Signal Filtering, Curve Fitting, and Ordinary Differential Equation Solving in Python

Rafael O. Soto

March 17, 2014
Phys-440
Illinois Institute of Technology, Chicago, IL 60616

Abstract

In this lab I implemented Fourier Transform techniques to filter a given signal, then I used curve fitting techniques in order to extract information from the same signal. Then I implemented an ordinary differential equation solver in order to recreate the signal from Newton's second law.

1. Introduction

Signal filtering is common throughout many scientific fields, it is often the case that data taken has to be filtered in order to acquire meaningful results. These processes are also have many engineering application. One of the simplest ways to filter out "noise" (unwanted artifacts of the signal) is to work in the frequency domain. One can do so by taking the Fourier transform of a discrete signal and filtering out the unwanted signals, these are usually less pronounced that the main signal when plotted in Fourier space. One can also use other filtering methods that act in the space domain, such as a Savitzky-Golay filter, I will be using Fourier techniques in this lab.

Once one has a clean signal it can be fit to a function in order to learn more about it. For example if the signal was a noisy sine wave, then fitting the clean discrete signal to a sin function can relay certain parameters of the wave such as the amplitude, frequency, and phase shift.

In this lab I will also use ODE(Ordinary Differential Equation) solving methods in order to try and reproduce the signal I received. This entails using built in libraries in SciPy that can solve a system of first order coupled ODEs that represent a simple harmonic system, later I will add in a time dependent external force and observe the behavior of the resulting function.

2. Analysis and Results

2.1. FFT Filtering

I used the discrete methods in scipy.fftpack in order to obtain the discrete Fourier transform of the signal which looked like this:

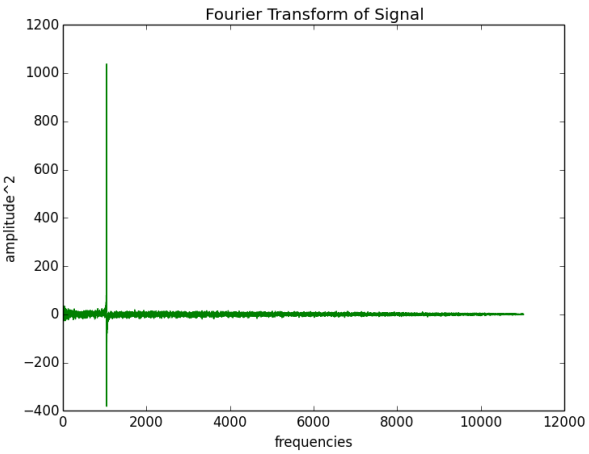

Figure 1: Fourier Transform before filtering.

The most pronounced peak is the main signal, the rest is just noise, so I set all of the smaller signals to zero in order to filter out the noise. After filtering out the unwanted components of the signal I recreated it using the inverse discrete Fourier transform method of fftpack.

The results of the filtering showed that the signal resembled a sinusoidal wave, and from the frequency domain graph it was clear that the frequency of this wave was 523 kHz, the plot of the smoothed signal is shown in conjunction to the curve fitting in figure 2.

2.2. *Curve Fitting*

The signal looked sinusoidal so I used a sine function in order to fit the signal. The way that signal filtering usually works is that you define a function and then the method attempts to minimize the difference between the function and the signal by adjusting certain parameters that are given. The method I used was curve_fit in the scipy.optimize library, this method takes as arguments a function to minimize, a signal, and initial values for the parameters. The closer that these values are to the actual signal the better the fit will be. This is how the function to be minimized needed to be initialized, the parameters that the method will return are optimized values for a,the amplitude, b, the angular frequency, c, the phase shift and d, the y displacement.

```
1 def func(x,a,b,c,d):
2       return -a*np.sin(b*x+c)+d
```

the results of the curve fitting gave the following parameters as shown in figure 2. Frequency is in kHz.

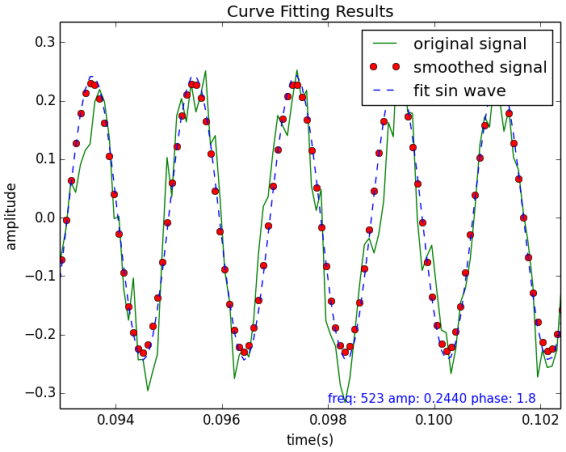

Figure 2: Results from FFT filtering and curve fitting superimposed over a section of the original signal.

2.3. *ODE Solving*

For this section I employed SciPy's odeint, which takes as an argument a system of first order ordinary differential equations and integrates them over a given time interval, using a given number of timesteps. The equation that I wanted to use to model the behavior of the previous signal was the equation for simple harmonic motion, in this case a simple spring.

$$F_k = -kx \rightarrow m\ddot{x} = -kx \tag{1}$$

Where $\ddot{x}$ is the second time derivative of x. In order for odeint to solve the second order ODE it needed to be broken down into two first order ODE. The results of the first order ode with initial conditions x[0]=0.277, x'[0]=0 are shown in figure 3.

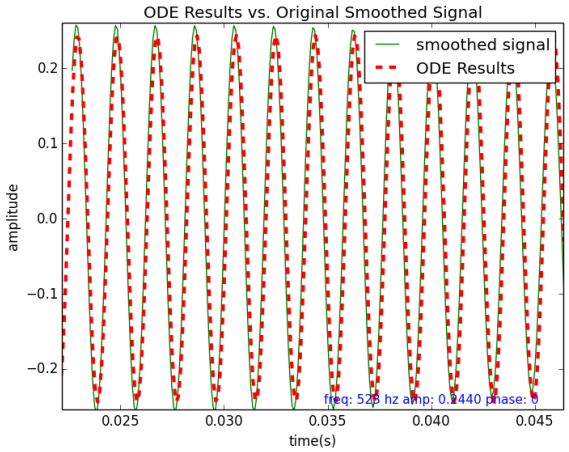

Figure 3: Results from ODE solver vs original signal.

2.4. *Solutions for Time-Independent External Force*

Here are the solutions for section 9.8 of A Survey of Computational Physics. [1]

1. This is the equation of motion once I added in the time-independent external force

$$m\ddot{x} = -kx + F_0 sin(\omega t) \tag{2}$$

2 & 3. Figure 4 is a plot of both the mode locking effect with a very large driving force, as well as a driving force that is close to the restoring force in order to produce beating.

4. The number of variations in intensity per unit time is indeed equal to the frequency difference.

$$\frac{(\omega - \omega_0)}{2\pi} \rightarrow \frac{(338 - 523)}{2\pi} \tag{3}$$

$$= 29 \approx 30 \tag{4}$$

According to figure four there are six beats, or six variations in intensity, in .2 seconds. If 30 is the number of beats per unit time, then adjusting this number for .2s we get 6, which is precisely what the graph depicts.

5. I made the runs in an Ipython Console and manually added the data to a numpy array.

6. Figure 5 is the plot and it clearly shows that the greatest amplitude occurs when the angular frequency of the driving force is equal omega naught, this behavior resembles resonance very well.

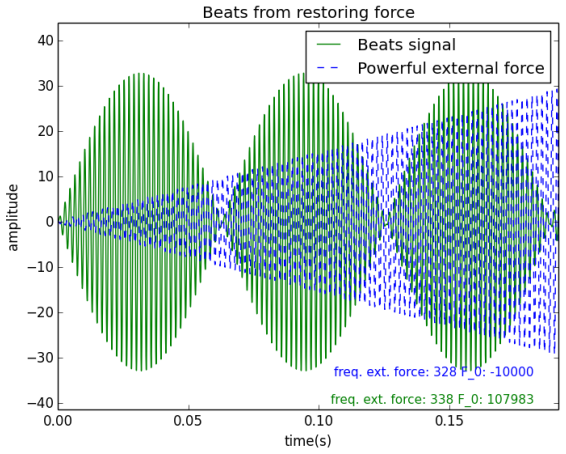

Figure 4: Beating and 500-pound-gorilla effect.

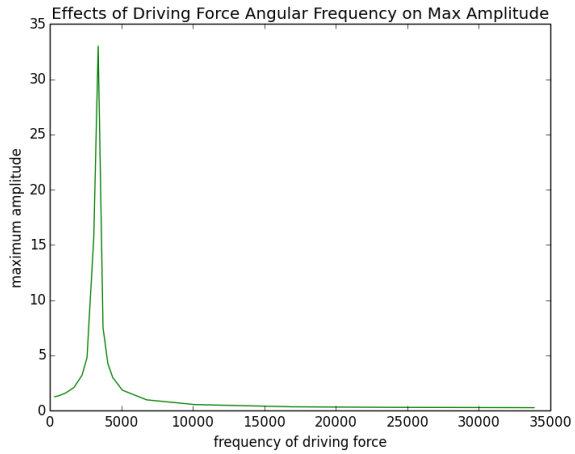

Figure 5: Varying the angular frequency of the driving force that causes beats.

3. Discussion

The results of the Fourier techniques shows that it is a viable method for filtering out unwanted frequencies and it was very easy to implement using SciPy. The curve fitting techniques in SciPy were also fairly easy to implement and gave a signal that very much resembled the original one and I am confident that the values obtained for the frequency and amplitude are correct. The ODE solving method further adds to the reliability of the results. The different results obtained when adding a time dependent external force were interesting. When the force was very small there was little effect, but when large it easily overpowered the original signal. The most interesting results occurred when the force was nearly equal to the restoring force. In this domain beats and resonances occurred, the resonance was maximal when the frequency of the driving force was equal to the frequency of signal, which makes qualitative sense. The graph produced by varying the frequency of the driving force led me to this conclusion. This lab was illuminating in that it had widely varying techniques from very different aspects of physics all applied to a single problem, this helped me see parallels between the different problems presented.

[1] Rubin, H. Landau, Manuel Jose Paez, Cristian C. Bordeianu *A Survey of Computational Physics* 2012: Princeton University Press.

[2] Python Software Foundation *Python v2.7.6 documentation* 1990-2014

[3] The People of Stack Exchange, Google *Internet.*

```
# -*- coding: utf-8 -*-
"""
Created on Tue Mar 11 16:05:19 2014

@author: waffles
"""

import numpy as np
import matplotlib.pyplot as plt

from scipy.signal import butter, lfilter



if __name__ == "__main__" :

from scipy.signal import freqz
from scipy.optimize import leastsq

signal=np.loadtxt('signal5.dat')
plt.plot(signal[:,0],signal[:,1])


t=signal[:,0]
x=signal[:,1]

#fourier filtering

signal=x.copy()
W=scipy.fftpack.fftfreq(signal.size, d=t[1]-t[2])
fsignal=scipy.fftpack.rfft(signal)

cut=fsignal.copy()
for i in range(len(fsignal)):
    if i>1103 or i<976: cut[i]=0

csignal=scipy.fftpack.irfft(cut)

plt.subplot(221)
plt.plot(t,signal)
plt.subplot(222)
plt.plot(fsignal)
plt.xlim(0,5000)
plt.subplot(223)
plt.plot(cut)
plt.xlim(0,5000)
plt.subplot(224)
plt.plot(t,csignal)
plt.show()
```

```
x=t[250:2000]

yn=csignal[250:2000]
popt, pcov=curve_fit(func,x,yn,(.27,3300,1,0))

ynew=func(x,*popt)

"""Plotting The Curve Fitting"""

fig=plt.figure()
ax=fig.add_subplot(111)
ax.set_title('Curve Fitting Results')
ax.set_xlabel('time(s)')
ax.set_ylabel('amplitude')

ax.plot(x,signal[250:2000],'g', label="original signal")
ax.plot(x,csignal[250:2000],'ro',lw=3, label="smoothed signal")
ax.plot(x,ynew, 'b--', label="fit sin wave")

ax.text(0.95, 0.01, "freq: %.3s amp: %.6s phase: %.3s" % (popt[1]/(2*np.pi), -popt[0],popt[2]),
        verticalalignment='bottom', horizontalalignment='right',
        transform=ax.transAxes,
        color='blue', fontsize=11)
plt.show()
plt.legend()


####ODE PART######


from scipy.integrate import odeint
omega=523*np.pi*2

def deriv(y,t):
    uprime=y[1]
    wprime=-y[0]*(omega**2)
    yprime=np.array([uprime,wprime])
    return yprime
```

```
start=0
end=1
numsteps=10000
time=np.linspace(start,end,numsteps)
y0=np.array([0.244,0])
y=scipy.integrate.odeint(deriv,y0,time)

plt.plot(time,y[:,0])

##ploting ode results with original smoothed signal.
fig=plt.figure()
ax=fig.add_subplot(111)
ax.set_title('ODE Results vs. Original Smoothed Signal')
ax.set_xlabel('time(s)')
ax.set_ylabel('amplitude')

ax.plot(x,csignal[250:2000],'g', label="smoothed signal")
ax.plot(time,y[:,0],'r--',lw=3, label="ODE Results")

ax.text(0.95, 0.01, "freq: %.3s hz amp: %.6s phase: %.3s" % (523, -popt[0],0),
        verticalalignment='bottom', horizontalalignment='right',
        transform=ax.transAxes,
        color='blue', fontsize=11)
plt.show()
plt.legend()

##ODE with restoring force.

omega=523*np.pi*2
omega2=omega + 100
F=omega**2 - 100

def deriv(y,t):
    uprime=y[1]
    wprime=-y[0]*(omega**2)+F*np.sin(omega2*t)
    yprime=np.array([uprime,wprime])
    return yprime

start=0
end=.5
numsteps=1000000
time=np.linspace(start,end,numsteps)
y0=np.array([0.244,0])
y=scipy.integrate.odeint(deriv,y0,time)

omega3=omega
F2=-1000000
```

```
    def deriv2(y,t):
        uprime=y[1]
        wprime=-y[0]*(omega**2)+F2*np.sin(omega3*t)
        yprime=np.array([uprime,wprime])
        return yprime

    time=np.linspace(start,end,numsteps)
    y0=np.array([0.244,0])
    y2=scipy.integrate.odeint(deriv2,y0,time)



    #plt.plot(time,y[:,0])

    fig=plt.figure()
    ax=fig.add_subplot(111)
    ax.set_title('Beats from restoring force')
    ax.set_xlabel('time(s)')
    ax.set_ylabel('amplitude')

    ax.plot(time,y[:,0],'g', label="Beats signal")
    ax.plot(time,y2[:,0],'b--', label="Powerful external force")

    ax.text(0.95, 0.01, "freq. ext. force: %.3s F_0: %.6s" % (omega2, F),
            verticalalignment='bottom', horizontalalignment='right',
            transform=ax.transAxes,
            color='green', fontsize=11)

    ax.text(0.95, 0.08, "freq. ext. force: %.3s F_0: %.6s" % (omega3, F2),
            verticalalignment='bottom', horizontalalignment='right',
            transform=ax.transAxes,
            color='blue', fontsize=11)
    plt.show()
    plt.legend()

    #results from different omegas, data gathered from ipython console
    omega=523*np.pi*2
    l=np.zeros((18,2))
    l[0]=(omega + 100)/10, 1.27
    l[1]=(omega + 100)/7, 1.3
    l[2]=(omega + 100)/5, 1.37
    l[3]=(omega + 100)/3, 1.61
    l[4]=(omega + 100)/2, 2.1
    l[5]=(omega + 100)/1.5, 3.21
    l[6]=(omega + 100)/1.3, 4.8
    l[7]=(omega + 100)/1.1, 15.7
    l[8]=(omega + 100)/1, 33
    l[9]=(omega + 100)*1.1, 7.5
    l[10]=(omega + 100)*1.2, 4.26
    l[11]=(omega + 100)*1.3, 3
```

```
    l[12]=(omega + 100)*1.5, 1.85
    l[13]=(omega + 100)*2, .98
    l[14]=(omega + 100)*3, .55
    l[15]=(omega + 100)*5, .35
    l[16]=(omega + 100)*7, .30
    l[17]=(omega + 100)*10, .27

    fig=plt.figure()
    ax=fig.add_subplot(111)
    ax.set_title('Effects of Driving Force Angular Frequency on Max Amplitude')
    ax.set_xlabel('frequency of driving force')
    ax.set_ylabel('maximum amplitude')

    ax.plot(l[:,0],l[:,1],'g', label="Beats signal")
```