Zach Tzavelis

ME465 - Sound and Space

02/13/2020

HW2: RMS Averaging Effect on Power Spectral Density

Exercise 1.3.1

The task was to analyze a 16 second signal (fs: 1024 Hz) with a frequency of 120 Hz hidden in noise (Figure 1). The task was focused on highlighting the impact of root mean squared (RMS) averaging of the power spectral density (Gxx) function. To compute the RMS of Gxx, the time domain signal was sliced into 16 one-second segments and Gxx was computed on each of them, before averaging all of them together. Figure 2 depicts the difference between the spectral density function of the entire signal verses the RMS averaged version (Gxx_a).

Figure 1: 120 Hz Signal Hidden in Noise

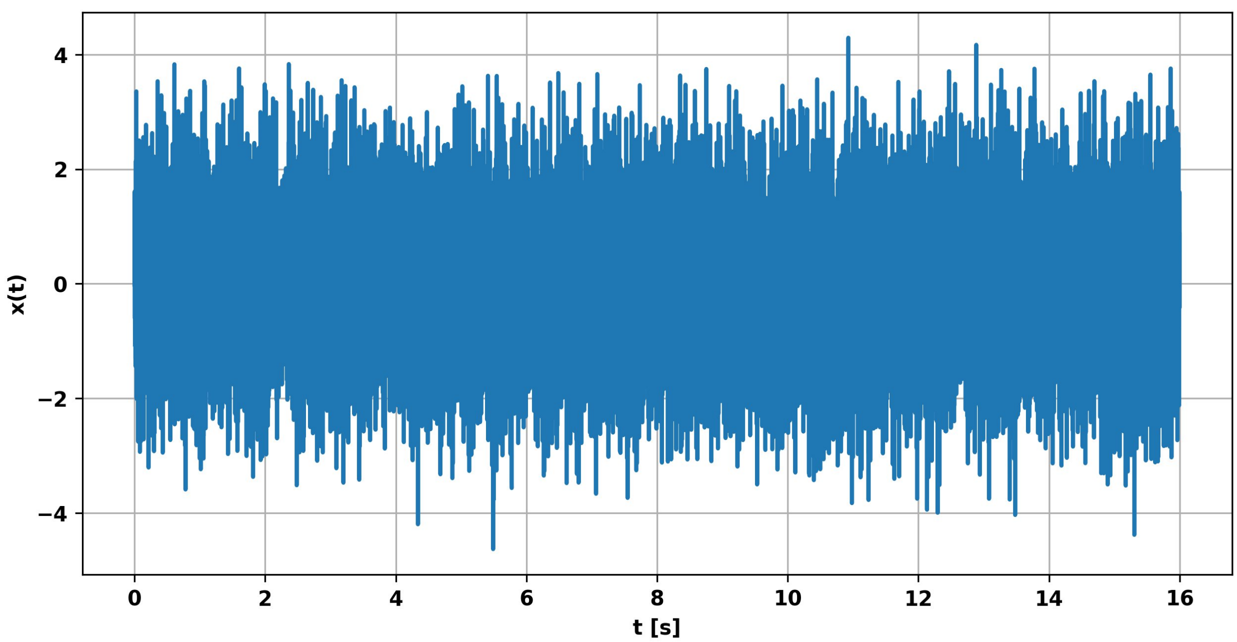
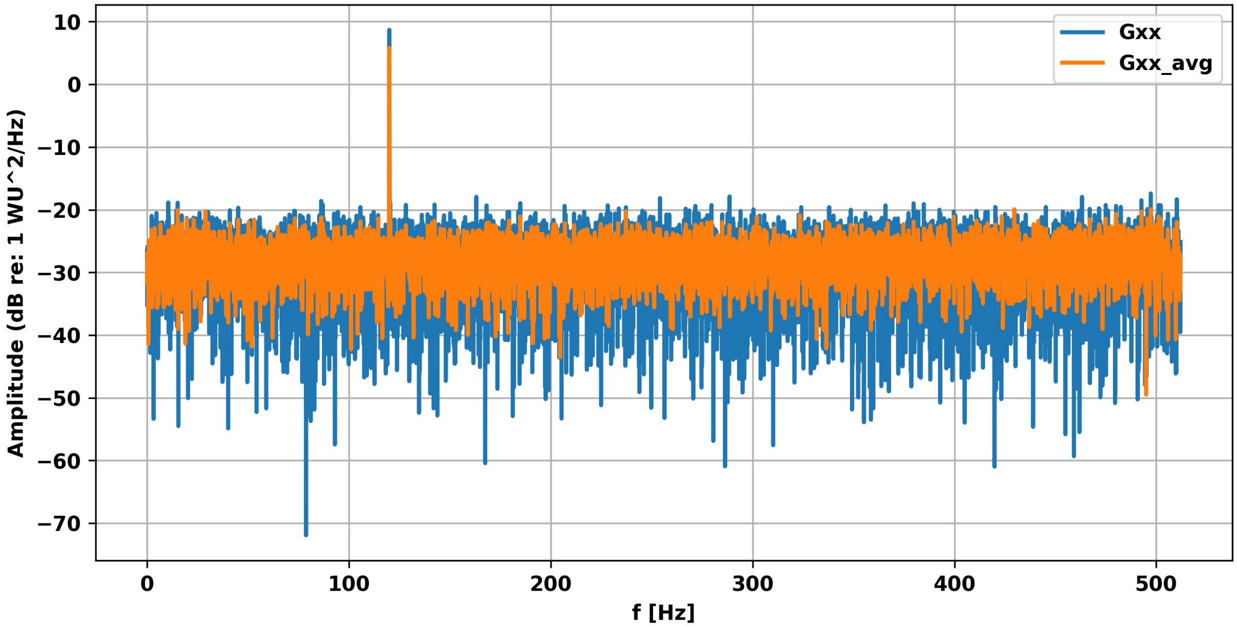


Figure 2: Power Spectral Density with and without RMS Averaging



The first notable result of RMS averaging Gxx is the reduced variability across the frequency spectrum — the amplitude of the signal at 120 Hz also decreases; the average amplitude value stays about the same, however. The reduction in amplitude across the frequency spectrum makes sense since the magnitude of Gxx_a is the simple average of 16 'smaller' Gxx signals; such an average would get rid of some of the variability in the amplitude of the 16 Gxxs.

Concluding Remarks

RMS averaging of Gxx seems like a good solution for getting rid of some amplitude noise, but only when that noise is relatively small compared to the signal. Increasing the amount of noise could bury the original signal, and performing an RMS on Gxx could result in missing the important signal at 120 Hz (in this case). It may be more useful to perform something like linear averaging of X since one can then take advantage of phase and magnitude information instead of just the magnitude — like in Gxx RMS averaging performed in this excercise.

tzavelis_hw2

February 12, 2020

[35]:
```
import numpy as np
import pandas as pd
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
from IPython.display import HTML, display
#import plotly

plt.rcParams['font.weight'] = 'bold'
plt.rcParams['axes.labelweight'] = 'bold'
plt.rcParams['lines.linewidth'] = 2
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_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[self.N//2+1:],
                            X[0: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:],
```

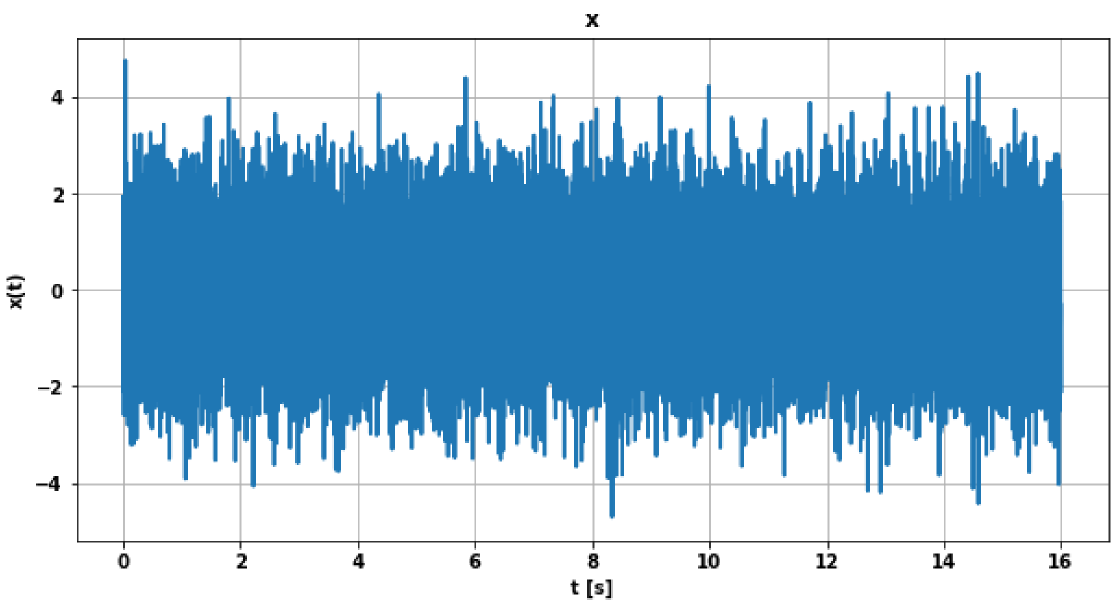
```
                           X[0:int(np.ceil(self.N/2))-1])) #put the fft values␣
↪in standard form so ifft can accept it
        x = np.fft.ifft(a=X, n=None, axis=-1, norm=None) / self.dt
        self.parseval_thrm(x,X)  #check Parsevals thrm
        x = pd.Series(data=x,
                      index=t,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

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

        X = self.sxx.values[i_zero:] * 2 #grab from the center value all the␣
↪way to the end an double it
        X[0] = X[0]/2
        if self.N%2 == 0: X[-1] = X[-1]/2 #even
```

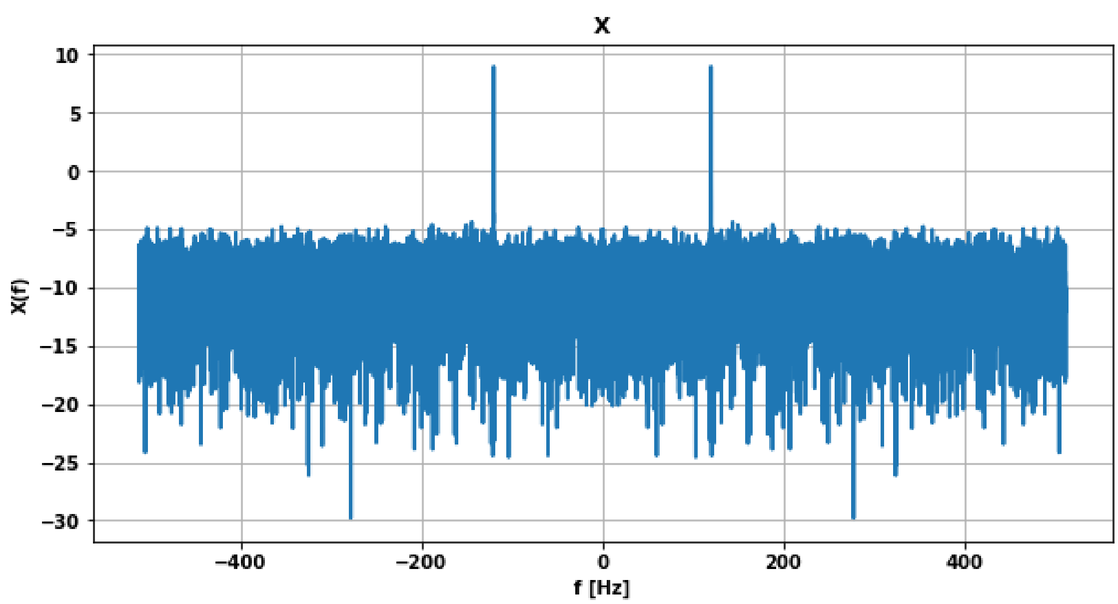
```
        gxx = pd.Series(data=X,
                  index=freq,
                  name='G_xx')

        # 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 spectrogram(self, n_intervals = 16):
        """
            Spectrogram!
        """
        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();
            m.sd()
            m.gxx.name = round(self.x.index[(int((i-1)*self.N/n_intervals) +␣
 ↪int(i*self.N/n_intervals))/2],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)
#          display(HTML(gxx_df.to_html()))
        gxx_df.name = 'Gxx_spectro'
        self.gxx_df = gxx_df
        self.signals.append(self.gxx_df)
        return gxx_df

    def rms_a_gxx(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();
            m.sd()
```

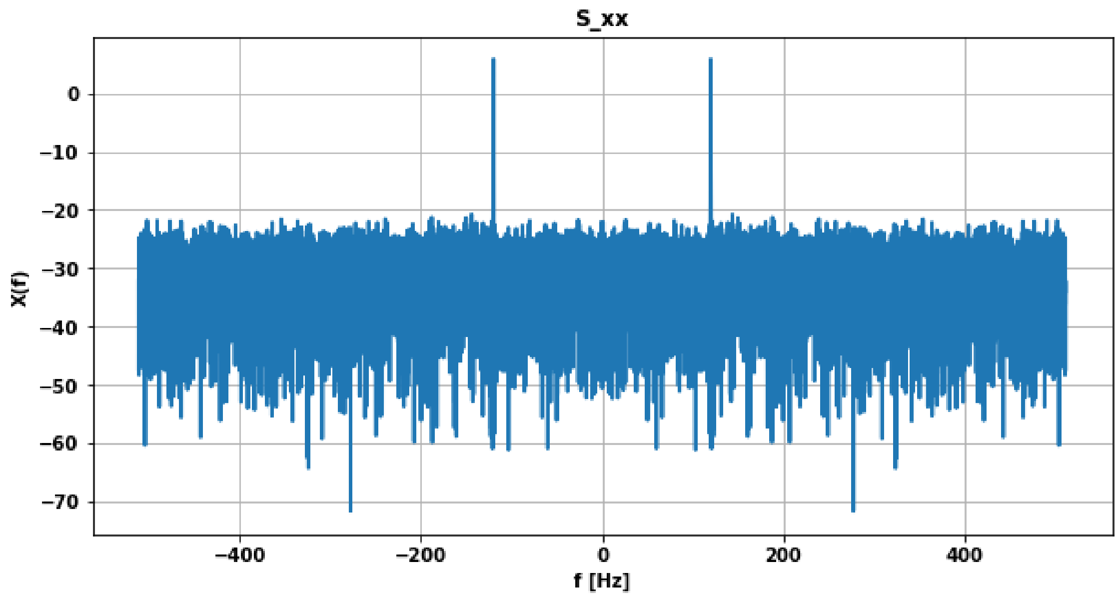
```
            frames.append(m.gxx)
        assert len(frames) == n_intervals, 'Could not perfectly cut the number␣
↪of samples by the n_interval: {}'.format(n_intervals)
        gxx_a = pd.concat(frames,axis='columns').mean(axis='columns')
        gxx_a.name = 'G_xx_a'
        self.gxx_a = gxx_a
        self.signals.append(gxx_a)
        return gxx_a

    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_a']:
                sig = 10*np.log10(sig); plt.ylabel('X(f)'); plt.xlabel('f [Hz]')
            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
```

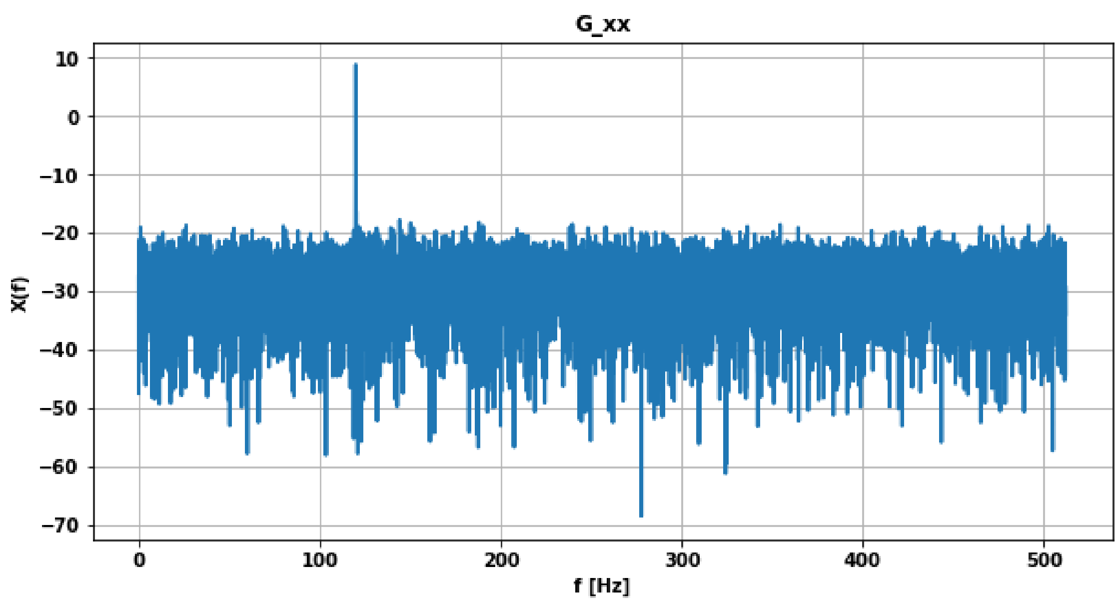
```
        Returns:
            Series
        """
        t = np.linspace(start=0, stop=L, num=N, endpoint=True, dtype=float)
        return pd.Series(data=A*np.sin(2*np.pi*f*t),
                         index=t,
                         name='x')

    @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')
```

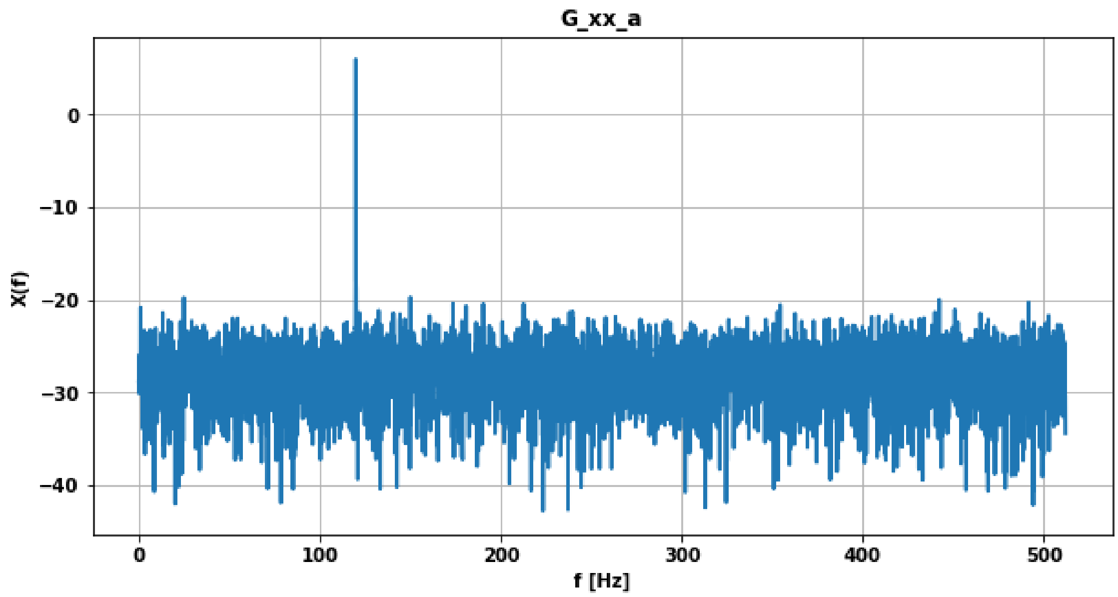
[44]:
```
#1.3.1
L = 16 # [s]
fs = 1024 #[hz]
N = int(L/(1/fs));  # generate the number of points based on the sampling␣
 ↪frequency which is higher than the actual signal frequency
f_sin = 120;

x = SignalTB.sin(A = 1, f = f_sin, N = N, L = L) + SignalTB.randn_sig(L=L,N=N)

s = SignalTB(x=x, fs=fs)
s.my_fft();
s.sd()
s.rms_a_gxx(n_intervals = 2)
#s.spectrogram(n_intervals = 2);
s.plot_signals();
```


x
x(t)
t [s]
4
2
0
−2
−4
0
2
4
6
8
10
12
14
16

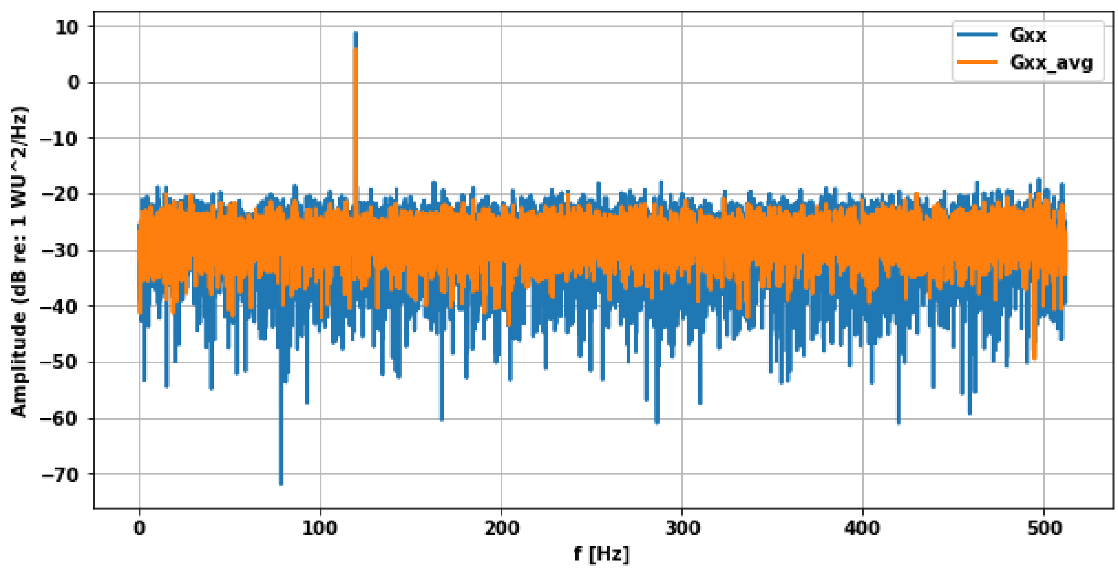

X
X(f)
f [Hz]
10
5
0
−5
−10
−15
−20
−25
−30
−400
−200
0
200
400


S_xx
X(f)
f [Hz]
0
−10
−20
−30
−40
−50
−60
−70
−400
−200
0
200
400


G_xx
X(f)
f [Hz]
10
0
−10
−20
−30
−40
−50
−60
−70
0
100
200
300
400
500



[41]:
```
fig = plt.figure(figsize=(10,5))
for myplot in [s.gxx,s.gxx_a]:
    plt.plot(10*np.log10(np.abs(myplot)));
plt.ylabel('Amplitude (dB re: 1 WU^2/Hz)'); plt.xlabel('f [Hz]');
plt.legend(['Gxx','Gxx_avg'])
plt.grid()
fig.savefig('./plots/gxx_a.png', dpi=300, bbox_inches='tight');
```



[43]:
```
fig = plt.figure(figsize=(10,5))
for myplot in [s.x]:
    plt.plot(myplot);
plt.ylabel('x(t)'); plt.xlabel('t [s]')
plt.grid()
fig.savefig('./plots/time_hw2.png', dpi=300, bbox_inches='tight');
```