KURVS Code Write-Up

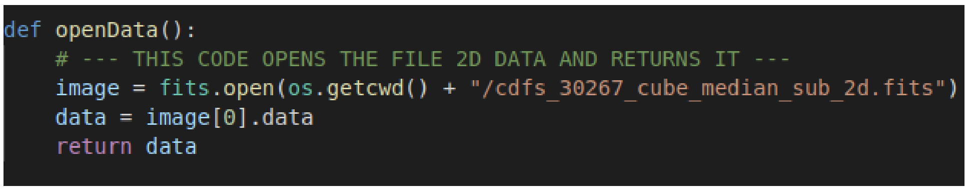
By Thomas Smail and Gautam Chauhan

Introduction

The aim of this project was to estimate the amount of dark matter in a galaxy. This was done using a piece of code that we wrote. That piece of code is what will be broken down and explained in this document.

KURVS Single Slice

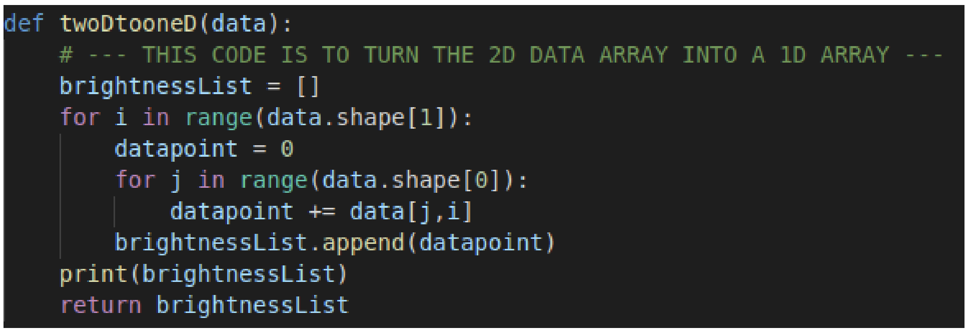
1.1.openData()



```python
def openData():
    # --- THIS CODE OPENS THE FILE 2D DATA AND RETURNS IT ---
    image = fits.open(os.getcwd() + "/cdfs_30267_cube_median_sub_2d.fits")
    data = image[0].data
    return data
```

- **openData()** is a simplistic function that passes the 2D fits file back to the code in a way that it can comprehend. data is passed back to the function as an array of values that looks like this: *[[2.47653322e-16 8.05208618e-16 -1.17465134e-15 ... -2.14149344e-15 -1.39929389e-15 -8.83203459e-16] [4.15455050e-16 -7.85519407e-16 -3.34919770e-16 ... 9.07941316e-17 -8.86547374e-16 3.28284145e-15] ...]* As can be seen the data is split into rows where each value is a pixel's intensity.

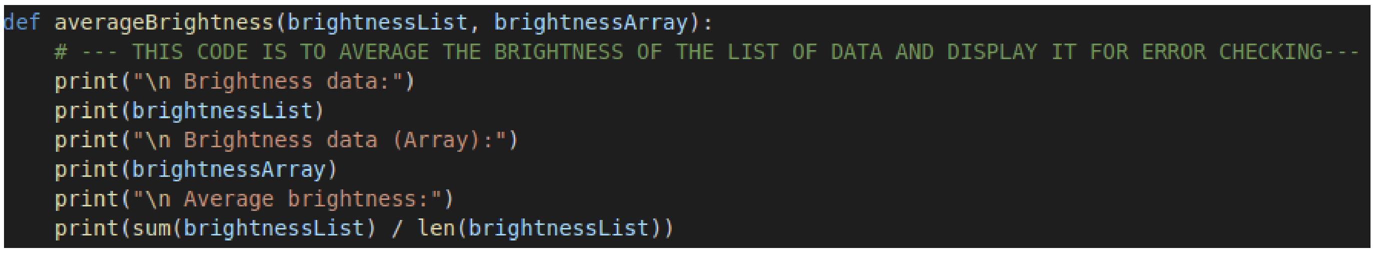
1.2.twoDtooneD()



```python
def twoDtooneD(data):
    # --- THIS CODE IS TO TURN THE 2D DATA ARRAY INTO A 1D ARRAY ---
    brightnessList = []
    for i in range(data.shape[1]):
        datapoint = 0
        for j in range(data.shape[0]):
            datapoint += data[j,i]
        brightnessList.append(datapoint)
    print(brightnessList)
    return brightnessList
```

- **twoDtooneD()** takes the passed data and converts it into a 1-dimensional list of all the pixels in the entire image, it does this by cycling through each column on each row and appending them to a new list which it then returns. This was an important step as it allowed us to manipulate the data with less complexity.

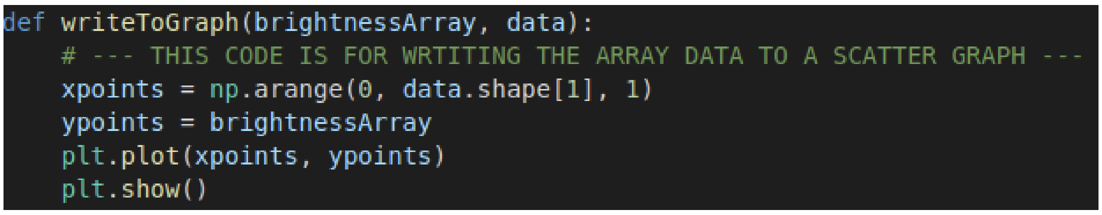
1.3.averageBrightness()



```
def averageBrightness(brightnessList, brightnessArray):
    # --- THIS CODE IS TO AVERAGE THE BRIGHTNESS OF THE LIST OF DATA AND DISPLAY IT FOR ERROR CHECKING---
    print("\n Brightness data:")
    print(brightnessList)
    print("\n Brightness data (Array):")
    print(brightnessArray)
    print("\n Average brightness:")
    print(sum(brightnessList) / len(brightnessList))
```

- **averageBrightness()** this function enables error checking in the data as it prints out the two different formats of the 1D spectrum which we had issues with when building the code.
- It also finds the average intensity of the light, so we can quickly check if something has gone wrong. This was particularly useful as we were using DS9 and so could check our expected pixel values with what we were getting from the data.

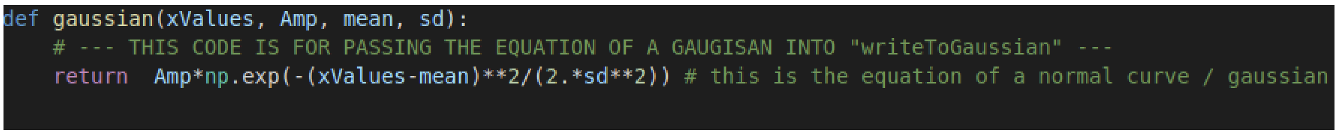
1.4.writeToGraph()



```
def writeToGraph(brightnessArray, data):
    # --- THIS CODE IS FOR WRTITING THE ARRAY DATA TO A SCATTER GRAPH ---
    xpoints = np.arange(0, data.shape[1], 1)
    ypoints = brightnessArray
    plt.plot(xpoints, ypoints)
    plt.show()
```

- **writeToGraph()** is a function that plots the entirety of the **brightnessArray** on the y and the corresponding indexes on the x axis, this again is for error checking and was put in as the program was being built to make sure that the data made sense and was what we predicted we were getting.

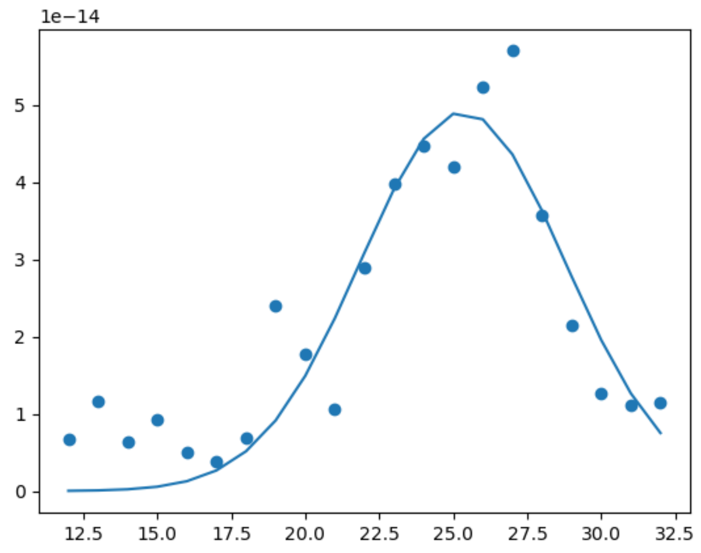
1.5.gaussian()

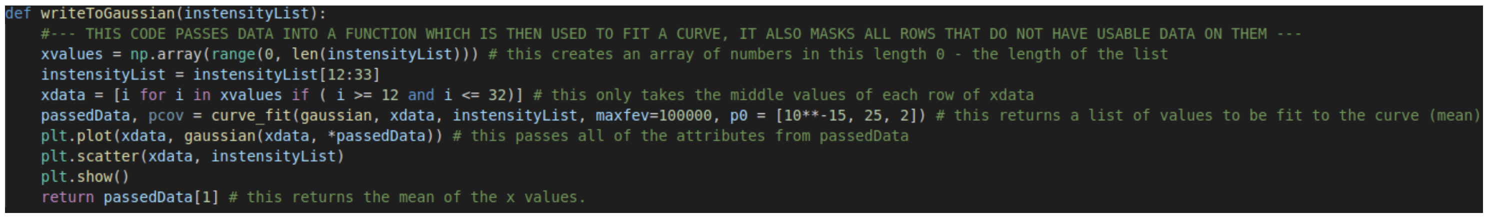


```
def gaussian(xValues, Amp, mean, sd):
    # --- THIS CODE IS FOR PASSING THE EQUATION OF A GAUGISAN INTO "writeToGaussian" ---
    return  Amp*np.exp(-(xValues-mean)**2/(2.*sd**2)) # this is the equation of a normal curve / gaussian
```

- **gaussian()** function is only in the code as it is required by **curve_fit**, in the next function, to plot a gaussian for our points. It takes the x values, amplitude, mean and standard deviation and returns an equation which is then plotted on the y axis in the next function.
- The amplitude corresponds to the height of the curves peak, the mean is the position of the middle of this peak and the standard deviation determines how wide the curve should be.

1.6.writeToGaussian()

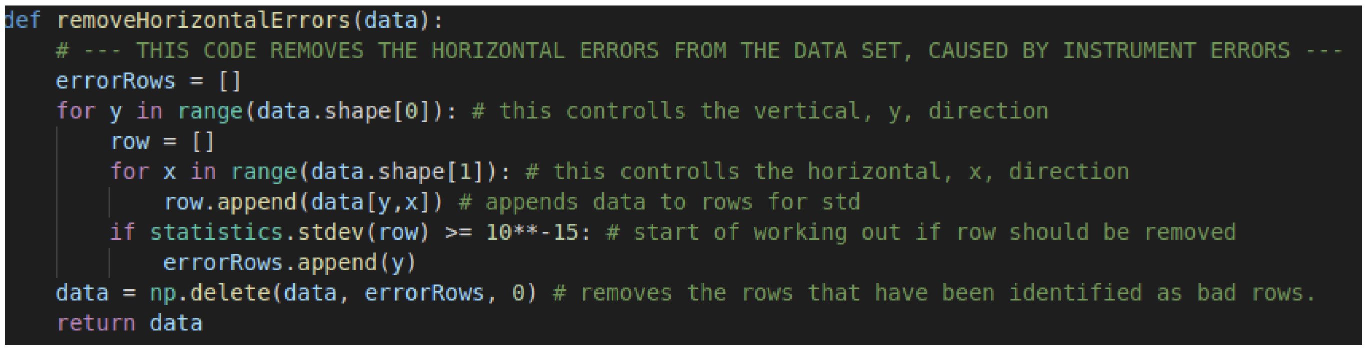




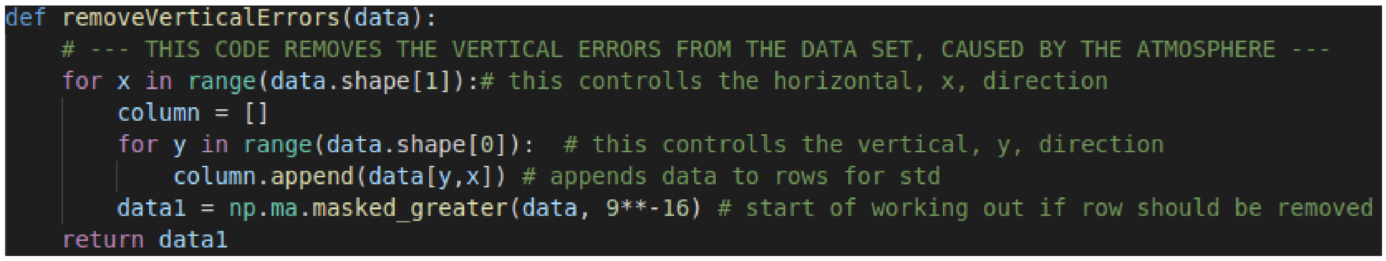
```
def writeToGaussian(instensityList):
    #--- THIS CODE PASSES DATA INTO A FUNCTION WHICH IS THEN USED TO FIT A CURVE, IT ALSO MASKS ALL ROWS THAT DO NOT HAVE USABLE DATA ON THEM ---
    xvalues = np.array(range(0, len(instensityList))) # this creates an array of numbers in this length 0 - the length of the list
    instensityList = instensityList[12:33]
    xdata = [i for i in xvalues if ( i >= 12 and i <= 32)] # this only takes the middle values of each row of xdata
    passedData, pcov = curve_fit(gaussian, xdata, instensityList, maxfev=100000, p0 = [10**-15, 25, 2]) # this returns a list of values to be fit to the curve (mean)
    plt.plot(xdata, gaussian(xdata, *passedData)) # this passes all of the attributes from passedData
    plt.scatter(xdata, instensityList)
    plt.show()
    return passedData[1] # this returns the mean of the x values.
```

- **writeToGaussian()** takes the **brightnessLis**t (**intensityList**) and filters out the parts of the galaxy that are useful and the parts that are not. For doing this we have taken the middle of the galaxy which is between index 12 – 32 in pixel values, we did attempt to do this with an automated function but was not able to achieve the same level of cleanness in the data.
- Once the data is cleaned (masked) then the data is passed into the gaussian function where it is then passed back and plotted on a graph along with another scatter graph. As can be seen a nice curve is produced by the function.

1.7.a.removeHorizontalErrors() - 1.7.b.removeVerticalErrors()



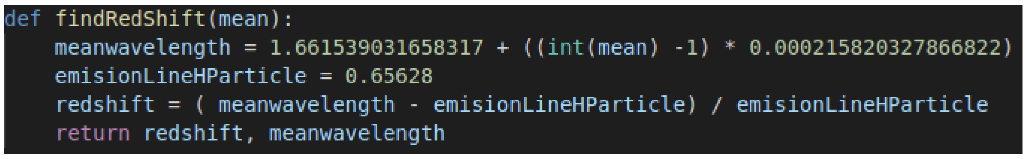
```
def removeHorizontalErrors(data):
    # --- THIS CODE REMOVES THE HORIZONTAL ERRORS FROM THE DATA SET, CAUSED BY INSTRUMENT ERRORS ---
    errorRows = []
    for y in range(data.shape[0]): # this controlls the vertical, y, direction
        row = []
        for x in range(data.shape[1]): # this controlls the horizontal, x, direction
            row.append(data[y,x]) # appends data to rows for std
        if statistics.stdev(row) >= 10**-15: # start of working out if row should be removed
            errorRows.append(y)
    data = np.delete(data, errorRows, 0) # removes the rows that have been identified as bad rows.
    return data
```



```
def removeVerticalErrors(data):
    # --- THIS CODE REMOVES THE VERTICAL ERRORS FROM THE DATA SET, CAUSED BY THE ATMOSPHERE ---
    for x in range(data.shape[1]):# this controlls the horizontal, x, direction
        column = []
        for y in range(data.shape[0]):  # this controlls the vertical, y, direction
            column.append(data[y,x]) # appends data to rows for std
        data1 = np.ma.masked_greater(data, 9**-16) # start of working out if row should be removed
    return data1
```

- **removeHorizontalErrors()** is replaced in the final code for multiple slices, however it is for removing errors caused by the telescope. It finds the standard deviation of each of the rows and if it is above a certain value then we can say it is anomalous and thus it is deleted.
- A similar method is used for **removeVerticalErrors()** however the data is masked instead.

1.8.findRedShift()



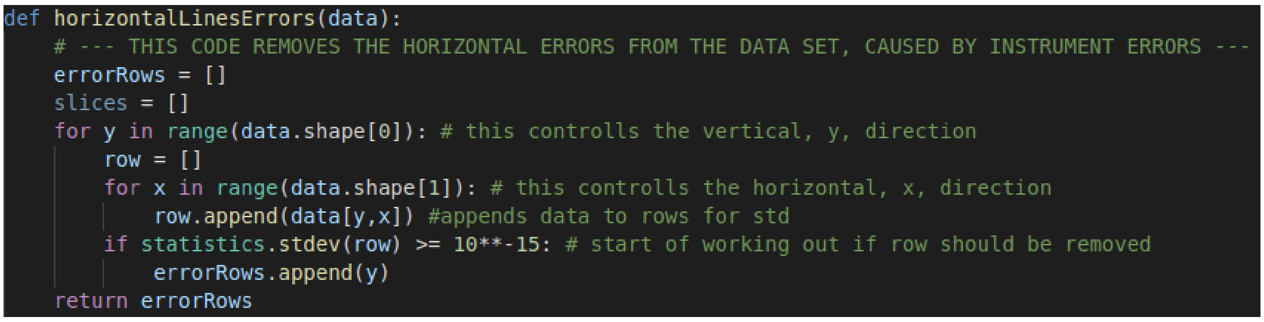
```python
def findRedShift(mean):
    meanwavelength = 1.661539031658317 + ((int(mean) -1) * 0.000215820327866822)
    emisionLineHParticle = 0.65628
    redshift = ( meanwavelength - emisionLineHParticle) / emisionLineHParticle
    return redshift, meanwavelength
```

- **findRedShift()** this finds the redshift of each slice, which we will make later, it also finds the **meanwavelength** (l**ambda_peak_values**) of each slice. It does this by using predefined values for the emission wavelength of a Hydrogen Particle as well as the wavelengths of the peaks of each slice.

KURVS Multi Slice

We started a new script for our multi slice imaging, however in retrospect this was somewhat unnecessary as it could have all been contained in one script. The next functions are all in a different script and have the old functions passed to it when necessary.

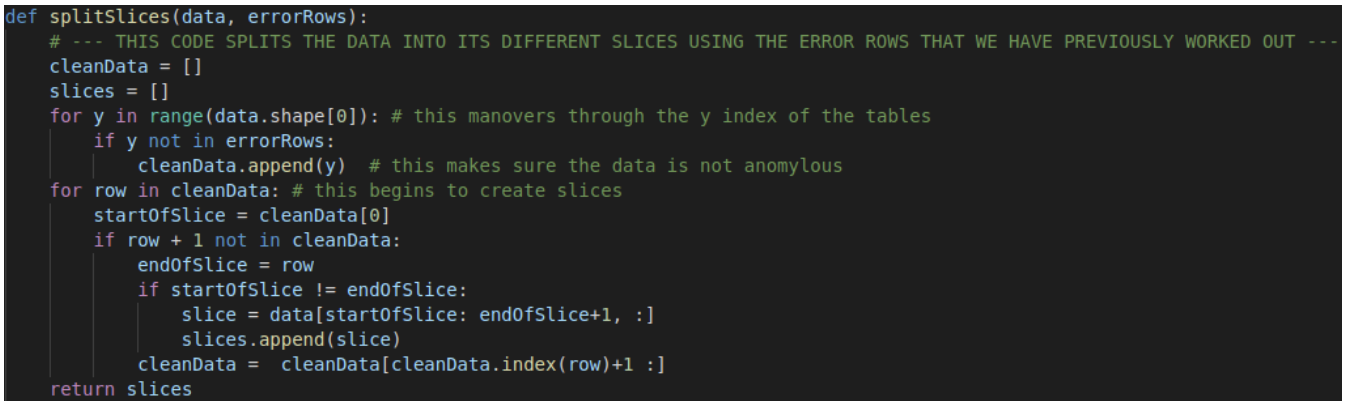
2.1.horizontalLinesErrors()



```python
def horizontalLinesErrors(data):
    # --- THIS CODE REMOVES THE HORIZONTAL ERRORS FROM THE DATA SET, CAUSED BY INSTRUMENT ERRORS ---
    errorRows = []
    slices = []
    for y in range(data.shape[0]): # this controlls the vertical, y, direction
        row = []
        for x in range(data.shape[1]): # this controlls the horizontal, x, direction
            row.append(data[y,x]) #appends data to rows for std
        if statistics.stdev(row) >= 10**-15: # start of working out if row should be removed
            errorRows.append(y)
    return errorRows
```

- **horizontalLinesErrors()** is similar to the previous function **removeHorizontalErrors()** however instead of deleting the erroneous data it returns the lines that have errors on them instead. By doing this, slices can be formed in the next function.

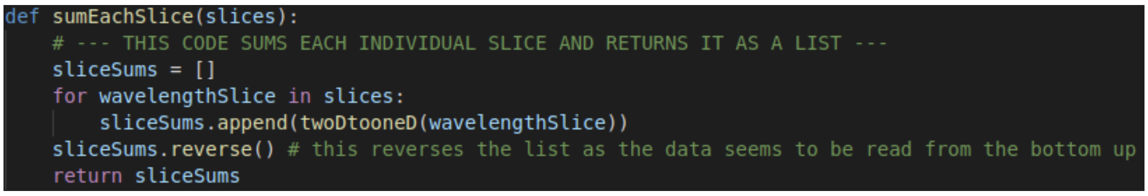
2.2.splitSlices()



```python
def splitSlices(data, errorRows):
    # --- THIS CODE SPLITS THE DATA INTO ITS DIFFERENT SLICES USING THE ERROR ROWS THAT WE HAVE PREVIOUSLY WORKED OUT ---
    cleanData = []
    slices = []
    for y in range(data.shape[0]): # this manovers through the y index of the tables
        if y not in errorRows:
            cleanData.append(y)  # this makes sure the data is not anomylous
    for row in cleanData: # this begins to create slices
        startOfSlice = cleanData[0]
        if row + 1 not in cleanData:
            endOfSlice = row
            if startOfSlice != endOfSlice:
                slice = data[startOfSlice: endOfSlice+1, :]
                slices.append(slice)
            cleanData =  cleanData[cleanData.index(row)+1 :]
    return slices
```

- **splitSlices()** takes the data as well as the **errorRows** and creates slices of the data using the **errorRows** as dividers between the slices. It checks if each of the rows is in **errorRows** and then appends the index of the good rows to a list, then it splits the data up into different lists which it inserts into **slices**. The **slices** list is then returned.

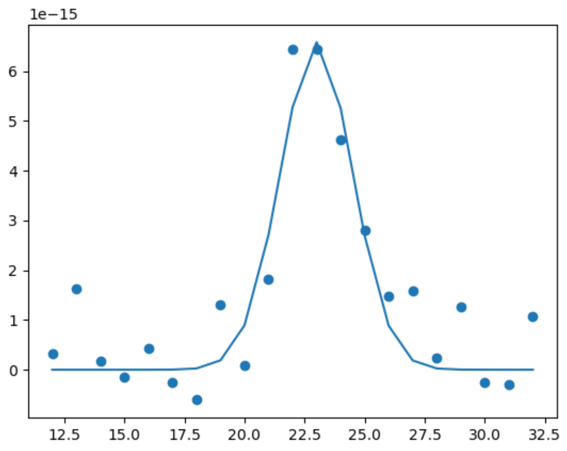
2.3.sumEachSlice()

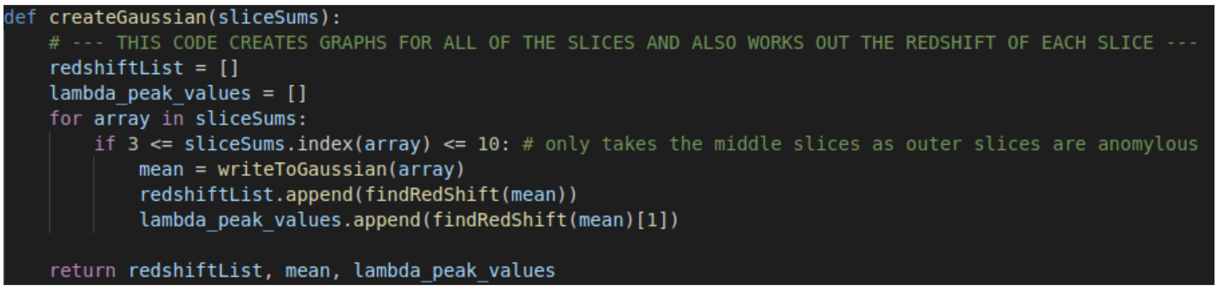


```python
def sumEachSlice(slices):
    # --- THIS CODE SUMS EACH INDIVIDUAL SLICE AND RETURNS IT AS A LIST ---
    sliceSums = []
    for wavelengthSlice in slices:
        sliceSums.append(twoDtooneD(wavelengthSlice))
    sliceSums.reverse() # this reverses the list as the data seems to be read from the bottom up
    return sliceSums
```

- **sumEachSlice()** takes the **slices** and for the number of **slices** in **slices** applies **twoDtooneD()** and then appends this to **sliceSums**. This is then reversed as we realised that the data was being read from bottom to top by **openData()**.

2.4.createGaussian()



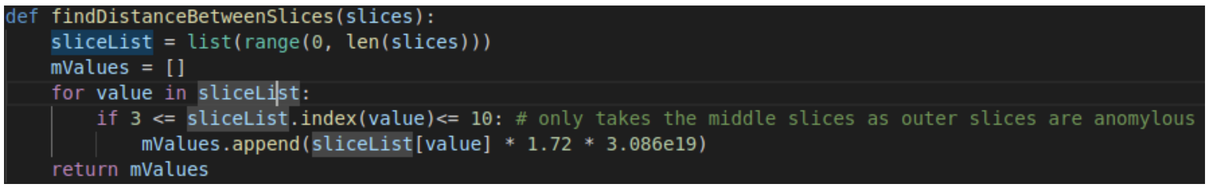


```python
def createGaussian(sliceSums):
    # --- THIS CODE CREATES GRAPHS FOR ALL OF THE SLICES AND ALSO WORKS OUT THE REDSHIFT OF EACH SLICE ---
    redshiftList = []
    lambda_peak_values = []
    for array in sliceSums:
        if 3 <= sliceSums.index(array) <= 10: # only takes the middle slices as outer slices are anomylous
            mean = writeToGaussian(array)
            redshiftList.append(findRedShift(mean))
            lambda_peak_values.append(findRedShift(mean)[1])

    return redshiftList, mean, lambda_peak_values
```

- **createGaussian()** cycles through **sliceSums** and creates a gaussian for each one using the **writeToGaussian()** function, whilst it is doing this it also finds the redshift and **lambda_peak_values** for each of the slices. It then returns these values so they can be used later in calculations. The graph above is an example of the code plotting the function.

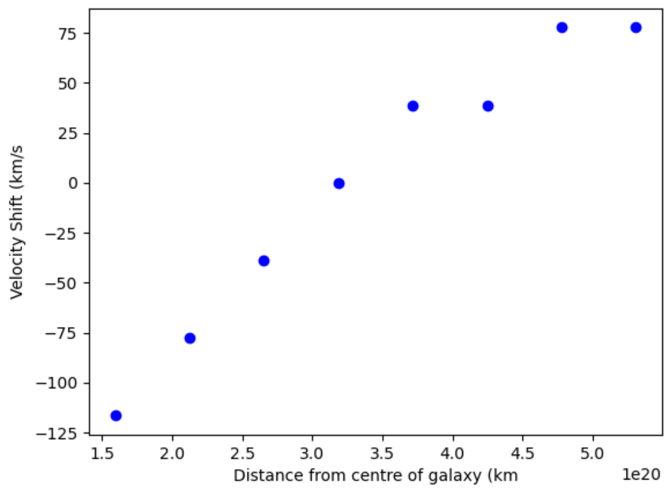
2.5.findDistanceBetweenSlices()

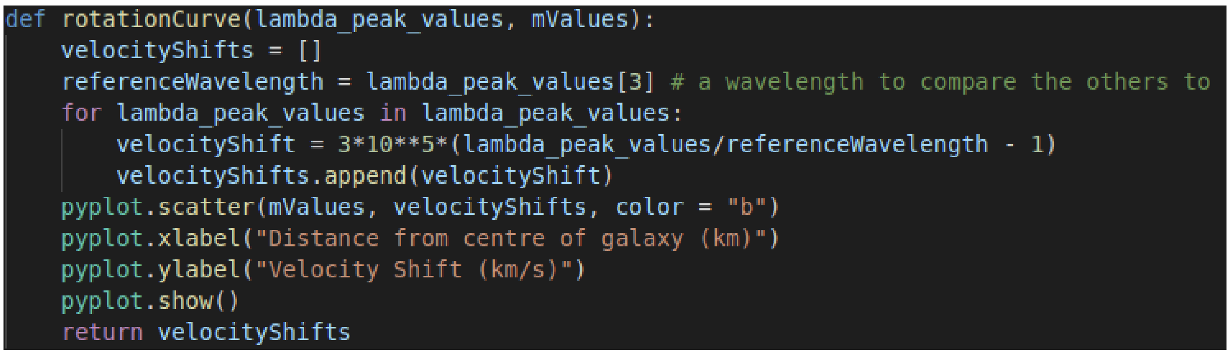


```python
def findDistanceBetweenSlices(slices):
    sliceList = list(range(0, len(slices)))
    mValues = []
    for value in sliceList:
        if 3 <= sliceList.index(value)<= 10: # only takes the middle slices as outer slices are anomylous
            mValues.append(sliceList[value] * 1.72 * 3.086e19)
    return mValues
```

- **findDistanceBetweenSlices()** finds the distance in meters between each of the slices that have been taken, it also masks the data on the edges as only the data in the middle is useful for us.
- It finds the distance between the slices by having a regular interval that is predefined, it then returns the list of **mValues**.

2.6.RotationCurve()

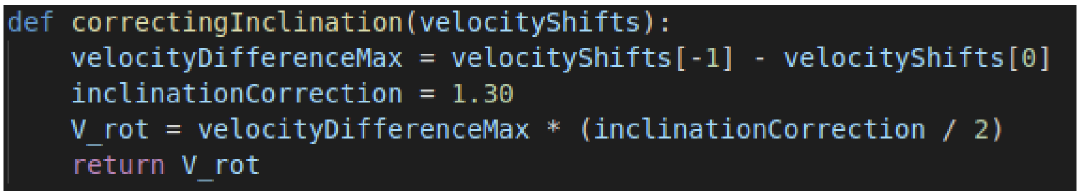




```python
def rotationCurve(lambda_peak_values, mValues):
    velocityShifts = []
    referenceWavelength = lambda_peak_values[3] # a wavelength to compare the others to
    for lambda_peak_values in lambda_peak_values:
        velocityShift = 3*10**5*(lambda_peak_values/referenceWavelength - 1)
        velocityShifts.append(velocityShift)
    pyplot.scatter(mValues, velocityShifts, color = "b")
    pyplot.xlabel("Distance from centre of galaxy (km)")
    pyplot.ylabel("Velocity Shift (km/s)")
    pyplot.show()
    return velocityShifts
```

- **rotationCurve()** takes the **lambda_peak_values** from earlier and the **mValues** and translates the lambda values to find the **velocityShifts** of the slices.
- The list of **velocityShifts** and **mValues** are then plotted on a scatter graph. The graph above is what is output when this code is run and returns the required results.

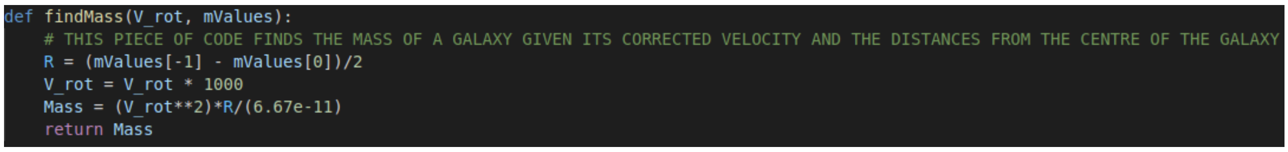
2.7.correctingInclination()



```python
def correctingInclination(velocityShifts):
    velocityDifferenceMax = velocityShifts[-1] - velocityShifts[0]
    inclinationCorrection = 1.30
    V_rot = velocityDifferenceMax * (inclinationCorrection / 2)
    return V_rot
```

- **correctingInclination()** this is necessary as the galaxy is on a slant to us and therefore, we must correct for it. It therefore takes the **velocityShift** values and takes the difference between the first and last value and times it by a predefined value for the galaxy to work out the corrected velocity difference to workout mass with.

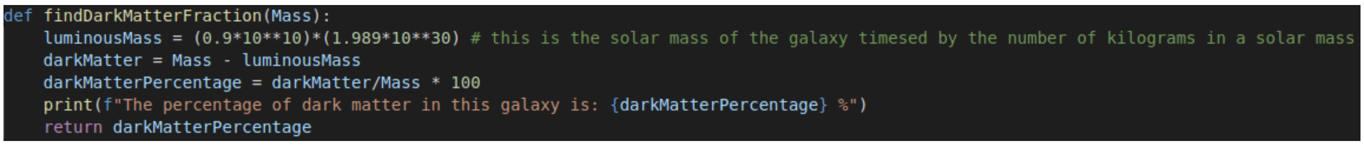
2.8.findMass()



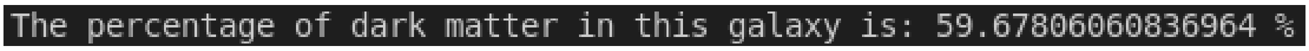
```python
def findMass(V_rot, mValues):
    # THIS PIECE OF CODE FINDS THE MASS OF A GALAXY GIVEN ITS CORRECTED VELOCITY AND THE DISTANCES FROM THE CENTRE OF THE GALAXY
    R = (mValues[-1] - mValues[0])/2
    V_rot = V_rot * 1000
    Mass = (V_rot**2)*R/(6.67e-11)
    return Mass
```

- **findMass()** is used to find the mass of the galaxy, this is done by translating **V_rot** into m/s and then passing it into an equation along with **R** which is half of the full extent of the data from the rotation curve. The **Mass** is then returned.

2.9.findDarkMatterFraction()



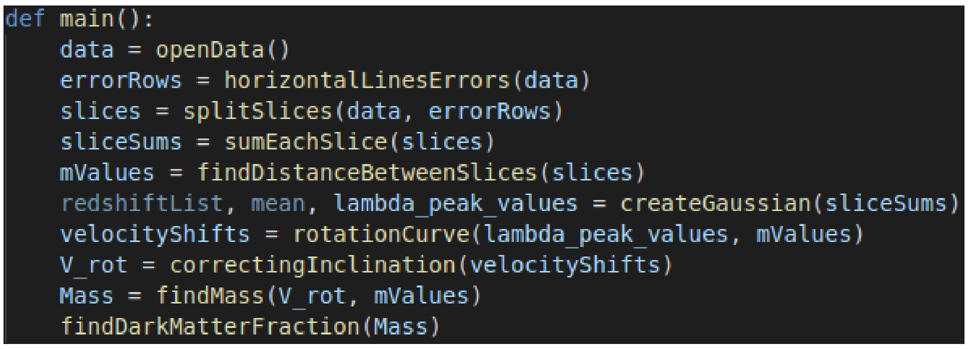
```python
def findDarkMatterFraction(Mass):
    luminousMass = (0.9*10**10)*(1.989*10**30) # this is the solar mass of the galaxy timesed by the number of kilograms in a solar mass
    darkMatter = Mass - luminousMass
    darkMatterPercentage = darkMatter/Mass * 100
    print(f"The percentage of dark matter in this galaxy is: {darkMatterPercentage} %")
    return darkMatterPercentage
```



```
The percentage of dark matter in this galaxy is: 59.67806060836964 %
```

- **findDarkMatterFraction()** is used to find out exactly what fraction of the observed galaxy's mass is made of dark matter, which is our final objective. This is done by working out the **luminousMass** which is predefined and then taking this off the total mass.
- Then we find out what fraction of **Mass** the leftover is and then return this value as a percentage.
- The screenshot above is the output we received when we passed the data through the script, we checked this against a fraction of dark matter – relative velocity graph and found our readings to be highly accurate.

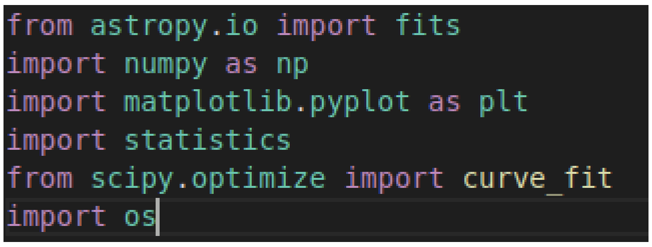
2.10.main()



```
def main():
    data = openData()
    errorRows = horizontalLinesErrors(data)
    slices = splitSlices(data, errorRows)
    sliceSums = sumEachSlice(slices)
    mValues = findDistanceBetweenSlices(slices)
    redshiftList, mean, lambda_peak_values = createGaussian(sliceSums)
    velocityShifts = rotationCurve(lambda_peak_values, mValues)
    V_rot = correctingInclination(velocityShifts)
    Mass = findMass(V_rot, mValues)
    findDarkMatterFraction(Mass)
```

- **main()** calls all the other functions required to determine the **darkMatterPercentage**. First it calls **openData()**, then it calls **horizontalLinesErrors()** to find the **errorRows**. After this the slices are split up from the data using **splitSlices()**.
- Then the returned value of this is passed into **createGaussian()** and **slices** is passed into **findDistanceBetweenSlices()**.
- The **lambda_peak_values** and **mValues** that have been returned from these functions are then passed into **rotationCurve()** to find **velocityShifts**. This is then passed into **correctingInclination()**, which returns **V_rot** which is passed with **mValues** to finally work out the **Mass** which is plugged into **findDarkMatterFraction()** which prints and returns our final value: **darkMatterPercentage**.

Libraries



```
from astropy.io import fits
import numpy as np
import matplotlib.pyplot as plt
import statistics
from scipy.optimize import curve_fit
import os
```

Without some of the libraries we used we would have had a much harder time completing the project. Therefore, it is important to breakdown and reference each one:

Astropy.io - from this library we used **fits** to open the fits file so we could get the data from it.

Numpy – This was used for a lot of different tasks, for creating arrays as well as some functions such as **exp** and **arange**.

Matplotlib – from here we took **pyplot**, so that we could plot all our graphs as otherwise this would have been exceedingly difficult.

Statistics – was used to find the standard deviations of a lot of our rows and columns to identify errors.

Scipy – from this library we took **optimize** and from there **curve_fit** so we could plot our gaussian.

OS - was used so that we could take the fits files from the current working directory.