Lab 4

Ondrea Robinson

November 3, 2019

In this lab we're going to look at 'real' data and finding the background from it. Before we begin, here's the Python implementation we'll be using:

```
#First these are the packages we'll be using
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from matplotlib import pylab
from pylab import *
import scipy
from scipy import stats
import h5py

#This is the file with our data that we save to an array
hf = h5py.File('gammaray_lab4.h5', 'r')
data = np.array(hf.get('data'))
hf.close()
```

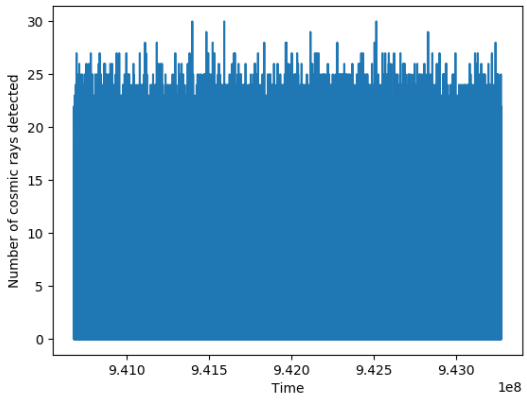
1 Problem 1

In this problem we are looking at the data from a gamma-ray satellite orbiting in low Earth orbit. It takes a reading of the number of particles detected every 100 milliseconds, and is in an approximately 90 minute orbit. While it is looking for gamma-ray bursts, virtually all of the particles detected are background cosmic rays.

1.1 1

First let's just see what our data looks like. We want to know how the data changes with respect to time, location (longitude) and solar phase. The variable we're looking at is our 'signal' variable counts. Each of the 4 columns of our data array corresponds to a variable; from index 0 to 3 we have time (gps seconds), solar phase (degrees), longitude (degrees) and finally counts. Because the data set is so large, when we graph all counts with respect to time we just see a large block:

```
time = data[0,:]
counts = data[3,:]

figure(0)
plt.plot(time, counts)
plt.xlabel('Time')
plt.ylabel('Number of cosmic rays detected')
```



We want to look closer and see better how the data changes in time so we can choose a smaller x range (time range) and create graphs for several time intervals. For these next graphs we're going to look at 15000 time steps each and we'll do 3 intervals of this starting from time step 'zero', the first time step in our array. Here's the code:

```
interval = 25000
for i in range(3):
    figure(i+1)
    plt.xlim((data[0,0] + i*interval), (data[0,0] + interval + i*interval))
    plt.plot(time, counts)
    plt.xlabel('Time')
    plt.ylabel('Number of cosmic rays detected')
    plt.show()
```

From this we get three sequential plots:

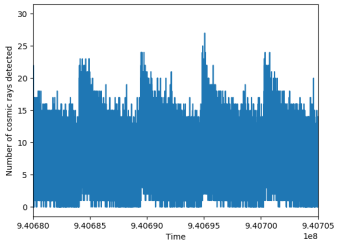


Figure 1: First interval

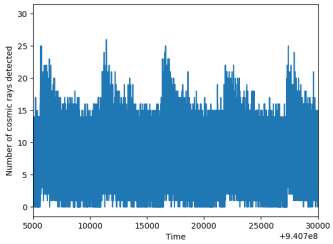


Figure 2: Second interval

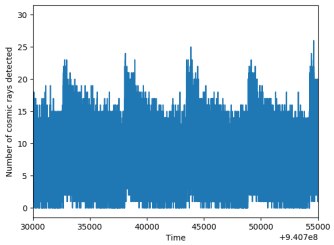


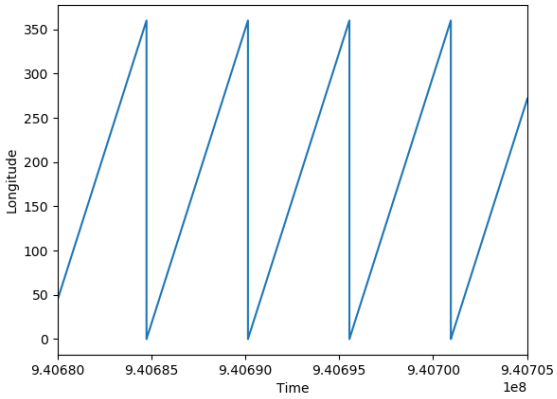
Figure 3: Third interval

In all three intervals we see periodicity with a period of around 5000 seconds. We know that it takes the satellite 90 minutes to orbit. This corresponds to $60*90 = 5400$ seconds per orbit. This is extremely similar to the period of our data fluctuations so that is where we'll start investigating. To create our background pdf we're going to have to incorporate this periodicity into our function to be accurate. This means we'll need to determine the time dependency of our background by comparing to other factors. We can't tell whether there are signals yet but after looking through about 15 graphs, most seem rather regular and periodic without strange or particularly noticeable outliers.

1.2 2

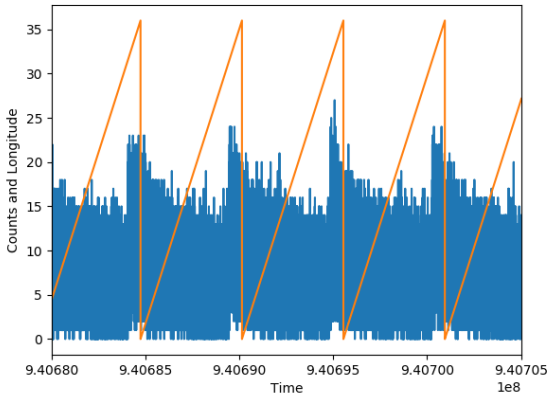
Let's investigate which factor will affect this the most. As we discussed, the factor that seems to match the closest is the time of the orbit to the period of the spike in counts. We want to see if there's any correlation so let's plot the longitude over time and see what that looks like then compare that graph against the graph of the counts. First longitude vs time:

```
longitude = data[2,:]
figure(5)
plt.xlim(data[0,0], (data[0,0]+ interval))
plt.plot(time, longitude)
plt.xlabel('Time')
plt.ylabel('Longitude')
```



We can see a similar periodicity with the period of about 5400 seconds like we hoped. Now let's see if it matches our counts:

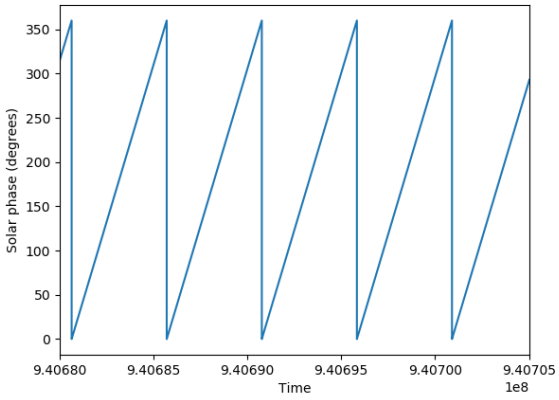
```
longitude = data[2,:]
figure(6)
plt.xlim(data[0,0], (data[0,0]+ 15000))
plt.plot(time, longitude/10, label='Longitude')
plt.plot(time, counts, label='Counts')
plt.show()
```



This graph deserves some explaining. Here the amplitude of the longitude function (orange) is divided by 10 just so it appears visibly next to the spikes of the counts plot (blue). This doesn't affect the width of the periods however. We can now see there's a clear periodicity of the measurement associated with the longitude of the satellite. To build our pdf we'll need to include this time dependency.

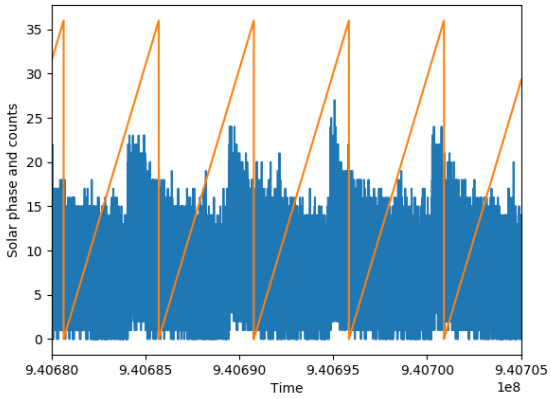
Let's also see how the solar phase affects the counts. First let's graph he phase with respect to time just to see how it changes:

```
phase = data[1,:]
interval = 25000

figure(10)
plt.xlim(data[0,0], (data[0,0]+ interval))
plt.plot(time, phase)
plt.xlabel('Time')
plt.ylabel('Solar phase (degrees)')
plt.show()
```



We also see a periodicity that has a similar period as our counts. Let's compare again to see whether there's a correlation.

```
figure(11)
plt.xlim(data[0,0], (data[0,0]+ interval))
plt.plot(time, counts)
plt.plot(time, phase/10)
plt.xlabel('Time')
plt.ylabel('Solar phase and counts')
plt.show()
```



We can see that the solar phase and the counts are out of phase and do not coincide as nicely as the longitude. For building our pdf we're going to focus on the longitude measurements rather than the solar phase.

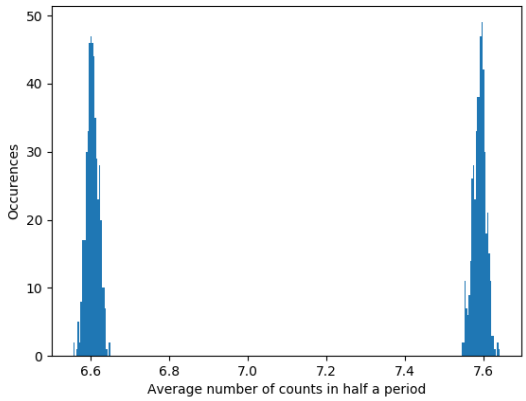
1.3 3

Now the hard part. We need to design a pdf with a built in time dependency. Without knowing about the periodicity of our data, we would normally assume a Poisson distribution because we're looking at occurrences in a time interval. But because of these regular fluctuations, the background will have to be modified to incorporate an average that changes with respect to time. We know the average number of counts spikes around 360 degrees and we return to 360 degrees every 5400 seconds.

We decided to make a pdf based on the average value of the counts for each $\frac{1}{16}$ of a period, over all periods. First let's go over this process for a half a period. What we want to do is count all of the counts for the first half of our period (27000 seconds) then average them. The average for one half of one period is added to an array. We then we do the second half of the period and add the average to our array. We do this $480 * 2 = 960$ times because there are 480 periods in our data set.

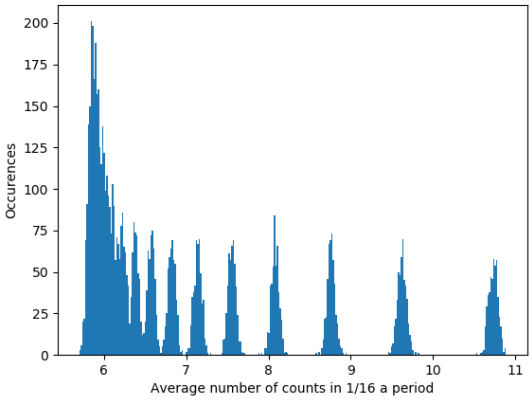
```
avgarray = ([])
avgover = 27000 # looking at half our interval (54000/2)
times = 2

i=0
while i < 480*times:
    sumvar=0
    for j in range(i*avgover, i*avgover + avgover):
        sumvar = sumvar + counts[j]
```

```
        avg1orbit = sumvar/avgover
        avgarray = np.append(avgarray, avg1orbit)
        i = i + 1

    figure(7)
    plt.hist(avgarray, bins=300)
    plt.xlabel('Average number of counts in half a period')
    plt.ylabel('Occurences')
```



Now looking at a similar distribution for 16 intervals:

```
    avgarray = ([])
    times = 16
    avgover = 3375 # looking at half our interval (54000/2)

    i=0
    while i < 480*times:
        sumvar=0
        for j in range(i*avgover, i*avgover + avgover):
            sumvar = sumvar + counts[j]

        avg1orbit = sumvar/avgover
        avgarray = np.append(avgarray, avg1orbit)
        i = i + 1

    figure(8)
    plt.hist(avgarray, bins=300)
    plt.xlabel('Average number of counts in 1/16 a period')
    plt.ylabel('Occurences')
    plt.show()
```

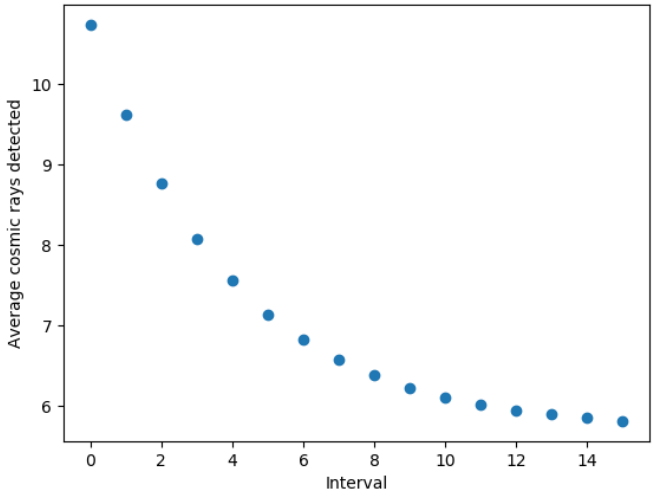


Now we have a distribution for each 1/16 of a period. Note that for each 1/16 there is a Poisson distribution around some average value of counts. These average values are not ordered sequentially so we need to find how the average changes over the course of a period (the time dependence). We use the following code to find the average value of counts for each 1/16 (i.e. 1/16, 2/16,...,1) for all 480 periods. We can then create a plot of these averages in the correct order to see how the average changes over time. Here's the code:

```
avgover = 3375
allintervals = np.array([])
def batchArrays(interval):
    low = interval*avgover
    high = low + avgover
    batch = np.array([])
    while high <= 25920000:
        sumvar=0
        for k in range(low, high):
            sumvar = sumvar + counts[k]
        sumvar = sumvar/avgover
        batch = np.append(batch, sumvar)
        low = low + 54000
        high = high + 54000
    return(batch)

averages = np.array([])
for i in range(times):
    average = np.mean(batchArrays(i))
    averages = np.append(averages, average)
print(averages)

lastfour = averages[12:16]
```

```
editedavg = np.append(lastfour, averages)
editedavg = editedavg[0:16]
```

Here's the plot of the average counts with respect to interval of the period:



We can easily fit a function to this plot. This function will give us the mean with respect to time within a period. We can plug that mean into our Poisson probability mass function so that we get a distribution that changes with respect to time. The function for a Poisson distribution is given by:

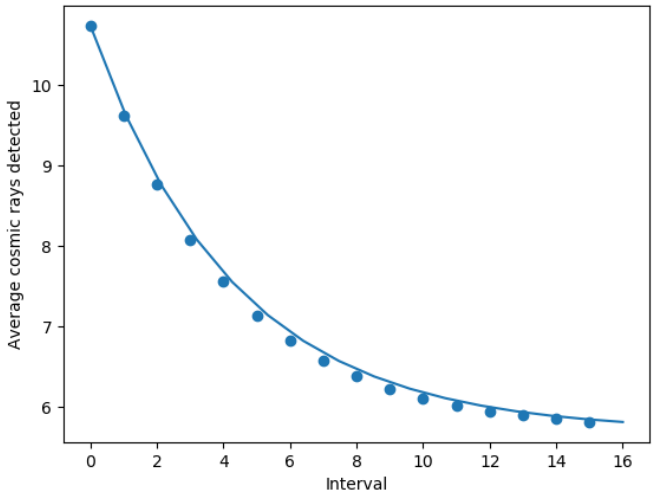
$$P(k) = \frac{\lambda^k e^{-\lambda}}{k!}$$

This is where λ is the average and k is the number of events given to the function. P returns the probability of k events happening in an interval. We can apply a best fit to our scatter plot above and we'll get an exponential decay function:

```
def exponential(x, a, k, b):
    return a*np.exp(x*k) + b

x_array = np.linspace(0, 16, 16)
y_array = editedavg

popt, pcov = scipy.optimize.curve_fit(exponential, x_array,
y_array, p0=[1,-0.5, 1])

figure()
x = np.arange(0, 16)
plt.scatter(x, editedavg)
```

```
plt.plot(x_array, exponential(x_array, *popt))
plt.xlabel('Interval')
plt.ylabel('Average cosmic rays detected')
plt.show()
```

This function fits our data pretty well:



The full function is given by

$$\lambda(t) = 5.04e^{-0.234t} + 5.7$$

Now if we're given a measurement value for a certain time we can compare it to the correct distribution. All we need to know is at what time in the period the measurement was taken and we can find the correct average value for the Poisson. We know that at the first data point we're about a quarter of a period from the start of the cycle. So to find out what part of the period we're in we can just take our raw time T (associated with a measurement), subtract 13500 (a quarter period)

1.4

Let's test out our distribution. We want to know what the threshold is for a 100 millisecond gamma ray burst is at different times. That's the measurement in 1 time step. So if we're looking at the beginning of a period, then the threshold will be higher and if we're looking toward the end, it will be lower. Here's the code we'll use:

```
mean = exponential(t, *popt) #t is in units of time within the period
prob = 1/(3.5e6)
print(stats.poisson.ppf(1-prob, mean))
```

For t=0 in the period, the threshold is 31, for t=1/2period the threshold is 23 and for the last 16 of the period the threshold is 21 (end of period). This is all for a 5σ detection in one time step.

2 Problem 2

In this problem we are going to look at a stack of telescope images (again simulated). We have 10 images and we will be looking for the faintest stars.

2.1

first let's look open and save our images then display them using the code:

```
hf1 = h5py.File('images.h5', 'r')
imdata = np.array(hf1.get('image1'))
imstack = np.array(hf1.get('imagestack'))

for i in range(10):
    figure(i+1)
    plt.imshow(imstack[:,:,i])
    plt.show()
```

We see 10 different images. We're going to show 2 rather than all 10 since they all look quite similar. Their differences aren't noticeable to the eye at least.

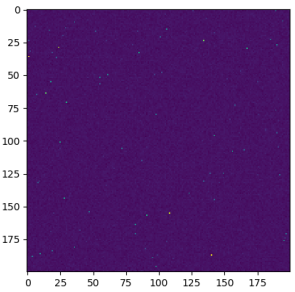


Figure 4

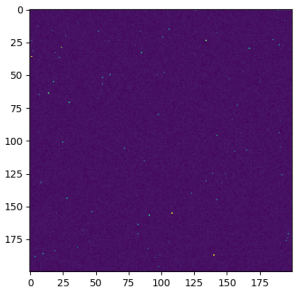


Figure 5

2.2

Now let's discuss. The information we have is brightness for each pixel. The bright pixels don't appear to have any spatial or time dependence. If they are changing as a function of time the change isn't easily visible. The pixels spatial distribution also appears to have no noticeable affect on brightness; the bright pixels appear to be normally distributed throughout the square. For these factors we can assume that our distribution is random. What makes finding our

pdf difficult however is that there is most likely signal contamination in our images. We're looking for faint stars. If these images include faint stars, our background pdf based on these images will have a high likelihood of producing faint stars. For this reason we have to manipulate the data so that we're comparing our signal to the distribution of all the normal stars.

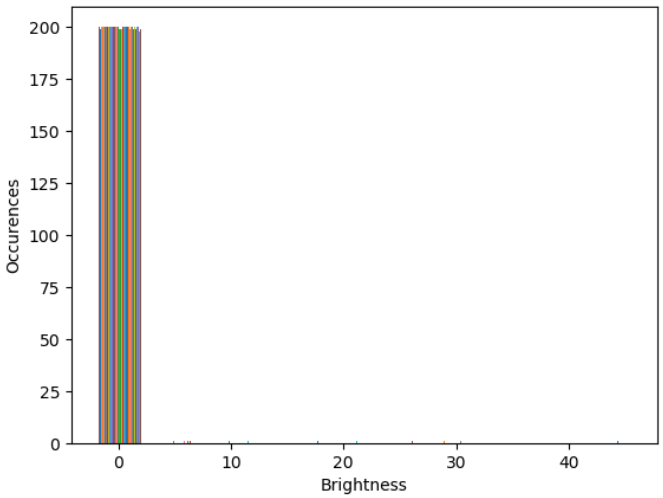
First let's define what a 'faint' star is. We'll assume that star brightness is normally distributed around some average. The average brightness of the entire image will most likely be close to zero because the pixels with stars are fewer in between than the pixels without. Let's say we set some minimum of star brightness to be considered 'normal'. We want to collect all the pixels that are above this threshold then average them to create our distribution for regular brightness.

So how we get our background pdf will have to be from averaged brightness per pixel. This isn't perfect since the faint stars we suspect are in the data will throw off our average but it will help decrease their affect. We're going to find the brightness distribution for each pixel for each image, stack our 10 distributions then average by dividing by 10.

2.3

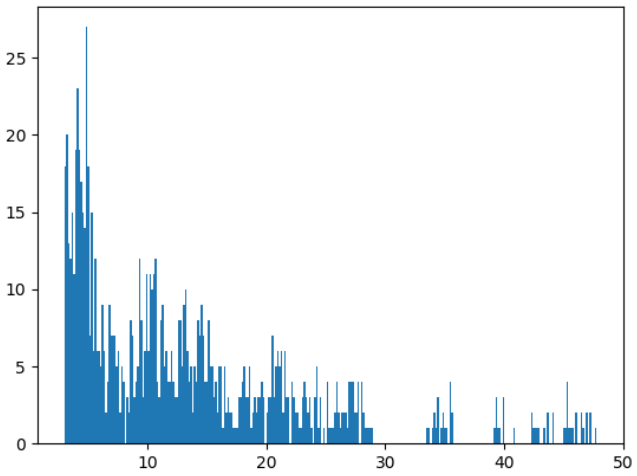
In order to set our threshold let's manipulate the data to see the minimum and maximum levels of brightness we have. Here's a histogram of the raw data for one image:

```
figure()
plt.hist(imstack[:,:,0])
plt.xlabel('Brightness')
plt.ylabel('Occurences')
plt.show()
```

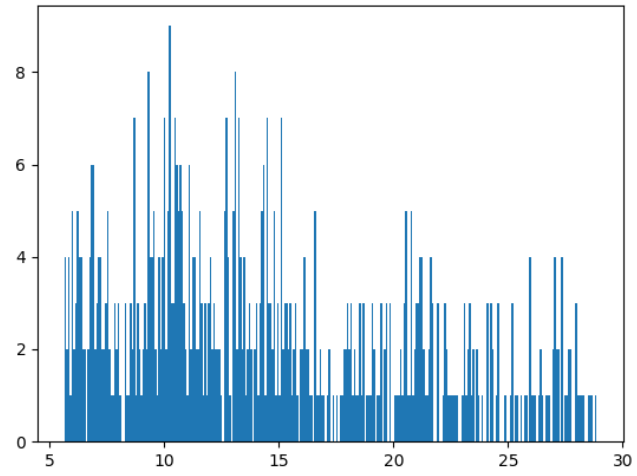


As expected most of the data is clustered around zero and the lower values of brightness. If you look closely however, there are little bumps of brightness further out that happen less frequently. These are our stars. The brightest is out at around 44 and the lowest is somewhere around 4.5. Here's a histogram of the brightness of all the stars from all the images:

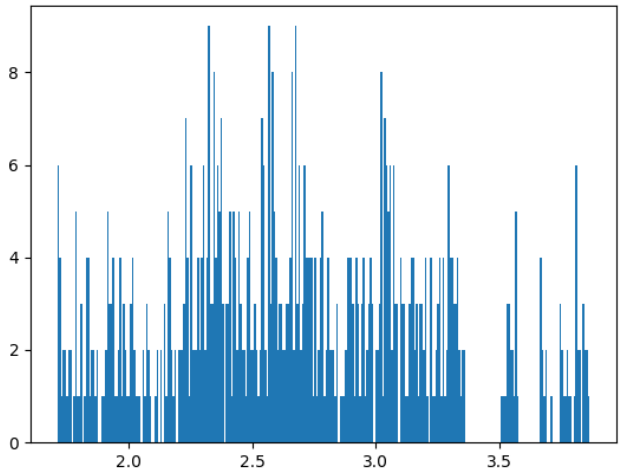
```
allpix = imstack[:,:,0].flatten()
for i in range(9):
    pixels = imstack[:,:,i+1].flatten()
    allpix = np.append(allpix, pixels)

starbright = np.array([])
for i in range(len(allpix)):
    if 3 < allpix[i]:
        starbright = np.append(starbright, allpix[i])

figure()
x = np.linspace(5, 35, len(starbright))
plt.hist(starbright, bins=300)
plt.show()
```



The mean of the brightness is 13.32. Now the question is what do we consider 'faint'. If we're looking for dim stars, there are quite a few of those all clustered around brightness 5. If we're looking for stars even fainter than those, we could just make a distribution around the mean we have. Again the threshold is a decision. Let's consider the stars around 4 or 5 to be faint. Since these are the faint stars we'll have to eliminate them from the background to cut down signal contamination. Let's use the same code but for a different threshold and make a distribution of the mean. After changing the threshold the mean of our data is 16.78 and the plot looks like:

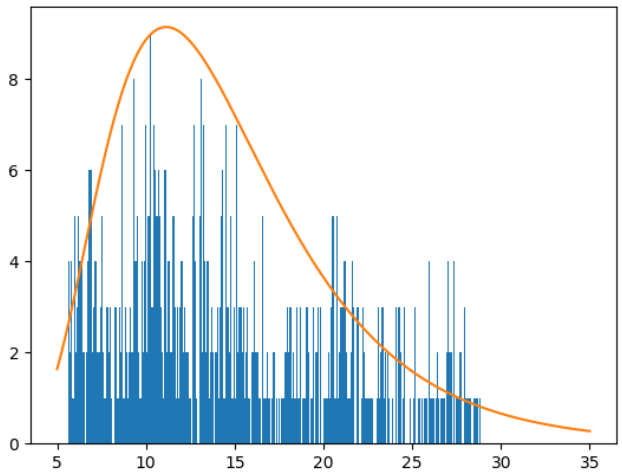


Looking at the shape of our distribution it appears more like a lognormal distribution than a normal distribution. If we plot the log of our function we get a graph that looks like:



Since the log of the distribution looks like a normal distribution we can use a lognormal distribution to model the function. Our pdf looks like:

```
figure()
x = np.linspace(5, 35, len(starbright))
plt.hist((starbright), bins=300)
plt.plot(x, 121*stats.lognorm.pdf(x, 0.431735892678718, loc=0,
scale=np.exp(2.5968)))
plt.show()
```



We can now use this background distribution to compare against for faint stars.

2.4

My partner and I were looking at the same data but for different signals. My partner had a different definition of a signal so their background pdf had to change to account for that. We're still looking at the same data but we want to to compare two different measurement values to it. To do that we need to distribute the data against the type of measurement we're taking. For each definition of a signal you need to take a background of the signal-like measurements you're comparing to. The signal-like measurements are found in the same data but are evaluated differently.