Technical Manual

SpecTracer

Contents

Introduction

This manual is intended to guide the user through the processing of the data stored in the FITs file generated by the spectrograph in the SARA telescope at Kitt Peak Observatory. The data in the file is extracted and processed in order to retrieve the star's spectrum. All the significant processes and algorithms are discussed in this manual. The contents are structured to coincide with the processing happening if the user used the program for the first time. Also, the manual is split into two columns. The leftmost column contains explanations and additional information, whereas the rightmost column shows the snippets of code in charge of carrying out the function being explained.

Creating a Virtual Environment

The program was built to work on Python 2.7.8. Therefore, it is important that the installed version of Python on the computer is this one, along with the adequate library versions. If this is not so, there will be serious problems in programming syntax and will make the program not work at all. As of this writing, Python 3 has already been released, and is the default for version when installing Anaconda distribution. Therefore, it is important to create a virtual environment running Python 2.7.8 and the libraries in the needed versions. The code on the right shows how to create the virtual environment from the console. It is important that conda is up to date. “Snakes” is an arbitrary name given to the virtual environment. It can be saved under any name.

```
>conda update conda

>conda create --name snakes python=2.7.8
anaconda
```

By installing the anaconda distribution, it ensured that all the libraries used by the program are available and installed. On the right is a way to check their versions along with the version they should be.

```
>python
>>>import pip
>>>pip.main([‘‘show’’,’’numpy’’])

numpy 1.9.0
scipy 0.14.0
matplotlib 1.4.0
astropy 0.4.2
```

If one of the versions is different, the command right below the version list should be executed.

```
>python -m pip install numpy==1.9.0
```

Finally, to use the virtual environment created, in the command line the instruction on the right should be typed.

```
>activate snakes
```

To verify the version this instruction should be executed. (Note: it should be a capital V).

```
>python -V
```

Now the code can be executed by:

```
>python [Path]SpecTracer.py
```

Once the environment is no longer needed this command should be executed.

```
>deactivate
```

Bias, Dark and Overscan Calibration

The program calibrates itself to account for noise and stray light that might have gotten into the detector.

The first source used to calibrate is the overscan. The overscan, is the region in the detector outside the lens or detection region. The values for each row of pixels in overscan are summed and averaged. Each individual average of the row is plotted. The mean value is obtained by fitting a constant through the data. This value will be subtracted from all the pixels in the FITS file.

Afterwards, the program asks the user to select a folder containing all the bias files. The bias files are created by taking a 0 second exposure with the detector. The program gets the data from all of the files and finds the median value per pixel. This value is also subtracted from the data when analyzed. The overscan is masked so it is not taken into account in the calculations.

The same process is repeated for the dark files. The dark files are created by taking an exposure, with the same duration as the observation measurements, with the lens covered. From the median value extracted from the images, the overscan mean value is subtracted along with the mean bias value per pixel. Also, the overscan regions are masked. This in order to avoid accounting for the same source of error twice.

All of this information gets stored in various file with the command shown on the right.

```
left_Over =
np.sum(flat_data[:,0:left], axis =
1)
right_Over =
np.sum(flat_data[:,right:length],
axis = 1)
overScan = left_Over+right_Over
            overScan =
overScan/((left+1)+(length-right))
#left and right are
coordinates,therefore the number of
pixels is the coordinate on th eleft
plus 1, and the total length minus
the starting coordinate of the right
overscan
z = np.polyfit(x_new, overScan, 0)
f = np.poly1d(z)
meanOverScan_val = f(0)


bias_data =
np.ndarray(shape=(len(biases_arr),le
n(temp_data), len(temp_data[0]))

bias_data[i] =
fits.getdata(biases_arr[i])-
meanOverScan_val

bias_data[:,0:left] = ma.masked
bias_data[:,right:length]=ma.masked

medianBias_val =
ma.median(bias_data, 0)

dark_data[i] =
fits.getdata(darks_arr[i])-
meanOverScan_val-medianBias_val

medianDark_val =
ma.median(dark_data, 0)

saveToFile('bias', medianBias_val)
saveToFile('dark', medianDark_val)
saveToFile('overscan',
meanOverScan_val)
saveToFile('overscanloc', [left,
right])
```

Order Modify

Once the path to the FITS file is given to the program, the data is extracted using the routine shown on the right. Also, the program checks that the dimensions of the data matrix are correct.

```
data =
fits.getdata(nameFile_orderCalib)
g_data = forceLandscape(data,
isLandscape)
```

The user is asked to select all the points that correspond to usable orders. The polyfit() function finds the coefficients of a polynomial that best fits the data given. In this case it was fixed to a second degree polynomial, since the orders closely resemble a segment of a parabola. This coefficients are then sent through another method: poly1d(). This one creates a working function using the coefficients which can be used to find values at any point in the fitted parabola.

```
fitX= np.append(fitX, xCoord)
fitY = np.append(fitY, yCoord)
```

```
z = np.polyfit(fitX,fitY,2)
```

```
f = np.poly1d(z)
```

Afterwards, the parabola is fitted to all of the orders and the user tweaks by multiplying by a coefficient, drifting left or right, or panning up or down.each individual order, in order to get a better fit.

```
g_proper_orderLoc=np.append(g_proper
_orderLoc, yCoord)
g_multiplier[choice] =
g_multiplier[choice]*correction
v_pan[choice]=v_pan[choice]+pan
g_drift[choice]=g_drift[choice]+dan
```

All of this information is stored as an array in numpy type files, which prevent accidental modification (as the information can only be accessed through the numpy library).

```
saveToFile ('correction',
g_multiplier)
g_orderLoc = g_orderLoc+v_pan
saveToFile ('displacement',
g_orderLoc)
saveToFile ('drift', g_drift)
```

Sensitivity Calibration

This routine corrects for variations in sensitivity from pixel to pixel. It does this order-wise and by normalizing the flux obtained on each pixel, on each order, by applying a gaussian filter.

In this routine all of the previously generated files are loaded, along with the image taken by the spectrograph of a flat lamp. The mean overscan value and the mean bias value per pixel are subtracted from the image. The fit created in the order tracing routine, is used to find all of the orders for which the normalization will be carried out.

```
flat_data =
fits.getdata(nameFile_flat)

flat_data = flat_data-
medianBias_val-meanOverScan_val
```

Each pixel is traced monotonically using the fit previously created, and tracing it from pixel 0 to the last one on the image, and then displacing it upwards one pixel to account for the thickness. This pixels are stored in an empty array, which allows keeping just the order of interest while discarding the rest of the image.

```
g = np.poly1d(z)
```

Once all of the pixels are stored for a particular order, the program moves up to the next order and, in a new empty array, stores the pixels for that next order. This is done iteratively until all of the orders specified are stored into the array.

```
t=g
t.c[0]=order_multiply[k]*g[2]
t.c[1]=t.c[1]-order_drift[k]
j = int(t(i))
pan = j + displace+count
flat_flux[k,pan,i]=flat_data[pan,i]
```

Each of the arrays holding each of the orders are processed by a predefined routine of the scipy library. The exact specifications of it can be found on the scipy website, listed in the annex. Roughly, this routine smooths the image, in order to minimize the variation in measurement from pixel to pixel.

```
sig = np.std(surface_val2[i])
surface_val2[i]=
ndimage.gaussian_filter(flat_flux[i]
, sigma = (0.05,75), order = 0)
```

The original data is then divided by the values on the surface, and this creates the

```
flat_field2 =
flat_flux/surface_val2
```

flat field with the coefficients of correction of the sensitivity

Once this is finished, the flat fielded coefficients of correction are stored in a file.

```
saveToFile('flatfield',
flat_field2)
```

Reducing the Spectrum

The file containing the image of the spectrum is accessed, along with the flat lamp image, and the image is loaded into a matrix. The bias, dark and overscan mean values are all subtracted from the image in order to account for the possible interferences in the data.

```
temp_data[k,pan, i]=
spect_data[pan,i]
temp_flat[k,pan,i]=flat_data[pan,i]
```

The fit created at the beginning is used to sweep along the orders and obtain the value for each pixel. Just like in the creation of the flat field, each value is stored into an array containing just the data of the order.

```
flatF_data = temp_data/flatF
flatF_flat = temp_flat/flatF
flatF_flat = flatF_flat.filled(0)
flatF_data = flatF_data.filled(0)
```

This data is then flat fielded with the coefficients of correction generated in the sensitivity calibration stage. To do this, the array holding the information for the order is divided by the flat field, and this corrects the fluctuations in sensitivity from pixel to pixel. Also the masked values are filled with a 0, this allows the array to be used freely without interference from the overscan, and negative values resulting from the subtraction of the overscan, bias and dark.

```
flat_gauss [k,count] =
flat_gauss[k,count]+flatF_flat[k,
pan, i]

mean =
ma.sum(flat_gauss[k]*gauss_step)/ma
.sum(flat_gauss[k])

sig =
ma.sum(flat_gauss[k]*(gauss_step-
mean)**2)/ma.sum(flat_gauss[k])

popt, pcov =
scipy.optimize.curve_fit(gaussian,g
auss_step, flat_gauss[k],
p0=[ma.max(flat_gauss[k]),mean,
sig])

weight = gaussian(gauss_step,
*popt)

sumWeight = np.sum(weight)


def gaussian (x,a,x0,sigma):
return a*np.exp(-(x-
x0)**2/(2*sigma**2))
```

The pixels are then added column-wise for each particular order to obtain the information for a particular wavelength. The addition is used using weights. Each pixel in the column is plotted and a gaussian bell curve is fitted through them. The bell curve is fitted using some the calculated mean, and the standard deviation of the data, along with the peak value which is the maximum of the data set. This allows the program to give more importance to pixels with higher amount of flux than the ones with less flux.

The program then sends the pixel value to a method to find the weight that differs least from it, or in other words, the one that corresponds to that data point. It finds the minimum difference point and returns the index, for that weight to be used.

```
adjusted_Weight =
gaussAdjust(weight,
flatF_data[k,pan,i])
```

```
def gaussAdjust(weights, point):
i = (np.abs(weights-
point)).argmin()
return weights[i]
```

The values for each column are then stored in a 1-dimensional array, which shows the flux at each pixel coordinate for a particular order. This process is then repeated for all of the orders fitted in the first routine.

```
spectrum[k,0, i] =
spectrum[k,0,i]+(flatF_data[k,pan,i
]*adjusted_Weight)/sumWeight
```

The normalization divides the array containing the un-normalized spectrum by the fitted line. The normalization function is obtained by fitting a line through the relative maxima of the spectrum

```
spectrumNormalized[order]=np.copy(sp
ectrumNormalized[order])/normalizati
on_function(x_Dimension)
```

Each one dimensional array containing the normalized reduction of each order is stored in a table in a fits file. Each column of the table contains each of the orders.

```
col  = fits.Column(name = label,
format = 'D' ,array =
spectrum[i,0])
column  = [col]
hdu = fits.PrimaryHDU(spectrum)
label = 'Order_' + str(i)
column.extend([fits.Column(name =
label, format = 'D' ,array =
spectrum[i,0])])
cols = fits.ColDefs(column)
tbhdu =
fits.BinTableHDU.from_columns(cols)
thdulist = fits.HDUList([hdu,
tbhdu])
thdulist.writeto(star_File)
```

Wavelength Calibration

This routine, relates pixel coordinates with wavelength for each order. To be able to do this, a calibration lamp spectrum is needed along with a calibration text file.

The calibration lamp's spectrum can be obtained by using all the previous program routines and reducing the spectrum. Then using the Show Spectrum button, the calibration text file can be created. To do it, the user should display the spectrum and identify the peaks on each order, write down their pixel coordinate along with the wavelength they represent. Since it is a calibration lamp, the wavelength of the peaks, representing the emission lines, are known and can be related to the pixel coordinate. The calibration text file should contain: the number of the order, the pixel coordinate and the wavelength. Starting a new line for each different peak. A sample is shown on the right. Each value should be separated by a comma and the file should be saves as a text file (.txt).

Once this is done for all of the orders of interest. The Wavelength Calibration routine can be started. The spectrum of the calibration lamp, the calibration text file, and the calibration lamp's spectrum (taken the night of the observation) are loaded. The user relates the known wavelength emission lines of the calibration lamp, with those of the same calibration lamp but taken in the night of observation. The wavelength is then related to the pixel coordinates at the night of the observation, and a regression is done. The regression is set to find the best fit line, using chi squared values, that is 2 orders smaller than the amount of data. This avoids having either a trivial perfect fir (i.e. two points are scattered and a

Sample text for calibration text file:

```
6,1007,7067.17
6,1162,7084.19
6,1546,7107.46
7,1180,6871.29
7,1213,6874.74
```

```
upperInd = event.ind
lowerInd = event.ind
crossRelatedPoints =
np.append(crossRelatedPoints,[x_Coo
rd[lowerInd],labels[upperInd]])
pointRelated =
plt.scatter([x_Coord[lowerInd]],
[labels[upperInd]],c = 'b')

x_val = crossRelatedPoints[:,0]
      y_val =
crossRelatedPoints[:,1]
```

1st degree polynomial can be fitted perfectly. It also avoids the trivial fit (i.e. one point is scattered, there is a large number of 1st degree polynomials that can be made to pass through that point).

Also it is coded so that only odd polynomials are fit. This avoids that the same wavelength can be related to 2 different pixel coordinates.

After the regression is done, the user must save it before proceeding onto the next order. The array containing the coefficients of the polynomial is stored in a file named after the order. This procedure should be carried out for each order.

```
z = np.polyfit(x_val,y_val,
(2*i+1))
f = np.poly1d(z)
y_Expected = f(x_val)

chi_Squared,_= chisquare (y_val,
y_Expected, ddof = points-1)


z_Arr.append(z)
f_Arr.append(f)
chi_Arr.append(chi_Squared)
bestFit = np.argmin(chi_Arr)


saveToSecondaryFile(name_File, z)
```

Show Spectrum

The reduced spectrum from the Spectrum Tracer routine is loaded from a fits file and displayed to the user. If the spectrum is that of a star and the wavelength calibration has been done, the x-axis switches to wavelength values for all the orders calibrated.

```
ax11.cla()
ax11.plot (scale,
spectrum_In[order])
ax11.set_title(title, fontsize=16)
ax11.set_xlabel(label, fontsize=14)
ax11.set_ylabel("Flux", fontsize =
14)
ax11.tick_params(axis='both', which
= 'major', labelsize=12)
fig11.tight_layout()
fig11.canvas.draw()
```

Annex

All of the libraries and predefined functions are explained in greater detail in the Scipy, Numpy and other databases. All of the used libraries are listed here, along with a link to their documentation.

Sys: https://docs.python.org/2/library/sys.html

Astropy.io.fits: http://docs.astropy.org/en/stable/io/fits/

Numpy: http://docs.scipy.org/doc/numpy/reference/index.html

Numpy.ma: http://docs.scipy.org/doc/numpy/reference/maskedarray.html

Matplotlib: http://matplotlib.org/contents.html

Tkinter: https://docs.python.org/2/library/tkinter.html

tkFileDialog: http://tkinter.unpythonic.net/wiki/tkFileDialog

Signal: https://docs.python.org/2/library/signal.html

Threading: https://docs.python.org/2/library/threading.html

Multiprocessing: https://docs.python.org/2/library/multiprocessing.html

OS: https://docs.python.org/2/library/os.html

mpl_toolkits.mplot3d: http://matplotlib.org/mpl_toolkits/mplot3d/index.html

Scipy: http://docs.scipy.org/doc/scipy/reference/

Scipy.ndimage.filters.gaussian_filter: http://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.ndimage.filters.gaussian_filter.html