

## tzavelis\_hw5

March 6, 2020

```
[2]: 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
    SignalTB.parseval_thrm(self.x, self.X, self.df, self.dt) #check Parsevals
→thrm

    self.signals.append(self.X)
    return X

@staticmethod
def my_ifft(X, N, df, dt):
    """
    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=0, stop=N*dt, num=N, endpoint=True)
    X = X.values # these are in sequential, non standard form

```

```

X = np.concatenate((X[int(np.ceil(N/2))-1:],
                    X[0:int(np.ceil(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) / dt
SignalTB.parseval_thrm(x,X,df,dt) #check Parsevals thrm
return pd.Series(data=x,
                  index=t,
                  name='x2')

@staticmethod
def parseval_thrm(x, X, df, dt):
    """
    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
    """
    x = np.absolute(x) #get rid of complex numbers to do calc
    X = np.absolute(X)
    td = round((x**2).sum() * dt, 1)
    fd = round((X**2).sum() * df, 1)
    assert td == fd, "Parseval Theorem not satisfied: {} != {}".
→format(td,fd)

def sd(self):
    """
    Description:
        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
        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 instantiate 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 instantiate 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,2) # 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 impulse(N,L,Tp):
    x = np.zeros(N)
    x[0:int(N*Tp/L)] = 1
    x = pd.Series(data=x,
                    index=np.linspace(start=0, stop=L, num=N, endpoint=True,
→ dtype=float),
                    name='x')
    return x

@staticmethod
def cw_sin(A,f,N,L,Tp):
    """
    Arguments:
        A: Amplitude
        f: Frequency of signal [hz]
        L: Total length of time [s]
        N: Number of points
        Tp: Pulse length

    Returns:
        Series
    """
    t = np.linspace(start=0, stop=L, num=N, endpoint=True, dtype=float)
    x = np.zeros(N)

    t_s = t[0:int(N*Tp/L)];
    x[0:int(N*Tp/L)] = A*np.sin(2*np.pi*f*t_s)

```

```

        return pd.Series(data=x,
                           index=t,
                           name='x')

    @staticmethod
    def lin_sin_s(A,N,L,Tp,f1,f2):
        """
        Arguments:
            A: Amplitude
            f: Frequency of signal [hz]
            L: Total length of time [s]
            N: Number of points
            Tp: Pulse length

        Returns:
            Series
        """
        t = np.linspace(start=0, stop=L, num=N, endpoint=True, dtype=float)
        x = np.zeros(N)

        t_s = t[0:int(N*Tp/L)];
        phi = 2*np.pi*( (f2-f1)/(2*Tp)*t_s**2+f1*t_s)
        x[0:int(N*Tp/L)] = A*np.sin(phi)

        return pd.Series(data=x,
                           index=t,
                           name='x')

    @staticmethod
    def log_sin_s(A,N,L,Tp,f1,f2):
        """
        Arguments:
            A: Amplitude
            L: Total length of time [s]
            N: Number of points
            Tp: Pulse length

        Returns:
            Series
        """
        t = np.linspace(start=0, stop=L, num=N, endpoint=True, dtype=float)
        x = np.zeros(N)

        t_s = t[0:int(N*Tp/L)];
        phi = 2*np.pi*f1*(f2/f1)**(t_s/Tp)*Tp/(np.log(f2/f1))-2*np.pi*f1*Tp/np.
        ↪ log(f2/f1)
        x[0:int(N*Tp/L)] = A*np.sin(phi)

```



```

x = pd.Series(data=x,
               index=t,
               name='x')

return x

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

    Returns:
        Series
    """
    fs = N/L
    df = fs/N

    p = np.random.uniform(0,2*np.pi,int(N/2)-1) #random reals
    Z = 1*np.sin(p) + 1*np.cos(p)*1j #random complex numbers with magnitude 1
    ↪ of 1

    Z = np.concatenate((Z, np.array([0]), np.flip(np.conj(Z))))

    if N%2 == 0: Z = np.concatenate((Z, np.array([0])))

    freq = np.arange(-np.ceil(N/2)+1, np.floor(N/2)+1) * df
    X = pd.Series(data=Z,index=freq); #construct the full freq response
    x = SignalTB.my_ifft(X=X, N=N, df=df, dt=dt) # get the full time noise

    y = np.zeros(N,dtype=complex)
    y[0:int(N*Tp/L)] = x.values[0:int(N*Tp/L)] # take only a subset of the
    ↪ noisy signal and put it in front of all the zeros

    return pd.Series(data=np.real(y),
                     index=x.index,
                     name='x')

@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):
    """
    Description:
        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

```

```

@staticmethod
def cross_corr(s0,s1):
    """
    Description:
        Cross correlation  $F^{-1}(S_{xy})$ 
    """
    sxy, gxy = SignalTB.csd(s0,s1)
    X = sxy.values # these are in sequential, non standard form
    X = np.concatenate((X[int(np.ceil(s0.N/2))-1:],
                        X[0:int(np.ceil(s0.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) / s0.dt
    SignalTB.parseval_thrm(x,X, s0.df, s0.dt) #check Parsevals thrm
    x = np.concatenate((x[int(np.floor(s0.N/2))+1:],
                        x[0:int(np.floor(s0.N/2))+1])) #put the fft values
    → in standard form so ifft can accept it
    t = np.arange(-np.ceil(s0.N/2)+1,np.floor(s0.N/2)+1) * s0.dt

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

    return cross_corr, sxy, gxy

```

```

[3]: # QUESTION 1
L = 10 # [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 = 400;
df = fs/N
dt = L/N

Tp = 3
f1 = 1
f2 = 200

x, Z = SignalTB.white_noise(N,L,Tp)
#plt.plot(np.angle(Z))
plt.plot(x)

```

```

    □
    → -----
ValueError                                Traceback (most recent call
    → last)

```

```

<ipython-input-3-0afc105afb8d> in <module>
    11 f2 = 200
    12
---> 13 x, Z = SignalTB.white_noise(N,L,Tp)
    14 #plt.plot(np.angle(Z))
    15 plt.plot(x)

```

ValueError: too many values to unpack (expected 2)

```

[11]: # QUESTION 1
L = 10 # [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 = 400;
df = fs/N
dt = L/N

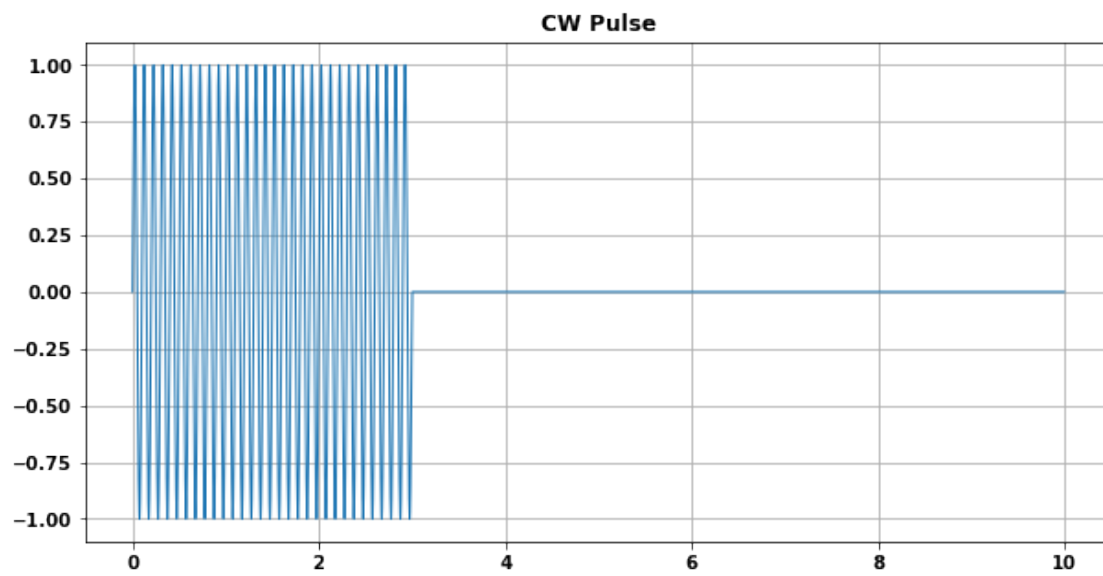
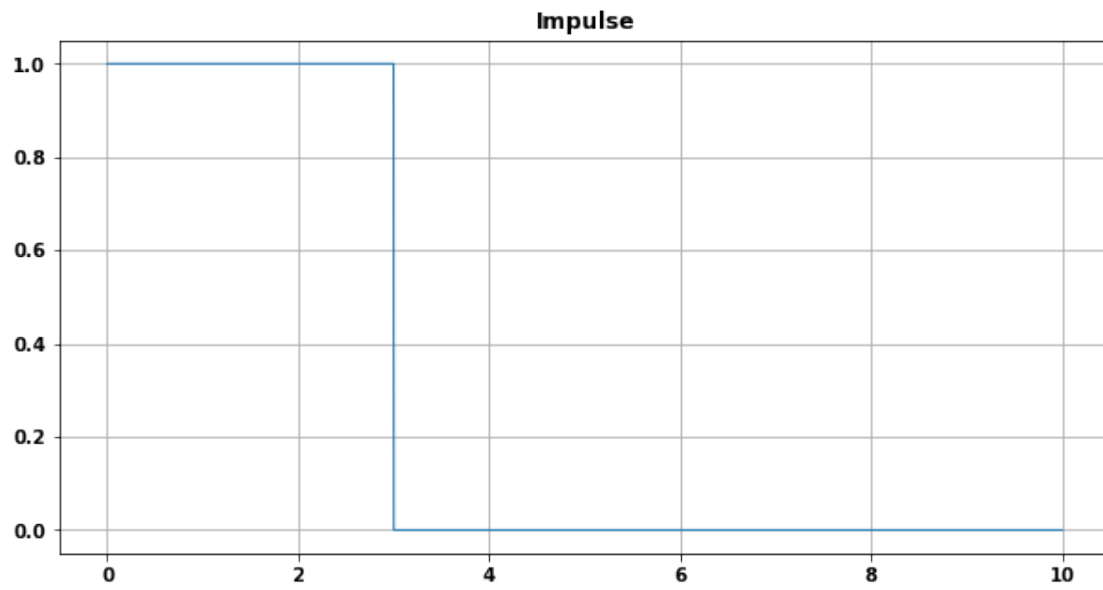
Tp = 3
f1 = 1
f2 = 200

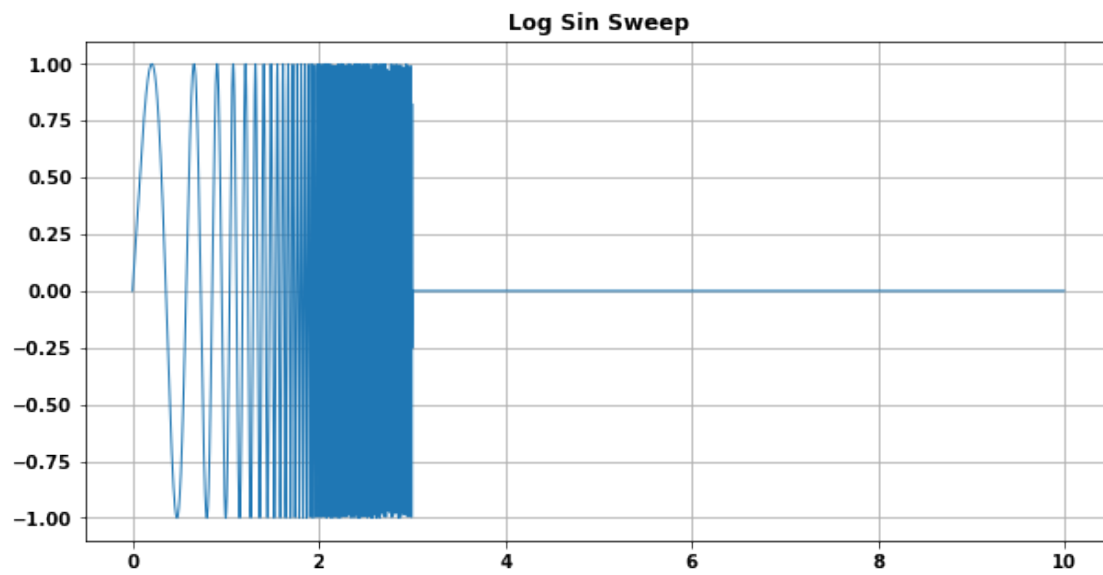
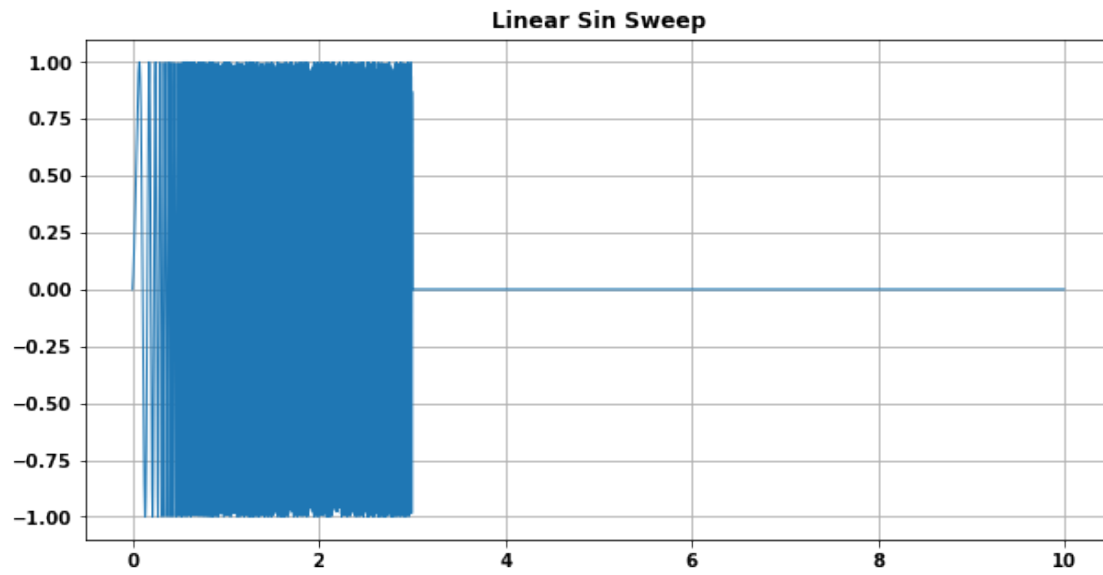
sigs = [SignalTB.impulse(N=N,L=L,Tp=Tp),
        SignalTB.cw_sin(A=1,f=10, N=N,L=L,Tp=Tp),
        SignalTB.lin_sin_s(A=1,N=N,L=L,Tp=Tp,f1=f1,f2=f2),
        SignalTB.log_sin_s(A=1,N=N,L=L,Tp=Tp,f1=f1,f2=f2),
        SignalTB.white_noise(N=N,L=L,Tp=Tp)]

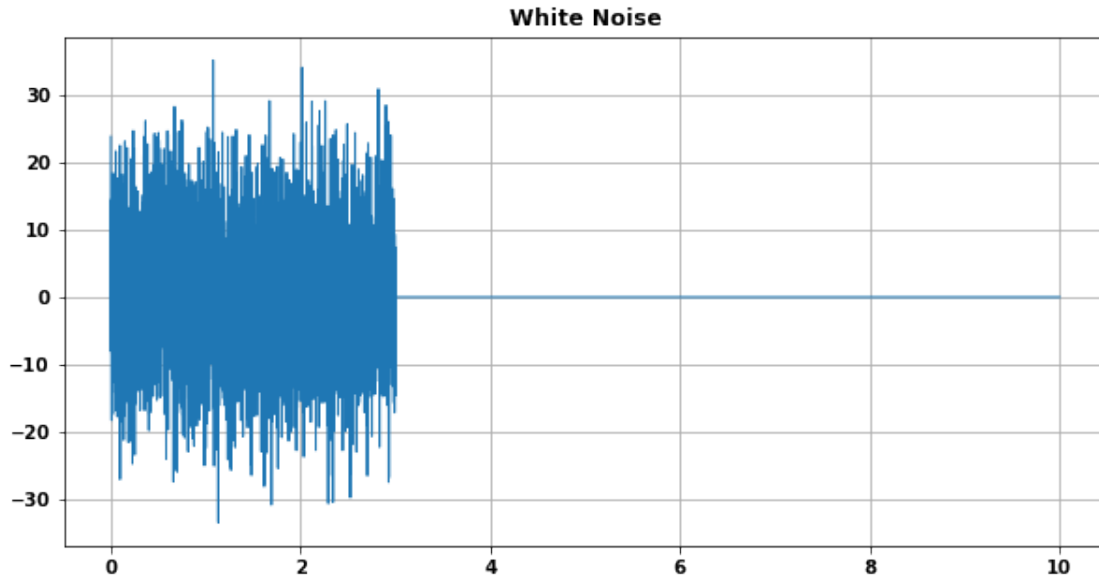
signals = [SignalTB(x=x, fs=fs) for x in sigs]

titles = ['Impulse', 'CW Pulse', 'Linear Sin Sweep', 'Log Sin Sweep', 'White
    ↪ Noise']
#q1
for i, s in enumerate(signals):
    plt.figure(figsize=(10,5))
    plt.title(titles[i])
    s.x.plot(); plt.grid()
    #s.my_fft()
    #plt.plot(np.abs(SignalTB.my_ifft(s.X, N, df, dt)), '--')

```







```
[37]: # QUESTION 1
L = 10 # [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 = 400;
df = fs/N
dt = L/N

Tp = 3
f1 = 1
f2 = 200

x = SignalTB.impulse(L=L,N=N,Tp=Tp)
s = SignalTB(x=x,fs=fs)
s.my_fft() #must calc fft to do sd
#power spectral density
sxx, gxx = s.sd()

fig = plt.figure()
plt.plot(10*np.log10(np.abs(gxx)));

Tp = 6
f1 = 1
f2 = 200

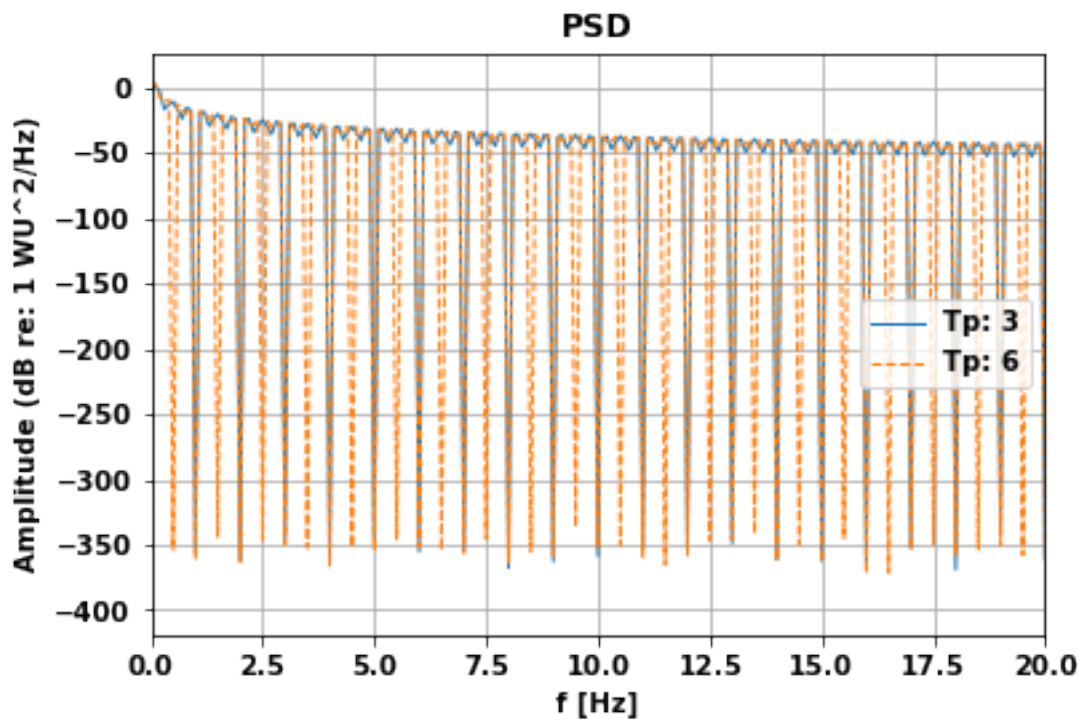
x = SignalTB.impulse(L=L,N=N,Tp=Tp)
```

```

s = SignalTB(x=x,fs=fs)
s.my_fft() #must calc fft to do sd
#power spectral density
sxx, gxx = s.sd()

plt.plot(10*np.log10(np.abs(gxx)), '--'); plt.xlim([0,20])
plt.xlim([0,20])
plt.title('PSD'); plt.ylabel('Amplitude (dB re: 1 WU2/Hz)'); plt.xlabel('f⊥
→[Hz]'); plt.grid(True)
plt.legend(['Tp: {}'.format(times[0]), 'Tp: {}'.format(times[1])])
fig.savefig('./plots/hw5_'+'zoomed_impulse'+'.png', dpi=300,⊥
→bbox_inches='tight')

```



```

[12]: #QUESTION 2
titles = ['Impulse', 'CW Pulse', 'Linear Sin Sweep', 'Log Sin Sweep', 'White⊥
→Noise']
linestyles = ['-', '--']
for i,t in enumerate(titles):
    fig, axs = plt.subplots(1,2,figsize=(15,5)); plt.grid()
    fig.suptitle(t)

    times = [Tp, 2*Tp]
    for j, length in enumerate(times):

```



```

sigs = [SignalTB.impulse(L=L,N=N,Tp=Tp),
        SignalTB.cw_sin(A=1,f=f2, N=N,L=L,Tp=length),
        SignalTB.lin_sin_s(A=1,N=N,L=L,Tp=length,f1=f1,f2=f2),
        SignalTB.log_sin_s(A=1,N=N,L=L,Tp=length,f1=f1,f2=f2),
        SignalTB.white_noise(N=N,L=L,Tp=Tp),
        ]
signals = [SignalTB(x=x, fs=fs) for x in sigs]

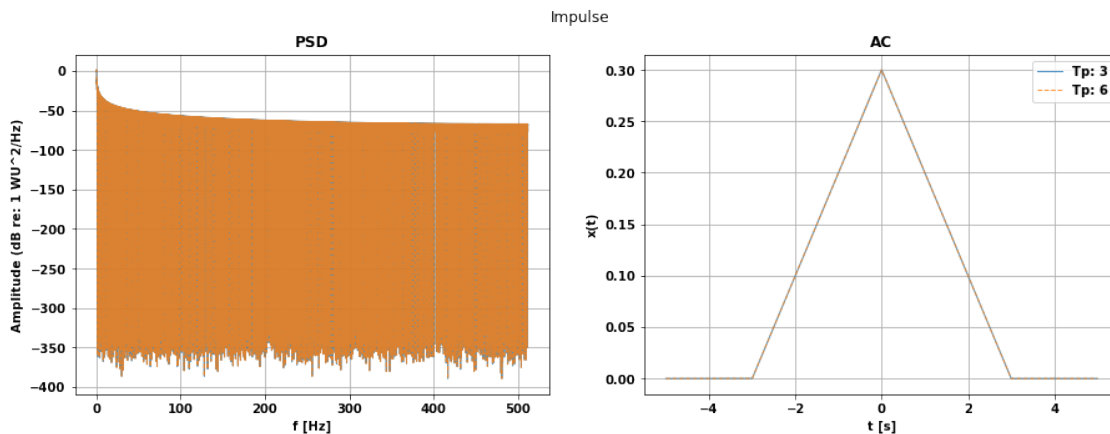
s = signals[i] #select the actual signal we care about
s.my_fft() #must calc fft to do sd

#power spectral density
sxx, gxx = s.sd()
axs[0].plot(10*np.log10(np.abs(gxx)),alpha=0.8,linestyle=linestyles[j]);
axs[0].set_title('PSD'); axs[0].set_ylabel('Amplitude (dB re: 1 WU^2/
→Hz)'); axs[0].set_xlabel('f [Hz]'); axs[0].grid(True)

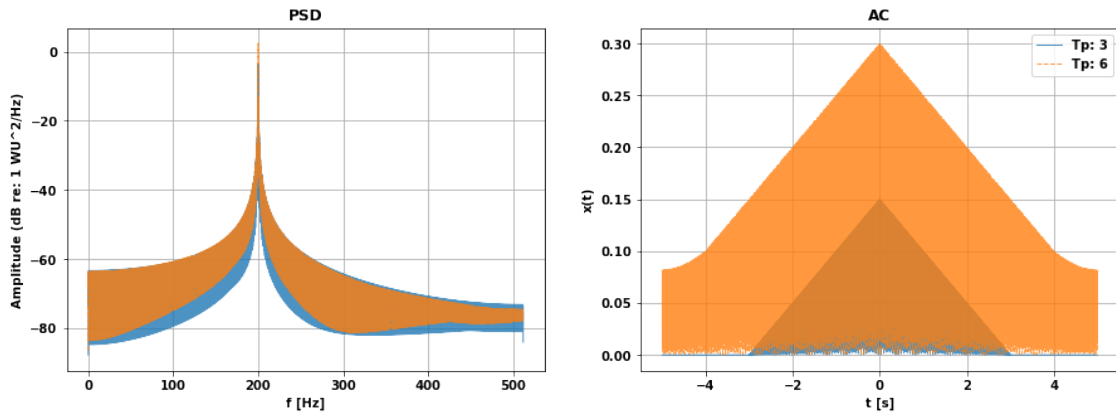
#autocorrelation
cross_corr, sxy, gxy = SignalTB.cross_corr(s,s)
axs[1].plot(np.abs(cross_corr),alpha=0.8,linestyle=linestyles[j]);
axs[1].set_title('AC'); axs[1].set_ylabel('x(t)'); axs[1].set_xlabel('t_
→[s]'); axs[1].grid(True);
plt.legend(['Tp: {}'.format(times[0]), 'Tp: {}'.format(times[1])])
fig.savefig('./plots/hw5_'+t+'.png', dpi=300, bbox_inches='tight');

```

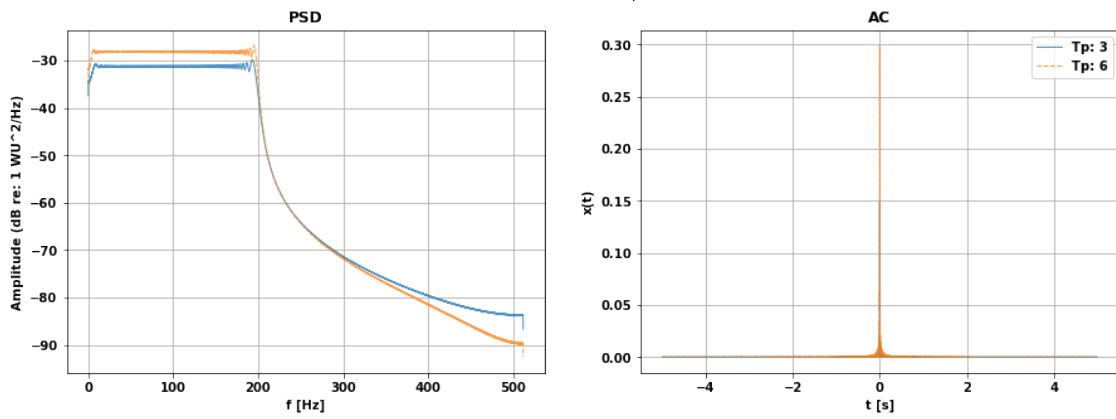
/home/m4rz910/anaconda3/lib/python3.7/site-packages/pandas/core/series.py:679:  
RuntimeWarning: divide by zero encountered in log10  
result = getattr(ufunc, method)(\*inputs, \*\*kwargs)



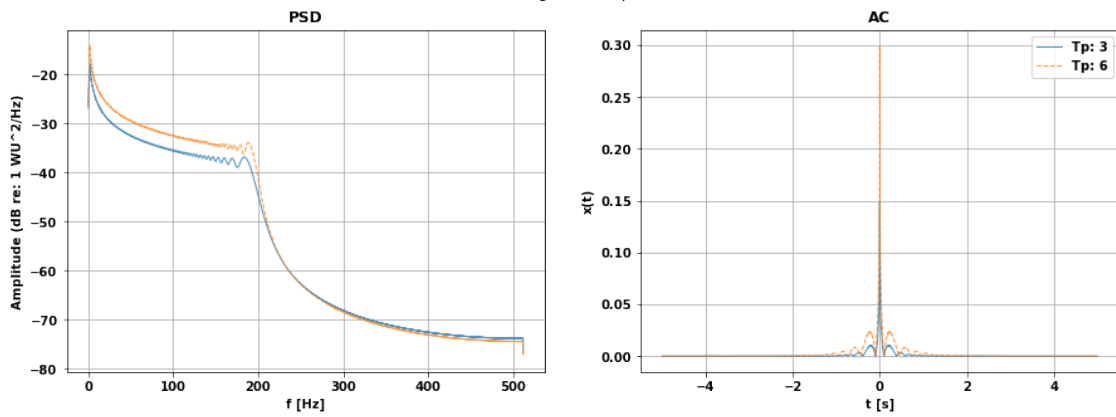
CW Pulse

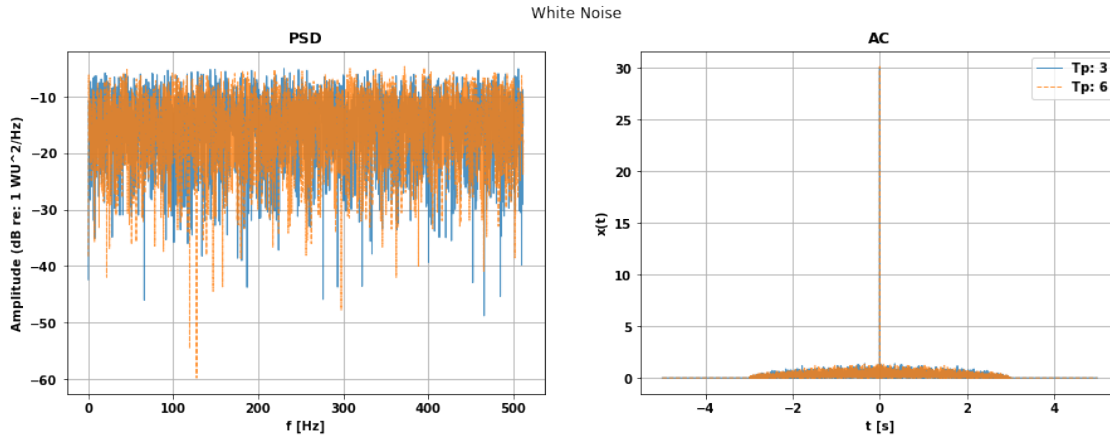


Linear Sin Sweep



Log Sin Sweep





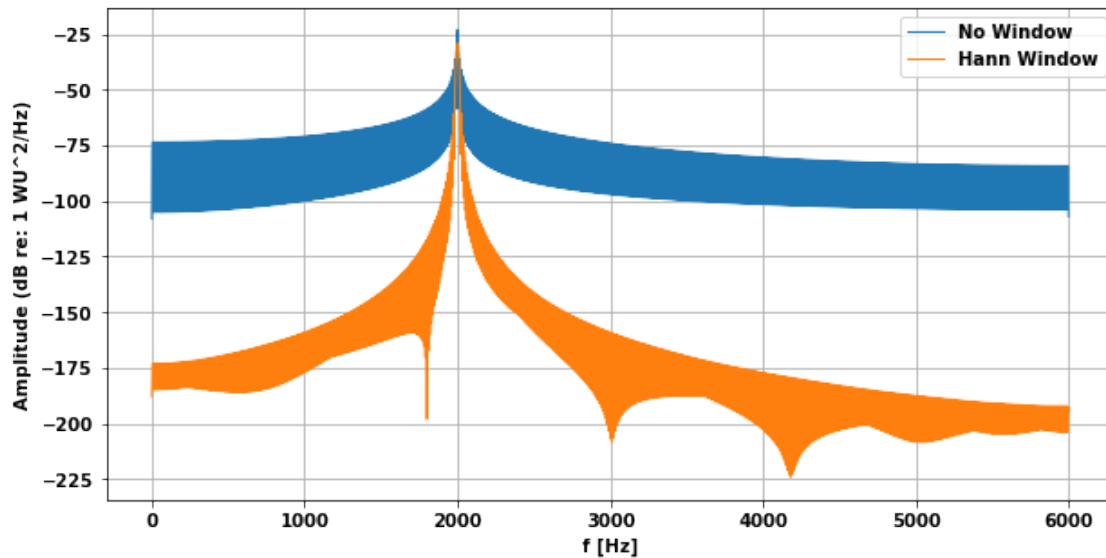
```
[40]: # 3. TAPER AND WINDOW
L = 1
Tp=0.1
fs=12000
f=2000
N = int(L/(1/fs));

fig = plt.figure(figsize=(10,5));plt.grid()
#without window
x = SignalTB.cw_sin(A=1,f=f,N=N,L=L,Tp=Tp)
s0 = SignalTB(x,fs)
s0.my_fft();
s0.sd();
#plt.plot(s0.x); #plt.xlim([0,.01])
plt.plot(10*np.log10(np.abs(s0.gxx)))

#with window
w = np.zeros(N)
w[0:int(N*Tp/L)] = np.hanning(int(N*Tp/L))

x = SignalTB.cw_sin(A=1,f=f,N=N,L=L,Tp=Tp)*w
s1 = SignalTB(x,fs)
s1.my_fft();
s1.sd();
plt.plot(10*np.log10(np.abs(s1.gxx)))
#plt.plot(s1.x);
#plt.xlim([0,5000])

plt.ylabel('Amplitude (dB re: 1 WU^2/Hz)'); plt.xlabel('f [Hz]');
plt.legend(['No Window', 'Hann Window']);
fig.savefig('./plots/hw5_'+ 'q3taper'+'.png', dpi=300, bbox_inches='tight');
```



```
[42]: # 4. Sine Sweep
# 3. TAPER AND WINDOW
L = 1
Tp=0.1
fs=12000
f1=100
f2=5000
N = int(L/(1/fs));

fig = plt.figure(figsize=(10,5));plt.grid()
#without window
x=SignalTB.lin_sin_s(A=1,N=N,L=L,Tp=Tp,f1=f1,f2=f2)
#x=SignalTB.log_sin_s(A=1,N=N,L=L,Tp=Tp,f1=f1,f2=f2)
#x = SignalTB.cw_sin(A=1,f=f,N=N,L=L,Tp=Tp)
s0 = SignalTB(x,fs)
s0.my_fft();
s0.sd();
#plt.plot(s0.x); #plt.xlim([0,.01])
plt.plot(10*np.log10(np.abs(s0.gxx)))

#with window
w = np.zeros(N)
w[0:int(N*Tp/L)] = np.hanning(int(N*Tp/L))

x=SignalTB.lin_sin_s(A=1,N=N,L=L,Tp=Tp,f1=f1,f2=f2)*w
#x=SignalTB.log_sin_s(A=1,N=N,L=L,Tp=Tp,f1=f1,f2=f2)*w
#x = SignalTB.cw_sin(A=1,f=f,N=N,L=L,Tp=Tp)*w
s1 = SignalTB(x,fs)
```

```

s1.my_fft();
s1.sd();
plt.plot(10*np.log10(np.abs(s1.gxx)))
#plt.plot(s1.x);
#plt.xlim([0,5000])

plt.ylabel('Amplitude (dB re: 1 WU^2/Hz)'); plt.xlabel('f [Hz]');
plt.legend(['No Window', 'Hann Window']);
fig.savefig('./plots/hw5_'+ 'q3taper'+ '.png', dpi=300, bbox_inches='tight');

```

