Section 4.1

In this section we want to create our master dark frame and our master flat field so we can calibrate our raw data.

In this first box we are simply importing all of our modules and ensuring that we are in the correct directory to load in our files (so this would change per person)

In [2]:
```
import numpy as np
import matplotlib.pyplot as plt
from astropy.io import fits
from scipy import stats
import os

cwd = os.getcwd()
if cwd[-11:] != "Lab_2_again":
    os.chdir("/Users/efrainmartinez/Downloads/SBU/SBU_Spring_2024_Semester/AST4
```

In this nex box we are importing in the 10 dark frames that were taken for the 30 second exposure times. We also import the data for our flat fields that were taken with a 0.5 second exposure time.

In [ ]:
```
dark_30s_prefix = "dark_30sec_.00000"
dark_30s_end = ".DARK.FIT"
dark_30s_data = []
for i in range(285, 295, 1):
        filename = dark_30s_prefix + str(i) + dark_30s_end
        list = fits.open('dark_frames/'+filename)
        image_data = list[0].data
        dark_30s_data.append(image_data)

flat_ad_prefix = "flat_field_ad_.0000000"
flat_ad_end = ".FIT"
flat_ad_data = []
for i in range(0, 10):
    filename = flat_ad_prefix + str(i) + flat_ad_end
    list = fits.open('flat_fields/'+filename)
    image_data = list[0].data
    flat_ad_data.append(image_data)
```

In this next box we want to create our master dark and our master flat field images. We normalize our median combined flat field image by the median of the combined image, thus giving us values around 1.0

In [ ]:
```
median_dark_30s = np.median(dark_30s_data, axis=0)

median_flat_ad = np.median(flat_ad_data, axis=0)
median = np.median(median_flat_ad)
norm_flat_ad = median_flat_ad / median
```

In this box we are simply importing the data taken of Kepler-1. We create three different if statements for when the number of the image file becomes a two digit or three digit number.

In [ ]:
```
image_30s_prefix = "exposure_30sec.00000"
image_30s_end = ".FIT"
image_30s_data = []
for i in range(0, 284):
    if i < 10:
        filename = image_30s_prefix + "00" + str(i) + image_30s_end
        list = fits.open('images/'+filename)
        image_data = list[0].data
        image_30s_data.append(image_data)
    elif i<100:
        filename = image_30s_prefix + "0" + str(i) + image_30s_end
        list = fits.open('images/'+filename)
        image_data = list[0].data
        image_30s_data.append(image_data)
    else:
        filename = image_30s_prefix + str(i) + image_30s_end
        list = fits.open('images/'+filename)
        image_data = list[0].data
        image_30s_data.append(image_data)

with fits.open('images/exposure_30sec.00000000.FIT') as hdul:
               print(repr(hdul[0].header))
```

Now we want to calibrate our data with our master dark frame and our master flat field. We first subtract out our dark frame to eliminate the effect from the dark current in each frame, and then we can divide by our master flat field to get our final calibrated data.

In [ ]:
```
final_30s_data = []

for i in range(0, len(image_30s_data)):
    final_30s_data.append((image_30s_data[i]-median_dark_30s) / norm_flat_ad)

flattened = final_30s_data[0].flatten()
mean_final = np.mean(flattened)
std_final = stats.tstd(flattened)
plt.imshow(final_30s_data[0], cmap = 'gray', vmin = mean_final-3*std_final, vma
plt.colorbar()
```

Now that we have our calibrated data, we ensure that we are in the directory of out repository called "new_calibrated_fits_files". We then run through the calibrated image data for each file and rewrite it to another FITS file. If the file already exists in the directory we overwrite the data.

In [ ]:
```
cwd = os.getcwd()
print(cwd)
if cwd[-25:] != "new_calibrated_fits_files":
    os.chdir("/Users/efrainmartinez/Downloads/SBU/SBU_Spring_2024_Semester/AST4
    cwd = os.getcwd()
print(cwd)
if os.path.exists("calib_30s.000.FIT") != True:
    for i in range(0, len(final_30s_data)):
        image_30s_prefix = "calib_30s."
```

```
        image_30s_suffix = ".FIT"
        if i < 10:
            hdu = fits.PrimaryHDU(final_30s_data[i])
            filename = image_30s_prefix + "00" + str(i) + image_30s_suffix
            hdu.writeto(filename, overwrite=True)
        elif i<100:
            hdu = fits.PrimaryHDU(final_30s_data[i])
            filename = image_30s_prefix + "0" + str(i) + image_30s_suffix
            hdu.writeto(filename, overwrite=True)
        else:
            hdu = fits.PrimaryHDU(final_30s_data[i])
            filename = image_30s_prefix + str(i) + image_30s_suffix
            hdu.writeto(filename, overwrite=True)
```

4.2

The following bash script iterates over all of our science data, solving for WCS using **astrometry.net** and using XO-2N's RA/Dec as an initial guess. The same was done using Kepler-1's RA/Dec.

In [ ]:
```
#! /bin/bash -u

for file in $(ls -1 *.FIT)
do
   solve-field --ra 117.082 --dec 50.298 --radius 2 ${file}
done
```

4.3

We choose an aperature diameter of 18.882172 pixels. This decision was made by using **ds9**'s region functions and determining an area that encapsulates all of Kepler-1.

Firstly, the input files are sorted numerically so that they are input sequentially in the output data file. Then, we take the specific catalogue file number (e.g. '020' from calib_30s.020s.cat) and retrieve the date of observation from the corresponding fits file. The relevant data (i.e. flux and flux error) is read in from the catalogue file using a RA and Dec mask corresponding to the coordinates of the object of interest. If the object is not found in an image, a placeholder value of 0.001 is given for the flux and flux error for that file. The JD of observation, flux, and flux error are output into a data file. The process then repeats as the program loops through every catalogue file.

In [ ]:
```
import os
import numpy as np
from astropy.io import fits
from astropy.time import Time

# Get a list of all files in the directory
file_list = os.listdir('new_source_extractor')

# Filter and sort files numerically based on the number part before the extens
file_list = [file_name for file_name in file_list if file_name.startswith('cal
```

```
file_list.sort(key=lambda x: int(x.split('.')[-3].split('_')[-1]))
ts = []
# Open the data file for writing
with open('kepler1.dat', 'w') as f:
    # Iterate over each file in the directory
    for file_name in file_list:
        # Extract file number
        file_number = file_name.split('.')[-3].split('_')[-1]

        # Extract time of observation from FITS header
        fits_file_path = os.path.join('../images', 'exposure_30sec.00000' + fi
        with fits.open(fits_file_path) as hdul:
            header = hdul[0].header
            time_of_observation = header['DATE-OBS']

        t = Time(time_of_observation, format='fits', scale='utc')
        t_plot = t.jd

        # Load data from file
        data = np.loadtxt(os.path.join('new_source_extractor', file_name))

        # Extract columns from the data
        index = data[:,0]
        right_ascensions = data[:, 3]
        decs = data[:, 4]
        flux = data[:, 5]
        flux_err = data[:, 6]

        # Define masks for right ascensions and declinations
        ra_mask = (right_ascensions < 286.81) & (right_ascensions > 286.795)
        dec_mask = (decs < 49.33) & (decs > 49.31)

        # Combine masks using logical AND
        combined_mask = ra_mask & dec_mask

        # Apply the combined mask to get the indices where the condition is Tru
        indices = np.where(combined_mask)

        # Extract the values based on the combined mask
        xo_flux = flux[indices[0]]
        xo_flux_err = flux_err[indices[0]]

        if len(xo_flux) == 0:
            xo_flux = np.array([0.001])
            xo_flux_err = np.array([0.001])


        #print(file_name, index[indices], xo_flux, xo_flux_err)

        # Stack flux and flux error into columns

        stacked_data = np.column_stack((t_plot,xo_flux, xo_flux_err))

        # Write the file name, time of observation, and the stacked data to the

        np.savetxt(f, stacked_data)
```

In [2]:

```
import matplotlib.pyplot as plt
import numpy as np
#286.8 49.32
```

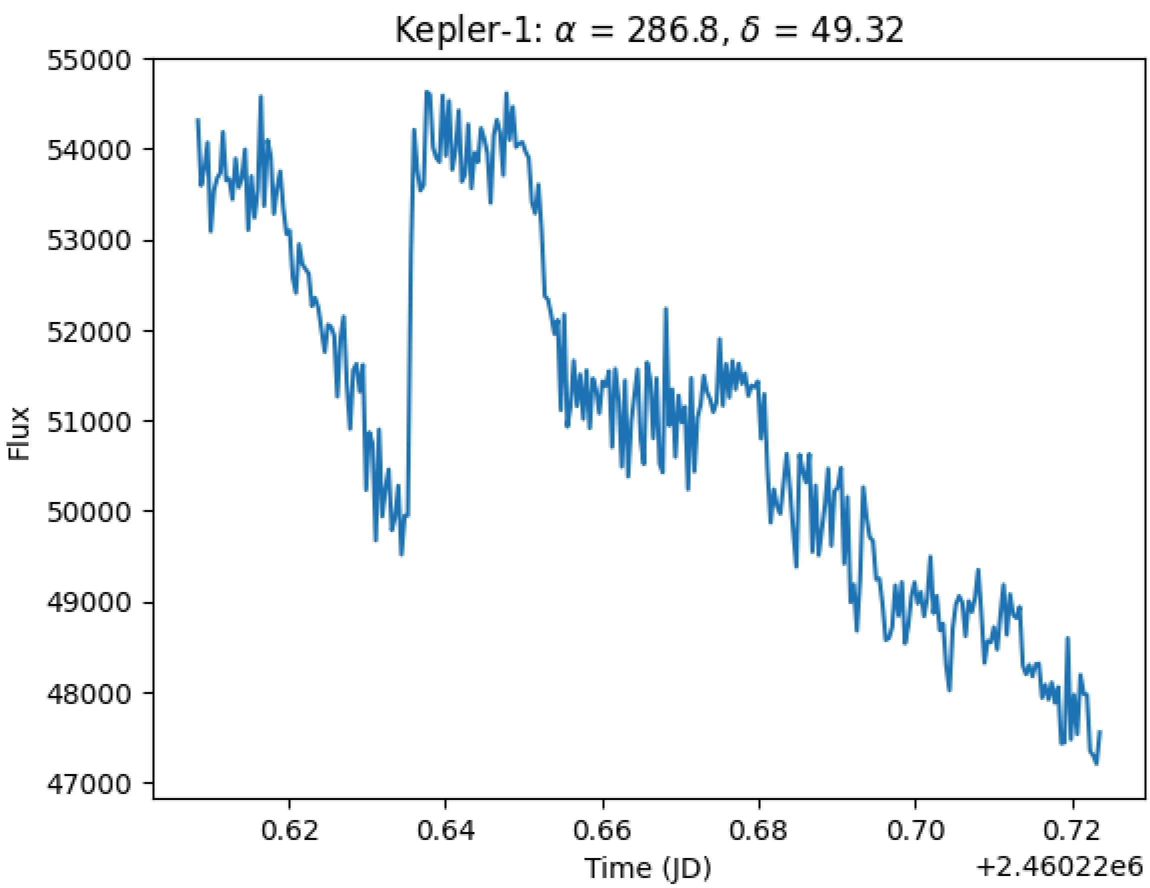
```
alpha = 286.8
dec = 49.32

data = np.loadtxt('NEW_SECTION_4_3/kepler1.dat')

time = data[:,0]
flux = data[:,1]
plt.xlabel('Time (JD)')
plt.ylabel('Flux')
plt.plot(time,flux[:])
plt.title(rf'Kepler-1: $\alpha$ = {alpha}, $\delta$ = {dec}')
```

Out[2]:

```
Text(0.5, 1.0, 'Kepler-1: $\\alpha$ = 286.8, $\\delta$ = 49.32')
```



Section 4.4

In this first box what I'm doing is just importing the data for our target star and each of the refence stars and saving the JD, flux, and flux errors in the respective arrays. We also print the header on the data file to gather information such as the filter used, the exposure time, and when the image was taken

In [ ]:

```
import numpy as np
import matplotlib.pyplot as plt
from astropy.io import fits

JDs = []
flux = []
```

```
flux_err = []

with open("NEW_SECTION_4_3/kepler1.dat") as file:
    lines = file.readlines()
    column0 = []
    column1 = []
    column2 = []
    for x in lines:
        column0.append(float(x.split(' ')[0]))
        column1.append(float(x.split(' ')[1]))
        column2.append(float(x.split(' ')[2]))
    JDs.append(column0)
    flux.append(column1)
    flux_err.append(column2)

for i in range(1, 11):
    filename = "NEW_SECTION_4_3/ref" + str(i) + ".dat"
    with open(filename, 'r') as file:
        lines = file.readlines()
        column0 = []
        column1 = []
        column2 = []
        for x in lines:
            column0.append(float(x.split(' ')[0]))
            column1.append(float(x.split(' ')[1]))
            column2.append(float(x.split(' ')[2]))
        JDs.append(column0)
        flux.append(column1)
        flux_err.append(column2)

print(repr(fits.open('images/exposure_30sec.00000283.FIT')[0].header))
```

In this box we're just eliminating the points in the data where no flux was recorded, and so the value for that value was replaced with a place holder of 0.001. So when we look at our data we run through the values and removed the flux, flux error, and JD associated with the exact 0.001 value.

In [ ]:

```
JDs_no_0 = []
flux_no_0 = []
flux_err_no_0 = []
for i in range(0, 11):
    temp_JD = []
    temp_flux = []
    temp_flux_err = []
    for j in range(0, len(flux[i])):
        if flux[i][j] != 0.001 and flux[i][j] > 0.:
            temp_JD.append(JDs[i][j])
            temp_flux.append(flux[i][j])
            temp_flux_err.append(flux_err[i][j])

    JDs_no_0.append(temp_JD)
    flux_no_0.append(temp_flux)
    flux_err_no_0.append(temp_flux_err)
```

Now that we have the correct data, we calculated the mean flux for each of the stars over the total observation period. We save this in `ave_flux` and then rescale our flux and flux error by dividing by the each set of data by its respective average flux. We have to do it a little differently since all of the arrays don't necessarily have the same shape, and so there was a bit of issues with list comprehension. To work around this rather than using the `np.mean` function we did it manually.

In [ ]:
```
ave_flux_no_0 = [sum(flux_no_0[x]) / len(flux_no_0[x]) for x in range(0, 11)]
scaled_flux_no_0 = [np.asarray(flux_no_0[x]) / ave_flux_no_0[x] for x in range
scaled_flux_err_no_0 = [np.asarray(flux_err_no_0[x]) / ave_flux_no_0[x] for x :
```

Here I am plotting each lightcurve separately so that we can analyze each one. We plot it here with the errorbars to see if any of the stars have more than random variability to them as well. From the look of the curves they all seem to follow the same pattern, having a lot of the same peaks and dips. Once we have scaled the fluxes we see they tend to have the around the same values as well.

In [ ]:
```
for i in range(0, 11):
#     plt.figure(figsize=(20, 10))
    plt.xlabel('Time (JD)')
    plt.plot(JDs[i],scaled_flux_no_0[i])
    plt.errorbar(JDs[i], scaled_flux_no_0[i], yerr=scaled_flux_err_no_0[i], lin
    plt.ylabel('Flux')
    if i == 0:
        plt.title("Kepler-1")
    else:
        plt.title(rf'REF {i}')
    plt.show()
```

Here I'm plotting the first five reference stars together and then the next 5 reference stars together, sepearating each of them by a step so that we can see each of them separately. Now it's really plain to see how the lightcurves all look similar and have similar peaks and dips.

In [ ]:
```
#This is just to stack the plots together
step = 0.3
for i in [0, 5]:
    scaled_flux_no_0[i+1] += step
    scaled_flux_no_0[i+2] += 2*step
    scaled_flux_no_0[i+3] += 3*step
    scaled_flux_no_0[i+4] += 4*step
    scaled_flux_no_0[i+5] += 5*step

for j in [0, 5]:
    plt.figure(figsize=(10, 6))
    for i in range(1, 6):
        i += j
        time = (np.asarray(JDs_no_0[i]) - JDs_no_0[i][0])
        centered_time = np.asarray(time - 0.05231309686350435) * 24
        plt.xlabel('Time from Mid-Transit (hours)')
        plt.plot(centered_time, scaled_flux_no_0[i][:len(time)], label=rf"REF -
        plt.ylabel('Scaled Flux')
        plt.title(rf'Reference Star Light Curves')
    time = (np.asarray(JDs_no_0[0]) - JDs_no_0[0][0])
    centered_time = np.asarray(time - 0.05231309686350435) * 24
    plt.plot(centered_time,scaled_flux_no_0[0], label="TrES-2b")
```

```
    plt.legend(loc='upper right', framealpha=1)
    plt.show()
```

Section 4.5

In this section our goal is to take the calibrated data of TrES-2b and the reference stars and begin to reduced them further to get our finalized lightcurve of the transit

In this first box we are starting with what we did in the previous section, importing in our files containing our data from SExtractor and saving them into the arrays flux, flux_err, and JDs.

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from astropy.io import fits

JDs = []
flux = []
flux_err = []

with open("NEW_SECTION_4_3/kepler1.dat") as file:
    lines = file.readlines()
    column0 = []
    column1 = []
    column2 = []
    for x in lines:
        column0.append(float(x.split(' ')[0]))
        column1.append(float(x.split(' ')[1]))
        column2.append(float(x.split(' ')[2]))
    JDs.append(column0)
    flux.append(column1)
    flux_err.append(column2)

for i in range(1, 11):
    filename = "NEW_SECTION_4_3/ref" + str(i) + ".dat"
    with open(filename, 'r') as file:
        lines = file.readlines()
        column0 = []
        column1 = []
        column2 = []
        for x in lines:
            column0.append(float(x.split(' ')[0]))
            column1.append(float(x.split(' ')[1]))
            column2.append(float(x.split(' ')[2]))
        JDs.append(column0)
        flux.append(column1)
        flux_err.append(column2)
```

Like in the last section, we need to go through and find all the values that are equal to 0.001 (no flux was given from SExtractor) or are less than 0 (no negative flux here). We removed these values along with the associated error and JD of that point.

In [ ]:
```
index = 0
while index < 11:
```

```
    j = 0
    for j in range(0, len(flux[index][:])):
        if j >= len(flux[index][:]):
            break
        if flux[index][j] == 0.001 or flux[index][j] < 0.:
            JDs[index].pop(j)
            flux[index].pop(j)
            flux_err[index].pop(j)
            j -= 1
    if 0.001 in flux[index]:
        index -= 1
    index += 1
```

Now that we have the correct data, we want to calculate the weighted means and errors of our reference stars so that we can calibrate our data properly. Since some points may have been removed if SExtractor didn't return a flux value, we ensure that the values we are calculating the means for all have the same JDs by using separate index values for each star. That way if one star has a flux point at a certain JD but not another, the star has its own index to keep it on track.

In [ ]:

```
# Here we want to line up our data so that when we calculate the weighted means
# mixing our points together

ind_targ = ind_1 = ind_2 = ind_2 = ind_3 = ind_4 = ind_5 = ind_6 = ind_7 = ind_
ind = [ind_targ, ind_1, ind_2, ind_3, ind_4, ind_5, ind_6, ind_7, ind_8, ind_9

weighted_means = []
weighted_errs = []

while np.max(ind) < len(JDs[0]):
    temp_flux = []
    temp_flux_err = []
    temp_flux = np.array(temp_flux)
    temp_flux_err = np.array(temp_flux_err)
    date = JDs[0][ind[0]]

    for i in range(1, 11):
        if ind[i] >= len(JDs[i]):
            break
        if JDs[i][ind[i]] == date:
            temp_flux = np.append(temp_flux, flux[i][ind[i]])
            temp_flux_err = np.append(temp_flux_err, flux_err[i][ind[i]])
            ind[i] += 1

    ind[0] += 1

    if len(temp_flux_err) != 0:
        w_mean = sum(temp_flux / (temp_flux_err**2)) / sum(1. / temp_flux_err*
        w_err = np.sqrt(1. / sum(1. / temp_flux_err**2))
    else:
        w_mean = 1.0
        w_err = 1.0
    weighted_means.append(w_mean)
    weighted_errs.append(w_err)
#     print(f"Weighted mean is {w_mean}")
```

Now in this box we are utilizing the weighted means we just calculated. We divide the flux of Kepler-1 by the weighted means to get a "corrected" lightcurve, and we use our error propagation for divison to calculate the new errors.

In [ ]:
```
r_s = []
r_s_err = []
for i in range(0, len(flux[0])):
    r_s.append(flux[0][i] / weighted_means[i])
    r_s_err.append(np.abs(flux[0][i] / weighted_means[i]) *
                   np.sqrt((flux_err[0][i]/flux[0][i])**2 + (weighted_errs[i]/w
```

Now in this box we want to normalize the flux of Kepler-1 to a baseline flux. We take a slice of the fluxes before the transit and calculate the median value. From there we normalize the flux and the errors to this value, and since it is a constant we take the error propagation as such.

In [ ]:
```
pre_transit = 15

pre_flux = r_s[:pre_transit]
median_flux = np.median(pre_flux)
flux_norm = r_s / median_flux
flux_norm_err = r_s_err / median_flux
```

Now that we have the final normalized data and the statistical uncertainty with it, we can plot the lightcurve for Kepler-1, centering the plot around our mid-transit time (which is calculated in the next section).

In [ ]:
```
time = (np.asarray(JDs[0])-JDs[0][0])*24 - 1.25551
plt.plot(time, flux_norm,'o')
plt.errorbar(time, flux_norm, linestyle='', c='r', yerr=flux_norm_err)
plt.title("Kepler 1 Lightcurve")
plt.ylabel("Normalized Flux")
plt.xlabel("Time from Mid-Transit (hours)")
plt.ylim(0.8, 1.2)
plt.show()
```

With all of the values we have calculated, we export them to a file called "section_4_5.csv" so that we can import these values to calculate the transit parameters.

In [ ]:
```
import csv

ind_targ = ind_1 = ind_2 = ind_2 = ind_3 = ind_4 = ind_5 = ind_6 = ind_7 = ind_
ind = [ind_targ, ind_1, ind_2, ind_3, ind_4, ind_5, ind_6, ind_7, ind_8, ind_9

headers = ['DATE-OBS', 'Target Flux', 'Target Flux Error', 'Rescaled Ref Fluxe
data = zip(JDs[0], flux[0], flux_err[0], weighted_means, r_s, r_s_err, flux_no

with open('section_4_5.csv', 'w') as file:
    writer = csv.writer(file)
    writer.writerow(headers)
    for row in data:
    #     stacked_data = np.column_stack((JDs[0], flux[0], flux_err[0], weight
        writer.writerow(row)
```

Section 5

In this section we want to utilize the values we have calibrated to calculate the parameters of the transit (duration, depth, planetary radius).

In this first box we are simply reading the csv written from Section 4.5 and importing the JDs, normalized fluxes, and their errors.

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import csv
from scipy import optimize
import batman

JDs = []
flux_norm = []
flux_norm_err = []

with open('section_4_5.csv', 'r') as file:
    reader = csv.reader(file)
    header = []
    header = next(reader)

    for row in reader:
        JDs.append(float(row[0]))
        flux_norm.append(float(row[6]))
        flux_norm_err.append(float(row[7]))
```

This box is where we calculate all of the important values of our transit. We do this with help from the **batman** model from Laura Kreidberg. The **batman** model takes in parameters for the exoplanet transit and creates a model of the lightcurve based on these parameters: mid-transit time, period, planet radius, semi-major axis, orbital inclination (degrees), eccentricity, longitude of periastron (degrees), limb darkening coefficients. Since we don't know the exact values of these parameters, we want to find the parameters that fit our data the best. To do this we define the function `chi_squared` and use `scipy.optimize.minimize` to find the parameters that minimize `chi_squared`, thus finding the parameters that create a model that best fits our data. From these values we only take the mid-transit parameter, as the planetary radius is based solely on the model and not our data, and the other values are not relevant.

To calculate the transit depth we utilize the points that are distributed around the mid-transit time given by the model and find their average. We also calculated the error on this value by calculating the systematic error that would be necessary for our model to be a good fit. This is all described more in the lab reports.

To calculate the transit duration we look at the model flux again. The **batman** model creates a flux that is equal to 1 in all points except for when the transit is occuring, so we take the transit duration to be the time difference from the two points where the model flux is not equal to 1 at the beginning and end of the transit

In [ ]:

```
time = np.asarray(JDs)-JDs[0]

def chi_squared(params):
    t0, per, rp, a, inc, ecc, w, u1, u2 = params

    # Transit parameters
    transit_params = batman.TransitParams()
    transit_params.t0 = t0
    transit_params.per = per
    transit_params.rp = rp
    transit_params.a = a
    transit_params.inc = inc
    transit_params.ecc = ecc
    transit_params.w = w
    transit_params.u = [u1, u2]
    transit_params.limb_dark = "quadratic"

    # Transit model
    transit_model = batman.TransitModel(transit_params, time)
    model_flux = transit_model.light_curve(transit_params)

    # Calculate sum of squared differences
    return np.sum((flux_norm - model_flux)**2 / model_flux)

# Initial guess for parameters
initial_guess = [0.05, 0.5, 0.1, 15.0, 87.0, 0.0, 90.0, 0.1, 0.3]


sigs = []
```

```
# Minimize squared difference to converge to parameters
result = optimize.minimize(chi_squared, initial_guess, method="Powell")

# Extract the parameters
optimized_params = result.x

# Plot observed data and model
transit_params = batman.TransitParams()
transit_params.t0 = optimized_params[0]
transit_params.per = optimized_params[1]
transit_params.rp = optimized_params[2]
transit_params.a = optimized_params[3]
transit_params.inc = optimized_params[4]
transit_params.ecc = optimized_params[5]
transit_params.w = optimized_params[6]
transit_params.u = [optimized_params[7], optimized_params[8]]
transit_params.limb_dark = "quadratic"

transit_model = batman.TransitModel(transit_params, time)
model_flux = transit_model.light_curve(transit_params)

# Print optimized parameters
print("Powell Method\n")
print(f"Model Mid Transit Time:       {optimized_params[0]*24:.5f} hours")

plt.figure(figsize=(10, 6))
for i in range(0, len(model_flux)):
    if model_flux[i] != 1.0:
        plt.axvline((time[i] - transit_params.t0)*24, c='r', linestyle='--')
        transit_start = time[i]
        break
for i in range(len(model_flux)-1, 0, -1):
    if model_flux[i] != 1.0:
        plt.axvline((time[i] - transit_params.t0)*24, c='r', linestyle='--')
        transit_stop = time[i]
        break

print(f"Transit Start:          {transit_start * 24:.5f} hours")
print(f"Transit Stop:           {transit_stop * 24:.5f} hours")
print(f"Transit Duration:       {(transit_stop-transit_start) * 24:.5f} hours"

centered_time = np.asarray(time - optimized_params[0]) * 24
plt.axvline(0.0, c='r', linestyle='--')
plt.scatter(centered_time, flux_norm, label='Data')
plt.plot(centered_time, model_flux, c='black', label='Model', linewidth=3)
plt.ylim(0.9, 1.1)
plt.xlabel('Time from Mid-Transit (hours)')
plt.ylabel('Normalized Flux')
plt.title('Transit Light Curve')

#-------------------------------------------------
t0 = optimized_params[0]

for i in range(0, len(time)):
    if time[i] < t0 and time[i+1] > t0:
        mid_start = i-3
        mid_end = i+4
        mid_flux = flux_norm[mid_start:mid_end]
        mid_time = time[mid_start:mid_end]
        mid_model_flux = model_flux[mid_start:mid_end]
```

```
mid_var = np.sum((np.asarray(mid_flux) - np.asarray(mid_model_flux))**2) / (len
mid_sigma_sys = np.sqrt(np.abs(np.asarray(mid_var)**2 - np.asarray(sigma_stat[m
mid_sigma_tot = np.sqrt(np.asarray(flux_norm_err[mid_start:mid_end])**2 + mid_s
mid_sigma = mid_sigma_tot[3] / np.sqrt(len(mid_flux))
mid_flux_value = np.sum(mid_flux) / len(mid_flux)

depth = 1-mid_flux_value
planet_radius = np.sqrt(depth) * 9.73116
depth_x_rad_err = depth * np.sqrt(0.036**2 + (mid_sigma / depth)**2) * 9.73116
rad_err = 0.5 * depth**(-0.5) * depth_x_rad_err

print(f"\nThese values were calculated with our data near the calculated mid t
print(f"Transit Depth: {depth:.10f} +/- {mid_sigma:.10f}")
print(f"Planet Radius: {planet_radius:.10f} +/- {rad_err:.10f} Jupiter Radii")

plt.errorbar(centered_time, flux_norm, yerr=sigma_tot, capsize=3, capthick=2, 
# plt.scatter(mid_time - t0, mid_flux, c='r')
plt.legend()
plt.show()
```