Data Analysis

Assessment 4

CARDIFF SCHOOL OF PHYSICS AND ASTRONOMY

Michael Norman, *C1325126,* **Email:** NormanM5@cardiff.ac.uk

1.0: Velocity determination:

1.1: Background Reduction

GS2000 is a binary star system comprised of two objects in orbit of each other, one is a star whilst the other is speculated to be either a neutron star or a black hole. The purpose of this short project will be to determine which. It is thought to contain a star and another much denser object as the emission spectra show a characteristic double-peaked emission peak as one would expect from an accretion disk formed as the denser object rips matter from the star which heats up dramatically due to friction as it spirals inwards. The absorption lines of the star can also be seen much more faintly in the spectrum. It is suggested that the start is of K5 stellar type, but testing with other templates will be necessary for a rigorous analysis.

In order to determine the velocity of the star, the spectral data provided, at differing orbital phases, will be cross correlated against a given spectrum stellar type template, initially K5, redshifted across an array of different positions. The template spectrums however are void of any other background bar the emission lines, whereas the raw GS2000 data is not, in order to successfully cross correlate, any extraneous background must first be subtracted from the data, and thus we will do this first across our GS2000 data.

In order to subtract the background, a spline must be made which encompasses the shape of the data whilst avoiding the features, to do this, noise must first be smoothed so a moving average of the data set will be taken, using the equation seen below in Equation 1.1 [2].

$$\hat{t}_i = \frac{1}{N_{\pm 1/2h}} \sum_{t_j \leq t_i - 1/2h}^{t_j \leq t_i + 1/2h} t_i \tag{1.1}$$

The error on the moving average was found using Equation 1.2 and extrapolating for each moving average point:

$$\sigma_{\hat{x}} = \frac{1}{\sqrt{N}} \sqrt{\frac{1}{N-1} \sum_{i}^{N} (x_i - \hat{x})} \tag{1.2}$$

The following code was used to find a weighted central moving average and its associated errors:

```
1.  def calcMovAvg(data, err, wind_size):
2.
3.      """ Calculates weighted moving average given data set with errors given """
4.
5.      wghts = (1./(err**2.))
6.      w_data = data*wghts
7.
8.      cumsum_data = np.cumsum(np.insert(w_data, 0, 0))
9.      cumsum_wghts = np.cumsum(np.insert(wghts, 0, 0))
10.     mov_avg = (cumsum_data[wind_size:] - cumsum_data[:-
    wind_size]) / (cumsum_wghts[wind_size:] - cumsum_wghts[:-wind_size])
11.     pad_mov_avg = np.zeros(wind_size + len(mov_avg) - 1)
12.     pad_mov_avg[(np.floor(wind_size/2) - 1):(np.floor(-wind_size/2))] = mov_avg
13.
14.     mov_avg_err = (np.sqrt(1/(cumsum_wghts[wind_size:] - cumsum_wghts[:-
    wind_size])*np.sum((data - pad_mov_avg))))
15.
16.     pad_mov_avg_err = np.zeros(wind_size + len(mov_avg) - 1)
17.     pad_mov_avg_err[(np.floor(wind_size/2) - 1):(np.floor(-wind_size/2))] = mov_avg_err
18.
19.     return pad_mov_avg, pad_mov_avg_err
```

Figure 1.1: Code used to calculate moving average and its errors.

The moving average was padded with zeros at either side in regions where the average window would extend outside the data range, these zeros would be discarded later.

Once the moving average was found, a spline was interpolated selecting only a sample of moving average points taken at regular intervals to remove any wanted features from the interpolation. The spline was weighted using the errors calculated by the moving average, and an error was found from the spline by returning its residuals. The error was found to be small compared to the error in the raw data, by the order of several magnitudes, and so will be discounted.

The numpy UnivariateSpline function was used to find the spline function as can be seen in the code snipped below, see **Figure 1.2:**

```
1.  def calcSpctSpln(spect):
2.
3.      spect.spln_y = np.zeros_like(spect.raw_y)
4.      spect.diff_y = np.zeros_like(spect.raw_y)
5.      spect.spln_err_y = np.zeros(spect.num)
6.
7.      """ Calculates moving average for spectrum """
8.
9.      n = np.arange(spect.num)
10.     for data_idx in n:
11.         spln_func = it.UnivariateSpline(spect.conv_x[data_idx][spect.wind_size/2:(-
    spect.wind_size/2) - 1:spect.spln_size], spect.mov_avg_y[data_idx][spect.wind_size/2:(-
    spect.wind_size/2) - 1:spect.spln_size], w = spect.mov_avg_y_err[data_idx][spect.wind_size/2:(-
    spect.wind_size/2) - 1:spect.spln_size] )
12.         spect.spln_err_y[data_idx] = np.sqrt(1/len(spect.mov_avg_y[data_idx][spect.wind_size/2:(-
    spect.wind_size/2) - 1:spect.spln_size])*spln_func.get_residual())
13.         spect.spln_y[data_idx] = spln_func(spect.conv_x[data_idx])
14.         spect.diff_y[data_idx] = spect.raw_y[data_idx] - spect.spln_y[data_idx]
15.     return spect
```

Once the spline was calculated the raw data x values were fed into the function to produce a spline value for each raw data point. The following is a plot of an example data set showing the data points and their error, the moving average and the spline. Parameters, such the moving average window and the period of spline function have been adjusted to find a good fit

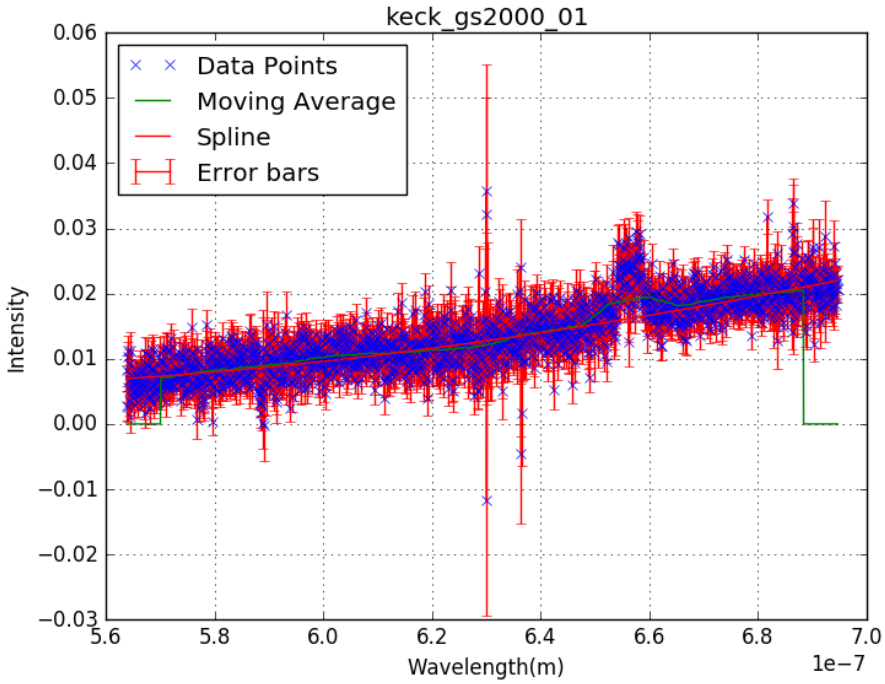

Figure 1.3: Illustration of spline and moving average values.

The same process was coded to repeat for each of the GS2000 data sets and the template spectrum k5, below is an example of the same process applied to the K5 template spectrum, which will be our initial test.

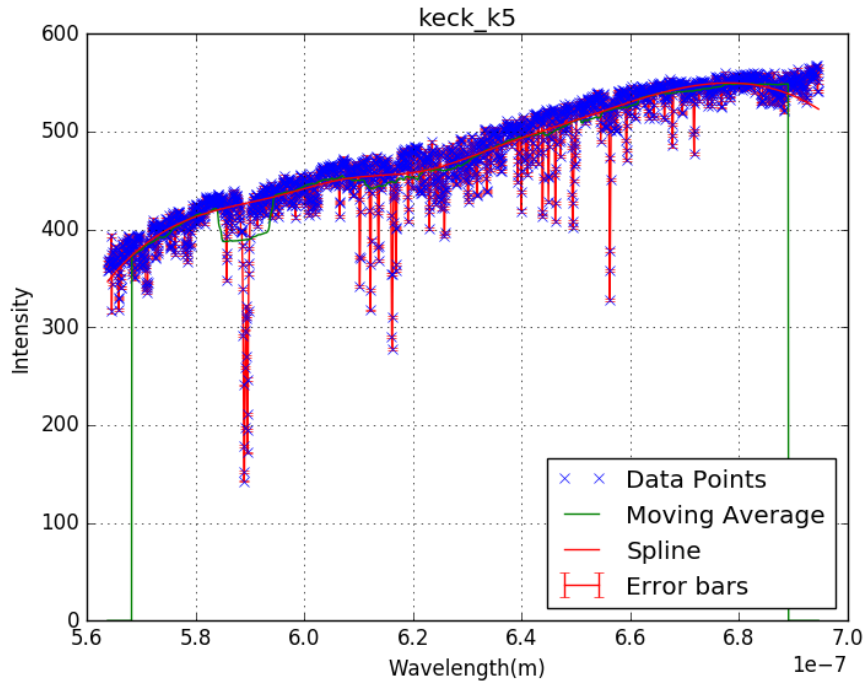

Figure 1.4: Illustration of spline and moving average values, on the K5 spectral template.

To complete the background subtraction, the calculated spline points were subtracted from original data sets to produce data as in the following **Figure 1.5.** The error was kept as original is it was found, perhaps erroneously, to be much higher than any error introduced by the background subtraction.

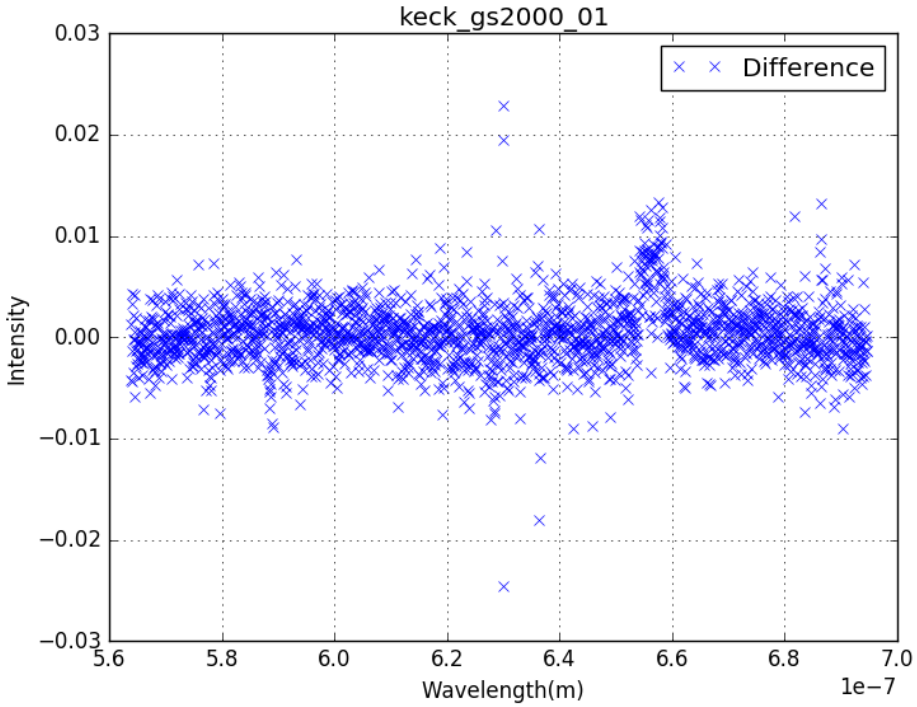

Figure 1.4: Illustration of GS200 data with background subtracted, the same method was applied to all spectra and templates.

1.2: Template shifting

Once the background subtraction was complete, the next step was to shift the template spectrum across an array of different velocities then interpolate a function from each velocity to create new y values for each new set of x-values. The Doppler shift was accomplished using the following **Equation 1.3.**

$$\Delta\lambda = \frac{v\lambda_0}{c} \tag{1.3}$$

A 2d array of new x values was calculated across a velocity range from -600kms^{-1} to 600kms^{-1} in 2000 intervals, this range was found using trial and error. A 2d array of y values was calculated from these x values, using numpy's InterpolatedUnivariateSpline function. The errors here were again ignored as they were assumed to be small compared to the error on the raw data.

```
115 def calcDopShft(data,veloc,c_veloc):
116     return data*(c_veloc+veloc)/c_veloc
117 
118 def calcSpctShft(temps, spcts, velocs, c_veloc, num_veloc, det):
119     temps.shft_x = np.zeros((temps.num, num_velocs, len(temps.conv_x[0])))
120     temps.int_fn =  np.zeros((temps.num, num_velocs))
121     temps.itrp_x = np.zeros((temps.num,det))
122     temps.itrp_y = np.zeros((temps.num, num_velocs, det))
123 
124     spcts.itrp_y = np.zeros((temps.num, spcts.num, num_velocs, len(spcts.conv_x[0])))
125 
126     v = np.arange(num_velocs)
127     t = np.arange(temps.num)
128     s = np.arange(spcts.num)
129 
130     for temp_idx in t:
131         for spct_idx in s:
132             temps.itrp_x[temp_idx] = np.linspace(temps.conv_x[temp_idx][0] - 0.5*(temps.conv_x[temp_idx][-1] - temps.conv_x[temp_idx][0]),
    temps.conv_x[temp_idx][-1] + 0.5*(temps.conv_x[temp_idx][-1] - temps.conv_x[temp_idx][0]), det)
133             for veloc_idx in v:
134                 temps.shft_x[temp_idx][veloc_idx] = calcDopShft(temps.conv_x[temp_idx], velocs[veloc_idx], c_veloc)
135                 spln_func = it.InterpolatedUnivariateSpline(temps.shft_x[temp_idx][veloc_idx], temps.diff_y[temp_idx])
136                 spcts.itrp_y[temp_idx][spct_idx][veloc_idx] = spln_func(spcts.conv_x[spct_idx])
137 
138     return temps
```

Figure 1.5: Template shifting function.

1.2: Scaling to fit

Once the template shifted x and y values were calculated, the next set was to find the scale factor needed to scale the shifted templates to fit the data in each velocity case. This was done using a chi squared minimisation technique as seen in the lecture notes. The following **equation 1.4[2]** was used:

$$A = \frac{\sum_i^N y_i P(x_i)/\sigma_i^2}{\sum_i^N [P(x_i)]^2/\sigma_i^2} \tag{1.4}$$

A scale factor was found for each velocity scaled to each GS2000 spectrum to create a 2d array of scale factors, then a chi squared value was calculated for each scale factor using the following **Equation 1.5:**

$$\chi^2 = \sum_i^N \left(\frac{y_i - AP(x_i)}{\sigma_i^2}\right) \tag{1.5}$$

Plotting these chi-squared values against velocity it's easy to see the minimum chi squared value suggesting this is the velocity which has the best fit for the given GS2000 data, see **figure 1.6.** 200 data points were cropped from both sides to remove erroneous data points.

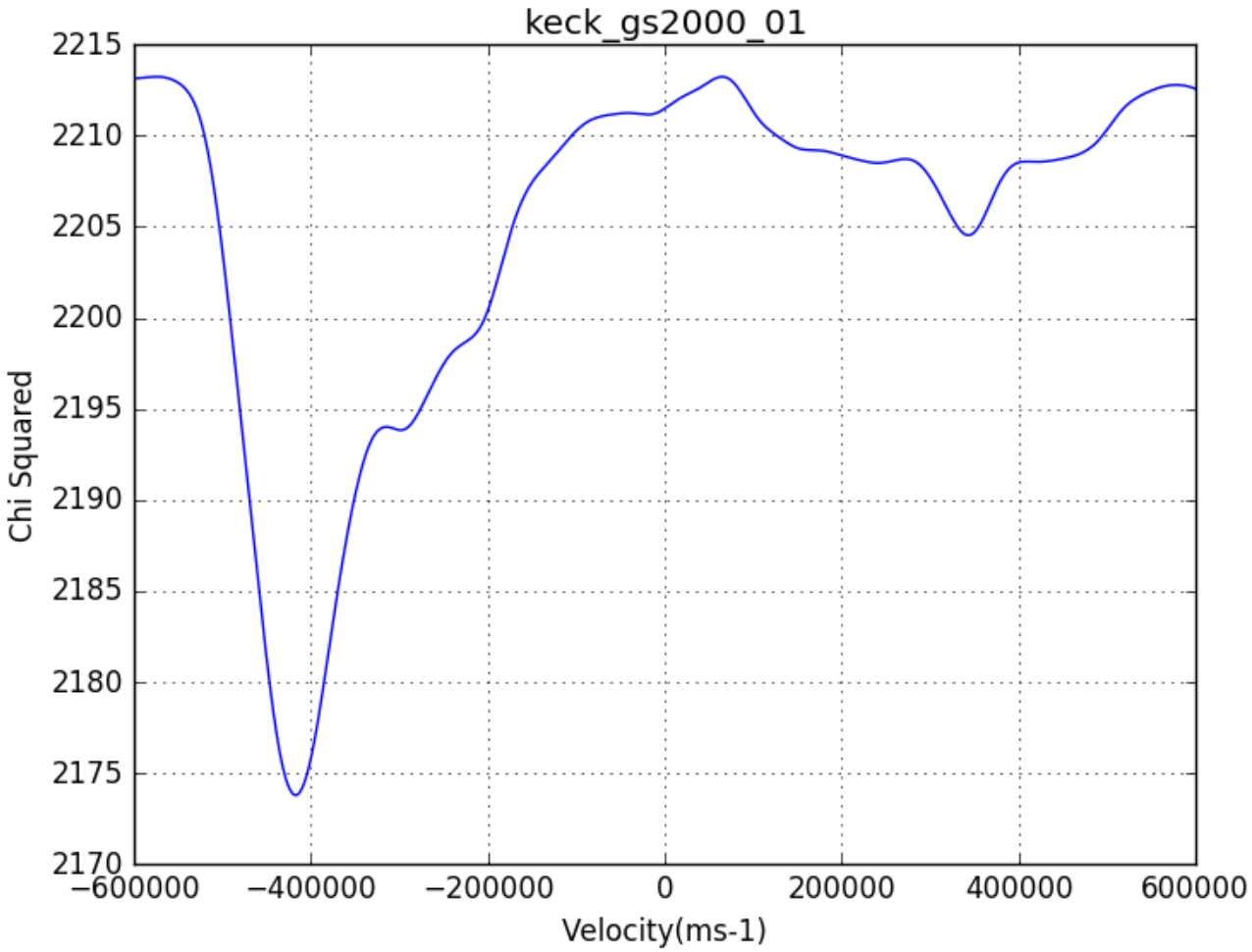

Figure 1.6: Chi Squared plotted against velocity for the first GS2000 data set.

To find the error on the minimum velocity to one-sigma, a trick of chi-squared space was used in which a value of one was added to the minimum chi-squared value, then the velocities which gave chi squared + 1 were used as error margins, with the largest difference taken as the error.

```
1.  for temp_idx in t:
2.      for spct_idx in s:
3.          print(spct_idx)
4.          for veloc_idx in v:
5.              scle_fcts[temp_idx][spct_idx][veloc_idx] = scleSpct(spcts.diff_y[spct_idx][200:-
    200], spcts.itrp_y[temp_idx][spct_idx][veloc_idx][200:-200], spcts.err_y[spct_idx][200:-200])
6.              chi_sqrd[temp_idx][spct_idx][veloc_idx] = chiSqr(spcts.diff_y[spct_idx][200:-
    200], spcts.itrp_y[temp_idx][spct_idx][veloc_idx][200:-200], spcts.err_y[spct_idx][200:-
    200], scle_fcts[temp_idx][spct_idx][veloc_idx] )
7.
8.          min_points[temp_idx][spct_idx] = min(chi_sqrd[temp_idx][spct_idx])
9.          chi_min_veloc[temp_idx][spct_idx] = velocs[np.argmin(chi_sqrd[temp_idx][spct_idx])]
10.
11.         chiFunc = it.InterpolatedUnivariateSpline(velocs, chi_sqrd[temp_idx][spct_idx])
12.         temp_veloc = np.linspace(min_veloc, max_veloc, 10000)
13.         temp_chi = chiFunc(temp_veloc)
14.         chi_err[temp_idx][spct_idx] = np.abs(chi_min_veloc[temp_idx][spct_idx] - max(temp_veloc[np
    .argwhere(temp_chi < (min_points[temp_idx][spct_idx] + 1))]))
```

Figure 1.7: Code snippet which calculates the velocity when chi squared is minimised and its error.

Phase	Chi Minimised Velocity	Error
-0.8828	-4.17E+05	1.16E+04
-0.3663	-1.04E+05	2.00E+04
0.2042	1.90E+05	1.36E+04
0.6271	3.38E+05	1.04E+04
1.0933	5.02E+05	1.10E+04
1.4514	5.33E+05	9.03E+03
1.9346	5.26E+05	9.75E+03
2.3242	3.93E+05	1.02E+04
2.7571	2.37E+05	1.01E+04
3.1466	5.13E+04	9.94E+03
3.5802	-1.96E+05	1.52E+04
4.0030	-3.98E+05	9.28E+03
4.5716	-4.80E+05	1.22E+04

Table 1.1: Chi Minimised Velocity vs phase for k5 template.

1.3: Fitting to velocity sin curve

Once the chi squared minimised velocities were calculated, the next step was to fit the sin curve equation provided in the question paper to the curve. The equation given is as follows in **Equation 1.6.**

$$V(\phi) = \gamma + K_x \sin(2\pi\phi) + K_Y \sin(2\pi\phi) \tag{1.6}$$

This was fitted to the chi squared minimised velocity values using another chi-squared least squares fit, the equation to be minimised is as follows:

$$\chi^2 = \sqrt{\frac{y_i - (\gamma + K_x \sin(2\pi\phi_i) + K_Y \sin(2\pi\phi_i)))}{\sigma_y^2}} \tag{1.7}$$

In order to minimise the above equation, it must be differentiated, in terms of its three variables $\gamma, K_x \; and \; K_Y$, to produce a set of three simulatanious equations which will then be set to zero to find the minima of the 3d surface. The error values are changed to weights using the equation $w = \frac{1}{\sigma^2}$. The following matrix equation shown in **Equation 1.7** was applied to calculate the best fit values for $\gamma, K_x \; and \; K_Y$.

$$\begin{pmatrix} \sum w_i & \sum w_i \sin(2\pi\phi_i) & \sum w_i \cos(2\pi\phi_i) \\ \sum w_i \sin(2\pi\phi_i) & \sum w_i \sin^2(2\pi\phi_i) & \sum w_i \cos(2\pi\phi_i) \sin(2\pi\phi_i) \\ \sum w_i \cos(2\pi\phi_i) & \sum w_i \sin(2\pi\phi_i) \cos(2\pi\phi_i) & \sum w_i \cos^2(2\pi\phi_i) \end{pmatrix} \begin{pmatrix} \gamma \\ K_x \\ K_y \end{pmatrix} = \begin{pmatrix} \sum w_i y_i \\ \sum w_i y_i \sin(2\pi\phi_i) \\ \sum w_i y_i \cos(2\pi\phi_i) \end{pmatrix} \tag{1.8}$$

Using this method values for $\gamma, K_x \; and \; K_Y$ were calculated, using a numpy matrix solving function. A value of 28138.792 ± 2889.353 was calculated for γ using the k5 template, a value of 3330.359 ± 4999.413 ms^{-1} was calculated for K_x, and a value of 28138.792 ± 4221.510 ms^{-1} was calculated for K_Y. The error on $\gamma, K_x \; and \; K_Y$ was calculated by diagonalisng the inverse of the Hessian matrix. Since the equation is linear the shape of the error distribution should be the same for all parameters. The hessian matrix is given using the following **Equation 1.8.**

$$H_{jk} = \frac{1}{2} \frac{\partial^2 \chi^2}{\partial \propto_j \partial \propto_k} \tag{1.9}$$

Using this the Hessian matrix was found to be:

$$H = \begin{pmatrix} \sum w_i & \sum w_i \sin(2\pi\phi_i) & \sum w_i \cos(2\pi\phi_i) \\ \sum w_i \sin(2\pi\phi_i) & \sum w_i \sin^2(2\pi\phi_i) & \sum w_i \cos(2\pi\phi_i) \sin(2\pi\phi_i) \\ \sum w_i \cos(2\pi\phi_i) & \sum w_i \sin(2\pi\phi_i) \cos(2\pi\phi_i) & \sum w_i \cos^2(2\pi\phi_i) \end{pmatrix} \quad (1.10)$$

The inverse of the Hessian matrix provides a covariance matrix, in which the diagonal elements provide the variance, and the covariance is shown in the off axis elements. The parameters have some dependence on each other however, as is illustrated by the fact that the off-axis elements of the hessian are not 0. In order to eliminate this dependence and find the true error, the matrix was diagonalised by calculating its eigenvectors which become the diagonal elements of the new diagonalized matrix as well as the variance for our values. From the variance the 1 sigma error was then calculated by taking the square root.

```
1.  s_val = np.sin(rad_phse); c_val = np.cos(rad_phse); w_veloc = chi_min_veloc*chi_wgts
2.
3.  A = np.sum(chi_wgts), np.sum(chi_wgts*s_val), np.sum(chi_wgts*c_val)
4.  B = np.sum(chi_wgts*s_val), np.sum(chi_wgts*s_val**2), np.sum(chi_wgts*s_val*c_val)
5.  C = np.sum(chi_wgts*c_val), np.sum(chi_wgts*c_val*s_val), np.sum(chi_wgts*c_val**2)
6.
7.  RHS = np.sum(w_veloc), np.sum(w_veloc*s_val), np.sum(w_veloc*c_val)
8.  LHS = np.array([A,B,C])
9.
10. gamma,kx,ky = np.linalg.solve(LHS, RHS)
11.
12. eig = np.linalg.eig(LHS)
13.
14. err_mat = np.diag(eig[0])
15. inv = np.linalg.inv(err_mat)
16.
17. gamma_err = np.sqrt(inv[0][0])
18. kx_err = np.sqrt(inv[1][1])
19. ky_err = np.sqrt(inv[2][2])
20.
21. arr_phse = np.linspace(min(org_phse), max(org_phse), 1000)
22. sin_fit = sinCurve(gamma, kx, ky, arr_phse)
23.
24. fit_err = sigmaErr(gamma,kx,ky,org_phse,chi_min_veloc, num_spcts)
```

Figure 1.8: Code snippet showing the generation of fit parameters and their associated errors.

Once the fir parameters were calculated a plot could be made for the K5 template fitted against the data, giving the following **Figure 1.9**.

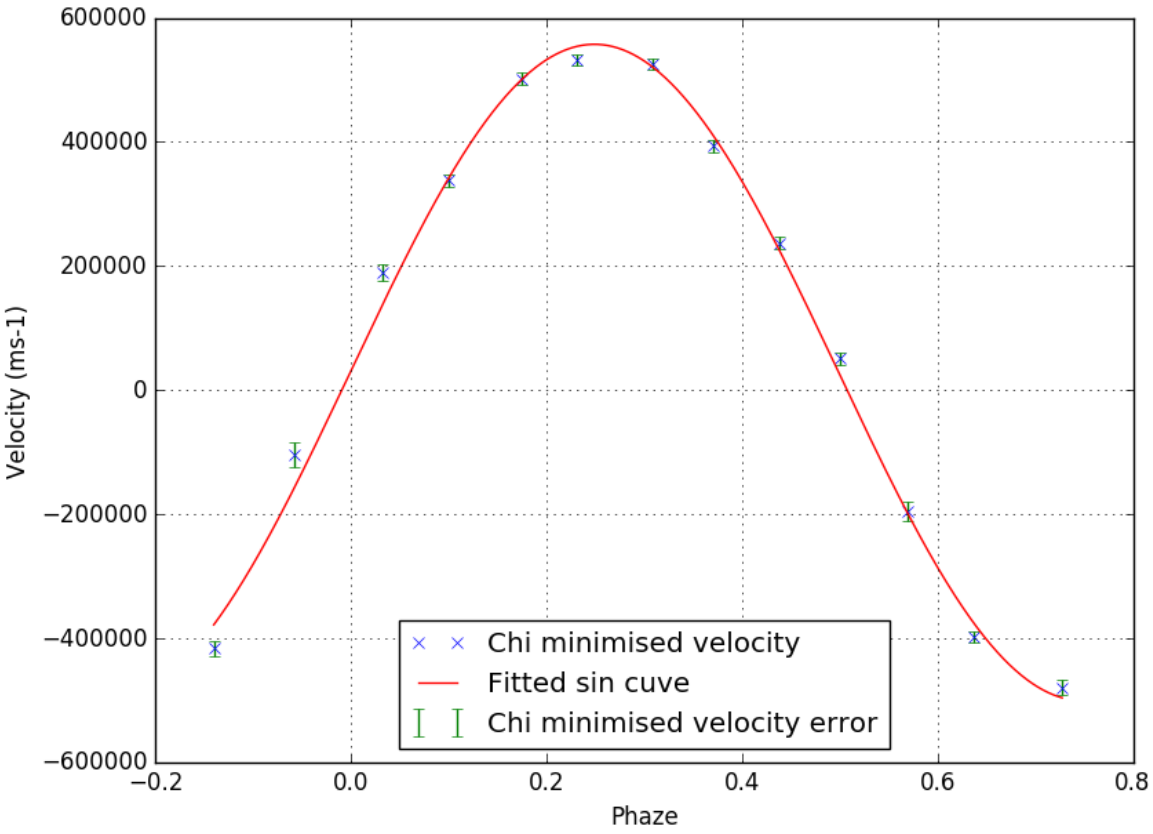

Figure 1.9: Fitted sine curve plotted against chi-minimised data and its error for the chi squared template.

An error on the fit was calculated using the following code:

```
1.  def sinCurve(gamma, kx, ky, phase):
2.      return gamma + kx*np.sin(2*np.pi*phase) + ky*np.cos(2*np.pi*phase)
```

Giving a value of 31114.255 for σ_y The process was then repeated for all the templates to investigate the relative fits of the different templates to the given Equation 1.6:

Spectral Type	σ_y
g5	552185
g9	527127
k0	540284
k1	501191
k2	520093
k4	32663
k5	31114
k7	26076
k8	26639
mo	25001

Table 1.1: Fits vs spectral type.

The spectral types best matching the curve are of small dwarf stars, with the best fit applying to the m0 spectral type. We will continue however, using the k5 template since it was the type suggested by the assignment.

1.3: Calculating radial velocity:

In order to calculate the radial velocity based on fitted parameters, we were provided with the following **Equation 1.8.:**

$$K = \sqrt{K_x^2 + K_y^2} \tag{1.11}$$

This was calculated by using the numpy.hypot function. An equation for the error in K was found using error propagation in quadrature. Giving the following **Equation 1.12**:

$$\sigma_k = \sqrt{\frac{K_x^2}{K}\sigma_{K_x} + \frac{K_y^2}{K}\sigma_{K_y}} \tag{1.12}$$

A value for K was calculated as 529.4 ± 5.0 kms^{-1}.

2.0: Mass determination:

2.1: Mass function determination

The mass function was determined using the equation provided in the question, **Equation 2.1**. The period was taken from the provided keck notes file, with error in this case, assumed to be negligible.

$$f(M_x) = \frac{PK^3}{2\pi G} \tag{2.1}$$

And the error was calculated again through quadrature, assuming the error on the period and the gravitational constant are negligible, giving the following **equation 2.2:**

$$\sigma_{f(M_x)} = \frac{3PK^2}{2\pi G}\sigma_k \tag{2.2}$$

The value found for the mass function was: 5.291 +- 0.150 $M_\odot$.

```
1.  def calcStlrMass(prd,rad_veloc,rad_veloc_err,grav_cnst):
2.      mass = (prd*rad_veloc**3)/((2*np.pi*grav_cnst))
3.      err = ((3*prd*(rad_veloc**2))/(2*np.pi*grav_cnst))*rad_veloc_err
4.      return mass, err
```

Figure 2.1: Function used to calculate stellar mass function and its error.

2.1: Single angle companion mass determination

The mass function corresponds to the mass as if the binary system was being viewed head on, an unlikely scenario, in reality it will probably be at some angle to the telescope. Thus, we have to use the equation 2.3:

$$f(M_x) = \frac{M_x^2 sin^3 i}{(M_x + M_C)^2} \tag{2.3}$$

Taking an average value for the mass of K5 types stars [3], a value for M_C is suggested at 0.625 solar masses. The equation must be solved as a cubic, which was done so using the numpy function np.roots. Only of the roots was real and positive and thus this was taken to be the answer. The error was calculated by running the value of $f(M_x) \pm errors$ through the solver. The cubic region was nearly linear so there are no maxima hiding within the errors. Taking the angle i to be 90° a value of 6.378 +- 0.153 $M_\odot$ was calculated as the mass of the companion. Much too massive to be a neutron star given the limit suggested by the assignment paper.

2.2: Monty Carlo estimation for companion mass probability distribution

In reality we cannot assume the binary system to be at a set angle. In the original paper [4], the value for i was given as 48°±11° these values will be used within a Monty Carlo estimation, to better calculate a likely estimate for the companion's mass. The error on the period was assumed negligible as it was very small compared to other sources of error, the error on the stellar mass was said to be: ±0.175 $M_\odot$, the error on the radial velocity was set as σ_k and the $\sigma_{f(M_x)}$, calculated using that and period.

Normal distribution was assumed in all cases and arrays of random samples with 10,000 was created for all variables then fed into the mass calculation formula to produce a distribution. See **Figure 2.3** below:

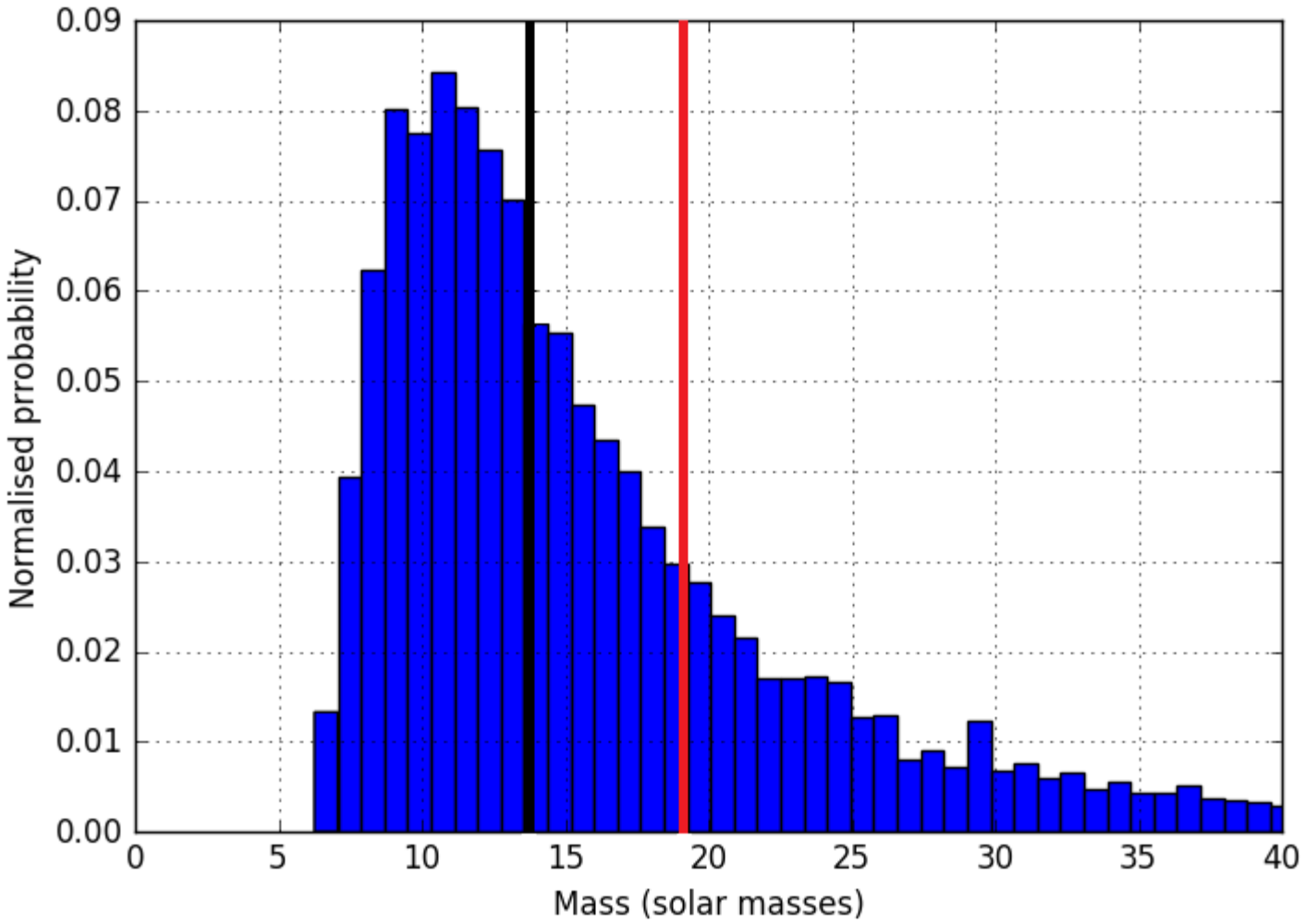

Figure 2.3: Mass Distribution created using Monty Carlo method, with the black line representing the median and the red the mean.

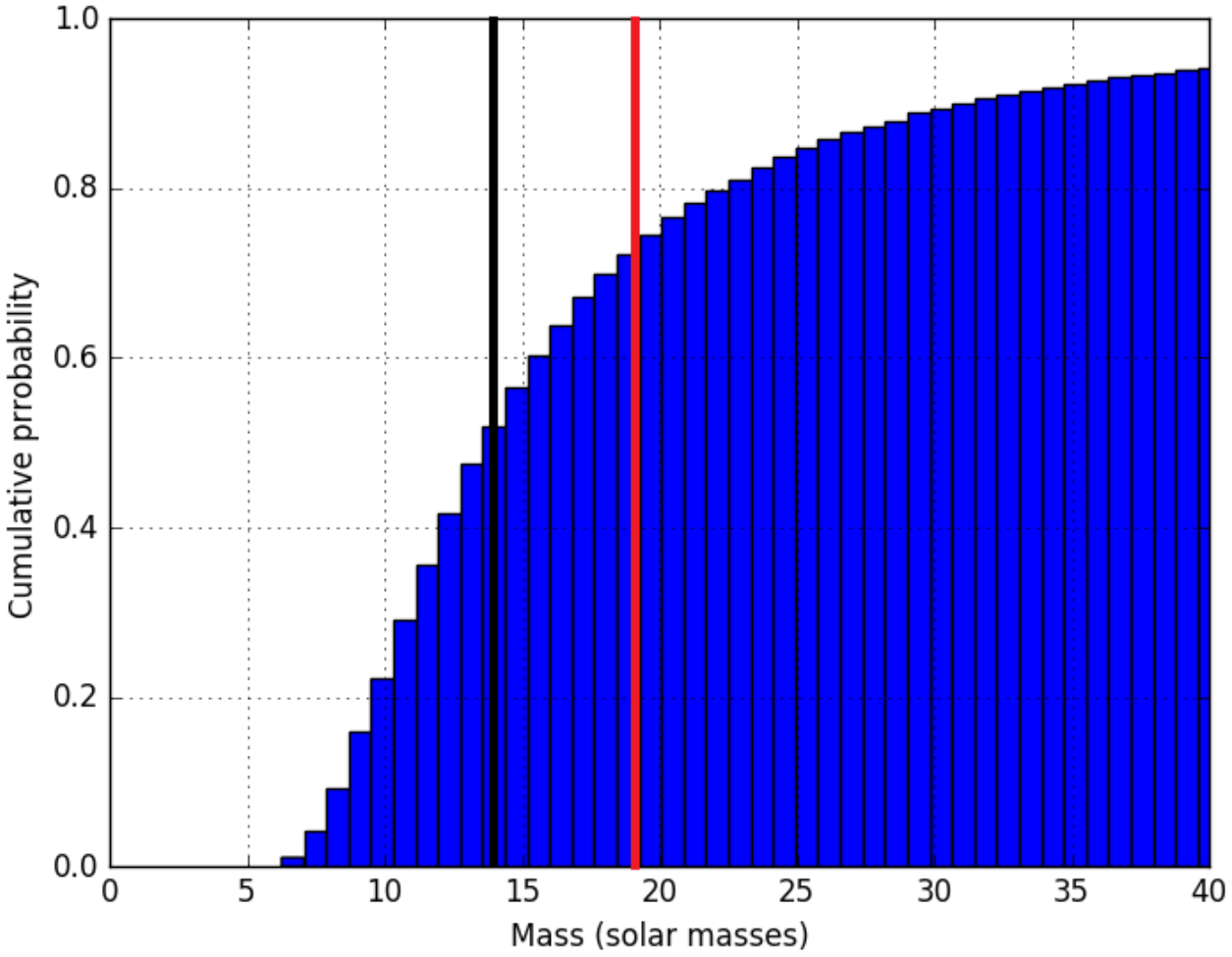

Figure 2.4: Cumulative probability mass distribution created using Monty Carlo distribution, with the black line representing the median and the red the mean.

The above plots 2.3 and 2.4 show the mass distribution and the cumulative mass distribution respectively, the black lines represent the median in each case and the red lines the mean, the mean of the data set was 19.2 and the median 14.0. The distribution has an extremely long tail in the positive x direction, enlarging the standard deviation to 30, which is clearly not a good estimate of error in the negative direction which falls of sharply.

The probability of the companion object having a mass of less than 3 is practically 0 by any measurable amount, thus the data suggests the object is a black hole in as certain terms as it is ever possible to be in statistics, unless of course there are some inherent errors unknown within the method, which I wouldn't say is unlikely. Though this is not a rigorous answer, I do not know how to generate a more scientific one from the data.

```
num_smpls = 10000

i_tri = np.random.normal(np.deg2rad(48), np.deg2rad(11), num_smpls)

#uniform(0.2,np.pi - 0.2, num_smpls)
mc_tri = np.random.normal(mass_c,mass_c_err, num_smpls)
rad_veloc_tri = np.random.normal(rad_veloc, rad_veloc_err, num_smpls)
mass_fn_tri, mass_fn_err_arr = calcStlrMass(prd, rad_veloc_tri, rad_veloc_err, grav_cnst)

smpl_mass_x = np.zeros(len(i_tri))
for smpl_idx in np.arange(len(i_tri)):
    smpl_mass_x[smpl_idx] = calcMass(mass_fn_tri[smpl_idx]/sol_mass,mc_tri[smpl_idx],i_tri[smpl_id
x])

```

```
mean = np.mean(smpl_mass_x)
med = np.median(smpl_mass_x)
std = np.std(smpl_mass_x)
```

Figure 2.5: Code used to generate probability distributions

References

[1] PX 4128 – Data Analysis, Continual Assessment 4

[2]PX 4128 - Data Analysis Notes

[3]Wikipedia, Stellar Classification, Accessed, 14/12/2017 https://en.wikipedia.org/wiki/Stellar_classification

[4] A Black Hole in the X-Ray Nova GS 2000+25, Alexei V. Filippenko, Thomas Matheson, and Aaron J. Barth, 1995

Appendix Full Code

```
# -*- coding: utf-8 -*-
"""
Created on Sat Dec  9 17:02:58 2017

@author: Michael
"""
import numpy as np
import scipy.interpolate as it
import matplotlib.pyplot as plt
import os

""" ~~~~~~~~~~ Functions ~~~~~~~~~~ """


def formatGraph(g_title,x_title,y_title):

    """ Ease of life function for graph formating """

    plt.title( g_title )
    plt.xlabel( x_title )
    plt.legend(loc='best')
    plt.ylabel(y_title)
    plt.grid()

def readSpct(spect):

    """ Reads spectrograms from inputted folder into dictionary"""

    num_files = 0

    for file in os.listdir(spect.dir_name):
        if file.endswith(spect.ext):
            num_files += 1

    file_names = [None]*num_files

    file_idx = 0

    x_data = [None]*num_files
    y_data = [None]*num_files
    y_err = [None]*num_files

    for file in os.listdir(spect.dir_name):
        if file.endswith(spect.ext):
            file_name = file.rstrip(spect.ext)
            file_names[file_idx] = file_name
            x_data[file_idx], y_data[file_idx], y_err[file_idx] = np.loadtxt(os.path.join(spect.di
r_name, file), unpack = True)
```

```
            file_idx += 1

    spect.num = num_files
    spect.names = file_names
    spect.raw_x = np.asarray(x_data)
    spect.raw_y = np.asarray(y_data)
    spect.err_y = np.asarray(y_err)

    return spect

def convAngToMet(spect):

    spect.conv_x = np.zeros_like(spect.raw_x)

    n = np.arange(spect.num)
    for data_idx in n:
        spect.conv_x[data_idx]= spect.raw_x[data_idx]*1E-10
    return spect

def calcMovAvg(data, err, wind_size):

    """ Calculates weighted moving average given data set with errors given """

    wghts = (1./(err**2.))
    w_data = data*wghts

    cumsum_data = np.cumsum(np.insert(w_data, 0, 0))
    cumsum_wghts = np.cumsum(np.insert(wghts, 0, 0))
    mov_avg = (cumsum_data[wind_size:] - cumsum_data[:-wind_size]) / (cumsum_wghts[wind_size:] - cumsum_wghts[:-wind_size])
    pad_mov_avg = np.zeros(wind_size + len(mov_avg) - 1)
    pad_mov_avg[(np.floor(wind_size/2) - 1):(np.floor(-wind_size/2))] = mov_avg

    mov_avg_err = (np.sqrt(1/(cumsum_wghts[wind_size:] - cumsum_wghts[:-wind_size])*np.sum((data - pad_mov_avg))))

    pad_mov_avg_err = np.zeros(wind_size + len(mov_avg) - 1)
    pad_mov_avg_err[(np.floor(wind_size/2) - 1):(np.floor(-wind_size/2))] = mov_avg_err

    return pad_mov_avg, pad_mov_avg_err

def calcSpctAvg(spect):

    """ Calculates moving average for spectrum """

    spect.mov_avg_y = np.zeros_like(spect.raw_y)
    spect.mov_avg_y_err = np.zeros_like(spect.raw_y)

    n = np.arange(spect.num)
    for data_idx in n:
        spect.mov_avg_y[data_idx], spect.mov_avg_y_err[data_idx] = calcMovAvg(spect.raw_y[data_idx],spect.err_y[data_idx],spect.wind_size)
    return spect

def calcSpctSpln(spect):

            spect.spln_y = np.zeros_like(spect.raw_y)
            spect.diff_y = np.zeros_like(spect.raw_y)
            spect.spln_err_y = np.zeros(spect.num)

            """ Calculates moving average for spectrum """

            n = np.arange(spect.num)
            for data_idx in n:
                spln_func = it.UnivariateSpline(spect.conv_x[data_idx][spect.wind_size/2:(-spect.wind_size/2) - 1:spect.spln_size], spect.mov_avg_y[data_idx][spect.wind_size/2:(-spect.wind_size/2) - 1:spect.spln_size], w = spect.mov_avg_y_err[data_idx][spect.wind_size/2:(-spect.wind_size/2) - 1:spect.spln_size] )
                spect.spln_err_y[data_idx] = np.sqrt(1/len(spect.mov_avg_y[data_idx][spect.wind_size/2:(-spect.wind_size/2) - 1:spect.spln_size])*spln_func.get_residual())
```

```
111.                spect.spln_y[data_idx] = spln_func(spect.conv_x[data_idx])
112.                spect.diff_y[data_idx] = spect.raw_y[data_idx] - spect.spln_y[data_idx]
113.            return spect
114.
115.        def calcDopShft(data,veloc,c_veloc):
116.            return data*(c_veloc+veloc)/c_veloc
117.
118.        def calcSpctShft(temps, spcts, velocs, c_veloc, num_veloc, det):
119.            temps.shft_x = np.zeros((temps.num, num_velocs, len(temps.conv_x[0])))
120.            temps.int_fn =  np.zeros((temps.num, num_velocs))
121.            temps.itrp_x = np.zeros((temps.num,det))
122.            temps.itrp_y = np.zeros((temps.num, num_velocs, det))
123.
124.            spcts.itrp_y = np.zeros((temps.num, spcts.num, num_velocs, len(spcts.conv_x[0])))
125.
126.            v = np.arange(num_velocs)
127.            t = np.arange(temps.num)
128.            s = np.arange(spcts.num)
129.
130.            for temp_idx in t:
131.                for spct_idx in s:
132.                    temps.itrp_x[temp_idx] = np.linspace(temps.conv_x[temp_idx][0] - 0.5*(temps.con
    v_x[temp_idx][-1] - temps.conv_x[temp_idx][0]), temps.conv_x[temp_idx][-
    1] + 0.5*(temps.conv_x[temp_idx][-1] - temps.conv_x[temp_idx][0]), det)
133.                    for veloc_idx in v:
134.                        temps.shft_x[temp_idx][veloc_idx] = calcDopShft(temps.conv_x[temp_idx], vel
    ocs[veloc_idx], c_veloc)
135.                        spln_func = it.InterpolatedUnivariateSpline(temps.shft_x[temp_idx][veloc_id
    x], temps.diff_y[temp_idx])
136.                        spcts.itrp_y[temp_idx][spct_idx][veloc_idx] = spln_func(spcts.conv_x[spct_i
    dx])
137.
138.            return temps
139.
140.        def calcShftInrp(spcts, num_velocs, inrp_det):
141.
142.            temps.itrp_x = np.zeros((temps.num, inrp_det))
143.            temps.itrp_y = np.zeros((temps.num, num_velocs, inrp_det))
144.
145.            spcts.itrp_y = np.zeros((temps.num, spcts.num, num_velocs, len(spcts.conv_x[0])))
146.
147.            t = np.arange(temps.num)
148.            s = np.arange(spcts.num)
149.
150.            for temp_idx in t:
151.                for spct_idx in s:
152.                    temps.itrp_x[temp_idx] = np.linspace(temps.conv_x[temp_idx][0] - 0.5*(temps.con
    v_x[temp_idx][-1] - temps.conv_x[temp_idx][0]), temps.conv_x[temp_idx][-
    1] + 0.5*(temps.conv_x[temp_idx][-1] - temps.conv_x[temp_idx][0]), inrp_det)
153.
154.        def plotTemp(temp, num_velocs):
155.
156.            """ PLots spectrograms from dictionary """
157.
158.            t = np.arange(temp.num)
159.
160.            for temp_idx in t:
161.                plt.figure(temp.names[temp_idx])
162.                plt.errorbar(temp.conv_x[temp_idx],temp.raw_y[temp_idx], temp.err_y[temp_idx], colo
    r= "red", label = "Error bars" )
163.                plt.plot(temp.conv_x[temp_idx],temp.raw_y[temp_idx], "x", label = "Data Points")
164.                plt.plot(temp.conv_x[temp_idx],temp.mov_avg_y[temp_idx], label = "Moving Average")
165.                plt.plot(temp.conv_x[temp_idx],temp.spln_y[temp_idx], label = "Spline")
166.                formatGraph(temp.names[temp_idx],"Wavelength(m)","Intensity")
167.
168.                plt_rng = temp.wind_size/2
169.
170.                plt.figure()
```

```
171.                plt.plot(temp.conv_x[temp_idx][plt_rng:-plt_rng],temp.diff_y[temp_idx][plt_rng:-
    plt_rng], "x", label = "Difference")
172.                formatGraph(temp.names[temp_idx],"Wavelength(m)","Intensity")
173.
174.        def plotSpct(spect, temp, scle_fcts, num_velocs):
175.
176.            """ PLots spectrograms from dictionary """
177.
178.            s = np.arange(spect.num)
179.
180.            for spct_idx in s:
181.                plt.figure(spect.names[spct_idx])
182.                plt.errorbar(spect.conv_x[spct_idx],spect.raw_y[spct_idx], spect.err_y[spct_idx], c
    olor= "red", label = "Error bars" )
183.                plt.plot(spect.conv_x[spct_idx],spect.raw_y[spct_idx], "x", label = "Data Points")

184.                plt.plot(spect.conv_x[spct_idx],spect.mov_avg_y[spct_idx], label = "Moving Average"
    )
185.                plt.plot(spect.conv_x[spct_idx],spect.spln_y[spct_idx], label = "Spline")
186.                formatGraph(spect.names[spct_idx],"Wavelength(m)","Intensity")
187.
188.                plt.figure()
189.                plt.plot(spect.conv_x[spct_idx],spect.diff_y[spct_idx], "x", label = "Difference")

190.                formatGraph(spect.names[spct_idx],"Wavelength(m)","Intensity")
191.
192.                #plt.plot(spect.conv_x[data_idx], spect.itrp_y[data_idx][0][268]*scle_fcts[data_idx
    ][0][268])
193.
194.        def scleSpct(data, temp, data_err):
195.            return sum((data*temp/(data_err**2)))/sum((temp)**2/(data_err**2))
196.
197.        def chiSqr(y,x,err_y,A):
198.            return np.sum(((y - (A*x))/err_y)**2)
199.        def formatGraph(g_title,x_title,y_title):
200.
201.            """ Ease of life function for graph formating """
202.
203.            plt.title( g_title )
204.            plt.xlabel( x_title )
205.            plt.legend(loc='best')
206.            plt.ylabel(y_title)
207.            plt.grid()
208.
209.        def sinCurve(gamma, kx, ky, phase):
210.            return gamma + kx*np.sin(2*np.pi*phase) + ky*np.cos(2*np.pi*phase)
211.
212.        def sigmaErr(gamma, kx, ky, phase, veloc, N):
213.                return np.sqrt((1./(N-3.))*np.sum((veloc - sinCurve(gamma, kx, ky, phase))**2))
214.
215.        def calcStlrMass(prd,rad_veloc,rad_veloc_err,grav_cnst):
216.            mass = (prd*rad_veloc**3)/((2*np.pi*grav_cnst))
217.            err = ((3*prd*(rad_veloc**2))/(2*np.pi*grav_cnst))*rad_veloc_err
218.            return mass, err
219.
220.        """ ~~~~~~~~~~ Class Setup ~~~~~~~~~~ """
221.
222.        class Spect:
223.
224.            names = ""
225.            num = []
226.
227.            dir_name = ""
228.            ext = ""
229.
230.            raw_x = []
231.            conv_x = []
232.
233.            shft_x = []
234.
```

```
            raw_y = []; mov_avg_y = []; spln_y = []; diff_y = []

            err_y = []; mov_avg_y_err = []; spln_err_y = [];

            wind_size = 0; spln_size = 0

            int_fn = []

            itrp_x = []
            itrp_y = []


        spcts = Spect()
        temps = Spect()

        """ ~~~~~~~~~~ Variables ~~~~~~~~~~ """

        spcts.dir_name = "./gs2000" #<--- Directory location of spectrum files.
        temps.dir_name = "./keck_temp" #<--- Directory location of template files.

        grav_cnst = 6.67508E-11

        spcts.ext = ".txt"
        temps.ext = ".txt"

        spcts.wind_size = 200 #<--- GS200 window size.
        spcts.spln_size = 300

        temps.wind_size = 157 #<--- Template window size.
        temps.spln_size = 220

        c_veloc = 299792458

        min_veloc = -600E3
        max_veloc = 600E3

        num_velocs = 2000
        grav_cnst = 6.67508E-11

        prd = 0.3440915*86400
        prd_err = 0.01

        sol_mass = 1.988E30

        org_phse = np.array([-0.1405,-
    0.0583,0.0325,0.0998,0.1740,0.2310,0.3079,0.3699,0.4388,0.5008,0.5698,0.6371,0.7276])

        """ ~~~~~~~~~~ Calculations ~~~~~~~~~~ """

        spcts = readSpct(spcts)
        temps = readSpct(temps)

        spcts = convAngToMet(spcts)
        temps = convAngToMet(temps)

        spcts = calcSpctAvg(spcts)
        temps = calcSpctAvg(temps)

        spcts = calcSpctSpln(spcts)
        temps = calcSpctSpln(temps)

        velocs = np.linspace(min_veloc,max_veloc,num_velocs)
        temps = calcSpctShft(temps, spcts, velocs, c_veloc, num_velocs, 2000)

        s = np.arange(spcts.num)
        v = np.arange(num_velocs)
        t = np.arange(temps.num)

        scle_fcts = np.zeros((temps.num, spcts.num, num_velocs))
        chi_sqrd = np.zeros((temps.num, spcts.num, num_velocs))
```

```
304.
305.       min_points = np.zeros((temps.num, spcts.num))
306.       chi_min_veloc = np.zeros((temps.num, spcts.num))
307.       chi_err = np.zeros((temps.num, spcts.num))
308.
309.       for temp_idx in t:
310.           for spct_idx in s:
311.               print(spct_idx)
312.               for veloc_idx in v:
313.                   scle_fcts[temp_idx][spct_idx][veloc_idx] = scleSpct(spcts.diff_y[spct_idx][200:
    -200], spcts.itrp_y[temp_idx][spct_idx][veloc_idx][200:-200], spcts.err_y[spct_idx][200:-200])
314.                   chi_sqrd[temp_idx][spct_idx][veloc_idx] = chiSqr(spcts.diff_y[spct_idx][200:-
    200], spcts.itrp_y[temp_idx][spct_idx][veloc_idx][200:-200], spcts.err_y[spct_idx][200:-
    200], scle_fcts[temp_idx][spct_idx][veloc_idx] )
315.
316.               min_points[temp_idx][spct_idx] = min(chi_sqrd[temp_idx][spct_idx])
317.               chi_min_veloc[temp_idx][spct_idx] = velocs[np.argmin(chi_sqrd[temp_idx][spct_idx])]

318.
319.               chiFunc = it.InterpolatedUnivariateSpline(velocs, chi_sqrd[temp_idx][spct_idx])
320.               temp_veloc = np.linspace(min_veloc, max_veloc, 10000)
321.               temp_chi = chiFunc(temp_veloc)
322.               chi_err[temp_idx][spct_idx] = np.abs(chi_min_veloc[temp_idx][spct_idx] - max(temp_v
    eloc[np.argwhere(temp_chi < (min_points[temp_idx][spct_idx] + 1))]))
323.
324.       chi_wgts = 1/(chi_err**2)
325.       plotSpct(spcts, temps, scle_fcts, num_velocs)
326.       plotTemp(temps, num_velocs)
327.
328.       for temp_idx in t:
329.           for spct_idx in s:
330.               plt.figure()
331.               plt.plot(velocs, chi_sqrd[temp_idx][spct_idx])
332.               formatGraph(spcts.names[spct_idx], "Velocity(ms-1)", "Chi Squared")
333.
334.       num_spcts = len(chi_min_veloc[0])
335.       num_temps = 1
336.
337.       s = np.arange(num_spcts); t = np.arange(num_temps)
338.
339.       rad_phse = 2*np.pi*org_phse
340.
341.       #~~~~~~~~~~~~ Question 4 ~~~~~~~~~~~~#
342.
343.       s_val = np.sin(rad_phse); c_val = np.cos(rad_phse); w_veloc = chi_min_veloc*chi_wgts
344.
345.       A = np.sum(chi_wgts), np.sum(chi_wgts*s_val), np.sum(chi_wgts*c_val)
346.       B = np.sum(chi_wgts*s_val), np.sum(chi_wgts*s_val**2), np.sum(chi_wgts*s_val*c_val)
347.       C = np.sum(chi_wgts*c_val), np.sum(chi_wgts*c_val*s_val), np.sum(chi_wgts*c_val**2)
348.
349.       RHS = np.sum(w_veloc), np.sum(w_veloc*s_val), np.sum(w_veloc*c_val)
350.       LHS = np.array([A,B,C])
351.
352.       gamma,kx,ky = np.linalg.solve(LHS, RHS)
353.
354.       eig = np.linalg.eig(LHS)
355.
356.       err_mat = np.diag(eig[0])
357.       inv = np.linalg.inv(err_mat)
358.
359.       gamma_err = np.sqrt(inv[0][0])
360.       kx_err = np.sqrt(inv[1][1])
361.       ky_err = np.sqrt(inv[2][2])
362.
363.       arr_phse = np.linspace(min(org_phse), max(org_phse), 1000)
364.       sin_fit = sinCurve(gamma, kx, ky, arr_phse)
365.
366.       fit_err = sigmaErr(gamma,kx,ky,org_phse,chi_min_veloc, num_spcts)
367.
368.       print("Fit Error:", '{0:.3f}'.format(fit_err))
```

```
369.
370.       print("Gamma:", '{0:.3f}'.format(gamma), "+-", '{0:.3f}'.format(gamma_err))
371.       print("kx:", '{0:.3f}'.format(ky), "+-", '{0:.3f}'.format(kx_err), "ms-1")
372.       print("ky:", '{0:.3f}'.format(gamma), "+-", '{0:.3f}'.format(ky_err), "ms-1")
373.
374.       for temp_idx in t:
375.           plt.figure("m0")
376.           plt.plot(org_phse, chi_min_veloc[temp_idx], "x", label = "Chi minimised velocity")
377.           plt.errorbar(org_phse, chi_min_veloc[temp_idx], chi_err[temp_idx], fmt = "None", label
    = "Chi minimised velocity error")
378.           plt.plot(arr_phse, sin_fit, label = "Fitted sin cuve")
379.           plt.savefig("m0_veloc")
380.
381.           formatGraph("","Phaze","Velocity (ms-1)")
382.
383.       #~~~~~~~~~~~~~ Question 5 ~~~~~~~~~~~~~# b
384.
385.       rad_veloc = np.hypot(kx,ky)
386.       rad_veloc_err = np.hypot(((ky/rad_veloc)*ky_err),((kx/rad_veloc)*kx_err))
387.
388.       print("Radial Velocity:", '{0:.3f}'.format(rad_veloc/1000), "+-
    ", '{0:.3f}'.format(rad_veloc_err/1000), "kms-1")
389.
390.       #~~~~~~~~~~~~~ Question 6 ~~~~~~~~~~~~~#
391.
392.       mass_fn, mass_fn_err = calcStlrMass(prd, rad_veloc, rad_veloc_err, grav_cnst)
393.       mass_fn_s = mass_fn/sol_mass; mass_fn_err_s = mass_fn_err/sol_mass
394.
395.       print("Mass Function:", '{0:.3f}'.format(mass_fn_s), "+-
    ", '{0:.3f}'.format(mass_fn_err_s), "solar masses")
396.
397.       #~~~~~~~~~~~~~ Question 7 ~~~~~~~~~~~~~#
398.
399.       mass_c = np.mean([0.45,0.8])
400.       mass_c_err = 0.8 - mass_c
401.
402.       def calcMass(mass_fn, mass_c, i):
403.           coeff = np.array([np.sin(i)**3, -mass_fn, -2*mass_fn*mass_c, -mass_fn*mass_c**2])
404.           roots = np.roots(coeff)
405.           return np.real(roots[np.isreal(roots)])[0]
406.
407.       mass_x = calcMass(mass_fn_s, mass_c, np.pi/2)
408.       mass_err_lrg = calcMass((mass_fn_s + mass_fn_err_s), mass_c, np.pi/2)
409.       mass_err_sml = calcMass((mass_fn_s - mass_fn_err_s), mass_c, np.pi/2)
410.
411.       mass_x_err = np.max([abs(mass_x - mass_err_lrg), abs(mass_x - mass_err_sml)])
412.
413.       print("Mass Object:", '{0:.3f}'.format(mass_x), "+-
    ", '{0:.3f}'.format(mass_x_err), "solar masses")
414.
415.       #~~~~~~~~~~~~~ Question 8 ~~~~~~~~~~~~~#
416.
417.       num_smpls = 10000
418.
419.       i_tri = np.random.normal(np.deg2rad(48), np.deg2rad(11), num_smpls)
420.
421.       #uniform(0.2,np.pi - 0.2, num_smpls)
422.       mc_tri = np.random.normal(mass_c,mass_c_err, num_smpls)
423.       rad_veloc_tri = np.random.normal(rad_veloc, rad_veloc_err, num_smpls)
424.       mass_fn_tri, mass_fn_err_arr = calcStlrMass(prd, rad_veloc_tri, rad_veloc_err, grav_cnst)
425.
426.       smpl_mass_x = np.zeros(len(i_tri))
427.       for smpl_idx in np.arange(len(i_tri)):
428.           smpl_mass_x[smpl_idx] = calcMass(mass_fn_tri[smpl_idx]/sol_mass,mc_tri[smpl_idx],i_tri[
    smpl_idx])
429.
430.       mean = np.mean(smpl_mass_x)
431.       med = np.median(smpl_mass_x)
432.       std = np.std(smpl_mass_x)
433.
```

```
434.        plt.figure()
435.        plt.hist(smpl_mass_x, bins = 2000, normed = True)
436.        formatGraph("","Mass (solar masses)", "Normalised prrobability")
437.        plt.xlim(0,40)
438.
439.        plt.figure()
440.        plt.hist(smpl_mass_x, bins = 2000, cumulative = True, normed = True)
441.        hist, bins = np.histogram(smpl_mass_x, bins = 10000, normed = True)
442.        plt.xlim(0,40)
443.        formatGraph("","Mass (solar masses)", "Cumulative prrobability")
444.
445.
446.        normal_x = np.linspace(0,40,1000)
```