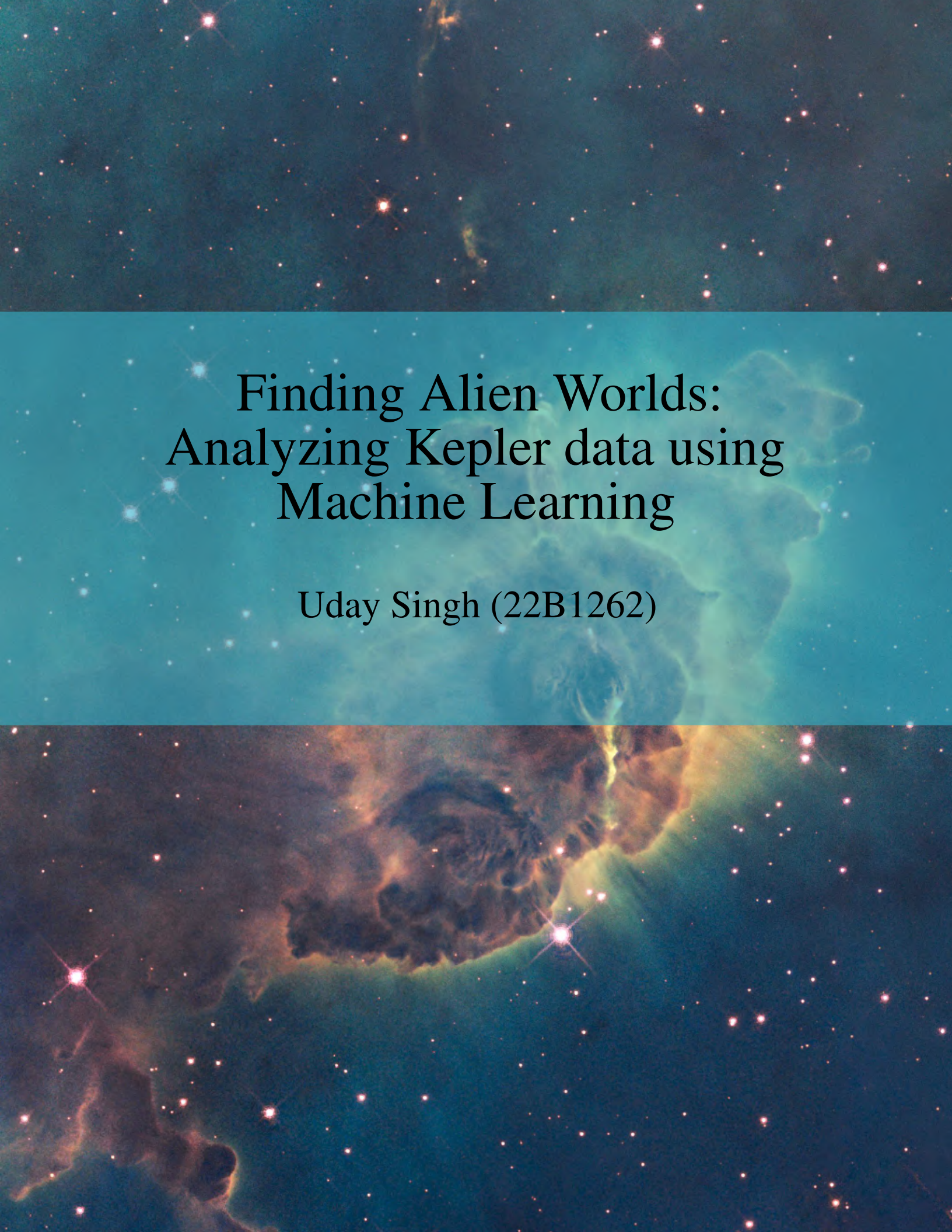


Finding Alien Worlds: Analyzing Kepler data using Machine Learning

Uday Singh (22B1262)

Finding Alien Worlds: Analyzing Kepler data using Machine Learning

Krittika Computational Astronomy Projects

Author: Uday Singh

Mentor: Suryansh Srijan

Second Mentor: Shreyas Kulkarni

Contents

1 Introduction 5

1.1 What are Exoplanets? 5

1.2 Why to Study them? 5

1.3 References 6

2 Transit Method for Exoplanet Detection 7

2.1 What is a Transit? 7

2.2 Limb Darkening 8

2.3 Transit Light Curves 9

3 Plotting a Light Curve using Kepler data 11

3.1 Getting the Data 11

3.2 FITS files 12

3.3 Plotting a Light Curve 13

3.4 Finding Time Period by Method 1 14

3.5 Finding Time Period by Method 2 17

4 Neural Network for Exoplanet Detection 19

4.1 Reading and Filtering Training Data 19

4.2 Neural Network Model 21

4.3 Testing the Model 23
4.4 Using Model on Kepler data for Exoplanet Detection 24

5 Simulation of Solar System 27
5.1 Euler-Richardson Algorithm 27
5.2 Simulation for 3 Planets and 1 Star 29
5.3 Light curve of one body Observed from another body 31

1. Introduction

1.1 What are Exoplanets?

Exoplanets or Extrasolar Planets, as the name suggests, are planets, beyond our solar system.

What does that really mean though? To get a clear idea, we need to know what a planet actually is. According to the **IAU (International Astronomical Union)**, to be called a Planet a celestial entity:

- Must be in orbit around a star
- Has sufficient mass for its self-gravity to overcome rigid body forces so that it assumes a hydrostatic equilibrium (nearly round) shape
- Has cleared the neighborhood around its orbit (The entity is massive enough to throw any other comparable-size objects out of its orbit, such that its orbit is “cleaned up” of said objects)

Since Pluto does not fulfill these conditions, It is not considered a planet.

Pluto is technically a "**dwarf planet**,” because it has not “cleared its neighboring region of other objects.” This means that Pluto still has lots of asteroids and other space rocks along its flight path, rather than having absorbed them over time, as the larger planets have done.

This is the currently accepted definition and is subject to change as debate and discoveries continue.

1.2 Why to Study them?

Other than the mind-numbing notion that we’re standing on but one of the trillions of such roughly spherical wanderers hidden in plain sight, Everything we think we know about the origins of Planets, each with their own composition, atmosphere, cycles, and systems are constantly being challenged as we discover these new worlds.

We know the Earth has limited resources. Largely science fiction at this point, but if we were to leave an uninhabitable earth in the future, if we could, where would we go?

Are we alone in the Universe? A question so famous, yet unanswered! We’re the first generation who has the tools at our fingertips to begin to answer this question with scientific observations, and turns out we’re lucky enough to witness the first steps of this cosmic-scale search.

To answer these big questions, we must know where to start, and that's what we're going to discuss here. We'll look at the beginning of the exoplanet hunt, the methods of detection developed over time, as well as the mathematics involved with them.

1.3 References

A few links to get you interested (In case words weren't enough ;-;)

- `https://www.youtube.com/watch?v=EUU0-ZpFoK4`
- `https://exoplanets.nasa.gov/alien-worlds/strange-new-worlds/`
- `https://exoplanets.nasa.gov/alien-worlds/exoplanet-travel-bureau/`

2. Transit Method for Exoplanet Detection

2.1 What is a Transit?

The study of exoplanets through Transits is the most successful method, mainly because of the successes of the **Kepler and TESS NASA missions**.

A transit (or astronomical transit) is a phenomenon when a celestial body passes directly between a larger body and the observer. As viewed from a particular point, the transiting body appears to move across the face of the larger body, covering a small portion of it.

When it comes to exoplanets, transits look nothing like the Transit of Venus, because we can't see a faraway star from limb to limb (end to end). So what do we see? as the transiting body passes in front of the larger light source, there is a drop in the brightness or flux received from the star due to the occulting. We see this in the form of transit light curves.

The increasing majority of transiting planets are being found from dedicated wide-angle searches. Since there is little to indicate a priori which stars may have planets, which of those that do might be oriented favourably for a transit to be observed, and when or how frequently such transits may occur, surveys simply monitor large numbers of stars, simultaneously and for long periods of time, searching for the tiny periodic drops in intensity that might be due to transiting planets. It may take multiple years to confirm transiting exoplanets.

The effect being sought is also small: a planet with R almost equal to RJ transiting a star of 1R results in a drop of the star flux of (ΔF/F) = 0.011 , or around 0.01 mag. For planets of Earth or Mars radius, ΔF = 0.000084 and 0.00003 respectively. Depths of up to 7 percent might occur for M dwarfs.

Ground-based searches are able to discover transits with depths up to about (ΔF/F) = 1%, revealing gas-giant planets around stars frequently bright enough for radial-velocity confirmation and mass measurements with 2m-class telescopes, or for the study of their atmospheric transmission and emission spectra from space-based observations. Surveys from space, beyond the effects of atmospheric seeing and scintillation, are discovering planets with transit depths of a few times, 10^4.

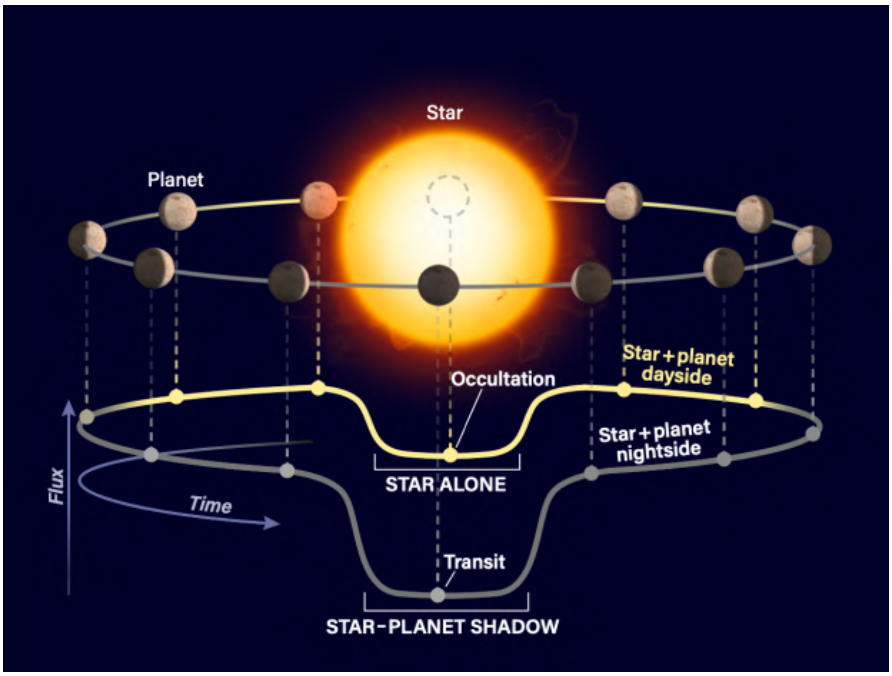


Figure 2.1: LightCurve of a Star

2.2 Limb Darkening

When you look at a light curve of a transit, you may expect to see a box-like structure, with sharp drops at the edges, but if you look at a transit at different wavelengths, you'll see that its shape curves slightly. The figure below shows the transit of a planet around the star HD 209458 at wavelengths ranging from about 3000 Angstroms (purple, at bottom) to about 10000 Angstroms (red, at top).

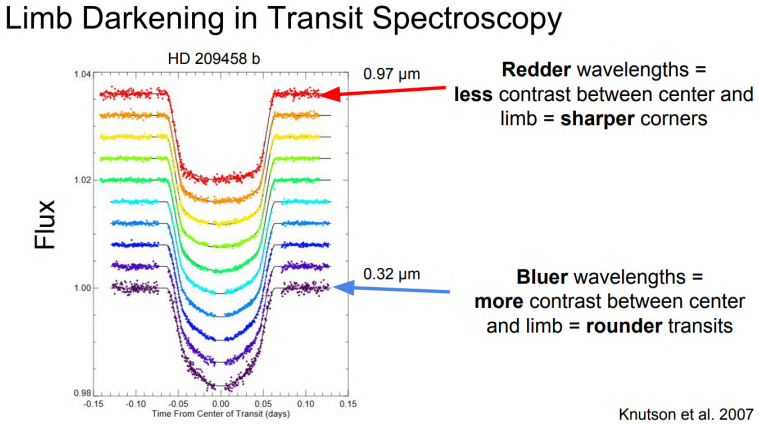


Figure 2.2: Effects fo Limb darkening on transmission spectrum

You might have noticed, that for shorter wavelengths, other than the obvious decrease in overall intensity emitted, the dip isn't as "sharp" as it is for larger wavelengths.

When you look at a star in the middle, the light you're seeing is from a deeper part of the star

which emits higher energy light. This part of the star is called the **photosphere**, and it is the deepest part of the star which is still transparent to certain visible photons and as you move outwards, the peripheral emissions of the star comprise larger wavelengths in a higher percentage, the outer layers of the sun radiate lower energy light, thus the dropoff in the intensity of higher frequency light is smoother as you go outwards, hence, as the transiting planet blocks this wavelength at the "limb" (periphery), it doesn't lead to as sharp a drop as red light.

One important feature of limb darkening is that it depends on the wavelength at which one observes: shorter wavelengths lead to stronger limb darkening, because of the stronger gradient in emitted radiation between the inner and outer layers of the sun.

The dip in the light created by a transit shows us a sort of mirror image of the surface brightness of the star: the dip is deeper where the surface is brighter.

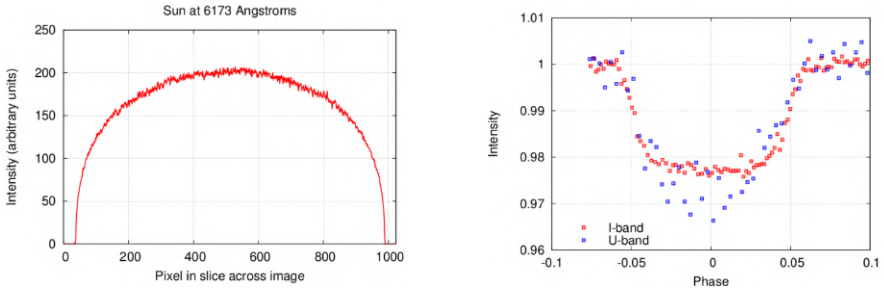


Figure 2.3: Limb darkening and transit samples from Kepler Mission(Left) and TESS Mission(Right)

2.3 Transit Light Curves

Aside from detection, there are a number of things that can be inferred from the properties of the transit light curve:

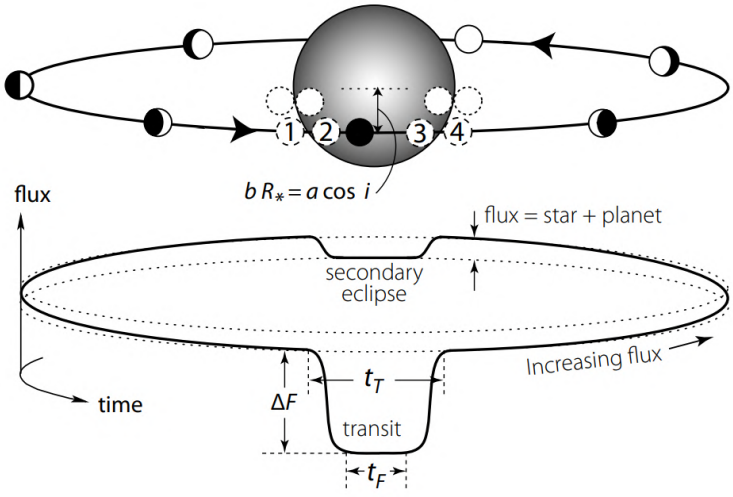


Figure 2.4: Transit Schematic

A light curve is a plot of flux versus time that shows the variability of light output from an object. This is one way to find planets **periodically transitting a star**.

There are four principle observables which characterise the duration and profile of the primary transit: the period P , the transit depth ΔF , the interval between the first and fourth contacts tT (Total Transit Duration) and the interval between the second and third contacts tF .

From these, three geometrical equations together describe the principle features of the transit light curve

The first follows from the ratio of the areas of the projected disks of the planet and star. The total transit time follows from the fraction of the orbital period P during which the projected distance between the centres of the star and planet is less than the sum of their radii. The formula for tF is derived similarly.

Using these formulae, coupled with Kepler's 3rd law, many properties of the star and planet can be estimated, such as stellar and planet density, inclination and even planet surface gravity! When combined with data from other observation techniques, such as radial velocity measurements or spectroscopy, additional details about the exoplanet's composition, atmosphere, and potential habitability can be inferred.

3. Plotting a Light Curve using Kepler data

Defining some terms:

- **Cadence:** the frequency with which summed data are read out. Files are either short cadence (a 1 minute sum) or long cadence (a 30 minute sum).
- **SAP Flux:** Simple Aperture Photometry flux; flux after summing the calibrated pixels within the optimal aperture
- **PDCSAP Flux:** Pre-search Data Conditioned Simple Aperture Photometry; these are the flux values nominally corrected for instrumental variations.
- **BJD:** Barycentric Julian Day; this is the Julian Date that has been corrected for differences in the Earth's position with respect to the Solar System Barycentre (center of mass of the Solar System).
- **HDU:** Header Data Unit; a FITS file is made up of Header or Data units that contain information, data, and metadata relating to the file. The first HDU is called the primary, and anything that follows is considered an extension.

3.1 Getting the Data

After importing required libraries like matplotlib ,Astropy and Astroquery , we need to find the data file.

```
keplerObs = Observations.query_criteria(target_name='kplr011446443', obs_collection='Kepler')
keplerProds = Observations.get_product_list(keplerObs[1])
yourProd = Observations.filter_products(keplerProds, extension='kplr011446443-2009131110544_slc.fits',
                                        mrp_only=False)
yourProd
```

Figure 3.1: kplr011446443 database

This is similar to searching for the data using the MAST Portal in that we will be using certain

keywords to find the file. The target name of the object we are looking for is kplr011446443, collected by the Kepler spacecraft. Now that we've found the data file, we can download it using the results of the above cell.

Now that we have the file, we can start working with the data. We will begin by assigning a shorter name to the file to make it easier to use. Then, using the info function from astropy.io.fits, we can see some information about the FITS Header Data Units:

```
12]: filename = "./mastDownload/Kepler/kplr011446443_sc_Q113313330333033302/kplr011446443-2009131110544_slc.fit
     fits.info(filename)

Filename: ./mastDownload/Kepler/kplr011446443_sc_Q113313330333033302/kplr011446443-2009131110544_slc.fits
No.    Name      Ver    Type      Cards   Dimensions   Format
  0  PRIMARY       1 PrimaryHDU      58   ()
  1  LIGHTCURVE    1 BinTableHDU    155   14280R x 20C   [D, E, J, E, E, E, E, E, E, J, D, E, D, E, D, E, D,
E, E, E]
  2  APERTURE      1 ImageHDU        48   (8, 9)   int32
```

Figure 3.2: FITS file

3.2 FITS files

This format is the standard data format used in astronomy, can contain one image, multiple images, tables and header keywords providing descriptive information about the data. The way it works is that this format can contain a text file with keywords that comprise the information about the observation and a multidimensional array that could be a table, or an image, or an array of images (data cube). This files can be managed in different ways, with an image preview use DS9, for handing the data in a program use the *Python* package *PyFITS*.

- **No. 0 (Primary):** This HDU contains meta-data related to the entire file.
- **No. 1 (Light curve):** This HDU contains a binary table that holds data like flux measurements and times. We will extract information from here when we define the parameters for the light curve plot.
- **No. 2 (Aperture):** This HDU contains the image extension with data collected from the aperture. We will also use this to display a bitmask plot that visually represents the optimal aperture used to create the SAPFLUX column in HDU1.

Headers and Data can be accessed using -

```
:  with fits.open(filename) as hdulist:
       header1 = hdulist[1].header

   print(repr(header1[0:24])) #repr() prints the info into neat columns

XTENSION= 'BINTABLE'           / marks the beginning of a new HDU
BITPIX  =                    8 / array data type
NAXIS   =                    2 / number of array dimensions
```

Figure 3.3: Accessing Headers

```
with fits.open(filename) as hdulist:
    binaryext = hdulist[1].data

binarytable = Table(binaryext)
binarytable[1:5]
```

Table length=4

TIME	TIMECORR	CADENCENO	SAP_FLUX	SAP_FLUX_ERR	SAP_BKG	SAP_BKG_ERR	PDCSAP_FLUX
float64	float32	int32	float32	float32	float32	float32	float32
120.52992386784899	0.00096672785	5501	401288.16	91.51187	2598.1086	0.5752603	406100.9
120.53060508973431	0.00096674974	5502	401425.53	91.53448	2598.0261	0.57525027	406242.22

Figure 3.4: Accessing Data

3.3 Plotting a Light Curve

Now that we have seen and accessed the data, we can begin to plot a light curve:

- Open the file using the command fits.open. This will allow the program to read and store the data we will manipulate to be plotted. Here we've also renamed the file with a phrase that is easier to handle (see line 1).
- Start by calibrating the time. Because the Kepler data is in BKJD (Kepler Barycentric Julian Day) we need to convert it to time in Julian Days (BJD) if we want to be able to compare it to other outside data. For a more detailed explanation about time conversions, visit the [page 13] or [page 17] of `https://archive.stsci.edu/kepler/manuals/archivemanual.pdf#page=13` of the Kepler Archive Manual. Read in the BJDREF times, both the integer (BJDREFI) and the floating point (BJDREFF). These are found as columns of data in the *binary extension* of the header.
- Read in the columns of times and fluxes (both uncorrected and corrected) from the data.

```
with fits.open(filename, mode="readonly") as hdulist:
    # Read in the "BJDREF" which is the time offset of the time array.
    bjdrefi = hdulist[1].header['BJDREFI']
    bjdreff = hdulist[1].header['BJDREFF']

    print(bjdrefi)
    print(bjdreff)

    # Read in the columns of data.
    times = hdulist[1].data['time']
    sap_fluxes = hdulist[1].data['SAP_FLUX']
    pdcsap_fluxes = hdulist[1].data['PDCSAP_FLUX']
```

Figure 3.5: Reading Flux and time

Now that the appropriate data has been read and stored, convert the times to BJDS by adding the BJDREF times to the data of times. Finally, we can plot the fluxes against time. We can also set a title and add a legend to the plot. We can label our fluxes accordingly and assign them colors and

styles ("-k" for a black line, "-b" for a blue line).

```
# Convert the time array to full BJD by adding the offset back in.
bjds = times + bjdrefi + bjdreff

plt.figure(figsize=(9,4))

# Plot the time, uncorrected and corrected fluxes.
plt.plot(bjds, sap_fluxes, '-k', label='SAP Flux')
plt.plot(bjds, pdcsap_fluxes, '-b', label='PDCSAP Flux')

plt.title('Kepler Light Curve')
plt.legend()
plt.xlabel('Time (days)')
plt.ylabel('Flux (electrons/second)')
plt.show()
```

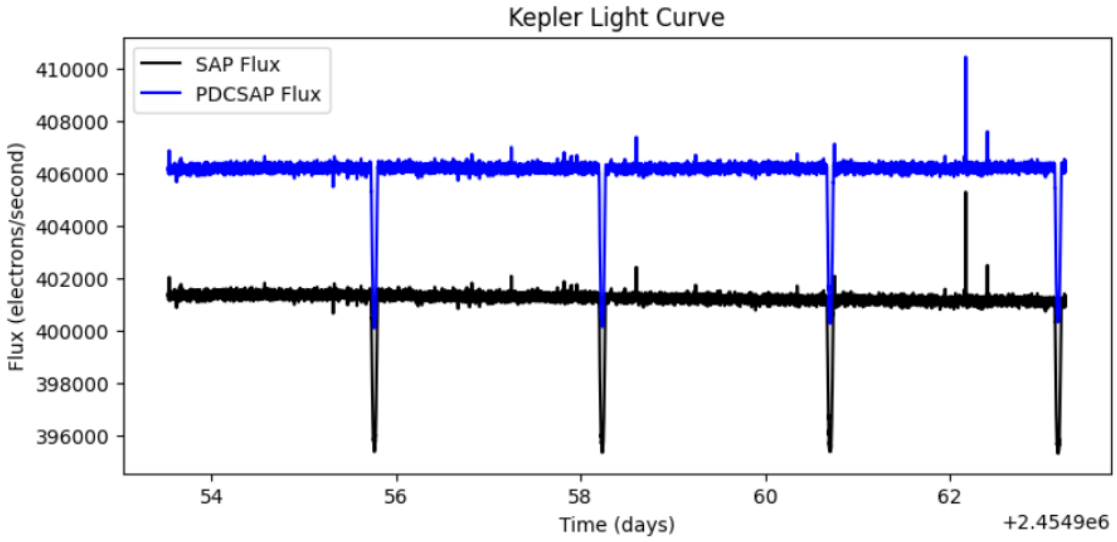


Figure 3.6: Final LightCurve

3.4 Finding Time Period by Method 1

To find the period quantitatively, using a Lomb-Scargle Periodogram seems slightly inappropriate; this light curve doesn't resemble a sinusoid in any way. Instead, we will use another Periodogram in astropy.timeseries called BoxLeastSquares.

The usage is similar to LombScargle; so it is left to you to find the period of the transits, and hence the orbital period of the exoplanet.

We will be using BoxLeastSqaures.

.

.

.

.

```
# There are some NaN values in the flux. So, we will find their indices to filter them out
sel_nan = np.isnan(sap_fluxes) #NaN = not a number
np.where(sel_nan==True)
# sap_fluxes[4064]
```

```
(array([ 4064,  4065,  4066,  4067,  4068,  4069,  4070,  4071,  4072,
         4073,  4074,  6679,  8414,  8415,  8416,  8417,  8418,  8419,
         8420,  8421,  8422,  8423,  8424, 11470, 11471, 11472, 11473,
        11474, 11475, 11476, 11477, 11478, 11479, 11480, 11481, 12147,
        12148], dtype=int64),)
```

```
from astropy.timeseries import BoxLeastSquares

# Implement BoxLeastSquares
model = BoxLeastSquares(bjds[~sel_nan], sap_fluxes[~sel_nan]) # ~ is the unary Not operator
periodogram = model.autopower(0.2)
```

```
# Plot the power Spectrum
periods = periodogram.period
power = periodogram.power

plt.figure(figsize=(8,6))
plt.plot(periods, power)
```

```
[<matplotlib.lines.Line2D at 0x252132e0f80>]
```

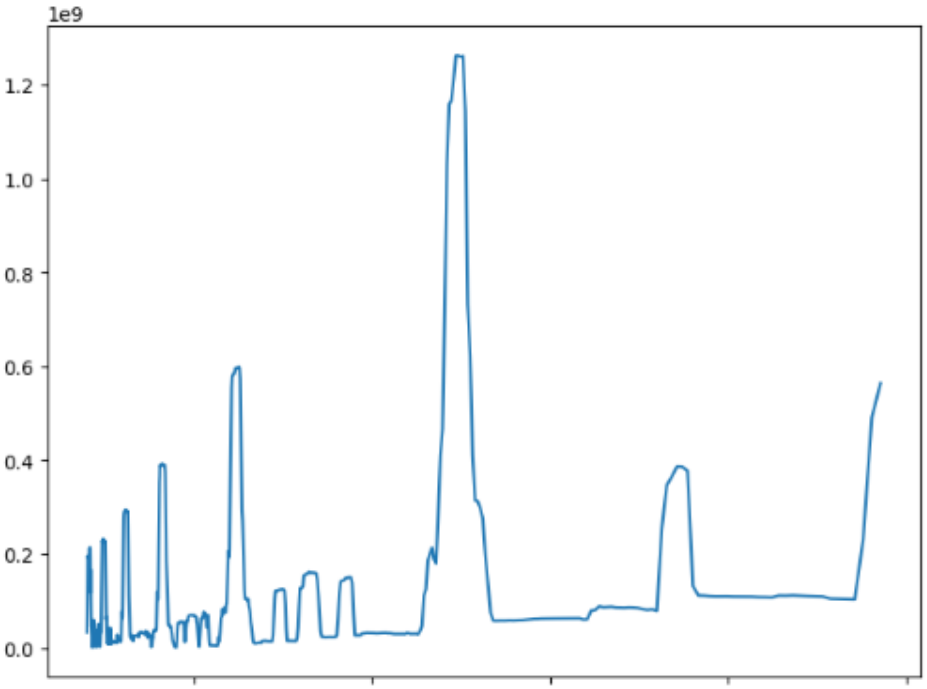


Figure 3.7:

```
best_period = periods[np.argmax(power)]
# Plot the Phase-Folded Light curve for the Exoplant Transit using the Period found.

# The added phase to the time is just to center the transit.
phase = np.remainder(bjds, best_period)/best_period

print(best_period)

plt.figure(figsize=(8,6))
plt.scatter(phase, sap_fluxes, s=1, color='k')
plt.xlabel("Phase")
plt.ylabel("Flux [e/s]")
```

```
2.4835596538949924
Text(0, 0.5, 'Flux [e/s]')
```

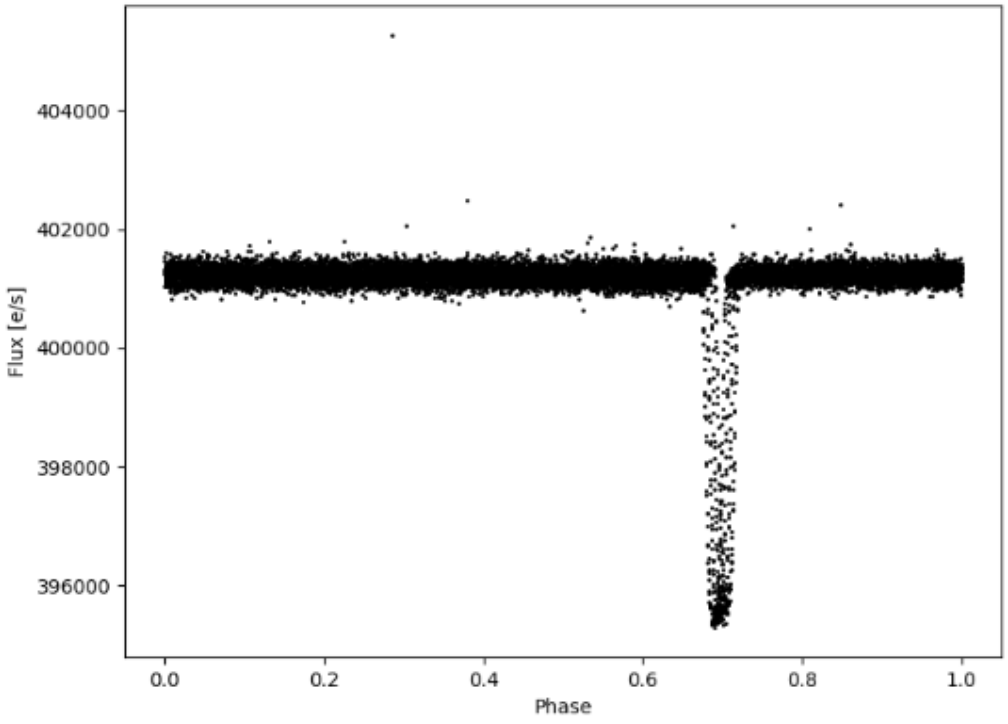


Figure 3.8: time period = 2.4835596538949924

3.5 Finding Time Period by Method 2

Time Period here is the difference between two consecutive dips in the light curve.

Divide the lightcurve into 4 parts and find the time at which the flux is minimum in that respective part. The difference between 2 consecutive min-time is the time period.

```
# method to find time period!
import numpy as np
print(sap_fluxes.size)
a = 14280*0.37
print(a)
b = 14280*0.62
print(b)
a = 5284
b = 8854
```

Figure 3.9:

```
f1 = sap_fluxes[0:a]
f2 = sap_fluxes[a+1:b]
value = np.where(f1==min(f1))
value
```

```
(array([3281], dtype=int64),)
```

```
t1 = bjds[3281]
t1
```

```
2454955.7640689346
```

```
value2 = np.where(f2==min(f2))
value2
```

```
(array([1623], dtype=int64),)
```

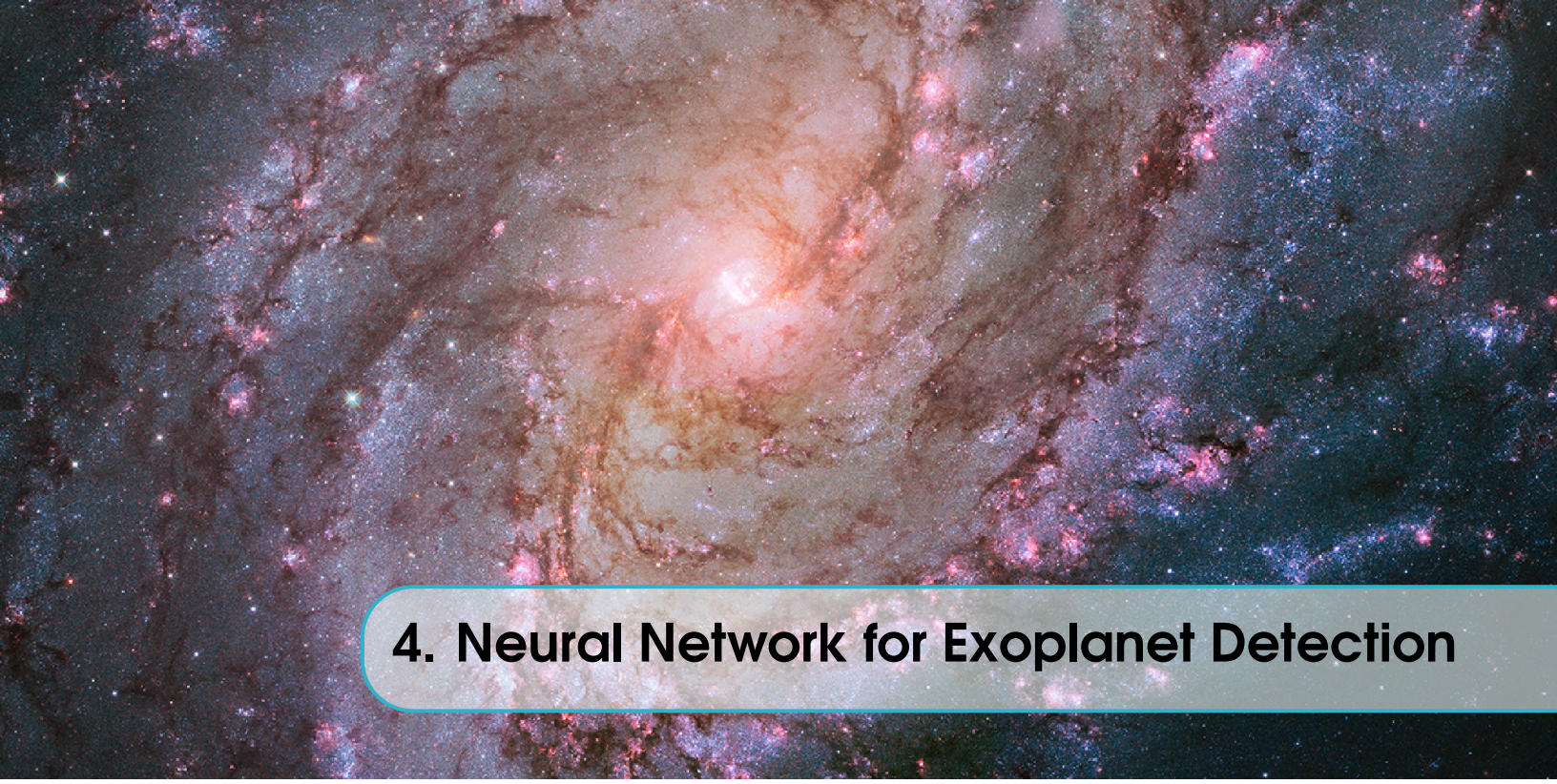
```
t2 = bjds[1623+1+5284]
t2
```

```
2454958.234569723
```

```
t2 - t1 # this is the time period!!!
```

```
2.4705007881857455
```

Figure 3.10: time period = 2.4705007881857455



4. Neural Network for Exoplanet Detection

Steps:

1. Filter and Normalise Training Data
2. Design a Neural Network
3. Test on Kepler Mission Data
4. Final Results and Possible Reasons

4.1 Reading and Filtering Training Data

First, filter all the nan values. Then apply mean filtering to all the rows(light curves) in the data for smoothening out light curves.

```
import pandas as pd
a = [5089,5091] # removing faulty rows
data = pd.read_csv("data.csv",header=None,skiprows=a)
data
```

	0	1	2	3	4	5	6	7	8	9	...	3188
0	1.0	9.385000e+01	8.381000e+01	2.010000e+01	-2.698000e+01	-3.956000e+01	-1.247100e+02	-1.351800e+02	-9.627000e+01	-7.989000e+01	...	-7.807000e+01
1	1.0	-3.888000e+01	-3.383000e+01	-5.854000e+01	-4.009000e+01	-7.931000e+01	-7.281000e+01	-8.655000e+01	-8.533000e+01	-8.397000e+01	...	-3.280000e+00

Figure 4.1: filter out the nan vlaues

In this filtered data we will also add Kepler mission data "KOI Q1 short" for training. It is assumed that all 10 light curves in kepler data are of exoplanets.

Normalizing(Scaling to 0-1) will be done inside the neural network model itself.

Final training data is formed by concatenating filtered data and Kepler KOI Q1 short.

.

.

.

.

```
#MEAN FILTERING OF DATA

k = 3  # Choose the value of k for deviations (2-4 as suggested)

# Exclude the first column from mean filtering
data_to_filter = data.iloc[:, 1:].copy()  # Make an explicit copy to avoid SettingWithCopyWarning

# Apply moving mean filter to the data excluding the first column
moving_mean = data_to_filter.rolling(window=5, axis=1, min_periods=1).mean()  # Use min_periods to avoid NaN

# Calculate the median and standard deviation
median = moving_mean.median(axis=1)
std_dev = moving_mean.std(axis=1)

# Thresholding: apply mean filtering for values within k*sigma deviations from the median
for i in range(len(data_to_filter)):
    deviation = np.abs(data_to_filter.iloc[i] - median[i])
    mask = deviation <= k * std_dev[i]
    filtered_values = moving_mean.iloc[i].where(mask, data_to_filter.iloc[i])  # Reverting to original if not within threshold
    data_to_filter.iloc[i] = filtered_values

# Reconstruct the DataFrame with the first column and the filtered data
filtered_data = pd.concat([data.iloc[:, 0], data_to_filter], axis=1)
```

Figure 4.2: Mean Filtering

```
for i in range(20):
    import matplotlib.pyplot as plt
    import numpy as np
    %matplotlib inline
    plt.figure(figsize=(12,4))

    plt.plot(list(filtered_data.iloc[i])[1:],'r', label='Flux')

    plt.title('Kepler Light Curve')
    plt.legend()
    plt.xlabel('Time (days)')
    plt.ylabel('Flux (electrons/second)')
    plt.show()
```

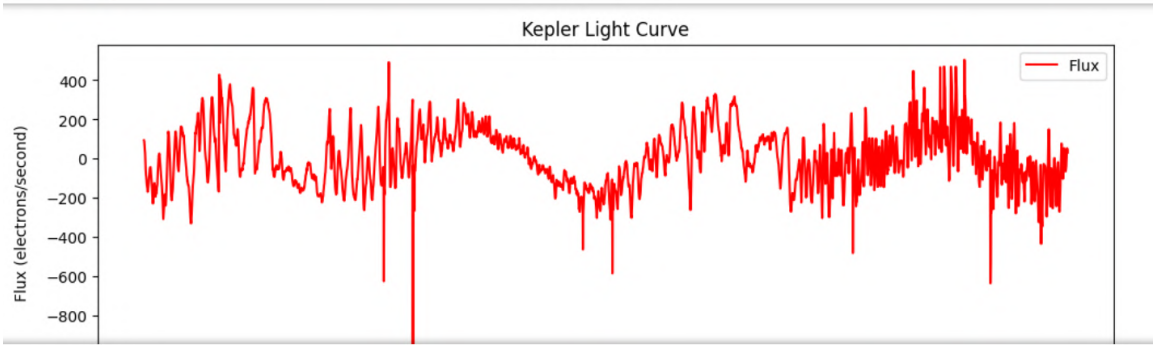


Figure 4.3: light curve after filtering

4.2 Neural Network Model

Model consists of 2 hidden layers , 1 input and 1 output layer. I have used MinMax Scaler to scale light curves between 0 to 1.

```
import pandas as pd  #to load data
import torch
import torch.nn as nn
from torch.utils.data import Dataset #to generate torch tensors and have torch compatible dataset
from sklearn.preprocessing import MinMaxScaler

# Creating a custom PyTorch Dataset
# The 'CustomDataset' class is our customized class for data preparation for our torch model. It is inherited from the parent abstract class Dataset.
# Dataset is the object type accepted by torch models.
class CustomDataset(Dataset):
    #constructor
    def __init__(self, csv_file):
                                                # defining labels(l) and features(f)
        f = csv_file.iloc[:,1:].values
        l = csv_file.iloc[:,0].values

        # Transpose the DataFrame to scale rows
        data_transposed = f.T
        # Initialize MinMaxScaler
        scaler = MinMaxScaler()
        # Scale each row individually
        scaled_values = scaler.fit_transform(data_transposed)

        # Remove the first column by index (0 here represents the first column)
        data_without_first_col = csv_file.drop(csv_file.columns[0], axis=1)

        # Transpose the scaled data back to the original orientation
        f_train = pd.DataFrame(scaled_values.T, columns=data_without_first_col.columns)
        # Convert DataFrame to NumPy array
        f_train = f_train.values
        l_train = l

        #converting to torch tensors
        self.F_train = torch.tensor(f_train, dtype = torch.float32)
        self.l_train = torch.tensor(l_train, dtype = torch.long)

    def __len__(self):
        return len(self.l_train)

    def __getitem__(self, idx):
        return self.F_train[idx], self.l_train[idx]


data_set = CustomDataset(filtered_data)

train_loader = torch.utils.data.DataLoader(dataset=data_set,
                                           batch_size=10,
                                           shuffle=True)

input_size =  3197        #layer1
hidden_size1 = 300        #hidden layer
hidden_size2 = 30
num_classes = 2           #last layer
num_epochs = 3
# batch_size = 10
learning_rate = 0.001
```

Figure 4.4: Data Prepration

```python
# Fully connected neural network with one hidden layer
class NeuralNet(nn.Module):
    def __init__(self, input_size, hidden_size1,hidden_size2,num_classes):
        super(NeuralNet, self).__init__()
        self.input_size = input_size
        self.l1 = nn.Linear(input_size, hidden_size1)
        self.relu = nn.ReLU()
        self.l2 = nn.Linear(hidden_size1, hidden_size2)
        self.l3 = nn.Linear(hidden_size2, num_classes)

    def forward(self, x):
        out = self.l1(x)
        out = self.relu(out)
        out = self.l2(out)
        out = self.relu(out)
        out = self.l3(out)

        # no activation and no softmax at the end
        return out

model = NeuralNet(input_size, hidden_size1,hidden_size2, num_classes)

# Loss and optimizer
criterion = nn.CrossEntropyLoss()  # this will apply solftmax for us
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)  # defines parameters and updates parameteres

# Train the model
n_total_steps = len(train_loader)
for epoch in range(num_epochs):
    for i, (features, labels) in enumerate(train_loader):
        # Forward pass
        outputs = model(features)
        loss = criterion(outputs, labels)

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if (i+1) % 100 == 0:
            print (f'Epoch [{epoch+1}/{num_epochs}], Step [{i+1}/{n_total_steps}], Loss: {loss.item():.4f}')

print('Model Trained!')
```

```
Epoch [1/3], Step [100/513], Loss: 0.0302
Epoch [1/3], Step [200/513], Loss: 0.1202
Epoch [1/3], Step [300/513], Loss: 0.0082
Epoch [1/3], Step [400/513], Loss: 0.0484
Epoch [1/3], Step [500/513], Loss: 0.0564
Epoch [2/3], Step [100/513], Loss: 0.0745
Epoch [2/3], Step [200/513], Loss: 0.0324
Epoch [2/3], Step [300/513], Loss: 0.0335
Epoch [2/3], Step [400/513], Loss: 0.0135
Epoch [2/3], Step [500/513], Loss: 0.4002
Epoch [3/3], Step [100/513], Loss: 0.0272
Epoch [3/3], Step [200/513], Loss: 0.0424
Epoch [3/3], Step [300/513], Loss: 0.0468
Epoch [3/3], Step [400/513], Loss: 0.0252
Epoch [3/3], Step [500/513], Loss: 0.0134
Model Trained!
```

Figure 4.5: Training Model on Data

4.3 Testing the Model

First load the Kepler data(FITS file) as shown in Chapter 3. I used the training data for testing itself and got an accuracy of 98.77049180327869%.

```
]: # Test the model
# In test phase, we don't need to compute gradients (for memory efficiency)

class testDataset(Dataset):
    #constructor
    def __init__(self,result_array):

        ftest = result_array.iloc[:,1:].values
        ltest = result_array.iloc[:,0].values

        # Transpose the DataFrame to scale rows
        data_transposed = ftest.T
        # Initialize MinMaxScaler
        scaler = MinMaxScaler()
        # Scale each row individually
        scaled_values = scaler.fit_transform(data_transposed)

        # Remove the first column by index (0 here represents the first column)
        data_without_first_col = result_array.drop(result_array.columns[0], axis=1)

        # Transpose the scaled data back to the original orientation
        f_test = pd.DataFrame(scaled_values.T, columns=data_without_first_col.columns)
        # Convert DataFrame to NumPy array
        f_test = f_test.values
        l_test = ltest

        #converting to torch tensors
        self.F_test = torch.tensor(ftest, dtype = torch.float32)
        self.l_test = torch.tensor(ltest, dtype = torch.long)

    def __len__(self):
        return len(self.l_test)

    def __getitem__(self, idx):
        return self.F_test[idx], self.l_test[idx]

data_set_test = testDataset(filtered_data)

test_loader = torch.utils.data.DataLoader(dataset=data_set_test,
                                          batch_size=5,
                                          shuffle=False)

with torch.no_grad():
    n_correct = 0
    n_samples = 0
    for features, labels in test_loader:

        outputs = model(features)
        # max returns (value ,index)
        _, predicted = torch.max(outputs.data, 1)
        n_samples += labels.size(0)
        n_correct += (predicted == labels).sum().item()
        if 1 in predicted:
            print(predicted)
            print(outputs)
    acc = 100.0 * n_correct / n_samples
    print(f'n_correct = {n_correct}')
    print(f'n_samples = {n_samples}')
    print(f'Accuracy of the network : {acc} %')
```

```
tensor([0, 0, 0, 0, 1])
tensor([[  557.7037,  -221.3559],
```

Figure 4.6: Testing the Model - 1

```
        outputs = model(features)
        # max returns (value ,index)
        _, predicted = torch.max(outputs.data, 1)
        n_samples += labels.size(0)
        n_correct += (predicted == labels).sum().item()
        if 1 in predicted:
            print(predicted)
            print(outputs)
    acc = 100.0 * n_correct / n_samples
    print(f'n_correct = {n_correct}')
    print(f'n_samples = {n_samples}')
    print(f'Accuracy of the network : {acc} %')
```

```
tensor([0, 0, 0, 0, 1])
tensor([[  557.7037,  -221.3559],
        [   22.8002,   -12.8237],
        [ 1312.2543, -1975.9498],
        [   10.3750,   -23.1964],
        [  -16.8857,   -15.6986]])
tensor([0, 1, 0, 0, 0])
tensor([[ 145.7410, -162.8263],
        [  -0.8046,   -0.4435],
        [  17.7974,  -15.4319],
        [  66.2252,  -21.5418],
        [ 194.2578, -265.3662]])
tensor([0, 0, 0, 1, 0])
tensor([[   7.8730,   -6.4084],
        [ 2728.4211, -4083.4561],
        [  132.4813,  -131.9061],
        [ -128.4061,    -7.0951],
        [   93.3761,   -56.4110]])
n_correct = 5061
n_samples = 5124
Accuracy of the network : 98.77049180327869 %
```

Figure 4.7: Testing the Model - 2

4.4 Using Model on Kepler data for Exoplanet Detection

Implementing the Neural Network model on Kepler data KOI Q12 long to check for any exoplanets. 1 is for an exoplanet and 0 is for not an exoplanet.

In the Snippets given below, we can see that according to our trained Neural Network model, we are detecting 0 exoplanets out of over 3000 light curve samples.

RESULTS:

- Zero detected exoplanets

POSSIBLE REASONS:

- The training data consisting of 5000 light curves have only 52 confirmed exoplanets, so the model is trained better at detecting non-exoplanet light curves than detecting exoplanets.
- The training data given has lots of noise which was not completely removed even after mean filtering.
- After plotting the first 20 lightcurves from the training data. I realized that not all of the light curves mentioned as exoplanets were exoplanets(since they didn't have periodic dips). Therefore training data is faulty.

```
acclist = []
import os
# from astropy.io import fits

# Directory containing the FITS files
folder_path = './KOI_Q12_long'

# Loop through files in the directory
for filename in os.listdir(folder_path):
    if filename.endswith('.fits'):  # Check if the file is a FITS file
        file_path = os.path.join(folder_path, filename)
        # Open the FITS file
        # print(filename)
        with fits.open(file_path, mode="readonly") as hdulist:
            sap_fluxes = hdulist[1].data['SAP_FLUX']

        sf_without_nan = sap_fluxes[~np.isnan(sap_fluxes)]

        a1 = sf_without_nan[:3197]
        l = np.array([1])
        a1r = a1.reshape(1, -1)  # Reshape array1 to a row
        lr = l.reshape(1, -1)
        result_array2 = np.concatenate((lr, a1r),axis=1)
        result_array = pd.DataFrame(result_array2)
        # result_array

        # import pandas as pd  #to load data
        # import torch
        # import torch.nn as nn
        # from torch.utils.data import Dataset #to generate torch tensors and have torch compatible dataset
        # from sklearn.preprocessing import StandardScaler #data splitting and pre-processing #Normalize
        # # from sklearn.preprocessing import MinMaxScaler
        # from sklearn.preprocessing import Normalizer

        # Creating a custom PyTorch Dataset
        # The 'CustomDataset' class is our customized class for data preparation for our torch model. It is inherited from the parent abstract class Da
        # Dataset is the object type accepted by torch models.

        class testDataset(Dataset):
            #constructor
            def __init__(self,result_array):

                ftest = result_array.iloc[:,1:].values
                ltest = result_array.iloc[:,0].values

                # Transpose the DataFrame to scale rows
                data_transposed = ftest.T
                # Initialize MinMaxScaler
                scaler = MinMaxScaler()
                # Scale each row individually
                scaled_values = scaler.fit_transform(data_transposed)

                # Remove the first column by index (0 here represents the first column)
                data_without_first_col = result_array.drop(result_array.columns[0], axis=1)

                # Transpose the scaled data back to the original orientation
                f_test = pd.DataFrame(scaled_values.T, columns=data_without_first_col.columns)
                # Convert DataFrame to NumPy array
                f_test = f_test.values
                l_test = ltest

                #converting to torch tensors
                self.F_test = torch.tensor(f_test, dtype = torch.float32)
                self.l_test = torch.tensor(l_test, dtype = torch.long)
```

Figure 4.8: Exoplanet Detection - 1

```
            # Convert DataFrame to NumPy array
            f_test = f_test.values
            l_test = ltest

            #converting to torch tensors
            self.F_test = torch.tensor(f_test, dtype = torch.float32)
            self.l_test = torch.tensor(l_test, dtype = torch.long)

        def __len__(self):
            return len(self.l_test)

        def __getitem__(self, idx):
            return self.F_test[idx], self.l_test[idx]

    data_set_test = testDataset(result_array)

    test_loader = torch.utils.data.DataLoader(dataset=data_set_test,
                                              batch_size=5,
                                              shuffle=False)


    # Test the model
    # In test phase, we don't need to compute gradients (for memory efficiency)

    with torch.no_grad():
        n_correct = 0
        n_samples = 0
        for features, labels in test_loader:

            outputs = model(features)
            # max returns (value ,index)
            _, predicted = torch.max(outputs.data, 1)
            n_samples += labels.size(0)
            n_correct += (predicted == labels).sum().item()

        acc = 100.0 * n_correct / n_samples
        # print(f'n_correct = {n_correct}')
        # print(f'n_samples = {n_samples}')
        # print(f'Accuracy of the network : {acc} %')
        acclist.append(acc)
print(acclist)
```

```
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
```

```
[ ]: 100.0 in acclist
     # print(len(acclist))
```

```
[ ]: False
```

Figure 4.9: Exoplanet Detection - 2

5. Simulation of Solar System

5.1 Euler-Richardson Algorithm

Is is better to compute the velocity at the middle of the interval rather than at the beginning or at the end. The Euler-Richardson algorithm is based on this idea. This algorithm is particularly useful for velocity-dependent forces but does as well as other simple algorithms for forces that do not depend on the velocity. The algorithm consists of using the Euler algorithm to find the intermediate position ymid and velocity vmid at a time tmid = t + Δt/2. We then compute the force, F(ymid, vmid, tmid) and the acceleration amid at t = tmid. The new position yn+1 and velocity vn+1 at time tn+1 are found using vmid and amid and the Euler algorithm.

We summarize the Euler-Richardson algorithm as:

$$a_n = \frac{F(y_n, v_n, t_n)}{m} \tag{5.1}$$

$$v_{\rm mid} = v_n + \frac{1}{2} a_n \Delta t \tag{5.2}$$

$$y_{\rm mid} = y_n + \frac{1}{2} v_n \Delta t \tag{5.3}$$

$$a_{\rm mid} = \frac{F(y_{\rm mid}, v_{\rm mid}, t + \frac{1}{2}\Delta t)}{m} \tag{5.4}$$

$$v_{n+1} = v_n + a_{\rm mid} \Delta t \tag{5.5}$$

$$y_{n+1} = y_n + v_{\rm mid} \Delta t \quad \text{(Euler-Richardson algorithm)} \tag{5.6}$$

Although we need to do twice as many computations per time step, the Euler-Richardson algorithm is much faster than the Euler algorithm because we can make the time step larger and still obtain better accuracy than with either the Euler or Euler-Cromer algorithms.

```python
# 3 Euler-Richardson Algorithm

class Particle3:
    def __init__(self, mass, initial_position, initial_velocity,dt):
        self.mass = mass
        self.position = initial_position
        self.velocity = initial_velocity
        self.dt = dt

    def force(self, position):
        # Define the force acting on the particle (Example: Simple harmonic motion F = -kx)
        k = 1  # Spring constant (for example)
        return -k * position

    def euler_richardson_step(self):
        # Calculate acceleration at the current position
        a_n = self.force(self.position) / self.mass
        v_mid = self.velocity +0.5*a_n*self.dt
        y_mid = self.position +0.5*self.velocity*self.dt
        a_mid = self.force(y_mid) / self.mass

        self.velocity = self.velocity + a_mid*self.dt
        self.position = self.position + v_mid*self.dt

        return self.position

# Initialize a particle
mass = 1
initial_position = 0.5
initial_velocity = 1
particle3 = Particle3(mass, initial_position, initial_velocity, 0.01)

l3 = []
for k in range(10000):
    l3.append(particle3.euler_richardson_step())

print(l3[-1])
```

```
-0.07334790451187599
```

```python
import matplotlib.pyplot as plt
%matplotlib inline
plt.figure(figsize=(12,2))

plt.plot( l3,'r', label='position')
plt.title('SHM')
plt.legend()
plt.xlabel('Time')
plt.ylabel('y')
plt.show()
```

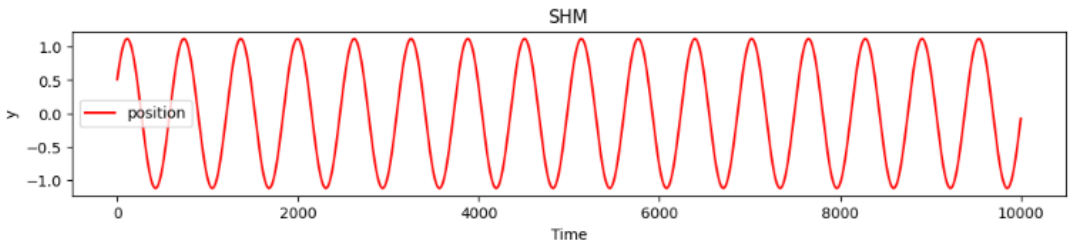


Figure 5.1: Euler Richardson Algorithm for a Particle in SHM

5.2 Simulation for 3 Planets and 1 Star

Simulating Venus, Earth, Mars, and the Sun. Taking into account Gravitational Forces between 4 bodies, their initial Positions, and their initial velocities.

Apparent Brightness is taken to be 1 / (distance ** 2).

For 4 BODY

```
]:
```

```
2]: import numpy as np
class sources():
    def __init__(self,m,x,y,ux,uy,dt):
        self.m = m
        self.ux = ux
        self.uy = uy
        self.x = x
        self.y = y
        self.dt = dt

    def force_x(self,m1,m2,m3,x,y,x1,y1,x2,y2,x3,y3):
        d1 = abs(((x1-x)**2 + (y1-y)**2)**0.5)
        d2 = abs(((x2-x)**2 + (y2-y)**2)**0.5)
        d3 = abs(((x3-x)**2 + (y3-y)**2)**0.5)
        angle = np.arctan2(y-y1, x-x1)
        angle2 = np.arctan2(y-y2, x-x2)
        angle3 = np.arctan2(y-y3, x-x3)
        return ((-1*6.67408*(10**-11)*self.m*m1)/(d1)**2)* np.cos(angle)+ ((-1*6.67408*(10**-11)*self.m*m2)/(d2)**2)*np.cos(angle2)+((-1*6.67408*(10**-1

    def force_y(self,m1,m2,m3,x,y,x1,y1,x2,y2,x3,y3):
        d1 = abs(((x1-x)**2 + (y1-y)**2)**0.5)
        d2 = abs(((x2-x)**2 + (y2-y)**2)**0.5)
        d3 = abs(((x3-x)**2 + (y3-y)**2)**0.5)
        angle = np.arctan2(y-y1, x-x1)
        angle2 = np.arctan2(y-y2, x-x2)
        angle3 = np.arctan2(y-y3, x-x3)
        return ((-1*6.67408*(10**-11)*self.m*m1)/(d1)**2)*np.sin(angle)+ ((-1*6.67408*(10**-11)*self.m*m2)/(d2)**2)*np.sin(angle2) +((-1*6.67408*(10**-1

    def euler_richardson_step(self,m1,m2,m3,x1,y1,x2,y2,x3,y3):
        # Calculate acceleration at the current position
        an_x = self.force_x(m1,m2,m3,self.x,self.y,x1,y1,x2,y2,x3,y3) / self.m
        an_y = self.force_y(m1,m2,m3,self.x,self.y,x1,y1,x2,y2,x3,y3) / self.m
        v_mid_x = self.ux +0.5*an_x*self.dt
        y_mid_x = self.x +0.5*self.ux*self.dt

        v_mid_y = self.uy +0.5*an_y*self.dt
        y_mid_y = self.y +0.5*self.uy*self.dt
        a_mid_y = self.force_y(m1,m2,m3,y_mid_x,y_mid_y,x1,y1,x2,y2,x3,y3) / self.m
        a_mid_x = self.force_x(m1,m2,m3,y_mid_x,y_mid_y,x1,y1,x2,y2,x3,y3) / self.m

        self.ux = self.ux + a_mid_x*self.dt
        self.x = self.x + v_mid_x*self.dt

        self.uy = self.uy + a_mid_y*self.dt
        self.y = self.y + v_mid_y*self.dt

# 1 hour time step

sun = sources(1.989e30, 0, 0, 0, 0, 3600)  # Sun
earth = sources(5.972e24, 1.496e11, 0, 0, 29780, 3600)  # Earth
venus = sources(4.867e24, 1.08e11, 0, 0, 35260, 3600)  # Venus
mars = sources(6.4171e23, 2.2794e11, 0, 0, 24077, 3600) #mars
```

Figure 5.2: Euler Richardson Algorithm for 4 bodies attracted to each other by Gravitational Forces

```
list1 = []
list2 = []
list3 = []
list4 = []
list5 = []
list6 = []
list7 = []
list8 = []


for k in range(20000):
    list1.append(earth.x)
    list2.append(earth.y)
    earth.euler_richardson_step(sun.m,venus.m,mars.m,sun.x,sun.y,venus.x,venus.y,mars.x,mars.y)

    list3.append(venus.x)
    list4.append(venus.y)
    venus.euler_richardson_step(sun.m,earth.m,mars.m,sun.x,sun.y,earth.x,earth.y,mars.x,mars.y)

    list5.append(sun.x)
    list6.append(sun.y)
    sun.euler_richardson_step(earth.m,venus.m,mars.m,earth.x,earth.y,venus.x,venus.y,mars.x,mars.y)

    list7.append(mars.x)
    list8.append(mars.y)
    mars.euler_richardson_step(earth.m,venus.m,sun.m,earth.x,earth.y,venus.x,venus.y,sun.x,sun.y)
```

Figure 5.3: Euler Richardson Algorithm for 4 bodies attracted to each other by Gravitational Forces-2

```
import matplotlib.pyplot as plt

import numpy as np

plt.figure(figsize=(8, 6))
plt.scatter(list1, list2, label='Earth', s = 1)
plt.scatter(list5, list6, color='yellow', label='Sun')
plt.scatter(list3, list4, color='orange', label='Venus',s = 1)
plt.scatter(list7, list8, color='black', label='Mars',s = 1)
plt.xlabel('X position')
plt.ylabel('Y position')
plt.title('Earth, Sun, Venus and Mars positions over time')
plt.legend()
plt.grid(True)
plt.show()
```

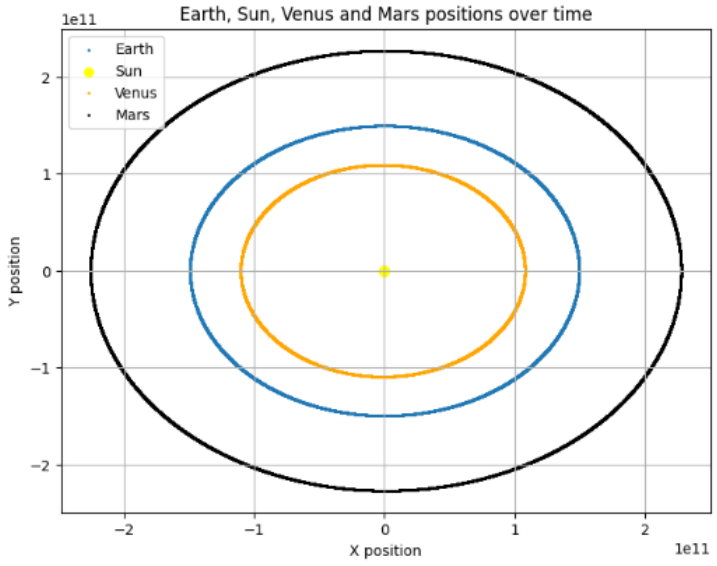


Figure 5.4: Graphical Representation

5.3 Light curve of one body Observed from another body

1. Light Curve of the Sun as observed from Earth (without Venus).
2. Light Curve of the Sun as observed from Earth (with Venus transiting).
3. Light Curve of Venus as observed from Earth.

[89]:
```python
# Function to calculate distance between two points
def distance(x1, y1, x2, y2):
    return ((x2 - x1) ** 2 + (y2 - y1) ** 2) ** 0.5

def calculate_apparent_brightness(distance):
    return 1 / (distance ** 2)  # Assuming inverse square law for brightness

# Calculate distances between Sun and Earth at each time step
distances_sun_earth = [distance(list5[i], list6[i], list1[i], list2[i]) for i in range(len(list1))]

# Calculate apparent brightness for each distance
apparent_brightness_sun_from_earth = [calculate_apparent_brightness(dist) for dist in distances_sun_earth]

# Plotting the light curve
import matplotlib.pyplot as plt

plt.figure(figsize=(8, 6))
plt.plot(apparent_brightness_sun_from_earth, label='Sun observed from Earth')
plt.xlabel('Time')
plt.ylabel('Apparent Brightness')
plt.ylim((3e-23,5e-23))
plt.title('Light Curve of Sun as Observed from Earth without venus')
plt.legend()
plt.grid(True)
plt.show()
```

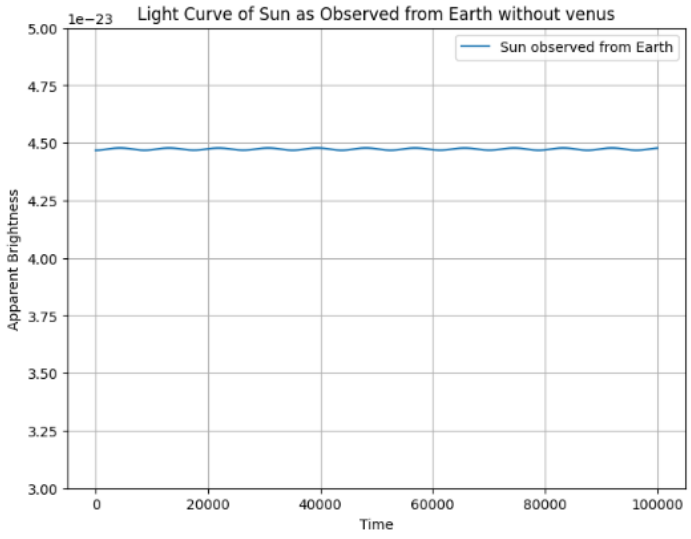


Figure 5.5:

```python
def distance1(x1, y1, x2, y2):
    return ((x2 - x1) ** 2 + (y2 - y1) ** 2) ** 0.5


def is_inside_triangle(px, py, v1x, v1y, v2x, v2y, v3x, v3y):
    def sign(p1x, p1y, p2x, p2y, p3x, p3y):
        return (p1x - p3x) * (p2y - p3y) - (p2x - p3x) * (p1y - p3y)

    b1 = sign(px, py, v1x, v1y, v2x, v2y) < 0.0
    b2 = sign(px, py, v2x, v2y, v3x, v3y) < 0.0
    b3 = sign(px, py, v3x, v3y, v1x, v1y) < 0.0

    return b1 == b2 == b3
```

```python
flux_list = []
for i in range(100000):
    a = list1[i]-list5[i]
    b = list2[i]-list6[i]
    if b == 0:
        slope = np.pi/2
    else:
        slope = np.arctan(-a/b)
    x_2 = np.cos(slope)*696340000
    y_2 = np.sin(slope)*696340000
    x_3 = -1*x_2
    y_3 = -1*y_2
    dist_s_e = distance1(list5[i], list6[i], list1[i], list2[i])
    if is_inside_triangle(list3[i],list4[i],list1[i],list2[i],x_2,y_2,x_3,y_3):
        flux_list.append((1 / (dist_s_e ** 2))*(1-0.002)) # venus transit blocks 0.2% of Sun's light
    else:
        flux_list.append(1 / (dist_s_e ** 2))

plt.figure(figsize=(16, 4))
plt.plot(flux_list, label='Sun observed from Earth')
plt.xlabel('Time')
plt.ylabel('Apparent Brightness')
plt.ylim((4e-23,5e-23))
plt.title('Light Curve of Sun as Observed from Earth with venus transit')
plt.legend()
plt.grid(True)
plt.show()
```

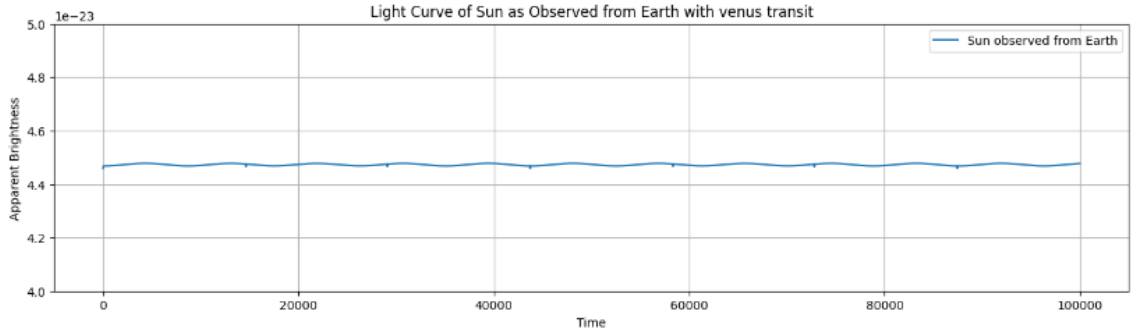


Figure 5.6:

```
theta = np.arcsin(696340000/distance1(list5[0], list6[0], list3[0], list4[0]))
theta
```

```
0.006447637266057416
```

```
p9 = []
for i in range(1000):
    if i != 0:
        k = np.arctan((list3[i]-list1[0])/(list4[i]-list2[0]))
        if k < 0.006447637266057416 :
            p = distance1(list3[i], list4[i], list1[0], list2[0])
            p9.append(p)
max(p9)
```

```
146513987332.24957
```

```
flux_list2 = []
for i in range(100000):

    a1 = distance1(list5[i], list6[i], list3[i], list4[i])
    b1 = distance1(list3[i], list4[i], list1[i], list2[i])

    if i == 0:
        flux_list2.append(0)
    elif  b1 > 146513987300 :
        flux_list2.append(0)
    else:
        flux_list2.append(1 / ((a1+b1) ** 2))

plt.figure(figsize=(16, 4))
plt.plot(flux_list2, label='Venus observed from Earth)')
plt.xlabel('Time')
plt.ylabel('Apparent Brightness')
# plt.ylim((4e-23,5e-23))
plt.title('Light Curve of Venus as Observed from Earth (light reflected by venus (source = sun) ')
plt.legend()
plt.grid(True)
plt.show()
```

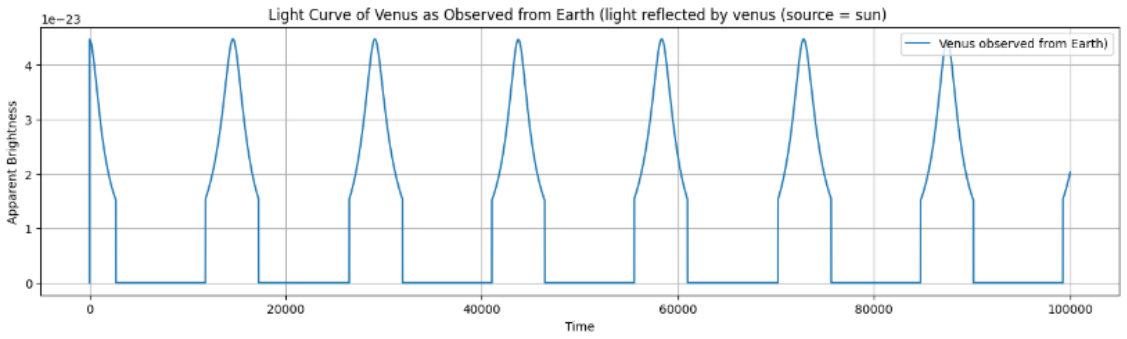


Figure 5.7: