IFSCube Manual

Daniel Ruschel Dutra

August 18, 2017

Contents

1 Introduction

1.1 What is IFSCube?

IFSCube is a python package designed to perform analysis tasks in data cubes of integral field spectroscopy. It was originally designed to work with the data from Gemini's Multi-Object Spectrograph (GMOS), and as such, many default parameters are set to work best with this kind of data. Using it with data cubes from other instruments should be possible by explicitly setting a few more parameters.

Keep in mind that this is not a closed software that will be called from the command line and perform the tasks that you might want. Instead it is intended to be used as part of your own scripts, hopefully making them a lot more concise, and saving you a lot of time. At the same time, since everything you can do in a python script can also be done using an interactive interpreter such as ipython, you can also perform your analysis on the fly.

1.2 Installation instructions

The preferred method for the installation of IFSCube is to use pip. Pip can install directly from the git repository using the following command:

```
pip install git+https://danielrd6@bitbucket.org/danielrd6/ifscube.git
```

IFSCube uses third party programs that are not distributed with the package, namely pPXF and Voronoi Binning, written by Michelle Cappellari. If you wish to use IFSCube in conjunction with these programs

you will have to install them first, by downloading and following the instructions at the following address: http://www-astro.physics.ox.ac.uk/ mxc/software/ Upgrade

If you want to upgrade an existing installation of IFSCube use

```
pip install --upgrade git+https://danielrd6@bitbucket.org/danielrd6/ifscube.git
```

If you are having trouble with the Fortran compiler you can force one with pip's install options, which are exemplified below.

To force a specific compiler:

```
pip install git+https://danielrd6@bitbucket.org/danielrd6/ifscube.git
--install-option=build --install-option='--fcompiler=gnu95'
```

IFSCube has been extensively tested with the astroconda[1] distribution, therefore it is highly recommended that you install it within the astroconda's Python 3 environment.

2 Basic Functions

The *cubetools* module contains all the major functions of the IFSCube package, including some basic functions that allow a quick inspection of the data cube. These functions are implemented as methods of the *gmosdc* class, so the recommended usage is always to first load your data as a *gmosdc* object. The examples in this section will use the ngc3081_cube.fits file, present in the examples directory.

```
from ifscube import cubetools as ct

mycube = ct.gmosdc('ngc3081_cube.fits')
```

The initialization method for *gmosdc* sets up a few basic variables that will be used in the analyses tasks, such as the wavelength, rest wavelength, science data, noise cube, etc. The rest wavelenght specifically, is only properly set if you enter the redshift when initalizing the object, such as in

```
mycube = ct.gmosdc('ngc3081_cube.fits', redshift=0.042)
```

If no redshift is given then the rest wavelength will be just a copy of the observed wavelength. In either case, all the methods within the *gmosdc* class will always interpret wavelengths as refering to the rest frame wavelength.

2.1 Image from wavelength cut

Data cubes have three dimensions, i.e. two spatial and one spectral. In order to have an image of the data cube one has to select a wavelength range, or collapse the entire cube along the spectral dimension. Often it is useful to have an image at a specific wavelength, and for that there is the *wlprojection* function. The latter takes the data cube, multiplies it by a filtering function and returns the resulting array. Two arguments are required by *wlprojection*, namely the central wavelength and the FWHM of the filtering function. Additionaly, there is the optional *filtertype* argument, which selects the type of filtering used to produce the image. Currently you can choose between a box function and a gaussian.

For instance, if you want to have an image of the data cube centred on 6000$\mathring{A}$ with a FWHM of 100$\mathring{A}$

```
im = a.wlprojection(6000, 100)
```

[1] https://astroconda.readthedocs.io/en/latest/

3 Fitting emission lines

One of the main purposes and motivations behind the development of IFSCube is the fitting of emission lines in the spectra of galaxies. This is a common task for 1D spectra, and therefore there is a plethora of software than can achieve it fairly well, from IRAF's splot onward. However, fitting data cubes requires a code that can run with little to no interaction from the user, since the modeling process is expected to run on several hundred spectra.

The *linefit* function of the *gmosdc* class was designed specifically for the fitting emission features in spectroscopic data cubes. It is basically a wrapper for the *scipy.optimize.minimize* function, with the addition of many tools for doing the minimization recursively over the entire cube. The reader that wants to gain a deeper understanding of the fitting process inside *linefit* is strongly advised to read the documentation for *minimize*.

The IFSCube distribution includes a heavily spatially under sampled data cube for the NGC 3081 galaxy, which will be used in the examples below. You can find it within the *examples* directory, under the path of your installation of IFSCube. All the examples in this manual will refer to this data cube.

3.1 Line fitting, a basic example.

The first thing you will need to do to start using the functions contained in IFSCube is to load your data cube as a gmosdc object. This will start the basic variables that will be used by analysis functions. IFSCube was designed to work with Multi Extension FITs files, and relies on the astropy.io.fits package to read them.

In a python shell, or in your script, you should start by importing the module and initializing the *gmosdc* object.

```
from ifscube import cubetools as ct

mycube = ct.gmosdc('mycube.fits')
```

In case you have a noise estimate in the form of a cube of σ, you can load it automatically by giving the extension number as a parameter to gmosdc. If available, this noise information will be used for plotting your spectra and for evaluating the reduced χ^2 of the line fitting.

```
mycube = ct.gmosdc('ngc3081_cube.fits', var_ext=2)
```

Now let us take a look at a particular spectrum of the data cube, using the *plotspec* method. This method takes two mandatory arguments, which are the horizontal and vertical spaxel coordinates. The following example will cause a plot of the spectrum at spaxel (3, 3) to appear.

```
mycube.plotspec(3, 3)
```

If all went well, you should see the plot in figure 1. The shaded smoothed region is the noise estimate from extension 2. If you do not have the noise extension this shaded region will simply not be plotted, but otherwise everything should work equally well.

If we zoom in around the Hα region you will get a clearer picture of the emission lines, which will be instrumental in defining the initial guess for the line fitting process. Suppose now that we are only interested in fitting the [N II] line, ignoring the neighboring [N II] and Hα lines. We should start by defining a list of initial parameters which we will call *p0*. This list has to be in the order of amplitude, central wavelength and sigma for a Gaussian profile fit.

```
p0 = [
    1e-14,  # The flux at the center of the line.
    6635.0,  # The central wavelength.
    3.0,  # The sigma in wavelength units.
    ]
```

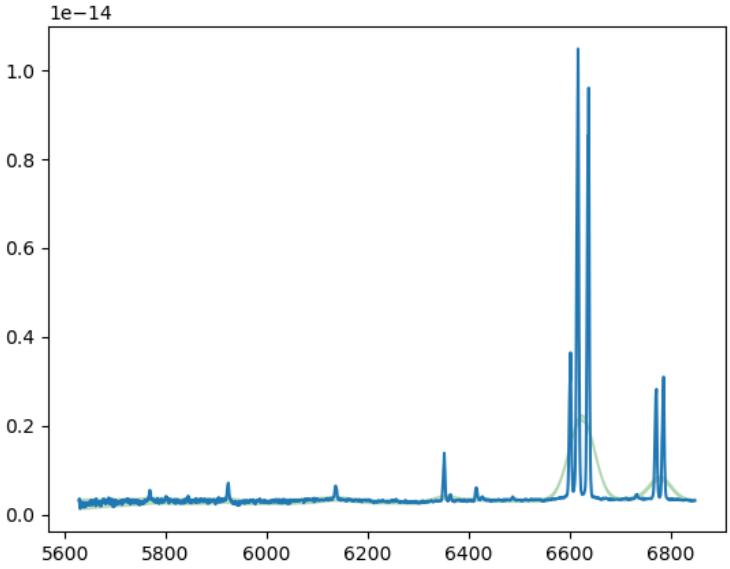

Figure 1: Example of the plotspec method.

Also, although not mandatory, it is always good to define a fitting window. This will help the code find a suitable continuum level and save time by not fitting a lot of zeros far away from the line of interest. By default, the continuum will be inferred from a polynomial fit to the spectrum with an aggressive sigma clipping. Most of the time this leads to good results, but you can change the parameters for the continuum fitting or provide a continuum entirely defined by you using the argument *copts* or setting the *mycube.cont* property. Both of these options will be discussed in more detail later. For this example we are fitting within 6500 and 6750 Å.

We should also choose a profile function for the emission line. Currently only Gaussian and Gauss-Hermite profiles are supported. We tell linefit which one to use by setting the argument *function='gaussian'*. For a Gauss-Hermite profile use 'gauss-hermite'.

Finally, since we are not trying to fit the whole data cube in our first run, we should set the *individual_spec* argument to some tuple defining the coordinate of the desired spaxel, just like in the *plotspec* function discussed above.

The complete call to the linefit function should look like the following.

```
mycube.linefit(p0, function='gaussian', fitting_window=(6500, 6750),
               individual_spec=(3, 3))
```

When fitting a single spaxel, this function returns a lot of information that will only be stored in the gmosdc object during data cube runs. This behavior is particularly useful for interactive runs, as it gives immediate access to the main fit products, thus facilitating the user's evaluation of the fitting procedure.

In order to see the result of this fit, you can use the plotfit method, which again takes the spaxel coordinates as arguments. This method will plot every component of the fit (in this case it will be only the Gaussian profile and the continuum), the observed spectrum and the modeled spectrum.

```
mycube.plotfit(3, 3)
```

Figure 2 shows a zoomed in view of plotfit's result. In this example the green line represents the observed

spectrum, the blue line is the modeled line profile added to the continuum, and the orange line is the continuum. Plotfit also prints the fit parameters in the terminal, as in the example below.

```
A          wl         s
  8.89e-15   6636.85     -2.05
```

Notice how the σ value turned out negative. Since only σ^2 is used in the Gaussian profile, the minimization algorithm can make no distinction between positive and negative values. This can be prevented by setting bounds to each parameter, which will be discussed in further detail in another section.

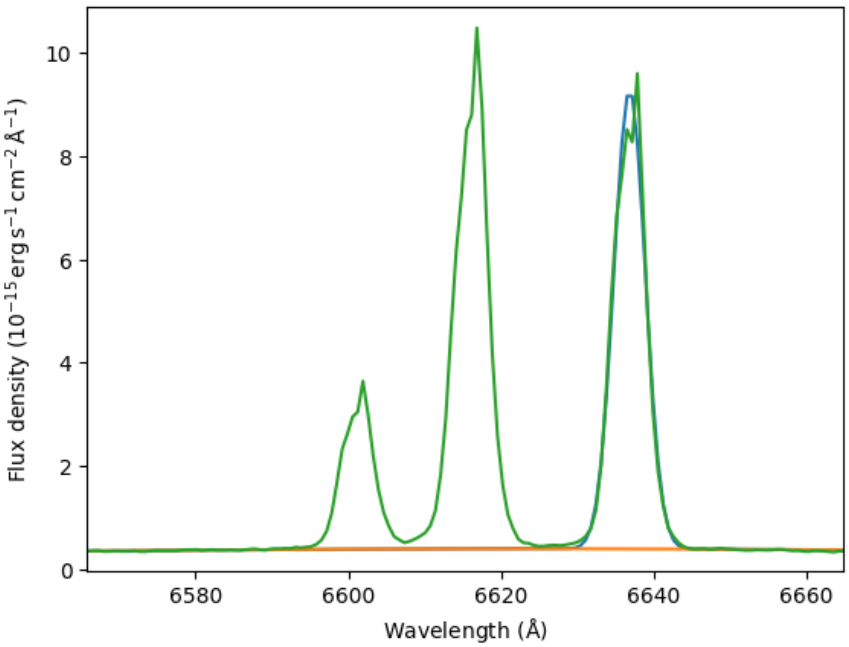

Figure 2: Example of the plotfit function.

If all went well and we are satisfied with our fitting, we can go ahead and remove the *individual_spec* argument, and let linefit run over the entire data cube. Removing this argument will cause linefit to return the fit parameters in a data cube of spatial dimensions equal to the original data, and depth equal to the number of parameters plus one. The last plane of this solution cube is the reduce χ^2 of the fit.

```
mycube.linefit(p0, function='gaussian', fitting_window=(6500, 6750))
```

Even if you do not choose to store the output in a variable, it will be stored in the mycube.em_model property. It is also advisable to save the results of the fit in a FITS file, by setting the arguments *write-fits=True* and *outimage='myfit.fits'*. This will allow you to use the *fit_scrutinizer* utility to check the results of your fits (see section 4.

3.2 Setting bounds

If perchance you went ahead and tried to fit the entire data cube with the parameters of the last section you mostly likely got disappointed with the results. Alas, life is not that simple, neither is minimization. A good fit, other than having a good initial guess, should have bounds. They prevent the minimization algorithm from wandering too far away from the physically sound parameters.

Bounds for linefit are a list of pairs, with each pair representing the minimum and maximum values of the parameters, respecting the same order of the initial guess. For instance, let us say that the amplitude should not be negative, neither should it be greater than 1e-13. Our initial guess for the central wavelength should be within 5 Å of the correct answer and the sigma should be above 1 Å and below 6 Å. Therefore

```
b = [
        [0, 1e-13],
        [6630, 6640],
        [1, 6],
]
```

We now pass this list as an argument to linefit and see how that turns out.

```
mycube.linefit(p0, function='gaussian', fitting_window=(6500, 6750), bounds=b)
```

3.3 Multiple lines

Following the basic example of a single line fit, we will now discuss how to fit multiple lines simultaneously. The basic process remains the same, the only thing that will change is the number of parameters in the initial guess and the number of bounds. The *linefit* function is prepared to work with an arbitrarily large number of components, as long as they have the same profile function, i.e. Gaussian or Gauss-Hermite. [2]

In the next example we will fit both [N II] and the Hα line. Our inital guess $p0$ will be

```
p0 = [
    1e-14,  # The flux at the center of the line.
    6601.0,  # The central wavelength.
    3.0,  # The sigma in wavelength units.

    1e-14,  # The flux at the center of the line.
    6617.0,  # The central wavelength.
    3.0,  # The sigma in wavelength units.

    1e-14,  # The flux at the center of the line.
    6635.0,  # The central wavelength.
    3.0,  # The sigma in wavelength units.
    ]
```

Of course this is a little tedious to write explicitly, and we can try to do it programatically, like

```
lambda_0 = [6601, 6617, 6635]
p0 = []
for i in range(3):
    p0 += [1e-14, lambda_0[i], 3]
```

Or even, if you would like to have a piece of code with all the components clearly distinguised you could define three lists, one for each of the components, and later state that $p0$ is the sum of all three.

```
n2a = [1e-14, 6601, 3]
ha = [1e-14, 6617, 3]
n2b = [1e-14, 6635, 3]
p0 = n2a + ha + n2b
```

[2] A complementary version of *linefit* that accepts astropy.models.FittableModel instances instead of the internally defined functions is currently under development.

The bounds can also be set programatically based on our initial guess. For instance, if you want to keep loose bounds for the amplitude and the sigma, but want to be stringent on the central wavelength you could write

```
b = []
for i in range(0, len(p0), 3):
    b += [[0, 1e-13]]  # Amplitude positive and below 1e-12
    b += [[p0[i + 1] - 5, p0[i + 1] + 5]]  # Central wavelength within 5 A
    b += [[1, 6]]  # Sigma between 1A and 6A
```

The call to linefit is exactly the same as before, the only difference lies in the definition of $p0$ and b. Figure 3 shows the result of this fit for the spaxel (3, 3).

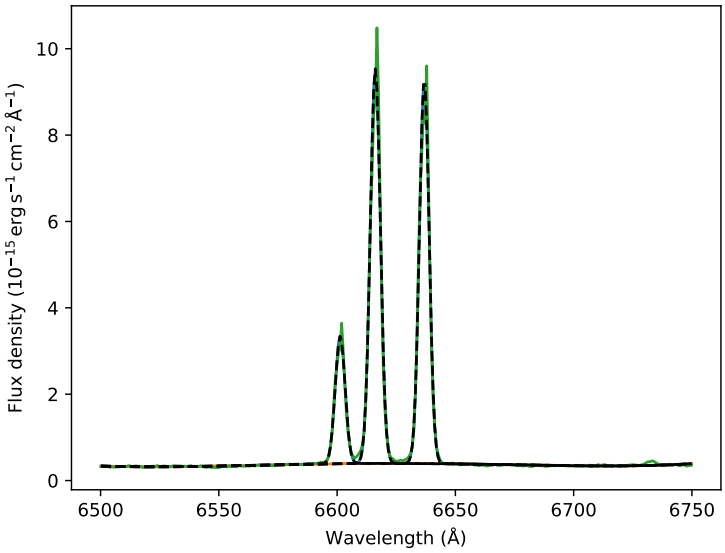

Figure 3: Example of a multiple component fit for the same spaxel (3, 3).

3.4 Constraints

The next important topic on a good fit is constraints. Constraints allow you to define relations between parameters, which is a

4 Checking fit results: fit_scrutinizer

The white zone is for loading and unloading only. If you got to load or unload, go to the white zone.

Frank Zappa

The fastest way to check the results of your fit is to use the fit_scrutinizer program. This program has graphical user interface (GUI) that lets you select each parameter of each component, and see the image of that parameter at the same time showing the spectrum of a particular spaxel. After installation of IFSCube by the pip installer, fit_scrutinizer will be appended to your path, making it accessible from any directory directly from the command line.

For instance, let us take a look at that first attempt at a data cube fit, saved as myfit.fits. The data cube that originated it is the ngc3081_cube.fits, therefore the calling sequence to fit_scrutinizer should read

```
fit_scrutinizer ngc3081_cube.fits myfit.fits
```

This will star a GUI similar to the one in figure 4, but without any of the plots yet. To start plotting your results you have to select a parameter from the list at the lower left corner, and a component from the list right next to it. In this example we selected the velocity for the only component available, component "0". Next we click "Image plot" to generate the image of the velocity for the first component in the upper left.

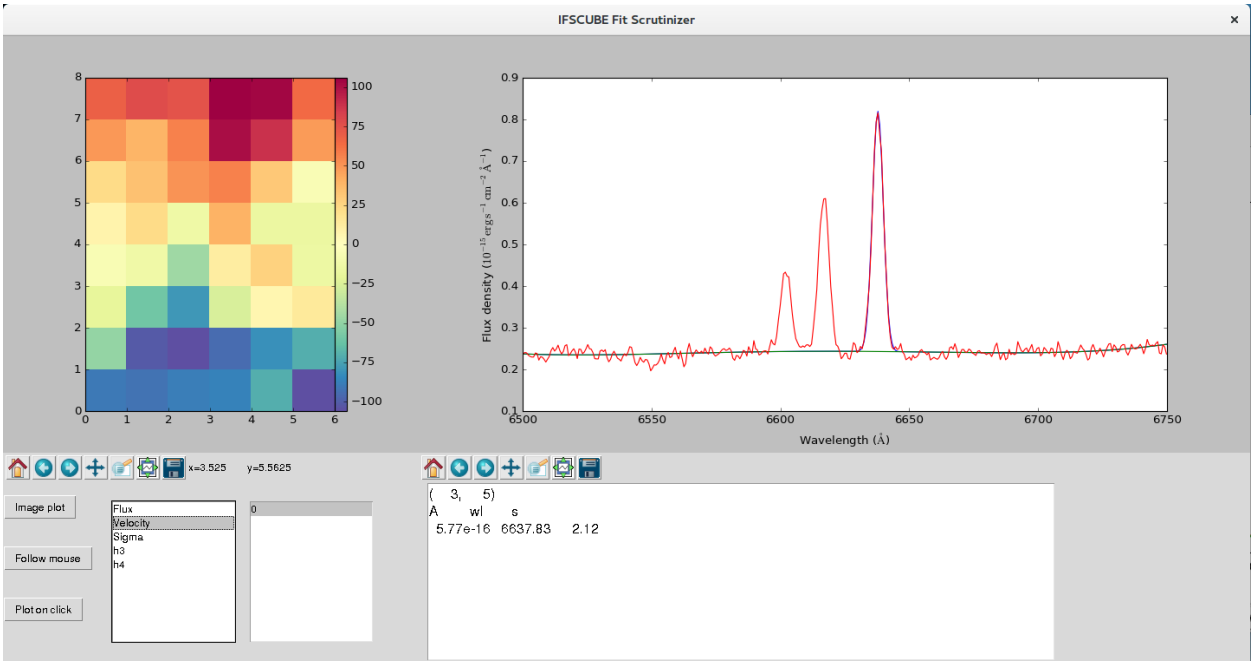

Figure 4: Example of the interface of the fit_scrutinizer program, showing the velocity image, and the spectrum in the spaxel (3, 5).

At this point only the image is visible, but no spectral plot will be produced until you click on either "Follow mouse" or "Plot on click". The former will cause a new spectral plot to be generated every time the mouse enters a new spaxel on the image at the upper left, while the latter will only plot the spectrum when you click on a spaxel.