09w1qqj6e

December 28, 2024

0.1 Team Memebers

- Aser Osama 202101266
- Aya Sherif 202100642
- Mariam Ismael 202101506
- Salma Hatem 202100381

1 Part 1

1.1 Setup

Importing file

[1]:
```
from google.colab import files


text_file = files.upload()
```

```
<IPython.core.display.HTML object>
```

```
Saving Test_text_file.txt to Test_text_file.txt
```

1. Computing an estimate of the probabilities of the different English characters (symbols) in this text file

[2]:
```
# saving the text in the file in a string
text = text_file["Test_text_file.txt"]

text = str(text)
text = text[2:len(text)-1]

# create dictionary with character and their coressponding probability
char_probability = {}
for i in range(97, 123):
    char_probability[chr(i)] = 0
char_probability['('] = 0
char_probability[')'] = 0
char_probability['.'] = 0
char_probability[','] = 0
```

```
char_probability['/'] = 0
char_probability['-'] = 0
char_probability[' '] = 0

for c in text:
    char_probability[c] += 1
for x in char_probability:
    char_probability[x] = char_probability[x]/len(text)
print(char_probability)
```

```
{'a': 0.07193732193732194, 'b': 0.012108262108262107, 'c': 0.026353276353276354,
'd': 0.045584045584045586, 'e': 0.10398860398860399, 'f': 0.013532763532763533,
'g': 0.014245014245014245, 'h': 0.022079772079772078, 'i': 0.06054131054131054,
'j': 0.0007122507122507123, 'k': 0.0007122507122507123, 'l':
0.04202279202279202, 'm': 0.021367521367521368, 'n': 0.0641025641025641, 'o':
0.06267806267806268, 'p': 0.037037037037037035, 'q': 0.0007122507122507123, 'r':
0.05982905982905983, 's': 0.05626780626780627, 't': 0.07193732193732194, 'u':
0.018518518518518517, 'v': 0.009259259259259259, 'w': 0.007834757834757835, 'x':
0.004985754985754986, 'y': 0.00641025641025641, 'z': 0.002136752136752137, '(':
0.002136752136752137, ')': 0.002136752136752137, '.': 0.004273504273504274, ',':
0.007834757834757835, '/': 0.0007122507122507123, '-': 0.011396011396011397, '
': 0.1346153846153846}
```

2. Calculating the entropy

[3]:
```
import math
entropy = 0
for c in char_probability:
    entropy += -char_probability[c] * math.log2(char_probability[c])

entropy
```

[3]:
```
4.25701056473807
```

3. Calculating the number of bits/symbol in a fixed length code and its effeciency

[4]:
```
bits_fixed_length =  math.ceil(math.log2(len(char_probability)))
effeciency_fixed_length = entropy/bits_fixed_length
print("fixed length code\t "+ str(bits_fixed_length) + " bits/symbol\n",␣
 ↪"effeciency\t\t", effeciency_fixed_length*100, " %")
```

```
fixed length code        6 bits/symbol
 effeciency              70.95017607896783  %
```

1.2 Huffman

4. Encoder

[5]:

```
# creating class for tree node
class Node:
    def __init__(self, char=None, probability=0):
        self.char = char
        self.probability = probability
        self.left = None
        self.right = None

# initializing nodes
tree = []
for x in char_probability:
    tree.append(Node(x, char_probability[x]))

# creating tree
while(len(tree) > 1):
    tree.sort(key = lambda p: -p.probability)  # sort probabilities descendingly
    left = tree.pop()
    right = tree.pop()
    node = Node('*',left.probability + right.probability)
    node.right = right
    node.left = left
    tree.append(node)



# creating code
char_code = {}
for x in char_probability:
    char_code[x] = ""

def code(node, node_code):
    if node:
        if node.char != '*':
            char_code[node.char] = node_code
        code(node.left, node_code + "0")
        code(node.right, node_code  + "1")

code(tree[0],'')

def encode_huffman(text):
    encoded_text = ""
    for s in text:
        encoded_text += char_code[s]

    return encoded_text

huffman_encoded_text = encode_huffman(text)
```

5. Decoder

[6]:
```python
# Decode the Huffman encoded text directly
def decode_huffman(encoded_text, root):
    decoded_text = ""
    current_node = root

    for bit in encoded_text:
        if bit == '0':
            current_node = current_node.left
        elif bit == '1':
            current_node = current_node.right

        # If a leaf node is reached, append the character and reset to root
        if current_node.char != '*':  # Leaf node
            decoded_text += current_node.char
            current_node = root  # Reset to root for the next character

    return decoded_text

# Decode the encoded text
decoded_text = decode_huffman(huffman_encoded_text, tree[0])

# Print the decoded text
print("Decoded Text:")
print(decoded_text)
```

```
Decoded Text:
in this paper, a novel decorrelation-based concurrent digital predistortion
(dpd) solution is proposed for dual-band transmitters (tx) employing a single
wideband power amplifier (pa), and utilizing just a single feedback receiver
path. the proposed decorrelation-based parameter learning solution is both
flexible and simple, and operates in a closed-loop manner, opposed to the widely
applied indirect learning architecture. the proposed decorrelation-based
learning and dpd processing can also be effectively applied to more ordinary
single-band transmissions, as well as generalized to more than two transmit
bands. through a comprehensive analysis covering both the dpd parameter learning
and the main path processing, it is shown that the complexity of the proposed
concurrent dpd is substantially lower compared with the other state-of-the-art
concurrent dpd methods. extensive set of quantitative simulation and rf
measurement results are also presented, using a base-station pa as well as a
commercial lte-advanced mobile pa, to evaluate and validate the effectiveness of
the proposed dpd solution in various real world scenarios, incorporating single-
band/dual-band tx cases. the simulation and rf measurement results demonstrate
excellent linearization performance of the proposed concurrent dpd, even
outperforming current state-of-the-art methods, despite the significantly lower
complexity.
```

```
[7]: with open("decode_file_huffman.txt", "w") as file:
         file.write(decoded_text)
     file.close()
```

comparing decoded stream with the original stream

```
[8]: def CompareTexts(text1, text2):
       if len(text1) != len(text2):
         return False
       for i in range(0, len(text1)):
         if text1[i] != text2[i]:
           print(i)
           return False
       return True

     print("Texts are the same") if CompareTexts(text, decoded_text) else␣
      ↪print("Texts are not the same")
```

```
Texts are the same
```

6. Calculating the efficiency of the Huffman code

```
[9]: def CalculateEfficiencyHuffman():
       average_code_length = 0
       for code in char_code:
         average_code_length += len(char_code[code]) * char_probability[code]
       return entropy / average_code_length

     print("Huffman code efficiency: ", CalculateEfficiencyHuffman())
```

```
Huffman code efficiency:  0.995476820934752
```

1.3 Shannon-Fano

7. Encoder

```
[10]: from pprint import pprint
      def ShannonFanoEncoder(symbols):
        codes = {}
        assignCodes(symbols, "", codes)
        return codes

      def assignCodes(symbols, prefix, codes):
          if len(symbols) == 1:
              codes[symbols[0][0]] = prefix
              return

          splitPoint = findSplitPoint(symbols)
          firstGroup = symbols[:splitPoint + 1]
```

```
    secondGroup = symbols[splitPoint + 1:]

    assignCodes(firstGroup, prefix + "0", codes)
    assignCodes(secondGroup, prefix + "1", codes)


def findSplitPoint(symbols):
    totalFrequency = 0
    for i in range(0, len(symbols)):
        totalFrequency += symbols[i][1]

    cumulativeFrequency = 0
    half_total = totalFrequency / 2
    for i in range(0, len(symbols)):
        cumulativeFrequency += symbols[i][1]
        if cumulativeFrequency >= half_total:
          return i if (abs(cumulativeFrequency - half_total) <␣
 ↪abs(cumulativeFrequency - (half_total + symbols[i][1]))) else i - 1


    return len(symbols) - 1

def EncodeText(text, codes):
  encoded_text = ""
  for char in text:
    encoded_text += codes[char]
  return encoded_text


char_probability1 = {
    'a': 0.3,
    'b': 0.25,
    'c': 0.2,
    'd': 0.15,
    'e': 0.1
}
text1 = "abcdabcdabdd"

sorted_char_probability = sorted(char_probability.items(), key=lambda x: x[1], ␣
 ↪reverse=True)
codes = ShannonFanoEncoder(sorted_char_probability)
shannon_encoded_text = EncodeText(text, codes)
```

Shannon-Fano codes

[11]:
```
print("{:<10} {:<10}".format('character', 'code'))
for key, value in codes.items():
```

```
    print("{:<10} {:<10}".format(key, value))
```

```
character  code
           000
e          001
a          0100
t          0101
n          0110
o          0111
i          1000
r          1001
s          1010
d          10110
l          10111
p          11000
c          11001
h          11010
m          11011
u          111000
g          111001
f          111010
b          111011
-          111100
v          1111010
w          1111011
,          1111100
y          1111101
x          11111100
.          11111101
z          111111100
(          111111101
)          1111111100
j          1111111101
k          1111111110
q          11111111110
/          11111111111
```

Decoder

[12]:
```
def ShannonFanoDecoder(encoded_text, codes):
    decoded_text = ""
    current_code = ""
    for char in encoded_text:
        current_code += char
        for code in codes:
            if codes[code] == current_code:
                decoded_text += code
                current_code = ""
```

```
                break
    return decoded_text


shannon_decoded_text = ShannonFanoDecoder(shannon_encoded_text, codes)
```

[13]:
```
with open("decode_file_shannon.txt", "w") as file:
    file.write(decoded_text)
file.close()
```

comparing decoded stream with the original stream

[14]:
```
def CompareTexts(text1, text2):
  if len(text1) != len(text2):
    return False
  for i in range(0, len(text1)):
    if text1[i] != text2[i]:
      print(i)
      return False
  return True

print("Texts are the same") if CompareTexts(text, shannon_decoded_text) else␣
 ↪print("Texts are not the same")
```

```
Texts are the same
```

Calculating the efficiency of the Shannon-Fano code

[15]:
```
def CalculateEfficiencyShannon():
  average_code_length = 0
  for code in codes:
    average_code_length += len(codes[code]) * char_probability[code]
  return entropy / average_code_length

print("Shannon-Fano code efficiency: ", CalculateEfficiencyShannon())
```

```
Shannon-Fano code efficiency:  0.9948140534108271
```

8-Efficiency Comparison: Shannon-Fanovs. Huffman Codes

[16]:
```
# Comparison Function using pre-computed variables
def compare_coding_methods(fixed_avg_length, huffman_avg_length,␣
 ↪shannon_fano_avg_length,
                           fixed_efficiency, huffman_efficiency,␣
 ↪shannon_fano_efficiency):
    # Display Results
    print(f"{'Method':<15} {'Average Length':<20} {'Efficiency (%)':<20}")
    print(f"{'Fixed-Length':<15} {fixed_avg_length:<20.4f} {fixed_efficiency *␣
 ↪100:<20.2f}")
```

```
    print(f"{'Huffman':<15} {huffman_avg_length:<20.4f} {huffman_efficiency *␣
↪100:<20.2f}")
    print(f"{'Shannon-Fano':<15} {shannon_fano_avg_length:<20.4f}␣
↪{shannon_fano_efficiency * 100:<20.2f}")

# Call the function with the pre-computed values
compare_coding_methods(bits_fixed_length, math.
↪ceil(CalculateEfficiencyHuffman()*entropy), math.
↪ceil(CalculateEfficiencyShannon()*entropy),
                       effeciency_fixed_length, CalculateEfficiencyHuffman(),␣
↪CalculateEfficiencyShannon())
```

```
Method          Average Length       Efficiency (%)
Fixed-Length    6.0000               70.95
Huffman         5.0000               99.55
Shannon-Fano    5.0000               99.48
```

#Part 2

[17]:
```
import numpy as np
```

1.3.1 Encoder

[18]:
```
G = np.array([[1, 1, 0, 1, 0, 0, 0],
              [0, 1, 1, 0, 1, 0, 0],
              [1, 1, 1, 0, 0, 1, 0],
              [1, 0, 1, 0, 0, 0, 1]])
```

[19]:
```
def hammingCodeEncoder(input_text):
  input_text += '0' * (len(input_text) % 4)
  encoded_bits =[]
  for i in range(0,len(input_text),4):
      m = []
      for j in range(4):
          m.append(int(input_text[i+j]))
      c = m @ G
      c = c % 2
      encoded_bits = np.concatenate((encoded_bits,c))
      BPSK = encoded_bits*2-1
  return encoded_bits, BPSK


def AWGN(signal, snr_db):
    snr = 10**(snr_db/10)
    power = np.mean((signal)**2)
    noise_power = power/snr     # snr = power/noise_power
    noise = np.random.normal(0,1,len(signal))
```

```
    noise *= noise_power
    return signal + noise
```

[20]:
```
encoded_bits, BPSK_signal = hammingCodeEncoder(huffman_encoded_text)
noisy_signal = AWGN(BPSK_signal, 10)
```

1.3.2 Decoder

[21]:
```
H = np.array([[1, 0, 0, 1, 0, 1, 1],
              [0, 1, 0, 1, 1, 1, 0],
              [0, 0, 1, 0, 1, 1, 1]])
```

[22]:
```
from pprint import pprint
def generate_error_codes():
  error_codes = {}
  i_mat = np.identity(7, dtype=int)
  i_z = [0, 0, 0, 0, 0, 0, 0]
  syndrome = tuple(i_z @ H.T % 2)
  error_codes[syndrome] = i_z
  for i in range(7):
    syndrome = tuple(i_mat[i] @ H.T % 2)
    error_codes[syndrome] = list(i_mat[i])
  return error_codes

error_codes = generate_error_codes()
# pprint(error_codes)

def hammingCodeDecoder(noisy_signal):
# Threshold the noisy BPSK signal to convert to binary
    received_signal = []
    for value in noisy_signal:
        if value > 0:
            received_signal.append(1)
        else:
            received_signal.append(0)

    decoded_messages = []
    corrected_codewords = []

    for i in range(0, len(received_signal), 7):
      received_codeword = received_signal[i:i+7]
      syndrome = tuple(received_codeword @ H.T % 2)
      error_code = [0, 0, 0, 0, 0, 0, 0]
      if syndrome in error_codes:
        error_code = error_codes[syndrome]
      # print(received_codeword, error_code, syndrome)
```

```python
        received_codeword = [received_codeword[j] ^ error_code[j] for j in␣
 ↪range(7)]
        original_message = received_codeword[3:]
        decoded_messages.append(original_message)
        corrected_codewords.append(received_codeword)

        string  = np.concatenate(decoded_messages).tolist()
        string = list(map(str, string))
        hamming_decoded_messages = ''.join(string)
    return hamming_decoded_messages
```

[23]:

```python
decoded_messages = hammingCodeDecoder(noisy_signal)
print("Decoded Messages:", decoded_messages)
# print("Corrected Codewords:", corrected_codewords)
```

```
Decoded Messages: 0111100110111001111110111010010111100110111100001011001010111 0
11101101100110001110110001111101010000000100011100001100110001111101101110001111 0
00100100010000010111010100001000010100011100010010001111101001100110001100111001
01000001110101100111110011011111010111100011000100000111010011001000011011000111
10001001101010111101000011100000000101111001010100100011110111010110001111000100 1
10101110100101111000110100011100100001000010000101010100100001101010000111010110
11111000010000010111011001000010111000110110110010100111110011111001100001011001
00101010111101110011101110010111100101001111110111001111010000101010011110010101
10101110110101000111100101011011110001101010111001110000000100010111011001000010 1
11100100000101110001011010111011111110111001111001110101000111001011010101011110 11
11001101010111100010101110111011001000010111101011000111111100111111011110011110
01010110101111011111101111010010011001011101101010001111001010110111100011010101 0
00010010000000101110100011111011111001010110001000110010111111011000101101011110
01101110011111101011111101110011111100110111100011010001110010000100001000010100
00001000111000011001100011111011011100011110001001000100000101110101000010000101
11100110101101101111110001110000101101011111000111010110100101111001010110101010
01000111101110101100011110001001101011101001010001011000110011111110101010011110
00111101110011100010111110001101110110010000101010001111111101110011110001010101
11011101100100001011000111000010110110111000010100101011110011011101101000111111
01000010000100000000100111101000100011100101111110110110011001001011001010111011 0
00111001110010000100001000010111001000101110011111100110101011100111000000111110
01010101011101111001110011110011100100001010111100100000111011000100011110010111
11000111010110100101111001010110101110101100001111111101111100001000111100111010
01100010101111110111001111110011011110001101000111001000010000100001010000001000
11100001100110001111101101110001111000100100010000010111010100001000010111110001
11010110100101111001010110101110110010000101000011100000001011110001101000000110 0
10100010001111001010110101000111101100110111011111100100100010100010100110100101 0
10001010000100011110001111110110001111100101010101110111100111001111001110010000
10111001000101111110100001100011011000011000000111100111010110010101010101000111
10010101101111000100010000010111011001000010111000110110110010100111110011101000
10001111000100101000101011101110101001010101110001111011110101110101001010101 10
00110010010110110111110011111101111000100001011100100010111111010000110001101110
```

```
0111111110110011011100010111010001011100011011011001010011110011111001010001011
10110010000010001011111101110011111101101000111010010110111111101110110100011100
01111101110001100011111110011001010001111110110001101110110011101111100101010010
00111010010100011100011101100010110011110010101101010001011000110011111110111001
111110011010000111000000010111100110101101101111110001110000101101011111000111010
11010010111100101011010111011001000010111001111110011011111101101011110011011110
01101110011111110111100011010000001100101000100011110010101100101011101011111001
01011101001010100111111100001011101001101110011111111011100101110011111100110100
01110001111101110011110001111011100111110001010101011000010100101110011111100110
11110001101000111001000010000100001010001110001001000111110100110011000110011100
10100001110000001010111010010101001110100001010100110011011001110001111101111101
11100101010101111101000010111000101101010001110001111101110011010110001000010101
01110011111001111111011100111111001101100011001111110010110101010011001101110000
100010010000101000000100110011111100100010011010110110010100011100010010001111101
0011001100011001110010100001110000000101111110001110011111111000000001000101111110
10011110111011000011001010001111110110001101010000111001011000010100101111011111
11111010110110011100011111001101110001111110110001101010001111111101110101111011
01110001111000100110111011001000010101100101001011111100011101010011101001100011
11110001100111001010110001010011101011110110001001011101011000110111011111001001
00010111100011000101000011001110000100000101011101111010010001111001010110101110
11010001011101010000100010001001100110111000111100010011011110011011011101010010
10101110001111101111010111010100101110110100011100011111011111000101100001101111
10111110101111101100001000100110100001110110110110010001100100001011111101000000
10101111111000110111100110101010111011100100010100111101101101111101110101101110
0001101110110010000101111011011011111001110000110111000011011100111110011010010
10100010100001000111100011111101100011001001010001001011000010100101110011111100
11011110001101000111001000010000100001010000111000000101010010001111011101011000
11110001001101011110011011110110110101100111100011101001001010110001110111110101
010111010000110111100000010101000001100110011101011001111000010001010111010111100
10001110000110111001000011011011100011110010101101010100011110010101101111000100
010000010111011001000011101111111000001110101101111100001000001011101100100001011
10011101110101000111101010000101000101111110111001111110011010100011111111011101
01111011011100011110001001101110110010000101011001010010111111000111010100111010
01100011111100011001110010101100010100111010111101100010010100000011111101000100
10100110001101101110000110100111101110000110011111011110001100111001011111001111
00100111010110011111101111011011100011110001001101111000010110010100100001101111
10110110010001100110110000101001011100111111001101111000110100011100100001000010
000101000111000100100011111010011001100011001110010100001110000000010101110100111
101100011001101100011101011001110000101100101001000011011111100111100101011010100
011111010011001100011001110010101001100110111000010001001000010100000010011001111
1100100010011010110110010111111000111001111111000000000100010101110100000001010011
10001111100001101110011111100110101000111010110100101110101000111000111101100111
00111100101010101111110100001011100010110101000111000111110111001111000111011100
11111000101010010111111
```

```
[24]: # bit_string = ""
# for message in decoded_messages:
```

```
#     for bit in message:
#          bit_string += str(bit)  # Convert each bit to a string and␣
↪concatenate
```

1.3.3 BER vs SNR

[25]:
```
def calculate_BER(original_data, decoded_data):
  original_data = np.array(list(original_data))
  decoded_data = np.array(list(decoded_data))
  errors = np.sum(np.array(original_data) != np.array(decoded_data))
  return errors / len(original_data)
```

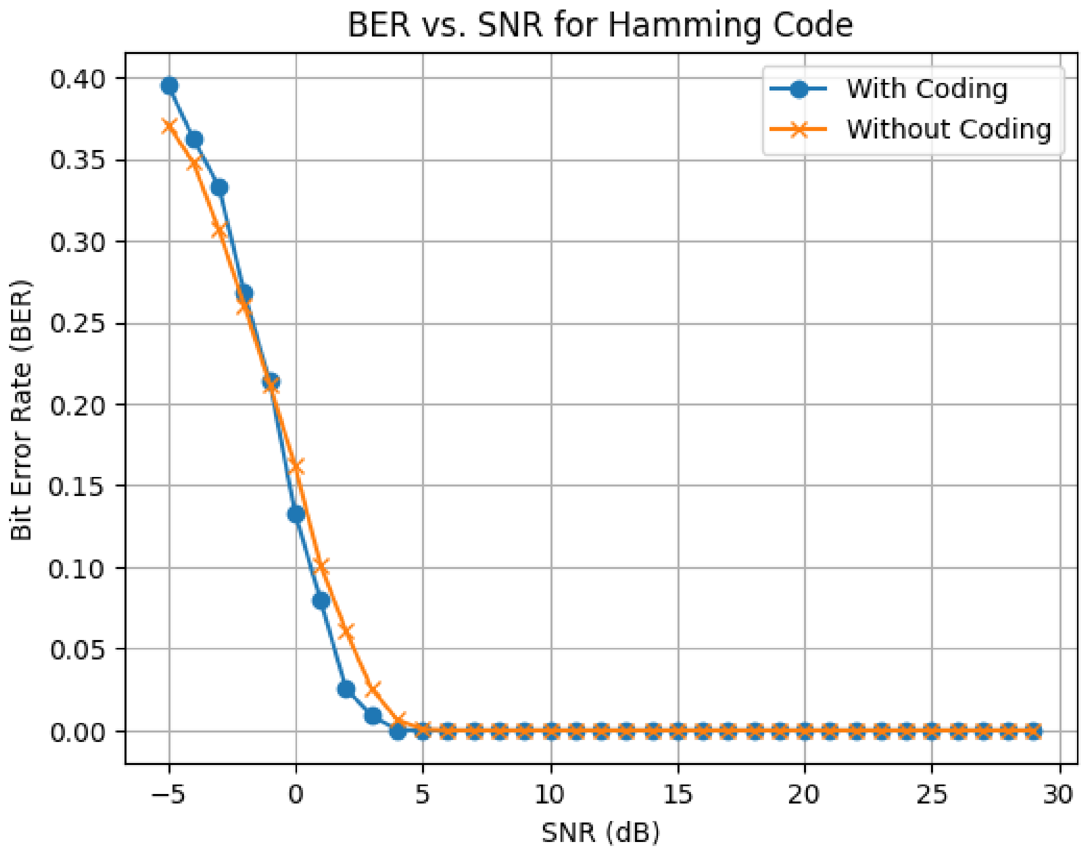
[26]:
```
def transmitnReceiveBits(bits_to_transmit,snr):
    data = np.array(list(bits_to_transmit), dtype=int)
    data_noisy_signal = AWGN(data*2-1, snr) #Convert to BPSK before adding noise
    decoded_data = []
    for value in data_noisy_signal:  #Threshold the noisy signal to get binary␣
↪values
      if value>0:
        decoded_data.append(1)
      else:
        decoded_data.append(0)

    decoded_data = list(map(str, decoded_data))
    decoded_data = ''.join(decoded_data)
    return decoded_data
```

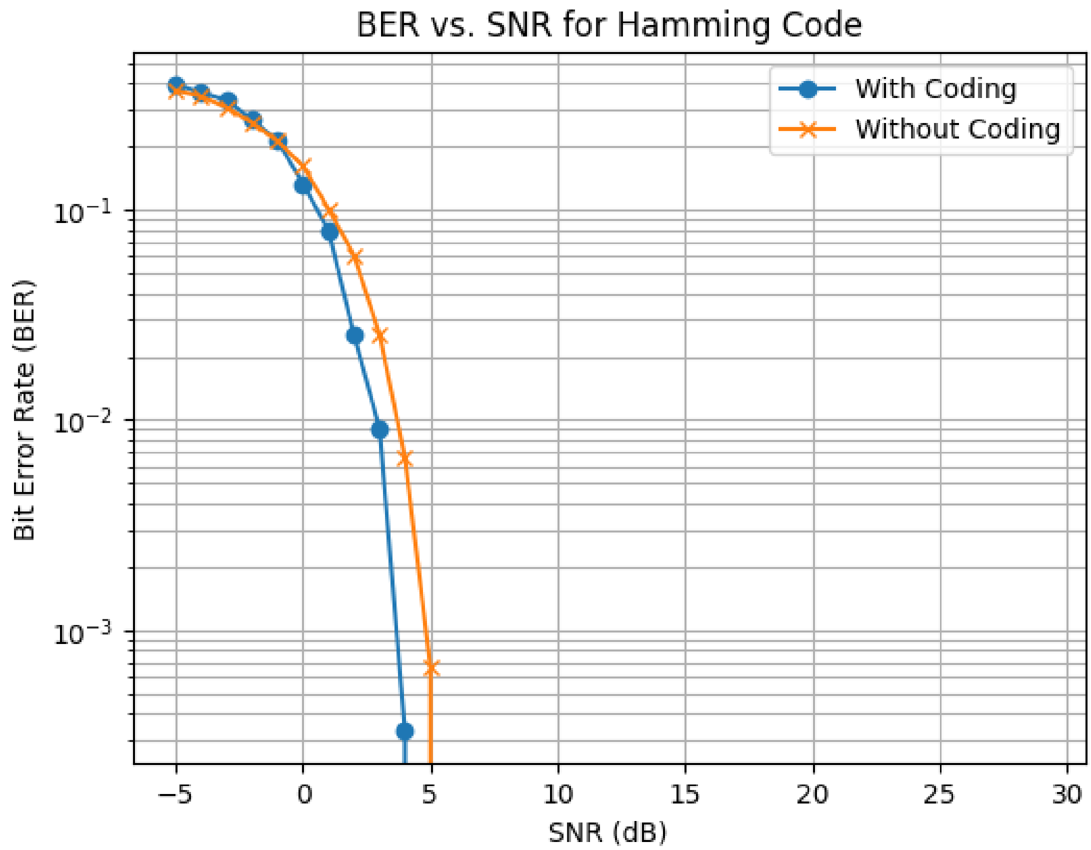
[27]:
```
def transmitnReceiveBitsWithEncoding(bits_to_transmit,snr):
    encoded_data, BPSK = hammingCodeEncoder(bits_to_transmit)
    data_noisy_signal = AWGN(BPSK, snr)
    decoded_data = hammingCodeDecoder(data_noisy_signal)
    return decoded_data
```

[ ]:
```
import matplotlib.pyplot as plt

snr_range = np.arange(-5,30,1)
ber_with_coding=[]
ber_without_coding =[]
# data = huffman_encoded_text;
data = shannon_encoded_text
for snr in snr_range:
  #with Hamming code
  print(snr)
  decoded_data_coding = transmitnReceiveBitsWithEncoding(data,snr)
  ber_with_coding.append(calculate_BER(data, decoded_data_coding))

  #without Hamming code
```

```
    received_data = transmitnReceiveBits(data,snr)
    ber_without_coding.append(calculate_BER(data, received_data))
    #print(ber_with_coding)
```

[29]:
```
#plotting
plt.figure()
plt.plot(snr_range, ber_with_coding, label="With Coding", marker="o")
plt.plot(snr_range, ber_without_coding, label="Without Coding", marker="x")
plt.xlabel("SNR (dB)")
plt.ylabel("Bit Error Rate (BER)")
plt.title("BER vs. SNR for Hamming Code")
plt.legend()
plt.grid(True, which="both")
plt.show()
#plotting
plt.figure()
plt.semilogy(snr_range, ber_with_coding, label="With Coding", marker="o")
plt.semilogy(snr_range, ber_without_coding, label="Without Coding", marker="x")
plt.xlabel("SNR (dB)")
plt.ylabel("Bit Error Rate (BER)")
plt.title("BER vs. SNR for Hamming Code")
plt.legend()
plt.grid(True, which="both")
plt.show()
```


BER vs. SNR for Hamming Code
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
Bit Error Rate (BER)
−5
0
5
10
15
20
25
30
SNR (dB)
With Coding
Without Coding



1.4 End-to-end Communication System

[30]:
```
def countDiffChar(a, b):
    min_len = min(len(a), len(b))
    c = 0

    for i in range(min_len):
        if a[i] != b[i]:
            c += 1

    c += abs(len(a) - len(b))

    return c
countDiffChar(text,text)
```

[30]: 0

Shannon Encoding SNR = 0 dB

[31]:

```python
data = shannon_encoded_text
snr = 0
received_data = transmitnReceiveBits(data,snr)
recieved_text = ShannonFanoDecoder(received_data, codes)
recieved_text_error_1 = countDiffChar(recieved_text, text)

received_data_hamming = transmitnReceiveBitsWithEncoding(data,snr)
recieved_text_hamming = ShannonFanoDecoder(received_data_hamming, codes)
recieved_text_hamming_error_1 = countDiffChar(recieved_text_hamming, text)

ber = calculate_BER(data, received_data)
ber_hamming = calculate_BER(data, received_data_hamming)
print("original text: ")
print(text)
print("\nShannon Encoding, No channel Encoding, SNR = 0 dB")
print(recieved_text)
print("\nShannon Encoding, Hamming channel Encoding, SNR = 0 dB")
print(recieved_text_hamming)
```

```
original text:
in this paper, a novel decorrelation-based concurrent digital predistortion
(dpd) solution is proposed for dual-band transmitters (tx) employing a single
wideband power amplifier (pa), and utilizing just a single feedback receiver
path. the proposed decorrelation-based parameter learning solution is both
flexible and simple, and operates in a closed-loop manner, opposed to the widely
applied indirect learning architecture. the proposed decorrelation-based
learning and dpd processing can also be effectively applied to more ordinary
single-band transmissions, as well as generalized to more than two transmit
bands. through a comprehensive analysis covering both the dpd parameter learning
and the main path processing, it is shown that the complexity of the proposed
concurrent dpd is substantially lower compared with the other state-of-the-art
concurrent dpd methods. extensive set of quantitative simulation and rf
measurement results are also presented, using a base-station pa as well as a
commercial lte-advanced mobile pa, to evaluate and validate the effectiveness of
the proposed dpd solution in various real world scenarios, incorporating single-
band/dual-band tx cases. the simulation and rf measurement results demonstrate
excellent linearization performance of the proposed concurrent dpd, even
outperforming current state-of-the-art methods, despite the significantly lower
complexity.

Shannon Encoding, No channel Encoding, SNR = 0 dB
sn tnao ntepnog poe. cigoaxeoo nuborrst c eeoenos ,dodt onnhtpp mnsgtbh on
zdiosopceb- nrm a-esh,euootrafeobnkonipceat pdpetibalntev.aekrmld mrea limrln
zmenaamreso-rralnetne- eiexuno-, atd pspl o-ptrcyoctdeita rri  oaamont q hoeo
mrcr e ,o-eru aeempraaptiml,rlntpnmctefeeaa  rld dfsga-n ceie gh d u
eaylaepbsm-tnonearhpohhplerit ht , cta e aeia anbhestsuhu te c noi e-neves al
thetb triof ic tuee nrs,wca learnta- aprhhswoparruenagirnsecteitcporrnopuemor-r
```

```
ed ,eesh rcennerteieifroceetolee cp  aeshinainhflos ohh. agi ued to lsro
-mpncssataeeclehesptnetdpse pfs bomheea)mgimaecipmalcmrni  t ilre thaa teso
trpviie ehnfhehascnlnemh enmoiaenoamoasu e ceouhccveve,eds matr gseaenen
ursisetl eoearnpmr phpenpstlia a  ss aadm ssiopbprt  a tnr,rieo ai
t-totoaeegxit)on  osrn iopnsed pocrnsnareatl n ss bmiho e sa enj lower alofapea
tn ah my tar c nhipflceathlatr iilueurren iedeeedrtdolbdiext feedridtaers ko
anartaaeepe srm-latsonen tiohioasto rer dt rstgtti  ar cosom r sncnceuep-moe pe-
rp,stntion pa ts --wiasaie lhyibsaleltlptavnfeonerdfileehaof leemreoe
thtenspehaefdatn lapenireroioati rseln l b ei,rhedtngntagnitiede einueo
edeasasmiweimrhscenarioencntoe zentrunnee sosng-bccywdn ra-n-fi
luirasdroneoapsitdnf nmdeans agime to rlnmniha,ohp rrhdcdcvrto-irliinhe,de
apzamipst nobuptnce of thnerrzoser ntmaoib tg dprx even
oumierformsogmepaiiatttfsrd flpnoniart trthodixenesppsnedi taernrbciadllnirkcr
aew ogmr ty.

Shannon Encoding, Hamming channel Encoding, SNR = 0 dB
in tpis aeeiertir  ch al deygreyer ocfmasdpntyrui rnolhecitalrire
seasmhpoot.lpiooia vd ss, is hwuosed fnr lnhtgemanr enntsds sssndtnsxo-
bprl,snxe apeleirtbis wetm n oclaiel.hfi  i ed a),nihho talcr iele jhhtndtaeytc
nledbaco(recr frr pr f. tcaeiroposelrnemorrelthibormasede  reameter
learnrewraweitiweps vnlaemgo-icmr nndeean-tqigape- re eesh pe emoosedreoonp m
errax oweod   ethms rb tir( a denied indsreca lfad ooeeneio uccifre-aeo e propn
o ieaoetriroaaerd-basep larsh md icienpdhoeoceosing asedadi-
oprthfehtivooosaeipliedasneddai,m rrp. sie-mr-bons e eanena ttmmtofit orool as
gipcispz otoo  car thane yl tranvbarmanlior thizneh a cohoarhol
veiieaddnseaerovde tsioneth the lntgios n ticefteening a ameypmaitti eoe pros
hsingy it se tnr.d thao tmiewsligxiay of efe ialcae  a rnbrprrepe gissis
s-phvfsaolalathyrr cotfared witl the othnneatig-tfp aaocre csonsxerenrrn-n mey
weez gismtaose set o,ekb onptangh tt muwonespe hseoh measuremeit r sults a-se
zppa e dnoal epsrng a base-shirsnmn i taebell ai ttrn(rt, ro s icrenvanmdmnmvrc
uitietpetarsfpset dposalicntes -a vtlcacfxnea br lpveati rmn dpd e sxeudae
trbmeioui rar.en- w  oeenk es,r ncorporttiee ilgoele-bphl(ie eeo-uand tx
isterofeiuisboplgn baaieo uincasfremenl inl ehheplnbtenteee
emrrtolepsenineceizaepxl erbmfhedaiof the pseune mentmseprrent dpdyitr lsg
tpeonrgminh teaahncatnsei,ofmeaaoennsimethodsea des-mcilaatppoaot
paoyosdoowtie.jor.eeoby
```

Shannon Encoding SNR = 3 dB

[32]:
```
data = shannon_encoded_text
snr = 3
received_data = transmitnReceiveBits(data,snr)
recieved_text = ShannonFanoDecoder(received_data, codes)
recieved_text_error_2 = countDiffChar(recieved_text, text)

received_data_hamming = transmitnReceiveBitsWithEncoding(data,snr)
recieved_text_hamming = ShannonFanoDecoder(received_data_hamming, codes)
recieved_text_hamming_error_2 = countDiffChar(recieved_text_hamming, text)
```

```python
ber = calculate_BER(data, received_data)
ber_hamming = calculate_BER(data, received_data_hamming)
print("original text: ")
print(text)
print("\nShannon Encoding, No channel Encoding, SNR = 3 dB")
print(recieved_text)
print("\nShannon Encoding, Hamming channel Encoding, SNR = 3 dB")
print(recieved_text_hamming)
```

```
original text:
in this paper, a novel decorrelation-based concurrent digital predistortion
(dpd) solution is proposed for dual-band transmitters (tx) employing a single
wideband power amplifier (pa), and utilizing just a single feedback receiver
path. the proposed decorrelation-based parameter learning solution is both
flexible and simple, and operates in a closed-loop manner, opposed to the widely
applied indirect learning architecture. the proposed decorrelation-based
learning and dpd processing can also be effectively applied to more ordinary
single-band transmissions, as well as generalized to more than two transmit
bands. through a comprehensive analysis covering both the dpd parameter learning
and the main path processing, it is shown that the complexity of the proposed
concurrent dpd is substantially lower compared with the other state-of-the-art
concurrent dpd methods. extensive set of quantitative simulation and rf
measurement results are also presented, using a base-station pa as well as a
commercial lte-advanced mobile pa, to evaluate and validate the effectiveness of
the proposed dpd solution in various real world scenarios, incorporating single-
band/dual-band tx cases. the simulation and rf measurement results demonstrate
excellent linearization performance of the proposed concurrent dpd, even
outperforming current state-of-the-art methods, despite the significantly lower
complexity.

Shannon Encoding, No channel Encoding, SNR = 3 dB
in thiun rier, a novel deaxcelttion-bcaaaedbaoimescomaoeitariphnheabasptn (ditl,
solution is proposed for dual-bandetransmitters (tx)aooiloying a single wideband
powed rlilifier (palg- and f plizing just a single feeelnrrk receiver path. the
proptsedrnecorrelntion-base-eiarameter learningraw-spont srmoms flexible nnd
sitptix and nunt sniaeie eilosed-loop canneroi ,n ved to the widely applied
indihoet learning architecturoy tme proposed decorrelation-based learning
phptden processieoe can asho be efbecaively applied t-nmce ordinar-ising,,band
transmissions, as weebias generaliw npetpmore thnn two transheeaonihhlf lteough
a compreheasive analysis covering bvf ohe dpl parameter easadingaihpevnedr n pat
ineeocessing, ihe s shown that the complexitos . eoaaneeoposed concurreeeatden
is ssemstantially ltver compardpolala laaehfer sta r-of-the-art concurregatden
methodsoh o-tensive set of qu oen sidefe sit-tpeunn aad rf
measgrementteesultseaie alio presented, usingiiemase-statioo pa as well as a
commemritl lte-advanc d mobile pa, to evtlgatea ndewalidate the effectrveness of
the prxef d dpd solgtio-aeiosarious real -lrld scenarioi, iacoipoven mr single-
```

```
band/dual-bandil- cases,itpe simulation and rftdteoprement results demonstrate
excellent linadnexation performance ob the proposed concuihtratden, even on l
ercegminverurrent statou.oiltopret mnd ods, despite thsnsecnisseran m. lower
comitcxioy.

Shannon Encoding, Hamming channel Encoding, SNR = 3 dB
in this paper, a novel decorrelation-based concurrent digital predistortion
(dpd) solution is proposed for dual-band treesdutters (tx) employing a single
wideband power amplifier (pa), an-eptilizble just a single feedback receiver
ptth. the proposed d,lrr  hen bormases n reameter learning solution is both
fleqoooe and simple, and operates in a closed-loop manner, opposed to the widely
applied indirect learning architecture. the proposed decorrelation-based
learning and dpd processing can also be effectively applied to more ordinary
single-band transmissions, is well as generalized to more than two transmit
bands. through a comprehensive analy at nt. cing both the dpd parameter learning
and the main path processing, it is shown that the complexity of the proposed
concurrent dpd is substantially lower compared with the other state-of-the-art
concurrent dpd methods. extensive set of quantitative simulatihpe hpav
measurement results are also presented, usionae emase-station pa as wwmias a
commercial lte-advanced mobile pa, to evaluate and validate the effectiveness of
the proposeiadpd solution in varg,t aral worhd scenarios, incorporating single-
band/dual-band tx cases. the simulation and rf measurement resissditibonstrate
excellent lineorization performance of the proposed concurrent dpd, even
outperforcing current state-of-the-art methods, despite the significantly lower
complexity.
```

Shannon Encoding SNR = 10 dB

[33]:
```python
data = shannon_encoded_text
snr = 10
received_data = transmitnReceiveBits(data,snr)
recieved_text = ShannonFanoDecoder(received_data, codes)
recieved_text_error_3 = countDiffChar(recieved_text, text)

received_data_hamming = transmitnReceiveBitsWithEncoding(data,snr)
recieved_text_hamming = ShannonFanoDecoder(received_data_hamming, codes)
recieved_text_hamming_error_3 = countDiffChar(recieved_text_hamming, text)

ber = calculate_BER(data, received_data)
ber_hamming = calculate_BER(data, received_data_hamming)
print("original text: ")
print(text)
print("\nShannon Encoding, No channel Encoding, SNR = 10 dB")
print(recieved_text)
print("\nShannon Encoding, Hamming channel Encoding, SNR = 10 dB")
print(recieved_text_hamming)
```

```
original text:
```

in this paper, a novel decorrelation-based concurrent digital predistortion (dpd) solution is proposed for dual-band transmitters (tx) employing a single wideband power amplifier (pa), and utilizing just a single feedback receiver path. the proposed decorrelation-based parameter learning solution is both flexible and simple, and operates in a closed-loop manner, opposed to the widely applied indirect learning architecture. the proposed decorrelation-based learning and dpd processing can also be effectively applied to more ordinary single-band transmissions, as well as generalized to more than two transmit bands. through a comprehensive analysis covering both the dpd parameter learning and the main path processing, it is shown that the complexity of the proposed concurrent dpd is substantially lower compared with the other state-of-the-art concurrent dpd methods. extensive set of quantitative simulation and rf measurement results are also presented, using a base-station pa as well as a commercial lte-advanced mobile pa, to evaluate and validate the effectiveness of the proposed dpd solution in various real world scenarios, incorporating single-band/dual-band tx cases. the simulation and rf measurement results demonstrate excellent linearization performance of the proposed concurrent dpd, even outperforming current state-of-the-art methods, despite the significantly lower complexity.

Shannon Encoding, No channel Encoding, SNR = 10 dB
in this paper, a novel decorrelation-based concurrent digital predistortion (dpd) solution is proposed for dual-band transmitters (tx) employing a single wideband power amplifier (pa), and utilizing just a single feedback receiver path. the proposed decorrelation-based parameter learning solution is both flexible and simple, and operates in a closed-loop manner, opposed to the widely applied indirect learning architecture. the proposed decorrelation-based learning and dpd processing can also be effectively applied to more ordinary single-band transmissions, as well as generalized to more than two transmit bands. through a comprehensive analysis covering both the dpd parameter learning and the main path processing, it is shown that the complexity of the proposed concurrent dpd is substantially lower compared with the other state-of-the-art concurrent dpd methods. extensive set of quantitative simulation and rf measurement results are also presented, using a base-station pa as well as a commercial lte-advanced mobile pa, to evaluate and validate the effectiveness of the proposed dpd solution in various real world scenarios, incorporating single-band/dual-band tx cases. the simulation and rf measurement results demonstrate excellent linearization performance of the proposed concurrent dpd, even outperforming current state-of-the-art methods, despite the significantly lower complexity.

Shannon Encoding, Hamming channel Encoding, SNR = 10 dB
in this paper, a novel decorrelation-based concurrent digital predistortion (dpd) solution is proposed for dual-band transmitters (tx) employing a single wideband power amplifier (pa), and utilizing just a single feedback receiver path. the proposed decorrelation-based parameter learning solution is both flexible and simple, and operates in a closed-loop manner, opposed to the widely applied indirect learning architecture. the proposed decorrelation-based

learning and dpd processing can also be effectively applied to more ordinary single-band transmissions, as well as generalized to more than two transmit bands. through a comprehensive analysis covering both the dpd parameter learning and the main path processing, it is shown that the complexity of the proposed concurrent dpd is substantially lower compared with the other state-of-the-art concurrent dpd methods. extensive set of quantitative simulation and rf measurement results are also presented, using a base-station pa as well as a commercial lte-advanced mobile pa, to evaluate and validate the effectiveness of the proposed dpd solution in various real world scenarios, incorporating single-band/dual-band tx cases. the simulation and rf measurement results demonstrate excellent linearization performance of the proposed concurrent dpd, even outperforming current state-of-the-art methods, despite the significantly lower complexity.

[34]:

```
print("Shannon Encoding, No channel Encoding, SNR = 0 dB")
print("number of erroneous symbols: ", recieved_text_error_1)
print("error percentage: ", round(recieved_text_error_1/len(text),2)*100)
print("\n")

print("Shannon Encoding, Hamming channel Encoding, SNR = 0 dB")
print("number of erroneous symbols: ", recieved_text_hamming_error_1)
print("error percentage: ", round(recieved_text_hamming_error_1/
 ↪len(text),2)*100)
print("\n")


print("Shannon Encoding, No channel Encoding, SNR = 3 dB")
print("number of erroneous symbols: ", recieved_text_error_2)
print("error percentage: ", round(recieved_text_error_2/len(text),2)*100)
print("\n")

print("Shannon Encoding, Hamming channel Encoding, SNR = 3 dB")
print("number of erroneous symbols: ", recieved_text_hamming_error_2)
print("error percentage: ", round(recieved_text_hamming_error_2/
 ↪len(text),2)*100)
print("\n")


print("Shannon Encoding, No channel Encoding, SNR = 10 dB")
print("number of erroneous symbols: ", recieved_text_error_3)
print("error percentage: ", round(recieved_text_error_3/len(text),2)*100)
print("\n")

print("Shannon Encoding, Hamming channel Encoding, SNR = 10 dB")
print("number of erroneous symbols: ", recieved_text_hamming_error_3)
print("error percentage: ", round(recieved_text_hamming_error_3/
 ↪len(text),2)*100)
```

```python
print("\n")
```

```
Shannon Encoding, No channel Encoding, SNR = 0 dB
number of erroneous symbols:  1315
error percentage:  94.0


Shannon Encoding, Hamming channel Encoding, SNR = 0 dB
number of erroneous symbols:  1312
error percentage:  93.0


Shannon Encoding, No channel Encoding, SNR = 3 dB
number of erroneous symbols:  1094
error percentage:  78.0


Shannon Encoding, Hamming channel Encoding, SNR = 3 dB
number of erroneous symbols:  979
error percentage:  70.0


Shannon Encoding, No channel Encoding, SNR = 10 dB
number of erroneous symbols:  0
error percentage:  0.0


Shannon Encoding, Hamming channel Encoding, SNR = 10 dB
number of erroneous symbols:  0
error percentage:  0.0
```