Priyanka Raavi

Student No: 200393260

Email: priyankaravi03@gmail.com

Part1: Transmission of Information without noise based on Shannon's model of Communication(Assign1)

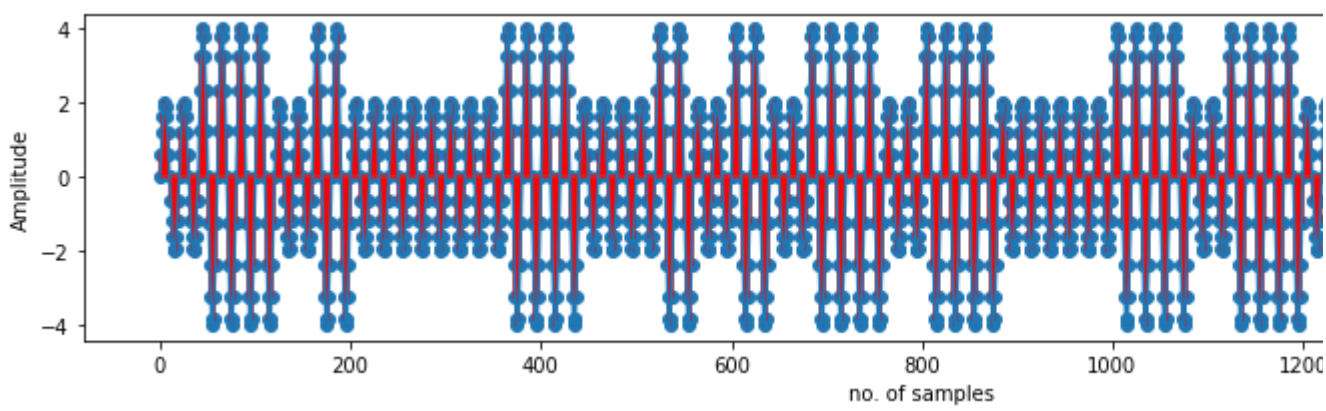
```
import math
import struct
import numpy as np
from scipy import signal as sg
import matplotlib
import matplotlib.pyplot as plt
import binascii
import random
```

```
spb=40    #Samples per bit
cpb=2     #Cycles per bit
T=spb/cpb #period of Samples
```

```
#converting text to bits
def text_to_bits(text, encoding='ISO-8859-1', errors='surrogatepass'):
    bits = bin(int(binascii.hexlify(text.encode(encoding, errors)), 16))[2:]
    return bits.zfill(8 * ((len(bits) + 7) // 8))
```

```
#encode function
def encode(asciimsg):
  binmsg=text_to_bits(asciimsg)
  message=[]
  for bit in binmsg:
    if bit.isspace():
      pass
    else:
      message.append(int(bit))
  Totalsamples=spb*len(message)
  x=np.arange(Totalsamples)
  carrier=np.sin(2*np.pi*(1/T)*(x))
  messageList=[]
  for bit in message:
    if bit is 0:
      for i in range(0,spb):
        messageList.append(2)
    else:
      for i in range(0,spb):
        messageList.append(4)
  message=messageList
  transsignal=carrier*message
  nestedList=[x,transsignal]
  %matplotlib inline
  fig=plt.figure(figsize=(15,3))
  plt.stem(x,transsignal,'r')
  plt.plot(x,transsignal)
  plt.xlabel("no. of samples")
  plt.ylabel("Amplitude")
  #print(nestedList)
  return nestedList
```

```
encodeList=encode("hello")       #list contain time and amplitude list
```




```
#convert bits to text
def text_from_bits(bits, encoding='ISO-8859-1', errors='surrogatepass'):
    n = int(bits, 2)
    return int2bytes(n).decode(encoding, errors)

def int2bytes(i):
    hex_string = '%x' % i
    n = len(hex_string)
    return binascii.unhexlify(hex_string.zfill(n + (n & 1)))


def decode(nestedList):
  transsignal= nestedList[1] #transsignal contain only the amplitud
  #x1=newlist[0]
  amplitudeList=[]
  messageinbits=len(transsignal)//spb #In this case, output of messageinbits (450/3
#creating a list of amplitudes for every bit in a message into a amplitudes.(i.e; amptitude
  for i in range(0,messageinbits):
    tempList=[] #tempList contains the list of amplitudes of
    tempList.clear()
    for j in transsignal[(i*spb):(i+1)*spb]:
      tempList.append(j)
    amplitudeList.append(tempList)
#Decoding Scheme
  threshold =2.5
  binaryList = []
  for i in range(0,len(amplitudeList)):
    crossing = 0
    for x in amplitudeList[i]:
      if x >= threshold:
        crossing = crossing + 1
    if crossing >=6:
      binaryList.append(1)
    else:
      binaryList.append(0)
#storing the list of binary values into a variable
  for i in range(0,len(binaryList)):
    binary= ''+str(binaryList[i])
  binarymsg =''.join(str(i) for i in binaryList) #final binary form of a message
  textmessage=text_from_bits(binarymsg)
  #print(binarymsg)
  return textmessage


print(decode(encodeList))
```

```
hello
```

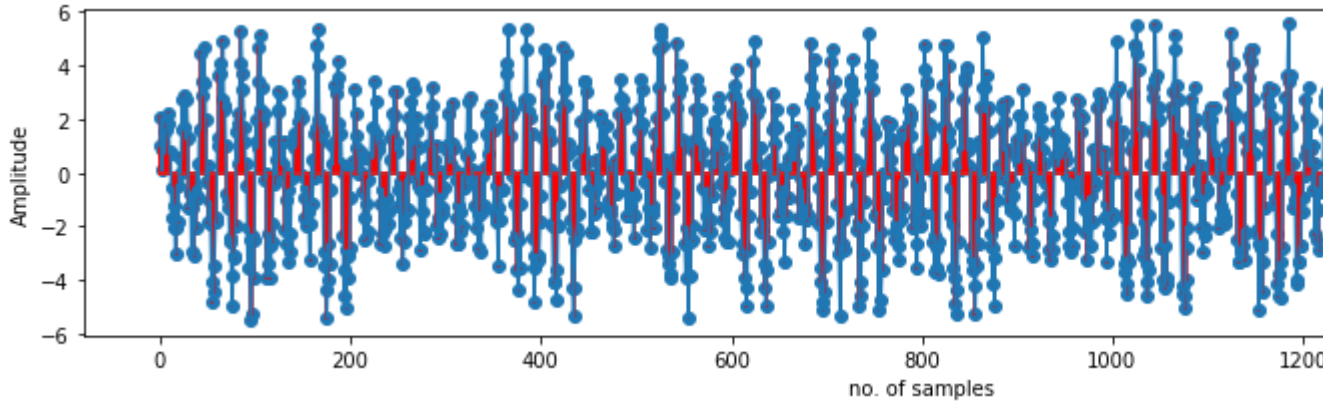
PartII: The Discrete Channel with Noise- Assignment3

```
#noise generation
def noise(Namplitude, signallength):
  noiseList = []
  for x in range(signallength):
    noiseList.append((random.randint(-100,100)*(Namplitude))/100)

  return noiseList
```

```
Namplitude = 1.6
asciimsg="hello"
slist = encode(asciimsg)
signal = slist[1]
signallength = len(signal)
noise(Namplitude, signallength)
signalwithnoise = []
signalwithnoise = signal + noise(Namplitude, signallength)   # adding signal with the noise
x=np.arange(signallength)
decodesignal=[x,signalwithnoise]
%matplotlib inline
fig=plt.figure(figsize=(15,3))
plt.stem(x,signalwithnoise,'r')
plt.plot(x,signalwithnoise)
plt.xlabel("no. of samples")
plt.ylabel("Amplitude")
```

```
Text(0, 0.5, 'Amplitude')
```




```
#SNR
def SNR(Namplitude):
  snr = 2.5/Namplitude
  return snr

#comparing 2 strings to get the character difference
def compareString(asciimsg,noisedmsg):
  counter = 0
  for i in range(len(asciimsg)):
    if asciimsg[i]!=noisedmsg[i]:
      counter = counter + 1
  return counter

#calculating accuracy
def getaccuracy(asciimsglength, Noise):
```

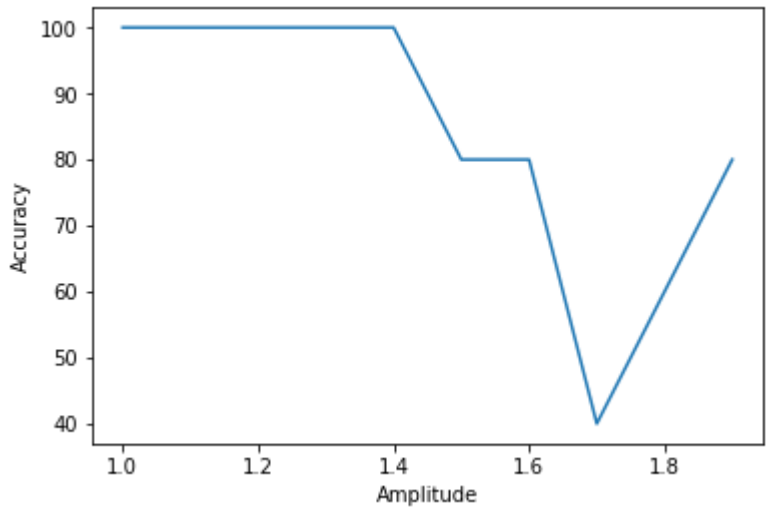
```
  differinchar = asciimsglength - Noise
  return (differinchar/asciimsglength)*100


#noised signal function to get calculate the accuracy at various points of SNR
def noisedSignal(asciimsg):

  List = encode(asciimsg)
  signal = List[1]
  signallength = len(signal)
  accuracylist = []
  amplitude = []
  accuracyAmplitude = []
  for i in range(10,20):
    Namplitude = i/10
    noiseList = noise
    noiseList = noise(Namplitude, signallength)
    signalwithnoise = []
    signalwithnoise = signal + noise(Namplitude, signallength)
    x=np.arange(signallength)
    decodesignal=[x,signalwithnoise]
    decodemsg = decode(decodesignal)
    counter = compareString(asciimsg,decodemsg)
    snr = SNR(Namplitude)
    accuracy = getaccuracy(len(asciimsg),counter)
    accuracylist.append(accuracy)
    amplitude.append(Namplitude)
  accuracyAmplitude.append(accuracylist)
  accuracyAmplitude.append(amplitude)
  return accuracyAmplitude


accuracyAmplitude = noisedSignal(asciimsg)
accuracylist = accuracyAmplitude[0]
amplitude = accuracyAmplitude[1]
%matplotlib inline
plt.plot(amplitude,accuracylist)
plt.xlabel("Amplitude")
plt.ylabel("Accuracy")
```

```
Text(0, 0.5, 'Accuracy')
```



PartIII: Model of Communication by adding Forward

To overcome the noise that had added to the transmission signal, we have three ways[3]:

1. Increase the signal
    - By increasing the power of the signal, it will decrease the intensity of the noise.
2. Increase Bandwidth
    - Increase in the Bandwidth will increase the size of the channel capacity.
3. Coding Theory
    - Among all these, using the coding theory will help us to rectify most of the errors that occur in the transmission. The coding theory can be done in three ways.

A. Error Correction Coding

B. Data Compression

C. Spectrum Spreading

A. Error Correction Coding(ECC):

```
  ECC is a process of adding redundant data to the message so that, it can retrieve
at the receiver side[2]. There are 2 types of ECC:
  1. Forward Error Correction(FEC)
  2. Backward Error Correction(BEC)
```

1. Forward Error Correction(FEC):
    - Rectifying the errors at the time of decoding the message signal that had received.
2. Backward Error Correction(BEC):
    - When the receiver has received the message, the algorithm checks the errors and try to rectify them, but if we still have errors, then the acknowledgment is sent back to the encoder to saying that the message has been corrupted and ask to send the message again.

PartA: Choose a Method

To test my algorithm, I have chosen to use the Forward Error Correction method. We have a lot of forward error-correcting codes available like Hamming code, Triple modular redundancy, Reed Solomon, Walsh- Hadamard code and soon.

After my research on all types of ECC, I have chosen to use the Triple modular redundancy because we can overcome the noise and get the message by using triple modular redundancy to the message and also it is a very easy and efficient method of encoding. It is an easy job for the encoder to encode the message and would be an easy way for the decoder to decode the message as well.

Triple modular Redundancy:

Triple modular redundancy is based on majority voting. Let me explain in detail[1].

In triple modular redundancy, we will encode each bit in the message by redundant each bit 3 times. For instance, we have to send a "hello" message to the receiver. We will encode the message as "hhheeellllllooo" and then will send this message to the receiver.

The receiver will receive the message and will decode by taking 3 bits at a time and reads each bit. Whichever bit comes redundant among the 3 bits, that will be taken as the original message bit. This redundancy of bits

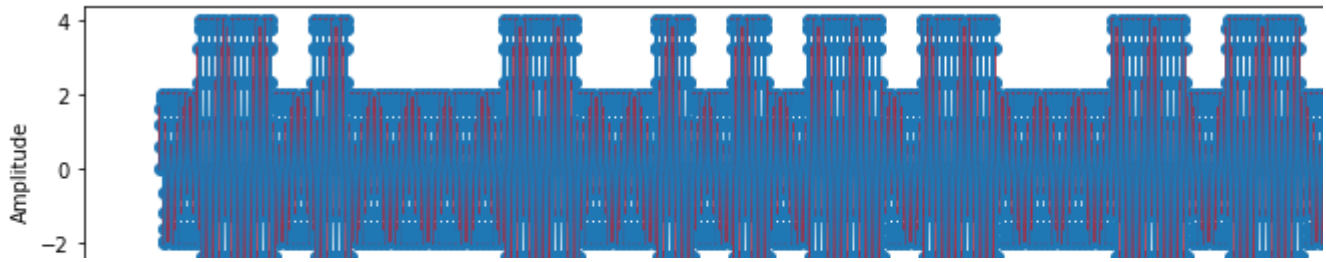
PartB: Implement an FEC Scheme

I have implemented the triple modular redundancy function right after reading the message and converting the message into bits using text_to_bits function. I took the list of bits of the message and implemented triple modular redundancy to that bits.

```
# implementing triple modular redundancy while encoding the function
def tripleredundancy(msgbits):
  message = [i for i in msgbits for j in range(3)]
  return message


#encode function
def encodefec(asciimsg):
  binmsg=text_to_bits(asciimsg)
  message=[]
  for bit in binmsg:
    if bit.isspace():
      pass
    else:
      message.append(int(bit))
  #calling triple redundancy function
  message= tripleredundancy(message)
  Totalsamples=spb*len(message)
  x=np.arange(Totalsamples)
  carrier=np.sin(2*np.pi*(1/T)*(x))
  messageList=[]
  for bit in message:
    if bit is 0:
      for i in range(0,spb):
        messageList.append(2)
    else:
      for i in range(0,spb):
        messageList.append(4)
  message=messageList
  transsignal=carrier*message
  nestedList=[x,transsignal]
  %matplotlib inline
  fig=plt.figure(figsize=(15,3))
  plt.stem(x,transsignal,'r')
  plt.plot(x,transsignal)
  plt.xlabel("no. of samples")
  plt.ylabel("Amplitude")
  return nestedList


EncodeFEC= encodefec("hello")
```



Part C: Decoding and Parity Check

I have implemented the decoding part of the triple redundancy at the point where after implementing my decoding scheme and storing all the list of bits into a variable. I took the range of 3 bits from the list of bits and calculated the majority of the bit value and stored that bit into a variable list and then converted the bits into a text message.

As I have used Triple redundancy Forward Error Correction method, there is no need to do the parity checks.

```
def decodefec(nestedList):
  transsignal= nestedList[1] #transsignal contain only the amplitud
  #x1=newlist[0]
  amplitudeList=[]
  messageinbits=len(transsignal)//spb #In this case, output of messageinbits (450/3
#creating a list of amplitudes for every bit in a message into a amplitudes.(i.e; amptitude
  for i in range(0,messageinbits):
    tempList=[] #tempList contains the list of amplitudes of
    tempList.clear()
    for j in transsignal[(i*spb):(i+1)*spb]:
      tempList.append(j)
    amplitudeList.append(tempList)
#Decoding Scheme
  threshold =2.5
  binaryList = []
  for i in range(0,len(amplitudeList)):
    crossing = 0
    for x in amplitudeList[i]:
      if x >= threshold:
        crossing = crossing + 1
    if crossing >=6:
      binaryList.append(1)
    else:
      binaryList.append(0)
#storing the list of binary values into a variable
  for i in range(0,len(binaryList)):
    binary= ''+str(binaryList[i])
  binarymsg =''.join(str(i) for i in binaryList) #final binary form of a message
  message=[]
  # implemeting the triple redundancy.
  for bit in binarymsg:
    if bit.isspace():
      pass
    else:
      message.append(int(bit))
  msglist=[]
  len_array =[]
  for i in range(0,int(len(message)/3)):
    len_array.append(i*3)
  for i in len_array:
    if(message[i] == message[i+1]):
      msglist.append(message[i])
    else:
      msglist.append(message[i+2])
  binarymessage =''.join(str(i) for i in msglist)
  textmessage=text_from_bits(binarymessage)
```

```
  return textmessage
```

```
DecodedMsg = decodefec(EncodeFEC)
print("received Msg:",DecodedMsg,"\n\n\n")
```

```
received Msg: hello
```

Again I have calculated the accuracy of the message decoding after using the error correction scheme, the same way as assignment3.

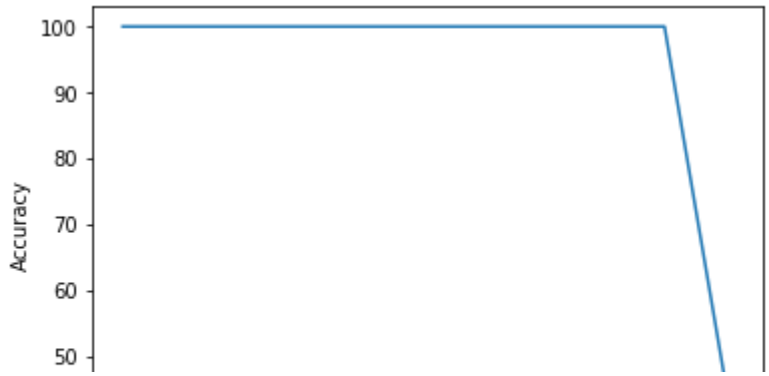
```
def signalNoiseFunction2(asciimsg):

  signalList = encodefec(asciimsg)
  signal = signalList[1]
  signalLength = len(signal)
  accuracyList = []
  amplitudeList = []
  accuracyAmplitudeList = []
  for i in range(10,20):
    noiseAmplitude = i/10
    noiseList = noise
    noiseList = noise(noiseAmplitude, signalLength)


    noiseAddedSignal = []
    noiseAddedSignal = signal + noise(noiseAmplitude, signalLength)
    x=np.arange(signalLength)
    decodesignal=[x,noiseAddedSignal]
    receivedMessage = decodefec(decodesignal)

    #comparing message and noise added message
    counter = compareString(asciimsg,receivedMessage)
    snr = SNR(noiseAmplitude)
    accuracy = getaccuracy(len(asciimsg),counter)
    accuracyList.append(accuracy)
    amplitudeList.append(noiseAmplitude)
  accuracyAmplitudeList.append(accuracyList)
  accuracyAmplitudeList.append(amplitudeList)
  return accuracyAmplitudeList
accuracyAmplitudeList = signalNoiseFunction2(asciimsg)
accuracyList = accuracyAmplitudeList[0]
amplitudeList = accuracyAmplitudeList[1]
%matplotlib inline
plt.plot(amplitudeList,accuracyList)
plt.xlabel("Amplitude")
plt.ylabel("Accuracy")
```

```
Text(0, 0.5, 'Accuracy')
```



Part D: Comparission and Discussion

As we can see, after adding the error correction scheme, the accuracy of the received message has been increased to 1.8, which means, till 1.8 amplitude of noise, the message can send accurately. That results there is an increase of around 30% in the accuracy after using error correction scheme than after adding noise to the signal.

Short Commings:

1. Due to the function used in the process of generation of noise, the results may vary a bit.

Limitations of Triple Modular Redundancy:

1. The encoding scheme may lead to increasing the prone of errors when operating at high speeds.
2. It may lead to accidents when using complex systems.

My original scheme would work only at the time when the noise does not interrupt my transmission. However, this scheme can be to work and provide a good amount of accuracy in the delivery of the message even when the noise interrupts the transmission of a message. So, my newly developed scheme had better advantages than my original scheme.

References

1.("Triple modular redundancy", 2019). https://en.wikipedia.org/wiki/Triple_modular_redundancy

2.("Error correction code", 2019) https://en.wikipedia.org/wiki/Error_correction_code

3.C. E. Shannon, "A mathematical theory of communication," in The Bell System Technical Journal, vol. 27, no. 3, pp. 379-423, July 1948. doi: 10.1002/j.1538-7305.1948.tb01338.x. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6773024&isnumber=6773023 References