American Journal of Operations Management and Information Systems
2026; 09(3): 120-137
http://www.sciencepublishinggroup.com/j/ajomis
doi: 10.11648/j.ajcst.20260903.13

A Comparative Performance Analysis of Source and Channel Coding Techniques for Digital Communication Systems

Anis Zebiane*

Department of Electrical, Computer, and Communication Engineering (ECCE), Notre Dame University - Louaize, Deir El Kamar, Lebanon

Email address:
abzebiane@ndu.edu.lb (Anis Zebiane)

To cite this article:
Anis Zebiane. (2026). A Comparative Performance Analysis of Source and Channel Coding Techniques for Digital Communication Systems. *American Journal of Operations Management and Information Systems, 09*(3), 120-137. https://doi.org/10.11648/j.ajcst.20260903.13

Received: 12 October 2025; **Accepted:** 12 October 2025 **Published:** 23 September 2026

Abstract: This research investigates and compares the performance of five major coding techniques - Huffman, Run-Length Encoding, Arithmetic, Convolutional, and Bose-Chaudhuri-Hocquenghem (BCH) coding - within a complete digital communication system comprising source coding, channel coding, Binary Phase Shift Keying (BPSK) modulation, and transmission over an Additive White Gaussian Noise (AWGN) channel. The study evaluates these methods in terms of compression efficiency, error correction capability, and overall system reliability under different signal-to-noise ratio (SNR) conditions. MATLAB simulations were conducted for both text and image data to quantify compression ratios and bit error rate (BER) performance across SNR values ranging from 0 dB to 24 dB. Results demonstrate that Run-Length Encoding achieves superior compression performance for highly repetitive text, whereas Arithmetic coding provides near-optimal compression efficiency for general, non-repetitive data distributions such as natural images. Among channel coding schemes, Convolutional codes exhibit better resilience to noise at low-to-moderate SNR levels compared to BCH codes, particularly when decoded using the Viterbi algorithm, while BCH codes retain an advantage where guaranteed multiple-error correction within fixed-length blocks is required. Moreover, integrating Arithmetic coding with Convolutional coding into a single hybrid pipeline enhances end-to-end robustness by balancing data reduction and error resilience, achieving a bit error rate of zero at SNR levels of 4 dB and above while sustaining channel capacities exceeding 1.8 Mbps. These findings provide valuable insights into the optimal combination of source and channel coding strategies for modern digital communication systems, emphasizing the trade-offs between computational complexity, compression efficiency, and transmission reliability, and offering practical guidance for system designers working on bandwidth-constrained or noise-limited communication links.

Keywords: Source Coding, Channel Coding, Huffman Coding, Arithmetic Coding, Run-Length Encoding, Convolutional Codes, BCH Codes, BPSK Modulation

1. Introduction

In computer science and information theory, data compression, source coding, or bit-rate reduction involves encoding information using fewer bits than the original representation. The project aims to show the complete digital communication system, from the input data to the outputted data as described in Figure 1.

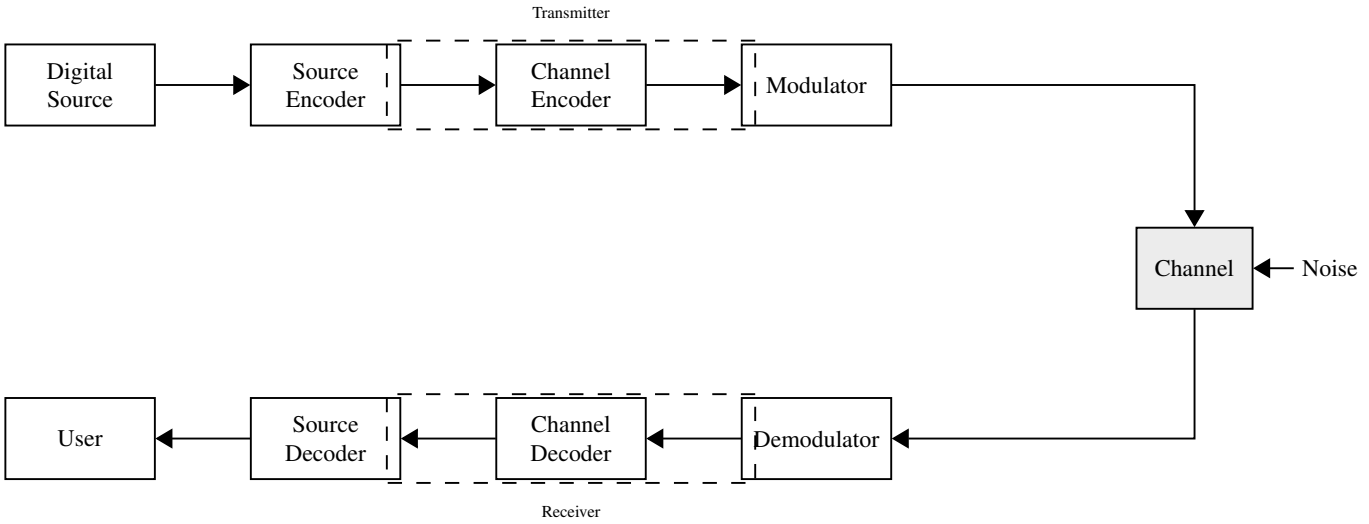


Figure 1. *A Complete Digital Communication System.*

Communication is distributed all around the world and having the ability to understand it, can open the door to additional improvements and modifications in the field. In the next sections we are gonna dig inside every block of the communication system presented in Figure 1.

2. Basic Theory

2.1. Source Coding

Source coding, also known as data compression, is a very important block of the digital communication system. There are two sorts of compression techniques: lossless (Huffman, Arithmetic, Run-Length, Lempel-Ziv...) and lossy (Discrete Cosine Transform - DCT, DWT ...). When a lossless compression method is applied, the reproduced picture is the same as the input image. The images are first compressed using lossless image methods, turning them into picture pixels. Then, each and every pixel is handled separately. Part of the first stage involves estimating the value of the subsequent picture pixel from the nearby pixels. In the second stage, the difference between the actual intensity of the next pixel and its expected value is encoded using multiple encoding techniques. A higher compression ratio is obtained using lossy compression as opposed to lossless compression. The compressed image produced by this method is not precisely the same as the original image; there is some loss of information. A large amount of information can be easily removed from an image when using lossy compression. Our main goal in this paper will be to focus on lossless source coding / compression techniques.

2.2. Channel Coding

A key component of reliable wireless signal transmission is channel coding. Wireless transmission uses non-guided media, such as the air or vacuum (space), as opposed to cable transmission, which uses a physical channel (there is no physical link in the transmission) As a result, while transmitting data, non-guided mediums like air and vacuum settings introduce large noise. In other words, transmission across a noisy channel is a common term for wireless communication. A transmitted message may have part of its bits reversed or altered due to this noisy channel. The message that is received differs from the one that was originally broadcast due to these bit alterations. In other words, a receiver experiences message failures due to a noisy channel.

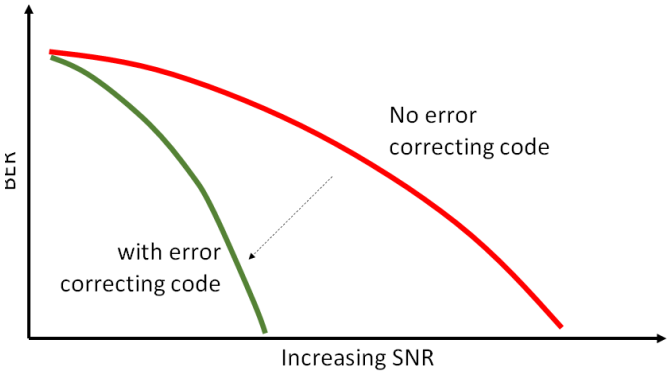


Figure 2. *BER vs. SNR with and without error correction.*

One of the best ways to deal with received message errors caused by bit flipping or changes during transmission on a noisy wireless channel is to employ channel coding. The original message can be recovered by using channel coding to fix the incorrect bits in a received message. This is the reason the program is frequently referred to as ”error correcting code.” There are two main types of channel codes, namely block codes such as Hamming codes, Golay codes, Bose-Chaudhuri-Hocquenghem (BCH) codes, and Reed Solomon codes (uses nonbinary symbols) and convolutional codes such as trellis coded modulation (TCM) and turbo codes.

2.3. Binary Phase Shift Keying (BPSK) Modulation - Demodulation with Additive White Gaussian Noise (AWGN)

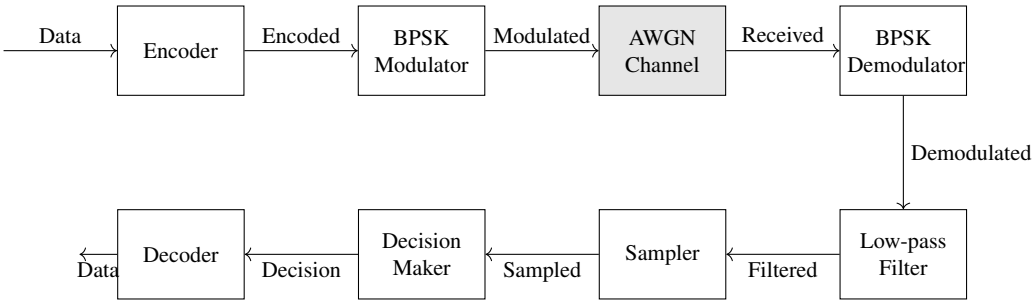


Figure 3. *Block diagram of the BPSK modulation and demodulation process.*

2.3.1. BPSK Modulation

Binary Phase Shift Keying (BPSK) is a digital modulation scheme where each bit in the binary data is represented by one of two phases of a carrier signal. The process can be broken down into the following steps:

1) *Mapping Data to Phases:* Each bit of the binary data is mapped to one of two possible phases of the carrier signal. Typically, a binary '1' is mapped to a phase of $0°$ (or 0 radians), and a binary '0' is mapped to a phase of $180°$ (or π radians).
2) *Modulation:* The carrier signal (a high-frequency sinusoidal wave) is modulated by changing its phase according to the mapped binary data. For example:
   a) For a bit '1', the carrier signal remains unchanged.
   b) For a bit '0', the carrier signal is inverted (phase shifted by $180°$).
3) *Transmission:* The phase-modulated carrier signal is transmitted over the communication channel.

The figure below illustrates the BPSK modulation process:

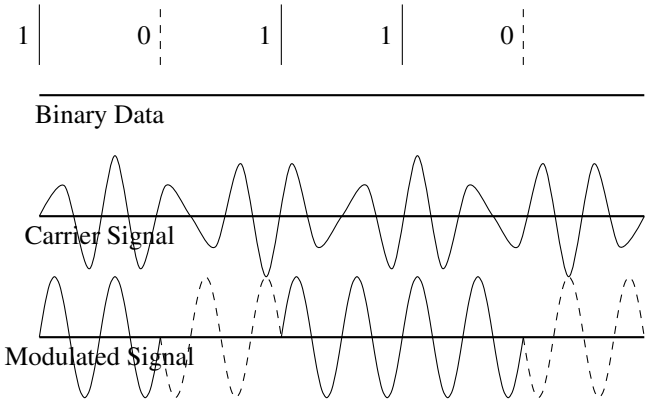


Figure 4. *BPSK Modulation Process.*

2.3.2. BPSK Demodulation

The demodulation process involves the following steps:

1) *Reception:* The modulated signal is received over the communication channel. Due to the presence of noise (e.g., Additive White Gaussian Noise, AWGN), the received signal is corrupted.
2) *Coherent Detection:* The received signal is multiplied by a reference carrier signal (synchronized with the transmitter's carrier). This process converts the phase variations into amplitude variations.
3) *Low-pass Filtering:* The signal is passed through a low-pass filter to remove high-frequency components, leaving a baseband signal that corresponds to the original binary data.
4) *Decision Making:* The filtered signal is sampled, and a decision circuit determines the binary value of each bit based on the amplitude:
   a) If the amplitude is positive, the bit is '1'.
   b) If the amplitude is negative, the bit is '0'.

The figure below illustrates the BPSK demodulation process:

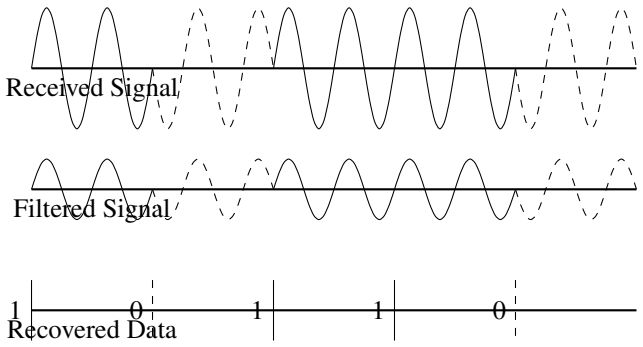


Figure 5. *BPSK Demodulation Process.*

2.3.3. Effect of Noise

Additive White Gaussian Noise (AWGN) is a common noise model used in communications. It is characterized by a constant spectral density and a Gaussian distribution of amplitude. The presence of AWGN in the communication channel affects the received signal by introducing random variations in amplitude and phase. This noise can cause errors in the demodulated data, leading to a higher Bit Error Rate (BER).

In BPSK, the effect of AWGN can be visualized by observing the received signal constellation. Ideally, without noise, the received signal points would be located at distinct positions (e.g., $+1$ and -1). However, with AWGN, these points are spread out around their ideal positions, making it harder to accurately determine the original transmitted bits.

The figure below illustrates the effect of AWGN on the BPSK signal:

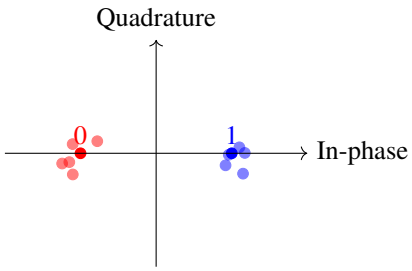


Figure 6. *Effect of AWGN on BPSK*

In the above figure, the ideal constellation points are shown in solid colors, while the noise-affected points are shown with some transparency to indicate the spreading effect of AWGN. The closer the noise-affected points are to the ideal points, the lower the BER, and vice versa. Techniques such as filtering and error correction coding can help reduce the impact of AWGN and improve the accuracy of the demodulated data.

3. Analysis

The main aim of this project is to compare two source coding techniques and two channel coding techniques. Out of the previously stated techniques, Huffman, Arithmetic and Run-Length Encoding (RLE) are the chosen source coding techniques to be compared, whereas for the channel coding we chose TCM and BCH algorithms.

3.1. Huffman Coding

Huffman coding, a widely used entropy encoding method for lossless data compression, was introduced by David A. Huffman in his paper ”A Method for the Construction of Minimum-Redundancy Codes” [2]. It constructs a variable-length code table where more frequent symbols are represented by shorter codewords, leading to reduced data size.

3.1.1. Algorithm

The Huffman algorithm constructs a binary tree based on symbol probabilities or frequencies:

1) *Initialization:* Create leaf nodes for each symbol, each with a priority based on its frequency (lower frequency = higher priority).
2) *Iteration:*
   a) Remove the two nodes with the highest priority from the queue.
   b) Combine them into a new internal node whose frequency is the sum of the two nodes' frequencies.
   c) Insert the new node back into the priority queue.
3) *Termination:* Repeat the iteration step until only one node remains (the root node).

The code for each symbol is obtained by traversing the tree from the root to the corresponding leaf, assigning ’0’ for a left branch and ’1’ for a right branch.

3.1.2. Decoding Process

Decoding Huffman-encoded data involves traversing the Huffman tree from the root to the leaves. Starting at the root, each ’0’ in the encoded data directs to the left child, and each ’1’ directs to the right child, until a leaf node (character) is reached.

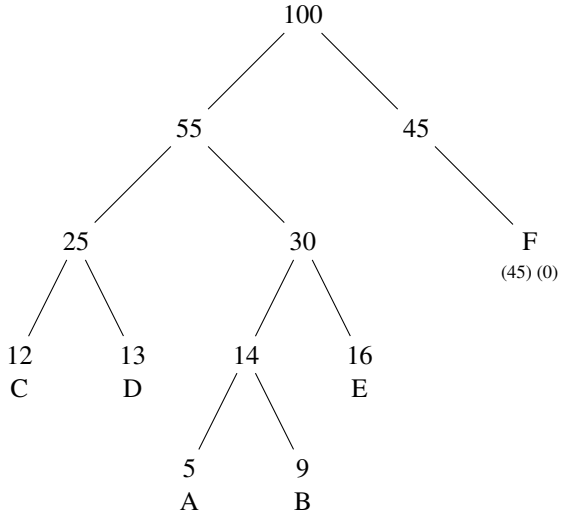
3.1.3. Example

Consider the character frequencies presented in Table 1.

Table 1. *Character Frequency.*

Character	Frequency
A	5
B	9
C	12
D	13
E	16
F	45

The Huffman tree is constructed as follows:



For each non-leaf node, assign 0 to the left edge and 1 to the right edge, this yields the following Huffman codes:

Table 2. *Character Codes.*

Character	Code
A	0100
B	0101
C	000
D	001
E	011
F	11

Now discussing the decoding, assume the encoded string is "11000101111", the decoding process would go as follows:
- 11: F
- 000: C
- 10: D
- 11: F

3.1.4. Advantages

1) *Optimality:* Huffman coding is optimal when the symbol probability distribution is known, achieving the minimum possible expected code length per symbol compared to any other prefix codes.
2) *Simplicity:* It is relatively simple to implement and understand, making it widely used in applications where data needs to be compressed and decompressed quickly.
3) *Flexibility:* It can be adapted to work with any set of symbols and probabilities, making it useful for different types of data.

3.1.5. Disadvantages

1) *Greedy Approach:* Huffman's method is a greedy algorithm, and it requires a complete knowledge of the source probability distribution before it can begin encoding, which can be a limitation in adaptive systems.
2) *Variable Code Length:* The code lengths vary according to frequency; less frequent characters can have longer codes, which might increase the size when encoding data with uniformly distributed characters.

3.1.6. Variations

1) *Adaptive Huffman Coding:* Unlike the standard Huffman coding which requires knowing all probabilities in advance, adaptive Huffman coding does not require knowledge of source probabilities beforehand and can adapt to changing data in real-time.
2) *Length-limited Huffman Coding:* This variation places a limit on the maximum length of the code words, which can be useful in communication systems where long codes are undesirable.

3.2. Arithmetic Coding

Arithmetic coding is a powerful entropy encoding method for lossless data compression. Unlike symbol-by-symbol encoding techniques like Huffman coding, arithmetic coding treats the entire input message as a single unit and assigns it a unique code within a continuous range.

3.2.1. Fundamental Concepts

1) *Probability Model:* A fundamental requirement for arithmetic coding is a probability model that estimates the likelihood of each symbol occurring in the input data. This model can be static (fixed probabilities) or adaptive (probabilities updated as the data is processed).
2) *Interval Representation:* The algorithm represents the entire probability space as a unit interval *[0, 1)*. Each symbol in the alphabet is assigned a subinterval within this range, proportional to its probability.
3) *Recursive Subdivision:* The encoding process involves recursively subdividing the current interval based on the probabilities of the symbols encountered in the input sequence. Each subdivision narrows down the interval representing the message.
4) *Tag Generation:* The final encoded output is a tag, a fractional value that uniquely identifies the final subinterval containing the entire message. This tag is typically chosen to be the shortest binary fraction that lies entirely within the subinterval.

3.2.2. Encoding Process:

1) *Initialization:* Begin with the entire probability space *[0, 1)* as the current interval.
2) *Symbol Processing:*
   a) Determine the symbol's probability range based on the probability model.
   b) Subdivide the current interval into subintervals, one for each symbol in the alphabet. The size of each subinterval is proportional to the symbol's probability.
   c) Update the current interval to the subinterval corresponding to the current symbol.
3) *Tag Generation:* Once all symbols have been processed, select a tag value that lies within the final subinterval. This tag represents the entire input sequence.

3.2.3. Decoding Process

1) *Initialization:* Begin with the entire probability space *[0, 1)* as the current interval and the received tag value.
2) *Symbol Decoding:*
   a) Identify the symbol whose probability range contains the tag value.
   b) Output this symbol as the next decoded symbol.
   c) Subdivide the current interval based on the decoded symbol's probability, similar to the encoding process.
   d) Update the current interval to the subinterval corresponding to the decoded symbol.
3) *Termination:* Repeat step 2 until the entire input sequence has been decoded.

3.2.4. Example

Consider the input sequence *CAEE* and the following probability table:

Table 3. *Symbol Probabilities and Ranges.*

Symbol	Probability	Cumulative Probability	Range
A	0.2	0.2	*[0, 0.2)*
C	0.3	0.5	*[0.2, 0.5)*
E	0.5	1.0	*[0.5, 1.0)*

1) *Initialization:* Start with the range *[0, 1)* as the current interval.
2) *Encoding 'C':*
    1) Range for 'C': *[0.2, 0.5)*
    2) Subdivide *[0, 1)* proportionally: - *[0, 0.2)*, *[0.2, 0.5)*, *[0.5, 1.0)*
    3) Update current range to *[0.2, 0.5)*.
3) *Encoding 'A':*
    1) Range for 'A': *[0, 0.2)*
    2) Subdivide *[0.2, 0.5)* proportionally: - *[0.2, 0.24)*, *[0.24, 0.3)*, *[0.3, 0.5)*
    3) Update current range to *[0.2, 0.24)*.
4) *Encoding 'E' (twice):*
    1) Range for 'E': *[0.5, 1.0)*
    2) First 'E': Subdivide *[0.2, 0.24)*, update to *[0.22, 0.24)*.
    3) Second 'E': Subdivide *[0.22, 0.24)*, update to *[0.23, 0.24)*.
5) *Tag Selection:* Choose any value within *[0.23, 0.24)*, e.g., 0.235.

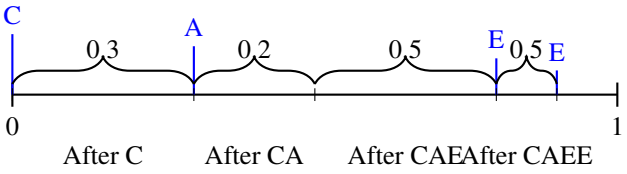


Figure 7. *Arithmetic Coding Process for the sequence CAEE.*

After processing all input symbols and continuously narrowing the interval, the final step in arithmetic coding is to select a tag that uniquely represents the entire sequence. From the last interval [0.23, 0.24) derived in the encoding steps, any number within this range can serve as the tag. For simplicity, we might choose:

Encoded Tag: 0.235

This tag value of 0.235 efficiently encapsulates the entire sequence 'CAEE' based on the defined probabilities and the resulting intervals. This tag is what would be transmitted or stored as the representation of the original data, achieving compression by using fewer bits than traditional encoding methods.

Using the tag from the encoding example, the decoding process would proceed as follows: - Start with [0, 1). The tag 0.235 first falls into the range for 'C' [0.2, 0.5). - Output 'C'. Adjust the range and continue. - Continue decoding the next symbols using the subdivisions applied during encoding, reversing the process until the entire message is reconstructed.

This method ensures that the entire message can be uniquely and efficiently decoded using the same probability model as in encoding.

Arithmetic coding excels with skewed symbol probability distributions, making it suitable for compressing text, images, and other data types with varying frequencies.

3.2.5. Advantages

1) *Optimal Compression:* Achieves compression ratios close to the theoretical entropy limit.
2) *Adaptability:* Can adapt to varying symbol probabilities, making it suitable for a wide range of data types.
3) *Efficiency:* Ideal for data with skewed symbol distributions.

3.2.6. Disadvantages

1) *Computational Complexity:* Requires high-precision arithmetic operations, making it more computationally expensive.
2) *Implementation Difficulty:* More complex to implement compared to simpler methods like Huffman coding.

3.2.7. Variations

1) *Static Arithmetic Coding:* Uses fixed probabilities for the entire encoding process.
2) *Adaptive Arithmetic Coding:* Updates symbol probabilities dynamically as data is processed, allowing for better adaptation to varying data characteristics.
3) *Binary Arithmetic Coding:* A simplified version that encodes binary data directly.

3.3. Run Length Coding

Run-Length Encoding (RLE) is a straightforward lossless data compression technique that replaces consecutive repeating characters (runs) with a single character followed by the count of repetitions. It's particularly effective for data with many long runs of the same character, such as simple images or line drawings.

3.3.1. Algorithm

The RLE algorithm operates as follows:

1) *Initialization:* Start with an empty output string or data structure.
2) *Traversal:* Iterate through the input data.
3) *Run Detection:* For each character, count the number of consecutive occurrences.
4) *Encoding:* If the count is greater than 1 (a run), append the character and its count to the output. Otherwise, append the character itself.

5) *Termination:* Continue until the end of the input data is reached.

3.3.2. Decoding Process

The RLE decoding process is equally straightforward:

1) *Initialization:* Start with an empty output string or data structure.
2) *Traversal:* Iterate through the encoded data.
3) *Decoding:* For each character-count pair:
    a. Repeat the character the specified number of times and append it to the output.
4) *Termination:* Continue until the end of the encoded data is reached.

3.3.3. Example 1

Consider the input string "AAABBBCCCDDEEEEE".

The RLE algorithm would process this as follows:

1) Start with an empty output.
2) Encounter 'A' and count three consecutive occurrences. Append 'A3' to the output.
3) Encounter 'B' and count three consecutive occurrences. Append 'B3' to the output.
4) Encounter 'C' and count three consecutive occurrences. Append 'C3' to the output.
5) Encounter 'D' and count two consecutive occurrences. Append 'D2' to the output.
6) Encounter 'E' and count five consecutive occurrences. Append 'E5' to the output.

The final RLE encoded output is "A3B3C3D2E5".

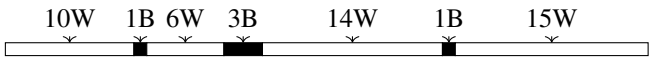
3.3.4. Example 2

Consider a simplified representation of a monochrome image where:

'W' represents a white pixel. 'B' represents a black pixel.

Input Pixel Data: 'WWWWWWWWWWWBWWWWWWBB
BWWWWWWWWWWWWWWWWBWWWWWWWWWW'

Graphical Representation and RLE Encoding:



The RLE encoded output is *W10B1W6B3W14B1W15*.

For the encoded output *W10B1W6B3W14B1W15*, the decoded output would be the original pixel data:

WWWWWWWWWWBWWWWWWBBBWWWWWW
WWWWWWWWWBWWWWWWWWW

3.3.5. Advantages

1) *Simplicity:* RLE is easy to understand and implement.
2) *Fast Encoding/Decoding:* The encoding and decoding processes are relatively fast.
3) *Effective for Repetitive Data:* RLE provides good compression for data with long runs of identical characters.

3.3.6. Disadvantages

1) *Inefficient for Non-Repetitive Data:* RLE can actually increase the size of data that doesn't contain many repetitions.
2) *Limited to Simple Runs:* RLE's basic form only handles runs of single characters.

3.3.7. Variations

There are variations of RLE, such as:

1) *Run-length Limited (RLL):* Limits the maximum length of a run to reduce the impact of long runs on decoding efficiency.
2) *Bitmap Compression:* Uses RLE to compress monochrome images efficiently.

3.4. Convolutional Algorithm - Trellis Diagram

Convolutional coding is an error-correcting technique that uses overlapping, sliding window methods on binary data to produce coded sequences. These codes are especially useful in communication systems where data is subject to errors during transmission.

3.4.1. Convolutional Coding Basics

Convolutional codes encode input data using shift registers and modulo-2 adders. Key parameters include:

1) *Code rate* (R): Ratio of input bits to output bits.
2) *Constraint length* (K): Number of memory elements in the encoder, influencing the encoder's complexity.

3.4.2. Trellis Diagram

The trellis diagram is a graphical representation of the state transitions in a convolutional encoder. Each node represents a state, and edges between nodes show state transitions with labels indicating input/output bits.

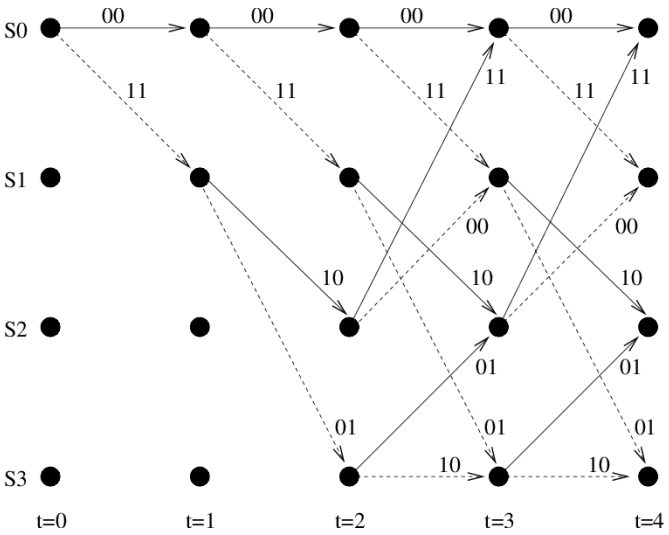


Figure 8. *Trellis diagram for rate 1/2, K=3 convolutional code [3].*

3.4.3. Decoding Process

Decoding is typically done using the Viterbi algorithm, which efficiently finds the most likely path through the trellis. The steps include:

1) *Initialization:* Set path metrics, starting with zero for the

initial state.
2) *Iteration:* Update paths and metrics based on received data.
3) *Termination:* Select the path with the lowest metric as the most likely sequence of transmitted data.

3.4.4. Encoding Example with Trellis Diagram

Consider a rate $1/2$ encoder with a constraint length of 3. The encoder has two outputs for each input bit, determined by the current and previous inputs. The generator polynomials are $g_1 = [1, 1, 1]$ and $g_2 = [1, 0, 1]$.

3.4.5. Encoding Process for Input Sequence "110"

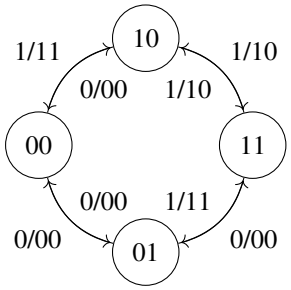
The following steps outline the encoding process:

i Initial State (00):
   1) Input: 1
   2) State Transition: 00 $\rightarrow$ 10
   3) Output: 11 (since g_1 and g_2 both produce 1)

ii State (10):
   1) Input: 1
   2) State Transition: 10 $\rightarrow$ 11
   3) Output: 10 (since g_1 produces 1 and g_2 produces 0)

iii State (11):
   1) Input: 0
   2) State Transition: 11 $\rightarrow$ 01
   3) Output: 00 (since g_1 and g_2 both produce 0)

3.4.6. Visualizing the Trellis



This simple diagram represents the state transitions for inputs '110' in a basic trellis for a convolutional encoder.

3.4.7. Decoding Example with Viterbi Algorithm

Assume the received sequence is "111000". The Viterbi algorithm will proceed as follows:

1) *Initialization:* Set the path metrics to zero for the initial state (00) and infinity for all other states.
2) *Iteration:* For each received pair of bits, update the path metrics:
   a) Step 1 (Received: 11):
      i. State (00) to (10): Metric = 0 + 0 (no error)
      ii. State (10) to (11): Metric = 0 + 1 (1 bit error)
   b) Step 2 (Received: 10):
      i. State (10) to (11): Metric = 0 + 1 = 1 (no error)
      ii. State (11) to (01): Metric = 0 + 0 = 0 (1 bit error)
   c) Step 3 (Received: 00):
      i. State (11) to (01): Metric = 0 + 0 = 0 (no error)
      ii. State (01) to (00): Metric = 0 + 1 = 1 (1 bit error)
3) *Termination:* Select the path with the lowest metric at the final step, which gives the most likely transmitted sequence.

For the given example, the most likely path corresponds to the transmitted sequence "110".

3.4.8. Advantages

1) *Error Correction:* Provides strong error correction capabilities, improving data integrity.
2) *Flexibility:* Suitable for a variety of communication systems.
3) *Efficiency:* Effective in channels with moderate noise levels.

3.4.9. Disadvantages

1) *Complexity:* Encoding and decoding can be computationally intensive.
2) *Latency:* Introduces delay due to the processing time required.
3) *Resource Usage:* Requires more memory and processing power compared to simpler codes.

3.4.10. Variations

1) *Turbo Codes:* Combines two or more convolutional codes for improved performance.
2) *LDPC Codes:* Uses a sparse matrix approach to provide strong error correction with lower complexity.
3) *Trellis-Coded Modulation (TCM):* Integrates modulation and convolutional coding for increased spectral efficiency.

3.5. BCH Algorithm

Bose-Chaudhuri-Hocquenghem (BCH) codes are a class of cyclic error-correcting codes that are constructed using polynomials over a finite field. These codes are capable of correcting multiple random error patterns and are widely used in digital communication systems.

3.5.1. Fundamental Concepts

1) *Finite Fields:* BCH codes are constructed over finite fields, also known as Galois fields (GF). The size of the field is typically 2^m, where m is a positive integer. n is the codeword length and k is the the message length.
2) *Generator Polynomial:* The generator polynomial $g(x)$ is a key element in constructing BCH codes. It is designed to have roots that correspond to error locations.
3) *Code Word:* A code word in BCH coding is a polynomial that is a multiple of the generator polynomial. The code words are used to represent the encoded data.
4) *Error Detection and Correction:* BCH codes can detect and correct multiple random errors using syndromes derived from received code words.

Out of the previously stated variables, we have different

combinations of these variable as shown in Table 4, where each combination have different performance than the other.

Table 4. *Different Possible Combinations for Codeword and Message Lengths.*

$\mathbf{m}$	$\mathbf{n = 2^m - 1}$	Maximum t	Typical k values
3	7	1	4, 3
4	15	3	11, 7, 5
5	31	5	26, 21, 16, 11
6	63	11	57, 51, 45, 39, 33, 27
7	127	23	Various

Usually, whenever the codeword length n increases, we expect a better performance for the BCH algorithm.

3.5.2. Encoding Process

1) *Initialization:* Define the generator polynomial $g(x)$ over GF(2^m).
2) *Data Polynomial:* Represent the input data as a polynomial $d(x)$.
3) *Multiplication:* Multiply $d(x)$ by x^{n-k}, where n is the code length and k is the message length.
4) *Division:* Divide the result by $g(x)$ to get the remainder $r(x)$.
5) *Code Word:* The encoded code word is $c(x) = d(x) \cdot x^{n-k} + r(x)$.

3.5.3. Decoding Process

1) *Syndrome Calculation:* Calculate the syndromes S_i for the received polynomial $r(x)$.
2) *Error Location Polynomial:* Use the Berlekamp-Massey algorithm to find the error location polynomial $\sigma(x)$.
3) *Error Locations:* Find the roots of $\sigma(x)$ using the Chien search to determine error locations.
4) *Error Correction:* Correct the errors in the received polynomial based on the error locations.

3.5.4. Encoding Example

Consider a BCH code with parameters $n = 15$, $k = 5$, and $t = 3$ (capable of correcting up to 3 errors).

1) *Generator Polynomial:* $g(x) = x^{10}+x^8+x^5+x^4+x^2+1$
2) *Data Polynomial:* $d(x) = x^4 + x^2 + 1$ (binary representation of the data 10101)
3) *Multiplication:* $d(x) \cdot x^{10} = x^{14} + x^{12} + x^{10}$
4) *Division:* Divide $x^{14} + x^{12} + x^{10}$ by $g(x)$ to get the remainder $r(x) = x^9 + x^8 + x^6 + x^5 + x^2$
5) *Code Word:* $c(x) = x^{14} + x^{12} + x^{10} + x^9 + x^8 + x^6 + x^5 + x^2$ (binary representation 101011110110010)

3.5.5. Decoding Example

Assume the received polynomial is $r(x) = x^{14} + x^{12} + x^9 + x^8 + x^6 + x^5 + x^2 + x$ (binary representation 101001110110010).

1) *Syndrome Calculation:* Calculate syndromes $S_1, S_2, S_3, S_4, S_5, S_6$ using the received polynomial.
2) *Error Location Polynomial:* Use the Berlekamp-Massey algorithm to determine the error location polynomial $\sigma(x) = x^3 + x + 1$.
3) *Error Locations:* Find the roots of $\sigma(x)$ using the Chien search to determine error locations at positions 2, 4, and 14.
4) *Error Correction:* Correct the errors in the received polynomial:
   a) Error at position 2: Flip bit from 0 to 1.
   b) Error at position 4: Flip bit from 0 to 1.
   c) Error at position 14: Flip bit from 1 to 0.
5) *Corrected Code Word:* $c(x) = x^{14}+x^{12}+x^{10}+x^9+x^8+x^6+x^5+x^2$ (binary representation 101011110110010)

3.5.6. Advantages

1) *Error Correction:* Capable of correcting multiple random errors.
2) *Flexibility:* Parameters can be adjusted to correct more errors at the expense of code rate.
3) *Performance:* Provides good error correction performance in noisy channels.

3.5.7. Disadvantages

1) *Complexity:* Encoding and decoding processes can be computationally intensive.
2) *Resource Usage:* Requires more memory and processing power compared to simpler codes.
3) *Latency:* Introduces delay due to the processing time required.

3.5.8. Variations

1) *Extended BCH Codes:* Increases the length of the code to provide better error correction.
2) *Reed-Solomon Codes:* A special case of BCH codes over GF(2^m) with applications in data storage and transmission.
3) *Primitive BCH Codes:* Uses a primitive polynomial to construct the generator polynomial.

4. Simulation Description

To proceed with the desired task, the input data was selected to be an image that will be converted to binary format suitable for the coding process.

Listing 1. *Reading and Converting Image file for Compression.*

```
img = imread('assign1.jpg');
if size(img, 3) == 3
img = rgb2gray(img);
end
binStream = reshape((dec2bin(img, 8) - '0').', 1,
[]);
img_vector = de2bi(img(:), 8, 'left-msb');
```

We started by comparing the Huffman and Arithmetic coding techniques. The code was the following:

Listing 2. *Huffman and Arithmetic Coding Comparison code.*

```
    % Huffman Encoding
    symbols = unique(binStream);
    prob=histcounts(double(binStream),'Normalization',
'probability');
    dict = huffmandict(symbols, prob);
    encodedHuffman = huffmanenco(binStream, dict);

    img_binary = img_vector(:);  % Create a binary
stream

    % Prepare data for arithmetic encoding
    %(increment to make positive integers)
    img_binary_for_encoding = img_binary + 1;

    % Arithmetic Encoding
    counts = [1, 1];  % Equal probability for '1' and
'2'
    encodedArithmetic = arithenco(double(
img_binary_for_encoding), counts);

    % BPSK Modulation setup
    bpskModulator = comm.BPSKModulator();
    bpskDemodulator = comm.BPSKDemodulator();

    SNR_dB = 0:2:24;
    berHuffResults = zeros(size(SNR_dB));
    berArithResults = zeros(size(SNR_dB));

    for idx = 1:length(SNR_dB)
    modulatedHuffman = step(bpskModulator, double(
encodedHuffman(:)));
    modulatedArithmetic = step(bpskModulator, double(
encodedArithmetic(:)));

    receivedHuffman = awgn(modulatedHuffman, SNR_dB(
idx), 'measured');
    receivedArithmetic = awgn(modulatedArithmetic,
SNR_dB(idx), 'measured');

    demodHuffman = step(bpskDemodulator,
receivedHuffman);
    demodArithmetic = step(bpskDemodulator,
receivedArithmetic);

    decodedHuffman = huffmandeco(demodHuffman, dict);
    decodedArithmetic = arithdeco(demodArithmetic,
counts, numel(img_binary_for_encoding));
    decodedArith = decodedArithmetic - 1;
    if length(decodedHuffman) > length(binStream)
    decodedHuffman = decodedHuffman(1:length(binStream
));
    end
    if length(decodedArith) > length(img_binary)
    decodedArith = decodedArith(1:length(img_binary));
    end
    % Calculate BER
    [~, berHuffResults(idx)] = biterr(binStream,
decodedHuffman');
    [~, berArithResults(idx)] = biterr(img_binary,
decodedArith);

    berHuffResults(idx) = berHuffResults(idx);
    berArithResults(idx) = berArithResults(idx);
    end
```

Next, we tried sending the image directly to the channel coding techniques that we discussed before. The code is the following

Listing 3. *Convolutional (Trellis) and BCH Coding.*

```
    % Convolutional coding parameters
    trellis = poly2trellis(7, [171 133]); % Example
trellis structure with rate 1/2

    % Convolutional Encoder
    convEncoder = comm.ConvolutionalEncoder(trellis);

    % Encode the image data
    encoded_bits_conv = step(convEncoder, bin_img);

    % BPSK Modulation
    bpskModulator = comm.BPSKModulator;
    modulated_signal_conv = step(bpskModulator,
encoded_bits_conv);

    % SNR range
    snr_range = 0:1:10;
    ber_values_conv = zeros(length(snr_range), 1);
    ber_values_bch = zeros(length(snr_range), 1);

    for snr_idx = 1:length(snr_range)
    snr = snr_range(snr_idx); % Current SNR value

    % Transmit through AWGN channel
    awgnChannel = comm.AWGNChannel('EbNo', snr);
    received_signal_conv = step(awgnChannel,
modulated_signal_conv);

    % BPSK Demodulation
    bpskDemodulator = comm.BPSKDemodulator;
    demodulated_bits_conv = step(bpskDemodulator,
received_signal_conv);

    % Convolutional Decoder
    viterbiDecoder = comm.ViterbiDecoder(trellis, ...
    'InputFormat', 'Hard', ...
    'TracebackDepth', 34, ...
    'TerminationMethod', 'Truncated'); % Ensure proper
 termination method

    decoded_bits_conv = step(viterbiDecoder,
demodulated_bits_conv);

    % Remove the initial&final bits added due to
convolutional encoding
    decoded_bits_conv = decoded_bits_conv(1:length(
bin_img));

    [num_of_error,ber_conv]=biterr(bin_img,
decoded_bits_conv);
    ber_values_conv(snr_idx) = ber_conv;

    end

    % Parameters for BCH coding
    n = 15; % Codeword length
    k = 7;  % Message length

    % BCH Encoder
    msg_length = length(bin_img);
    num_blocks = ceil(msg_length / k);
    padded_length = num_blocks * k;
    padded_bin_img = [bin_img; zeros(padded_length -
msg_length, 1)]; % Zero padding
    msg_blocks = reshape(padded_bin_img, k, num_blocks
)';

    bchEncoder = comm.BCHEncoder(n, k);
    bchDecoder = comm.BCHDecoder(n, k);

    encoded_bits_bch = [];
    for i = 1:num_blocks
    encoded_block = step(bchEncoder, msg_blocks(i, :)
');
    encoded_bits_bch = [encoded_bits_bch;
encoded_block];
    end

    modulated_signal_bch = step(bpskModulator,
encoded_bits_bch);

    for snr_idx = 1:length(snr_range)
    snr = snr_range(snr_idx);

    awgnChannel = comm.AWGNChannel('EbNo', snr);
    received_signal_bch = step(awgnChannel,
modulated_signal_bch);

    demodulated_bits_bch = step(bpskDemodulator,
received_signal_bch);

    % BCH Decoder
    decoded_bits_bch = [];
```

```
    for i = 1:num_blocks
    decoded_block = step(bchDecoder,
demodulated_bits_bch((i-1)*n + 1:i*n));
    decoded_bits_bch = [decoded_bits_bch;
decoded_block];
    end

    % Remove padding
    decoded_bits_bch = decoded_bits_bch(1:msg_length);

    [num_of_error,ber_bch]=biterr(bin_img,
decoded_bits_bch);
    ber_values_bch(snr_idx) = ber_bch;

    end
```

To complete our study, and for some reasons that we will be discussing very soon in the next sections, the next additional codes were written.

Listing 4. *Performance Evaluation code for the Source Coding techniques in terms of Compression Ratio.*

```
    % Sample Text
    txt = '
AAAAAAAAAAAAAAAAAANNNNNNNNNNNIIIIIIIIIIIISSSSSSSSSSSS'
;
    % Original Text Size
    originalTextSize = numel(txt) * 8;
    % Bits

    % 1. Huffman Encoding for Text
    textVector = uint8(txt);
    symbols = unique(textVector);
    prob = histcounts(textVector, [symbols-0.5,
symbols(end)+0.5]) / numel(textVector);
    dict = huffmandict(symbols, prob);
    huff_txt = huffmanenco(textVector, dict);

    % Huffman Encoded Text Size
    huffmanTextSize = numel(huff_txt);

    % 2. Run-Length Encoding for Text
    [rle_txt_str, rle_txt_run] = rle_encode_text(txt);

    % Run-Length Encoded Text Size
    rleTextSize = (numel(rle_txt_str) + numel(
rle_txt_run)) * 8;

    % 3. Arithmetic Encoding for Text
    [~, sortedSymbols] = sort(-prob); % Sort symbols
in descending order of probability
    textSeq = zeros(1, numel(textVector));
    for i = 1:numel(textVector)
    textSeq(i) = find(symbols == textVector(i));
    end
    counts = floor(prob * 1000); % Scale probabilities
 to avoid floating-point issues
    arith_txt = arithenco(textSeq, counts);

    % Arithmetic Encoded Text Size
    arithTextSize = numel(arith_txt);

    % Compression Ratios for Text
    huffmanRatioText = originalTextSize /
huffmanTextSize;
    rleRatioText = originalTextSize / rleTextSize;
    arithRatioText = originalTextSize / arithTextSize;



    % Sample Image
    img = imread('assign1.jpg'); % Replace 'assign1.
jpg' with your actual image path

    % Original Image Size
    originalImageSize = numel(img) * 8; % Assuming 8
bits per pixel

    % 1. Huffman Encoding for Images
    imgVector = img(:);  % Convert to a vector
    imgSymbols = unique(imgVector);
    imgProb = histcounts(imgVector, numel(imgSymbols))
 / numel(imgVector);
    imgDict = huffmandict(imgSymbols, imgProb);
    huff_img = huffmanenco(imgVector, imgDict);

    % Huffman Encoded Image Size
    huffmanImageSize = numel(huff_img);

    % 2. Run-Length Encoding for Images
    [rle_img_val, rle_img_run] = rle_encode_text(img);

    % Run-Length Encoded Image Size
    rleImageSize = (numel(rle_img_val) + numel(
rle_img_run)) * 8;

    symbols = unique(imgVector);
    symbolIndices = 1:length(symbols); % Create
indices for each unique symbol

    % Create a map of actual pixel values to their
respective indices
    [~, mapIndices] = ismember(imgVector, symbols);

    % Calculate the histogram of the mapped indices
    counts = histcounts(mapIndices, 1:length(symbols)
+1);

    % Arithmetic Encoding
    encodedArith = arithenco(mapIndices, counts);

    arithImageSize=numel(encodedArith);

    % Compression Ratios for Images
    huffmanRatioImage = originalImageSize /
huffmanImageSize;
    rleRatioImage = originalImageSize / rleImageSize;
    arithRatioImage = originalImageSize /
arithImageSize;

    % Function for Run-Length Encoding
    function [values, runs] = rle_encode_text(text)
    values = text(1);
    runs = 1;
    for i = 2:numel(text)
    if text(i) == values(end)
    runs(end) = runs(end) + 1;
    else
    values = [values, text(i)];
    runs = [runs, 1];
    end
    end
    end
```

5. Theoretical Results

5.1. Huffman vs. Arithmetic Source Coding

Both Huffman and Arithmetic coding are widely used entropy encoding techniques for lossless data compression. While each method has its specific advantages and disadvantages, their theoretical performance, particularly in terms of compression efficiency, varies based on data characteristics and application requirements.

5.1.1. Compression Efficiency

Compression efficiency is a critical factor in evaluating the performance of an entropy encoding technique. It refers to the effectiveness of the algorithm in reducing the size of the input data.

5.1.2. Huffman Coding Efficiency

Huffman coding assigns variable-length codes to input characters based on their frequencies. The code length

is inversely proportional to the frequency of the character, ensuring that more frequent characters have shorter codes.

1) *Optimality*: Huffman coding is optimal for symbol-by-symbol coding when the probabilities of the symbols are powers of two. In such cases, it achieves the minimum possible average code length, which is equal to the entropy of the source.
2) *Suboptimal Cases*: In cases where the symbol probabilities are not powers of two, Huffman coding can be suboptimal. The average code length can be higher than the entropy of the source, leading to less efficient compression compared to Arithmetic coding [4].

5.1.3. Arithmetic Coding Efficiency

Arithmetic coding represents the entire input sequence as a single number in the interval [0, 1). This approach can theoretically achieve the entropy limit of the source more closely than Huffman coding.

1) *Optimality*: Arithmetic coding is optimal for any probability distribution. It can encode data with average code length approaching the entropy of the source, regardless of the symbol probabilities.
2) *Higher Compression Ratios*: Due to its ability to handle fractional probabilities and its more precise interval subdivision, Arithmetic coding often achieves higher compression ratios than Huffman coding, particularly for large alphabets or skewed distributions [5].

5.1.4. Compression Ratio Comparison

The compression ratio is a measure of the effectiveness of a compression algorithm, defined as the ratio of the size of the original data to the size of the compressed data. Higher compression ratios indicate more effective compression.

5.1.5. Huffman Coding Compression Ratio

Huffman coding's compression ratio is influenced by the distribution of symbol frequencies in the input data. For data with symbol probabilities close to powers of two, Huffman coding performs well and achieves compression ratios close to the theoretical limit.

1) *Typical Scenarios*: For text data with moderately varying symbol frequencies, Huffman coding achieves reasonable compression ratios but may fall short of the optimal entropy limit.
2) *Limitation*: In scenarios with highly skewed symbol distributions or large alphabets, Huffman coding's compression ratio can be significantly lower than the optimal ratio due to its inability to handle fractional probabilities [4].

5.1.6. Arithmetic Coding Compression Ratio

Arithmetic coding excels in achieving high compression ratios, especially in cases where symbol probabilities vary significantly or are not close to powers of two.

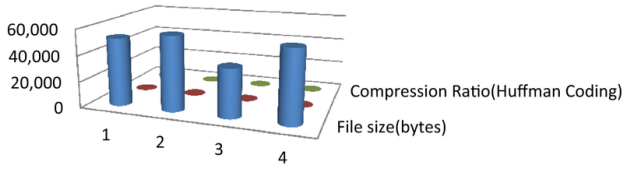
1) *Optimal Performance*: Arithmetic coding approaches the entropy limit more closely than Huffman coding, resulting in higher compression ratios for a wide range of data types.
2) *Adaptability*: Its ability to handle fractional probabilities allows it to maintain high compression efficiency across different data distributions and alphabet sizes [5].

5.1.7. Image Compression

In the context of image compression, arithmetic coding has been shown to outperform Huffman coding in terms of compression ratio. This is particularly evident in multimedia standards like JPEG and JPEG2000, where arithmetic coding is used for its superior efficiency in handling the diverse probability distributions of pixel values.

1) *JPEG and JPEG2000*: Studies have demonstrated that arithmetic coding can achieve better compression ratios compared to Huffman coding when used in the entropy coding phase of image compression standards. This improvement is due to its finer granularity in representing probability distributions [5].

A study have been done [6], it showed the performance of both Huffman and Arithmetic coding techniques with files of different sizes (the files used were text files). The obtained results were summarized in Figure 9.



	1	2	3	4
File size(bytes)	52,331	57,206	37,137	55,050
Compression Ratio(Arithmetic Coding)	1.12	1.1	1.58	1.13
Compression Ratio(Huffman Coding)	1.06	1.25	1.51	1.09

Figure 9. *Compression ratio of arithmetic coding and Huffman coding.*

5.1.8. Computational Complexity

The computational complexity of encoding and decoding processes is another important factor in the performance comparison.

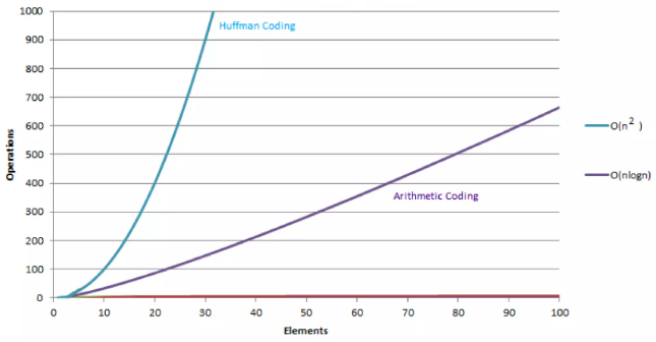


Figure 10. *Big O Time Complexity of Huffman and Arithmetic.*

5.1.9. Huffman Coding Complexity

Huffman coding is generally less computationally intensive, making it suitable for real-time applications.

1) *Encoding Complexity*: The time complexity for constructing the Huffman tree is $O(n \log n)$, where n is the number of unique symbols.
2) *Decoding Complexity*: Decoding involves traversing the Huffman tree, which has a complexity of $O(1)$ per symbol, making it very efficient.

5.1.10. Arithmetic Coding Complexity

Arithmetic coding, while more efficient in terms of compression, is computationally more intensive.

1) *Encoding Complexity*: Arithmetic coding involves high-precision arithmetic operations and multiple interval subdivisions, resulting in higher computational overhead.
2) *Decoding Complexity*: Decoding requires similar high-precision calculations to determine the intervals and retrieve the original symbols, which can be computationally demanding.

5.1.11. Practical Considerations

When choosing between Huffman and Arithmetic coding, practical considerations such as implementation complexity, processing speed, and application requirements play a significant role.

1) *Huffman Coding*: Due to its simplicity and lower computational requirements, Huffman coding is preferred for real-time applications and environments with limited computational resources.
2) *Arithmetic Coding*: Despite its computational complexity, Arithmetic coding is preferred in scenarios where achieving the highest possible compression ratio is critical, such as in multimedia compression and transmission over bandwidth-limited channels [7].

5.1.12. Conclusion

Both Huffman and Arithmetic coding have their specific use cases based on their theoretical and practical performance characteristics. Huffman coding excels in simplicity and speed, making it ideal for real-time applications. In contrast, Arithmetic coding provides better compression efficiency, especially for complex data distributions, at the cost of increased computational complexity.

5.2. Convolutional (Trellis) vs. BCH Channel Coding

Convolutional coding and Bose-Chaudhuri-Hocquenghem (BCH) coding are two prominent error-correcting techniques used in digital communication systems. Both have distinct advantages and specific use cases, particularly in handling noisy transmission channels.

5.2.1. Error Correction Efficiency

Error correction efficiency is critical for channel coding as it directly impacts the ability to recover the original data accurately after transmission through noisy channels.

5.2.2. Convolutional Coding Efficiency

Convolutional coding uses shift registers and a trellis diagram for encoding data, making it effective for correcting errors in a continuous stream of data. The efficiency of convolutional coding is enhanced by the Viterbi algorithm, which provides optimal decoding for these codes.

1) *Real-time Performance*: Convolutional codes are well-suited for real-time applications due to their continuous nature and lower latency in decoding.
2) *Robustness*: These codes are particularly robust in correcting burst errors, making them ideal for applications like mobile and satellite communications [8].

5.2.3. BCH Coding Efficiency

BCH codes are block codes designed to correct multiple random errors. They are highly efficient in terms of error correction capability and are widely used in storage devices and data transmission systems.

1) *Error Correction Capability*: BCH codes can correct multiple errors within a block, providing a high level of data integrity.
2) *Adaptability*: These codes can be tailored to correct different numbers of errors by adjusting the parameters of the generator polynomial [9].

5.2.4. Error Correction Comparison

The error correction capability of a coding scheme is a key performance metric, indicating how well the code can recover the original data in the presence of errors.

5.2.5. Convolutional Coding Error Correction

Convolutional codes use the Viterbi algorithm for decoding, which efficiently finds the most likely sequence of states and corrects errors. This method is effective in continuous data streams and low-latency environments.

1) *Burst Error Correction*: Convolutional codes are particularly effective at correcting burst errors, making them suitable for mobile and satellite communications where such errors are common.
2) *Continuous Data Streams*: These codes perform well in scenarios where data is transmitted in a continuous stream, such as real-time video and audio transmission [8].

5.2.6. BCH Coding Error Correction

BCH codes are designed to correct multiple random errors in a block of data. The error correction capability depends on the degree of the generator polynomial used in the code construction.

1) *Multiple Error Correction*: BCH codes can correct multiple random errors within each block of data, providing robust error correction for applications requiring high data integrity.
2) *High Data Integrity*: These codes are ideal for

applications where data integrity is critical, such as data storage and retrieval systems [9].

5.2.7. Computational Complexity

The computational complexity of encoding and decoding processes impacts the choice between convolutional and BCH coding.

5.2.8. Convolutional Coding Complexity

Convolutional coding, combined with the Viterbi algorithm for decoding, provides a relatively low computational complexity, making it suitable for applications requiring fast processing.

1) *Encoding Complexity*: The encoding process is straightforward and involves simple shift register operations.
2) *Decoding Complexity*: The Viterbi algorithm ensures efficient decoding with a complexity of $O(n \cdot 2^k)$, where n is the code length and k is the constraint length [8].

5.2.9. BCH Coding Complexity

BCH coding involves more complex polynomial arithmetic, both for encoding and decoding, which can be computationally intensive.

1) *Encoding Complexity*: The encoding process involves polynomial division, which increases with the degree of the generator polynomial.
2) *Decoding Complexity*: Decoding BCH codes typically requires algorithms like Berlekamp-Massey, which have higher computational demands [9].

5.2.10. Practical Considerations

When choosing between convolutional and BCH coding, practical considerations such as implementation complexity, processing speed, and application requirements are crucial.

1) *Convolutional Coding*: Ideal for real-time applications with continuous data streams, such as audio and video transmission, due to lower latency and computational requirements.
2) *BCH Coding*: Best suited for applications requiring high data integrity, such as data storage and critical data transmissions, where the ability to correct multiple errors outweighs the increased computational complexity [10].

5.2.11. Conclusion

Convolutional coding excels in real-time applications due to its lower latency and computational efficiency, which are further enhanced by the Viterbi algorithm's ability to efficiently decode convolutional codes. This capability has been extensively studied and validated, as demonstrated by research such as [11], which underscores the effectiveness of trellis-based decoding techniques in maximizing error correction performance. BCH coding, on the other hand, provides superior error correction for applications requiring high data integrity, despite its higher computational complexity. It remains the preferred choice for scenarios where the ability to correct multiple errors outweighs the increased computational demands. Additionally, a comparative performance analysis of block and convolutional codes conducted by C. Shakti, K. Asvir, M. Himanshu [12] highlights the advantages of convolutional codes in terms of their error correction capabilities, particularly in noisy communication channels where the power of convolutional coding, especially when implemented with trellis diagrams and the Viterbi algorithm, is particularly notable. .

6. Results and Interpretation

6.1. Compression Performance Analysis

Figure 11 displays the compression ratios achieved by Huffman, Run-Length, and Arithmetic encoding techniques when applied to both text and image data. The visualization compares the effectiveness of these encoding methods in reducing data size across different media types.

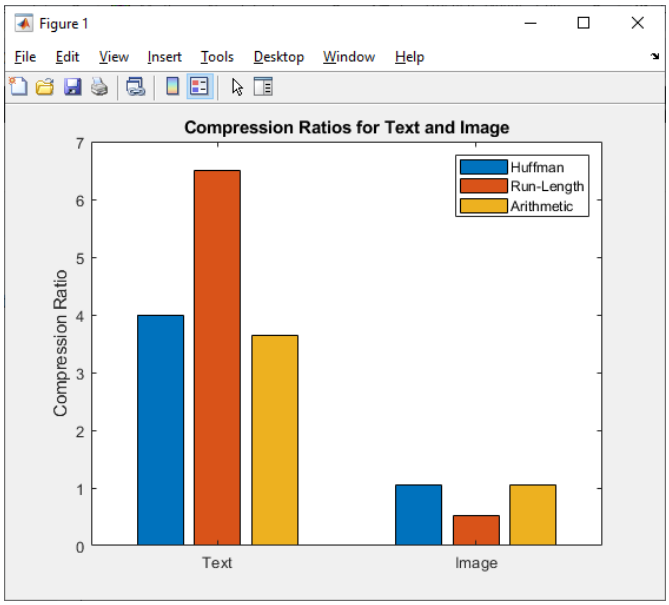


Figure 11. *Compression Ratios for Text and Image Data.*

Discussion: As shown in Figure 11, Run-Length Encoding significantly outperforms Huffman and Arithmetic encodings in text compression, which can be attributed to its efficiency in handling repeated characters. For image data, the performance varies, suggesting that the choice of encoding technique should consider the specific characteristics of the data to achieve optimal compression.

6.2. Performance Comparison Between Huffman and Arithmetic Encoding

6.2.1. Bit Error Rate Comparison

Figure 12 illustrates the Bit Error Rate (BER) vs. Signal-to-Noise Ratio (SNR) for Huffman and Arithmetic encoding methods. This graph provides insights into the error resilience of each encoding strategy under varying channel conditions.

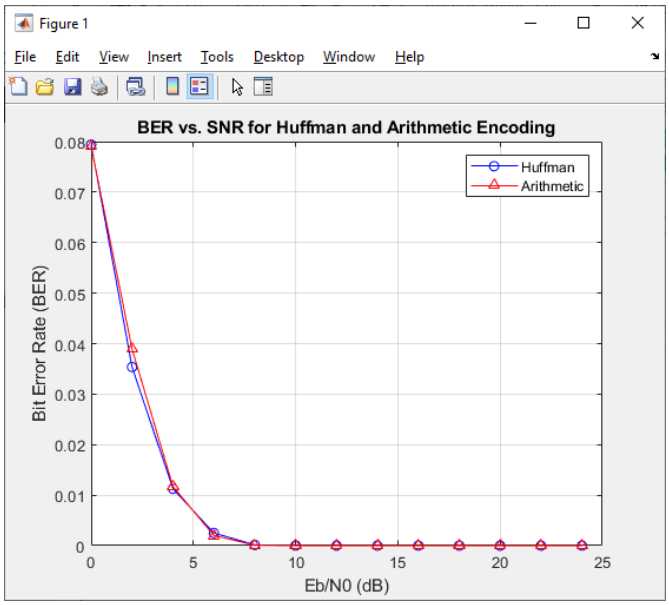


Figure 12. BER vs. SNR for Huffman and Arithmetic Encoding.

6.2.2. Decoded Image Quality at SNR = 10 dB

Figure 13 compares the quality of images decoded using Huffman and Arithmetic encoding at an SNR of 10 dB.

Figure 13. Decoded Image Quality Comparison using Huffman and Arithmetic Encoding at SNR = 10 dB.

Analysis: The images decoded with Arithmetic encoding retain better clarity and detail compared to those decoded with Huffman encoding, demonstrating Arithmetic's superior error-handling capabilities in maintaining image integrity at higher SNR levels.

6.3. Comparison Between Convolutional and BCH Coding

6.3.1. Bit Error Rate Comparison

Figure 14 shows the BER performance of Convolutional coding and BCH coding with BPSK modulation across different SNR levels.

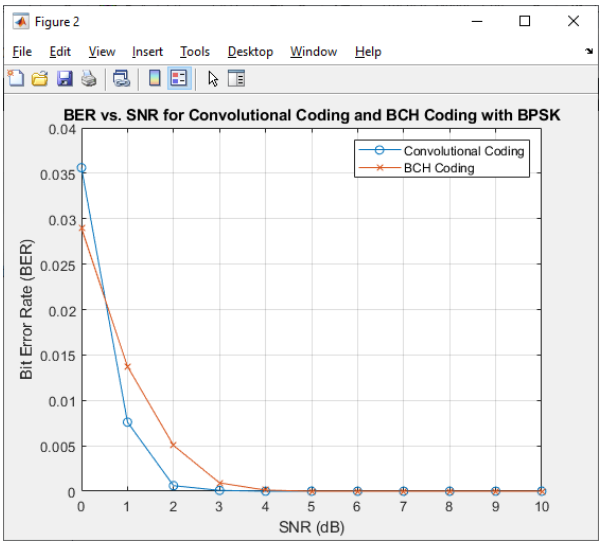


Figure 14. BER vs. SNR for Convolutional Coding and BCH Coding with BPSK.

6.3.2. Decoded Image Quality at Low SNR

Figure 15 assesses the image quality decoded at a low SNR using both Convolutional and BCH coding techniques.

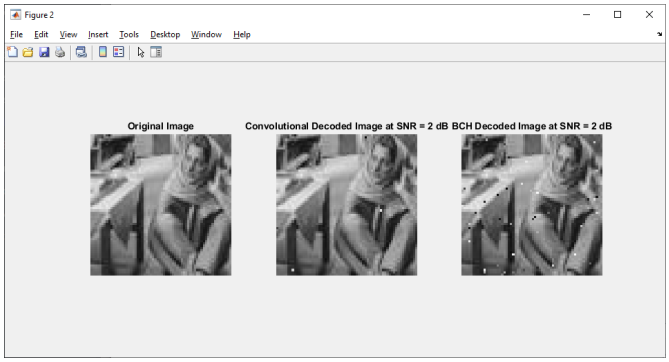


Figure 15. Comparison of Decoded Image Quality at Low SNR: Convolutional vs. BCH Coding.

Analysis This visual comparison in Figure 15 shows that Convolutional coding provides a clearer and more accurate image recovery at low SNR levels compared to BCH coding, highlighting its effectiveness in applications where maintaining image quality is critical. A notable observation from the BER performance of Convolutional coding versus BCH coding, as shown in Figure 14, is the initial higher BER at SNR = 0. This phenomenon can be attributed to the Viterbi algorithm used in Convolutional coding, which exploits the inherent memory structure within the code to provide optimal error correction. This initial high BER reflects the algorithm's adjustment phase, which is crucial for achieving superior error correction at subsequent higher SNR values. The inherent memory exploitation allows for effective error prediction and correction, leading to significant improvements in BER as the SNR increases.

This characteristic is particularly beneficial for applications where maintaining data integrity across variable noise levels is crucial, affirming the choice of Convolutional coding for

environments with fluctuating SNR conditions.

6.4. Advanced Combined Coding Technique

Given the findings from previous tests, a hybrid approach combining Arithmetic encoding for source coding with Convolutional coding for channel coding was implemented to harness the strengths of both techniques, to obtain the complete digital modulation system depicted in Figure 1. This section discusses the results of this combination under simulated conditions.

6.4.1. Bit Error Rate Improvement with Combined Coding

Figures 16 and 17 illustrate the BER versus SNR for the combined Arithmetic and Convolutional coding. These graphs provide a direct comparison with previous results using only individual coding techniques.

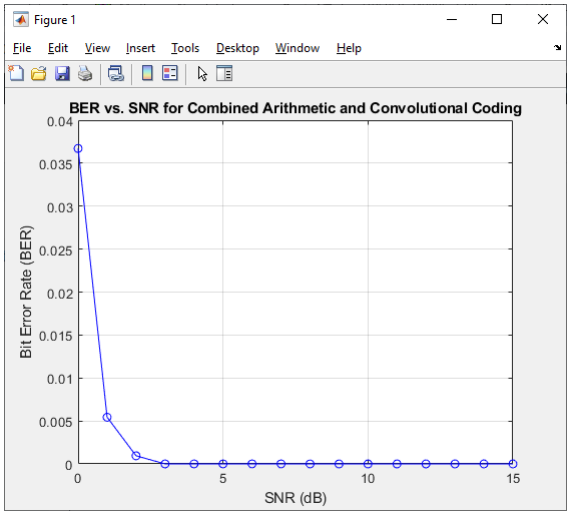


Figure 16. BER vs. SNR for Combined Arithmetic and Convolutional Coding (Normal Scale).

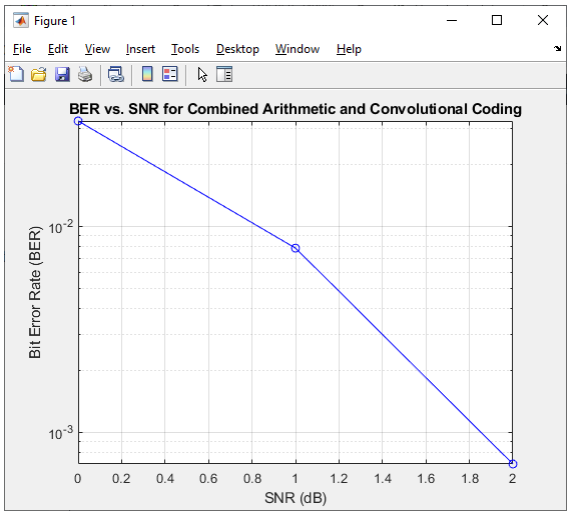


Figure 17. BER vs. SNR for Combined Arithmetic and Convolutional Coding (Logarithmic Scale).

The results demonstrate a dramatic improvement in BER performance, particularly at higher SNR levels, showcasing the effectiveness of integrating Arithmetic encoding's compression capabilities with Convolutional coding's robust error correction methods. The combined approach outperforms the individual techniques, especially in maintaining lower error rates across a wider range of SNR values.

6.4.2. Visual Quality Analysis at Moderate SNR

Figure 18 displays the original image alongside its decoded version at an SNR of 6 dB using the combined coding method.

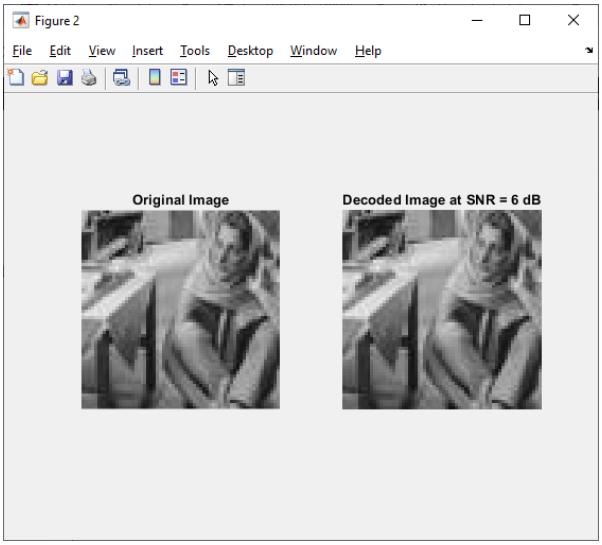


Figure 18. Comparison of Original and Decoded Image Quality at SNR = 6 dB with Combined Coding.

The decoded image exhibits significantly less degradation and maintains higher fidelity relative to what was observed with individual coding techniques at similar SNR levels. This underscores the hybrid approach's advantage in preserving image quality under noisy conditions.

6.4.3. Experimental Channel Capacity and BER Analysis

The experimental data collected on channel capacity and BER at various SNR levels provides critical insights into the performance of the combined coding system. This section explores the relationship between SNR, channel capacity, and the probability of error, aiming to identify the maximum channel capacity that maintains a zero error probability.

Table 5. Channel Capacity and BER for Different SNR Values.

SNR (dB)	Channel Capacity (bps)	BER
0	1,000,000	0.03421
1	1,175,637	0.0054321
2	1,370,105	0.00076294
3	1,582,682	0.000061035
4	1,812,246	0
5	2,057,373	0

As evident from Table 5, there is a clear trend of increasing channel capacity and decreasing BER with higher SNR levels. Particularly at SNR levels of 4 dB and above, the system achieves a BER of zero, indicating an optimal error-free state at these capacities. This performance showcases the effectiveness of the combined coding approach in maximizing channel utilization while minimizing transmission errors.

Practical Implications The ability to achieve high channel capacity without errors at reasonable SNR levels is crucial for applications such as high-definition video streaming, where maintaining both high quality and continuous data flow is essential. This data provides a benchmark for systems engineers to design networks that can operate efficiently under varied conditions, optimizing both the throughput and reliability of data transmissions.

Conclusion The combined use of Arithmetic and Convolutional coding not only improves error correction capabilities but also optimally enhances channel capacity. This balance is vital for applications where both high throughput and low error rates are critical, affirming the benefit of integrating advanced coding techniques in modern communication systems.

7. Conclusion

This report has evaluated various encoding techniques and their applicability in digital communication systems, focusing on compression efficiency, error correction, and system reliability. Our findings highlight the distinct advantages of each coding method:

- *Run-Length Encoding* excels in text compression, particularly for data with repetitive patterns.

- *Arithmetic Encoding* ensures high data integrity, making it suitable for applications requiring precise data transmission.

- *Convolutional Coding* offers unmatched error correction capabilities, essential in high-noise communication environments.

- The *combination of Arithmetic and Convolutional coding* effectively enhances both compression and error correction, leading to superior performance in terms of both bit error rates and channel capacities.

These results underscore the importance of selecting appropriate coding techniques to optimize the efficiency and reliability of communication systems. The integration of source and channel coding, particularly through hybrid approaches like Arithmetic-Convolutional coding, presents a robust solution for enhancing the performance of modern digital communications.

Future research should continue to explore these and new coding strategies, especially in response to the evolving demands of data-heavy and high-speed communication networks. This study provides crucial insights that can assist system designers and engineers in developing more robust and efficient communication infrastructures.

In conclusion, strategic coding technique selection based on specific system requirements and operational conditions is vital for achieving optimal performance in communication systems. This approach not only improves the overall system functionality but also ensures high levels of data integrity and transmission efficiency.

ORCID

0009-0001-6435-708X (Anis Zebiane)

Abbreviations

BER	Bit Error Rate
SNR	Signal-to-Noise Ratio
BPSK	Binary Phase Shift Keying
AWGN	Additive White Gaussian Noise
BCH	Bose-Chaudhuri-Hocquenghem
RLE	Run-Length Encoding
TCM	Trellis-Coded Modulation
LDPC	Low-Density Parity-Check
RS	Reed-Solomon
GF	Galois Field
DCT	Discrete Cosine Transform
DWT	Discrete Wavelet Transform

Author Contributions

Anis Zebiane: Conceptualization, Data curation, Formal Analysis, Investigation, Methodology, Software, Validation, Visualization, Writing – original draft, Writing – review & editing

Conflicts of Interest

The authors declare no conflicts of interest.

References

[1] Fu-Hua Huang, Chapter 1. Available at: https://vtechworks.lib.vt.edu/server/api/core/bitstreams/8a89ad07-8e74-4ef3-87d3-69f53dce15ac/content

[2] Huffman, D. (1952). ”A Method for the Construction of Minimum-Redundancy Codes”. Proceedings of the IRE 40 (9):1098-1101. https://doi.org/10.1109/JRPROC.1952.273898

[3] Tuominen, Johanna and Plosila, Juha. Asynchronous Viterbi Decoder in Action Systems.

[4] Revisiting Huffman Coding: Toward Extreme Performance on Modern GPU Architectures. Available at: https://arxiv.org/abs/2010.10039

[5] Evaluation of Huffman and Arithmetic Algorithms for Multimedia Compression Standards. Available at: https://arxiv.org/abs/1109.0216

[6] Jancy, S. and Jayakumar, C.. (2019). Sequence Statistical Code Based Data Compression Algorithm for Wireless Sensor Network. Wireless Personal Communications. 106. https://doi.org/10.1007/s11277-019-06171-x

[7] Comparison of Huffman and Arithmetic Coding. Available at: https://codingtube.tech

[8] Proakis, J. G., Salehi, M., & Bauch, G. (2001). Digital communications. McGraw-Hill.

[9] Lin, S., & Costello, D. J. (2004). Error control coding. Pearson Education India.

[10] Ko, C. C., & Chen, J. (2018). A new BCH coded modulation scheme based on combined block coding and RS codes. Electronics, 7(6), 89. https://doi.org/10.3390/electronics7060089

[11] Costello, D. J., Jr., Dolinar, S. J., & Kunz, T. G. (1983). Performance of trellis-coded modulation with Viterbi decoding for satellite and space communication. IEEE Journal on Selected Areas in Communications, 1(6), 954-975. https://doi.org/10.1109/JSAC.1983.1145922

[12] C. Shakti, K. Asvir, M. Himanshu,(2016). Comparative Performance Analysis of Block and Convolution Codes. Indian Journal of Science and Technology. 9. https://doi.org/10.17485/ijst/2015/v8i1/106868

[13] Zhu, Y., Liu, Y., Zhu, Y., Huang, M., Jiang, J., & Zhang, Y. (2025). Lossy Infrared Image Compression Based on Wavelet Coefficient Probability Modeling and Run-Length-Enhanced Huffman Coding. Sensors, 25(8), 2491. https://doi.org/10.3390/s25082491

[14] Lei, X. (2024). The Comparison of RS Codes, LDPC Codes, and Turbo Codes. Highlights in Science, Engineering and Technology, 81, 496–504. https://doi.org/10.54097/8tn0ya36

[15] Patel, D., et al. (2023). Performance Evaluation of Conventional and Neural Network-Based Decoder for an Audio of Low-Girth LDPC Code. Journal of Electrical and Computer Engineering, 2023, Article 1071142. https://doi.org/10.1155/2023/1071142