Zach Tzavelis

ME465 - Sound and Space

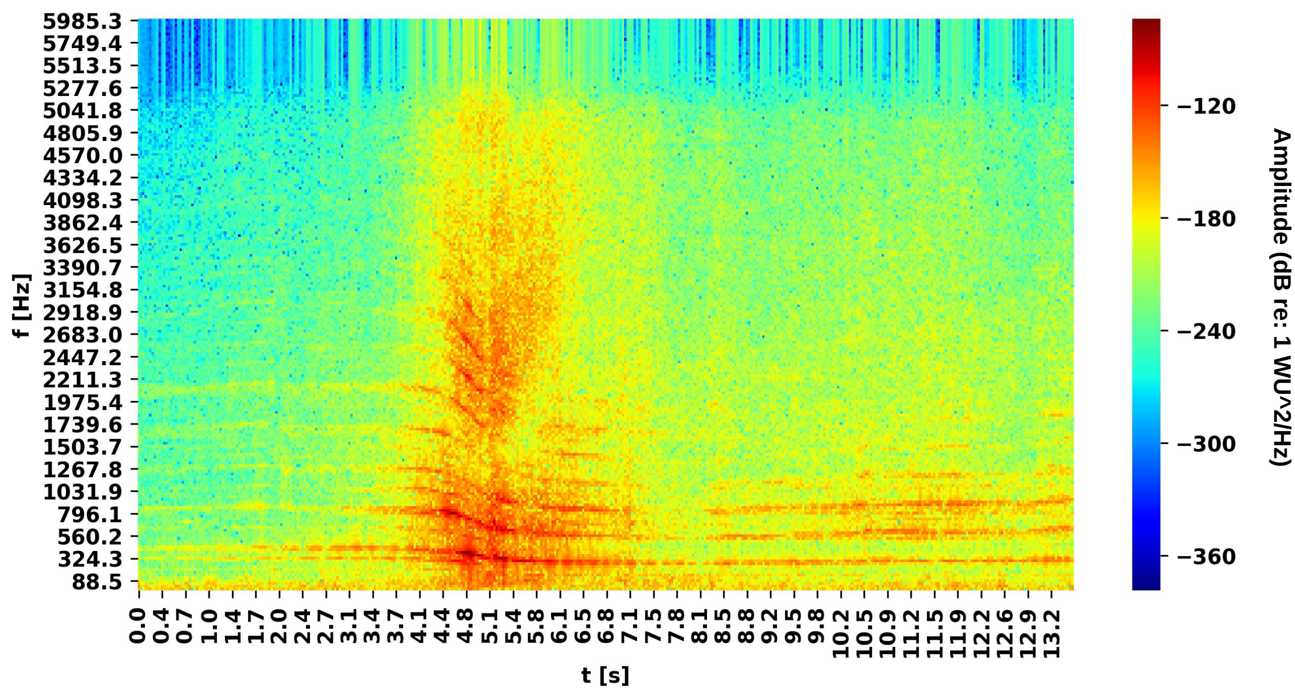
02/28/2020

<u>HW4: Spectrogram and Pulse Noise Revisited</u>

Spectrogram Analysis

The purpose of the assignment was to analyze a time domain signal of a formula one car as it passes by a microphone. A spectrogram was created which shows how the power spectral density changes over time (Figure 1). The colors represent the amplitude of the power spectral density in decibels.

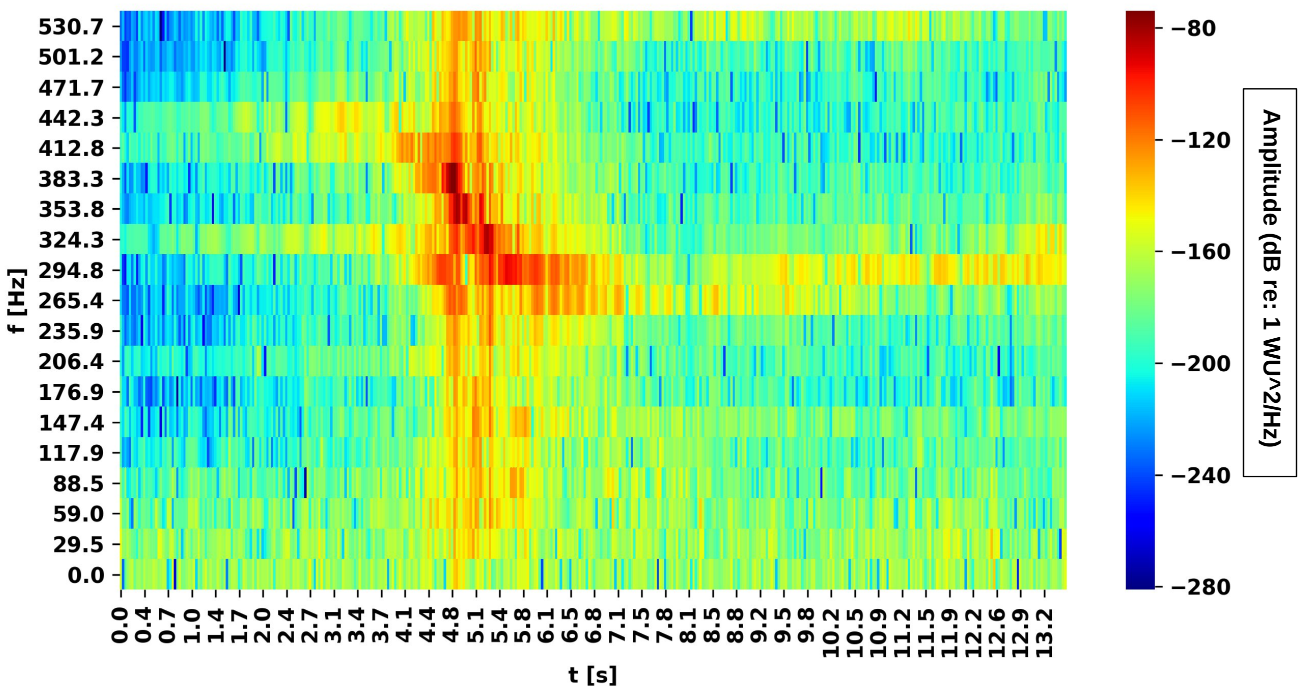
<u>Figure 1: Spectrogram of a Formula One Car (Overlap: 25%, Number of Intervals: 300)</u>



Below ~2000 Hz there are a series of curves which span the total length of the signal — these are the car's harmonics. The reason that they change over time is due to the fact that the car is initially traveling toward the microphone, and then away, causing a change in perceived frequency at the microphone — this is called the Doppler effect. Although the car's emitted frequencies are constant in its inertia frame, as the car moves toward the microphone the velocity of the car and the waves 'add' as more waves pass the microphone per second than they do as the car drives a way. Fewer waves hit the source per second as it drives away because the wave hitting the microphone and the car are traveling in opposite directions. Zooming into the spectrogram (Figure 1a) at the harmonic starting at 442 Hz ((f_h), one can see it is most intense at 295 Hz (f_0) when it is right in front of the microphone and assumed to see no relative velocity, and then turns to 383 Hz (f_l) afterward; using this information we can use Equation 1 to calculate the velocity that the car is traveling at — in this case its 70.6 m/s or about 157 mph, which seems reasonable for a race car.

Equation 1: $v = c\frac{(f_h - f_l)}{(f_0)}$

Figure 1a: Zoomed-in Spectrogram



Pulse in Noise Revisited

The purpose of Part B in this exercise was to learn how to use auto-correlation to discover properties about a wave such as the amount of times it has a repeating underlying signal (and its length). Figure 2 shows the entire auto-correlation of the 'signal in noise'; it can be seen that there is a single large peak in the middle and smaller ones spanning the length of time. The single largest peak corresponds with when there is no shift between the two signals, and we expect there to be a large amplitude here because the signals completely overlap.

Figure 2: Auto-correlation of Signal In Noise

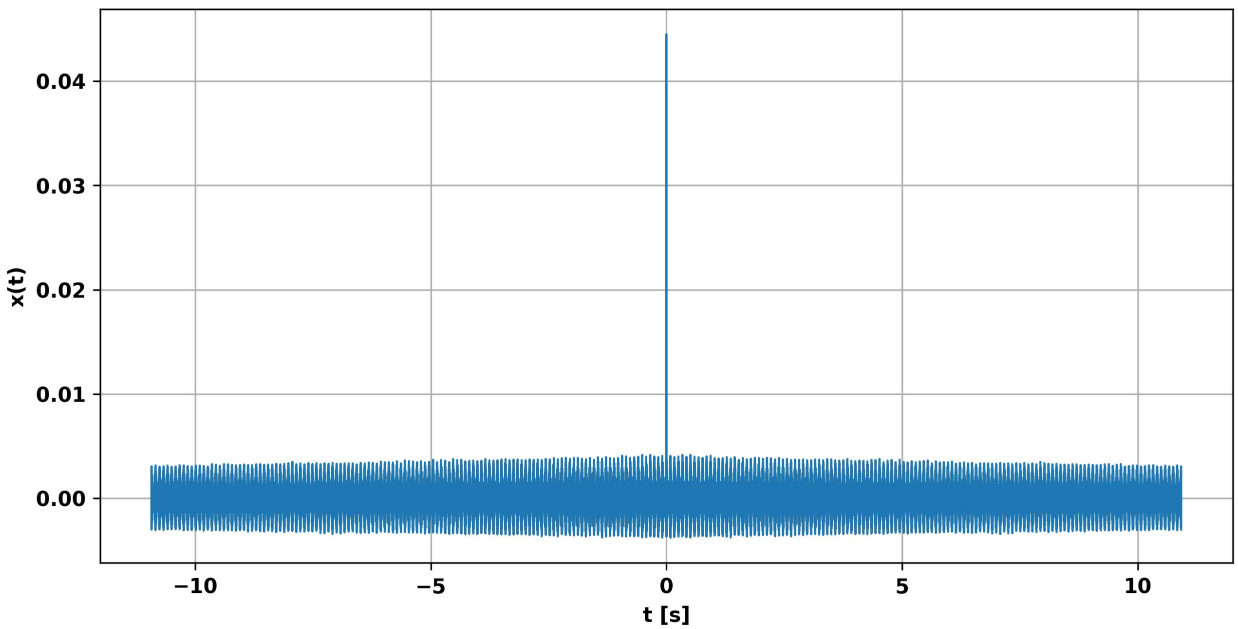
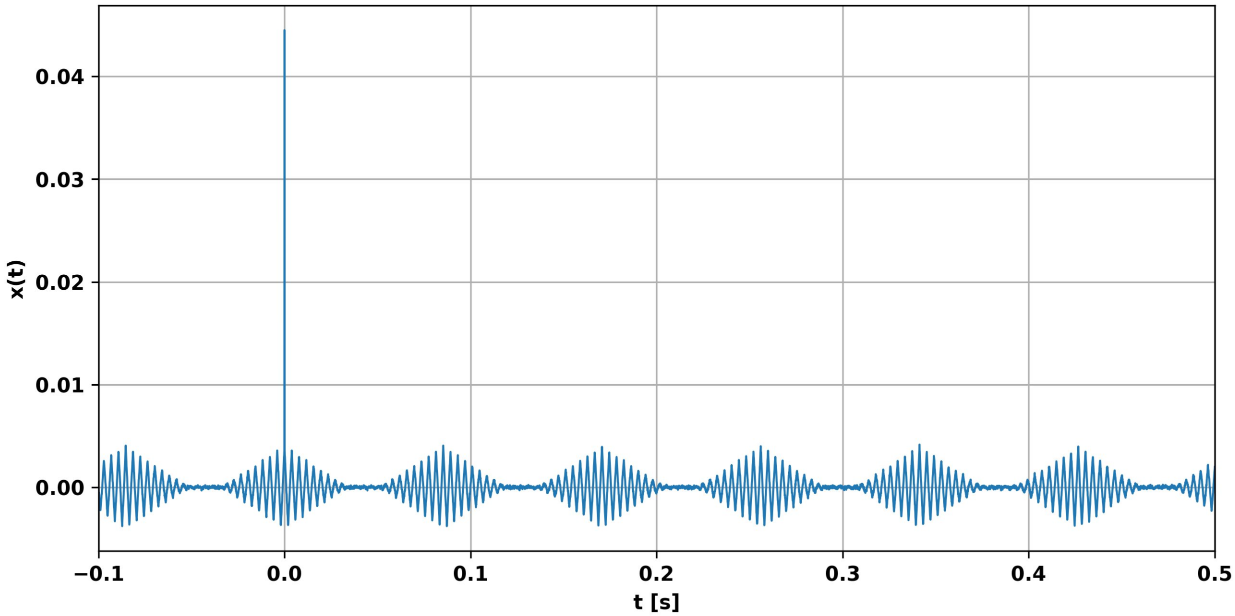


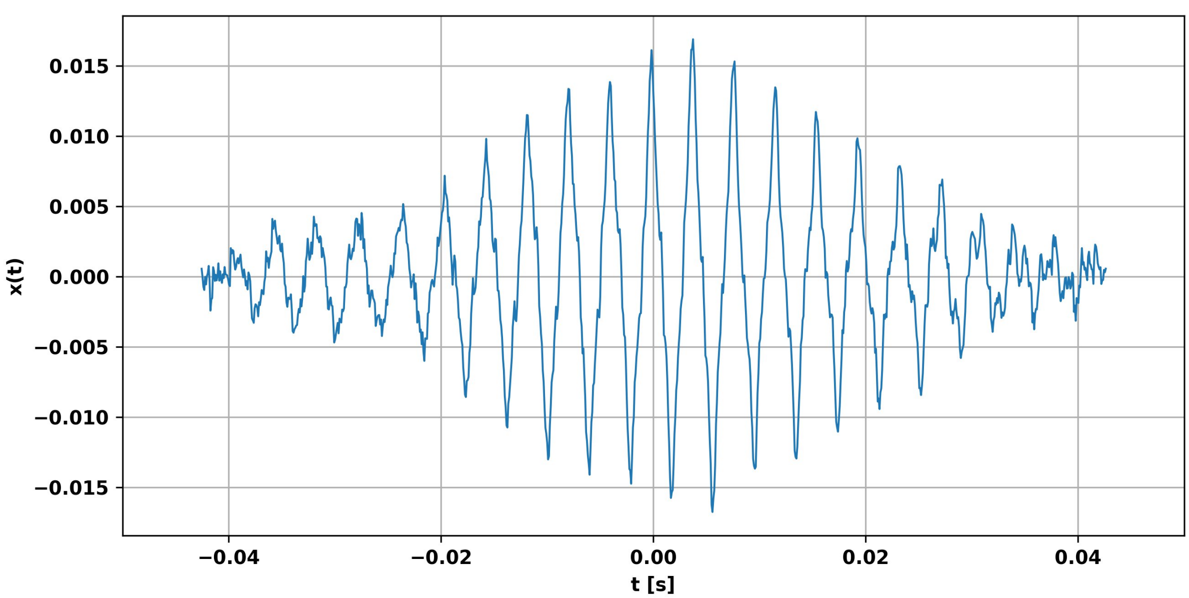
Figure 3 shows a zoomed view of Figure 2 in order to show the 256 smaller peaks (not counting the large one at t=0) each of which are of length: 0.069 seconds. These smaller peaks correspond to when a part of the repeating signal aligns with another one that is not at the same time step, causing a large amplitude to show up in the auto-correlation.

Figure 3: Zoomed In Auto-correlation of Signal In Noise



Part D of the assignment asked to study the cross-cross correlation between the source signal (with no noise) and the microphone signal (signal in noise) . To accomplish this, the first 1024 samples of the signal in noise signal was taken and cross-correlated with the source signal. Figure 4 shows the results from the cross correlation, which revealed a peak at delay of 0.0037 seconds; the delay represents the time the signal took to travel from the source to the microphone. Consequently, since the speed of sound is 343 m/s, the distance between the source and microphone is 1.28 m (343 * 0.0037)

Figure 4: Cross Correlation of Pulse with Pulse Noise



Concluding Remarks

The first major take away of this assignment is that the relative motion between two objects can cause a perceived change in frequency - known as the Doppler Effect. The second major take away is that you can use auto-correlation to find out information (number and length of hidden signal) about a repeating signal within a larger signal. The third take away is that you can use cross correlation between a source signal and microphone signal to find the distance between the two objects.

tzavelis_hw4

February 28, 2020

[272]:

```
import numpy as np
import pandas as pd
from scipy import signal
import soundfile as sf
import simpleaudio as sa
import sounddevice as sd
from scipy.io import wavfile
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns

plt.rcParams['font.weight'] = 'bold'
plt.rcParams['axes.labelweight'] = 'bold'
plt.rcParams['lines.linewidth'] = 1
plt.rcParams['axes.titleweight'] = 'bold'

class SignalTB:
    """
        My signal toolbox (SignalTB)!
    """

    def __init__(self, x, fs):
        """
        Arguments:
            x: Time Series
            fs: Sample Frequency
        """
        self.fs = fs; # [hz]
        self.x = x    # time domain series
        self.X = None # frequency domain series
        self.sxx = None
        self.gxx = None
        self.gxx_rms_a = None
        self.gxx_linear_a = None
        self.signals = [self.x] #useful container

        self.N = self.x.shape[0]   # number of samples
```

```
        self.L = self.x.index[-1] - self.x.index[0]   # total time of signal [s]
        self.dt = self.L/self.N  # [s]
        self.df = self.fs/self.N

    def get_signals():
        return filter(lambda x: x is not None, [self.X, self.x, self.sxx, self.
↪gxx])

    def my_fft(self):
        """
        Description:
            This method calculates the fft of a time domain signal using␣
↪numpy's fft function and
        adjusting it appropriately to multiplies it by dt.

        Returns:
            Series of frequency domain signal
        """
        freq = np.arange(-np.ceil(self.N/2)+1,
                         np.floor(self.N/2)+1) * self.df
        X = np.fft.fft(a=self.x.values, n=None, axis=-1, norm=None) * self.dt
        X = np.concatenate((X[int(np.floor(self.N/2))+1:],
                            X[0:int(np.floor(self.N/2))+1])) # rearrange the␣
↪frequencies from standard form to sequential. Remember that 1:self.N//2 does␣
↪not grab that second index value
        X = pd.Series(data=X,
                      index=freq,
                      name='X')
        self.X = X
        self.parseval_thrm(self.x,self.X)  #check Parsevals thrm
        self.signals.append(self.X)
        return X

    def my_ifft(self):
        """
        Description:
            This method calculates the ifft of a time domain signal using␣
↪numpy's ifft function and
        adjusting it appropriately to multiplies it by dt.

        Returns:
            Series of frequency domain signal
        """
        t = np.linspace(start=self.x.index[0], stop=self.x.index[-1], num=self.
↪N, endpoint=True)
        X = self.X.values # these are in sequential, non standard form
        X = np.concatenate((X[int(np.ceil(self.N/2))-1:],
```

```
                            X[0:int(np.ceil(self.N/2))-1])) #put the fft values␣
↪in standard form so ifft can accept it
        x = np.fft.ifft(a=X, n=None, axis=-1, norm=None) / self.dt
        self.parseval_thrm(x,X)  #check Parsevals thrm
        x = pd.Series(data=x,
                      index=t,
                      name='x2')
        self.signals.append(x)
        return x

    def parseval_thrm(self, x, X):
        """
        Description:
             Checks to make sure Parseval's Theorem holds between a time domain␣
↪and FFT holds true

        Arguments:
            x: time domain signal
            X: frequency domain signal
        """
        td = round((x**2).sum() * self.dt, 1)
        fd = round((np.absolute(X)**2).sum() * self.df, 1)
        assert td == fd , "Parseval Theorem not satisfied: {} != {}".
↪format(td,fd)

    def sd(self):
        """
        Descrition:
            Spectral Density
        """
        sxx = np.abs(self.X)**2 / self.L; sxx.name = 'S_xx'; #display('sxx',sxx)
        # mean squared check
        X_ms = round(1/self.L * np.sum(np.abs(self.X)**2)*self.df,1)
        sxx_ms = round(np.sum(sxx)*self.df,1)
        assert X_ms == sxx_ms, 'Mean Squared Value Error: {} != {}'.
↪format(X_ms,sxx_ms)
        self.sxx = sxx
        self.signals.append(self.sxx)

        #gxx
        freq = np.arange(0, np.floor(self.N/2)+1) * self.df;␣
↪#display('freq',freq)
        i_zero = int(np.ceil(self.N/2)-1); #display('i_zero',i_zero)

        X = self.sxx.values[i_zero:] * 2 #grab from the center value all the␣
↪way to the end and double it
```

```
        X[0] = X[0]/2
        if self.N%2 == 0: X[-1] = X[-1]/2 #even
        gxx = pd.Series(data=X,
                        index=freq,
                        name='G_xx')

        # mean squared check
        gxx_ms = round(np.sum(gxx) * self.df,1)
        assert sxx_ms == gxx_ms, 'Mean Squared Value Error: {} != {}'.
↪format(sxx_ms,gxx_ms)
        self.gxx = gxx # uts of db
        self.signals.append(self.gxx)
        return self.sxx, self.gxx

    def rms_a(self, n_intervals = 16):
        """
            RMS Averaging for Gxx
        """
        frames=[]
        for i in range(1,n_intervals+1):
            x = self.x.iloc[int((i-1)*self.N/n_intervals):int(i*self.N/
↪n_intervals)]
            m = SignalTB(x=x, fs=self.fs)
            m.my_fft(); #calc fft
            m.sd() #calculate sxx and gxx
            frames.append(m.gxx) #save each gxx for averaging
        assert len(frames) == n_intervals, 'Could not perfectly cut the number␣
↪of samples by the n_interval: {}'.format(n_intervals)
        gxx_rms_a = pd.concat(frames,axis='columns').mean(axis='columns') #␣
↪calculates the mean of at each row (frequency)
        gxx_rms_a.name = 'G_xx_rms_a'
        self.gxx_rms_a = gxx_rms_a
        self.signals.append(gxx_rms_a)
        return gxx_rms_a

    def linear_a(self, n_intervals = 16):
        """
            Linear Averaging for X, then calculation of Gxx
        """
        frames=[]
        for i in range(1,n_intervals+1):
            x = self.x.iloc[int((i-1)*self.N/n_intervals):int(i*self.N/
↪n_intervals)]
            m = SignalTB(x=x, fs=self.fs)
            m.my_fft(); #calc fft
            frames.append(m.X) #save the fft
```

```python
        assert len(frames) == n_intervals, 'Could not perfectly cut the number of samples by the n_interval: {}'.format(n_intervals)
        X_a = pd.concat(frames,axis='columns').mean(axis='columns') #average all the X's at each frequency

        #generate a temporary object so that you can perform computations
        m = SignalTB(x=self.x.iloc[int((i-1)*self.N/n_intervals):int(i*self.N/n_intervals)], fs=self.fs) # the time signal passed in doesn't mean anything, its just necessary to instatiate object
        m.X = X_a #set the new averaged X_a as the frequency domain signal in the temporary object
        m.sd()
        m.gxx.name = 'G_xx_linear_a'
        self.gxx_linear_a = m.gxx
        self.signals.append(m.gxx)
        return m.gxx

    def time_a(self, n_intervals = 16):
        """
            Time Averaging for x, then calculation of Gxx
        """
        frames=[]
        for i in range(1,n_intervals+1):
            x = self.x.iloc[int((i-1)*self.N/n_intervals):int(i*self.N/n_intervals)]
            x = pd.Series(data=x.values,
                          index=self.x.index[0:int(self.N/n_intervals)]) # make sure that all the objects have the same time index. This is important for taking the average and when we instatiate a new object.
            frames.append(x) #save the fft
        assert len(frames) == n_intervals, 'Could not perfectly cut the number of samples by the n_interval: {}'.format(n_intervals)
        x_a = pd.concat(frames,axis='columns').mean(axis='columns'); #display(x_a);

        m = SignalTB(x=x_a, fs=self.fs) #generate a temporary object
        m.my_fft()
        m.sd()
        m.gxx.name = 'G_xx_time_a'
        self.gxx_time_a = m.gxx
        self.signals.append(m.gxx)
        return m.gxx

    def spectrogram(self, n_intervals = 16, overlap = 0.25):
        """
            Spectrogram!
```

```
        """
        p_size = int(np.floor(self.N/n_intervals));#print('psize: {}'.
↪format(p_size));print('n_intervals*p_size: {}'.format(n_intervals*p_size))
        x = self.x.iloc[0:int(n_intervals*p_size)]

        frames=[]
        for i in range(1,n_intervals+1):
            if i == 1:
                f = 0
                l = p_size
            else:
                f = l - int(np.floor(overlap*p_size))
                l = f + p_size
            sig = x.iloc[f:l]
            m = SignalTB(x=sig, fs=self.fs)
            m.my_fft();
            m.sd()
            r = (int((i-1)*self.N/n_intervals) + int(i*self.N/n_intervals))/2
            m.gxx.name = round(r*m.dt,1)  # name the slice at the middle
            frames.append(m.gxx)
        assert len(frames) == n_intervals, 'Could not perfectly cut the number␣
↪of samples by the n_interval: {}'.format(n_intervals)
        gxx_df = pd.concat(frames,axis='columns').sort_index(ascending=False)
        gxx_df.name = 'Gxx_spectro'

        gxx_df.index = gxx_df.index.values.round(decimals=1)
        self.gxx_df = gxx_df
        self.signals.append(self.gxx_df)

        return gxx_df

    def plot_signals(self, xrange=None):
        """
        Description:
            Plots all of the signals in the self.signals container

        Returns:
            Nothing
        """

        for i, sig in enumerate(self.signals):
            if type(sig) != pd.DataFrame:
                if sig.dtype == complex: sig = np.absolute(sig) # ALWAYS the␣
↪magnitude of this in case its a complex number
            fig = plt.figure(figsize=(10,5))
            plt.title(sig.name)
            if sig.name in ['x','x2','time domain signal']:
```

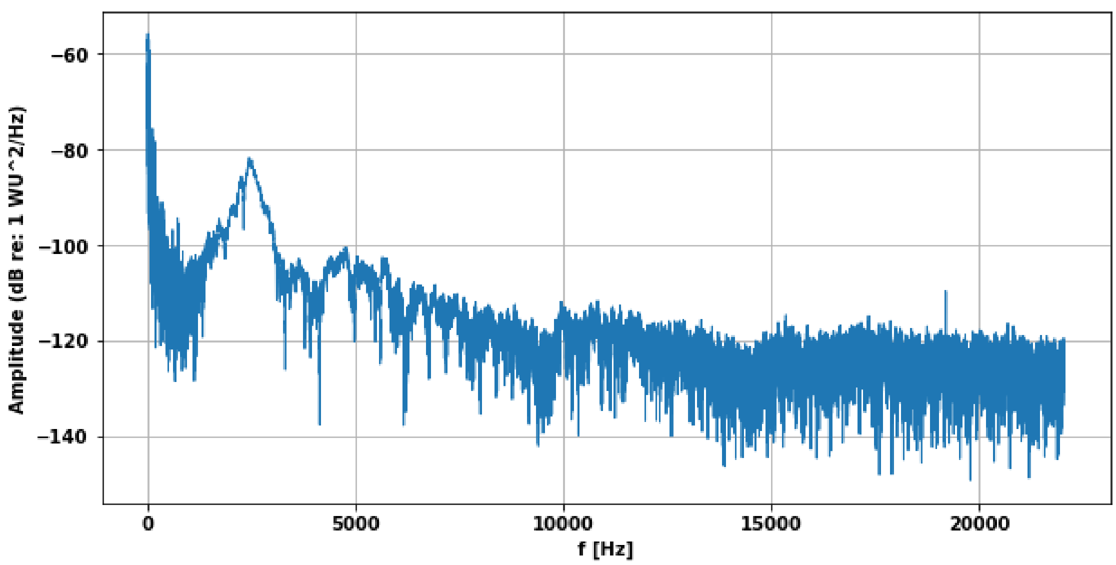
```
                plt.ylabel('x(t)'); plt.xlabel('t [s]')
            elif sig.name in␣
↪['X','S_xx','G_xx','G_xx_rms_a','G_xx_linear_a','G_xx_time_a']:
                sig = 10*np.log10(sig); plt.ylabel('X(f)'); plt.xlabel('f␣
↪[Hz]'); #plt.ylim([-30:])
            elif sig.name in ['Gxx_spectro']:
                sns.heatmap(sig, cmap="jet"); plt.ylabel('f [Hz]'); plt.
↪xlabel('t [s]')
                continue
            if xrange != None:
                sig[xrange[0]:xrange[1]].plot();
            else:
                sig.plot();
            plt.grid()

    #Useful functions to generate signals
    @staticmethod
    def sin(A,f,L,N):
        """
        Arguments:
            A: Amplitude
            f: Frequency of signal [hz]
            L: Total length of time [s]
            N: Number of points

        Returns:
            Series
        """
        t = np.linspace(start=0, stop=L, num=N, endpoint=True, dtype=float)
        return pd.Series(data=A*np.sin(2*np.pi*f*t),
                         index=t,
                         name='x')

    @staticmethod
    def randn_sig(L,N):
        """
        Arguments:
            L : Total length of time [s]
            N : Number of points

        Returns:
            Series
        """
        return pd.Series(data=np.random.randn(N,),
                         index=np.linspace(start=0, stop=L, num=N,␣
↪endpoint=True),
                         name='x')
```

```
    @staticmethod
    def csd(s0,s1):
        """
        Descrition:
            Cross Spectral Density
        """
        #calculate the fft of the objects
        s0.my_fft(); s1.my_fft()

        #sxy
        sxy = np.conj(s0.X)*s1.X / s0.L; sxy.name = 'S_xy';

        #gxy
        freq = np.arange(0, np.floor(s0.N/2)+1) * s0.df; #display('freq',freq)
        i_zero = int(np.ceil(s0.N/2)-1); #display('i_zero',i_zero)

        X = sxy.values[i_zero:] * 2 #grab from the center value all the way to␣
↪the end and double it
        X[0] = X[0]/2
        if s0.N%2 == 0: X[-1] = X[-1]/2 #even
        gxy = pd.Series(data=X,
                        index=freq,
                        name='G_xy')
        return sxy, gxy

    def cross_corr(self,s0,s1):
        """
        Description:
            Cross correlation F^-1(Sxy)
        """
        sxy, gxy = SignalTB.csd(s0,s1)
        X = sxy.values # these are in sequential, non standard form
        X = np.concatenate((X[int(np.ceil(self.N/2))-1:],
                            X[0:int(np.ceil(self.N/2))-1])) #put the fft values␣
↪in standard form so ifft can accept it
        x = np.fft.ifft(a=X, n=None, axis=-1, norm=None) / self.dt
        self.parseval_thrm(x,X)  #check Parsevals thrm
        x = np.concatenate((x[int(np.floor(self.N/2))+1:],
                            x[0:int(np.floor(self.N/2))+1])) #put the fft␣
↪values in standard form so ifft can accept it
        t = np.arange(-np.ceil(self.N/2)+1,np.floor(self.N/2)+1) * self.dt

        cross_corr = pd.Series(data=x,
                               index=t,
                               name='Cross Correlation')
        return cross_corr, sxy, gxy
```

1 HW 4 - Excercise

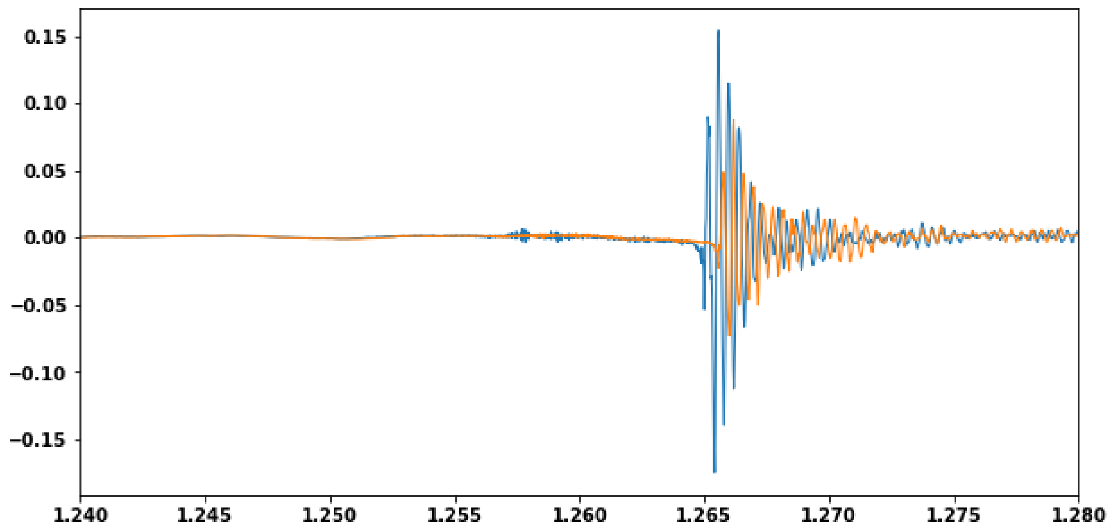
[34]:
```
data, fs = sf.read('./hw4_excercise_files/cross_corr_example.wav');
N = len(data); display(N)
L = N/fs;
df = pd.DataFrame(data=data,
                  index=np.linspace(start=0, stop=L, num=N, endpoint=True))

s0 = SignalTB(x=df[1], fs=fs)
s1 = SignalTB(x=df[0], fs=fs)
cross_corr, sxy, gxy = s0.cross_corr(s0,s1)

fig = plt.figure(figsize=(10,5))
plt.plot(10*np.log10(np.abs(gxy)));
plt.ylabel('Amplitude (dB re: 1 WU^2/Hz)'); plt.xlabel('f [Hz]');#plt.
 ↪xlim([250,260])
plt.grid()
#fig.savefig('./plots/hw3_2_gxx_averages.png', dpi=300, bbox_inches='tight');
```

132300



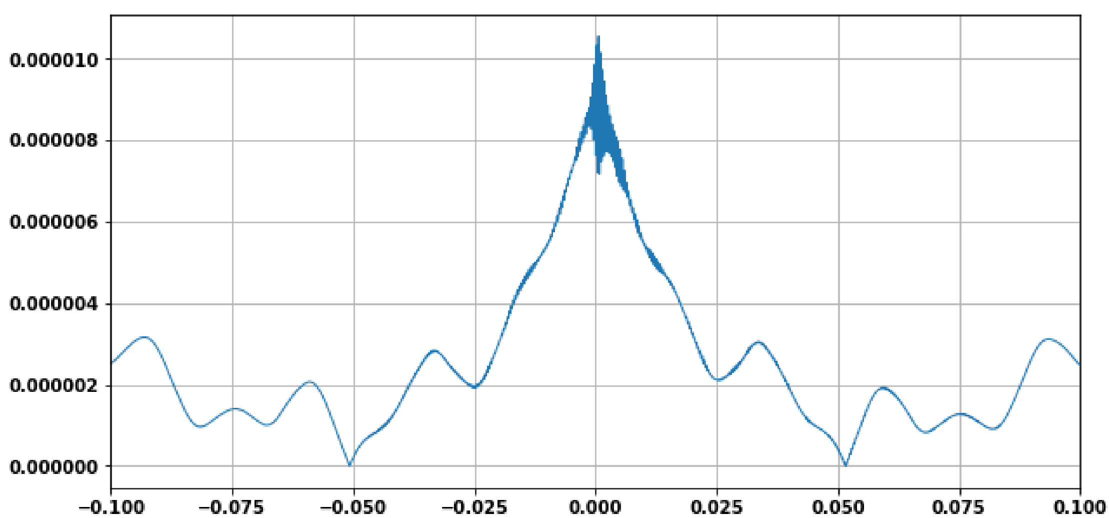
[35]:
```
plt.figure(figsize=(10,5))
s0.x.plot()
s1.x.plot()
plt.xlim((1.24,1.28))
```

[35]: (1.24, 1.28)



[37]:
```
plt.figure(figsize=(10,5))
#plt.plot(10*np.log10(np.abs((s0.cross_corr(sxy)))));plt.grid(); plt.xlim([-.
↪025,.025])
plt.plot(np.abs(cross_corr));plt.grid(); plt.xlim([-.1,.1]);
np.abs(cross_corr).idxmax()
```

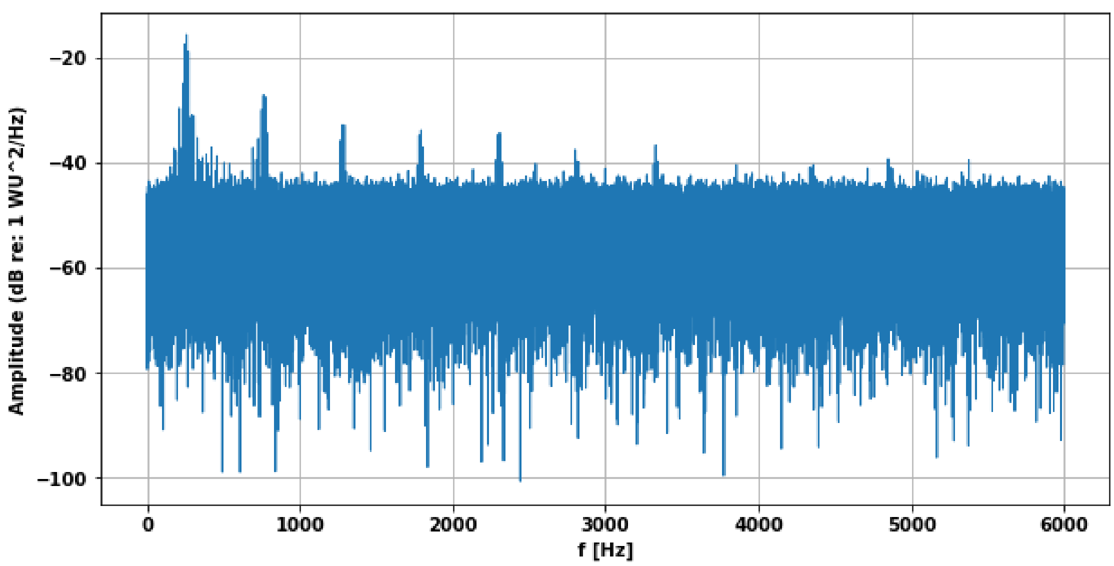
[37]: 0.0005895691609977325



2 HW 4

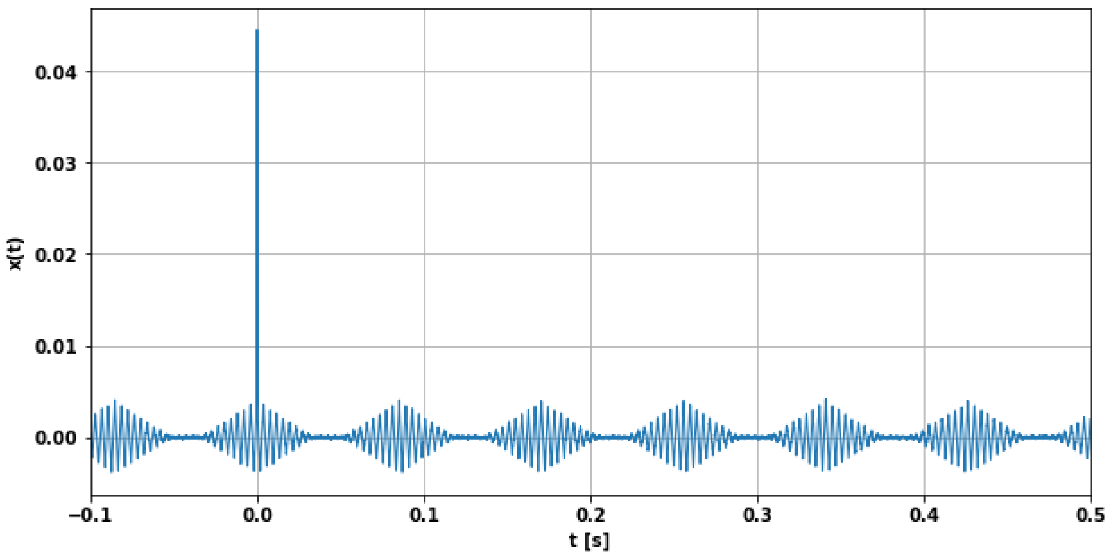
[291]:
```
data, fs = sf.read('./hw3_files/HW2_pulsenoise.wav');
N = len(data); print(N)
L = N/fs;
s = pd.Series(data=data,
              index=np.linspace(start=0, stop=L, num=N, endpoint=True))

s0 = SignalTB(x=s, fs=fs)
s1 = SignalTB(x=s, fs=fs)
cross_corr, sxy, gxy = s0.cross_corr(s0,s1)

fig = plt.figure(figsize=(10,5))
plt.plot(10*np.log10(np.abs(gxy)));
plt.ylabel('Amplitude (dB re: 1 WU^2/Hz)'); plt.xlabel('f [Hz]');#plt.
 ↪xlim([250,260])
plt.grid()
```

262196



[297]:
```
fig = plt.figure(figsize=(10,5))
plt.plot(cross_corr);plt.grid(); plt.xlim([-0.1,.5]); #plt.ylim([0,0.0037])
plt.ylabel('x(t)'); plt.xlabel('t [s]')
fig.savefig('./plots/hw4_crosscorr.png', dpi=300, bbox_inches='tight');
cross_corr.idxmax()
```

[297]: 0.0



[293]:
```
p_length = 0.12-0.05 #by looking at the plot
n_intervals = len(signal.find_peaks(cross_corr, distance=t_len*s0.fs)[0]) - 1␣
↪#subtract one to not count center one
print('pulse length: {}\nn_intervals: {}'.format(p_length,n_intervals))
```

```
pulse length: 0.06999999999999999
n_intervals: 256
```

[137]:
```
s0.x.iloc[:len(data)]
```

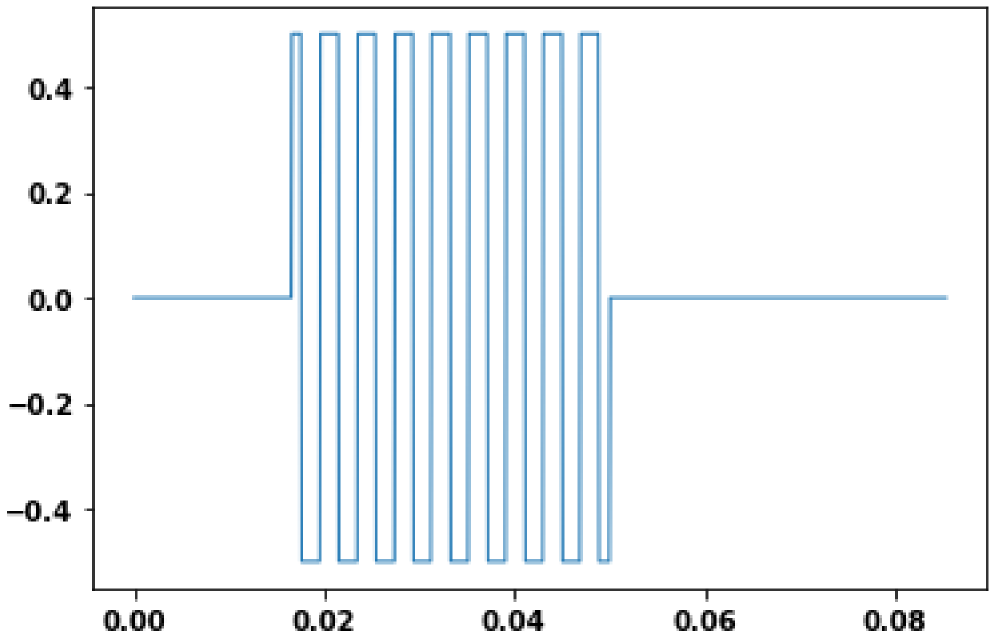
[137]:
```
0.000000     0.116852
0.000083    -0.080841
0.000167     0.509888
0.000250     0.027252
0.000333     0.050537
               …
21.849333   -0.017609
21.849417    0.222748
21.849500    0.124054
21.849583   -0.018585
21.849667    0.153748
Length: 262196, dtype: float64
```

[298]:
```
data, fs = sf.read('./hw3_files/HW2_pulse.wav');
N = len(data); print(N)
L = N/fs; print(L)
s = pd.Series(data=data,
              index=np.linspace(start=0,stop=L,num=N,endpoint=True))
```

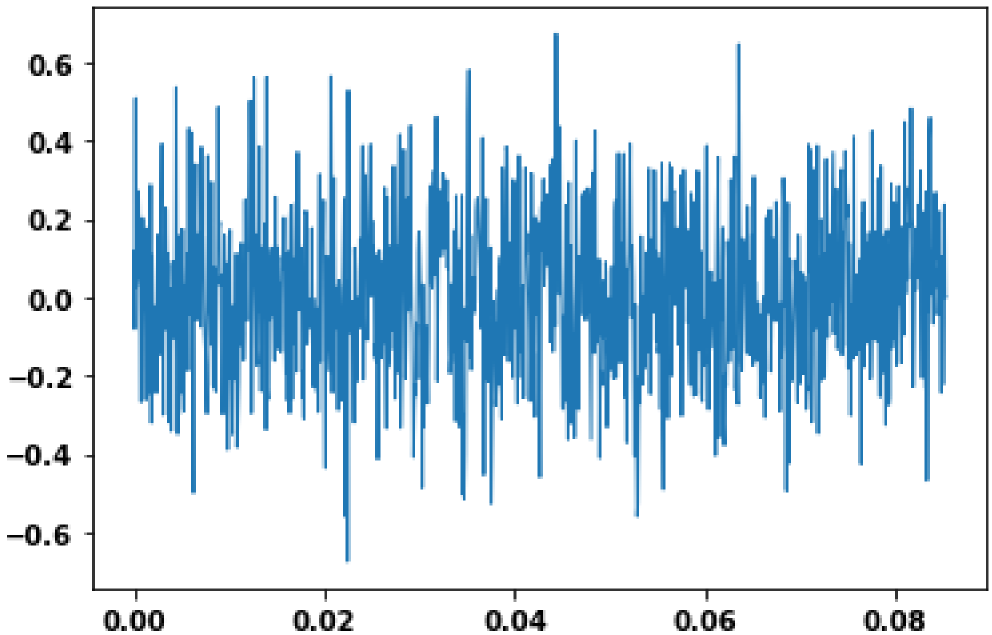
```
s3 = SignalTB(x=s, fs=fs)
plt.plot(s3.x)
```

```
1024
0.08533333333333333
```

[298]: [<matplotlib.lines.Line2D at 0x7f0fd8431400>]



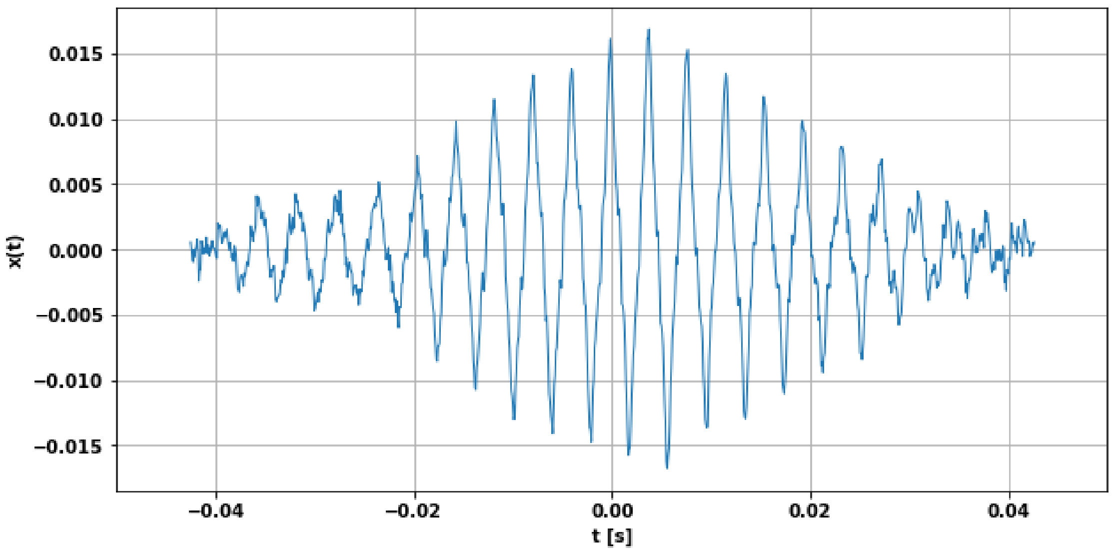
[144]:
```
s0_s = SignalTB(x=s0.x.iloc[0:len(data)], fs=fs)
plt.plot(s0.x.iloc[0:len(data)]);
```



[300]:
```
cross_corr, sxy, gxy = s0_s.cross_corr(s3,s0_s)
```

[303]:
```
fig = plt.figure(figsize=(10,5))
plt.plot(cross_corr);plt.grid(); plt.xlim([-0.05,0.05]); #plt.ylim([0,0.0037])
plt.ylabel('x(t)'); plt.xlabel('t [s]')
fig.savefig('./plots/hw4_crosscorr.png', dpi=300, bbox_inches='tight');

cross_corr.idxmax() * 343
```

[303]: 1.2849987973936088

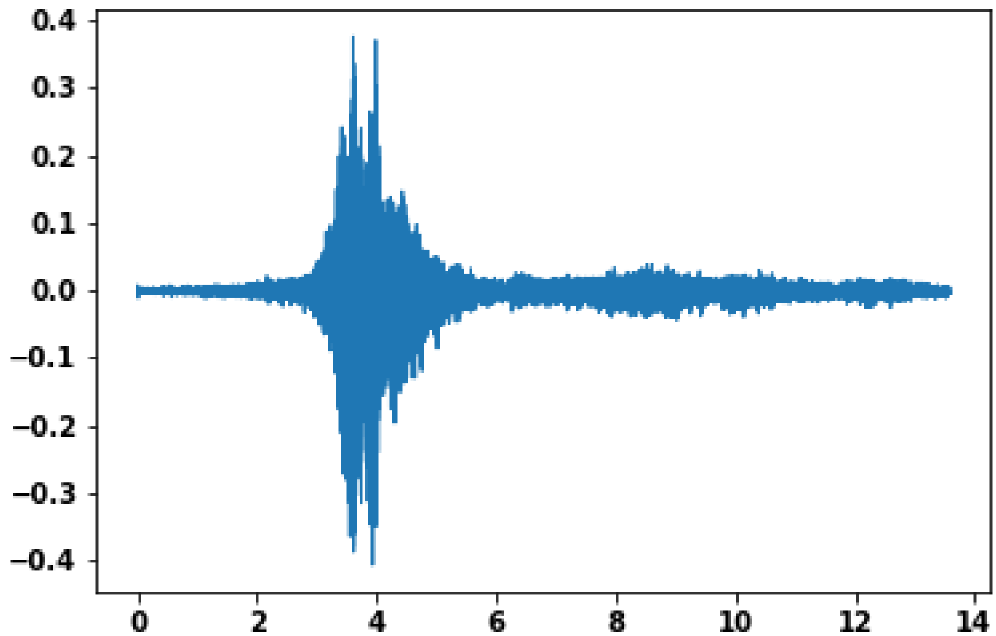


3 spectrogram

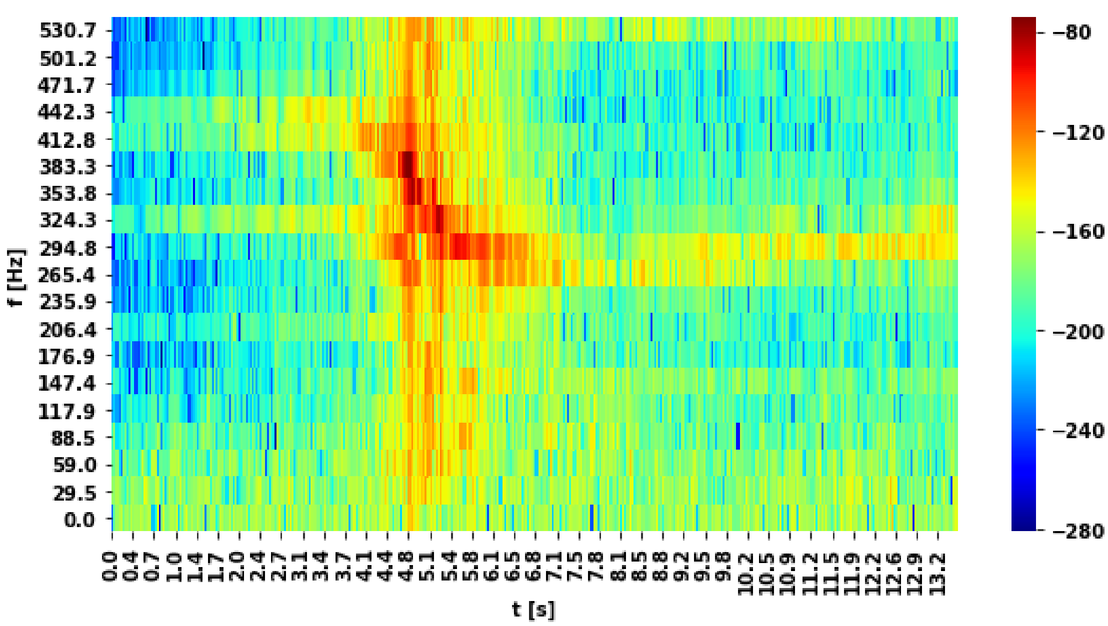
[306]:
```python
data, fs = sf.read('./hw4/HW4_spect.wav');
N = len(data); print(N)
L = N/fs; print(L)
s = pd.Series(data=data,
              index=np.linspace(start=0,stop=L,num=N,endpoint=True))
s = SignalTB(x=s, fs=fs)
plt.plot(s.x)
```

```
163011
13.58425
```

[306]:
```
[<matplotlib.lines.Line2D at 0x7f0fd8948710>]
```




```
[340]: gxx_spectro = s.spectrogram(n_intervals = 400,overlap = 0.25)
fig = plt.figure(figsize=(10,5))
sns.heatmap(10*np.log(gxx_spectro.iloc[185:]),cmap="jet"); plt.ylabel('f [Hz]');
 ↪ plt.xlabel('t [s]');# plt.set_label('Amplitude (dB re: 1 WU^2/Hz)')
fig.savefig('./plots/spectrogram.png', dpi=300, bbox_inches='tight');
```



[341]:

```python
c = 353
vl= 294
vh= 442
v = 343 * (vl+vh)/(vh-vl)
v
```

[341]: 1705.7297297297298

[343]:

```python
vl= 383
vh= 442
f0 = 295
df = vh-vl
df*c/f0
```

[343]: 70.6