

Investigating the Travelling Salesperson problem and the Tacoma Bridge collapse

Importing necessary modules

```
import numpy as np
import numpy.random as random
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
from matplotlib.pyplot import figure
import random
import copy
%matplotlib inline
```

Python Version

```
import sys
import matplotlib
print("Python version: {}".format(sys.version))
print("Matplotlib version: {}".format(matplotlib.__version__))
```

```
Python version: 3.5.0 (default, Sep 13 2015, 23:30:27)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.56)]
Matplotlib version: 1.4.3
```

Abstract

In this report, the Travelling Salesperson problem and Tacoma Bridge collapse was investigated. The Travelling Salesperson used simulated annealing algorithm to find the shortest path distance between 30 cities. The shortest path found was 2194.6483. The Tacoma Bridge collapse was modelled using a set of differential equations. These equations were then solved using the Taylor method and the Cromer method. The driven wind force was introduced to the differential equations which meant the natural frequency of the bridge could be found. This was found to be 0.12Hz.

Introduction

This report aims to investigate 2 optimisation problems. These problems are commonly known as the Tacoma Bridge collapse and the Travelling Salesperson problem. The Travelling Salesperson is categorised as NP-hard. NP hardness is used in computational theory to define a property of a class of problems. A problem is defined as NP-hard if an algorithm for solving it can also be used to solve any NP-problem. Therefore, NP-hard is at least if not harder than any NP-problem [NP-hard problem].

The Tacoma Bridge collapse occurred in November 1940 [Tacoma Narrows Bridge collapses 2009], only a short while after its opening; it was the third largest suspension bridge at the time. The bridge was constructed in a non-typical fashion, it used two narrow plate girders to stiffen the deck instead of using the traditional approach of trusses. It was

clear from the beginning that the bridge was too flexible, even in moderate winds. The collapse of the bridge was primarily caused by the phenomenon known as resonance. Resonance is when a periodic force becomes in phase with the natural frequency of a system. These periodic forces add up to large oscillations over a period of time. In this case for the Tacoma Bridge collapse, the driving force comes from the phenomena known as vortex shedding and aerodynamic flutter. This is where a fluid, such as air, moving past an object oscillates, producing vortices behind the object. This causes zones of low pressure, and when these occur at frequencies close to the natural frequency of the bridge, small amounts of wind can cause large oscillations. Any amount of twisting in the bridge creates areas of low pressure/ vortices which amplify the twisting motion. This is a self-induced oscillation at a definite frequency where energy is extracted from the wind driving force by the motion of the structure [Bond, 2022].

The motion of the collapse of the Tacoma Bridge can be modelled using a set of differential equations. These equations can be solved computationally using the Taylor and Cromer method. The motion is first modelled without the addition of a driving force and then later, wind is introduced to model a more accurate representation of the collapse. From these models, we are able to determine the natural frequency of the bridge by finding the greatest vertical displacement and its corresponding rotational frequency. The natural frequency of the bridge was stated to be 0.2Hz [Tacoma Narrows Bridge Failure].

The Travelling Salesperson problem was first studied in the 1920s by an economist and mathematician by the name of Karl Menger [Cummings, 2000]. It was previously known as the 'messenger problem' and was defined as the task of finding the shortest distance for a known path which had a finite number of points. The TSP is categorised as a NP-hard problem which means it cannot be solved in polynomial time. For further clarification, this means if the computation time is greater than a polynomial function of the input size, it cannot be solved in polynomial time. This project aims to find the shortest path possible for 30 capital cities in America.

Firstly, a random path is generated for 'n' number of cities. Then, a function to compute a pairwise exchange is made in order to shuffle the order of the cities created in the random path so that we can compare the path lengths for the original path and the new pairwise exchange path. The method of simulated annealing is used to determine whether the new path should be accepted or rejected. Finally, the shortest path can be found by creating an optimal function that consists of the functions mentioned above.

Problem 1: The Tacoma Bridge

In this problem, we use a set of differential equations to model the oscillations that lead up to the collapse of the Tacoma Bridge. First, the oscillations are modelled without any consideration of wind. Then wind is accounted for and we investigate how varying parameters such as the amplitude and frequency effect the motion of the bridge.

Cables hold tight when extended, however, lose resistance when compressed, therefore, Hooke's law for a conventional spring cannot be used to model the tension force. Instead, McKenna's model [McKenna and C.O Tuama] is used and is as follows:

$$f(y) = \frac{K}{a} (e^{ay} - 1) \tag{1}$$

By using trigonometry as well as Euler's equation of motion for a rigid body and taking into account the moment of inertia along with the tension force, the following differential equations are achieved.

$$\frac{d^2 y}{dt^2} = -d \times \frac{dy}{dt} - \frac{K}{ma} (e^{a(y - l \sin(\theta))} + e^{a(y + l \sin(\theta))} - 2) \tag{2}$$

$$\frac{d^2 \theta}{dt^2} = -d \times \frac{d\theta}{dt} + \frac{3 \cos(\theta)}{l} \frac{K}{ma} (e^{a(y - l \sin(\theta))} - e^{a(y + l \sin(\theta))}) \tag{3}$$

These differential equations would be extremely hard to solve analytically so the Taylor and Cromer method is used to computationally solve them.

In order to solve these differential equations we need to define a few equations:

$$\frac{dy}{dt} = z \tag{4}$$

$$\frac{d\theta}{dt} = \gamma$$

therefore

$$\frac{dz}{dt} = -d \times \frac{dy}{dt} - \frac{K}{ma} (e^{a(y - l \sin(\theta))} + e^{a(y + l \sin(\theta))} - 2) \tag{5}$$

and

$$\frac{d\gamma}{dt} = -d \times \frac{d\theta}{dt} + \frac{3 \cos(\theta)}{l} \frac{K}{ma} (e^{a(y - l \sin(\theta))} - e^{a(y + l \sin(\theta))}) \tag{6}$$

This is useful because instead of solving 2 second order differential equations, we are now solving equations of the first order.

Taylor and Cromer method

The Taylor method, also known as the Euler method, uses the current position of the system to calculate the next position. The Cromer method on the other hand, uses a mix of the current and later positions in order to determine the position of the system at later time. The Taylor method is defined as explicit while the Cromer method is semi-explicit [Euler-cromer method]. The equations used to approximate the solution is as follows:

Taylor method

$$\theta_{n+1} = \theta_n + dt \times \gamma_n \tag{7}$$

$$y_{n+1} = y_n + dt \times z_n \tag{8}$$

Cromer Method

$$\theta_{n+1} = \theta_n + dt \times \gamma_{n+1} \tag{9}$$

$$y_{n+1} = y_n + d t \cdot z_{n+1}$$

The Cromer method is a faster and more efficient method compared to the Taylor method due to the fact that it only works with first order differential equations however, this also means it tends to be less accurate. Although Taylor method is more accurate, it usually has a longer run time due to the higher order derivatives being calculated.

Initial parameters

```
tstart=0
tend=200
```

```
d=0.01
a=0.1
M=2500
K=1000
l=12
```

d = Friction Coefficient

a = Force Parameter

M = Mass of the bridge

K = Spring constant

l = width of the bridge

Defining functions for z and gamma:

Defining the functions beforehand makes the functions created later cleaner and easier to interpret.

```
def dz(z,y,theta):
    return -d * z - (K / (M * a)) * (np.exp(a * (y - l * np.sin(theta))) +
np.exp(a * (y + l * np.sin(theta))) - 2)
```

```
def dg (gamma, theta, y):
    return -d * gamma + ((3 * np.cos(theta) * K)/(l * M * a)) * (np.exp(a * (y
- l*np.sin(theta))) - np.exp(a*(y + l * np.sin(theta))))
```

Solving the differential

```
def tacoma(dt=0.01, cromer=False, y0=0, theta0=0, z0=0, gamma0=0):
```

```
    # Create the variables to store the results
    times=np.arange(tstart, tend+dt, dt) # Create the array of model
times
    Nt=len(times) # Number of time-steps
    y=np.zeros(Nt)
    theta=np.zeros(Nt)
```

```

z=np.zeros(Nt)
γ=np.zeros(Nt)

# Set the initial conditions

y[0]= y0
θ[0]=θ0
z[0]=z0
γ[0]=γ0

# Loop over all of the times and integrate the model
for n in range(Nt-1):

    γ[n+1]= γ[n] + (dg(γ[n], θ[n], y[n])) * dt

    z[n+1]=z[n] + (dz(z[n], y[n], θ[n])) * dt

    if cromer:
        θ[n+1] = θ[n] + dt * γ[n+1]

        y[n+1] = y[n] + dt * z[n+1]

    else:
        θ[n+1]= θ[n] + dt * γ[n]

        y[n+1]= y[n] + dt * z[n]
fig, ax = plt.subplots()
ax.plot(times, θ, '-b', label='theta against time')
ax.plot(times,y, '-r', label='y against theta')
plt.legend()
plt.title('cromer = {0}, theta0 = {1}'.format(cromer,θ0))
plt.show()

return times, θ, y

```

The above function models the oscillations of the bridge if no wind is present.

Graphs with varying parameters

```
times, θ, y = tacoma(dt=0.001, cromer=False, y0=0, θ0=0.01, z0=0,
γ0=0)
```

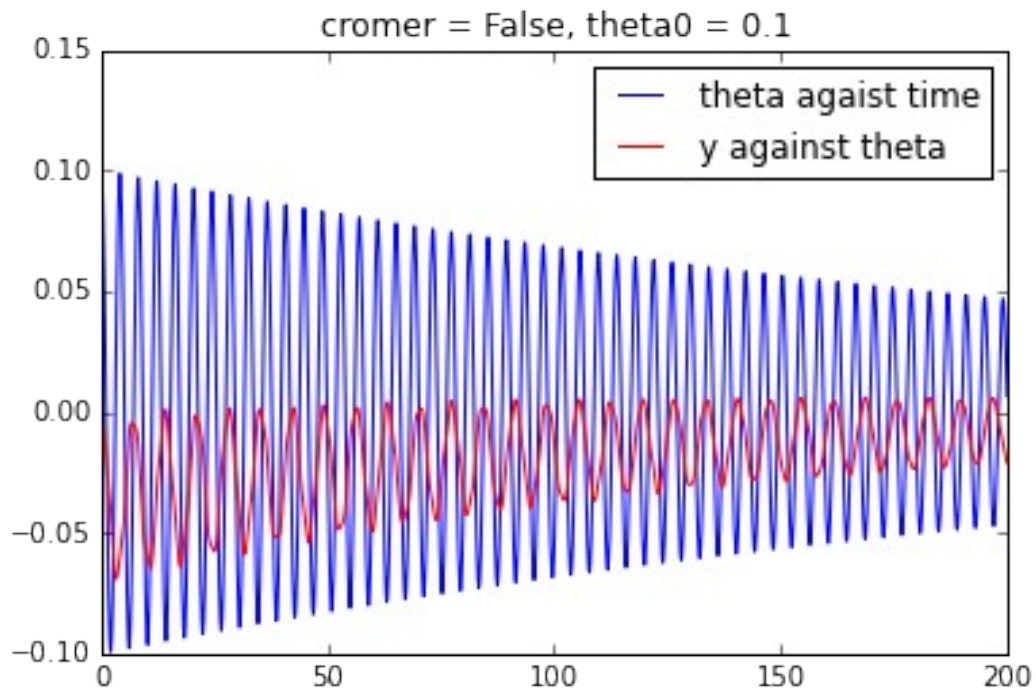
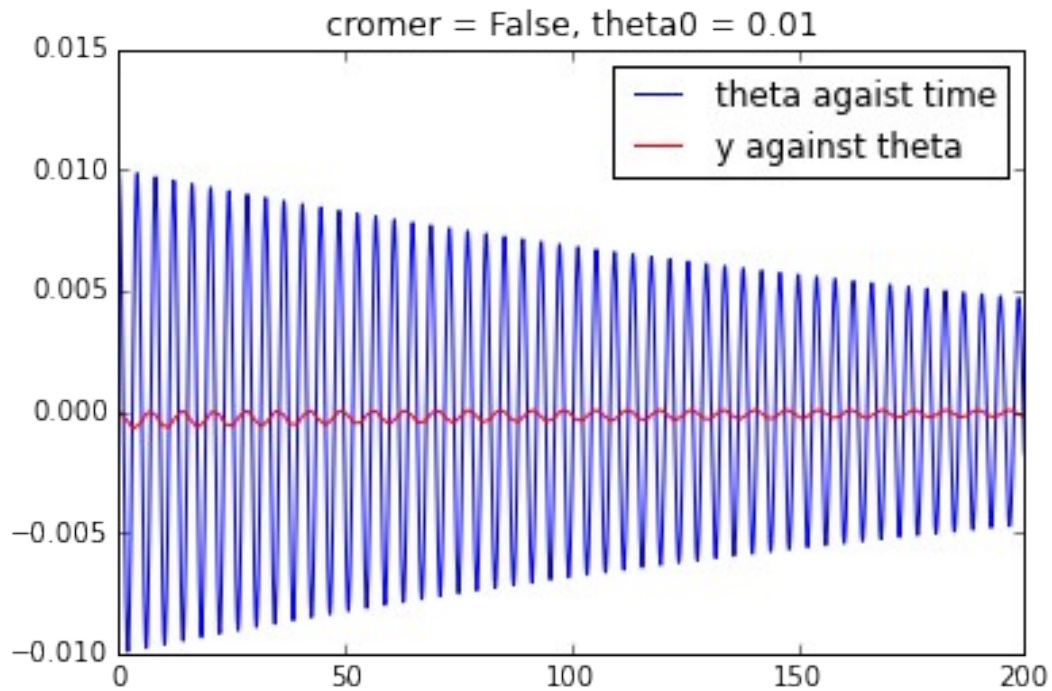
```
times, θ, y = tacoma(dt=0.001, cromer=False, y0=0, θ0=0.1, z0=0, γ0=0)
```

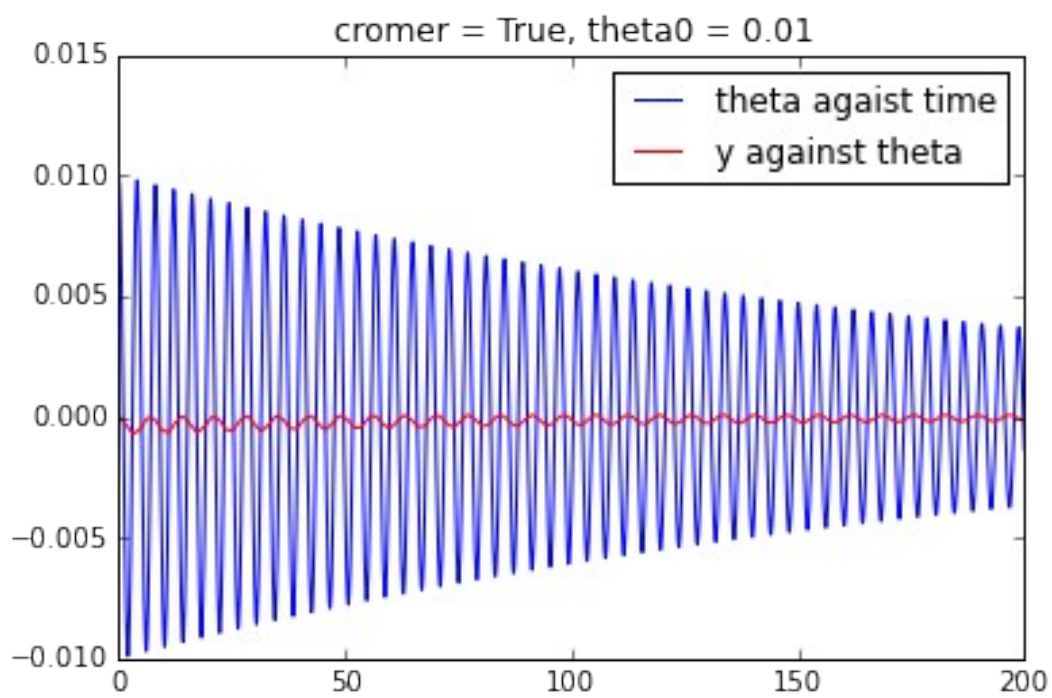
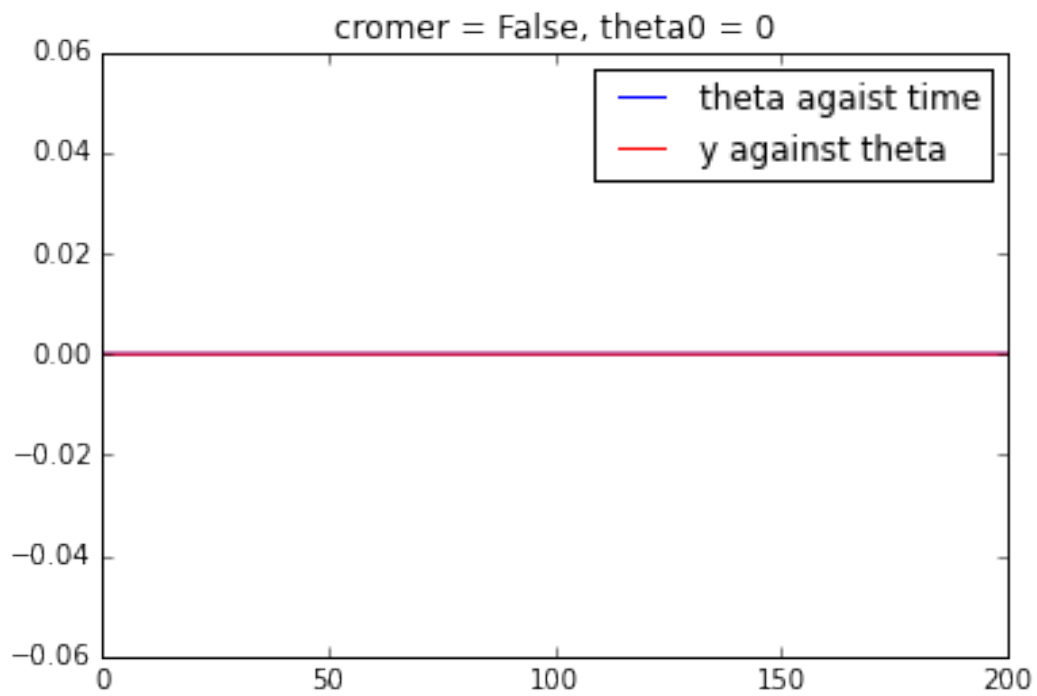
```
times, θ, y = tacoma(dt=0.001, cromer=False, y0=0, θ0=0, z0=0, γ0=0)
```

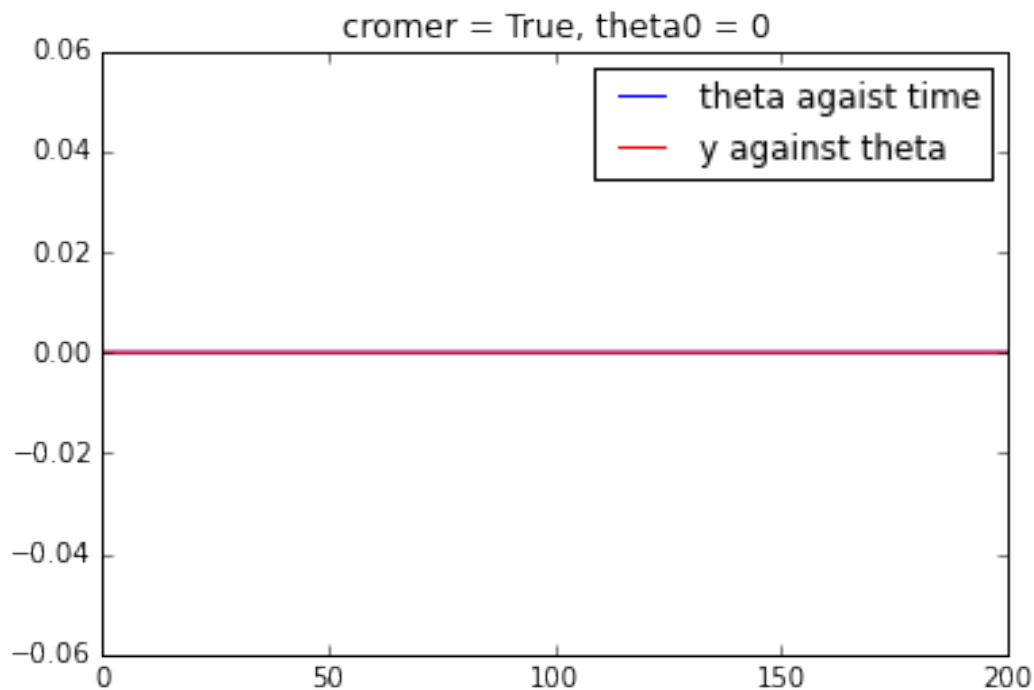
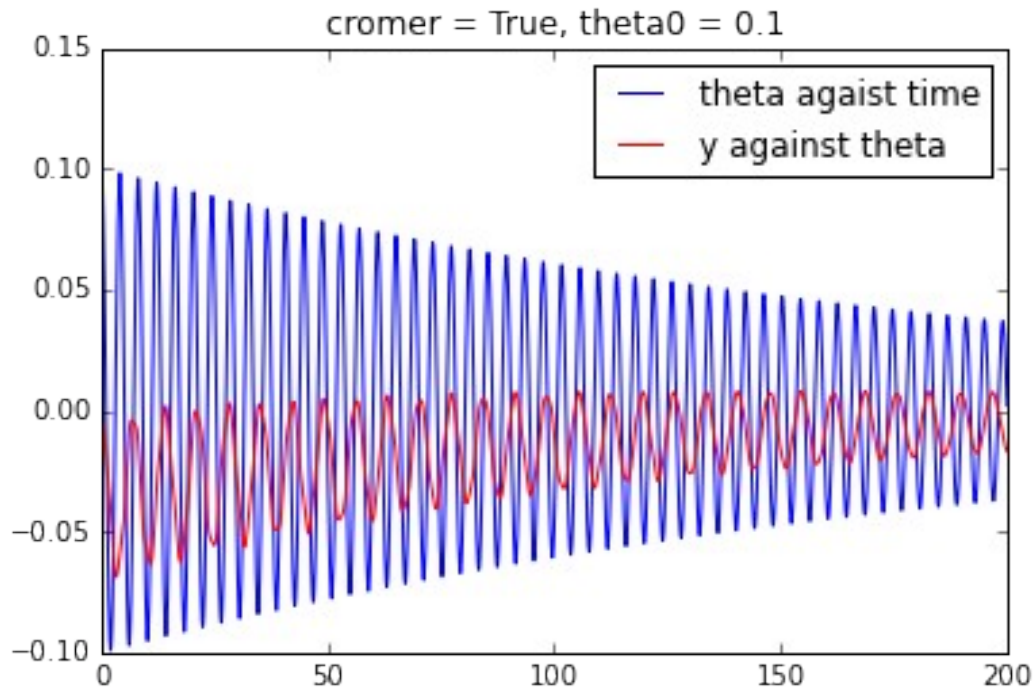
```
times,  $\theta$ , y = tacoma(dt=0.001, cromer=True, y0=0,  $\theta_0=0.01$ , z0=0,  $\gamma_0=0$ )
```

```
times,  $\theta$ , y = tacoma(dt=0.001, cromer=True, y0=0,  $\theta_0=0.1$ , z0=0,  $\gamma_0=0$ )
```

```
times,  $\theta$ , y = tacoma(dt=0.001, cromer=True, y0=0,  $\theta_0=0$ , z0=0,  $\gamma_0=0$ )
```







The graphs above compare the Cromer method to the Taylor method in order to verify whether these methods can be used to solve the coupled differential equations as well as establishing a baseline set of results to compare to. Both y and θ decrease over time in the absence of a driven force. This is expected as energy is dissipated over time. The graphs for the corresponding parameters for each Taylor and Cromer method are more or less

identical which would suggest that they were successful in solving the differential equations with the given parameters.

Wind

Now that the method has been verified and a baseline has been established, the driving force of wind can be introduced to more accurately model the events that occurred before the collapse of the Tacoma Bridge. This is simply done by adding an additional term to vertical component. The additional term is:

$$A \sin(\omega t)$$

```
def wind_tacoma(dt=0.01, cromer=False, y0=0, theta0=0, z0=0, gamma0=0, A=0,
omega=0):

    # Create the variables to store the results
    times=np.arange(tstart, tend+dt, dt) # Create the array of model
times
    Nt=len(times) # Number of time-steps
    y=np.zeros(Nt)
    theta=np.zeros(Nt)
    z=np.zeros(Nt)
    gamma=np.zeros(Nt)
    t=np.arange(tstart, tend+dt, dt)

    # Set the initial conditions

    y[0]=y0
    theta[0]=theta0
    z[0]=z0
    gamma[0]=gamma0

    for n in range(Nt-1):

        gamma[n+1]= gamma[n] + (dg(gamma[n], theta[n], y[n])) * dt

        z[n+1]=z[n] + (dz(z[n], y[n], theta[n]) + A * np.sin(omega*t[n])) * dt

    # wind introduced by adding A sin(omega t ) to the z component

    if cromer:
        theta[n+1] = theta[n] + dt * gamma[n+1]

        y[n+1] = y[n] + dt * z[n+1]

    else:
```

```
θ[n+1]= θ[n] + dt * γ[n]
```

```
y[n+1]= y[n] + dt * z[n]
```

```
return times, θ, y
```

```
times, θ, y = wind_tacoma(dt=0.001, cromer=True, y0=0, θ0=0.01, z0=0,
γ0=0, A=2, ω=3 )
fig, ax = plt.subplots()
ax.plot(times, θ, '-b', label='theta agaist time')
ax.plot(times,y, '-r', label='y against time')
plt.legend()
plt.title('A = 2, omega = 3')
plt.show()
```

```
times, θ, y = wind_tacoma(dt=0.001, cromer=True, y0=0, θ0=0.01, z0=0,
γ0=0, A=1, ω=3 )
fig, ax = plt.subplots()
ax.plot(times, θ, '-b', label='theta agaist time')
ax.plot(times,y, '-r', label='y against time')
plt.legend()
plt.title('A = 1, omega = 3')
plt.show()
```

```
times, θ, y = wind_tacoma(dt=0.001, cromer=True, y0=0, θ0=0.01, z0=0,
γ0=0, A=2, ω=2 )
fig, ax = plt.subplots()
ax.plot(times, θ, '-b', label='theta agaist time')
ax.plot(times,y, '-r', label='y against time')
plt.legend()
plt.title('A = 2, omega = 2')
plt.show()
```

```
times, θ, y = wind_tacoma(dt=0.001, cromer=True, y0=0, θ0=0.01, z0=0,
γ0=0, A=2, ω=2.5 )
fig, ax = plt.subplots()
ax.plot(times, θ, '-b', label='theta agaist time')
ax.plot(times,y, '-r', label='y against time')
plt.legend()
plt.title('A = 2, omega = 2.5')
plt.show()
```

```
times, θ, y = wind_tacoma(dt=0.001, cromer=True, y0=0, θ0=0.01, z0=0,
γ0=0, A=2, ω=2.8 )
fig, ax = plt.subplots()
ax.plot(times, θ, '-b', label='theta agaist time')
ax.plot(times,y, '-r', label='y against time')
plt.legend()
```

```
plt.title('A = 2, omega = 2.8')
plt.show()
```

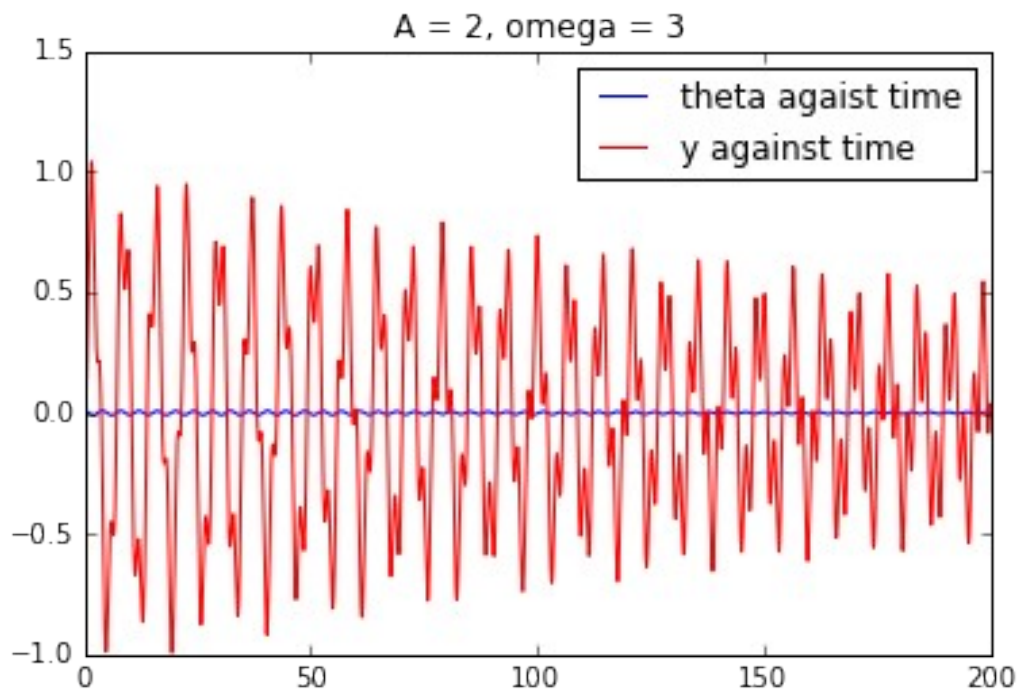
```
times,  $\theta$ , y = wind_tacoma(dt=0.001, cromer=True, y0=0,  $\theta_0=0.01$ , z0=0,
y0=0, A=2,  $\omega=3.3$ )
fig, ax = plt.subplots()
ax.plot(times,  $\theta$ , '-b', label='theta agaist time')
ax.plot(times,y, '-r', label='y against time')
plt.legend()
plt.title('A = 2, omega = 3.3')
plt.show()
```

[<matplotlib.lines.Line2D at 0x7f78ele60f98>]

[<matplotlib.lines.Line2D at 0x7f78d0703128>]

<matplotlib.legend.Legend at 0x7f78elf362e8>

<matplotlib.text.Text at 0x7f7887fc4080>

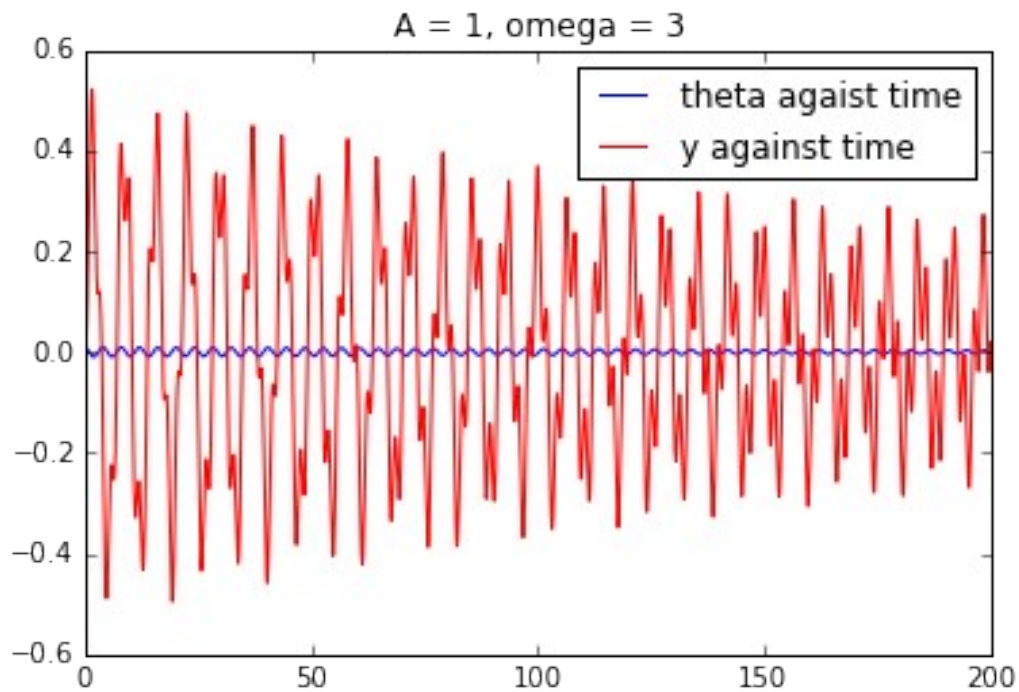


[<matplotlib.lines.Line2D at 0x7f78ele846a0>]

[<matplotlib.lines.Line2D at 0x7f78e3068f98>]

<matplotlib.legend.Legend at 0x7f78c0bef2e8>

<matplotlib.text.Text at 0x7f7873586ba8>

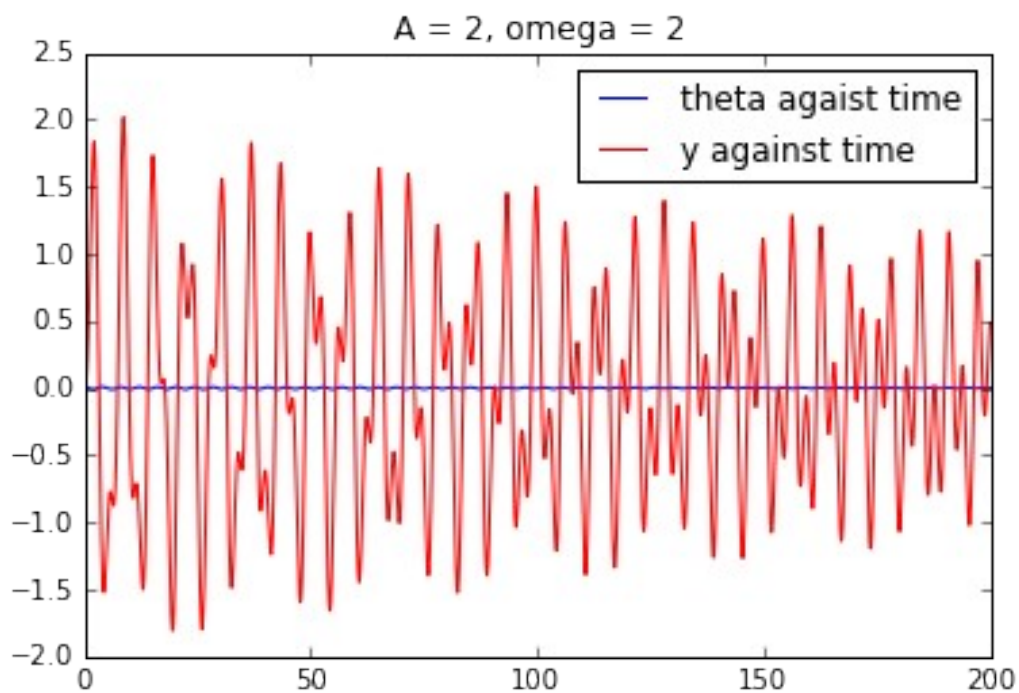


[<matplotlib.lines.Line2D at 0x7f78d0b5b048>]

[<matplotlib.lines.Line2D at 0x7f78e1e51e80>]

<matplotlib.legend.Legend at 0x7f78e1e51048>

<matplotlib.text.Text at 0x7f78d1319f28>

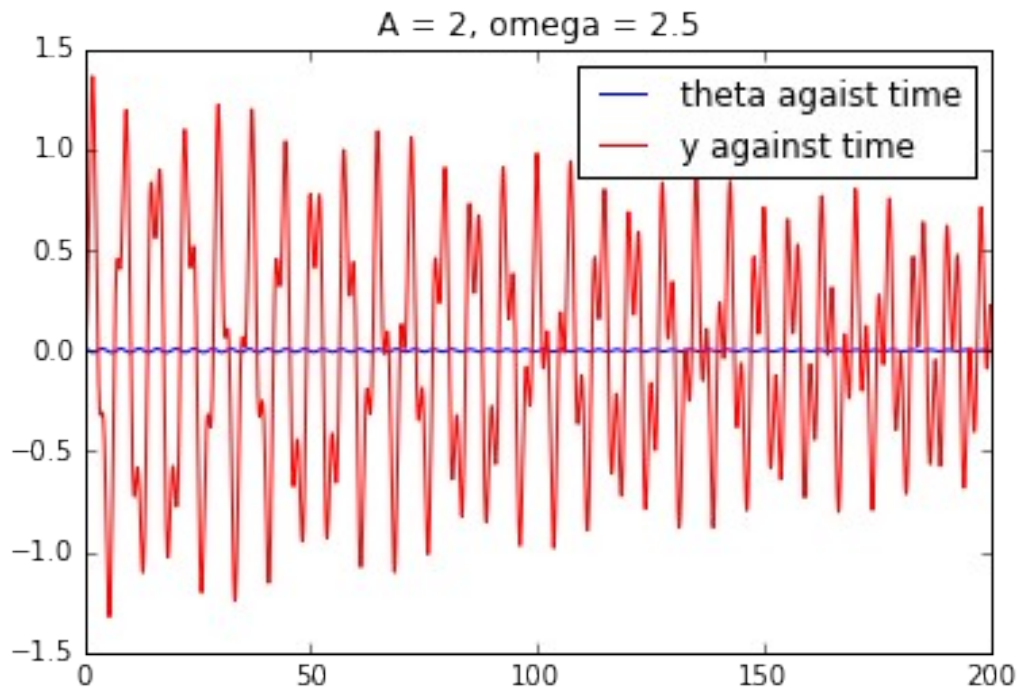


[<matplotlib.lines.Line2D at 0x7f78d132a860>]

[<matplotlib.lines.Line2D at 0x7f78e4189550>]

<matplotlib.legend.Legend at 0x7f78e4189828>

<matplotlib.text.Text at 0x7f78d0bba860>

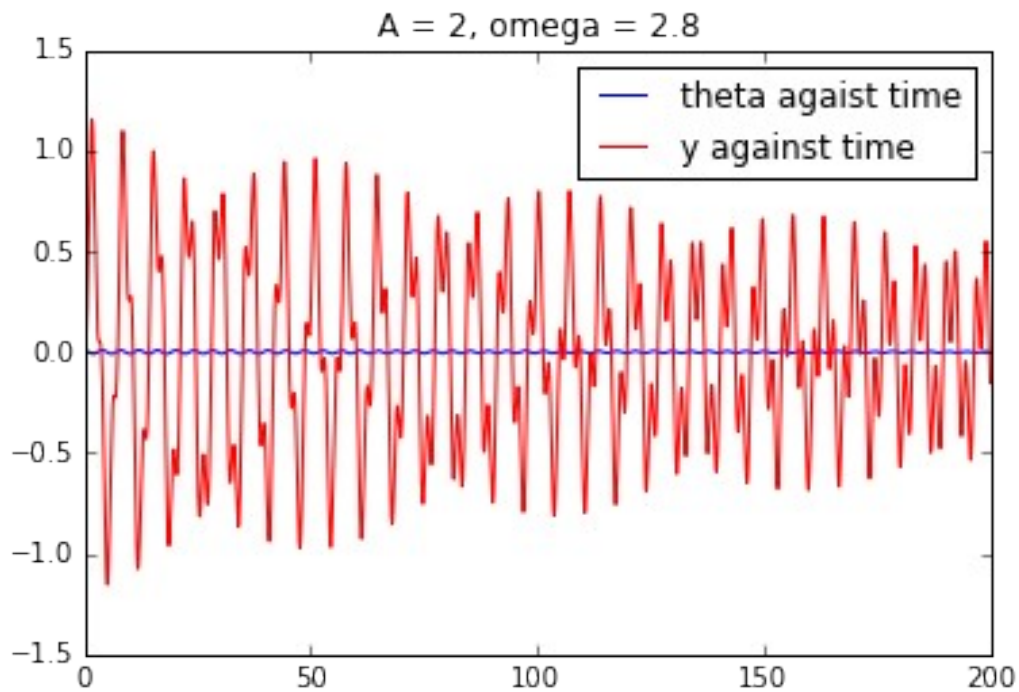


[<matplotlib.lines.Line2D at 0x7f78d2ebd0f0>]

[<matplotlib.lines.Line2D at 0x7f78d13f5ef0>]

<matplotlib.legend.Legend at 0x7f78d13d9208>

<matplotlib.text.Text at 0x7f78d06e3dd8>

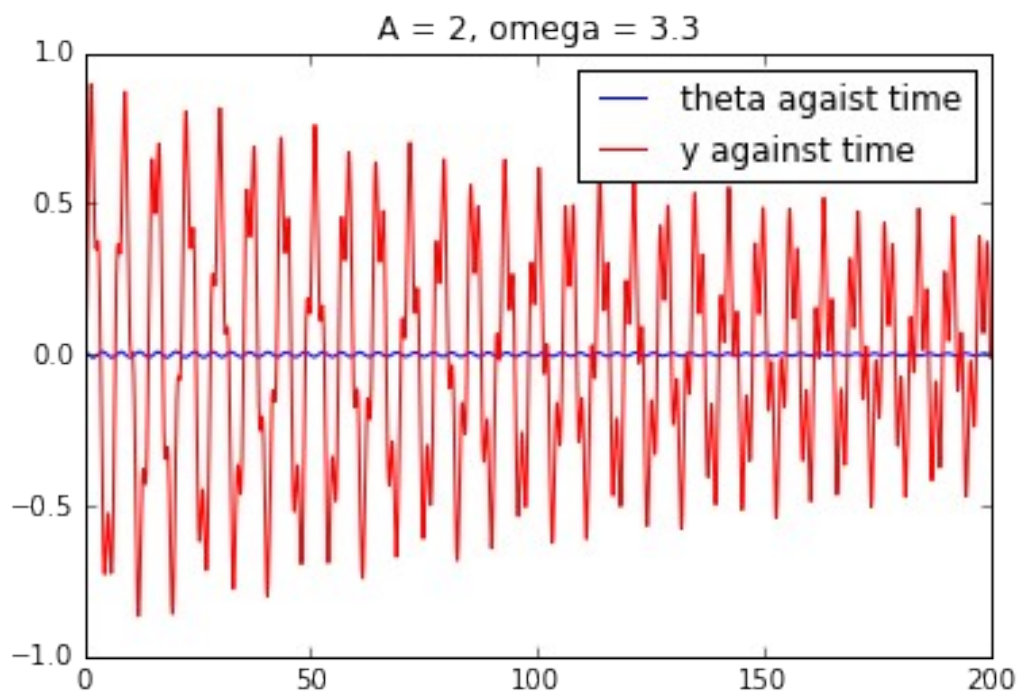


[<matplotlib.lines.Line2D at 0x7f78c087e7b8>]

[<matplotlib.lines.Line2D at 0x7f78e5712a20>]

<matplotlib.legend.Legend at 0x7f78e5712cf8>

<matplotlib.text.Text at 0x7f78c20a5f98>



We can see from the graph that as the amplitude of the wind component increases, so does the vertical displacement.

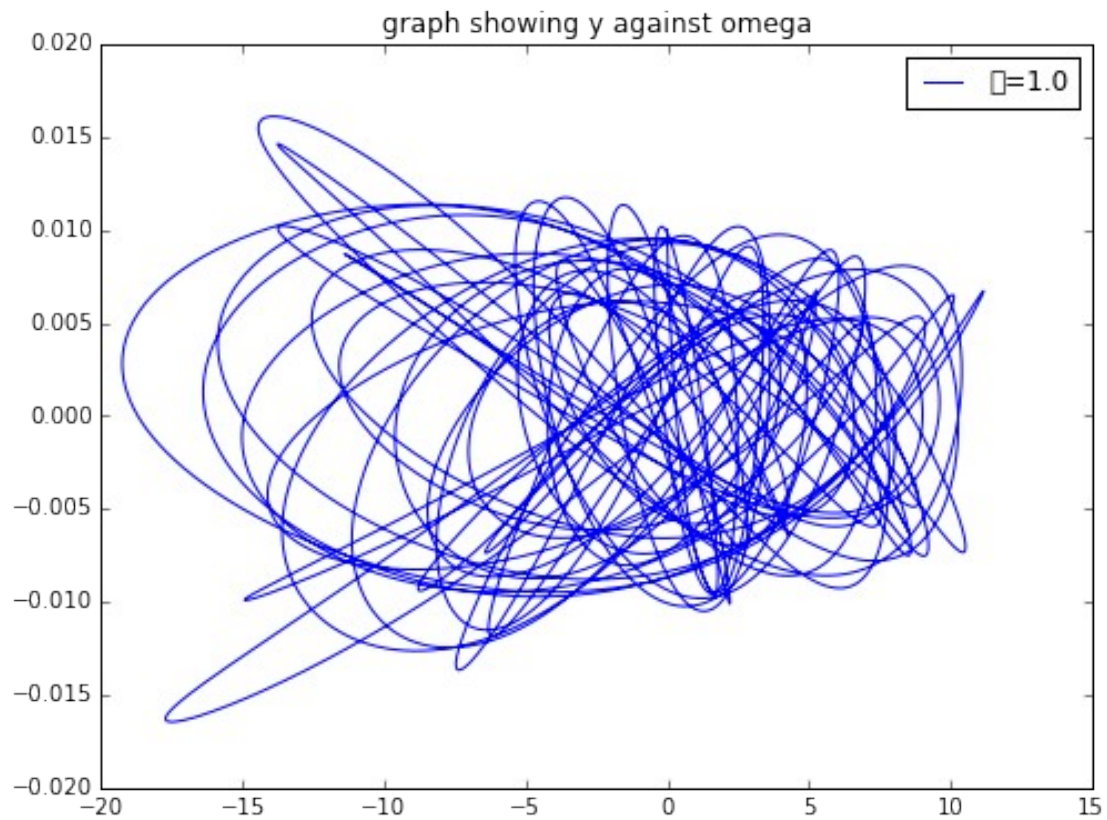
Investigation

Now we can determine the natural frequency of the bridge by varying ω and plotting a graph of θ against y and locating which value of ω produces the greatest displacement in y .

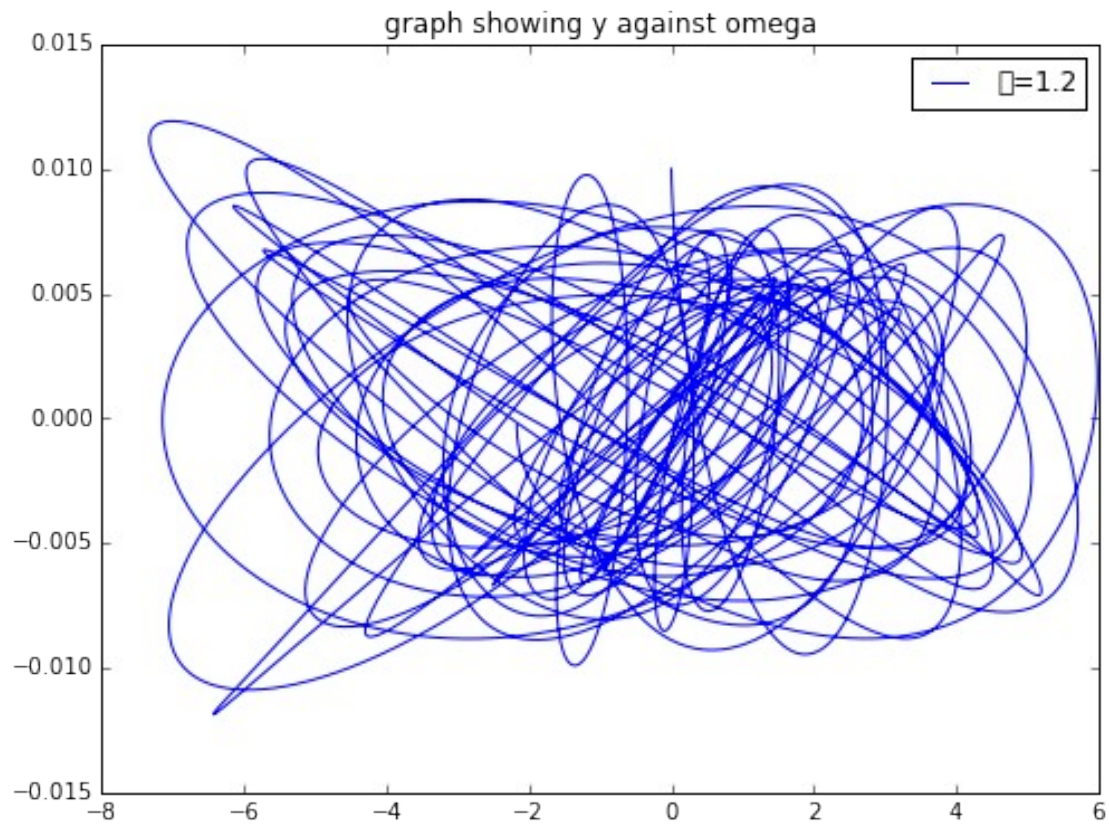
```
ω_start=1
ω_end=4.2
ω=np.arange(ω_start, ω_end, 0.2)

for i in ω:
    times, θ, y = wind_tacoma(dt=0.001, cromer=False, y0=0, θ0=0.01,
z0=0, γ0=0, A=2, ω=i )
    figure(figsize=(8,6),dpi=80)
    plt.plot(y,θ , '-b', label='ω={0}'.format(i))
    plt.xlabel('')
    plt.ylabel('')
    plt.title('graph showing y against omega')
    plt.legend()
    plt.show()

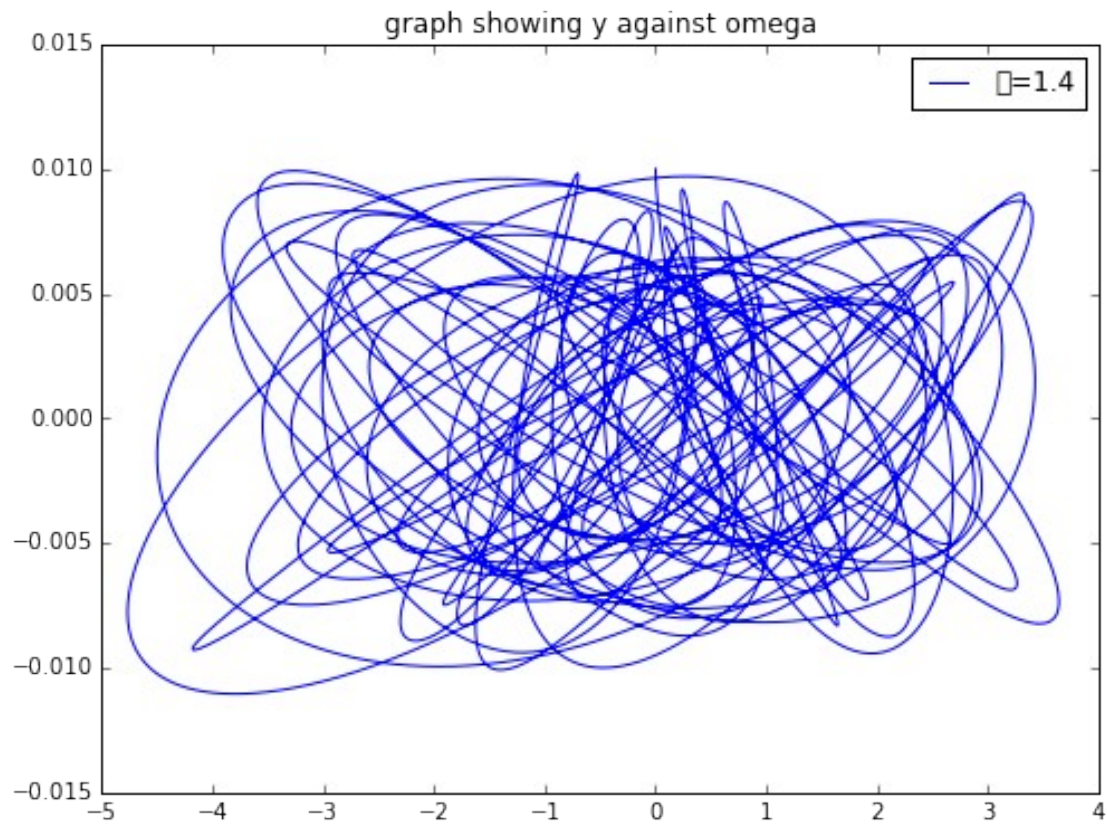
<matplotlib.figure.Figure at 0x7f78c00b7588>
[<matplotlib.lines.Line2D at 0x7f78c2d880f0>]
<matplotlib.text.Text at 0x7f7873584f28>
<matplotlib.text.Text at 0x7f78c084eb38>
<matplotlib.text.Text at 0x7f78d06f6898>
<matplotlib.legend.Legend at 0x7f78c2d88cf8>
```

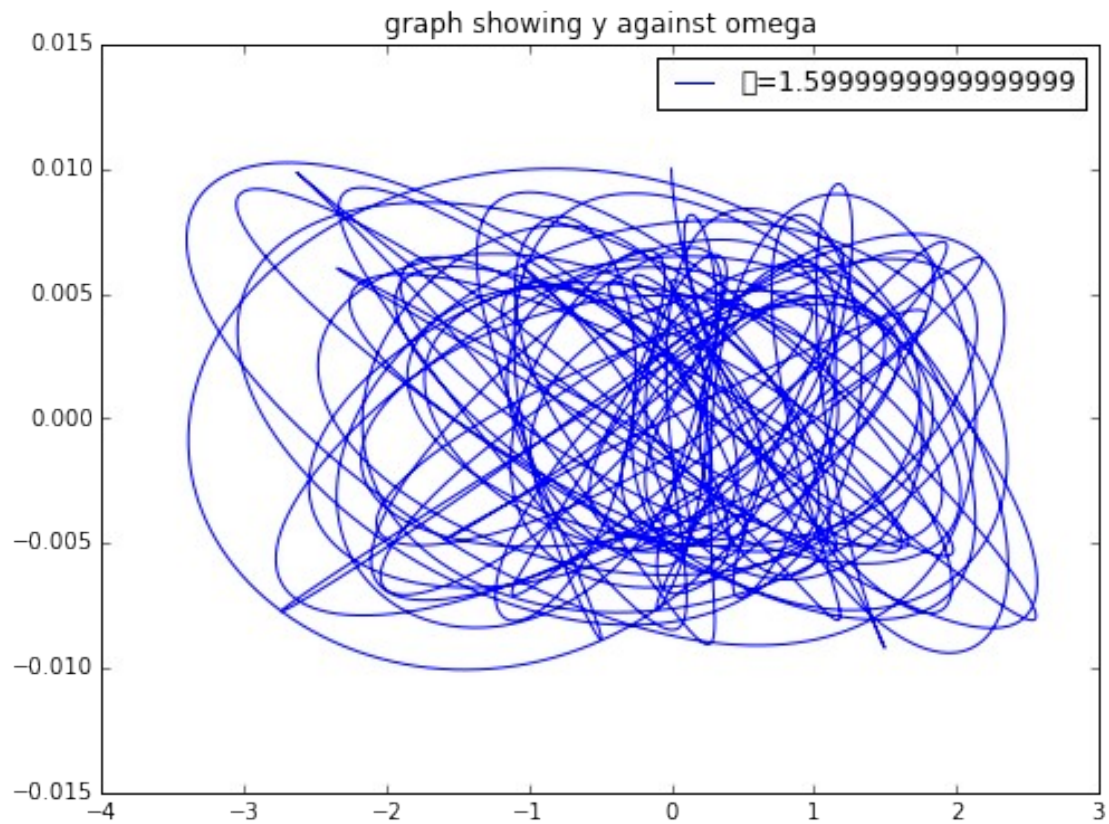
```
<matplotlib.figure.Figure at 0x7f78e1e2bd30>  
[<matplotlib.lines.Line2D at 0x7f78c2d845f8>]  
<matplotlib.text.Text at 0x7f78c084e6a0>  
<matplotlib.text.Text at 0x7f78d0716be0>  
<matplotlib.text.Text at 0x7f78b004fcc0>  
<matplotlib.legend.Legend at 0x7f78e580a8d0>
```

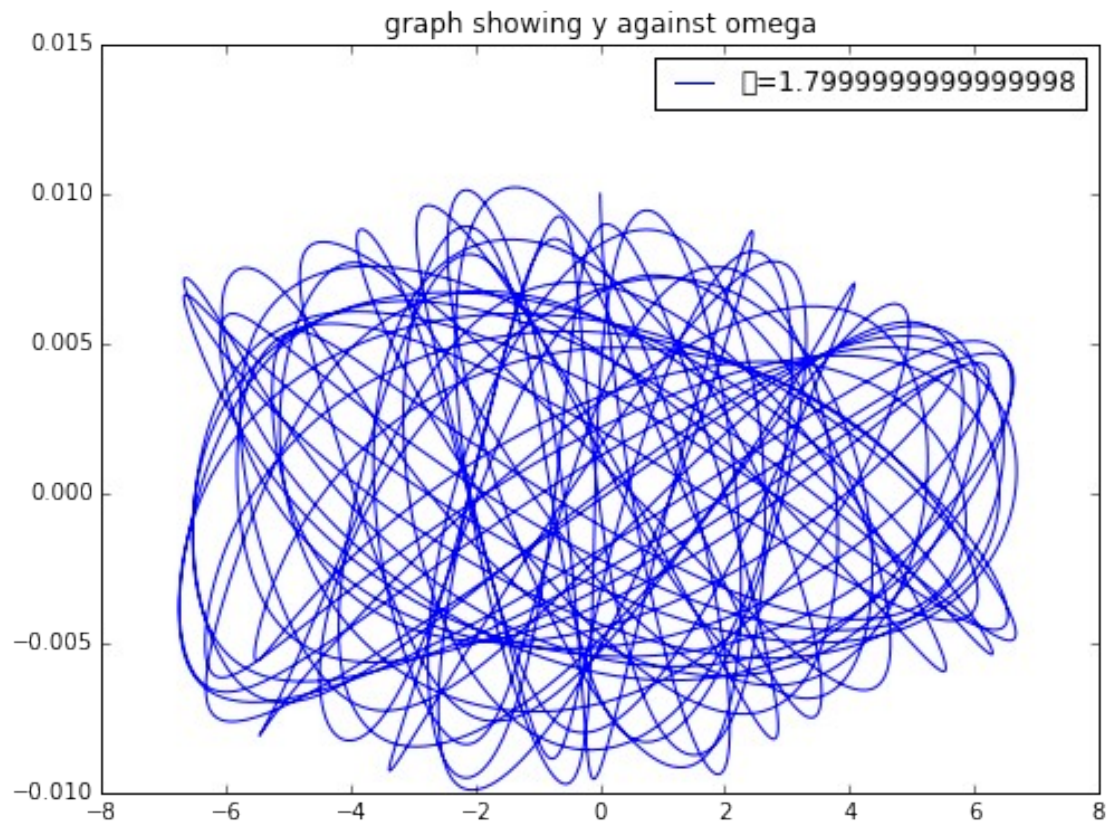
```
<matplotlib.figure.Figure at 0x7f78e17bd940>  
[<matplotlib.lines.Line2D at 0x7f7872e0d7f0>]  
<matplotlib.text.Text at 0x7f78d0b87780>  
<matplotlib.text.Text at 0x7f78c0bdcfd0>  
<matplotlib.text.Text at 0x7f78d0710358>  
<matplotlib.legend.Legend at 0x7f7872e0d0f0>
```



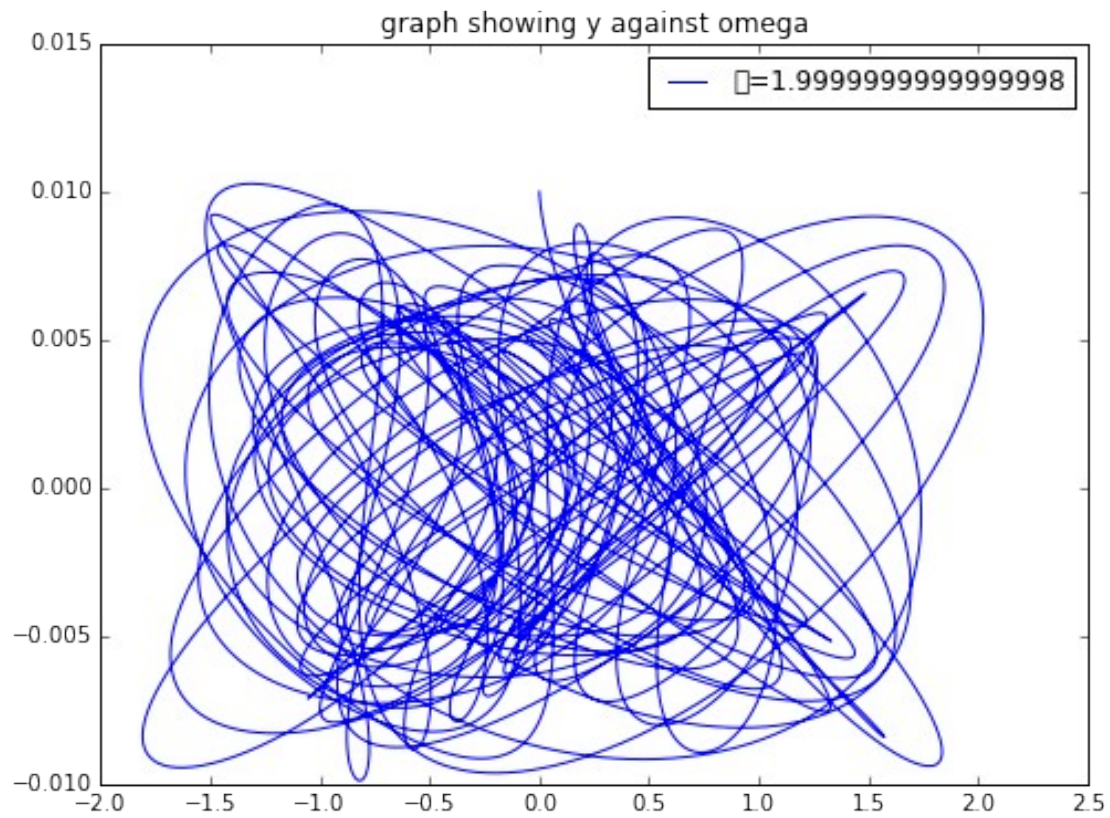
```
<matplotlib.figure.Figure at 0x7f78e5819d30>  
[<matplotlib.lines.Line2D at 0x7f78e1d8d6d8>]  
<matplotlib.text.Text at 0x7f78d13414e0>  
<matplotlib.text.Text at 0x7f78e1d07c18>  
<matplotlib.text.Text at 0x7f78c192eba8>  
<matplotlib.legend.Legend at 0x7f7873579c18>
```



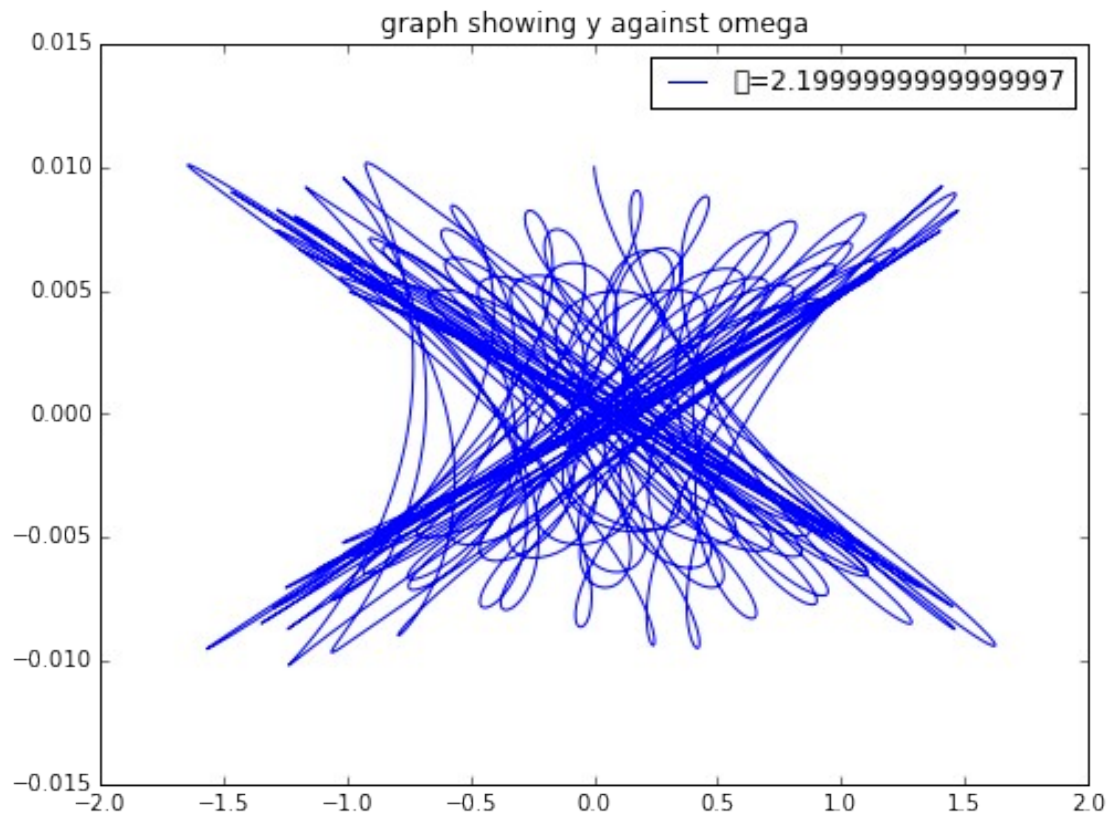
```
<matplotlib.figure.Figure at 0x7f78e1f366d8>  
[<matplotlib.lines.Line2D at 0x7f78d146ad68>]  
<matplotlib.text.Text at 0x7f78c01248d0>  
<matplotlib.text.Text at 0x7f78e1e8a7b8>  
<matplotlib.text.Text at 0x7f78e1e5a160>  
<matplotlib.legend.Legend at 0x7f7887fdcc50>
```



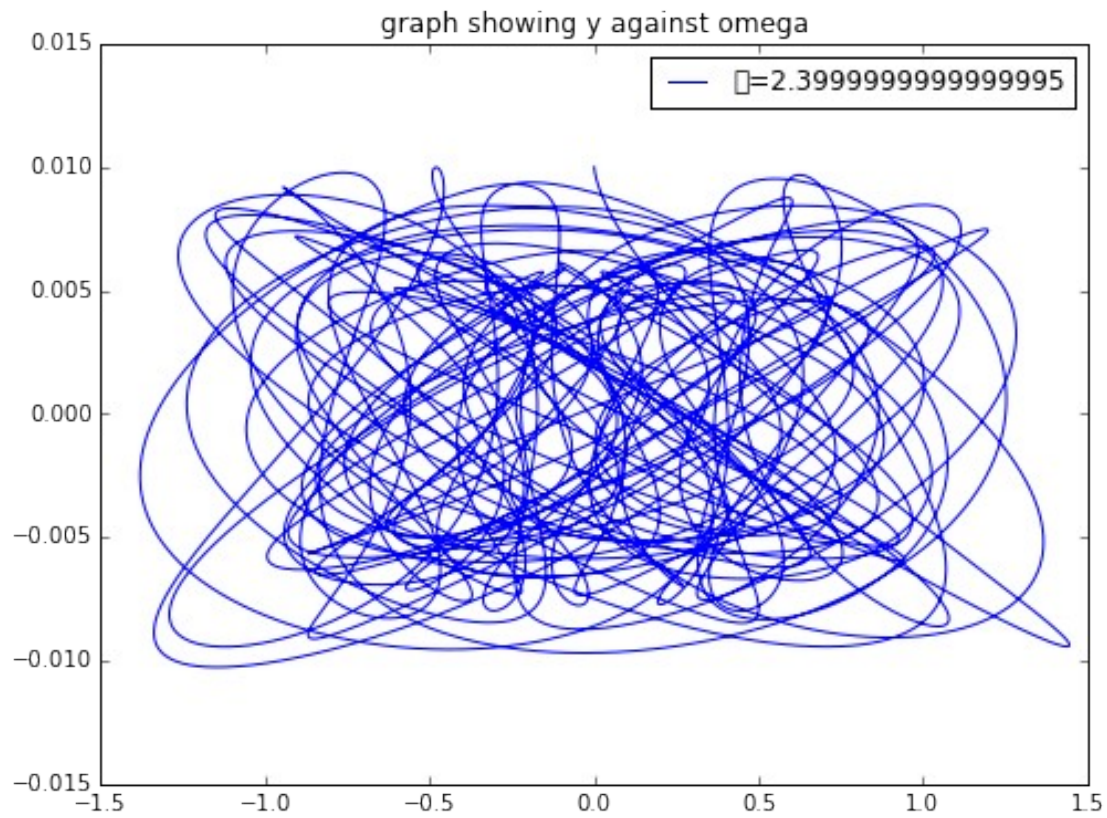
```
<matplotlib.figure.Figure at 0x7f78d1460860>  
[<matplotlib.lines.Line2D at 0x7f78d140fd30>]  
<matplotlib.text.Text at 0x7f78d0bd0ef0>  
<matplotlib.text.Text at 0x7f78d1458a20>  
<matplotlib.text.Text at 0x7f78d0707358>  
<matplotlib.legend.Legend at 0x7f78d140f240>
```

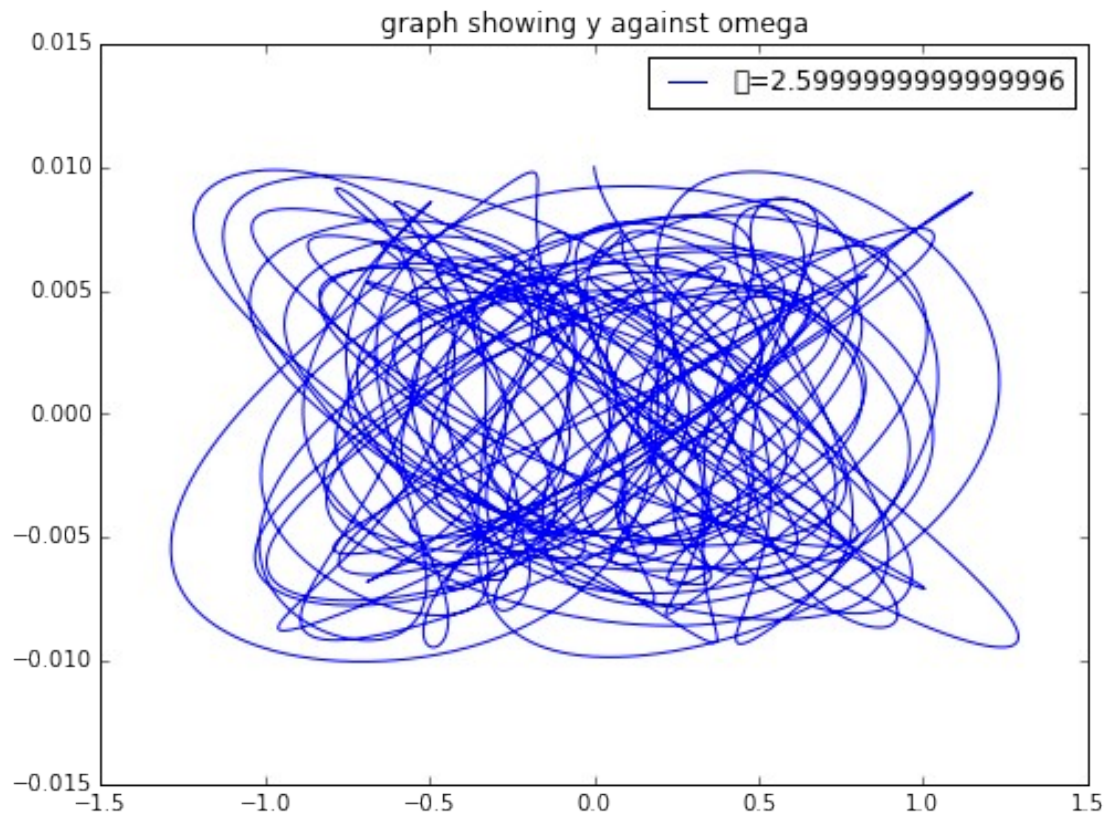
```
<matplotlib.figure.Figure at 0x7f78d13299b0>  
[<matplotlib.lines.Line2D at 0x7f78e1e6aa90>]  
<matplotlib.text.Text at 0x7f78e58126a0>  
<matplotlib.text.Text at 0x7f78c1fed160>  
<matplotlib.text.Text at 0x7f78c20cd828>  
<matplotlib.legend.Legend at 0x7f78d0b88a90>
```



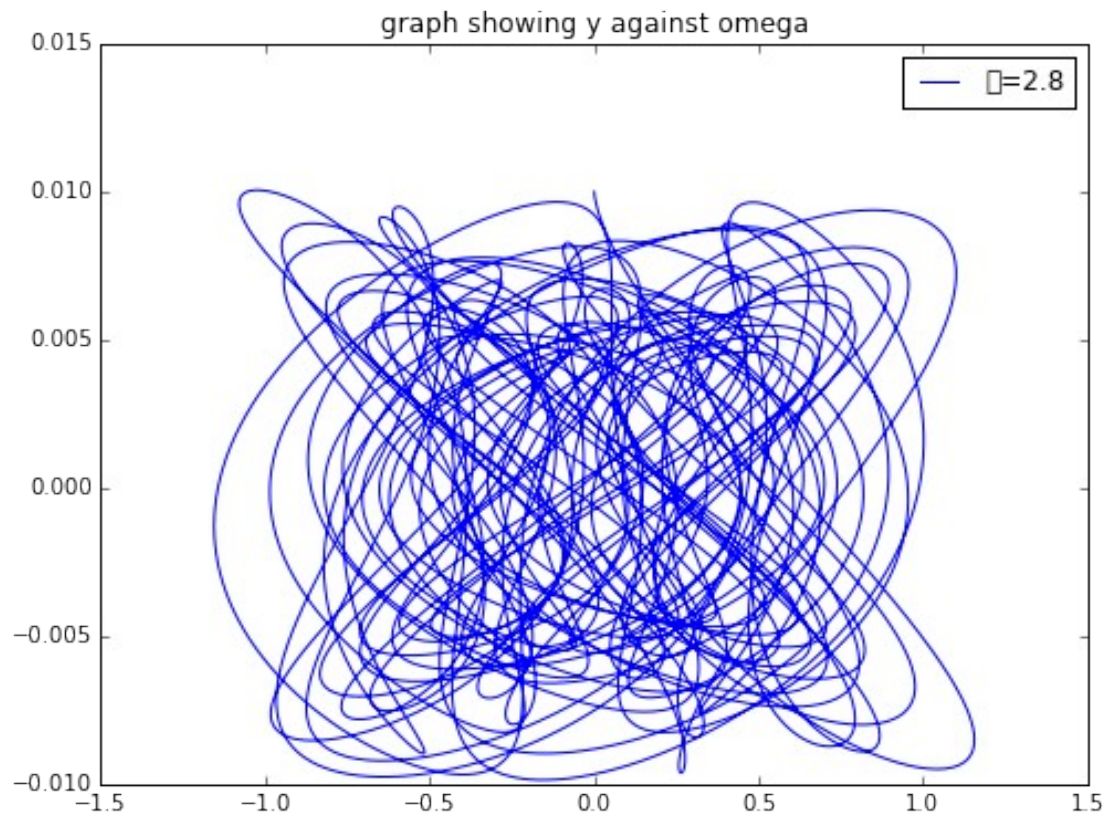
```
<matplotlib.figure.Figure at 0x7f78d133e8d0>  
[<matplotlib.lines.Line2D at 0x7f78e17d6240>]  
<matplotlib.text.Text at 0x7f78c2d8ba20>  
<matplotlib.text.Text at 0x7f78e1ceb630>  
<matplotlib.text.Text at 0x7f78e1e5e978>  
<matplotlib.legend.Legend at 0x7f78d06ea400>
```



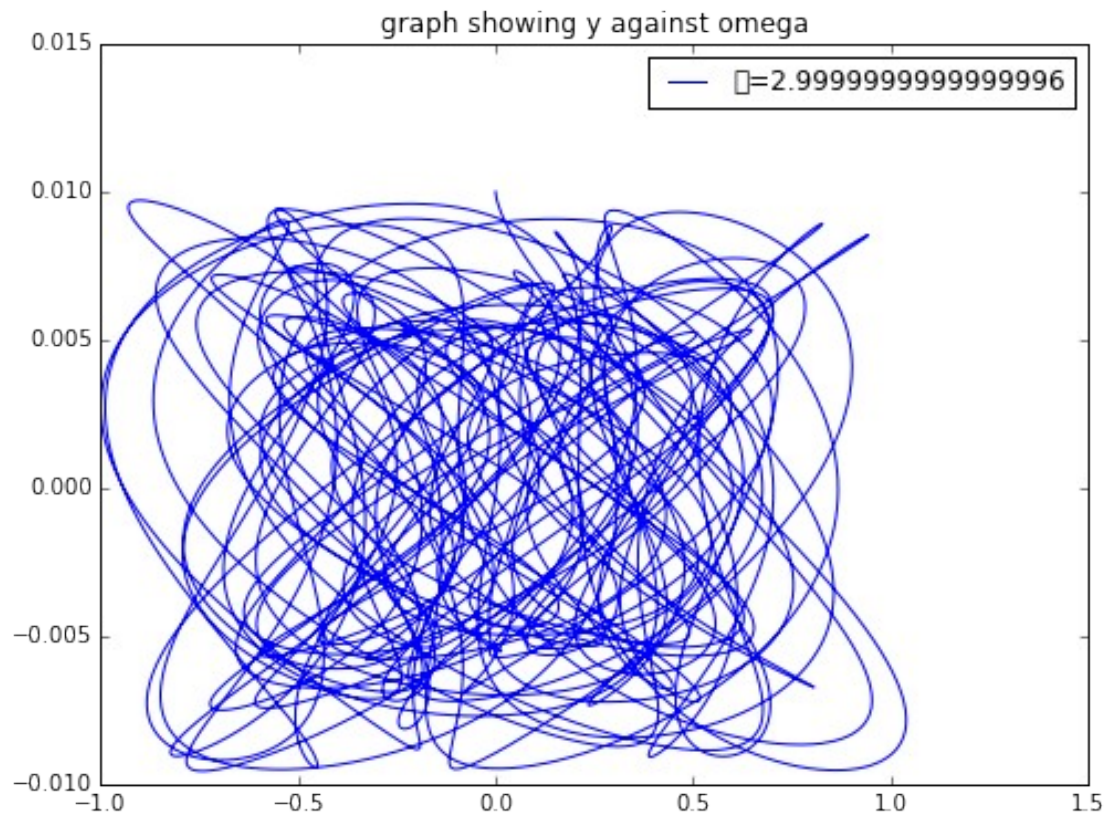
```
<matplotlib.figure.Figure at 0x7f78c2d92ba8>  
[<matplotlib.lines.Line2D at 0x7f78d132a978>]  
<matplotlib.text.Text at 0x7f78e1f3dc50>  
<matplotlib.text.Text at 0x7f78c2d857f0>  
<matplotlib.text.Text at 0x7f78e306d9b0>  
<matplotlib.legend.Legend at 0x7f78d06fa2b0>
```



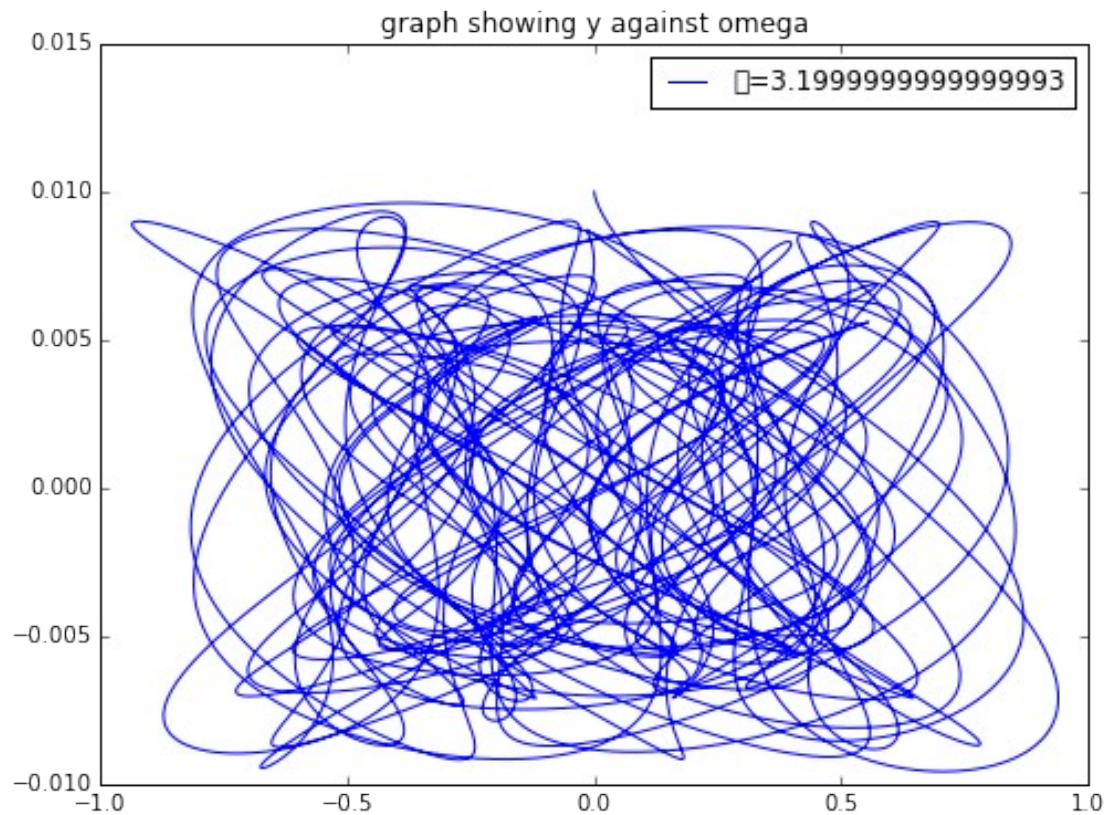
```
<matplotlib.figure.Figure at 0x7f78c2d9d400>  
[<matplotlib.lines.Line2D at 0x7f78e301d198>]  
<matplotlib.text.Text at 0x7f78c1fedcf8>  
<matplotlib.text.Text at 0x7f78e4193b70>  
<matplotlib.text.Text at 0x7f78c0874c50>  
<matplotlib.legend.Legend at 0x7f78e301dac8>
```

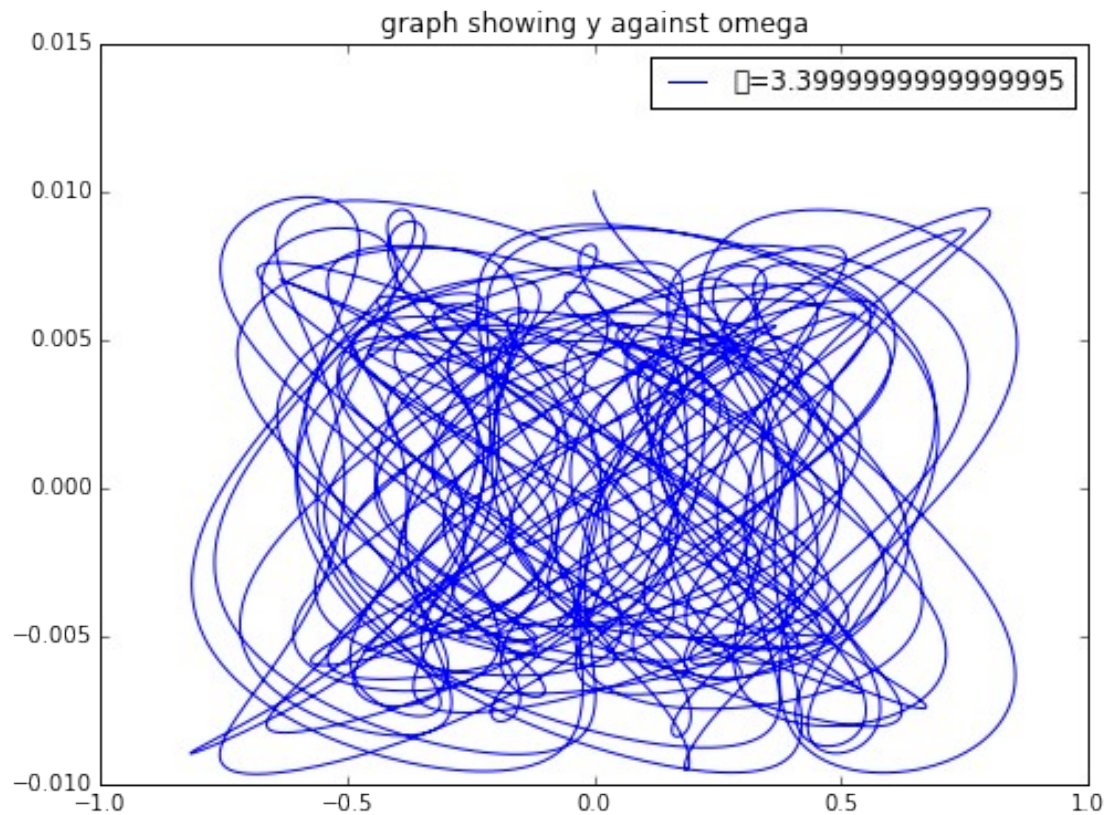
```
<matplotlib.figure.Figure at 0x7f78c1ff7a20>  
[<matplotlib.lines.Line2D at 0x7f78d05f06d8>]  
<matplotlib.text.Text at 0x7f78e3016da0>  
<matplotlib.text.Text at 0x7f78e2fea0f0>  
<matplotlib.text.Text at 0x7f78d0612ac8>  
<matplotlib.legend.Legend at 0x7f78d05f0f60>
```



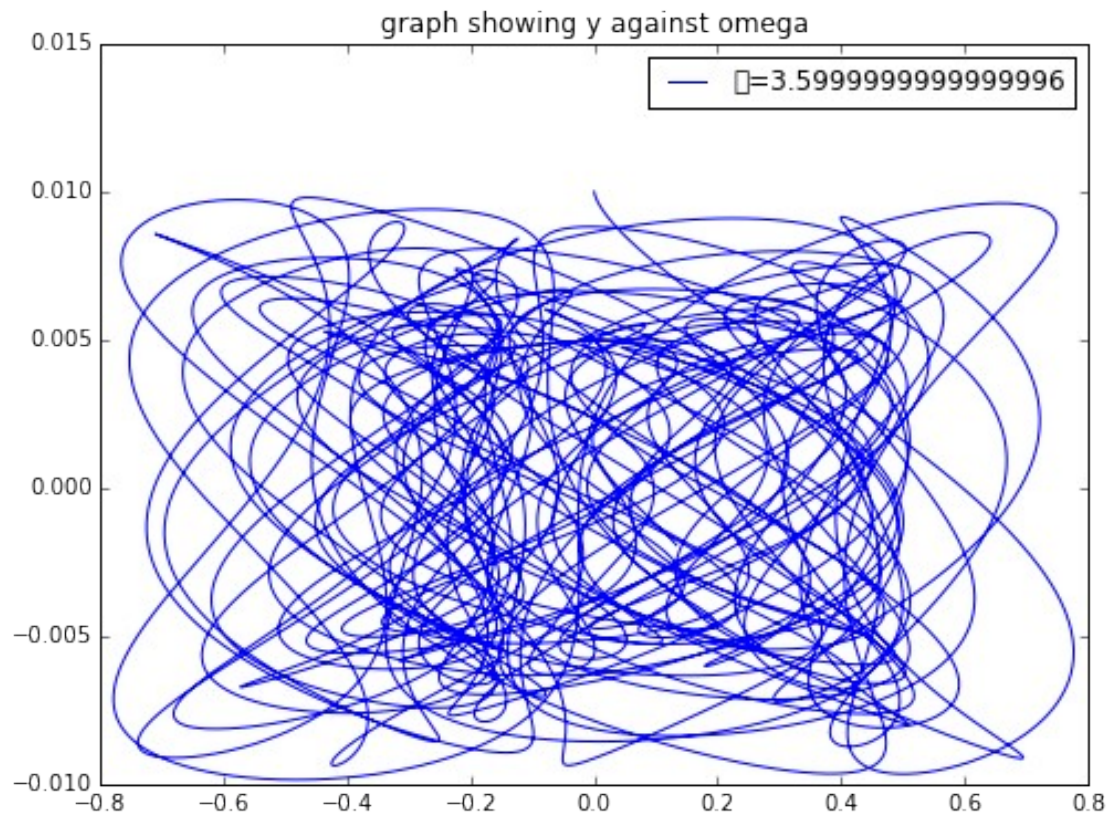
```
<matplotlib.figure.Figure at 0x7f78e1c6ae48>  
[<matplotlib.lines.Line2D at 0x7f78e301d198>]  
<matplotlib.text.Text at 0x7f78e41937b8>  
<matplotlib.text.Text at 0x7f78e30651d0>  
<matplotlib.text.Text at 0x7f78c087a358>  
<matplotlib.legend.Legend at 0x7f78e301d390>
```



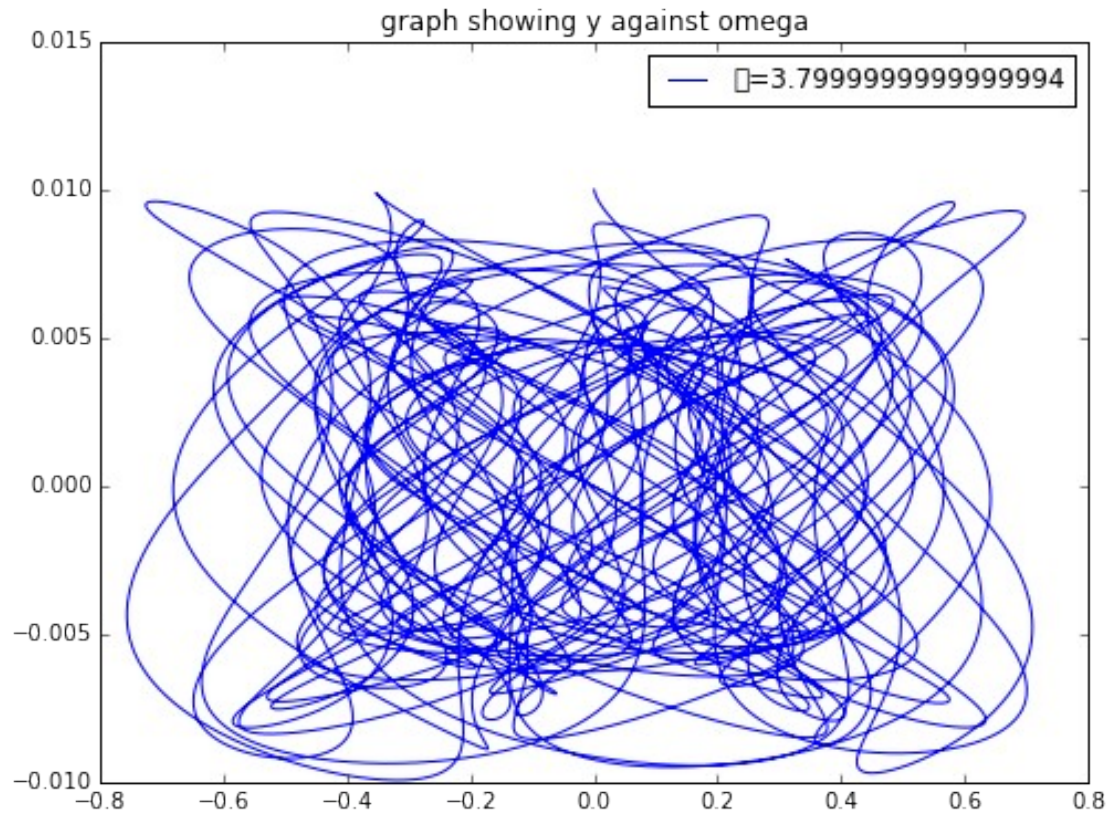
```
<matplotlib.figure.Figure at 0x7f78d132a4e0>  
[<matplotlib.lines.Line2D at 0x7f78c1fe65f8>]  
<matplotlib.text.Text at 0x7f78e1c703c8>  
<matplotlib.text.Text at 0x7f78e56f2dd8>  
<matplotlib.text.Text at 0x7f78c20c1cc0>  
<matplotlib.legend.Legend at 0x7f78e306db38>
```



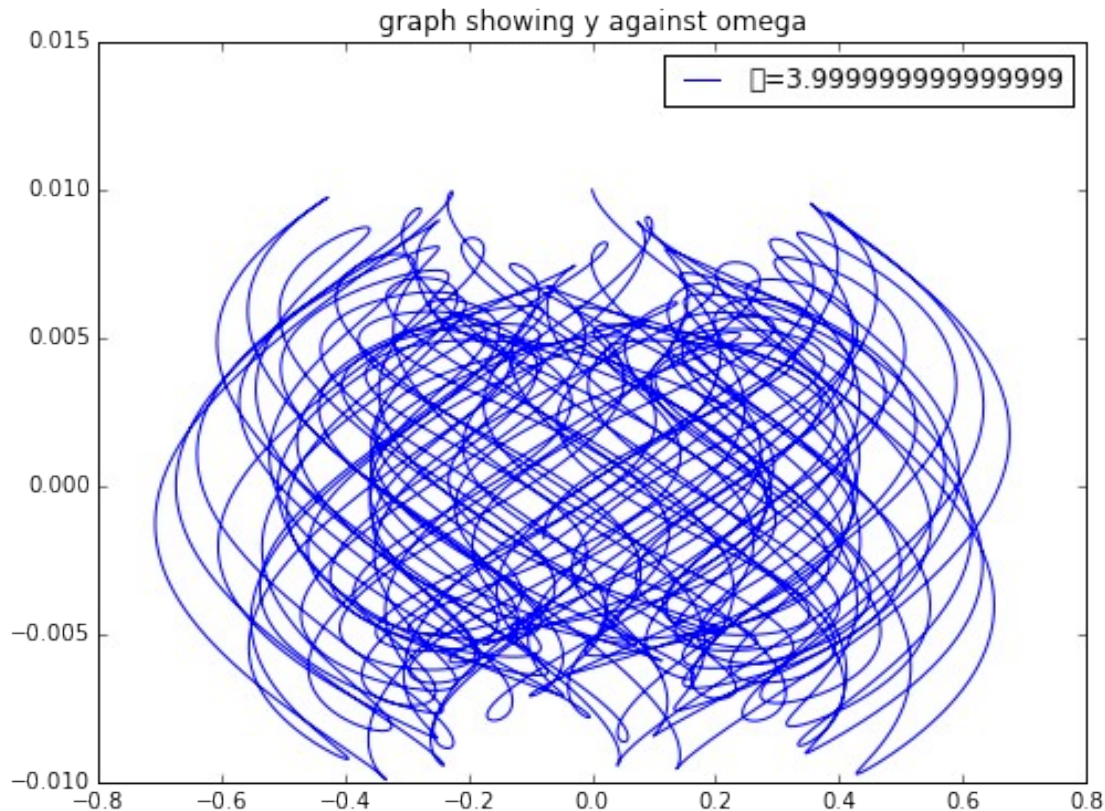
```
<matplotlib.figure.Figure at 0x7f78c20bde48>  
[<matplotlib.lines.Line2D at 0x7f78e180c780>]  
<matplotlib.text.Text at 0x7f78d070d780>  
<matplotlib.text.Text at 0x7f78c20b04a8>  
<matplotlib.text.Text at 0x7f78c2d8df98>  
<matplotlib.legend.Legend at 0x7f78e1e5e3c8>
```

```
<matplotlib.figure.Figure at 0x7f78d13a1f98>  
[<matplotlib.lines.Line2D at 0x7f78d30f07b8>]  
<matplotlib.text.Text at 0x7f78d2ec9dd8>  
<matplotlib.text.Text at 0x7f78d2dfc160>  
<matplotlib.text.Text at 0x7f78d0b81e10>  
<matplotlib.legend.Legend at 0x7f78c00b9320>
```



```
<matplotlib.figure.Figure at 0x7f78d12d8160>  
[<matplotlib.lines.Line2D at 0x7f78d146abe0>]  
<matplotlib.text.Text at 0x7f78d0b81160>  
<matplotlib.text.Text at 0x7f78e4165400>  
<matplotlib.text.Text at 0x7f78c0bdd9b0>  
<matplotlib.legend.Legend at 0x7f78d2eca588>
```



From the graphs above, we can see that the greatest y displacement occurs when ω is around 1. This means we can repeat the process with ω ranging from 0.9 - 1.1 to find a more accurate natural frequency.

```

ω_start=0.9
ω_end=1.1
ω=np.arange(ω_start, ω_end, 0.01)

for i in ω:
    times, θ, y = wind_tacoma(dt=0.001, cromer=False, y0=0, θ0=0.01,
z0=0, γ0=0, A=2, ω=i )
    plt.plot( y, θ, label='ω={0}'.format(i))
    plt.xlabel('y')
    plt.ylabel('theta')
    plt.title('graph showing y against omega')
    plt.legend()

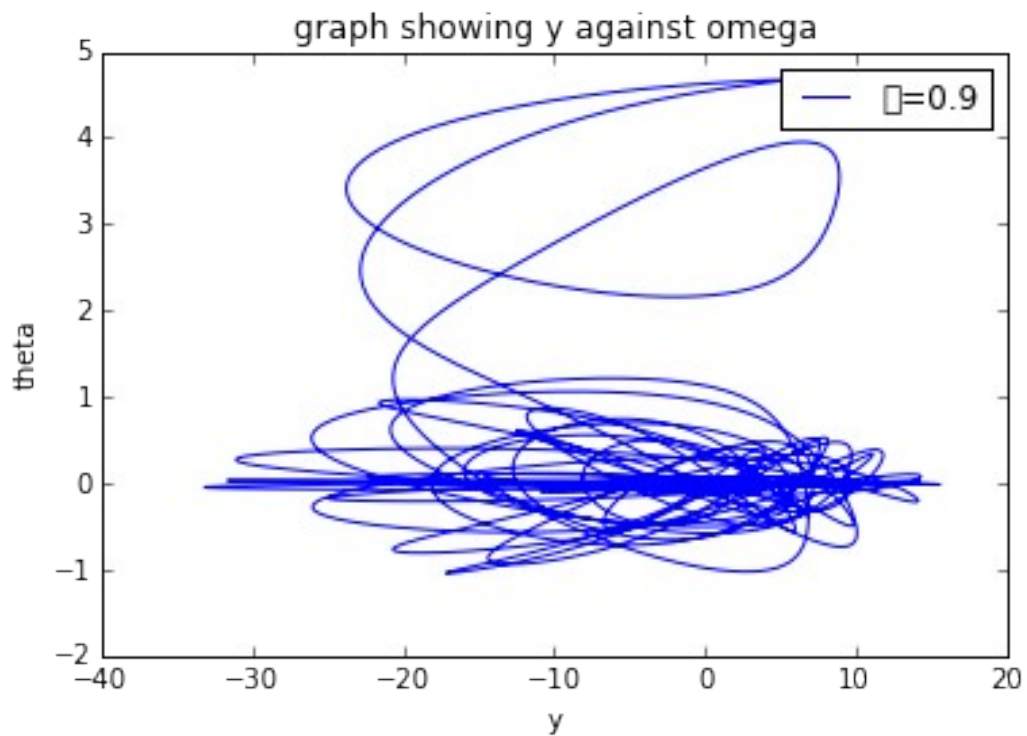
plt.show()

[<matplotlib.lines.Line2D at 0x7f7872dfc748>]
<matplotlib.text.Text at 0x7f78d0b78c88>
<matplotlib.text.Text at 0x7f78e418def0>

```

<matplotlib.text.Text at 0x7f78d0717c50>

<matplotlib.legend.Legend at 0x7f78c0e06eb8>



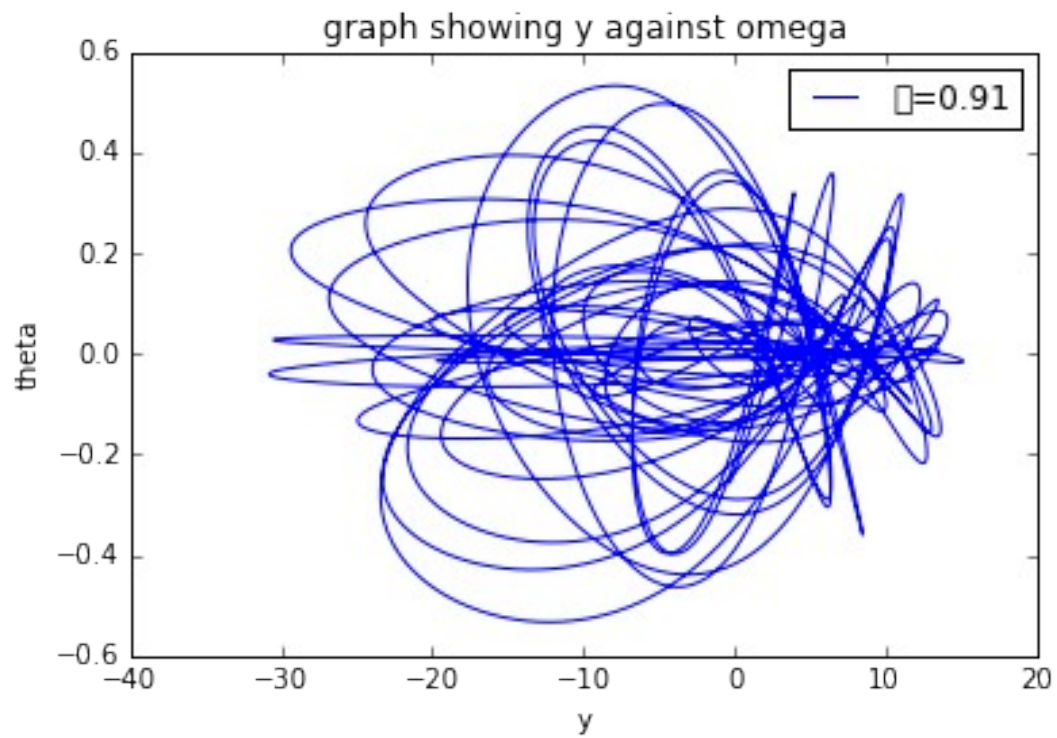
[<matplotlib.lines.Line2D at 0x7f78c0befa58>]

<matplotlib.text.Text at 0x7f78e1e8aac8>

<matplotlib.text.Text at 0x7f7873564518>

<matplotlib.text.Text at 0x7f78e418eac8>

<matplotlib.legend.Legend at 0x7f78e1e77b38>



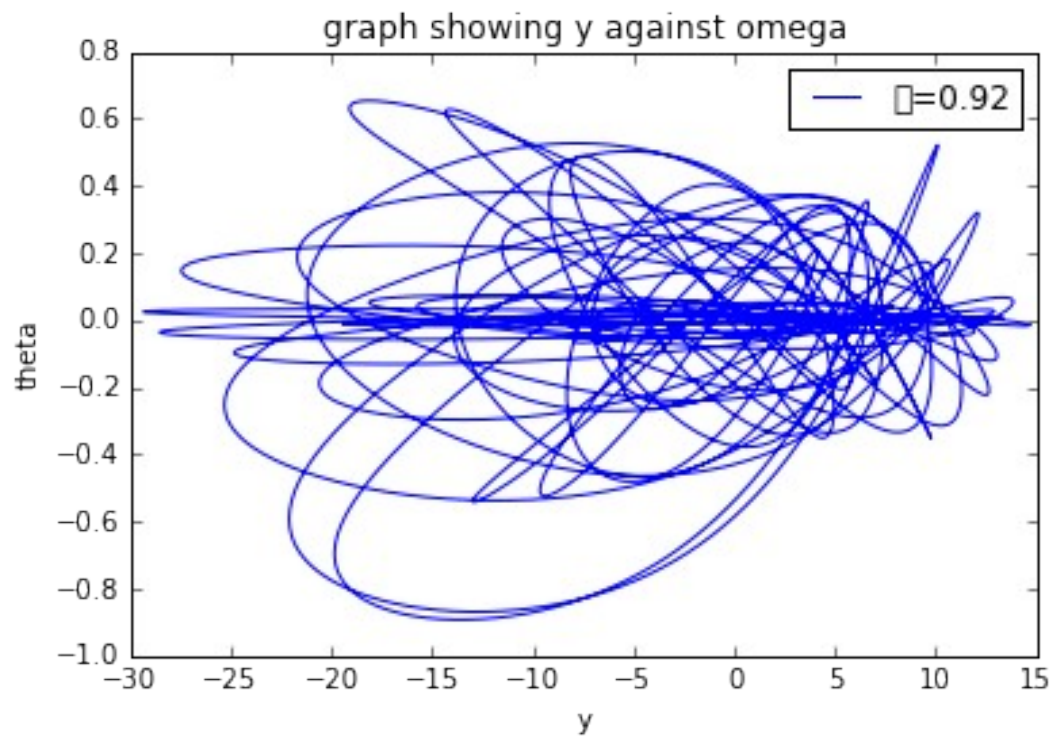
[<matplotlib.lines.Line2D at 0x7f78c00ce390>]

<matplotlib.text.Text at 0x7f78e17d7be0>

<matplotlib.text.Text at 0x7f78d13d2978>

<matplotlib.text.Text at 0x7f7872e0de80>

<matplotlib.legend.Legend at 0x7f78c00cea90>



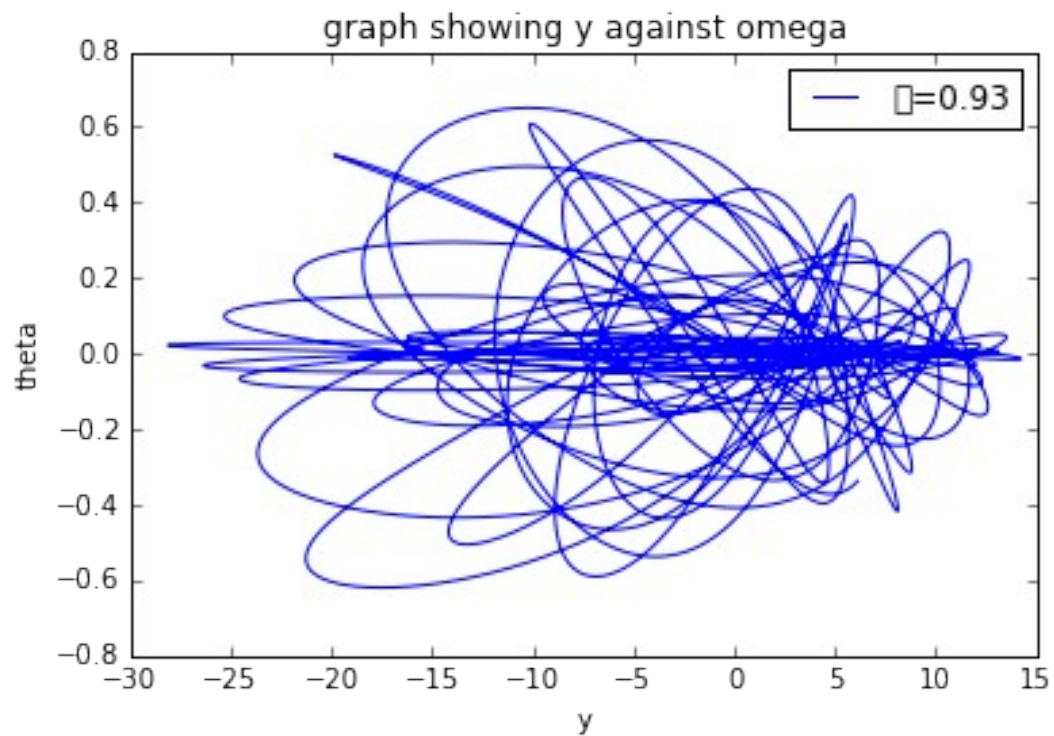
[<matplotlib.lines.Line2D at 0x7f78d0716240>]

<matplotlib.text.Text at 0x7f78d0710cf8>

<matplotlib.text.Text at 0x7f7887fe1e48>

<matplotlib.text.Text at 0x7f78e4189940>

<matplotlib.legend.Legend at 0x7f78e41810b8>



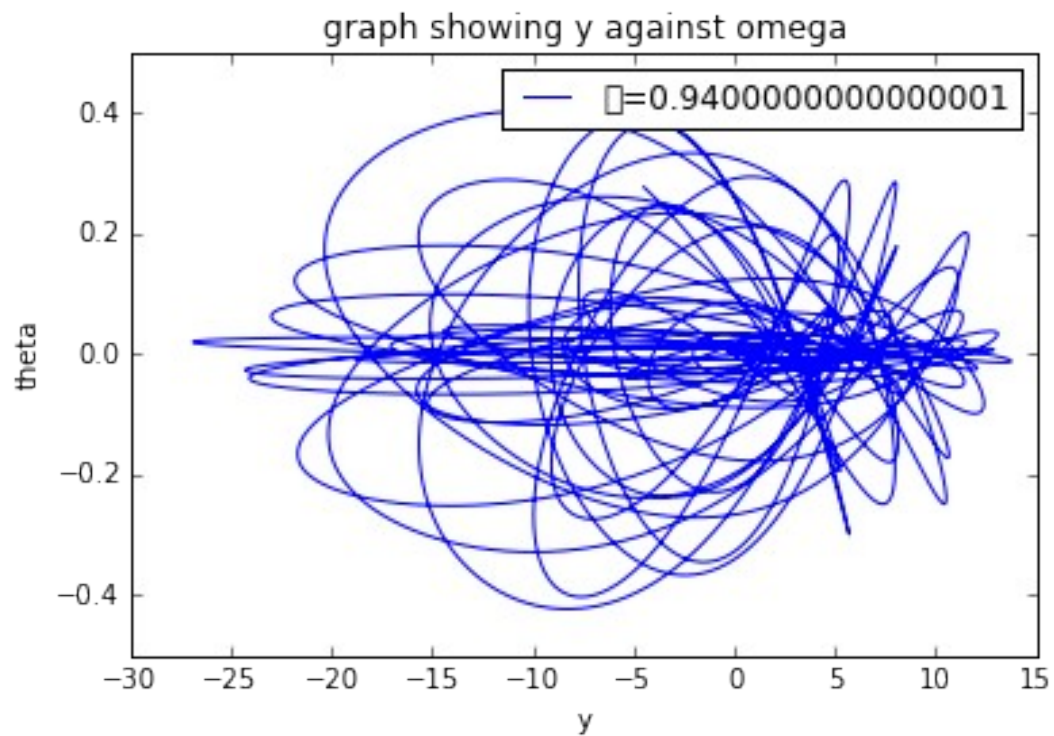
[<matplotlib.lines.Line2D at 0x7f78d1342630>]

<matplotlib.text.Text at 0x7f78e41895c0>

<matplotlib.text.Text at 0x7f78c0beb4a8>

<matplotlib.text.Text at 0x7f78d140e5f8>

<matplotlib.legend.Legend at 0x7f78c20f3a20>



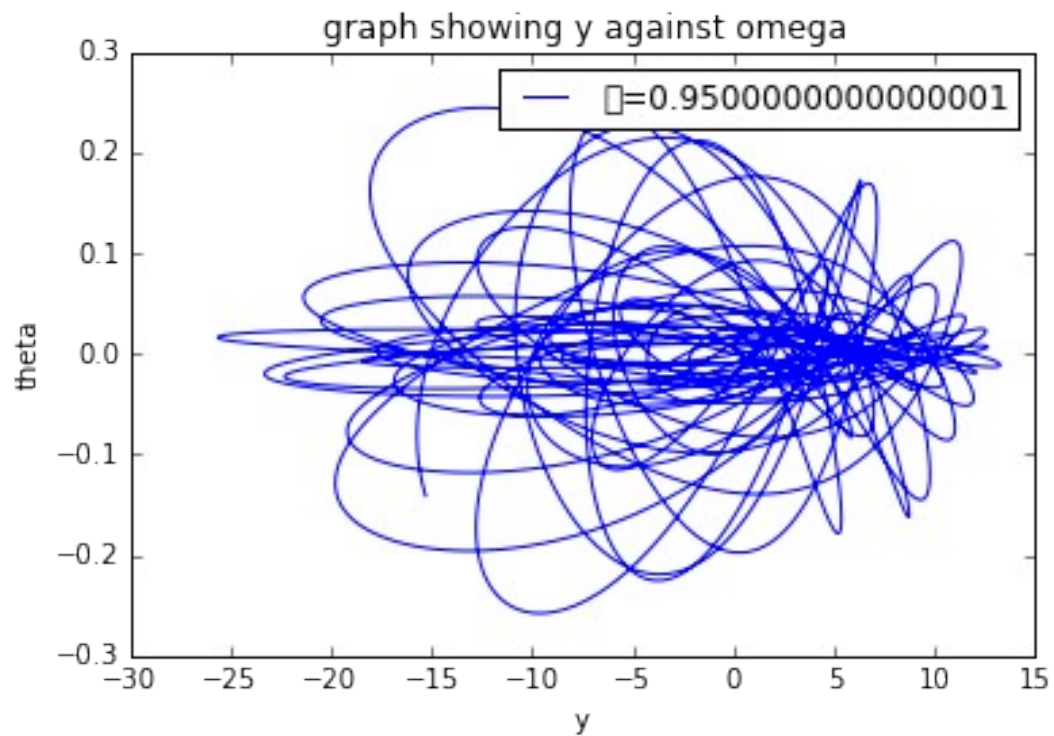
[<matplotlib.lines.Line2D at 0x7f78e5797828>]

<matplotlib.text.Text at 0x7f78c08607b8>

<matplotlib.text.Text at 0x7f78d134d4a8>

<matplotlib.text.Text at 0x7f78c1fc44e0>

<matplotlib.legend.Legend at 0x7f78e57a7160>



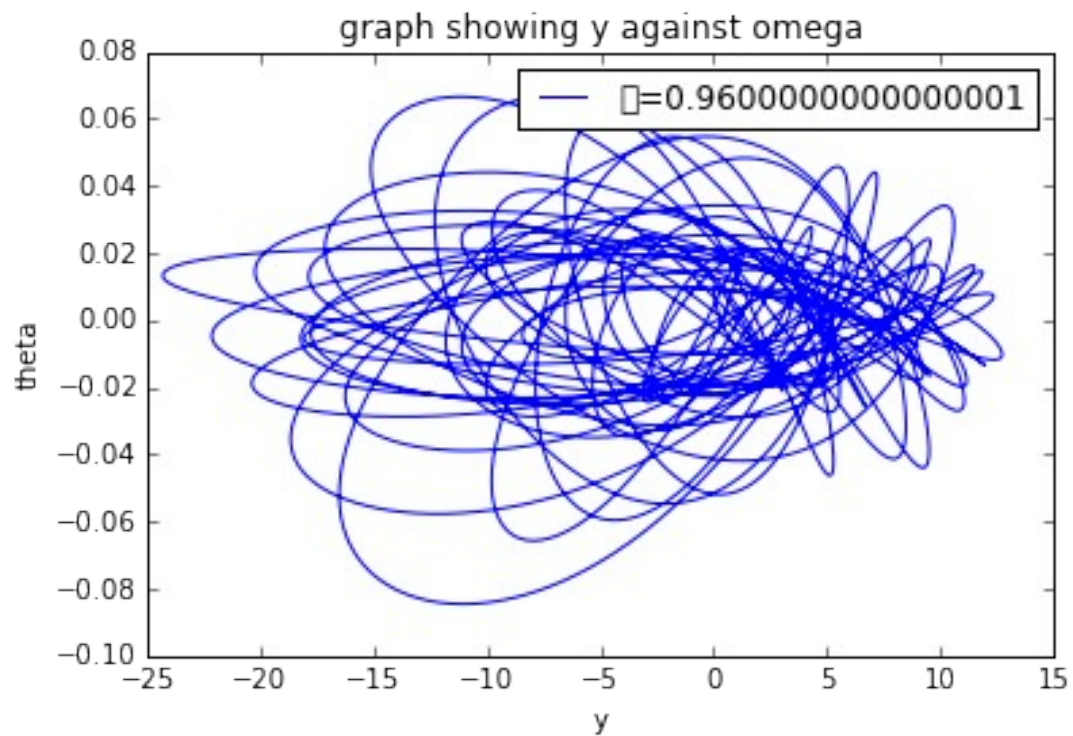
[<matplotlib.lines.Line2D at 0x7f78e1c85438>]

<matplotlib.text.Text at 0x7f78e5788860>

<matplotlib.text.Text at 0x7f78e579c358>

<matplotlib.text.Text at 0x7f78e1c877b8>

<matplotlib.legend.Legend at 0x7f78e1c85390>



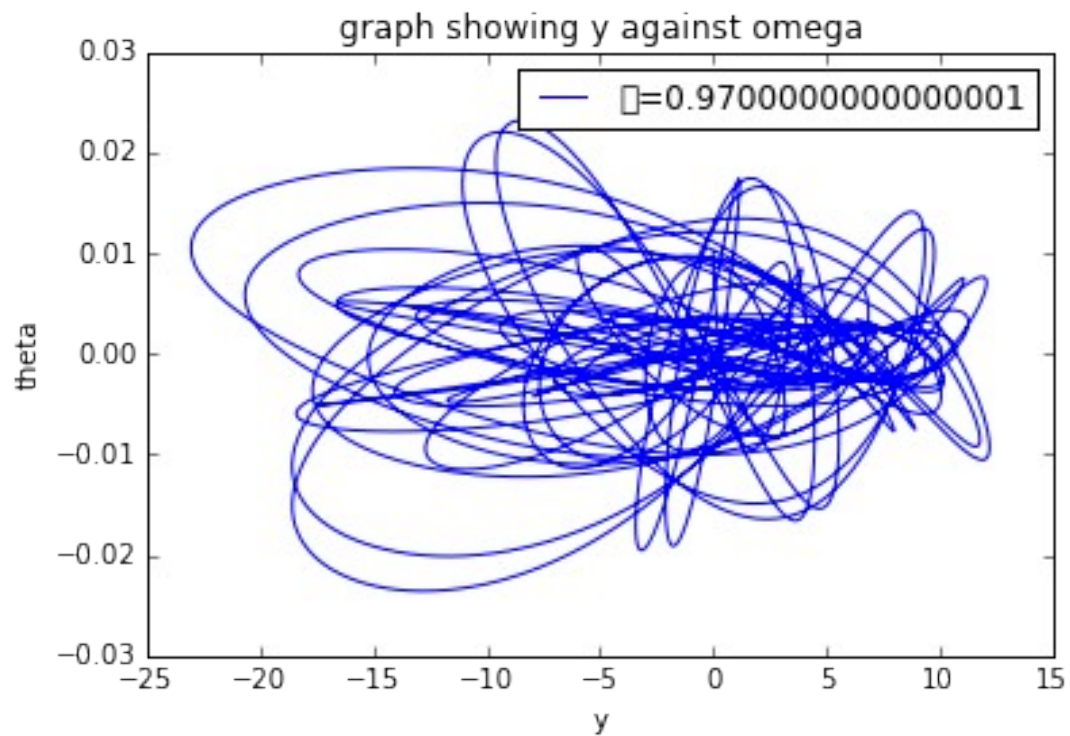
[<matplotlib.lines.Line2D at 0x7f78c2d8b7f0>]

<matplotlib.text.Text at 0x7f78c2d873c8>

<matplotlib.text.Text at 0x7f78d3761c88>

<matplotlib.text.Text at 0x7f78d06e3b38>

<matplotlib.legend.Legend at 0x7f78e4181748>



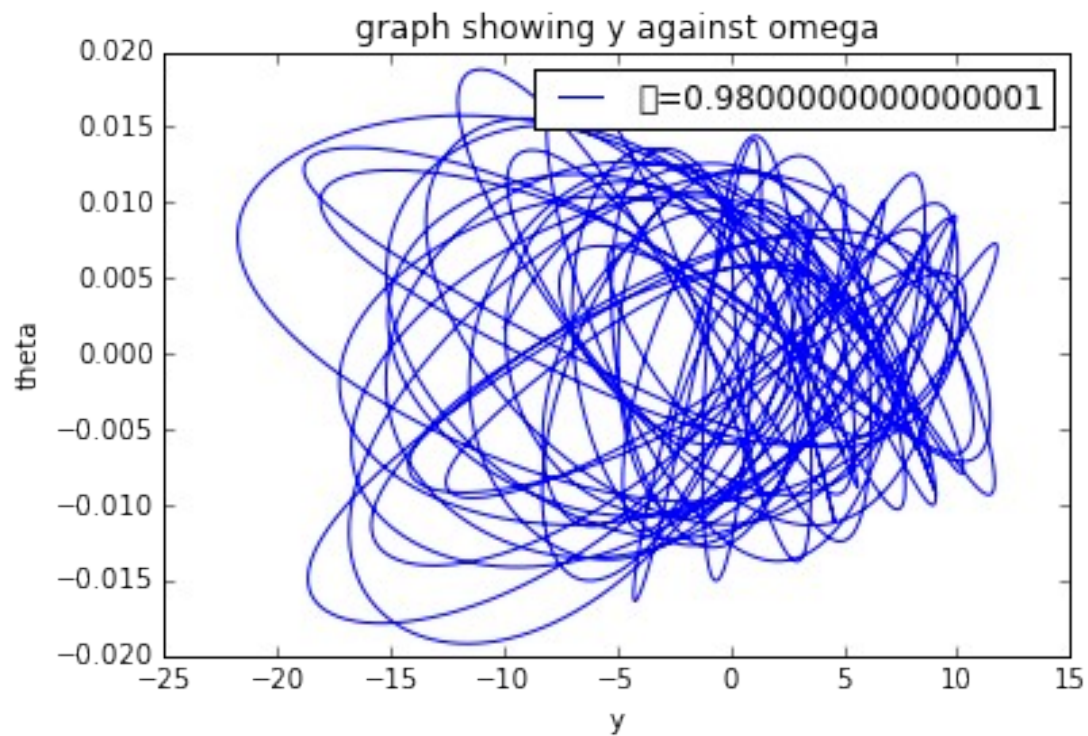
[<matplotlib.lines.Line2D at 0x7f7887fe1668>]

<matplotlib.text.Text at 0x7f78e58141d0>

<matplotlib.text.Text at 0x7f7872e09358>

<matplotlib.text.Text at 0x7f78e1f1b9b0>

<matplotlib.legend.Legend at 0x7f7887fe1ba8>



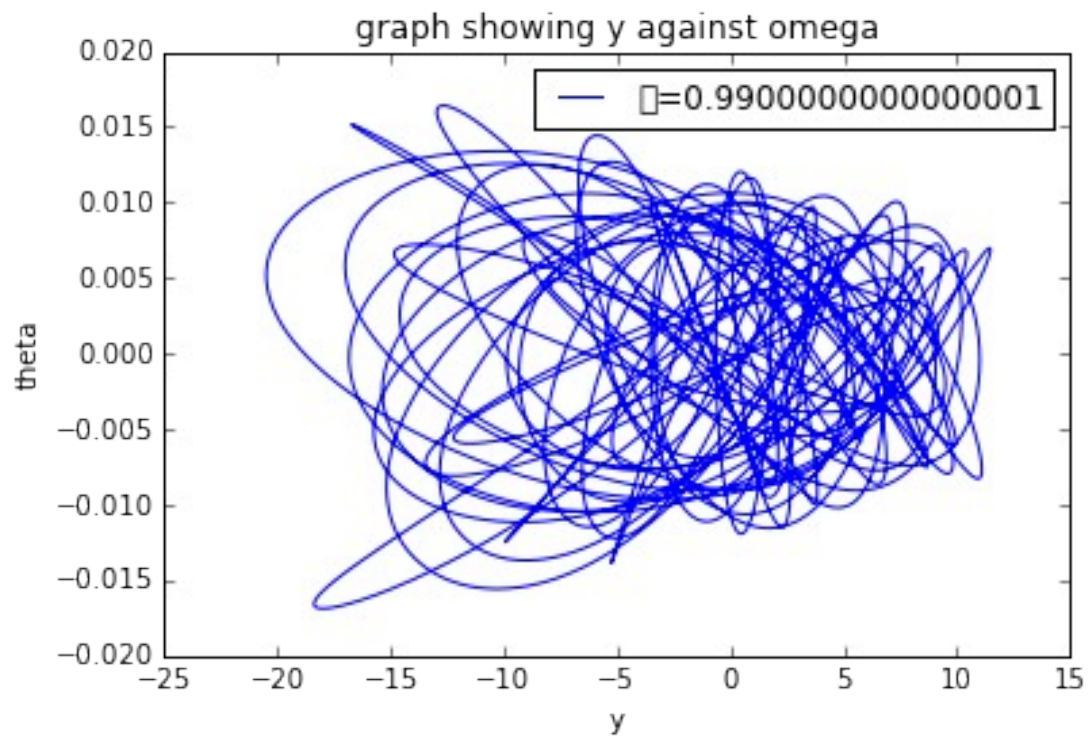
[<matplotlib.lines.Line2D at 0x7f78c0befb38>]

<matplotlib.text.Text at 0x7f78c0bd14e0>

<matplotlib.text.Text at 0x7f78d30b9710>

<matplotlib.text.Text at 0x7f78e418e860>

<matplotlib.legend.Legend at 0x7f78c0bef048>



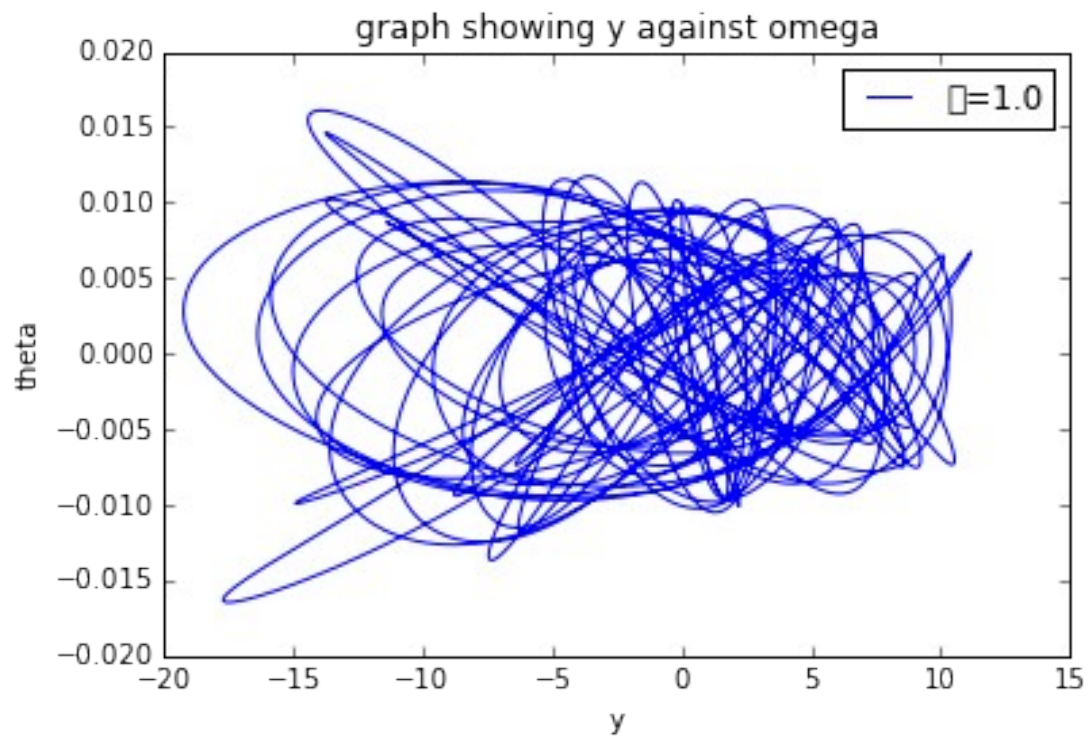
[<matplotlib.lines.Line2D at 0x7f787357da20>]

<matplotlib.text.Text at 0x7f78c192e550>

<matplotlib.text.Text at 0x7f7873579b38>

<matplotlib.text.Text at 0x7f78d070c668>

<matplotlib.legend.Legend at 0x7f78c0e06860>



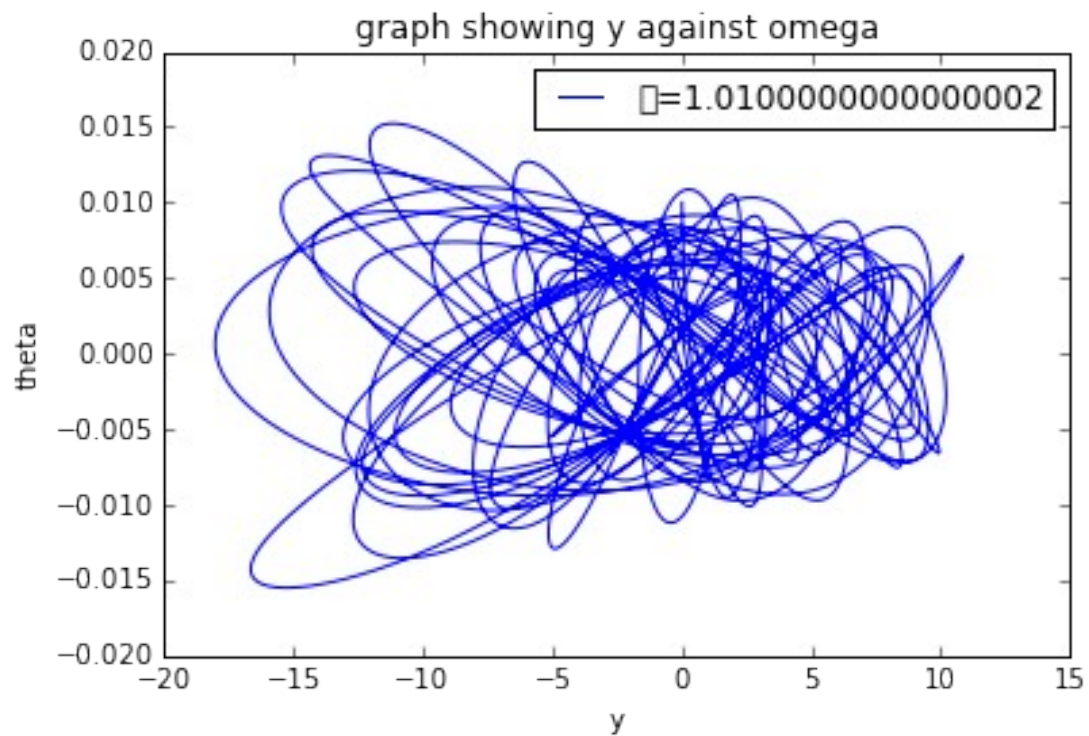
[<matplotlib.lines.Line2D at 0x7f78d13f8400>]

<matplotlib.text.Text at 0x7f78e1f36ef0>

<matplotlib.text.Text at 0x7f78e418a518>

<matplotlib.text.Text at 0x7f7872e277f0>

<matplotlib.legend.Legend at 0x7f7887fc4518>



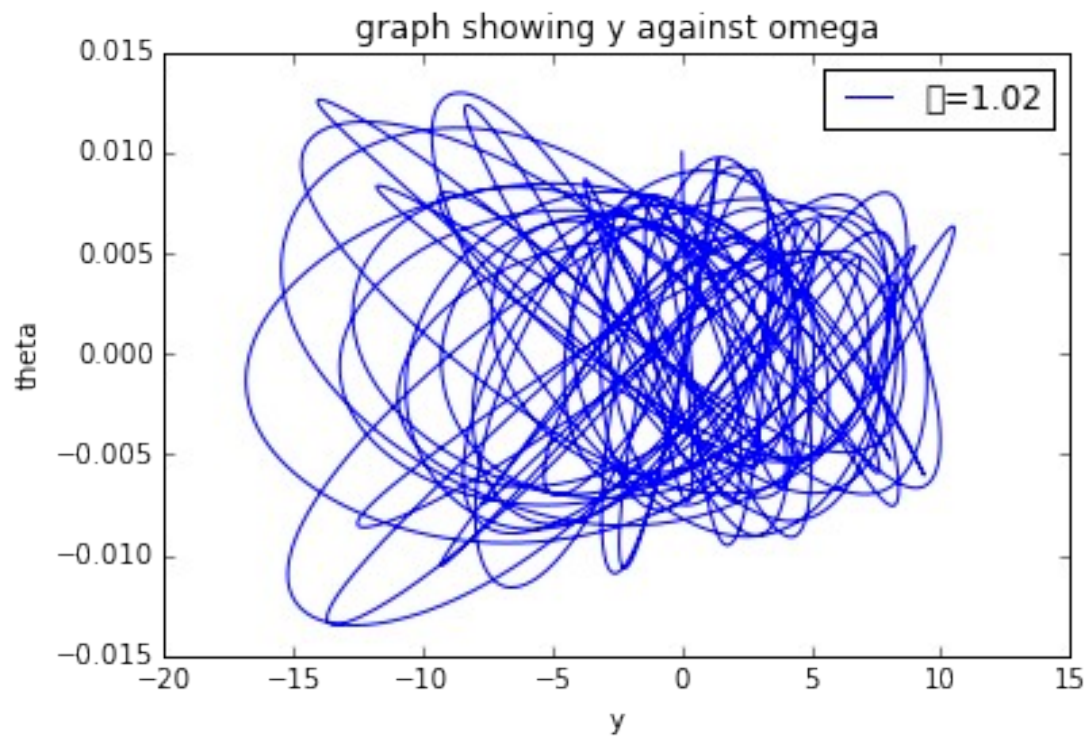
[<matplotlib.lines.Line2D at 0x7f78e5844a90>]

<matplotlib.text.Text at 0x7f78e58080b8>

<matplotlib.text.Text at 0x7f78e17cc0b8>

<matplotlib.text.Text at 0x7f78e4160b00>

<matplotlib.legend.Legend at 0x7f78d06f7978>



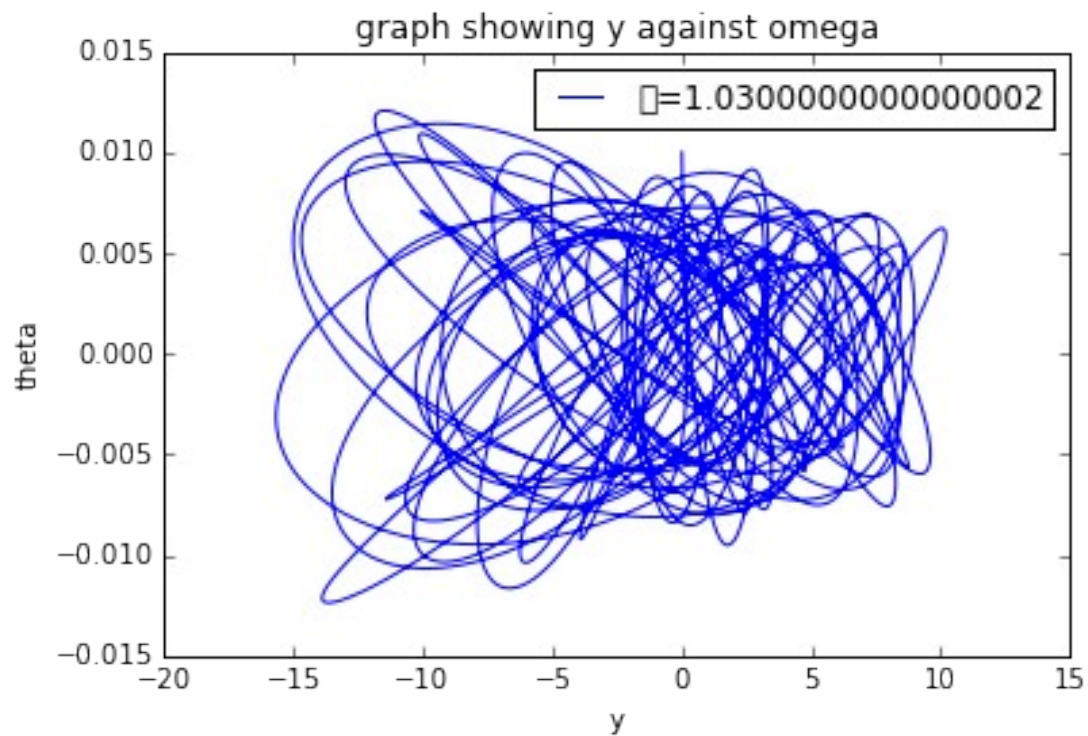
[<matplotlib.lines.Line2D at 0x7f78c00ddef0>]

<matplotlib.text.Text at 0x7f78c1fe6940>

<matplotlib.text.Text at 0x7f78c20a52e8>

<matplotlib.text.Text at 0x7f78d13d5588>

<matplotlib.legend.Legend at 0x7f7887fbf4a8>



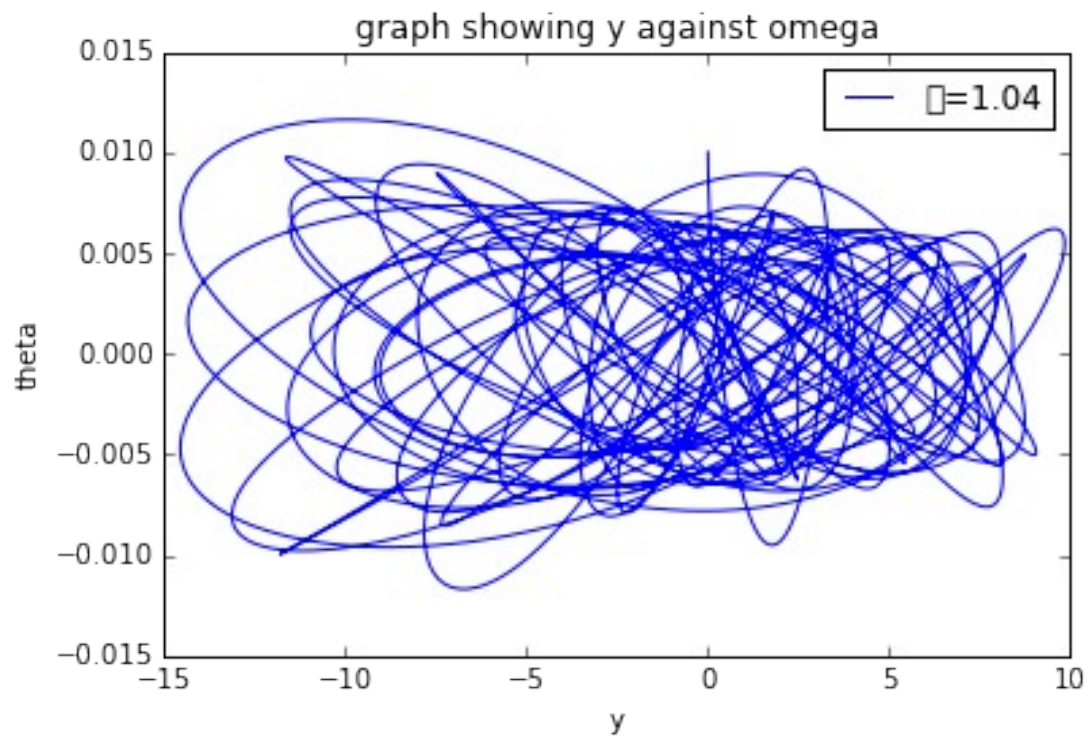
[<matplotlib.lines.Line2D at 0x7f78c087b940>]

<matplotlib.text.Text at 0x7f78e1c626d8>

<matplotlib.text.Text at 0x7f78c2d92748>

<matplotlib.text.Text at 0x7f78c2d926d8>

<matplotlib.legend.Legend at 0x7f78c085e278>



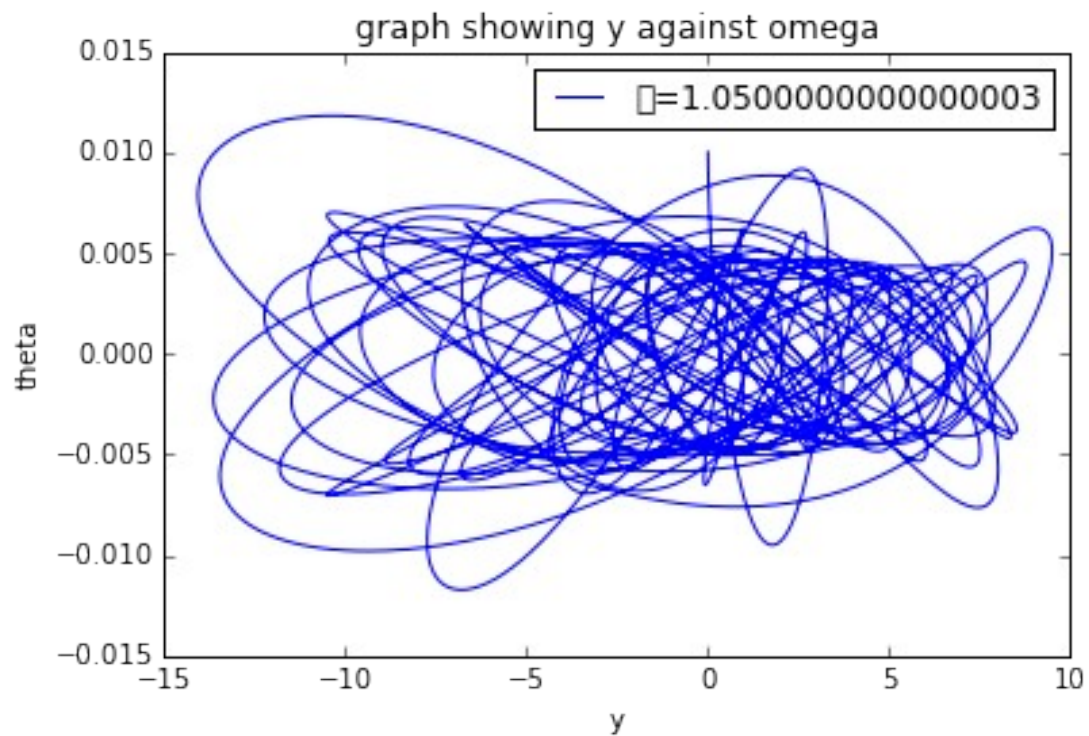
[<matplotlib.lines.Line2D at 0x7f78d060fb38>]

<matplotlib.text.Text at 0x7f78e1c6a470>

<matplotlib.text.Text at 0x7f78d06fa780>

<matplotlib.text.Text at 0x7f78e1c52cf8>

<matplotlib.legend.Legend at 0x7f78d05f6cc0>



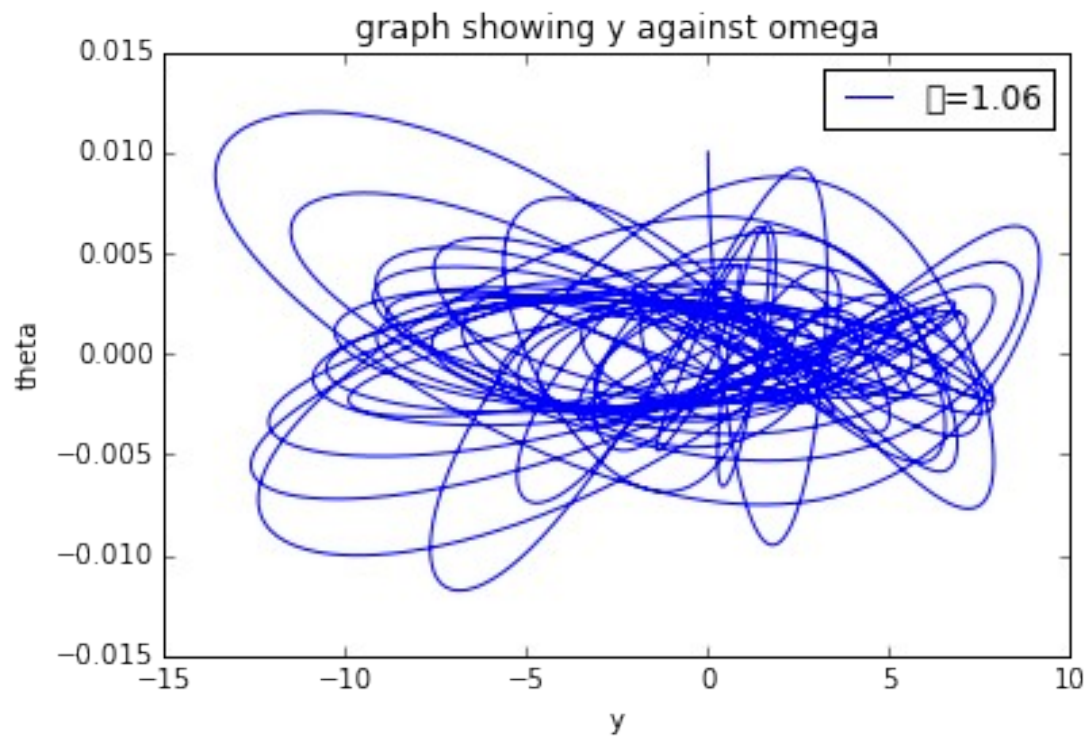
[<matplotlib.lines.Line2D at 0x7f78e1ed9748>]

<matplotlib.text.Text at 0x7f78e1c7fda0>

<matplotlib.text.Text at 0x7f78e301e0b8>

<matplotlib.text.Text at 0x7f78e1eb1470>

<matplotlib.legend.Legend at 0x7f78e1ed9f98>



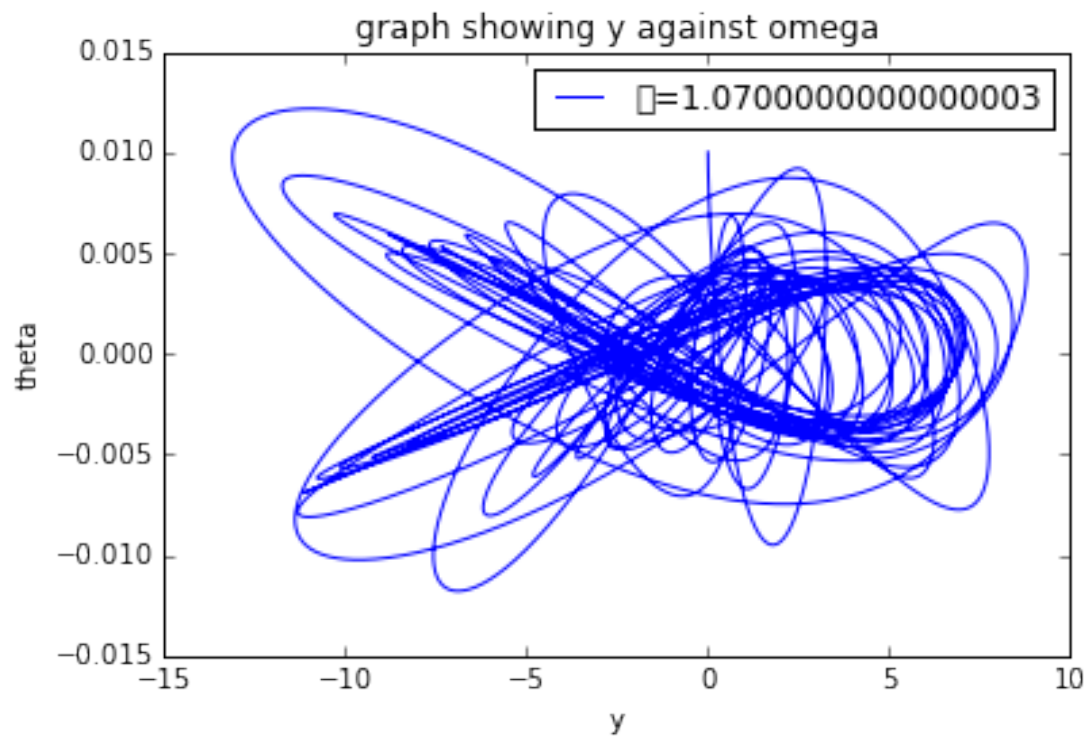
[<matplotlib.lines.Line2D at 0x7f78d05f6a58>]

<matplotlib.text.Text at 0x7f78e1c17d68>

<matplotlib.text.Text at 0x7f78e1c18978>

<matplotlib.text.Text at 0x7f78d0614780>

<matplotlib.legend.Legend at 0x7f78d05f6a90>



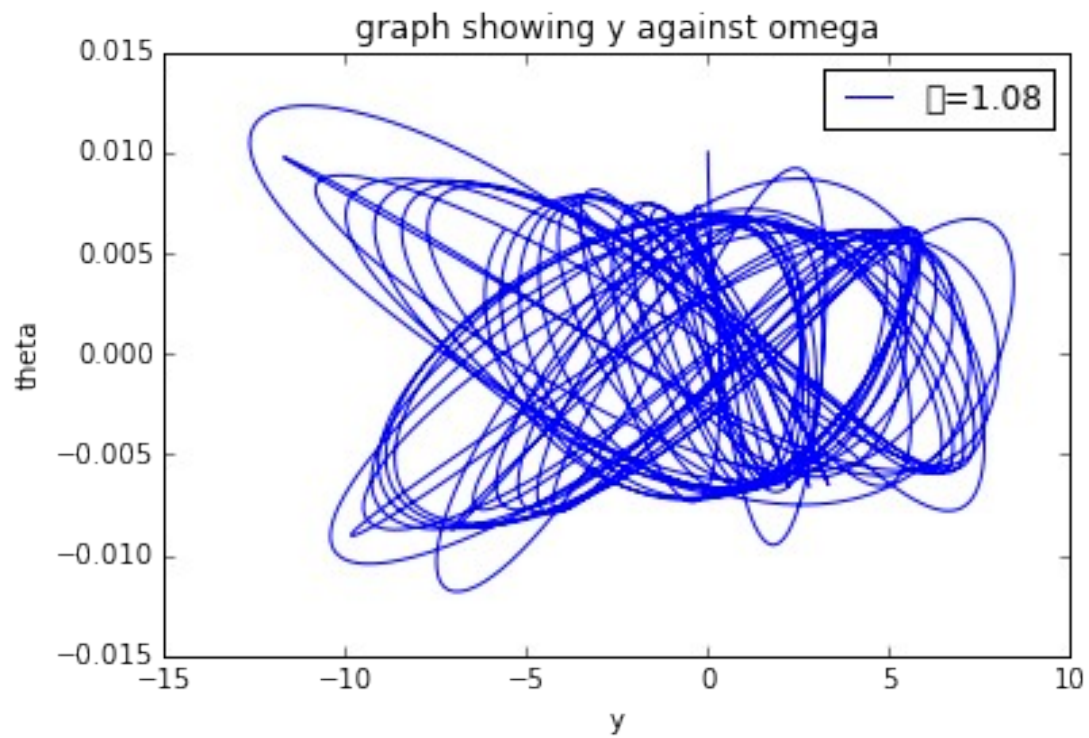
[<matplotlib.lines.Line2D at 0x7f78d0bd2240>]

<matplotlib.text.Text at 0x7f78d05e5080>

<matplotlib.text.Text at 0x7f78e1c35b00>

<matplotlib.text.Text at 0x7f78e3005668>

<matplotlib.legend.Legend at 0x7f78c1933630>



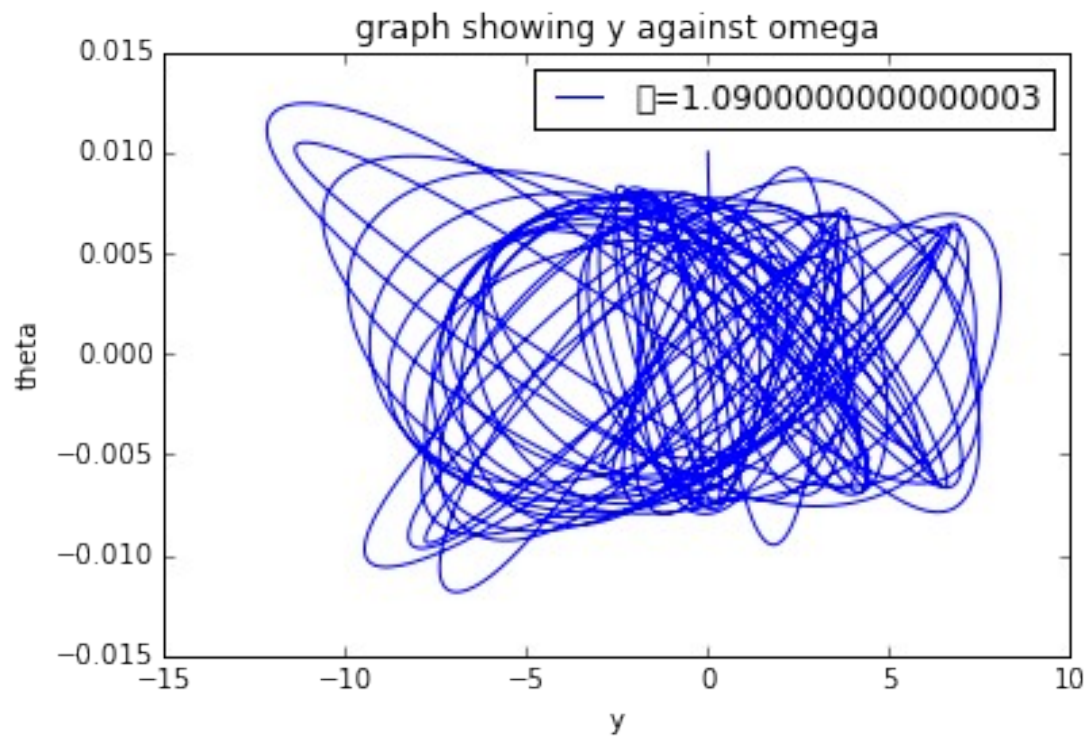
[<matplotlib.lines.Line2D at 0x7f78e5723390>]

<matplotlib.text.Text at 0x7f78d13a9908>

<matplotlib.text.Text at 0x7f78e1c3d908>

<matplotlib.text.Text at 0x7f78d070df28>

<matplotlib.legend.Legend at 0x7f78d132af60>



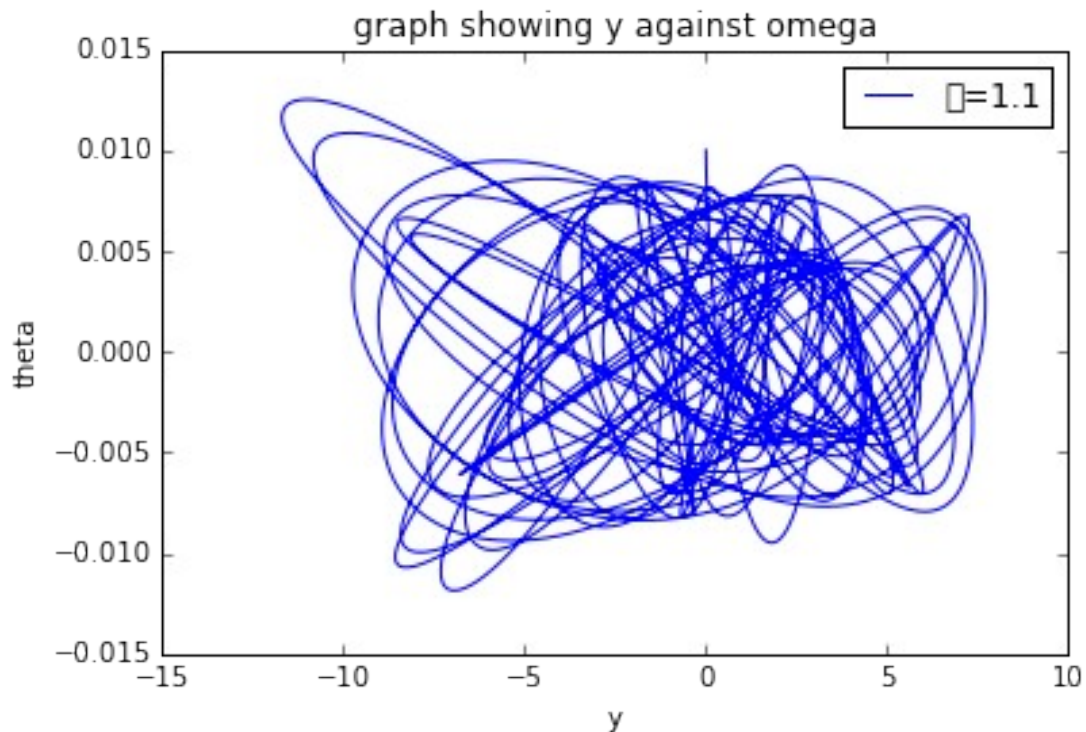
[<matplotlib.lines.Line2D at 0x7f7887fb85c0>]

<matplotlib.text.Text at 0x7f78c20b9358>

<matplotlib.text.Text at 0x7f78e419a128>

<matplotlib.text.Text at 0x7f78c1fbb5c0>

<matplotlib.legend.Legend at 0x7f78c0bdd668>



Try again with ω in the range of 0.7 - 0.9

```

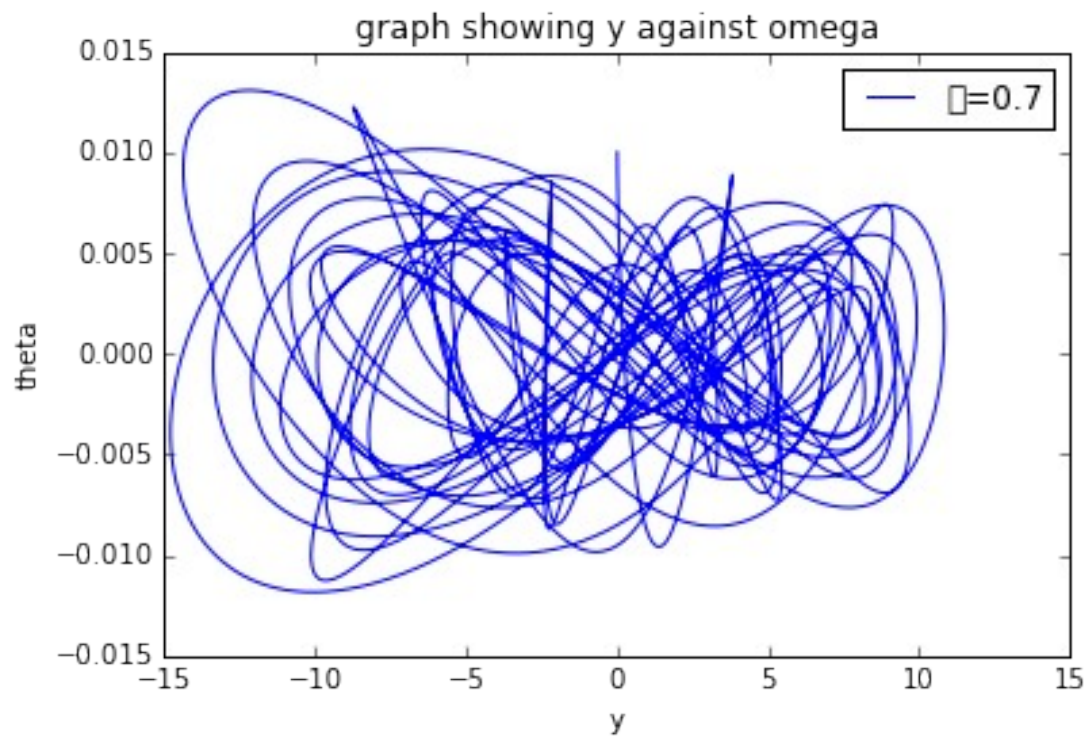
ω_start=0.7
ω_end=0.9
ω=np.arange(ω_start, ω_end, 0.01)

for i in ω:
    times, θ, y = wind_tacoma(dt=0.001, cromer=True, y0=0, θ0=0.01,
z0=0, γ0=0, A=2, ω=i )
    plt.plot( y, θ, label='ω={0}'.format(i))
    plt.xlabel('y')
    plt.ylabel('theta')
    plt.title('graph showing y against omega')
    plt.legend()

plt.show()

[<matplotlib.lines.Line2D at 0x7f78e1cebc18>]
<matplotlib.text.Text at 0x7f78e2fa9c88>
<matplotlib.text.Text at 0x7f78e1d54588>
<matplotlib.text.Text at 0x7f78e58086a0>
<matplotlib.legend.Legend at 0x7f78c0bcaa90>

```



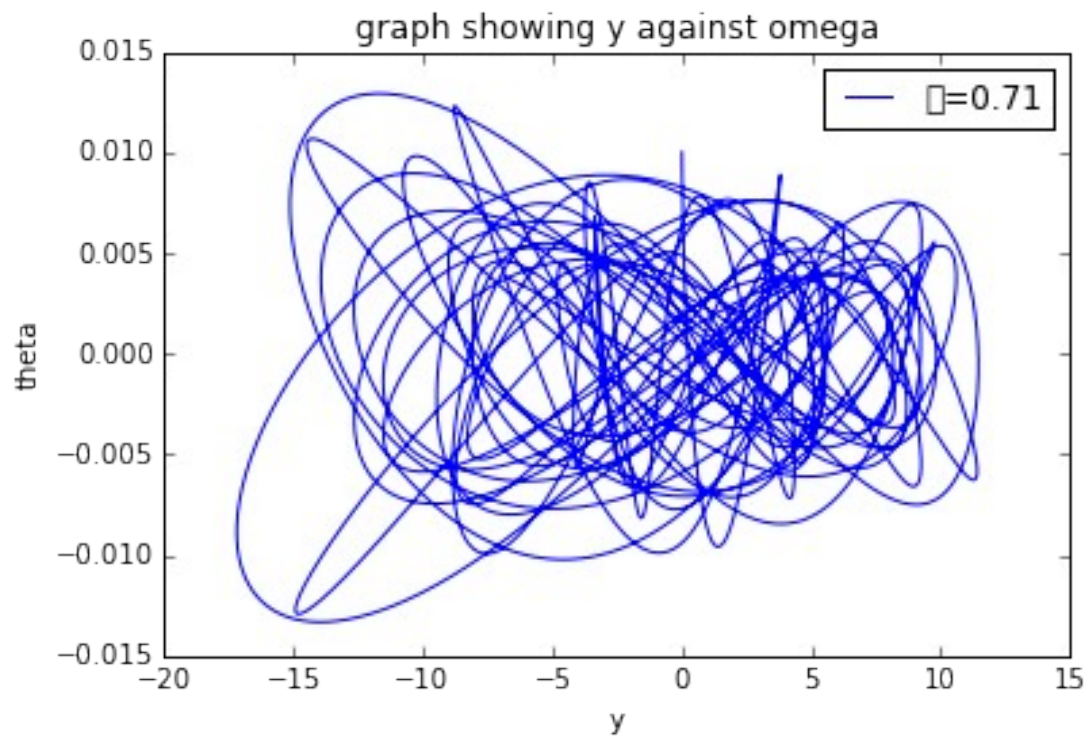
[<matplotlib.lines.Line2D at 0x7f78ele39cc0>]

<matplotlib.text.Text at 0x7f78elcfaef0>

<matplotlib.text.Text at 0x7f78d0ba7518>

<matplotlib.text.Text at 0x7f78d0b8c048>

<matplotlib.legend.Legend at 0x7f78d13217f0>



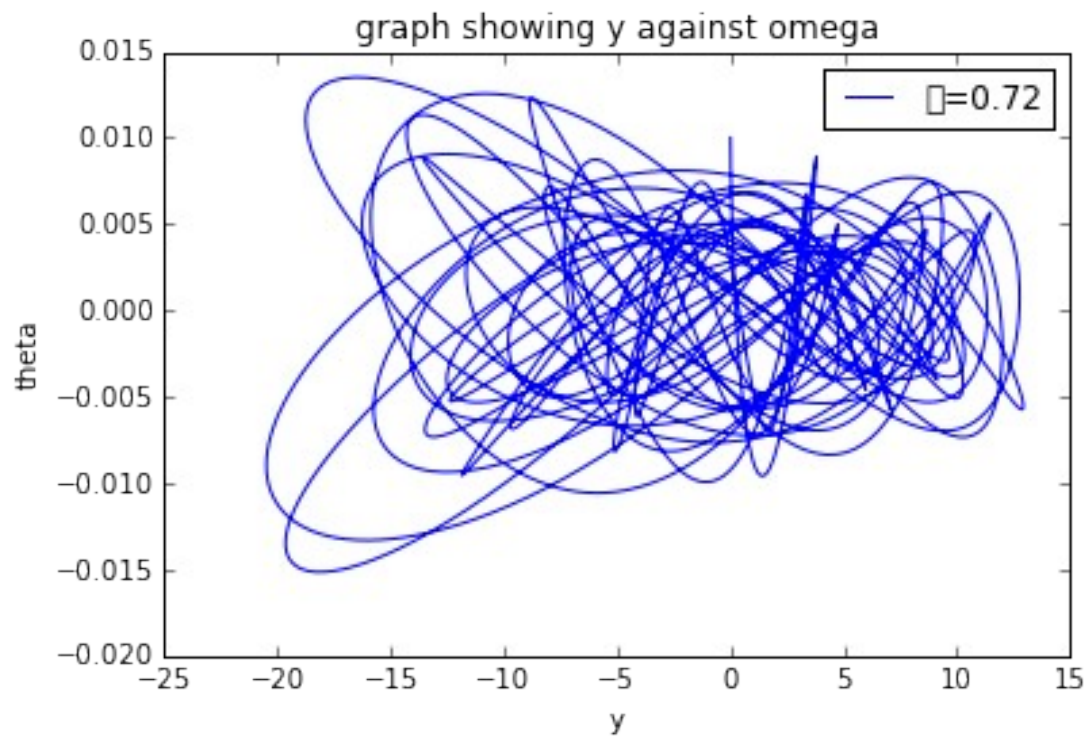
[<matplotlib.lines.Line2D at 0x7f78d30f1f28>]

<matplotlib.text.Text at 0x7f7887fdc080>

<matplotlib.text.Text at 0x7f78e418d1d0>

<matplotlib.text.Text at 0x7f78d1319eb8>

<matplotlib.legend.Legend at 0x7f78c0e62c18>



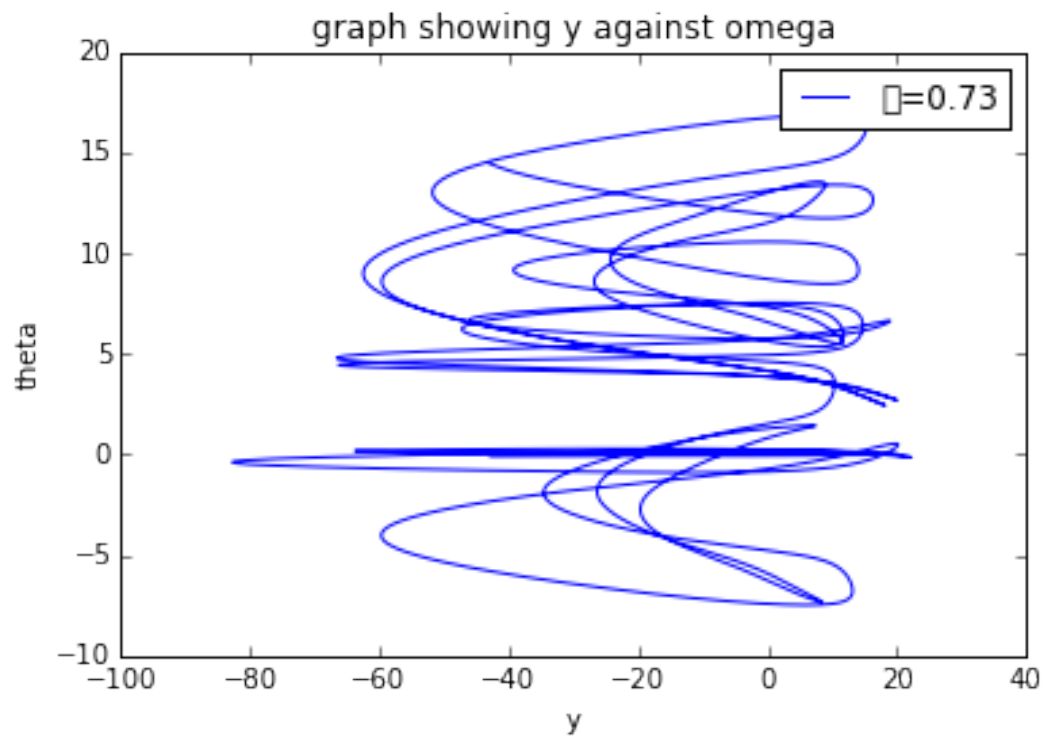
[<matplotlib.lines.Line2D at 0x7f78c0bdf208>]

<matplotlib.text.Text at 0x7f78e418ecf8>

<matplotlib.text.Text at 0x7f78d145c6d8>

<matplotlib.text.Text at 0x7f78735862b0>

<matplotlib.legend.Legend at 0x7f78c0bf51d0>



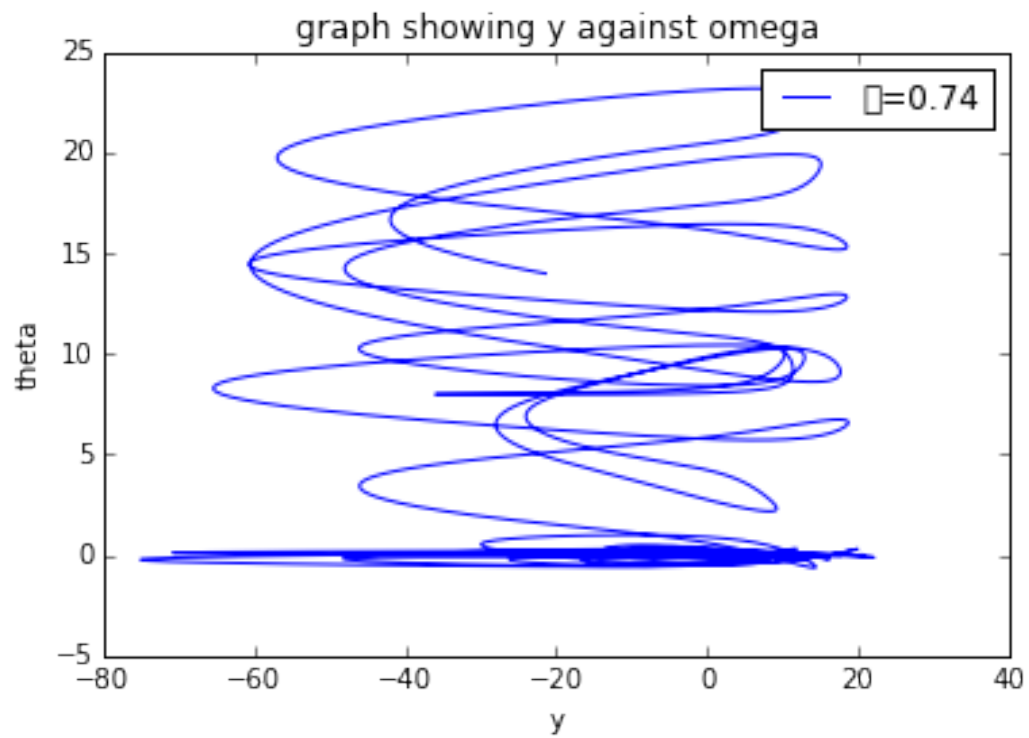
[<matplotlib.lines.Line2D at 0x7f78c2d87cc0>]

<matplotlib.text.Text at 0x7f78c2d80898>

<matplotlib.text.Text at 0x7f78d2ee10f0>

<matplotlib.text.Text at 0x7f7873558c88>

<matplotlib.legend.Legend at 0x7f78e17d7b70>



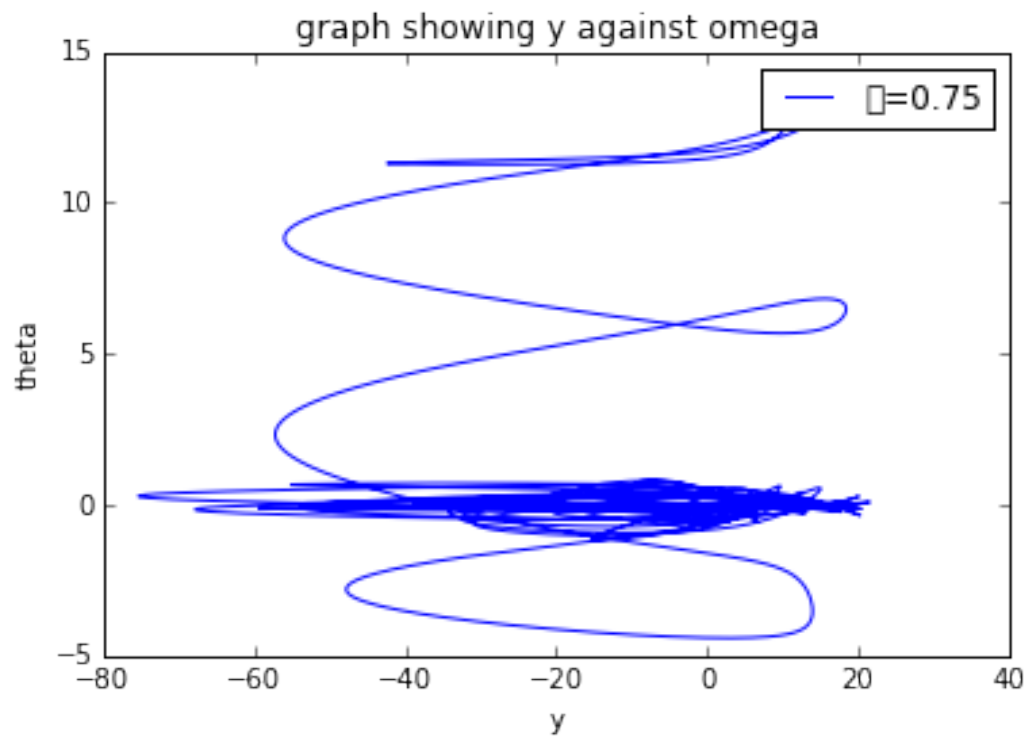
[<matplotlib.lines.Line2D at 0x7f78c20cd6d8>]

<matplotlib.text.Text at 0x7f78c0bfe438>

<matplotlib.text.Text at 0x7f78d2e20a90>

<matplotlib.text.Text at 0x7f78d06e3f28>

<matplotlib.legend.Legend at 0x7f78d0b84748>



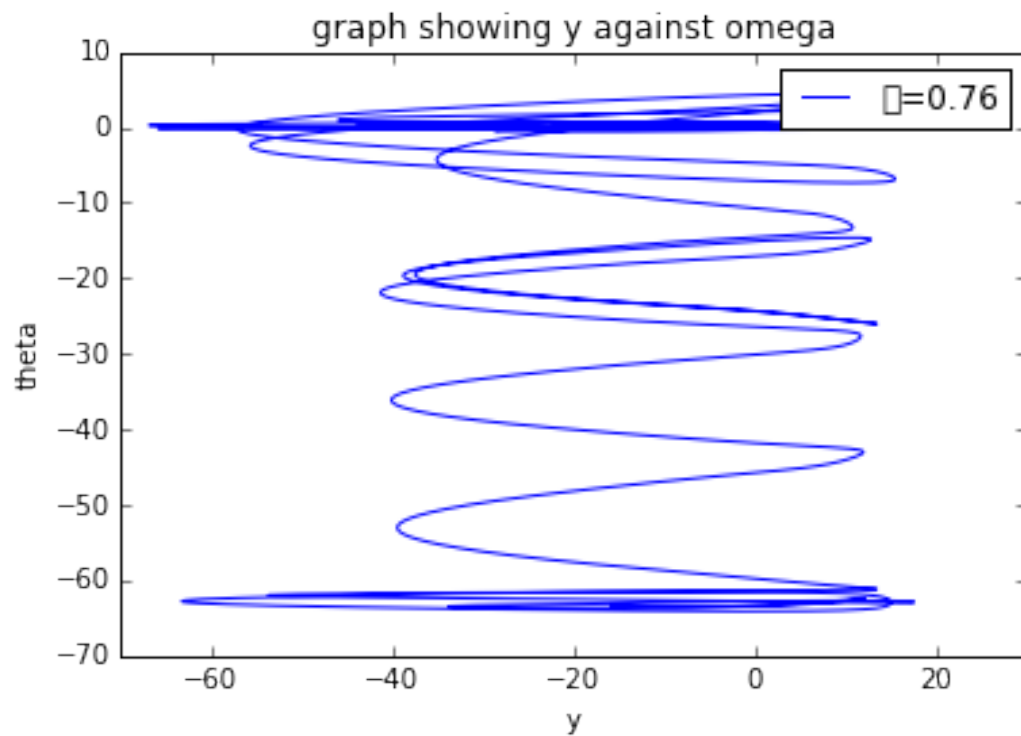
[<matplotlib.lines.Line2D at 0x7f78e3011400>]

<matplotlib.text.Text at 0x7f78d0716eb8>

<matplotlib.text.Text at 0x7f78c1fc42e8>

<matplotlib.text.Text at 0x7f78c1fb8470>

<matplotlib.legend.Legend at 0x7f78e3011cf8>



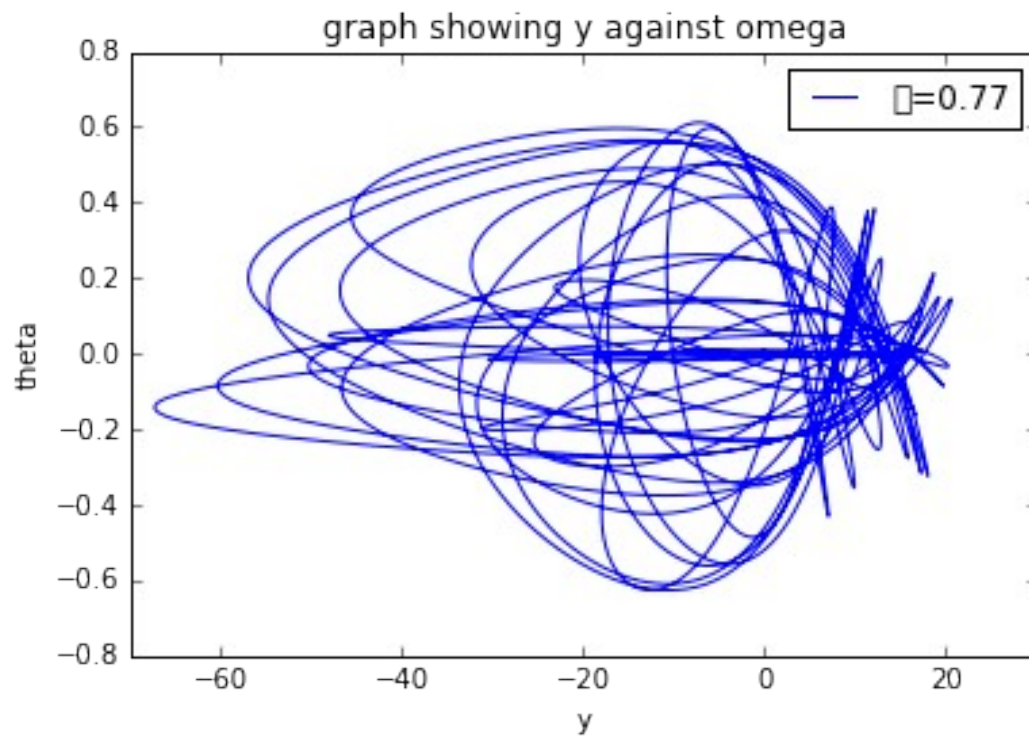
[<matplotlib.lines.Line2D at 0x7f78e4189da0>]

<matplotlib.text.Text at 0x7f78e30034e0>

<matplotlib.text.Text at 0x7f78e02dea20>

<matplotlib.text.Text at 0x7f78d140e550>

<matplotlib.legend.Legend at 0x7f78e4189978>



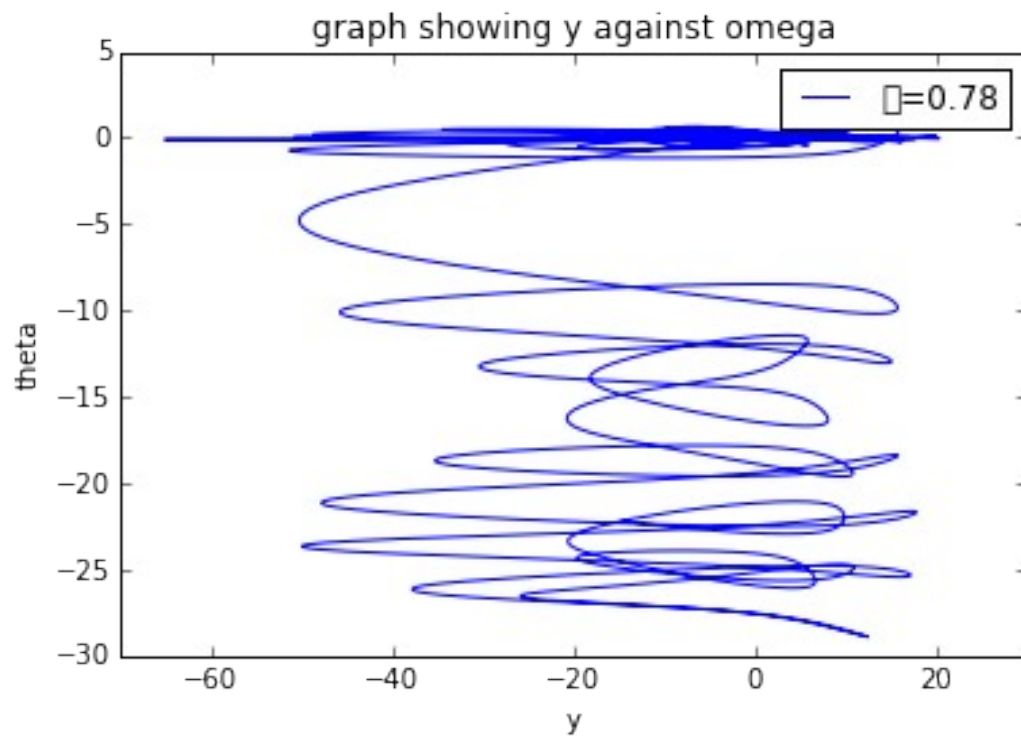
[<matplotlib.lines.Line2D at 0x7f78e17d7ba8>]

<matplotlib.text.Text at 0x7f78e580a438>

<matplotlib.text.Text at 0x7f78e57a8588>

<matplotlib.text.Text at 0x7f78c20b2fd0>

<matplotlib.legend.Legend at 0x7f7873558ac8>



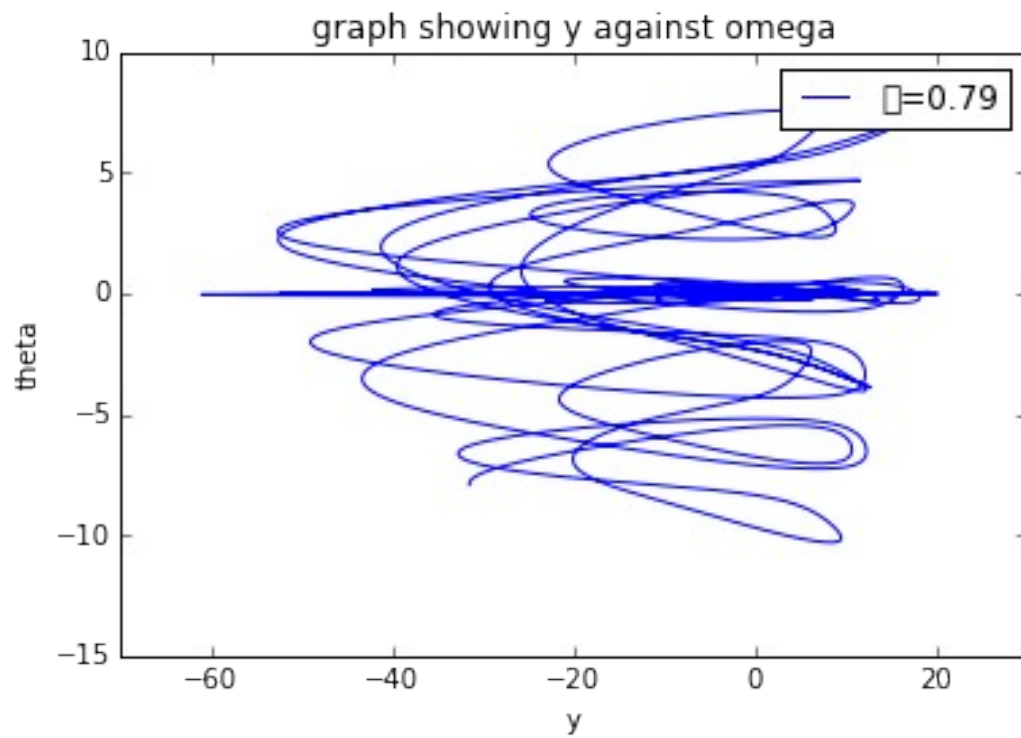
[<matplotlib.lines.Line2D at 0x7f7872e0dc50>]

<matplotlib.text.Text at 0x7f78b004ffd0>

<matplotlib.text.Text at 0x7f78e57a8da0>

<matplotlib.text.Text at 0x7f78e17bdba8>

<matplotlib.legend.Legend at 0x7f7872e0da20>



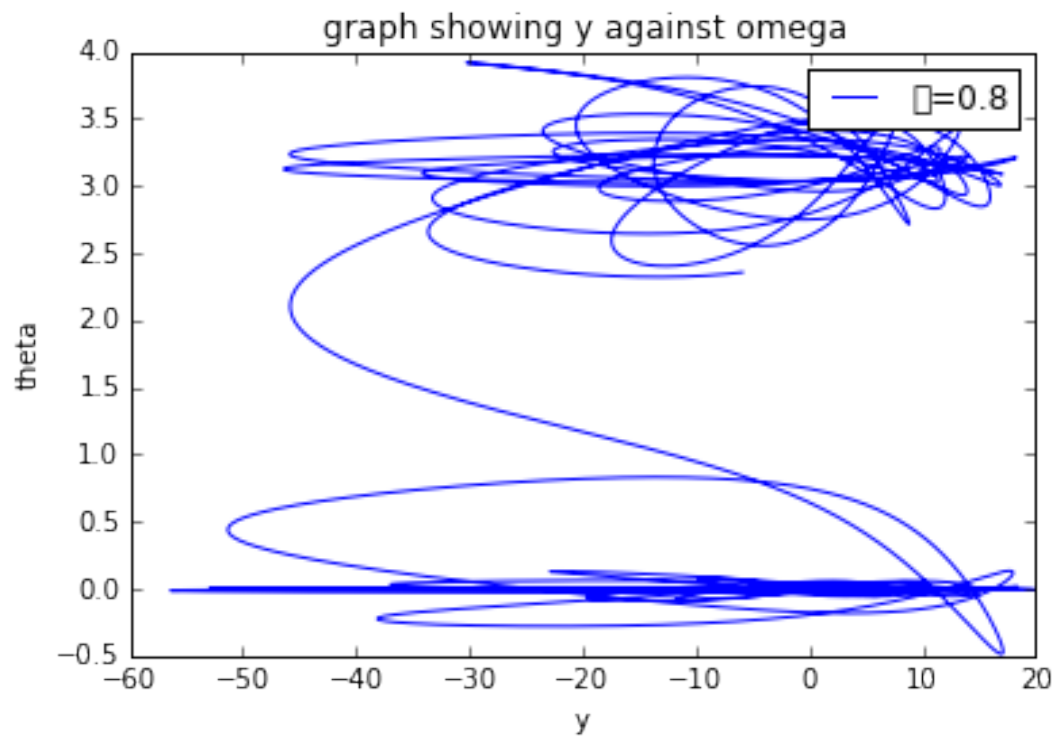
[<matplotlib.lines.Line2D at 0x7f78c192ea90>]

<matplotlib.text.Text at 0x7f7887fe4160>

<matplotlib.text.Text at 0x7f78e3061550>

<matplotlib.text.Text at 0x7f78e1f29a90>

<matplotlib.legend.Legend at 0x7f78e418e1d0>



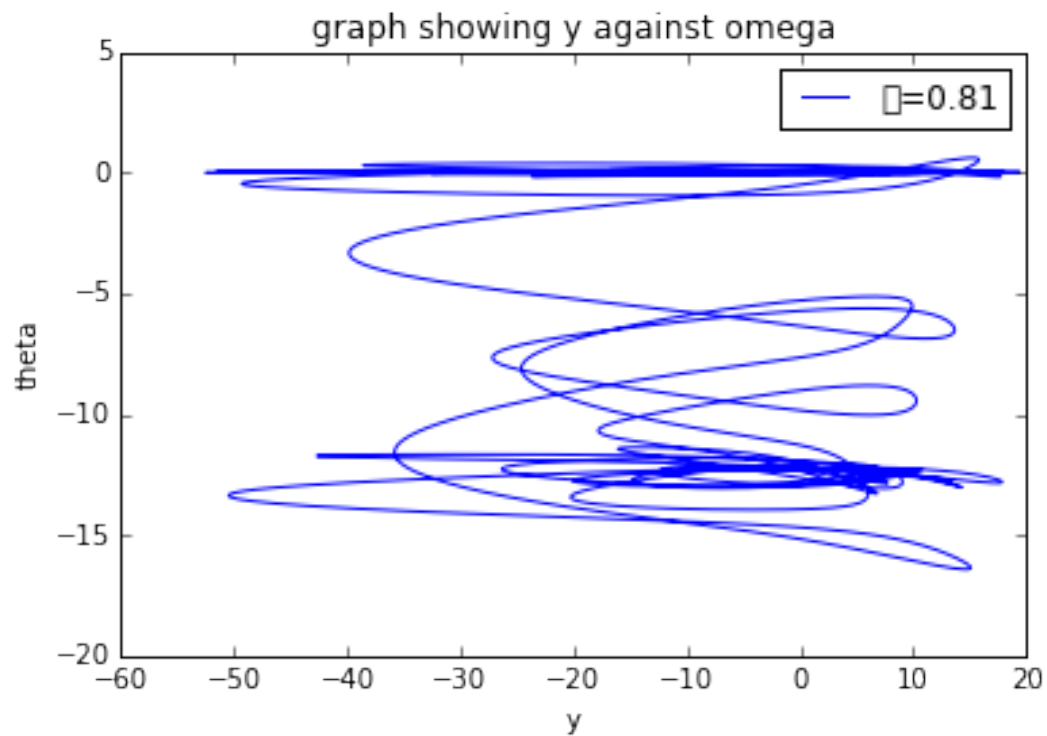
[<matplotlib.lines.Line2D at 0x7f78c08dd400>]

<matplotlib.text.Text at 0x7f78d0710278>

<matplotlib.text.Text at 0x7f7873579080>

<matplotlib.text.Text at 0x7f78c0e06ba8>

<matplotlib.legend.Legend at 0x7f78c08ddc88>



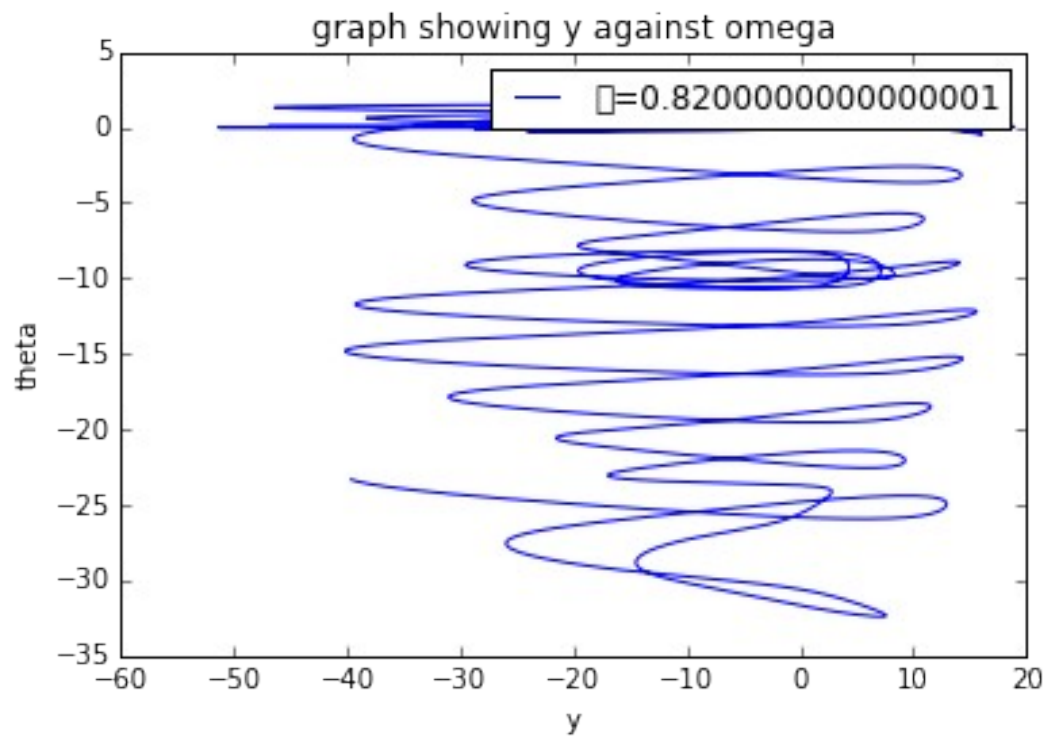
[<matplotlib.lines.Line2D at 0x7f78c0bca710>]

<matplotlib.text.Text at 0x7f78e3068898>

<matplotlib.text.Text at 0x7f78d30ea6a0>

<matplotlib.text.Text at 0x7f78d12f9ef0>

<matplotlib.legend.Legend at 0x7f78e5808080>



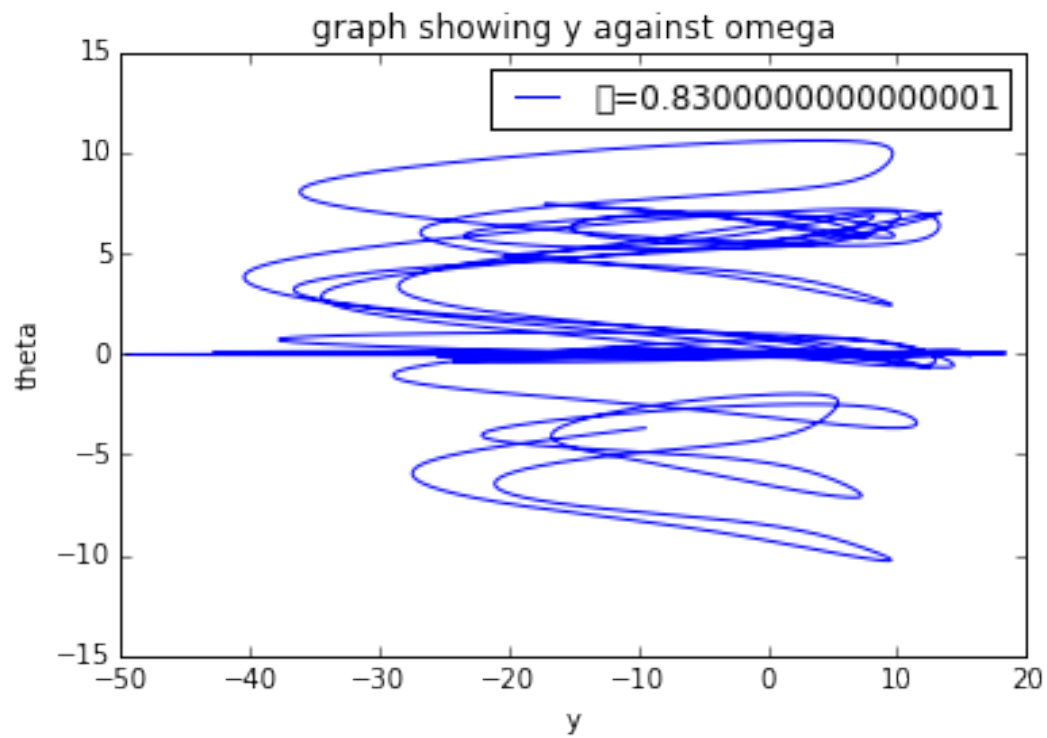
[<matplotlib.lines.Line2D at 0x7f78e2feaef0>]

<matplotlib.text.Text at 0x7f78d2ebb358>

<matplotlib.text.Text at 0x7f7872dfcfd0>

<matplotlib.text.Text at 0x7f787357dda0>

<matplotlib.legend.Legend at 0x7f78e2fea9b0>



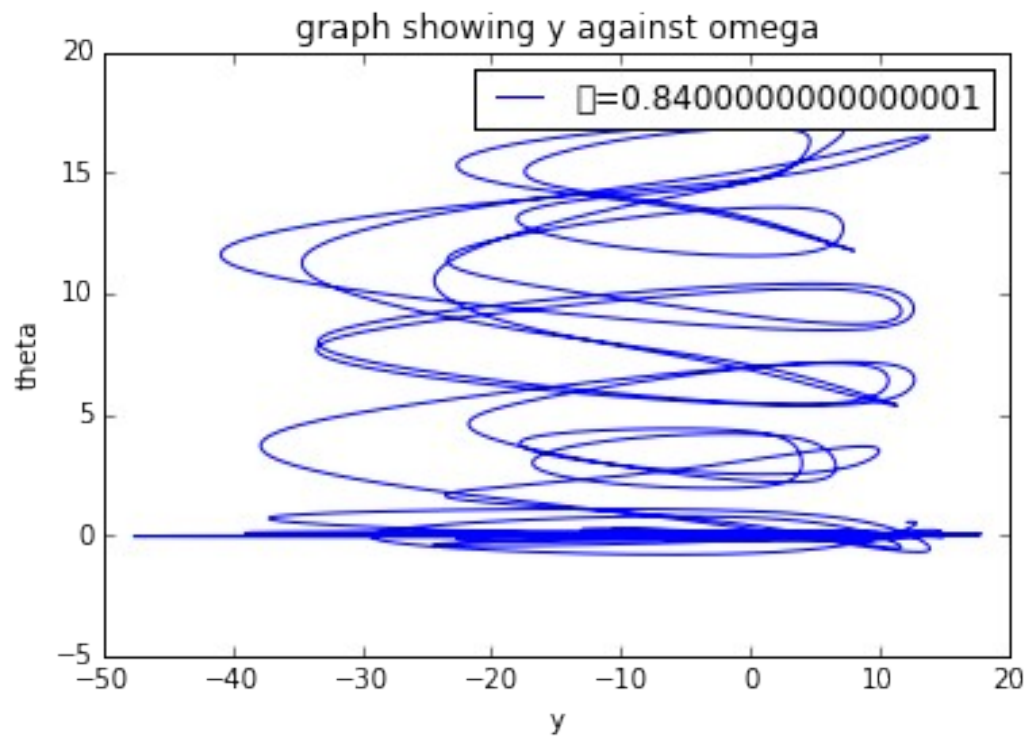
[<matplotlib.lines.Line2D at 0x7f78e1c355f8>]

<matplotlib.text.Text at 0x7f78e1c679e8>

<matplotlib.text.Text at 0x7f78e2ffe240>

<matplotlib.text.Text at 0x7f78d05f2f60>

<matplotlib.legend.Legend at 0x7f78e1c19278>



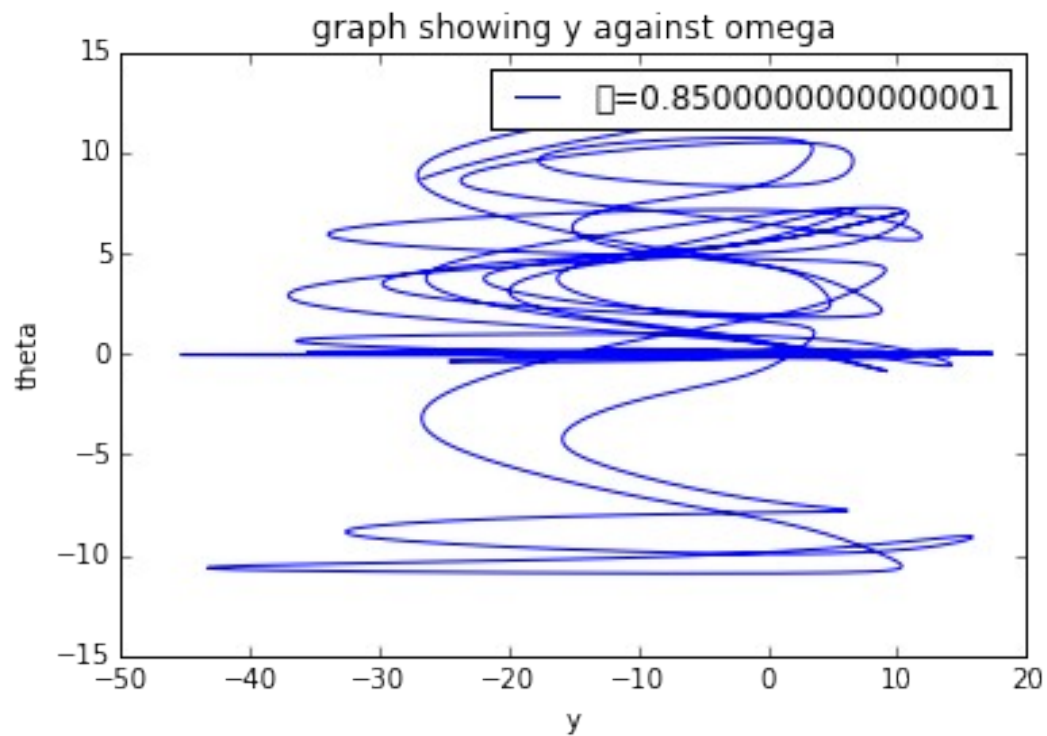
[<matplotlib.lines.Line2D at 0x7f78e1c63828>]

<matplotlib.text.Text at 0x7f78e1cfa630>

<matplotlib.text.Text at 0x7f78c2d9aac8>

<matplotlib.text.Text at 0x7f78e1eb1710>

<matplotlib.legend.Legend at 0x7f78e300a160>



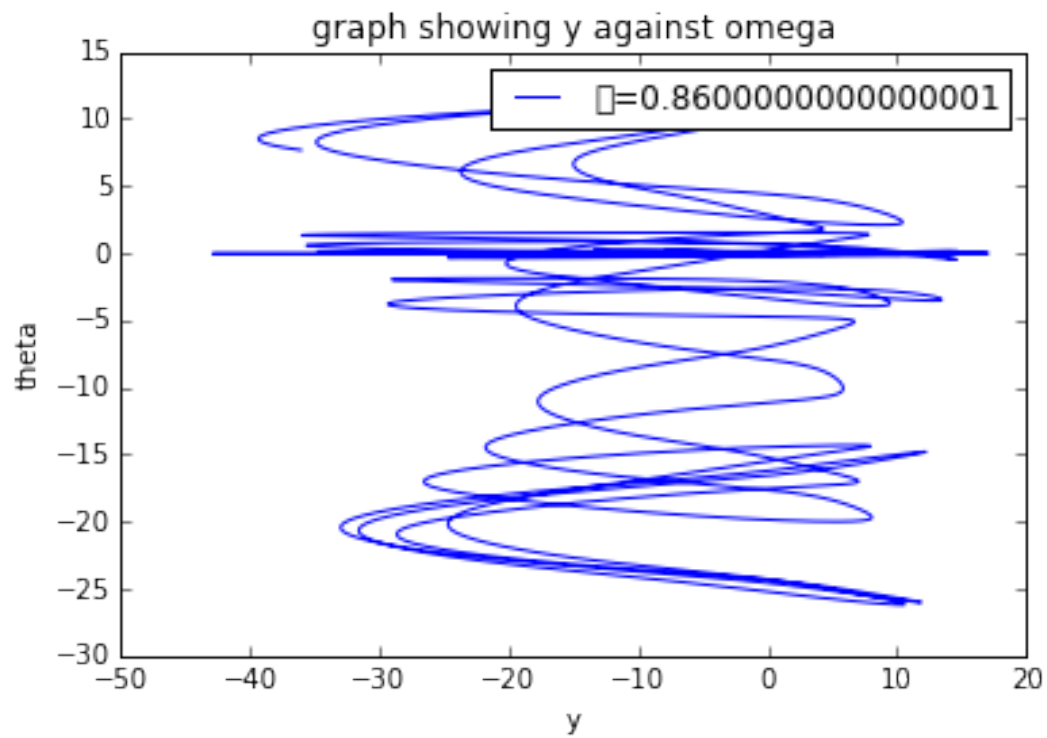
[<matplotlib.lines.Line2D at 0x7f78e1eda860>]

<matplotlib.text.Text at 0x7f78e2ff5f28>

<matplotlib.text.Text at 0x7f78e1c1f588>

<matplotlib.text.Text at 0x7f78d06133c8>

<matplotlib.legend.Legend at 0x7f78e1eda860>



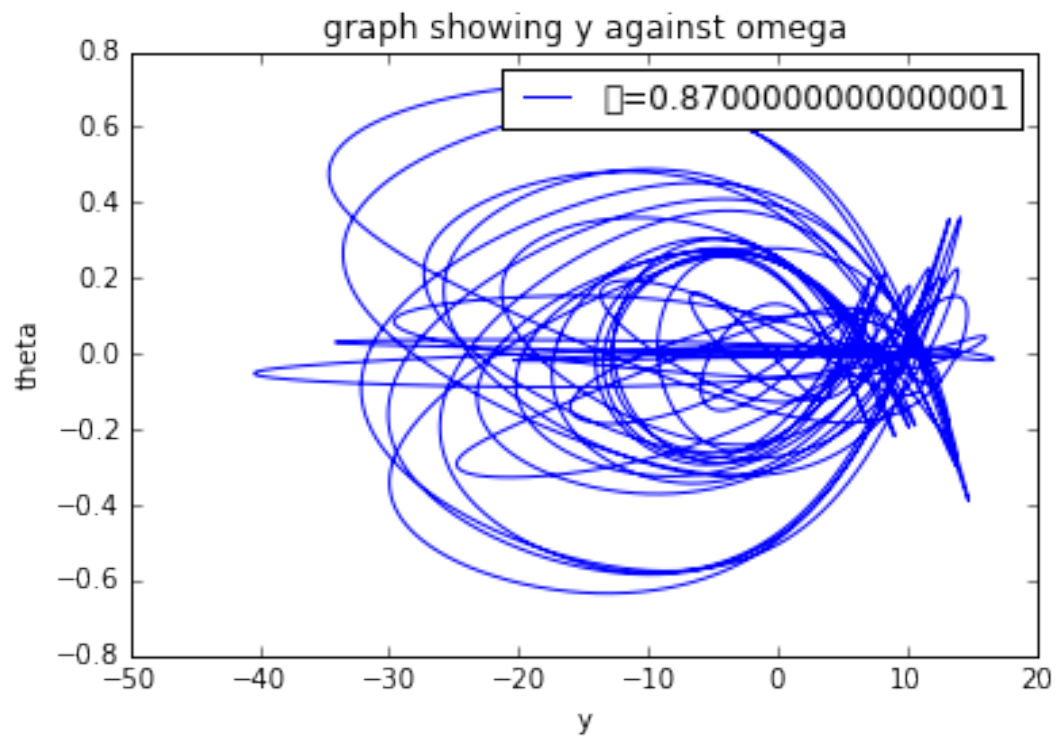
[<matplotlib.lines.Line2D at 0x7f78d2f16fd0>]

<matplotlib.text.Text at 0x7f78d05f69e8>

<matplotlib.text.Text at 0x7f78e2ff3f28>

<matplotlib.text.Text at 0x7f78d376c4a8>

<matplotlib.legend.Legend at 0x7f78d2ef9940>



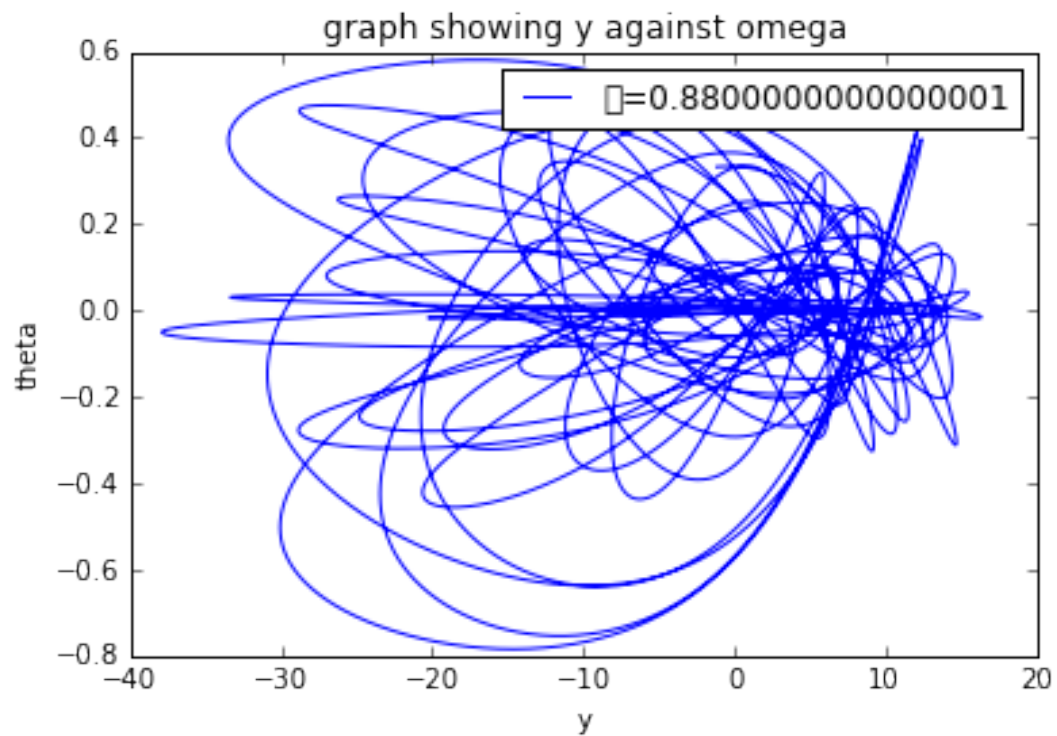
[<matplotlib.lines.Line2D at 0x7f78d375ad68>]

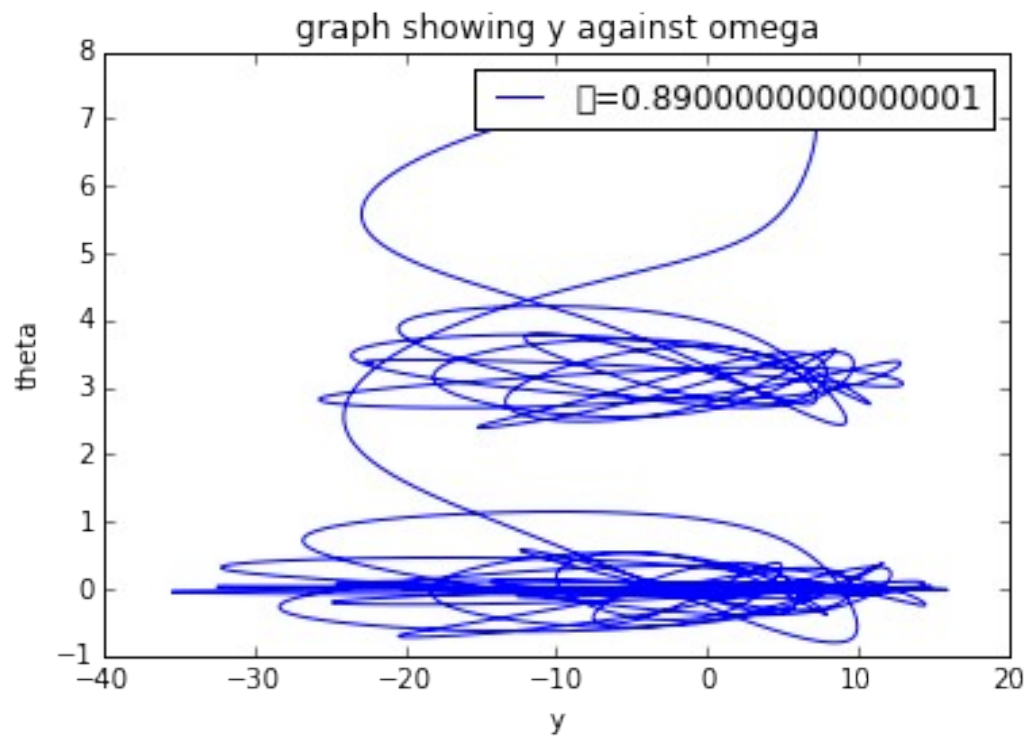
<matplotlib.text.Text at 0x7f78d2efba90>

<matplotlib.text.Text at 0x7f78b0567a20>

<matplotlib.text.Text at 0x7f78d3742128>

<matplotlib.legend.Legend at 0x7f78d376ab38>





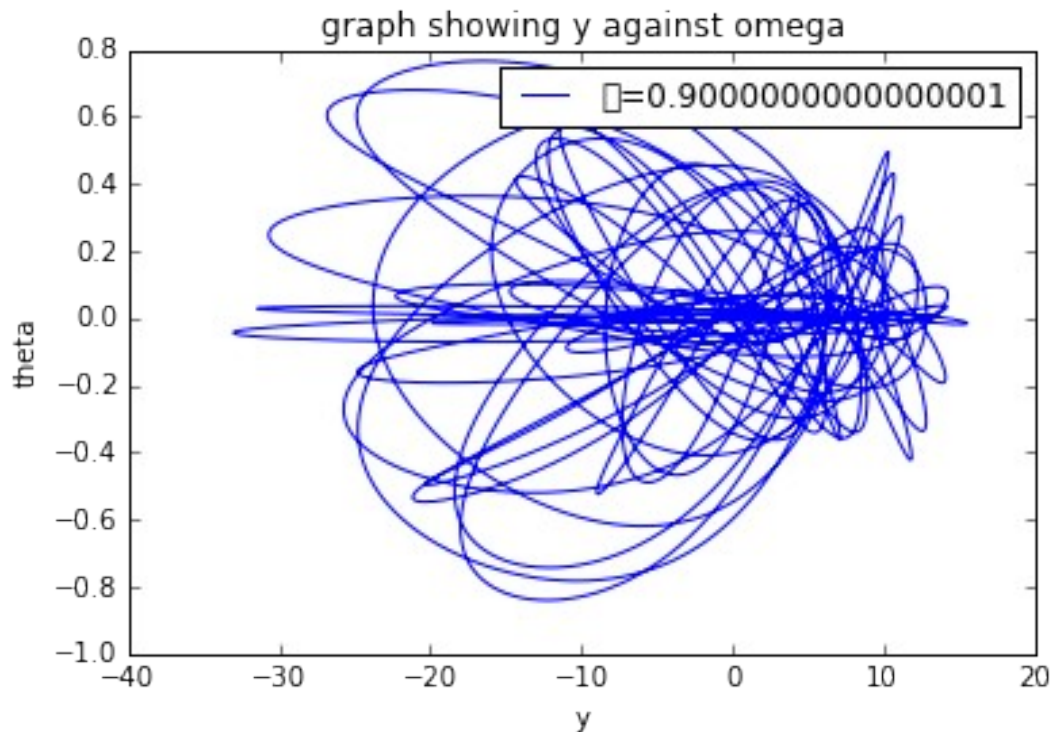
[<matplotlib.lines.Line2D at 0x7f78e1c70dd8>]

<matplotlib.text.Text at 0x7f78e1eab4e0>

<matplotlib.text.Text at 0x7f78b053b860>

<matplotlib.text.Text at 0x7f78e301a5f8>

<matplotlib.legend.Legend at 0x7f78c2d92e10>



```
import numpy as np

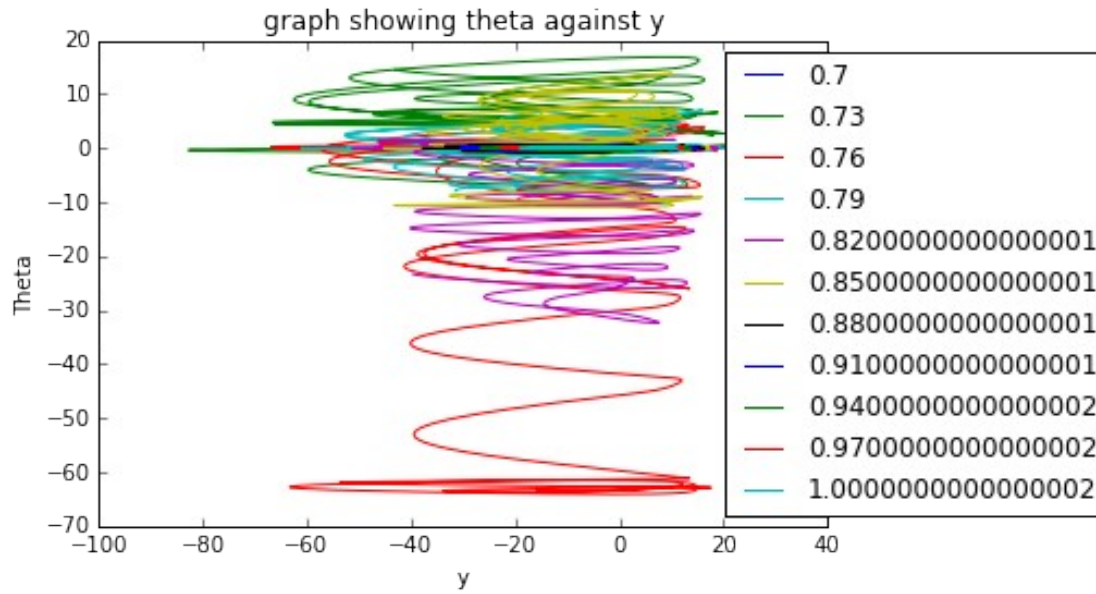
fig, ax = plt.subplots()

ω_start=0.7
ω_end=1
ω=np.arange(ω_start, ω_end, 0.03)

for i in ω:
    times, θ, y = wind_tacoma(dt=0.001, cromer=True, y0=0, θ0=0.01,
z0=0, γ0=0, A=2, ω=i )
    ax.plot( y,θ, label='{0}'.format(i))
    ax.set_xlabel('y')
    ax.set_ylabel('Theta')
    ax.set_title('graph showing theta against y')

plt.rcParams['figure.figsize']=[10,10]

ax.legend(loc='right', bbox_to_anchor=(1.4, 0.5))
plt.show();
```



The graphs above clearly highlights that the largest displacement in y occurs when $\omega = 0.73$.

Using the following equation:

$$\omega = 2\pi v$$

the natural frequency of the bridge can be established. The natural frequency of the bridge which is achieved from this investigation is 0.12Hz.

Other occurrences of similar behaviour

Millennium bridge

On opening day, the millennium bridge showed similar occurrences to the Tacoma bridge but for slightly different reasons. The bridge opened on June 10th 2000 and straight away, thousands of people were crossing it [Explaining why the millennium bridge wobbled- Cornell University 2005]. At first there was no signs of motion in the bridge, but then slowly, the bridge started to sway and oscillate which gradually intensified. The pedestrians on the bridge began to walk in unison in order to keep their balance and combat the swaying of the bridge. This accidental synchronisation of the steps of the pedestrians was what caused this intensifying of the swaying. The difference between the Millennium bridge incident and Tacoma bridge is that the cause of the collapse of Tacoma bridge was due to the vertical oscillations of the bridge while the Millennium bridge swayed in the lateral plane.

Broughton Suspension bridge

On the 12th of April 1831, the Broughton Suspension bridge collapsed due to mechanical resonance induced by soldiers marching across [Tietz, 2020]. The synchronisation of the marching troops induced a vertical force causing one of the iron columns supporting the

suspension bridge to fall. After this incident, it was ordered by the British Army to 'break step' when soldiers were marching across a bridge.

Problem 2: The Travelling Salesperson

The Travelling Salesperson problem aims to optimise the path a person is meant to take in order to complete it in the shortest distance possible. Finding this by hand is challenging as there are $n!$ number of combination for 'n' number of cities were $n!$ can be calculated using the equation:

$$n! = n \times (n-1) \times (n-2) \times \dots (n-(n-1))$$

Then each path must be drawn out and compared to one another to find which combination creates the shortest path length.

Therefore, for just 8 cities there are 40320 different path combinations and for 30 cities, there are $2.652528598 \times 10^{32}$ different paths. It is clear from this that it is impossible for the problem to be solved by using such a brute force method. Therefore, a different approach must be implemented.

Simulated annealing

Simulated annealing is a common method for optimization problems. It is used to find a global minima when local minima are present. It models the physical process of heating and then slowly lowering the temperature to decrease defects. The algorithm is essentially 'hill climbing' but instead of picking the best move, it picks a random move based on a probability distribution. If the selected move improves the solution, then it is always accepted, however, a move that does not improve the solution can still be accepted. This is useful as it helps to avoid falling into a local minima. A temperature schedule is used to systematically decrease the temperature as the algorithm of the simulated annealing process runs. Therefore, the new path is accepted/rejected depending on a probability distribution which is proportional to the temperature [Baird]. The probability distribution is given by the equation:

$$P = e^{\frac{-d_{new} - d_{old}}{T}}$$

```
import json
import copy

from IPython.core.interactiveshell import InteractiveShell

InteractiveShell.ast_node_interactivity = "all"

map = mpimg.imread("map.png")

with open('capitals.json', 'r') as capitals_file:
    capitals = json.load(capitals_file)
```

```

capitals_list = list(capitals.items())

capitals_list = [(c[0], tuple(c[1])) for c in capitals_list]

def coords(path):
    _, coords = zip(*path)
    return coords

def show_path(path_, starting_city):

    path=coords(path_)
    x, y = list(zip(*path))

    _, (x0, y0) = starting_city

    plt.imshow(map)
    plt.plot(x0, y0, 'y*', markersize=15)
    plt.plot(x + x[:1], y + y[:1])
    plt.axis("off")
    fig = plt.gcf()
    fig.figure=((10,10))
    plt.show()

```

The functions above were given beforehand.

Temperature Schedule

A temperature schedule is used to systematically decrease the temperature as the algorithm of the simulated annealing process runs. The purpose of this is to prevent the algorithm from converging to a local minima. This is given by the equation:

$$Temp = \alpha^t t_0$$

The function is shown below.

```

def temp_sched(alpha, t0, t):

    return (alpha**t) * t0

```

Random path

The function below generates a random list of 'n' cities from the 'capitals_list' list.

```

def rand_path(n):
    path=random.sample(capitals_list,n)
    return path

path=rand_path(8) #random of path with 8 cities
show_path(path,path[0])

```



Pairwise Exchange

The pairwise function has been defined below. This function is used to shuffle the position of 2 cities in the randomly generated path in order to change the order. This is then used in the annealing algorithm in order to compare certain attributes of different path orders.

```
def pairwise_exchange(path):
    path2=path.copy()
    pos1=np.random.randint(len(path)) #chooses a random city in the
    path2 path
    pos2=np.random.randint(len(path)) #chooses another random city

    while pos1==pos2: #if the same city is chosen for each position,
        choose another city for position 1
        pos1=np.random.randint(len(path))

    path2[pos1],path2[pos2]=path2[pos2],path2[pos1]
    #swap the positions of these 2 cities in the path to create the
    new path

    return path2
```

```
path = rand_path(8)
print(path)
```

```
path2 = pairwise_exchange(path)
print(path2)
```

```
[('Raleigh', (662.0, 328.8)), ('Providence', (735.2, 201.2)), ('Little
Rock', (469.2, 367.2)), ('Atlanta', (585.6, 376.8)), ('Denver',
(293.6, 274.0)), ('Oklahoma City', (392.8, 356.4)), ('Tallahassee',
(594.8, 434.8)), ('Hartford', (719.6, 205.2))]
[('Raleigh', (662.0, 328.8)), ('Providence', (735.2, 201.2)),
('Oklahoma City', (392.8, 356.4)), ('Atlanta', (585.6, 376.8)),
('Denver', (293.6, 274.0)), ('Little Rock', (469.2, 367.2)),
('Tallahassee', (594.8, 434.8)), ('Hartford', (719.6, 205.2))]
```

Above shows that the pairwise exchange function was successful in switching two cities around to create a new order for the path.

Path Length

The function below finds the length of the randomly generated path. This is done by using pythagoras. City 1 and City 2 are taken from the list, the change in horizontal component and vertical component are found, constructing a right angled triangle so that the hypotenuse can be calculated. This is the distance between the 2 cities. This process is then repeated for the 2nd and 3rd city in the list and so on until it has run through all the cities in the path, completing the loop.

```
def path_length(path):
    l = 0 #inialise the length
    route = list(path) + [path[0]] #closes the path so it returns to
its original starting city
    pair = zip(route, route[1:])
    for i,j in pair:

        h= (i[1][0])
        v= (i[1][1])
        h2= (j[1][0])
        v2= (j[1][1])

        distance= np.sqrt(((h - h2)**2) + ((v - v2)**2))
        #using pythagoras to find the distance between a pair of
cities
        l = l + distance

    return l

path = rand_path(8)
path_length(path) # shows the path length of the randomly generated
path with 8 cities
```


1909.7902722180791

Simulated Annealing

```
def annealing( $\alpha$ , t0, t, n):

    path = rand_path(n) #generate random path
    print('Distance of original path : ', path_length(path))
    Distances=[]
    Temperatures=[]

    for i in range(t):

        exc_path = pairwise_exchange(path) #the same random path is
now been switched to a different order

        new_d = path_length(exc_path) #length of new path

        old_d = path_length(path) #length of originally generated
random path
        Temp = temp_sched( $\alpha$ , t0,i)

        P = np.exp(-new_d/Temp)*np.exp(old_d/Temp) #probability
distribution

        if P >= random.uniform(0,1): #generates random number between
0 and 1 that determines if new path is accepted

            path = exc_path #if accepted the new path becomes the old
path and the process is repeated
            Distances.append(new_d)
            Temperatures.append(Temp)

    show_path(path,path[0])
    plt.plot(Temperatures,Distances)
    plt.xscale("log")
    plt.yscale("log")
    plt.xlabel("Temperatures")
    plt.ylabel("Distances")
    plt.autoscale(enable=True, axis='both', tight=None)
    plt.show()

    print('shorter path :',Distances[-1]) #prints the new distance
of the shorter path
```

annealing(0.97,1000,200,8)

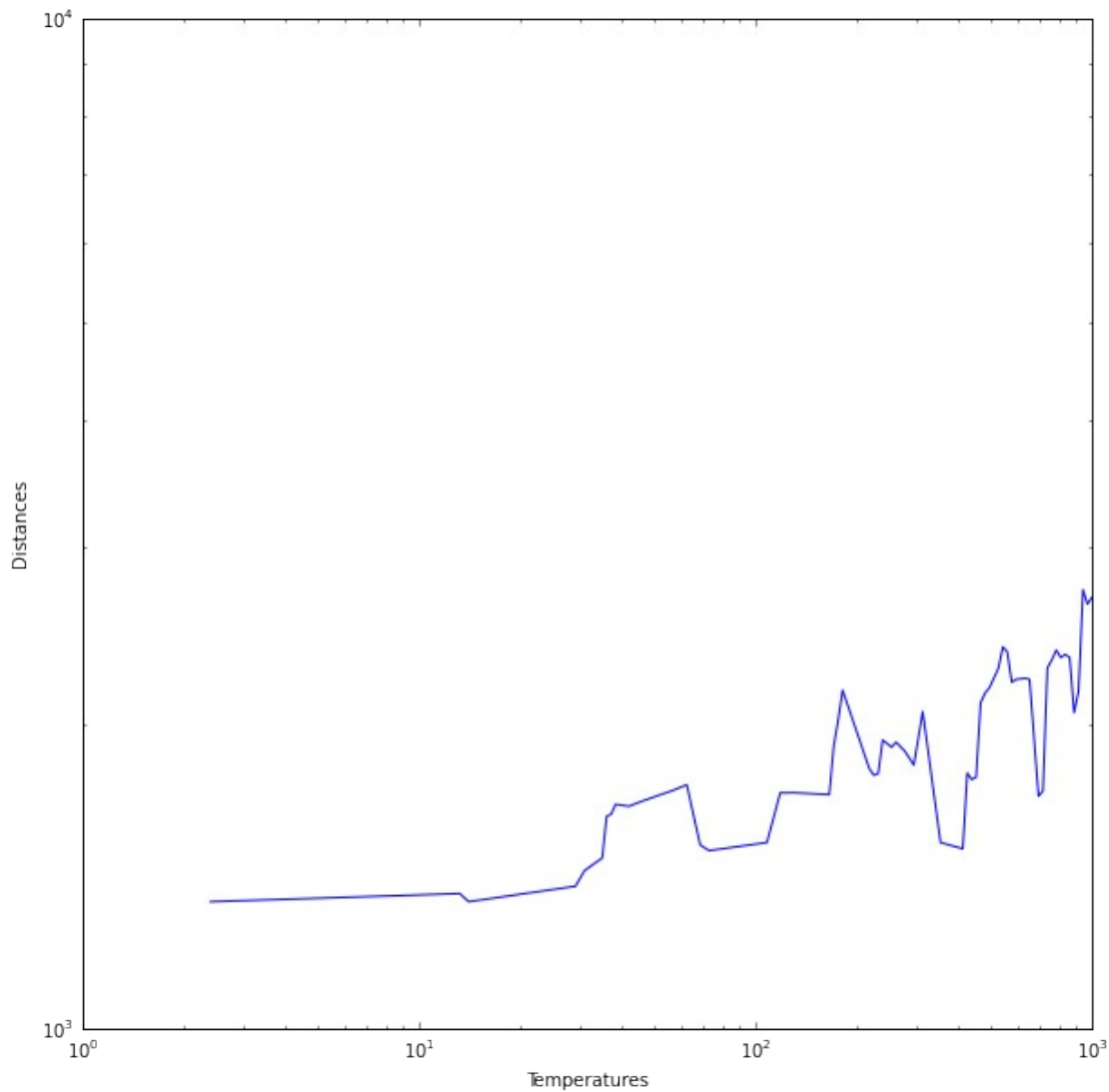
annealing(0.95,1000,200,8)

annealing(0.9,1000,200,8)

annealing(0.8,1000,200,8)

Distance of original path : 2476.1724061



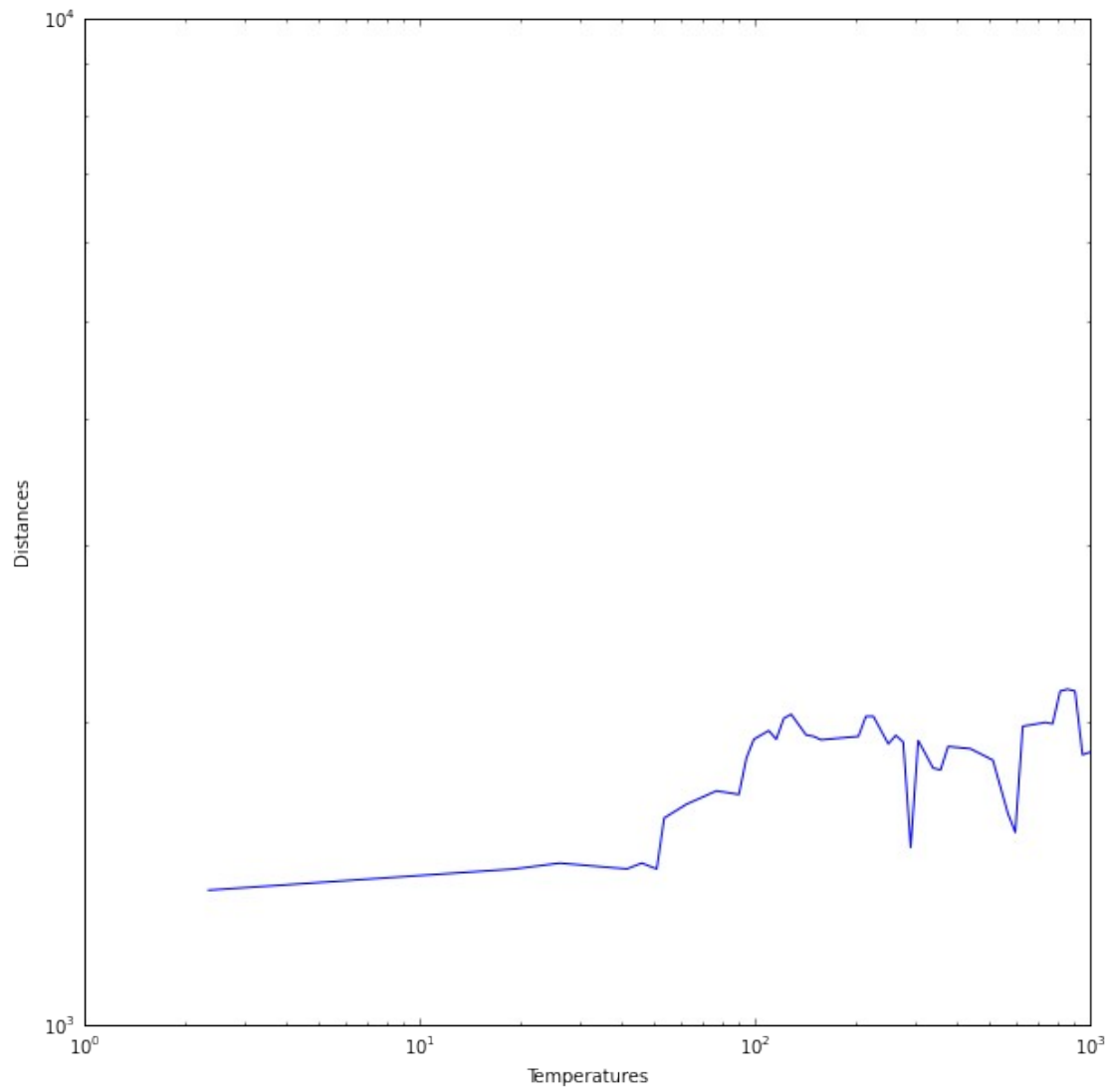


shorter path : 1335.39198114

Distance of original path : 1771.02780113

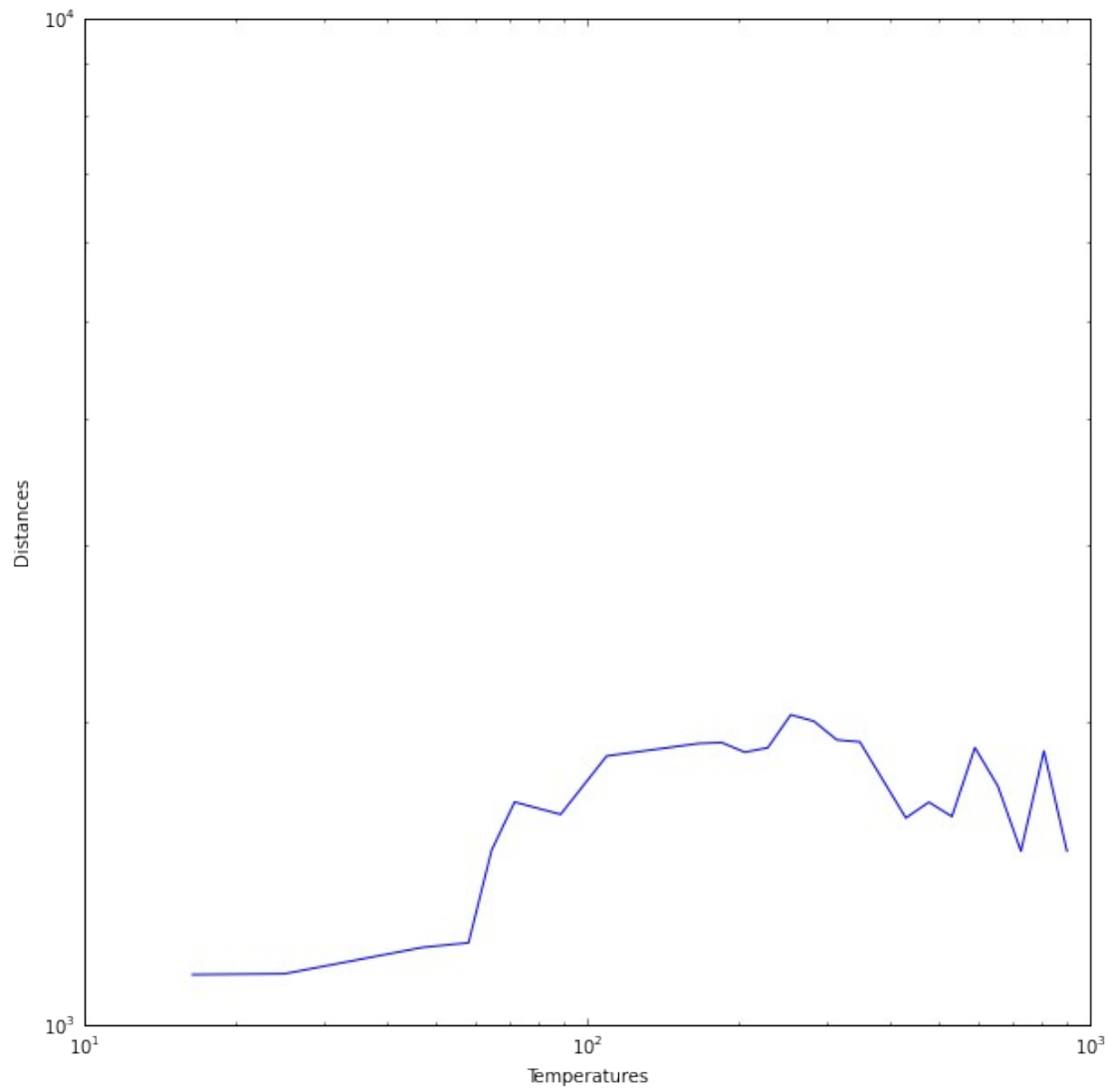
```
/Applications/Pineapple.app/Contents/Frameworks/Python.framework/  
Versions/3.5/lib/python3.5/site-packages/ipykernel/__main__.py:18:  
RuntimeWarning: overflow encountered in exp  
/Applications/Pineapple.app/Contents/Frameworks/Python.framework/Versi  
ons/3.5/lib/python3.5/site-packages/ipykernel/__main__.py:18:  
RuntimeWarning: invalid value encountered in double_scalars
```





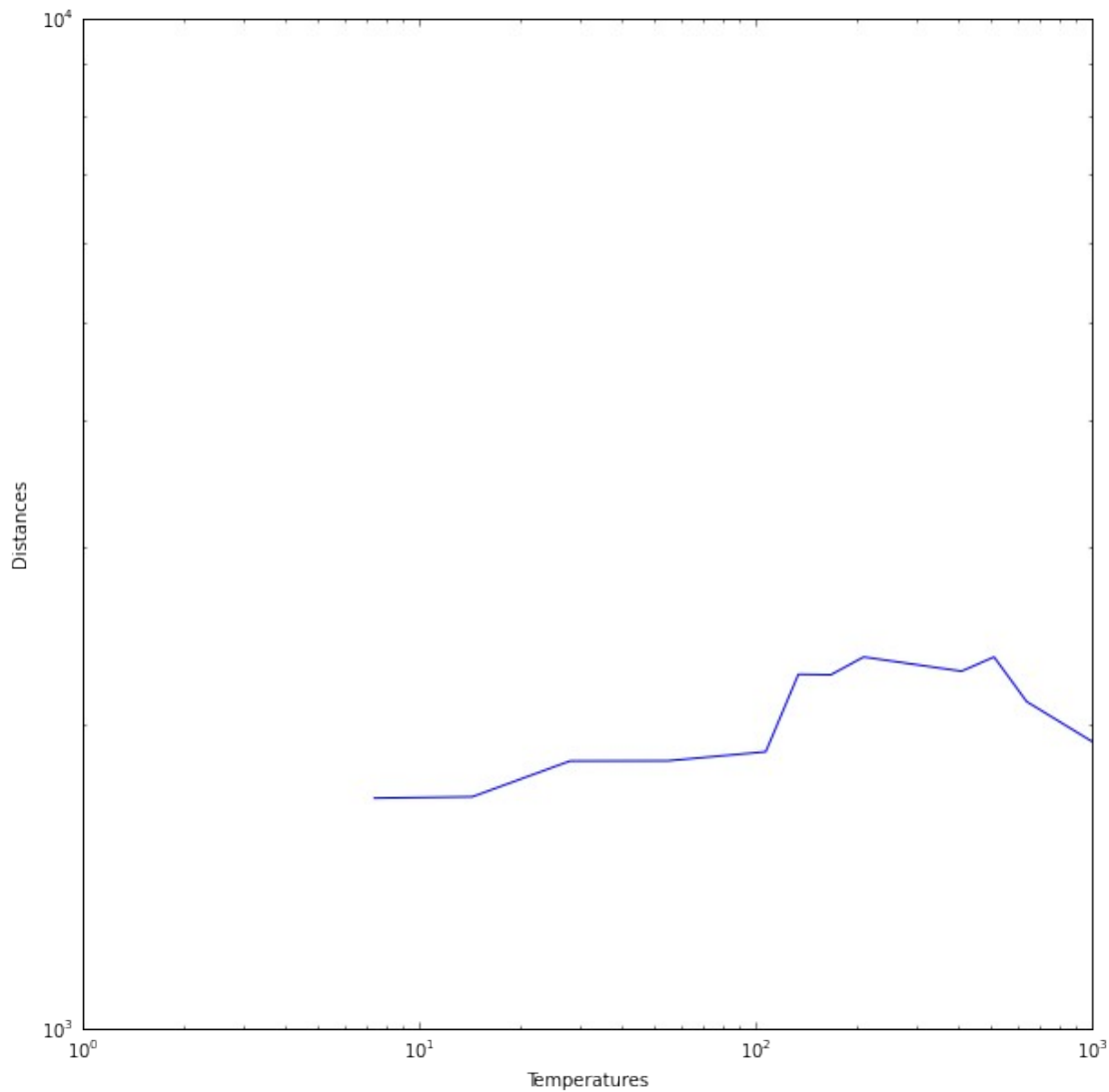
shorter path : 1360.87847146
Distance of original path : 1355.62029213





shorter path : 1121.9122967
Distance of original path : 1950.9284593





shorter path : 1691.03715208

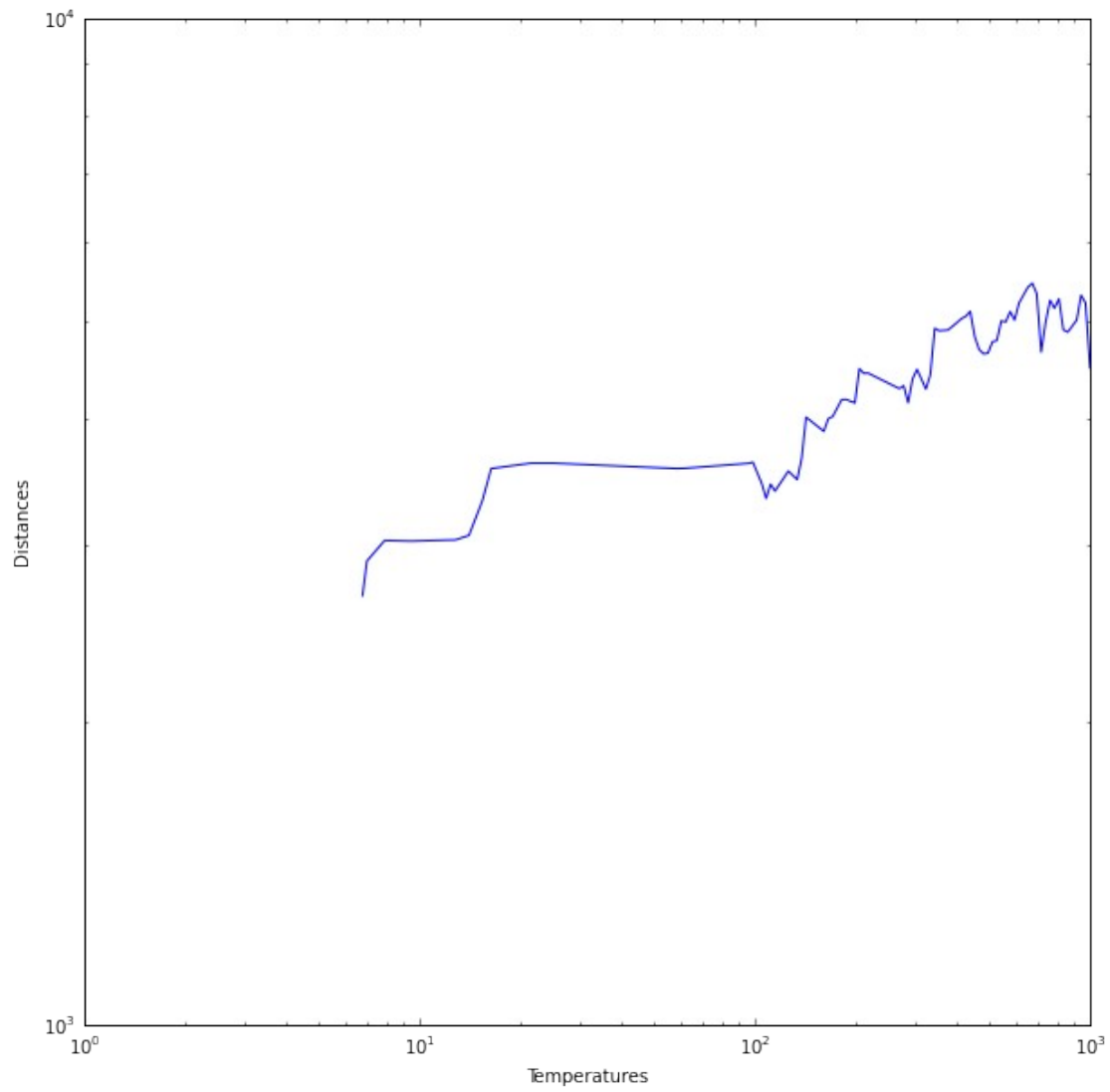
`annealing(0.97,1000,200,20)` *#test using original parameters for 20 cities*

`annealing(0.97,1000,200,len(capitals_list))` *#test using original parameters for all the cities in 'capitals_list'*

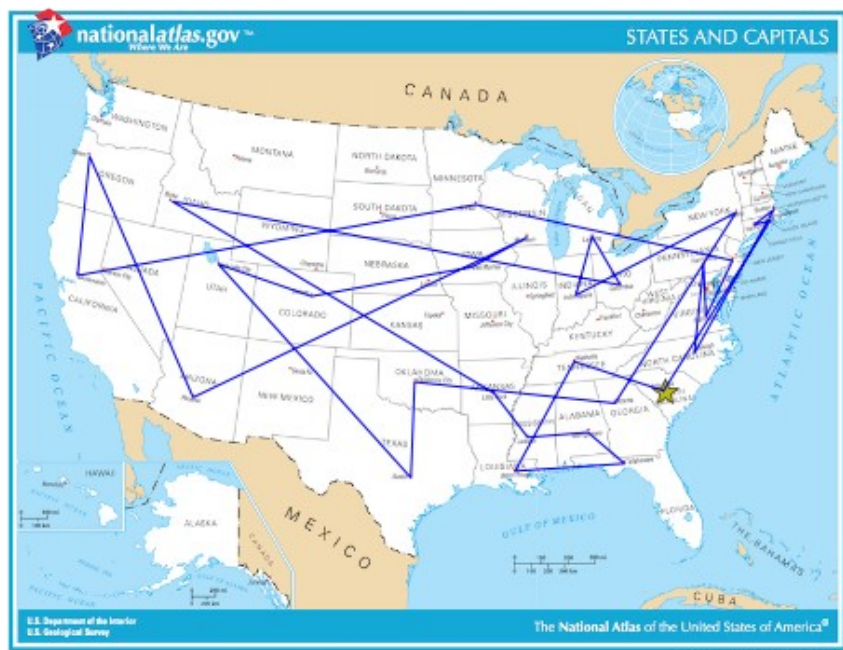
Distance of original path : 4659.25724672

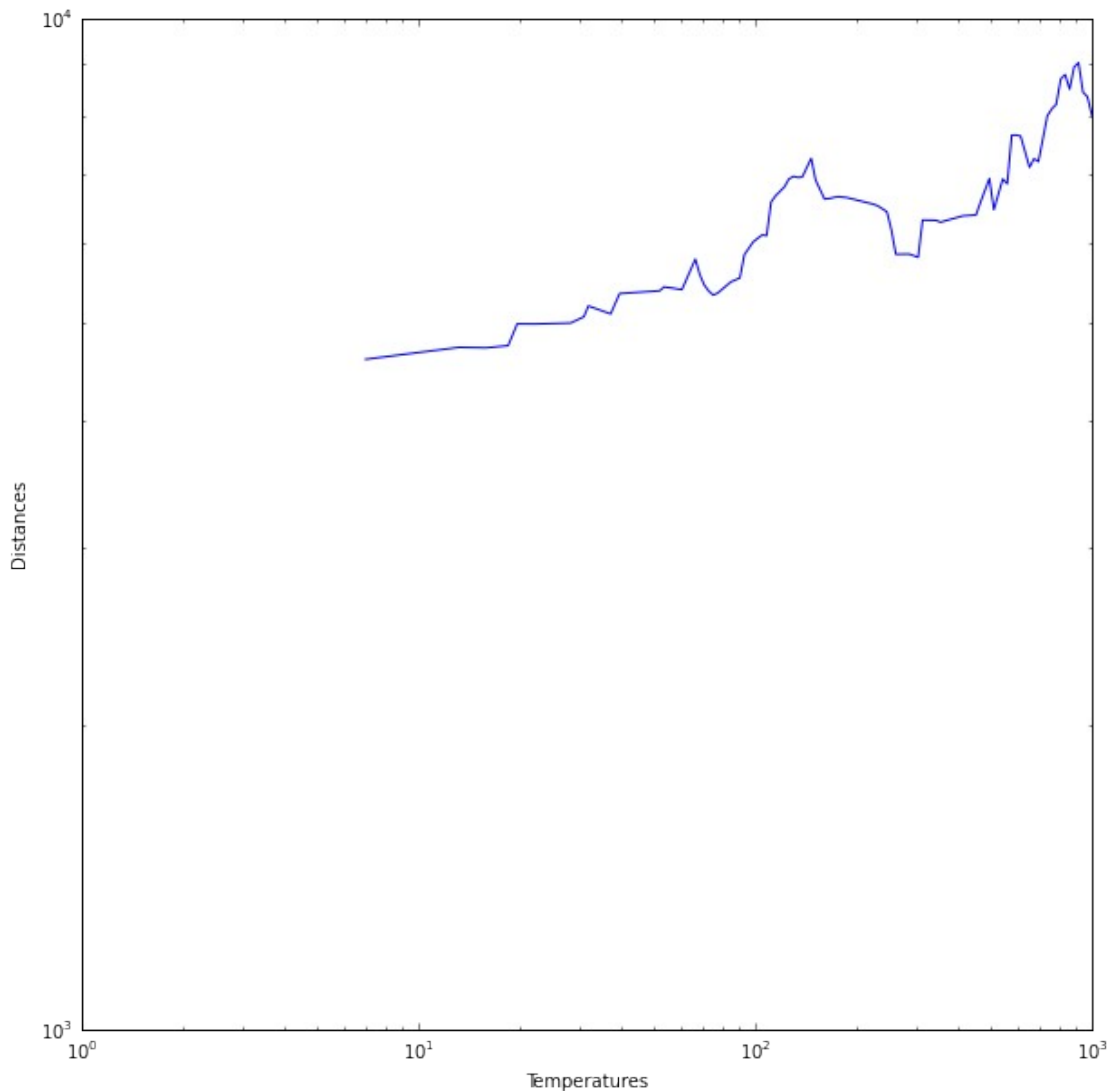
```
/Applications/Pineapple.app/Contents/Frameworks/Python.framework/
Versions/3.5/lib/python3.5/site-packages/ipykernel/__main__.py:18:
RuntimeWarning: overflow encountered in exp
/Applications/Pineapple.app/Contents/Frameworks/Python.framework/Versi
ons/3.5/lib/python3.5/site-packages/ipykernel/__main__.py:18:
RuntimeWarning: invalid value encountered in double_scalars
```





shorter path : 2669.46155236
Distance of original path : 8192.29780223





shorter path : 4600.49421374

The graphs above clearly show that the distance of the path decreases over a period of iterations which means that the annealing function is working.

FINDING THE BEST PARAMETERS

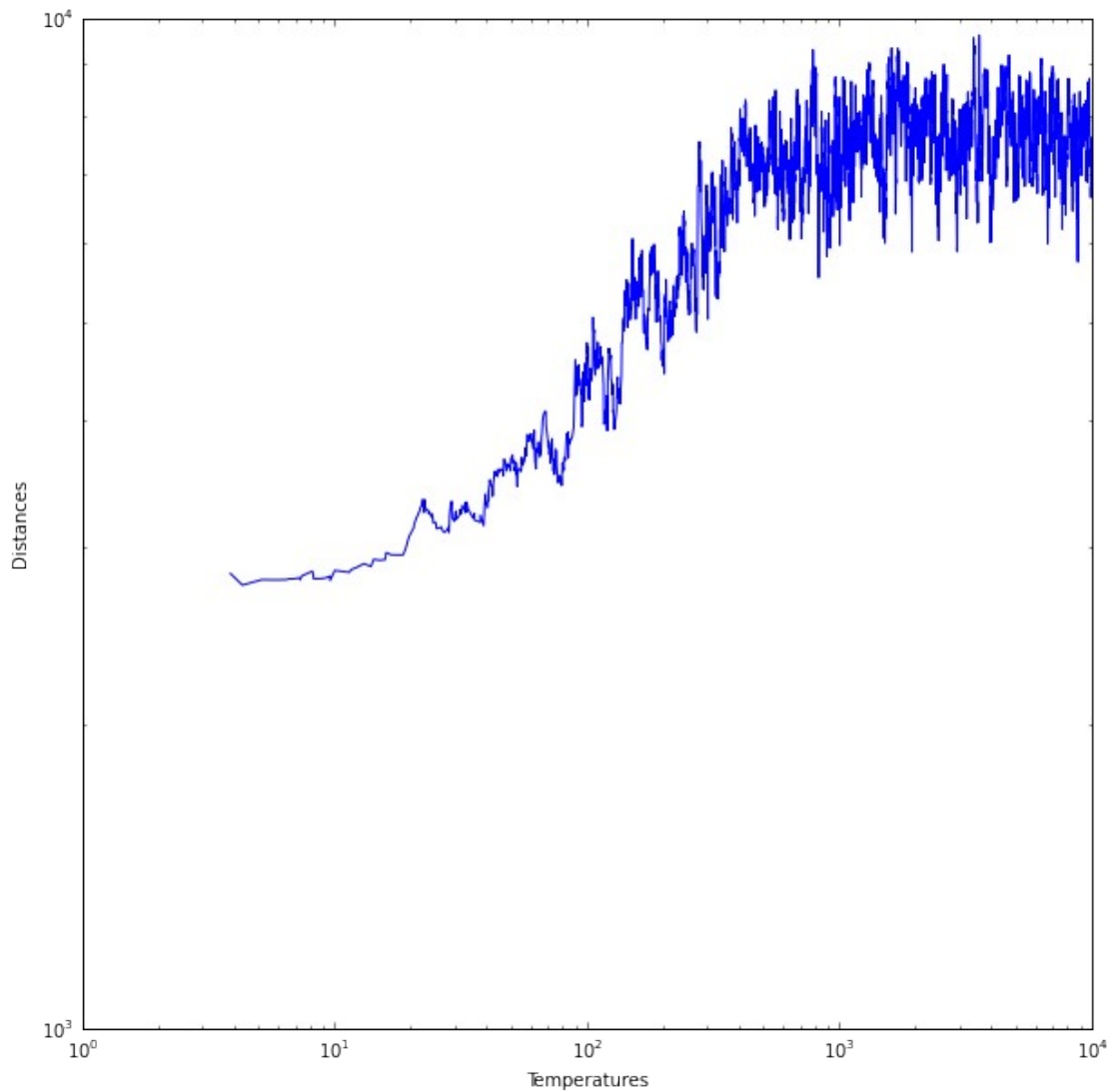
The best parameters were found using trial and error. The individual parameters were modified and many different combinations were attempted in order to come to the set of parameters below. This could also have been possible by using nested for loops to change the parameters within a set range to give the best combination of parameters to achieve the shortest path however, this would have been too computationally expensive.

`annealing(0.999,10000,8000,len(capitals_list))` *#this set of parameters achieved the shortest distance*

Distance of original path : 7134.35277263

```
/Applications/Pineapple.app/Contents/Frameworks/Python.framework/  
Versions/3.5/lib/python3.5/site-packages/ipykernel/__main__.py:18:  
RuntimeWarning: overflow encountered in exp  
/Applications/Pineapple.app/Contents/Frameworks/Python.framework/Versi  
ons/3.5/lib/python3.5/site-packages/ipykernel/__main__.py:18:  
RuntimeWarning: invalid value encountered in double_scalars
```





shorter path : 2821.47050225

Shortest path

Running the annealing function just once does not always give the shortest path possible due to an element of randomness. The function below repeats the annealing function with the most optimal parameters a set number of times. This would produce list of short distances meaning the absolute shortest distance from this list can be selected.

```
def shortest_path( $\alpha$ , t0, t, N):
    path = rand_path(len(capitals_list)) #produce a random path of all
the cities

    Distances=[]
    Temperatures=[]
    paths=[]
    best_path=[]
```

```

for j in range(N): #repeat the for loop below N times
    for i in range(t):
        # same as annealing function
        exc_path = pairwise_exchange(path)

        new_d = path_length(exc_path)

        old_d = path_length(path)
        Temp = temp_sched( $\alpha$ , t0,i)

        P = np.exp(-new_d/Temp)*np.exp(old_d/Temp)

        if P >= random.uniform(0,1):
            path = exc_path
            Distances.append(new_d) #adds the newly improved
distances to the empty list
            Temperatures.append(Temp)
            paths.append(path)

        d_min = min(Distances) #finds the lowest value in the Distances
list
        b = Distances.index(d_min) #finds the position of this distance in
the list
        best_path = paths[b] #finds the correspondig path to the shortest
distance

        show_path(best_path,best_path[0]) #shows the path of the 'shortest
path'
        print('The distance of the shortest path obtained is',d_min) #
prints the distance of the shortest path

shortest_path(0.999,10000,8000,20)

```

```

/Applications/Pineapple.app/Contents/Frameworks/Python.framework/
Versions/3.5/lib/python3.5/site-packages/ipykernel/__main__.py:19:
RuntimeWarning: overflow encountered in exp
/Applications/Pineapple.app/Contents/Frameworks/Python.framework/Versi
ons/3.5/lib/python3.5/site-packages/ipykernel/__main__.py:19:
RuntimeWarning: invalid value encountered in double_scalars

```



The distance of the shortest path obtained is 2280.23619976

The function above repeats the optimal parameters 20 times and selects the shortest distance from this. The number of repeats could be increased to obtain more path lengths to choose from but it becomes too computationally expensive.

Longest path

The longest path can be found by copying the annealing function previously stated and switching the direction of the probability inequality. This ensures that if the new path is longer than the original path, it will always be accepted.

```
def long( $\alpha$ , t0, t, n):
    path = rand_path(n) #generate random path
    print('Distance of original path : ', path_length(path))
    Distances=[]
    Temperatures=[]

    for i in range(t):
        exc_path = pairwise_exchange(path) #the same random path is
now been switched to a different order
```

```

new_d = path_length(exc_path) #length of new path

old_d = path_length(path) #length of originally generated
random path
Temp = temp_sched( $\alpha$ , t0,i)

P = np.exp(-new_d/Temp)*np.exp(old_d/Temp) #probability
distribution

    if P <= random.uniform(0,1): #generates random number between
0 and 1 that determines if new path is accepted

        path = exc_path #if accepted the new path becomes the old
path and the process is repeated
        Distances.append(new_d)
        Temperatures.append(Temp)

show_path(path,path[0])

```

```

    print('longer path :',Distances[-1])    #prints the new distance
of the longer path

```

```

long(0.999,10000,8000,len(capitals_list))

```

```

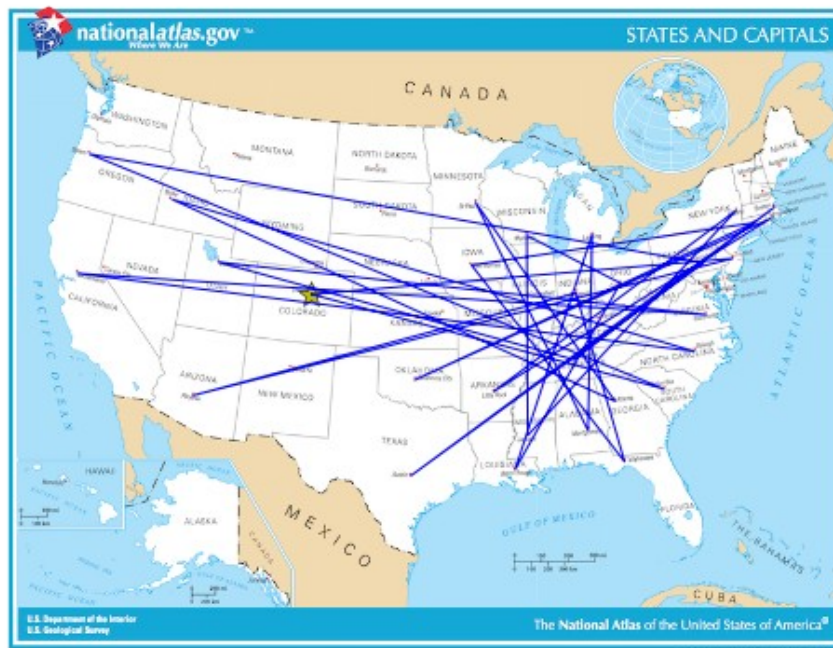
Distance of original path : 7956.22673286

```

```

/Applications/Pineapple.app/Contents/Frameworks/Python.framework/
Versions/3.5/lib/python3.5/site-packages/ipykernel/__main__.py:18:
RuntimeWarning: overflow encountered in exp
/Applications/Pineapple.app/Contents/Frameworks/Python.framework/Versi
ons/3.5/lib/python3.5/site-packages/ipykernel/__main__.py:18:
RuntimeWarning: invalid value encountered in double_scalars

```

longer path : 10635.7451445

Below shows the `longest_path` function. This function repeats the `long` function `N` times and retrieves this longest path and path length from the list of paths and distances created.

```
def longest_path( $\alpha$ , t0, t, N):
    path = rand_path(len(capitals_list)) #produce a random path of all
    the cities
```

```
    Distances=[]
    Temperatures=[]
    paths=[]
    best_path=[]
```

```
    for j in range(N): #repeat the for loop below N times
        for i in range(t):
            # same as annealing function
            exc_path = pairwise_exchange(path)

            new_d = path_length(exc_path)

            old_d = path_length(path)
            Temp = temp_sched( $\alpha$ , t0,i)
```

```

P = np.exp(-new_d/Temp)*np.exp(old_d/Temp)

if P <= random.uniform(0,1):
    path = exc_path
    Distances.append(new_d) #adds the newly improved
distances to the empty list
    Temperatures.append(Temp)
    paths.append(path)

d_max = max(Distances) #finds the highest value in the Distances
list
b = Distances.index(d_max) #finds the position of this distance in
the list
best_path = paths[b] #finds the correspondig path to the shortest
distance

show_path(best_path,best_path[0]) #shows the path of the 'longest
path'
print('The distance of the longest path obtained is',d_max) #
prints the distance of the longest path

longest_path(0.999,10000,8000,20)

```

```

/Applications/Pineapple.app/Contents/Frameworks/Python.framework/
Versions/3.5/lib/python3.5/site-packages/ipykernel/__main__.py:19:
RuntimeWarning: overflow encountered in exp
/Applications/Pineapple.app/Contents/Frameworks/Python.framework/Versi
ons/3.5/lib/python3.5/site-packages/ipykernel/__main__.py:19:
RuntimeWarning: invalid value encountered in double_scalars

```



The distance of the longest path obtained is 10783.0599787


CONCLUSION

In conclusion, the natural frequency obtained from the investigation into the collapse of the Tacoma Bridge was 0.12Hz. The literature value for the natural frequency of the bridge was 0.2Hz [Tacoma Narrows Bridge Failure] which means the percentage difference between the calculated value and the literature value is 40%. This percentage difference is large enough to deem the investigation unsuccessful in determining the natural frequency of the bridge. This is not a surprise due to the simplistic nature of the model. The bridge was modelled in 2D instead of 3D to avoid complications in doing the calculations. Using this model would therefore disregard any forces acting in the horizontal direction which in reality is not the case. Furthermore, using a 2D model also does not properly acknowledge the structure of the bridge as well as the materials used and their material properties. If the investigation is repeated, it would be best to further dive into the causes of oscillations namely, aerodynamic and aeroelastic flutter, in order to give a better, more accurate expression for the driven force and therefore, we would be able to achieve a better estimate for the natural frequency of the bridge.

The investigation into the Travelling salesperson problem was somewhat successful in creating a function that produces the shortest path possible from a set number of cities. The shortest path achieved was 2215.1046. Although the point of the algorithm is to avoid

the use of the brute force method to find the shortest distance, it is still used when finding the most optimal parameters. This seems counter-intuitive and inefficient as it goes against the purpose of the investigation. Computational methods could have been used to find these optimal parameters such as the use of nested for loops with varying ranges for each of the parameters, however, the run time for this becomes extremely long and the method becomes inefficient. Moreover, the `shortest_path` function does not always generate the shortest path possible. This is due to the random nature of the function. Currently, the number repeats in the function is 20, increasing this number to 100 would more consistently produce the shortest path possible but this becomes immensely computationally expensive.

Both problems have been immensely influential in the progression of physics and engineering. The collapse of the Tacoma bridge caused engineers to rethink the structural design of the bridges. With years of research into aerodynamics, and the new mathematical knowledge of vibrations and wave phenomena, a new era of more stable suspension bridges was created. The TSP algorithm is used globally by couriers, providing great financial benefit by creating the shortest path possible, thus greatly limiting the fuel cost of these companies.

Below is the shortest path and distance which the function could produce after a numerous number of repeats. 

REFERENCES

Tacoma Narrows Bridge collapses (2009) History.com. A&E Television Networks. Available at: <https://www.history.com/this-day-in-history/tacoma-narrows-bridge-collapses> (Accessed: January 22, 2023).

NP-hard problem (no date) from Wolfram MathWorld. Available at: <https://mathworld.wolfram.com/NP-HardProblem.html> (Accessed: January 22, 2023).

Bond, P. (2022) Vortex shedding 101: What is it & why does it matter?, Pi Engineering. Available at: <https://www.piengineering.ca/blog/vortex-shedding-101/> (Accessed: January 22, 2023).

Tacoma Narrows Bridge Failure (no date) enDAQ. Available at: <https://endaq.com/pages/tacoma-narrows-bridge-failure> (Accessed: January 22, 2023).

Cummings, N. (2000) Home, The OR Society. Available at: <https://www.theorsociety.com/about-or/or-methods/heuristics/a-brief-history-of-the-travelling-salesman-problem/> (Accessed: January 22, 2023).

P.J. McKenna and C.O Tuama, Amer. Math. Monthly 108, 738 (2001)

Baird, L. (no date) Simulated annealing , What is Simulated Annealing? Available at: <https://www.cs.cmu.edu/afs/cs.cmu.edu/project/learn-43/lib/photoz/.g/web/glossary/anneal.html> (Accessed: January 22, 2023).

Harmonic motion (no date) Harmonic motion 3. Available at:
<https://www.ippp.dur.ac.uk/~krauss/Lectures/NumericalMethods/Oscillator/Lecture/os3.html> (Accessed: January 22, 2023).

A brief introduction to numerical methods for Differential ... - lehman (2011). Available at:
<https://lehman.edu/academics/cmacs/documents/NumericalIntegrationTutorial.pdf>
(Accessed: January 22, 2023).

Euler-cromer method (no date) Computational Methods of Physics. Available at:
https://www.physics.udel.edu/~bnikolic/teaching/phys660/numerical_ode/node2.html
(Accessed: January 22, 2023).

Tietz, T. (2020) The Broughton Suspension Bridge and the Resonance Disaster, SciHi Blog.
Available at: <http://scihi.org/broughton-suspension-bridge-resonance-disaster/>
(Accessed: January 22, 2023).

Explaining why the millennium bridge wobbled- Cornell University (2005) ScienceDaily.
ScienceDaily. Available at:
<https://www.sciencedaily.com/releases/2005/11/051103080801.htm> (Accessed: January 22, 2023).