**Experiment No:** 02

**Experiment Name:** Write a program to simulate convolutional coding based on their encoder structure.

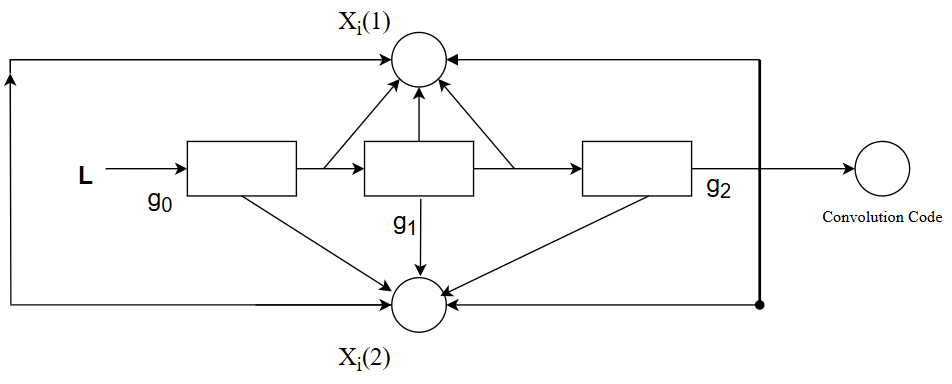
**Theory:**

**Convolution Coding:**  
Convolution codes, or Trellis Code, introduce memory into the coding process to improve the error-correcting capabilities of the codes. The coding and decoding processes that are applied to error-correcting block codes are memoryless. The encoding and decoding of the block depend only on that block and is independent of any other block. They do this by making the parity-checking bits dependent on the bit values in several consecutive blocks.

Say, we have a message source that generates a sequence of information digits ​. We will assume that the information digits are binary, i.e., information bits. These information bits are fed into a convolution encoder. As an example, consider the encoder shown below. This encoder is a finite-state machine that has (a finite) memory.

Its current output depends on the current input and on a certain number of past inputs. In the example, its memory is 2 bits, i.e., it contains a shift-register that keeps stored the values of the last two information bits. Moreover, the encoder has several modulo-2 adders. The output of the encoder is the codeword bits that will be transmitted over the channel. In our example, for every information bit, two codeword bits are generated. Hence the encoder rate is =1/2 bits.

In general, the encoder can take ​ information bits to generate codeword bits, yielding an encoder rate of = / bits.



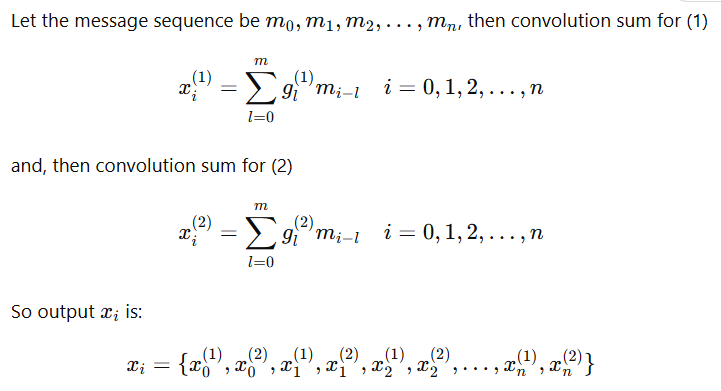
To make sure that the outcome of the encoder is a deterministic function of the sequence of input bits, we ask the memory cells of the encoder to contain zeros at the beginning of the encoding process. Moreover, once information bits have been encoded, we stop the information bit sequence and will feed T dummy zero-bits as inputs instead, where T is chosen to be equal to the memory size of the encoder. These dummy bits will make sure that the state of the memory cells are turned back to zero. Here in the above diagram,

L = the message length,

m = number of shift registers,

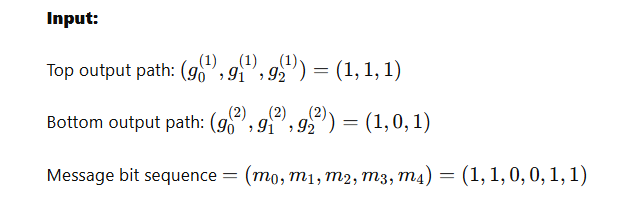
n = number of modulo-2 adders.

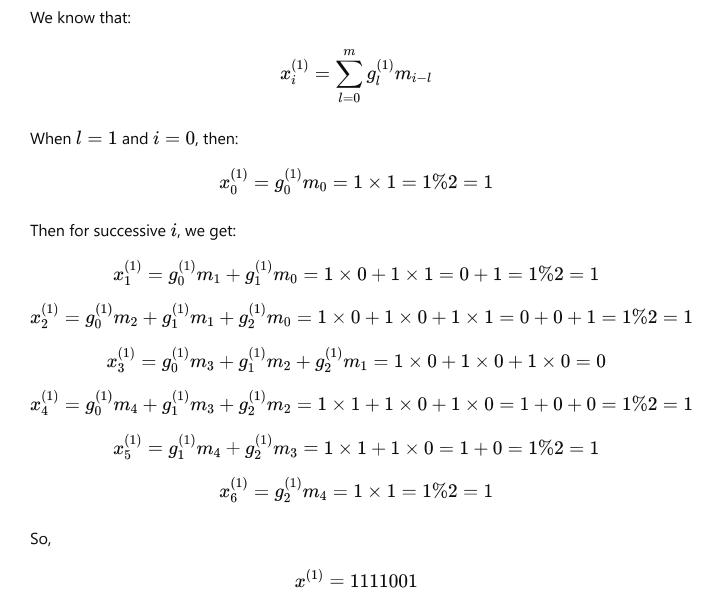
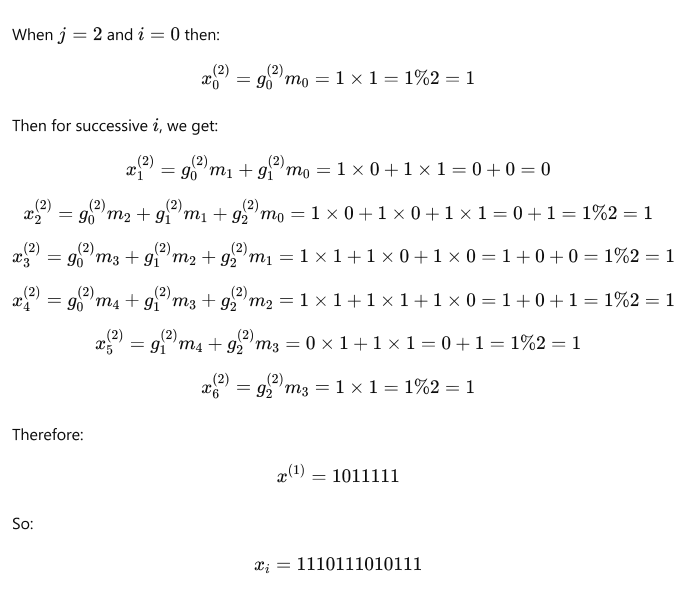
Output = n(m +L) bits, and code rate,

**Constraint length,**  . If (1)​,,------ are the states of shift register, then the input-top adder output path is given by,

(1)​,,------are the state of the shift register, then the input-bottom adder output path is given by (1)​,,------

**Math:**





**Python Source Code:**

|  |
| --- |
|  |

**Experiment No:** 03

**Experiment Name:** Write a program to implement Lempel-Ziv code.

**Theory:**

Lempel-Ziv is a universal lossless data compression algorithm created by Abraham Lempel, Jacob Ziv. It was the algorithm of the widely used Unix file compression utility compress and is used in the GIF image format.

Lempel-Ziv algorithm is accomplished by parsing the source data stream into segments that are the shortest subsequences not encountered previously. To illustrate, let us consider an input binary sequence as follows: 000101111001010010001 ...

Let us assume 0 and 1 are already stored so, subsequence store: 0, 1

Data to be parsed: 000101110001...

The encoding process begins at left. As 0 and 1 are already stored, the shortest subsequence of data stream one written as

**subsequence stored**: 0, 1, 00

**Data to be parsed**: 01011110010100101...

The second and the next sequences are:

**Subsequence stored**: 0, 1, 00, 01

**Data to be parsed**: 01110010100101...

We continue this until the given data stream is completely parsed. Now the binary code blocks of the sequences are:

* **Numerical positions**: 1 2 3 4 5 6 7 8 9
* **Subsequence**: 0 1 00 01 011 10 010 100 101
* **Numerical representation**: 11 12 42 21 41 61 62
* **Binary encoded blocks**: 0010 0011 1001 0100 1000 1100 1101

**Figure**: Illustrating the encoding process performed by the Lempel-Ziv algorithm.

From the figure, the first row shows the numerical position of individual subsequences in the code. A sequence of data stream 010 consists of the concatenation of the subsequence 01 in position 4 and symbol 0 in position 1; hence the numerical representation is 41. Similarly others are.

The decoder is just simple as the encoder. Use the pointer to identify the root subsequence and appends the innovation symbol. Such as, the binary block encoded block 1101 in position 9. The last bit is the innovation symbol.

In contrast to Huffman coding, the Lempel-Ziv algorithm uses fixed length codes to represent a variable number of source system.

**Python Source Code:**

|  |
| --- |
|  |