Department of Electronics & Electrical Communication Engineering
Indian Institute of Technology Kharagpur

Multimedia Systems And Applications (EC60104)

Assignment 1 : Lossless Image Compression

Group No. : 1
Members: Adipta Halder (21EC37023)
Mayukh Shubhra Saha (21EC37009)
Abhradeep De (21EC37022)

Original Grayscale Image

lena.png (225 * 225)

GitHub Repository

All scripts and relevant input and output files are present in a GitHub repository.
Link : https://github.com/mayukhss262/Multimedia_Assignments
Please refer to the README for the file structure and contents and commands to run the scripts.

Part 1

Aim

(a) Obtain the bit-plane mapped images for all the eight bit planes.
(b) Represent the image by a 8-bit gray code defined as follows:

$g_7 = b_7$
$g_i = b_i \oplus b_{i+1}, \qquad 0 \leq i \leq 6$

where, $b_7 b_6 \ldots\ldots b_1 b_0$ represents the binary values. Obtain the gray-coded bit plane images for all the eight planes.

(c) Compare the two sets of bit-plane images in (a) and (b). In what sense should gray-coded bit-planes be better? Justify your answer.

Brief Theory

In an 8-bit grayscale image, each pixel intensity (0–255) can be decomposed into 8 bit planes, where bit plane k ($0 \leq k \leq 7$) contains the k-th bit of every pixel's binary representation. The most significant bit (MSB) plane (bit 7) carries the most visual information and structural correlation, while lower-order planes appear increasingly noisy and stochastic.

A limitation of standard binary bit planes is that adjacent intensity values (e.g., 127 = 01111111 and 128 = 10000000) differ in all 8 bits, causing abrupt transitions across bit planes even in smooth image regions. Gray code representation addresses this by ensuring that consecutive integers differ by exactly one bit (e.g., 127 → 01000000, 128 → 11000000 in Gray code).

Converting pixel values to Gray code before bit-plane extraction significantly increases spatial correlation within each bit plane, particularly in the lower-order planes, making them more suitable for image compression.

Solution Approaches

1(a)

The bit plane decomposition of the 8-bit grayscale image is done by a Python script. It extracts each of the 8 constituent bit planes (bit 0 through bit 7) from the source image (**lena.png**) and saves them as individual binary images.

The script expects a single input image named lena.png located in the same directory as itself. The image can be of any format supported by **PIL (PNG, JPEG, BMP, etc)**. If the input is a color image, the script automatically converts it to 8-bit grayscale during loading. It generates 8 output images, one for each bit

plane, stored in a dedicated subdirectory named **bit_plane_images/** within the script's directory. Each output file follows the naming convention **bit_plane_{i}.png** where i ranges from 0 (LSB) to 7 (MSB). The output images are binary in nature—pixels are either 0 (black) or 255 (white)—representing whether the corresponding bit in the original pixel value is 0 or 1, respectively.
The main() function of the script begins by dynamically resolving the script's directory using **os.path.dirname(os.path.abspath(__file__))**. This ensures that the script always correctly locates input and output paths from any working directory. The input image path and output directory path are constructed relative to this base path. Before processing, the script verifies the existence of the input image file. If absent, execution halts with an informative error message. The output directory bit_plane_images/ is created if it does not already exist using os.makedirs().

The PIL library loads the input image and converts it to grayscale mode 'L', yielding pixel values in the range [0, 255]. This PIL Image object is then converted to a NumPy array for efficient vectorized bit manipulation. The image dimensions are logged for verification.

The core logic resides in the **get_bit_plane()** function. For a given bit position k, the function constructs a bitmask as 1 << k (e.g., 0b00000001 for k=0, 0b10000000 for k=7). A bitwise AND operation image_array & mask isolates the k-th bit of every pixel simultaneously via NumPy's vectorized operations. The result is then right-shifted by k positions (>> bit_position) to normalize the extracted bit to either 0 or 1. Finally, multiplication by 255 scales the binary values to full 8-bit intensity for proper image visualization. The resulting array is cast to np.uint8 for compatibility with image file formats. The main() function iterates over all 8 bit positions (0 through 7), invoking get_bit_plane() for each. Each extracted bit plane is converted back to a PIL Image using **Image.fromarray()** and saved to the output directory with the appropriate filename.

1(b)

Decomposing the 8-bit grayscale image into Gray-coded bit planes is handled by another Python script. It first transforms the pixel values of the source image from

standard binary to Gray code representation, and then extracts each of the 8 bit planes from this Gray-coded image. Gray code encoding ensures that adjacent integer values differ by exactly one bit, which significantly increases spatial correlation within individual bit planes—particularly in the lower-order planes—making them more suitable for compression techniques like run-length encoding and entropy coding.

The script expects a single input image named lena.png located in the same directory as the script. Similar to the previous script, the input can be any image format supported by PIL and can be either color or grayscale. Color images are automatically converted to 8-bit grayscale (mode 'L') during the loading phase. The script generates 9 output images stored in a dedicated subdirectory named **gray_coded_bit_plane_images/** within the script's directory. The outputs include **gray_code_image.png**, which is the complete Gray-coded representation of the original image. The other outputs are named **gray_bit_plane_{i}.png** (for i = 0 to 7). Eight individual bit-plane images extracted from the Gray-coded pixel values. Each file corresponds to bit position i, where 0 is the LSB and 7 is the MSB. These are binary images with pixel values of either 0 (black) or 255 (white).

The main() function begins by determining the script's directory using **os.path.dirname(os.path.abspath(__file__))**, ensuring that all file paths are resolved relative to the script's location regardless of the current working directory. The input image path and output directory are constructed from this base path.

The script checks for the existence of the input image and terminates with an error message if it is absent. The output directory is created via os.makedirs() if it does not already exist. The PIL library loads the input image and converts it to grayscale mode 'L', producing pixel values in the range [0, 255]. The image is then converted to a NumPy array for efficient vectorized operations. Image dimensions are logged during this phase.

The main logic is implemented in the **binary_to_gray()** function. The function applies the standard Gray code formula **G = B ⊕ (B >> 1)**, where B is the original binary value and ⊕ denotes the XOR operation. For each pixel, this computes the Gray code equivalent by XORing the binary value with its right-shifted (by one

position) version. This operation is performed element-wise across the entire NumPy array using vectorized bitwise operations. The resulting Gray-coded array has the property that consecutive intensity values (e.g., 127 and 128) now differ by only a single bit, unlike their binary counterparts which may differ in all 8 bits.

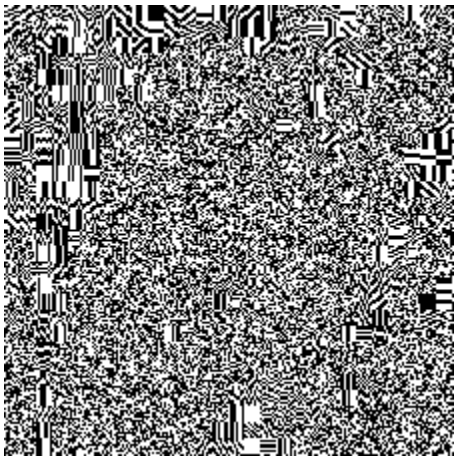
Before extracting bit planes, the script saves the complete Gray-coded image as **gray_code_image.png**. This provides visibility into the intermediate representation and allows verification that the Gray code conversion was applied correctly.

The **get_bit_plane()** function is identical to that in the standard bit-plane script. For a given bit position k, it constructs a bitmask 1 << k, applies a bitwise AND with the image array to isolate the k-th bit, right-shifts the result to normalize it to 0 or 1, and scales by 255 for proper 8-bit visualization. The key difference is that this function is now applied to the Gray-coded pixel array rather than the original binary pixel values.

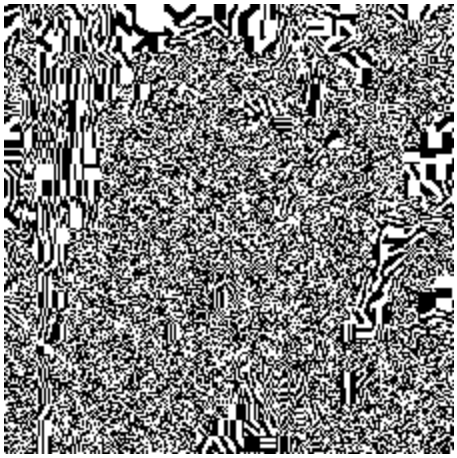
The main() function iterates over all 8 bit positions (0 through 7), extracting each bit plane from the Gray-coded image and saving it with the naming convention **gray_bit_plane_{i}.png**.

Results & Observations

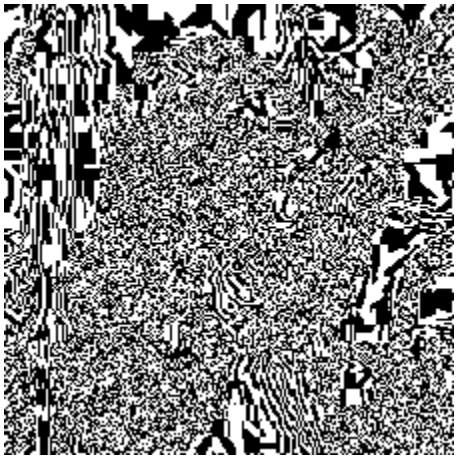
1(a)



Bit Plane 0 (LSB)



Bit Plane 1



Bit Plane 2

Bit Plane 3

Bit Plane 4

Bit Plane 5

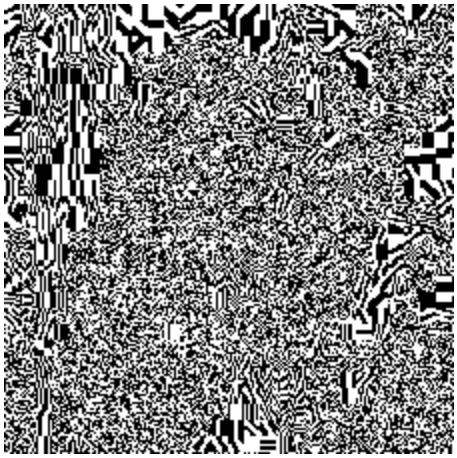
Bit Plane 6

Bit Plane 7 (MSB)

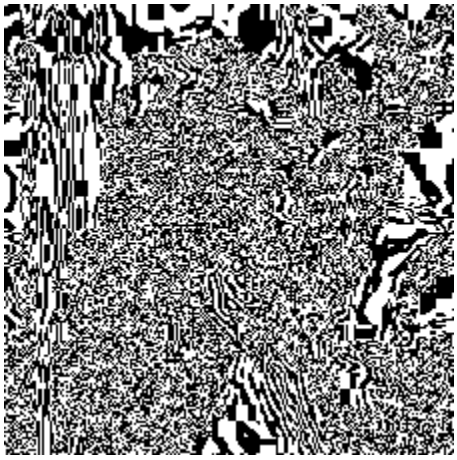
The higher bit planes resemble the original image closely, while the lower bit planes are increasingly random and noisy.

1(b)

Original Image Represented In Gray Code



Gray Coded Bit Plane 0 (LSB)



Gray Coded Bit Plane 1

Gray Coded Bit Plane 2

Gray Coded Bit Plane 3

Gray Coded Bit Plane 4

Gray Coded Bit Plane 5

Gray Coded Bit Plane 6

Gray Coded Bit Plane 7 (MSB)

The higher bit planes resemble the original image closely, while the lower bit planes are increasingly random and noisy.

1(c): Gray Code vs. Binary Representation for Bit-Plane Decomposition

Gray code representation offers significant advantages over standard binary encoding for bit-plane image decomposition due to its fundamental property that adjacent integer values differ by exactly one bit, whereas binary representation can require multiple simultaneous bit changes (e.g., $127_{10} = 01111111_2$ to $128_{10} =$

10000000_2 changes all 8 bits). This characteristic directly impacts spatial correlation in natural images where neighboring pixels have similar intensities: smooth gradients in binary code cause rapid oscillations in lower bit planes creating high-frequency noise patterns, while gray code maintains regional coherence with minimal bit transitions across gradual intensity changes. Consequently, gray-coded bit planes exhibit longer runs of consecutive 0s and 1s, making them substantially more compressible under run-length encoding schemes, with reduced entropy in individual bit planes leading to better performance with Huffman and arithmetic coding. The practical benefit is quantifiable—a horizontal gradient from 0 to 255 produces approximately 255 transitions in the binary LSB plane versus significantly fewer in the gray code equivalent, translating to higher compression ratios when raw binary data is processed. Additionally, gray code provides improved error resilience (single-bit errors affect only one intensity level versus potentially large visual artifacts in binary) and enables more meaningful progressive transmission when sending bit planes from MSB to LSB, making it the preferred representation in image compression systems utilizing bit-plane decomposition, including standards like JPEG 2000.

Part 2

Aim

(a) On the above bit-plane images, perform 1-D run-length coding. If each run is to be represented by a 6-bit value, calculate the compression ratio (compressed bits: uncompressed bits) for (i) binary-mapped bit-planes and (ii) gray-coded bit-planes.

(b) From the statistics of gray-coded bit-planes, obtain the probabilities of the run-lengths. Assign Huffman codes to the run-lengths and calculate the compression ratio for the resulting encoded bit stream. Compare this result with that of (a)-(ii).

Brief Theory

Run-Length Encoding (RLE) is a lossless data compression technique that exploits spatial redundancy by replacing consecutive sequences of identical symbols with a single (symbol, count) pair or, in the case of binary data, a sequence of run lengths.

For a binary bit-plane image, RLE traverses each row and encodes contiguous runs of 0s and 1s as their respective lengths. For example, a row 0000011111100 would be encoded as the run-length sequence [5, 6, 2] (assuming the row starts with 0 and values alternate). If a row starts with 1, an initial run-length 0 is added, signifying the first run of 0s has a length of 0. The compression efficiency of RLE is directly proportional to the average run length—images with high spatial correlation (i.e., large homogeneous regions) yield longer runs and thus better compression, while noisy or high-frequency content produces short runs that may result in data expansion. In bit-plane decomposition, **RLE is particularly effective on higher-order bit planes (MSB) which exhibit strong spatial structure**, and on Gray-coded bit planes where the single-bit-change property between adjacent intensities increases run lengths in smooth gradient regions.

Huffman coding is an entropy-based, variable-length prefix coding algorithm that achieves near-optimal compression by assigning shorter codewords to more frequent symbols and longer codewords to less frequent symbols. The algorithm constructs a binary tree bottom-up by repeatedly merging the two nodes with the lowest probabilities until a single root node remains; the path from root to each leaf (left = 0, right = 1) defines the codeword for that symbol. The resulting code is instantaneously decodable (**prefix-free**) and produces an average code length L that satisfies the inequality $\mathbf{H(X) \leq L < H(X) + 1}$, where H(X) is the source entropy in bits per symbol. When applied to RLE-encoded bit planes, Huffman coding exploits the non-uniform distribution of run lengths—short runs (particularly in noisy LSB planes) and certain characteristic lengths (in structured MSB planes) occur more frequently and receive shorter codes. Using a common codebook across all bit planes simplifies the decoder but sacrifices optimality, while individual codebooks per bit plane achieve tighter compression at the cost of increased overhead for storing multiple code tables.

Solution Approaches

2(a)

A Python script implements 1D Run-Length Encoding (RLE) on bit-plane images to achieve lossless compression. The primary objective is to encode both standard

binary bit planes (generated by 1a) and Gray-coded bit planes (generated by 1b) using a row-wise RLE scheme with 6-bit run length values, then compare their compression performance. The script quantifies the compression ratio for each bit plane.

The script expects two input directories relative to its location: **bit_plane_images/** containing the 8 binary bit-plane images, and **gray_coded_bit_plane_images/** containing the 8 Gray-coded bit-plane images. Within each directory, the script searches for PNG files whose filenames contain bit_plane (e.g., bit_plane_0.png, gray_bit_plane_7.png). Each input image is expected to be a binary image with pixel values of either 0 or 255, representing the extracted bit values.

The script generates two categories of outputs for each input folder. First, it creates compressed binary files with the **.rl** extension in the output directories **RL_bit_plane_images/** and **RL_gray_coded_bit_plane_images/,** for binary and Gray-coded bit planes respectively. **Each .rl file contains the packed 6-bit run-length values as raw binary data.**

Next, the script generates a **compression_ratios.txt** report in each output directory, documenting the uncompressed size (in bits), compressed size, number of runs, and compression ratio for each bit plane.
The core encoding logic is in the run_length_encode_row() function, which operates on individual rows of the binary image. The function first normalizes pixel values to strict binary (0 or 1) using the expression (row>0).astype(np.uint8). **The encoding scheme assumes that all rows conceptually begin with a run of 1s. If a row actually starts with 0, the encoder outputs a zero-length run as the first symbol, signaling that the implicit leading 1-run has length zero.** This alternating convention eliminates the need to explicitly store the symbol value (0 or 1) for each run—the decoder knows that runs alternate starting from 1. The function iterates through each pixel, incrementing a counter while the value remains unchanged. When a transition occurs (0→1 or 1→0), the current run length is appended to the output list and a new run begins. To accommodate the 6-bit representation constraint (maximum value 63), runs exceeding 63 pixels are split: a run of 63 is emitted, followed by a zero-length run of the opposite value,

and counting resumes. This splitting ensures all run lengths fit within 6 bits while maintaining correct decoding semantics.

The **run_length_encode_image()** function applies row-level encoding across all rows of the image, concatenating the run-length sequences into a single flat list. The row-by-row (1D) approach treats each scanline independently, which is simpler than 2D schemes but still captures horizontal spatial redundancy effectively.

The run-length values are packed into a binary file format where each 6-bit value is concatenated into a continuous bit stream and then segmented into 8-bit bytes. The function maintains a bit buffer that accumulates incoming 6-bit values via left-shifting and bitwise OR operations. Whenever the buffer contains 8 or more bits, a complete byte is extracted and appended to the output. Any remaining bits after processing all runs are zero-padded and flushed as a final byte.

The uncompressed size is calculated as height × width bits (since each bit-plane pixel is inherently 1 bit). The compressed size is computed as the number of runs multiplied by 6 bits. **The compression ratio is defined as compressed_bits / uncompressed_bits**, where values less than 1 indicate successful compression and values greater than 1 indicate expansion. Results are logged to both the console and a formatted text file.

2(b)

Two scripts (for two approaches) are written for this part. Both scripts implement Huffman coding on run-length encoded (RLE) Gray-coded bit-plane images to achieve entropy-based compression. The fundamental goal is to apply variable-length prefix coding to the RLE symbols (run lengths) extracted from Gray-coded bit planes, thereby exploiting the non-uniform distribution of run lengths to reduce the average bits per symbol. The two scripts differ in their codebook strategy:

2b_huffman_common.py - constructs a single Huffman codebook from the aggregated symbol statistics of all 8 bit planes

2b_huffman_individual.py - constructs an independent Huffman codebook for each bit plane based solely on its own statistics.

Input Specification
Both scripts expect the input directory **gray_coded_bit_plane_images/** to exist, containing the 8 Gray-coded bit-plane images. The scripts search for PNG files matching the pattern **gray_bit_plane_*.png**. Each image is loaded as grayscale and converted to a NumPy array for processing.

Both scripts generate a detailed text report documenting the compression results. 2b_huffman_common.py generates **2b_huffman_common_results.txt**, while 2b_huffman_individual.py generates **2b_huffman_individual_results.txt**. The reports include a description of the encoding scheme, the complete Huffman codebook(s) with run length, frequency, probability, and assigned codeword, per-bit-plane compression statistics including symbol count, compressed size, compression ratio, and average code length.

The Huffman tree is represented using a **HuffmanNode** class containing the symbol, frequency, and left/right child references. The **lt** method enables comparison by frequency for use with Python's heapq min-heap. The **build_huffman_tree()** function initializes the heap with leaf nodes for each unique symbol, then iteratively extracts the two lowest-frequency nodes, merges them into an internal node with combined frequency, and reinserts the merged node. This bottom-up process continues until a single root node remains.

The **generate_huffman_codes()** function performs a recursive depth-first traversal of the Huffman tree, accumulating the binary path (0 for left, 1 for right) as it descends. When a leaf node is reached, the accumulated path becomes the Huffman codeword for that symbol. The function handles the edge case of a single-symbol alphabet by assigning the code "0".

Both scripts use the alternating RLE scheme where rows are assumed to start with a run of 1s. If a row actually begins with 0, a zero-length run is prepended to signal this. The encoder iterates through each row, counting consecutive identical values and recording run lengths upon each transition. Unlike the 2a script, there is no

6-bit cap on run lengths—values are stored as arbitrary integers. The image-level function aggregates runs from all rows into a single list.

The critical distinction between the two scripts lies in how the Huffman codebook is derived.

Common Codebook (2b_huffman_common.py) - In the first phase, it iterates over all 8 bit-plane images, performs RLE on each, and aggregates all run-length symbols into a single combined pool. A frequency distribution is computed over this combined pool, and a **single Huffman tree** is constructed from these aggregated statistics. In the second phase, this common codebook is applied to encode each bit plane individually, and compression ratios are computed per plane. This approach ensures that the same codeword is assigned to the same run length regardless of which bit plane it appears in, simplifying the decoder at the cost of suboptimal code assignment for individual planes whose distributions differ from the aggregate.

Individual Codebooks (2b_huffman_individual.py) - The script processes each bit-plane image independently in a single pass. For each image, it performs RLE, computes the frequency distribution of run lengths within that image alone, constructs a Huffman tree from those statistics, and generates a codebook specific to that bit plane. This results in **8 distinct codebooks**, each optimally tuned to the probability distribution of its respective bit plane. The trade-off is increased decoder complexity and storage overhead for maintaining multiple codebooks.

Results & Observations

2(a)

```
================================================================================
        RUN-LENGTH ENCODING COMPRESSION RESULTS
================================================================================

Encoding: 6-bit run values, 1D row-by-row, rows assumed to start with 1

--------------------------------------------------------------------------------
File                          Uncomp Bits     Comp Bits       Ratio
--------------------------------------------------------------------------------
bit_plane_0.png               50625           150402          2.9709
bit_plane_1.png               50625           144468          2.8537
bit_plane_2.png               50625           131802          2.6035
bit_plane_3.png               50625           116442          2.3001
bit_plane_4.png               50625           96300           1.9022
bit_plane_5.png               50625           66924           1.3220
bit_plane_6.png               50625           47400           0.9363
bit_plane_7.png               50625           24540           0.4847
--------------------------------------------------------------------------------

Average Compression Ratio:                    1.9217

Note: Ratio < 1 means compression achieved, Ratio > 1 means expansion.
```

Compression Ratios Report For Each Bit Plane Using 6-bit RLE

```
================================================================================
        RUN-LENGTH ENCODING COMPRESSION RESULTS
================================================================================

Encoding: 6-bit run values, 1D row-by-row, rows assumed to start with 1

--------------------------------------------------------------------------------
File                            Uncomp Bits     Comp Bits       Ratio
--------------------------------------------------------------------------------
gray_bit_plane_0.png            50625           144246          2.8493
gray_bit_plane_1.png            50625           132582          2.6189
gray_bit_plane_2.png            50625           116736          2.3059
gray_bit_plane_3.png            50625           93330           1.8436
gray_bit_plane_4.png            50625           73884           1.4594
gray_bit_plane_5.png            50625           40428           0.7986
gray_bit_plane_6.png            50625           30552           0.6035
gray_bit_plane_7.png            50625           24540           0.4847
--------------------------------------------------------------------------------

Average Compression Ratio:                      1.6205

Note: Ratio < 1 means compression achieved, Ratio > 1 means expansion.
```

Compression Ratios Report For Each Gray-Coded Bit Plane Using 6-bit RLE

It is seen that the RL encoded outputs only lead to compression near the upper bit planes. All lower bit planes actually see expansion of data. Overall, the binary compression ratio is **1.92**, and the Gray-coded compression ratio is **1.62**.

This RLE scheme uses a fixed 6-bit representation for each run length, meaning every run requires exactly 6 bits regardless of its actual length or frequency of occurrence. For compression to occur, the total number of runs multiplied by 6 bits must be less than the original image size. However, in bit-plane images, particularly the lower planes, the pixel values exhibit **high spatial randomness** with frequent alternations between 0 and 1, resulting in **very short runs.** When average run length drops below 6 pixels, the 6 bits spent encoding each run exceeds the bits saved, **leading to data expansion** rather than compression.

Also, Gray code representation provides superior RLE compression (or in this case, lesser expansion) because it **increases spatial correlation within bit planes**. Gray code guarantees that consecutive integers differ by exactly one bit; therefore, in smooth gradient regions, most bit planes will exhibit no transitions at all, producing longer runs. The lower-order bit planes benefit most significantly: in binary representation, their average run length is only **2 pixels**, while in Gray code representation, this rises to above **6 pixels**.

2(b)

```
================================================================================
SUMMARY
================================================================================

Bit Plane                Symbols      Huffman Ratio   Avg Code Len
--------------------------------------------------------------------------------
gray_bit_plane_0         24041        1.0978          2.3118
gray_bit_plane_1         22097        1.0692          2.4495
gray_bit_plane_2         19456        1.0098          2.6276
gray_bit_plane_3         15523        0.8956          2.9210
gray_bit_plane_4         12274        0.7799          3.2167
gray_bit_plane_5         6616         0.5857          4.4819
gray_bit_plane_6         4858         0.4637          4.8318
gray_bit_plane_7         4000         0.4764          6.0297
--------------------------------------------------------------------------------
Average                               0.7973          3.6088

Note: Compression ratio = compressed bits / original bits
      Ratio < 1 means compression, Ratio > 1 means expansion
```

Summary of Compression Results Achieved Using Huffman Coding With Common Codebook

```
================================================================================
SUMMARY
================================================================================

Bit Plane                Huffman Ratio   Entropy      Avg Code Len
--------------------------------------------------------------------------------
gray_bit_plane_0         1.0203          2.1377       2.1485
gray_bit_plane_1         1.0109          2.2923       2.3160
gray_bit_plane_2         0.9844          2.5111       2.5615
gray_bit_plane_3         0.8893          2.8316       2.9001
gray_bit_plane_4         0.7682          3.1094       3.1685
gray_bit_plane_5         0.5416          4.1251       4.1443
gray_bit_plane_6         0.4234          4.3900       4.4125
gray_bit_plane_7         0.3961          4.9742       5.0137
--------------------------------------------------------------------------------
Average                  0.7543          3.2964       3.3332

Note: Compression ratio = compressed bits / original bits
      Ratio < 1 means compression, Ratio > 1 means expansion
```

Summary of Compression Results Achieved Using Huffman Coding With Individual Codebooks For Each Bit Plane

With Huffman coding, the compression ratios improve significantly. Most Gray-coded bit planes now show compression (in both approaches). The individual codebook approach achieves an overall compression ratio of **0.75**, while the common codebook approach achieves an overall compression ratio of **0.79**.

Huffman coding achieves compression ratios below 1 where fixed-length RLE could not because it employs variable-length codewords tailored to the statistical distribution of symbols. In the 6-bit RLE scheme, every run length—whether it occurs once or ten thousand times—consumes exactly 6 bits, ignoring the probability structure of the data. Huffman coding exploits the observation that run-length distributions are highly non-uniform: short runs (1–5 pixels) dominate in noisy bit planes while longer runs (50+ pixels) dominate in structured planes. By assigning shorter codewords (1–3 bits) to frequently occurring run lengths and longer codewords (8–12 bits) to rare run lengths, Huffman coding minimizes the average bits per symbol.

It is noted that the lowest bit planes still show very slight data expansion - this is due to very low possible compression due to high randomness, and that extra symbols are added for the dummy 0 run-lengths, which are added if the row begins with 0s instead of 1s.

The individual codebook approach consistently achieves better compression ratios than the common codebook approach. This is because each codebook is optimized for its specific symbol distribution. Higher-order bit planes have longer average run lengths and fewer unique symbols, resulting in highly skewed distributions that benefit significantly from tailored codebooks. Lower-order bit planes (0, 1) have more uniform, noise-like distributions where the difference between common and individual codebooks is less pronounced. But, the common codebook operates on a single codebook, while in the other case, 8 separate codebooks need to be built, stored and transmitted, which is a higher overhead.

Part 3

Aim

(a) Using the same monochrome image, obtain the predicted image and the error image using $a_1 = a_2 = a_3 = a_4 = 0.25$.
(b) Compute the histograms and the entropies of the original image and the error image.

Brief Theory

Predictive coding compresses sequential data by exploiting **redundancy**. In digital images, this is **spatial correlation**—neighboring pixels have statistical dependency. Adjacent pixels are similar due to:

1. **Physical continuity** of objects
2. **Smooth illumination**
3. **Limited high-frequency content**

The core idea is that if a pixel is accurately predicted from neighbors, only the

prediction error needs encoding, not the pixel value.

In a lossless predictive coding system, each pixel $s(n_1, n_2)$ at position (n_1, n_2) is predicted as $\hat{s}(n_1, n_2)$ using previously decoded pixels. The prediction error (also called the residual) is defined as:

$$e(n_1, n_2) = s(n_1, n_2) - \hat{s}(n_1, n_2)$$

This error signal is then encoded using entropy coding techniques (Huffman, Arithmetic, etc.). At the decoder, the **original pixel is perfectly reconstructed (**error captures the exact difference) via:

$$s(n_1, n_2) = \hat{s}(n_1, n_2) + e(n_1, n_2)$$

The predictor used in this implementation is a **4-neighbor linear predictor**, which computes the predicted value as a weighted sum of four causal neighbors:

$$\hat{s}(n_1, n_2) = a_1 \cdot s(n_1\text{-}1, n_2\text{-}1) + a_2 \cdot s(n_1\text{-}1, n_2) + a_3 \cdot s(n_1\text{-}1, n_2\text{+}1) + a_4 \cdot s(n_1, n_2\text{-}1)$$

Coefficient Constraint: $a_1 + a_2 + a_3 + a_4 = 1$

This constraint ensures that the predicted value remains within the valid intensity range [0, 255] for 8-bit images. In this assignment, we use **equal weights**: $a_1 = a_2 = a_3 = a_4 = 0.25$, which assumes isotropic (direction-independent) correlation.

Causality Principle: The predictor can only use pixels that have been previously transmitted/decoded. This ensures the decoder can replicate the prediction exactly.

Shannon Entropy measures the average information content (bits per symbol) of a source: $H = -\Sigma\, P(i) \cdot \log_2(P(i))$, where $P(i)$ is the probability of intensity value i.

For natural images, the error image $e(n_1, n_2)$ has a **sharply peaked distribution** centered at zero, because good predictions result in small errors. This concentration leads to: **H_error << H_original**

Theoretical Compression Ratio (Shannon's Theorem) = H_original / H_error

Solution Approach

STEP 1: Image Loading and Preprocessing

- Load monochrome image from file or accept numpy array
- Convert to grayscale (mode 'L') if necessary
- Store as 16-bit signed integer array for arithmetic operations

STEP 2: Predicted Image Computation (Handle Boundary cases)

FOR each pixel position (n_1, n_2):
  IF ($n_1 == 0$ AND $n_2 == 0$):
    $\hat{s}(n_1, n_2) = 0$ # First pixel, no neighbors

  ELSE IF ($n_1 == 0$): # First row
    $\hat{s}(n_1, n_2) = s(n_1, n_2-1)$ # Use left neighbor only

  ELSE IF ($n_2 == 0$): # First column
    $\hat{s}(n_1, n_2) = s(n_1-1, n_2)$ # Use top neighbor only

  ELSE IF ($n_2 == W-1$): # Last column (no top-right)
    # Redistribute a_3 weight among a_1, a_2, a_4
    total = $a_1 + a_2 + a_4$
    $\hat{s}(n_1, n_2) = (a_1/\text{total})\cdot s(n_1-1,n_2-1) + (a_2/\text{total})\cdot s(n_1-1,n_2) + (a_4/\text{total})\cdot s(n_1,n_2-1)$

  ELSE: # Interior pixels - full 4-neighbor prediction
    $\hat{s}(n_1, n_2) = 0.25\cdot s(n_1-1,n_2-1) + 0.25\cdot s(n_1-1,n_2) +$
      $0.25\cdot s(n_1-1,n_2+1) + 0.25\cdot s(n_1,n_2-1)$

  # Round to integer (lossless due to perceptual threshold)
  $\hat{s}(n_1, n_2) = \text{round}(\hat{s}(n_1, n_2))$

STEP 3: Error Image Computation

FOR each pixel position (n_1, n_2):

$e(n_1, n_2) = s(n_1, n_2) - \hat{s}(n_1, n_2)$

STEP 4: Histogram Generation

- Count frequency of each intensity value (0-255 for original)
- Count frequency of each error value (can be negative)
- Normalize to obtain probability distribution P(i)

STEP 5: Entropy Calculation

$H = -\Sigma\ P(i) \cdot \log_2(P(i))$ for all i with $P(i) > 0$

STEP 6: Statistical Analysis

- Compute error statistics: mean, std deviation, range
- Calculate compression potential: CR = H_original / H_error
- Verify lossless property: $s(n_1, n_2) = \hat{s}(n_1, n_2) + e(n_1, n_2)$

STEP 7: Visualization and Reporting

- Generate 2×3 subplot figure showing images and histograms
- Save outputs: predicted image, error image, plots, report

Implementation Details

Core Class: LosslessPredictiveCoder: The LosslessPredictiveCoder class encapsulates the entire predictive coding pipeline with attributes for storing the original image (int16), predicted image (int16), error image (int16), prediction coefficients, and image dimensions. The class provides methods for computing predictions, calculating error residuals, generating histograms, computing Shannon entropy, verifying lossless reconstruction, creating visualizations, generating statistical reports, and exporting results. This object-oriented design ensures modularity and allows the predictor to be easily instantiated with different images and coefficient sets while maintaining all intermediate results for analysis.

init(self, image_input, coefficients): The constructor initializes the coder by validating that prediction coefficients sum to 1.0 (ensuring valid intensity range preservation), loading the input image using PIL and converting it to grayscale

mode 'L' if necessary, storing the image as a 16-bit signed integer array to accommodate negative error values, and validating that the input is 2D grayscale. The function raises explicit errors for invalid coefficient sums or non-grayscale inputs, implementing defensive programming to catch configuration mistakes early in execution.

compute_predicted_image(self): This function generates predicted pixel values by iterating through all positions and applying the 4-neighbor linear model $\hat{s}(n_1,n_2) = a_1 \cdot s(n_1-1,n_2-1) + a_2 \cdot s(n_1-1,n_2) + a_3 \cdot s(n_1-1,n_2+1) + a_4 \cdot s(n_1,n_2-1)$, with special handling for boundaries: the first pixel (0,0) is predicted as 0, first-row pixels use only the left neighbor, first-column pixels use only the top neighbor, and last-column pixels redistribute the top-right coefficient weight among the three available neighbors to maintain the sum-to-one constraint. The predicted values are computed in floating-point for precision and then rounded to integers using np.round() before storage, which introduces a maximum ±0.5 error that is imperceptible and fully compensated by the error image. The algorithm runs in $O(H \times W)$ time with a single pass through all pixels.

compute_error_image(self): This function computes the residual by performing element-wise subtraction $e(n_1,n_2) = s(n_1,n_2) - \hat{s}(n_1,n_2)$ using NumPy array operations. The resulting error image contains signed integers in the range [-255, +255] for 8-bit inputs, where positive errors indicate underprediction, negative errors indicate overprediction, and zero indicates perfect prediction. For natural images with high spatial correlation, the error distribution is sharply peaked around zero with a Laplacian-like shape, resulting in significantly lower entropy than the original image and enabling efficient compression.

compute_histogram(self, image): This function generates intensity histograms by flattening the input array and counting occurrences of each value using np.histogram(), with dynamic bin range selection based on whether the input is an original image (bins from -0.5 to 255.5) or an error image (bins from minimum to maximum observed values). The function returns bin centers rather than edges for easier interpretation and plotting, calculating centers as the midpoint of each bin interval. This histogram serves as an empirical probability distribution used for entropy calculation and visualization.

compute_entropy(self, image): This function calculates Shannon entropy $H = -\Sigma P(i) \cdot \log_2(P(i))$ by first counting the frequency of each unique pixel value using Counter, converting these counts to probabilities by dividing by total pixel count, filtering out zero probabilities to avoid undefined logarithms, and summing the negative products of probabilities and their base-2 logarithms. The resulting entropy value represents the theoretical minimum average number of bits per pixel required for optimal lossless encoding, with typical values of 7-8 bits/pixel for natural images and 3-5 bits/pixel for error images.

verify_lossless_reconstruction(self): This function validates the lossless property by reconstructing the original image as the sum of predicted and error images, computing the maximum absolute difference between the original and reconstructed arrays, and checking if this difference is below a numerical precision threshold of 1e-10. The function returns a boolean indicating whether the test passed and the maximum error value for debugging purposes. For correct integer-based implementations, the maximum error should be exactly zero, confirming that the predictive coding scheme introduces no information loss.

theoretical_compression_ratio(self): This function computes the theoretical compression ratio as CR = H_original / H_error based on Shannon's source coding theorem, which states that the average code length cannot be less than the entropy. The ratio indicates how much smaller the optimally entropy-coded error image would be compared to the optimally entropy-coded original image.

compute_statistics(self): This function generates comprehensive error image statistics including minimum and maximum error values, mean (which should be near zero for an unbiased predictor), standard deviation (indicating prediction consistency), mean absolute error (MAE), root mean square error (RMSE). These metrics provide quantitative assessment of prediction quality, with good predictors typically showing mean near 0, small standard deviation, and 10-20% zero-error pixels for natural images.

visualize_results(self, save_path, dpi): This function creates a comprehensive 2×3 subplot visualization showing the original image, predicted image, and error image with 127 offset in the top row, and histograms of the original image, error image, and an entropy comparison bar chart in the bottom row.

create_test_image(): This utility function generates a simple 4×4 test array with values ranging from 100 to 113 arranged in a smooth gradient pattern, providing a minimal test case for algorithm validation where expected errors should be in the range [-5, +5] due to high local correlation.

create_sample_image(size): This function synthesizes a grayscale image of specified dimensions by combining horizontal and vertical gradients with diagonal stripe patterns, serving as a default input when no image file is provided and ensuring the program can execute without external dependencies.

main(): This function implements the command-line interface using Python's argparse module to accept optional arguments for image path (--image), output directory (--output), test mode (--test), and custom coefficients (--coefficients), instantiates the LosslessPredictiveCoder with appropriate inputs, executes the prediction and error computation pipeline, prints the analysis report to console, and saves all outputs to the specified directory with default fallbacks for each parameter.

Outputs

Statistical Analysis Output

=== LOSSLESS PREDICTIVE CODING ANALYSIS ===
Image: lena.png
Dimensions: 225 x 225
Coefficients: a1=0.25, a2=0.25, a3=0.25, a4=0.25

--- ENTROPY ANALYSIS ---
Original Image (UPLOADED BY USER) Entropy: 7.4663 bits/pixel
Error Image Entropy: 4.9676 bits/pixel
Entropy Reduction: 33.47%

--- COMPRESSION POTENTIAL ---
Theoretical CR (Shannon): 1.50:1
Average bits/pixel (FOR ANY MONOCHROME IMAGE (0-255)): 8.00
Average bits/pixel (error, optimal encoding): 4.97

--- ERROR STATISTICS ---
Error Range: [-121, 162]
Error Mean: -0.0225
Error Std Dev: 12.8817
Mean Absolute Error (MAE): 6.8273
Root Mean Square Error (RMSE): 12.8817
Error Q1: -3.00
Error Median: 0.00
Error Q3: 3.00
Pixels with zero error: 9067 (17.91%)
--- VERIFICATION ---
Reconstruction Test: PASS
Max reconstruction error: 0.0

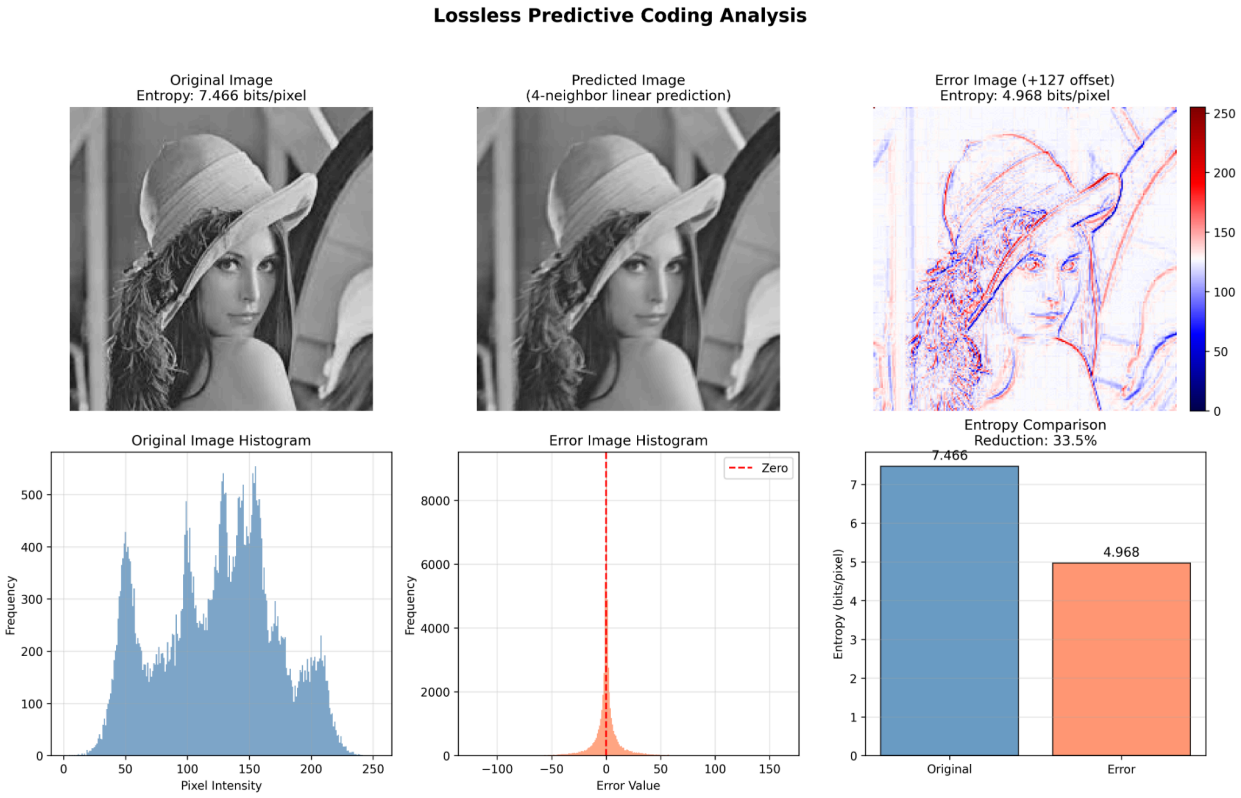


Figure 1 (top left): Original Image, Figure 2: Predicted Image, Figure 3: Error Image (with +127 offset), Figure 4: Histogram Analysis of original image, Figure 5: Histogram Analysis of error image