Signal Processing Visualization

DSP visualization is a rather broad term. It could mean anything from spectral analysis (ex. Power Spectral Desnity), to time-frequency representations (ex. spectrograms) to domain-specific visualization (ex. range-doppler maps). I will focus on visualizations I have made very often. Typically, I have used visualizations to separating signal from noise, and occasionally classify what that signal is. At work, I was given a data set of ”ambient noise” to dissect. Luckily I work in acoustics, so alongside my visualizations I was also able to listen to the data. My initial results did not ”sound” like the actual signal, so I begin to look deeper. After playing with my visualization tool, I discovered my window tended to collapse a unique signal into a tone, or reduce the resolution of lower-frequency components to an unacceptable level. I stumbled upon the time-frequency resolution (Heisenberg).

Objective

I do not wish to debate the differences between Matlab and Python (cost, ease-of-use, maintainability, etc). Coming from years of Matlab experience, I simply wish to see the difficulty of translating a somewhat complex visualization from Python to Matlab.

From a signal processing perspective, my goal is to be able to easily and accurately identify the two signals I am summing

Contents

1 Matlab

Matlab's built-in spectrogram tools perform very well, but as always there is room for improvement. Notably, setting a log-scale on the frequency axis comes out rather ugly. In the case of the low-frequency resolution, it also leaves a large white bar which is not very appealing. I solved this by using an unsupported helper function, but that is by no means a stable solution. Matlab's Wavelet tool works spectacularly, albeit it is very computationally complex (inherent to the algorithm).

```
clc; clear all; close all;
fs   = 8E3; t = 0:1/fs:1;
fs1  = 3E3; PRF1 = 4E-3 > mod(t, 1E-2);  y1 = sin(2*pi*fs1.*t).*PRF1;
fs2  = 50;  PRF2 = 6E-2 > mod(t, 5E-1);  y2 = sin(2*pi*fs2.*t).*PRF2;
y = y1 + y2 ;%+ randn(1,length(y1));

subplot(4,1,1); plot(t,y1,t,y2+2); title('Time Series Data');  legend('HF Signal', 'LF Signal');
addpath('C:\Users\jpsullivan\Documents\MATLAB\Examples\R2019a\wavelet\CWTTimeFrequencyExample');
pause(1);
subplot(4,1,2); tic; [S,F,T] = spectrogram(y, 2^6, 2^6-1, 2^7, fs, 'yaxis'); a = toc;
helperCWTTimeFreqPlot(S,T,F,'surf','Fine Time resolution','Hz')
set(gca, 'yscale','log'); title(['Fine Time Resolution t= ',num2str(a,2), ' s']);
subplot(4,1,3); tic; [S,F,T] = spectrogram(y, 2^11, 2^11-1, 2^12, fs, 'yaxis'); a = toc;
helperCWTTimeFreqPlot(S,T,F,'surf','Fine Frequency Resolution','Hz')
set(gca, 'yscale','log'); title(['Fine Frequency Resolution t= ', num2str(a,2), ' s']);
subplot(4,1,4); tic; [cfs,f] = cwt(y, fs, 'frequencylimits', [20, 5E3]); a = toc; %, 'frequencylimits', [20, 5E3]
helperCWTTimeFreqPlot(cfs,t,f,'surf','CWT Output','Hz')
set(gca, 'yscale','log'); title(['Wavelet Transform t= ',num2str(a,2), ' s'])
```

Alternatively, similar results without using the sketchy example function (less pretty log-spacing)

```
clc; clear all; close all;
addpath('/Users/JP-Macbook/Documents/MATLAB/Examples/R2019a/wavelet/FinancialDataExample')
fs   = 8E3; t = 0:1/fs:1;
fs1  = 3E3; PRF1 = 4E-3 > mod(t, 1E-2);  y1 = sin(2*pi*fs1.*t).*PRF1;
fs2  = 50;  PRF2 = 6E-2 > mod(t, 5E-1);  y2 = sin(2*pi*fs2.*t).*PRF2;
y = y1 + y2 ;%+ randn(1,length(y1));

subplot(4,1,1); plot(t,y1,t,y2+2); title('Time Series Data');  legend('HF Signal', 'LF Signal');
pause(0.5);
subplot(4,1,2); tic; [S,F,T] = spectrogram(y, 2^6, 2^6-1, 2^7, fs, 'yaxis'); a = toc;
helperCWTTimeFreqPlot(S,T,F,'surf','Fine Time resolution','Hz'); ylim([20, fs/2]);
set(gca, 'yscale','log'); title(['Fine Time Resolution t= ',num2str(a,2), ' s']);
clearvars S F T; pause(0.5);
subplot(4,1,3); tic; [S,F,T] = spectrogram(y, 2^11, 2^11-1, 2^12, fs, 'yaxis'); a = toc;
helperCWTTimeFreqPlot(S,T,F,'surf','Fine Frequency Resolution','Hz'); ylim([20, fs/2]);
set(gca, 'yscale','log'); title(['Fine Frequency Resolution t= ', num2str(a,2), ' s']);
clearvars S F T; pause(0.5);
subplot(4,1,4); tic; [cfs,f] = cwt(y, fs, 'frequencylimits', [20, 5E3]); a = toc;
helperCWTTimeFreqPlot(cfs,t,f,'surf','CWT Output','Hz'); ylim([20, fs/2]);
set(gca, 'yscale','log'); title(['Wavelet Transform t= ',num2str(a,2), ' s']);
```

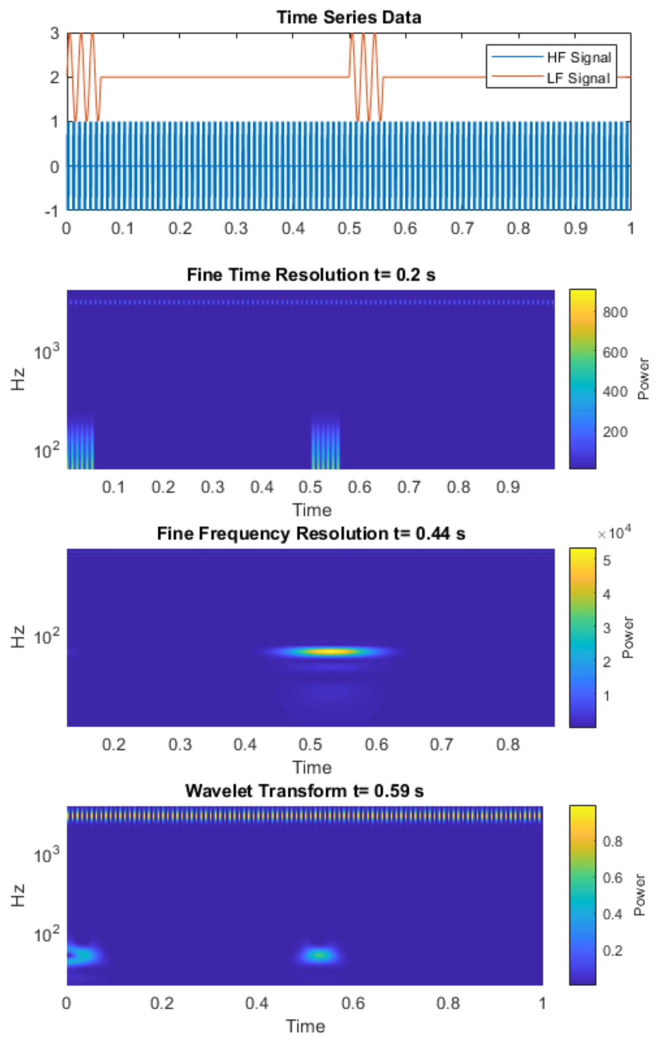

Figure 1: Matlab Spectral Analysis

1. The upper subplot shows the time series data, baseline to compare temporal resolution
2. fine time resolution, catching HF, poor LF performance
3. Fine frequency resolution. HF is low-power tone (barely visible). Smearing LF signal
4. Wavelet transform: Better time resolution for both signals, and better power estimates.

2 Python

There are numerous methods for signal analysis in Python. I will focus on using Matplot-lib and two methods for Wavelet's. Namely the scipy/signal/cwt package, and the PyWavelets package. Both of which are included within the Anaconda distribution of Python, with no extra installation steps required. There is also a wrapper function available via *Pip* called Scalogram which I attempted to use.

There are numerous visualizations online of beautiful Scalograms using these packages, however I was not able to replicate my Matlab results in Python. I believe I may be misusing the "scales," and attempting to calculate very small (or large) frequency components, causing my program to time out. I was able to generate some similar Spectrogram plots in Python. However, when I tried to set the frequency axis to log a portion of the plot would always revert to whitespace. There are a handful of Stackoverflow posts on this, but many simply say it is a known bug and report a fix that did not quite work for me.

I was able to produce Spectrograms that captured the essence of the signal processing dilemna in Python, but I was unable to really work with the Wavelet tools. Using the given scale-frequency code, I manually calculated the scales that corresponded to my frequency of interest and manually performed a single-scale CWT (single filterbank). The results are. . . not ideal. The calculation about as long to run as the Matlab counterpart. Admittidely, I am sure the parameters could be fine tuned. Also, there is a strong modulation present in both the high-frequency and low-frequency signals, that is otherwise absent in the Matlab analysis. Essentially, Matlab is doing a bunch of stuff under the hood that I really do not appreciate.

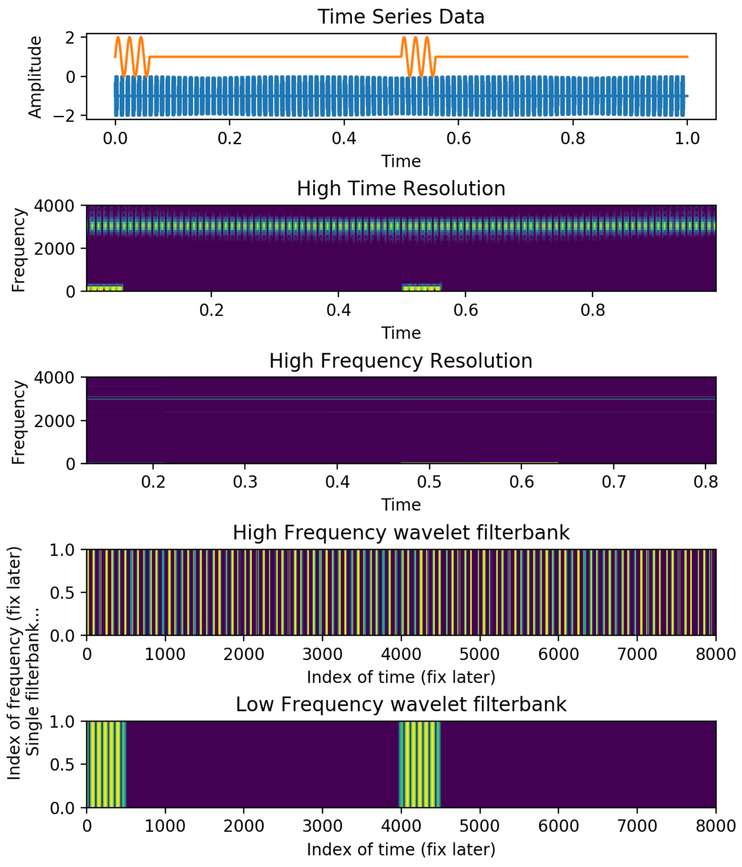

Figure 2: Python Spectral Analysis

```
import numpy as np
from math import sin, pi
import matplotlib.pyplot as plt
import matplotlib.mlab as mlab
from scipy import signal
import pywt
import scaleogram as scg

from time import sleep;
# clc; clear all; close all;
# addpath('/Users/JP-Macbook/Documents/MATLAB/Examples/R2019a/wavelet/FinancialDataExample')
# fs   = 8E3; t = 0:1/fs:1;
# fs1  = 3E3; PRF1 = 4E-3 > mod(t, 1E-2);  y1 = sin(2*pi*fs1.*t).*PRF1;
# fs2  = 50;  PRF2 = 6E-2 > mod(t, 5E-1);  y2 = sin(2*pi*fs2.*t).*PRF2;
# y = y1 + y2 ;%+ randn(1,length(y1));

# subplot(4,1,1); plot(t,y1,t,y2+2); title('Time Series Data');  legend('HF Signal', 'LF Signal');

# Sample signals
fs   = 8e3;
T    = 1;
N    = fs*T;
t    = np.linspace(0, T, N)
# # Signal 1
fs1  = 3E3                      # High frequency, "fast" signal
fs2  =  50
# y1 = np.array([0]);
y2 = np.array([0]);
prf1 = (4E-3 > np.mod(t, 1E-2))
prf2 = (6E-2 > np.mod(t, 5E-1))
y1   = np.sin(2*pi*fs1*t)*prf1
y2   = np.sin(2*pi*fs2*t)*prf2
# for time in t:
#     prf1 = 4E-3 > (time % 1E-2)       # Apply modulating PRF
#     val1 = sin(2*pi*fs1*time)*prf1;
#     np.append(y1, [val1])    # Hard-gated tone
#     prf2 = 6E-2 > (time % 5E-1)
#     val2 = sin(2*pi*fs1*time)*prf2;
#     np.append(y1, 2)    # Hard-gated tone

#     print(f'{val1:1.4f}   {val2:1.4f}')

print(f'DEBUG SHAPES: t: {t.shape}  prf1: {prf1.shape}    y1: {y1.shape}    y2: {y2.shape}')

print("trying to plot...\n")
plt.subplot(5, 1, 1)
plt.title("Time Series Data")
plt.plot(t, y1 - 1)
plt.plot(t, y2 + 1)
plt.xlabel("Time"); plt.ylabel("Amplitude")
Npts = 2**6
NFFT = 2**7 # This is intuitively backwards

h = plt.subplot(5,1,2)
# h.set_yscale('log')
Pxx, freq, t2 = mlab.specgram(y1+y2, Fs = fs, NFFT = Npts, noverlap = round(Npts - Npts/3))
plt.title("High Time Resolution")
plt.xlabel("Time"); plt.ylabel("Frequency")
j = h.pcolor(t2, freq, 20*np.log10(Pxx), clim = [-100, -80])
j.set_clim(vmin = -100, vmax = -50)
```

```
# h.set_yscale('log')



# Spectrogram number 2
Npts = 2**11
NFFT = 2**12 # This is intuitively backwards
h = plt.subplot(5,1,3)
Pxx, freq, t2 = mlab.specgram(y1+y2, Fs = fs, NFFT = Npts, noverlap = round(Npts - Npts/3))
plt.title("High Frequency Resolution")
plt.xlabel("Time"); plt.ylabel("Frequency")
j = h.pcolor(t2, freq, 20*np.log10(Pxx))
maxval = 20*np.log10(Pxx).max();
j.set_clim(vmin = maxval-40, vmax = maxval)
# h.set_yscale('log')
print('Computing wavelet')
widths = np.arange(2,31)
# Wavelet transform
scales = np.array([0.5333335])
print(scales); wind = 'gaus1'
freqs = pywt.scale2frequency(wind, scales)*fs;
print(f'Scales: {freqs}')
# '''
coef, freqs = pywt.cwt(y1+y2, sampling_period = 1/fs, scales = scales, wavelet = wind)
print('Plotting wavelet')
print(f'Shape of coef: {coef.shape}')
h =  plt.subplot(5,1,4)
j = h.pcolor(20*np.log10(abs(coef)+1e-8))
maxval = 20*np.log10(abs(coef)+1e-8).max()
j.set_clim(vmin = maxval-20, vmax = maxval)
plt.xlabel('Index of time (fix later)')
plt.ylabel('Index of frequency (fix later)                          \nSingle filterbank...
')
plt.title('High Frequency wavelet filterbank')
scales = np.array([32])
print(scales); wind = 'gaus1'
freqs = pywt.scale2frequency(wind, scales)*fs;
print(f'Scales: {freqs}')
# '''
coef, freqs = pywt.cwt(y1+y2, sampling_period = 1/fs, scales = scales, wavelet = wind)
print('Plotting wavelet')
print(f'Shape of coef: {coef.shape}')
h =  plt.subplot(5,1,5)
j = h.pcolor(20*np.log10(abs(coef)+1e-8))
maxval = 20*np.log10(abs(coef)+1e-8).max()
plt.xlabel('Index of time (fix later)')
# plt.ylabel('Index of frequency (fix later)\nSingle filterbank...')
plt.title('Low Frequency wavelet filterbank')
j.set_clim(vmin = maxval-20, vmax = maxval)

# plt.colorbar(j);
# '''
'''
# # ATTEMPT with SciPy
# cwtmatrix = signal.cwt(y1+y2, signal.ricker, widths);
'''

# ATTEMPT with Scaleogram
'''
scales = np.logspace(1, 10, num=5, dtype=np.int32)
```

```
print(scales)
#scales = np.arange(15,600, 4)
ax = scg.cws(t, y1+y2, scales, figsize=(12,6), ylabel="Period [Seoconds]", xlabel='Seconds', yscale='log')
ticks = ax.set_yticks([2,4,8, 16,32])
ticks = ax.set_yticklabels([2,4,8, 16,32])
print('Done')
'''
plt.show(block = True)
```