Python CCD Processing Handbook **by Andrew Bradshaw, 6/4/12**

This series of instructions is meant to give you an introduction into image processing and plotting in python. It does not contain final science-grade analysis, but is rather a demonstration of possible methods.

All analysis should be possible within the Enthought python distribution, which is available for academics here: http://download.enthought.com/epd_7.2/

Image Reduction

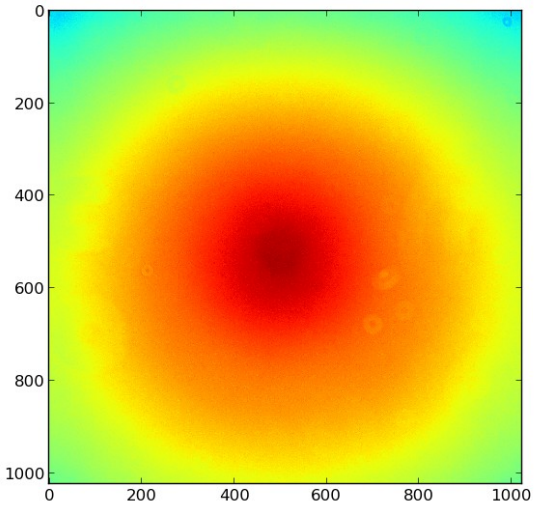
Given that we have a series of images, flats, biases, and darks, we would like to perform image arithmetic on them to get the optimum image out. Say our list of files is

```
bias-1.FIT bias-4.FIT dark-2.FIT dark-5.FIT flat-v-3.FIT m13-v-60s-1.FIT
bias-2.FIT bias-5.FIT dark-3.FIT flat-v-1.FIT flat-v-4.FIT m13-v-60s-2.FIT
bias-3.FIT dark-1.FIT dark-4.FIT flat-v-2.FIT flat-v-5.FIT m13-v-60s-3.FIT
```

Now we would like to first of all make a master flat, a master bias, and a master dark. This is necessary in order to reduce the effect of cosmic rays. To do this in python, execute the code:

```
import pyfits
darks = array([pyfits.getdata("dark-%d.fit" % n) for n in range(1,6)])
print shape(darks)
(5, 1024, 1024)
dark = median(darks,axis=0)
print shape(dark)
(1024, 1024)
imshow(dark,cmap=cm.gray)
```

Where the last command is just to see what the final image looks like. If it doesn't display much, try using imshow(log(dark),cmap=cm.gray) in order to stretch the brightness levels. The master flat is shown to the right; notice that the detector is defintely not flat!



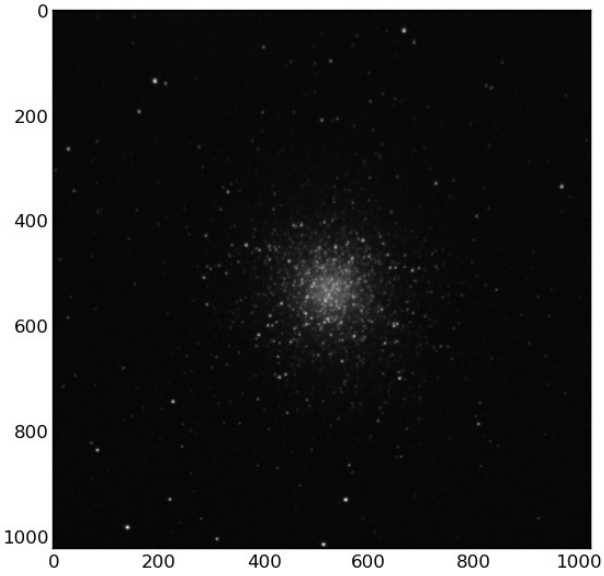
Do a similar median combine of the flats and biases. And load in all of the science frames but do not take the median. Now we are ready to perform the image arithmetic operation:

$$final = \frac{science - dark}{flat - bias}$$

in python this looks like:

```
final = ((science-dark)/(flat-bias))*mean(flat-bias)
shape(final)
(3, 1024, 1024)
```

Where the final image should be an array of however many science frames you had. Also note that the denominator (flat-bias) should be normalized, so that your pixel values stay high.



Once you have the final array of corrected images, you can then median combine the final images into one image, which is shown on the right.

Image analysis

Now that we have cleaned up our images a bit, we can do some image analysis! Say we want to find all of the stars in our image. There are lots of ways to do this, inside of python and out. A kinda good way to find centroids in python is simply using scipy's center of mass function.

The method is this: first tag all of the pixels which are larger than the threshold, defined here to be the mean of the image plus 2 standard deviations. This will create a map of pixels which satisfy the threshold, which we hold in the variable array called labels (the number of stars is held in "num"). Then perform the center of mass calculation on each of these pixel chunks:

```
import scipy.ndimage as snd
threshold = img.mean() + 2*img.std()
labels, num = snd.label(img > threshold, np.ones((3,3)))
centers = snd.center_of_mass(img, labels, range(1,num+1))
```

The center of mass function has now found the centers of some stars, but the centers is in a clunky list format. Just extract the x and y values using the command:

```
x = array(centers)[:,0]
y = array(centers)[:,1]
```

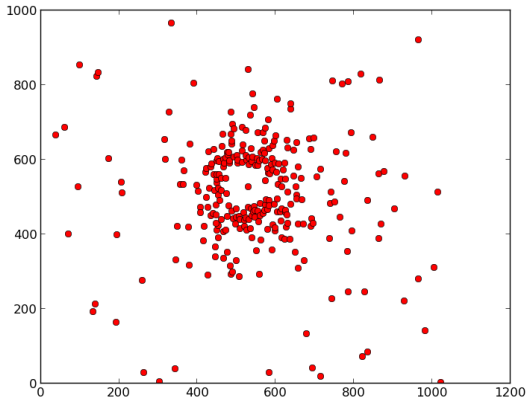
The (potential) star center (x, y) values are plotted below.

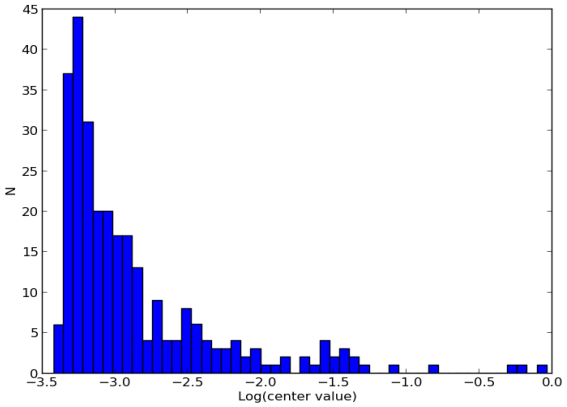
Now we have some numbers we can play with! How about making a histogram of the distance from the center?

```
r = sqrt((x-512)**2+(y-512)**2)
hist(r, bins=50)
```

Or how about the distribution of the flux values at the centers of each star? Use the x and y values of the centers of stars to subscript the image, and hold each flux measurement in an array called values. Note that we had to convert the X and Y centers to integers in order to be able to subscript our image array with them, and that we take the logarithm of the pixel value to give a rough magnitude of the stars (the zero-point offset is unknown without calibration).

```
xint = x.astype(int)
yint = y.astype(int)
values = img[xint,yint]
hist(log(values),bins=50)
```





There are lots of other things we could do here, now that our data is digitized and cleaned. What other operations could you do with the data now?

Another filter, and color data!

Let's load in another set of color images, and play with those too. Read in the flats for the B data, and the B data itself, then perform the image arithmetic that we did above. The bias and dark frames are filter-independent, but the flat image is not, because the filter throughput is factored into each pixel's sensitivity. So perform the reduction on B as we did in V:

```
bflats = array([pyfits.getdata("flat-b-%d.FIT" % n) for n in range (1,6)])
b = array([pyfits.getdata("m13-b-60s-%d.FIT" % n) for n in range (1,4)])
finalb = (b-dark)/(bflat-bias)*mean(bflat-bias)
```

Assuming the B and V images are perfectly aligned (a poor assumption), we can now look at how the center values of the stars are distributed in B and V at the same time! This is very useful, because

we can plot the star brightness as a function of the ratio of the brightness in B and V, which is a proxy for temperature (called color). This is shown below.

```
imgv = img        # our final v image
imgb = median(finalb,axis=0)     # the median of our reduced B images
bflux = imgb[xint,yint]
vflux = imgv[xint,yint]
plot(log(bflux)-log(vflux),log(bflux),'ro')
axis([-2.5,0,-1.0,-5]) #flip the axis
```

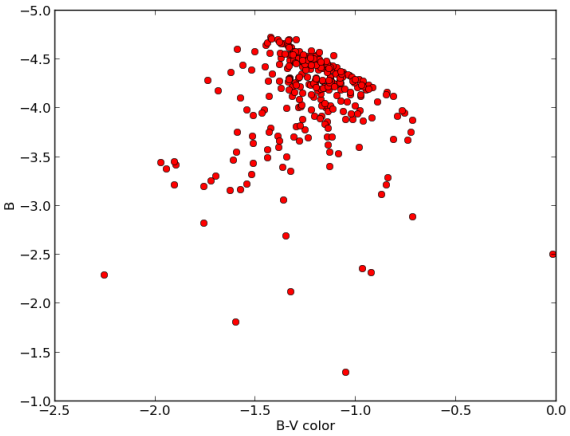
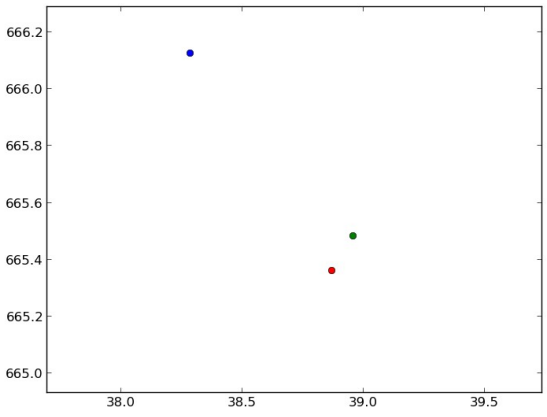


Image alignment

We previously assumed that the images in B and V are aligned with one another, but this is probably not the case. In fact, each exposure is probably offset from one another! We can see this easily by looking at the distribution of our star center values in B and V, which is shown on the right. We must correct for this offset.

```
threshold = imgb.mean() + 3*imgb.std()
labels,num = snd.label(imgb > threshold, np.ones((3,3)))
centers = snd.center_of_mass(imgb,  labels, range(1,num+1))
# the same "star center-finding" process from above
xb=array(centers)[:,0]
yb=array(centers)[:,1]
plot(xb,yb,'go')
plot(x,y,'bo') #b for blue, g for green
```



To correct for the offset, use scipy to interpolate the image and then shift. This is done with the following commands:

```
 from scipy.ndimage import interpolation
newb = interpolation.shift(bimg,[.6, -.7])
pyfits.writeto('out.fits',newb,hdr)
```

Where in the last line we have saved the new offset image as 'out.fits' so that we can use it later for analysis. The 'hdr' is the header from the old fits file, which can be obtained by entering

"foo,hdr=pyfits.getdata('old_b_image.fits',header=True)". The offset-corrected star center is shown as a red point in the image to the right (more corrections would be necessary).

Note that the new offset fits image will have some edge effects, which may need to be thrown out or ignored in the analysis.

PSF Fitting

Say we want to better understand what the Point Spread Function (PSF) looks like across our image. First we want to select some representative stars in our image, so open the .fits file in DS9. Create a small region around some stars which are bright (but unsaturated) and which don't have any friends nearby. Save the regions to a file (making sure to save it in X Y format) and then open it in python. My filename is ds9.reg, so I open it like:

```
import numpy
regions = numpy.loadtxt("ds9.reg")
xc,yc = regions[:,0],regions[:,1]
```

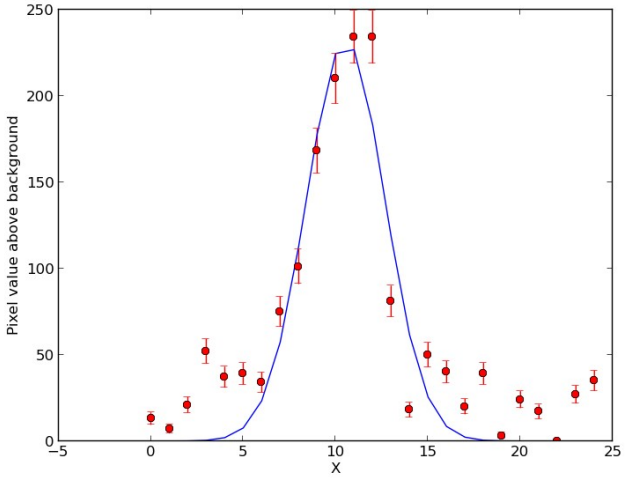
Let us illustrate the PSF fitting process using just one of the centers, say, the first one in the list. We will be fitting a Gaussian to only a line plot, also just for illustrative purposes. With your data you perhaps would want to fit more than one star, and in two dimensions, and maybe not even a Gaussian! But this is the gist of it:

```
import scipy.optimize
y = vimg[xc[0],yc[0]-10:yc[0]+10] # take star values in a 1 by 20 pixel strip
x = linspace(0,len(y)-1,len(y)) # make an array of x values, same length as y
```

Now define an "anonymous function" in python (called a lambda form, or lambda), which will take in a set of parameters and output the function operated upon those parameters. Here we define a Gaussian as our fitting function. We also define a lambda that is the fitted function minus the data; this function is what we will be minimizing using scipy.

```
fitfunc = lambda p, x: p[0]*scipy.exp(-(x-p[1])**2/(2.0*p[2]**2))
errfunc = lambda p, x, y: fitfunc(p,x)-y
parameters,foo = scipy.optimize.leastsq(errfunc,(2000,10,3),args=(x,y))
# foo is a throwaway variable
```

The last line is the fitting part, it is minimizing our error function using least squares by taking in three arguments: 1) the function to be minimized, 2) our initial guess for the parameters (amplitude, mean, and std dev), and 3) the x and y values of our data. The plot of our fit and the data (with error bars) is included here, and the plotting code is below:



```
clf()
xlabel('X')
ylabel('Pixel value above background')
plot(x,fitfunc(parameters,x))
errorbar(x,y,yerr=sqrt(y),fmt='ro')
```

Aperture Photometry and Calibration

Finally, it is necessary to get a magnitude for the stars in our field. To do this properly see the details starting on page 310 in Chromey. Here we will demonstrate the method in python using various apertures on a galaxy. The method can certainly be improved upon, but that is left to the student.

First we need to choose some apertures to measure the image flux. We can do this in DS9; first open the image and then go to the Region menu, and then under the Shape sub-menu click on "Annulus". Now clicking anywhere in the image will plop down an annulus whose size can be adjusted. Place several annuli with different inner and outer radii, centered on the galaxy, as shown in the image on the right. Once you have all of the regions you want, Save the regions to a file, this time using the "pros" format (which can be easily read into python).

Read these parameters into python using numpy, selecting only the 3rd-6th columns which represent the x/y center and inner/outer radii.

```
import numpy
import pyfits
ann=numpy.loadtxt('annulus.reg',usecols=(2,3,4,5))
xc=ann[0,0]
yc=ann[0,1]
inner=ann[:,2] #copy out the radius columns to their own arrays
outer=ann[:,3]
img=pyfits.getdata('M51 V Combined Average.FIT')
```

Now that we know where to look, we want to grab all of the pixels in the image which are within each annulus. One way to do this is with a 'for loop', which will run through all pixels (i,j). Then add a condition which checks whether the pixel (i,j) is in each of the annuli, and if so, then add its flux and increment the counter for that annulus. In python this looks like:

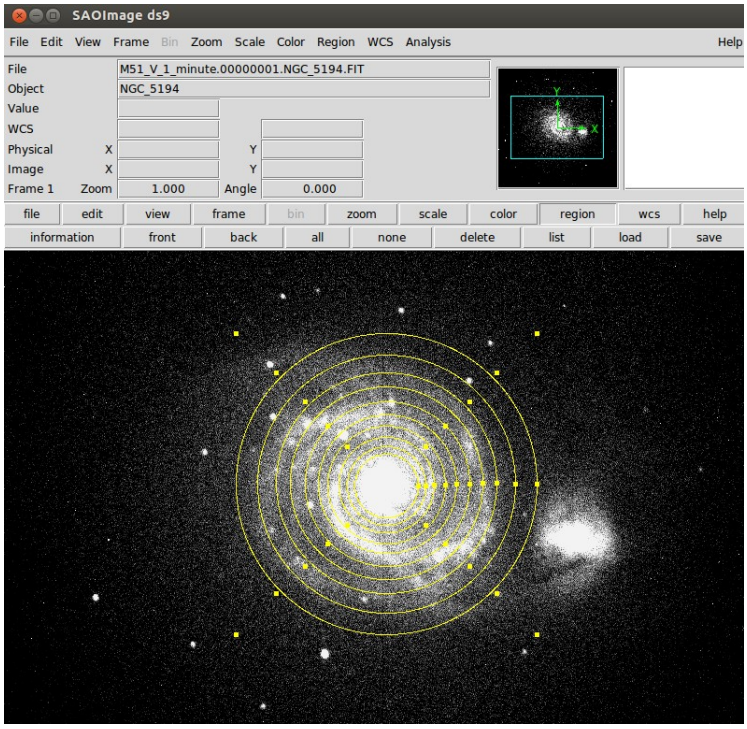
```
npix = numpy.zeros(len(ann))
pix = numpy.zeros(len(ann))
for i in range(img.shape[0]):
      for j in range(img.shape[1]):
            r = numpy.sqrt((i-xc)**2+(j-yc)**2)
            for k in range(len(ann)):
                  if inner[k]<r<outer[k]:
                        pix[k]+=img[i,j]
                        npix[k]+=1
```

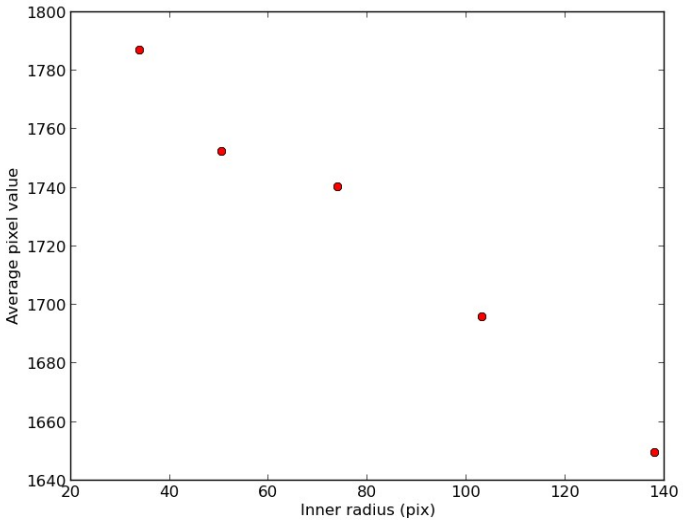
Where we have created an array of zeros to be filled in with aperture data using numpy.zeros. The for statement can be entered into the command line by just hitting enter after typing "for i in range(blah):"; the colon at the end makes python return a new line but not evaluate it, allowing you to enter further commands.

This process results in two filled-in arrays of the flux within the annulus and the number of pixels within each annulus. We can divide the first by the second to calculate the average pixel value inside each annulus! We can also plot this, which is shown below:

```
In [91]: pix,npix,pix/npix
Out[91]:
(array([ 3523697., 12751457., 17776646., 7576695., 35294274.]),
array([ 1972., 7328., 10484., 4324., 21400.]),
array([ 1786.86460446, 1740.10057314, 1695.59767264, 1752.24213691,
1649.26514019]))
import matplotlib.pyplot as plt
plt.plot(inner,pix/npix,'ro'), plt.xlabel('Inner radius (pix)'),
plt.ylabel('Average pixel value')
plt.show()
```

Now you can imagine using this method to measure a star's flux: choose an aperture to sum up all of the star's flux, and then also choose a sky annulus sufficiently far away from the center. For galaxies, you can (and should) do this for more than 5 aperture sizes to become more precise, and of course subtract the background using a method of your choice.





Making a Color Composite

If you have several bands of images, you can pretty easily make a color composite. Just match up the filters (e.g. I, V, B) with the typical R-G-B colors and then make plot them using the imshow command. In python you can do this easily using an extra piece of code called img_scale.py which someone has kindly written for us (available here: http://andrewnomy.com/pub/img_scale.py). So first import all of the necessary functions and also your images:

```
import pyfits, img_scale, numpy, pylab
i=pyfits.getdata('r-image.fits')    # and also for a 'green' and 'blue' image
```

Then you need to define how to scale the images, so choose a maximum and minimum value (which I do by using the mean and std. deviations), and use the image scaling code to do an arcsin stretch:

```
imin,imax=i.mean()+.75*i.std(),i.mean()+5*i.std() # also do for G and B
img=numpy.zeros((1024,1024,3))
img[:,:,0]=img_scale.asinh(i,scale_min=imin,scale_max=imax)
img[:,:,1]=img_scale.asinh(v,scale_min=vmin,scale_max=vmax)
img[:,:,2]=img_scale.asinh(b,scale_min=bmin,scale_max=bmax)
```

And then plot the result! An example of a color composite done this was is shown on the right (of M51, the 'Whirlpool Galaxy').

```
pylab.clf()
pylab.imshow(img)
pylab.show()
```