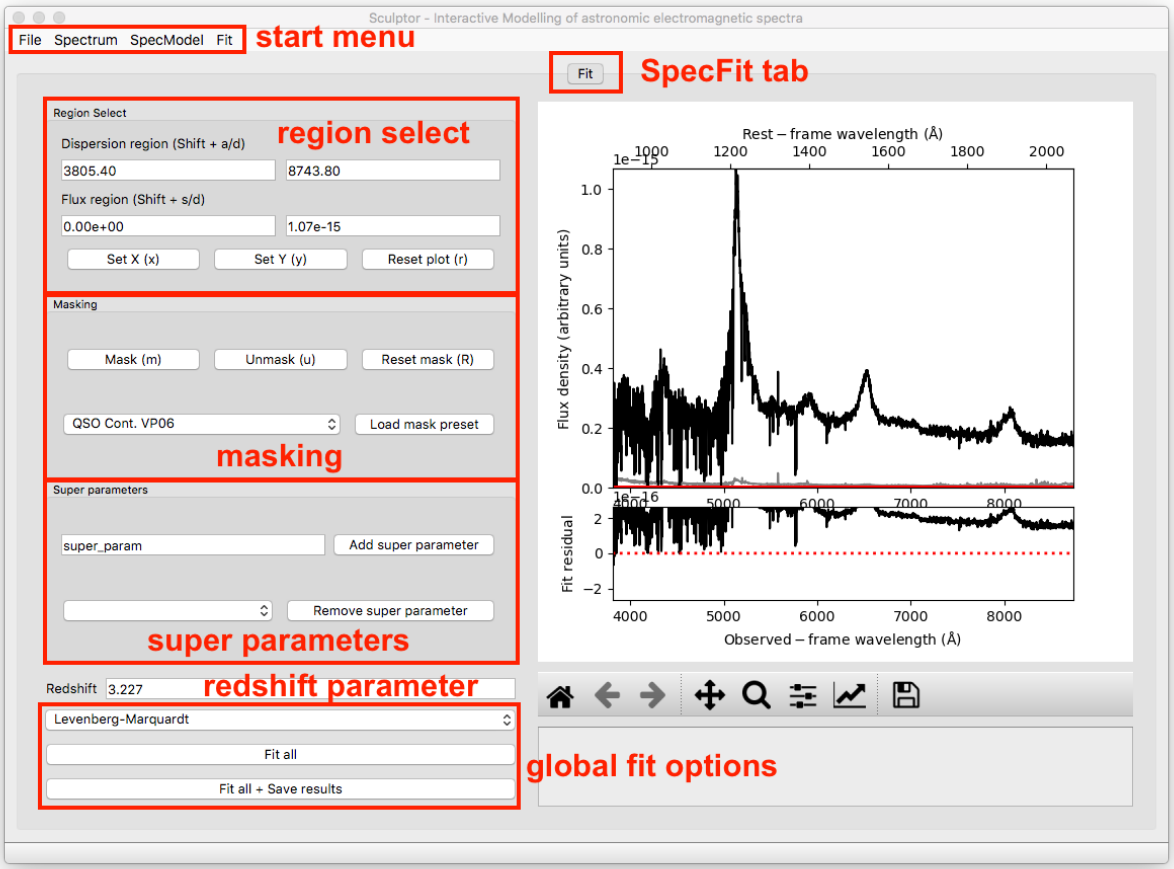
Disclaimer: The following examples are intended to get familiar with the Sculptor GUI. The fit to the quasar spectrum in this example is exemplary and should not be considered *science grade*. Due to the rapid development of Sculptor, your current version of the GUI, might look slightly different.

A full model fit to an astronomical spectrum is internally set up as a *SpecFit* python object. This object holds general information on the spectrum (dispersion, flux density, redshift, etc.) and also defines the optimization method used when carrying out the fit (e.g., Levenberg-Marquardt). Spectral models, *SpecModel* python objects, are then added to the SpecFit object. Each SpecModel object can hold various fit models (e.g., multiple Gaussian emission line models), which are fit simultaneously to the spectrum. The SpecModel objects are *ordered* and the model fit of the first SpecModel is subtracted before the second SpecModel is fit and so on. This information is central to the way that Sculptor works and will help to understand the GUI.

Start by opening up the Sculptor example:

```
run_sculptor --example=True
```

This will open the main GUI and read in an SDSS spectrum of quasar J030341.04-002321.8 at redshift z=3.227. It should look something like this:



2.1 1-The Start Menu

On top of the window you will find the start menu with the *File*, *Spectrum*, *SpecModel*, and *Fit* drop down menus.

2.1.1 File

The *File* dropdown menu allows to *Load* and *Save* the full spectral fit (SpecFit object) and will open a File Dialog Window to select the folder to save to or to load from. It also allows to *Exit* the GUI. Keyboard shortcuts are indicated.

2.1.2 Spectrum

The *Spectrum* dropdown menu offers various ways to import a spectrum, overwriting the current spectrum and removing all masks.

2.1.3 SpecModel

The fitting is done via the *SpecModel* object and without any *SpecModels* nothing can be fit. The *SpecModel* dropdown menu offers to *Add* a *SpecModel*, *Remove* the current (active tab) *SpecModel* or *Remove all SpecModels* altogether.

2.1.4 Fit

The *Fit* dropdown menu allows to specify further fit parameters when fitting the SpecModel with MCMC (*Set MCMC parameters*). It also offers the possibility to resample the spectrum on a pixel by pixel basis using its flux density uncertainties and fitting each resampled spectrum with the specified fit method (*Run resample and fit*). We will discuss this option at a later point in detail. One can specify a few settings for this option via the *Set resample and fit parameters* menu item.

2.2 2-The SpecFit Tab

The SpecFit tab, called *Fit*, provides an overview over all fit SpecModels and the full spectral fit (sum of all individual SpecModels) in the figure to the right. We will now go through the different regions to the left of the figure.

2.2.1 Region Select

The *Region Select* box allows to specify ranges in the dispersion of flux direction by direct input or interactively by pressing the *Shift* button together with *W*, *S*, *A*, or *D* while hovering with the cursor over the figure to the right. By doing so the values shown in the white fields are automatically updated. If the cursor is outside the figure no input will be passed.

The first two buttons below, which can also be accessed via keyboard shortcuts, allow to set the dispersion and flux density ranges of the figure to the right using the regions defined above. The last button *Reset plot* resets the plot ranges to show the full spectrum.

2.2.2 Masking

The masking box provides capabilities for interactive masking of the spectrum. **In the SpecFit tab masking removes regions from the fit (greyed out visually).** The *Mask* button masks the dispersion range defined in the *Region Select*. The *Unmask* button unmasks the dispersion region defined in the *Region Select* and the *Reset Mask* button resets the mask unmasking the entire dispersion range.

Furthermore one can select a pre-defined mask in the drop-down menu and then mask out the pre-defined dispersion ranges via the *Load mask preset* button.

2.2.3 Super Parameters

The *Super parameters* box allows to *Add* and *Remove* super parameters, which are defined on the highest level and then added to all SpecModels and individual models insight the SpecModel. Super parameters are defined here. If they are fitted by a SpecModel the values are adjusted globally and all future fits will now start with the updated values.

A use case for a super parameters could be the radial velocity of a star, for example. One can imagine that the first SpecModel fits the radial velocity via an absorption line shift relative to vacuum wavelength. All subsequent models use this velocity shift as an input value to analyze further spectral features.

Only special cases require the use of super parameters and because Sculptor fits SpecModels subsequently (one after another) care has to be taken, when using this advanced capability.

2.2.4 Redshift Parameter

Extragalactic sources (e.g., galaxies, quasars, etc.) will be cosmologically redshifted. The SpecFit object has a redshift attribute, which can be set/updated here. The user can enter the value and set/update the internal value by hitting *Enter*.

The *redshift parameter* can be passed to spectral models as a keyword argument (kwarg), when they are added to a SpecModel object. This allows to build in set the redshift parameter when building a new model. Contrary to *super parameters* the global redshift value will **not** be updated when fit by any model. However, the *redshift parameter* sets the rest-frame axis on top of the figure to the right.

2.2.5 Global Fit Options

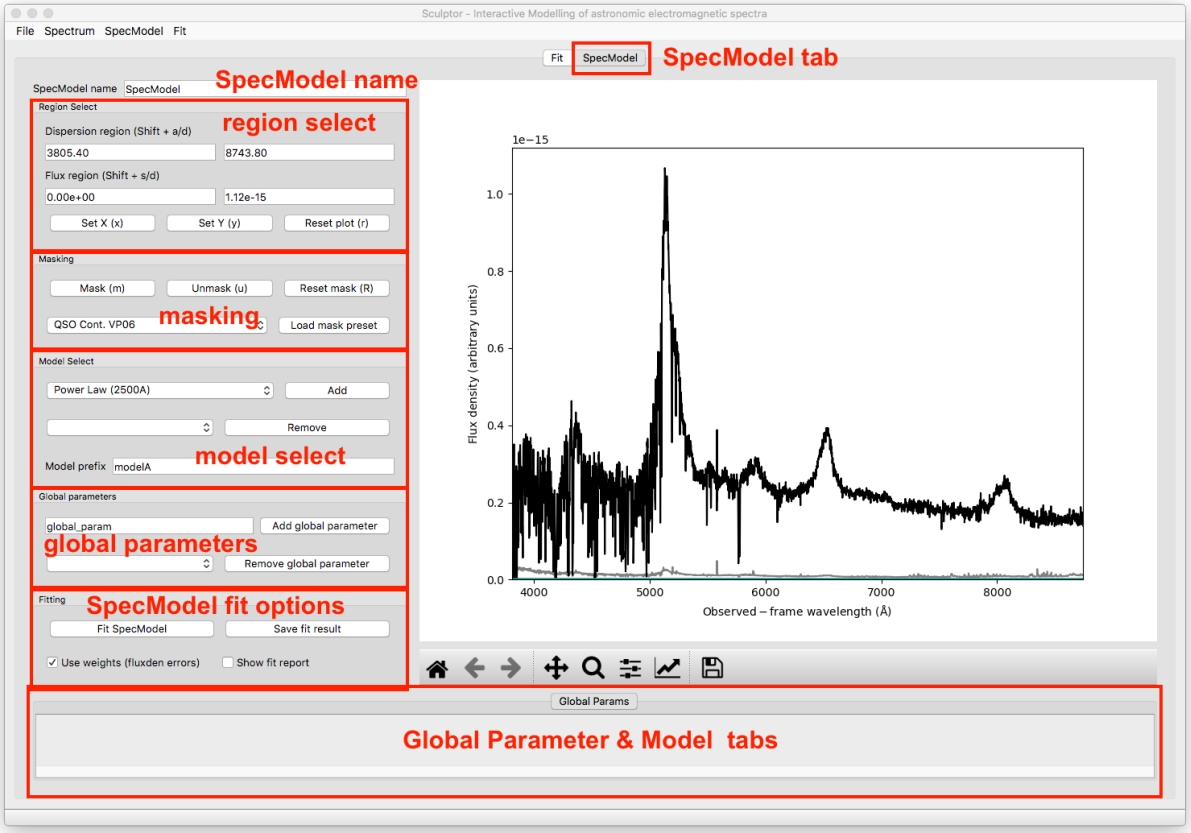
To the bottom left of the *SpecFit* tab is a drop-down menu and two buttons. The drop-down menu allows you to select the fit algorithm available in LMFIT. As a default the Levenberg-Marquardt algorithm is selected. For more on the differences of the minimizers see the LMFIT documentation (The minimze function).

One special option is the Maximum likelihood fit via Monte-Carlo Markov Chain, which uses emcee. Additional options for the MCMC runs are available under the *Fit* start menu item.

The *Fit all* button consecutively fits all SpecModels, whereas the *Fit all + Save results* button saves the fit results to a folder, which is selected by the user in a File Dialog. The results contain a png image of the figure shown in the SpecFit tab as well as a LMFIT fit report with the best fit values and covariances for each SpecModel saved in a ".txt" file.

2.3 3-The SpecModel Tab

As a next step we click on *SpecModel* from the *Start Menu* and click on *Add SpecModel*. This will add a new SpecModel tab to the GUI and automatically switches to it. The figure now displays the spectrum without the residual plot below and would show only model fluxes and masks related to the active SpecModel.



2.3.1 SpecModel Name

This input field allows you to change the name of the SpecModel from the default value "SpecModel". To apply the name change hit *Enter*. The name change is successful, when you see the name of the active tab change to your input.

2.3.2 Region Select

The region select controls work exactly in the same way as for the SpecFit tab. However, all changes to the flux and dispersion range are, of course, only applied to the SpecModel figure to the right.

2.3.3 Masking

The masking controls work in the same way as before with one important difference: mask regions now *mask in* ranges that should be considered in the SpecModel fit, whereas in the SpecFit tab masking excluded dispersion regions from all fits. The masked-in dispersion ranges are highlighted in color.

Custom user-defined masks can be added with new python modules as part of the **sculptor-extensions** package, included in the github repository. An example file *my_extension.py* adds the *QSO Cont. VP06* mask to Sculptor, which defines pure continuum regions for quasar modeling.

2.3.4 Model Select

The model select controls allow you to *Add* and *Remove* models selected by their name from the drop down menus. Before a model is added the model prefix (default: "modelA") can be specified for better readability of the results later on. For example, if someone wanted to the the Hydrogen Balmer line Hbeta, it would be appropriate to call the prefix "Hbeta". **Model prefixes cannot contain spaces.**

The models that can be added to the spectrum include a range of basic models (e.g., gaussian, power law, constant, etc.) included with Sculptor. Custom models can be defined by the user in new python modules as part of the **sculptor_extensions** package, included in the github repository. An example of such an extension is provided with the *my_extension.py* module.

2.3.5 Global Parameters

Similar to *Super parameters*, which are added to all models in *all* SpecModels, the *Global parameters* are added to all model functions in the *active* SpecModel. The controls allow to provide a custom name for a global parameter, *Add* the global parameter to the SpecModel or select an existing global parameter from the drop down menu and then *Remove* it.

Whereas the use cases for *Super parameters* are probably rare, use cases for *Global parameters* are much more common. For example, if we want to model a few emission lines, which we know should have the same width. We can easily define a new global parameter *FWHM_common* and set it to be the FWHM for all Gaussian emission line models in the SpecModel.

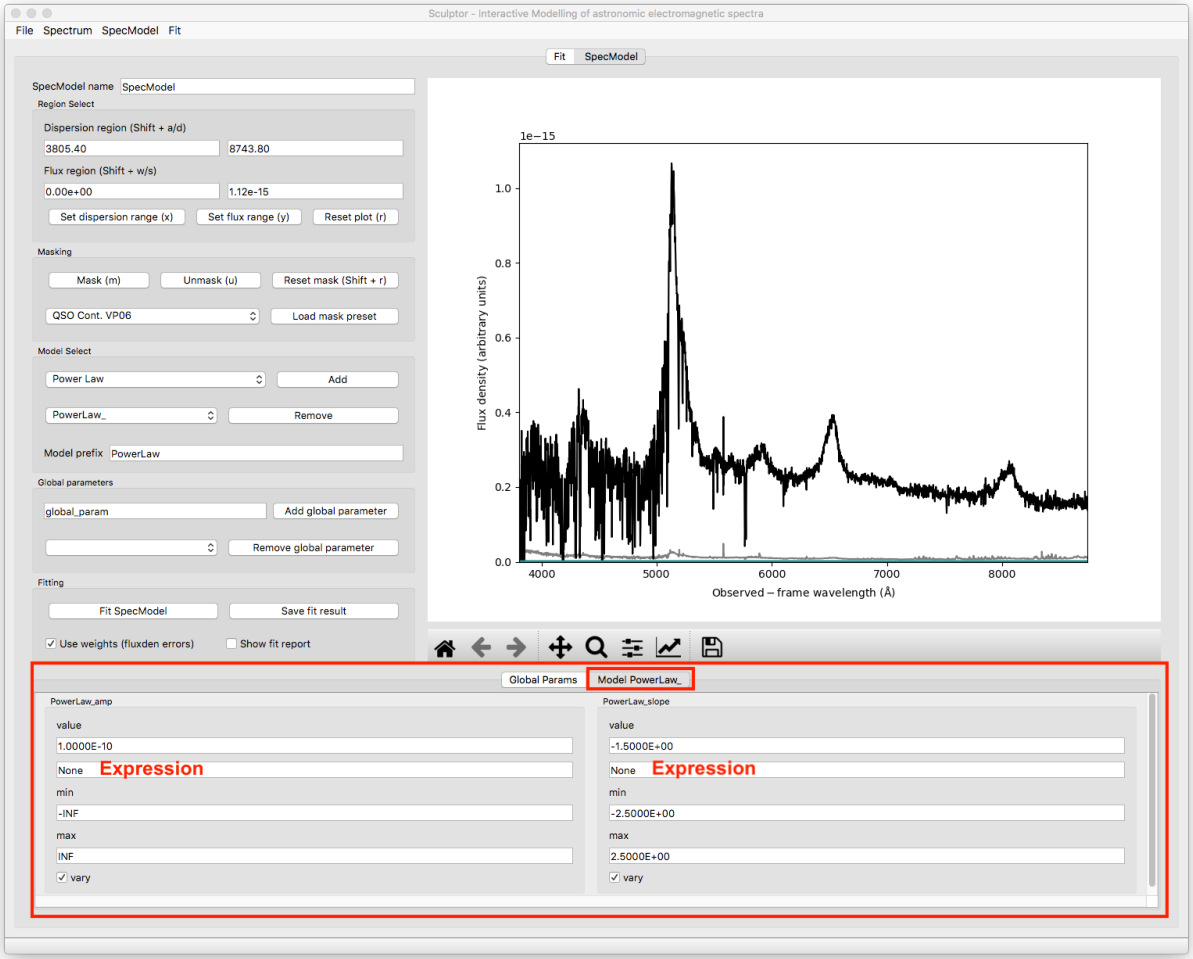
2.3.6 Fitting

The fitting controls consist of two button and two toggle switches. The switch *Use weights (fluxden errors)* is by default enabled and uses the flux density uncertainty as weights in the fit. The second switch, *Show fit report* prints the LMFIT fit report in a pop-up window on the screen. This provides the user with metrics for the goodness of the fit, the best fit values and the fit covariances. If the MCMC method is chosen in the SpecFit tab, the fit report also plots the posterior distributions of all fit parameters. If the number of model parameters is large, the corner plot will be unfortunately hard to read.

The *Fit SpecModel* button fits the active SpecModel and the *Save fit result* button saves the fit results (fit report and figure png) for the active SpecModel.

2.3.7 Global Parameter & Model Tabs

At the bottom of the SpecModel Tab is a field, which shows an empty tab called *Global Params*. All added global parameters will be shown in this tab. To further understand the function of this field let's add a power law model to the SpecModel.



Adding the model added another tab to the field appropriately named after the chosen model prefix (PowerLaw). Navigating to the tab shows all the parameters for this specific model. In the case chosen here the model has two free parameters, the amplitude (PowerLaw_amp) and the slope (PowerLaw_slope). For each parameter the user can now control the initial *Value*, the parameter range set by *min* and *max* and whether the parameter should be varied during the fit (*vary* toggle switch).

The field that currently shows the text *None* is the expression field. It allows to use mathematical constraints on the parameters. The LMFIT documentation for this topic is found here.

In the global parameter example above we described that we want to set the FWHM of multiple gaussian emission lines to our super parameter *FWHM_common*. This done via the expression field by entering the name of the global parameter and applying the change with *Enter*.

The expressions can only contain numbers, names of other parameters in the model and +, -, /, *, (,). If the expression is invalid the input will not be forwarded to the model and the text field will reset to previous expression after fitting the spectrum.

CHAPTER

THREE

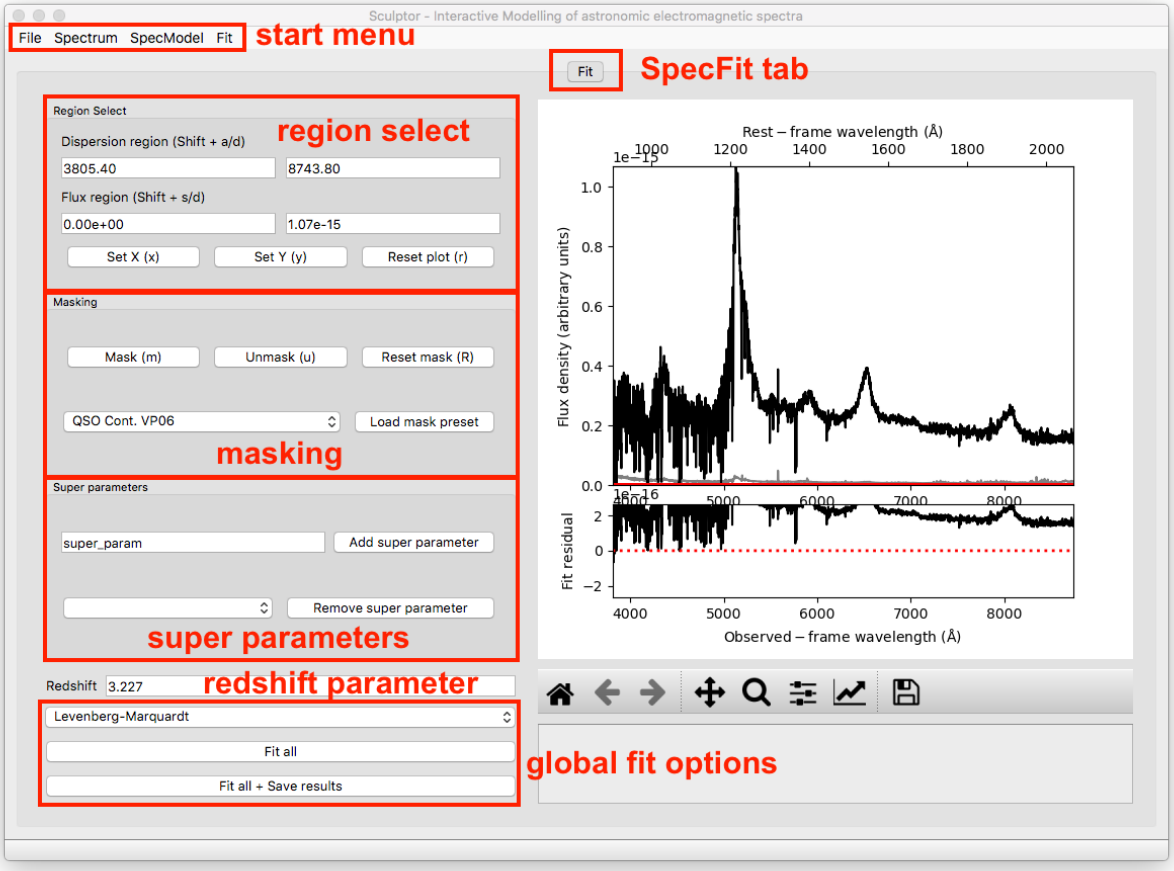
SPECTRAL FITTING WITH THE SCULPTOR GUI

In this example we will fit the SDSS spectrum of quasar J030341.04-002321.8 at redshift z=3.227 step by step. The example is aimed at first time users to provide insight into the Sculptor workflow and is designed to present a starting point.

We will begin by starting up sculptor with the example spectrum already imported:

```
run_sculptor --example=True
```

The GUI will start with the SpecFit tab open displaying the quasar spectrum.



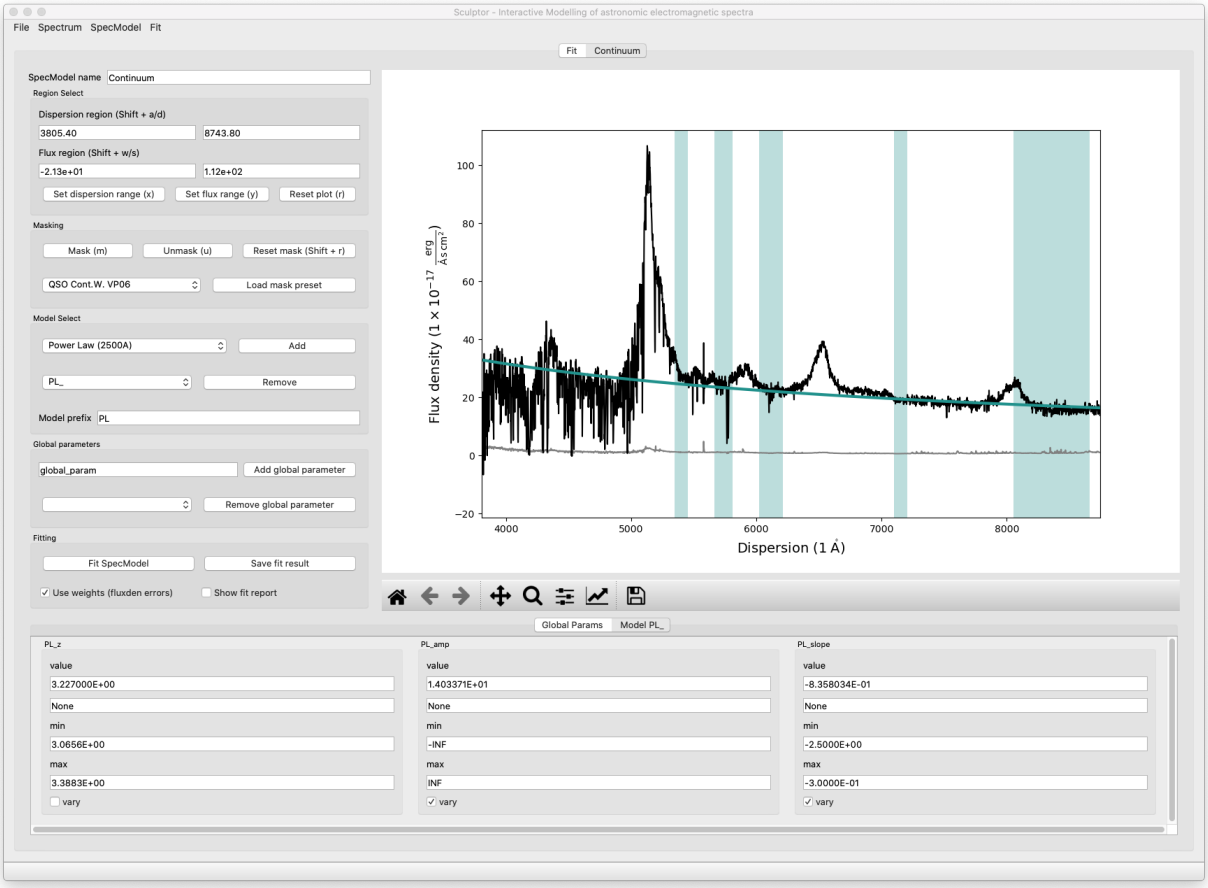
3.1 1-The quasar continuum model

In this example we will be working with the Sculptor basic models and the models defined in the Sculptor extension *qso.py*, which were specifically included for this example.

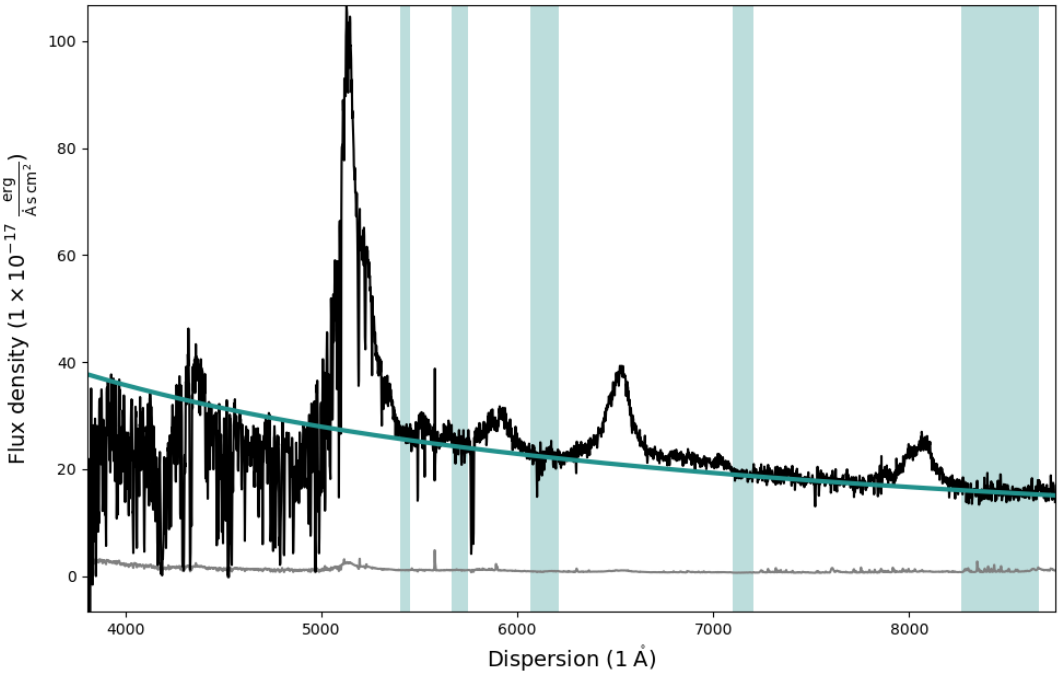
3.1.1 Steps

- Start by adding a *SpecModel* and naming it *Continuum*.
- Select the mask *QSO Cont.W. VP06*, which refers to quasar continuum regions according to the paper by Vestergaard & Peterson 2006.
- Add the *Power Law (2500A)* model with the prefix *PL*
- Click on *Fit SpecModel* for a first fit.

At this point the GUI showing *SpecModel* tab named *Continuum* should look like this:



Now we can interactively adjust the masked-in fit regions to exclude absorption and emission features and refit the spectrum again. Our final fit looks slightly better now:



3.2 2-Modeling the SiIV emission line

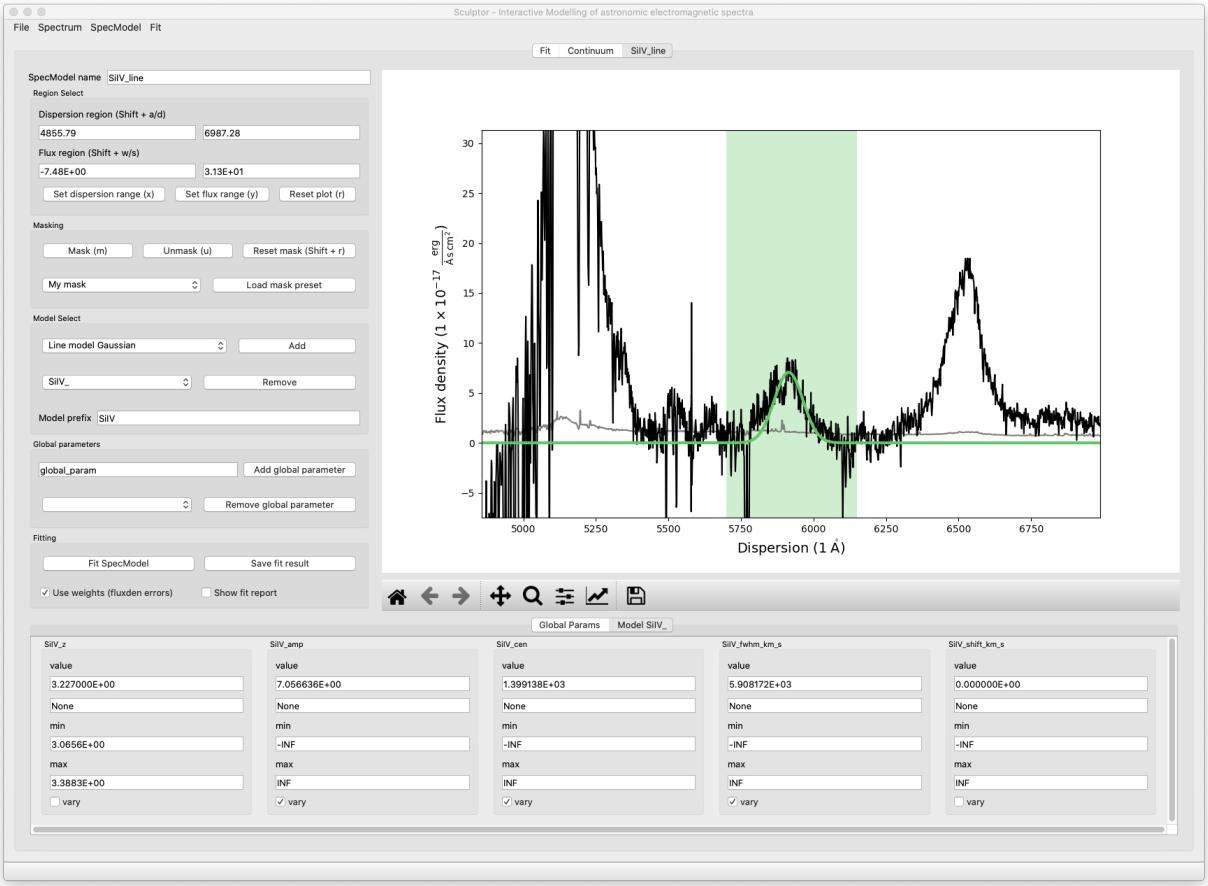
We will now add a model for the SiIV emission line at ~5900A observed frame with one Gaussian.

3.2.1 Steps

- Start by adding a another *SpecModel* and naming it *SiIV_line*.
- Manually enter 5700 / 6150 into the dispersion region windows and apply with *Enter*.
- Click *m* to mask the specified dispersion region.
- Add the *Line model Gaussian* model with the prefix *SiIV*.
- Enter 1399.8 into the *SiIV_cen* value field and apply the change with *Enter*.
- Click on *Fit SpecModel* to fit the line.

Note that the redshift (*SiIV_z*) and the velocity shift (*SiIV_vshift*) parameters have set default values used in the fit, but will not be fit themselves by default (*vary* checkmark is not enabled). You could choose to vary these parameter instead. However, if more than one of the three parameters (*SiIV_z*, *SiIV_cen*, *SiIV_vshift*) is set to *vary*, this will cause problems with the fit as they are degenerate.

The final SiIV fit will look something like this:



3.3 3-Modeling the CIV emission line

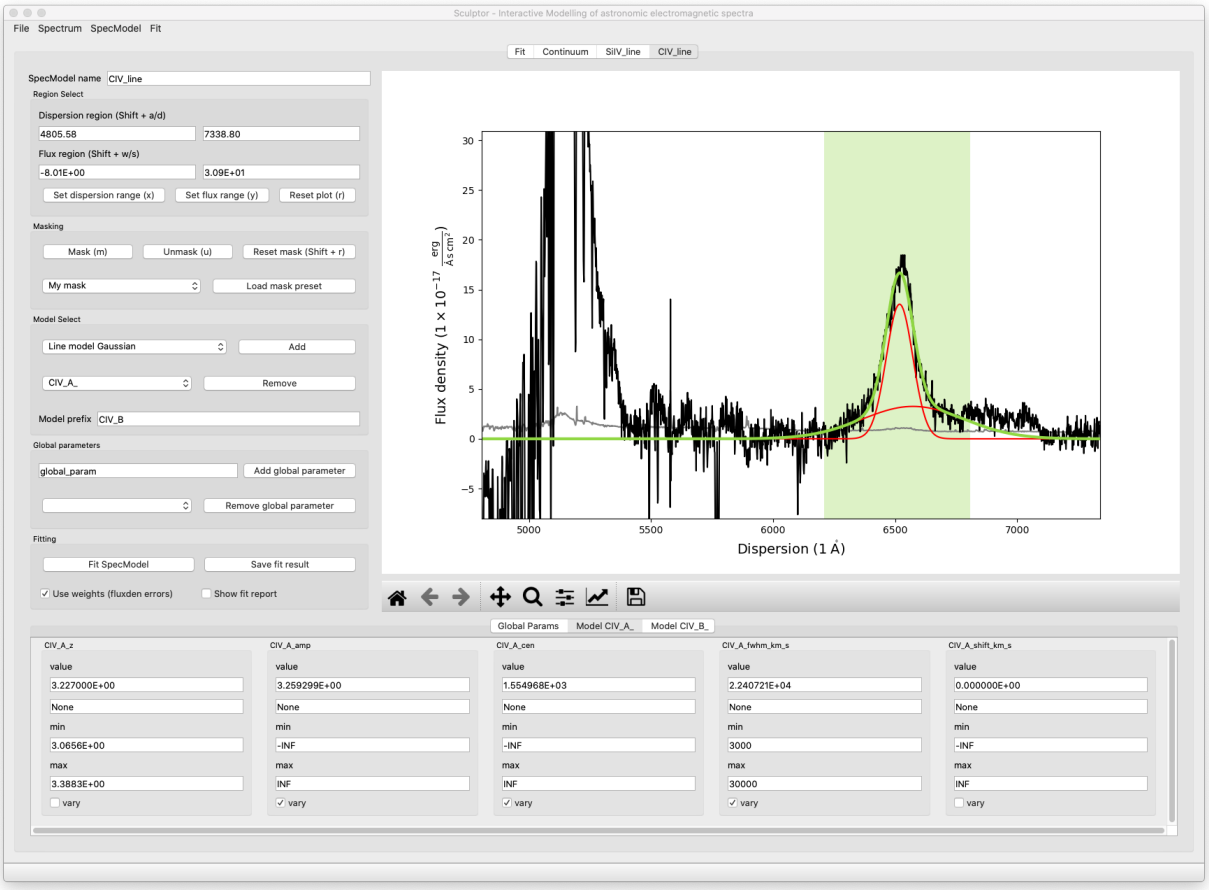
We have now successfully modeled the SiIV line. Let us do the same for the CIV line, but this time we will approximate it using 2 Gaussian models.

3.3.1 Steps

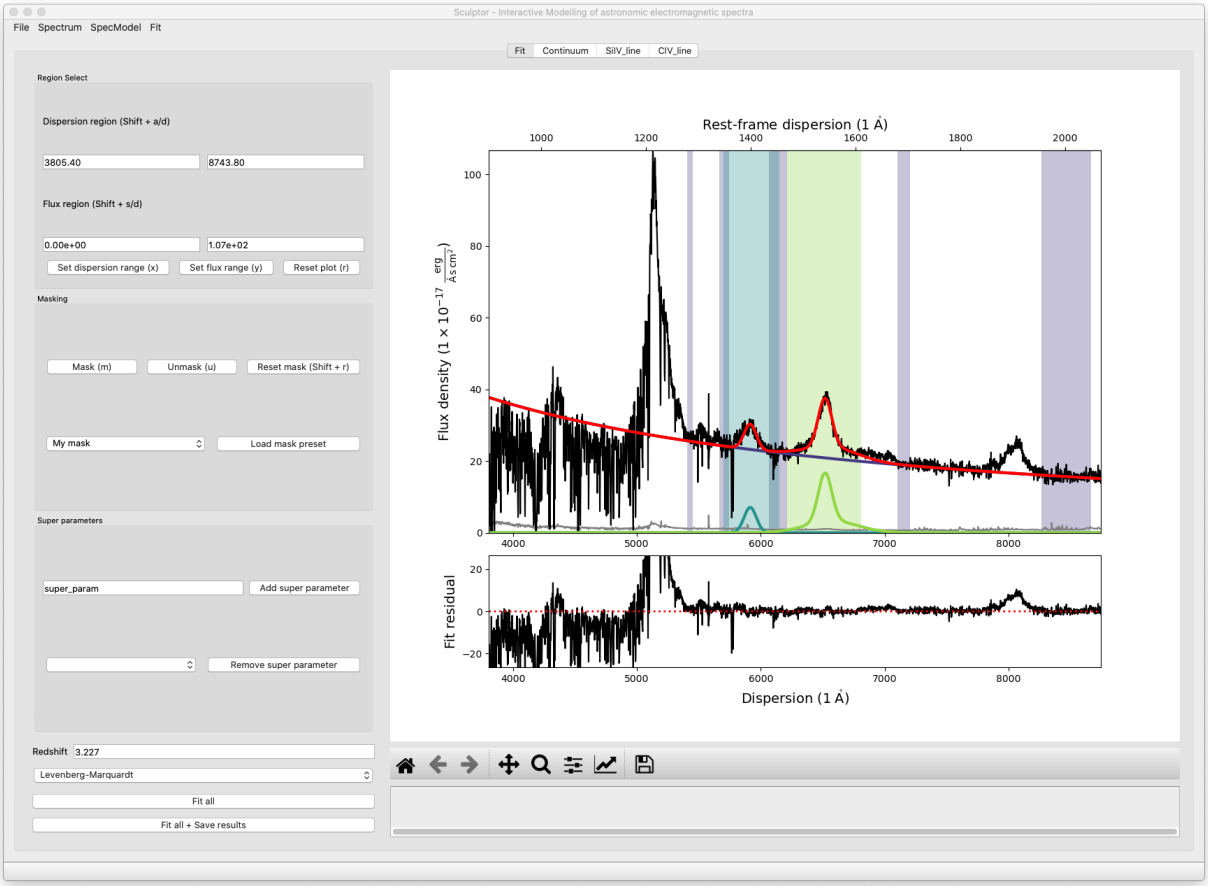
- Start by adding a another *SpecModel* and naming it *CIV_line*.
- Manually enter 6210 / 6810 into the dispersion region windows and apply with *Enter*.
- Click *m* to mask the specified dispersion region.
- Add the *Line model Gaussian* model with the prefix *CIV_A* (component A).
- Add the *Line model Gaussian* model with the prefix *CIV_B* (component B).
- Enter 1549.06 into the *CIV_cen* value field for both models and apply the changes with *Enter*.
- Navigate into the *CIV_A* model tab and restrict the *CIV_A_FWHM* values in the range of 2000 to 10000, then hit *Enter* to apply.
- Navigate into the *CIV_B* model tab and restrict the *CIV_B_FWHM* values in the range of 300 to 3000, then hit *Enter* to apply.

- Click on *Fit SpecModel* to fit the line.
- **Check the fit results, specifically the FWHM values. You will find that the fit reached the maximum value you specified. This means that you should probably relax the upper FWHM boundary.Choose *CIV_A_FWHM* max=30000, and *CIV_B_FWHM max=8000* for now an click on *Fit SpecModel* again. The resulting fit should be a better approximation of the line.**

The final fit of the CIV SpecModel will look similar to this:



You can also navigate to the *SpecFit* tab (*"Fit"*) and look at the total fit to the quasar spectrum:



In the lower panel of the figure in the SpecFit tab you will also see the fit residual after all your models have been subtracted.

3.4 4-Saving and loading model fits

If you are happy with your first fit, you can save the model and the fit results.

3.4.1 Steps

- In the start menu click *File*->*Save*. This will open an file dialog.
- Create a new directory (e.g., you can name it *myfirstfit*)
- Save your model by clicking *Open* in the file dialog bottom right corner.
- Following this example Scuptor created four model files (*[SpecModel Index]_[prefix]*_model.json *) for the *PL_*, the *SiIV_*, the *CIV_A_*, and the *CIV_B_* model as well as three result files (*[SpecModel Index]_fitresult.json*). The SpecModel Index in our example runs from 0-2 over the "Continuum", "SiIV line", "CIV line" SpecModels. The *fit.hdf5* file holds further information important for the SpecFit class and the SpecModel classes, including the spectrum itself.

You can now try to exit Sculptor and then open the Sculptor again:

```
run_sculptor
```

This should bring up an empty Sculptor GUI. To **load** your previously saved model click *File*->*Load* and select the folder, where you saved the model.

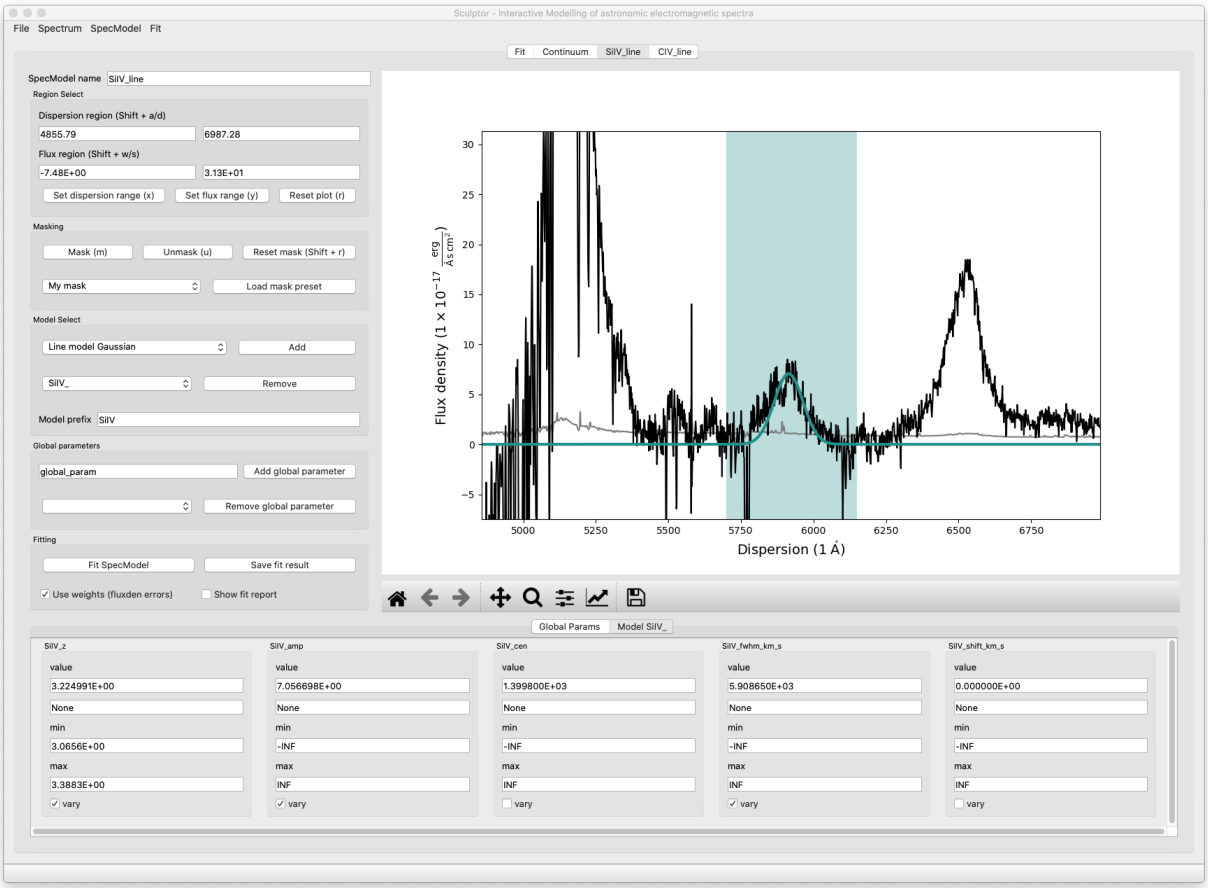
3.5 5-Fitting the SiIV line redshift

Previously we have allowed the central wavelength of the SiIV line to be the varying parameter. To calculate the redshift of the SiIV line we can calculate the offset between the fitted central wavelength and 1399.8 A. However, with the Gaussian model we have used, we can directly fit the redshift parameter.

3.5.1 Steps

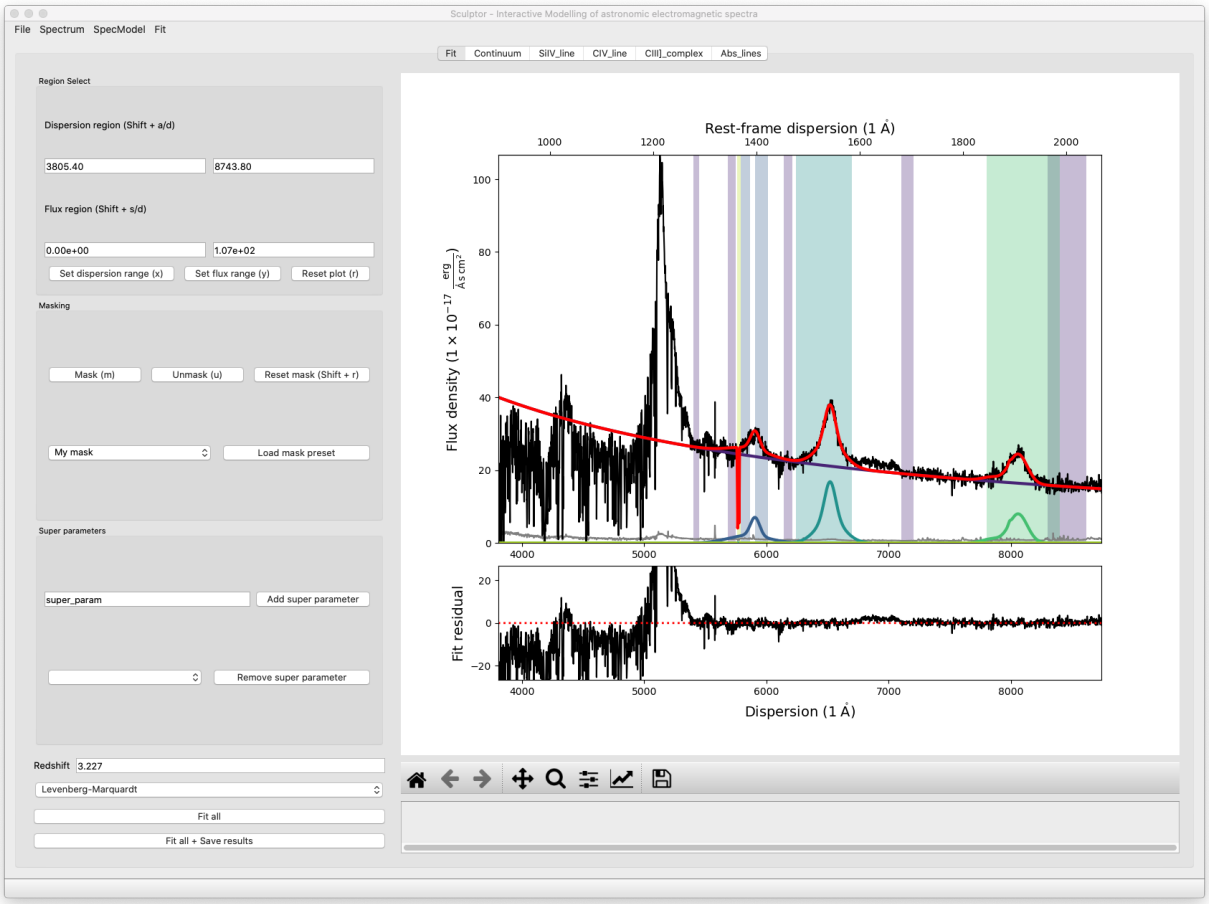
- Navigate to the *SiIV line* SpecModel tab and to the *Model SiIV_* parameters at the bottom.
- Set the value for *SiIV_cen* to 1399.8 and disable *vary* for the parameter.
- Then enable vary for the *SiIV_z* parameter.
- Click on *Fit SpecModel* to fit the line.

Now the redshift parameter was fit, while the central wavelength was considered constant:



3.6 6-A full fit of the example spectrum

We provide a full fit of the example spectrum in the *sculptor/examples* directory with the name *example_spectrum_fit*. We invite you to load this fit into Sculptor and explore the use of the *Global parameters* as well as the fitting of absorption lines.



CHAPTER

FOUR

AN INTRODUCTION TO THE SPECONED CLASS

In this notebook we will demonstrate the functionality of the SpecOneD class included in the speconed module as part of *Sculptor*. SpecOneD objects are designed to hold the spectral information of an astronomical source along with ancillary information, such as the data header generated by the observatory or the reduction software or the physical units of the dispersion and flux density axis.

4.1 Introducing the SpecOneD object

We begin by introducing the SpecOneD object and its attributes. As a first step we import the SpecOneD module from sculptor and generate an emptye SpecOneD object.

[1]:
```
# Import the speconed module from sculptor
from sculptor import speconed as sod

# Instantiate an empty SpecOneD object
spec = sod.SpecOneD()
```

We have now instantiated an empty SpecOneD object called *spec*. Let's take a quick look at the class attributes by iterating over them.

[2]:
```
for key in vars(spec).keys():
    print(f'{key}')
```

```
fluxden
fluxden_err
dispersion
fluxden_ivar
mask
dispersion_unit
fluxden_unit
header
fits_header
obj_model
telluric
```

Every SpecOneD object will be instantiated with a range of numpy.ndarrays holding information on the dispersion axis (dispersion), the flux density (fluxden), the flux density error (fluxden_err), a mask (mask). All of these arrays have to have the same length.

In addition, the two attributes "dispersion_unit" and "fluxden_unit" contain the information on the physical units of the dispersion axis as well as the flux density (and flux density error). These attributes are populated with astropy.units.Quantity (astropy.units.Unit, astropy.units.CompositeUnit, astropy.units.IrreducibleUnit) enabling to use the

astropy.units module for unit conversion. This is also an important aspect for the analysis of the spectral models later on.

The "header" attribute is a pandas.DataFrame with additional information of the spectrum. If the header was populated through a fits file the original fits header is available at the "fits_header" attribute. Note that when initializing a SpecOneD object you should pass your fits header under the *header* keyword.

Spectra reduced by *PypeIt* and read in from the *PypeIt* fits format might carry additional information about the object_model and the telluric model from the telluric correction, which will be saved in the "obj_model" and "telluric" attributes.

4.2 Manual initialization of a SpecOneD object

While the SpecOneD class offers functionality to import spectra from iraf or PypeIt standard data reductions and also supports the SDSS format, it is important to know how to initialize spectra manually for custom use. For this example we use the SDSS spectrum of the quasar J030341.04-002321.8 available in the sculptor data folder.

[3]:
```
# Let's begin by importing astropy fits and units functionality
from astropy.io import fits
import astropy.units as units

# Define the name of the example spectrum
spec_filename = '../../sculptor/data/example_spectra/J030341.04-002321.8_0.fits'

# Read in the example spectrum with astropy
hdu = fits.open(spec_filename)

# Extract the dispersion axis, the flux density and the error from the fits file
dispersion = hdu[1].data['loglam']
flux_density = hdu[1].data['flux']
flux_density_ivar = hdu[1].data['ivar']

# Extract the fits header of the spectrum
header = hdu[0].header

# Before we initialize the SpecOneD object we need to convert the dispersion axis to a␣
↪linear scale
dispersion = 10**dispersion

# The header provides information on the units of the spectrum (Let's print it out below!)
print('Wavelength unit: {}'.format(header['WAT1_001']))
print('Flux density unit: {}'.format(header['BUNIT']))

dispersion_unit = 1.*units.AA
fluxden_unit = 1e-17 * units.erg/units.s/units.cm**2/units.AA

# Now we can initialize a SpecOneD object manually
spec = sod.SpecOneD(dispersion=dispersion, fluxden=flux_density, fluxden_ivar=flux_
↪density_ivar,
                    fluxden_unit=fluxden_unit, dispersion_unit=dispersion_unit,␣
↪header=header)
```

```
Wavelength unit: wtype=linear label=Wavelength units=Angstroms
Flux density unit: 1E-17 erg/cm^2/s/Ang
```

We have done it! We have manually initialized a SpecOneD object

Let us now take a closer look if everything worked. We begin by checking the types and shapes of the main SpecOneD attributes: *flux density*, *dispersion*, *mask*, *flux density error*, the units and the header

[4]:
```
print('Flux density -  type {}, size {} \n'.format(type(spec.fluxden), spec.fluxden.
↪shape))
print('Dispersion axis -  type {}, size {} \n'.format(type(spec.dispersion), spec.
↪dispersion.shape))
print('Flux density error -  type {}, size {} \n'.format(type(spec.fluxden_err), spec.
↪fluxden_err.shape))
print('Mask -  type {}, size {} \n'.format(type(spec.mask), spec.mask.shape))

print('Dispersion unit -  value {} and type {} \n'.format(spec.dispersion_unit,␣
↪type(spec.dispersion_unit)))
print('Flux density unit - value {} and type {} \n'.format(spec.fluxden_unit, type(spec.
↪fluxden_unit)))

print('Header DataFrame \n', spec.header)
```

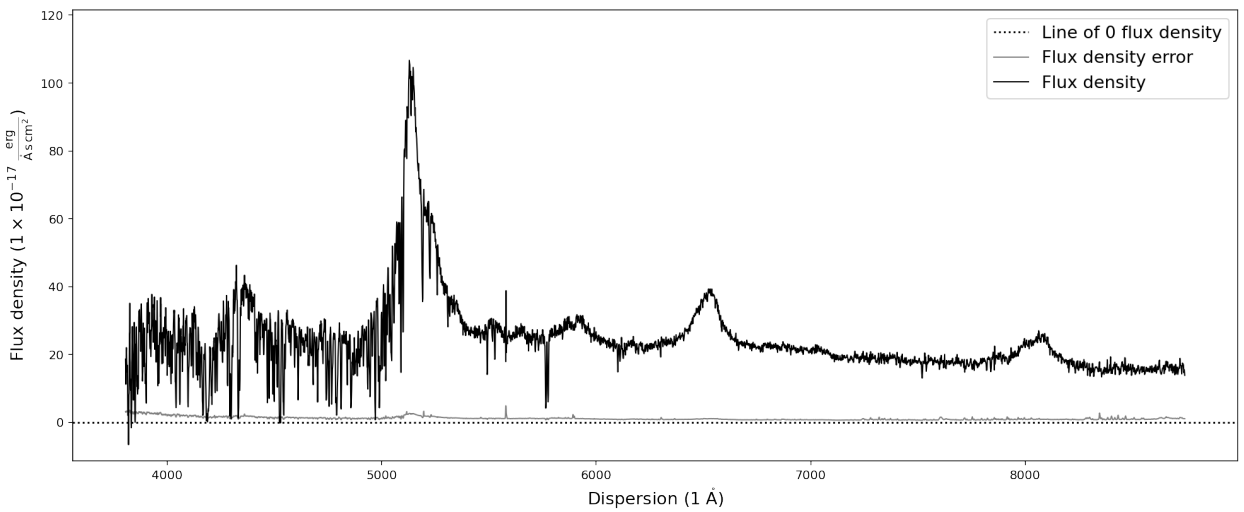
```
Flux density -  type <class 'numpy.ndarray'>, size (3614,)

Dispersion axis -  type <class 'numpy.ndarray'>, size (3614,)

Flux density error -  type <class 'numpy.ndarray'>, size (3614,)

Mask -  type <class 'numpy.ndarray'>, size (3614,)

Dispersion unit -  value 1.0 Angstrom and type <class 'astropy.units.quantity.Quantity'>

Flux density unit - value 1e-17 erg / (Angstrom cm2 s) and type <class 'astropy.units.
↪quantity.Quantity'>

Header DataFrame
                                    value
SIMPLE                              True
BITPIX                                 8
NAXIS                                  0
EXTEND                              True
TAI                       4477028053.74
...                                   ...
EXPID02  b1-00006799-00006802-00006803
EXPID03  b1-00006800-00006802-00006803
EXPID04  r1-00006798-00006802-00006803
EXPID05  r1-00006799-00006802-00006803
EXPID06  r1-00006800-00006802-00006803

[138 rows x 1 columns]
```

Everything seems to look reasonable here. However, a visual inspection of the spectrum will be necessary to convince us that it was read in correctly. For this purpose the SpecOneD class offers a simple plot functionality:

4.3 Using .plot() for quick visualization of the spectrum

The spectrum can be plotted with the built-in plot functionality, which uses matplotlib. Note that the units in the axis labels will be automatically populated by the units from the SpecOneD object (spec.dispersion_unit, spec.fluxden_unit).

[5]:
```
spec.plot()
```



We expect the source to be a quasar at redshift z=3.227. Therefore we expect the Lyman-alpha emission line around 5140A. A look at the plot convinces us that the spectrum was read in correctly, including the axis units.

4.4 Saving and reading in SpecOneD objects

Sculptor and the Sculptor GUI require a SpecOneD object as input. The GUI allow to import spectra from a few common fits formats or from a saved SpecOneD object.

SpecOneD allows to save spectra to the *hdf* format, which is the native way to save spectra within the SpecOneD class. The spectral data, meta data and the header will be saved in a file, which can be easily imported to the Sculptor GUI.

[6]:
```
# Saving the SpecOneD spectrum in its native format
spec.save_to_hdf('temp_spectrum.hdf5')
```

```
/opt/anaconda3/envs/sculptor-env/lib/python3.9/site-packages/pandas/core/generic.py:2606:
↪ PerformanceWarning:
your performance may suffer as PyTables will pickle object types that it cannot
map directly to c-types [inferred_type->mixed-integer,key->block0_values] [items->Index([
↪'value'], dtype='object')]

  pytables.to_hdf(
```

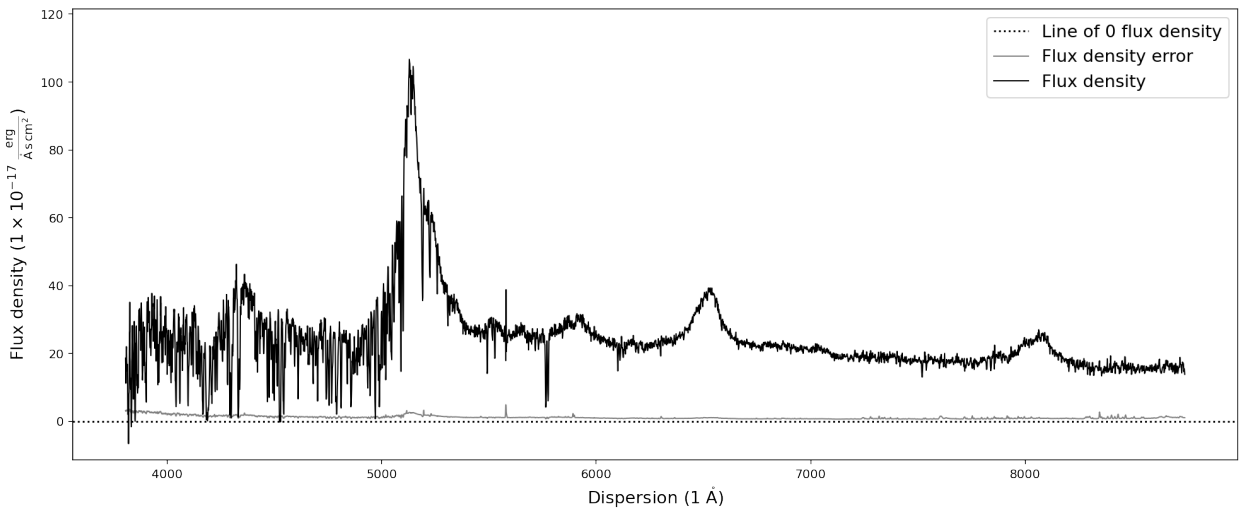
In a similar fashion we can read in the spectrum again:

[7]:
```
# Before we read in a SpecOneD object, we need to initialize an empty object
new_spec = sod.SpecOneD()
# Then we use the read_from_hdf function and the filename
new_spec.read_from_hdf('temp_spectrum.hdf5')
```

(continues on next page)

(continued from previous page)

```
# As a test we plot the spectrum
new_spec.plot()
# and test if the header was saved and read in correctly
print('Flux density unit according to header is {}'.format(new_spec.header.loc['BUNIT',
↪'value']))

# However, the fits header attribute is not populated
print('The fits header attribute is: {}'.format(new_spec.fits_header))

# To keep the notebook directoy clean we delete the temporary spectrum file
! rm temp_spectrum.hdf5
```




```
Flux density unit according to header is 1E-17 erg/cm^2/s/Ang
The fits header attribute is: None
```

Great! The spectrum was saved and read in correctly.

Note: Because we did not initialize the new spectrum from a fits file the *fits_header* attribute is *None*.

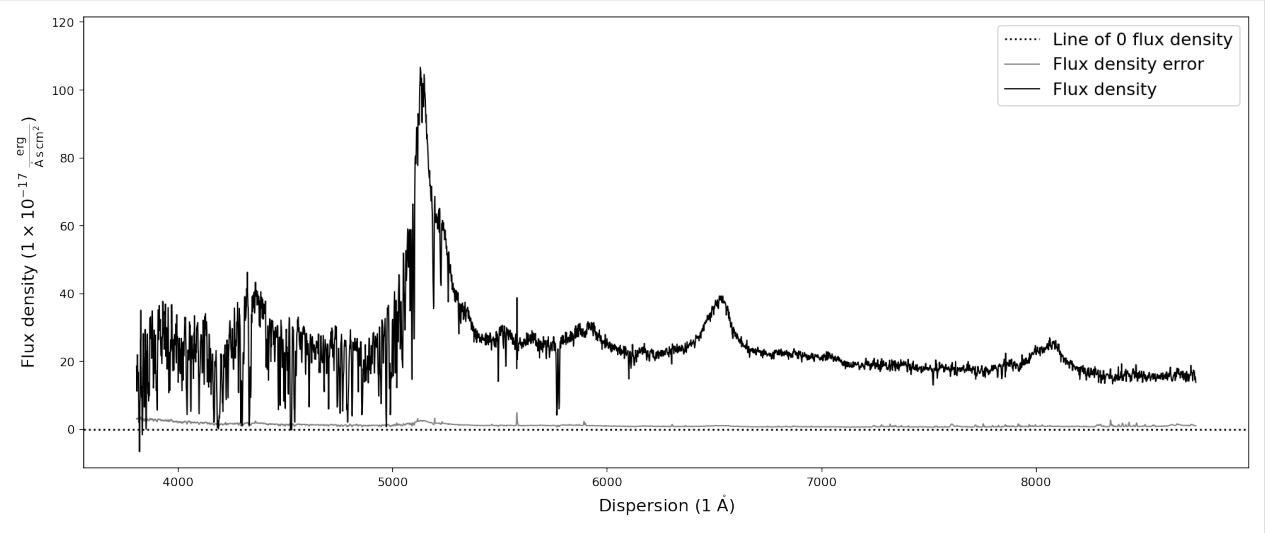
4.5 Reading in spectra from other formats

In the current version of Sculptor the SpecOneD module has three implemented methods to read in spectra from * an IRAF 1D fits file (in many cases the flux density and dispersion units need to be set manually) * an PypeIT 1D fits file * an SDSS fits file

These read function automizes the steps of the manual initialization above. We now go through quick examples:

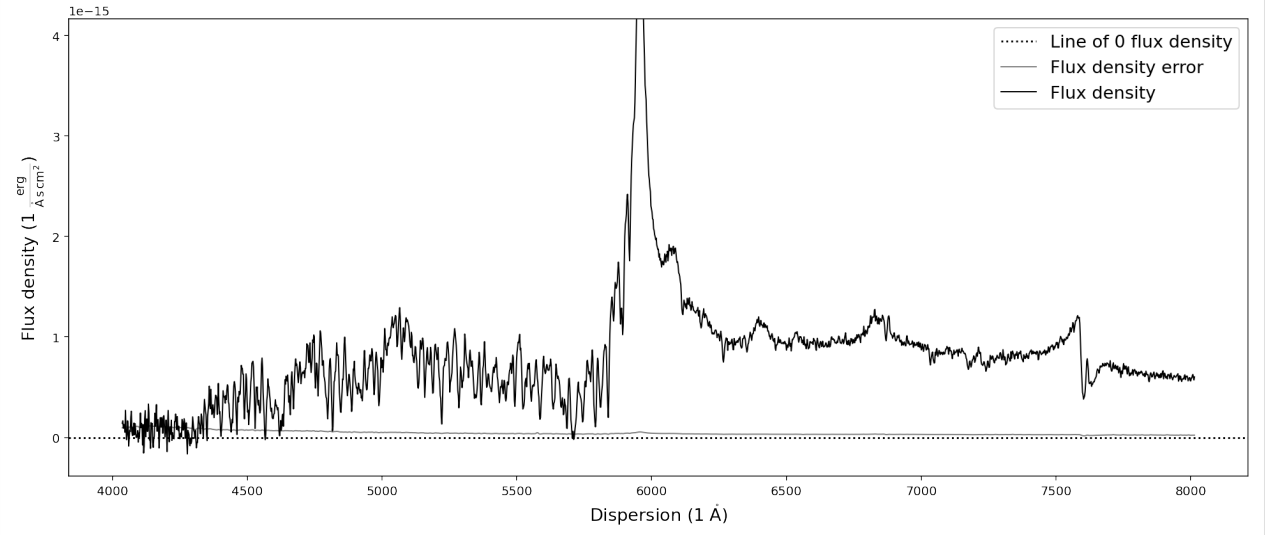
We begin by readin in the SDSS quasar spectrum from before in only two lines:

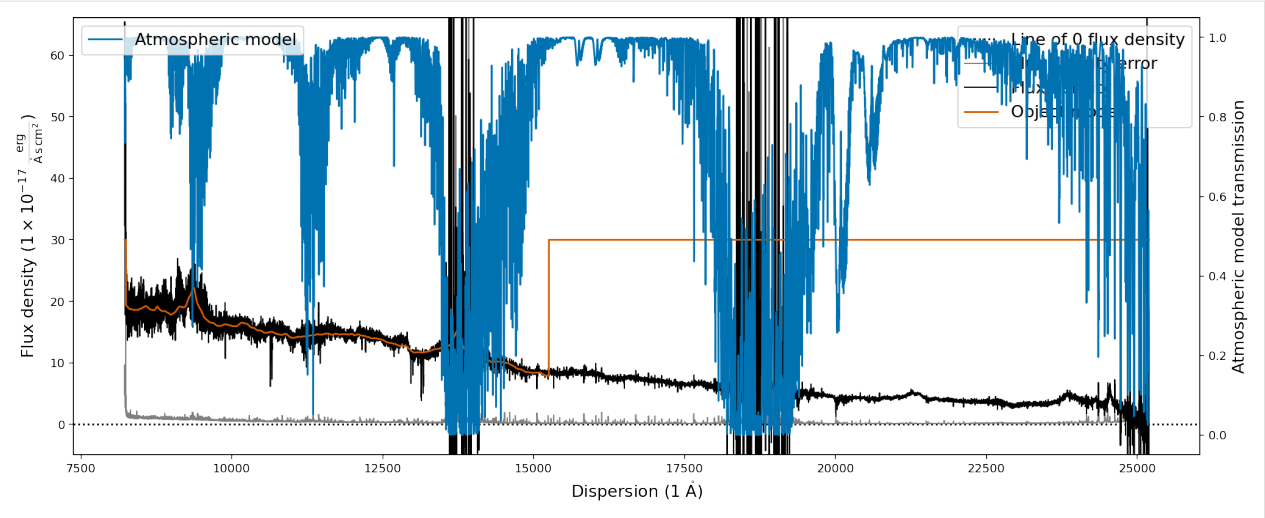
[8]:
```
# Initializing an empty SpecOneD object
sdss_spec = sod.SpecOneD()
# Read in the spectrum using the SDSS read function
sdss_spec.read_sdss_fits('../../sculptor/data/example_spectra/J030341.04-002321.8_0.fits
↪')
# Plot the spectrum
sdss_spec.plot()
```



The Sculptor data directory also includes example (quasar) spectra in IRAF and PypeIT fits format. Reading them in as a SpecOneD object is just as simple as in the case of the SDSS spectrum.

[9]:
```
# Initializing an empty SpecOneD object
iraf_spec = sod.SpecOneD()
# Read in the spectrum using the IRAF read function
iraf_spec.read_from_fits('../../sculptor/data/example_spectra/J2125-1719_OPT_A.fits')
# Plot the spectrum
iraf_spec.plot()

# Initializing an empty SpecOneD object
pypeit_spec = sod.SpecOneD()
# Read in the spectrum using the PypeIt read function
pypeit_spec.read_pypeit_fits('../../sculptor/data/example_spectra/J2125-1719_NIR.fits')
# Plot the spectrum
pypeit_spec.plot()
```





The first plot shows the optical spectrum of the ultra-luminous quasar J2125-1719, whereas the second shows the near-infrared spectrum.

The second plot however does show additional information, the object model (orange) and the atmospheric transmission (blue) from the PypeIt telluric correction routines. It also shows that the default *.plot()* capabilities of the SpecOneD class are not appropriate for generating publication grade figures as the legends in the second plot are not well placed.

4.6 Unit conversions

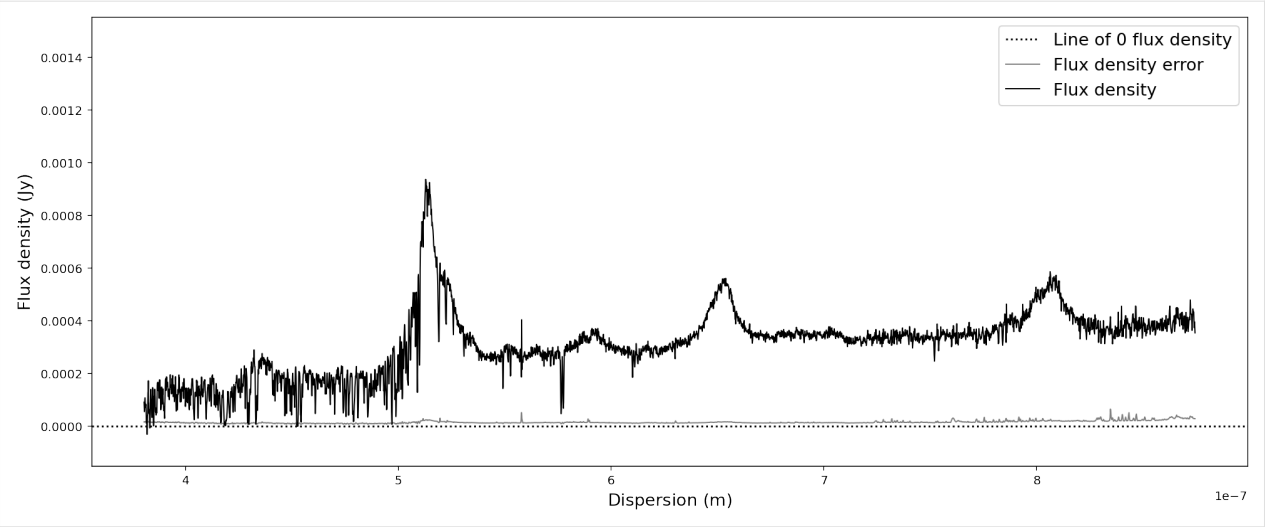
With the flux density and dispersion units included as astropy unit classes within the SpecOneD object, we have all the great unit conversion functionality of astropy.units at our disposal. We want to briefly demonstrate how this works. Let us begin again with our default SDSS quasar spectrum *sdss_spec*. The default units were

[10]:
```
print(sdss_spec.fluxden_unit)
print(sdss_spec.dispersion_unit)
```

```
1e-17 erg / (Angstrom cm2 s)
1.0 Angstrom
```

The function *convert_spectral_units* allows us to convert the spectral units by specifying the new dispersion and new flux density unit.

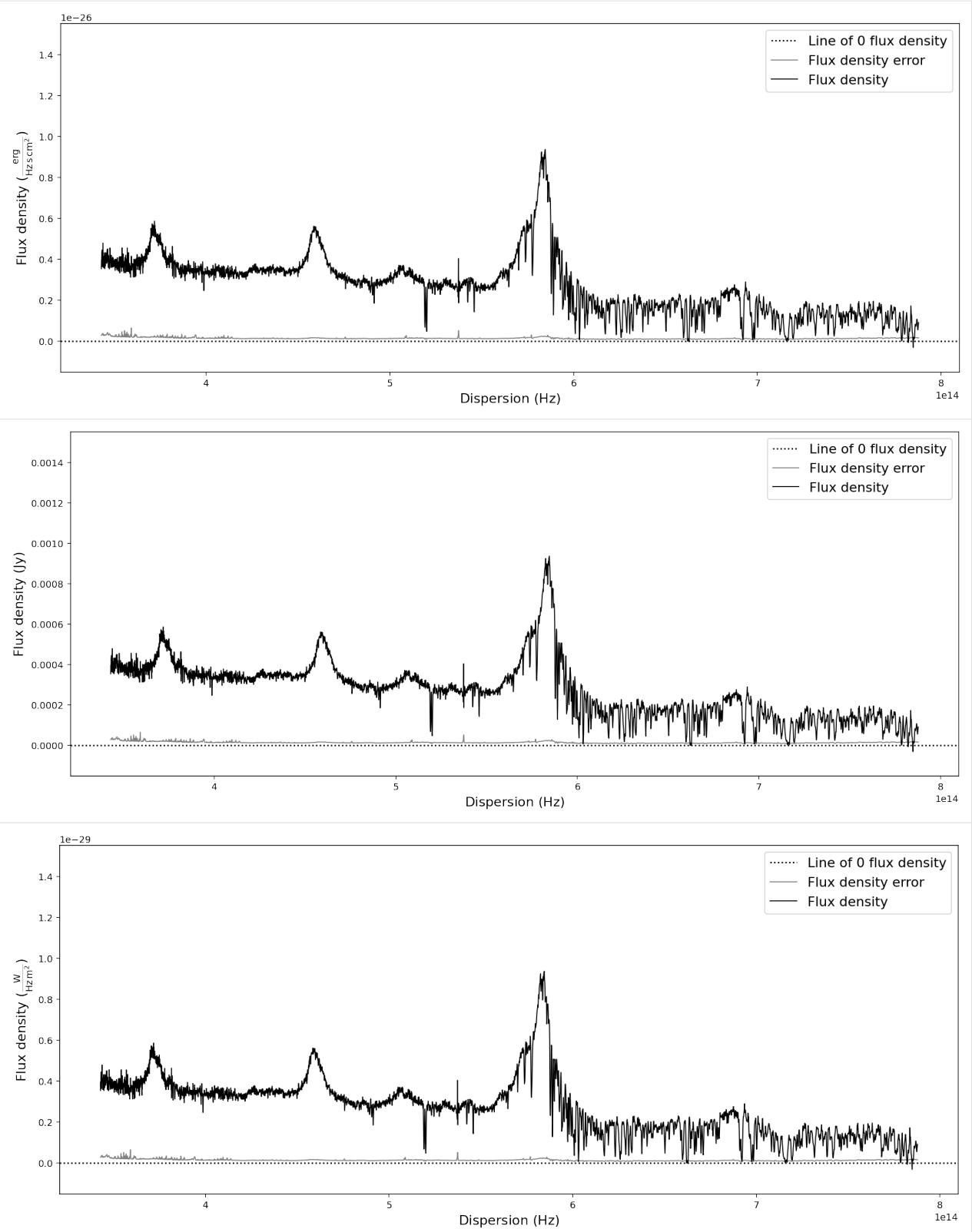
[11]:
```
# Convert the dispersion to meters and the flux density to Jansky
sdss_spec.convert_spectral_units(1*units.m,1*units.Jy)
# Display the spectrum
sdss_spec.plot()
```

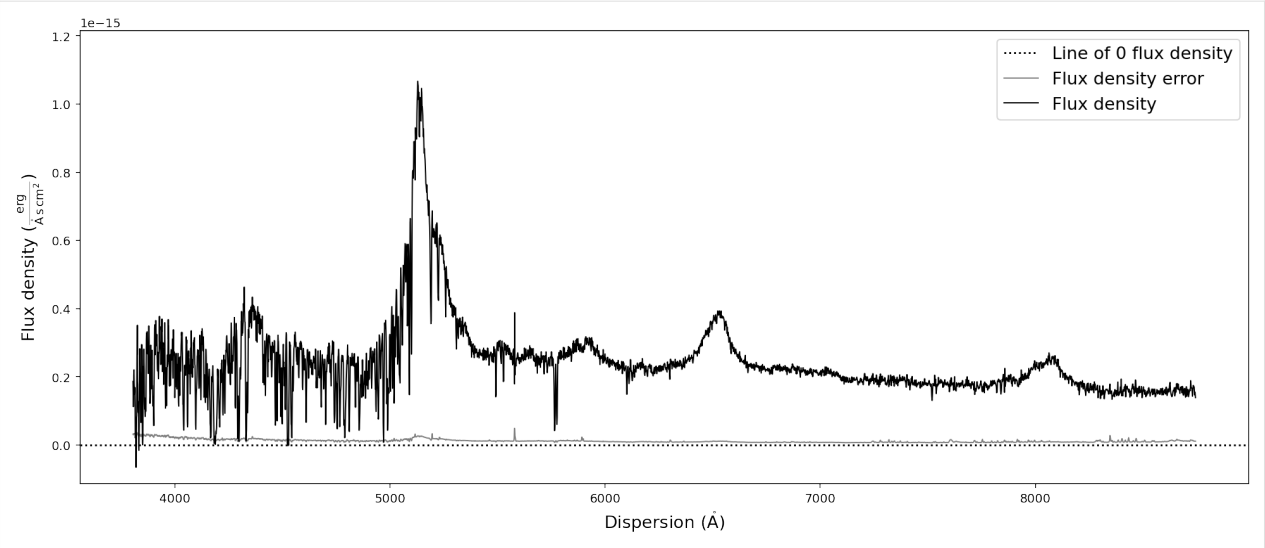


As long as the new units are reasonable for a flux density and a dispersion axis, you will be able to convert the spectrum. Internally the conversion uses *spectral* and *spectral_density* equivalencies, which will lead to an error if one would choose units that do not obey these equivalencies. Please consult the astropy documentation for details.

For our case, the SpecOneD class comes with default conversion functions for common spectral units:

[12]:
```
# Convert SpecOneD spectrum to flux density per unit frequency (Hz) in cgs units.
sdss_spec.to_fluxden_per_unit_frequency_cgs()
# Display the spectrum
sdss_spec.plot()

# Convert SpecOneD spectrum to flux density per unit frequency (Hz) in Jy.
sdss_spec.to_fluxden_per_unit_frequency_jy()
# Display the spectrum
sdss_spec.plot()

# Convert SpecOneD spectrum to flux density per unit frequency (Hz) in SI units.
sdss_spec.to_fluxden_per_unit_frequency_si()
# Display the spectrum
sdss_spec.plot()

# Convert SpecOneD spectrum to flux density per unit wavelength (Angstroem) in cgs units.
sdss_spec.to_fluxden_per_unit_wavelength_cgs()
# Display the spectrum
sdss_spec.plot()
```


1e−26
Line of 0 flux density
Flux density error
Flux density
Flux density ($\frac{erg}{Hz\,s\,cm^2}$)
Dispersion (Hz)
1e14
Line of 0 flux density
Flux density error
Flux density
Flux density (Jy)
Dispersion (Hz)
1e14
1e−29
Line of 0 flux density
Flux density error
Flux density
Flux density ($\frac{W}{Hz\,m^2}$)
Dispersion (Hz)
1e14



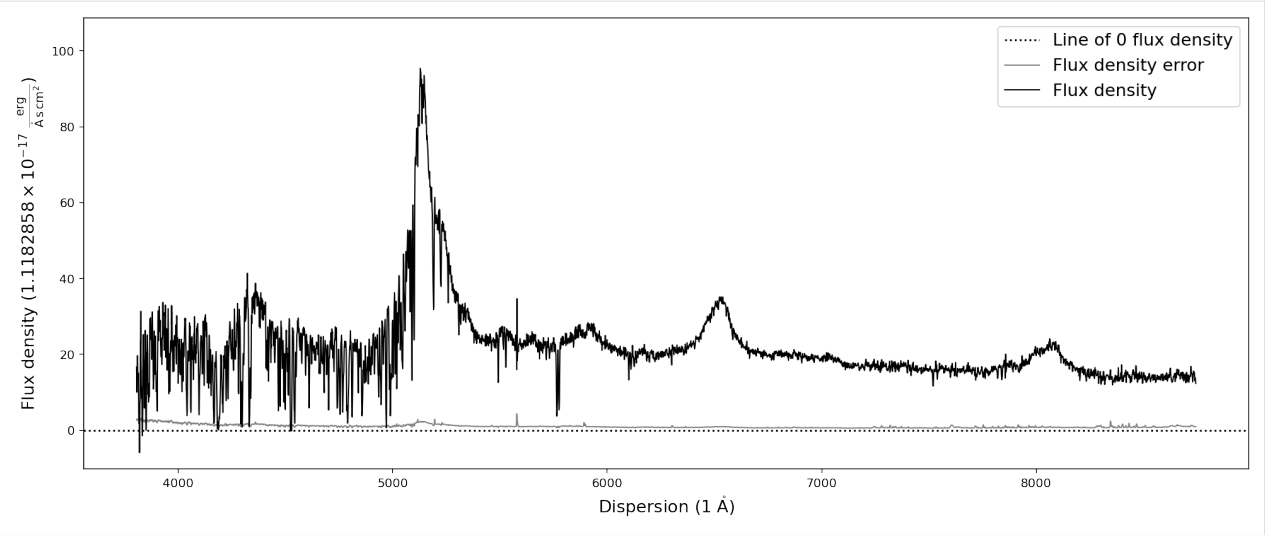
After all these conversion the last one brought us back to flux density per unit wavelength in the same units that we originally read the spectrum in - with one major difference: the 1e-17 factor in front of the flux density has been multiplied to the numpy.ndarray of the flux density and been removed from the flux density unit. However, in these particular units the flux density has very low numerical values, which could pose a problem to fitting algorithms. Therefore, we want to normalize the flux density numerical values and hide large factors in the unit.

4.7 "Normalizing" the flux density numpy.ndarray

Within SpecOneD we provide three functions that allow to "normalize" the flux density numpy.ndarray. The first function calculates the median flux density error and normalizes the flux density unit accordingly.

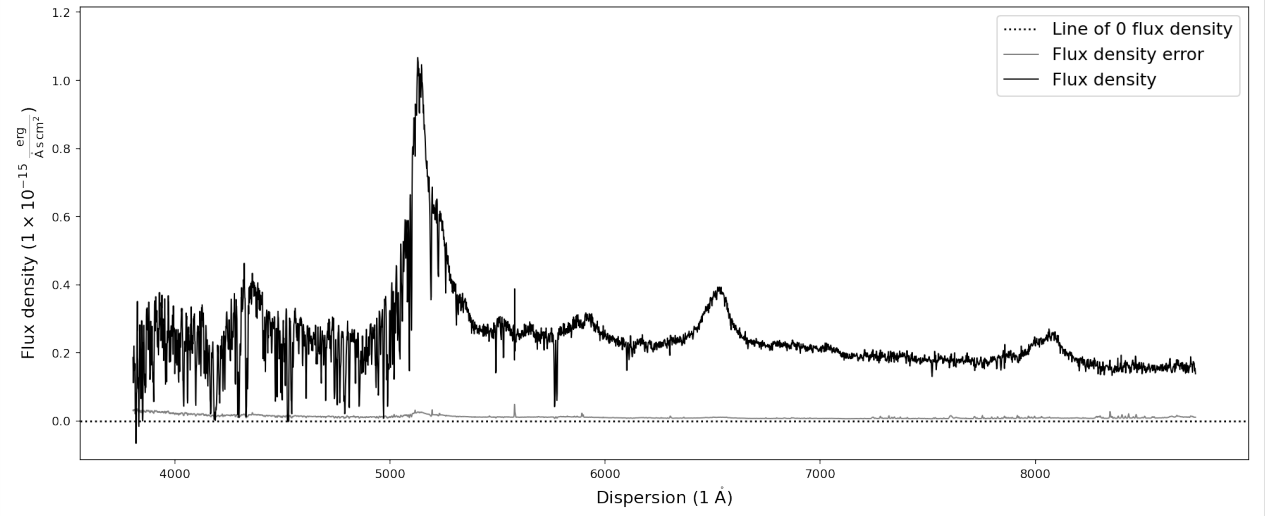
Note that these function return a new SpecOneD object by default. In fact, the majority of the SpecOneD functions that manipulate or modify the spectrum return a modified copy by defaul. This approach is designed to leave the original spectrum *untouched*. However, you can overwrite the original spectrum by using the *inplace* keyword argument of those functions and set it to *True*.

[13]:
```
# Normalize flux density unit by flux density error
nspec = sdss_spec.normalize_fluxden_by_error()
nspec.plot()
```

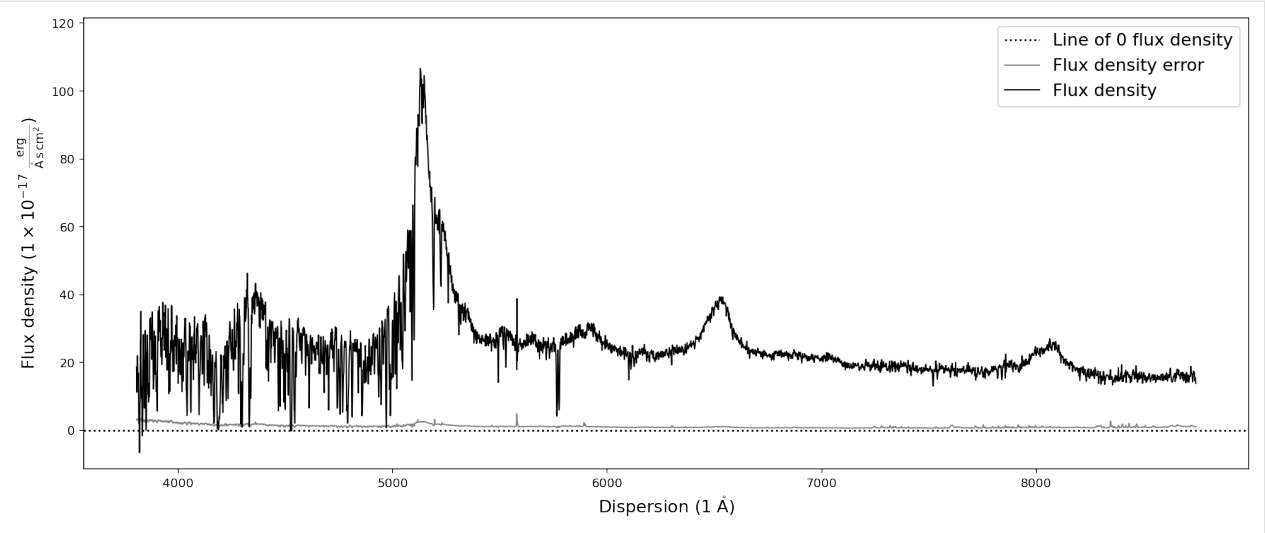


As you can see, the flux density axis now shows that its unit is ~1.12 * 1e-17 erg/s/cm^2/A. However, in many cases we want to set a specific pre-factor to the flux density unit or modify it by a specific factor. For this purpose two additional functions exist:

[14]:
```
# Normalize the spectrum BY a specific unit factor of 1e-15.
nspec = sdss_spec.normalize_fluxden_by_factor(1e-15)
nspec.plot()

# Applying the same factor again, might lead to ridiculous pre-factors
print(nspec.normalize_fluxden_by_factor(1e-15).fluxden_unit)

# Normalize the spectrum TO a specific unit pre-factor
nspec = sdss_spec.normalize_fluxden_to_factor(1e-17)
nspec.plot()
```




```
1e-30 erg / (Angstrom cm2 s)
```



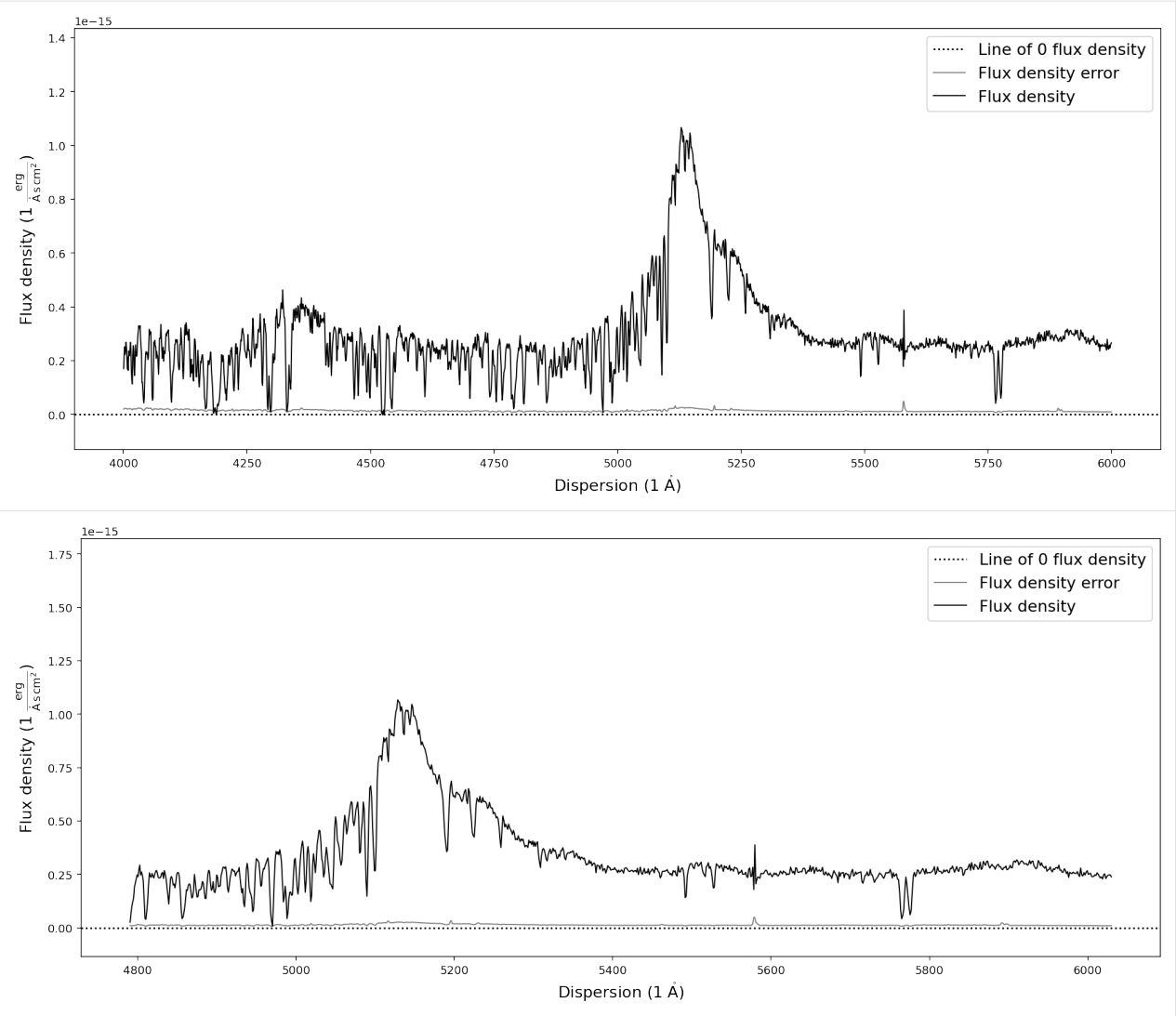
4.8 Manipulating spectra

In order to prepare the science spectrum for analysis one often has to manipulate the spectral data. This includes resampling the spectra to a pre-defined resolution, trimming the spectra, or matching multiple spectra to construct a composite spectrum. The SpecOneD class provides functionality for all of these cases.

4.8.1 Trimming spectra

Trimming spectral data with SpecOneD is implemented in the function *trim_dispersion*. A tuple of two float values provide the lower and upper limits of the new spectrum. By default the numerical limits are assumed to be in units of the dispersion axis (*mode='physical'*) and trim the spectrum at the closest points in dispersion units. Alternatively, one can change the mode to *mode='pixel'* and then trim the spectra according to the pixel number. By default the function will return a new spectrum. In case the user wants to overwrite the old spectrum, one can specify this by *inplace=True* in the function. If we return a new spectrum, we can immediately plot it by chaining the *.plot()* function to the trimming function.

[15]:
```
# Trim and plot the SDSS quasar spectrum in dispersion units
sdss_spec.trim_dispersion([4000,6000]).plot()

# Trim and plot the SDSS quasar spectrum in pixel units
sdss_spec.trim_dispersion([1000,2000], mode='pixel').plot()
```

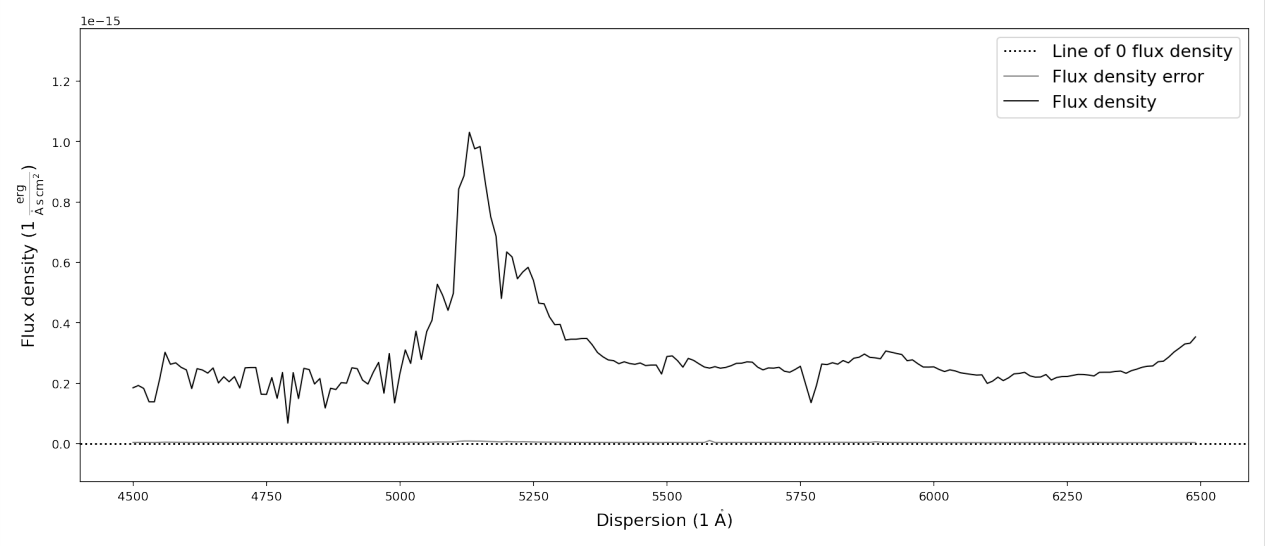


4.8.2 Resampling

In some cases one wants to resample the spectrum to a constant resolution to the resolution of another spectrum. For these cases Sculptor uses SpectRes package by Adam Carnall. The SpecOneD class provides a wrapper function called *resample* that applies the SpectRes resample method to the SpecOneD object.

As an argument it takes the new resolution as an numpy.ndarray.

[16]:
```
import numpy as np

# Setting up the new dispersion axis
new_dispersion = np.arange(4500,6500,10)

sdss_spec.resample(new_dispersion).plot()
```

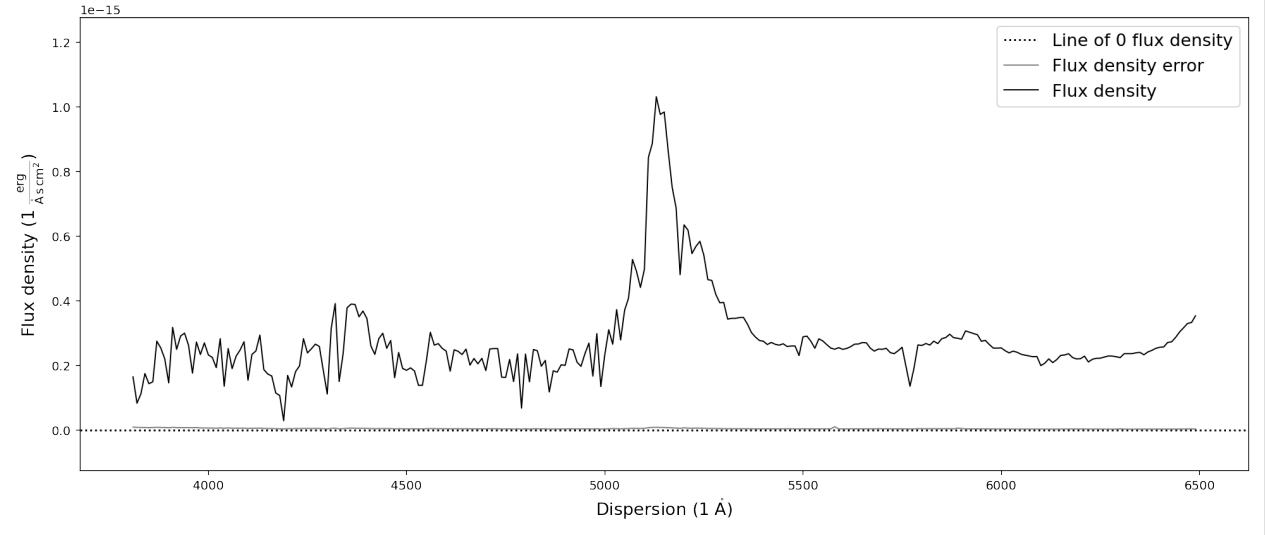


We have now created a lower resolution version of the previous SDSS quasar spectrum in the dispersion range of 4500A to 6500A. However, what happens if the new dispersion range is not covered by the spectrum?

In the current implementation this will automatically return an Error! However, one can set the keyword argument *force=True*, which will force the new dispersion range to be within the original spectral dispersion, trimming it automatically. This will prompt a warning message, so that the user is aware of the modifications by the function.

[17]:
```
# Setting up the new dispersion axis
new_dispersion = np.arange(2000,6500,10)

sdss_spec.resample(new_dispersion, force=True).plot()
```

```
[WARNING] Forcing new spectral dispersion axis to be within the limits of the old␣
↪spectral dispersion.
```

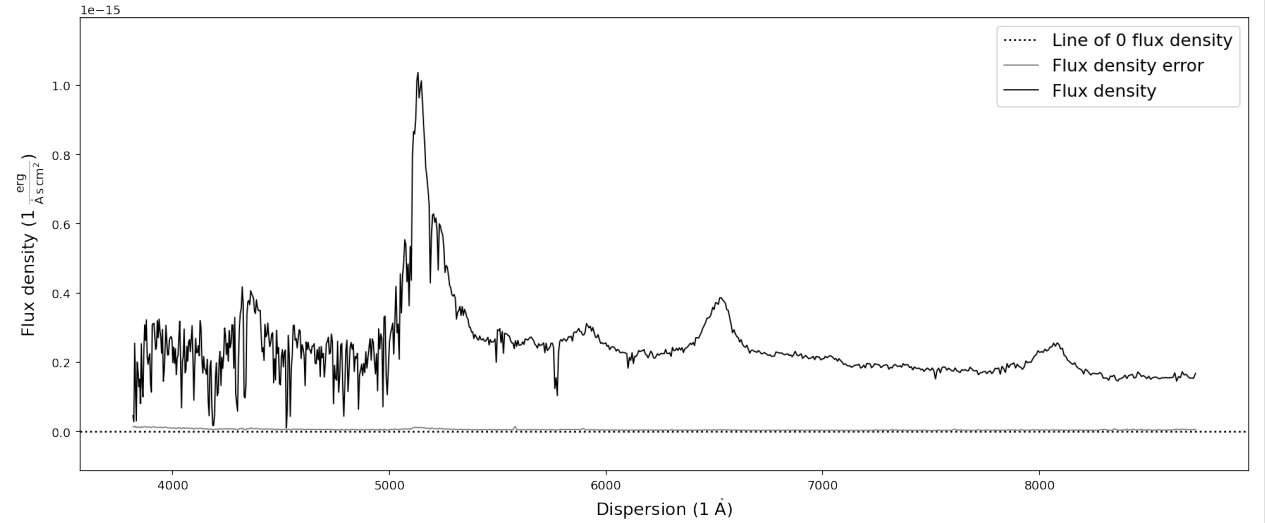


4.8.3 Reducing the spectral resolution via resampling & binning

In some cases it becomes necessary to reduce the resolution of a spectrum to increase the signal-to-noise ratio. Within the SpecOneD class this can either be achieved by resampling the spectrum to a dispersion with wider pixel sizes or by binning multiple pixels.

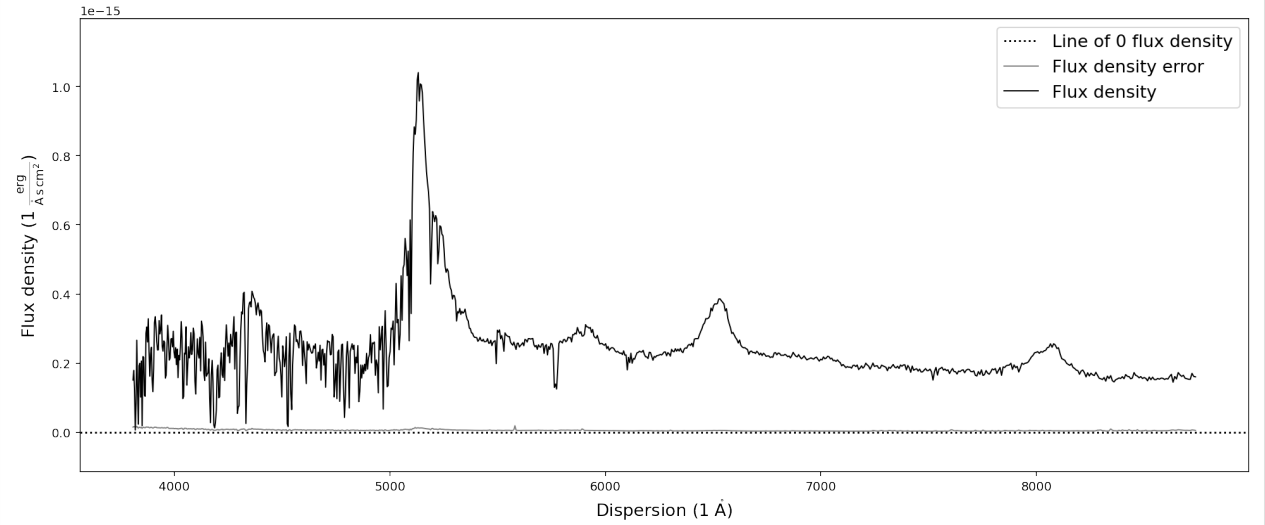
The *resample_to_resolution* function uses the *resample* method discussed above to automatically the spectrum to a new dispersion axis with the resolution in km/s provided by the argument. The *buffer* keyword argument (default value = 2) determines how many pixels at the beginning and the end of the spectrum will be omitted in the resampling process.

[18]:
```
# Resample the spectrum to a resolution of 300 km/s
sdss_spec.resample_to_resolution(300).plot()
```



Binning the spectrum by an integer number of pixels can be achieve with the *bin_by_npixels* function. One has to simply specify the number of pixels that should be binned.

[19]:
```
sdss_spec.bin_by_npixels(4).plot()
```

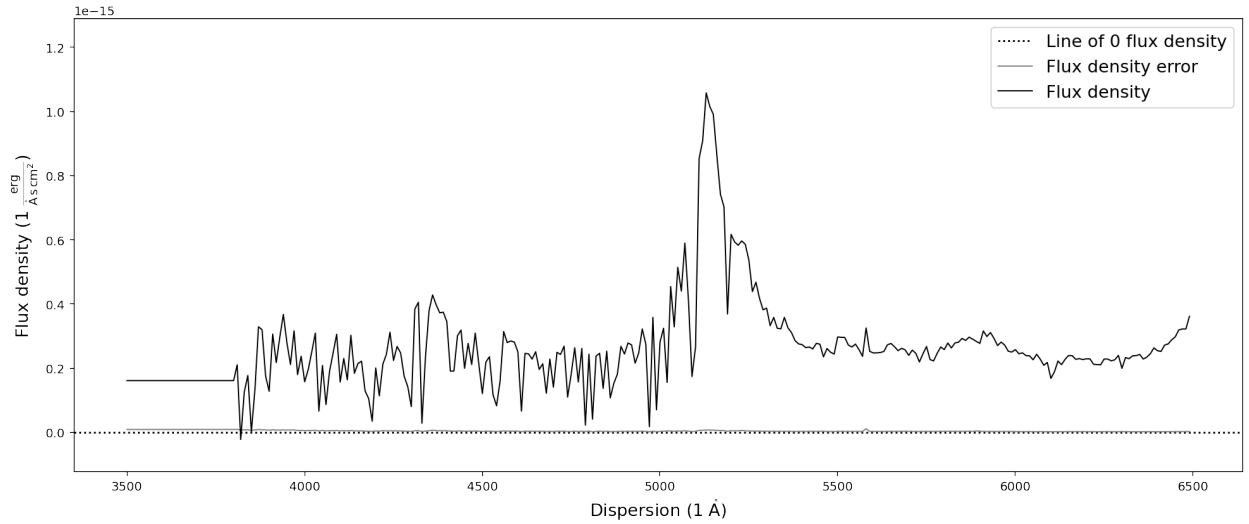


4.8.4 Interpolating spectra

While interpolating spectra onto a new spectral axis will cause the flux density and flux density errors to be correlated, it can be useful in many cases (generating figures). In the SpecOneD module we use interpolation of spectral data onto new dispersion axis in a range of contexts. The functions *broaden_by_gaussian*, *calculate_passband_flux*, *calculate_passband_ab_magnitude* all use the *interpolate* function internally.

The function uses the scipy.interpolate.interp1d method and takes the new dispersion as the argument. By default it will linearly interpolate (*kind='linear'*) the flux density and flux density error to the new dispersion axis, padding values outside the original dispersion range with a constant (*fill_value='const'*).

[20]:
```
# Setting up the new dispersion axis
new_dispersion = np.arange(3500,6500,10)

sdss_spec.interpolate(new_dispersion).plot()
```



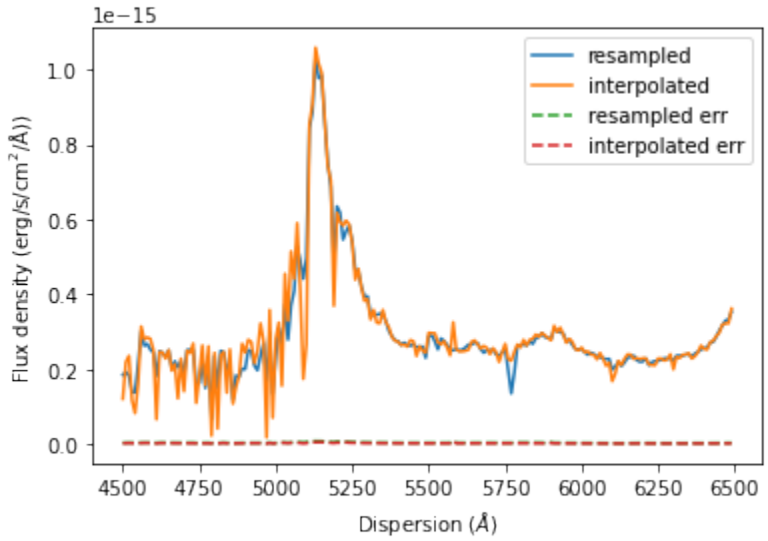
There are differences between the *resample* and *interpolate* methods that are especially evident when sampled with a low resolution. In addition the SpectRes *resample* calculates the resulting flux density error more accurately, while the *interpolate* function only adjusts the signal to noise ratio by the resolution factor (sqrt("old resolution"/"new resolution")).

[21]:
```
# Setting up the new dispersion axis
new_dispersion = np.arange(4500,6500,10)

resampled = sdss_spec.resample(new_dispersion)
interpolated = sdss_spec.interpolate(new_dispersion)

import matplotlib.pyplot as plt

plt.plot(resampled.dispersion, resampled.fluxden, label='resampled')
plt.plot(interpolated.dispersion, interpolated.fluxden, label='interpolated')
plt.plot(resampled.dispersion, resampled.fluxden_err, '--', label='resampled err')
plt.plot(interpolated.dispersion, interpolated.fluxden_err, '--', label='interpolated err
↪')
plt.xlabel(r'Dispersion ($\AA$)')
plt.ylabel(r'Flux density ($\rm{erg}/\rm{s}/\rm{cm}^2/\AA$)')
plt.legend()
```

(continues on next page)

(continued from previous page)

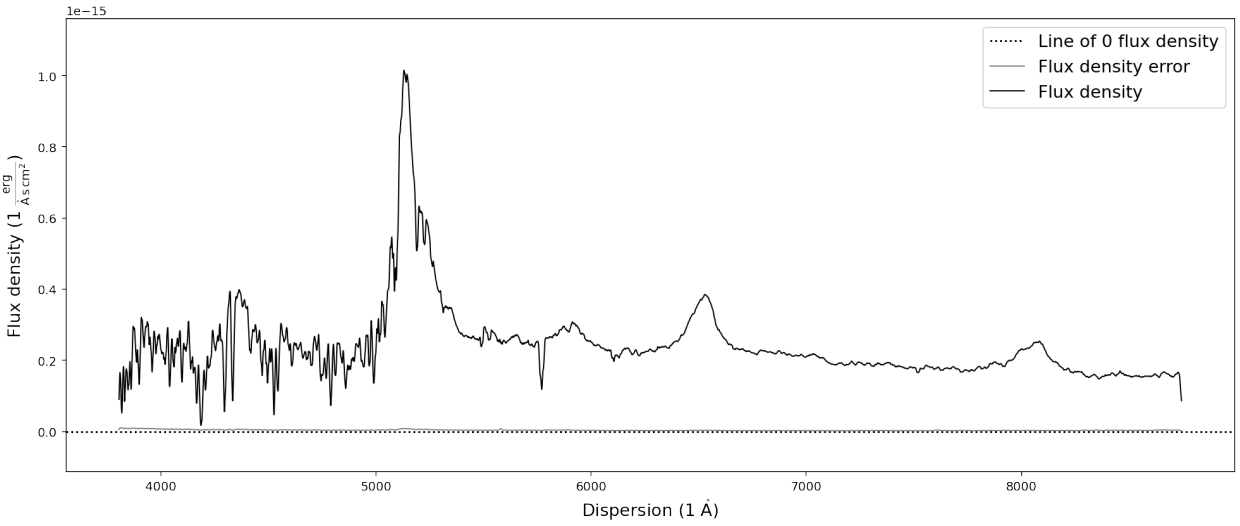
```
plt.show()
```



4.8.5 Smoothing spectra

In addition the methods described above one can also smooth the spectrum using a boxcar or gaussian kernel. SpecOneD uses the astropy.convolution functionality to achieve this. The function takes the width (in pixels) of the kernel as an argument. Keyword arguments determine the kernel (default *kernel="boxcar"*) and whether the signal-to-noise ratio will be adjusted (default *scale_sigma=True*) by sqrt(width).

[22]:
```
# Smooth the spectrum with a boxcar kernel of size 5 pixels and plot it
sdss_spec.smooth(10).plot()
```



4.8.6 Matching spectra to the same dispersion axis

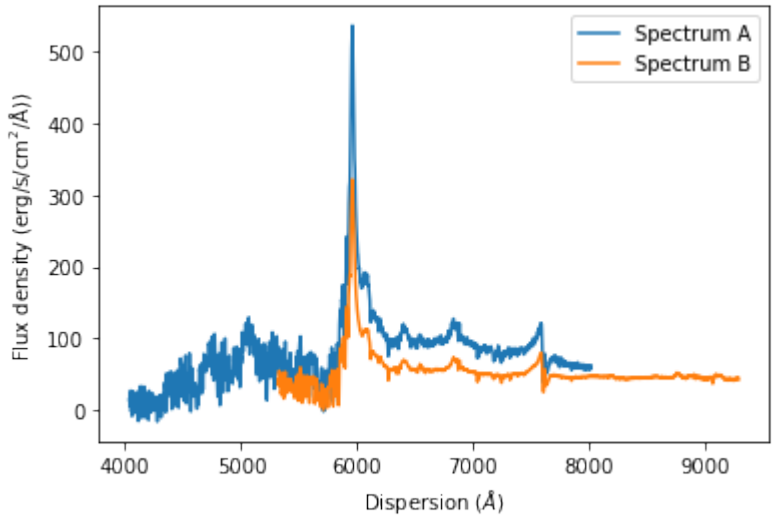
One of the applications of resampling spectra is to match two spectra of the same source, but observed with at different resolution to the same dispersion.

For this purpose the SpecOneD class has a function called *match_dispersions*. The documentation for this function reads:

```
Match the dispersion of the current spectrum and the secondary
spectrum.

Both, current and secondary, SpecOneD objects are modified in this
process. The dispersion match identifies the maximum possible overlap
in the dispersion direction of both spectra and automatically trims
them to that range.

If the current (primary) spectrum overlaps fully with the secondary
spectrum the dispersion of the secondary will be interpolated/resampled
to the primary dispersion.

If the secondary spectrum overlaps fully with the primary, the primary
spectrum will be interpolated/resampled on the secondary spectrum
resolution, but this will only be executed if 'force==True' and
'match_secondary==False'.

If there is partial overlap between the spectra and 'force==True'
the secondary spectrum will be interpolated/resampled to match the
dispersion values of the primary spectrum.

If there is no overlap a ValueError will be raised.
```

We now demonstrate how use this function with a quick example. We use two spectra from the example folder of the ultra-luminous quasar J2125-1719. These optical spectra cover a different dispersion range, so we will see how the function works in this case.

[23]:
```
# Read in the spectra of J2125-1719
specoptA = sod.SpecOneD()
specoptA.read_from_fits('../../sculptor/data/example_spectra/J2125-1719_OPT_A.fits')
specoptB = sod.SpecOneD()
specoptB.read_from_fits('../../sculptor/data/example_spectra/J2125-1719_OPT_B.fits')

# The optical spectra will be scaled to 1e-17 erg/s/cm^2/A
specoptA.convert_spectral_units(1.*units.AA, 1e-17*units.erg/units.s/units.cm**2/units.
↪AA)
specoptB.convert_spectral_units(1.*units.AA, 1e-17*units.erg/units.s/units.cm**2/units.
↪AA)

# Plot the spectra before matching them
plt.plot(specoptA.dispersion, specoptA.fluxden, label='Spectrum A')
plt.plot(specoptB.dispersion, specoptB.fluxden, label='Spectrum B')
plt.xlabel(r'Dispersion ($\AA$)')
plt.ylabel(r'Flux density ($\rm{erg}/\rm{s}/\rm{cm}^2/\AA$))')
plt.legend()
plt.show()
```

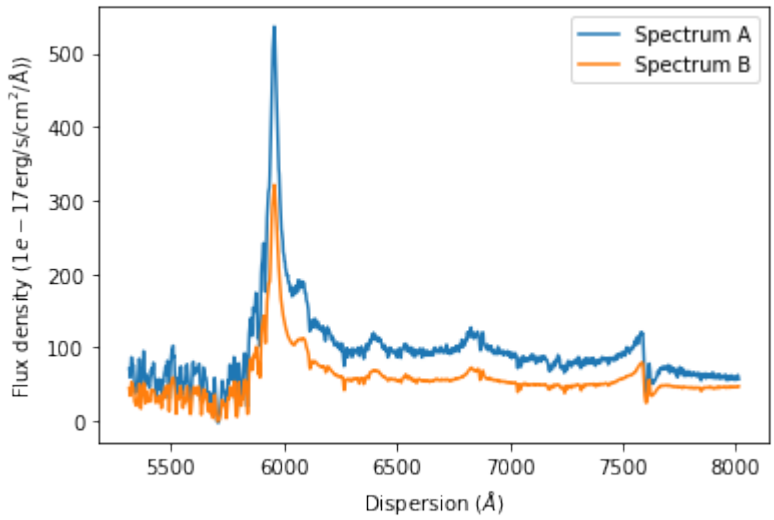


Now we will use the match dispersion function to match the dispersion of spectrum A to spectrum B. The *match_secondary* keyword indicates whether the secondary dispersion axis will always be matched to the primary spectrum of if the reverse is allowed. The default matching method is interpolation (*method='interpolate'*).

If the primary spectrum (here specoptA) is not fully contained within the dispersion range of the secondary spectrum (here specoptB), we need to set *force=True* to allow the primary spectrum to be reduced to the overlap region of both spectra.

Note that the *match_dispersions* function does NOT have a *inplace* keyword argument. This functions ALWAYS modifies both original spectra!

[24]:
```
specoptA.match_dispersions(specoptB, match_secondary=True,
                           force=True, method='interpolate',
                           interp_method='linear')
# Plot the spectra AFTER matching them
plt.plot(specoptA.dispersion, specoptA.fluxden, label='Spectrum A')
plt.plot(specoptB.dispersion, specoptB.fluxden, label='Spectrum B')
plt.xlabel(r'Dispersion ($\AA$)')
plt.ylabel(r'Flux density ($1e-17\rm{erg}/\rm{s}/\rm{cm}^2/\AA$))')
plt.legend()
plt.show()
```



The plot illustrates that both spectra have been automatically cut to their overlap region. In this range the dispersion axes of both spectra are now identical.

[25]:
```
print(specoptA.dispersion)
print(specoptB.dispersion)
```

```
[5316.3259873  5318.30567772 5320.28536813 ... 8008.70494917 8010.68463959
 8012.66433   ]
[5316.3259873  5318.30567772 5320.28536813 ... 8008.70494917 8010.68463959
 8012.66433   ]
```

4.8.7 Renormalize the flux density between two spectra

The SpecOneD class also includes methods to renormalize the spectral flux density either to a secondary spectrum or a passband magnitude. For now we will demonstrate how one can match the flux density of one spectrum to a different one in their overlap region. For this example we use the optical spectra of quasar J2125-1719 above.

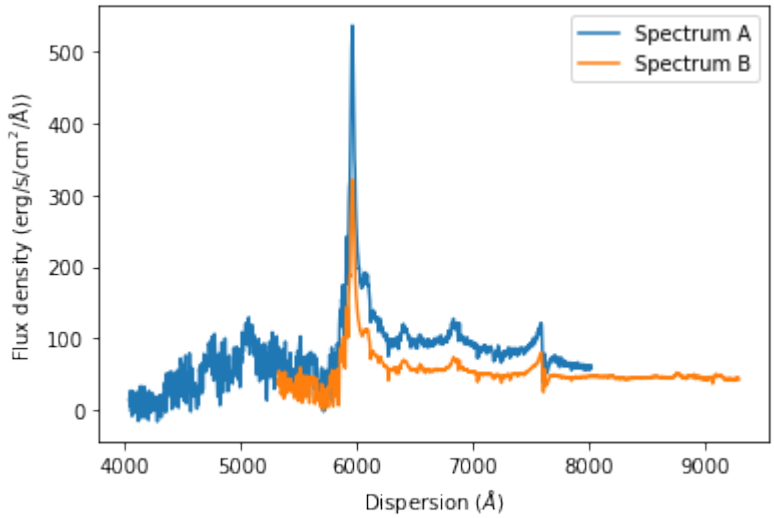
The SpecOneD function to carry out the renormalization is aptly named *renormalize_by_spectrum*.

[26]:
```
# Read in the spectra of J2125-1719
specoptA = sod.SpecOneD()
specoptA.read_from_fits('../../sculptor/data/example_spectra/J2125-1719_OPT_A.fits')
specoptB = sod.SpecOneD()
specoptB.read_from_fits('../../sculptor/data/example_spectra/J2125-1719_OPT_B.fits')

# The optical spectra will be scaled to 1e-17 erg/s/cm^2/A
specoptA.convert_spectral_units(1.*units.AA, 1e-17*units.erg/units.s/units.cm**2/units.
↪AA)
specoptB.convert_spectral_units(1.*units.AA, 1e-17*units.erg/units.s/units.cm**2/units.
↪AA)

# Plot the spectra before renomalizing the flux density
plt.plot(specoptA.dispersion, specoptA.fluxden, label='Spectrum A')
plt.plot(specoptB.dispersion, specoptB.fluxden, label='Spectrum B')
plt.xlabel(r'Dispersion ($\AA$)')
```

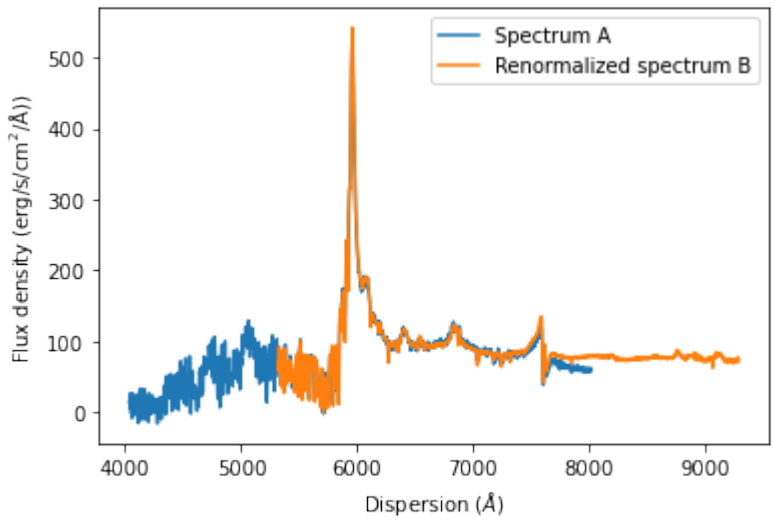
(continues on next page)

(continued from previous page)

```
plt.ylabel(r'Flux density ($\rm{erg}/\rm{s}/\rm{cm}^2/\AA$)')
plt.legend()
plt.show()
```



[27]:
```
# Renormalize the flux density of spectrum B to spectrum A in the overlap region of␣
↪6000A-7000A.
nspecoptB = specoptB.renormalize_by_spectrum(specoptA, dispersion_limits=[6000,7000])

# Plot the spectra after renomalizing the flux density
plt.plot(specoptA.dispersion, specoptA.fluxden, label='Spectrum A')
plt.plot(nspecoptB.dispersion, nspecoptB.fluxden, label='Renormalized spectrum B')
plt.xlabel(r'Dispersion ($\AA$)')
plt.ylabel(r'Flux density ($\rm{erg}/\rm{s}/\rm{cm}^2/\AA$)')
plt.legend()
plt.show()
```



If the dispersion limits for the overlap region are not specified, the full dispersion overlap of both spectra will be automatically chosen. **If one decides to choose dispersion limits that are not appropriate, the function will NOT**

stop you from doing so and return an incorrectly scaled spectrum.

The default *output_mode* (*output_mode='spectrum'*) returns a SpecOneD object. Alternatively one can specify *output_mode='flux_factor'* and the function will return the factor by which the spectrum should be scaled to match the flux density of the other one as a float value.

[28]:
```
# Calculate the scaling factor to scale spectrum B to spectrum A in the overlap region␣
↪of 6000A-7000A.
print(specoptB.renormalize_by_spectrum(specoptA, dispersion_limits=[6000,7000], output_
↪mode='flux_factor'))
```

```
1.6855747992533456
```

CHAPTER

FIVE

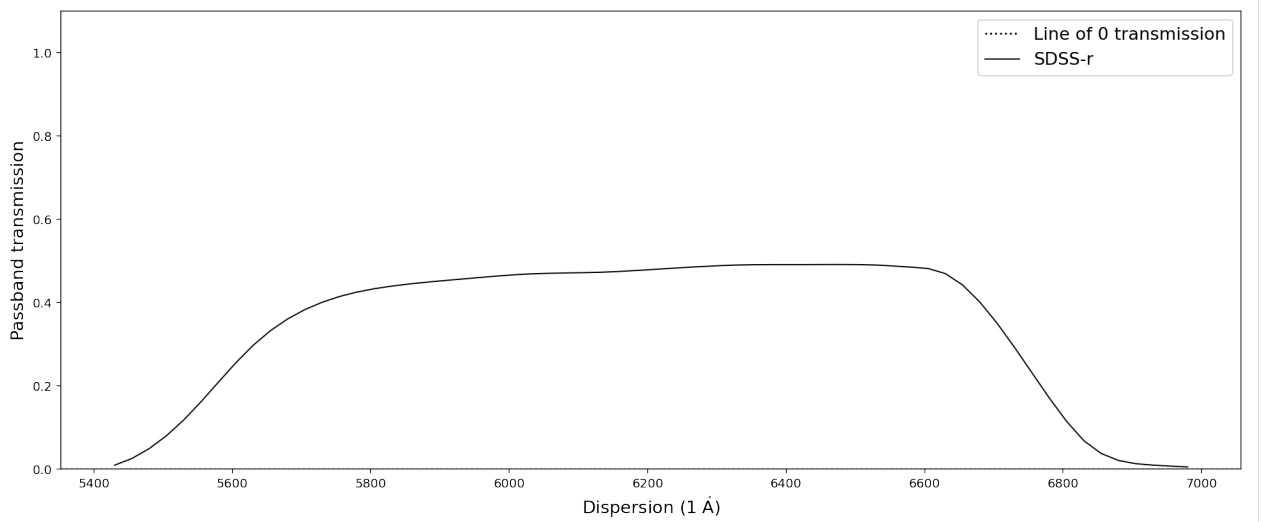
THE PASSBAND CLASS

The PassBand class is a child class of SpecOneD designed to store and manipulate filter transmission curves from various telescopes. The Sculptor package includes a range of filter transmission curves from various sources in the data/passbands folder.

These transmission curves can be automatically loaded into PassBand objects by using their name (omitting the .dat extension). They provide important functionality to the SpecOneD class for calculating passband fluxes or magnitudes and allow absolute flux normalization by the passband AB magnitude.

Let us start by loading the SDSS r-band filter provided by Sculptor.

[29]:
```
# Initialize the SDSS r-band filter as a PassBand object
pb = sod.PassBand('SDSS-r')

# Plot the SDSS r-band using the built in plot functionality
pb.plot()
```



Many SpecOneD functions that manipulate the dispersion axis also work on passbands. This includes *trim_dispersion* or *match_dispersion*, for example. However, some functionality requires flux densities and flux density errors and thus won't work with the PassBand class.

5.1 Calculating the spectral flux through a passband

5.1.1 Passband flux

For this example we will use the quasar spectrum of J030341.04-002321.8 again. We have already initialized the SDSS r-band as pb above, which we will be using in this example. The function that calculates the total passband flux is *calculate_passband_flux*.

Disclaimer: This function is written for passbands in quantum efficiency. Therefore, the (h*nu)^-1 term is not included in the integral.

Therefore, if you want to use passbands that are not in quantum efficiency you need to write your own function.

To match the passband and the spectrum to the same dispersion axis we can use either the *resample* or the *interpolate* methods described above. The lead to slightly different results, as can be seen below. If the spectrum does not fully overlap the passband one can set *force=True* to calculate the flux only in the overlap region. For this example we do not need to do that. (Setting *force=True* also prompts multiple warnings instructing the user to be especially careful with the intepretation of the results!)

[30]:
```
# Read in the sdss quasar spectrum
spec.read_sdss_fits('../../sculptor/data/example_spectra/J030341.04-002321.8_0.fits')

# Calculate the passband flux using the interpolate method to match the dispersions.
print(spec.calculate_passband_flux(pb, force=False, match_method='interpolate'))
# Calculate the passband flux using the resample method to match the dispersions.
print(spec.calculate_passband_flux(pb, force=False, match_method='resample'))
```

```
1.4209075815160446e-13 erg / (cm2 s)
1.4108874707002708e-13 erg / (cm2 s)
```

5.1.2 Passband magnitudes (AB system)

The SpecOneD class also allows to calculate the passband magnitude in the AB system with the *calculate_passband_ab_magnitude* function. It works pretty much identical to the *calculate_passband_flux* above, but returns the AB magnitude through the filter passband.

[31]:
```
# Calculate the passband flux in AB magnitudes using the interpolate method to match the␣
↪dispersions.
print(spec.calculate_passband_ab_magnitude(pb, force=False, match_method='interpolate'))
# Calculate the passband flux in AB magnitudes using the resample method to match the␣
↪dispersions.
print(spec.calculate_passband_ab_magnitude(pb, force=False, match_method='resample'))
```

```
17.60633670754543
17.61403048355622
```

5.2 Renormalizing the flux density to a passband AB magnitude - Absolute flux calibration

In some cases the absolute flux calibration of the observations might be inaccurate. Renormalizing the flux density of the spectrum through a passband to the passband photometry provides the best solution for an absolute flux calibration in these cases.

For this purpose the SpecOneD class offers the *renormalize_by_ab_magnitude* function, which words very similar to the *renormalize_by_spectrum* function above.

For the SDSS quasar we use as an example, the DR16 r-band magnitude is r=17.65. According to the SDSS website the r-band magnitude is close to AB, so we assume that r(AB)=17.65 in this case. We use the r-band from above and first decide to use the *output_mode='flux_factor'*. By default the *match_method* is set to *interpolate*.

[32]:
```
# SDSS r-band magnitude in AB
rmag_ab = 17.65

# Calculate the scaling factor for absolute flux calibration
print(spec.renormalize_by_ab_magnitude(rmag_ab, pb, output_mode='flux_factor'))
# Calculate the scaling factor for absolute flux calibration using the resample match␣
↪method
print(spec.renormalize_by_ab_magnitude(rmag_ab, pb, output_mode='flux_factor', match_
↪method='resample'))
```

```
0.9605825280212889
0.9674136110722544
```

The flux factors already suggest that the SDSS quasar spectrum has a pretty good absolute flux calibration.

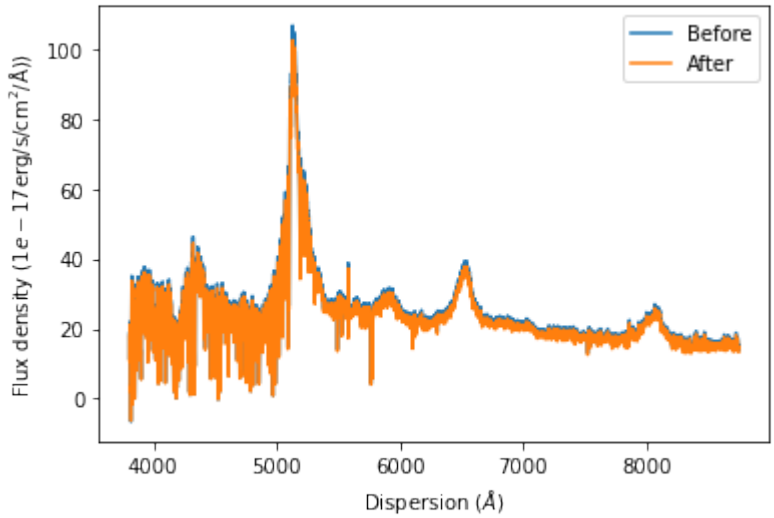
Now that we checked the flux factor output, we can run a simple test. First, we calculate the AB magnitude of the SDSS quasar spectrum in the r-band, then we normalize it to the r-band magnitude and lastly we calculate the AB magnitude again to check if the normalization was successfull. We will use the *interpolate* match method in all cases as it is the default.

[33]:
```
# Calculate the passband flux in AB magnitudes using the interpolate method to match the␣
↪dispersions.
print('r-band magnitude (before): ', spec.calculate_passband_ab_magnitude(pb))

# Renormalize the spectrum to the r-band magnitude
nspec = spec.renormalize_by_ab_magnitude(rmag_ab, pb)
print('Absolute flux calibration to r=17.65')

# Calculate the passband flux in AB magnitudes using the interpolate method to match the␣
↪dispersions.
print('r-band magnitude (after): ', nspec.calculate_passband_ab_magnitude(pb))
```

```
r-band magnitude (before):  17.60633670754543
Absolute flux calibration to r=17.65
r-band magnitude (after):  17.649999999999995
```

We have now absolute flux calibrated the SDSS quasar spectrum to numerical accuracy. We end this demonstration of the capabilies of the SpecOneD and PassBand classes by plotting the SDSS quasar spectrum before and after absolute flux calibration.

[34]:
```
# Plot the spectra after renomalizing the flux density
plt.plot(spec.dispersion, spec.fluxden, label='Before')
plt.plot(nspec.dispersion, nspec.fluxden, label='After')
plt.xlabel(r'Dispersion ($\AA$)')
plt.ylabel(r'Flux density ($1e-17\rm{erg}/\rm{s}/\rm{cm}^2/\AA$)')
plt.legend()
plt.show()
```



CHAPTER

SIX

PREPARING A COMPOSITE SPECTRUM FOR SCULPTOR MODELING USING THE SPECONED CLASS

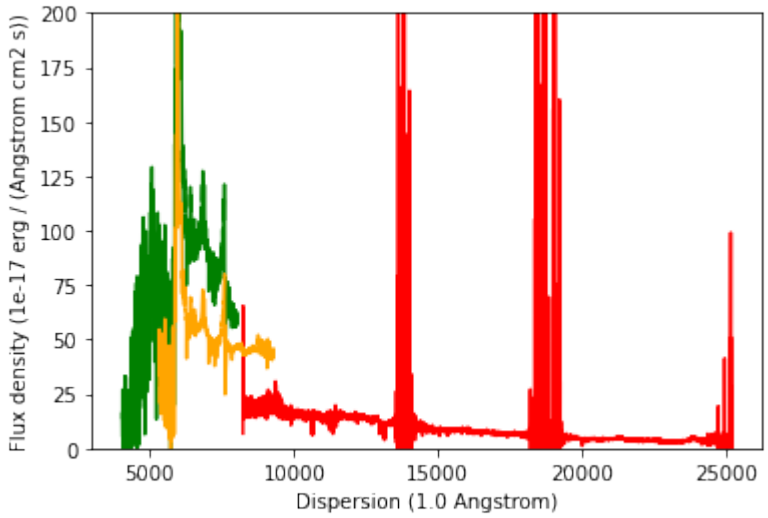
In this example we will be preparing two optical and one near-infrared spectrum for further spectral modeling using the Sculptor GUI. The spectra are of the ultra-luminous quasar J2125-1719 and are published in Schindler et al. 2020.

First, we import the SpecOneD module from Sculptor and read in the three spectra from the example folder.

[1]:
```
from sculptor import speconed as sod

from astropy import units as u

# Read in the spectra of J2125-1719
specnir = sod.SpecOneD()
specnir.read_pypeit_fits('../../sculptor/data/example_spectra/J2125-1719_NIR.fits')
specoptA = sod.SpecOneD()
specoptA.read_from_fits('../../sculptor/data/example_spectra/J2125-1719_OPT_A.fits')
specoptB = sod.SpecOneD()
specoptB.read_from_fits('../../sculptor/data/example_spectra/J2125-1719_OPT_B.fits')

# The optical spectra will be scaled to 1e-17 erg/s/cm^2/A
specoptA.convert_spectral_units(1.*u.AA, 1e-17*u.erg/u.s/u.cm**2/u.AA)
specoptB.convert_spectral_units(1.*u.AA, 1e-17*u.erg/u.s/u.cm**2/u.AA)
```

We could use SpecOneD's plot function to look at the spectra individually, but in this case we want to use matplotlib to show all spectra in one plot.

[2]:
```
import matplotlib.pyplot as plt

plt.plot(specnir.dispersion, specnir.fluxden, 'red')
plt.plot(specoptA.dispersion, specoptA.fluxden, 'green')
plt.plot(specoptB.dispersion, specoptB.fluxden, 'orange')
plt.ylim(0, 200)
plt.xlabel('Dispersion ({})'.format(specnir.dispersion_unit))
plt.ylabel('Flux density ({})'.format(specnir.fluxden_unit))
plt.show()
```



6.1 Deredden the science spectrum

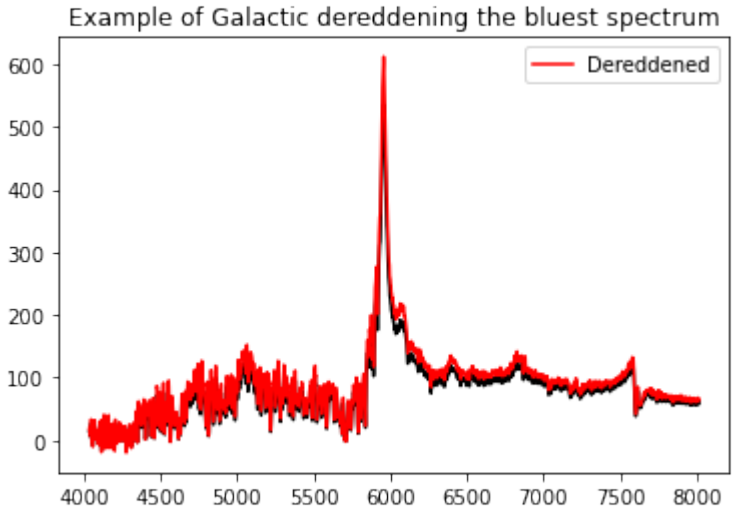
According to the IRSA dust map (https://irsa.ipac.caltech.edu/applications/DUST/) the quasar J2125-1719 (21h25m40.97s -17d19m51.3s Equ J2000) has a A_V=0.1606 using the Schlafly & Finkbeiner 2011 (ApJ 737, 103) Galactic dust map.

As we want use the dereddened magnitudes for absolute flux calibration, we deredden the three individual spectra before.

We demonstrate how one would use the speconed module to "deredden" the spectrum using the Fitzpatrick & Massa 2007 extinction curve. To do this speconed uses the python extinction package (https://github.com/kbarbary/extinction).

[3]:
```
a_v = 0.1606
r_v = 3.1
specnir = specnir.remove_extinction(a_v, r_v, extinction_law='fm07')

plt.plot(specoptA.dispersion, specoptA.fluxden,'k')

specoptA = specoptA.remove_extinction(a_v, r_v, extinction_law='fm07')
specoptB = specoptB.remove_extinction(a_v, r_v, extinction_law='fm07')


plt.plot(specoptA.dispersion, specoptA.fluxden,'r', label='Dereddened')
plt.legend()
plt.title('Example of Galactic dereddening the bluest spectrum')
plt.show()
```

```
[Warning] For Fitzpatrick & Massa 2007 R_V=3.1
[Warning] For Fitzpatrick & Massa 2007 R_V=3.1
[Warning] For Fitzpatrick & Massa 2007 R_V=3.1
```



6.2 Absolute flux calibration to broad band photometry

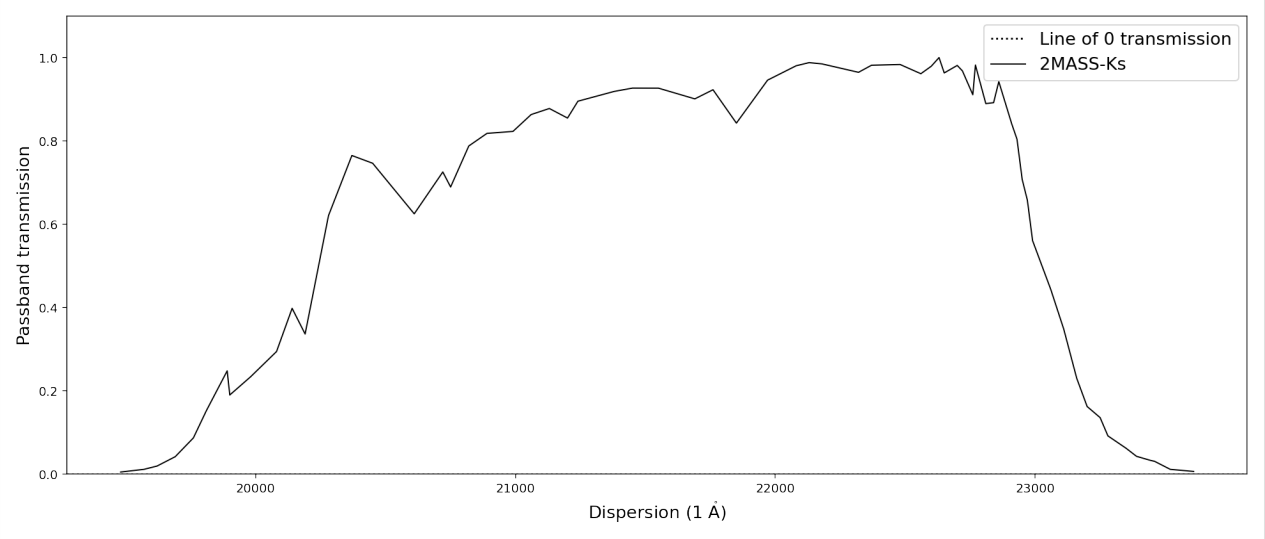
It is evident that the flux normalization of the three spectra does not agree with another. For a first test we will now scale the spectra according to their broad band magnitudes.

6.2.1 Absolute flux calibration of the near-infrared spectrum

We will start with the near-infrared spectrum, for which 2MASS magnitudes are available. In AB the magnitudes are J=16.13, H=16.14, K=16.01. We will begin by loading the 2MASS passbands.

(A range of passbands are included already with the Sculptor package in sculptor/data/passbands. The PassBand class behaves very similar to the SpecOneD class, but is mainly used for calculating magnitudes from spectra. The passbands can also be plotted like SpecOneD objects, e.g. 'pb_Ks.plot()')

[4]:
```
pb_J = sod.PassBand(passband_name='2MASS-J')
pb_H = sod.PassBand(passband_name='2MASS-H')
pb_Ks = sod.PassBand(passband_name='2MASS-Ks')

pb_Ks.plot()
```



As a test, let us calculate the magnitudes of the spectrum in the three 2MASS filter before flux normalization.

[5]:
```
J_ab_before = specnir.calculate_passband_ab_magnitude(pb_J)
H_ab_before = specnir.calculate_passband_ab_magnitude(pb_H)
Ks_ab_before = specnir.calculate_passband_ab_magnitude(pb_Ks)
print('J_AB = {:.2f}'.format(J_ab_before))
print('H_AB = {:.2f}'.format(H_ab_before))
print('Ks_AB = {:.2f}'.format(Ks_ab_before))
```

```
J_AB = 16.76
H_AB = 16.79
Ks_AB = 16.89
```

We choose the K-band magnitude for the flux normalization as the overlap region between the band and the spectrum is least affected by telluric features. Then we recalculate the magnitudes of the spectrum in the three 2MASS filter bands.

[6]:
```
Ks_ab = 16.01
J_ab = 16.13
H_ab = 16.14
nspecnir = specnir.renormalize_by_ab_magnitude(Ks_ab, pb_Ks)

J_ab_after = nspecnir.calculate_passband_ab_magnitude(pb_J)
H_ab_after = nspecnir.calculate_passband_ab_magnitude(pb_H)
Ks_ab_after = nspecnir.calculate_passband_ab_magnitude(pb_Ks)
print('J_AB = {:.2f}'.format(J_ab_after))
print('H_AB = {:.2f}'.format(H_ab_after))
print('Ks_AB = {:.2f}'.format(Ks_ab_after))
```

```
J_AB = 15.89
H_AB = 15.92
Ks_AB = 16.01
```

Let us compare the magnitude differences between the J-, H-band magnitudes from before and after the normalization. The choice of the K-band is furthermore motivated by a comparison of the near-infrared with the optical spectra after all have been normalized to broad band fluxes (see below).

[7]:
```
print('Absolute J-band differences before and after normalization: {:.2f} mag vs {:.2f}␣
↪mag'.format(
    abs(J_ab_before-J_ab), abs(J_ab_after-J_ab)))
print('Absolute H-band differences before and after normalization: {:.2f} mag vs {:.2f}␣
↪mag'.format(
    abs(H_ab_before-H_ab), abs(H_ab_after-H_ab)))
print('Absolute Ks-band differences before and after normalization: {:.2f} mag vs {:.2f}␣
↪mag'.format(
    abs(Ks_ab_before-Ks_ab), abs(Ks_ab_after-Ks_ab)))
```

```
Absolute J-band differences before and after normalization: 0.63 mag vs 0.24 mag
Absolute H-band differences before and after normalization: 0.65 mag vs 0.22 mag
Absolute Ks-band differences before and after normalization: 0.88 mag vs 0.00 mag
```

While the K-band flux normalization leaves some residuals in the J- and H-bands, they are much closer to their 2MASS values than before.

6.2.2 Absolute flux calibration of the optical spectra

We continue with the optical spectra, starting with the green spectrum (specoptA) and use the Pan-STARRS r-band magnitude (r=16.50, AB) for normalization.

[8]:
```
r_ab = 16.50
pb_r = sod.PassBand(passband_name='PS1-r')
nspecoptA = specoptA.renormalize_by_ab_magnitude(r_ab, pb_r)
```

The orange spectrum (specoptB) starts and ends at larger wavelengths. Therefore, we use the Pan-STARRS i-band magnitude (i=16.42, AB) for normalization.

[9]:
```
i_ab = 16.50
pb_i = sod.PassBand(passband_name='PS1-i')
nspecoptB = specoptB.renormalize_by_ab_magnitude(i_ab, pb_i)
```

6.2.3 Flux calibrated spectra

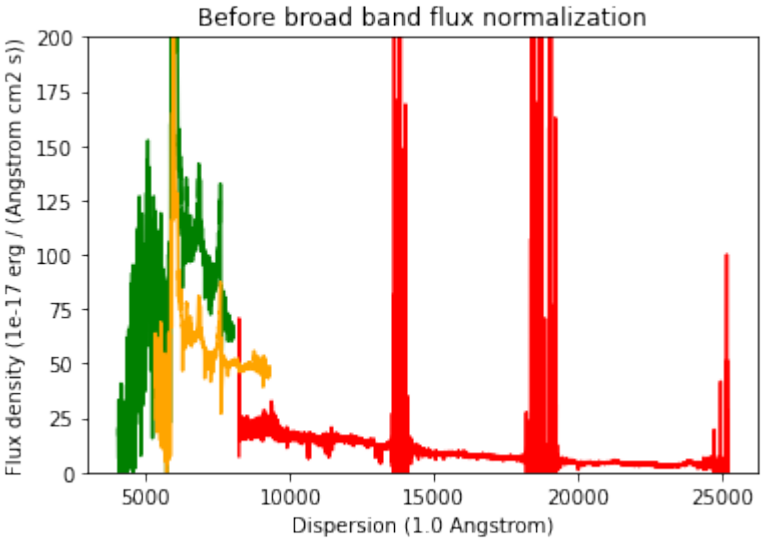
Below we display the spectra normalized by their broad band r, i, and Ks photometry. A comparison to the previous figure displays that the broad band flux normalization has been successful.

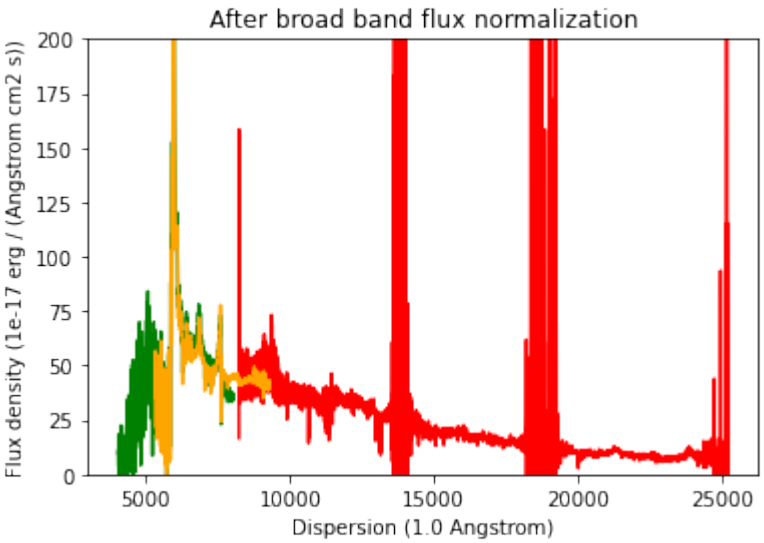
[10]:
```
import matplotlib.pyplot as plt

plt.plot(specnir.dispersion, specnir.fluxden, 'red')
plt.plot(specoptA.dispersion, specoptA.fluxden, 'green')
plt.plot(specoptB.dispersion, specoptB.fluxden, 'orange')
plt.ylim(0, 200)
plt.xlabel('Dispersion ({})'.format(specnir.dispersion_unit))
plt.ylabel('Flux density ({})'.format(specnir.fluxden_unit))
plt.title('Before broad band flux normalization')
plt.show()

plt.plot(nspecnir.dispersion, nspecnir.fluxden, 'red')
plt.plot(nspecoptA.dispersion, nspecoptA.fluxden, 'green')
```

(continues on next page)

(continued from previous page)

```
plt.plot(nspecoptB.dispersion, nspecoptB.fluxden, 'orange')
plt.ylim(0, 200)
plt.xlabel('Dispersion ({})'.format(nspecnir.dispersion_unit))
plt.ylabel('Flux density ({})'.format(nspecnir.fluxden_unit))
plt.title('After broad band flux normalization')
plt.show()
```





6.3 Building a composite spectrum

In this step we will "stitch" the spectra together using the normalized K-band spectrum as a reference. At first we renomalize the flux level of the optical spectrum B to the broad band normalized near-infrared spectrum in their overlap wavelength range 8500-9000 A.

[11]:
```
nspecoptB = specoptB.renormalize_by_spectrum(nspecnir, dispersion_limits=[8500, 9000])
```

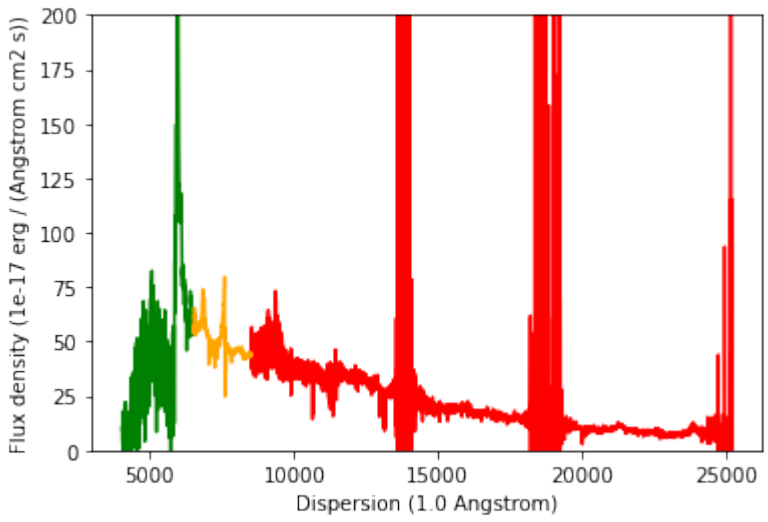
Then we normalize the optical spectrum A to the normalized optical spectrum B in their overlap wavelength range 6000-7000 A.

[12]:
```
nspecoptA = specoptA.renormalize_by_spectrum(nspecoptB, dispersion_limits=[6000, 7000])
```

Now that the spectra have the same flux level in their overlap regions, we trim their dispersion axes before we stitch them together.

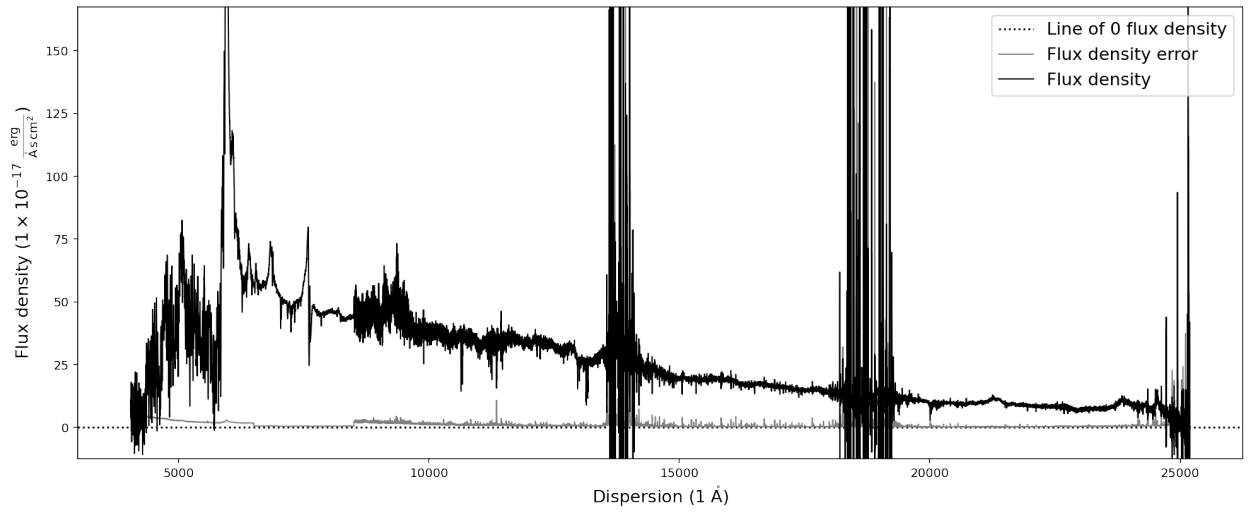
[13]:
```
nspecoptA = nspecoptA.trim_dispersion([0,6500])
nspecoptB = nspecoptB.trim_dispersion([6500,8500])
nspecnir = nspecnir.trim_dispersion([8500,400000])

# Lets. plot them to have a quick look.
plt.plot(nspecnir.dispersion, nspecnir.fluxden, 'red')
plt.plot(nspecoptA.dispersion, nspecoptA.fluxden, 'green')
plt.plot(nspecoptB.dispersion, nspecoptB.fluxden, 'orange')
plt.ylim(0, 200)
plt.xlabel('Dispersion ({})'.format(nspecnir.dispersion_unit))
plt.ylabel('Flux density ({})'.format(nspecnir.fluxden_unit))
plt.show()
```

```
[WARNING] Lower limit is below the lowest dispersion value. The lower limit is set to␣
↪the minimum dispersion value.
[WARNING] Upper limit is above the highest dispersion value. The upper limit is set to␣
↪the maximum dispersion value.
```



This doesn't look too bad. Now we build the composite spectrum by appending all dispersion, flux density, and flux density error arrays. This is done manually. Possibly, future versions of Sculptor may be able to automatize this process.

[14]:
```
import numpy as np

# Building the dispersion array
comp_dispersion = nspecoptA.dispersion
comp_dispersion = np.append(comp_dispersion, nspecoptB.dispersion)
comp_dispersion = np.append(comp_dispersion, nspecnir.dispersion)



# Building the flux density array
comp_fluxden = nspecoptA.fluxden
comp_fluxden = np.append(comp_fluxden, nspecoptB.fluxden)
comp_fluxden = np.append(comp_fluxden, nspecnir.fluxden)

# Building the flux density error array
comp_fluxden_err = nspecoptA.fluxden_err
comp_fluxden_err = np.append(comp_fluxden_err, nspecoptB.fluxden_err)
comp_fluxden_err = np.append(comp_fluxden_err, nspecnir.fluxden_err)

# Initialize a new SpecOneD object for the composite
composite = sod.SpecOneD(dispersion=comp_dispersion,
                         fluxden=comp_fluxden,
                         fluxden_err=comp_fluxden_err,
                          # We now need to specify the physical units
                          # of the dispersion and flux density axis.
                          # Here, we simply copy them from the near-infrared
                          # spectrum.
                         dispersion_unit=nspecnir.dispersion_unit,
                         fluxden_unit=nspecnir.fluxden_unit)

# Let's plot our new composite spectrum
composite.plot(show_fluxden_err=True)
```



6.4 Saving the final composite spectrum

As the final step of this example we save the dereddened composite spectrum to a SpecOneD hdf5 file for use with Sculptor. The relative path below will save it in the example spectra within the sculptor data folder.

[15]:
```
# Save a version of the spectrum in the example spectra folder
composite.save_to_hdf('../../sculptor/data/example_spectra/J2125-1719_composite.hdf')
```

CHAPTER

SEVEN

SCRIPTING SCULPTOR 01 - MODELLING THE EXAMPLE SPECTRUM IN A SCRIPT

In this notebook we will introduce on how to use Sculptor's SpecFit and SpecModel classes to generate a Sculptor fit within a python script.

The code in this example that generates the model fit is also available as a pure python script in the examples folder: *example_fit_setup_script.py*

We begin by importing the modules we need for this example. In addition to the SpecFit, SpecModel, and SpecOneD classes we need *pkg_resources* to easily access the SDSS quasar example spectrum, which is provided in sculptor/data/example_spectra.

[1]:
```
# Matplotlib statements to make plots nice
# %matplotlib notebook
from IPython.display import set_matplotlib_formats
import matplotlib.pyplot as plt
plt.rcParams['figure.dpi'] = 300
plt.rcParams['savefig.dpi'] = 300
# set_matplotlib_formats('svg')
```

[2]:
```
import pkg_resources

from sculptor import specfit as scfit
from sculptor import speconed as scspec
from sculptor import specmodel as scmod
```

```
[INFO] Import "sculptor_extensions" package: my_extension
[INFO] Import "sculptor_extensions" package: qso
[INFO] SWIRE library found.
[INFO] FeII iron template of Vestergaard & Wilkes 2001 found. If you will be using these␣
↪templates in your model fit and publication, please add the citation to the original␣
↪work, ADS bibcode: 2001ApJS..134...1V
[INFO] FeII iron template of Tsuzuki et al. 2006 found. If you will be using these␣
↪templates in your model fit and publication, please add the citation to the original␣
↪work, ADS bibcode: 2006ApJ...650...57T
[INFO] FeII iron template of Boroson & Green 1992 found. If you will be using these␣
↪templates in your model fit and publication, please add the citation to the original␣
↪work, ADS bibcode: 1992ApJS...80..109B
```

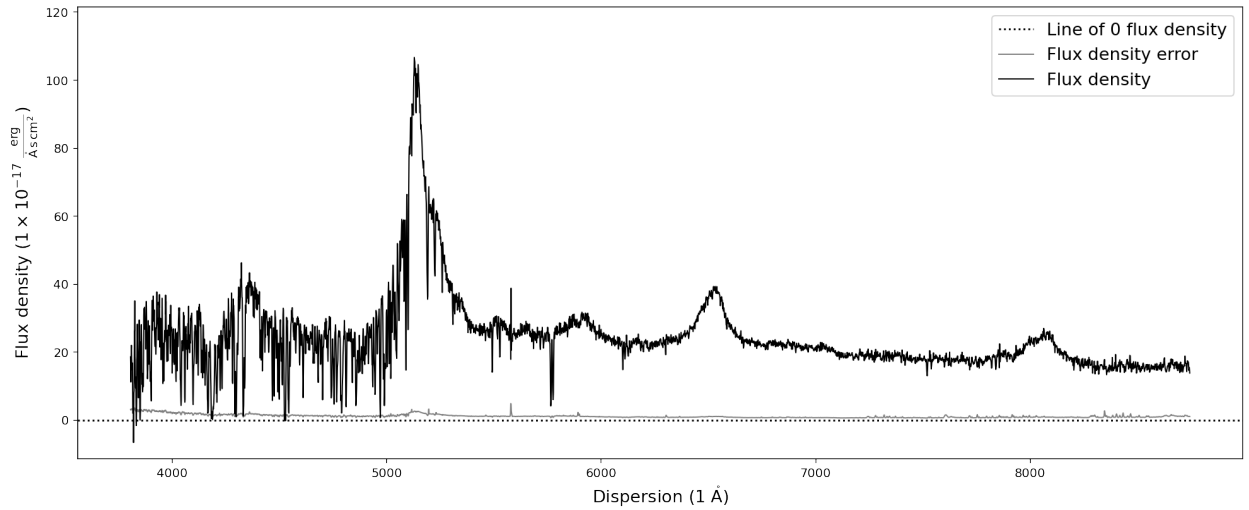
These imports automatically intialize some global variables, which allow you to use the model masks and functions defined in Sculptor and its extensions. We will take a look at how to write Sculptor extensions in a later tutorial. Sculptor comes with two extensions "my_extension", which provides an example on how to set up your own extension,

and the "qso" extension, which we will use heavily in this example. The "qso" extension also comes with additional data, e.g. FeII iron templates. If you want to use them in your work Sculptor reminds you where to find the original papers, so you can cite them appropriately.

7.1 Initializing the SpecFit object

Let us begin by importing the SDSS quasar spectrum. The quasar is at a redshift of z=3.227, which we will use for initializing the SpecFit class.

[3]:
```
# Initialize a new SpecOneD object and read in the quasar spectrum
spec = scspec.SpecOneD()
filename = pkg_resources.resource_filename('sculptor', 'data/example_spectra/J030341.04-
↪002321.8_0.fits')
spec.read_sdss_fits(filename)

redshift = 3.227

# Plot the quasar spectrum for confirmation that everything worked well.
spec.plot()
```

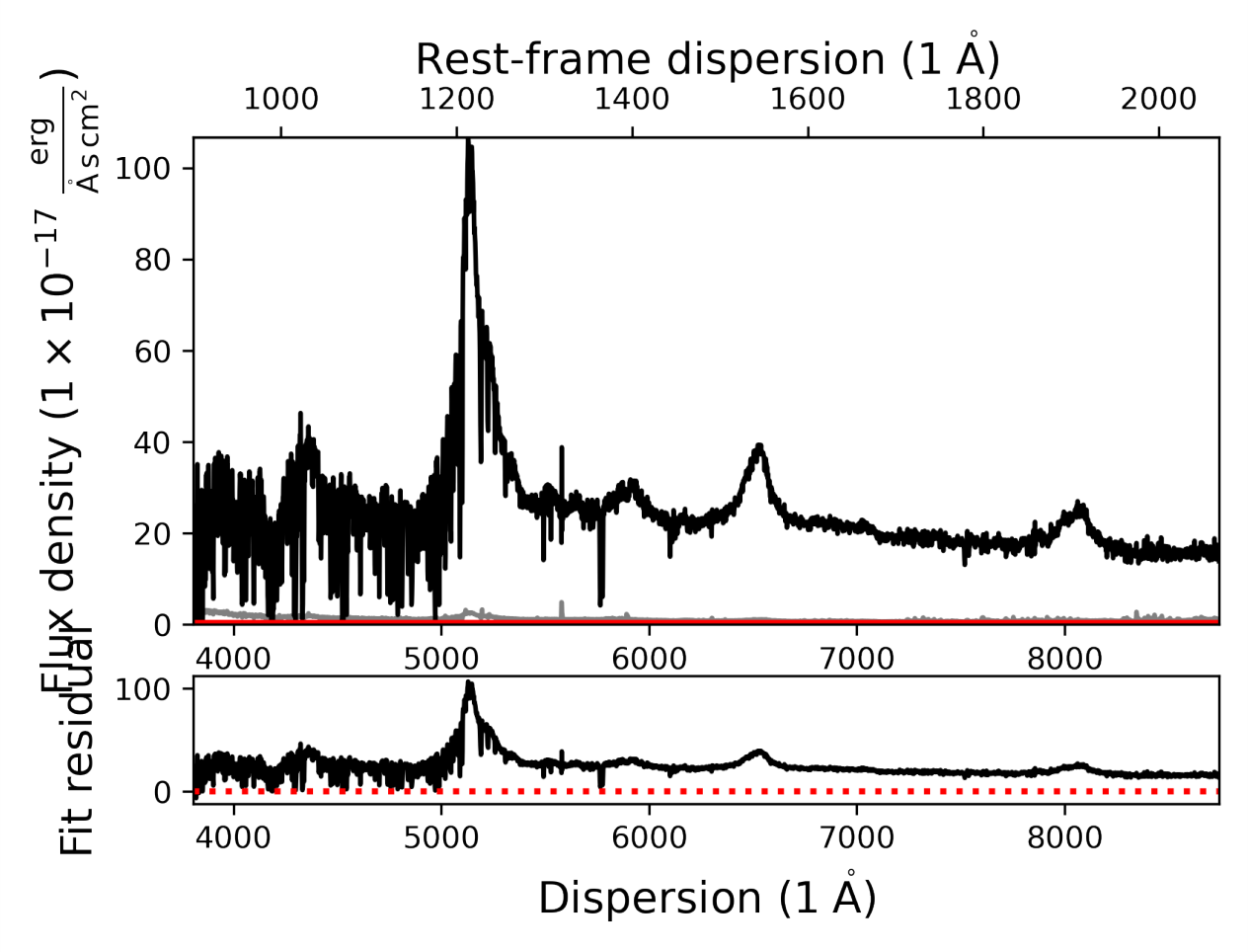


In the next step we initialize the SpecFit object with the SpecOneD spectrum and also supply the redshift information.

[4]:
```
# Initialize SpecFit object
fit = scfit.SpecFit(spec, redshift)
```

The SpecFit object currently does not include any models and if we use the *.plot()* method, it will only show us the quasar spectrum. However, we can check the rest-frame dispersion axis, if the redshift keyword was used correctly.

[5]:
```
fit.plot()
```



7.2 Adding our first SpecModel object - Fitting the continuum

In the next step we want to add a simple continuum model to our fit. However, we currently don't really know which models and masks are available to us based on Sculptor and its loaded extensions. Let's get a list of all model names and mask names. This information is available as a global variable in the sculptor.specmodel module:

[6]:
```
print('MODEL FUNCTIONS:')
for model_func in scmod.model_func_list:
    print (model_func)
print('\n')
print('MASKS:')
for mask in scmod.mask_presets:
    print(mask)
```

```
MODEL FUNCTIONS:
Constant (amp)
Power Law (amp, slope)
Gaussian (amp, cen, sigma, shift)
Lorentzian (amp, cen, gamma, shift)
My Model
```

(continues on next page)

(continued from previous page)

```
Power Law (2500A)
Power Law (2500A) + BC
Power Law (2500A) + BC (fractional)
Line model Gaussian
SiIV (2G components)
CIV (2G components)
MgII (2G components)
HBeta (2G components)
HAlpha (2G components)
[OIII] doublet (2G)
[NII] doublet (2G)
[SII] doublet (2G)
CIII] complex (3G components)
SWIRE Ell2
SWIRE NGC6090
FeII template 1200-2200 (VW01, cont)
FeII template 1200-2200 (VW01, split)
FeII template 2200-3500 (VW01, cont)
FeII template 2200-3500 (VW01, split)
FeII template 2200-3500 (T06, cont)
FeII template 2200-3500 (T06, split)
FeII template 3700-7480 (BG92, cont)
FeII template 3700-5600 (BG92, split)


MASKS:
My mask
QSO Continuum+FeII
QSO Cont.W. VP06
QSO Fe+Cont.W. CIV Shen11
QSO Fe+Cont.W. MgII Shen11
QSO Fe+Cont.W. HBeta Shen11
QSO Fe+Cont.W. HAlpha Shen11
```

Using these models when writing Sculptor scripts usually requires to fully understand their parameters and functionality. Therefore, it is adviseable to study their source code, before attempting to write scripts.

In order to fit a model to the spectrum, we need to first add a SpecModel object to our fit (SpecFit object). Then we can access this SpecModel and add model functions to it. In order to fit the SpecModel

[7]:
```
# Add the continuum SpecModel
fit.add_specmodel()
# We access the initialized SpecModel object by using the first item in the SpecFit.
↪specmodels list
contmodel = fit.specmodels[0]
# Let us rename this SpecModel 'Continuum'
contmodel.name = 'Continuum'

# Define the model function name
model_name = 'Power Law (2500A)'
# Define the model prefix
# It is important to keep track of the prefix to later access this model in the analysis
model_prefix = 'PL_'
```

(continues on next page)

(continued from previous page)

```
# Add the model function to the SpecModel
contmodel.add_model(model_name, model_prefix)
```

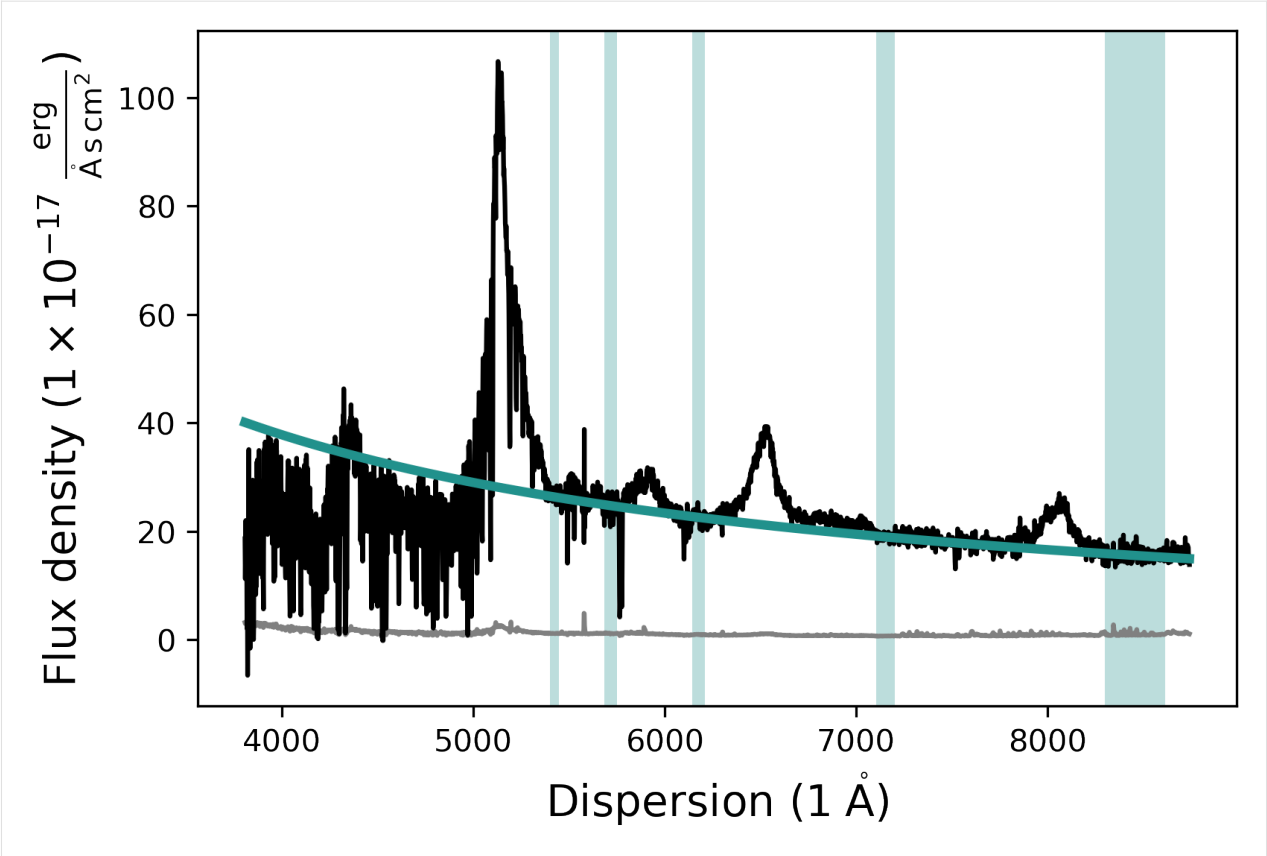
We have now added the model function to the SpecModel 'Continuum'. However, we cannot yet fit the model successfully as we have not defined the regions to which the continuum model should be fit. Let us add them manually based. The exact fit regions for the continuum model of this quasar have been determined beforehand.

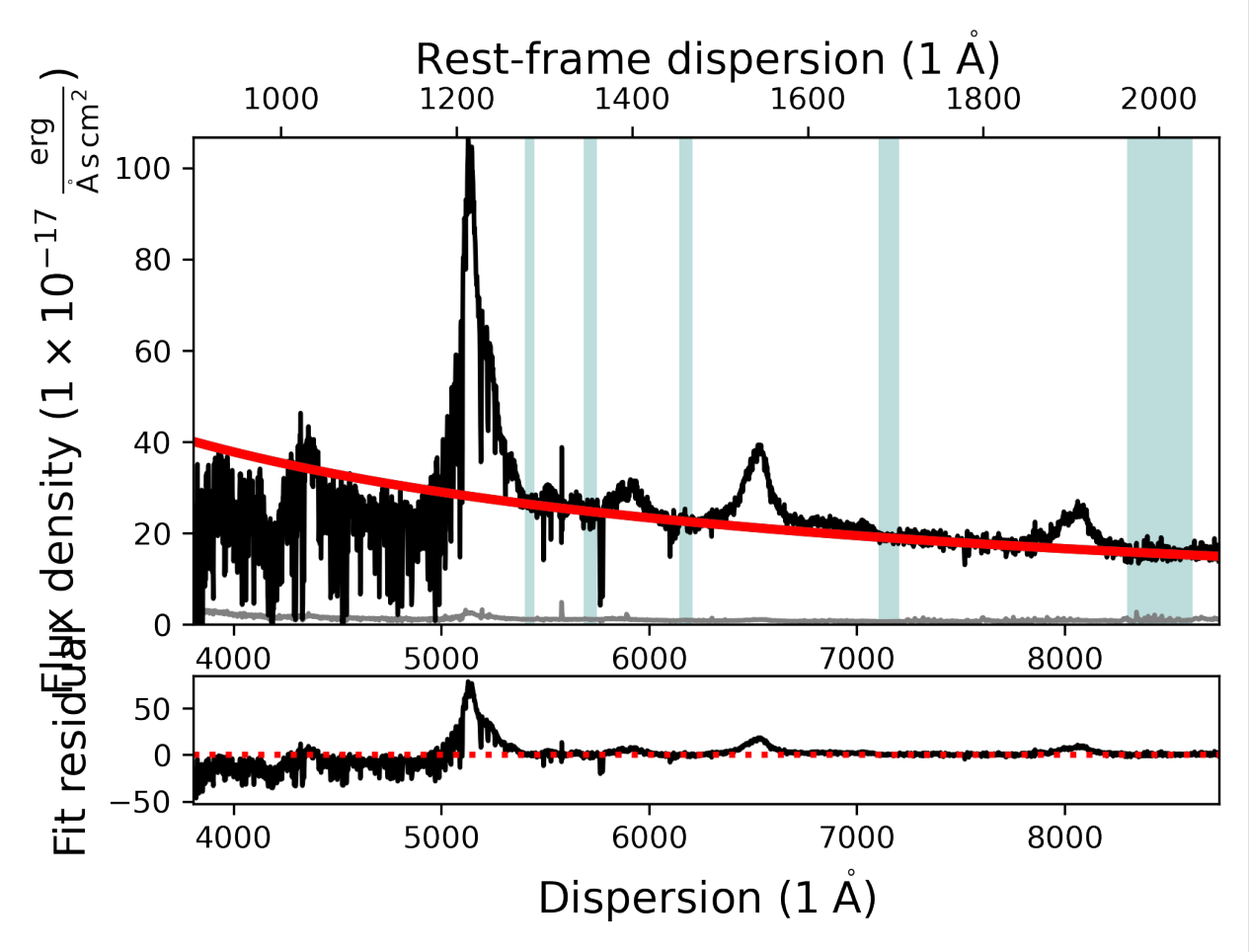
[8]:
```
contmodel.add_wavelength_range_to_fit_mask(8300, 8620)
contmodel.add_wavelength_range_to_fit_mask(7105, 7205)
contmodel.add_wavelength_range_to_fit_mask(5400, 5450)
contmodel.add_wavelength_range_to_fit_mask(5685, 5750)
contmodel.add_wavelength_range_to_fit_mask(6145, 6212)
```

```
[INFO] Manual mask range 8300 8620
[INFO] Manual mask range 7105 7205
[INFO] Manual mask range 5400 5450
[INFO] Manual mask range 5685 5750
[INFO] Manual mask range 6145 6212
```

Now that the fit mask has been updated we can fit the continuum model and plot the resulting fit. We can either use the *contmodel.plot()* function to only show the SpecModel components or use the *fit.plot()* function to show all SpecModels fits.

[9]:
```
# Fit the continuum model
contmodel.fit()
# Plot the fitted continuum model and the spectrum
contmodel.plot()
# Plot the SpecFit fit with all SpecModels (We only have 1 at the moment)
fit.plot()
```


Flux density (1×10^{-17} $\frac{\text{erg}}{\text{Å s cm}^2}$)
100
80
60
40
20
0
4000
5000
6000
7000
8000
Dispersion (1 Å)



7.3 Adding and manipulating a SpecModel - Fitting the SiIV line

In our next step we will add an emission line model to fit the SiIV emission line at a rest-frame wavelength of ~1400A. To do this we start by adding another SpecModel to our fit and specify the wavelength regions (in observed-frame) we want to use for the fit.

[10]:
```
# Add the SiIV emission line model
fit.add_specmodel()
# Access the SpecModel object by choosing the second SpecModel object in the SpecFit
# specmodels list.
siiv_model = fit.specmodels[1]
# Rename the SpecModel
siiv_model.name = 'SiIV_line'

# Add wavelength regions to the SpecModel for the fit
siiv_model.add_wavelength_range_to_fit_mask(5790, 5870)
siiv_model.add_wavelength_range_to_fit_mask(5910, 6015)
```

```
[INFO] Manual mask range 5790 5870
[INFO] Manual mask range 5910 6015
```

However, we will use the pre-defined 'SiIV (2G components)' model function from the Sculptor qso extension, which has pre-defined model prefixes. Therefore, we pass a *None* as the model prefix here.

The model functions allow to pass additional keyword arguments that modify the redshift or amplitude of the emission line model. The redshift is automatically passed by the SpecModel class if we don't specify a different value here. In this case we only want to pass an amplitude.

[11]:
```
model_name = 'SiIV (2G components)'
model_prefix = None
siiv_model.add_model(model_name, model_prefix, amplitude=20)
```

The 'SiIV (2G components)' model function added two emission line models with the prefixes 'SiIV_A_' and 'SiIV_B_' to our SpecModel object. We can check whether the models were successfully added to the SpecModel object by accessing the SpecModel 'model_list'. The items in the model list are LMFIT *Model* objects.

[12]:
```
for model in siiv_model.model_list:
    print(model)
```

```
<lmfit.Model: Model(line_model_gaussian, prefix='SiIV_A_')>
<lmfit.Model: Model(line_model_gaussian, prefix='SiIV_B_')>
```

We see that two model functions of the type 'line_model_gaussian' have been added to the SiIV SpecModel. By default the model function holds the redshift parameter fixed during the fit. However, for our purposes we want the redshift to be a variable. To change this we need to access the parameters of the model function.

The parameters for each of the models in the 'model_list' are stored in the 'params_list'. Each item in the 'params_list' is a LMFIT *Parameters* object, holding the individual parameters of the associated model from the 'model_list'.

[13]:
```
for params in siiv_model.params_list:
    print(params)
    print('\n')
```

```
Parameters([('SiIV_A_z', <Parameter 'SiIV_A_z', value=3.227 (fixed), bounds=[3.
↪0656499999999998:3.38835]>), ('SiIV_A_flux', <Parameter 'SiIV_A_flux', value=106446.
↪7019431226, bounds=[0:106446701.94312261]>), ('SiIV_A_cen', <Parameter 'SiIV_A_cen',␣
↪value=1399.8 (fixed), bounds=[-inf:inf]>), ('SiIV_A_fwhm_km_s', <Parameter 'SiIV_A_
↪fwhm_km_s', value=2500, bounds=[300:20000]>)])


Parameters([('SiIV_B_z', <Parameter 'SiIV_B_z', value=3.227 (fixed), bounds=[3.
↪0656499999999998:3.38835]>), ('SiIV_B_flux', <Parameter 'SiIV_B_flux', value=106446.
↪7019431226, bounds=[0:106446701.94312261]>), ('SiIV_B_cen', <Parameter 'SiIV_B_cen',␣
↪value=1399.8 (fixed), bounds=[-inf:inf]>), ('SiIV_B_fwhm_km_s', <Parameter 'SiIV_B_
↪fwhm_km_s', value=2500, bounds=[300:20000]>)])
```

To access a single LMFIT *Parameter* object from the *Parameters* we need to use its name, which consists of the model prefix and the parameter name, which are shown above.

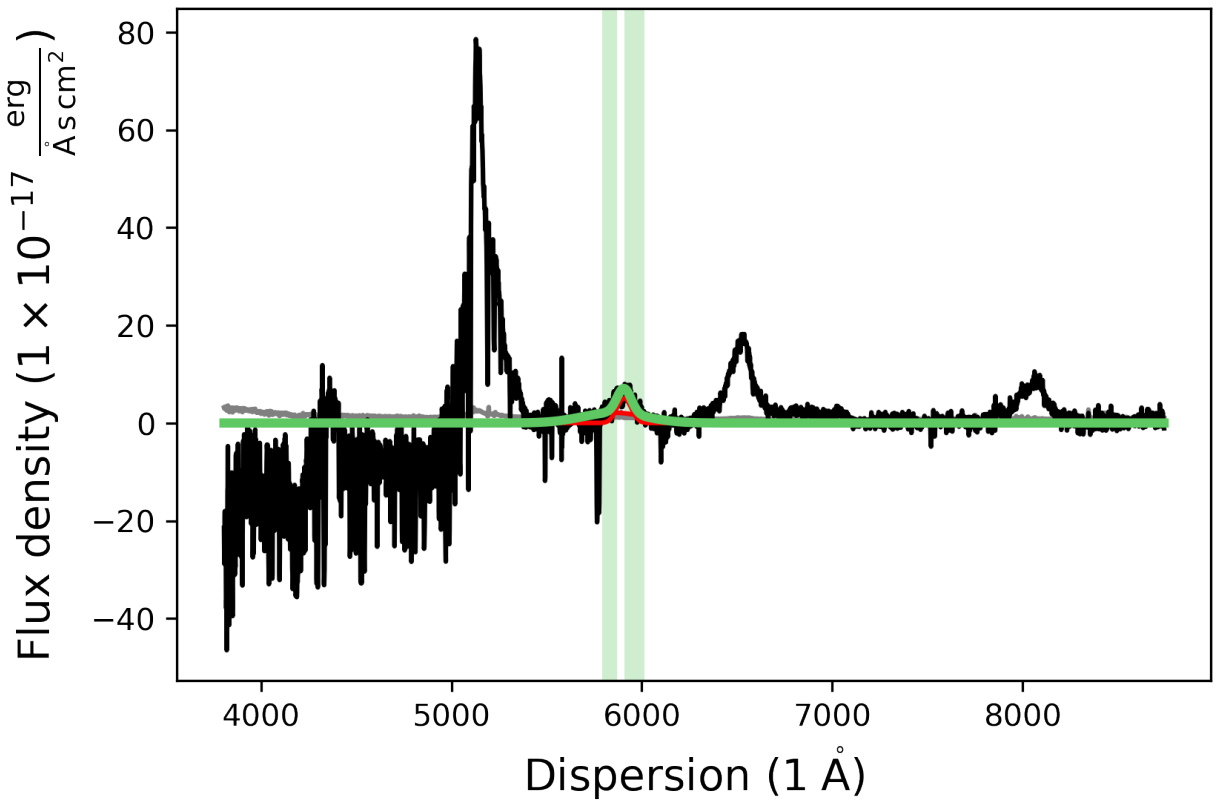
By accessing a specific parameters we can change its attributes, defined in the LMFIT documentation: * name (str) – Name of the Parameter. * value (float, optional) – Numerical Parameter value * vary (bool, optional) – Whether the Parameter is varied during a fit (default is True). * min (float, optional) – Lower bound for value (default is -numpy.inf, no lower bound). * max (float, optional) – Upper bound for value (default is numpy.inf, no upper bound). * expr (str, optional) – Mathematical expression used to constrain the value during the fit (default is None).

We will now change the *vary* attribute of the redshift parameters 'SiIV_A_z' of model function 0, and 'SiIV_B_z' of model function 1 to *True*.

[14]:
```
# Make the redshifts variable parameters
params = siiv_model.params_list[0]
params['SiIV_A_z'].vary = True
params = siiv_model.params_list[1]
params['SiIV_B_z'].vary = True
```

Then we fit the SiIV SpecModel.

[15]:
```
# Fit the SiIV SpecModel
siiv_model.fit()
# Display the fitted SpecModel
siiv_model.plot()
```



7.4 Fitting the CIV line

In the next step we add the CIV model, basically repeating the same procedure.

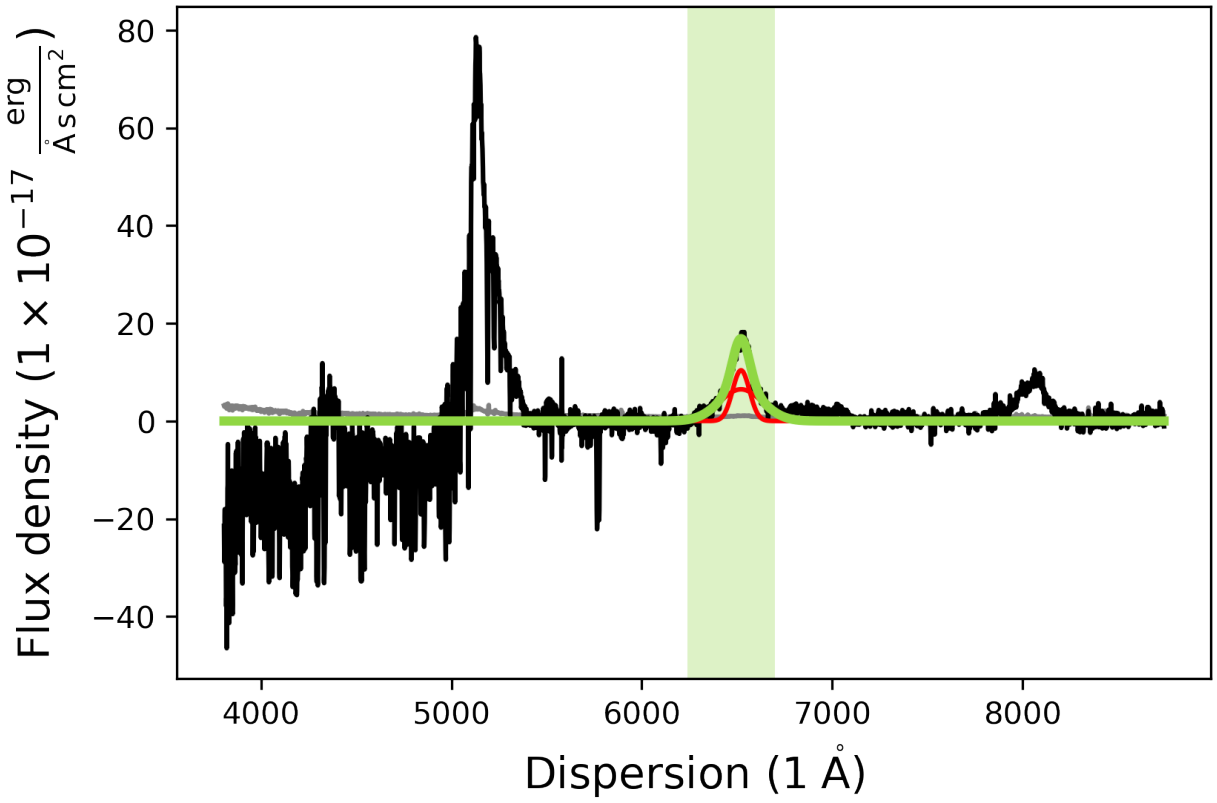
[16]:
```
# Add the CIV emission line model
fit.add_specmodel()
civ_model = fit.specmodels[2]
civ_model.name = 'CIV_line'

civ_model.add_wavelength_range_to_fit_mask(6240, 6700)
```

(continues on next page)

(continued from previous page)

```
model_name = 'CIV (2G components)'
model_prefix = None
civ_model.add_model(model_name, model_prefix, amplitude=10)

# Make the redshift a variable parameter
params = civ_model.params_list[0]
params['CIV_A_z'].vary = True
params = civ_model.params_list[1]
params['CIV_B_z'].vary = True

civ_model.fit()

civ_model.plot()
```

```
[INFO] Manual mask range 6240 6700
```

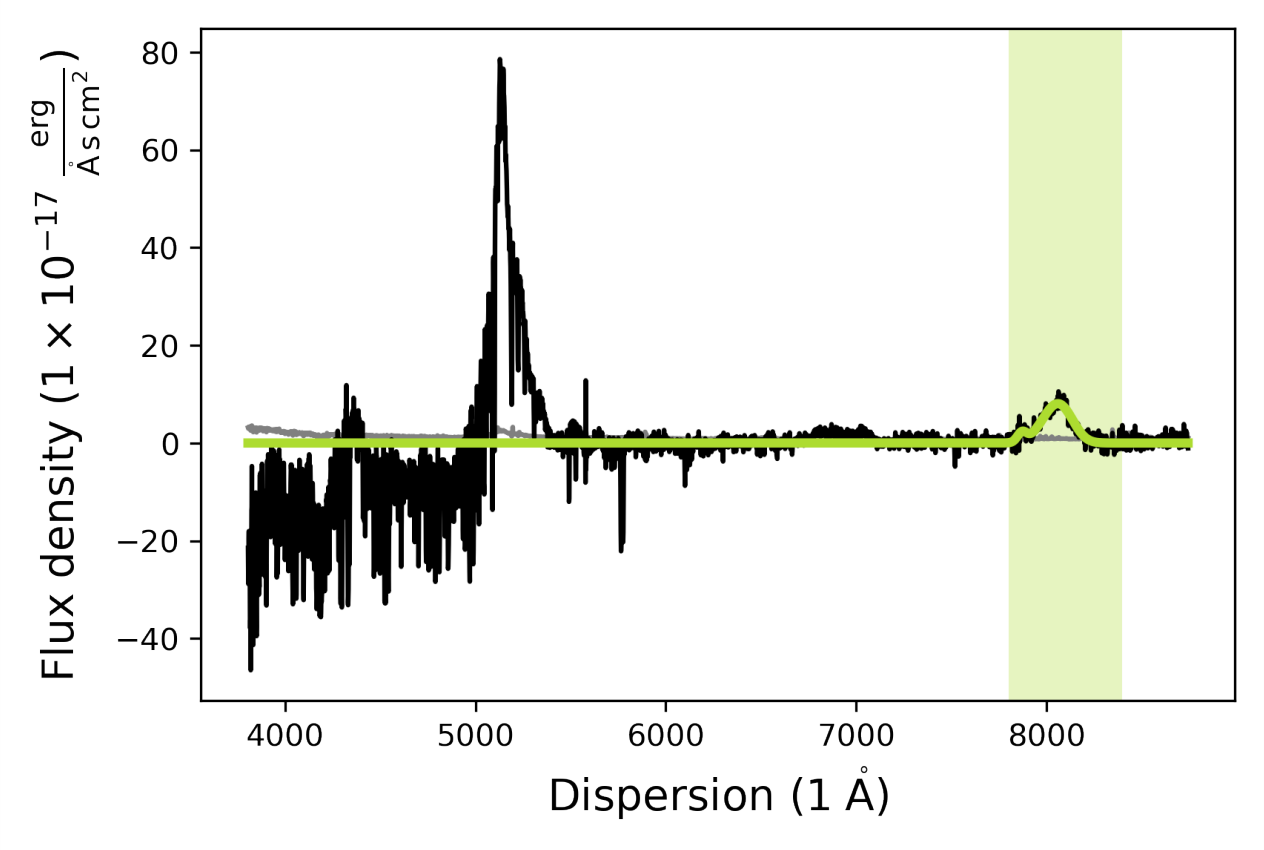


7.5 Fitting the CIII] line

After we successfully added the CIV line to the fit, we will now add a model for the SiIII], AlIII, and CIII] lines. While the previous SiIV and CIV model functions were comprised of two individual model functions called 'line_model_gaussian', the CIII] complex model is one model function comprised of three Gaussians with set central wavelength values.

[17]:
```
# Add the CIII] complex emission line model
fit.add_specmodel()
ciii_model = fit.specmodels[3]
ciii_model.name = 'CIII]_complex'

ciii_model.add_wavelength_range_to_fit_mask(7800, 8400)

model_name = 'CIII] complex (3G components)'
model_prefix = None
ciii_model.add_model(model_name, model_prefix, amplitude=2)

params = ciii_model.params_list[0]
params['CIII_z'].vary = True

# Initial fit
ciii_model.fit()
# Second fit to make sure the model converged
ciii_model.fit()

# Plot the model
ciii_model.plot()
```

```
[INFO] Manual mask range 7800 8400
```



7.6 Adding a basic line model (Gaussian) - Fitting absorption lines

Just blueward of the SiIV line this quasar has two prominent absorption features, which we also want to model. For this purpose we use the basic 'Line model Gaussian' model function and provide the parameters values as keyword arguments ourselves.

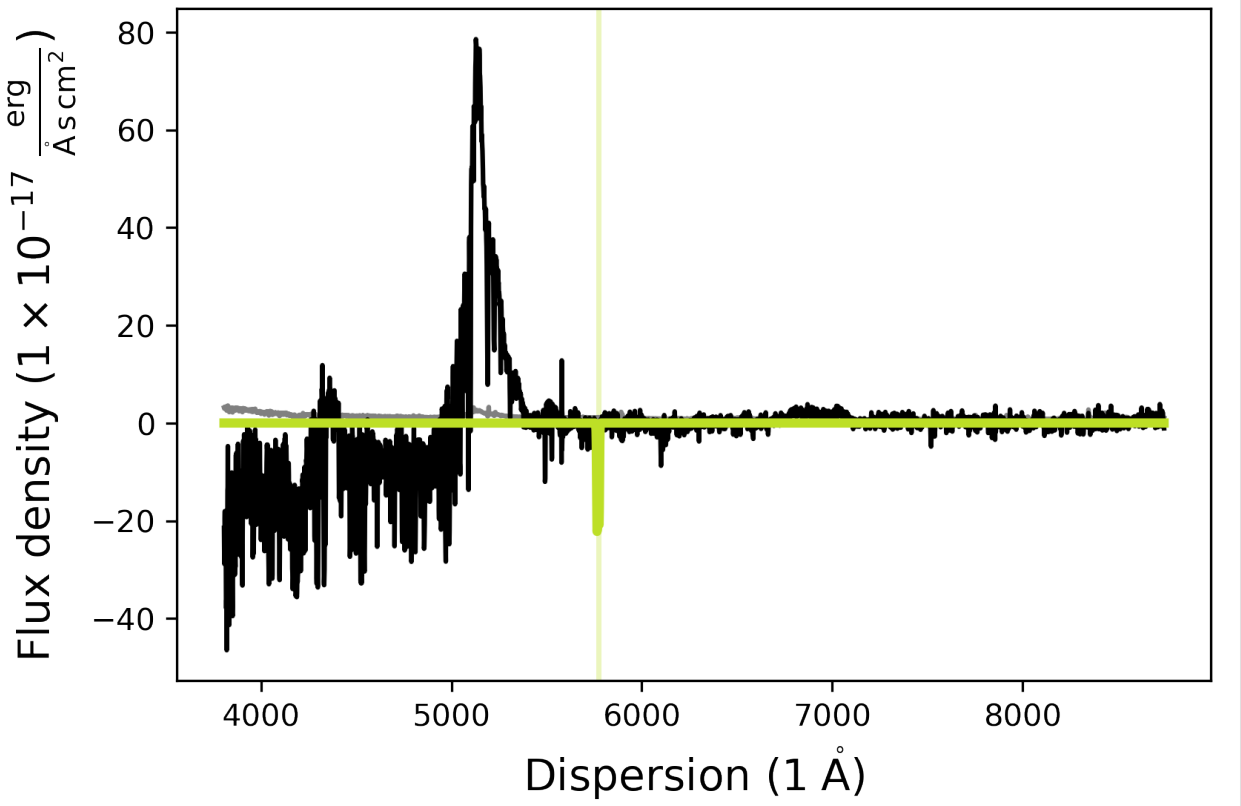
[18]:
```
# Add absorption line models
fit.add_specmodel()
abs_model = fit.specmodels[4]
abs_model.name = 'Abs_lines'

abs_model.add_wavelength_range_to_fit_mask(5760, 5790)

model_name = 'Line model Gaussian'
model_prefix = 'Abs_A'
abs_model.add_model(model_name, model_prefix, amplitude=-15,
                    cenwave=5766, fwhm=200, redshift=0)
model_name = 'Line model Gaussian'
model_prefix = 'Abs_B'
abs_model.add_model(model_name, model_prefix, amplitude=-15,
                    cenwave=5776, fwhm=200, redshift=0)

abs_model.fit()
```

(continues on next page)

(continued from previous page)

```
abs_model.plot()
```

```
[INFO] Manual mask range 5760 5790
```

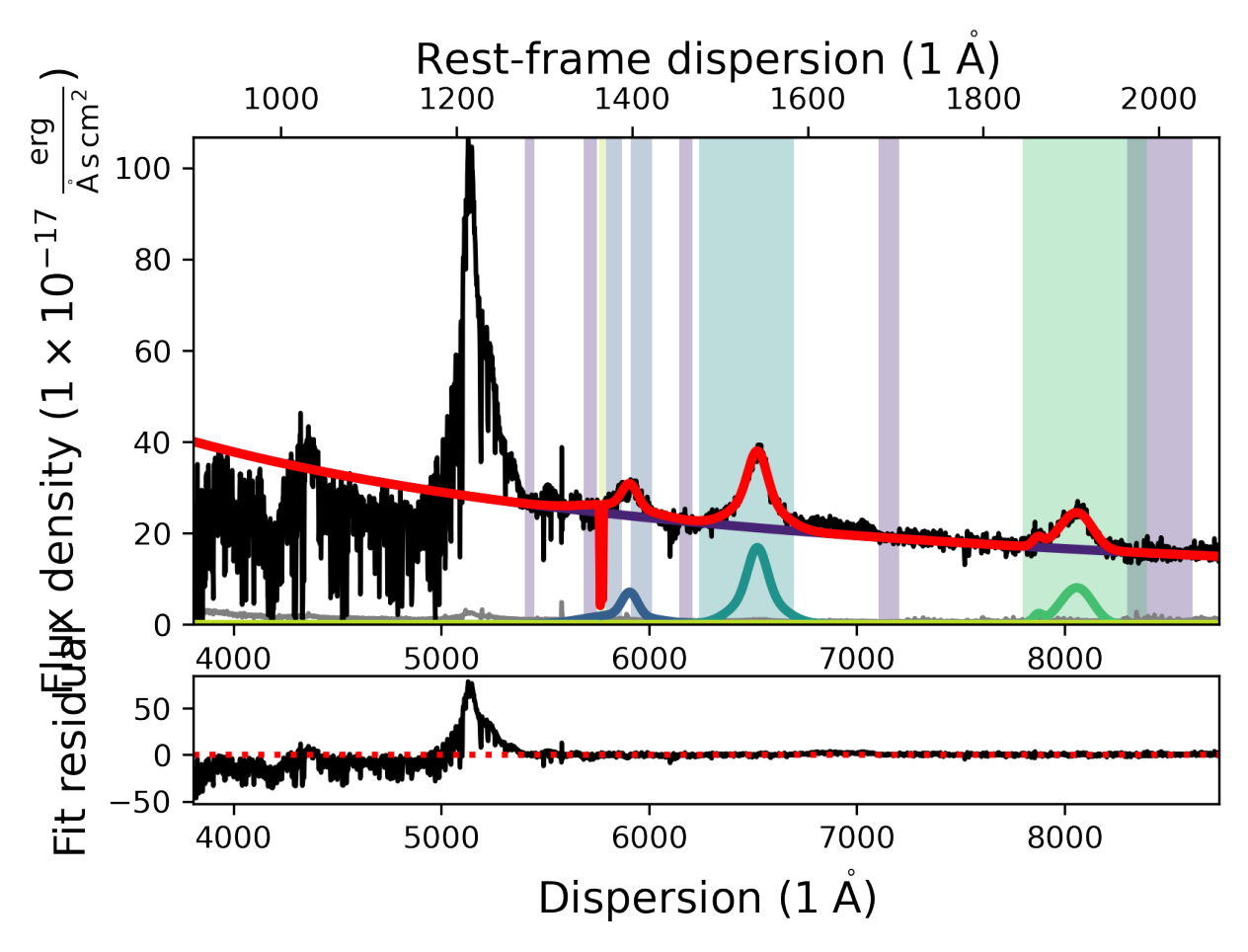


The default *.plot()* function dispersion and flux ranges may not always be appropriate to check whether the model was actually fit appropriately. Using the *matplotlib notebook* functionality one can zoom into the region of the two absorption lines to check whether the SpecModel fit did a good job.

7.7 Visualizing the full quasar model fit

Let us now visualize the entire fit by calling the SpecFit plot function:

[19]:
```
fit.plot()
```



7.8 Accessing SpecModel fit results and saving them

The fitted quasar model looks reasonable. However, in order to analyze the model we need to understand how to access the fitted parameters. Once a SpecModel has been fit, we can access the fit results simply by calling the *.fit_result* attribute of the SpecModel. It returns the LMFIT fit report for the SpecModel, giving us insight into the model that was fit, the fit statistics, the variables, and the correlations.

[20]:
```
# Printing the fit result for the CIV model
civ_model.fit_result
```

[20]:
```
<lmfit.model.ModelResult at 0x1842bc280>
```

We can save the fit result of each individual SpecModel using the *save_fit_report* function. One has to supply the folder path where the fit report should be saved. As an example we can save the fit report for the CIV SpecModel to the current folder.

[21]:
```
# Save the CIV SpecModel fit report to the current folder
civ_model.save_fit_report('.')

# Check if the fit report was saved
! ls
```

```
TestSpectralBroadening.ipynb      specmodel_CIV_line_fit_report.txt
scripting_sculptor_1.ipynb        speconed_demonstration.ipynb
scripting_sculptor_2.ipynb        spectrum_preparation.ipynb
scripting_sculptor_3.ipynb
```

Let's have a brief look at what was saved exactly:

[22]:
```
h = open("specmodel_CIV_line_fit_report.txt", "r")
for line in h:
    print (line)
h.close()
```

```
[[Fit Statistics]]

    # fitting method   = leastsq

    # function evals   = 121

    # data points      = 309

    # variables        = 6

    chi-square         = 431.940013

    reduced chi-square = 1.42554460

    Akaike info crit   = 115.498142

    Bayesian info crit = 137.898189

[[Variables]]

    CIV_A_z:          3.20972764 +/- 0.00149858 (0.05%) (init = 3.227)

    CIV_A_flux:       1820.82409 +/- 100.100791 (5.50%) (init = 100)

    CIV_A_cen:        1549.06 (fixed)

    CIV_A_fwhm_km_s:  12122.2834 +/- 611.964256 (5.05%) (init = 2500)

    CIV_B_z:          3.20984418 +/- 7.2015e-04 (0.02%) (init = 3.227)

    CIV_B_flux:       1151.33782 +/- 114.160367 (9.92%) (init = 100)

    CIV_B_cen:        1549.06 (fixed)

    CIV_B_fwhm_km_s:  4791.00832 +/- 222.068758 (4.64%) (init = 2500)

[[Correlations]] (unreported correlations are < 0.100)

    C(CIV_A_flux, CIV_B_flux)           = -0.957

    C(CIV_A_fwhm_km_s, CIV_B_flux)      =  0.913
```

(continues on next page)

(continued from previous page)

```
    C(CIV_B_flux, CIV_B_fwhm_km_s)      =  0.910

    C(CIV_A_flux, CIV_B_fwhm_km_s)      = -0.907

    C(CIV_A_flux, CIV_A_fwhm_km_s)      = -0.820

    C(CIV_A_fwhm_km_s, CIV_B_fwhm_km_s) =  0.781

    C(CIV_A_z, CIV_B_z)                 = -0.557

    C(CIV_A_z, CIV_A_fwhm_km_s)         =  0.241

    C(CIV_A_z, CIV_B_flux)              =  0.235

    C(CIV_A_z, CIV_A_flux)              = -0.225

    C(CIV_A_z, CIV_B_fwhm_km_s)         =  0.201

    C(CIV_A_fwhm_km_s, CIV_B_z)         = -0.132

    C(CIV_B_z, CIV_B_flux)              = -0.123

    C(CIV_B_z, CIV_B_fwhm_km_s)         = -0.120

    C(CIV_A_flux, CIV_B_z)              =  0.119
```

Wonderful! The full fit report, which we displayed above, was saved in the .txt file 'specmodel_CIV line_fit_report.txt'. Let's remove the file before we proceed.

[23]:
```
! rm specmodel_CIV_line_fit_report.txt
! ls
```

```
TestSpectralBroadening.ipynb scripting_sculptor_3.ipynb
scripting_sculptor_1.ipynb   speconed_demonstration.ipynb
scripting_sculptor_2.ipynb   spectrum_preparation.ipynb
```

7.9 Full fits, fitting algorithms, and saving your results

7.9.1 Performing a global consecutive fit and saving the SpecModel fit results

In some cases it might be useful to fit individual SpecModels and save their results. However, in most cases we want to fit all SpecModels consecutively and save all results. For this purpose the SpecFit object has a function called *fit*. It can take a keyword argument *save_results*, which defaults to *False*. If we set it to *True* the fit results will automatically be saved to the current folder. By providing a different *foldername* keyword argument the user can choose the folder, where the fit results will be saved.

[24]:
```
# Fit all SpecModels consecutively and save their results to the current folder
fit.fit(save_results=True)
```

(continues on next page)

(continued from previous page)

```
# Check the current folder for the .txt files
! ls
```

```
TestSpectralBroadening.ipynb      specmodel_2_FitAll_fit_report.txt
scripting_sculptor_1.ipynb        specmodel_3_FitAll_fit_report.txt
scripting_sculptor_2.ipynb        specmodel_4_FitAll_fit_report.txt
scripting_sculptor_3.ipynb        speconed_demonstration.ipynb
specmodel_0_FitAll_fit_report.txt spectrum_preparation.ipynb
specmodel_1_FitAll_fit_report.txt
```

Note that in this case, the SpecModels have not been saved with their given names, but rather with their list inidices in the specfit.specmodels list. However, if you used easy to understand model prefixes a simple look into the .txt files will let you recover the fit parameters easily.

To keep the notebook directory clean, let's remove the files again:

[25]:
```
# Remove the SpecModel fit report files
! rm *.txt
# Check the directory again
! ls
```

```
TestSpectralBroadening.ipynb scripting_sculptor_3.ipynb
scripting_sculptor_1.ipynb   speconed_demonstration.ipynb
scripting_sculptor_2.ipynb   spectrum_preparation.ipynb
```

7.9.2 Choosing the fit algorithm available in LMFIT

The SpecFit object has a string attribute called *fitting_method*, which allows you to specify with which algorithm LMFIT will fit your model to the data. An overview over which algorithms are implemented in LMFIT can be found here. The list of names available using Sculptor is saved as a global variable dictionary *fitting_methods* in the SpecModel module, that translates the human readable names of the algorithms into the LMFIT method options. Note that not all of them are fully tested in the Sculptor framework and may lead to errors, if used inappropriately.

[26]:
```
for key in scmod.fitting_methods:
    print('Name: {} \n Method {}'.format(key, scmod.fitting_methods[key]))
```

```
Name: Levenberg-Marquardt
 Method leastsq
Name: Nelder-Mead
 Method nelder
Name: Maximum likelihood via Monte-Carlo Markov Chain
 Method emcee
Name: Least-Squares minimization
 Method least_squares
Name: Differential evolution
 Method differential_evolution
Name: Brute force method
 Method brute
Name: Basinhopping
 Method basinhopping
Name: Adaptive Memory Programming for Global Optimization
 Method ampgo
Name: L-BFGS-B
```

(continues on next page)

(continued from previous page)

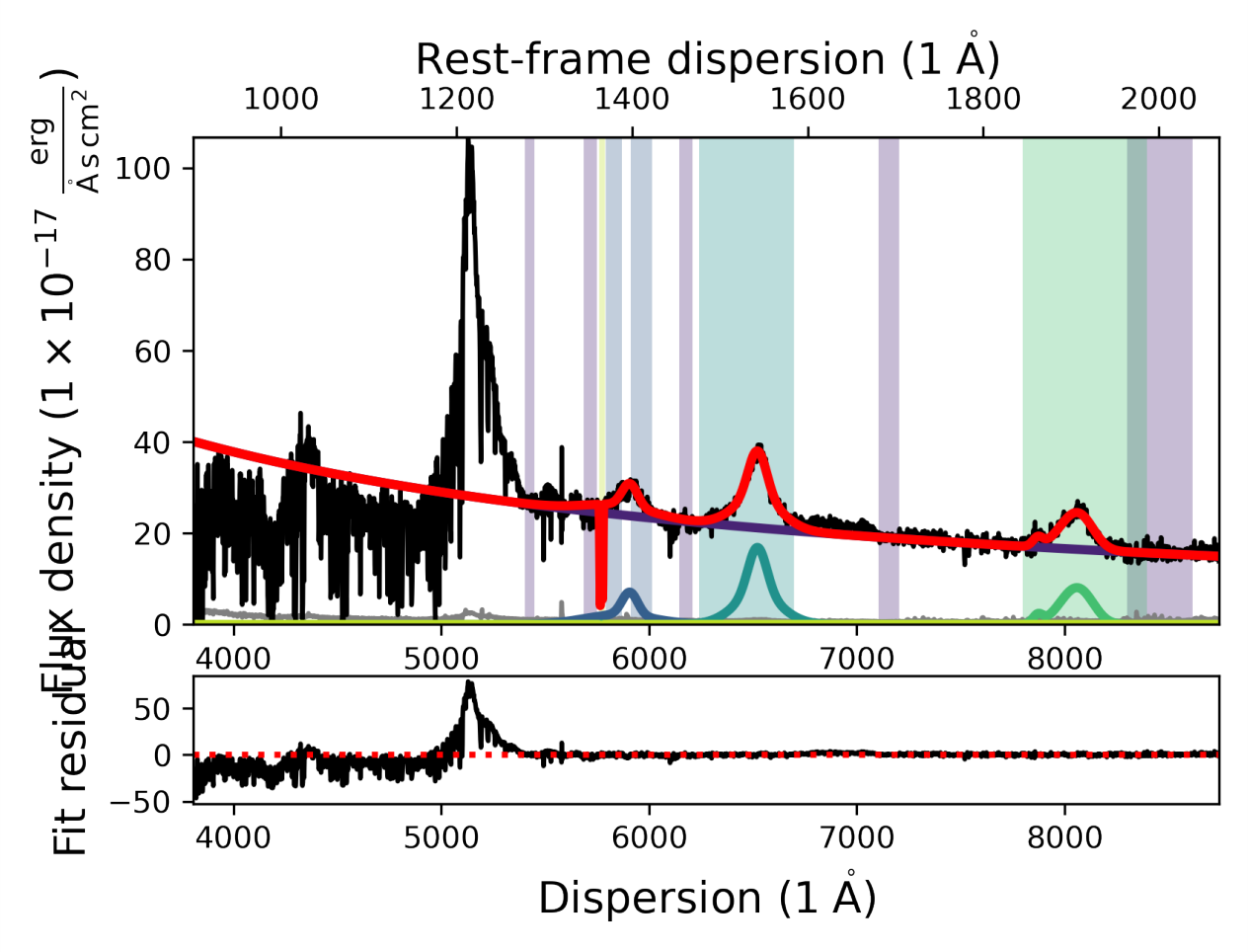
```
 Method lbfgsb
Name: Powell
 Method powell
Name: Conjugate-Gradient
 Method cg
Name: Cobyla
 Method cobyla
Name: BFGS
 Method bfgs
Name: Truncated Newton
 Method tnc
Name: Newton GLTR trust-region
 Method trust-krylov
Name: Trust-region for constrained obtimization
 Method trust-constr
Name: Sequential Linear Squares Programming
 Method slsqp
Name: Simplicial Homology Global Optimization
 Method shgo
Name: Dual Annealing Optimization
 Method dual_annealing
```

For more information on the different fitting algorithms, please refer to the LMFIT documentation.

The default method of Sculptor's SpecFit class is always 'Levenberg-Marquardt'. However, this can be easily changed:

[27]:
```
# Current fitting method
print('Default fitting method: ', fit.fitting_method)

# Change fitting method to Nelder-Mead
fit.fitting_method = 'Nelder-Mead'

# Check fitting method again
print('New fitting method: ', fit.fitting_method)

# Fit all SpecModels
fit.fit()
# Display result
fit.plot()
```

```
Default fitting method:  Levenberg-Marquardt
New fitting method:  Nelder-Mead
```



Some of the fitting methods need extra parameters. A prominent example for this is the 'Maximum likelihood via Monte-Carlo Markov Chain', which uses emcee to perform a maximum likelihood fit using MCMC. For this particular method default parameters are implemented. However, we will devote an entire notebook to show how you can use this fitting option to produce science grade results.

7.9.3 Saving the model fit

One of the goals of the Sculptor package is to enable easy reproducibility of model fits to astronomical spectra and their analysis. Therefore, you can save your entire fit (SpecFit object) to a folder with the SpecFit *save* function. It takes the folderpath+foldername as an attribute and will create the folder if it does not find it in the specified directory.

[28]:
```
# Quick look into the current directory
! ls

# Save the SpecFit object
fit.save('example_fit_notebook')

# Check if the folder was created and contents were saved
! ls
```

```
TestSpectralBroadening.ipynb scripting_sculptor_3.ipynb
scripting_sculptor_1.ipynb   speconed_demonstration.ipynb
```

(continues on next page)

(continued from previous page)

```
scripting_sculptor_2.ipynb   spectrum_preparation.ipynb
```

```
/opt/anaconda3/envs/sculptor-env/lib/python3.9/site-packages/pandas/core/generic.py:2606:
↪ PerformanceWarning:
your performance may suffer as PyTables will pickle object types that it cannot
map directly to c-types [inferred_type->mixed-integer,key->block0_values] [items->Index([
↪'value'], dtype='object')]

  pytables.to_hdf(
```

```
[INFO] Saving SpecModel fit result
[INFO] Saving new model file: example_fit_notebook/0_PL__model.json
[INFO] Saving SpecModel fit result
[INFO] Saving new model file: example_fit_notebook/1_SiIV_A__model.json
[INFO] Saving new model file: example_fit_notebook/1_SiIV_B__model.json
[INFO] Saving SpecModel fit result
[INFO] Saving new model file: example_fit_notebook/2_CIV_A__model.json
[INFO] Saving new model file: example_fit_notebook/2_CIV_B__model.json
[INFO] Saving SpecModel fit result
[INFO] Saving new model file: example_fit_notebook/3_CIII__model.json
[INFO] Saving SpecModel fit result
[INFO] Saving new model file: example_fit_notebook/4_Abs_A_model.json
[INFO] Saving new model file: example_fit_notebook/4_Abs_B_model.json
TestSpectralBroadening.ipynb scripting_sculptor_3.ipynb
example_fit_notebook         speconed_demonstration.ipynb
scripting_sculptor_1.ipynb   spectrum_preparation.ipynb
scripting_sculptor_2.ipynb
```

[29]:
```
# Check the contents of the saved SpecFit folder
! ls ./example_fit_notebook/
```

```
0_PL__model.json          2_fitresult.json          specmodel_0_specdata.hdf5
0_fitresult.json          3_CIII__model.json        specmodel_1_specdata.hdf5
1_SiIV_A__model.json      3_fitresult.json          specmodel_2_specdata.hdf5
1_SiIV_B__model.json      4_Abs_A_model.json        specmodel_3_specdata.hdf5
1_fitresult.json          4_Abs_B_model.json        specmodel_4_specdata.hdf5
2_CIV_A__model.json       4_fitresult.json          spectrum.hdf5
2_CIV_B__model.json       fit.hdf5
```

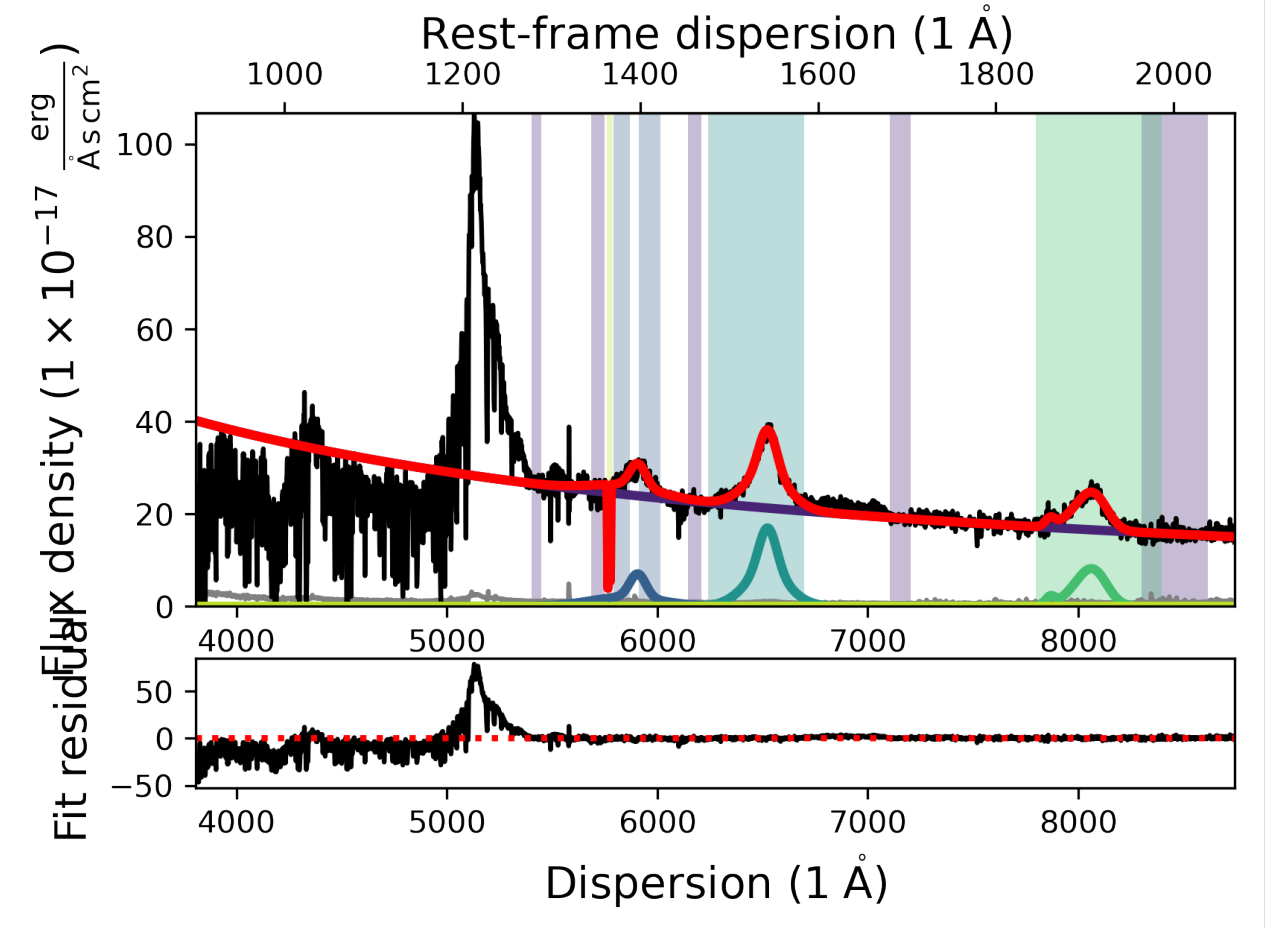
At this point we do not want to go into detail into all the saved files. The *fit.hdf5* saved the main attributes of the SpecFit and SpecModel objects. The individual models and their fit results were saved as *.json* files via the LMFIT functionality for saving models, parameters, and results. The input spectrum is saved as the *spectrum.hdf5* file and the spectra of the SpecModel objects, along with the mask regions and model spectra are saved in the *specmodel_X_specdata.hdf5* files.

7.9.4 Loading a SpecFit object from disk

We can now use the Sculptor GUI to now load the full model fit (SpecFit object) into the GUI from the folder we saved it to and then manipulate it to tune the fit.

Of course we can also instantiate a new SpecFit object and then load the previous result. This is quite simple:

[30]:
```
# Instantiate an empty SpecFit object
new_fit = scfit.SpecFit()

# Load the saved model fit using the folder name
new_fit.load('example_fit_notebook')

# Fit all SpecModels and then display the fit to see if everything worked
new_fit.fit()
new_fit.plot()

# We can also check if it saved the fitting method we changed earlier
print(new_fit.fitting_method)
```




```
Nelder-Mead
```

The full fit you see displayed above should resemble the one a few cells earlier, where we tested changing the fitting method. With this we conclude the demonstration on how to use scripts to construct, fit and save Sculptor model fits.

(We delete the save Sculptor fit to keep the notebook directory clean)

[31]:
```
# Delete the Sculptor model fit folder
! rm -r example_fit_notebook
# Check the directory
! ls
```

```
TestSpectralBroadening.ipynb scripting_sculptor_3.ipynb
scripting_sculptor_1.ipynb   speconed_demonstration.ipynb
scripting_sculptor_2.ipynb   spectrum_preparation.ipynb
```

[ ]:
```

```

CHAPTER

EIGHT

SCRIPTING SCULPTOR 02 - ANALYSING MODEL FITS WITH SPECANALYSIS

The next step after successfully fitting a model to the astronomical spectrum is the analysis of the model fit. The Sculptor *SpecAnalysis* module is designed to make this task easy with your Sculptor fit. While fitting a single function (e.g., a Gaussian) to an emission or absorption feature may be easy to analyze, deriving important quantities, such as the width, peak flux or equivalent width from multi-component line fits can be more tedious. Luckily, the *SpecAnalysis* module includes all of this functionality already.

However, before we begin, we need to import the most important python packages. We will use numpy and the astropy.units and astropy.cosmology packages as well as the three sculptor modules *SpecOneD*, *SpecFit*, and - the topic of this notebook - *SpecAnalysis*.

DISCLAIMER: At the moment the SpecAnalysis functions are written for model spectra in WAVELENGTH units. If you are working with a science spectrum in FREQUENCY units, please convert all model spectra to wavelength before using the *SpecAnalysis* functionality.

Note: A lot of the functionality of the *SpecAnalysis* module presented here can also be used without using Sculptor to generate your model fits. You only need your model spectrum in a file. Then you can read it in as a *SpecOneD* object and analyse it using this module.

[1]:
```
import numpy as np
import pandas as pd
from sculptor import speconed as scspec
from sculptor import specfit as scfit
from sculptor import specanalysis as scana

from astropy import units
from astropy.cosmology import FlatLambdaCDM
```

```
[INFO] Import "sculptor_extensions" package: my_extension
[INFO] Import "sculptor_extensions" package: qso
[INFO] SWIRE library found.
[INFO] FeII iron template of Vestergaard & Wilkes 2001 found. If you will be using these␣
↪templates in your model fit and publication, please add the citation to the original␣
↪work, ADS bibcode: 2001ApJS..134...1V
[INFO] FeII iron template of Tsuzuki et al. 2006 found. If you will be using these␣
↪templates in your model fit and publication, please add the citation to the original␣
↪work, ADS bibcode: 2006ApJ...650...57T
[INFO] FeII iron template of Boroson & Green 1992 found. If you will be using these␣
↪templates in your model fit and publication, please add the citation to the original␣
↪work, ADS bibcode: 1992ApJS...80..109B
```

As before, importing the SpecFit package prompts a few status messages of Sculptor to appear.

For extragalactic sources it is important to specify the cosmology, to allow derivation of luminosities or absolute magnitudes.

[2]:
```
# Define Cosmology for cosmological conversions
cosmo = FlatLambdaCDM(H0=70, Om0=0.3, Tcmb0=2.725)
```

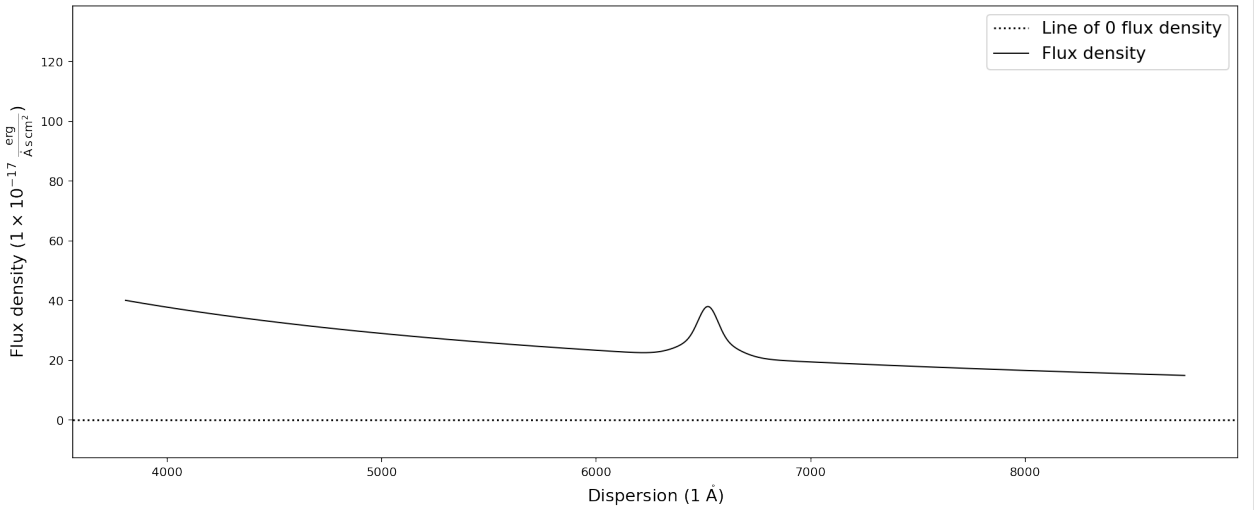
In the next step we import the saved Sculptor model fit.

[3]:
```
# Instantiate an empty SpecFit object
fit = scfit.SpecFit()
# Load the Sculptor model fit from its folder
fit.load('../example_spectrum_fit')
```

Now, we begin with building model spectra (*SpecOneD* objects) from the models in our fit. To do this you need to know the model prefixes you have chosen to uniquely identify your model functions. It does not matter if the model functions are in separate *SpecModel* instances in your model fit (*SpecFit* object) or not. The *SpecAnalysis* function *.build_model_flux* will go through all *SpecModels* to look for them. The first argument of the function will be the *SpecFit* object and the second one will be a list of strings denoting the model functions prefixes to choose.

Our goal is to analyze the CIV emission line (model prefixes 'CIV_A_', 'CIV_B_') and the quasar continuum ('PL_'). However, to illustrate the point above, we will first build the model spectrum of both continuum and CIV line and display it.

[4]:
```
# Building the model flux from the CIV emission line and power law continuum model
# This is done to illustrate how the .build_model_flux function finds model functions
# independent of their SpecModel association in the SpecFit object.
example_model_spec = scana.build_model_flux(fit, ['CIV_A_', 'CIV_B_', 'PL_'])
example_model_spec.plot()
```



As you can see above the *.build_model_flux* function used both the power law continuum and the emission line Gaussians from two different *SpecModels* in the *SpecFit* object to construct the model spectrum. However, in most cases one would want to choose to analyze the continuum independent of the emission lines. Therefore, we build two model spectra below.

[5]:
```
# Build the continuum model spectrum
cont_spec = scana.build_model_flux(fit, ['PL_'])
# Build the CIV emission line model spectrum
civ_spec = scana.build_model_flux(fit, ['CIV_A_', 'CIV_B_'])
```

Sculptor offers some very high level functions for the analysis of both continuum and emission line features. However, we will first illustrate below how to derive important properties by hand using the lower level *SpecAnalysis* functionality.

8.1 Analyzing continuum properties

We begin by analyzing the continuum. In the case of quasar spectroscopy, an important quantity is the flux/luminosity/magnitude at 1450A (rest-frame).

8.1.1 Calculating the average continuum flux density

To calculate the flux density at a specific point in the model spectrum we use the *SpecAnalysis* function **.get_average_fluxden**. It calculates the average flux density of a spectrum (SpecOneD object, first argument) in a window centered at the specified dispersion (second argument) and with a given width (keyword argument, *width=10* default). The dispersion and the width need to be in the same quantity as the dispersion axis, i.e. the unit in which the dispersion axis is plotted if you would plot your model spectrum. A redshift keyword argument (*redshift=0*, default) can be specified, which automatically translates the rest-frame central dispersion value and the rest-frame width to the observed frame.

Note: *Of course you can take your observed spectrum, import it as a SpecOneD object, and calculate the average continuum flux density directly from the science spectrum in regions unaffected by emission or absorption features with this function.*

[6]:
```
# Calculate the average flux density at 1445-1455A rest-frame
fluxden_1450 = scana.get_average_fluxden(cont_spec, 1450,
                                         redshift=fit.redshift)

print('Average flux density at 1450A: ',fluxden_1450)
```

```
Average flux density at 1450A:  2.2770917934343433e-16 erg / (Angstrom cm2 s)
```

Note how the function automatically returns the **physical** value of the flux density **including** the unit. All of the *SpecAnalysis* functions are designed to work with the astropy.units package to automatically return results with units. This allows for easy conversions into different unit system and for double checking the that the returned values are actually reasonable.

8.1.2 Calculating the monochromatic continuum luminosity

In the next step we will convert the average flux density to an average **monochromatic luminosity** using the cosmology (astropy.Cosmology object) defined above.

[7]:
```
lum_mono = scana.calc_lwav_from_fwav(fluxden_1450,
                                     redshift=fit.redshift,
                                     cosmology=cosmo)

print('Monochromatic luminosity at 1450A: ', lum_mono)
```

```
Monochromatic luminosity at 1450A:  8.864701317998224e+43 erg / (Angstrom s)
```

8.1.3 Calculating the apparent monochromatic AB magnitude

We calculate the apparent magnitude from the averaged flux density. To do this we use the *SpecAnalsysis* function *calc_apparent_mag_from_fluxden*. It takes the averaged flux density as the first argument and the central observed dispersion (wavelength or frequency) of the flux density as arguments. Note that the dispersion value should be a quantity (astropy.Quantity).

[8]:
```
abmag = scana.calc_apparent_mag_from_fluxden(
    fluxden_1450,
    1450*(1.+fit.redshift)*units.AA)

print('Apparent AB magnitude at 1450A: ', abmag)
```

```
Apparent AB magnitude at 1450A:  17.761599161710915 mag
```

8.1.4 Calculating the absolute monochromatic AB magnitude

From here we can calculate the sources AB magnitude at 1450A in three different ways: 1. Calculate the absolute magnitude from the the apparent magnitude 2. Calculate the absolute magnitude from the flux density 3. Calculate the absolute magnitude from the monochromatic luminosity

Let's briefly go through all three possibilities.

Calculate the absolute magnitude from the the apparent magnitude

We calculated the apparent magnitude above. Now we only need to convert it to the absolute magnitude. In this step we need to specify the cosmology as well as to indicate how the flux should be corrected due to the change of the filter bandpass - the K-correction. For monochromatic magnitudes this is simply a scaling factor of (1+z) in flux.

We use the *SpecAnalysis* function *calc_absolute_mag_from_apparent_mag*, which takes the apparent magnitude as the first argument, the cosmology (astropy.Cosmology) as the second argument and the redshift as the third argument. At present the function only allows to add the kcorrection for spectra with a power-law type continuum (power law index *a_nu*).

Note: In the case of monochromatic magnitudes, the K-correction is specified by setting the *kcorrection* keyword argument to *True* and the power law slope to *a_nu=0*, independent of the source's continuum shape. (Incidentally these are the default values.)

(For apparent to absolute magnitude conversions for bandpass magnitudes and non-power law continua, you can still use this function setting *kcorrection=False* and then apply the appropriate K-correction factor in magnitudes yourself.)

[9]:
```
abs_abmag = scana.calc_absolute_mag_from_apparent_mag(abmag, cosmo,
                                                      fit.redshift,
                                                      kcorrection=True,
                                                      a_nu=0)

print('Absolute magnitude at 1450 (from apparent magnitude): ', abs_abmag)
```

```
Absolute magnitude at 1450 (from apparent magnitude):  -27.889171135994392 mag
```

Calculate the absolute magnitude from the monochromatic luminosity

We do not need to calculate the apparent monochromatic magnitude to get to the absolute monochromatic magnitude. We can directly compute this from the flux density at 1450A using the *SpecAnalysis.calc_absolute_mag_from_fluxden* function.

However, the function goes through the same steps as we did above. Therefore, we also need to specify the K-correction and power law slope keywords on top of the monochromatic flux density (first argument), the observed dispersion (second argument), the cosmology (third argument), and the redshift (fourth argument).

[10]:
```
abs_abmag2 = scana.calc_absolute_mag_from_fluxden(
    fluxden_1450, 1450*(1.+fit.redshift) * units.AA,
    cosmo, fit.redshift, kcorrection=True, a_nu=0)

print('Absolute magnitude at 1450 (from monochr. flux density): ', abs_abmag2)
```

```
Absolute magnitude at 1450 (from monochr. flux density):  -27.889171135994392 mag
```

Calculate the absolute magnitude from the monochromatic luminosity

We already calculated the monochromatic luminosity above. Of course we can directly calculate the absolute magnitude from the monochromatic luminosity. In the *SpecAnalysis* module the function *calc_absolute_mag_from_monochromatic_luminosity* does that for you. You specify the monochromatic luminosity and the corresponding **rest-frame wavelength** value as the first and second arguments. The cosmological corrections were already applied, when calculating the monochromatic luminosity above.

[11]:
```
abs_abmag3 = scana.calc_absolute_mag_from_monochromatic_luminosity(lum_mono, 1450*units.
↪AA)

print('Absolute magnitude at 1450 (from monochr. luminosity): ', abs_abmag3)
```

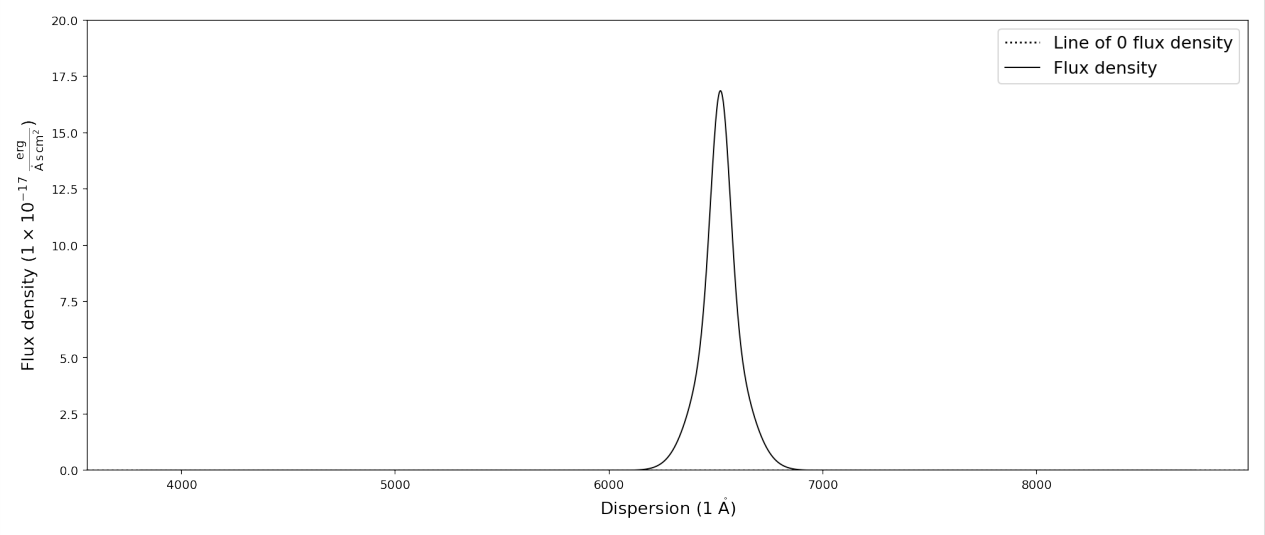
```
Absolute magnitude at 1450 (from monochr. luminosity):  -27.88917113599439 mag
```

As you can see all three functions to calculate the absolute magnitude lead to the same value of ~-27.9 mag.

8.2 Analyzing line properties

We now turn to calculate line properties. In our example, we will focus on the CIV emission line of the example quasar spectrum. We modeled the CIV line with two Gaussian components and stored the model flux in a *SpecOneD* object, called civ_spec. Let's first check if the model flux looks reasonable.

[12]:
```
# Plot the CIV model flux spectrum, adjusting the y-axis limits
civ_spec.plot(ymin=0, ymax=20)
```



The model flux specturm look reasonable, so we can continue!

8.2.1 The peak flux density

In order to calculate the peak flux density, we can rely on the *numpy* package:

[13]:
```
civ_peak_fluxden = np.max(civ_spec.fluxden)*civ_spec.fluxden_unit
print('CIV peak flux density: {:.2e}'.format(civ_peak_fluxden))
```

```
CIV peak flux density: 1.69e-16 erg / (Angstrom cm2 s)
```

8.2.2 The peak redshift

Although this calculation is quite simply, we have decided to add this functionality to the *SpecAnalysis* module with the *get_peak_redshift* function. It requires the line model spectrum and the rest-frame wavelength of the feature.

[14]:
```
civ_z = scana.get_peak_redshift(civ_spec, 1549.06)
print('CIV peak redshift: {:.3f}'.format(civ_z))
```

```
CIV peak redshift: 3.210
```

8.2.3 The line Full Width at Half Maximum (FWHM)

For line models with a single Gaussian component, the FWHM can be directly retrieved from the model function. However, for composite lines it is more difficult to define and then calculate a FWHM. The *SpecAnalysis* implementation for calculating the FWHM **get_fwhm**, follows the simplest definition of the FWHM:

It calculates the peak flux density and then sets up a spline interpolation subtracting half of the peak flux density from the model spectrum. If the line model flux is well defined the resulting spline should have exactly two roots (model flux = 0). The function returns the dispersion difference between the two roots. If the line feature has multiple components more or less than two roots can be found in which case a np.NaN value will be returned.

Correction for spectral resolution: The keyword argument *resolution* allows to specify the spectral resolution in R = Lambda/Delta Lambda. If the *resolution* value is not *None*, the spectral resolution will be subtracted from the FWHM

of the line feature (assuming FWHM_corr^2 = FWHM^2 - Resolution_km_s^2). If you specify the resolution, an info message will let you know that the function has corrected the FWHM for it.

[15]:
```
# Calculate the CIV FWHM
civ_fwhm = scana.get_fwhm(civ_spec)
print('CIV FWHM: {:.2f}'.format(civ_fwhm))

# Calculate the CIV FWHM, taking into account a spectral resolution of R=1000
civ_fwhm = scana.get_fwhm(civ_spec, resolution=1000)
print('CIV FWHM (accounting for spectral resolution): {:.2f}'.format(civ_fwhm))
```

```
CIV FWHM: 6351.97 km / s
[INFO] FWHM is corrected for the provided resolution of R=1000
CIV FWHM (accounting for spectral resolution): 6344.89 km / s
```

Additionally, there are two different methods for calculating the FWHM from the model spectrum. The default method 'spline' uses a spline to interpolate the original spectrum and find the zero points using a root finding algorithm on the spline. The second method 'sign' finds sign changes in the half peak flux subtracted spectrum. Their results may differ depending on the dispersion resolution. From an initial test it seems that the computational cost of both methods is similar.

[16]:
```
civ_fwhm = scana.get_fwhm(civ_spec)
print('CIV FWHM ("sign" method): {:.2f}'.format(civ_fwhm))

civ_fwhm = scana.get_fwhm(civ_spec, method='sign')
print('CIV FWHM ("spline" method): {:.2f}'.format(civ_fwhm))
```

```
CIV FWHM ("sign" method): 6351.97 km / s
CIV FWHM ("spline" method): 6350.92 km / s
```

8.2.4 The integrated line flux

The integrated line flux is easily calculated by the *get_integrated_flux* function of the *SpecAnalysis* module. By default the line model spectrum will be integrated over its full length. However, you can specify the *disp_range* keyword argument, setting the integration boundaries in units of the dispersion axis manually.

[17]:
```
civ_flux = scana.get_integrated_flux(civ_spec)
print('CIV integrated flux: {:.2e}'.format(civ_flux))
```

```
CIV integrated flux: 2.97e-14 erg / (cm2 s)
```

8.2.5 The integrated line luminosity

In a similar fashion you can calculate the integrated line luminosity. In addition to the line model spectrum, the function takes the source redshift and the cosmology as the second and third argument.

[18]:
```
civ_line_lum = scana.calc_integrated_luminosity(civ_spec,
                                                fit.redshift,
                                                cosmo)
print('CIV integrated line luminosity: {:.2e} '.format(civ_line_lum))
```

```
CIV integrated line luminosity: 2.74e+45 erg / s
```

8.2.6 The line equivalent width (observed-frame/rest-frame)

An important property of an emission or absorption feature is its equivalent width. For this calculation we need both the continuum model flux (cont_spec) and the line model flux (civ_spec). In order to calculate the rest-frame equivalent width for extragalactic sources the *redshift* keyword also needs to be specified. In the *SpecAnalysis* module the *get_equivalent_width* function calculates the equivalent width. For our example we use the CIV redshift that we determined above in this calculation. If we don't specify the redshift, the function returns the observed-frame equivalent width.

[19]:
```
# Calculate the rest-frame equivalent width
civ_ew = scana.get_equivalent_width(cont_spec, civ_spec, redshift=civ_z)
print('CIV EW (rest-frame): {:.2f}'.format(civ_ew))

# Calculate the observed-frame equivalent width
civ_ew = scana.get_equivalent_width(cont_spec, civ_spec)
print('CIV EW (observed-frame): {:.2f}'.format(civ_ew))
```

```
CIV EW (rest-frame): 33.38 Angstrom
CIV EW (observed-frame): 140.51 Angstrom
```

8.2.7 Non-parametric measurements

Furthermore we provide a range of non-parametric measurements for emission line features. These are measurements of the fraction of cumulative flux in velocity space. The redshift and rest-frame wavelength of the emission line define the velocity zero point. Negative velocities indicate flux blueward of the line and positive velocities indicate flux redward of the line.

In the current implementation the velocities at 5%, 10%, 50%, 90% and 95% of the cumulative flux fraction are calculated. The calculation also returns the mean resolution at 50% of the cumulative flux fraction as an indication for the uncertainty introduced by the spectral resolution.

From the median velocity (velocity at 50% of the cumulative flux fraction) we can derive its frequency, wavelength and redshift, which are all included in this analysis function.

The dispersion range keyword argument can be used to supply the physical dispersion limits in which the analysis should be executed.

The non-parametric measurements are NOT included in the default line measurements (scana.emfeat_measures_default) for the high-level analysis function (see below). Specifically for the analysis of resampled results or MCMC chains the non-parametric measurements are more computationally expensive.

[20]:
```
# Use the emission feature analysis function to analyze the CIV line
civ_result = scana.analyze_emission_feature(
    fit, 'CIV', ['CIV_A_', 'CIV_B_'], 1549.06, cont_model_names=['PL_'],
    redshift=fit.redshift, emfeat_meas=['nonparam'], cosmology=cosmo)

# Print the results from the dictionary
for key in civ_result.keys():
    print('{} = {:.2e}'.format(key, civ_result[key]))
```

```
CIV_v50 = -1.24e+03 km / s
CIV_v05 = -8.37e+03 km / s
CIV_v10 = -6.35e+03 km / s
CIV_v90 = 3.89e+03 km / s
CIV_v95 = 5.92e+03 km / s
```

(continues on next page)

(continued from previous page)

```
CIV_vres_at_line = 6.88e+01 km / s
CIV_freq_v50 = 4.60e+14 Hz
CIV_wave_v50 = 6.52e+03 Angstrom
CIV_redsh_v50 = 3.21e+00
```

8.3 High-level SpecAnalysis routines

We have written high-level *SpecAnalysis* routines that automatically calculate a range of continuum and line properties for you using the functions described above.

8.3.1 SpecAnalysis.analyze_continuum

The *analyze_continuum* routine allows for automatic continuum analysis. It takes four important arguments: * the SpecFit object * a list of model prefix names that make up the continuum model * a list of rest-frame wavelengths (float) for which fluxes, luminosities and magnitudes should be calculated * the cosmology (astropy.Cosmology)

The most important keyword argument is *cont_meas*, which stands for continuum measurements. The full list of continuum measurements available in the SpecAnalysis module is:

[21]:
```
print(scana.cont_measures_default)
```

```
['fluxden_avg', 'Lwav', 'appmag', 'absmag']
```

By default all of these will be calculated. For the full list of keyword arguments, please directly consult the documentation.

The function returns a dictionary with the results.

In the following example, we will analyze the continuum of the example spectrum at rest-frame wavelengths of 1450A (as above) and 1280A.

[22]:
```
# Define Cosmology for cosmological conversions
cosmo = FlatLambdaCDM(H0=70, Om0=0.3, Tcmb0=2.725)

# Import the saved example spectrum
fit = scfit.SpecFit()
fit.load('../example_spectrum_fit')

# Use the continuum analysis function
cont_result = scana.analyze_continuum(fit, ['PL_'],
                                      [1450, 1280],
                                      cosmo, width=10)

# Print the results from the dictionary
for key in cont_result.keys():
    print('{} = {:.2e}'.format(key, cont_result[key]))
```

```
1450_fluxden_avg = 2.28e-16 erg / (Angstrom cm2 s)
1450_Lwav = 8.86e+43 erg / (Angstrom s)
1450_appmag = 1.78e+01 mag
1450_absmag = -2.79e+01 mag
1280_fluxden_avg = 2.64e-16 erg / (Angstrom cm2 s)
```

(continues on next page)

(continued from previous page)

```
1280_Lwav = 1.03e+44 erg / (Angstrom s)
1280_appmag = 1.79e+01 mag
1280_absmag = -2.78e+01 mag
```

8.3.2 SpecAnalysis.analyze_emission_feature

Disclaimer:At this point we have only implemented an automatic high-level analysis for emission features. A more general function for both absorption and emission lines or, alternatively, a separate function for absorption lines is on our To-Do list.

In similar fashion to the *analyze_continuum* function above, *analyze_emission_feature* takes four important arguments: * the SpecFit object * the name of the emission feature, which will be used to name the resulting measurements in the output dictionary * a list of model names to create the emission feature flux from * the rest-frame wavelength of the emission feature

The cosmology is a keyword argument for this function (*cosmology*, astropy.Cosmology). Another important keyword argument is *emfeat_meas*, which stands for emission feature measurements and is a list of strings. The full list of possible emission feature measurements (and the default value) is:

[23]:
```
print(scana.emfeat_measures_default)
```

```
['peak_fluxden', 'peak_redsh', 'EW', 'FWHM', 'flux', 'lum']
```

For a full list of all available keyword arguments, please directly consult the documentation of the *SpecAnalysis* module.

In the following example we will repeat our analysis of the CIV line from above. Note, that in order to calculate the equvialent width we also need to specify the cont_model_names to generate the continuum model flux.

[24]:
```
# Use the emission feature analysis function to analyze the CIV line
civ_result = scana.analyze_emission_feature(
    fit, 'CIV', ['CIV_A_', 'CIV_B_'], 1549.06, cont_model_names=['PL_'],
    redshift=fit.redshift, emfeat_meas=None, cosmology=cosmo)

# Print the results from the dictionary
for key in civ_result.keys():
    print('{} = {:.2e}'.format(key, civ_result[key]))
```

```
CIV_peak_fluxden = 1.69e-16 erg / (Angstrom cm2 s)
CIV_peak_redsh = 3.21e+00
CIV_EW = 3.32e+01 Angstrom
CIV_FWHM = 6.35e+03 km / s
CIV_flux = 2.97e-14 erg / (cm2 s)
CIV_lum = 2.74e+45 erg / s
```

CHAPTER

NINE

SCRIPTING SCULPTOR 03 - FITTING AND ANALZYING MODELS USING MCMC

This tutorial notebook focuses on using *Sculptor's* modules to fit and analyze a model using the maxmimum likelihood Markov Chain Monte Carlo method implemented in LMFIT using the emcee backend. You can find more information on how LMFIT uses emcee here.

[1]:
```
%matplotlib inline

import corner
import matplotlib.pyplot as plt

from astropy.cosmology import FlatLambdaCDM

from sculptor import specfit as scfit
from sculptor import specmodel as scmod
```

```
[INFO] Import "sculptor_extensions" package: my_extension
[INFO] Import "sculptor_extensions" package: qso
[INFO] SWIRE library found.
[INFO] FeII iron template of Vestergaard & Wilkes 2001 found. If you will be using these␣
↪templates in your model fit and publication, please add the citation to the original␣
↪work, ADS bibcode: 2001ApJS..134...1V
[INFO] FeII iron template of Tsuzuki et al. 2006 found. If you will be using these␣
↪templates in your model fit and publication, please add the citation to the original␣
↪work, ADS bibcode: 2006ApJ...650...57T
[INFO] FeII iron template of Boroson & Green 1992 found. If you will be using these␣
↪templates in your model fit and publication, please add the citation to the original␣
↪work, ADS bibcode: 1992ApJS...80..109B
```

9.1 Fitting a model using MCMC

We begin by loading the example spectrum fit to an SDSS quasar spectrum we have been using in previous tutorials.

[2]:
```
# Instantiate an empty SpecFit object
fit = scfit.SpecFit()
# Load the example spectrum fit
fit.load('../example_spectrum_fit')
```

In the next step we need to change the fitting method. Just as a reminder the full list of fitting methods is available as a globa variable in the *SpecModel* module:

[3]:
```
for key in scmod.fitting_methods:
    print('Name: {} \n Method {}'.format(key, scmod.fitting_methods[key]))
```

```
Name: Levenberg-Marquardt
 Method leastsq
Name: Nelder-Mead
 Method nelder
Name: Maximum likelihood via Monte-Carlo Markov Chain
 Method emcee
Name: Least-Squares minimization
 Method least_squares
Name: Differential evolution
 Method differential_evolution
Name: Brute force method
 Method brute
Name: Basinhopping
 Method basinhopping
Name: Adaptive Memory Programming for Global Optimization
 Method ampgo
Name: L-BFGS-B
 Method lbfgsb
Name: Powell
 Method powell
Name: Conjugate-Gradient
 Method cg
Name: Cobyla
 Method cobyla
Name: BFGS
 Method bfgs
Name: Truncated Newton
 Method tnc
Name: Newton GLTR trust-region
 Method trust-krylov
Name: Trust-region for constrained obtimization
 Method trust-constr
Name: Sequential Linear Squares Programming
 Method slsqp
Name: Simplicial Homology Global Optimization
 Method shgo
Name: Dual Annealing Optimization
 Method dual_annealing
```

Let us now set the fitting method to 'Maximum likelihood via Monte-Carlo Markov Chain'.

[4]:
```
# Setting the fit method to MCMC via emcee
fit.fitting_method = 'Maximum likelihood via Monte-Carlo Markov Chain'
```

This fitting method takes additional keyword arguments that specify * the number of MCMC steps (*steps*), * the number of steps considered to be the burn in phase (*burn*), which will be discarded, * the number of walkers (*nwalkers*), which should be a much larger number than your variables * the number of workers (*workers*) for multiprocessing (*workers=4* will spawn a multiprocessing-based pool with 4 parallel processes), * thin sampling accepting only 1 in every *thin* (int) samples, * whether the objective function has been weightes by measurement uncertainties (*is_weighted*, boolean), * whether a progress bar should be printed (*progress*, boolean), * and the *seed* (default: *seed=1234*).

Note: The *is_weighted* keyword will be set to *True* by default.

For a full documentation of the keyword arguments, please visit the LMFIT emcee documentation. The list of keyword arguments can be accessed via *SpecFit.emcee_kws*.

[5]:
```
for key in fit.emcee_kws:
    print(key, fit.emcee_kws[key])
```

```
steps 1000
burn 300
thin 20
nwalkers 50
workers 1
is_weighted True
progress False
seed 1234
```

Additional keywords can be added to this dictionary, which will be passed to the *SpecModel* fit function, once the *SpecFit* fitting method has been selected to be MCMC (see above).

The default values **do not automatically constitute a good default choice**. The choice of those parameters is highly dependent on the model fit (e.g., number of variable parameters). In order to properly fit the CIV emission line we will adjust them:

[6]:
```
# Set the MCMC keywords
fit.emcee_kws['steps'] = 2000
fit.emcee_kws['burn'] = 500
# We are fitting 6 parameters so nwalker=50 is fine
fit.emcee_kws['nwalkers'] = 25
# No multiprocessing for now
fit.emcee_kws['workers'] = 1
fit.emcee_kws['thin'] = 2
fit.emcee_kws['progress'] = True
# Take uncertainties into account
fit.emcee_kws['is_weighted'] = True
```

Before fitting any model using MCMC it is strongly advised to fit the model with a standard algorithm (e.g., 'Levenberg-Marquardt') after setting it up and then start the MCMC fit from the best-fit parameters.

In our case we already fitted all models to the quasar spectrum before saving it. Therefore, we can immediately fit the CIV emission line model.

[7]:
```
# In case we have forgotten the index of the CIV SpecModel in the fit.specmodels list.
for idx, specmodel in enumerate(fit.specmodels):
    print(idx, specmodel.name)
```

```
0 Continuum
1 SiIV_line
2 CIV_line
3 CIII]_complex
4 Abs_lines
```

[8]:
```
# Select the CIV emission line SpecModel
civ_model = fit.specmodels[2]

# Fit the SpecModel using the MCMC method and emcee_kws modified above
civ_model.fit()
```

(continues on next page)

(continued from previous page)

```
# Print the fit result
print(civ_model.fit_result.fit_report())
```

```
100%|| 2000/2000 [00:06<00:00, 288.42it/s]
```

```
The chain is shorter than 50 times the integrated autocorrelation time for 7␣
↪parameter(s). Use this estimate with caution and run a longer chain!
N/50 = 40;
tau: [60.71255358 65.93961611 64.03676477 66.17055916 63.93059079 73.11907444
 61.57293084]
[[Model]]
    (Model(line_model_gaussian, prefix='CIV_B_') + Model(line_model_gaussian, prefix=
↪'CIV_A_'))
[[Fit Statistics]]
    # fitting method   = emcee
    # function evals   = 50000
    # data points      = 309
    # variables        = 7
    chi-square         = 301.595330
    reduced chi-square = 0.99866003
    Akaike info crit   = 6.50516627
    Bayesian info crit = 32.6385552
[[Variables]]
    CIV_B_z:          3.21141536 +/- 8.0272e-04 (0.02%) (init = 3.209845)
    CIV_B_flux:       1091.60304 +/- 132.570578 (12.14%) (init = 1151.246)
    CIV_B_cen:        1549.06 (fixed)
    CIV_B_fwhm_km_s:  4704.44704 +/- 244.820184 (5.20%) (init = 4790.818)
    CIV_A_z:          3.20746100 +/- 0.00166939 (0.05%) (init = 3.209726)
    CIV_A_flux:       1880.60255 +/- 118.497872 (6.30%) (init = 1820.904)
    CIV_A_cen:        1549.06 (fixed)
    CIV_A_fwhm_km_s:  11663.2794 +/- 674.461140 (5.78%) (init = 12121.84)
    __lnsigma:       -0.03756195 +/- 0.03824749 (101.83%) (init = 0.01)
[[Correlations]] (unreported correlations are < 0.100)
    C(CIV_B_flux, CIV_A_flux)           = -0.963
    C(CIV_B_flux, CIV_B_fwhm_km_s)      =  0.942
    C(CIV_B_fwhm_km_s, CIV_A_flux)      = -0.929
    C(CIV_B_flux, CIV_A_fwhm_km_s)      =  0.912
    C(CIV_A_flux, CIV_A_fwhm_km_s)      = -0.803
    C(CIV_B_fwhm_km_s, CIV_A_fwhm_km_s) =  0.797
    C(CIV_B_z, CIV_A_z)                 = -0.620
    C(CIV_B_z, CIV_A_fwhm_km_s)         = -0.576
    C(CIV_B_z, CIV_B_flux)              = -0.550
    C(CIV_B_z, CIV_B_fwhm_km_s)         = -0.520
    C(CIV_B_z, CIV_A_flux)              =  0.487
    C(CIV_A_z, CIV_A_fwhm_km_s)         =  0.374
    C(CIV_B_flux, CIV_A_z)              =  0.269
    C(CIV_B_fwhm_km_s, CIV_A_z)         =  0.242
    C(CIV_A_z, CIV_A_flux)              = -0.186
```

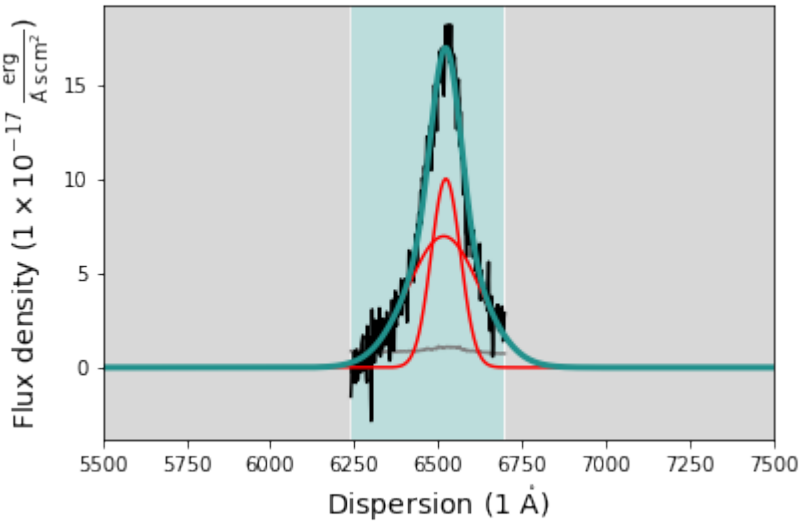
We visualize the fit result by plotting the CIV model fit.

[9]:
```
# Plot the fitted model
```

(continues on next page)

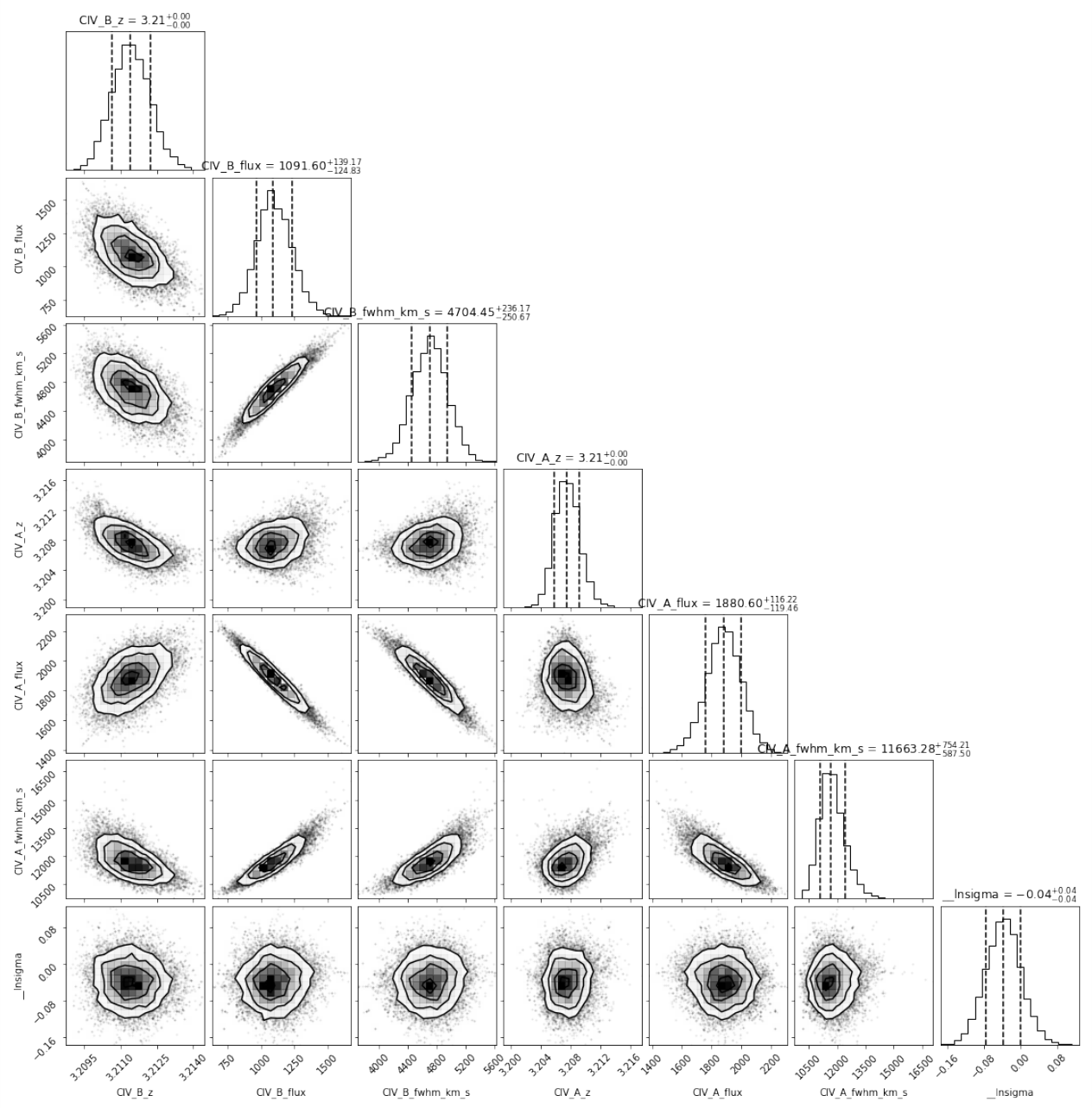
(continued from previous page)

```
civ_model.plot(xlim=[5500,7500])
```



However, much more important is that we have a look at the sampled parameter space and the posterior distributions of the fit parameters. To do this we first retrieve the sampled flat chain and the plot it using the *corner* package.

[10]:
```
# Retrieve the MCMC flat chain of the CIV model fit
data = civ_model.fit_result.flatchain.to_numpy()
# Visualize the flat chain fit results using the typical corner plot
corner_plot = corner.corner(data,
                            labels=civ_model.fit_result.var_names,
                            quantiles=[0.16, 0.5, 0.84],
                            show_titles=True,
                            title_kwargs={"fontsize": 12}
                            )
plt.show()
```



The posterior distributions of the fit parameters look quite well behaved. It seems we have appropriately sampled the parameter space. The plot illustrates strong covariance in some variable parameters pairs (e.g., CIV_A_amp and CIV_A_fwhm_km_s). For further analysis of the MCMC fit, we will now save the flat chain in a file using the *SpecModel.save_mcmc_chain* function, which takes a folder path as an argument. We will save the flat chain in the example fit folders.

[11]:
```
# Save the MCMC flatchain to a file for analysis
civ_model.save_mcmc_chain('../example_spectrum_fit')

! ls ../example_spectrum_fit/*.hdf5
```

```
../example_spectrum_fit/fit.hdf5
```

(continues on next page)

(continued from previous page)

```
../example_spectrum_fit/specmodel_0_specdata.hdf5
../example_spectrum_fit/specmodel_1_specdata.hdf5
../example_spectrum_fit/specmodel_2_specdata.hdf5
../example_spectrum_fit/specmodel_3_specdata.hdf5
../example_spectrum_fit/specmodel_4_specdata.hdf5
../example_spectrum_fit/specmodel_CIV_line_mcmc_chain.hdf5
../example_spectrum_fit/spectrum.hdf5
```

9.2 Analyzing the MCMC fit results

It is perfectly fine to work with the parameter fit results from the MCMC fit. However, we do have the full flat chain information and, therefore, it may be better to get posterior distributions for all the properties of the continuum or feature we are interested in.

In a previous notebook tutorial we covered how to use *SpecAnalysis* to analyze our model fits. To construct posterior distributions of the CIV emission line properties we will now use the high-level *SpecAnalysis* function *analyze_mcmc_results*, which uses the *SpecAnalysis.analyze_continuum* and *SpecAnalysis.analyze_emission_feature* we have introduced earlier.

[12]:
```
# Import the SpecAnalysis and Cosmology modules
from sculptor import specanalysis as scana
from astropy.cosmology import FlatLambdaCDM
```

We begin our analysis by specfiying the Cosmology we will be using and then import the model fit.

[13]:
```
# Define Cosmology for cosmological conversions
cosmo = FlatLambdaCDM(H0=70, Om0=0.3, Tcmb0=2.725)

# Instantiate an empty SpecFit object
fit = scfit.SpecFit()
# Load the example spectrum fit
fit.load('../example_spectrum_fit')
```

The *analyze_mcmc_results* function is designed to automatically analyze the output of the MCMC analysis. The user specifies the folder with the MCMC output data and the function will search for any *_mcmc_chain.hdf5* files. As a second argument the function requires the full model fit information (SpecFit object). The third and fourth argument are dictionaries, *continuum_listdict* and *emission_feature_listdict*, which specify which continuum models and features should be analyzed.

The *continuum_listdict* and the *emission_feature_listdict* hold the arguments for the *SpecAnalysis.analyze_continuum* and *SpecAnalysis.analyze_emission_feature* functions that will be called by the MCMC analysis procedure.

The following parameters should be specified in the *continuum_listdict*: * 'model_names' - list of model function prefixes for the full continuum model * 'rest_frame_wavelengths' - list of rest-frame wavelengths (float) for which fluxes, luminosities and magnitudes should be calculated

The other arguments for the *SpecAnalysis.analyze_continuum* are provided to the MCMC analysis function separately.

The following parameters should be specified in the *emission_feature_listdict*: * 'feature_name' - name of the emission feature, which will be used to name the resulting measurements in the output file * 'model_names' - list of model names to create the emission feature model flux from * 'rest_frame_wavelength' - rest-frame wavelength of the emission feature

Additionally, one can specify: * 'disp_range' - 2 element list holding the lower and upper dispersion boundaries flux density integration

For a list of all arguments and keyword arguments, have a look at the function header:

```
:param foldername: Path to the folder with the MCMC flat chain hdf5 files.
:type foldername: string
:param specfit: Sculptor model fit (SpecFit object) containing the
    information about the science spectrum, the SpecModels and parameters.
:type specfit: sculptor.specfit.SpecFit
:param continuum_dict: The *continuum_listdict* holds the arguments for
    the *SpecAnalysis.analyze_continuum* function that will be called by
    this procedure.
:type continuum_dict: dictionary
:param emission_feature_dictlist: The *emission_feature_listdict* hold the
    arguments for the *SpecAnalysis.analyze_emission_feature* functions that
    will be called by this procedure.
:type emission_feature_dictlist: dictionary
:param redshift: Source redshift
:type: float
:param cosmology: Cosmology for calculation of absolute properties
:type cosmology: astropy.cosmology.Cosmology
:param emfeat_meas: This keyword argument allows to specify the list of
    emission feature measurements.
    Currently possible measurements are ['peak_fluxden', 'peak_redsh', 'EW',
    'FWHM', 'flux']. The value defaults to 'None' in which all measurements
    are calculated
:type emfeat_meas: list(string)
:param cont_meas: This keyword argument allows to specify the list of
    emission feature measurements.
    Currently possible measurements are ['peak_fluxden', 'peak_redsh', 'EW',
    'FWHM', 'flux']. The value defaults to 'None' in which all measurements
    are calculated
:type cont_meas: list(string)
:param dispersion: This keyword argument allows to input a dispersion
    axis (e.g., wavelengths) for which the model fluxes are calculated. The
    value defaults to 'None', in which case the dispersion from the SpecFit
    spectrum is being used.
:type dispersion: np.array
:param width: Window width in dispersion units to calculate the average
    flux density in.
:type width: [float, float]
:param concatenate: Boolean to indicate whether the MCMC flat chain and
    the analysis results should be concatenated before written to file.
    (False = Only writes analysis results to file; True = Writes analysis
    results and MCMC flat chain parameter values to file)
:type concatenate: bool
```

IMPORTANT: Only model functions that are sampled together can be analyzed together. Therefore, only model functions from ONE SpecModel can be analyzed together.

THIS MEANS: For example, if you wanted to analyze the continuum model and the CIV emission line together using MCMC, you would have to fit them with ONE SpecModel. In our setup we have separate SpecModels for the continuum and the CIV line. Therefore, we cannot do that. However, to calculate the CIV equivalent width we need the continuum model. In this case we still need to specify the continuum model prefix for the analysis

routine. It will build the best-fit continuum model and use it for the CIV analysis. As the best-fit continuum model was subtracted before the CIV line was fit using MCMC, this is the correct way of analzying the results as well.

What does the *analyze_mcmc_results* actually do?

It identifies for which of the specified models (continuum/features) it can find the necessary MCMC flat chain information. If multiple continuum or feature models have been specified it will go through them one by one. For each entry in the flat chain file it will read in the model function parameters, build the model fluxes, and then analyze the model using the *SpecAnalysis.analyze_continuum* and *SpecAnalysis.analyze_emission_feature* functions. If you have very long MCMC chains, this process can easily take several minutes. A progress bar will keep you informed.

What is the result of the *analyze_mcmc_results* function?

The function will write an "Enhanced Character Separated Values" csv file to the same folder with the MCMC flat chain data. The csv file can be read in and then further manipulated with *astropy*. **Unit information on physical quantities is saved along with the values to the csv file.**

Let us now set up both dictionaries for our example:

[14]:
```
continuum_listdict = {'model_names': ['PL_'],
                      'rest_frame_wavelengths': [1450, 1280]}

emission_feature_listdict = [{'feature_name': 'CIV',
                              'model_names' : ['CIV_A_', 'CIV_B_'],
                              'rest_frame_wavelength': 1549.06}
                                  ]
```

And then run the *analyze_mcmc_results* function:

[15]:
```
scana.analyze_mcmc_results('../example_spectrum_fit', fit,
                           continuum_listdict,
                           emission_feature_listdict,
                           fit.redshift, cosmo)
```

```
  0%|          | 14/18750 [00:00<02:14, 138.92it/s]
```

```
[INFO] Starting MCMC analysis
[INFO] Working on output file ../example_spectrum_fit/specmodel_CIV_line_mcmc_chain.hdf5
[INFO] Analyzing emission feature CIV
```

```
100%|| 18750/18750 [02:09<00:00, 145.09it/s]
```

It can take a while until the MCMC anlysis finishes. The results are written into the same folder with the MCMC flat chain data. In our case we can find the analyzed results (*mcmc_analysis_CIV.csv*) in

[16]:
```
! ls ../example_spectrum_fit/
```

```
0_PL__model.json                  mcmc_analysis_CIV.csv
0_fitresult.json                  specmodel_0_FitAll_fit_report.txt
1_SiIV_A__model.json              specmodel_0_specdata.hdf5
1_SiIV_B__model.json              specmodel_1_FitAll_fit_report.txt
1_fitresult.json                  specmodel_1_specdata.hdf5
2_CIV_A__model.json               specmodel_2_FitAll_fit_report.txt
2_CIV_B__model.json               specmodel_2_specdata.hdf5
2_fitresult.json                  specmodel_3_FitAll_fit_report.txt
3_CIII__model.json                specmodel_3_specdata.hdf5
3_fitresult.json                  specmodel_4_FitAll_fit_report.txt
```

(continues on next page)

(continued from previous page)

```
4_Abs_A_model.json                specmodel_4_specdata.hdf5
4_Abs_B_model.json                specmodel_CIV_line_mcmc_chain.hdf5
4_fitresult.json                  spectrum.hdf5
fit.hdf5
```

The analyzed data is saved in the enhanced csv format from astropy. We use *astropy.table.QTable* to read the file in, retaining the unit information on the analyzed properties.

[17]:
```
from astropy.table import QTable

t = QTable.read('../example_spectrum_fit/mcmc_analysis_CIV.csv', format='ascii.ecsv')
t
```

[17]:
```
<QTable length=18750>
   CIV_peak_fluxden      CIV_peak_redsh   ...        CIV_lum
erg / (Angstrom cm2 s)                    ...        erg / s
       float64              float64       ...        float64
---------------------- ------------------ ... ----------------------
1.7025458953823565e-16 3.2114519863017534 ... 2.7272731887439896e+45
1.6721664115085754e-16  3.210481058669064 ...  2.716855532259717e+45
 1.723933433168788e-16 3.2114519863017534 ...  2.757010117996179e+45
1.7182794771269916e-16  3.210481058669064 ...  2.743871695879414e+45
1.6878815917188721e-16  3.210481058669064 ... 2.7316768614052704e+45
1.7320789412335437e-16 3.2114519863017534 ... 2.7826405341256157e+45
1.6987702663876353e-16 3.2114519863017534 ...  2.753574262946658e+45
1.7002049199831065e-16 3.2114519863017534 ... 2.7486143761102343e+45
1.6755933939029979e-16  3.210481058669064 ... 2.7467976530734148e+45
                   ...                ... ...                    ...
1.7230147389554056e-16 3.2114519863017534 ...  2.749285946205694e+45
 1.713119163521492e-16 3.2114519863017534 ...  2.768326236236442e+45
 1.699526629230714e-16  3.210481058669064 ...  2.761994684366228e+45
1.6981870118562772e-16 3.2114519863017534 ... 2.7377373005089886e+45
1.7190133988262977e-16  3.210481058669064 ... 2.7992449011384977e+45
1.7353516300351097e-16  3.210481058669064 ... 2.7691512911872792e+45
1.7204870649017424e-16 3.2114519863017534 ... 2.7260831013864665e+45
1.7059536939278595e-16 3.2114519863017534 ... 2.7229014157897432e+45
 1.699557133889922e-16  3.210481058669064 ... 2.7681753906977356e+45
1.6877095368987712e-16 3.2114519863017534 ...  2.749041419139746e+45
```

With the table data we can visualize the posterior distributions for all measurements and further analyze them.

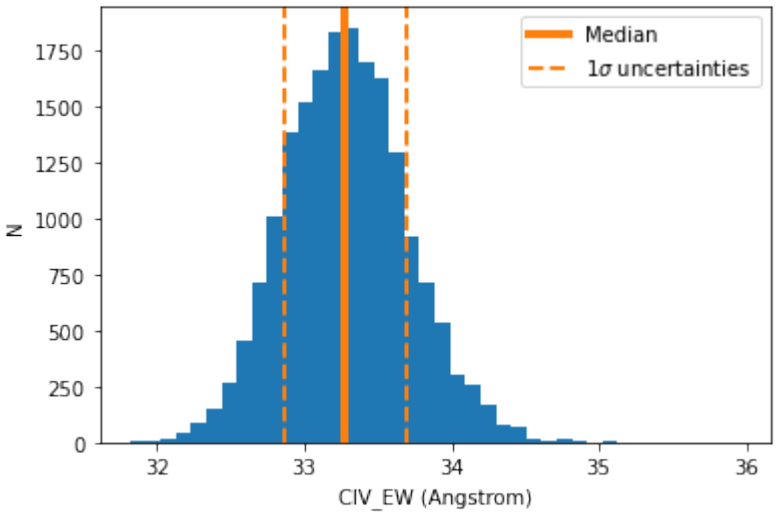
[18]:
```
import numpy as np
import matplotlib.pyplot as plt

prop = 'CIV_EW'

# Calculate median, lower and upper 1-sigma range
med = np.median(t[prop])
low = np.percentile(t[prop],16)
upp = np.percentile(t[prop],84)

print('Property: {}'.format(prop))
print('Median: {:.2e}'.format(med))
```

(continues on next page)

(continued from previous page)

```
print('Lower 1-sigma: {:.2e}'.format(low))
print('Upper 1-sigma: {:.2e}'.format(upp))

fig = plt.figure()
ax = fig.add_subplot(1,1,1)

ax.hist(t[prop].value, bins=40)
ax.axvline([med.value], ymin=0, ymax=1, color='#ff7f0e', lw=4, label='Median')
ax.axvline(low.value, ymin=0, ymax=1, color='#ff7f0e', lw=2, ls='--', label=r'$1\sigma$␣
↪uncertainties')
ax.axvline(upp.value, ymin=0, ymax=1, color='#ff7f0e', lw=2, ls='--')
plt.xlabel('{} ({})'.format(prop, med.unit))
plt.ylabel('N')
plt.legend()

plt.show()
```

```
Property: CIV_EW
Median: 3.33e+01 Angstrom
Lower 1-sigma: 3.29e+01 Angstrom
Upper 1-sigma: 3.37e+01 Angstrom
```



CHAPTER

TEN

THE SPECONED MODULE

This module introduces the SpecOneD class, it's functions and the related PassBand class. The main purpose of the SpecOneD class and it's children classes is to provide python functionality for the manipulation of 1D spectral data in astronomy.

class sculptor.speconed.**PassBand**(*passband_name=None*, *dispersion=None*, *fluxden=None*, *fluxden_err=None*, *header=None*, *dispersion_unit=None*, *fluxden_unit=None*)

The PassBand class, a child of the SpecOneD class, is a data structure for storing and manipulating astronomical filter transmission curves.

Parameters

- **passband_name** (*str*) – Name of the passband. The passband names provided with the Sculptor package are in the format [INSTRUMENT]-[BAND] and can be found in the Sculptor data folder.
- **dispersion** (*numpy.ndarray*) – A 1D array providing the dispersion axis of the passband.
- **fluxden** (*numpy.ndarray*) – A 1D array providing the transmission data of the spectrum in quantum efficiency.
- **fluxden_err** (*numpy.ndarray*) – A 1D array providing the 1-sigma error of the passband's transmission curve.
- **header** (*pandas.DataFrame*) – A pandas DataFrame containing additional information on the spectrum.
- **dispersion_unit** (*astropy.units.Unit or astropy.units.Quantity or astropy.units.CompositeUnit or astropy.units.IrreducibleUnit*) – The physical unit (including normalization factors) of the dispersion axis of the passband.
- **fluxden_unit** (*astropy.units.Unit or astropy.units.Quantity or astropy.units.CompositeUnit or astropy.units.IrreducibleUnit*) – The physical unit (including normalization factors) of the transmission curve and associated properties (e.g. flux density error) of the spectrum.

convert_spectral_units(*new_dispersion_unit*)

Convert the passband to new physical dispersion units.

This function only converts the passband dispersion axis.

Parameters
new_dispersion_unit –

Type
astropy.units.Unit or astropy.units.Quantity or astropy.units.CompositeUnit or astropy.units.IrreducibleUnit

Returns

load_passband(*passband_name*, *tolerance=0.005*)

Load a passband from the sculptor/data/passbands folder.

The passband names are in the following format: [INSTRUMENT]-[BAND]

Parameters

- **passband_name** (*str*) – Name of the passband, e.g. WISE-W1
- **tolerance** (*float*) – Value below which the passband throughput will be ignored when reading the passband in. In many cases the original passband files contain a large range of 0 values below and above the passband. The default value for the tolerance is 0.005, i.e. 0.5% throughput.

Returns

plot(*mask_values=False*, *ymin=0*, *ymax=1.1*)

Plot the passband.

This plot is aimed for a quick visualization of the passband spectrum and not for publication grade figures.

Parameters

- **mask_values** (*bool*) – Boolean to indicate whether the mask will be applied when plotting the spectrum (default:True).
- **ymin** (*float*) – Minimum value for the y-axis of the plot (flux density axis). This defaults to 'None'. If either ymin or ymax are 'None' the y-axis range will be determined automatically.
- **ymax** (*float*) – Maximum value for the y-axis of the plot (flux density axis). This defaults to 'None'. If either ymin or ymax are 'None' the y-axis range will be determined automatically.

Returns

class sculptor.speconed.**SpecOneD**(*dispersion=None*, *fluxden=None*, *fluxden_err=None*, *fluxden_ivar=None*, *header=None*, *dispersion_unit=None*, *fluxden_unit=None*, *obj_model=None*, *telluric=None*, *mask=None*)

The SpecOneD class provides a data structure for 1D astronomical spectra with extensive capabilities for spectrum analysis and manipulation.

Parameters

- **dispersion** (*numpy.ndarray*) – A 1D array providing the dispersion axis of the spectrum.
- **fluxden** (*numpy.ndarray*) – A 1D array providing the flux density data of the spectrum.
- **fluxden_err** (*numpy.ndarray*) – A 1D array providing the 1-sigma flux density error of the spectrum.
- **fluxden_ivar** (*numpy.ndarray*) – A 1D array providing the inverse variance of the flux density for the spectrum.
- **header** (*pandas.DataFrame*) – A pandas DataFrame containing additional information on the spectrum.
- **dispersion_unit** (*astropy.units.Unit or astropy.units.Quantity or astropy.units.CompositeUnit or astropy.units.IrreducibleUnit*) – The physical unit (including normalization factors) of the dispersion axis of the spectrum.

- **fluxden_unit** (*astropy.units.Unit or astropy.units.Quantity or astropy.units.CompositeUnit or astropy.units.IrreducibleUnit*) – The physical unit (including normalization factors) of the flux density and associated properties (e.g. flux density error) of the spectrum.
- **obj_model** (*numpy.ndarray*) – Object model from the telluric correction routine of PypeIt.
- **telluric** (*numpy.ndarray*) – Telluric (atmospheric transmission) model from the telluric correction routine of PypeIt.
- **mask** (*numpy.ndarray*) – A boolean 1D array specifying regions that allows to mask region in the spectrum during spectral manipulation or for display purposes.

Raises

ValueError – Raises a ValueError if the supplied header is not a pandas.DataFrame

apply_extinction(*a_v*, *r_v*, *extinction_law='ccm89'*, *inplace=False*)

Apply extinction to the spectrum (flux density ONLY).

This function makes use of the python extinction package: https://github.com/kbarbary/extinction .

Their documentation is available at https://extinction.readthedocs.io/en/latest/ .

Please have a careful look at their implementation and regarding details on the use of a_v and r_v. Possible extinction laws to use are "ccm89", "odonnell94", "calzetti00", "fitzpatrick99", "fm07".

Parameters

- **a_v** (*float*) – Extinction value ein the V band.
- **r_v** (*float*) – Ratio of total to selective extinction r_v = a_v/E(B-V)
- **extinction_law** (*str*) – Extinction law name as implemented in the extinction package, see documentation.
- **inplace** (*bool*) – Boolean to indicate whether the active SpecOneD object will be modified or a new SpecOneD object will be created and returned.

Returns

Returns the binned spectrum as a SpecOneD object if inplace==False.

Return type

SpecOneD

average_fluxden(*dispersion_range=None*)

Calculate the average flux density over the full spectrum or the specified dispersion range

Parameters

dispersion_range (*[float, float]*) – Dispersion range over which to average the flux density.

Returns

Average flux density of the full spectrum or specified dispersion range

Return type

float

bin_by_npixels(*npix*, *inplace=False*)

Bin the spectrum by an integer number of pixel.

The spectrum is binned by npix pixel. A new dispersion axis is calculated asumming that the old dispersion values marked the center positions of their bins.

The flux density (obj_model, telluric) are averaged over the new bin width, whereas the flux density error is accordingly propagated.

The spectrum mask will be automatically reset.

Parameters

- **npix** (*int*) – Number of pixels to be binned.
- **inplace** (*boolean*) – Boolean to indicate whether the active SpecOneD object will be modified or a new SpecOneD object will be created and returned.

Returns
Returns the binned spectrum as a SpecOneD object if inplace==False.

Return type
SpecOneD

broaden_by_gaussian(*fwhm*, *inplace=False*)

The spectrum is broadened by a Gaussian with the specified FWHM (in km/s).

The convolution of the current spectrum and the Gaussian is performed in logarithmic wavelength. Therefore, the spectrum is first converted to flux per logarithmic wavelength, then convolved with the Gaussian kernel and then converted back.

The conversion functions will automatically take care of the unit conversion and input spectra can be in flux density per unit frequency or wavelength.

This function normalizes the output of the convolved spectrum in a way that a Gaussian input signal of FWHM X broadened by a Gaussian kernel of FWHM Y, results in a Gaussian output signal of FWHM sqrt(X**2+Y**2) with the same amplitude as the input signal. Due to the normalization factor of the Gaussian itself, this results in a lower peak height.

The input spectrum and the Gaussian kernel are matched to the same dispersion axis using the 'interpolate' function.

Parameters

- **fwhm** (*float*) – FWHM of the Gaussian that the spectrum will be convolved with in km/s.
- **inplace** (*bool*) – Boolean to indicate whether the active SpecOneD object will be modified or a new SpecOneD object will be created and returned.

Returns
Returns the binned spectrum as a SpecOneD object if inplace==False.

Return type
SpecOneD

calculate_passband_ab_magnitude(*passband*, *match_method='interpolate'*, *force=False*)

Calculate the AB magnitude of the spectrum in the given passband.

AB magnitudes are calculated following equation 4 in https://arxiv.org/pdf/astro-ph/0210394.pdf

Parameters

- **passband** (PassBand) – The astronomical passband with throughput in quantum efficiencies.
- **match_method** (*str*) – Method for matching the dispersion axis of the spectrum to the passband.
- **force** (*bool*) – Boolean to indicate if they spectra will be forced to match if the spectrum does not fully cover the passband. The forced match will result in an inner match of the spectrum's and the passband's dispersion axes. User discretion is advised.

Returns
AB magnitude of the spectrum in the specified passband :rtype: float

calculate_passband_flux_density(*passband*, *match_method='interpolate'*, *force=False*)

Calculate the integrated flux in the specified passband.

Parameters

- **passband** (PassBand) – The astronomical passband with throughput in quantum efficiencies.
- **match_method** (*str*) – Method for matching the dispersion axis of the spectrum to the passband.
- **force** (*bool*) – Boolean to indicate if they spectra will be forced to match if the spectrum does not fully cover the passband. The forced match will result in an inner match of the spectrum's and the passband's dispersion axes. User discretion is advised.

Returns
Integrated spectrum flux in the passband

Return type
Quantity

check_dispersion_overlap(*secondary_spectrum*)

Check the overlap between the active spectrum and the supplied secondary spectrum.

This method determines whether the active spectrum (primary) and the supplied spectrum (secondary) have overlap in their dispersions. Possible cases include: i) The current spectrum dispersion is fully within the dispersion range of the 'secondary' spectrum -> 'primary' overlap. ii) The secondary spectrum dispersion is fully within the dispersion range of the current spectrum -> 'secondary' overlap. iii) and iv) There is only partial overlap between the spectra -> 'partial' overlap. v) There is no overlap between the spectra -> 'none' overlap. In the case of no overlap np.NaN values are returned for the minimum and maximum dispersion limits.

Parameters
secondary_spectrum (SpecOneD) –

Returns
overlap, overlap_min, overlap_max Returns a string indicating the dispersion overlap type according to the cases above 'overlap and the minimum and maximum dispersion value of the overlap region of the two spectra.

Return type
(str, float, float)

check_units(*spectrum*)

Raise a ValueError if current and input spectrum have different dispersion of flux density units.

Parameters
spectrum (SpecOneD) –

Returns

convert_spectral_units(*new_dispersion_unit*, *new_fluxden_unit*, *verbosity=0*)

Convert the spectrum to new physical dispersion and flux density units.

The function converts the flux density, the dispersion, the flux density error and the inverse variance. Object model and telluric if they exist will not be converted.

Parameters

- **new_dispersion_unit** (*astropy.units.Unit or astropy.units.Quantity or astropy.units.CompositeUnit or astropy.units.IrreducibleUnit*) – New dispersion unit (or quantity)
- **new_fluxden_unit** (*astropy.units.Unit or astropy.units.Quantity or astropy.units.CompositeUnit or astropy.units.IrreducibleUnit*) – New flux density unit (or quantity)

Returns

copy()

Copy the current SpecOneD object to a new instance and return it.

Returns

Return type
SpecOneD

create_dispersion_by_resolution(*resolution*)

This function creates a new dispersion axis in wavelength sampled by a fixed resolution, given in km/s.

This should work for all spectra with flux densities per unit wavelength/frequency.

Parameters
resolution –

Returns
Returns new dispersion axis with a resolution in km/s as given by the input value.

Return type
numpy.ndarray

get_fluxden_error_from_ivar()

Calculate the flux density 1-sigma error from the inverse variance of the flux density/

Returns

get_ivar_from_fluxden_error()

Calculate inverse variance of the flux density from the flux density 1-sigma error.

Returns

get_specplot_ylim()

Calculate the minimum and maximum flux density values for plotting
the spectrum.

The minimum value is set to -0.5 * median of the flux density. The maximum value is set 4 times the 84-percentile value of the flux density. This is an 'approximate guess' for a quick visualization of the spectrum and may not be optimal for all purposes. For pulication grade plots, the user should devise their own plots.

Returns
(ylim_min, ylim_max) Return the minimum and maximum values for the flux density (y-axis) for the plot function.

Return type
(float, float)

interpolate(*new_dispersion*, *kind='linear'*, *fill_value='const'*, *inplace=False*, *verbosity=0*)

Interpolate spectrum to a new dispersion axis.

The interpolation is done using scipy.interpolate.interp1d.

Interpolating a spectrum to a new dispersion axis automatically resets the spectrum mask.

Parameters

- **new_dispersion** (*numpy.ndarray*) – 1D array with the new dispersion axis
- **kind** (*str*) – String that indicates the interpolation function (default: 'linear')
- **fill_value** (*str*) – A string indicating whether values outside the dispersion range will be extrapolated ('extrapolate') or filled with a constant value ('const') based on the median of the 10 values at the edge.
- **inplace** (*bool*) – Boolean to indicate whether the active SpecOneD object will be modified or a new SpecOneD object will be created and returned.
- **verbosity** (*int*) – Integer indecating the verbosity level

Returns

mask_between(*limits*, *inplace=False*)

Mask spectrum between specified dispersion limits.

Parameters

- **limits** (*[float, float]*) – A list of two floats indicating the lower and upper dispersion limit to mask between.
- **inplace** (*bool*) – Boolean to indicate whether the active SpecOneD object will be modified or a new SpecOneD object will be created and returned.

Returns

mask_by_snr(*signal_to_noise_ratio*, *inplace=False*)

Mask all regions with a signal to noise below the specified limit

Parameters

- **signal_to_noise_ratio** (*float*) – All regions of the spectrum with a value below this limit will be masked.
- **inplace** (*bool*) – Boolean to indicate whether the active SpecOneD object will be modified or a new SpecOneD object will be created and returned.

Returns

SpecOneD

match_dispersions(*secondary_spectrum*, *match_secondary=True*, *force=False*, *method='interpolate'*, *interp_method='linear'*)

Match the dispersion of the current spectrum and the secondary spectrum.

Both, current and secondary, SpecOneD objects are modified in this process. The dispersion match identifies the maximum possible overlap in the dispersion direction of both spectra and automatically trims them to that range.

If the current (primary) spectrum overlaps fully with the secondary spectrum the dispersion of the secondary will be interpolated/resampled to the primary dispersion.

If the secondary spectrum overlaps fully with the primary, the primary spectrum will be interpolated/resampled on the secondary spectrum resolution, but this will only be executed if 'force==True' and 'match_secondary==False'.

If there is partial overlap between the spectra and 'force==True' the secondary spectrum will be interpolated/resampled to match the dispersion values of the primary spectrum.

If there is no overlap a ValueError will be raised.

Parameters

- **secondary_spectrum** (SpecOneD) – Secondary spectrum
- **match_secondary** (*bool*) – The boolean indicates whether the secondary will always be matched to the primary or whether reverse matching, primary to secondary is allowed.
- **force** (*bool*) – The boolean sets whether the dispersions are matched if only partial overlap between the spectral dispersions exists.
- **method** (*str*) –
- **interp_method** (*str*) –

Returns

Raises
ValueError – A ValueError will be raised if there is no overlap between the spectra.

normalize_fluxden_by_error(*inplace=False*)

Normalize the flux density, flux density error and object model numerical values by the median value of the flux density error array.

The flux density unit will be scaled accordingly. Hence, this normalization does not affect the physical values of the flux density and only serves to normalize the values in the flux density array.

This enables more efficient calculations on the flux density array by avoiding small numerical values.

Parameters
inplace (*bool*) – Boolean to indicate whether the active SpecOneD object will be modified or a new SpecOneD object will be created and returned.

Returns
SpecOneD

normalize_fluxden_by_factor(*factor*, *inplace=False*)

Normalize the flux density, flux density error and object model numerical values by the specified numerical factor.

The flux density unit will be scaled accordingly. Hence, this normalization does not affect the physical values of the flux density and only serves to normalize the values in the flux density array.

This enables more efficient calculations on the flux density array by avoiding small numerical values.

Parameters

- **factor** (*float*) – Scale factor by which the flux density, flux density error and object model will be divided and the flux density unit will be multiplied with.
- **inplace** (*bool*) – Boolean to indicate whether the active SpecOneD object will be modified or a new SpecOneD object will be created and returned.

Returns
SpecOneD

normalize_fluxden_to_factor(*factor*, *inplace=False*)

Normalize the flux density, flux density error and object model numerical values to the specified unit factor.

The flux density unit will be scaled accordingly. Hence, this normalization does not affect the physical values of the flux density and only serves to normalize the values in the flux density array.

For example normalizing the flux density to a factor 1e-17 will assure that the flux density unit is 1e-17 times the original unit of the flux density.

This enables more efficient calculations on the flux density array by avoiding small numerical values.

Parameters

- **factor** (*float*) – Scale factor by which the flux density, flux density error and object model will be divided and the flux density unit will be multiplied with.
- **inplace** (*bool*) – Boolean to indicate whether the active SpecOneD object will be modified or a new SpecOneD object will be created and returned.

Returns
SpecOneD

peak_dispersion()

Return the dispersion of the maximum flux density value in the spectrum.

Returns
Dispersion value of maximum flux density

Return type
float

peak_fluxden()

Return the maximum flux density value in the spectrum.

Returns
Maximum flux density value

Return type
float

plot(*show_fluxden_err=True*, *mask_values=True*, *ymin=None*, *ymax=None*, *show_obj_model=True*, *show_telluric=True*)

Plot the spectrum.

This plot is aimed for a quick visualization of the spectrum not for publication grade figures.

Parameters

- **show_fluxden_err** (*bool*) – Boolean to indicate whether the error will be plotted or not (default:True).
- **mask_values** (*bool*) – Boolean to indicate whether the mask will be applied when plotting the spectrum (default:True).
- **ymin** (*float*) – Minimum value for the y-axis of the plot (flux density axis). This defaults to 'None'. If either ymin or ymax are 'None' the y-axis range will be determined automatically.
- **ymax** (*float*) – Maximum value for the y-axis of the plot (flux density axis). This defaults to 'None'. If either ymin or ymax are 'None' the y-axis range will be determined automatically.
- **show_obj_model** (*bool*) – Boolean to indicate whether the object model will be plotted or not (default:True).
- **show_telluric** (*bool*) – Boolean to indicate whether the atmospheric model will be plotted or not (default:True).

Returns

read_from_fits(*filename*)

Read in an iraf fits spectrum as a SpecOneD object.

Parameters
filename (*str*) – Filename of the fits file.

Returns

Raises
ValueError – Raises an error when the filename could not be read in.

read_from_hdf(*filename*)

Read in a SpecOneD object from a hdf5 file.

Parameters
filename (*str*) – Filename from which to read the new SpecOneD object in.

Returns

read_pypeit_fits(*filename*, *exten=1*)

Read in a pypeit fits spectrum as a SpecOneD object.

Parameters

- **filename** (*string*) – Filename of the fits file.
- **exten** (*int*) – Extension of the pypeit fits file to read. This defaults to exten=1.

Returns

Raises
ValueError – Raises an error when the filename could not be read in.

read_sdss_fits(*filename*)

Read in an SDSS/BOSS fits spectrum as a SpecOneD object.

Parameters
filename (*str*) – Filename of the fits file.

Returns

Raises
ValueError – Raises an error when the filename could not be read in.

remove_extinction(*a_v*, *r_v*, *extinction_law='ccm89'*, *inplace=False*)

Remove extinction from spectrum (flux density ONLY).

This function makes use of the python extinction package: https://github.com/kbarbary/extinction .

Their documentation is available at https://extinction.readthedocs.io/en/latest/ .

Please have a careful look at their implementation and regarding details on the use of a_v and r_v. Possible extinction laws to use are “ccm89”, “odonnell94”, “calzetti00”, “fitzpatrick99”, “fm07”.

Parameters

- **a_v** (*float*) – Extinction value ein the V band.
- **r_v** (*float*) – Ratio of total to selective extinction r_v = a_v/E(B-V)
- **extinction_law** (*str*) – Extinction law name as implemented in the extinction package, see documentation.
- **inplace** (*bool*) – Boolean to indicate whether the active SpecOneD object will be modified or a new SpecOneD object will be created and returned.

Returns
Returns the binned spectrum as a SpecOneD object if inplace==False.

Return type
SpecOneD

renormalize_by_ab_magnitude(*magnitude*, *passband*, *match_method='interpolate'*, *force=False*, *output_mode='spectrum'*, *inplace=False*)

Scale the spectrum flux density and 1-sigma errors to the specified magnitude in the provided passband.

Parameters

- **magnitude** (*float*) – Magnitude to scale the spectrum to.
- **passband** (PassBand) – The astronomical passband with throughput in quantum efficiencies.
- **match_method** (*str*) – Method for matching the dispersion axis of the spectrum to the passband.
- **force** – Boolean to indicate if they spectra will be forced to match if the spectrum does not fully cover the passband. The forced match will result in an inner match of the spectrum's and the passband's dispersion axes. User discretion is advised. :type force: bool
- **output_mode** (*str*) – Output mode of the function. The default mode "Spectrum" returns the rescaled spectrum as a SpecOneD object or if inplace=True updates the provided spectrum. The alternative output mode "flux_factor" returns the factor to scale the flux with as a float.
- **inplace** (*bool*) – Boolean to indicate whether the active SpecOneD object will be modified or a new SpecOneD object will be created and returned.

Returns
Normalized spectrum or flux density normalization factor

renormalize_by_spectrum(*spectrum*, *dispersion_limits=None*, *output_mode='spectrum'*, *inplace=False*)

Scale the spectrum flux density and 1-sigma errors to match the provided spectrum in the full overlap region or in a specified dispersion range.

The original SpecOneD spectrum and the normalization spectrum should be in the same units. If this is not the case, the normalization spectrum will be converted to the same units as the original SpecOneD spectrum.

The dispersion limits are unitless (list of two floats) but need to be in the same units as the SpecOneD dispersion axis (dispersion_unit).

Parameters

- **spectrum** (SpecOneD) – The provided spectrum to scale the SpecOneD spectrum to.
- **dispersion_limits** (*(float, float)*) – A list of two floats indicating the lower and upper dispersion limits between which the spectra are normalized.
- **output_mode** (*str*) – Output mode of the function. The default mode "Spectrum" returns the rescaled spectrum as a SpecOneD object or if inplace=True updates the provided spectrum. The alternative output mode "flux_factor" returns the factor to scale the flux with as a float.
- **inplace** (*bool*) – Boolean to indicate whether the active SpecOneD object will be modified or a new SpecOneD object will be created and returned.

Returns

resample(*new_dispersion*, *force=False*, *inplace=False*)

Function for resampling spectra (and optionally associated uncertainties) onto a new wavelength basis.

This code is making use of SpectRes https://github.com/ACCarnall/SpectRes by Adam Carnall - damc@roe.ac.uk

The mask will be automatically reset.

If obj_model and telluric exist for the spectrum these will be linearly interpolated onto the new dispersion axis and NOT resampled.

Parameters

- **new_dispersion** (*numpy.ndarray*) – Array containing the new wavelength sampling desired for the spectrum or spectra.
- **force** (*bool*) – Boolean to force the resampling of the spectrum by reducing the new dispersion axis range to the old dispersion axis range.
- **inplace** (*bool*) – Boolean to indicate whether the active SpecOneD object will be modified or a new SpecOneD object will be created and returned.

Returns

Returns the resampled spectrum as a SpecOneD object if inplace==False.

Return type

SpecOneD

resample_to_resolution(*resolution*, *buffer=2*, *inplace=False*)

Resample spectrum at a specific resolution specified in km/s.

This should work for all spectra with flux densities per unit wavelength/frequency.

Parameters

- **resolution** (*float*) – Specified resolution in km/s
- **buffer** (*int*) – Integer value indicating how many pixels at the beginning and the end of the current spectrum will be omitted in the resampling process.
- **inplace** (*bool*) – Boolean to indicate whether the active SpecOneD object will be modified or a new SpecOneD object will be created and returned.

Returns

Returns the resampled spectrum as a SpecOneD object if inplace==False.

Return type

SpecOneD

reset_mask()

Reset the spectrum mask.

Returns

save_to_csv(*filename*, *outputformat='linetools'*)

Save SpecOneD object to a csv file.

Output formats: - default: All array-like SpecOneD data will be saved to the csv file. Standard columns include dispersion, flux density, and flux density error. Additional columns can be telluric or object model from spectra reduced with pypeit. - "linetools": In the linetool format dispersion, flux density and flux density error (if exists) are saved in three columns with names 'wave', 'flux', and 'error'.

WARNING: At present the SpecOneD data will be saved independent of its physical units. They will not be automatically converted to a common format. User discretion is advised as unit information might get lost if saving to csv.

Parameters

- **filename** (*str*) – Filename to save the SpecOneD object to.
- **outputformat** (*str*) – Format of the csv file. Possible formats include "linetools". All other inputs will save it to the default format.

Returns

save_to_hdf(*filename*)

Save a SpecOneD object to a hdf5 file.

SpecOneD hdf5 files have three extensions: - data: holding the array spectral information like dispersion, flux density, flux density error, flux density inverse variance, or mask - spec_meta: holding information on the spectral meta data, currently this includes the units of the dispersion and flux density axis. - header: If a header exists, it will be saved here.

Parameters
filename (*str*) – Filename to save the current SpecOneD object to.

Returns

smooth(*width*, *kernel='boxcar'*, *scale_sigma=True*, *inplace=False*)

Smoothing the flux density of the spectrum using a boxcar oder gaussian kernel.

This function uses astropy.convolution to convolve the spectrum with the selected kernel.

If scale_sigma=True, the fluxden error is scaled down according to sqrt(width).

Parameters

- **width** – Width (in pixels) of the kernel)
- **kernel** (*str*) – String indicating whether to use the Boxcar ("boxcar") or Gaussian ("gaussian") kernel.
- **scale_sigma** (*bool*) – Boolean to indicate whether to scale the fluxden error according to the width of the boxcar kernel.
- **inplace** (*bool*) – Boolean to indicate whether the active SpecOneD object will be modified or a new SpecOneD object will be created and returned.

Type
width: int

Returns

to_fluxden_per_unit_frequency_cgs()

Convert SpecOneD spectrum to flux density per unit frequency (Hz) in cgs units.

Returns

to_fluxden_per_unit_frequency_jy()

Convert SpecOneD spectrum to flux density per unit frequency (Hz) in Jy.

Returns

to_fluxden_per_unit_frequency_si()

Convert SpecOneD spectrum to flux density per unit frequency (Hz) in SI units.

Returns

to_fluxden_per_unit_wavelength_cgs()

Convert SpecOneD spectrum to flux density per unit wavelength (Angstroem) in cgs units.

Returns

trim_dispersion(*limits*, *mode='physical'*, *inplace=False*)

Trim the spectrum according to the dispersion limits specified.

Parameters

- **limits** (*[float, float] or [int, int]*) – A list of two floats indicating the lower and upper dispersion limit to trim the dispersion axis to.
- **mode** (*str*) – A string specifying whether the limits are in 'physical' values of the dispersion axis (e.g. Angstroem) or in pixel values.
- **inplace** (*bool*) – Boolean to indicate whether the active SpecOneD object will be modified or a new SpecOneD object will be created and returned.

Returns
Spectrum trimmed to the specified limits

Return type
SpecOneD

sculptor.speconed.**gaussian**(*x*, *amp*, *cen*, *sigma*, *shift*)

1-D Gaussian function

CHAPTER

ELEVEN

THE SPECMODEL MODULE

This module introduces the SpecModel class and its functionality. The SpecModel class is designed to fit models to an astronomical spectrum using LMFIT.

The SpecModel is always associated with a SpecFit object, which provides the foundational functionality for the fitting.

Notes

This module is in active development.

class sculptor.specmodel.**SpecModel**(*specfit*, *spectrum=None*, *redshift=0*)

Class holding information on models for the SpecFit class

specfit

Associated SpecFit object

Type
SpecFit

xlim

Wavelength limits for plotting

Type
list of float

ylim

Flux density limits for plotting

Type
list of float

spec

Astronomical spectrum as a SpecOneD object

Type
SpecOneD

redshift

Cosmological redshift of the astronomical object

Type
float

use_weights

Boolean to indicate whether fluxden errors will be used as weights for the fit.

Type
bool

`model_list`

List of LMFIT models

Type
list of Models

`params_list`

List of LMFIT Parameters for all LMFIT models.

Type
list of Parameters

`global_params`

Global parameters to be added to the all models in the Specmodel. Their main use is to provide variables and constraints for multiple individual models.

Type
Parameters

`color`

Color to use in the SpecModel plot.

Type
str

`model`

LMFIT SpecModel model. The global model including all models in the model_list.

Type
Model

`params`

LMFIT SpecModel parameters. The global parameter list including all parameters from all models.

Type
Parameters

`fit_result`

LMFIT ModelResult for the fit to the SpecModel

Type
ModelResult

`add_global_param`(*param_name*, *value=None*, *vary=True*, *min=-inf*, *max=inf*, *expr=None*)

Adding a "Global Parameter" to the SpecModel object

Parameters

- **param_name** (*str*) – Name of the global parameter
- **value** (*float,optional*) – Initial value of the global parameter
- **vary** (*bool,optional*) – Boolean to indicate whether the global parameter should be varied during the fit
- **min** (*float,optional*) – Minimum value for the global parameter
- **max** (*float,optional*) – Maximum value for the global parameter
- **expr** (*str, optional*) – Optional expression for the global parameter

Returns
None

add_mask_preset_to_fit_mask(*mask_preset_key*)

Adding a preset mask from the models_and_masks module to the fit.

Parameters
mask_preset_key (*str*) – Name of the preset mask in the mask_preset dictionary.

Returns
None

add_model(*model_name*, *prefix*, ***kwargs*)

Add a model to the SpecModel by using the built-in Sculptor models.

Parameters

- **model_name** –
- **prefix** –

Returns

add_wavelength_range_to_fit_mask(*disp_x1*, *disp_x2*)

Adding a wavelength region to the fit mask.

The dispersion region between the two dispersion values will be added to the fit mask.

Parameters

- **disp_x1** (*(float)*) – Dispersion value 1
- **disp_x2** (*(float)*) – Dispersion value 2

Returns

build_model()

Build the Specmodel model and parameters for the fit

Returns
None

delete_model(*index=None*)

Delete model (Model, Parameters) from the SpecModel object.

Parameters
index (*(int)*) – Index of model to remove from model_list and Parameters to remove from params_list (default index==”None”). If the index is None the last added model will be removed.

Returns
None

fit()

Fit the SpecModel to the astronomical spectrum

Returns
None

load(*foldername*, *specmodel_id*)

Load a SpecModel from the specified folder.

Parameters

- **foldername** (*str*) – Specified folder in which the SpecModel will be saved.
- **specmodel_id** (*str*) – Unique SpecModel identifier used in creating the filenames for the save files.

Returns
None

plot(*xlim=None*, *ylim=None*)

Plot the SpecModel

Returns
None

remove_global_param(*param_name*)

Remove "Global Parameter" from SpecModel object

Parameters
param_name (*str*) – Parameter name of the global parameter to remove.

Returns
None

reset_fit_mask()

Reset the fit mask based on the supplied astronomical spectrum.

Returns
None

reset_plot_limits(*fluxden=True*, *dispersion=True*)

Reset the plot limits based on the dispersion and flux density ranges of the spectrum.

Parameters

- **fluxden** (*boolean*) – Boolean to indicate whether to reset the flux density axis limits (default: True).
- **dispersion** (*boolean*) – Boolean to indicate whether to reset the dispersion axis limits (default: True).

Returns
None

save(*foldername*, *specmodel_id=0*)

Save the SpecModel object to a specified folder

Parameters

- **foldername** (*str*) – Specified folder in which the SpecModel will be saved.
- **specmodel_id** (*str*) – Unique SpecModel identifier used in creating the filenames for the save files.

Returns
None

save_fit_report(*foldername*, *specmodel_id=None*, *show=False*)

Save the fit report to a file in the specified folder

Parameters

- **foldername** (*str*) – Specified folder in which the fit report will be saved.

- **specmodel_id** (*str*) – Unique SpecModel identifier used in creating the filename for the fit report.
- **show** (*bool*) – Boolean to indicate whether the fit report should also be printed to the screen.

Returns
None

save_mcmc_chain(*foldername*, *specmodel_id=None*)

Save the values of the MCMC flat chain as an hdf5 file.

Fixed parameters in the model fit will be automatically added to the output file.

Parameters

- **foldername** (*str*) – Specified folder in which the fit report will be saved.
- **specmodel_id** (*str*) – Unique SpecModel identifier used in creating the filename for the fit report.

Returns
None

update_model_params_for_global_params()

Global parameters are added to the Model parameters.

Returns
None

update_params_from_fit_result()

Update all parameter values in the parameter list based on the fit result.

Individual model parameter, global parameters and even the super parameters of the associated SpecFit object will be updated based on the fit.

Returns
None

sculptor.specmodel.**fitting_methods = {'Adaptive Memory Programming for Global Optimization': 'ampgo', 'BFGS': 'bfgs', 'Basinhopping': 'basinhopping', 'Brute force method': 'brute', 'Cobyla': 'cobyla', 'Conjugate-Gradient': 'cg', 'Differential evolution': 'differential_evolution', 'Dual Annealing Optimization': 'dual_annealing', 'L-BFGS-B': 'lbfgsb', 'Least-Squares minimization': 'least_squares', 'Levenberg-Marquardt': 'leastsq', 'Maximum likelihood via Monte-Carlo Markov Chain': 'emcee', 'Nelder-Mead': 'nelder', 'Newton GLTR trust-region': 'trust-krylov', 'Powell': 'powell', 'Sequential Linear Squares Programming': 'slsqp', 'Simplicial Homology Global Optimization': 'shgo', 'Truncated Newton': 'tnc', 'Trust-region for constrained obtimization': 'trust-constr'}**

Dictionary of fitting methods

Fitting methods available for fitting in SpecFit based on the list of methods in LMFIT.

Type
dict

CHAPTER

TWELVE

THE SPECFIT MODULE

This module introduces the SpecFit class and its functionality. The SpecFit class is designed to facilitate complex model fits to astronomical spectra.

It is initialized with the supplied astronomical spectrum and can hold multiple SpecModel objects, which themselves hold the fit models and their parameters.

class sculptor.specfit.**SpecFit**(*spectrum=None*, *redshift=0*)

Base class for fitting of astronomical spectroscopic data.

The SpecFit class takes a SpecOneD object of an astronomical spectrum and allows complex models to be fit to it using the LMFIT module.

SpecModel objects will be added to the SpecFit class to hold information on the different models and parameters. Each SpecModel object will be consecutively fit to the astronomical spectrum.

spec

Astronomical spectrum as a SpecOneD object

Type
SpecOneD

xlim

Wavelength limits for plotting

Type
list of float

ylim

Flux density limits for plotting

Type
list of float

redshift

Cosmological redshift of the astronomical object

Type
float

fitting_method

Fitting method (default: 'Levenberg-Marquardt')

Type
str

colors

Float values to set colors for plotting

Type
numpy.ndarray of floats

super_params

Parameter list of "Super Parameters", which are global for the specfit class and are added as "Global Parameters" to all SpecModels

Type
lmfit.parameters

specmodels

List of SpecModel objects added to the SpecFit class.

Type
list of *SpecModel*

add_specmodel()

Add a SpecModel to the SpecFit class

Returns
None

add_super_param(*param_name*, *value=None*, *vary=True*, *min=-inf*, *max=inf*, *expr=None*)

Adding a "Super Parameter" to the SpecFit object.

Parameters

- **param_name** (*str*) – Name of the super parameter
- **value** (*float,optional*) – Initial value of the super parameter
- **vary** (*bool,optional*) – Boolean to indicate whether the super parameter should be varied during the fit
- **min** (*float,optional*) – Minimum value for the super parameter
- **max** (*float,optional*) – Maximum value for the super parameter
- **expr** (*str,optional*) – Optional expression for the super parameter

Returns
None

copy()

Copy the SpecFit object

Returns
SpecFit

delete_specmodel(*index=None*)

Delete the latest SpecModel or the one indicated by the index keyword argument from the SpecFit class.

Parameters
index (*int*) – Index of the SpecModel to delete in specmodels

Returns
None

fit(*save_results=False*, *foldername='.'*)

Fit all SpecModels consecutively

Parameters

- **save_results** (*bool*) – Boolean to indicate whether fit results will be saved.
- **foldername** (*str,optional*) – If "save_results==True" the fit results will be saved to the folder specified in foldername. This variable defaults to the current folder. If set to "None" fit results will not be saved.

Returns
None

get_result_dict()

Get the best-fit parameter values and return them as a dictionary

Returns
(dict) result_dict Dictionary with best-fit parameter values.

import_spectrum(*filename*, *filetype='IRAF'*)

Import an astronomical spectrum into SpecFit class

Currently the allowed 'filetype' options are: 'IRAF', 'PypeIt', 'SpecOneD', 'SDSS'

Note that the SpecFit class can be initialized a SpecOneD spectrum object, that can be constructed manually from the spectral format of choice.

Parameters

- **filename** (*str*) – Full file name of the astronomical spectrum
- **filetype** (*str*) – String specifying the type of the spectrum to select the appropriate read method.

Returns
None

load(*foldername*)

Load a full spectral fit (SpecFit) from a folder

This function overwrites all SpecModels, SpecFit parameters, and the astronomical spectrum.

Parameters
foldername (*str*) – Folder from which the SpecFit class will be loaded.

Returns
None

normalize_spectrum_by_error()

Normalize the flux density, flux density error and object model arrays of the spectrum by the median value of the flux density error array.

The flux density unit will be scaled accordingly. Hence, this normalization does not affect the physical values of the flux density and only serves to normalize the values in the flux density array.

This enables more efficient calculations on the flux density array by avoiding small numerical values.

Returns

normalize_spectrum_by_factor(*factor*)

Normalize the flux density, flux density error and object model arrays of the spectrum by the specified numerical factor.

The flux density unit will be scaled accordingly. Hence, this normalization does not affect the physical values of the flux density and only serves to normalize the values in the flux density array.

This enables more efficient calculations on the flux density array by avoiding small numerical values.

Parameters
factor –

Returns

normalize_spectrum_to_factor(*factor*)

Normalize the flux density, flux density error and object model arrays of the spectrum to the specified unit factor.

The flux density unit will be scaled accordingly. Hence, this normalization does not affect the physical values of the flux density and only serves to normalize the values in the flux density array.

This enables more efficient calculations on the flux density array by avoiding small numerical values.

Parameters
factor –

Returns

plot()

Plot the astronomical spectrum with all SpecModels

Returns
None

remove_super_param(*param_name*)

Remove "Super Parameter" from SpecFit object.

Parameters
param_name (*str*) – Parameter name of the super parameter to remove.

Returns
None

resample(*n_samples=100*, *save_result_plots=True*, *foldername='.'*, *seed=1234*)

Resample and fit the spectrum.

Resample the spectral flux on a pixel by pixel basis by assuming a Gaussian distribution of flux values around the measured flux value with a sigma equal to the flux uncertainty.

Fit all SpecModels to the resampled spectrum and record the best-fit values of all fit parameters. The fits are initialized with the current parameter values from all SpecModels.

All n_samples results for each parameter are saved in a hdf5 file with the filename 'resampled_fitting_results_[n_samples]_raw.hdf5'. Median, lower (15.9 percentile) and upper (84.1 percentile) values are calculated from each parameter distribution and saved in a csv/hdf5 file with the name 'resampled_fitting_results_[n_samples].hdf5/.csv'.

If fit result plots are enabled ('save_result_plots=True') then the best-fie value distributions for each parameters, including their median, lower and upper values are saved to '[foldername]/[parameter name]_results.pdf'.

Parameters

- **n_samples** (*int*) – Number of samples to draw
- **save_result_plots** (*bool*) – Boolean indicating whether histograms for all parameters should be saved in the specified folder.
- **foldername** (*str*) – Path to the folder where the result plots will be saved. This defaults to '.'.
- **seed** (*int*) – Random seed for initializing the numpy random number generator

Returns
None

save(*foldername*)

Save the spectral fit (SpecFit) to a folder.

Parameters
foldername (*str*) – Folder to which the SpecFit class will be saved.

Returns
None

update_specmodel_spectra()

Update all SpecModel spectra

This function updates the SpecModel spectra consecutively. Model fits from each SpecModel will be automatically subtracted/divided.

Note: Not only the dispersion and the fluxden, but also the mask will be updated.

Returns
None

update_specmodels()

Update SpecFit parameters in all SpecModels

CHAPTER

THIRTEEN

THE SPECANALYSIS MODULE

The SpecAnalysis module provides a range of functions for spectral analsysis in the context of the Sculptor packages SpecOneD, SpecFit, and SpecModel classes.

sculptor.specanalysis.**analyze_continuum**(*specfit*, *model_names*, *rest_frame_wavelengths*, *cosmology*, *redshift=None*, *dispersion=None*, *cont_meas=None*, *width=10*)

Calculate measurements of the continuum at a range of specified wavelengths for a spectral fit (SpecFit object).

At present this analysis assumes that the spectra are in the following units: flux density - erg/s/cm^2/AA dispersion - AA

Parameters

- **specfit** (*sculptor.specfit.SpecFit*) – SpecFit class object to extract the model flux from
- **model_names** (*list(string)*) – List of model names to create the emission feature flux from.
- **rest_frame_wavelengths** (*list(float)*) – Rest-frame wavelength of the emission feature
- **cosmology** (*astropy.cosmology.Cosmology*) – Cosmology for calculation of absolute properties
- **redshift** (*float*) – Redshift of the object. This keyword argument defaults to 'None', in which case the redshift from the SpecFit object is used.
- **dispersion** (*np.array*) – This keyword argument allows to input a dispersion axis (e.g., wavelengths) for which the model fluxes are calculated. The value defaults to 'None', in which case the dispersion from the SpecFit spectrum is being used.
- **cont_meas** (*list(string)*) – This keyword argument allows to specify the list of emission feature measurements. Currently possible measurements are ['peak_fluxden', 'peak_redsh', 'EW', 'FWHM', 'flux']. The value defaults to 'None' in which all measurements are calculated
- **width** (*[float, float]*) – Window width in dispersion units to calculate the average flux density in.

Returns
Dictionary with measurement results (with units)

Return type
dict

sculptor.specanalysis.**analyze_emission_feature**(*specfit*, *feature_name*, *model_names*, *rest_frame_wavelength*, *cont_model_names=None*, *redshift=None*, *dispersion=None*, *emfeat_meas=None*, *disp_range=None*, *cosmology=None*, *fwhm_method='spline'*)

Calculate measurements of an emission feature for a spectral fit (SpecFit object).

At present this analysis assumes that the spectra are in the following units: flux density - erg/s/cm^2/AA dispersion - AA

Parameters

- **specfit** (sculptor.specfit.SpecFit) – SpecFit class object to extract the model flux from
- **feature_name** (*string*) – Name of the emission feature, which will be used to name the resulting measurements in the output dictionary.
- **model_names** (*list*) – List of model names to create the emission feature flux from.
- **rest_frame_wavelength** (*float*) – Rest-frame wavelength of the emission feature
- **cont_model_names** (*list*) – List of model names to create the continuum flux model from. The continuum spectrum is for example used in the calculation of some emission feature properties (e.g. equivalent width).
- **redshift** (*float*) – Redshift of the object. This keyword argument defaults to 'None', in which case the redshift from the SpecFit object is used.
- **dispersion** (*np.array*) – This keyword argument allows to input a dispersion axis (e.g., wavelengths) for which the model fluxes are calculated. The value defaults to 'None', in which case the dispersion from the SpecFit spectrum is being used.
- **emfeat_meas** (*list*) – This keyword argument allows to specify the list of emission feature measurements. Currently possible measurements are ['peak_fluxden', 'peak_redsh', 'EW', 'FWHM', 'flux']. The value defaults to 'None' in which all measurements are calculated
- **disp_range** (*list*) – 2 element list holding the lower and upper dispersion boundaries for the integration
- **cosmology** (*astropy.cosmology class*) – Cosmology for calculating luminosities
- **fwhm_method** (*string*) – Method to use for calculating the FWHM. Possible values are 'sign' or 'spline' (default).

Returns
Dictionary with measurement results (with units)

Return type
dict

sculptor.specanalysis.**analyze_mcmc_results**(*foldername*, *specfit*, *continuum_dict*, *emission_feature_dictlist*, *redshift*, *cosmology*, *emfeat_meas=None*, *cont_meas=None*, *dispersion=None*, *width=10*, *concatenate=False*)

Analyze MCMC model fit results of specified continuum/feature models.

Results will be written to an enhanced csv file in the same folder, where the MCMC flat chain data resides.

Important: Only model functions that are sampled together can be analyzed together. This means that only model functions from ONE SpecModel can also be analyzed together. Additionally, only model functions for which all variable parameters have sampled by the MCMC fit are analyzed.

The following parameters should be specified in the *continuum_listdict*:

- 'model_names' - list of model function prefixes for the full continuum model
- 'rest_frame_wavelengths' - list of rest-frame wavelengths (float) for which fluxes, luminosities and magnitudes should be calculated

The other arguments for the *SpecAnalysis.analyze_continuum* are provided to the MCMC analysis function separately.

The following parameters should be specified in the *emission_feature_listdict*:

- 'feature_name' - name of the emission feature, which will be used to name the resulting measurements in the output file.
- 'model_names' - list of model names to create the emission feature model flux from.
- 'rest_frame_wavelength' - rest-frame wavelength of the emission feature.

Additionally, one can specify:

- 'disp_range' - 2 element list holding the lower and upper dispersion boundaries flux density integration.

Parameters

- **foldername** (*string*) – Path to the folder with the MCMC flat chain hdf5 files.
- **specfit** (sculptor.specfit.SpecFit) – Sculptor model fit (SpecFit object) containing the information about the science spectrum, the SpecModels and parameters.
- **continuum_dict** (*dictionary*) – The *continuum_listdict* holds the arguments for the *SpecAnalysis.analyze_continuum* function that will be called by this procedure.
- **emission_feature_dictlist** (*list of dictionary*) – The *emission_feature_listdict* hold the arguments for the *SpecAnalysis.analyze_emission_feature* functions that will be called by this procedure.
- **redshift** (*float*) – Source redshift
- **cosmology** (*astropy.cosmology.Cosmology*) – Cosmology for calculation of absolute properties
- **emfeat_meas** (*list(string)*) – This keyword argument allows to specify the list of emission feature measurements. Currently possible measurements are ['peak_fluxden', 'peak_redsh', 'EW', 'FWHM', 'flux']. The value defaults to 'None' in which all measurements are calculated
- **cont_meas** (*list(string)*) – This keyword argument allows to specify the list of emission feature measurements. Currently possible measurements are ['peak_fluxden', 'peak_redsh', 'EW', 'FWHM', 'flux']. The value defaults to 'None' in which all measurements are calculated
- **dispersion** (*np.array*) – This keyword argument allows to input a dispersion axis (e.g., wavelengths) for which the model fluxes are calculated. The value defaults to 'None', in which case the dispersion from the SpecFit spectrum is being used.
- **width** (*[float, float]*) – Window width in dispersion units to calculate the average flux density in.
- **concatenate** (*bool*) – Boolean to indicate whether the MCMC flat chain and the analysis results should be concatenated before written to file. (False = Only writes analysis results to file; True = Writes analysis results and MCMC flat chain parameter values to file)

Returns
None

sculptor.specanalysis.**analyze_resampled_results**(*specfit*, *foldername*, *resampled_df_name*, *continuum_dict*, *emission_feature_dictlist*, *redshift*, *cosmology*, *emfeat_meas=None*, *cont_meas=None*, *dispersion=None*, *width=10*, *concatenate=False*)

Analyze resampled model fit results for all specified continuum and feature models.

Results will be written to an enhanced csv file in the same folder, where the resampled raw data resides.

The following parameters should be specified in the *continuum_listdict*:

- 'model_names' - list of model function prefixes for the full continuum model
- 'rest_frame_wavelengths' - list of rest-frame wavelengths (float) for which fluxes, luminosities and magnitudes should be calculated

The other arguments for the *SpecAnalysis.analyze_continuum* are provided to the MCMC analysis function separately.

The following parameters should be specified in the *emission_feature_listdict*:

- 'feature_name' - name of the emission feature, which will be used to name the resulting measurements in the output file.
- 'model_names' - list of model names to create the emission feature model flux from.
- 'rest_frame_wavelength' - rest-frame wavelength of the emission feature.

Additionally, one can specify:

- 'disp_range' - 2 element list holding the lower and upper dispersion boundaries flux density integration.

Parameters

- **specfit** (sculptor.specfit.SpecFit) – Sculptor model fit (SpecFit object) containing the information about the science spectrum, the SpecModels and parameters.
- **foldername** (*string*) – Path to the folder with the resampled raw hdf5 file.
- **resampled_df_name** (*str*) – Filename of the resampled raw DataFrame saved in hdf5 format.
- **continuum_dict** (*dictionary*) – The *continuum_listdict* holds the arguments for the *SpecAnalysis.analyze_continuum* function that will be called by this procedure.
- **emission_feature_dictlist** (*list of dictionary*) – The *emission_feature_listdict* hold the arguments for the *SpecAnalysis.analyze_emission_feature* functions that will be called by this procedure.
- **redshift** (*float*) – Source redshift
- **cosmology** (*astropy.cosmology.Cosmology*) – Cosmology for calculation of absolute properties
- **emfeat_meas** (*list(string)*) – This keyword argument allows to specify the list of emission feature measurements. Currently possible measurements are ['peak_fluxden', 'peak_redsh', 'EW', 'FWHM', 'flux']. The value defaults to 'None' in which all measurements are calculated
- **cont_meas** (*list(string)*) – This keyword argument allows to specify the list of emission feature measurements. Currently possible measurements are ['peak_fluxden', 'peak_redsh',

'EW', 'FWHM', 'flux']. The value defaults to 'None' in which all measurements are calculated

- **dispersion** (*np.array*) – This keyword argument allows to input a dispersion axis (e.g., wavelengths) for which the model fluxes are calculated. The value defaults to 'None', in which case the dispersion from the SpecFit spectrum is being used.
- **width** (*[float, float]*) – Window width in dispersion units to calculate the average flux density in.
- **concatenate** (*bool*) – Boolean to indicate whether the MCMC flat chain and the analysis results should be concatenated before written to file. (False = Only writes analysis results to file; True = Writes analysis results and MCMC flat chain parameter values to file)

Returns
None

sculptor.specanalysis.**build_model_flux**(*specfit*, *model_list*, *dispersion=None*)

Build the model flux from a specified list of models that exist in the SpecModels of the SpecFit object.

The dispersion axis for the model flux can be specified as a keyword argument.

Parameters

- **specfit** (sculptor.specfit.SpecFit) – SpecFit class object to extract the model flux from
- **model_list** (*list(string)*) – List of model names to create the model flux from. The models must be exist in the SpecModel objects inside the SpecFit object.
- **dispersion** (*np.array*) – New dispersion to create the model flux for

Returns
SpecOneD objects with the model flux

Return type
sod.SpecOneD

sculptor.specanalysis.**calc_absolute_mag_from_apparent_mag**(*appmag*, *cosmology*, *redshift*, *kcorrection=True*, *a_nu=0*)

Calculate the absolute magnitude from the apparent magnitude using a power law k-correction.

Parameters

- **appmag** (*float*) – Apparent AB magnitude
- **cosmology** (*astropy.cosmology.Cosmology*) – Cosmology as an astropy Cosmology object.
- **redshift** (*float*) – Redshift of the source.
- **kcorrection** (*bool*) – Boolean to indicate whether the magnitude should be k-corrected assuming a power law spectrum. This keyword argument defaults to 'True'.
- **a_nu** (*float*) – Power law slope as a function of frequency for the k-correction. This defaults to '0', appropriate for monochromatic measurements.

Returns
Absolute AB magnitude (monochromatic)

Return type
astropy.units.Quantity

sculptor.specanalysis.**calc_absolute_mag_from_fluxden**(*fluxden*, *dispersion*, *cosmology*, *redshift*, *kcorrection=True*, *a_nu=0*)

Calculate the absolute AB magnitude from the monochromatic flux density at a given dispersion value.

Parameters

- **fluxden** (*astropy.units.Quantity*) – Monochromatic flux density at a given wavelength.
- **dispersion** (*float*) – Dispersion value (usually in wavelength).
- **cosmology** (*astropy.cosmology.Cosmology*) – Cosmology as an astropy Cosmology object.
- **redshift** (*float*) – Redshift of the source.
- **kcorrection** (*bool*) – Boolean to indicate whether the magnitude should be k-corrected assuming a power law spectrum. This keyword argument defaults to 'True'.
- **a_nu** (*float*) – Power law slope as a function of frequency for the k-correction. This defaults to '0', appropriate for monochromatic measurements.

Returns
Absolute AB magnitude (monochromatic)

Return type
astropy.units.Quantity

sculptor.specanalysis.**calc_absolute_mag_from_monochromatic_luminosity**(*l_wav*, *wavelength*)

Calculate the absolute monochromatic magnitude from the monochromatic luminosity per wavelegnth.

Parameters

- **l_wav** (*astropy.units.Quantity*) – Monochromatic luminosity per wavelength
- **wavelength** (*float*) – Wavelength of the monochromatic luminosity

Returns
Absolute monochromatic magnitude

Return type
astropy.units.Quantity

sculptor.specanalysis.**calc_apparent_mag_from_fluxden**(*fluxden*, *dispersion*)

Calculate the apparent AB magnitude from the monochromatic flux density at a specified dispersion value.

Parameters

- **fluxden** (*astropy.units.Quantity*) – Monochromatic flux density at a given wavelength.
- **dispersion** (*float*) – Dispersion value (usually in wavelength).

Returns
Returns apparent AB magnitude.

Return type
astropy.units.Quantity

sculptor.specanalysis.**calc_integrated_luminosity**(*input_spec*, *redshift*, *cosmology*, *disp_range=None*)

Calculate the integrated model spectrum luminosity.

Parameters

- **input_spec** (sculptor.speconed.SpecOneD) – Input spectrum

- **redshift** (*float*) – Redshift of the source.
- **cosmology** (*astropy.cosmology.Cosmology*) – Cosmology as an astropy Cosmology object
- **disp_range** (*[float, float]*) – 2 element list holding the lower and upper dispersion boundaries for the integration

Returns
Return the integrated luminosity for (dispersion range in) the input spectrum.

Return type
astropy.units.Quantity

sculptor.specanalysis.**calc_lwav_from_fwav**(*fluxden*, *redshift*, *cosmology*)

Calculate the monochromatic luminosity from the monochromatic flux density.

Parameters

- **fluxden** (*astropy.units.Unit or astropy.units.Quantity or astropy.units.CompositeUnit or astropy.units.IrreducibleUnit*) – Monochromatic flux density at a given wavelength.
- **redshift** (*float*) – Redshift of the source.
- **cosmology** (*astropy.cosmology.Cosmology*) – Cosmology as an astropy Cosmology object

Returns
Monochromatic luminosity in units of erg s^-1 Angstroem^-1

Return type
astropy.units.Quantity

sculptor.specanalysis.**get_average_fluxden**(*input_spec*, *dispersion*, *width=10*, *redshift=0*)

Calculate the average flux density of a spectrum in a window centered at the specified dispersion and with a given width.

The central dispersion and width can be specified in the rest-frame and then are redshifted to the observed frame using the redsh keyword argument.

Warning: this function currently only works for spectra in wavelength units. For spectra in frequency units the conversion to rest-frame will be incorrect.

Parameters

- **input_spec** (sculptor.speconed.SpecOneD) – Input spectrum
- **dispersion** (*float*) – Central dispersion
- **width** (*float*) – Width of the dispersion window
- **redshift** (*float*) – Redshift argument to redshift the dispersion window into the observed frame.

Returns
Average flux density

Return type
astropy.units.Quantity

sculptor.specanalysis.**get_equivalent_width**(*cont_spec*, *line_spec*, *disp_range=None*, *redshift=0*)

Calculate the rest-frame equivalent width of a spectral feature.

Warning: this function currently only works for spectra in wavelength units. For spectra in frequency units the conversion to rest-frame will be incorrect.

Parameters

- **cont_spec** (sculptor.speconed.SpecOneD) – Continuum spectrum
- **line_spec** (sculptor.speconed.SpecOneD) – Spectrum with the feature (e.g. emission line)
- **disp_range** (*[float, float]*) – Dispersion range (2 element list of floats) over which the equivalent width will be calculated.
- **redshift** – Redshift of the astronomical source

Returns
Rest-frame equivalent width

Return type
astropy.units.Quantity

sculptor.specanalysis.**get_fwhm**(*input_spec*, *disp_range=None*, *resolution=None*, *method='spline'*)

Calculate the FWHM (in km/s) of an emission feature from the spectrum.

The user can specify a dispersion range to limit the FWHM calculation to this part of the spectrum. If a resolution (R) is specified the FWHM is corrected for the broadening due to the resolution.

The function will subtract a flux density value of half of the maximum and then find the two roots (flux density = 0) of the new flux density axis. If the emission feature has multiple components more than two roots can be found in which case the a np.NaN value will be returned.

Parameters

- **input_spec** (sculptor.speconed.SpecOneD) – Input spectrum
- **disp_range** (*[float, float]*) – Dispersion range to which the calculation is limited.
- **resolution** (*float*) – Resolution in R = Lambda/Delta Lambda
- **method** (*string*) – Method to use in retrieving the FWHM. There are two methods available. The default method 'spline' uses a spline to interpolate the original spectrum and find the zero points using a root finding algorithm on the spline. The second method 'sign' finds sign changes in the half peak flux subtracted spectrum.

Returns
FWHM of the spectral feature

Return type
astropy.units.Quantity

sculptor.specanalysis.**get_integrated_flux**(*input_spec*, *disp_range=None*)

Calculate the integrated flux of a spectrum.

The keyword argument disp_range allows to specify the dispersion boundaries for the integration. The standard numpy.trapz function is used for the integration.

Parameters

- **input_spec** (sculptor.speconed.SpecOneD) – SpecOneD object holding the spectrum to integrate

- **disp_range** (*[float, float]*) – 2 element list holding the lower and upper dispersion boundaries for the integration

Returns
Integrated flux

Return type
astropy.units.Quantity

sculptor.specanalysis.**get_nonparametric_measurements**(*input_spec*, *line_rest_wavel*, *redshift*, *disp_range=None*)

Measure the velocities at different ratios of the total emission line flux.

These velocity measurements are referenced in the literature by (e.g.) Whittle+1985, Liu+2013, Zakamska & Greene 2014.

This function calculates the cumulative integral of the emission line flux and then determines the closest dispersion values in velocity space to the 5%, 10%, 50%, 90% and 95% total flux ratios.

Parameters

- **input_spec** (sculptor.speconed.SpecOneD) – Input spectrum
- **line_rest_wavel** (*float*) – rest-frame wavelength fo the line in Angstroem
- **redshift** (*float*) – Redshift of the source
- **disp_range** (*[float, float]*) – Observed-frame dispersion range to which the calculation is limited.

Returns
median velocity, 5% velocity, 10% velocity, 90% velocity, 95% velocity, velocity resolution at median velocity, frequency of median velocity, wavelength of median velocity, redshift of median velocity

Return type
astropy.units.Quantity, astropy.units.Quantity, astropy.units.Quantity, astropy.units.Quantity, astropy.units.Quantity, astropy.units.Quantity, astropy.units.Quantity, astropy.units.Quantity, astropy.units.Quantity

sculptor.specanalysis.**get_peak_redshift**(*input_spec*, *rest_wave*)

Calculate the redshift of the flux density peak in the spectrum by specifying the expected rest frame wavelength of the emission feature.

Parameters

- **input_spec** (sculptor.speconed.SpecOneD) – Input spectrum
- **rest_wave** – Rest-frame wavelength of the expected emission feature.

Returns
Redshift of the peak flux density

Return type
float

sculptor.specanalysis.**k_correction_pl**(*redshift*, *a_nu*)

Calculate the k-correction for a power law spectrum with spectral index (per frequency) a_nu.

Parameters

- **redshift** (*float*) – Cosmological redshift of the source
- **a_nu** (*float*) – Power law index (per frequency)

Returns
K-correction

Return type
float

CHAPTER

FOURTEEN

THE MASKS & MODELS MODULE

sculptor.masksmodels.**constant**(*x*, *amp*)

Constant model

Parameters

- **x** (*np.ndarray*) – Dispersion
- **amp** (*float*) – Amplitude of the constant model

Returns
Constant model

Return type
np.ndarray

sculptor.masksmodels.**gaussian**(*x*, *amp*, *cen*, *sigma*, *shift*)

Basic Gaussian line model

The Gaussian is not normalized.

Parameters

- **x** (*np.ndarray*) – Dispersion
- **amp** (*float*) – Amplitude of the Gaussian
- **cen** (*float*) – Central dispersion of the Gaussian
- **sigma** (*float*) – Width of the Gaussian in sigma
- **shift** (*float*) – Shift of the Gaussian in dispersion units

Returns
Gaussian line model

Return type
np.ndarray

sculptor.masksmodels.**lorentzian**(*x*, *amp*, *cen*, *gamma*, *shift*)

Basic Lorentzian line model

Parameters

- **x** (*np.ndarray*) – Dispersion
- **amp** (*float*) – Amplitude of the Lorentzian
- **cen** (*float*) – Central dispersion of the Lorentzian
- **gamma** – Lorentzian Gamma parameter

- **shift** (*float*) – Shift of the Lorentzian in dispersion units

Returns
Gaussian line model

Return type
np.ndarray

sculptor.masksmodels.**mask_presets = {'My mask': {'mask_ranges': [[1265, 1290], [1340, 1375], [1425, 1470], [1680, 1705], [1905, 2050]], 'name': 'My mask', 'rest_frame': True}, 'QSO Cont.W. VP06': {'mask_ranges': [[1265, 1290], [1340, 1375], [1425, 1470], [1680, 1705], [1950, 2050]], 'name': 'QSO Cont. VP06', 'rest_frame': True}, 'QSO Continuum+FeII': {'mask_ranges': [[1350, 1360], [1445, 1465], [1690, 1705], [2480, 2650], [2925, 3090], [4200, 4230], [4435, 4700], [5100, 5535], [6000, 6250], [6800, 7000]], 'name': 'QSO Continuum+FeII', 'rest_frame': True}, 'QSO Fe+Cont.W. CIV Shen11': {'mask_ranges': [[1445, 1465], [1700, 1705]], 'name': 'QSO Cont. CIV Shen11', 'rest_frame': True}, 'QSO Fe+Cont.W. HAlpha Shen11': {'mask_ranges': [[6000, 6250], [6800, 7000]], 'name': 'QSO Cont. HAlpha Shen11', 'rest_frame': True}, 'QSO Fe+Cont.W. HBeta Shen11': {'mask_ranges': [[4435, 4700], [5100, 5535]], 'name': 'QSO Cont. HBeta Shen11', 'rest_frame': True}, 'QSO Fe+Cont.W. MgII Shen11': {'mask_ranges': [[2200, 2700], [2900, 3090]], 'name': 'QSO Cont. MgII Shen11', 'rest_frame': True}}**

Automatic import of extensions into Sculptor

sculptor.masksmodels.**model_func_dict = {'CIII_complex_model_func': <function CIII_complex_model_func>, 'constant': <function constant>, 'gaussian': <function gaussian>, 'line_model_gaussian': <function line_model_gaussian>, 'line_model_gaussian_nii_doublet': <function line_model_gaussian_nii_doublet>, 'line_model_gaussian_oiii_doublet': <function line_model_gaussian_oiii_doublet>, 'line_model_gaussian_sii_doublet': <function line_model_gaussian_sii_doublet>, 'lorentzian': <function lorentzian>, 'my_model': <function my_model>, 'power_law': <function power_law>, 'power_law_at_2500': <function power_law_at_2500>, 'power_law_at_2500_plus_bc': <function power_law_at_2500_plus_bc>, 'power_law_at_2500_plus_fractional_bc': <function power_law_at_2500_plus_fractional_bc>, 'template_model': <function template_model>}**

Dictionary of model setup function names

Type
dict

sculptor.masksmodels.**model_func_list = ['Constant (amp)', 'Power Law (amp, slope)', 'Gaussian (amp, cen, sigma, shift)', 'Lorentzian (amp, cen, gamma, shift)', 'Power Law (2500A)', 'Power Law (2500A) + BC', 'Power Law (2500A) + BC (fractional)', 'Line model Gaussian', 'SiIV (2G components)', 'CIV (2G components)', 'MgII (2G components)', 'HBeta (2G components)', 'HAlpha (2G components)', '[OIII] doublet (2G)', '[NII] doublet (2G)', '[SII] doublet (2G)', 'CIII] complex (3G components)', 'FeII template 2200-3500 (T06, cont)', 'FeII template 2200-3500 (T06, split)', 'FeII template 3700-7480 (BG92, cont)', 'FeII template 3700-5600 (BG92, split)', 'My Model']**

Dictionary of model functions

Type
dict

sculptor.masksmodels.**model_setup_list = [<function setup_constant>, <function setup_power_law>, <function setup_gaussian>, <function setup_lorentzian>, <function setup_power_law_at_2500>, <function setup_power_law_at_2500_plus_bc>, <function setup_power_law_at_2500_plus_fractional_bc>, <function setup_line_model_gaussian>, <function setup_line_model_SiIV_2G>, <function setup_line_model_CIV_2G>, <function setup_line_model_MgII_2G>, <function setup_line_model_Hbeta_2G>, <function setup_line_model_Halpha_2G>, <function setup_doublet_line_model_oiii>, <function setup_doublet_line_model_nii>, <function setup_doublet_line_model_sii>, <function setup_line_model_CIII_complex>, <function setup_iron_template_MgII_T06>, <function setup_split_iron_template_MgII_T06>, <function setup_iron_template_OPT_BG92>, <function setup_split_iron_template_OPT_BG92>, <function setup_my_model>]**

Dictionary of mask presets

Type
dict

sculptor.masksmodels.**power_law**(*x*, *amp*, *slope*)

Parameters

- **x** (*np.ndarray*) – Dispersion
- **amp** (*float*) – Amplitude of the power law
- **slope** (*float*) – Slope of the power law

Returns
Power law model

Return type
np.ndarray

sculptor.masksmodels.**setup_constant**(*prefix*, ***kwargs*)

Set up a simple constant function model.

Parameters

- **prefix** (*string*) – Model prefix
- **kwargs** – Keyword arguments

Returns
LMFIT model and parameters

Return type
(lmfit.Model, lmfit.Parameters)

sculptor.masksmodels.**setup_gaussian**(*prefix*, ***kwargs*)

Set up a simple non-normalized Gaussian function model.

Parameters

- **prefix** (*string*) – Model prefix
- **kwargs** – Keyword arguments

Returns
LMFIT model and parameters

Return type
(lmfit.Model, lmfit.Parameters)

sculptor.masksmodels.**setup_lorentzian**(*prefix*, ***kwargs*)

Set up a simple Lorentzian function model.

Parameters

- **prefix** (*string*) – Model prefix
- **kwargs** – Keyword arguments

Returns
LMFIT model and parameters

Return type
(lmfit.Model, lmfit.Parameters)

sculptor.masksmodels.**setup_power_law**(*prefix*, ***kwargs*)

Set up a simple power law model.

Parameters

- **prefix** (*string*) – Model prefix
- **kwargs** – Keyword arguments

Returns
LMFIT model and parameters

Return type
(lmfit.Model, lmfit.Parameters)

CHAPTER

FIFTEEN

THE QSO-EXTENSION MODULE

The Sculptor Quasar Extension

This module defines models, masks and analysis routines specific for the analysis of type-I QSOs (quasars, type-I AGN).

At first we define the basic model functions and their setups. The setup functions initialize LMFIT models and parameters using the basic model functions defined here.

Complex models can be constructed by combining multiple of the basic model functions. For example, we define a setup function to initialize a model for the Hbeta and [OIII] lines consistent of six Gaussian emission lines.

sculptor_extensions.qso.**CIII_complex_model_func**(*x*, *z*, *cen*, *cen_alIII*, *cen_siIII*, *flux*, *fwhm_km_s*, *shift_km_s*, *flux_alIII*, *fwhm_km_s_alIII*, *shift_km_s_alIII*, *flux_siIII*, *fwhm_km_s_siIII*, *shift_km_s_siIII*)

Model function for the CIII] emission line complex, consisting of the Gaussian line models with a combined redshift parameter.

The width of the line is set by the FWHM in km/s.

The Gaussians are not normalized.

Parameters

- **x** (*np.ndarray*) – Dispersion of the template model
- **z** (*float*) – Redshift for CIII], AlIII, SiIII]
- **cen** (*float*) – CIII] central wavelength
- **cen_alIII** (*float*) – AlIII central wavelength
- **cen_siIII** (*float*) – SiIII] central wavelength
- **amp** (*float*) – Amplitude of the CIII] line
- **fwhm_km_s** (*float*) – Full Width at Half Maximum (FWHM) of CIII] in km/s
- **shift_km_s** (*float*) – Doppler velocity shift of the central wavelength
- **amp_alIII** – Amplitude of the AlIII line
- **fwhm_km_s_alIII** (*float*) – Full Width at Half Maximum (FWHM) of AlIII in km/s
- **shift_km_s_alIII** – Doppler velocity shift of the central wavelength
- **amp_siIII** – Amplitude of the SiIII] line
- **fwhm_km_s_siIII** (*float*) – Full Width at Half Maximum (FWHM) of SiIII] in km/s
- **shift_km_s_siIII** – Doppler velocity shift of the central wavelength

Returns
CIII] complex model

sculptor_extensions.qso.**add_redshift_param**(*redshift*, *params*, *prefix*)

Add the redshift parameter to the LMFIT parameters.

Parameters

- **redshift** (*float*) – Redshift
- **params** (*lmfit.Parameters*) – Model parameters
- **prefix** (*string*) – Model prefix

Returns

sculptor_extensions.qso.**balmer_continuum_model**(*x*, *z*, *amp_be*, *Te*, *tau_be*, *lambda_be*)

Model of the Balmer continuum (Dietrich 2003)

This model is defined for a spectral dispersion axis in Angstroem.

This functions implements the Balmer continuum model presented in Dietrich 2003. The model follows a blackbody below the Balmer edge (3646A) and is zero above.

The amplitude parameter amp_be defines the amplitude at the balmer edge. The strength of the Balmer continuum can be estimated from the fluxden density at 3675A after subtraction of the power-law continuum component for reference see Grandi et al.(1982), Wills et al.(1985) or Verner et al.(1999).

At wavelengths of 3646A higher order Balmer lines are merging. This has not been included in this model and thus it will produce a sharp break at the Balmer edge.

Parameters

- **x** (*np.ndarray*) – Dispersion of the Balmer continuum
- **z** (*float*) – Redshift
- **amp_be** (*float*) – Amplitude of the Balmer continuum at the Balmer edge
- **Te** (*float*) – Electron temperature
- **tau_be** (*float*) – Optical depth at the Balmer edge
- **lambda_be** (*float*) – Wavelength of the Balmer edge

Returns
Balmer continuum model

Return type
np.ndarray

sculptor_extensions.qso.**calc_bolometric_luminosity**(*cont_lwav*, *cont_wav*, *reference='Shen2011'*)

Calculate the bolometric luminosity from the monochromatic continuum luminosity (erg/s/A) using bolometric correction factors from the literature.

The following bolometric corrections are available cont_wav = 1350, reference = Shen2011 cont_wav = 3000, reference = Shen2011 cont_wav = 5100, reference = Shen2011

The Shen et al. 2011 (ApJ, 194, 45) bolometric corrections are based on the composite spectral energy distribution (SED) in Richards et al. 2006 (ApJ, 166,470).

Parameters

- **cont_lwav** (*astropy.units.Quantity*) – Monochromatic continuum luminosity in erg/s/A.

- **cont_wav** (*astropy.units.Quantity*) – Wavelength of the monochromatic continuum luminosity in A.
- **reference** (*string*) – A reference string to select from the available bolometric corrections.

Returns
Returns a tuple of the bolometric luminosity in erg/s and a reference string indicating the publication and continuum wavelength of the bolometric correction.

Return type
astropy.units.Quantity, string

sculptor_extensions.qso.**calc_eddington_luminosity**(*bh_mass*)

Calculate the Eddington luminosity for a given black hole mass.

Parameters
bh_mass (*u.Quantity*) – Black hole mass as an astropy Quantity

Returns
Returns the Eddington luminosity in cgs units (erg/s).

Return type
u.Quantity

sculptor_extensions.qso.**calc_eddington_ratio**(*lbol*, *bh_mass*)

Calculate the Eddington ratio for a provided bolometric luminosity and black hole mass.

Parameters

- **lbol** (*u.Quantity*) – Bolometric luminosity
- **bh_mass** (*u.Quantity*) – Black hole mass

Returns

sculptor_extensions.qso.**calc_velocity_shifts**(*z*, *z_sys*, *relativistic=True*)

Calculate the velocity difference of a feature redshift with respect to the systemic redshift.

This function is currently simply a wrapper around the linetools functions calculating the velocity difference.

Parameters

- **z** (*float*) – The redshift of the spectroscopic feature (e.g., absorption or emission line).
- **z_sys** (*float*) – The systemic redshift
- **relativistic** – Boolean indicating whether the doppler velocity is calculated assuming relativistic velocities.

Type
bool

Returns
Returns the velocity difference in km/s.

Return type
u.Quantity

sculptor_extensions.qso.**correct_CIV_fwhm_for_blueshift**(*civ_fwhm*, *blueshift*)

Correct the CIV FWHM for the CIV blueshifts using the relation determined in Coatman et al. (2017, MNRAS, 465, 2120).

The correction follows Eq. 4 of Coatman et al. (2017).

Parameters

- **civ_fwhm** (*astropy.units.Quantity*) – FWHM of the CIV line in km/s
- **blueshift** (*astropy.units.Quantity*) – Blueshift of the CIV line in km/s

Returns
Corrected FWHM in km/s

Return type
astropy.units.Quantity

sculptor_extensions.qso.**line_model_gaussian**(*x*, *z*, *flux*, *cen*, *fwhm_km_s*)

Gaussian line model

The central wavelength of the Gaussian line model is determined by the central wavelength cen and the redshift, z. These parameters are degenerate in a line fit and it is adviseable to fix one of them (to predetermined values e.g., the redshift or the central wavelength).

The width of the line is set by the FWHM in km/s.

The Gaussian is normalized.

Parameters

- **x** (*np.ndarray*) – Dispersion of the continuum model
- **z** (*float*) – Redshift
- **flux** (*float*) – Amplitude of the Gaussian
- **cen** (*float*) – Central wavelength
- **fwhm_km_s** (*float*) – FWHM of the Gaussian in km/s

Returns
Gaussian line model

Return type
np.ndarray

sculptor_extensions.qso.**line_model_gaussian_nii_doublet**(*x*, *z*, *flux_a*, *flux_b*, *fwhm_km_s_a*, *fwhm_km_s_b*)

Doublet line model for the [NII] lines at 6549.85 A and 6585.28 A.

This model ties the redshift of the forbidden transitions of [NII] at 6549.85 A and 6585.28 A together. FWHM and fluxes are free parameters for each Gaussian line model.

Parameters

- **x** (*np.ndarray*) – Dispersion of the continuum model
- **z** (*float*) – Redshift
- **flux_a** (*float*) – Amplitude of the 1st Gaussian component
- **flux_b** (*float*) – Amplitude of the 2nd Gaussian component
- **fwhm_km_s_a** (*float*) – FWHM of the 1st Gaussian component in km/s
- **fwhm_km_s_b** (*float*) – FWHM of the 2nd Gaussian component in km/s

Returns
Gaussian doublet line model

Return type
np.ndarray

sculptor_extensions.qso.**line_model_gaussian_oiii_doublet**(*x*, *z*, *flux*, *fwhm_km_s*, *fluxratio*)

Doublet line model for the [OIII] lines at 4960.30 A and 5008.24 A.

This model ties the redshift, the FWHM and the fluxes (ratio 1:3) of the forbidden transitions of [OIII] at 4960.30 A and 5008.24 A together.

Parameters

- **x** (*np.ndarray*) – Dispersion of the continuum model
- **z** (*float*) – Redshift
- **flux** (*float*) – Amplitude of the Gaussian
- **cen** (*float*) – Central wavelength
- **fwhm_km_s** (*float*) – FWHM of the Gaussian in km/s

Returns
Gaussian doublet line model

Return type
np.ndarray

sculptor_extensions.qso.**line_model_gaussian_sii_doublet**(*x*, *z*, *flux_a*, *flux_b*, *fwhm_km_s_a*, *fwhm_km_s_b*)

Doublet line model for the [SII] lines at 6718.29 A and 6732.67 A.

This model ties the redshift, of the forbidden transitions of [SII] at 6718.29 A and 6732.67 A together. FWHM and fluxes are free parameters for each Gaussian line model.

Parameters

- **x** (*np.ndarray*) – Dispersion of the continuum model
- **z** (*float*) – Redshift
- **flux_a** (*float*) – Amplitude of the 1st Gaussian component
- **flux_b** (*float*) – Amplitude of the 2nd Gaussian component
- **fwhm_km_s_a** (*float*) – FWHM of the 1st Gaussian component in km/s
- **fwhm_km_s_b** (*float*) – FWHM of the 2nd Gaussian component in km/s

Returns
Gaussian doublet line model

Return type
np.ndarray

sculptor_extensions.qso.**power_law_at_2500**(*x*, *amp*, *slope*, *z*)

Power law model anchored at 2500 AA

This model is defined for a spectral dispersion axis in Angstroem.

Parameters

- **x** (*np.ndarray*) – Dispersion of the power law
- **amp** (*float*) – Amplitude of the power law (at 2500 A)
- **slope** (*float*) – Slope of the power law
- **z** (*float*) – Redshift

Returns
Power law model

Return type
np.ndarray

sculptor_extensions.qso.**power_law_at_2500_plus_bc**(*x*, *amp*, *slope*, *z*, *amp_be*, *Te*, *tau_be*, *lambda_be*)

QSO continuum model consisting of a power law anchored at 2500 A and a balmer continuum contribution.

This model is defined for a spectral dispersion axis in Angstroem.

The amplitude of the Balmer continuum is set independently of the power law component by the amplitude of the balmer continuum at the balmer edge amp_be.

Parameters

- **x** (*np.ndarray*) – Dispersion of the continuum model
- **amp** (*float*) – Amplitude of the power law (at 2500 A)
- **slope** (*float*) – Slope of the power law
- **z** (*float*) – Redshift
- **amp_be** (*float*) – Amplitude of the Balmer continuum at the Balmer edge
- **Te** (*float*) – Electron temperature
- **tau_be** (*float*) – Optical depth at the Balmer edge
- **lambda_be** (*float*) – Wavelength of the Balmer edge

Returns
QSO continuum model

Return type
np.ndarray

sculptor_extensions.qso.**power_law_at_2500_plus_fractional_bc**(*x*, *amp*, *slope*, *z*, *f*, *Te*, *tau_be*, *lambda_be*)

QSO continuum model consisting of a power law anchored at 2500 A and a balmer continuum contribution.

This model is defined for a spectral dispersion axis in Angstroem.

The amplitude of the Balmer continuum is set to be a fraction of the power law component at the Balmer edge (3646A) using the variabe f.

Parameters

- **x** (*np.ndarray*) – Dispersion of the continuum model
- **amp** (*float*) – Amplitude of the power law (at 2500 A)
- **slope** (*float*) – Slope of the power law
- **z** (*float*) – Redshift
- **f** (*float*) – Amplitude of the Balmer continuum as a fraction of the power law component
- **Te** (*float*) – Electron temperature
- **tau_be** (*float*) – Optical depth at the Balmer edge
- **lambda_be** (*float*) – Wavelength of the Balmer edge

Returns
QSO continuum model

Return type
np.ndarray

sculptor_extensions.qso.**se_bhmass_civ_c17_fwhm**(*civ_fwhm*, *cont_lwav*, *cont_wav*)

Calculate the single-epoch virial BH mass based on the CIV FWHM and monochromatic continuum luminosity at 1350A.

This relationship follows Eq.6 of from Coatman et al. (2017, MNRAS, 465, 2120)

The FWHM of the CIV line was corrected by the CIV blueshift using their Eq. 4. In this study the CIV line was modeled with sixth order Gauss-Hermite (GH) polynomials using the normalization of van der Marel & Franx (1993) and the functional forms of Cappelari et al. (2002). GH polynomials were chose because they are flexble enough to model the often very asymmetric CIV line profile.

The authors state that using commonly employed three Gaussian components, rather than the GH polynomials, resulted in only marginal differences in the line parameters.

Parameters

- **civ_fwhm** (*astropy.units.Quantity*) – FWHM of the CIV line in km/s
- **cont_lwav** (*astropy.units.Quantity*) – Monochromatic continuum luminosity at 1350A in erg/s/A.
- **cont_wav** (*astropy.units.Quantity*) – Wavelength of the monochromatic continuum luminosity in A.

Returns
Returns a tuple of the BH mass estimate based on the CIV FWHM and a reference string for the single-epoch scaling relationship.

Return type
astropy.units.Quantity, string

sculptor_extensions.qso.**se_bhmass_civ_vp06_fwhm**(*civ_fwhm*, *cont_lwav*, *cont_wav*)

Calculate the single-epoch virial BH mass based on the CIV FWHM and monochromatic continuum luminosity at 1350A.

The monochromatic continuum luminosity at 1450A can be used without error or penalty in accuracy.

This relationship is taken from Vestergaard & Peterson 2006, ApJ 641, 689

The FWHM of the CIV line was measured with the methodology described in Peterson et al. 2004. The line width measurements to establish the CIV single-epoch relation are corrected for resolution effects as described in Peterson et al. 2004.

"The sample standard deviation of the weighted average zero point offset, which shows the intrinsic scatter in the saample is +-0.36 dex. This value is more representative of the uncertainty zer point than is the formal error."

Parameters

- **civ_fwhm** (*astropy.units.Quantity*) – FWHM of the CIV line in km/s
- **cont_lwav** (*astropy.units.Quantity*) – Monochromatic continuum luminosity at 1350A/1450A in erg/s/A.
- **cont_wav** (*astropy.units.Quantity*) – Wavelength of the monochromatic continuum luminosity in A.

Returns
Returns a tuple of the BH mass estimate based on the CIV FWHM and a reference string for the single-epoch scaling relationship.

Return type
astropy.units.Quantity, string

sculptor_extensions.qso.**se_bhmass_civ_vp06_sigma**(*civ_sigma*, *cont_lwav*, *cont_wav*)

Calculate the single-epoch virial BH mass based on the CIV line dispersion (sigma) and monochromatic continuum luminosity at 1350A.

The monochromatic continuum luminosity at 1450A can be used without error or penalty in accuracy.

This relationship is taken from Vestergaard & Peterson 2006, ApJ 641, 689

The FWHM of the CIV line was measured with the methodology described in Peterson et al. 2004. The line width measurements to establish the CIV single-epoch relation are corrected for resolution effects as described in Peterson et al. 2004.

Peterson et al. (2004) note a number of advantages to using sigma rathern than the FWHM as the line width measure.

"The sample standard deviation of the weighted average zero point offset, which shows the intrinsic scatter in the saample is +-0.33 dex. This value is more representative of the uncertainty zer point than is the formal error."

Parameters

- **civ_sigma** (*astropy.units.Quantity*) – Line dispersion (sigma) of the CIV line in km/s
- **cont_lwav** (*astropy.units.Quantity*) – Monochromatic continuum luminosity at 1350A/1450A in erg/s/A.
- **cont_wav** (*astropy.units.Quantity*) – Wavelength of the monochromatic continuum luminosity in A.

Returns
Returns a tuple of the BH mass estimate based on the CIV sigma and a reference string for the single-epoch scaling relationship.

Return type
astropy.units.Quantity, string

sculptor_extensions.qso.**se_bhmass_hbeta_vp06**(*hbeta_fwhm*, *cont_lwav*, *cont_wav=<Quantity 5100. Angstrom>*)

Calculate the single-epoch virial BH mass based on the Hbeta FWHM and monochromatic continuum luminosity at 5100A.

This relationship is taken from Vestergaard & Peterson 2006, ApJ 641, 689

Note that the Hbeta line width to establish this single-epoch virial estimator was established by using the FWHM of only the broad component. The line width was corrected for the spectral resolution as described in Peterson et al. 2004.

The relationship is based on line width measurements of quasars published in Boroson & Green 1992 and Marziani 2003.

"The sample standard deviation of the weighted average zero point offset, which shows the intrinsic scatter in the saample is +-0.43 dex. This value is more representative of the uncertainty zer point than is the formal error."

Parameters

- **hbeta_fwhm** (*astropy.units.Quantity*) – FWHM of the Hbeta line in km/s
- **cont_lwav** (*astropy.units.Quantity*) – Monochromatic continuum luminosity at 5100A in erg/s/A
- **cont_wav** (*astropy.units.Quantity*) – Wavelength of the monochromatic continuum luminosity (default = 5100A).

Returns
Returns a tuple of the BH mass estimate based on the Hbeta FWHM and a reference string for the single-epoch scaling relationship.

Return type
astropy.units.Quantity, string

sculptor_extensions.qso.**se_bhmass_mgii_s11_fwhm**(*mgii_fwhm*, *cont_lwav*, *cont_wav*)

Calculate the single-epoch virial BH mass based on the MgII FWHM and monochromatic continuum luminosity at 3000A.

This relationship is taken from Shen et al. 2011, ApJ, 194, 45

To model the FeII contribution beneath MgII line the authors use empirical FeII templates from Borosn & Grenn 1992, Vestergaard & Wilkes 2001, and Salviander 2007.

Salviander modified the Vestergaard & Wilkes (2001) template in the region of 2780–2830A centered on MgII, where the Vestergaard & Wilkes (2001) template is set to zero. For this region, they incorporate a theoretical FeII model ofSigut & Pradhan (2003) scaled to match the Vestergaard & Wilkes(2001) template at neighboring wavelengths.

As the subtraction of the underlying FeII continuum can have systematic effects on the measurement of the MgII FWHM and therefore the BH mass estimate it is advised to always employ the same continuum construction procedure as in the reference sample that established the single-epoch virial relationship.

Parameters

- **mgii_fwhm** (*astropy.units.Quantity*) – FWHM of the MgII line in km/s
- **cont_lwav** (*astropy.units.Quantity*) – Monochromatic continuum luminosity in erg/s/A.
- **cont_wav** (*astropy.units.Quantity*) – Wavelength of the monochromatic continuum luminosity in A.

Returns
Returns a tuple of the BH mass estimate based on the MgII FWHM and a reference string for the single-epoch scaling relationship.

Return type
astropy.units.Quantity, string

sculptor_extensions.qso.**se_bhmass_mgii_vo09_fwhm**(*mgii_fwhm*, *cont_lwav*, *cont_wav*)

Calculate the single-epoch virial BH mass based on the MgII FWHM and monochromatic continuum luminosity at 1350A, 2100A, 3000A, or 5100A.

This relationship is taken from Vestergaard & Osmer 2009, ApJ 641, 689

To determine the FWHM of the MgII line, the authors modeled the FeII emission beneath the MgII line with the Vestergaard & Wilkes 2001 and the Boroson & Green 1992 iron templates.

Most of the MgII lines were modeled with a single Gaussian component, in cases of high-quality spectra two Gaussian components were used. For the single-Gaussian components the authors adopted the measurements of the FWHM and uncertainties tabulated by Forster et al. (their Table 5). For multi-Gaussian components the FWHM of the full modeled profile was measured.

As the subtraction of the underlying FeII continuum can have systematic effects on the measurement of the MgII FWHM and therefore the BH mass estimate it is advised to always employ the same continuum construction procedure as in the reference sample that established the single-epoch virial relationship.

"The absolute 1 sigma scatter in the zero points is 0.55dex, which includes the factor ~2.9 uncertainties of the reverberation mapping masses to which these mass estimation relations are anchored (see Vestergaard & Peterson 2006 and Onken et al. 2004 for details)"

Parameters

- **mgii_fwhm** (*astropy.units.Quantity*) – FWHM of the MgII line in km/s
- **cont_lwav** (*astropy.units.Quantity*) – Monochromatic continuum luminosity in erg/s/A.
- **cont_wav** (*astropy.units.Quantity*) – Wavelength of the monochromatic continuum luminosity in A.

Returns
Returns a tuple of the BH mass estimate based on the MgII FWHM and a reference string for the single-epoch scaling relationship.

Return type
astropy.units.Quantity, string

sculptor_extensions.qso.**setup_SWIRE_Ell2_template**(*prefix*, ***kwargs*)

Setup the SWIRE library Ell2 galaxy template model

The dispersion axis for this model is in Angstroem.

Parameters

- **prefix** (*string*) – The input parameter exists for conformity with the Sculptor models, but will be ignored. The prefix is automatically set by the setup function.
- **kwargs** –

Returns
LMFIT model and parameters

Return type
(lmfit.Model, lmfit.Parameters)

sculptor_extensions.qso.**setup_SWIRE_NGC6090_template**(*prefix*, ***kwargs*)

Setup the SWIRE library NGC6090 galaxy template model

The dispersion axis for this model is in Angstroem.

Parameters

- **prefix** (*string*) – The input parameter exists for conformity with the Sculptor models, but will be ignored. The prefix is automatically set by the setup function.
- **kwargs** –

Returns
LMFIT model and parameters

Return type
(lmfit.Model, lmfit.Parameters)

sculptor_extensions.qso.**setup_doublet_line_model_nii**(*prefix*, ***kwargs*)

Set up a double line model for the [NII] emission lines at 6549.85 A and 6585.28 A.

This model is defined for a spectral dispersion axis in Angstroem.

Parameters
prefix – The name of the doublet line model. If prefix is None then a

pre-defined name will be assumed. :type prefix: string :param kwargs: :return: Return a list of LMFIT models and a list of LMFIT parameters :rtype: (list, list)

sculptor_extensions.qso.**setup_doublet_line_model_oiii**(*prefix*, ***kwargs*)

Set up a double line model for the [OIII] emission lines at 4960.30 A and 5008.24 A.

This model is defined for a spectral dispersion axis in Angstroem.

Parameters
prefix – The name of the doublet line model. If prefix is None then a

pre-defined name will be assumed. :type prefix: string :param kwargs: :return: Return a list of LMFIT models and a list of LMFIT parameters :rtype: (list, list)

sculptor_extensions.qso.**setup_doublet_line_model_sii**(*prefix*, ***kwargs*)

Set up a double line model for the [SII] emission lines at 6718.29 A and 6732.67 A.

This model is defined for a spectral dispersion axis in Angstroem.

Parameters
prefix – The name of the doublet line model. If prefix is None then a

pre-defined name will be assumed. :type prefix: string :param kwargs: :return: Return a list of LMFIT models and a list of LMFIT parameters :rtype: (list, list)

sculptor_extensions.qso.**setup_galaxy_template_model**(*prefix*, *template_filename*, *templ_disp_unit*, *templ_fluxden_unit*, *fwhm=2500*, *redshift=0*, *amplitude=1*, *intr_fwhm=900*, *dispersion_limits=None*)

Initialize a galaxy template model

Parameters

- **prefix** (*string*) – Model prefix
- **template_filename** (*string*) – Filename of the iron template
- **fwhm** (*float*) – FWHM the template should be broadened to
- **redshift** (*float*) – Redshift
- **amplitude** (*float*) – Amplitude of the template model
- **intr_fwhm** (*float*) – Intrinsic FWHM of the template
- **dispersion_limits** (*(float, float)*) –

Returns
LMFIT model and parameters

Return type
(lmfit.Model, lmfit.Parameters)

sculptor_extensions.qso.**setup_iron_template_MgII_T06**(*prefix*, ***kwargs*)

Setup the Tsuzuki 2006 iron template model around MgII (2200-3500A)

The dispersion axis for this model is in Angstroem.

Parameters

- **prefix** (*string*) – The input parameter exists for conformity with the Sculptor models, but will be ignored. The prefix is automatically set by the setup function.
- **kwargs** –

Returns
LMFIT model and parameters

Return type
(lmfit.Model, lmfit.Parameters)

sculptor_extensions.qso.**setup_iron_template_MgII_VW01**(*prefix*, ***kwargs*)

Setup the Vestergaard & Wilkes 2001 iron template model around MgII (2200-3500A)

The dispersion axis for this model is in Angstroem.

Parameters

- **prefix** (*string*) – The input parameter exists for conformity with the Sculptor models, but will be ignored. The prefix is automatically set by the setup function.
- **kwargs** –

Returns
LMFIT model and parameters

Return type
(lmfit.Model, lmfit.Parameters)

sculptor_extensions.qso.**setup_iron_template_OPT_BG92**(*prefix*, ***kwargs*)

Setup the Boroson & Green 1992 iron template model around Hbeta (3700-7480A)

The dispersion axis for this model is in Angstroem.

Parameters

- **prefix** (*string*) – The input parameter exists for conformity with the Sculptor models, but will be ignored. The prefix is automatically set by the setup function.
- **kwargs** –

Returns
LMFIT model and parameters

Return type
(lmfit.Model, lmfit.Parameters)

sculptor_extensions.qso.**setup_iron_template_T06**(*prefix*, ***kwargs*)

Setup the Tsuzuki 2006 iron template model

The dispersion axis for this model is in Angstroem.

Parameters

- **prefix** (*string*) – The input parameter exists for conformity with the Sculptor models, but will be ignored. The prefix is automatically set by the setup function.
- **kwargs** –

Returns
LMFIT model and parameters

Return type
(lmfit.Model, lmfit.Parameters)

sculptor_extensions.qso.**setup_iron_template_UV_VW01**(*prefix*, ***kwargs*)

Setup the Vestergaard & Wilkes 2001 iron template model around CIV (1200-2200A)

The dispersion axis for this model is in Angstroem.

Parameters

- **prefix** (*string*) – The input parameter exists for conformity with the Sculptor models, but will be ignored. The prefix is automatically set by the setup function.

- **kwargs** –

Returns
LMFIT model and parameters

Return type
(lmfit.Model, lmfit.Parameters)

sculptor_extensions.qso.**setup_iron_template_model**(*prefix*, *template_filename*, *templ_disp_unit*, *templ_fluxden_unit*, *fwhm=2500*, *redshift=0*, *amplitude=1*, *intr_fwhm=900*, *dispersion_limits=None*)

Initialize an iron template model

Parameters

- **prefix** (*string*) – Model prefix
- **template_filename** (*string*) – Filename of the iron template
- **fwhm** (*float*) – FWHM the template should be broadened to
- **redshift** (*float*) – Redshift
- **amplitude** (*float*) – Amplitude of the template model
- **intr_fwhm** (*float*) – Intrinsic FWHM of the template
- **dispersion_limits** (*(float, float)*) –

Returns
LMFIT model and parameters

Return type
(lmfit.Model, lmfit.Parameters)

sculptor_extensions.qso.**setup_line_model_CIII_complex**(*prefix*, ***kwargs*)

Set up a 3 component Gaussian line model for the CIII], AlIII and SiIII] emission lines.

Note that a special model function exists as all three Gaussian line models share a common redshift parameter.

This model is defined for a spectral dispersion axis in Angstroem.

Parameters

- **prefix** (*string*) – The input parameter exists for conformity with the Sculptor models, but will be ignored. The prefix is automatically set by the setup function.
- **kwargs** –

Returns
Return model and model parameters

sculptor_extensions.qso.**setup_line_model_CIV_2G**(*prefix*, ***kwargs*)

Set up a 2 component Gaussian line model for the CIV emission line.

This model is defined for a spectral dispersion axis in Angstroem.

This setup models the broad CIV line emission as seen in type-I quasars and AGN. Due to the broad nature of the line CIV the CIV doublet is assumed to be unresolved.

Parameters

- **prefix** (*string*) – The input parameter exists for conformity with the Sculptor models, but will be ignored. The prefix is automatically set by the setup function.

- **kwargs** –

Returns
Return a list of LMFIT models and a list of LMFIT parameters

Return type
(list, list)

sculptor_extensions.qso.**setup_line_model_Halpha_2G**(*prefix*, ***kwargs*)

Set up a 2 component Gaussian line model for the HAlpha emission line.

This model is defined for a spectral dispersion axis in Angstroem.

Parameters
prefix – The name of the emission line model. If prefix is None then a

pre-defined name will be assumed. :type prefix: string :param kwargs: :return: Return a list of LMFIT models and a list of LMFIT parameters :rtype: (list, list)

sculptor_extensions.qso.**setup_line_model_Hbeta_2G**(*prefix*, ***kwargs*)

Set up a 2 component Gaussian line model for the HBeta emission line.

This model is defined for a spectral dispersion axis in Angstroem.

Parameters
prefix – The name of the emission line model. If prefix is None then a

pre-defined name will be assumed. :type prefix: string :param kwargs: :return: Return a list of LMFIT models and a list of LMFIT parameters :rtype: (list, list)

sculptor_extensions.qso.**setup_line_model_MgII_2G**(*prefix*, ***kwargs*)

Set up a 2 component Gaussian line model for the MgII emission line.

This model is defined for a spectral dispersion axis in Angstroem.

This setup models the broad MgII line emission as seen in type-I quasars and AGN. Due to the broad nature of the line MgII the MgII doublet is assumed to be unresolved.

Parameters

- **prefix** (*string*) – The input parameter exists for conformity with the Sculptor models, but will be ignored. The prefix is automatically set by the setup function.
- **kwargs** –

Returns
Return a list of LMFIT models and a list of LMFIT parameters

Return type
(list, list)

sculptor_extensions.qso.**setup_line_model_SiIV_2G**(*prefix*, ***kwargs*)

Set up a 2 component Gaussian line model for the SiIV emission line.

This setup models the broad SiIV line emission as seen in type-I quasars and AGN. Due to the broad nature of the line SiIV, the SiIV doublet is assumed to be unresolved and blended with the OIV emission line.

This model is defined for a spectral dispersion axis in Angstroem.

Parameters

- **prefix** (*string*) – The input parameter exists for conformity with the Sculptor models, but will be ignored. The prefix is automatically set by the setup function.
- **kwargs** –

Returns
Return a list of LMFIT models and a list of LMFIT parameters

Return type
(list, list)

sculptor_extensions.qso.**setup_line_model_gaussian**(*prefix*, ***kwargs*)

Initialize the Gaussian line model.

Parameters

- **prefix** (*string*) – Model prefix
- **kwargs** – Keyword arguments

Returns
LMFIT model and parameters

Return type
(lmfit.Model, lmfit.Parameters)

sculptor_extensions.qso.**setup_power_law_at_2500**(*prefix*, ***kwargs*)

Initialize the power law model anchored at 2500 A.

Parameters

- **prefix** (*string*) – Model prefix
- **kwargs** – Keyword arguments

Returns
LMFIT model and parameters

Return type
(lmfit.Model, lmfit.Parameters)

sculptor_extensions.qso.**setup_power_law_at_2500_plus_bc**(*prefix*, ***kwargs*)

Initialize the quasar continuum model consistent of a power law anchored at 2500A and a balmer continuum contribution.

Parameters

- **prefix** (*string*) – Model prefix
- **kwargs** – Keyword arguments

Returns
LMFIT model and parameters

Return type
(lmfit.Model, lmfit.Parameters)

sculptor_extensions.qso.**setup_power_law_at_2500_plus_fractional_bc**(*prefix*, ***kwargs*)

Initialize the quasar continuum model consistent of a power law anchored at 2500A and a balmer continuum contribution.

The Balmer continuum amplitude at the Balmer edge is set to be a fraction of the power law component.

Parameters

- **prefix** (*string*) – Model prefix
- **kwargs** – Keyword arguments

Returns
LMFIT model and parameters

Return type
(lmfit.Model, lmfit.Parameters)

sculptor_extensions.qso.**setup_split_iron_template_MgII_T06**(*prefix*, ***kwargs*)

Setup the Tsuzuki 2006 iron template model subdivided into three separate models at 2200-2660, 2660-3000, 3000-3500.

The dispersion axis for this model is in Angstroem.

Parameters

- **prefix** (*string*) – The input parameter exists for conformity with the Sculptor models, but will be ignored. The prefix is automatically set by the setup function.
- **kwargs** –

Returns
Return a list of LMFIT models and a list of LMFIT parameters

Return type
(list, list)

sculptor_extensions.qso.**setup_split_iron_template_MgII_VW01**(*prefix*, ***kwargs*)

Setup the Vestergaard & Wilkes 2001 iron template model subdivided into three separate models at 2200-2660, 2660-3000, 3000-3500.

The dispersion axis for this model is in Angstroem.

Parameters

- **prefix** (*string*) – The input parameter exists for conformity with the Sculptor models, but will be ignored. The prefix is automatically set by the setup function.
- **kwargs** –

Returns
Return a list of LMFIT models and a list of LMFIT parameters

Return type
(list, list)

sculptor_extensions.qso.**setup_split_iron_template_OPT_BG92**(*prefix*, ***kwargs*)

Setup the Boroson & Green 1992 iron template model subdivided into three separate models at 3700-4700, 4700-5100, 5100-5600.

The dispersion axis for this model is in Angstroem.

Parameters

- **prefix** (*string*) – The input parameter exists for conformity with the Sculptor models, but will be ignored. The prefix is automatically set by the setup function.
- **kwargs** –

Returns
Return a list of LMFIT models and a list of LMFIT parameters

Return type
(list, list)

sculptor_extensions.qso.**setup_split_iron_template_UV_VW01**(*prefix*, ***kwargs*)

Setup the Vestergaard & Wilkes 2001 iron template model subdivided into three separate models at 1200-1560, 1560-1875, 1875-2200.

The dispersion axis for this model is in Angstroem.

Parameters

- **prefix** (*string*) – The input parameter exists for conformity with the Sculptor models, but will be ignored. The prefix is automatically set by the setup function.
- **kwargs** –

Returns
Return a list of LMFIT models and a list of LMFIT parameters

Return type
(list, list)

sculptor_extensions.qso.**setup_subdivided_iron_template**(*templ_list*, *fwhm=2500*, *redshift=0*, *amplitude=1*)

Setup iron template models from a predefined list of templates and dispersion ranges.

Parameters

- **templ_list** – List of template names for which models will be set up
- **fwhm** (*float*) – Goal FWHM of the template model
- **redshift** (*float*) – Redshift of the template model
- **amplitude** (*float*) – Amplitude of the template model

Type
list

Returns
Return a list of LMFIT models and a list of LMFIT parameters

Return type
(list, list)

sculptor_extensions.qso.**template_model**(*x*, *amp*, *z*, *fwhm*, *intr_fwhm*, *templ_disp=None*, *templ_fluxden=None*, *templ_disp_unit_str=None*, *templ_fluxden_unit_str=None*)

Template model

Parameters

- **x** (*np.ndarray*) – Dispersion of the template model
- **amp** (*float*) – Amplitude of the template model
- **z** (*float*) – Redshift
- **fwhm** (*float*) – FWHM the template should be broadened to
- **intr_fwhm** (*float*) – Intrinsic FWHM of the template
- **templ_disp** (*np.ndarray*) – Dispersion axis of the template. This must match the same dispersion unit as the model
- **templ_fluxden** (*templ_fluxden: np.ndarray*) – Flux density of the template.
- **templ_disp_unit_str** (*str*) – Dispersion unit of the template as a string in astropy cds format.
- **templ_fluxden_unit** – Flux density unit of the template as a string in astropy cds format.

Returns
Template model as a Scipy interpolation

CHAPTER

SIXTEEN

EXAMPLE EXTENSION MODULE

An example Sculptor Extension

This module defines models, masks and analysis routines as an example to create your own Sculptor extension.

At first we define the basic model functions and their setups. The setup functions initialize LMFIT models and parameters using the basic model functions defined here.

When youy define a mask you need to specify the name under which it will appear in the Sculptor GUI, the rest_frame keyword and the mask ranges to that will be included in the SpecModel masking or excluded in the SpecFit masking. With rest_frame=True the mask regions will automatically be adjusted for the object redshift specified in the SpecFit class. With rest-frame=False the mask will not be redshifted.

sculptor_extensions.my_extension.**model_func_dict = {'my_model': <function my_model>}**

List of model names

Type
list of str

sculptor_extensions.my_extension.**model_func_list = ['My Model']**

Dictionary of model setup function names

Type
dict

sculptor_extensions.my_extension.**model_setup_list = [<function setup_my_model>]**

Dictionary of mask presets

Type
dict

sculptor_extensions.my_extension.**my_mask = {'mask_ranges': [[1265, 1290], [1340, 1375], [1425, 1470], [1680, 1705], [1905, 2050]], 'name': 'My mask', 'rest_frame': True}**

Dictionary of model functions

Type
dict

sculptor_extensions.my_extension.**my_model**(*x*, *z*, *amp*, *cen*, *fwhm_km_s*, *shift_km_s*)

Gaussian line model as an example for a model

The central wavelength of the Gaussian line model is determined by the central wavelength cen, the redshift, z, and the velocity shift shift_km_s (in km/s). These parameters are degenerate in a line fit and it is adviseable to fix two of them (to predetermined values e.g., the redshift or the central wavelength).

The width of the line is set by the FWHM in km/s.

The Gaussian is not normalized.

Parameters

- **x** (*np.ndarray*) – Dispersion of the continuum model
- **z** (*float*) – Redshift
- **amp** (*float*) – Amplitude of the Gaussian
- **cen** (*float*) – Central wavelength
- **fwhm_km_s** (*float*) – FWHM of the Gaussian in km/s
- **shift_km_s** (*float*) – Doppler velocity shift of the central wavelength

Returns
Gaussian line model

Return type
np.ndarray

sculptor_extensions.my_extension.**setup_my_model**(*prefix*, ***kwargs*)

Example of a model setup function for the Gaussian emission line model.

The 'prefix' argument needs to be included. You can use a variety of keyword arguments as you can see below.

Parameters

- **prefix** (*string*) – Model prefix
- **kwargs** – Keyword arguments

Returns
LMFIT model and parameters

Return type
(lmfit.Model, lmfit.Parameters)

CHAPTER

SEVENTEEN

LICENSE

PYTHON MODULE INDEX

S

INDEX

D

F

G

I

K

L

M

N

P

R

S

T

U

X

Y