SpectraPy Documentation

Release 1.0.1

Marco Fumana

Aug, 2021

CONTENTS:

CHAPTER

ONE

SPECTRAPY INTRODUCTION

SpectraPy is a Python3 library, which collects algorithms and methods for data reduction of astronomical spectra obtained by a through slits spectrograph.

The library is designed to be **spectrograph independent**. It comes with a set of already configured spectrographs, but it can be easily configured to reduce data of other instruments.

Current implementation of SpectraPy is focused on the extraction of 2D spectra: it produces wavelength calibrated spectra, rectified for instrument distortion. The library can be used on both longslit (LS) and multi object spectrograph (MOS) data.

To achieve the spectra extraction, the main components used by the library are:

1. the *Instrument configuration* file
2. the *Mask description* file
3. one set of geometrical *Models*, which describes the spectra geometry on the raw frames.

Using these components the library is able to follow spectra distortions and extract rectified wavelength calibrated 2D spectra.

The extraction steps, can be roughly summarized in:

1. *Models calibration*: the calibration of the geometrical models. **If the instrument is stable enough**, this step is done once for all, for a given instrument configuration.
2. *Model Data Tuning*: the geometrical models must be adjusted on the current data. This steps performs slightly adjustments on the data in use
3. *Spectra Extraction*: the extraction of the 2D spectra.

In this tutorial we will show how to create your *Models* from scratch, apply them to your data and obtain 2D spectra extracted.

We will also illustrate how to handle different issues related to MOS ans LS data.

1.1 Acknowledging or citing SpectraPy

If you use SpectraPy in your work, we would be grateful if you could include an acknowledgment in papers and/or presentations

1.1.1 In publications

If you use SpectraPy for research presented in a publication, we ask that you please cite the SpectraPy DOI 10.20371/inaf/sw/2021_00001

1.1.2 In projects or presentations

If you are using SpectraPy as part of a code project, or if you are giving a presentation/talk featuring work/research that makes use of SpectraPy and would like to acknowledge SpectraPy, we suggest using this badge

CHAPTER

TWO

SPECTRAPY INSTALLATION PROCEDURE

2.1 Installing Prerequisites

- Python3.7 (or greater)
- Astopy
- Matplotlib
- PyDS9
- pyregion
- Cython
- DS9 (8.1 or greater)

We suggest to download and install Anaconda Python Distribution and update your `$PATH` environment variable to use the proper python

In the `BASH` shell

```
> export PATH=${CONDA_INSTAL_DIR}/bin:$PATH
```

or in the `TCSH` shell

```
> setenv PATH ${CONDA_INSTAL_DIR}/bin:$PATH
```

And install the additional packages

```
> ${CONDA_INSTAL_DIR}/bin/pip install pyds9
> ${CONDA_INSTAL_DIR}/bin/pip install pyregion
```

Astropy and Cython are already included in the *conda distribution*.

2.2 Install SpectraPy

Extract files from the tar package

```
> tar -xvzf spectrapy-0.11.3.tar.gz
```

and use the proper python3 to install it

```
> python3 setup.py install
```

SpectraPy is now installed and can be used from the python3 console

Note: To work with spectrapy, we suggest to use the ipython3 console (available in the conda distribution previously installed). This is a powerful interactive shell, which provides a lot of facilities.

CHAPTER

THREE

INSTRUMENT CONFIGURATION

SpectraPy requires basic information about the instrument and the format of the data files. All these information are collected in the **instrument configuration file** (`icf`), the file is formalized by one ini file organized in 4 different sessions:

1. *Detector*
2. *Grism*
3. *Files*
4. *Description* (optional)

3.1 Detector

The `detector` section contains information related to the detectors geometry. The following information are required by the library:

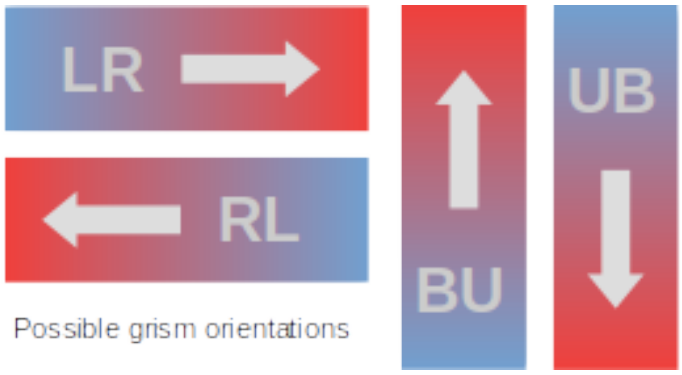
- `pixel_scale` ["/pixel]: the nominal pixel scale of the instrument
- `xpixels`: the number of pixels along the X axis
- `ypixels`: the number of pixels along the Y axis

3.2 Grism

The `grism` section defines the properties of the dispersion element:

- **`dispersion_direction`: this keyword defines the dispersion direction, which goes from bluest regions of the frame to reddest. Possible values of this keyword are:**
    1. `LR`: dispersion along X axis, from left to right
    2. `RL`: dispersion along X axis, from right to left
    3. `BU`: dispersion along Y axis, from bottom to top
    4. `LR`: dispersion along Y axis, from top to bottom
- `linear_dispersion` [A/pixel]: the nominal linear dispersion of the grism
- `reference_lambda`: wavelength position of a bright isolate line near the nominal central wavelength of the grism[1].

[1] Strictly speaking the `reference_lambda` is not a property of the grism, but it is a value required by the **SpectraPy wavelength calibration algorithm**. During the calibration process, we will choose a bright isolated line (sky line or arc line) as *reference point* for the slit. Since the real slit position is not visible on dispersed frames, this line will be used by SpectraPy as *"ideal"* slit position, and as reference point for the models.



3.3 Files

Every instrument produces data in several FITS formats and data can be stored in extensions that are not the Primary HDU. The `files` section is used by the library to retrieve data from the proper extension. The extensions can be defined by name or by number[2].

- `data_hdu`: extension containing science data. If not specified, the primary extension is used
- `var_hdu`: extension containing variance on data. If not specified, no variance is used.
- `err_hdu`: this is used when no variance is associated with data, but an error layer is provided[3].
- `flag_hdu`: extension containing bit mask on data. Pixels with mask value > 0, will be masked out by SpectraPy library during data calibration or data extraction. This layer can be used to mask out bad pixels, cosmic ray, ...

3.4 Description

The `Description` section is not mandatory and it used just for human purposes. It just contains a description for user

3.5 The icf files

In the `conf/instruments` directory there are a set of already prepared instrument configuration files. Here we can see an example of MODS1R/MODS2R instrument configuration file used in this manual.

```
[Description]
instrument = MODS1R/2R
grism = G670L

[Detector]
pixel_scale = 0.123
pixel_size = 0.015
xpixels = 8288
ypixels = 3088

[Grism]
```

(continues on next page)

[2] The Primary extension is 0 (like astropy.io.fits does)
[3] SpectraPy assumes var_hdu = err_hdu [2]

(continued from previous page)

```
dispersion_direction = RL
#A/pixels
linear_dispersion = 0.8
#Ne line
reference_lambda = 6929.47

[Files]
data_hdu = Primary
```

CHAPTER

FOUR

MASK DESCRIPTION

The mask description files (`mdf` files) are used by SpectraPy to know the geometry of the slits mask and to locate spectra on the raw frames. The *mdf* file is an ASCII file, containing the following columns[1]:

1. `ID` the **unique** slit ID[2].
2. `DIMX` and `DIMY` the dimensions in millimeters of the slit.
3. `X` and `Y` the position (always in millimeters) of the **center** of the slit in the field of view.[3]
4. `ROT` the rotation angle of the slit. Rotation angle is clockwise and starts from cross-dispersion direction.
5. `WID` and `LEN` the width and length of the slit in arcsec. These entries are not mandatory, some instruments don't provides them. In case these information are not available the value must be 0.[4]
6. `REF` the reference slit flag. This value is 1 in case the slit is a reference. Reference slits can be ignored during data calibration and extraction

Here an example of one `mdf` of a MODS mask

```
#ID     DIMX    DIMY    X       Y       ROT     WID     LEN     REF
49      0.720   3.600   -73.62  -86.11   0.0    1.2      6.0    0
85      0.720   6.000   -40.32  -91.09   0.0    1.2     10.0    0
119     0.720   3.600   -86.62  -48.47  40.0    1.2      6.0    0
Ref1    2.400   2.400   -20.70   74.29   0.0    4.0      4.0    1
```

4.1 Mask utility scripts

SpectraPy provides a set of scripts (stored in the `scripts` directory) to automatically create mdf files for VIMOS, LUCI and MODS data. Available scripts are:

1. `vimos2mdf.py`
2. `luci2mdf.py`
3. `mods2mdf.py`

The `vimos2mdf.py` and `luci2mdf.py` require in input the fits data file, because in the VIMOS and LUCI cases, mask geometry is described in the header of the files.

[1] No empty entries are allowed
[2] SpectraPy handles IDs as strings.
[3] The library assumes the center of this coordinates system in the center of the FOV.
[4] unlike the `DIMX` and `DIMY` parameters `WID` and `LEN` are not oriented along the X,Y axes, but they are oriented according with the grism dispersion direction: `LEN` is the slit length along the cross-dispersion direction, `WID` is the width along the dispersion direction

Since no mask information is available in the header of MODS files, MODS scripts takes in input the `mms` (the file obtained by the MODS mask preparation tool).

Finally, for LUCI and MODS spectrograph, a set of already prepared LS masks is available in the `conf/masks` directory.

CHAPTER

FIVE

MODELS

SpectraPy algorithms rely on 3 geometrical model:

1. The *Optical Model*, used to describe the geometrical distortions along the FOV
2. The *Curvature Model*, used to describe the spectra displacement with respect to the ideal dispersion direction, perfectly aligned along pixels
3. The *Inverse Dispersion Solution Model*, which describes the wavelength to pixel relation

5.1 Optical Model

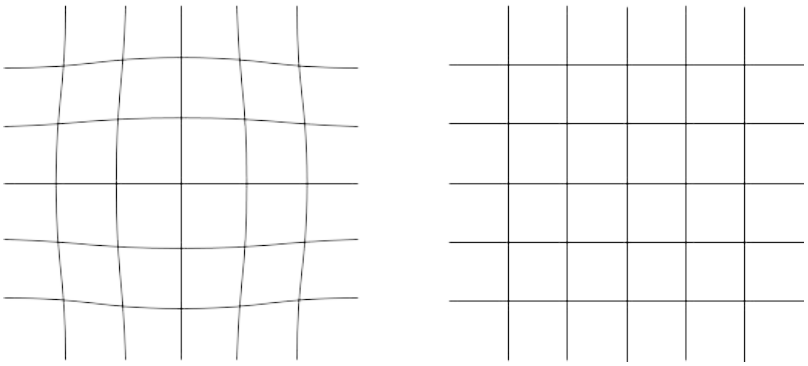
The Optical Model (aka `OPTModel`) describes the optical distortion in the FOV and it converts mask slit positions (in millimeters) into pixels on the detector. The Optical model does not dependent on the grism in use, since it just locates the reference lambda (the virtual slit for SpectraPy) on the detector ignoring any information about dispersion.

The model is defined by a pair of global 2D polynomials: one for the x and one for the y.

In the current implementation X and Y polynomials have the same shape.

$$x_{[pix]} = \sum_{i=0}^{N} \sum_{j=0}^{M} X_{i,j} x_{[mm]}^{i} y_{[mm]}^{j}$$

$$y_{[pix]} = \sum_{i=0}^{N} \sum_{j=0}^{M} Y_{i,j} x_{[mm]}^{i} y_{[mm]}^{j}$$



This model is used to describe the distorted images in a rectified coordinate system.

5.2 Curvature Model

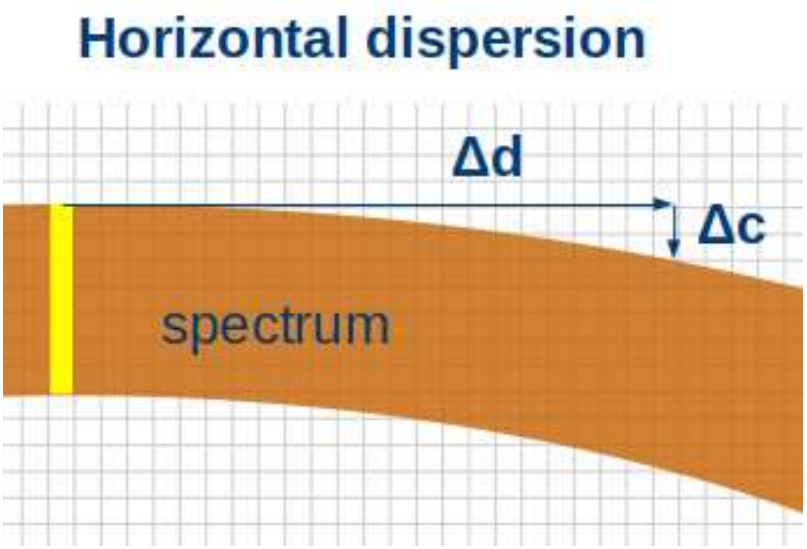
Due to optical distortions on real data, spectra are not perfectly aligned along the pixels and these distortions change within the FOV.

The Curvature Model (aka `CRVModel`) describes the deviation of the spectra traces, with respect to the perfect straight line (horizontal in the case of LR or RL dispersion directions and vertical in the case of BU or UB dispersion directions). The model estimates this displacement (in pixel) along the cross dispersion direction.

For each slit, one mono dimensional polynomial is used to describe the displacement along the cross dispersion direction Δc, starting from the slit reference position (located by the Optical Model)

$$\Delta c_{[pix]} = \sum_{i=0}^{N} c_{x,y,i} \Delta d^{i}_{[pix]}$$

The Δd is the displacement with respect to the reference lambda position in pixel (x, y).



Each slit has own set of $c_{x,y,i}$ coefficients, different slit by slit, because each slit is in a difference position in the FOV.

SpectraPy uses a global model to describe coefficients variation along the FOV.

The local $c_{x,y,i}$ coefficients are obtained by the evaluation of this global model at the reference lambda position on detector.

$$c_{x,y,i} = \sum_{h=0}^{N} \sum_{k=0}^{M} C_{i,h,k} x^{h}_{[pix]} y^{k}_{[pix]}$$

Note: This approach has the advantage to be **mask independent**. Once we have calibrated the global CRVModel, if the instrument is stable, we can apply the same model to describe every masks.

5.3 Inverse Dispersion Solution Model

Once the spectra are located on Detector (*Optical Model*) and geometrically described (*Curvature Model*) the wavelength calibration of the 2D can be carried out.

The Invese Dispersion Solutions Model (aka `IDSModel`) is the model used to obtain the relation between pixel positions and wavelengths. It moves along the curve described by the combination of OptModel and CRVModel and it associates expected wavelength value to pixels of this curve.

The IDSModel mathematical description is quite similar to the CRVModel: for each slit, one mono dimensional polynomial locates the wavelength position . Since each slit is in a difference position in the FOV, and distortions changes within the FOV, each slit has own set of $d_{x,y,i}$ coefficients.

$$d_{x,y,i} = \sum_{h=0}^{N} \sum_{k=0}^{M} D_{i,h,k} x_{[pix]}^{h} y_{[pix]}^{k}$$

The set of $d_{x,y,i}$ coefficients are used to measure the wavelength with respect to the reference lambda position.

$$\Delta d_{[pix]} = \sum_{i=0}^{N} d_{x,y,i} (\lambda - \lambda_{ref})^{i}$$

The Δd is the displacement with respect to the reference lambda position located at the pixel (x, y).

Even in this case a global 2D polynomial is used to describe the coefficients variation along the FOV and the local $c_{x,y,i}$ are obtained by the evaluation of this global model

CHAPTER

SIX

CATALOGS

Catalog files are ASCII files containing expected lines positions, lines starting with # are ignored. These file are used both during *Inverse Dispersion Solution calibration* and during IDSDataCalib to know where emission lines should be.

In the directory `conf/catalogs` there are a set of already prepared catalogs, or we can create our custom catalog, according to our needs.

The catalog file, must contain 3 columns:

- **pos**: the expected line position (in Angstrom)
- **label**: the line name, a label to identify it (just for human purposes)
- **flag**: the line flag internally used by SpectraPy (positive integer)

Here, as example, a part of a line catalog provided by SpectraPy

```
11591.684       sky     1
11627.846       sky     1
11650.746       sky     1
11696.348       sky     1
11716.151       sky     1
12007.078       sky     0
12030.885       sky     0
12055.878       sky     1
12121.5         double  2
12196.386       sky     1
```

The `flag` is an integer number used by SpectraPy, in the current version of SpectraPy, just 3 flags are allow: 0, 1, 2.

- **0** if the line is faint or very close to a stronger line.
- **1** these are bright isolated lines
- **2** these are lines very close and not resolved by the instrument, but they can be useful to anchor the solution in regions of the spectrum with few lines

During the *Inverse Dispersion Solution calibration*, the positions of the lines in the frame are manually decided by user. At this stage we can use all the lines in the catalog, in case of doubts we can remove the DS9 regions related with the ambiguous line. Different flags are plotted with different colors:

- 0 (faint lines): **pink**
- 1 (reliable lines): **green**
- 2 (not resolved lines): **red**

The *Inverse Dispersion Solution calibration* procedure is totally automatic, at this stage **only** the reliable lines (the lines with flag=1) will be used. SpectraPy automatically discards other lines, to obtain one solution as much reliable as possible.

CHAPTER

SEVEN

MODELS CALIBRATION

In the following sections, we will see how to use SpectraPy to obtain the final 2D extracted spectra. As training example we will use a MODS1R MOS frame. In some cases one longslit LUCI frame is also used to highlight the differences between MOS and LS cases.

7.1 Optical Model calibration

In order to define the distortions map of the FOV, we must initialize the *Optical Model*. This step requires: the instrument configuration file and the mask description file. We will start by loading an arc lamp frame for MODS1R

```
>>> # The MODS instrument configuration file
>>> mods1r = "conf/instruments/mods_G670L.icf"
>>> # The mask description file
>>> mods_mask = "examples/data/mods1r/ID532016.mdf"
>>> mods_arc = "examples/data/mods1r/mods1r.20180121.0073.fits.bz2"
```

We must initialize the `ModelsCalibration` class: **this is the class used to calibrate all the models**.

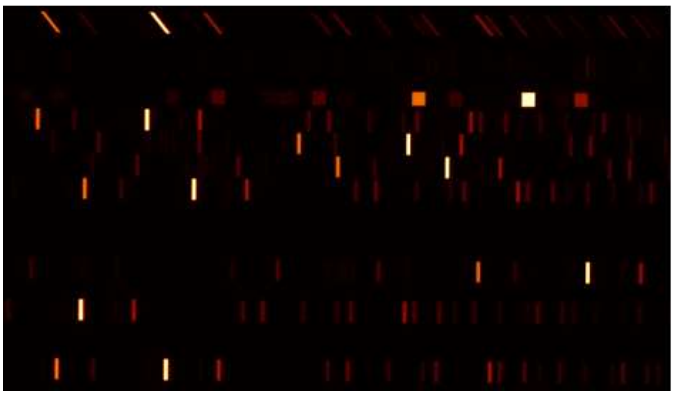
```
>>> from spectrapy.modelscalib.calib import ModelsCalibration
>>> calib = ModelsCalibration(mods1r, mask=mods_mask)
```

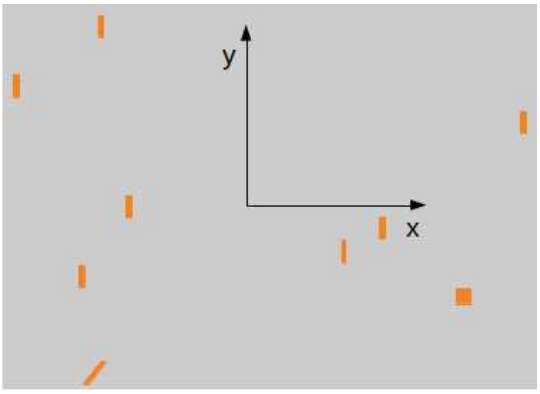
The last call opens a DS9 instance used by SpectraPy to display images and regions. All the calibration processes consist in moving and adjusting regions on the frame in order to compute proper models. The idea is that the `ModelsCalibration` instance shows us the current models solutions plotting regions on the DS9 frame. We can adjust these regions and refit the model using the new regions positions.

First of all, we must create a new Optical Model from scratch, because no OPTModel is yet available for this instrument configuration. In this case, we choose to describe the OPTModel with a polynomial of order 2 in both (x and y) directions.

```
>>> opt = calib.new_opt_model(2, 2)
```

OPTModel assumes the mask oriented with the X axis horizontal (from left to right) and Y axis vertical (from bottom to up). In the next figure we show on the left a dispersed frame, on the right the mask as described by the Optical Model.

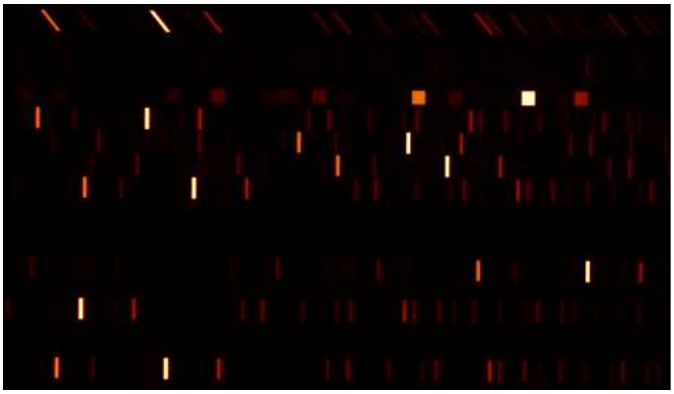


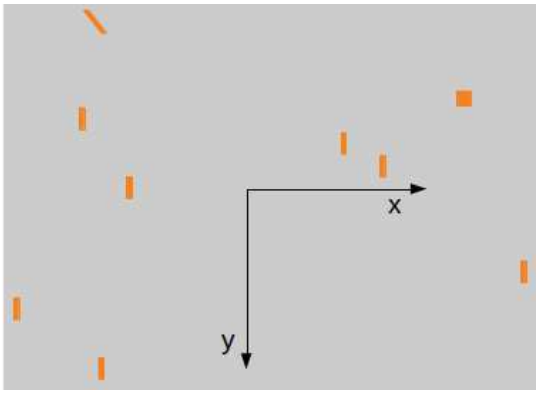


As we can see, in this case the mask and the image don't match, slits are not in the expected position, we can check this discrepancy looking at the tilted slit or at the reference square slit.

Due to instrument particularities and optical reflections, this initial assumption (with the axes oriented like in the picture above) can be not true for all the instruments. To solve this problem, the OPTModel can be flipped on both directions by the methods: `flipx` and `flipy`. In this MODS1R case, we flipped the model vertically.

```
>>> # For optical reasons we must (at least in MODS) flip the model along the Y axis
>>> opt.flipy()
```



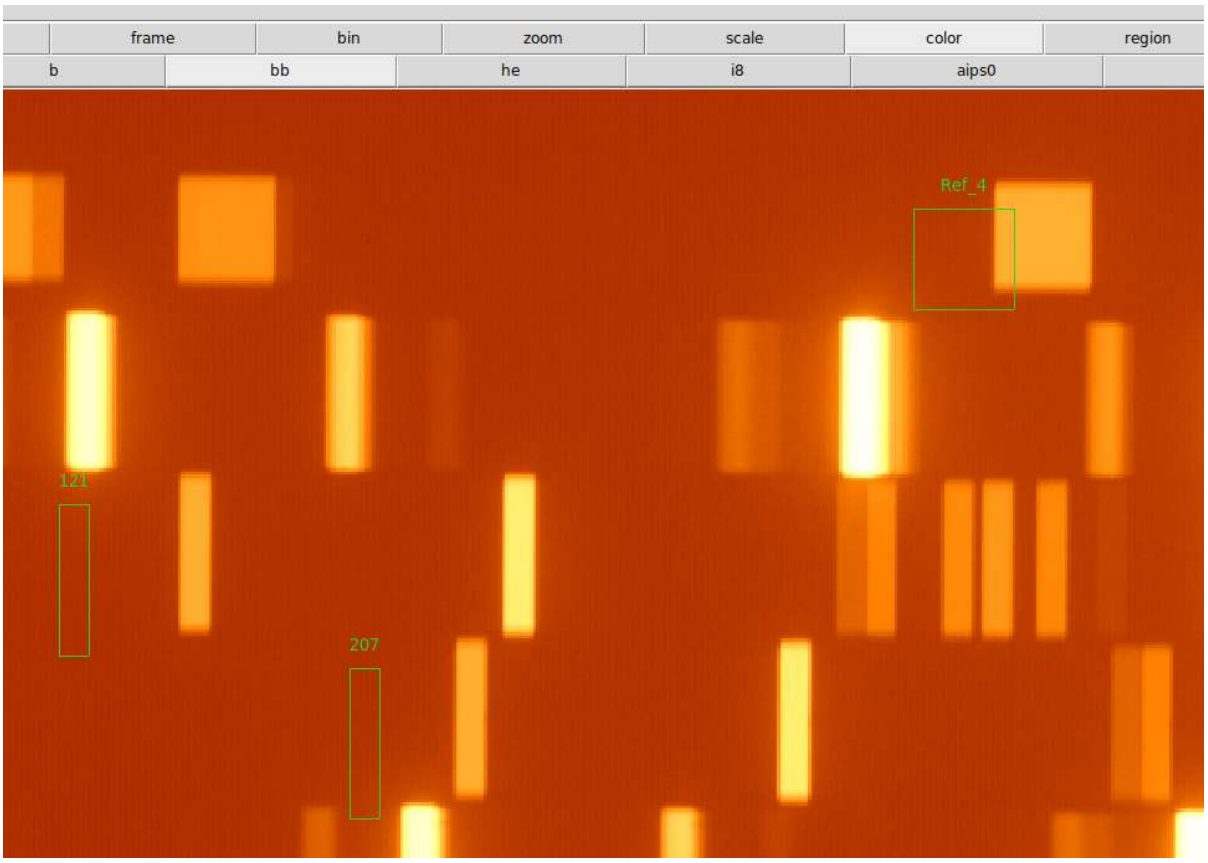


The `OPTModel` locates the mask slits on the FOV. Since we are working with dispersed images, slits are not visible on the frame. For this reason we will use an arc frame to tune the model: we will use the lambda reference position on the frame as *virtual slit*.

Note: As lambda reference position we suggest to choose a bright isolated line in the mid region of the dispersion range.

```
>>> calib.load_image(mods_arc)
>>> calib.plot_opt_model()
```

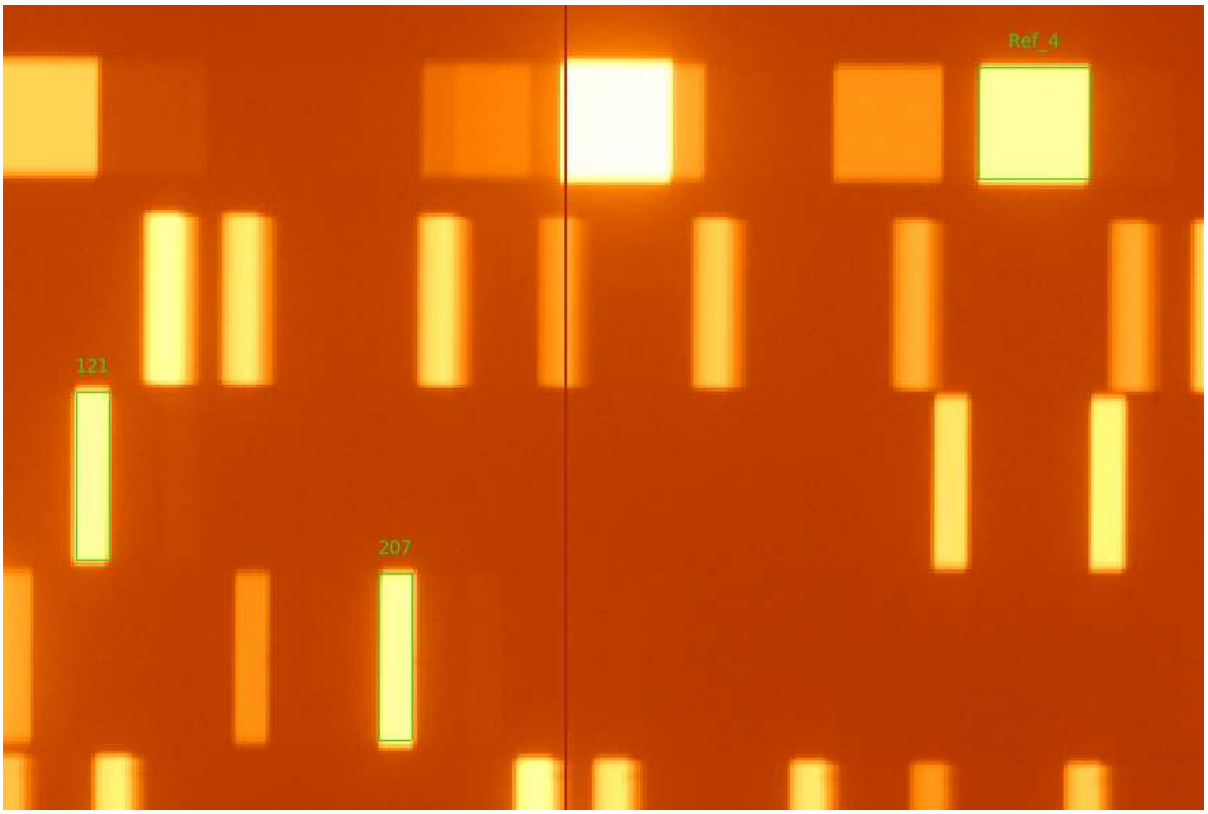
The arc image and the expected slits as green boxes will be displayed in the DS9 viewer.



The first time, usually slits will not be in the proper position, we have to tune the model moving[1] these slits on the expected reference line position[2], using the standard DS9 regions commands

[1] In case regions are frozen and you are not able to move them, select the `Region` option in the DS9 `Edit` menu.

[2] In case you want follow the exercise, without losing time moving regions, you can use already prepared region file in `examples/data/mods1r/regions/opt.reg`. You must delete all current regions and replace them with regions contained into the file.



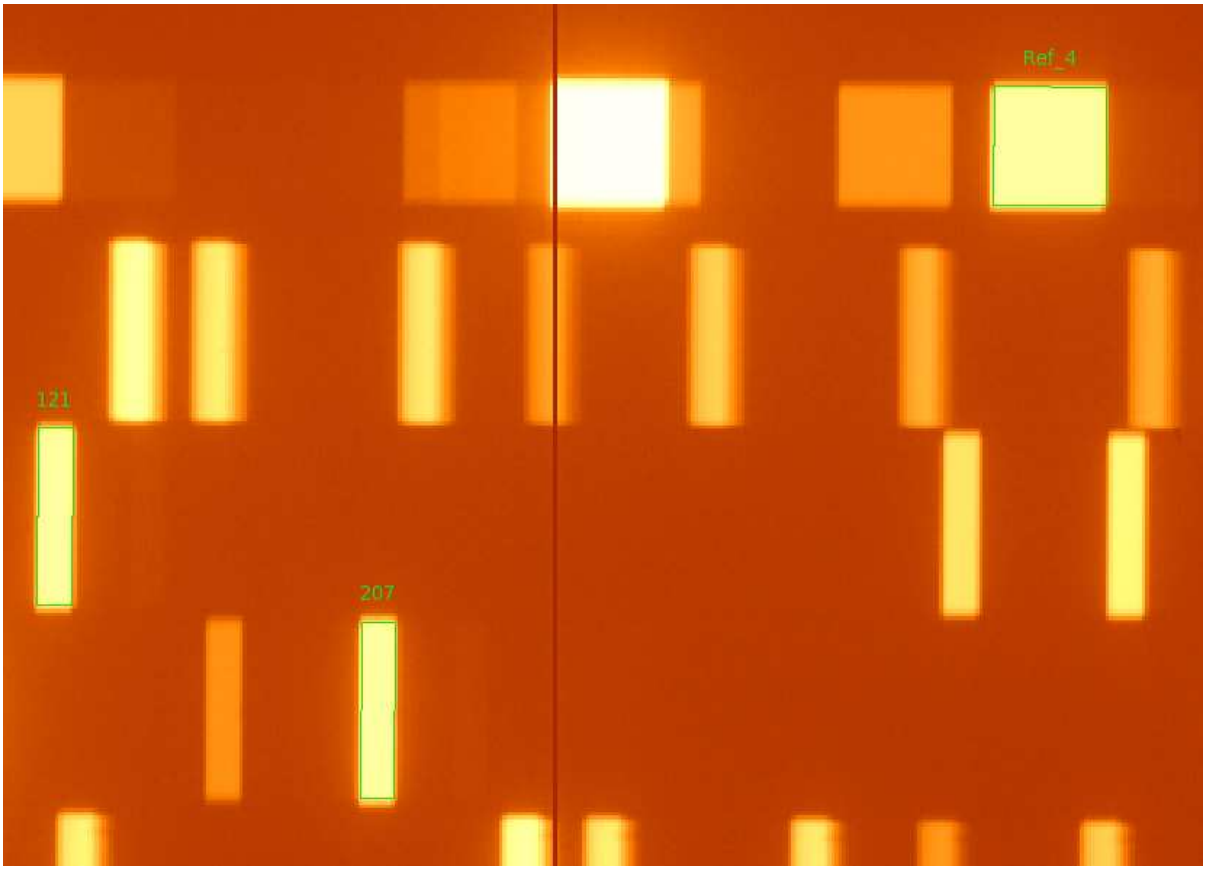
Once we have moved all the boxes over the correct lines, we must recompute the model solution, refitting it. The library reads from DS9 the current position of the lists, and adjust the model according with these positions.

Note: In case we don't want to use some slit and discard it from the fits, we can just remove this region from DS9.

```
>>> calib.fit_opt_model()
```

We can visually check the results, plotting again of the recomputed model on the frame in use.

```
>>> calib.plot_opt_model()
```



If slits regions, still remain in the proper position, the model is good and we can save it.

```
>>> opt.writeto("examples/tmp/MODS1R.opt", overwrite=True)
```

In case the model does not match the proper slit positions, we can try to increase the polynomial order of the model and repeat the previous operations.

7.1.1 The longslit case

In this section, we will calibrate the optical model for a longlist case. For this example we will use LUCI1 frames acquired with a slit 0.75" width and 60" long, using the low resolution grism. The main difference is that the number of slits in the FOV is just 1.

```
>>> luci_mask = "conf/masks/luci_LS_0.75.mdf"
>>> luci1 = "conf/instruments/luci_G200LoRes_1.93_1.8.icf"
>>> luci_file = "examples/data/luci1LoRes/luci1.20180202.0181.fits.bz2"
```

In this case we are using a science frame, since in the instrument configuration file we choose an OH sky line as reference position.

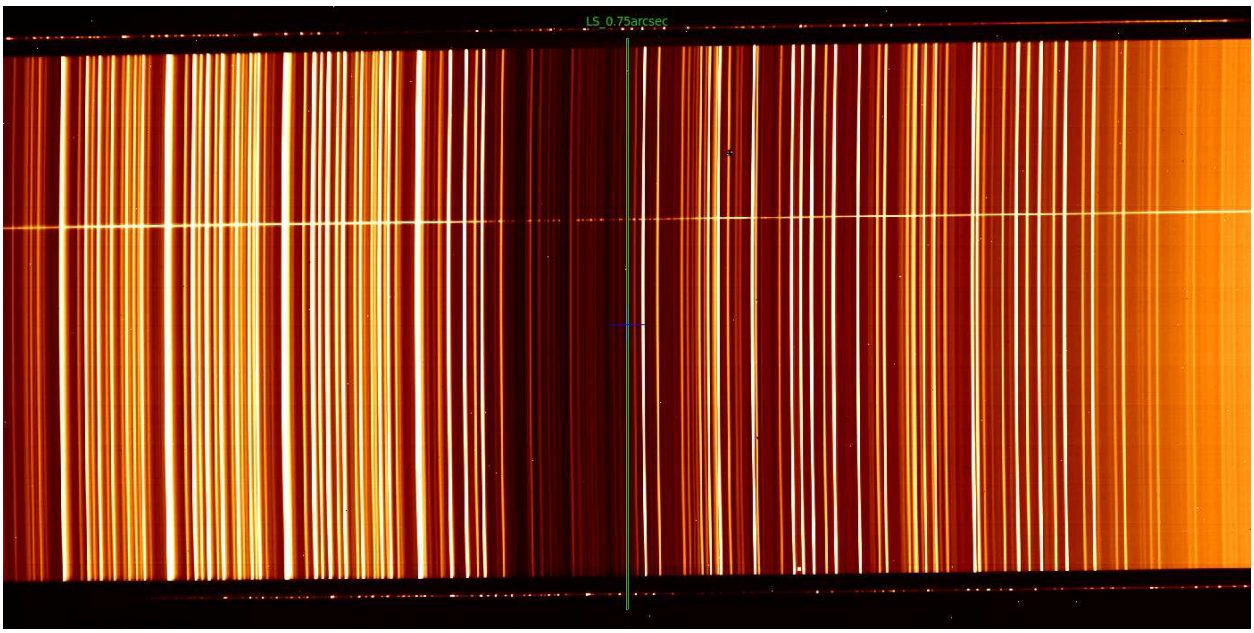
Again we create the OPTModel from scratch and display the slit.

```
>>> from spectrapy.modelscalib.calib import ModelsCalibration
>>> calib = ModelsCalibration(luci1, mask=luci_mask)
>>> calib.load_image(luci_file)
```

(continues on next page)

(continued from previous page)

```
>>> opt = calib.new_opt_model(1, 1)
>>> calib.plot_opt_model(edit=True)
```



In this case we have just one slit, and the slit mutal positions can not be used to derive the scaling factor of the Optical Model. You can see how the nominal scale factor is not enough accurate since the slit size (green box on the frame) doesn't fit the size on the dispersed frame.

So we have to adjust both the slit position and the slit dimensions. This is the reason why we set the parameter `edit=True`. With this flag ON we can move also the slit corners to fit the *real* slit position.[3]

Note: Be careful: slit is described by a polygon. **DON'T ADD** corners to this polygon, just **move the already existing corners**, otherwise the fit will fail!

This resizing allows us to calibrate the OPTModel scale. Once done we can refit the models

```
>>> calib.fit_opt_model()
```

And save the fit result

```
>>> opt.writeto("examples/tmp/LUCI1.opt", overwrite=True)
```

[3] Already prepared region file is available in examples/data/luci1LoRes/regions/opt.reg

7.2 Curvature Model calibration

Once the `OPTModel` has been calibrated, i.e. the reference lambda is properly located for each list, we can handle the spectra curvatures.

The Curvature Model calibration can be performed either using the previous `ModelsCalibration` class instance or initializing a new class, loading the saved OPTModel.

```
>>> #The variables already initialized
>>> mods_mask = "examples/data/mods1r/ID532016.mdf"
>>> mods1r = "conf/instruments/mods_G670L.icf"
>>> mods_arc = "examples/data/mods1r/mods1r.20180121.0073.fits.bz2"
>>> from spectrapy.modelscalib.calib import ModelsCalibration

>>> opt = "examples/tmp/MODS1R.opt"
>>> calib = ModelsCalibration(mods1r, mask=mods_mask, opt=opt)
```

Like in the `OPTModel` case, we create a *Curvature Model* from scratch and display it

```
>>> crv=calib.new_crv_model(1, 2, 2)
```

The previous line of code creates a curvature model which is locally described by a straight line, i.e. each spectra trace is described by one 1st order polynomial. The coefficients of these lines change along the FOV, i.e. the curvature of each trace is slightly different spectrum by spectrum. In this example we decide to describe **this variation** by a 2D polynomial of order 2 by 2.

Since we want to follow the trace of the spectra, the best frame to use for the `CRVModel` is a through slit flat, which shows clearly the trace edges of the spectra.

```
>>> mods_flat="examples/data/mods1r/mods1r.20180121.0067.fits.bz2"
>>> calib.load_image(mods_flat)
>>> calib.set_trace_limits(800, 800)
```

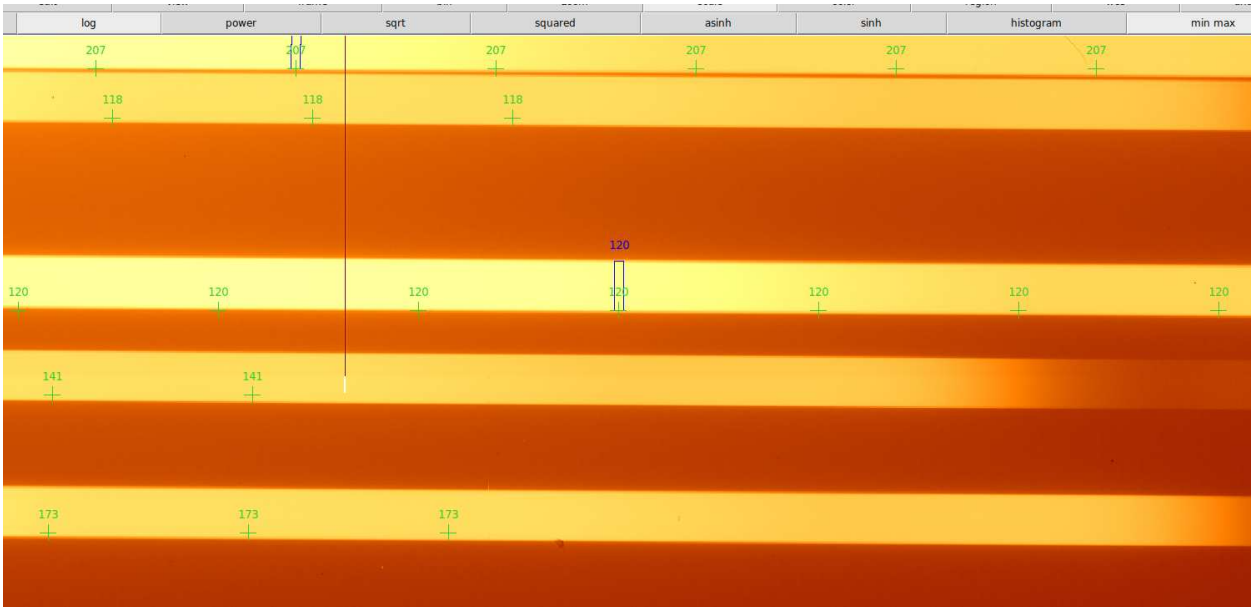
The `CRVModel`, starting from the reference position defined by the `OPTModel`, follows the geometry of the spectra both in the blue and the red directions.

The last line of code defines (in pixels) the tracing range we are going to calibrate. In this case we decide to calibrate 800 pixels (both directions) along the dispersion direction around the reference line[4].

```
>>> calib.plot_crv_model(9)
```

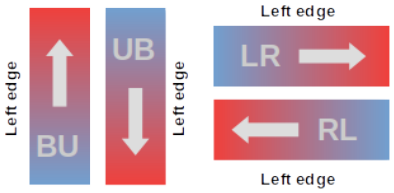
In this tutorial we decided to use 9 points (marked by green crosses) equally spaced along these 1600 pixels.

[4] The first parameter of the `set_trace_limits` defines the number of pixels in the blue direction, the second parameter the pixels in the red



DS9 will show the slits position (the blue region) and 9 crosses for each slits along the expected position of the spectra traces. Like we did for the OPTModel, we must adjust in DS9 the crosses along the *left side* the spectra traces.[5]

By default only the *left* edges of the spectra are used to calibrate the CRVModel. The *left* and *right* edges of the spectra are defined by SpectraPy on frames with BU dispersion direction (frames where the dispersion direction goes from bottom to up). According with that: in the frames with UB the dispersion direction the *left* edges for SpectraPy are the right edges on the frames, if the dispersion direction is LR is the upper side is the *left* edge and in the RL case the *left* edge is the lower edge on the frame.

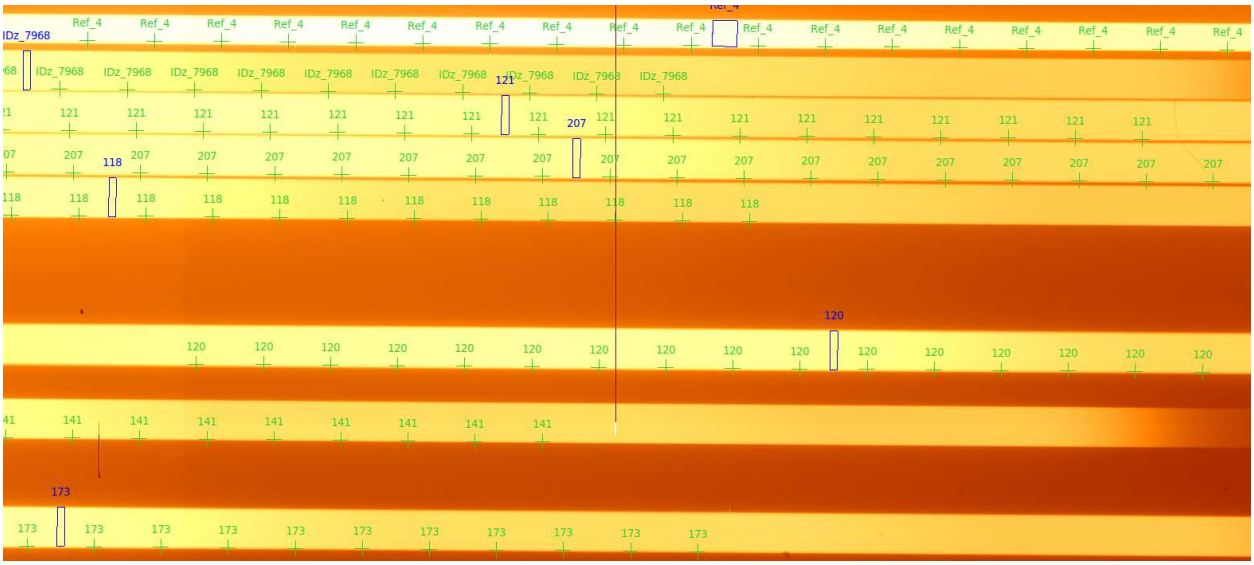


Note: Don't be afraid to delete some crosses if they follow out of frame or they are in a region where the spectrum signal is too faint, the fitting procedure will use only available crosses

```
>>> calib.fit_crv_model()
>>> calib.plot_crv_model(20)
```

Once done, we can refit the model and check it again. During check we can increase the number of points (20 in this case) to better visualize the new solution

[5] An already prepared region file is available in examples/data/mods1r/regions/crv.reg



If we are satisfied by this solution, we can save it pass on to the IDSModel calibration.

```
>>> crv.writeto("examples/tmp/MODS1R.crv", overwrite=True)
```

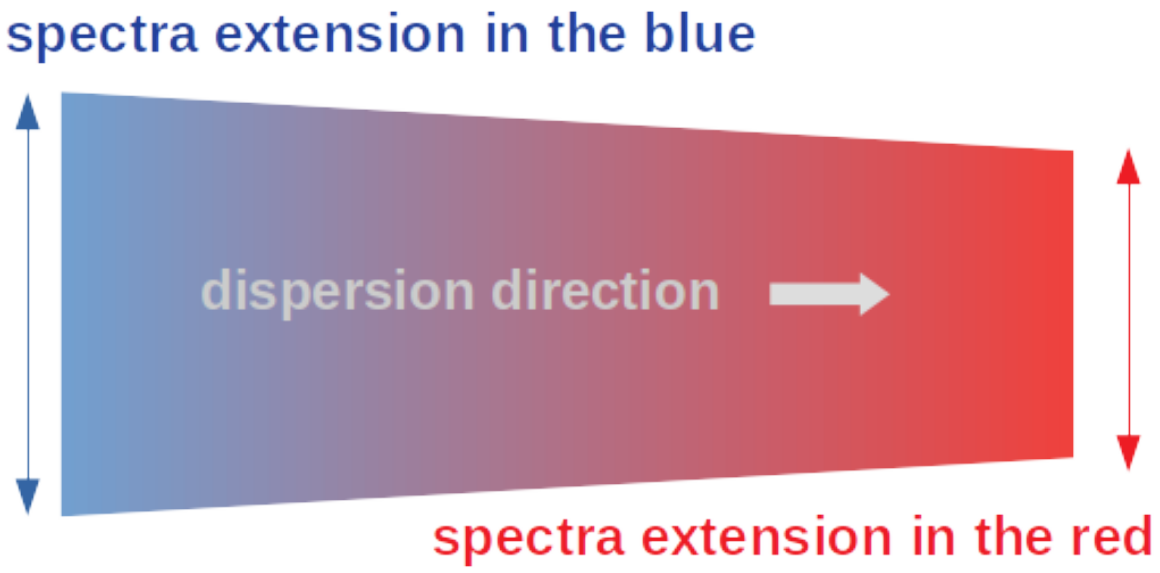
Otherwise we can increase the degree of the polynomial and repeat the previous steps.

Note: The degree of the local polynomial (1 in this example) can be increased according with the number of crosses used for each slit.

Note: The degrees of the global 2D polynomial is strictly related to the number of slit in the FOV and we MUST take into account of the slit number when we decide its degree.

7.2.1 The longslit case

The longlist case shows a single slit very extended along the cross dispersion direction. Due to optical distortions, the spectra produced by this kind of slit could have different sizes in the blue and in the red area. Namely the distance between the left and right edge in blue region slightly differs from the one in the red region.



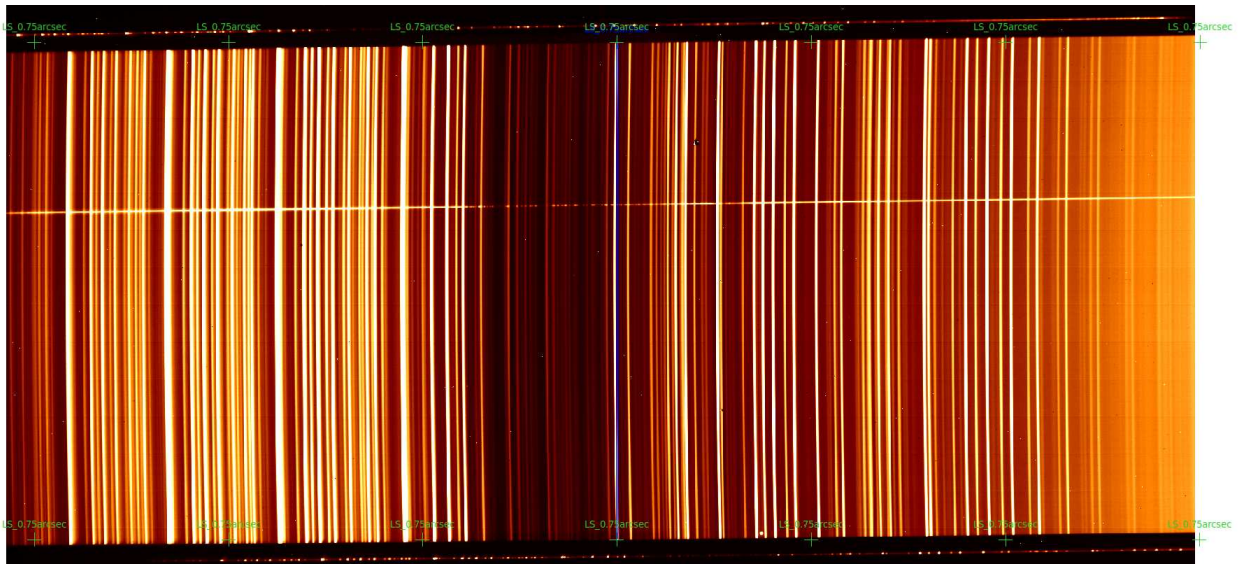
SpectraPy allow us to address this issue fitting both edges of the spectra.

```
>>> luci_mask = "conf/masks/luci_LS_0.75.mdf"
>>> luci1 = "conf/instruments/luci_G200LoRes_1.93_1.8.icf"
>>> luci_file = "examples/data/luci1LoRes/luci1.20180202.0181.fits.bz2"
>>> from spectrapy.modelscalib.calib import ModelsCalibration

>>> opt = "examples/tmp/LUCI1.opt"
>>> calib = ModelsCalibration(luci1, mask=luci_mask, opt=opt)
>>> crv=calib.new_crv_model(2, 0, 1)
```

The last call defines a local curvature model of the 2nd order, described by a global 2D model of:

- order 0 on X axis: we have just a single slit, it can not change in the FOV moving along X axis
- order 1 on Y axis: we want a model capable of fitting both edges of the spectrum, i.e. this model can change the spectra curvature along the cross dispersion direction (the Y axis)

```
>>> calib.set_trace_limits(1000, 1000)
>>> calib.load_image(luci_file)
>>> calib.plot_crv_model(7,  pos=(0, 1))
```



Unlike the MOS case, we **want** to plot both edges of the spectra, this is achieved by the `pos` parameter of the `plot_crv_model` methos.

SpectraPy describes the slit with a Bezier curve parameterized by a real value *t* with goes from 0 up to 1. So the left edge of the slit is the slit at *t=0* and the right edges is the slit at *t=1*. The parameter `pos=(0,1)` in the `plot_crv_model` call, define which region of the slit we want trace (the edges in this case).

Note: *pos* parameter can be any number between 0 and 1, that means SpectraPy can show you tracing on every point of the slit. This feature can be useful to work out with very problematic data.

In this case we must adjust both edges of the slit and refit the models[6]

```
>>> calib.fit_crv_model()
```

And check the results

```
>>> calib.plot_crv_model(100, pos=(0, 0.325, 1))
```

And finally, as usual, save the model

```
>>> crv.writeto("examples/tmp/LUCI1.crv", overwrite=True)
```

7.3 Inverse Dispersion Solution calibration

Once the spectra have been located on detector by the *Optical Model* and geometrically described by the *Curvature Model*, the wavelength calibration of the 2D spectra can be carried out.

The *Inverse Dispersion Solution Model* is the model which, following the spectrum curvatures, gives us a relation between wavelength (in Angstrom) and detector pixels.

Like for the `CRVModel`, the `IDSModel` can be calibrated either using the previous `ModelsCalibration` instance or initializing a new class which loads the already saved `OPTModel` and `CRVModel` instances.

[6] An already prepared region file is available in `examples/data/luci1LoRes/regions/crv.reg`

```
>>> #The variables already initialized
>>> mods_mask = "examples/data/mods1r/ID532016.mdf"
>>> mods1r = "conf/instruments/mods_G670L.icf"
>>> mods_arc = "examples/data/mods1r/mods1r.20180121.0073.fits.bz2"
>>> from spectrapy.modelscalib.calib import ModelsCalibration

>>> opt = "examples/tmp/MODS1R.opt"
>>> crv = "examples/tmp/MODS1R.crv"
>>> calib = ModelsCalibration(mods1r, mask=mods_mask, opt=opt, crv=crv)
```

The mathematical description of the `IDSModel` is similar to the `CRVModel`: for each slit, one mono dimensional polynomial locates the wavelength positions along the curves described by the CRVModel . The FOV variations of the coefficients of this polynomial are described by a 2D polynomial.

We start creating the `IDSModel` from scratch

```
>>> ids = calib.new_ids_model(3, 2, 2)
```

In this way, we created a model described locally by a 3th order polynomial and globally by 2x2 bi-dimensional polynomial.

In this example for the `IDSModel` calibration, we use the arc frame

```
>>> calib.load_image(mods_arc)
>>> NeHg_cat="conf/catalogs/NeHg_hr.dat"
```

In addition to the arc frame, we need a **line catalog** to know the expected arc lines position. Catalogs are ASCII files containing line positions, their formats is described the *Catalogs* paragraph.

Note: Each arcs lamps, can be acquired in different frames separately. During IDS model calibration, it could be useful to add together frames of different arc lamps, in order to span a wider wavelength range. In case we provide a list of files to the `load_image` method, SpectraPy sums them together and displays the stacked image in DS9.

7.3.1 The global mode

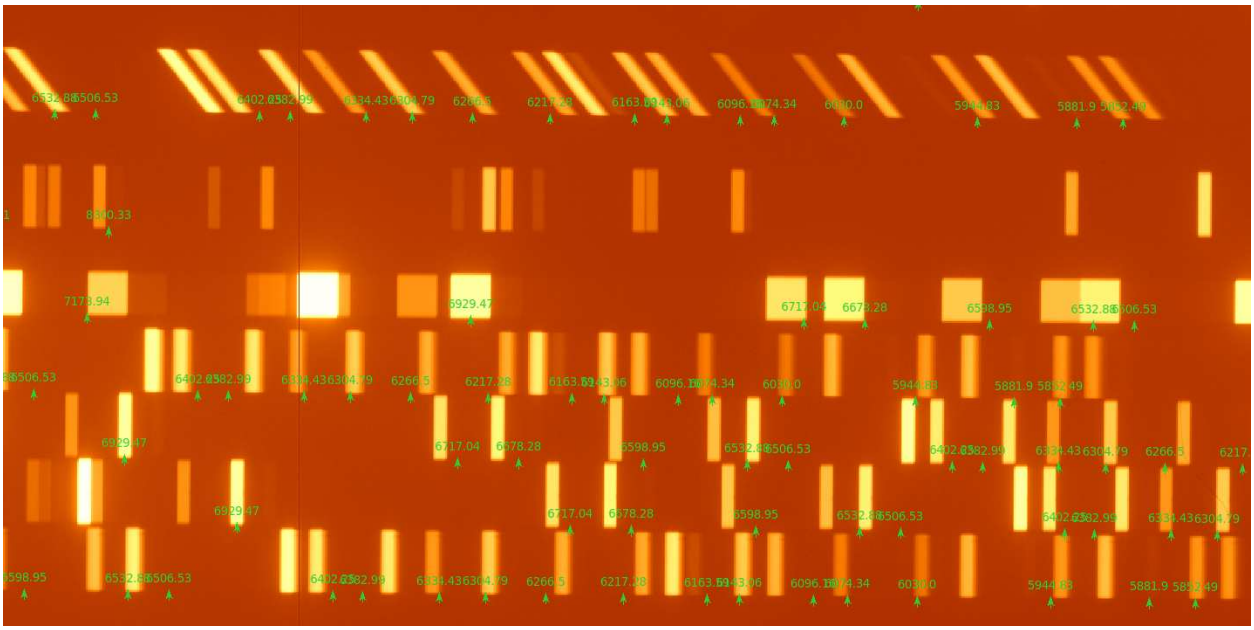
Now we can start with the model calibration. First of all we will show the global approach, by this strategy we will calibrate all the MOS slits at the same time.

```
>>> calib.plot_ids_model(NeHg_cat)
```

As we can see in DS9 we will see the expected DS9 lines positions. Since lines catalogs are related to a specific instrument configuration, they can span a wavelength range larger than range covered by the data, this is the reason why we see line region positions of of the frame.

Spectrapy allow us to load just a part of the catalog, using only lines in the relevant range. This is done by the `wstart` and `wend` parameter. In this case we ca limit the lines to the MODS red regions

```
>>> calib.plot_ids_model(NeHg_cat, wstart=5000., wend=10000.)
```

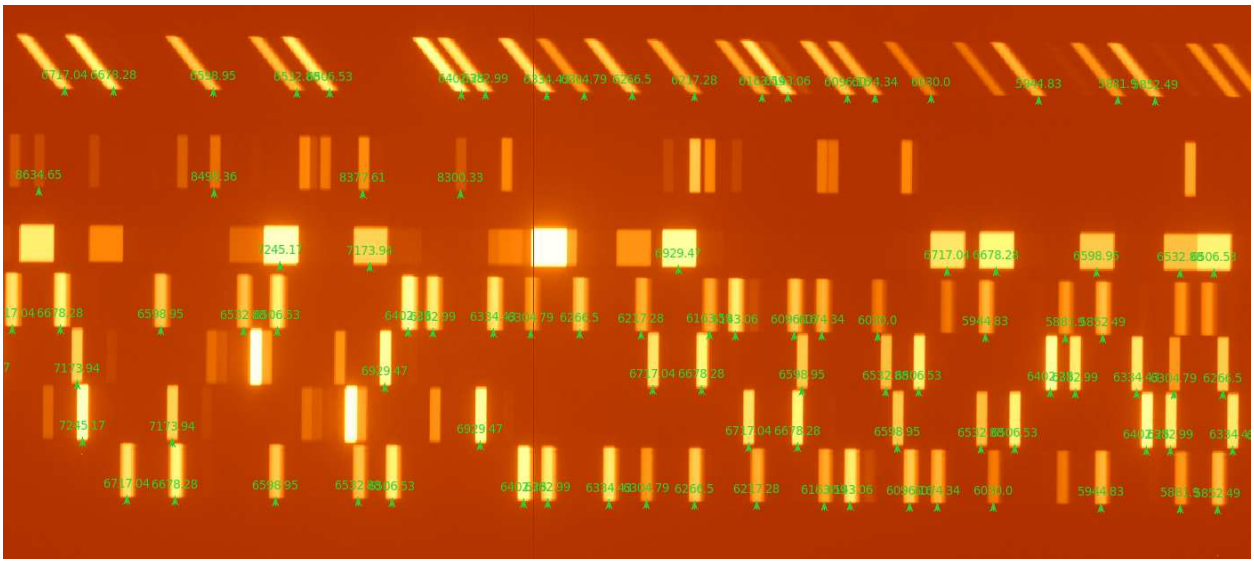


The method `plot_ids_model` displays the frame and over it, green arrows on the expected line positions. For each list, we must move the arrowhead in the proper position and then refit the model.[7] In this figure we can see how arrow positions don't match the line positions. The adjustment is not a simple global shift, we are suppose to adjust the model slit by slit, since the distortions are different along the FOV. Moreover each arrow position on the slit, must be moved of a different amount.

Note: During this adjustment, don't be afraid to delete lines which can not be properly adjusted, e.g. lines which fall out of frame.

```
>>> calib.fit_ids_model()
```

After the fit recomputing, the line positions according with the new model will be plotted.



In case the new computed model well fits the line positions, we can save it.

[7] Already prepared region file is available in `examples/data/mods1r/regions/ids_global.reg`

```
>>> ids.writeto("examples/tmp/MODS1R.ids",  overwrite=True)
```

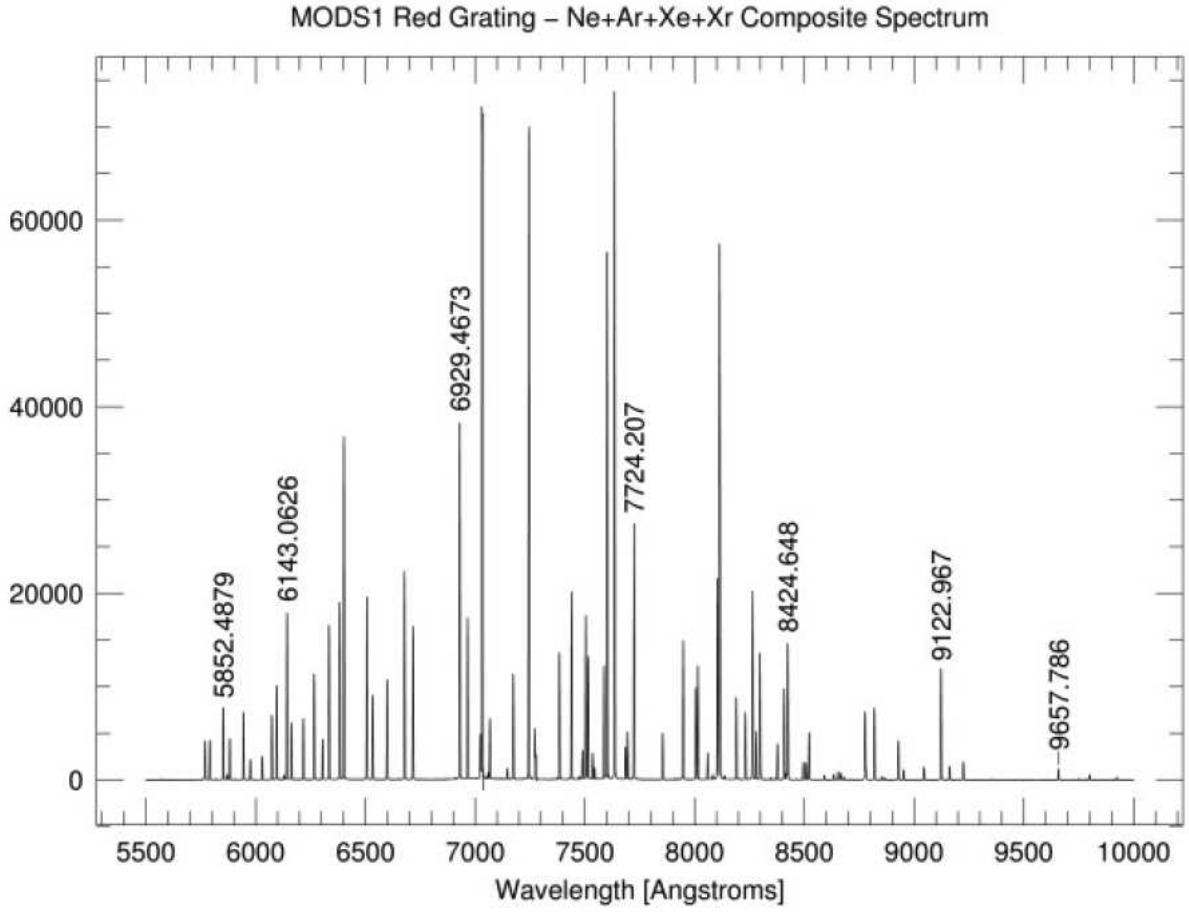
Otherwise we can change the orders of the model and repeat the previous steps.

7.3.1.1 Spectra slicing

The first time we are dealing with new data, wavelength calibration can be a very tricky exercise. We have to label emission lines in the raw frame, associating to each of them the proper wavelength value. In literature an on the web, we can find very useful references to complete this task .

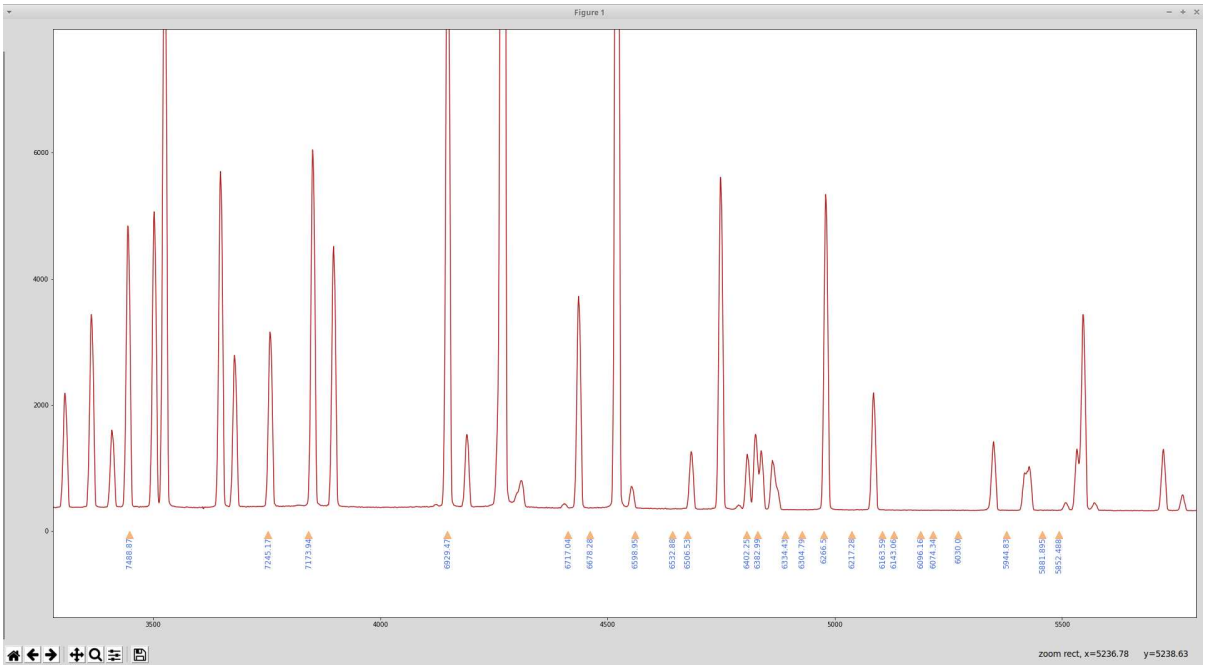
For the optical range we can suggest atomic data tables (National Institute of Standards and Technology) or for the near infrared the Rousselot et al. article could be a very useful reference for the OH lines positions.

We can find plots of lamp spectra already calibrated, like this on the LBTO page.



The comparison of this plots with 2D raw frames can be not so straightforward. For this reason SpectraPy allow us to obtain a 1D slice of a single slit using the `plot_slice` method. We each slit we can cut a slice passing the slit ID to the `ModelsCalibration.plot_slice` method. In the following example we are plotting a slice for the slit **49**.

```
>>> %pylab
>>> calib.plot_slice('49')
```



In this case this auxiliary plot shows the 1D slice of the slit 49 allowing us to compare the reference plot, with this slice and adjust properly the regions on the raw frame.

7.3.2 The interactive mode

In some cases, adjusting the entire mask could be very tricky. For example if we are dealing with a very crowded mask, this approach could not be the best choice.

For this reason, beside this **global approach**, the SpectraPy library provides also an **interactive mode**. This mode allow us to calibrate the model slit by slit.

The **interactive mode** starts from the slit closest to center of the FOV (where the distortions should be smaller), so we can focus only on this slit and adjust the solution for it; once done we can switch to the next slit, the slit closest to the current one. Moving to the next slit, SpectraPy will apply to the next slit the solution computed on the previous slit. Since the 2 slits are close each other, the new solution is normally reasonable even for this slit. In this way the manual adjustments to be made are quite small.

We will show this methodology by an example. We starts reading the already computed model and creating the IDS model from scratch, like we did in the global approach.

```
>>> calib = ModelsCalibration(mods1r, mask=mods_mask, opt=opt, crv=crv)
>>> calib.load_image(mods_arc)
>>> NeHg_cat="conf/catalogs/NeHg_hr.dat"
>>> ids = calib.new_ids_model(3, 2, 2)
```

and start the iteration process

```
>>> calib.ids_iter(NeHg_cat, wstart=5000.)
```

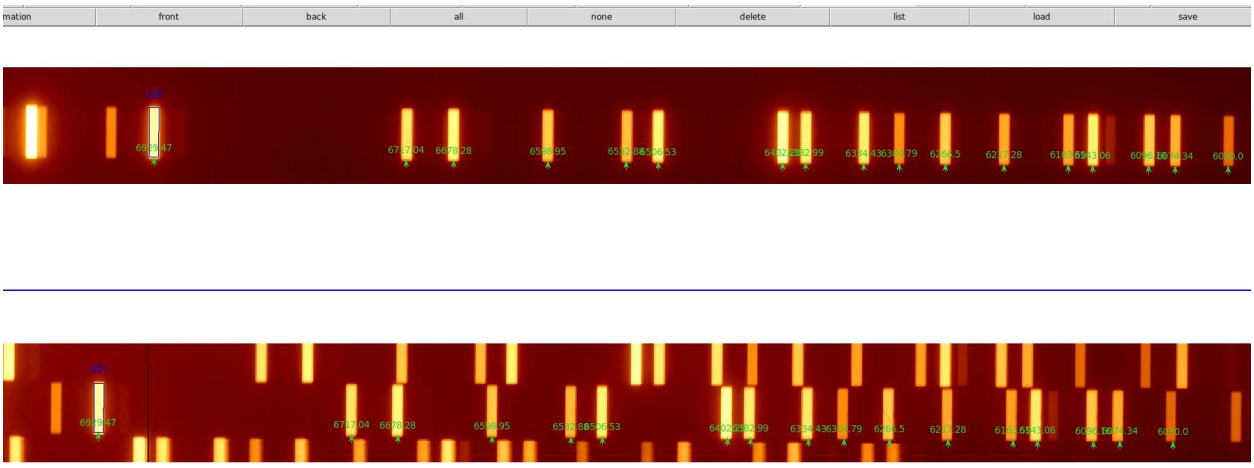
SpectraPy will display in DS9 only the central slit.

We must adjust the solution for this slit[8]; once done, we can switch to the next slit, by calling the `next` method

```
>>> calib.next()
```

Calling `next` SpectraPy performs the following actions:

- it fits the solution for the current slit already adjust
- it applies this new solution to the next slit
- it splits DS9 in 2 frames: in upper frame (or left in case dispersion is bottom-up) there is the slit already adjusted as reference, in the lower frame the new slit to adjust



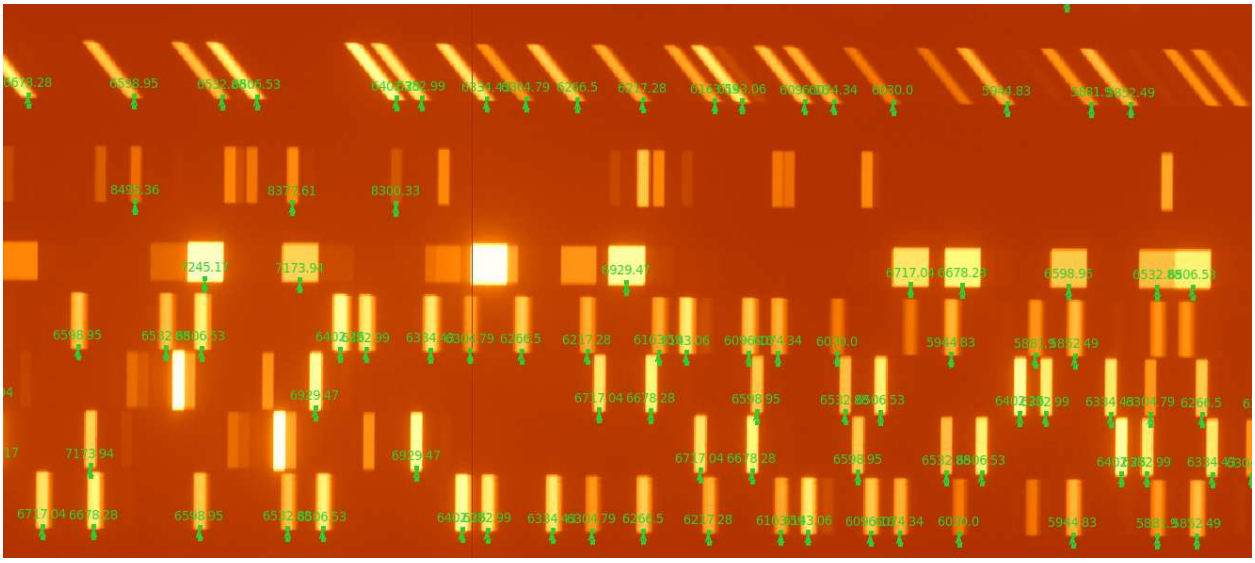
We must work on this second frame and adjust the solution for this new slit. Once done we can iterate the process calling again the `next()` method.

When we reach the last slit and adjusted it, we can stop the iteration process and fit the overall model. This is done calling the `stop_iter` method.

```
>>> calib.stop_iter()
```

Calling `stop_iter` we automatically perform the fit of the IDSModel and restore the visualization with the whole single frame

[8] The already prepared region file is available in `examples/data/mods1r/regions/ids120.reg`. One file for each region is available in the same directory.



If we are satisfied of the solution, we can save the model.

```
>>> ids.writeto('/tmp/MODS1R.ids',  overwrite=True)
```

Note: In case we need to go back and refine the solution of some slit, the `prev` method is also available .

Note: Every time we go back and forward with the `prev` and `next` method, the current slit solution is refitted by default. In case we may want to browse the slits solutions, **without refit their solutions**, we can call call `prev` and `next` methods with the parameter `fit=False`.

Note: In case we want stop the iteration without refit the model, we can also call the `stop_iter` method with the parameter `fit=False`.

7.3.3 The longslit case

Unlike the MOS case, in the longlist case the slit can be quite long, mask flexures and instrument distortions can produce spectrum with curved lines like in frame below. That means the solution of the IDSModel is different in the center of the slit with respect of to the edges.

To address this issue, SpectraPy allows to describe the dispersion solutions, defining the IDSModel solution in several parts of the slits .

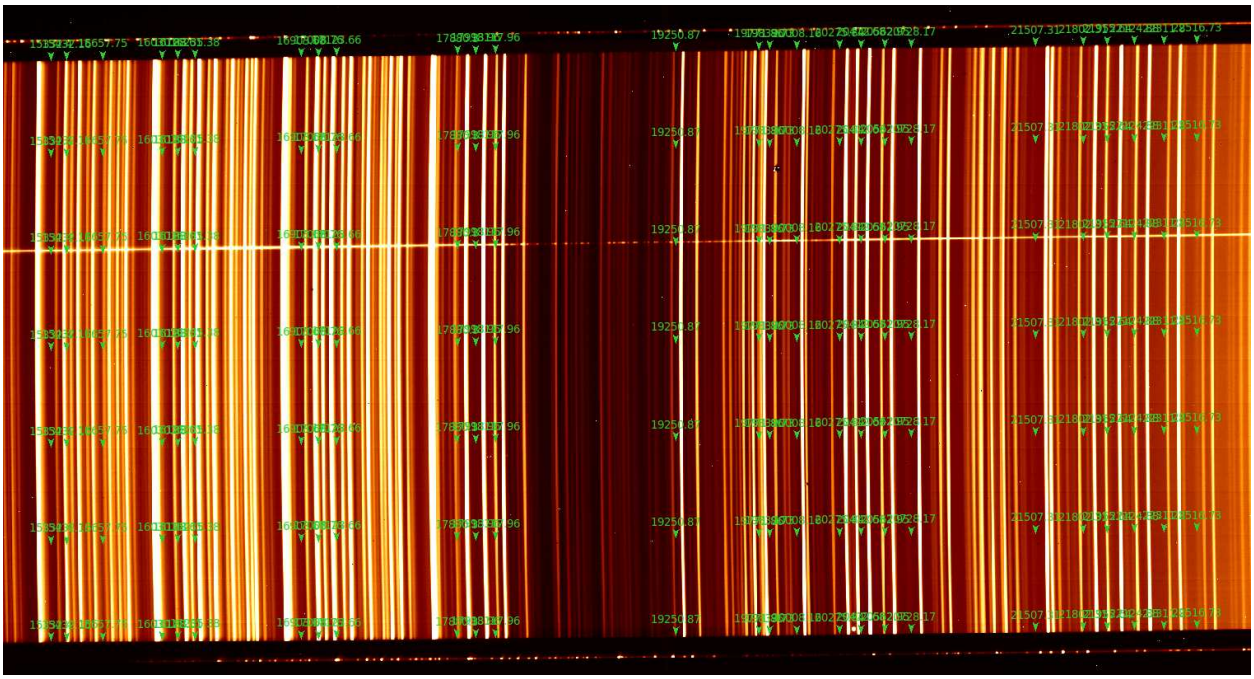
```
>>> luci_mask = "conf/masks/luci_LS_0.75.mdf"
>>> luci1 = "conf/instruments/luci_G200LoRes_1.93_1.8.icf"
>>> luci_file = "examples/data/luci1LoRes/luci1.20180202.0181.fits.bz2"
>>> from spectrapy.modelscalib.calib import ModelsCalibration

>>> opt="examples/tmp/LUCI1.opt"
>>> crv="examples/tmp/LUCI1.crv"
>>> calib = ModelsCalibration(luci1, mask=luci_mask, opt=opt, crv=crv)
```

We decide to generate a local IDSModel described by 3rd order polynomial, which does not change along the X axis (we have just 1 slit), but which changes along the cross dispersion direction since we want to follow the line curvatures

```
>>> ids = calib.new_ids_model(3, 0, 2)
```

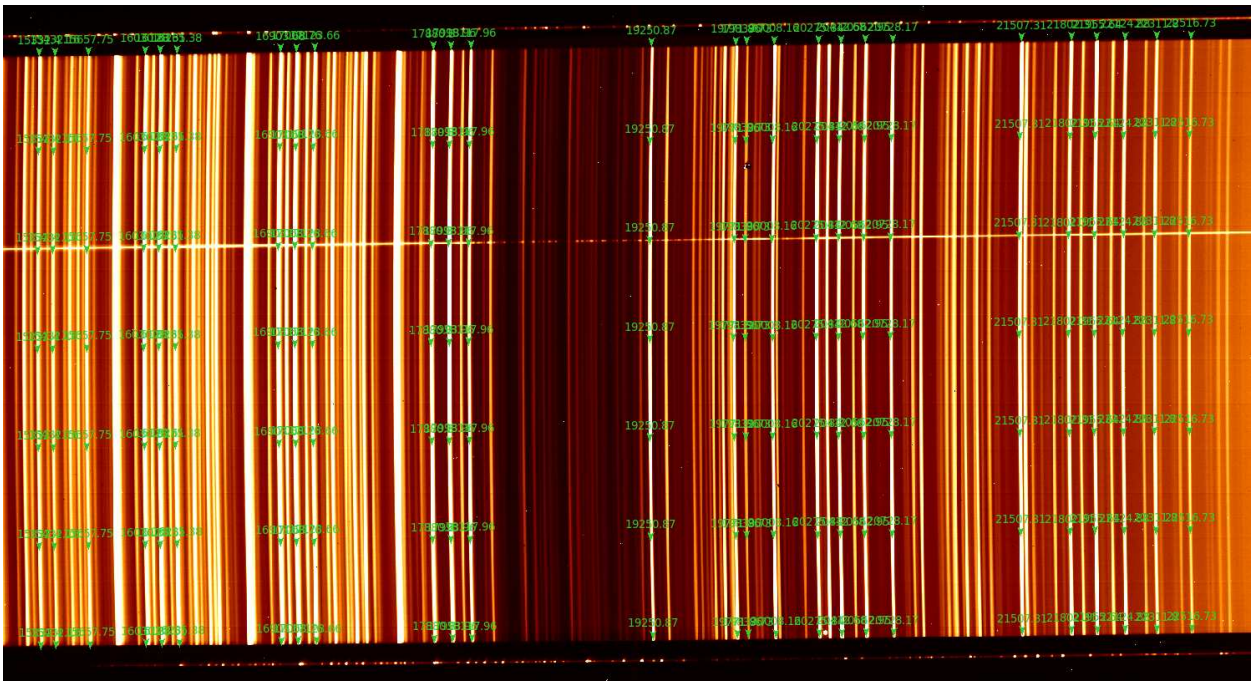
Now we can start the calibration procedure, splitting the slit in many pieces (7 in this example)

```
>>> sky_cat = "conf/catalogs/sky_lr.dat"
>>> calib.load_image(luci_file)
>>> calib.plot_ids_model(sky_cat, wstart=15000., nsplit=7)
```



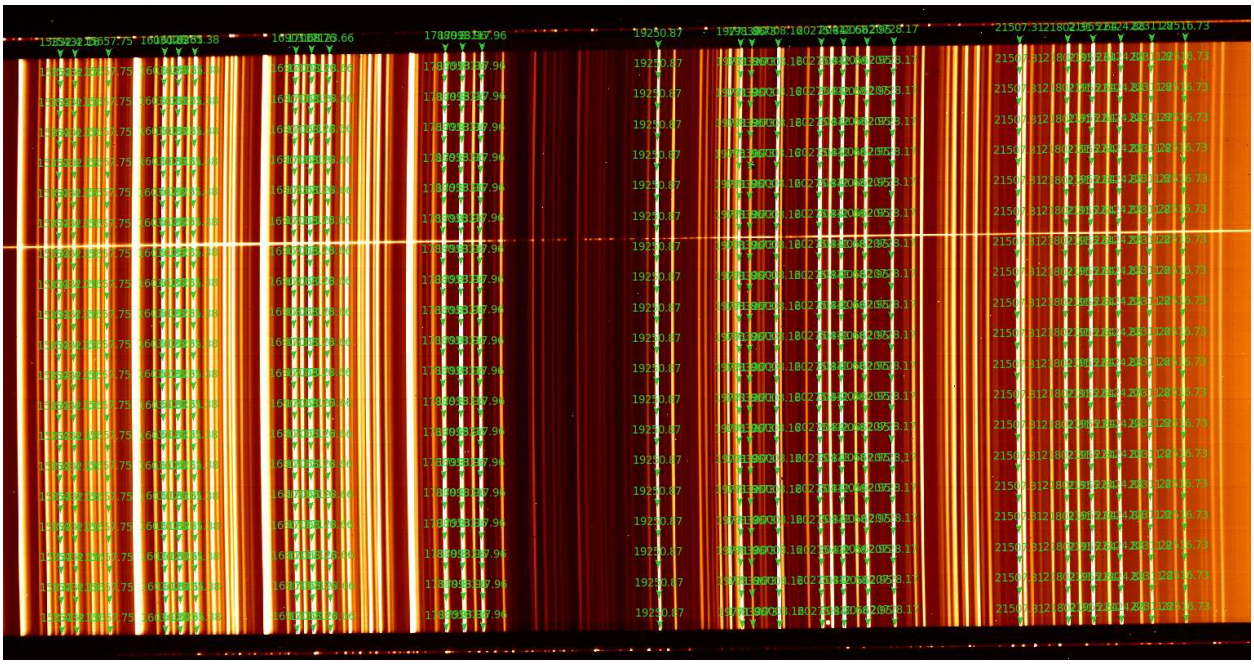
The same line position is showed 7 times along the slit. Adjusting the line position for each piece, we will instruct the model to follow the lines curvatures[9]

[9] An already prepared region file is available in `examples/data/luci1/regions/ids.reg`



Once done, we refit the model and check the solution on the frame. We can also increase the number of slits for a better check.

```
>>> calib.fit_ids_model()
>>> calib.plot_ids_model(sky_cat, nsplit=20)
```



And save the model

```
>>> ids.writeto("examples/tmp/LUCI1.ids", overwrite=True)
```

CHAPTER

EIGHT

MODEL DATA TUNING

In the ideal case, well calibrated models are ready to be used on real data. In the real world, instrument distortions, mask position uncertainty and other instrument effects affect the data in slightly different ways, producing slightly different distortions on a night by night basis. For this reason models must be adjusted on real data, this is what we call **models data tuning**.

Unlike before, now the computations of the spectra edges as well the line positions for the wavelength calibration, are automatically performed on real data and no more on the base of region files defined by the user.

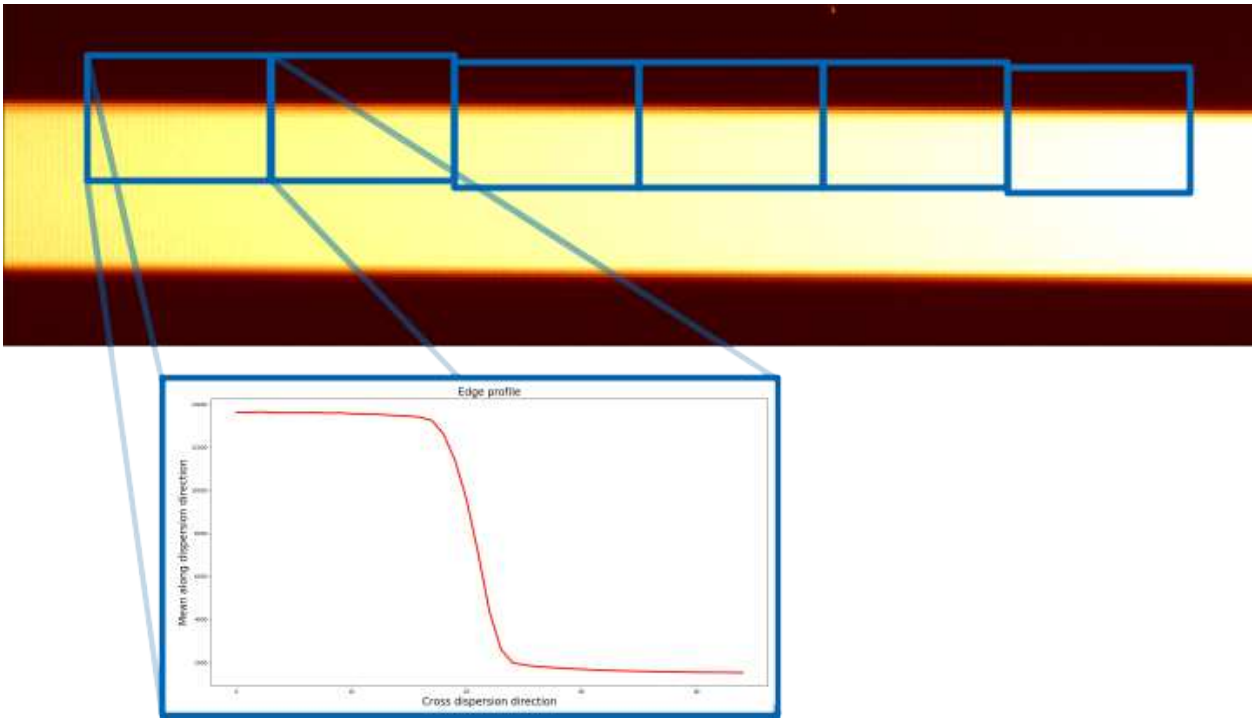
Since we are working only with dispersed data, this kind of tuning **can not** be performed on the *Optical Model*, but only on *Curvature Model* and *Inverse Dispersion Solution Model*.

In these section we will see how SpectraPy performs this tuning on real data.

8.1 Curvature Model data tuning

The accurate calibration of the spectra curvatures is performed by SpectraPy fitting the edges of the spectra on the frames. These are the main steps of the edges computing:

1. SpectraPy uses, as first guesses for the edge location, the solution of *Curvature Model* obtained by the *Models calibration* procedures
2. then it moves along this curve, cutting thumbnails containing the edge of the spectrum (blue boxes in the figure)
3. the library collapses these thumbnails along the dispersion direction and computes edge profile (the red curve in the figure) of each thumbnail
4. these edge values are used by SpectraPy to fit the model



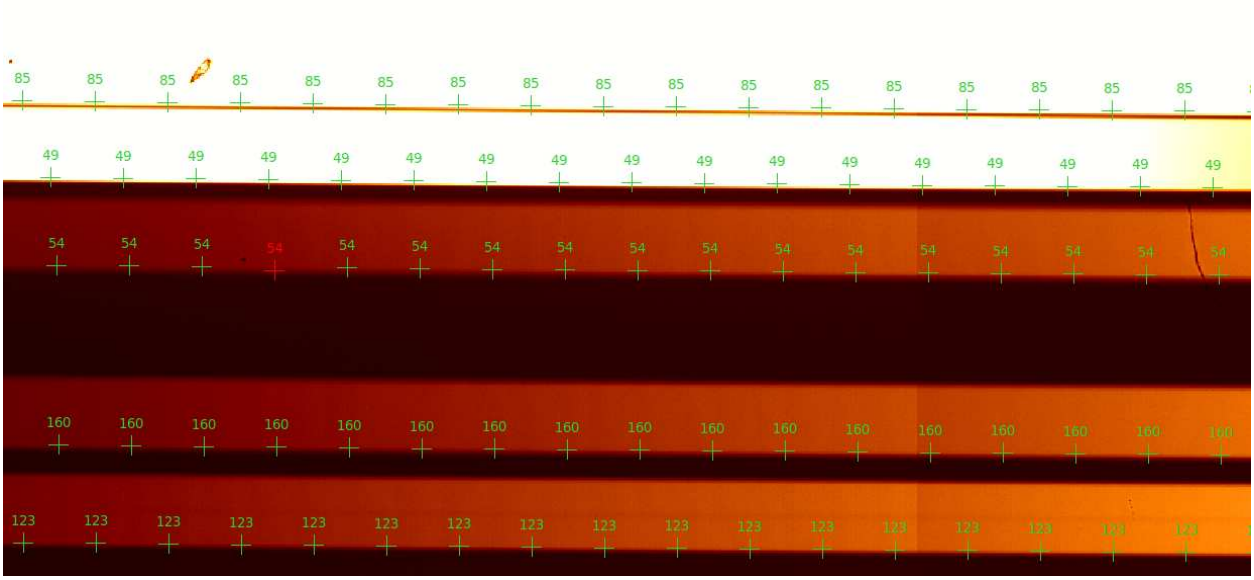
The `TraceCalibration` class is the object used by SpectraPy, to fit the edge of the spectra trace

```
>>> mods_mask = "examples/data/mods1r/ID532016.mdf"
>>> mods1r = "conf/instruments/mods_G670L.icf"

>>> mods_flat = "examples/data/mods1r/mods1r.20180121.0067.fits.bz2"

>>> opt = "examples/data/mods1r/models/MODS1R.opt"
>>> crv = "examples/data/mods1r/models/MODS1R.crv"

>>> from spectrapy.datacalib.tracingcalib import TraceCalibration
>>> trace_calib = TraceCalibration(mods1r, mods_mask, opt, crv)
```

Now we must define the limits of the tracing computations, and the sizes of the thumbnails

```
>>> # Define the trace limits
>>> trace_calib.set_trace_limits(1500, 3000)

>>> # Load the image
>>> trace_calib.load_image(mods_flat)

>>> # Define the thumbnails sizes and perform the computation
>>> trace_calib.compute_spectra_traces(slit_win=20, pix_bin=50, var=False)
```

In this example we selected a range of 1500 pixels in the blue and 3000 in the red. We decided to cut thumbnails every 50 pixels along the dispersion direction (the `pix_bin` parameter) and each thumbnail is 20 pixels large (along the cross dispersion direction). For each thumbnail the library creates the profile and computes the edge of this profile.



For visual check, SpectraPy displays in DS9 the computed edges. It places green crosses on the computed edge positions. It also shows red crosses in case of failures.

Like we did during *Models calibration*, in case we are satisfied by the results, we can use these positions to recompute the CRV model (the red crosses will be excluded from computation)

```
>>> trace_calib.fit_crv_model()
```

check the result

```
>>> trace_calib.plot_crv_model(50)
```

and save the new model

```
>>> trace_calib.writeto("examples/tmp/MODS1R.data.crv", overwrite=True)
```

Note: In case your instrument is **stable enough**, i.e. **through slits flats match science data**, we suggest to use through slits flats to compute spectra curvatures, because the spectra edges are sharper with respect to science data

8.1.1 The longslit case

Like we did during *Models calibration*, in the longslit case we want to fit both edges. This is done using the `right` flag in the `compute_spectra_traces` call.

```
>>> luci_mask = "conf/masks/luci_LS_0.75.mdf"
>>> luci = "conf/instruments/luci_G200LoRes_1.93_1.8.icf"
>>> sc_file = "examples/data/luci1LoRes/luci1.20180202.0181.fits.bz2"

>>> opt = "examples/tmp/LUCI1.opt"
>>> crv = "examples/tmp/LUCI1.crv"

>>> from spectrapy.datacalib.tracingcalib import TraceCalibration
>>> trace_calib = TraceCalibration(luci, luci_mask, opt, crv)
```

(continues on next page)

(continued from previous page)

```
>>> trace_calib.set_trace_limits(950, 1000)

>>> # We are using a science frame
>>> trace_calib.load_image(sc_file)

>>> # The right flag is now set True
>>> trace_calib.compute_spectra_traces(slit_win=20, pix_bin=50, right=True)
```

In this example, we also show how to to use a science frame to fit the trace edges instead of trough slit flat.

```
>>> trace_calib.fit_crv_model()
```

Like in the MOS case we can check the results and save them

```
>>> trace_calib.plot_crv_model(100, pos=(0, 0.325, 1))
>>> trace_calib.writeto("examples/tmp/LUCI1.data.crv", overwrite=True)
```

Note: The `right=True` flag, could be used also in the MOS case. It could be useful when we are working with crowded masks, where slits are very close one to the other, and the edges are not well defined. The `right` flag, doubles the traces (for each slit we have 2 edges now), increasing the model constraints, and this could help the fitting procedure.

8.2 Inverse Dispersion Solution data tuning

The wavelength solution is tuned on real data, searching the *real* line positions on frames.

The Optical Model gives us the slit extension on the frame along the cross dispersion direction, i.e. for each slit we know how large is the 2D spectrum.

To compute the wavelength solution, SpectraPy cuts N slices of 2D spectrum: one slice for each pixel along the cross dispersion direction. Namely, if the 2D spectrum is 23 pixels large, SpectraPy will create 23 slice of the 2D spectrum. SpectraPy can move along the slice following the Curvature Model Solution and it handles each slice like a 1D spectrum. Moving along the slice it computes the **real** line positions (relying on one line catalog). Then it computes the **slice wavelength solution** using these measured lines positions.

In details, for each slice:

- SpectraPy picks nominal line positions from the input catalog
- It uses the *Inverse Dispersion Solution Model* solution (coming from *Models calibration*) as first guess to move on the **expected** line position
- It measures the **real** line position.
- for each slice the library refits the 1d polynomial using these **real** positions.

The `WavelengthCalibration` class, is the tool used to achieve this calibration. The final product of this procedure is a structure called **Extraction Table**, saved as FITS table

```
>>> # The last CRV Module
>>> crv = "examples/tmp/MODS1R.data.crv"

>>> opt = "examples/data/mods1r/models/MODS1R.opt"
```

(continues on next page)

(continued from previous page)

```
>>> mods_mask = "examples/data/mods1r/ID532016.mdf"
>>> mods1r = "conf/instruments/mods_G670L.icf"

>>> # The first guess
>>> ids = "examples/data/mods1r/models/MODS1R.ids"

>>> from spectrapy.datacalib.wavelengthcalib import WavelengthCalibration
>>> wave_calib = WavelengthCalibration(mods1r, mods_mask, opt, crv, ids)
```

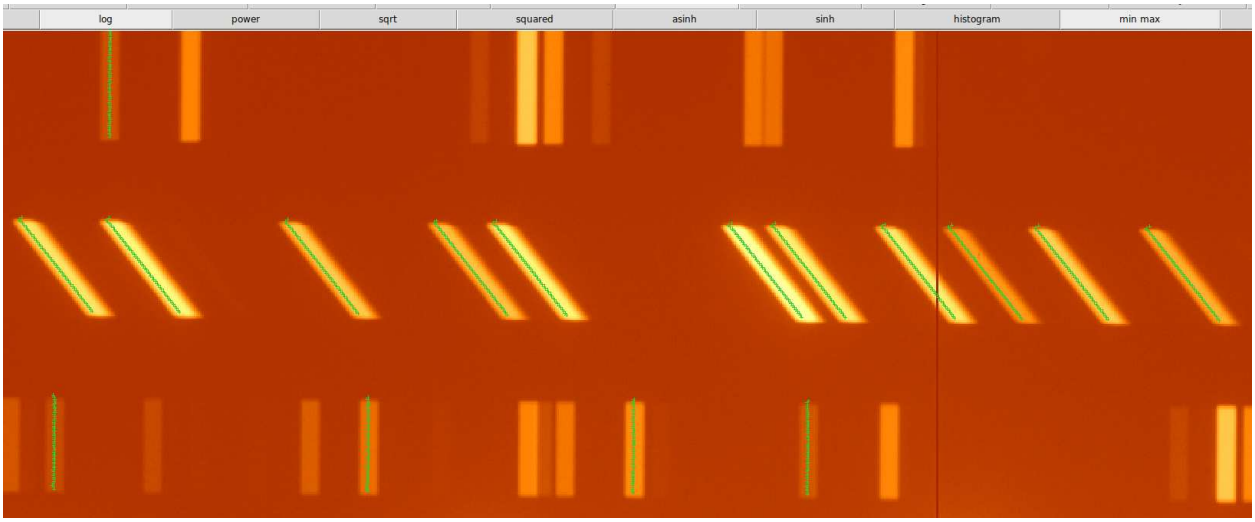
Now we must define the range of lambda we want to calibrate (in Angstrom)

```
>>> wave_calib.set_lambda_range(5000., 9000.)
```

and load a line catalog. Since in the MOS example we are using an arc frame as calibrator, we select the proper line catalog

```
>>> mods_arc = "examples/data/mods1r/mods1r.20180121.0073.fits.bz2"
>>> wave_calib.load_image(mods_arc)
>>> NeHg_cat="conf/catalogs/NeHg_hr.dat"

>>> # Do the computation
>>> ID532016_exr = wave_calib.compute_spectra_wave(NeHg_cat)
```

During this computation we will see appear crosses in the DS9 viewer on the **measured** lines positions: one cross for each line, each slit and each slice.



Note: Since this is a completely automatic process, only reliable catalog lines, i.e. lines with flag 1 (see Catalog section for details) will be used.

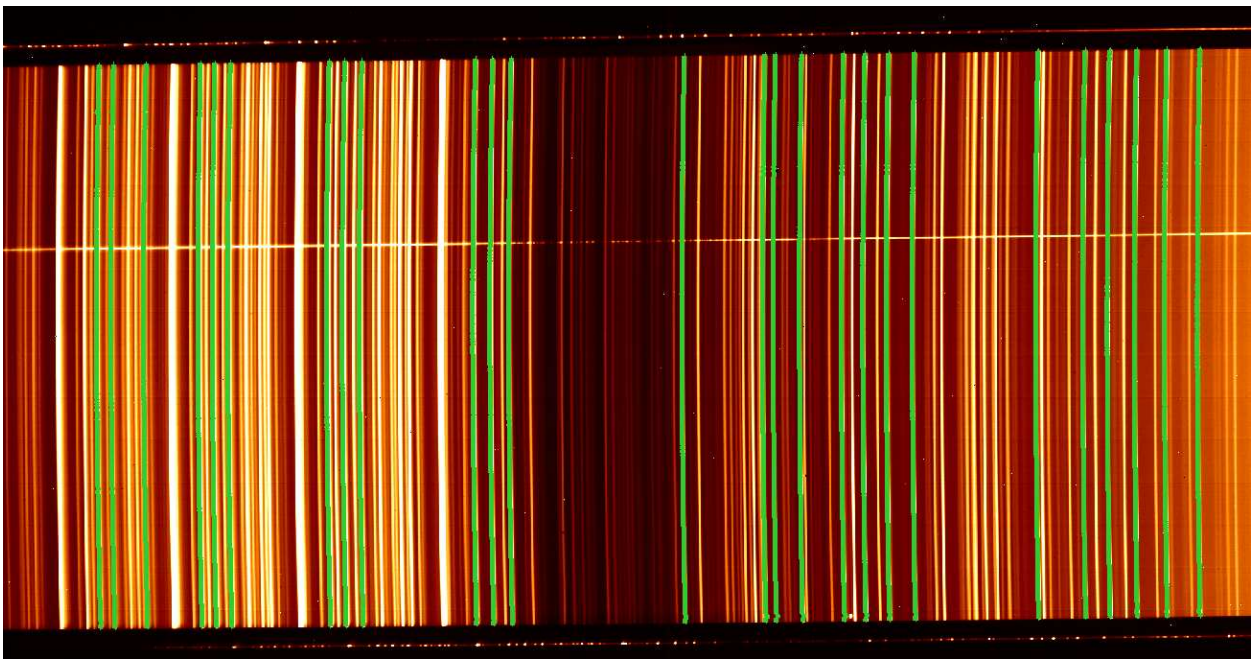
In case we are satisfied by the results, we can save it

```
>>> ID532016_exr.writeto("examples/tmp/ID532016.exr", overwrite=True)
```

8.2.1 The longslit case

For longslit data, the procedure is the same. We just reports the list of commands, but there are not differences compared to the MOS case.

```
>>> luci_mask = "conf/masks/luci_LS_0.75.mdf"
>>> luci = "conf/instruments/luci_G200LoRes_1.93_1.8.icf"
>>> sc_file = "examples/data/luci1LoRes/luci1.20180202.0181.fits.bz2"

>>> crv = "examples/tmp/LUCI1.data.crv"

>>> opt = "examples/tmp/LUCI1.opt"
>>> ids = "examples/tmp/LUCI1.ids"

>>> from spectrapy.datacalib.wavelengthcalib import WavelengthCalibration
>>> wave_calib = WavelengthCalibration(luci, luci_mask, opt, crv, ids)
>>> wave_calib.set_lambda_range(15000., 22000.)
>>> wave_calib.load_image(sc_file)
>>> sky_cat = "conf/catalogs/sky_lr.dat"

>>> LS075_exr = wave_calib.compute_spectra_wave(sky_cat)
```



and save the extraction table

```
>>> LS075_exr.writeto("examples/tmp/LS075.exr",overwrite=True)
```

Note: The Extraction Table contains the solution **for a given mask**, i.e. it is no more a generic model like OPTModel, CRVModel or IDSModel. It contains the 3 models applied to the current mask. If we want to extract spectra acquired with the same instrument configuration, but with another mask, we can tune the valid models on our new data, and we **must create a new Extraction Table**.

CHAPTER

NINE

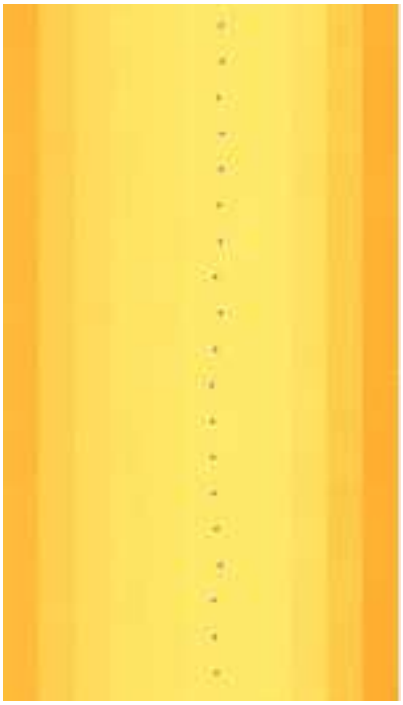
SPECTRA EXTRACTION

9.1 The Extraction Table

The `ExtractionTable` is the final product of the IDSDataCalib and it is the key point to perform the spectra extraction. Given the mask and the instrument configuration, that table contains the all information to achieve the 2D spectra extraction. This table is stored as a FITS table.

As described in the previous sections, during IDS data tuning, each slit is sliced and the solution of each slice is computed independently by the others.

If we focus on a single line, the IDS solution for this line can be jagged and not as smooth as in the frame appears. It can also happen that the computation of the solution for some row fails (cosmic, bad pixels or other effects can affect the results on that row). Here we can see the effect previously described, in blue the expected positions.



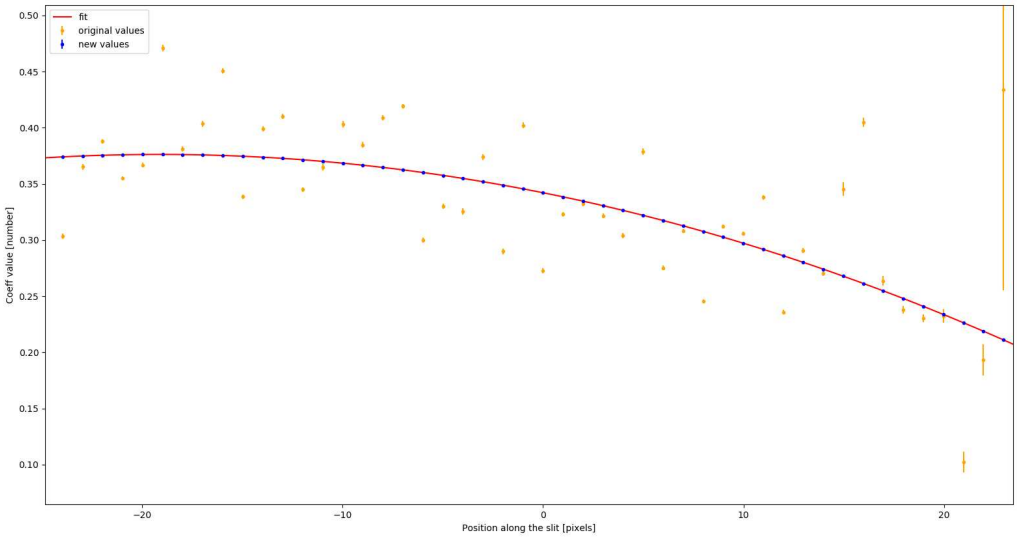
To avoid these problems SpectraPy provides a methods to rectify the solutions contained in the Extraction Table and fill any gaps in case of row failures.

9.2 Extraction Table rectification

This rectification process is performed slit by slit, since the Extraction Table handles each slice separately. For each slit there are sets of coefficients which are slightly different slice by slice.

> Focusing on a single slit, and fixed the order of the polynomial coefficient, SpectraPy gathers together the coefficients of all the slices. For each coefficient group, SpectraPy fits a polynomial in order to find a smoother variation of the current coefficient. The final result, is to have a smoother overall solution for the entire slit.

In the figure below for a given slit, we can see in detail the values of the 0th coefficient slice by slice; the coefficient of each slice is slightly different from the value of the next slice (the orange points). The red curve is the fit along the slits (slices position are in arbitrary units). The evaluation of the fit on each slice positions gives us the new 0th coefficients (the blue points).



Here the lines of code to rectify the solutions

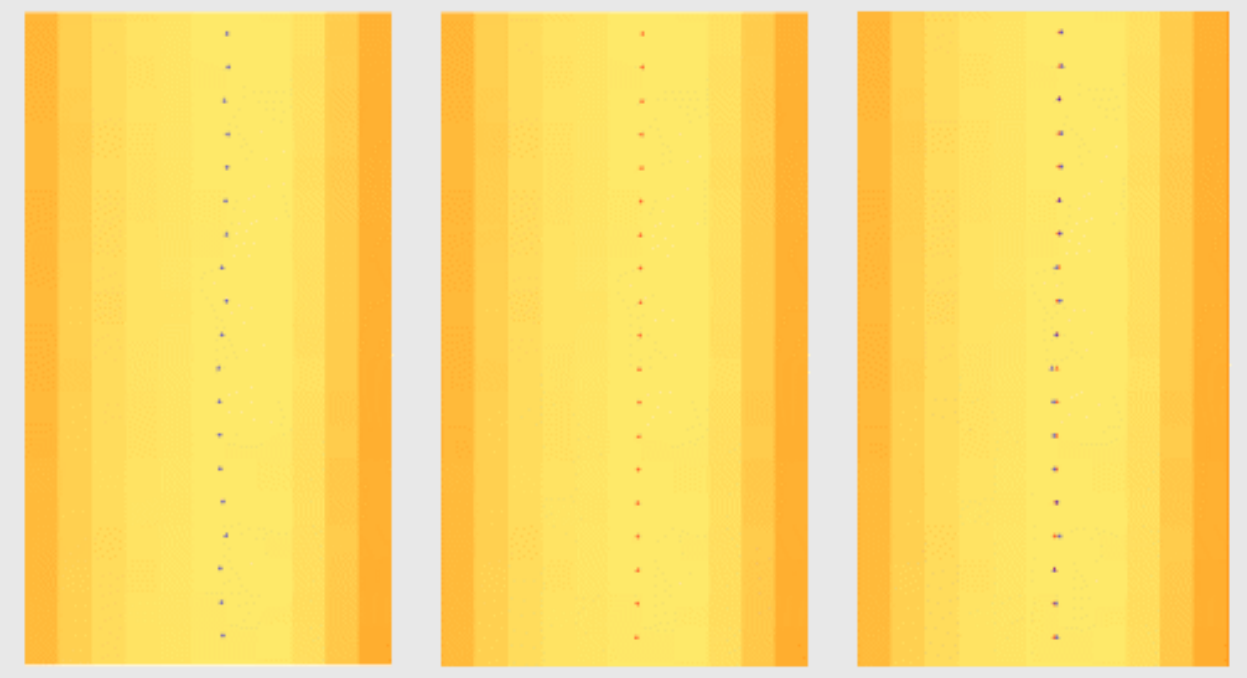
```
>>> from spectrapy.datacalib.extractiontable import ExtTable
>>> exr = ExtTable.load("examples/tmp/ID532016.exr")
>>> exr.rectify(deg=2, margin=1)
```

In this case we choose a polynomial of 2nd order (parameter `deg=2`) and we excluded the 1 pixel (both sides) at the edges of the slit (`margin=1`), this is done just to avoid possible mask cutting issues.

Or, if we prefer, we can just refit the single slit

```
>>> exr['121'].rectify(deg=2, margin=2)
```

If we display the result of the rectified Extraction Table we can see the solution on a single line is smoother than before (the blue crosses before rectification, the red crosses after rectification).



9.3 Check the Wavelength Solution

Even if the wavelegenth solution contained in the Extraction Table is very precise, this solution could be not the best once applied on some science images. For example, this can append because distortions could be different night by night, or occasionally mask can slightly drift away along the night. In order to achieve these kind of problems, or at least to aware of them, the current SpectraPy release provides us one class to check and inspect the quality of the wavelength solution applied on a given image.

These checks are performed by the `WavelengthCalibCheck` class.

We must initialize the class with the Extraction Table to check, load the desiderd image and run the `check_ext_table` method. This method can be run, applying multiple constraints:

- the input catalog: to select the lines to check
- the wavelength range: to limit the lambda range to check
- the slits IDs: to select only some slits instead of the whole mask
- the rows range: to check just a region of the slit

```
>>> from spectrapy.check.wavelengthcheck import WavelengthCalibCheck
>>> exttable = "examples/tmp/ID532016.exr"
>>> arc_frame = "examples/data/mods1r/mods1r.20180121.0073.fits.bz2"
>>> catalog = "conf/catalogs/NeHg_hr.dat"
>>> check = WavelengthCalibCheck(exttable)
>>> check.load_image(arc_frame)
>>> results = check.check_ext_table(catalog, wrange=(6000., 9000.), slits=('49', '54'),␣
↪row_start=10, row_end=20)
```

In this example we decide to:

- check only slits 49 and 54

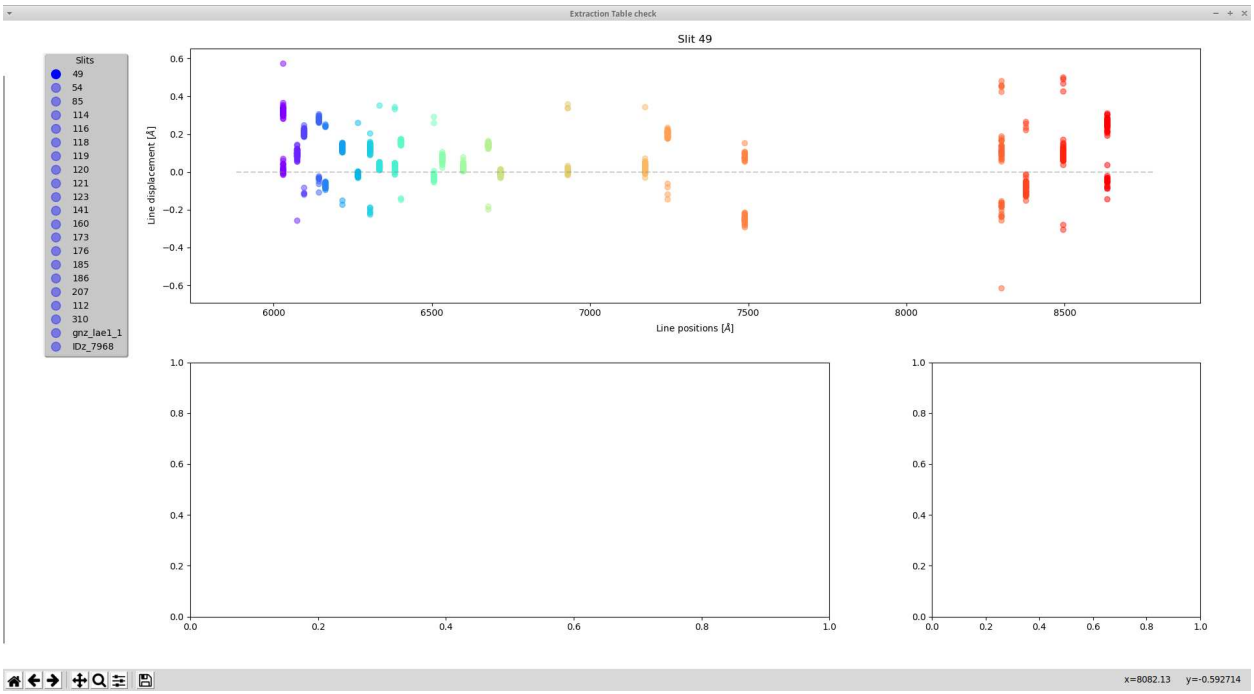
- use catalog lines in the range between 6000 and 9000 Angstroms
- check just the slits rows between 10 and 20

For each selected slit and row, the `check_ext_table` method starts from the expected positions, given by the Extraction Table. then it computes the real line position around these expect positions. This is done for each available line the catalog.

This method collects returns in a class, which allow us to display and browse these results.

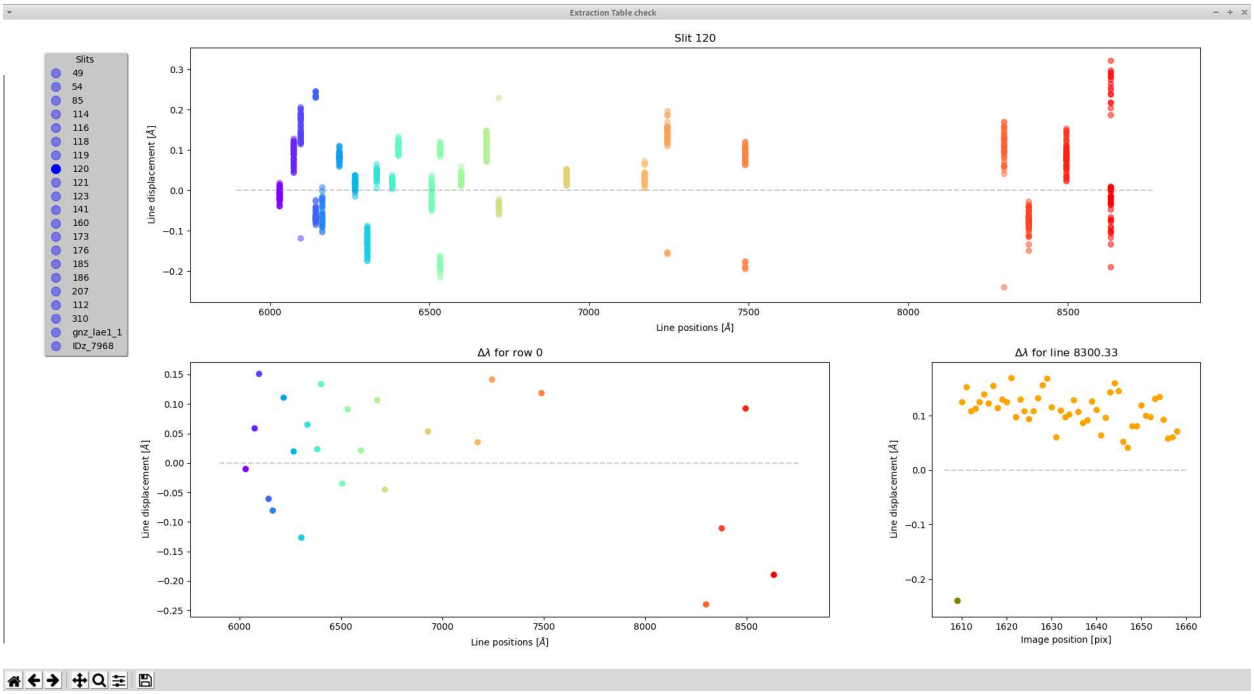
Here below, how to display results

```
>>> results.plot(legend=True)
```



The class shows us a `matplotlib` figure containing 3 axes. In the main frame, the upper one, we can see the displacements between expected line positions and local measurement of the line position. In case the legend parameter is `True`, a legend with the slits ID is showed on the left side of the frame. This legend allow us to browse along slits, selecting the desired slit, clicking on the blue bullets. We can also browse along slits pressing `n` key for the next slit and `p` for the previous.

We can also inspect more details of the solution. Clicking with the left mouse button on a single marker of the main frame, the solutions related the a single row will be displayed in the bottom left panel. In case we want focus on the quality of the solution for a single line we can click with the right mouse button on a marker of that line, and the line solution will be displayed on the bottom right panel.



9.4 Spectra Extraction

The `ExtractionTable` table contains all the information to extract spectra for a given mask and instrument configuration.

Applying the `ExtractionTable` on data, we obtain a multi-extension FITS file, which contains the 2D spectra wavelength calibrated and corrected by optical distortions. In case variance (or errors) on raw data are provided, the rebinning procedure provides also the variance on the re-binned spectra. For each rebinned pixel a quality rate is computed; this is a real number from 0 to 1 which gives us the percentage of pixels used during the rebinning procedure. These rates are stored in the multi extension as well the variance and the data.

To perform the extraction we must initialize a re-binning engine

```
>>> from spectrapy.extraction.exponentialfilter import ExponentialFilter
>>> engine = ExponentialFilter(2, 1000)
```

The current implementation of SpectraPy provides the Exponential filter as re-binning engine, which requires:

- the radius of the resampling kernel (2 pixels in this case)
- the sub-pixels accuracy in the re-sampling for each pixel (1000 by default)

The resampling procedure requires a lot of CPU time and the extraction can take a very long time. For this reason beside the pure Python class we develop also a Cython class which is faster. In the Cython class the radius and the sub-pixels parameters can not be tuned, the class uses 2 pixels as radius and 1000 as subpixels. This choice is quite standard.

In case, for any reason, we want change these parameters, we must set the `cython` flag to `False` during the class initialization.

The re-sampling engine must also know the extraction range we are interested in, and the re-binning step of the re-sampled spectra. In this example we decided to extract from 6000 up to 9000 Angstrom, creating spectra linearly re-sampled every 0.8 Angstrom.

```
>>> engine.set_extraction_range(6000., 9000., 0.8)
```

Note: A good choice for resampling value (0.8 Angstrom in this example), could be the nominal grism dispersion value.

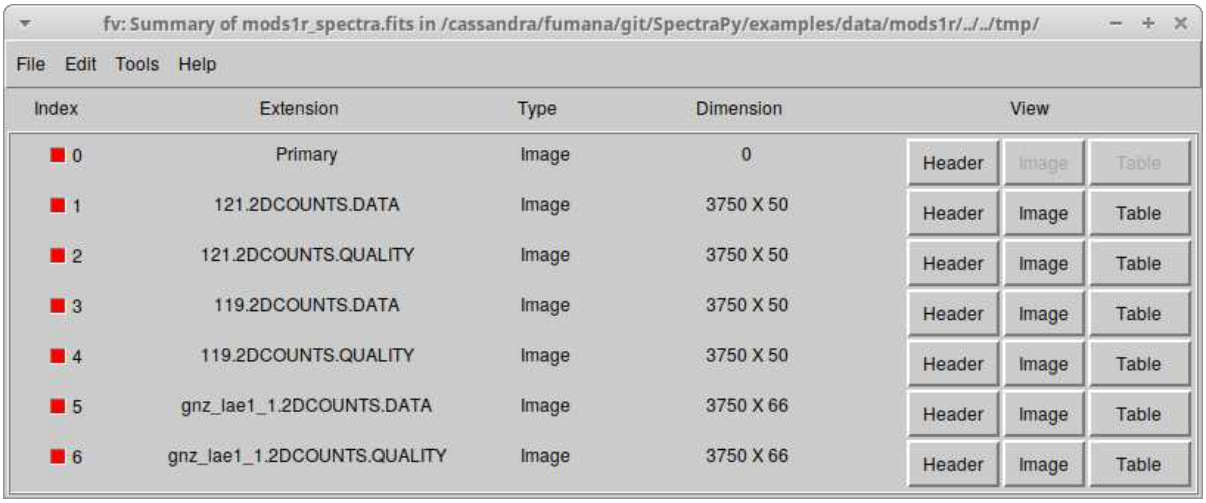
Now, we can do the real job starting the extraction. By default the extraction will extract all the spectra described in the `ExtractionTable`, but we can also decide to extract a sub-sample of them providing a list of ID, as defined in the *Mask description* file.

```
>>> from spectrapy.datacalib.extractiontable import ExtTable
>>> #Load the extraction table and rectify it
>>> exr=ExtTable.load('examples/tmp/ID532016.exr')

>>> #Extract the spectra
>>> objlist = ('121', '119', 'gnz_lae1_1')
>>> mods_arc = "examples/data/mods1r/mods1r.20180121.0073.fits.bz2"
>>> spectra2d = engine.extract(exr, mods_arc, objlist)
```

and save the spectra.

```
>>> spectra2d.writeto('examples/tmp/mods1r_spectra.fits')
```

This produces a multi extension FITS file the file in figure below, where extension name is combination of: slit id, data content (in the current version only 2DCOUNTS is available) and type of data (data, variances of quality flags).



The selection can be performed also on a sub region along the cross dispersion direction. For example in the longslit LUCI case, we can decide to extract just the region around the object itself.

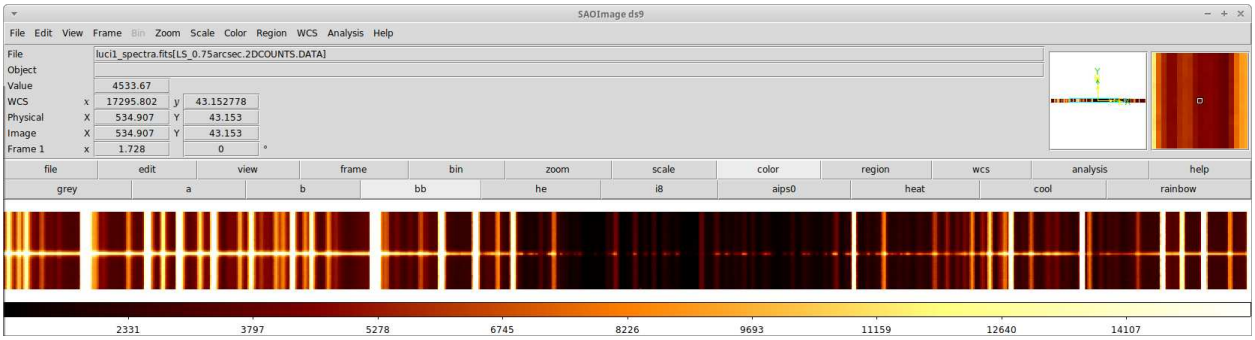
```
>>> from spectrapy.extraction.exponentialfilter import ExponentialFilter
>>> from spectrapy.datacalib.extractiontable import ExtTable

>>> exr=ExtTable.load('examples/tmp/LS075.exr')
>>> engine = ExponentialFilter()

>>> # Set the extraction range
>>> engine.set_extraction_range(15000., 22000., 4.3)
```

(continues on next page)

(continued from previous page)

```
>>> # Select a sub regione along the spatial direction
>>> sc_file = "examples/data/luci1LoRes/luci1.20180202.0181.fits.bz2"
>>> spectra2d = engine.extract(exr, sc_file, row_start=250, row_end=310)

>>> #And save it
>>> spectra2d.writeto('examples/tmp/luci1_spectra.fits', overwrite=True)
```

The data content of the extracted spectra appears like this



9.5 Spectra Extraction Adjustment

Models are computed on a set of frames and then applied on the science frames (usually another set of frames) to perform the spectra extraction. In some cases the match between the 2 set of data can be perfect.

Data can show tiny differences along the night, due to different instrument distortions at different positions of the telescope. Or due to slight shifts of the mask along the night.

In order to compensate these changes, SpectraPy allow us to adjust the solution contained in the ExtractionTable on the data we are extracting. The `extract` method of the `ExponentialFilter` class has 2 optional parameters to perform this adjustment: `xadjust` and `lines`.

If the `xadjust` parameter is `True` the recipe recomputes both left and right edges, at the lambda reference position, of the spectra selected by the `objlist`. Then it computes the difference between these new edge positions and the edge positions expected by the ExtractionTable. The **shift along the cross dispersion direction** is the sigma clipped mean of these differences.

The parameter `lines` is used to compute the shift along the dispersion direction. If this parameter is a valid line catalog containing just few very bright isolated lines, SpectraPy uses these lines to compute the differences between theirs expected positions (according with the ExtractionTable solution) and theirs real position on the frame. The average the differences between expected and real positions is the **shift amount along the dispersion direction**.

Here just an example of the use of these parameters

```
>>> spectra2d_adj = engine.extract(exr, sc_file, row_start=250, row_end=310,␣
↪xadjust=True, lines=[16235.376, 17880.298, 19350.119, 21802.312])
```