Lab 1 - Introduction to Discrete Time Signals and Systems

Instructor: Prof. Lillian Jane Ratliff

Teaching Assistants: Ashwin Srinivas Badrinath and Kevin Lin

Team Members:

In [1]:
```
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import IPython
from scipy.io import wavfile
import matplotlib.pyplot as plt
import scipy.signal
from scipy import *
import copy
import pylab as pl
from scipy import signal
import time as t
from IPython import display
```

1) Implementing Discrete Time Filters to Filter Time-Series Data

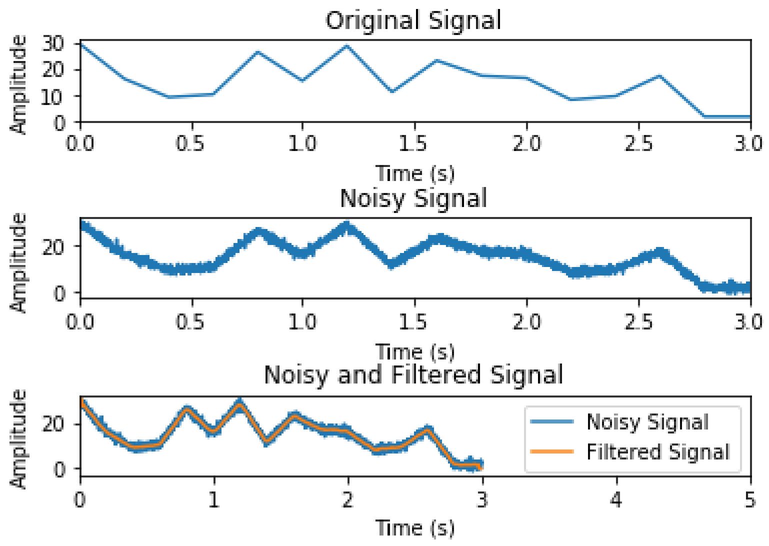
In this part, we will be looking at various discrete time filters and how they are used to make more sense of time-series data. These are very common, basic and helpful operations that one encounters in anything related to signal processing.

Implementing a Mean Filter

In [78]:
```
# choose relevant parameters
srate = 1000 # sampling rate in Hz
time  = np.arange(0,3,1/srate) # associated time vector that corresponds to 3
 seconds
n     = len(time) # length of the time vector
p     = 15 # poles for random interpolation
pi = np.pi # value of pi
k = 20

# here are some base signals to work with
base1 = np.interp(np.linspace(0,p,n),np.arange(0,p),np.random.rand(p)*30)
base2 = 5*np.sin(2*pi*5*time)

# create some random noise to be added to the above base signals

noise = np.random.randn(len(base1))# TO DO: create some random noise

# add noise to the base signals to create new noisy signals
signal1 = base1 + noise
signal2 = base2 + noise

# implement the running mean filter
filtsig1 = signal1[0:k]
filtsig1 = np.append(filtsig1,np.zeros(len(signal1)-k))

print(len(filtsig1))
print(len(signal1))
for y in range (k,len(signal1)-k):

    filtsig1[y] = np.mean(signal1[y-k:y]+signal1[y:y+k])/2



#np.zeros(n) # initialize filtered signal vector for signal 1
filtsig2 = signal2[0:k]
filtsig2 = np.append(filtsig2,np.zeros(len(signal2)-k))


for y in range (k,len(signal2)-k):

    filtsig2[y] = np.mean(signal2[y-k:y]+signal2[y:y+k])/2



#np.zeros(n) # initialize filtered signal vector for signal 2

#TO DO: finish your implementation of the running mean filter

# compute the time window size in ms and print it
windowsize = len(filtsig1)

# TO DO: compute the duration of the time-window that
# slides across the signal in ms
print("The time window size used was ",windowsize,"ms")

# TO DO: plot the required plots
# For base signal 1:
# In a single plot and three subplots, plot the original signal, noisy signal
 and
# filtered signal overliad on the noisy signal to see the difference
fig7 = plt.figure(7)
fig7.subplots_adjust(hspace = 1.2,wspace = .1)

plt.subplot(3,1,1)
plt.plot(time,base1)
plt.xlim(0,3)
plt.title('Original Signal')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')

plt.subplot(3,1,2)
plt.plot(time,signal1)
plt.xlim(0,3)
plt.title('Noisy Signal')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')

plt.subplot(3,1,3)
plt.plot(time,signal1, label = 'Noisy Signal')
plt.plot(time,filtsig1, label = 'Filtered Signal')
plt.legend()
plt.xlim(0,5)
plt.title('Noisy and Filtered Signal')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')

# For base signal 2:
# In a single plot and three subplots, plot the original signal, noisy signal
 and
# filtered signal overliad on the noisy signal to see the difference
```

```
3000
3000
The time window size used was  3000 ms
```

Out[78]:
```
Text(0, 0.5, 'Amplitude')
```



Discussion

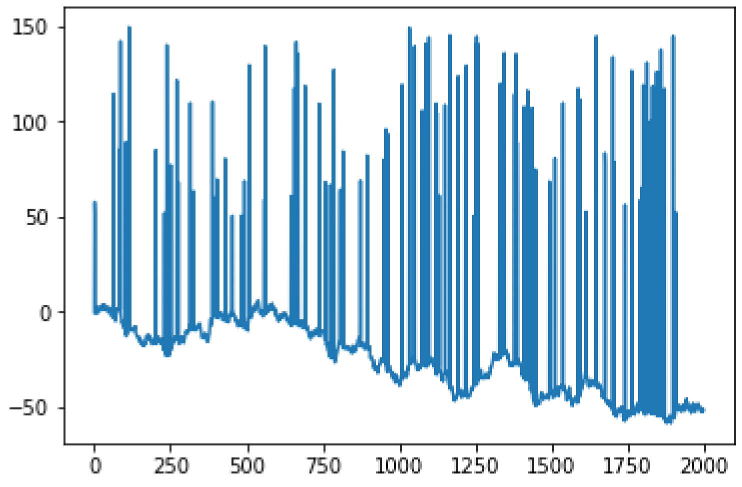
When you amplify the noise more, it makes the noisy signal deviate away from the filtered signal more. When you increase k drastically, the filtered signal starts deviating from the original signal. When you have heavy spikes in the data, the mean isn't reflective on the actual signal because some values will be affected by large outliers.

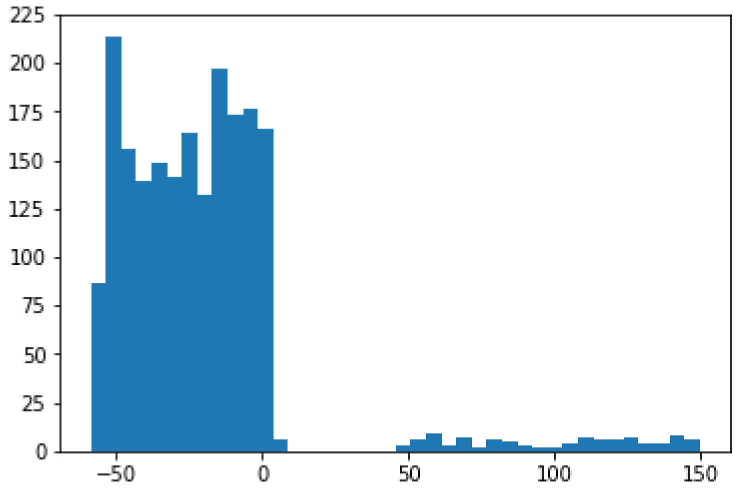
Implementing a Median Filter to Remove Spikes

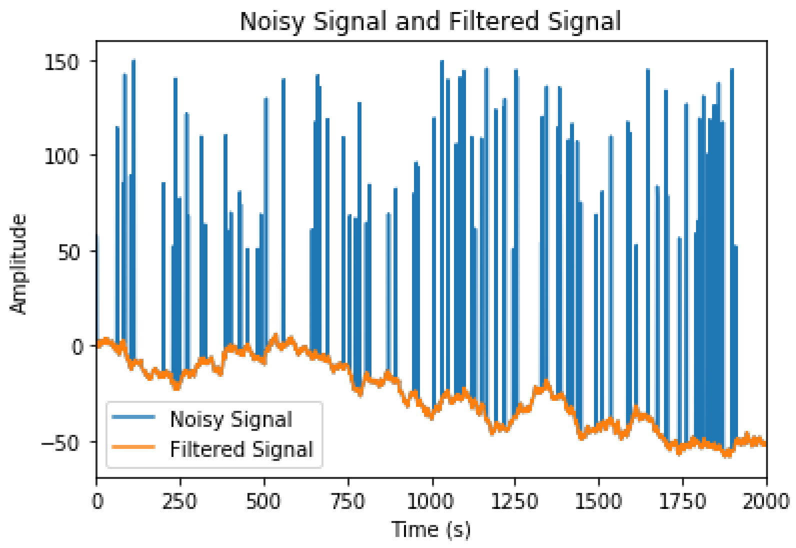
In [79]:
```python
# create signal
n = 2000
signal = np.cumsum(np.random.randn(n))

# proportion of time points to replace with noise
propnoise = .05

# find noise points
noisepnts = np.random.permutation(n)
noisepnts = noisepnts[0:int(n*propnoise)]

# generate signal and replace points with noise
signal[noisepnts] = 50+np.random.rand(len(noisepnts))*100

fig3=plt.figure(3)
plt.plot(range(0,n),signal)

# use hist to pick threshold
fig5=plt.figure(5)

plt.hist(signal,40)

# visual-picked threshold
threshold = 25


# find data values above the threshold
suprathresh = []
for y in signal:
    if y > threshold:
        suprathresh = np.append(suprathresh,np.where(signal ==y))

# initialize filtered signal
filtsig = copy.deepcopy(signal)

# loop through suprathreshold points and set to median of k
k = 20 # actual window is k*2+1
for ti in range(0,len(suprathresh)):

    index = int(suprathresh[ti])
    filtsig[index] = np.median(filtsig[index-k:index+k])


# TO DO: plot your results as directed
fig1 = plt.figure(1)
fig1.subplots_adjust(hspace = 1.2,wspace = .1)

plt.subplot(1,1,1)
plt.plot(np.arange(0,n),signal, label = "Noisy Signal")
plt.plot(np.arange(0,n),filtsig, label = "Filtered Signal")

plt.legend()
plt.xlim(0,n)
plt.title('Noisy Signal and Filtered Signal')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
```

```
C:\Users\Tyan\Anaconda3\lib\site-packages\numpy\core\fromnumeric.py:2920: Run
timeWarning: Mean of empty slice.
  out=out, **kwargs)
C:\Users\Tyan\Anaconda3\lib\site-packages\numpy\core\_methods.py:85: RuntimeW
arning: invalid value encountered in double_scalars
  ret = ret.dtype.type(ret / rcount)
```

Out[79]:
```
Text(0, 0.5, 'Amplitude')
```







Discussion

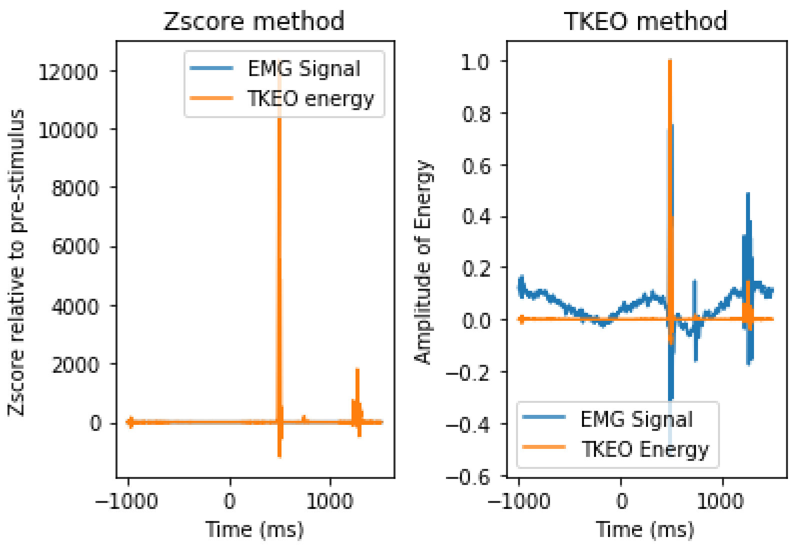
The advantage of using the median filter is that it is not affected by large outliers such as drastic spikes. The advantage for using the mean filter is that when you have a signal without that much noise, the mean is more reflective of the original signal because it considers all points together, whereas the median filter takes one data point at that specific point in time.

Denoising an EMG signal

In [4]:
```
# import data
emgdata = scipy.io.loadmat('EMG.mat')

# extract needed variables
emgtime = emgdata['emgtime'][0]
emg  = emgdata['emg'][0]
print(emg)
print(emgtime)
# initialize filtered signal
emgf = copy.deepcopy(emg) # this is where the result of
# # the TKEO algorithm should be stored

# # apply the TKEO algorithm
for y in range(1,len(emg)-1):
    emgf[y] = (emg[y]**2) - (emg[y-1]*emg[y+1])
# TO DO: You can implement this with a for loop or use a vectorized approach

## convert both signals to zscore
zeroindex = np.where(emgtime == 0)
zeroindex = int(zeroindex[0])
print(zeroindex)
print(emgtime[zeroindex])
sumEmg = 0
sumEmgf = 0
x = emg[0: zeroindex]
stdEmg = np.std(emg[0: zeroindex])
print(stdEmg)

for y in range (0,zeroindex):
    sumEmg = sumEmg + emg[y]
    sumEmgf = sumEmgf + emgf[y]
meanEmg = sumEmg/zeroindex
meanEmgf = sumEmgf/zeroindex

emgZ = copy.deepcopy(emg)
emgfZ = copy.deepcopy(emgf)
for y in range (0,len(emg)):
    emgZ[y] = (emg[y] - meanEmg)/stdEmg
    emgfZ[y] = (emgf[y]- meanEmgf)/stdEmg

fig2 = plt.figure(2)
fig2.subplots_adjust(hspace = 1.2,wspace = .4)

plt.subplot(1,2,1)
plt.plot(emgtime,emgZ, label = "EMG Signal")
plt.plot(emgtime,emgfZ, label = "TKEO energy")

plt.legend()
plt.title('Zscore method')
plt.xlabel('Time (ms)')
plt.ylabel('Zscore relative to pre-stimulus')

plt.subplot(1,2,2)
plt.plot(emgtime,emg/(np.amax(emg)), label = "EMG Signal")
plt.plot(emgtime,(emgf)/(np.amax(emgf)), label = "TKEO Energy")
plt.legend()
plt.title('TKEO method')
plt.xlabel('Time (ms)')
plt.ylabel('Amplitude of Energy')

# # find timepoint zero
# time0 = # TO DO: np.argmin() can come in handy

# # convert original EMG to z-score from time-zero
# emgZ = # TO DO:subtract the mean of the emg signal from 0 to time0 from the
 original emg
# # and divide that by the standard deviation of the signal from 0 to time0

# # same for filtered EMG energy
# emgZf = # TO DO: Repeat for the filtered signal


# # TO DO: plot your results as directed
```

```
[59.18904  57.842308 57.46825  ... 56.037285 56.66291  53.86403 ]
[-1000.        -998.046875  -996.09375  ...  1496.09375   1498.046875
  1500.      ]
512
0.0
21.380733
```

Out[4]:
```
Text(0, 0.5, 'Amplitude of Energy')
```



Discussion

The mean filter would be inaccurate at some points because there are points where there are drastic spikes in the data. However, the median filter would do better because it would ignore these spikes. We would use the median filter because we found it handles spikes in data really well.

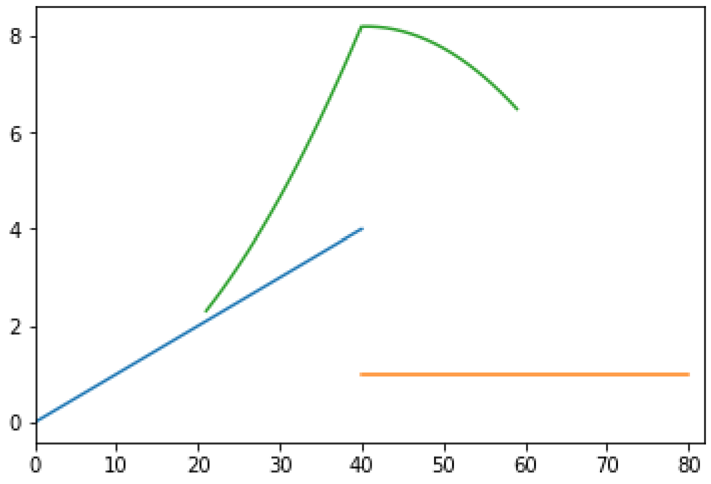
2) Convolution

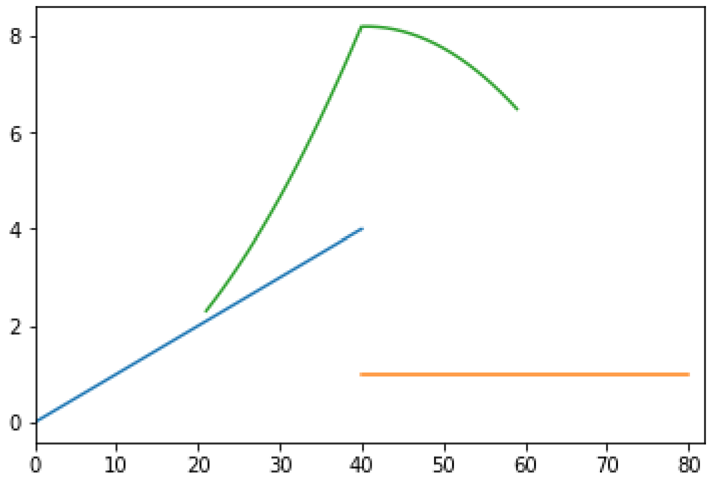
In this section you will be implementing your own convolution sum and plotting an animated plot that shows the process unfolding.

In [5]:
```
# define the basic signal generating functions
def u(t): # this returns a step signal
    return 1*(t>=0)
def r(t): # this return a ramp signal
    fs = 10
    if t >= 0:
        return t/fs
```

In [6]:
```
# TO DO: fill in the signal and kernel below as directed

# Use a 4s long ramp sampled at 10 samples/s for the signal
# Use a 4s long step signal for the kernel
fs = 10
time = np.arange(0, (fs * 4) + 1,1)
ramp = []
unitStep = []

for x in time:
    unitStep.append(u(x))
    ramp.append(r(x))

#signal
signal1= ramp

# convolution kernel
kernel = unitStep

# convolution sizes
nSign = len(signal1)
nKern = len(kernel)
nConv = nSign + nKern - 1

## convolution in animation

half_kern = int( np.floor(nKern/2) )

# flipped version of kernel
kflip = kernel[::-1] #-np.mean(kernel)

# zero-padded data for convolution
dat4conv = np.concatenate( (np.zeros(half_kern),signal1,np.zeros(half_kern)) ,
axis=0)

# initialize convolution output
conv_res = np.zeros(nConv)


# run convolution
for ti in range(half_kern,nConv-half_kern):

    # get a chunk of data
    th = dat4conv[ti-half_kern:ti+half_kern + 1]
    tempdata = th
    # TO DO: store the slice of th signal for the current time step
    sum1 = np.dot(tempdata,kflip)
    print(sum1)
    # compute dot product (don't forget to flip the kernel backwards!)
    conv_res[ti] = np.sum(tempdata * kflip) / fs

    # draw plot
    pl.cla() # clear the axis
    plt.plot(signal1)
    plt.plot(np.arange(ti-half_kern,ti+half_kern+1),kflip)
    plt.plot(np.arange(half_kern+1,ti),conv_res[half_kern+1:ti])
```

```
        plt.xlim([0,nConv+1])


        display.clear_output(wait=True)
        display.display(pl.gcf())
        t.sleep(.01)


# cut off edges
conv_res = conv_res[half_kern:-half_kern]
```





Discussion

It is important because when you increase the sampling rate, you are increasing the overall sum because you are adding more data points together, therefore, you have to divide by the sampling rate. The convolution sum is discrete which is why we have to normalize it, but the convolution integral is continuous and uses dx which is infinitely small and we don't have to divide by a sampling rate.

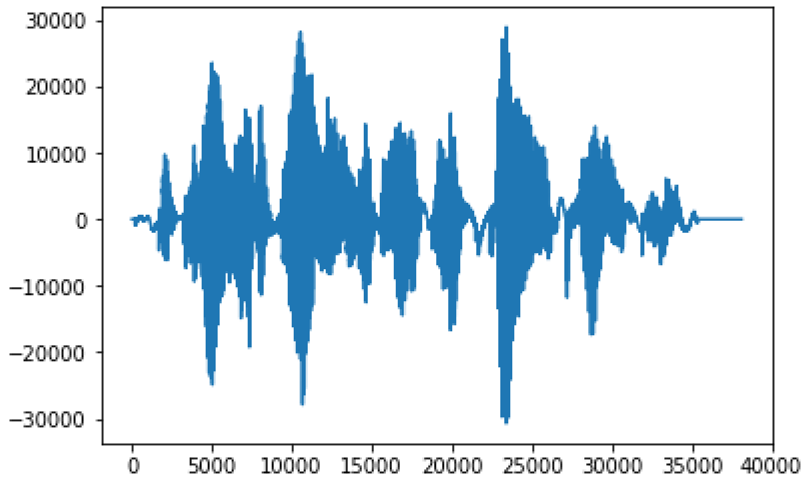
3) Analog vs Digital Transmission

This section is meant to begin the Analog vs Digital comparison and get you thinking about why we bother with discrete signals. One reason is that signal processing and computation is all done on computers wich operate using digital circuitry and hence are discrete. Even otherwise, the nature of digital signals still makes them more advantageous to use in quite a lot of cases.

In [140]:
```
rate, s = wavfile.read('speech.wav')
plt.plot(s);
IPython.display.Audio(s, rate=rate)
```

Out[140]:

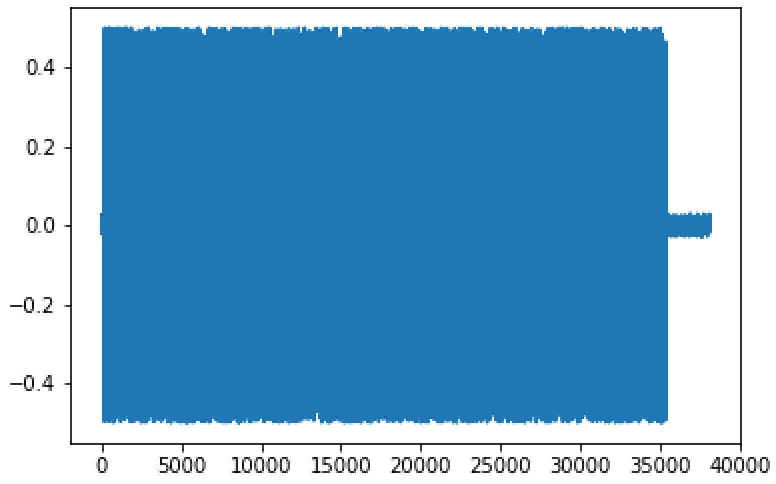
0:00 / 0:02



In [141]:
```
# the analog signal is simply rescaled between -100 and +100
# largest element in magnitude:
norm = 1.0 / max(np.absolute([min(s), max(s)])) # the normalizing factor
sA = 100.0 * s * norm # the "analog" signal

# the digital version is clamped to the integers
sD = np.round(sA) # the "digital" signal
```

In [142]:
```
# TO DO: plot the difference between the analog and digital signal

difference = []
for x in range(0, len(sD)):
    difference.append(float(sD[x]) - sA[x])

fig3 = plt.figure(3)
fig3.subplots_adjust(hspace = 1.2,wspace = .4)

plt.subplot(1,1,1)
plt.plot(np.arange(0, len(sD)), difference)
```

Out[142]:
```
[<matplotlib.lines.Line2D at 0x1c99a49e978>]
```



In [143]:
```
# function to calculate SNR
def SNR(noisy, original):
    # power of the error
    err = np.linalg.norm(noisy - original) # TO DO: use np.linalg.norm() to fi
nd the power of the noisy signal
    #power of the signal
    sig = np.linalg.norm(original) # TO DO: use np.linalg.norm() to find the p
ower of the original signal
    # SNR in dBs
    snr = (np.log10(sig/err)) * 10 # TO DO: return the log (base 10) of the ra
tio and amplify if by a factor of 10
    return snr

# TO DO: print the snr of the signal a directed
print(SNR(sD, sA))
```

```
17.124344123539522
```

In [144]:
```
def repeater(x, noise_amplitude, attenuation):
    # first, create the noise
    noise = np.random.uniform(-noise_amplitude, noise_amplitude, len(x))
    # TO DO: fill in the steps as directed by the documentation:

    # attenuation
    x *= attenuation
    # noise
    x += noise
    # gain compensation
    x /= attenuation
    return x
```

In [145]:
```
def analog_tx(x, num_repeaters, noise_amplitude, attenuation):
# TO DO: modify x to represent analog transmission over the given number of re
peaters
    for i in range(num_repeaters):
        x = repeater(x, noise_amplitude, attenuation)
    return x
```

In [146]:
```
def digital_tx(x, num_repeaters, noise_amplitude, attenuation):
# TO DO: modify x to represent digital transmission over the given number of r
epeaters
# hint: np.round() will come in handy when you're trying
# to round your signals values to the nearest integer
    for i in range(num_repeaters):
        x = np.round((repeater(x, noise_amplitude, attenuation)))
    return x
```

Discussion

One advantage that analog signals have over digital signals is that there is no loss in data. They both sound noisy, however the analog signal doesn't sound as clear. It's near impossible to hear what he is saying in the analog signal. This makes sense because the SNR of the analog signal is much greater than the SNR of the digital signal.

In [148]:
```
# keep these parameters
NUM_REPEATERS = 20
NOISE_AMPLITUDE = 0.2
ATTENUATION = 0.5

# TO DO: find the final signal that is recieved after transmission and store t
he
# "analog" and "digital" versions in yA and yB respectively and print the SNR

yA = analog_tx(sA, NUM_REPEATERS, NOISE_AMPLITUDE, ATTENUATION)

yD = digital_tx(sD, NUM_REPEATERS, NOISE_AMPLITUDE, ATTENUATION)
print(SNR(yA, sA))
print(SNR(yD, sD))
```

```
inf
17.117995918392353

C:\Users\Tyan\Anaconda3\lib\site-packages\ipykernel_launcher.py:8: RuntimeWar
ning: divide by zero encountered in double_scalars
```