Information Theory
Project 2
Convolutional Coding/Decoding

Summary of Project:

This project is related to Viterbi decoding, used for decoding the corrupted data after passing through the Binary Symmetric Channel. It has two main parts : one is encoding bit sequence using given code and then decoding using the Viterbi decoding. First 'K' bits sequence is generated randomly with equal probability and encoded with the given code. This encoded bits are then passed through BSC with certain error probability to generate the corrupted bit sequence. Viterbi decoding is used for decoding these corrupted bit sequence in order to get the original bit sequence. After decoding, error in decoding is noted and the graph of BER vs error probability after decoding is plotted.

Generator Matrix:

The project is for (3,1,2) convolutional code with the generator matrix given below:

$$G(D) = [\ 1+D,\ 1+D^2,\ 1+D+D^2\ ]$$

State Diagram:

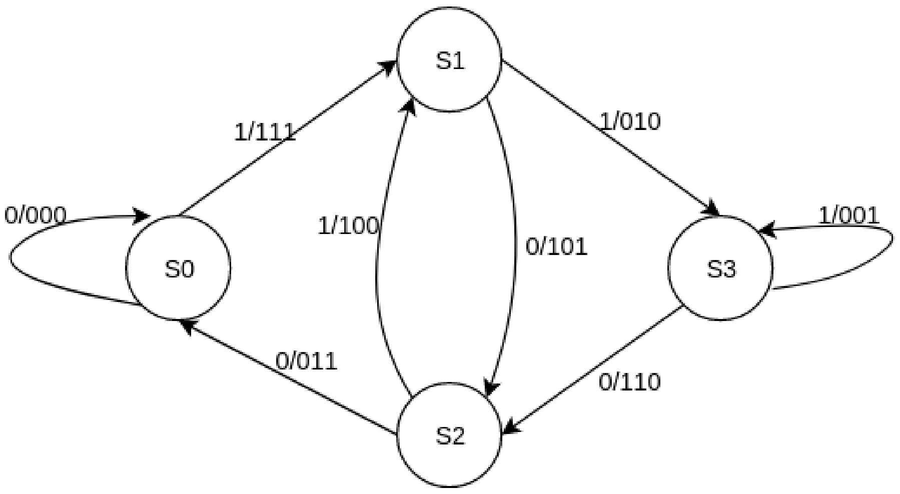


Fig 1: state diagram for given generation matrix

Coding Language: Python

Code Description:

The code is divided into two parts:

1. v**iterbi.py –** It has the class named 'viterbi' that implement the viterbi decoding along with the encoding and BSC

2. **main.py** - Here, state diagram is defined and input bit sequences are generated and passed to viterbi class for encoding and decoding. The graph for BER vs error probability is also plotted.

Functions in viterbi.py:

1. **encode(self, msg_bits):** This function returns encoded bit sequence of the original message bits denoted by **msg_bits**. The encoding is performed through the state machine. Initially, the state is assumed to be 'S0'. Then for each bit in msg_bits the output state and output bits are tracked and the sequence of output bits are stored in **encoded_bits** which represents the encoded bits for original message bits.

2. **hamming_distance(self, code1, code2):**

The hamming distance of two codes **code1** and **code2** is calculated in this function. For this each bit of two codes are compared and if they are not equal hamming distance is increased by 1.

3. **backward_connected_vertex(self, current_state):**
   Given the current state this function provides the information of the previous state and the output obtained on transition from previous state to current state. The function iterates through all the states in state machine and tries to find the current state in tuple. On getting the current state, it returns the corresponding key state and output value.

4. **next_state(self, prev_state):**
   It returns the states that are possible in next step for the given state.

5. **msg_bit_extract(self, prev_state, current_state):**
   The main objective of this function is to give the information of the input or message bit for the given transition from the previous state to current state. For the two possible states of the previous state, it search for the matching current state and passed the value.

6. **decode(self, received_bits):**
   This function implement the concept of Viterbi decoding. There are two main parts in decoding. First maintains the path in trellis tree by observing cost and state and next is traceback the possible path.
   At first the received bits are divided into group of three bits (as this code has three output streams). Then, for each received bits stream, two tables are maintained- cost and state table. For the cost table, hamming distance of the received bit and the output bits for the two possible paths for each state is compared and the minimum of two are recorded in table. The state table (named as trace_scr in code) saves the state information from which the current state is arrived i.e. saves the previous state for which the arrival cost or weight is minimum. hamming_distance() and backward_connected_vertex() functions are used for this purpose.
   After getting complete cost and state table, trace back is started from the state S0 of the last step. The previous state information is extracted from state table for the current state and by using the state information of current and previous state original message is extracted using function msg_bit_extract().
   Then, decoded message bit sequence are returned.

7. **error(self, original_msg_bits, decoded_msg_bits):**
   This function calculates the errors in between original message bit and decoded message bits. This is done by comparing two bit sequence by calling '**hamming_distance(self, code1, code2)**' function. The error is calculated as:

$$\text{error} = \frac{hamming\,distance\left(msg\,bit\,,decoded\,bit\right)}{length\,of\,msg\,bit}$$

8. **add_noise(self, encoded_bits, pe):**
   The function is implementation for the BSC channel. First, the flip-bits equal to length of encoded bits are generated randomly for the given BER (pe) and for flip bits = 1 , the encoded bits are flipped, otherwise left as it is. These encoded bits after adding noise are saved as received_bits.

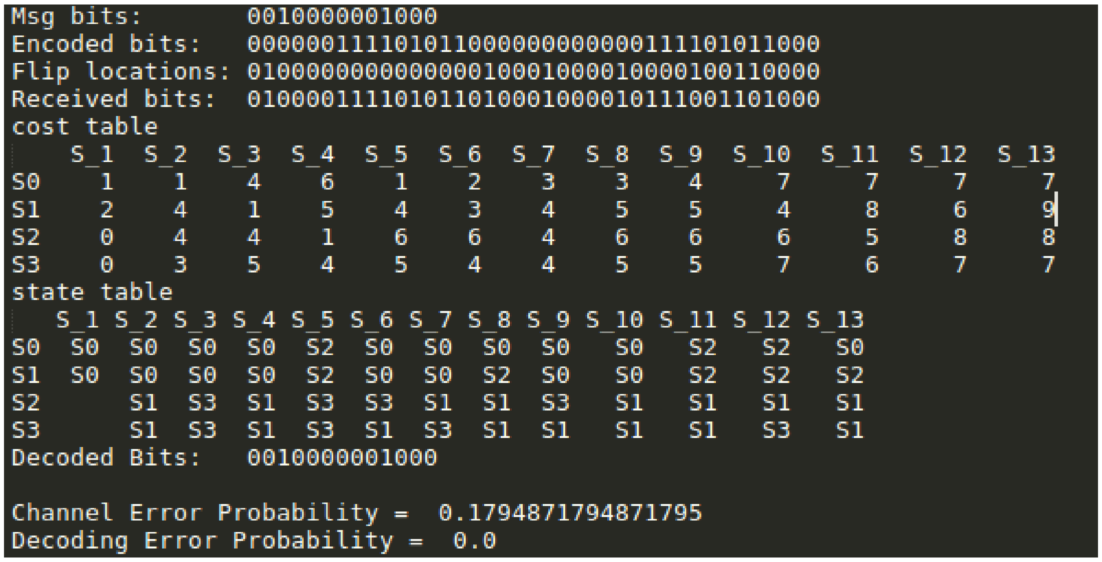
State Machine:

The state machine is defined in the main.py in the form of python dataframe as follows:

```
g = {
        'S0': [('S0', '000'), ('S1', '111')],
        'S1': [('S2', '101'), ('S3', '010')],
        'S2': [('S0', '011'), ('S1', '100')],
        'S3': [('S2', '110'), ('S3', '001')],
        }
```

Here, 'S0': [('S0', '000'), ('S1', '111')] represents ‘S0’ as the current state with two possible next states ‘S0’ and ‘S1’ with their corresponding outputs ‘000’ and ‘111’ respectively. Also first element in array i.e. ('S0', '000') is obtained when the input message is ‘0’ and next element ('S1', '111') otherwise. It is true for all other elements also.

Result for K = 10 bits for one instance:
The screenshot of output of the program for small message bit sequence is as follows:



```
Msg bits:       0010000001000
Encoded bits:   000000111101011000000000000000111101011000
Flip locations: 010000000000000001000100001000010011000 0
Received bits:  010000111101011010001000010111001101000
cost table
    S_1  S_2  S_3  S_4  S_5  S_6  S_7  S_8  S_9  S_10  S_11  S_12  S_13
S0    1    1    4    6    1    2    3    3    4     7     7     7     7
S1    2    4    1    5    4    3    4    5    5     4     8     6     9
S2    0    4    4    1    6    6    4    6    6     6     5     8     8
S3    0    3    5    4    5    4    4    5    5     7     6     7     7
state table
   S_1 S_2 S_3 S_4 S_5 S_6 S_7 S_8 S_9 S_10 S_11 S_12 S_13
S0  S0  S0  S0  S0  S2  S0  S0  S0  S0   S0   S2   S2   S0
S1  S0  S0  S0  S0  S2  S0  S0  S2  S0   S0   S2   S2   S2
S2      S1  S3  S1  S3  S3  S1  S1  S3   S1   S1   S1   S1
S3      S1  S3  S1  S3  S1  S3  S1  S1   S1   S1   S3   S1
Decoded Bits:   0010000001000

Channel Error Probability =  0.1794871794871795
Decoding Error Probability =  0.0
```

The output is for following constrains:
length of message bits, k = 10 bits
flush bits = 3
BER = 0.2

The Msg bits represent the message bits with 3 flush bits appended at the end. Encoded bits are bit sequence after passing through the convolutional code (3,2,1). It is observed that the length of encoded bit is 3 times the message bits. Flip locations denotes the locations where the encoded bits are flipped after passing through BSC. These flip location streams are generated randomly with BER = 0.2. Received bits are bits received after passing through channel which has flipped bits or errors.
The cost and state table store the cost and previous state for transition from one state to next respectively. Both table represent state in each row and time or step number in each columns. If we see the first row in cost table, then it represents the cost or weight in order to get to that state in each step.

Similarly, if we see the state table, first row represents the corresponding previous state from which the state S0 is reached at that step.
For decoding, state table is traced back starting from S0 from last step (S_13 for this example). The value for last step is S0, so it goes back to S0 row for S_12 step and also extract the message bits for transition from S0 to S0 state from state machine. The process continues for all the steps and decoded bits are generated represented by Decoded Bits in output shown above.
Channel and decoding error probability provides the information about the error made by channel and decoding algorithm respectively. It is observed that although the channel has the error of 0.179 decoding is performed without any error.

Result for K = 1024 bits:
The graph for BER (Pb) of channel vs error probability after decoding (Pe, in log scale) is plotted to observe the performance of the viterbi decoding. The parameters used to obtain the graph is as follows:
message bit length, K = 1024 bits
flush bits = 3 bits
BER range, Pb = 0 to 0.5 at step size of 0.01
number of iteration = 100 times

Table for average Pe obtained for different Pb after 100 iterations:

Pb	Pe
0.0	0.0
0.01	0.0
0.02	0.0
0.03	0.0001
0.04	0.0003
0.05	0.0007
0.06	0.0015
0.07	0.0032
0.08	0.0058
0.09	0.0089
0.1	0.0129
0.11	0.0187
0.12	0.0265
0.13	0.0389
0.14	0.0495
0.15	0.066
0.16	0.08
0.17	0.0982
0.18	0.1173
0.19	0.1412
0.2	0.1639
0.21	0.1883
0.22	0.2079
0.23	0.2312
0.24	0.2562
0.25	0.2741
0.26	0.2956
0.27	0.3178

Pb	Pe
0.28	0.3434
0.29	0.36
0.3	0.3711
0.31	0.3933
0.32	0.4068
0.33	0.4131
0.34	0.4283
0.35	0.4379
0.36	0.4529
0.37	0.4583
0.38	0.4638
0.39	0.4686
0.4	0.481
0.41	0.4814
0.42	0.4862
0.43	0.4869
0.44	0.4917
0.45	0.493
0.46	0.4969
0.47	0.4985
0.48	0.4996
0.49	0.4991

The graph shows that error probability increase as BER of the channel increase. This means algorithm can correct the code with less error and as error in code increases, message bits are decoded with certain error.

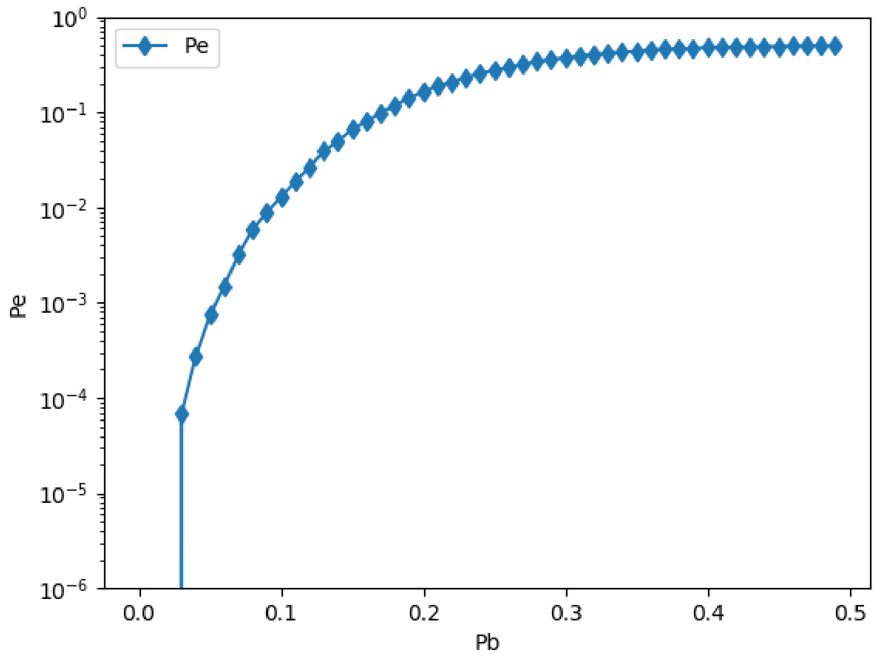


Fig 2 : Graph showing average decoding error probability, Pe in log scale for different BER, Pb of BSC channel after 100 iterations