# ASTRONOMICAL DATA PROCESSING
# using DS9, python, and IRAF

*ASTR 230 - 2022 Spring Semester*
DUE DATE: TUESDAY, March 29, 7PM

## 1 Overview and Purpose.

This laboratory is intended to give you a working knowledge of astronomical image processing. Astronomy is at a crossroads, and there are several ways to reduce images. One of the primary packages developed for decades by the National Observatories for imaging and spectral reduction is known as IRAF (*Image Reduction and Analysis Facility*). IRAF is powerful and there is a package to do just about anything you want. However, its internal programming language has never caught on, its ways of plotting data are rather arcane, and its system administration is not necessarily straightforward. Most importantly, funding to support it has dried up, program developers see it as based on an old framework, and its future is unclear. Nonetheless, it is the language that remains prevalent at many observatories around the world. If you are using a professional telescope, there is a good chance that IRAF is there, and if it is an NOAO facility, it probably is how you assess whether or not your data are any good. As an observational astronomer using equipment that operates in the optical or infrare, you are at a significant disadvantage if you have no facility with its workings.

IDL is a powerful analysis package that has enjoyed widespread usage in many areas of astronomy. However, it is not free, not owned by astronomical interests, and its use seems to be slowly waning. The statistics package R is free and is as powerful as IDL, but astronomers are generally unfamiliar with it. MATLAB is similar to R and to IDL, with a more mathematical flavor, and Mathematica is another option along those lines. Neither MATLAB nor Mathematica have widespread use in observational astronomy. Java is another language, but is used primarily in web-based applications, or to code up instructions for machines to run automatically.

This leaves python. Like R and IDL, Python is an interpretive programming language, meaning that it operates line-by-line rather than by compiling code. It was invented by (and for) computer scientists, whereas IRAF was invented for astronomers, IDL for data analysts and R for statisticians. Like C++ and R, python is object-oriented, so that you can define an object that has attributes. It has caught on especially for interfacing with instruments and dealing with data that have a mixture of numbers and text, for example with satellite mission data. Its plotting interface is typically a package that mimics MATLAB, and python is used to install software packages on linux machines, scripts for web pages, and even in video games. As an interpreted language, python is typically a factor of $\sim$ 5-10 slower than compiled languages like C and fortran for computation. There are claims that Julia, a new programming language, is the next frontier. It is hard to predict where all this will go. You can spend a lot of time learning something and then have it become obsolete. So as your teachers we have to decide how to best prepare you for the future.

There are many issues with python, including a lack of backward compatitibility between versions (getting better, perhaps, but don't expect something you figure out now to work in five years), and its dependence upon a series of contributed packages that are not necessarily compatible with

one another and can be poorly managed. System administration can be complicated if, as often happens, multiple versions are needed. Because python is used by your operating system, it is possible to break the entire computer if the wrong versions are put in the wrong places. Python has many different object classes and they often don't behave as you'd like. Error messages are long and cryptic, and can suddenly appear in line 478 of some imported module you did not write when you change python versions because now someone decided to 'deprecate' a way of managing a class (that is, it is *your* fault that their code doesn't work). Personally, I find R to be a much more intuitive and stable interpretive language than python, more likely to work the first time if you try something logical and less likely to break later. Nonetheless, python seems to have risen to favor among many young astronomers, so much so among its adherents that it is sometimes the only programming language they know, and as such they tend to be rather evangelical about it.

In this lab we will learn to reduce astronomical images and spectra by using python in the Jupyter (for *Ju*lia/*py*thon/*R*) environment. From the name, you can tell that Jupyter is meant to work with Julia, Python, and R, and is a good way to visualize programming. Jupyter grew out of IPython, a somewhat buggy initial attempt at such an interface. We will reduce some CCD images using just python so you can see how it works. There is a repository of astronomical python packages called astropy, and we will use some of these. The sophistication and flexibility of the astropy packages varies, and is behind those of IRAF in most areas, though astropy continues to improve as it matures, and IRAF is not being developed actively. Generally I have avoided making you use a package that may still be changing rapidly. Things like file openings and array manipulation ought to stay stable. There is an effort to convert the most important IRAF packages to python, but that is a large task, and is being done piecemeal.

Python can display images, and this can be helpful to see right away what your data reduction steps did. However, python doesn't have the interactive mouse control and quick response offered by DS9 (a.k.a. saoimage). DS9 is a standalone application that can access catalogs around the world, display images and spectra, and do initial data exploration. IRAF usually displays images through DS9, though there is a less-used tool ximtool as well. The lab will take you through how to use DS9, and that is normally how you'll want to investigate what an image looks like. Once we are done with the image reductions, we'll turn to spectra and photometry, and it is a good time to learn a bit of IRAF.

You will need a laptop with python on it to do the lab. If this is an issue, let me know and we'll try to set you up on a linux machine that has it. Expect to have to do some system administration to get everything to work right. That is the way of the world now. You'll probably need to install ds9, and hopefully we can get you a copy of IRAF. These are available on the lab computer if you need them. Some of you will want accounts there and others won't.

*You should learn how to use a non-GUI text editor.* There are several options for this, but two of them stand out - 'vi' (or vim) and 'emacs'. Typically you learn one of these really well. I chose 'vi' because if you ever have to do serious system administration on a unix machine, the only base-level editor that works when the machine is really sick is one that is similar to vi. I don't know much about emacs. Both of them are very powerful, and both take a while to learn to use well. Pick one and get on it. They are probably 5-10 times faster to use than any point-and-click editor you may initially find more comfortable. It's a bit like learning how to drive or how to type - eventually you just think about how you want the text in the file to appear and your fingers make it happen.

With a text editor you will be able to make a python program and run it from a command line. You can also save a text file from your Jupyter session and then edit it later as a text file.

Now if you all wait until the last minute and all try to do a lot of data reduction at once, I don't know how well things will run. The solution of course, is to not wait until the last minute. It is not a good idea to wait until the last minute anyway for obvious reasons. As a computer lab, you need to expect some bugs to occur, both from your lack of experience as well as operating system glitches and setup problems. I will address the latter when you bring them to my attention but this will not happen immediately.

To be explicitly clear – *I am not going to do system administration on your laptop.* Most people are able to get python, Jupyter, DS9, and IRAF to work on their machines without too much difficulty. IRAF is probably the trickiest. The latest version ought to be available on github (https://github.com/iraf-community/iraf/releases/tag/v2.17). You are going to need the x11iraf part as well to get an xgterm, which is an x-terminal that has graphical capabilities. To get ds9 to work on a Mac you'll probably need Xquartz. There is also a 'conda' version of IRAF you can find on-line, but that is becoming increasingly problematic with lack of support, and only works on 32-bit machines.

We may be able to offer some other advice, but if you want the convenience of doing the lab on your own machine it is up to you to make it work. Don't expect anything to work under Windows, science generally operates in a Unix-like environment. Macs have an underlying operating system based on unix, but not everything works right away, necessarily. I run all these things both on a Mac laptop and on unix workstations. We do have IRAF and DS9 on the lab computer, and can give you an account there with a bit of notice, but this won't happen immediately. If you delayed starting your lab until the last weekend before it was due, well....

This can be a long lab, not so much from the standpoint of doing it if you know what to do, but rather that its purpose is to have you think about each step. As such, it is not intended as a sequential series of tasks, but rather one where you develop new skills and use them. You need to ponder your results and make sure they are giving you something sensible. Because of the 'Wild West' nature of current astronomical data reduction, we'll just have to see how it goes. You have several weeks, and the intent is that you work on this lab when you are not observing. I think you will learn the material better if you don't try a blitz at the end. Remember that you will need all these skills for the final project. There are more specialized aspects of data reduction such as psf-fitting for photometry and optimal extraction for spectra we are not covering here, but you may end up using for your project. But this lab should give us all a base to work from.

As a parenthetical note of advice, in addition to either **vi or emacs**, you ought to learn how to program in a standard compiled coding language such as **C++ or fortran** before you graduate. Most of the major codes that run on supercomputers and the like are based on one of these languages. One can also do a lot of list manipulation and scripting directly within the unix environment by using **awk, sed, perl, and bash programming**. Python grew up in the era of perl, which descends in part from awk. You can think of python, R, MATLAB, and IDL as convenient interfaces to a programming environment geared for data analysis. That is, they have built-in functions that save effort over programming in C or fortran. Personally, I am (i) a guru with vi but know no emacs; (ii) very good with IRAF, know some R, am pushing through python on the side, and know hardly any MATLAB or IDL; (iii) am a guru with awk, decent with sed, know only the perl that goes with

awk, haven't done much bash; and (iv) am quite good with fortran, but know very little C++. You ought to have something in each of these categories. All of them if you want to hang your hat on being a computer whiz! I have a computer page at http://sparky.rice.edu/~hartigan/comp.html that has links to awk, sed, R, IDL, IRAF, python, and gnuplot, where I store notes for myself. They might be of use to you. At some point all of these languages want to loop, define variables, and calculate stuff. The syntax is the challenge typically.

The lab consists of a series of goals. You ought to use the Jupyter interface to do your reductions, and create the files needed. We'll figure out a way to share the files on Rice box so I can see what you have done. As the astronomical software landscape changes so does the lab, meaning that there may well be bugs in it. Please bring these to my attention so they can be fixed.

Here we go.

## 2 Image Reduction and Analysis

**A. Jupyter**

Open a terminal and type 'jupyter notebook'. Your internet browser should open and you'll see something like 'localhost:8888/tree' in the title bar. On the right side click on New→Python 3. Note you could have other kernels, say a Javascript one, one that uses R, or even fortran. But you have to download these and install them. Here, since we are just going to use Jupyter for python, you shouldn't have to do anything special because the python kernel comes packaged with Jupyter. Type the following lines, each followed by Shift-Return:

```
import numpy as np
a=np.pi*3480./66.33
print(a)
```

You should see it print out the value of $\pi(3480./66.33) = 164.82349516798553$. You can now save these two commands to a notebook file using the File→SaveAs command. It will create a file.ipynb in the current directory. This is a text file and you can look at it with vi or emacs, or print it out. It has an awkward javascript format that is rather difficult to read though. You can close Jupyter, and open the notebook file next time with 'jupyter notebook file.ipynb'.

Jupyter (like vi) has two modes, a command mode and an insert mode. You can get to insert mode by just clicking on the line, though there are other ways. To get to command mode you hit the escape key. I see the bar on the left of the line to be blue in command mode, and green when using input mode. So, 'escape' then 'shift-enter' executes the line and goes to the next line. 'shift-tab' executes the line but does not move forward. If you see an '*' it means it is working on your latest command. If it hangs, you'll need to go to 'kernel' and 'restart kernel'. Don't be afraid to restart the kernel, you don't lose all the commands - it just throws out any arrays and what-not you've created. But the commands will recreate those quickly enough.

To save the work as a text file of python commands, File→Download as→python will create file.py in the Downloads directory. Equivalently, from the command line you can type 'jupyter nbconvert

–to python nbname', which converts file.ipynb to file.py. This conversion adds some comment lines '#' and blank lines that are not necessary. Delete them in vi or emacs if you like. The file.py is a python code file you can run via 'python nbname.py'.

Try saving and running your python program in this way.

Customization: Jupyter has a bunch of menu-based items for inserting, deleting, and so on. These work, but after a while you'll find them to be awkward and slow. You can define your own macros to, say, open a new line below the current one with a single keystroke. If you have no preference at this point, *you may as well use the ones I use.* They are based on the vi editor, so you don't have to learn things twice. The shortcut file is named notebook.json, and is located at the bottom of the web page sparky.rice.edu/∼hartigan/py.html if you want to give it a try. Store it in ∼.jupyter/nbconfig/notebook.json on your computer. Once notebook.json's shortcuts are defined, 'j' moves you down, 'k' up, 'O' adds a line below, 'o' a line above, 'dd' deletes a line, 'a' appends, 'i' inserts, 'yy' yanks a line to a buffer, and 'p' puts that line where your cursor is located, 'J' moves a line down, 'K' moves the line up, 'Cmd-J' and 'Cmd-K' merge cells, and 'Ctl-w' in command mode splits cells at the cursor.

**B. Basic Imaging Display and Visualization with DS9**

For starters, look at the contents of 'ASTR230 FILES'. You'll begin by displaying the HH 111 images, which are reduced HST images of a stellar jet in the emission lines of H-alpha and [S II]. Copy the images to your computer, and install ds9. My version is 8.2. There are probably newer versions.

Launch it via 'ds9 &'. File→Open to open the image. Right click-drag to change contrast. Experiment with the push-button options of ds9 and learn how to pan, zoom, change the greyscale mapping, change the colormap, blink, etc. Many of these tasks are also controlled by the mouse, which is nice and fast compared to using the menu. If you left-click on the image you will get a circular marker, which can be resized (or deleted) by selecting it. I don't like the markers much, so I usually start by going to Edit-Pan, which then makes the left button click pan around the image instead of drawing a marker. Zooms (left button; shift-left-button to de-zoom; first click 'Edit' then 'zoom' to set this up, or use the panner display at top), pans (middle button) and greyscale adjusting (click-drag the right mouse button) can all be done this way.

One of the skills you need to learn is how to display images well. Look at the scale bar. There is a lower value z1 and a higher value z2 that set the limits on the greyscales. As you click-drag you can change the greyscales between these limits, but to get any contrast at higher values you must raise z2, and at lower values you must lower z1. However, if z1 and z2 are too far apart, you will not have enough grey scales in-between to see details at intermediate values. You can change greyscales under Scale→User and Scale→ Scale Parameters. Note you are *not* changing the image with these operations! You are simply changing how the image is displayed on the screen.

For images of nebulae, try using the logarithmic option to see if you can better display faint emission in the images. Finally, store hardcopies by File→Export.

*NOTE: When writing up the lab I will be looking for a single, self-contained pdf file. If you need to show an image, embed the picture in that file. If you have results from a jupyter notebook, put the results into the pdf. You should reference where your .fits files are stored in case I want to look at them, but your plots, images, tables, and other results need to be clear enough in the pdf alone to*

*answer the questions.*

**Q1 [scaling]:** *For the images hh111.ha and hh111.sii (HST images of a collimated jet from a young star)* **(a)** *What are the best z1 & z2 values you can find in a linear stretch display which show detail within the brightest parts of the jet?* **(b)** *What are the best z1 & z2 values for a linear stretch that show the three fainter bow shocks to the left?* **(c)** *Does a logarithmic stretch help to display the image? For each case, save your appropriately-zoomed picture as a greyscale JPEG and include it in the writeup. The goal is to see the structure in the jet knots well. That means you should be able to see individual pixels in the zoom.* **(d)** *Explain in your own words exactly how the computer uses the values of z1 and z2 to map pixel values into greyscales. Explain why the z1 and z2 values for part (a) did not work for part (b), and vice-versa.*

Another way to make images for publication is to scale so that one can see the background and faint emission well, and leave most of the actual object looking saturated as white. Then we draw contours on top of the white part. Let's try this.

Zoom in a factor of 16 onto the brightest knot in [S II], and display it to show detail in the background and faint levels, with the object all white. Click Analysis→Contours. Under 'Contour Parameters', pick levels that begin at some value and increase by factors of x, where x is some value you pick. Experiment with various contour smoothing, colors, and thickness until you get something that looks like it is doing a good job. Click-drag so you can see the white part and verify that the contours are in the right place. As you move the cursor the information panel will show you the pixel value.

**Q2 [contours]:** *Store your favorite contour representation.*

Notice ds9 has several other analysis features, including smoothing, statistics, and so on. It has ways to bring in data from IR, Xray, and other catalogs as well.

### C. Image Reduction with Python and Jupyter

### I. Image Loading and Headers

The main data format of astronomical images is a FITS file. FITS files consist of a header file and a data file. Often there are multiple extensions to fits files, each with their own header and data sections. To begin,

*from astropy.io import fits*

loads the 'fits' module of astropy.

*import numpy as np*

loads the 'numpy' package of python and nicknames it 'np'. So if you want the value of $\pi$ that is done by np.pi, for example. To get the sine of 30 degrees you'd write np.sin(30*np.pi/180.), because the sine function takes arguments in radians.

*ffile = fits.open("input_file.fits")*

creates an object 'ffile' that is a list of each header-data-unit (HDU) in the fits file. To see how many it found, type

*ffile.info().*

In our case there should only be a [0]. This data object is by default returned as a 32-bit floating

point array. You should be able to see the dimensions of the fits file.

*fdata = np.array(ffile[0].data)*

creates an array of the data in the fits file. Fits images behave the way you'd expect: when NAXIS1=500 and NAXIS2=20, then the image has 500 pixels in x and 20 pixels in y. Displaying it in ds9 will show a skinny horizontal image. Unfortunately, python flips the axes in its array, so what was a row is now a column and vice-versa. You have to keep track of this. When it writes a fits image back out it flips the image back again. Also, the upper left pixel the in the fits file is [1,1], but python decided to index everything to 0, so that pixel is now [0,0]. Combining this nonsense together, we see that pixel [2,12] in the fits image is now pixel [11,1] in the python data array. To see the dimensions of the array, type

*np.shape(fdata)*

Note that np.shape(data) is a 'tuple', which is an array whose values cannot be changed. There is a np.shape(data)[0] and a np.shape(data)[1] you can look at which represent the y-axis length, and the x-axis length of the image, respectively. You can use these values as integers in loops when you program.

*fheader=ffile[0].header*

is the header, a series of keywords like NAXIS1 and OBJECT and their values, which can be strings or numbers. The '0' is for header unit 0, which is all we have here. Often though, with HST data there will be one unit for data, one for uncertainties, one for data quality flags, and so on.

*fheader*

ought to print out the entire fits header. You can see it has a series of keywords and their values. For example,

*fheader['OBJECT']*

will print out the object name recorded when the data were taken. You could change it via

*fheader['OBJECT'] = "newname"*

and then write the array back into the fits file via

*fits.writeto("filename", fdata, fheader, overwrite=True)*

It is relatively easy to add a new keyword and its description to the header file with a command like

*fheader.set('OBJECT2', "newname","This is a new keyword")*

A similar command would insert this new keyword in a specific place, say, after the keyword 'OBJECT':

*fheader.insert('OBJECT',('OBJECT2', "newname","This is a new keyword"))*

To delete the new keyword, you'd type

*fheader.remove('OBJECT2')*

**Q3 [header]:** *The value of the RA in the HST data for m101.ha.1.fits is given in degrees. Use python to grab this value from the header, and convert it to hrs:min:sec format and print it out. Note the np.int command does a 'greatest integer less than or equal to'. The pixel size is 0.1*

*arcseconds. How many digits should you bother to keep in the RA seconds? Submit your python code snippet as part of the lab writeup.*

**II. Image Statistics and Plotting**

We'll need to do some plotting, so

*import matplotlib.pyplot as plt*

to load the MATLAB-like plotting routines. You can use these to plot slices of the data, and to display the image. I find the image display to be rather crude compared with ds9, but it does work. You can try it here with

```
plt.figsize=(12,12)
plt.imshow(fdata,cmap='gray',vmin=300,vmax=10000,aspect='equal')
```

The vmin and vmax are the z1 and z2 values to specify and the cmap is the color map. You need the figsize or by default the plot comes out really small, and the aspect is needed or it will distort the image. You could specify a section, say [1000:2000,500:600] for those ranges in y and x, or [:,500:600] for all of y and a 101 pixel limit in x. I can never seem to get the axis labels to behave when plotting an image section. Remember the array axis flipping. The question is what to choose as the values for vmin and vmax (IRAF does all of this automatically by the way). We can do statistics on sections of the image to get some idea as to how to set things. The commands np.mean, np.std, and np.median compute the mean, standard deviation, and median, respectively, of some image section. For example, you can try

*np.mean(fdata[1000:1500,:])*

to find the mean in the section of all x, and y from pixel 1001 to 1500. Notice pixel 1001 is actually number 1000 in the python array. Also, the upper python range limit 1500 is actually not included, so the top python array value is actually 1499, which is pixel 1500 in the fits file. Setting vmin and vmax to be $\pm$ 1 sigma from the mean is a reasonable first attempt. Or you can just skip this and always display in ds9. Histograms can be useful, but only use them on 1-D arrays or it will try to group each line in a different color, and probably hang and you'll have to restart the kernel. Try

*plt.hist(fdata[1000,500:600])*

You can do 2-D sections with histogram but you have to flatten them first. You can specify the histogram range and number of bins. For example,

*plt.hist(fdata.flatten(),50,range=[3750,4000]*

should give you a histogram plot from 3750 to 3999, with 50 bins.

**III. Bias Subtraction, Trimming, and Flatfielding**

It is time to start reducing data. Raw CCD data have a bias strip on the side. Its purpose is to measure a zero point that can be subtracted from each row. Download the raw data, leo*.fits, and read one of these into python as you did above. These are images taken with the McDonald 30" telescope, a facility you may be able to use for your projects if it is open for observing. First you have to figure out where the bias strip is. That's easy to do. By displaying in ds9 you'll see it is over at the highest 40 pixels or so in x. Plot out a strip with python:

*plt.plot(fdata[1000,:])*

**Q4 [image exploration]:** *Looks like there is a spike at x=1, something at x=400 and x=1400 or so, and something going on at the high end. Look carefully at the image in ds9. What are these things? Try plotting an adjacent line, and one offset by 100 pixels. Do the features go away?*

It looks like a good bias section is x = 2050 through 2079 inclusive. We should also trim the image and only keep the range x = 5 through 2045 and y = 3 through 2047, where I decided upon these numbers by displaying the data. To keep things straight, and in case we'd like to change them later, in Jupyter let's define the following limits for trimming and for bias extraction, in the coordinate system of the fits file:

```
x1=5 #first good pixel in x after trim
x2=2045 #last pixel to include in x after trim
y1=3 #first good pixel in y after trim
y2=2047 #last pixel to include in y after trim
bx1=2050 #first pixel of bias section
bx2=2079 #last pixel of bias section
```

Here, the # indicate a comment (no need to type that in). Now we can copy the section we want to keep into ndata.

```
ndata=fdata[y1-1:y2,x1-1:x2] #grabs the good section, but still need to subtract bias
```

Now we want to go row by row and subtract off the bias. We'll pick median, though you can do a mean. The following command should do the loop correctly, given the offsets and array flips that python uses:

```
for y in range(y1-1,y2):
ndata[y+1-y1,:]=ndata[y+1-y1,:] - np.median(fdata[y,bx1-1:bx2])
```

The indentation is how python determines what block of code to do in the loop. Notice that the range is for the original data file fdata, whereas ndata starts at 0 and simply ends up being the size it was before, with the median over the bias range subtracted from the values. Be careful when running this in Jupyter. Because ndata is defined in terms of ndata, you don't want to run this line multiple times. If you think it is getting confused, have a look at the values for fdata, ndata, and the medians it finds. You can always kill the kernel and walk through the commands to this point to correct any problems that way.

Now that the data have been corrected for bias-overscan and have been trimmed, sometimes you subtract a separate bias image to account for any pattern in the bias. This CCD doesn't need that correction, so we skip that step. The last thing to do now is to flatfield the data. I already reduced the flats for you by trimming and correcting for bias the same way as above, coadding all the individual flats for each filter together, and then normalizing by some number to make the average of the flat $\sim 1$. These flat images are named 'rflat.fits' for the r filter, and similarly for the other filters. So we are ready to just bring in that image and divide our data by it.

```
fltr=fits.open("rflat.fits")
rfdata=fltr[0].data
for y in range(y1-1,y2):
```

*ndata[y+1-y1,:]=ndata[y+1-y1,:] / rfdata[y+1-y1,:]*

Finally, we should update the header to store the trim section and bias section we used, and then write the data to a new file name.

*nheader=fheader* to copy the original header.
*nheader['BIASSEC'] = "[bx1:bx2,y1:y2]"*

where bx1, bx2, y1, and y2 were defined as above.

*nheader['TRIMSEC'] = "[x1:x2,y1:y2]"*

where x1, x2, y1, and y2 defined the trim section
*fits.writeto("leor1_ff.fits", tdata, nheader, overwrite=True)*

Here, the _ff is to remind you that this file is flatfielded and trimmed.

**Q5 [flatfield statistics]:** *Find three sections of ndata (the unflatfielded but trimmed image) at least 100x100 in size that appear to have only sky (no stars). Calculate the standard deviation of the pixel values in those areas using np.std. Now do the same for the flatfielded version rfdata. Did flatfielding help smooth out the background? Print out the commands you used and the results you got for the standard deviations.*

**IV. Image Shifting and Coadding**

Construct flatfielded leor1_ff, leor2_ff, leor3_ff, and leor4_ff images. Display these in ds9, each in a different frame but with the same z1 and z2 values. Press TAB to cycle through the images. You'll notice that the galaxies move. This movement is caused by the telescope not tracking perfectly, and by deliberate shifts to cover more area. The goal is to combine all the images, using an average of some sort where they overlap.

We have to do several things to get this right. First, obviously we need to shift the images to line up with each other. After that, we could simply add them, but then we have the problem that the counts are higher when more images have data at that point. So we really need to subtract a background value for the skyglow. But that varies between exposures. There are some bad columns and rows, and it would be good to not add those in to the final data or they will proliferate and mess up the image. There may be some places where there are no data. If these are off in a corner we may just want to put zero there so the image doesn't look jagged when we combine all the individual ones together. But if there is a narrow bad column with no data maybe we can interpolate across the column. Finally, one image may have more counts for all the stars than another image, so it may be important to multiply the sky-subtracted image by some normalization constant before adding it to the rest of the images.

There are many ways to try to accomplish all these tasks. It would take you quite a while to program all this even if you know some python, so I've made something you can try. The program is called imshalign.py. Have a look at it in a text editor. Let's think about what we need. First, we have to have the images flatfielded and bias-subtracted, but you have those now. Then we need to figure out (i) pixel shifts, (ii) sky values, (iii) bad pixel masks, and (iv) normalization scales for each frame. So let's get busy.

The sky value can be a median over some section that has no stars in it, and the pixel shift you

can get to within a pixel or so by displaying in ds9, moving your cursor over a star that is common between two images and reading off its position. There are fancier ways to do the shift, some that involve fractional pixel shifts (like we'll do with IRAF photometry below). But it is probably not worth the effort here, given that the stellar images all spread out over several pixels.

Display each r-band image in a different frame in ds9. Choose Edit→Pan and click on a star that is common to all four images. Blink by using TAB. Tweak the alignment by using the arrow keys. When they are aligned, you should be able to hold down the TAB key and see the stars not move as the images blink. Put the cursor somewhere in the image and note the (x,y) coordinates as you hit TAB. The changes give you the shifts (don't click, which will recenter them). Now find a spot in the image that is mostly sky, choose Edit→Region. Click to add a region, and change its size to a good spot with sky. Read off the median value, making sure it looks like what you'd expect. Record your shifts and sky values, and create the following text file 'flist' that should look something like:

```
name          xsh   ysh   skyval bpfile scale
leor1_ff.fits -36   53    966.2  bpmask  1.0
leor2_ff.fits  0    -46   128.4  bpmask  0.8
leor3_ff.fits -10   -49   515.7  bpmask  1.2
leor4_ff.fits  43   -94  1359.7  bpmask  1.1
```

where you put in your measurements for xsh, ysh, and skyval. For the bpfile, it is probably ok to use a single file, bpmask, for each image, as the bad sections and columns are the same ones between images. There are cosmic rays that scatter across each frame, but these are removed in the median combinations. The format of the bpfile is as follows:

```
x1    x2  y1    y2
480   480 142  255
830  1831  164 164
200  205   450 455
```

Each line has an inclusive range of where the pixels are bad. For example, the top line indicates that column 480 (note the fixed x-value) is bad between rows 142 and 255, inclusive. The second entry marks row 164 as bad between columns 830 and 1831, inclusive, and the last entry is a 6x6 pixel box. Pan around the image in ds9 with a high enough zoom to see individual pixels. Decide where you'd like to mask out bad pixels. Sometimes it helps to choose Frame→Match→Image to line them up, and then use TAB to cycle through them.

Find a star in-common between the four frames. With Edit→Pan, click on the star to center it in each image. Then Region→Shape to pick a circle, and Edit→Region, and click on the image to make a region. A green circle should appear. Highlight the region (four dots appear at the corners), and Region→Get Information to open up a dialog box. Choose a Radius of 10 and hit Apply. Move the region across your star and record either the mean, median, or sum. Then do the same for the sky. Subtract the two to get some idea as to the counts in each frame for that star. In this way determine a normalization factor that will bring the counts into approximate agreement between frames. You should get the same answer for different stars.

Note that to pan you need Edit→Pan, and for regions you need Edit→Region. If you intended to pan but were doing region and you clicked, you'll get another region you don't want. In that case just highlight it and hit Delete.

Okay, time to try the code. Be sure the formats of the 'flist' and 'bpmask' files are correct, including their first header lines. You execute the code by typing

*python imshalign.py*

at the command line in a window outside of Jupyter. It should work, but let me know if there are bugs in it. It's python, so it is rather slow. Name the output file 'leor_med.fits'. Display this image in ds9 to be sure it looks good. Line it up with the individual components (leor1_ff.fits, leor2_ff.fits, etc.). The signal-to-noise should have improved some, the cosmic rays should be gone, and the bad-pixel columns largely gone. Stellar images should be as sharp or sharper than the originals. If you see doubled stars, or suddenly have 6 galaxies instead of 3, your alignment is off. If you see big breaks in the sky where one image ends, then the skyvalues are wrong. If the bad columns haven't gone away, then your bpmask needs to be fixed. It can be educational to deliberately break some of these, and see how the output changes.

**Q6 [Shifting and Combining]:** *Show your final best setup file 'flist' in the writeup. Store the final bad pixel mask 'bpmask' and your resulting image 'leor_med.fits'. Capture a nice representation of the image for display in your writeup.*

One of the images was quite a bit further north than the others. For this one there will be no good data where there are bad columns because there are no other images there to pick from. In this case we would like to interpolate across the columns. There are a couple of ways to do this but the easiest that works here because there aren't a lot of them to deal with is to identify the columns to fix, and simply overwrite them in the array. Once that is done, there are going to be several corners where there are no data at all. The image looks a bit odd with its missing corners, so let's put zeros in there. I'd recommend the np.nan_to_num task. Do the final fiddling in Jupyter and record the final image as 'leor_final.fits'

**Q7 [Final Image]:** *Include the lines of python code you used to interpolate across the columns and fill in the blank areas. It need not be long - mine took 7 lines, including opening the file, fixing it, and storing it. Also include your final image 'leor_final.fits'.*

**Q8 [Asteroid!]:** *One problem with medians is that they don't record moving objects like asteroids. There are more than one of these in these images! Align the r-images, blink them, and see if you can find one. Make a 2-panel figure that displays the starfield around the asteroid, and has a circle that identifies the asteroid in two frames, showing motion.*

**Q9 [imshalign.py]:** *There's a fair bit of stuff going on in imshalign.py. The code makes use of 'nan', which mean 'not a number'. These are essentially flags to indicate no data, and are used as masks. You'll need to look carefully at the code to see how they are being implemented. Annotate it with your own comments (add '#' after lines), and write a paragraph that describes exactly what the code is doing. Submit both your annotated code and your paragraph.*

### V. Multi-Color Image Construction

As a result of your efforts in part III., you now have reduced images of the field in r. Do the same now for v and b so you can make a color composite. Display these images and choose File→Save

Image→JPEG and choose the highest quality. The goal is to insert one of them into an R-channel, one in a V-channel, and one in a B-channel. I don't care how you accomplish this task. I used to use Photoshop, which is available on Rice computers at MUDD. But since they've gone to a rip-off subscription model, I'm transitioning to GIMP, which is free and appears to be a lot like Photoshop used to be. There may be other options.

DS9 can do this by Frame→New Frame RGB, and then open the RGB tab to choose either R, G, or B. Load and scale the image as usual. For alignment you definitely don't want WCS, as the coordinates in the headers here are not that precise. Try aligning with Image instead. If the R, G, and B stars don't line up you may have to crop them in python to make them the same size and to align them. Photoshop is made to do things like this very well.

Depending on how you scale things, your composites are going to look different. Ideally all your stars are white-ish. You can make the image look more or less red, blue, etc., by changing the contrasts. Note that these are 'true color' in the sense that the colors actually represent the spectrum in some way. There are also 'false-color' images. To get one of these, just load an image into ds9, and choose something from the Color menu. Now your pixel values are mapped to a specific color, so each intensity is a color (like a contour map in a way). But this differs from a color composite, where the brightness related to the combined fluxes in the three bands, but the color depends on the ratios of the fluxes in the bands.

**Q10 [Multicolor]:** *Store your best attempts at a multicolor composite. You can try different scales, zooms etc. You will likely have to pick different z1 and z2 values before you store the image from ds9 to highlight bright areas, say, of the galaxies.*

# 3 C. IRAF Reductions

IRAF is now distributed via a community resource group on github (https://github.com/iraf-community/iraf/releases/tag/v2.17). This will be the most convenient for you, though it may take a bit of effort to get the installation to work. You will also need x11iraf to get the xgterm, the terminal that displays graphics. Alternatively, I have it on the lab computer, and we can arrange to get you an account there. For most tasks you will need to physically be at the computer, so you will all need to share.

### A. Setup and Operation

Generic IRAF runs under an environment called 'cl' that is similar to unix. Depending on the installation, sometimes 'ecl' offers a bit more functionality with regard to editing history commands and so on. Once you get used to running IRAF you will want to string commands together and execute them as a large batch. You'll do this by creating a text file of IRAF commands, e.g., with vi.

The computer pongo.rice.edu has IRAF, DS9, and xgterm installed on it. Once you have an account, you ought to be able to log in and run everything. If you are seated at pongo, there may be icons you can click to bring these windows up. You can also invoke ds9, xgterm, etc. from the command line of another xterm (or xgterm) by typing what you want followed by a space and an ampersand. The ampersand if needed so you can keep using the window from which you type the command (e.g. *ds9 &*). IRAF is run from the xgterm. You can run it from a pure xterm, but if

you use a task with graphics it will dump gibberish to the screen when you try to plot something. *BE SURE TO LOG OFF WHEN YOU ARE DONE.* We don't want any locked screens so your fellow students cannot do the lab.

I'm not exactly sure how IT will set up your accounts. Once you log in, you ought to have at least one xterm window. Type "cd" and then "mkdir iraf" to make an iraf directory for yourself. The version of IRAF on pongo is the Space Telescope version, and requires python 2.7, and therefore a python environment of 2.7. A command called "iraf" activates this environment and opens an xgterm for you. If this is not set up, type "conda activate iraf27; xgterm -sb -bg white -geom 105x58+30+12". The prompt should change, showing (iraf27). Now in the xgterm window type "cd iraf; cl". When you get a "cl>" prompt it means you are in IRAF. To get out of iraf, type "logout". For IRAF to work properly you need to start the cl in a directory where the login.cl file is located. This file is made from the unix prompt using the command 'mkiraf' in your iraf directory and needs to be done only once. IRAF will warn you if you do not have a login.cl file, and will work from other directories, but some tasks may encounter bugs if you operate this way. So start the cl from the right place, i.e., your iraf directory, and then change directories *within* IRAF. As you gain experience with IRAF you may review your login.cl file and customize it to fit your needs (e.g., to choose your favorite editor, like vi or emacs).

Once you know how to get into IRAF, go to the directory "/home/hartigan/astr230," where several images exist. Later, you will work in your home directory or in a subdirectory therein. For starters, look at the README file in /home/hartigan/astr230, which describes images stored in the directory. To read an ascii file in the IRAF cl window, type "page README," or use the UNIX 'more' command by typing "!more README" (in the cl window, typing an exclamation mark then a UNIX command allows you to execute UNIX commands within IRAF). You can also do all of this from outside of IRAF in an xterm or xgterm by typing 'more /home/hartigan/astr230/README'.

These images have differing sizes. You will need to make ds9 at least as large as your biggest image. To see how large ds9 is at present, type 'show stdimage' in IRAF. You may get something like 'imt512'. To see what this means, type 'gdevices', and note that the NX NY columns give the size of the display in pixels. To change to a different size, say imt2048, type 'set stdimage = imt2048'. Check with 'show stdimage' that your display has changed. Generally you want to use a display just larger than your image size. You can always find the sizes of the images by typing 'imhead imagename'. Try this "imhead m42b".

DS9 usually works pretty well with IRAF but if you want RA and DEC to display correctly in DS9 or do statistics with pixel values with DS9 you are better off loading images directly into DS9. IRAF has an entire task devoted to statistics in images (imstat). There are help pages for all the tasks.

Look at several of the images with the IRAF "display" command, using both the autoscaling and manual scaling modes in the parameter file options (i.e., change things in the display parameter file as shown to you in the class tutorial using epar). Before using "display", you may wish to use "imstat" or "imhist" to see how to set z1 and z2 in your display. If you get confused while editing the display parameter file, type "unlearn display," to reset the default values.

**B. Photometry**

It is time to do some photometry in order to estimate the magnitude of a star. Type 'digiphot'

and then 'apphot' to get into the aperture photometry package. Visualize a CCD image as a 3D surface, where the x,y plane is pixel coordinate and the z-direction is number of counts. A stellar image appears like a 'mountain'. Our goal is to measure the volume of the mountain, i.e., the total counts from the star. To do this we must also estimate the height of the 'plain' that the mountain sits upon.

The aperture photometry package works by adding up all the counts in a circular aperture centered on the star (the mountain), and subtracting off the 'average' height of the background determined from the pixel values in an annulus around the star (the 'plain'). To see how the background/sky fitting is done, type 'fitskypars'. There are a variety of algorithms that are possible here. Any of centroid, gauss, median, or mode should work. The reason we don't just take an average of the counts in the annulus is that there may be other stars there, and we don't want these to affect the photometry of our target object. Note the inner 'annulus' parameter and the width, or 'dannulus' parameter.

Now type 'photpars'. The 'aperture' value allows you to set any number of apertures to add up the counts on your target. The bigger the aperture, the brighter the magnitude. The 'zmag' parameter is simply a zero point; a zmag of 25 means the program interprets 1 count as 25th magnitude. This number depends on the exposure time, telescope aperture, sky clarity, etc. For this exercise you need only change 'aperture'. Finally, the program will try to center up on the star of your choice before it does the photometry. Parameters for this are located in 'centerpars'.

Display your leor_final.fits image and type 'radprof leor_final 15 1'. The '15' is the radius of the profile to display, and the '1' is the pixels per step. There may be some warnings about missing keywords you can ignore. You should see a circular cursor on the ds9 display. Move the cursor over the moderately bright star we'll call 'star 1' located about 275 pixels above the right end of the hamburger-shaped galaxy (at about x=1408, y=524 in my composite) and press the space bar.

If it works you ought to get a nice graph that shows the radial profile of the star's brightness. If you get a bunch of junk printed to your window and no plot, then you have probably tried to run this from an xterm and not an xgterm. You'll need to exit IRAF with 'logout', type 'xgterm &', and move into the iraf directory in the new window 'cd ∼iraf', and type 'cl'. Then go find where your image is and display it as before and retry the photometry commands.

Place the cursor in the ds9 window and type 'q' to get back to your text window. By the way, the output line on your text window gives you info such as the location of your star in fractional pixels and the FWHM of its point-spread function (fwhmpsf) in pixels.

Now you are ready to try some photometry. Type 'phot leor_final', and place the cursor over any star of interest. Press the space bar. The results for the center and magnitudes in each aperture appear on the screen and are also stored in a '.mag' file for later use. You can interactively change the apertures, skyalgorithms, etc. Type '?' at the circle prompt to see your various options.

Do a similar estimate for star 2, which is considerably fainter, located about 30 pixels to the upper right of star 1. Check to be sure the bright star is not messing up the photometry of the fainter one. Notice that these magnitudes are instrumental magnitudes, meaning they have an arbitrary zero point, in this case defined so 1-count = 25th magnitude. This is fine for relative measurements like we are doing here, but we'd need to identify some standard star in the field to measure a real magnitude.

**Q11:** *Describe how the sky-fitting algorithms you chose work, focusing on how the algorithm manages to ignore any stars that may fall within the annulus. Record your choices of apertures and sky-fitting annuli. Evaluate which choices you think work well and which did not. What is your best guess as to the difference in magnitude between stars 1 and 2? Estimate, and justify, an uncertainty for your measurements.*

Each time you do photometry in IRAF your results were stored in a '.mag' file. In it you will see everything from the position in fractional pixels, to the instrumental magnitude in each aperture, and the sky values it found. If you choose to do a photometry project you may end up wanting photometry for entire clusters. No problem. You could click on a whole bunch of stars, do the photometry, and then extract the results from the .mag file via

pdump leor_final.mag.1 xcenter,ycenter,mag,merr yes | cat > leo_r_phot.txt

This will create the file leo_r_phot.txt which you can edit in a unix window with a text editor, and then read into python or R to plot.

There are two upgrades to the above procedure. First, you might want an automatic way to find stars, that way you don't have to click a thousand times. Second, it will be more accurate if you could extract the psf from bright stars in the field, map out how that psf changes across the image, and then fit that psf shape to each center where you think there is a star. You might also attempt to deblend stars with these profiles if their photometry apertures overlap. The package for all of this is *daophot*. It is a bit much at this point, but expect to use it for the best photometry. Daophot lives within IRAF, but is also a package that works outside of IRAF.

### C. Spectra

ASTR 230 is in the process of commissioning a new spectrograph that you will (with luck) be using for both your short project and possibly for your final project. For now, we will be practicing on data from the old spectrograph. The new one ought to have the same steps needed for reduction, though we'll have to see once it is working.

Now that you are an expert at image manipulation and plotting, it is time to tackle spectroscopy. In the remainder of this lab you will learn how to reduce and analyze spectral data. The final goal is to obtain a 1D spectrum in counts vs. wavelength, for the targets of interest. We split the tasks up into five segments:

Step 1: Extract a 1D spectrum of an object from a 2D file.

In the directory /home/hartigan/astr230/spec you will find a spectrum of the binary star $\sigma$ Ori named 'sigmaori.fit' taken with the spectrograph at George. This is a FITS file. The spectrum is oriented so the wavelength axis goes along the x-axis, and the spatial location along the slit is the y-axis. You may want to copy it to your own directory.

Display this image and determine an aperture (i.e., a range in y) that includes most of the light for the primary. The center of that range we will call ycen, and the width dy. Type 'twod' and 'long' to get into the longslit package, and then type 'help' to see the various options. We want to deal with one-dimensional spectra, so we want to use 'scopy', which will give us a total of the counts in the aperture at each wavelength. To do this, we first need to set the aperture size. Type 'lpar longslit' to see the value of nsum, and change this to equal dy. Then 'lpar scopy' to review

the parameters there. We will want the aperture to equal ycen. Run scopy and extract a spectrum into the output file. Note that the dispersion axis (the wavelength direction) runs along lines.

Display the spectrum in the 'onedspec' package by typing 'splot output'. You can do a lot with splot. For now, try typing 'w' and then '?' to see how to control the display. Typing 'q' twice exits help and returns you to the splot window. 'w a' redisplays, and 'w' followed by two 'e's expands the graph. If you simply type '?' in the splot window you will get a summary of all the cursor commands. There are also a menu of : commands under the '?' menu; for example, :hist will plot the spectrum as a histogram instead of connecting dots. Typing 'q' twice returns you to the splot window, and one more 'q' puts you back into the iraf window.

Once you are satisfied with your extraction of the primary, do the same for the secondary. Examine the headers using 'imhead file lo+' to see that the extraction info was stored correctly. You can rename the extractions with imrename, as you did with images. You'll want to subtract out any background sky or scattered light, so also extract a spectrum of the background and subtract this from the primary and secondary.

**Q12:** *Store your sky-subtracted 1D spectra of the primary and secondary on disk, and plot them in your writeup. Report what aperture sizes you used to extract the spectra. Call these images 'pri1' and 'sec1', respectively.*

Step 2: Construct a comparison lamp image from separate Hg and Ne lamps.

Display the Hg image Hg.fit. There are four very bright lines in the lower right, and a lot of faint stuff to the left. There is also a star that snuck into the slit (that horizontal streak around y=54)! The star won't bother us much, but we have to get rid of the faint lines with x-values less than about 120. These are not Hg lines. After looking carefully through a line list it appears that they are lines of argon. They are not as bright as the Neon lines, that cover roughly the same wavelength region, so we will ignore them for now. If you have a project that requires precise wavelengths they might come in handy, so there is a file nearhg.dat that has all three if you need it.

We will need to add the right half of the Hg image to the Ne image. Do this by replacing the left half of the Hg image with a constant using 'imreplace'. You'll want the constant to be about equal to the rest of the background in the Hg image or there will be a 'shelf' where you cut the image off. Remember that 'imstat' allows you to determine mean values within an image section – perhaps useful for choosing your constant.

**Q13:** *Store your final wavelength image of* Hg+Ne*, with the red lines from the Hg image removed, on disk. Call this image 'wave'. Include a picture of it in your writeup.*

Step 3: Extract the 1D lamp spectrum

From this Hg+Ne spectrum we now need to fit a wavelength solution. To first order you might expect the wavelength per pixel to be constant across the chip, but we can do much better by fitting polynomials. In the task 'identify' change coordlist to "linelists$nehg.dat". Then type 'page linelists$nehg.dat'. This is a list of all the bright lines that your prof identified in the lamp spectra we have. You've undoubtedly noticed that the lines are tilted – the wavelength solution will be different at the top than it is at the bottom. So you'll need to extract a 1D spectrum taken over the same rows as your two stars were. Do so as in step 1.

**Q14:** *Store the two lamp spectra as wavepri and wavesec, respectively, and plot these spectra in*

*your report.*

Step 4: Fit a wavelength solution to the 1D lamp spectrum.

This is a bit tricky, but is also fun once you get the hang of it. Start by splotting one of the extractions from step 3. The Hg line at the highest pixel, about x=225 or so, is the 4046.56 line. The next one down, around x=210 is 4358.328, and the next bright one is 5460.7348, around x=155. Then the Ne lines start to come in. The brightest of these, at pixel 105 or so, is the 6402.246 transition. Some of these lines are blended in this low-resolution spectrum, and will be resolved if you take high-res data or use a narrower slit. So not all of the lines listed in nehg.dat are going to be good for calibration with this data set. Your task now is to figure out which ones are good and which are not.

In onedspec, type 'lpar identify' and look through the parameters. There is a fair bit of stuff there, and 'help identify' will explain it all. To begin, let's fit a straight line to these four points. Each time you mark an emission line, IRAF tries to find the center of the feature in pixels. This gives an ordered pair (x,lambda) which we will be fitting with a polynomial. Set the fitting function to equal 'cheb', and the order 2, which is a line because it has two terms, a constant and a slope. The feature widths are narrow – try 3 pixels.

Type 'identify wavepri', move the cursor to the first line you wish to mark, and type 'm'. If the program finds a feature it will mark it with a hash, and prompt you for the wavelength. Type '4046' – you should see '4046.56 Hg' as the identifier in the bottom of the window. IRAF has scanned the file linelists$nehg.dat and found the closest line to the one you requested. This is a nice feature, as it means you don't have to type in the exact wavelength, just enough to ID it. Mark all four lines (if you err, type 'd' over the line to delete your mark).

We are ready to fit a line to these four points. Type 'f'. You should see a residual plot. This is how much each point deviates from a straight line. Note the RMS in the header. Type 'q' to return to the fit. You should now see a plot of intensity vs. wavelength, rather than intensity vs. pixels like you had before.

This is a good start, but we need more lines. Rather than ID them yourself, as long as we have a preliminary fit, let's allow IRAF to find them. Type 'e'. You should see a whole bunch of lines marked now. Before we fit these we must delete the blends, and try to mark something at the red end to fix the solution. To delete blends the best thing to do is to look at each one and make sure the profile is symmetrical and has a reasonable amount of flux. You can either use the 'window' command for each feature, or put the cursor to the rightmost line and type 'z' to zoom in. If you don't like the way the line looks, delete it with a 'd'. Go to the next line with a '+' or 'n'. When you have gone through the lines, type 'p' to return to see the entire spectrum again. It can be very helpful to have the file nehg.dat in another window, or a hardcopy, as you are checking lines to help identify blends.

Ok, let's see how you did. Type 'f' to fit and look at the residuals. Any that are way off are probably misidentified and should be deleted. In the residual window place the cursor over the errant points and type 'd', and then 'f'. If you have done this right, the residuals probably show a small wave, indicating a higher order fit would be useful. In the residual window type ':order 3' and then 'f' to fit a quadratic, ':order 4' and then 'f' for a cubic.

When you are satisfied, save your fit and write it to the database. If you look in the directory

database your solution is stored as idwavepri. We need to do this for the secondary as well. Rather than go through the whole procedure, let's start with the solution for wavepri and shift it. The appropriate task is 'reidentify'. Edit the parameters of reidentify to set the coordlist to 'linelists$nehg.dat', set the reference to wavepri and answer = yes to fit interactively. Type 'reidentify'. As before 'f' to fit, and 'd' to remove points. The last number printed out is the sigma of the fit in Angstroms.

Now we must apply the wavelength solution to your objects. When you have a whole lot of data you can use 'refspec' to help assign calibration lamps to object spectra. In our case we simply want to use wavepri for the primary, and wavesec for the secondary. So we edit the header to assign the keyword REFSPEC1 to 'wavepri' via 'hedit pri1 REFSPEC1 wavepri add+ ver-', and similarly for the secondary.

You apply the dispersion solution with 'dispcor'. Look at the parameters for this task. Since there is not much flux less than 4000Å, and longer than 7000Å there is a lot of terrestrial absorption, we should truncate the spectra in these regions. Input 4000 for w1 and 7000 for w2. It is helpful if all the spectra have the same lengths, so set nw=256. To apply the dispersion solution type 'dispcor pri1 pri2'. Have a look at pri2: you should see a spectrum of the primary plotted vs. wavelength. Now do the secondary and store this as sec2. Look at the ratio of the two spectra.

**Q15:** *Record the sigma of your wavelength fits, and include the files idwavepri and idwavesec in your report. Summarize how well the fitting procedure went. Plot your wavelength calibrated spectra pri2 and sec2 of the primary and secondary. Which of the two stars is hotter – i.e., bluer?*

Step 5: Flux-calibrate

Your output from step 4 includes the overall response of the telescope+grating+CCD. We will use the spectrum of Regulus, a spectrophotometric standard star, to correct for this response. The spectrum 'aleo2' has been extracted from a 2D image and wavelength calibrated like you did above for sigma Ori.

Edit the parameters for the 'standard' task. Set star_name to 'hr3982', which is one of Regulus's names. The calibration data caldir should be that for the bright standards, 'onedstds$bstdscal/'. While you are at it, look at the flux file for Regulus in onedstds$bstdscal/hr3982.dat. Each line is a wavelength, a magnitude, and a bandpass. The standard task simply adds up the flux it sees in our spectrum, and multiplies it by the appropriate value to bring Regulus to the level it should be at for each wavelength. If it asks you for an airmass, just input 1.0: we will not worry about that correction since sigma Ori and Regulus were at nearly the same airmass. Likewise, just enter a blank line for the extinction file. The observatory is 'george'.

Run the standard task with the input = aleo2 and the output = std. This file std you made is simply a calibration table. Now we want to fit a function to it by using 'sensfunc'. Try running this and look at the fit it gives you. It is possible to delete points, change orders, and so on as you did with the wavelength fits. Type 'f' to fit, and 'q' to quit as you did before. When you are done, use splot to look at your sensitivity function. IRAF appends the aperture number to the name, so you'll get something like 'sens.0070' as output. Rename this to 'sens' with imrename.

Now we need to apply this sensitivity function to your spectra using 'calibrate'. Set extinct=no and ignoreaps=yes and make sure you have the name of the sensitivity function correct, and then run 'calibrate pri2 pri3' to create a flux-calibrated spectrum pri3 from the wavelength calibrated

spectrum pri2. Do the same for the secondary and for Regulus, to create sec3 and aleo3.

Look at your results using splot. Within splot type 'f', then 'l' and 'q' to plot the spectra on a log scale, which is useful because the flux varies so much across the spectrum. Make hardcopies of all three spectra. The easiest way to make hardcopies of splot windows is to have the splot window active, type the colon, the enter ".snap eps" which will generate an encapsulated postscript file in the current working directory which can be printed with the UNIX lpr command. The filename will end in ".eps" in this case. Now 'splot pri3', and use 'w' and 'e' to zoom in on the feature near 6560Å, which is the H$\alpha$ line. Go to the left side of the line where it meets the continuum and type 'k', and then do so again for the right side of the line. The value of 'eqw' is the equivalent width in Å. Record this for the primary and for Regulus. Which has the stronger H$\alpha$ line?

Finally, sometimes we don't care about the shape of the spectrum, but are more interested in the lines. To see how this works, 'splot pri3', and type 's'. Enter 50 for the smoothing box. If you do this procedure 2 or 3 times you should get a nice smooth curve. Any residual features can be removed by positioning the cursor where you want it and typing 'j'. When you have a smooth response curve type 'i', and store it as resp. Divide pri3 by resp and put the result in pri4. The equivalent widths may be easier to measure here. Note we could have done this procedure from the pri2 file and obtained the same answer.

**Q16:** *Print out a plot of your sensitivity function from splot, and hardcopies of the aleo3, pri3, and sec3 files. Report the results of your equivalent width calculations. Print out what pri4 looks like.*

# 4 Final Comments and the Report

In this exercise you no doubt have (or will) encountered problems with IRAF, python, and DS9. It takes persistent effort and practice to get results. But once you get accustomed to the way things are set up, things will go surprisingly fast and you will be able to figure most new things out on your own. The hardest part is probably to get started, so don't get too discouraged if you encounter problems – try to work around or skip them and move on. You have a whole month+ to do this at the location of your convenience (and it is not weather-dependent), so the sooner you start to overcome the problems, the better you will succeed. *This is not a canned or push- button lab...you will have to think through some problems, so plan on it.* The spectroscopy part is rather involved, so don't leave this to the last minute.

Your suggestions for improvement will make the lab better next time.