Tutorial for Stacking FITS files and Bootstrapping with Python

Jonathan Mercedes Feliz[1]

[1]Dept. of Physics, University of Connecticut, Storrs, CT, 06269, USA
contact: jonathan.mercedes_feliz@uconn.edu

Abstract

This tutorial should be able to demonstrate how to access and stack FITS files from any catalog or data set using Python, within the Jupyter Notebook module. Being able to add a buffer to center all your files equally, stacking and bootstrapping. Providing a step by step procedure using an example of a small project.

Introduction

Specifically we will be using FITS files from 3D-HST to look at and stack 24 micron postage stamps for detection, which is a good proxy for the dust attenuation bump. This is to ultimately get a total star formation rate vs mass relationship between quiescent galaxies within the sample through a selection process. Specifically stacking to measure flux and calculate infrared luminosity, which we will then use to bootstrap for error analysis.

Getting Started

The first thing you will want to do is make sure you have Python 2.7 (which is what I will be using) or Anaconda installed on your computer. You can do this by either going on https://www.anaconda.com/distribution/ or following the instructions on Python Beginner's Guide. Make sure you install the right installer depending on your OS system. You will need many python libraries and modules for the following tutorial but all you need to do is install them by accessing your terminal and following the documentation Installing Python Modules supplied by Python. Accessing Jupyter Notebook is incredibly easy depending on which of the two avenues above you choose, if you have Anaconda installed, just open the Anaconda-Navigator and launch the Notebook application. If you installed Python directly, then you can follow Code Academy's how-to article to access them at: Code Academy Article.

Now the important thing to do as you start your venture is to make sure to import whatever module you will need and how you will be referencing back to them:

```
import astropy
from astropy.table import Table, Column, join
from astropy.coordinates import SkyCoord
from astropy.table import Column
from astropy.io import ascii
import numpy as np
import matplotlib.pyplot as plt
from astropy.utils.data import get_pkg_data_filename
from astropy.visualization import astropy_mpl_style
plt.style.use(astropy_mpl_style)
from astropy.io import fits
from scipy import stats
import gzip
import os
import pandas as pd
import glob
import gc
import subprocess
from photutils import aperture_photometry, CircularAperture, CircularAnnulus
%matplotlib inline
```

You could also define any functions that could be helpful and are necessary for your analysis in the next cell, like for example the following:

```
def formatNumber(num):
    if num % 1 == 0:
        return int(num)
    else:
        return num

def change_column_order(df, col_name, index):
    cols = df.columns.tolist()
    cols.remove(col_name)
    cols.insert(index, col_name)
    return df[cols]

def checker(image_data):
    p=np.isnan(image_data)
    ay=np.count_nonzero(p == True)
    ya=np.count_nonzero(p == False)
    full = 88209.0
    percent = (ya)/full

    return percent

def distance_from_center(x,y):
    distance = (x-148)**2 + (y-148)**2
    return distance

def find_nearest(array, value):
    array = np.asarray(array)
    idx = (np.abs(array - value)).argmin()
    return array[idx]
```

These functions will be useful for me but essentially the first one just changes a float to an integer. The second will come in handy when we talk about adding a buffer to your FITS image, but all it does is it takes a data frame and you can change the position of a column elsewhere. The third one checks how much of a FITS image is filled with NaN, but this is only for a 297×297 square image. The fourth is again only useful for a 297×297 square image, but it gets the distance away from the center pixel. The final function finds the nearest element in an array that is similar to a specific value I decide to give. Most of these might not make sense for you to use, but they will be used by me to some degree as I go through the motions. Now that I have all of my modules and functions ready to go, I start with accessing the catalogs that I will be working with.

Reading in Catalog Path/Within Computer

```
catalog_path="/Users/jmf/Desktop/UConn/Research/merged_catalogs/" # Trailing / is required

cosmos = Table.read(catalog_path+'cosmos.zbest.v4.1.5.fits')
aegis = Table.read(catalog_path+'aegis.zbest.v4.1.5.fits')
uds = Table.read(catalog_path+'uds.zbest.v4.1.5.fits')
goodss = Table.read(catalog_path+'goodss.zbest.v4.1.5.fits')
goodsn = Table.read(catalog_path+'goodsn.zbest.v4.1.5.fits')

cosmos_sfr = ascii.read(catalog_path+'cosmos_3dhst.v4.1.5.zbest.sfr', data_start=0,header_start=0,delimiter=' ')
aegis_sfr = ascii.read(catalog_path+'aegis_3dhst.v4.1.5.zbest.sfr', data_start=0,header_start=0,delimiter=' ')
uds_sfr = ascii.read(catalog_path+'uds_3dhst.v4.1.5.zbest.sfr', data_start=0,header_start=0,delimiter=' ')
goodss_sfr = ascii.read(catalog_path+'goodss_3dhst.v4.1.5.zbest.sfr',data_start=0,header_start=0,delimiter=' ')
goodsn_sfr = ascii.read(catalog_path+'goodsn_3dhst.v4.1.5.zbest.sfr', data_start=0,header_start=0,delimiter=' ')
```

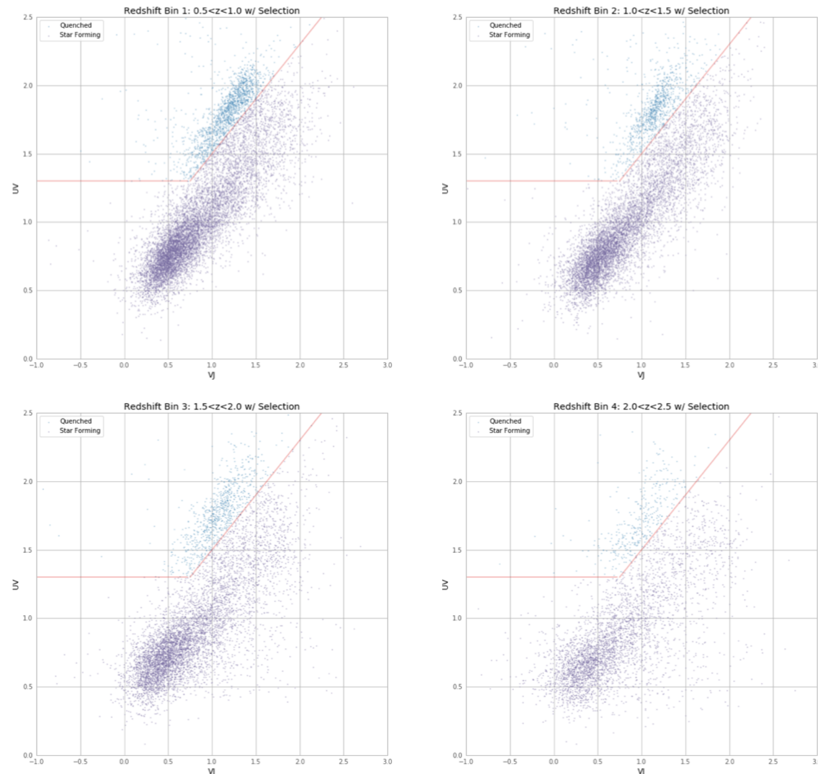
Where I have read in the 5 fields of 3D-HST as well as some information from an ASCII table. I now create a flag condition, I call it the UVJflag, where I make a selection from looking at the UVJ diagram. The selection being that quenched galaxies are found to be $UV > 1.3$ and $UV > 0.8 \cdot VJ + 0.7$, while star forming galaxies are located elsewhere. If UVJflag is equal to 1 then it is a quenched galaxy, and when it is equal to 0 then it is a star forming galaxy. I go ahead and make a for loop where I go through each of the catalogs and assign a UVJ flag to galaxies that meet my selection requirements as well as take any necessary information that we will need into a .txt file:

Making the UVJ flag for all galaxies, all redshifts

```
bleh=[]
for k in range(5):
    if k==0:
        cosmos = Table.read(catalog_path+'cosmos.zbest.v4.1.5.fits')
        idv=cosmos['ID'][0]
        cosmos_sfr = ascii.read(catalog_path+'cosmos_3dhst.v4.1.5.zbest.sfr'
                                , data_start=0,header_start=0,delimiter=' ')
        UVJcondition=(cosmos['UV'][0]>1.3) & (cosmos['UV'][0]>(0.8*cosmos['VJ'][0])+0.7)
        UVJflag=np.zeros_like(len(UVJcondition))
        UVJflag=UVJcondition*1.0
        use=cosmos['USE'][0]
        field='COSMOS'
        mass=cosmos['LMASS'][0]
        uv=cosmos['UV'][0]
        vj=cosmos['VJ'][0]
        z=cosmos['Z_BEST'][0]
        luvcal=(1.5)*(cosmos_sfr['L_2800'])
    elif k==1:
        aegis = Table.read(catalog_path+'aegis.zbest.v4.1.5.fits')
        idv=aegis['ID'][0]
        aegis_sfr = ascii.read(catalog_path+'aegis_3dhst.v4.1.5.zbest.sfr', data_start=0,header_start=0
                               ,delimiter=' ')
        UVJcondition=(aegis['UV'][0]>1.3) & (aegis['UV'][0]>((0.8*aegis['VJ'][0])+0.7))
        UVJflag=np.zeros_like(len(UVJcondition))
        UVJflag=UVJcondition*1.0
        use=aegis['USE'][0]
        field='AEGIS'
        mass=aegis['LMASS'][0]
        uv=aegis['UV'][0]
        vj=aegis['VJ'][0]
        z=aegis['Z_BEST'][0]
        luvcal=(1.5)*(aegis_sfr['L_2800'])
    elif k==2:
        uds = Table.read(catalog_path+'uds.zbest.v4.1.5.fits')
        idv=uds['ID'][0]
        uds_sfr = ascii.read(catalog_path+'uds_3dhst.v4.1.5.zbest.sfr', data_start=0,header_start=0
                             ,delimiter=' ')
        UVJcondition=(uds['UV'][0]>1.3) & (uds['UV'][0]>((0.8*uds['VJ'][0])+0.7))
        UVJflag=np.zeros_like(len(UVJcondition))
        UVJflag=UVJcondition*1.0
        use=uds['USE'][0]
        field='UDS'
        mass=uds['LMASS'][0]
        uv=uds['UV'][0]
        vj=uds['VJ'][0]
        z=uds['Z_BEST'][0]
        luvcal=(1.5)*(uds_sfr['L_2800'])
```

```
    elif k==3:
        goodsn = Table.read(catalog_path+'goodsn.zbest.v4.1.5.fits')
        idv=goodsn['ID'][0]
        goodsn_sfr = ascii.read(catalog_path+'goodsn_3dhst.v4.1.5.zbest.sfr', data_start=0,header_start=0
                                ,delimiter=' ')
        UVJcondition=(goodsn['UV'][0]>1.3) & (goodsn['UV'][0]>((0.8*goodsn['VJ'][0])+0.7))
        UVJflag=np.zeros_like(len(UVJcondition))
        UVJflag=UVJcondition*1.0
        use=goodsn['USE'][0]
        field='GOODSN'
        mass=goodsn['LMASS'][0]
        uv=goodsn['UV'][0]
        vj=goodsn['VJ'][0]
        z=goodsn['Z_BEST'][0]
        luvcal=(1.5)*(goodsn_sfr['L_2800'])
    else:
        goodss = Table.read(catalog_path+'goodss.zbest.v4.1.5.fits')
        idv=goodss['ID'][0]
        goodss_sfr = ascii.read(catalog_path+'goodss_3dhst.v4.1.5.zbest.sfr', data_start=0,header_start=0
                                ,delimiter=' ')
        UVJcondition=(goodss['UV'][0]>1.3) & (goodss['UV'][0]>((0.8*goodss['VJ'][0])+0.7))
        UVJflag=np.zeros_like(len(UVJcondition))
        UVJflag=UVJcondition*1.0
        use=goodss['USE'][0]
        field='GOODSS'
        mass=goodss['LMASS'][0]
        uv=goodss['UV'][0]
        vj=goodss['VJ'][0]
        z=goodss['Z_BEST'][0]
        luvcal=(1.5)*(goodss_sfr['L_2800'])
    for i in range(len(idv)):
        bleh.append([field,idv[i],UVJflag[i],uv[i],vj[i],mass[i],z[i],use[i],luvcal[i]])
np.savetxt('fields-ids01.txt',bleh, fmt='%s', header='field id uvj-flag#q=1,sf=0 uv vj mass z use calculatedL_UV')
```

You can see how the UVJ diagram looks with the distinction between SF (star forming) and Q (quenched) galaxies for 4 separate redshift bins below:



Stacking

Before we even start to stack our FITS files, we need to make sure that they all have the same dimensions and that they all have the same center. If they don't have the same dimensions and center then you will not capture every pixel correctly. If you don't know what is the size of your FITS image then you could easily check with `print(image_data.shape)`, and this will show you the dimensions of it.

Now that you know the sizes of your data you can decide a size to assign to them all. For example, in my case `COSMOS, GOODS-N,` and `GOODS-S` has an image size of (283×283) while `AEGIS` and `UDS` has (295×295). I decided to add a buffer to all of them so that they're (297×297). The reason why these are all odd by odd dimensions is because the center would be exactly in the middle, if not then the center would be offset. This means that for the `AEGIS` and `UDS` FITS files I will add two rows and columns of NaN surrounding the data, as to not affect the overall information, which I essentially do here:

```
fields=np.genfromtxt('fields-ids01.txt',usecols=[0],dtype=None)
idv= np.genfromtxt('fields-ids01.txt',usecols=[1],dtype=None)
uvjflag=np.loadtxt('fields-ids01.txt',usecols=[2])
uv=np.loadtxt('fields-ids01.txt',usecols=[3])
vj=np.loadtxt('fields-ids01.txt',usecols=[4])
mass=np.loadtxt('fields-ids01.txt',usecols=[5])
z=np.loadtxt('fields-ids01.txt',usecols=[6])
use=np.loadtxt('fields-ids01.txt',usecols=[7])

shine=[]
fields1=fields[np.where((uvjflag==0.) & (mass>8.8) & (z>0.5) & (z<1) & (use==1))]
idv1=idv[np.where((uvjflag==0.) & (mass>8.8) & (z>0.5) & (z<1) & (use==1))]

for i in range(len(fields1)):
    catalog__path="/Volumes/WHITAKER_DATA/3DHST/"+fields1[i]+"/MIPS/phot/OUT_ALL/"+fields1[i]+"_MIPS_sci/"
    y=formatNumber(idv1[i])
    ##  changing a zipped fits file to a regular one
    file_call = 'gunzip '+'/Volumes/WHITAKER_DATA/3DHST/'+fields1[i]+"/MIPS/phot/OUT_ALL/"+fields1[i]+'_MIPS_sci/'
    +str(y)+'/'+str(y)+'_phot.fits.gz'
    output = subprocess.call(file_call,shell=True)
    image_file=catalog__path+str(y)+"/"+str(y)+"_phot.fits"
    image_data = fits.getdata(image_file, ext=0)
    image_oh = fits.open(image_file)

    image_x, image_y = image_data[:,-1], image_data[:,-2]

    datas=pd.DataFrame(np.array(image_data).byteswap().newbyteorder())
    if image_data.shape==(295,295):
        datas['295']=np.nan
        datas['-1']=np.nan
        datas=change_column_order(datas,'-1',0)

        datas.loc[-1]=np.nan
        datas.loc[295]=np.nan
        datas = datas.sort_index(ascending=True)


        hdu = fits.PrimaryHDU(datas)
        hdul = fits.HDUList([hdu])
        hdul.writeto(fields1[i]+str(idv1[i])+'.fits',overwrite=True)
```

Where the first few lines are just inputting variables from the .txt file I created earlier. The two lines after `shine=[]` is selecting only galaxies that are above $\log M_\star > 8.8$, are SF and within a redshift bin of $0.5 < z < 1.0$. The `for` loop allows for us to go through the catalogs and access all of the FITS files that satisfy that condition, open up the data and concatenates two columns filled with NaNs. We use one of the functions we've created previously `change_column_order` and just move one of the new NaN columns to the immediate right. The rest is to create two rows of NaNs and then order the the data array so we can have the buffer truly encompass the data. Finally

we just write our new buffered data into a FITS file using the last three lines, where within the `.writeto()` argument we set overwrite to `True` so we can rewrite the file whenever we need to fix or add any code within the loop. For `COSMOS, GOODS-N,` and `GOODS-S` we do essentially the same but we add 14 columns and rows:

```
    else:
        datas['283']=np.nan
        datas['284']=np.nan
        datas['285']=np.nan
        datas['286']=np.nan
        datas['287']=np.nan
        datas['288']=np.nan
        datas['289']=np.nan
        datas['-1']=np.nan
        datas['-2']=np.nan
        datas['-3']=np.nan
        datas['-4']=np.nan
        datas['-5']=np.nan
        datas['-6']=np.nan
        datas['-7']=np.nan
        datas=change_column_order(datas,'-1',0)
        datas=change_column_order(datas,'-2',0)
        datas=change_column_order(datas,'-3',0)
        datas=change_column_order(datas,'-4',0)
        datas=change_column_order(datas,'-5',0)
        datas=change_column_order(datas,'-6',0)
        datas=change_column_order(datas,'-7',0)

        datas.loc[-1]=np.nan
        datas.loc[-2]=np.nan
        datas.loc[-3]=np.nan
        datas.loc[-4]=np.nan
        datas.loc[-5]=np.nan
        datas.loc[-6]=np.nan
        datas.loc[-7]=np.nan
        datas.loc[283]=np.nan
        datas.loc[284]=np.nan
        datas.loc[285]=np.nan
        datas.loc[286]=np.nan
        datas.loc[287]=np.nan
        datas.loc[288]=np.nan
        datas.loc[289]=np.nan
        datas = datas.sort_index(ascending=True)


        hdu = fits.PrimaryHDU(datas)
        hdul = fits.HDUList([hdu])
        hdul.writeto(fields1[i]+str(idv1[i])+'.fits',overwrite=True)
```

Now that we've added a buffer to all of our selected FITS files, we're ready to start stacking. There are generally two different types of stacks you can choose to do. One is a mean stack while the other is a median stack. Doing a median stack is less sensitive to extreme things so for this example we'll be doing that and not using a mean stack. Essentially all a stack is, is like shooting a dart through each pixel for every z value of this "cube" of images and we flatten it, returning us back to a 2-D image. What I'll be doing is within a redshift bin, making 11 mass bins, with 0.2 dex interval between them.

```python
# z1m1={9,9.2}, z1m2={9.2,9.4}, z1m3={9.4,9.6}, z1m4={9.6,9.8}, z1m5={9.8,10}, z1m6={10,10.2},
# z1m7={10.2,10.4}, z1m8={10.4,10.6},z1m9={10.6,10.8}, z1m10={10.8,11}, z1m11={11,11.2}

z1m1=[False]*len(mass)
z1m2=[False]*len(mass)
z1m3=[False]*len(mass)
z1m4=[False]*len(mass)
z1m5=[False]*len(mass)
z1m6=[False]*len(mass)
z1m7=[False]*len(mass)
z1m8=[False]*len(mass)
z1m9=[False]*len(mass)
z1m10=[False]*len(mass)
z1m11=[False]*len(mass)

for i in range(len(mass)):
    if mass[i]<9.2:
        z1m1[i]=True
    elif mass[i]>=9.2 and mass[i]<9.4:
        z1m2[i]=True
    elif mass[i]>=9.4 and mass[i]<9.6:
        z1m3[i]=True
    elif mass[i]>=9.6 and mass[i]<9.8:
        z1m4[i]=True
    elif mass[i]>=9.8 and mass[i]<10:
        z1m5[i]=True
    elif mass[i]>=10 and mass[i]<10.2:
        z1m6[i]=True
    elif mass[i]>=10.2 and mass[i]<10.4:
        z1m7[i]=True
    elif mass[i]>=10.4 and mass[i]<10.6:
        z1m8[i]=True
    elif mass[i]>=10.6 and mass[i]<10.8:
        z1m9[i]=True
    elif mass[i]>=10.8 and mass[i]<11:
        z1m10[i]=True
    elif mass[i]>=11 and mass[i]<11.2:
        z1m11[i]=True


z1mbin1,z1mbin2,z1mbin3,z1mbin4,z1mbin5 = mass[z1m1], mass[z1m2], mass[z1m3], mass[z1m4], mass[z1m5]
z1mbin6,z1mbin7,z1mbin8,z1mbin9,z1mbin10 = mass[z1m6], mass[z1m7], mass[z1m8], mass[z1m9], mass[z1m10]
z1mbin11= mass[z1m11]

id1mbin1, id1mbin2, id1mbin3, id1mbin4, id1mbin5 = idv1[z1m1], idv1[z1m2], idv1[z1m3], idv1[z1m4], idv1[z1m5]
id1mbin6, id1mbin7, id1mbin8, id1mbin9, id1mbin10 = idv1[z1m6], idv1[z1m7], idv1[z1m8], idv1[z1m9], idv1[z1m10
id1mbin11= idv1[z1m11]

field1, field2, field3, field4, field5 = fields1[z1m1], fields1[z1m2], fields1[z1m3], fields1[z1m4], fields1[z
field6, field7, field8, field9, field10 = fields1[z1m6], fields1[z1m7], fields1[z1m8], fields1[z1m9], fields1[
field11 = fields1[z1m11]

id1mbin1,id1mbin2,id1mbin3,id1mbin4,id1mbin5= id1mbin1.astype('str'),id1mbin2.astype('str'),id1mbin3.astype('s
id1mbin6,id1mbin7,id1mbin8,id1mbin9,id1mbin10= id1mbin6.astype('str'),id1mbin7.astype('str'),id1mbin8.astype('
id1mbin11= id1mbin11.astype('str')

com1,com2,com3,com4,com5,com6=galaxies_bin_maker(field1,id1mbin1),galaxies_bin_maker(field2,id1mbin2),galaxies
com7,com8,com9,com10,com11=galaxies_bin_maker(field7,id1mbin7),galaxies_bin_maker(field8,id1mbin8),galaxies_bi

z1matching1,z1matching2,z1matching3,z1matching4,z1matching5,z1matching6=galaxy_list_maker(images1,com1),galaxy
z1matching7,z1matching8,z1matching9,z1matching10,z1matching11=galaxy_list_maker(images1,com7),galaxy_list_make
```

We've introduced a `for` loop to separate out all of the galaxies into the mass bins they fall into, using the `galaxies_bin_maker()` and `galaxies_list_maker()` functions. Where the former creates a list of the field and ID number of the galaxies in a mass bin, while the latter takes that list and the directory containing all the selected galaxies and creates another list with only the mass binned galaxies:

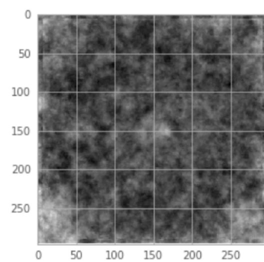
```python
def galaxies_bin_maker(binned_field,binned_ids):
    com=[]
    for i in range(len(binned_field)):
        paa = binned_field[i]+binned_ids[i]
        paa = str(paa)
        com.append(paa)

    return com

def galaxy_list_maker(images,galaxies_bin):
    matching1=[]
    for i in range(len(galaxies_bin)):
        matching = [s for s in images if str(galaxies_bin[i]) in s]
        matching1.append(matching)
        matching1=[''.join(x) for x in matching1]
    return matching1
```

Stacking is now possible now that we have broken down our selected galaxies into mass bins we just use a few lines of code. We first create an empty list which I do by writing `cube=[]` and we need to incorporate a `for` loop to iterate through the list we created in the previous cell using the `galaxies_list_maker()` function. We then make sure to access the data, using the `fits.getdata()` function, and we have `memmap=False` to avoid running into a problem where if you're opening too many FITS files in a loop the memory associated with that process gets full. So this just guarantees that after every iteration is done with the FITS data it erases it from the memory to continue. We append each FITS image onto the `cube` list essentially creating a cube as my variable name suggests. The stack is ultimately completed by using the `np.nanmedian()` function, which guarantees us that the NaNs we introduced from the buffer is ignored.

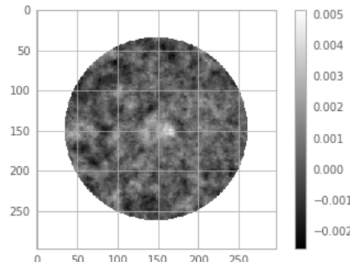
```python
def new_stack(matchedarray,binnum,LUV):
    cube = []
    for i in range(len(matchedarray)):
        im = fits.getdata(matchedarray[i],ext=0,memmap=False)
        cube.append(im)
    cube = np.asarray(cube)
        # creates my stacked cube

    imagen=np.nanmedian(cube,axis=0)
        # creates 1 version of the median stack

    flux = aperture_phot(imagen)
    meanluv = np.nanmean(LUV[binnum])

    print('Stack Done')

    return flux, meanluv
```

The `flux` line is the crux of why we wanted to stack, for detection. Which we measure from a stack to get a given flux using aperture photometry. Which we do with the `aperture_phot()` function, which allows us to get flux through the following steps:

```python
def aperture_phot(image_data):
    # '' / 0.18'' per pix converts it into pixels, so a 24'' radius circle is 133.3 pixels
    annulus_apertures = CircularAnnulus((148,148), r_in=(20/0.18), r_out=(25/0.18))
    phot_table = aperture_photometry(image_data, annulus_apertures)

    # take the sum of the flux value within the annulus and divide by the total pixel count to get
    # average flux per pixel
    bkg_mean = phot_table['aperture_sum'] / annulus_apertures.area()

    # correction, subtract background from the total image
    imo = image_data - bkg_mean

    # mask the NaN values and then check if peak of distribution is near 0
    newimage_data = imo[~np.isnan(imo)]
    n,bins,patches = plt.hist(newimage_data)

    # create an aperture of radius 3.5'' and get the sum of what's inside, that
    # is the flux we're looking for
    aperture = CircularAperture((148.,148.), r=(3.5/0.18))
    photo_table = aperture_photometry(imo, aperture)

    flux = photo_table['aperture_sum'][0]

    return flux
```



Stacked Before Image



20” Image

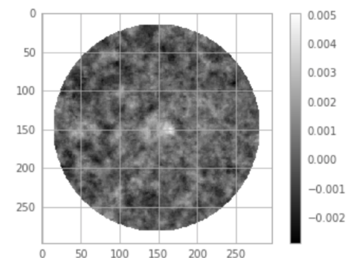
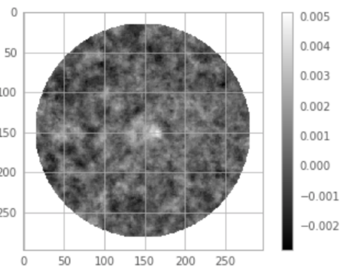
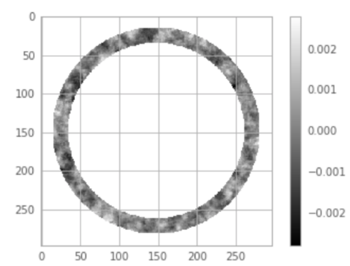


Image after subtracting average flux of the annulus



24” Image



Annulus

Bootstrapping

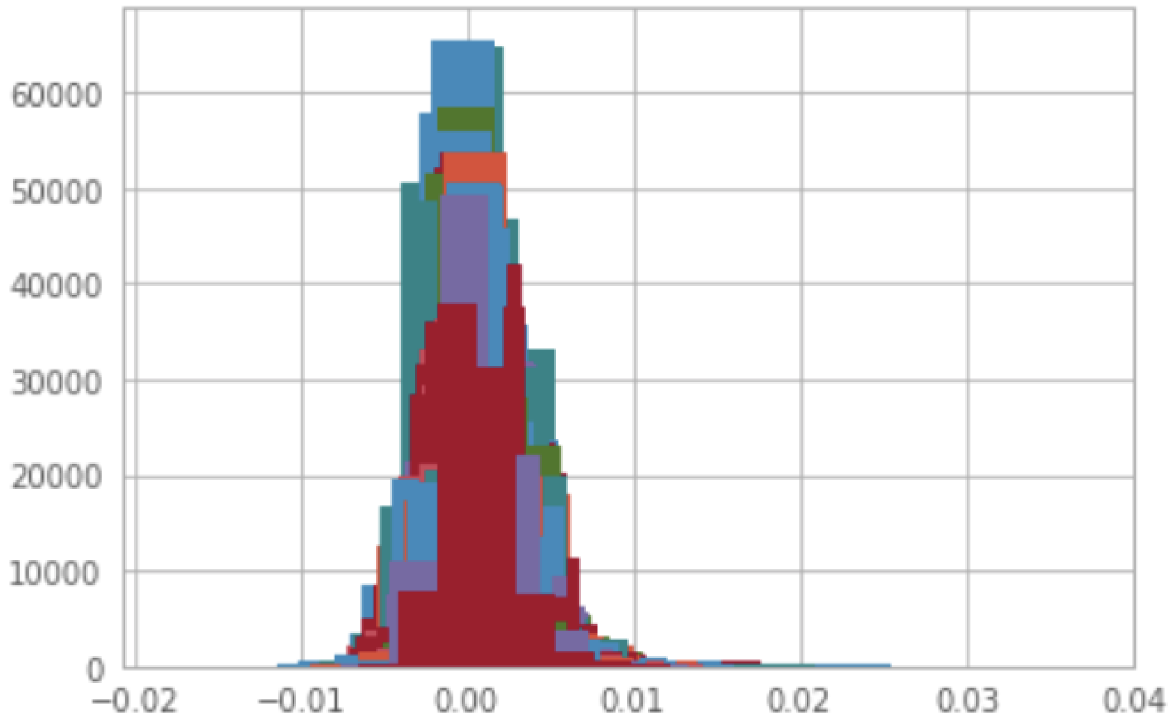
Stacking might generate some error which we’d expect, and errorbars will most likely come from bootstrapping. Where I need to resample the data with some deviation and do a certain amount of iterations. To be more pedantic, this method of bootstrapping, is a statistical technique for estimating certain quantities about a population (pool of data) by averaging estimates from multiple small data samples. That the samples are constructed by drawing pick (an observation) from a large data sample one at a time and returning them to the data sample after they have

been chosen. This allows for a given selection to be included in a small sample more than once. Essentially this approach is generally called sampling with replacement. This can be broken down to 5 steps:

1. Choose a number of bootstrap runs (iterations) to perform
2. Choose the size of the sample
3. While the size of the sample is less than the chosen size
   i Randomly select an observation from the dataset
   ii Add it to the sample
4. Throw the observation back to the pool
5. Repeat until sample is filled

```
def bootstrap(emptylist,matchedarray,N):
    for k in range(N):
        pick = np.random.randint(len(matchedarray),size=len(matchedarray))
        # creates an array of indexes equal to the length of the mass bin array

        seed = np.empty_like(matchedarray)
        for j in range(len(pick)):
            seed[j] = matchedarray[pick[j]]
        # creates my randomized array filled with the directory of all the galaxies

        cube = []
        for i in range(len(matchedarray)):
            im = fits.getdata(seed[i],ext=0,memmap=False)
            cube.append(im)
        cube = np.asarray(cube)
        # creates my stacked cube

        imagen=np.nanmedian(cube,axis=0)

        flux = aperture_phot(imagen)


        emptylist.append(flux)

    emptylist = np.asarray(emptylist)
    print('Bootstrap Done')
    return emptylist
```

The way I introduce bootstrapping to this example is by allowing for my stack within a certain mass bin to reuse a galaxy if the pick deems it so. By using `np.random.randit()` it returns an array the length of the mass bin but filled with indexes randomly picked from the original mass bin. Which we then allow for a sample `seed` to select from the original dataset and stack like we've previously done. We could see how the bootstrap works with how the flux values deviate through the multiple iterations and mass bins in the histogram below:



Log v. Linear

Normally errorbars ($\pm\delta y$) in a linear plot are lines extended equally above and below a point (y). But if you were to plot this on a logarithmic plot, then absolute error bars that were once symmetric in the linear $x-y$ plot, becomes asymmetric. The lower bar is longer than the upper one, which can give misleading information about the quantities they represent. Which can become a huge problem if the points vary by several orders of magnitudes. Errors in log are done by recognizing that what's being plotted isn't exactly y but a function $z = \log(y)$, so the error δz can be found to be:

$$\begin{aligned}
\delta z &= \delta[\log(y)] \\
\delta z \approx \mathrm{d}z &= \mathrm{d}[\log(y)] \\
&= \frac{1}{\log(10)}\frac{\mathrm{d}y}{y} \\
&= \frac{1}{2.303}\frac{\mathrm{d}y}{y} \\
\delta z &\approx 0.434\frac{\delta y}{y}
\end{aligned}$$

Where we see that the correct error, shown to be the *relative error*, will display correct errorbars on logarithmic plots, shown below.

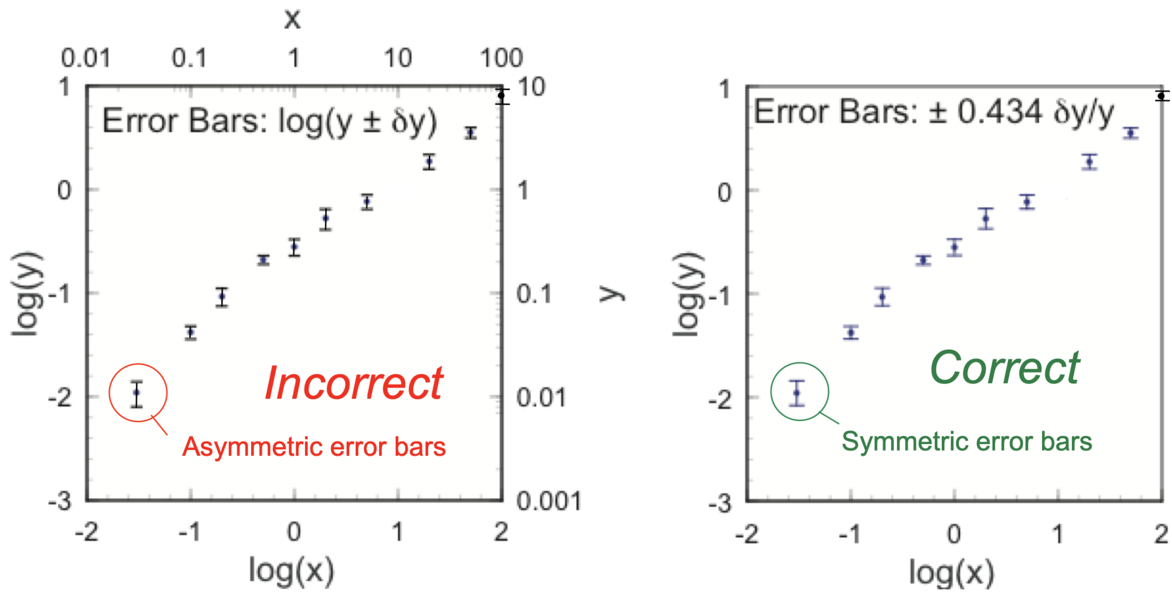


Figure 1: *Courtesy* of Professor Eric M. Stuve, a more in depth discussion and explanation about this can be found at Estimating and Plotting Logarithmic Error Bars

Relevance

You should now be able to stack your own FITS files by implementing what I've done to some degree on your own data. As well as being able to bootstrap at well which is important for just about anything, depending on what you're doing. Especially if you have a small sample size and you want to gain the variance or any statistical analysis you need. Introduction to the Bootstrap Method gives a much more detailed explanation of bootstrapping and its benefits.