In [1]:
```python
from scipy.fft import fft, rfft
from scipy.fft import fftfreq, rfftfreq
import plotly.graph_objs as go
from plotly.subplots import make_subplots
import matplotlib.pyplot as plt
import tensorflow as tf
import pandas as pd

import numpy as np
from scipy.signal import argrelextrema
from scipy.interpolate import CubicSpline
import random
import math
from scipy.interpolate import CubicSpline
import statistics
from statistics import mode
get_ipython().run_line_magic('matplotlib', 'inline')
```

In [2]:
```python
def extend_signal(signal,K):
   signal=np.array(signal)
   signal2=[]
   N=len(signal)

   for n in range(int(K*N)):
       signal2.append(signal[n%N])
   signal2=np.array(signal2)

   return signal2

def signal_weight(signal):
   X=0
   MinX=100000.0
   for x in signal:
       if x<MinX:
            MinX=x



   for x in signal:
       X=X+((x-MinX)*(x-MinX))

   return X


class EllipseEquation():

    t=[]
    dist=[]
    accel=[]
    vel=[]
    Theta=[]
    vel_x=[]
    dist_x=[]
    x_coords=[]
    y_coords=[]
    x_offset=0
    G=6.673e-11
    dt=0
    def __init__(self):
        return self
    def __init__(self,a,e,M):
        self.e=e
        self.a=a
```

```
        self.b=math.sqrt(a*a*(1-e*e))
        self.M=M
        self.P=2*math.pi*math.sqrt(self.a**3/(self.G*self.M))
    def calc_r(self,theta):
        return self.a*(1-self.e**2)/(1+self.e*math.cos(theta))
    def calc_v(self,r):
        return math.sqrt(self.G*self.M*(2/r-1/self.a))


    def calc(self,N,x_offset=0):
        #print ('Delta time in seconds:'+repr(dt))
        self.x_offset=x_offset
        self.t=[]
        self.dist=[]
        self.accel=[]
        self.Theta=[]
        self.vel=[]
        self.vel_x=[]
        self.dist_x=[]
        self.x_coords=[]
        self.y_coords=[]
        theta=0
        T=0
        X=0
        Dtheta=0
        D=0
        count=0
        theta_add=0
        self.dt=self.P/N
        while theta>=0 and theta<2*math.pi and count<10000:
            r=self.calc_r(theta)
            v=self.calc_v(r)

            x=r*math.cos(theta)+x_offset
            y=r*math.sin(theta)
            self.x_coords.append(x)
            self.y_coords.append(y)
            #print(x)
            d_theta=2*math.pi/N
            overflow_count=0

            while overflow_count<10:
                overflow_count+=1
                theta_dash=theta+d_theta
                r_dash=self.calc_r(theta_dash)
                v_dash=self.calc_v(r_dash)
                x_dash=r_dash*math.cos(theta_dash)+x_offset
                y_dash=r_dash*math.sin(theta_dash)

                d_dist=math.sqrt((x_dash-x)**2+(y_dash-y)**2)
                dt_dash=v/d_dist
                dt_ratio=dt_dash*self.dt
                #print(dt_ratio)
                if(dt_ratio<1.01 and dt_ratio>=1):
                    break
                d_theta=d_theta*dt_ratio
            #print(d_theta)
            theta_dash=theta+d_theta
            r_dash=self.calc_r(theta_dash)
            v_dash=self.calc_v(r_dash)
            x_dash=r_dash*math.cos(theta_dash)+x_offset
            y_dash=r_dash*math.sin(theta_dash)

            D=math.sqrt((x)**2+(y)**2)
```

```
            self.t.append(T)
            self.dist.append(D)
            self.vel.append(v)
            self.accel.append((v_dash-v)/self.dt)
            self.Theta.append(theta)
            T=T+self.dt
            count+=1
            theta=theta_dash
            #print(theta)


        self.t=np.array(self.t)
        self.dist=np.array(self.dist)
        self.vel=np.array(self.vel)
        self.Theta=np.array(self.Theta)
        self.x_coords=np.array(self.x_coords)
        self.y_coords=np.array(self.y_coords)

        return count
class EllipticalDifferenceEquation(EllipseEquation):

        Mass=0
        semimajoraxis=0
        amplitude=1.0
        offset=0
        obs_amplitude=1.0
        obs_offset=0
        obs_period=0
        time=[]
        flux=[]
        resultant=[]
        dist=[]
        e1=0
        e2=0
        G=6.673e-11
        phase=0
        N=1000
        N_phase1=0
        N_phase2=0
        index0=0
        index1=0
        def __init__(self,time,flux,Mass,e1,e2):
            self.e1=e1
            self.e2=e2
            self.Mass=Mass
            self.time=np.array(time)
            self.flux=np.array(flux)
        def normalise(self,array_input):
            max_array=np.max(array_input)
            min_array=np.min(array_input)
            amplitude=max_array-min_array
            offset=(max_array+min_array)/(2*(amplitude))
            return array_input/amplitude-offset
        def apply_phase(self,array_input,phase):
            N=len(array_input)
            array_input=extend_signal(array_input,2)
            return array_input[range(int(phase*N),N+int(phase*N))]
        def calc_arrays(self,array1,array2,amplitude,phase):
            N=len(array1)
            amplitude=amplitude/N
            array_resultant=[]
            for i in range(N-1):
                j=i+phase
                if j>=N:
```

```
                j=i+phase-N
            array_resultant.append(array1[i]-amplitude*array2[j])
        return np.array(array_resultant)



    def calc_weights(self,list_array,delta_phase):
        N=len(list_array[0])

        phase0=0
        amplitude=0
        min_args=[0,0,0]
        min_weight=9999999
        while phase0<N:
            phase1=int(0.1*N/2)
            while phase1<(int(N/2)-int(0.1*N/2)):
                amplitude=int(N/2)
                while amplitude<N:
                    array=self.calc_arrays(list_array[1],list_array[2],amplitude/N,p
                    weight=signal_weight(self.calc_arrays(list_array[0],array,1.0,ph
                    #print(weight,phase1,phase0,amplitude)
                    if(weight<min_weight):
                        min_args=[phase1,phase0,amplitude]
                        min_weight=weight
                        #print(min_args)
                    amplitude+=delta_phase
                phase1+=delta_phase
            phase0+=delta_phase
        print('.',end='')


        return min_args
    def calc(self):
        print('Calculating ...')
        N=self.N

        print('index length:'+repr(N)+' index start:'+repr(self.index1)+' end index:

        self.flux=np.array(self.flux[range(self.index1,self.index0)])




        self.obs_period=self.time[self.index0]-self.time[self.index1]
        print('Observational Period:'+repr(self.obs_period))

        self.time=np.array(self.time[range(self.index1,self.index0)])
        self.time=self.time-self.time[0]


        spline_data=CubicSpline(self.time,self.flux)
        N=100 #data points
        self.time = np.linspace(0, self.obs_period, num=int(N)) #time base total 0.1
        self.flux=spline_data(self.time)
        self.N=N


        self.flux=self.normalise(self.flux)

        self.semimajoraxis=math.pow((self.G*self.Mass)*(self.obs_period/(2*math.pi))

    def calc_main(self,ARGS=[],E1=2,E2=7,deltaN=1):

        N=100
```

```
if(ARGS==[]):
    list_ellipse=[]



    min_weight=999999
    e1=0.1


    EE=EllipseEquation(self.semimajoraxis,e1,self.Mass)

    i=0
    while e1<0.89:
        EE.e=e1
        EE.calc(N)
        list_ellipse.append(self.normalise(-EE.dist))
        i+=1
        e1+=0.1
    I=i
    i=0
    while i<I:
        j=0
        while j<I:

            args=self.calc_weights([self.flux,list_ellipse[i],list_ellipse[j
            self.dist=self.calc_arrays(list_ellipse[i],list_ellipse[j],args[
            self.dist=self.normalise(self.dist)
            self.flux=self.calc_arrays(self.flux,self.dist,1.0,args[0])
            weight=signal_weight(self.flux-self.dist)
            if(weight<min_weight):
                min_weight=weight
                E1=i
                E2=j
                ARGS=args
            j+=1
            print(' :'+repr(j),end='')
        i+=1
        print(' :'+repr(i),end='')
    self.dist=self.calc_arrays(list_ellipse[E1],list_ellipse[E2],ARGS[2],ARG
    self.dist=self.normalise(self.dist)
    self.flux=self.calc_arrays(self.flux,self.dist,1.0,ARGS[0])
    print(ARGS)
elif len(ARGS)==3:
    EE1=EllipseEquation(self.semimajoraxis,E1,self.Mass)
    EE2=EllipseEquation(self.semimajoraxis,E2,self.Mass)
    n=2
    while EE1.calc(N+n)>101:
        n=n-1
    n=2
    while EE2.calc(N+n)>101:
        n=n-1
    self.dist=self.calc_arrays(EE1.dist,EE2.dist,ARGS[2],ARGS[1])
    self.dist=self.normalise(self.dist)
    self.dist=self.normalise(self.dist)
    #self.flux=self.calc_arrays(self.flux,self.dist,1.0,ARGS[0])
elif len(ARGS)==2:
    EE1=EllipseEquation(self.semimajoraxis,E1,self.Mass)
    EE2=EllipseEquation(self.semimajoraxis,E2,self.Mass)
    n=2
    while EE1.calc(N+n)>101:
        n=n-1
    n=2
    while EE2.calc(N+n)>101:
        n=n-1
    args=self.calc_weights([self.flux,EE1.dist,EE2.dist],deltaN)
    self.dist=self.calc_arrays(EE1.dist,EE2.dist,args[2],args[1])
```

```
            self.dist=self.normalise(self.dist)
            self.flux=self.calc_arrays(self.flux,self.dist,1.0,args[0])
            #weight=signal_weight(self.flux-self.dist)
            print(args)
        elif len(ARGS)==5:
            EE1=EllipseEquation(self.semimajoraxis,ARGS[2],self.Mass)
            EE2=EllipseEquation(self.semimajoraxis,ARGS[3],self.Mass)

            if(ARGS[2]>0.9):
                n=2000
            else:
                n=2
            while EE1.calc(N+n)>101:
                n=n-1
            if(ARGS[3]>0.9):
                n=2000
            else:
                n=2
            while EE2.calc(N+n)>101:
                n=n-1
            print(len(EE1.dist))
            print(len(EE2.dist))


            self.dist=self.calc_arrays(EE1.dist,EE2.dist,int(ARGS[1]*100),int(ARGS[0
            self.dist=self.normalise(self.dist)
            self.dist=self.normalise(self.dist)
            self.flux=self.apply_phase(self.flux,ARGS[4])
            #self.flux=self.flux,self.dist,1.0,0.0)
            #weight=signal_weight(self.flux-self.dist)
            print(ARGS)









    def printout(self):
        spline_data1=CubicSpline(self.time[range(len(self.flux))],self.flux)
        self.dist=np.array(self.dist)
        self.dist=self.dist[range(len(self.flux))]
        spline_data2=CubicSpline(self.time[range(len(self.flux))],self.dist)
        N=1000 #data points

        self.time = np.linspace(0, self.obs_period, num=int(N)) #time base total 0.1

        self.flux=spline_data1(self.time)
        self.dist=spline_data2(self.time)
        self.N=N
        print('eccentricity1:'+repr(self.e1)+' eccentricity2:'+repr(self.e2))
        print('Retry again with initial eccentricities listed above.')

        #print('Phase 1: '+repr(M1+20)+' value: '+repr(phase_weights[M1]))
        #print('Phase 2: '+repr(M2)+' value: '+repr(weights[M2]))
        weight_flux=signal_weight(self.flux)
        weight_resultant=signal_weight(self.flux-self.dist)
        print('Weight Flux:  '+repr(weight_flux))
        print('Weight Resultant:  '+repr(weight_resultant))
        ratio_weight=weight_resultant/weight_flux
        if(ratio_weight>1):
            period_error=0
```

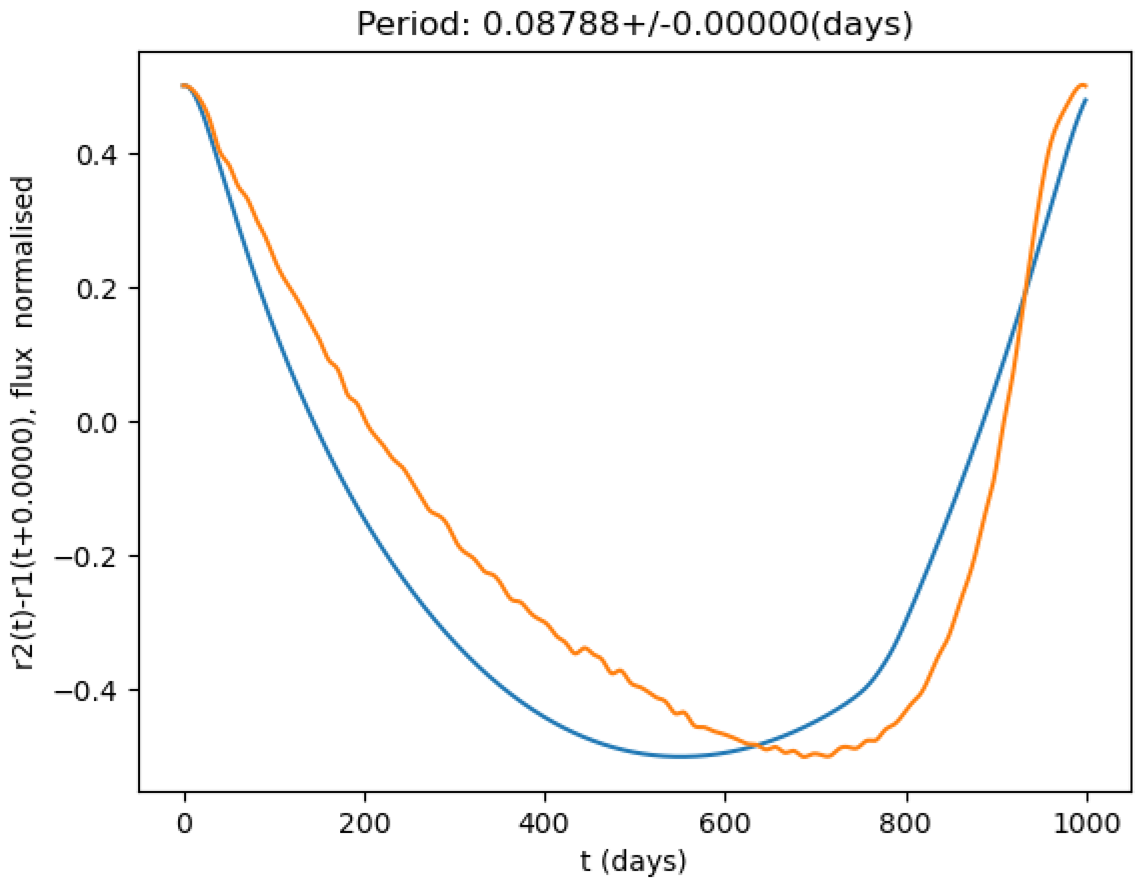
```python
        else:
            period_error=0.5*self.obs_period*ratio_weight

        freq_error=-1/(self.obs_period+period_error)+1/(self.obs_period-period_error
        print('Frequency: '+format(1/self.obs_period,'0.3f')+'+/-'+format(freq_error
        mag_error=-2.81*math.log10(self.obs_period+period_error)+2.81*math.log10(sel
        print('Average Visual Magnitude: ',format(-2.81*math.log10(self.obs_period)-
        mag_error=-3.725*math.log10(self.obs_period+period_error)+3.725*math.log10(s
        print('Average Visual Magnitude: ',format(-3.725*math.log10(self.obs_period)

        self.resultant=self.flux+self.dist
        plt.plot(-self.dist)
        plt.plot(self.flux)
        plt.xlabel('t (days)')
        plt.ylabel('r2(t)-r1(t+'+format(self.N_phase1/self.N,'.4f')+'), flux  normal
        plt.title('Period: '+format(self.obs_period,'0.5f')+'+/-'+format(period_erro
        plt.show()



    def optimise_e(self,e=1,count_max=5,phase_max1=0.1,phase_max2=0.05,phase_min=0.0
        N=self.N
        if (e==1):
            e1=self.e1
        else:
            e1=self.e2
        print('Calculating Phase,Eccentricity...')
        list_resultant_weight=[]
        counter_e1=0
        delta_e1=0.1
        while counter_e1<count_max:

            calc_phase([self.ellipse2.dist,self.ellipse1.dist,self.flux],0.5,10)



            self.dist=dist5
            weight_flux=signal_weight(self.flux)
            weight_resultant=signal_weight(dist5-self.flux)
            list_resultant_weight.append(weight_resultant)
            if(counter_e1>0):
                difference_weight_resultant=list_resultant_weight[counter_e1]-list_r
                if(abs(difference_weight_resultant)>1):
                    if(difference_weight_resultant<0):
                        delta_e1=-0.05/abs(difference_weight_resultant)
                    else:
                        delta_e1=0.05/abs(difference_weight_resultant)
                else:
                    delta_e1=difference_weight_resultant*delta_e1
                #print('Delta e1: '+repr(delta_e1))
            if(abs(delta_e1)<0.0001):
                break
            e1=e1+delta_e1
            if(e==1):
                self.ellipse1.e=e1
            else:
                self.ellipse2.e=e1
            n=N+2

            if(e==1):
                N2=self.ellipse1.calc(n)
            else:
                N2=self.ellipse2.calc(n)
            while N!=N2:
                n=n-1
                if(e==1):
```

```python
                    N2=self.ellipse1.calc(n)
                else:
                    N2=self.ellipse2.calc(n)

            counter_e1+=1
            print(counter_e1)
        if(e==1):
            self.e1=e1
        else:
            self.e2=e1
        self.N_phase1=M1
        self.N_phase2=M2



    def calc_period(self):
         ##################################### PERIOD CALCULATION #################
        #use adaptive smoothing to find optimal maxima for period calculation
        use_maxima=True #False untested
        print('Calculating Period...')
        Smooth_factor=20

        N_list=[]
        Index0_list=[]
        Index1_list=[]
        N_fluxmax=len(self.flux)
        if(N_fluxmax>1000): # cut short due to errors with np.max
             N_fluxmax=1000
        #to avoid local maxima choose maxima with flux above %90 maximum
        fluxmax=0.95*np.max(self.flux[range(N_fluxmax)])
        fluxmin=0.95*np.min(self.flux[range(N_fluxmax)])
        print('Flux Maximum:  '+repr(fluxmax)+' Flux Minimum:  '+repr(fluxmin))
        while Smooth_factor<60:
            test_flux=np.convolve(self.flux,np.ones(Smooth_factor)/Smooth_factor,mod
            maxima=argrelextrema(test_flux,np.greater)
            minima=argrelextrema(test_flux,np.less)
            maxima=maxima[0]
            minima=minima[0]
            temp_maxima=[]
            for maxima_index in maxima:
                if(self.flux[maxima_index]>fluxmax):
                    temp_maxima.append(maxima_index)
            maxima=temp_maxima
            temp_minima=[]
            for minima_index in maxima:
                if(self.flux[minima_index]<fluxmin):
                    temp_minima.append(minima_index)
            minima=temp_minima
            #print(maxima)
            i=1
            N=0
            while N<1000: #period cutoff
                if use_maxima:
                    if(i+1>len(maxima)):
                        break
                    N=maxima[i]-maxima[i-1]
                else:
                    if(i+1>len(minima)):
                        break
                    N=minima[i]-minima[i-1]
                i=i+1
            i=i-1
            if(N>10): # use values that represent a proper period
                N_list.append(N)
                if use_maxima:
                    Index0_list.append(maxima[i])
```

```
                    Index1_list.append(maxima[i-1])
                else:
                    Index0_list.append(minima[i])
                    Index1_list.append(minima[i-1])

            #print(N)
            Smooth_factor=Smooth_factor+1


        #choose most frequent from (N_list)
        self.N=mode(N_list)

        print('N:'+repr(self.N))

        #search N_list for index of N
        i=0
        for j in range(len(N_list)):
            if(N_list[j]==self.N):
                i=j
                break
        self.index0=Index0_list[i]
        self.index1=Index1_list[i]
        ############################################################################

def shuffle(inputs_train,labels_train,N_counter):
    counter=0
    while counter<N_counter:
        index1=random.randint(0,len(inputs_train)-1)
        index2=random.randint(0,len(inputs_train)-1)
        temp1=inputs_train[index1]
        temp2=labels_train[index1]
        labels_train[index1]=labels_train[index2]
        labels_train[index2]=temp2
        inputs_train[index1]=inputs_train[index2]
        inputs_train[index2]=temp1
        counter+=1
        #print(inputs_train,labels_train)
    return (inputs_train,labels_train)

rel_flux=(0.516874,0.514718,0.513875,0.51701,0.515947,0.516506,0.514698,0.514998,0.51023
time_base=(0,0.000569,0.00114,0.001709,0.002279,0.002849,0.003418,0.003989,0.004559,0.00

spline_data=CubicSpline(time_base,rel_flux)
O=2000 #data points
t = np.linspace(0, 0.145668, num=int(O)) #time base total 0.145668 days
signal=spline_data(t)


M=1.9891E30 #Mass in kg
e1=0.70 #eccentricity of ellipse 1
e2=0.30 #eccentricity of ellipse 2

EDE_main=EllipticalDifferenceEquation(t,signal,M,e1,e2)
EDE_main.calc_period()
EDE_main.calc()
EDE_main.calc_main([ -0.77417165, 0.1677008, 0.71303195 ,0.75767905,0])




EDE_main.printout()
```

```
Calculating Period...
Flux Maximum:  0.7822310891874247 Flux Minimum:  0.4762603901995125
N:1206
Calculating ...
```

```
index length:1206 index start:555 end index:1761
Observational Period:0.08788174487243622
101
101
[-0.77417165, 0.1677008, 0.71303195, 0.75767905, 0]
eccentricity1:0.7 eccentricity2:0.3
Retry again with initial eccentricities listed above.
Weight Flux:  217.11087124421908
Weight Resultant:  744.0257294044169
Frequency: 11.379+/-0.0000
Average Visual Magnitude:  1.5376+/-0.0000
Average Visual Magnitude:  1.9440+/-0.0000
```



In [3]:

```
print("Loading light_curve_model.keras")
model = tf.keras.models.load_model("./light_curve_model110.keras")
input_dist=EDE_main.flux[range(99)]
input_dist=tf.reshape(input_dist,(1,99))

pred = model.predict(input_dist)
print(pred[0])
```

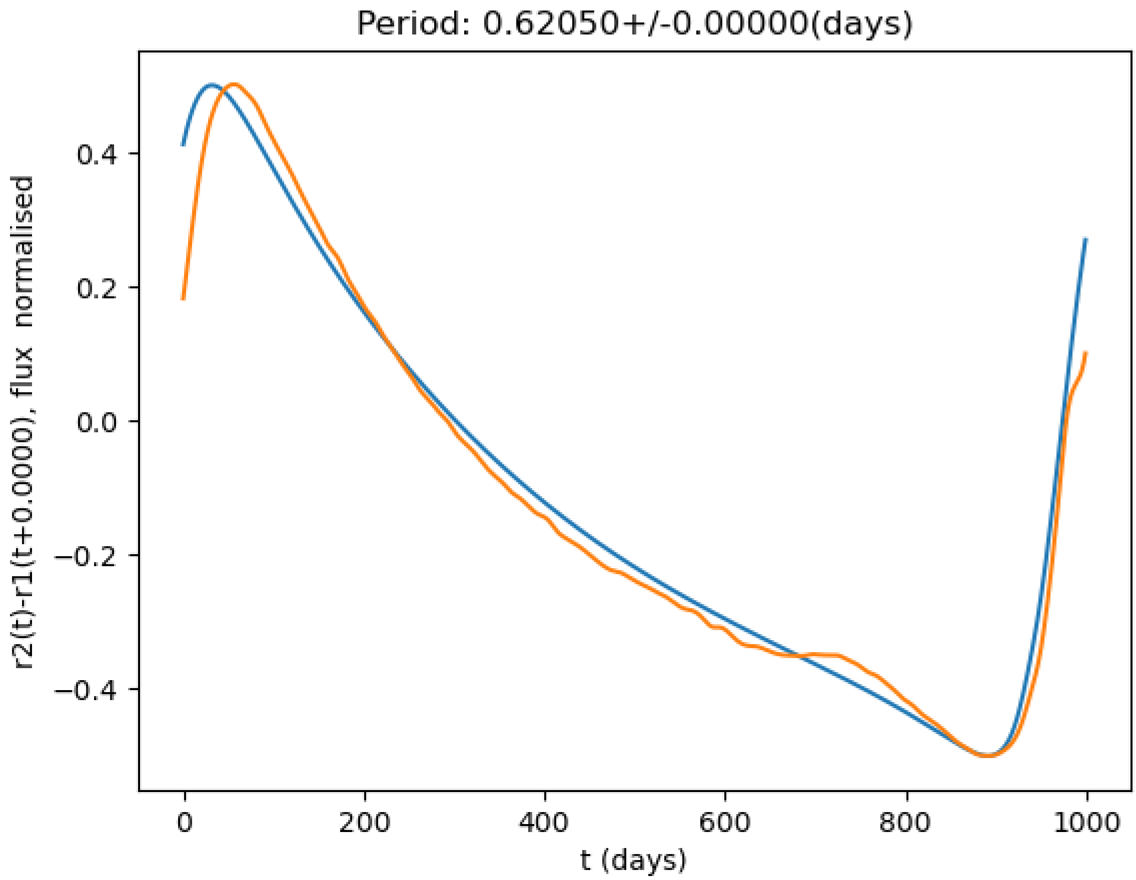
```
Loading light_curve_model.keras
1/1 [==============================] - 1s 649ms/step
[0.97988474 0.8039203  0.59598666 0.60092145 0.22974396]
```

In [5]:

```
from astropy.io import fits
from astropy.table import Table
filename = r"./kplr005559631-2010234115140_slc.fits"


fits.info(filename)
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']

    # Read in the columns of data.
```

```python
    times = hdulist[1].data['time']
    sap_fluxes = hdulist[1].data['SAP_FLUX']
    pdcsap_fluxes = hdulist[1].data['PDCSAP_FLUX']


print(times)
N=len(pdcsap_fluxes)
t=np.linspace(0,times[N-1]-times[0],N)


M=1.9891E30 #Mass in kg
e1=0.70 #eccentricity of ellipse 1
e2=0.30 #eccentricity of ellipse 2

EDE_rrlyrae=EllipticalDifferenceEquation(times,sap_fluxes,M,e1,e2)
EDE_rrlyrae.calc_period()
EDE_rrlyrae.calc()
EDE_rrlyrae.calc_main([1-0.91, 0.83, 0.61, 0.67,-0.06])




EDE_rrlyrae.printout()
```

```
Filename: ./kplr005559631-2010234115140_slc.fits
No.    Name      Ver    Type      Cards   Dimensions   Format
  0  PRIMARY       1 PrimaryHDU      58   ()
  1  LIGHTCURVE    1 BinTableHDU    155   45390R 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   (5, 5)   int32
[567.37368236 567.37436347 567.37504458 ... 598.2879318  598.288613
 598.289294  ]
Calculating Period...
Flux Maximum:  20648.8919921875 Flux Minimum:  10065.671191406249
N:911
Calculating ...
index length:911 index start:44260 end index:45171
Observational Period:0.6204979998656199
101
101
[0.08999999999999997, 0.83, 0.61, 0.67, -0.06]
eccentricity1:0.7 eccentricity2:0.3
Retry again with initial eccentricities listed above.
Weight Flux:  230.26596992293094
Weight Resultant:  937.7589135346624
Frequency: 1.612+/-0.0000
Average Visual Magnitude:  -0.8476+/-0.0000
Average Visual Magnitude:  -1.2180+/-0.0000
```

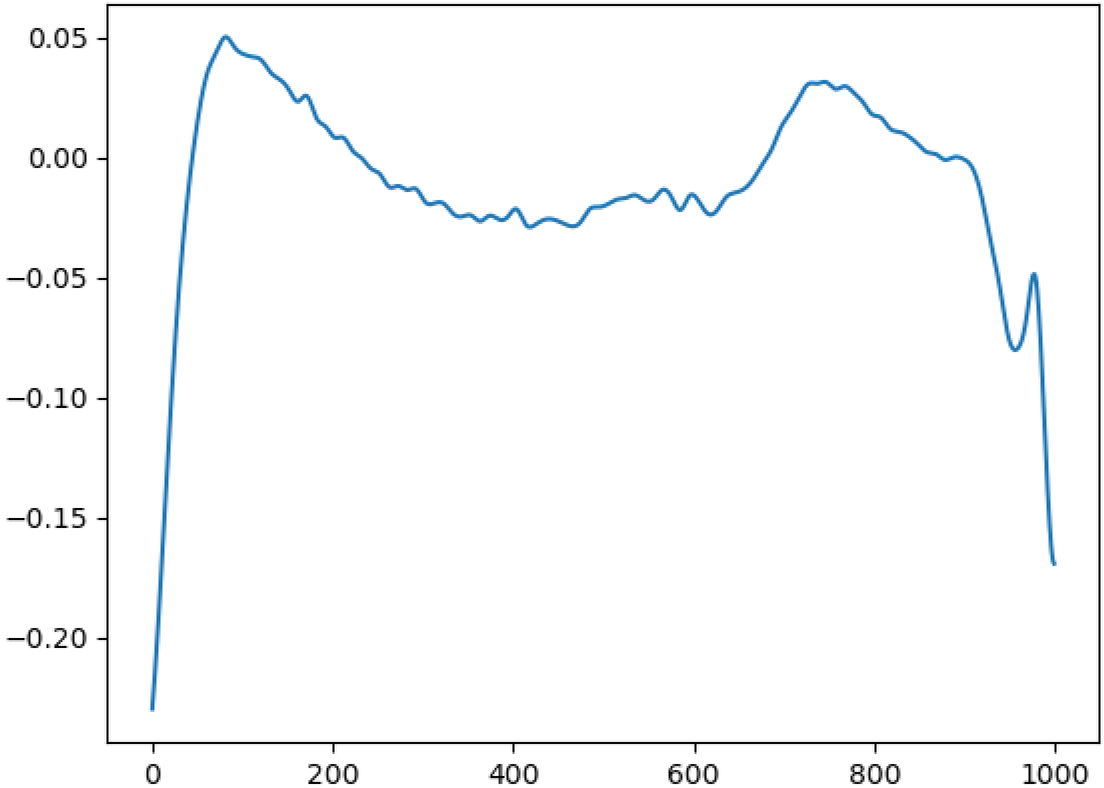


In [6]:
```
print("Loading light_curve_model.keras")
model = tf.keras.models.load_model("./light_curve_model130.keras")
input_dist=EDE_rrlyrae.flux[range(99)]
input_dist=tf.reshape(input_dist,(1,99))

pred = model.predict(input_dist)
print(pred[0])
```

```
Loading light_curve_model.keras
1/1 [==============================] - 0s 94ms/step
[0.99948746 0.00535165 0.9232514  0.69283575 0.12081055]
```

In [7]:
```
plt.plot(EDE_rrlyrae.resultant)
```

Out[7]:
```
[<matplotlib.lines.Line2D at 0x276fad209d0>]
```



In [ ]: