Curve_Fitting_Chaotic_Data

March 9, 2020

1 Using Damped Oscillation Data

2 Ring-Down Data

I'm using a data set from when the outer magnetic coils were running (See March 4, 2020 entry) {Side note: Is there an easy way to link these notebooks to entries in Onenote?}

This data should look like an exponential times a sine function like

$$f(x) = Ae^{Bx}\sin(\omega x + \delta)$$

so this should be a good testbed for curve fitting.

The next cell loads the data from the file into a variable named `data`. Note that the data is not is csv format, it is space delimited.

[2]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline


data = np.loadtxt(fname='more_data.txt', delimiter=' ')
```

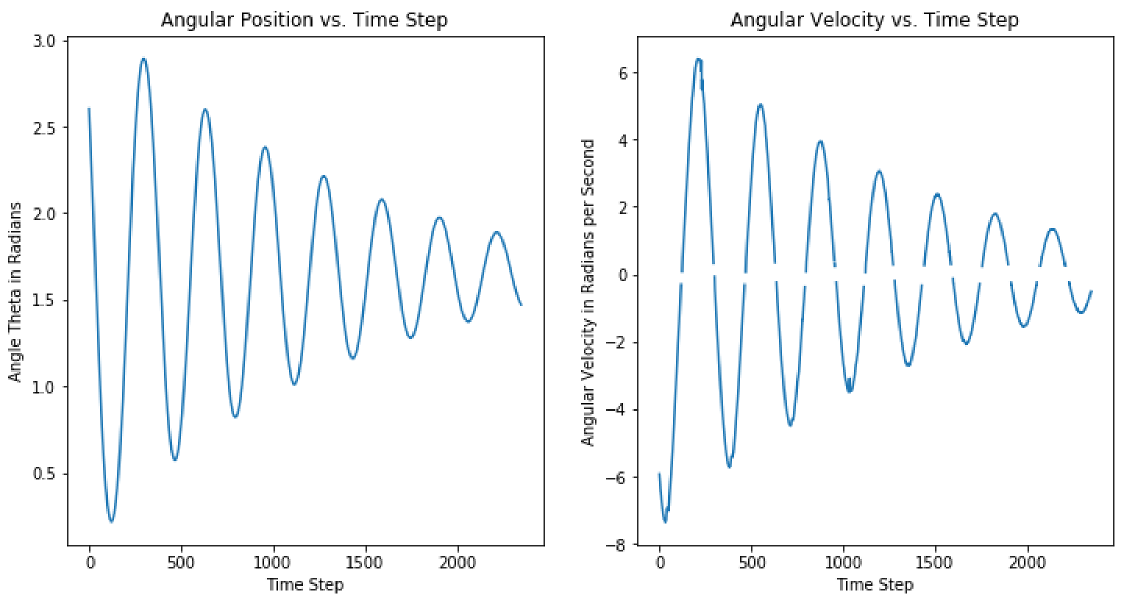
2.1 Plotting the Data

See March 4, 2020 entry for more info about this code (live links *would* be nice, but I think I can only link to the web pages and not entries in a Onenote file)

[3]:
```
print("Total data points in this file:", len(data[:,0]))

# Specifies which data points to start and stop with
N_starting_data = 150
N_ending_data = 2500

raw_theta = data[N_starting_data:N_ending_data,1]
raw_angular_velocity = data[N_starting_data:N_ending_data,2]
time_step_number = np.arange(0,len(raw_theta))
```

```
raw_theta_max = 2048

#Convert raw data to angular data
theta = raw_theta*np.pi/raw_theta_max

#Not sure if this is radians per second
angular_velocity = raw_angular_velocity*np.pi/raw_theta_max

# Plot data
fig, [ax1,ax2] = plt.subplots(1,2, figsize=(12,6))

ax1.plot(time_step_number, theta)
ax1.set_title("Angular Position vs. Time Step")
ax1.set_xlabel("Time Step")
ax1.set_ylabel("Angle Theta in Radians")

ax2.plot(time_step_number, angular_velocity)
ax2.set_title("Angular Velocity vs. Time Step")
ax2.set_xlabel("Time Step")
ax2.set_ylabel("Angular Velocity in Radians per Second")
plt.show()
```

```
Total data points in this file: 2560
```



The gaps in the velocity graph above are due to the fact that the teensy returns `nan` for velocities near zero. I don't think this will be a problem but I should keep this in the back of my mind when analyzing it.

3 Fitting A Curve

I'm going to use `curve_fit` from `scipy.optimize`.

To use this I need to define a function to fit the data. The requirements for this function are:

1. The first variable should be the independent variable
2. The next arguments should be all of the possible parameters
3. The function should return a number based on the input arguments

3.1 Define a Fitting Function

The format for defining a function is `def FUNCTION_NAME(ARGUMENT_VARIABLES):`. Any words in all-caps should be replaced with names you have made up for your situation. Don't forget the colon at the end of the line. This indicates that all lines that follow this one and are indented will be part of the function.

The next lines should calculate the result. Below I name it `my_result` but you can use any name you want.

The last line should start with `return` which tells Python that the following value is returned to the function call (e.g. if you had the following code: `stuff = fit_func(1, 2, 3, 4, 5)`, the function will return the result and assign it to the variable I've called `stuff`).

The `curve_fit` function is going to use this function to try and fit your data.

3.2 Finding the Best Parameters

The next step is to pass the fitting function and your data. The `curve_fit` routine will return the best-fit parameters and the covariance of the parameters (this tells us how good they fit the data, but we are going to ignore the covariance for now).

Python Tip: If a function returns a list of items, if you know how many items it returns you can assign each item to a variable. For example, I know `curve_fit` will return a list of two items, the parameters and the covariance. Rather than assigning the output from the function to a single variable and then having to unpack the list, I can assign the output from the function to two different variables.

The variable `parameters` is a list of the parameters of my fitting function, so I need to extract them. I'll use these parameters and plot the fit vs. my original data.

I know there is an exponential decay, but I'm going to try and fit the curve to a straight cosine function, just to see what I get. This will also give me a good starting guess for the frequency ω.

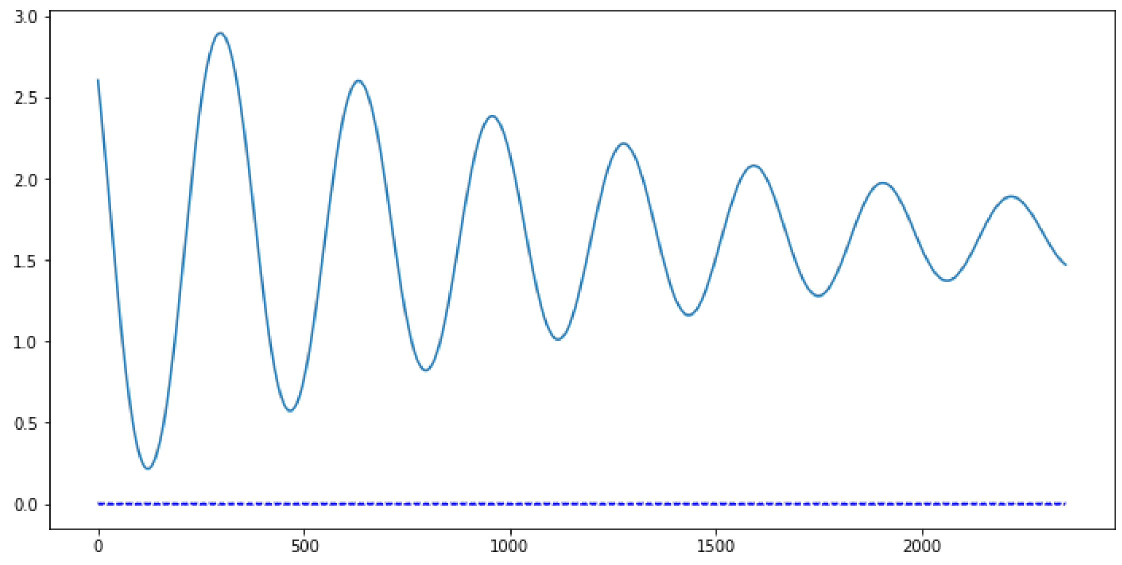
My fitting function is

$$f(x) = A\cos(\omega x + \phi)$$

Ooops, there is an offset form zero, so I need to add another parameter to my fitting function.

```
[4]: from scipy.optimize import curve_fit

def fit_func(x, A, omega, phi):
    my_result = A*np.cos(omega*x + phi)
    return my_result


parameters, param_covariance = curve_fit(fit_func, time_step_number, theta)

print(parameters)


Amp = parameters[0]
Om = parameters[1]
Ph = parameters[2]


y = fit_func(time_step_number, Amp, Om, Ph)


# Plot data
fig, ax = plt.subplots(1,1, figsize=(12,6))

ax.plot(time_step_number, theta)
ax.plot(time_step_number, y, 'b--')
plt.show()
```

```
[-0.00362662  0.99863913  2.06034   ]
```



3.3 Fitting Function Version 2

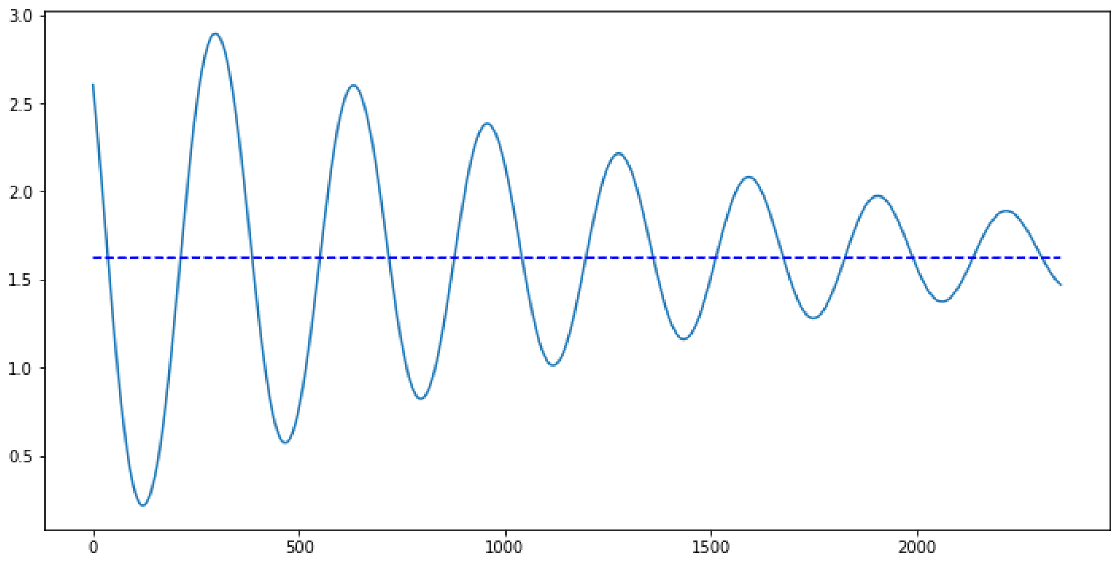
My fitting function is now

$$A\cos(\omega x + \phi) + B$$

The B term is just the zero-offset

[5]:
```python
from scipy.optimize import curve_fit

def fit_func(x, A, omega, phi, B):
    my_result = A*np.cos(omega*x + phi) + B
    return my_result


parameters, param_covariance = curve_fit(fit_func, time_step_number, theta)

print(parameters)


Amp = parameters[0]
Om = parameters[1]
Ph = parameters[2]
B = parameters[3]

y = fit_func(time_step_number, Amp, Om, Ph, B)


# Plot data
fig, ax = plt.subplots(1,1, figsize=(12,6))

ax.plot(time_step_number, theta)
ax.plot(time_step_number, y, 'b--')
plt.show()
```

```
[-9.97612961e-04  9.99915832e-01  2.12101972e+00  1.62222188e+00]
```



Oookay! Not getting a good fit. I'm guessing I need to input some guesses for the parameters.

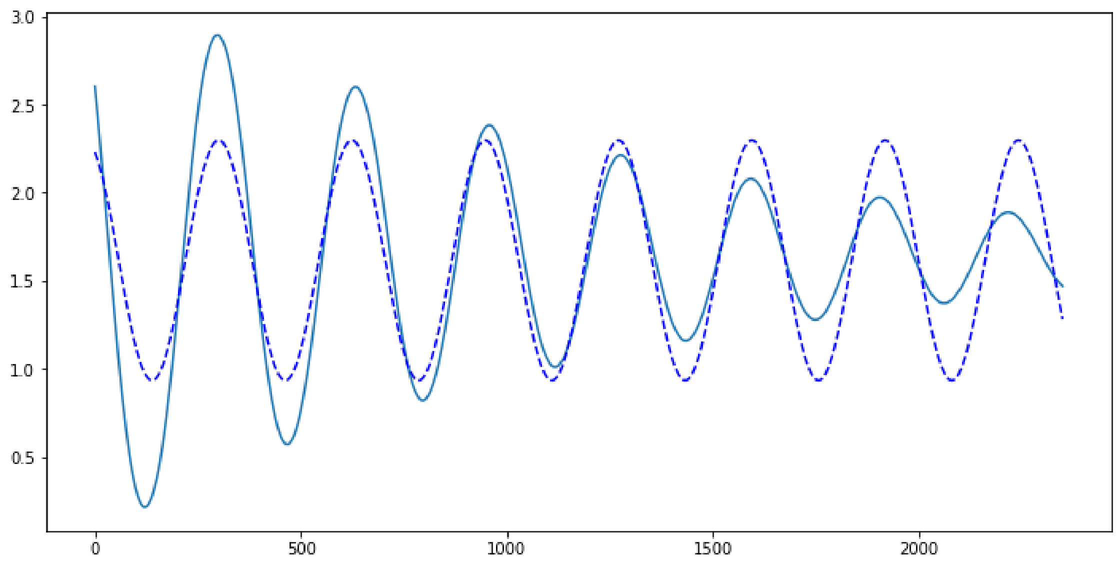
To specify an initial guess, add `p0=[1,2,3,4]` (or whatever numbers you want to start at) as an additional argument in `curve_fit`.

I'll try `B=1.6` from the previous fit. It looks like the amplitude `A` is around 1 so I'll use that as a guess. Since the period of the wave is roughly 400 time steps, I'll try `omega = 2*pi/400` as my guess for `omega`. I don't have a good guess for `phi` so I'll try 0.

[6]:
```
from scipy.optimize import curve_fit

def fit_func(x, A, omega, phi, B):
    my_result = A*np.cos(omega*x + phi) + B
    return my_result


parameters, param_covariance = curve_fit(fit_func, time_step_number, theta,␣
 ↪p0=[1, 2*np.pi/400, 0,1.6])

print(parameters)


Amp = parameters[0]
Om = parameters[1]
Ph = parameters[2]
B = parameters[3]

y = fit_func(time_step_number, Amp, Om, Ph, B)
```

```
# Plot data
fig, ax = plt.subplots(1,1, figsize=(12,6))

ax.plot(time_step_number, theta)
ax.plot(time_step_number, y, 'b--')
plt.show()
```

```
[-0.68127497  0.01942023 -2.69227735  1.61565024]
```

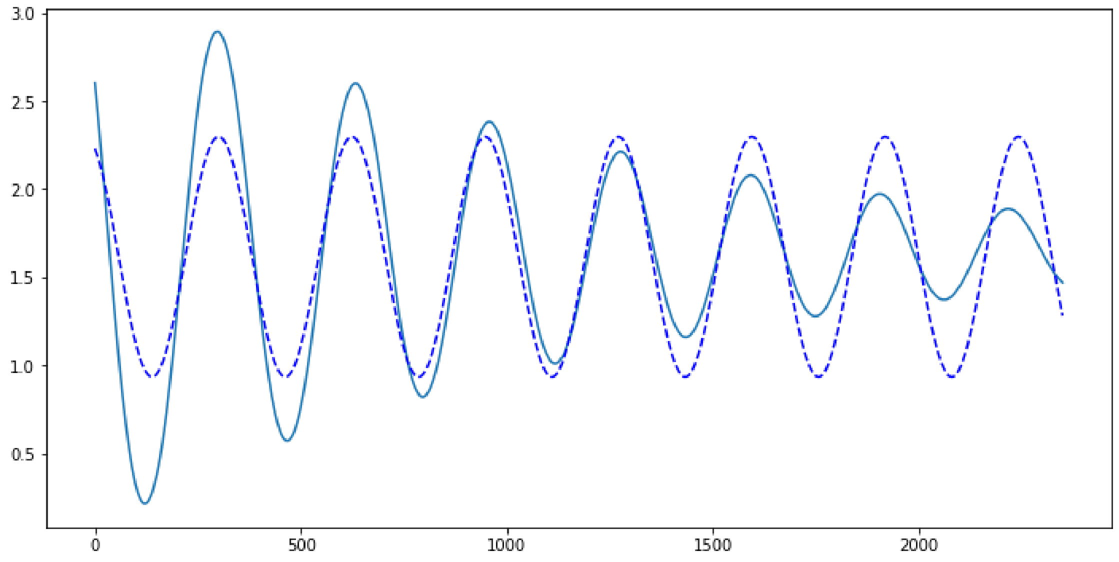


Not too bad. I wonder if I try a differet set of guesses if I'll get the same results. This will show how resilient the answer is.

[7]:
```
from scipy.optimize import curve_fit

def fit_func(x, A, omega, phi, B):
    my_result = A*np.cos(omega*x + phi) + B
    return my_result


parameters, param_covariance = curve_fit(fit_func, time_step_number, theta,␣
 ↪p0=[5, 2*np.pi/320, 5,1.6])

print(parameters)


Amp = parameters[0]
Om = parameters[1]
Ph = parameters[2]
```

```
B = parameters[3]

y = fit_func(time_step_number, Amp, Om, Ph, B)


# Plot data
fig, ax = plt.subplots(1,1, figsize=(12,6))

ax.plot(time_step_number, theta)
ax.plot(time_step_number, y, 'b--')
plt.show()
```

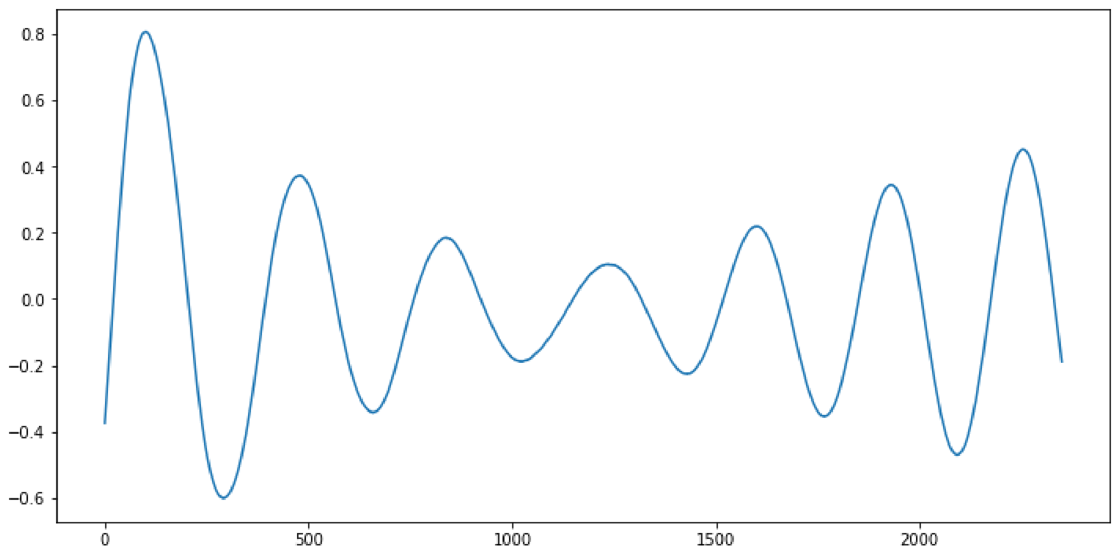
```
[0.68127478 0.01942027 6.73243746 1.61564963]
```



The only difference is in the `phi` term, which is understandable. Any value of `phi` that is $\phi = n2\pi + \phi_0$ (i.e. add an integral multiple of π) will work.

I should take a look at the residuals (which is `residuals = fitted_curve - actual data`). I know this result above is wrong but it would be instructive to look a the residuals.

[9]:
```
resid = y - theta

fig, ax = plt.subplots(1,1,figsize=(12,6))

ax.plot(time_step_number, resid)
plt.show()
```



That isn't what I expected (I expected an exponential decay) but it makes sense in hindsight.

Is there a way to see that exponential decay that is missing from the curve fit? Maybe comparing peaks?

3.4 Adding in the Exponential Decay

I know there is an exponential decay term due to damping so my fitting function should look something like:

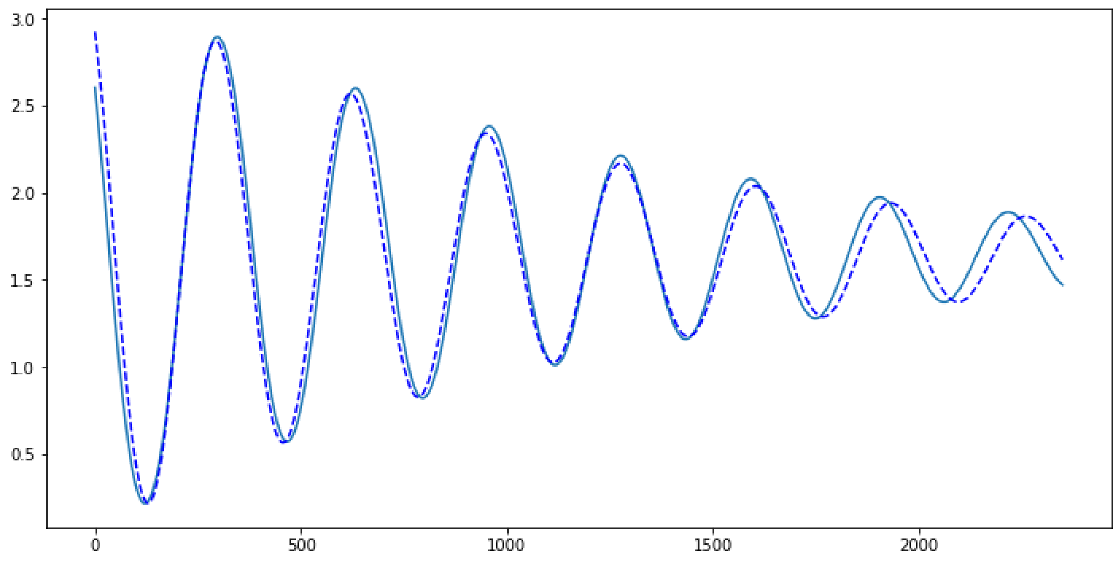
$$Ae^{kx+\phi_2}\cos(\omega x+\phi)+B$$

Let's see what this yields

[12]:
```
from scipy.optimize import curve_fit

def fit_func(x, A, omega, phi, B, k, phi2):
    my_result = A*np.exp(k*x+phi2)*np.cos(omega*x + phi) + B
    return my_result


parameters, param_covariance = curve_fit(fit_func, time_step_number, theta,␣
 ↪p0=[1, 2*np.pi/320, 0.5,1.6,0,0])

print(parameters)
```

```
Amp = parameters[0]
Om = parameters[1]
Ph = parameters[2]
B = parameters[3]
K = parameters[4]
Ph2 = parameters[5]

y = fit_func(time_step_number, Amp, Om, Ph, B, K,Ph2)


# Plot data
fig, ax = plt.subplots(1,1, figsize=(12,6))

ax.plot(time_step_number, theta)
ax.plot(time_step_number, y, 'b--')
plt.show()
```

```
[ 1.00781539e+00  1.91710313e-02  6.23345098e-01  1.63648565e+00
 -8.55856880e-04  4.53417534e-01]
```



4 Comparing Exponential Decay vs. Cosine

Let a subscript of `E` represent exponential decay fit and a subscript of `C` represent the pure cosine fit.

4.0.1 Amplitudes

$A_C = 0.681$

$A_E = 1.008$

This difference isn't surprising because the exponential decay term makes fitting the data more accurately easier.

4.0.2 Omega

$\omega_C = 0.01942$

$\omega_E = 0.01917$

Both agree pretty well. Looks good

4.0.3 Phi

$\phi_C = 6.732$

$\phi_E = .623$

This is roughly a factor of 2π difference. The added exponential decay probably shifts things a little.

4.0.4 Zero Offset

$B_C = 1.616$

$B_E = 1.636$

4.0.5 Decay Constant

$k_E = -8.558E - 4$

Nothing to compare this too but it should be the 1/number of time steps to decay to 1/e of initial max.

$1/k_E \approx 1200$ which is in the right ball park.

4.0.6 Phi_1

$\phi_1 = .4534$

Nothing to compare this too and I don't have a good feel for what to compare it to.

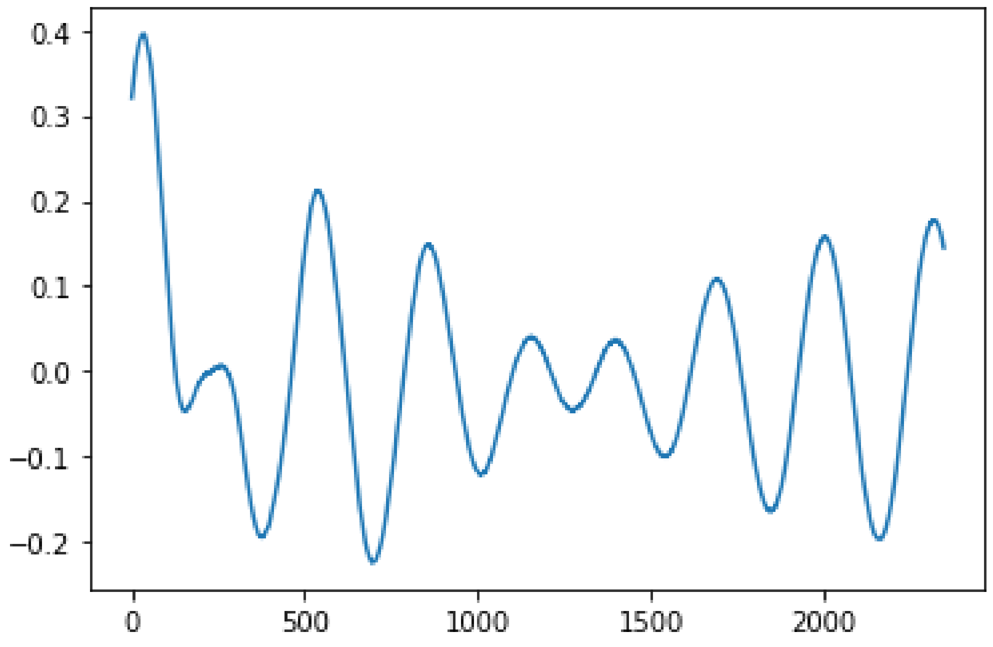
Let's take a look at the residuals

[14]:
```
# Period is roughly 1/k_E

print(1/-8.558E-4)
```

```
-1168.4973124561814
```

[13]:
```
residuals = (y - theta)
plt.plot(time_step_number,residuals)
```

[13]:
```
[<matplotlib.lines.Line2D at 0x1017d4cef0>]
```



Hmmmm. Not sure how to read that. It looks to be about half as big as before.

To Do: - Use different variables in each new cell so I can refer back to previous results by printing them in place rather than having to rerun a bunch of cells. Come up with a naming scheme

---

I'm going to try taking the real part of

$$e^{i\omega x} + e^{-i\omega x}$$

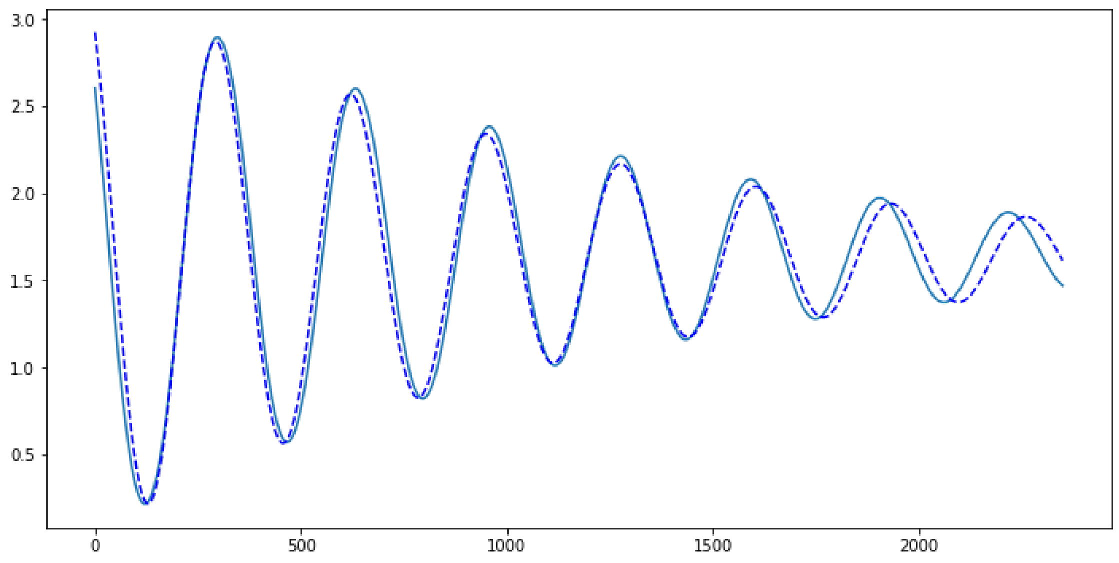
to see if this gives any different results.

[15]:
```
from scipy.optimize import curve_fit

def fit_func(x, A1, A2, omega, B, y0, phi):
    my_result = np.real(np.exp(-B*x)*(A1*np.exp(1j*(omega*x + phi)) + A2*np.
 ↪exp(1j*(omega*x + phi))) + y0)
```

```python
    return my_result


parameters, param_covariance = curve_fit(fit_func, time_step_number, theta,␣
 ↪p0=[2,2,2*np.pi/250, .1, 1.6, 0.5])

print(parameters)


A1 = parameters[0]
A2 = parameters[1]
omega = parameters[2]
B = parameters[3]
y0 = parameters[4]
phi = parameters[5]

y = fit_func(time_step_number, A1,A2,omega,B,y0, phi)


# Plot data
fig, ax = plt.subplots(1,1, figsize=(12,6))

ax.plot(time_step_number, theta)
ax.plot(time_step_number, y, 'b--')
plt.show()
```

```
[ 1.48405625e+02 -1.46819621e+02  1.91710102e-02  8.55875333e-04
  1.63648631e+00  6.23356316e-01]
```

```
/Users/toddzimmerman/anaconda3/envs/jupyter_lab_37/lib/python3.7/site-
packages/ipykernel_launcher.py:4: RuntimeWarning: overflow encountered in
multiply
  after removing the cwd from sys.path.
```



Nope. Nothing significantly different.

To Do: - Take a look in Classical text for a possible approximate solution (it looks like frequency varies)

```
[ ]:
```