

MSc in Digital Humanities

Image Processing/Analysis



Image coding, compression, transmission

George Pavlidis, Paraskevi Tzouveli, Christos Tsinos

Course Outline

1. Introduction to Compression
2. Fundamentals of Information Theory
3. Entropy and Information Quantification
4. Entropy Encoding Techniques
5. Transform-Based Compression
6. Modern Approaches to Image Compression
7. Image Transmission

Why Do We Need Image Compression?



- Storage constraints
 - Large image files consume significant disk space
- Transmission efficiency
 - Large files slow down network transfer speeds
- Compression optimizes both storage and bandwidth usage
- Example file sizes
 - Relatively small colour image
 - $1024 \text{ px} \times 1024 \text{ px} \times 3 \text{ Bpp} = 3 \text{ MB}$
 - High-resolution scanned A4 documents
 - $21 \text{ cm} \times (600 \text{ dpi}/2.54) \times 29,7 \text{ cm} \times (600 \text{ dpi}/2.54) \times 3 \text{ Bpp} \approx 100 \text{ MB}$

** Bps = Bytes per pixel / bpp = bits per pixel*

Types of Image Compression



- Lossless Compression
 - Retains all original data
 - Examples: BMP, PNG, GIF, TIF, RAW, LZW, JPEG2000/JP2/JFIF
- Lossy Compression
 - Irreversibly removes data
 - Examples: JPEG, JPEG2000/JP2/JFIF, WebP, AVIF, HEIF, GIF

Fundamentals of Information Theory



- Claude Shannon's Communication Theory
 - Information is a measure of reduced uncertainty
 - Shannon's **Entropy** definition

$$H(X) = - \sum_i P(x_i) \log_2 P(x_i), \quad x_i \in \mathcal{X}$$

- Entropy determines theoretical limits for compression
- Higher entropy → More randomness → More bits required

Reprinted with corrections from *The Bell System Technical Journal*, Vol. 27, pp. 379–423, 623–656, July, October, 1948.

A Mathematical Theory of Communication

By C. E. SHANNON

INTRODUCTION

THE recent development of various methods of modulation such as PCM and PPM which exchange bandwidth for signal-to-noise ratio has intensified the interest in a general theory of communication. A basis for such a theory is contained in the important papers of Nyquist¹ and Hartley² on this subject. In the present paper we will extend the theory to include a number of new factors, in particular the effect of noise in the channel, and the savings possible due to the statistical structure of the original message and due to the nature of the final destination of the information.

The fundamental problem of communication is that of reproducing at one point either exactly or approximately a message selected at another point. Frequently the messages have *meaning*; that is they refer to or are correlated according to some system with certain physical or conceptual entities. These semantic aspects of communication are irrelevant to the engineering problem. The significant aspect is that the actual message is one *selected from a set* of possible messages. The system must be designed to operate for each possible selection, not just the one which will actually be chosen since this is unknown at the time of design.

If the number of messages in the set is finite then this number or any monotonic function of this number can be regarded as a measure of the information produced when one message is chosen from the set, all choices being equally likely. As was pointed out by Hartley the most natural choice is the logarithmic function. Although this definition must be generalized considerably when we consider the influence of the statistics of the message and when we have a continuous range of messages, we will in all cases use an essentially logarithmic measure.

C. E. Shannon, "A mathematical theory of communication," in *The Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, July 1948, doi: 10.1002/j.1538-7305.1948.tb01338.x.

Fundamentals of Information Theory



- Shannon's entropy
 - $H(X) = -\sum_i P(x_i) \log_2 P(x_i), x_i \in \mathcal{X}$
 - X is a random variable
 - taking values from a set $\mathcal{X}$ of possible values $\{x_1, x_2, \dots, x_n\}$
- **X : the question or uncertain situation**
 - X is the whole “game” or “situation”.
 - Example: “Tomorrow’s weather” as a variable that can be *sunny*, *rainy*, *cloudy*, etc.
 - We don’t know yet what will actually happen; we only know the possibilities.

Fundamentals of Information Theory



- Shannon's entropy
 - $H(X) = -\sum_i P(x_i) \log_2 P(x_i), x_i \in \mathcal{X}$
- x_i : **each possible answer**
 - These are the concrete outcomes:
 - x_1 = “sunny”
 - x_2 = “rainy”
 - x_3 = “cloudy”,...etc
 - The index i just counts them: $\{1, 2, \dots, n\}$
- So: **each x_i is one possible answer the universe might choose**

Fundamentals of Information Theory



- Shannon's entropy
 - $H(X) = -\sum_i P(x_i) \log_2 P(x_i), x_i \in \mathcal{X}$
- $P(x_i)$: **“how common” or “how likely” each answer is**
 - To each possible answer x_i , we attach a probability $P(x_i)$.
 - Example:
 - $P(x_1) = 0.6$ (60% chance sunny)
 - $P(x_2) = 0.3$ (30% chance rainy)
 - $P(x_3) = 0.1$ (10% chance cloudy)
 - The $P(x_i)$ must add up to 1 (100%): $P(x_1) + P(x_2) + \dots + P(x_n) = 1$
- So $P(x_i)$ **tells us how expected or unexpected each outcome is**

Fundamentals of Information Theory



- Shannon's entropy
 - $H(X) = -\sum_i P(x_i) \log_2 P(x_i), x_i \in \mathcal{X}$
- $\log_2 P(x_i)$: **turning likelihood into “surprise”**
 - When something is **very likely** ($P(x_i)$ close to 1), $\log_2 P(x_i)$ is close to 0 → **little surprise**.
 - When something is **unlikely** ($P(x_i)$ small), $\log_2 P(x_i)$ is a big negative number → **a lot of surprise** when it happens.
- We then put a minus sign: $-\log_2 P(x_i)$
 - that quantity becomes a **positive number** measuring:
 - ***“How surprised would I be if outcome x_i actually happens?”***

Fundamentals of Information Theory



- Shannon's entropy
 - $H(X) = -\sum_i P(x_i) \log_2 P(x_i), x_i \in \mathcal{X}$
- This quantity $-\log_2 P(x_i)$ is called the **information** or **surprise** of that single i outcome.
- The **minus** is just to make the surprise positive
- The **base 2** means we measure information in **bits** (yes/no)
 - **Outcome** x_i
 - **Probability** $P(x_i)$
 - **Surprise if it happens:** $I(x_i) = -\log_2 P(x_i)$
 - *surprise of outcome x_i in bit units*

Fundamentals of Information Theory



- Shannon's entropy
 - $H(X) = -\sum_i P(x_i) \log_2 P(x_i), x_i \in \mathcal{X}$
- But before the event happens, we **don't know which outcome** we will get. We only know:
 - If a very likely outcome happens, we'll get **little surprise**.
 - If a very unlikely outcome happens, we'll get **a lot of surprise**.
 - Each outcome has:
 - its own **probability** $P(x_i)$
 - and its own **surprise** $I(x_i)$
 - So what we really want is:
 - *The average surprise we can expect, given the whole distribution of possibilities*

Fundamentals of Information Theory



- Shannon's entropy
 - $H(X) = -\sum_i P(x_i) \log_2 P(x_i), x_i \in \mathcal{X}$
- ***“If I play this ‘uncertainty game’ again and again, on average how surprised will I be each time?”***
- So the **Σ (sum)** is there because:
 - We must **add up the contributions of all possible outcomes**, each weighted by how often they occur, to get the **average surprise**

Information Theory Introduction



1. Concept of Information & Surprise

- ♦ What is "information"?
 - Information is something that **reduces our uncertainty** about an event
 - More **surprising** events carry **more information**
 - More **predictable** events carry **less information**
 - An important equation about "information content"

$$I(X) = -\log_2 P(X)$$

Information Theory Introduction



1. Concept of Information & Surprise

- **◆ Example: Rolling a Dice**
 - **A fair 6-sided die**
 - corresponds to 1 bit per event
 - (since both outcomes are equally likely)
 - gives us $I(X) = -\log_2(1/6) \approx 2.58$ **bits of information**
 - **A biased die (favoring one number)**
 - gives **less information** because it's more predictable



Information Theory Introduction

- ⚠ **Some key insights**

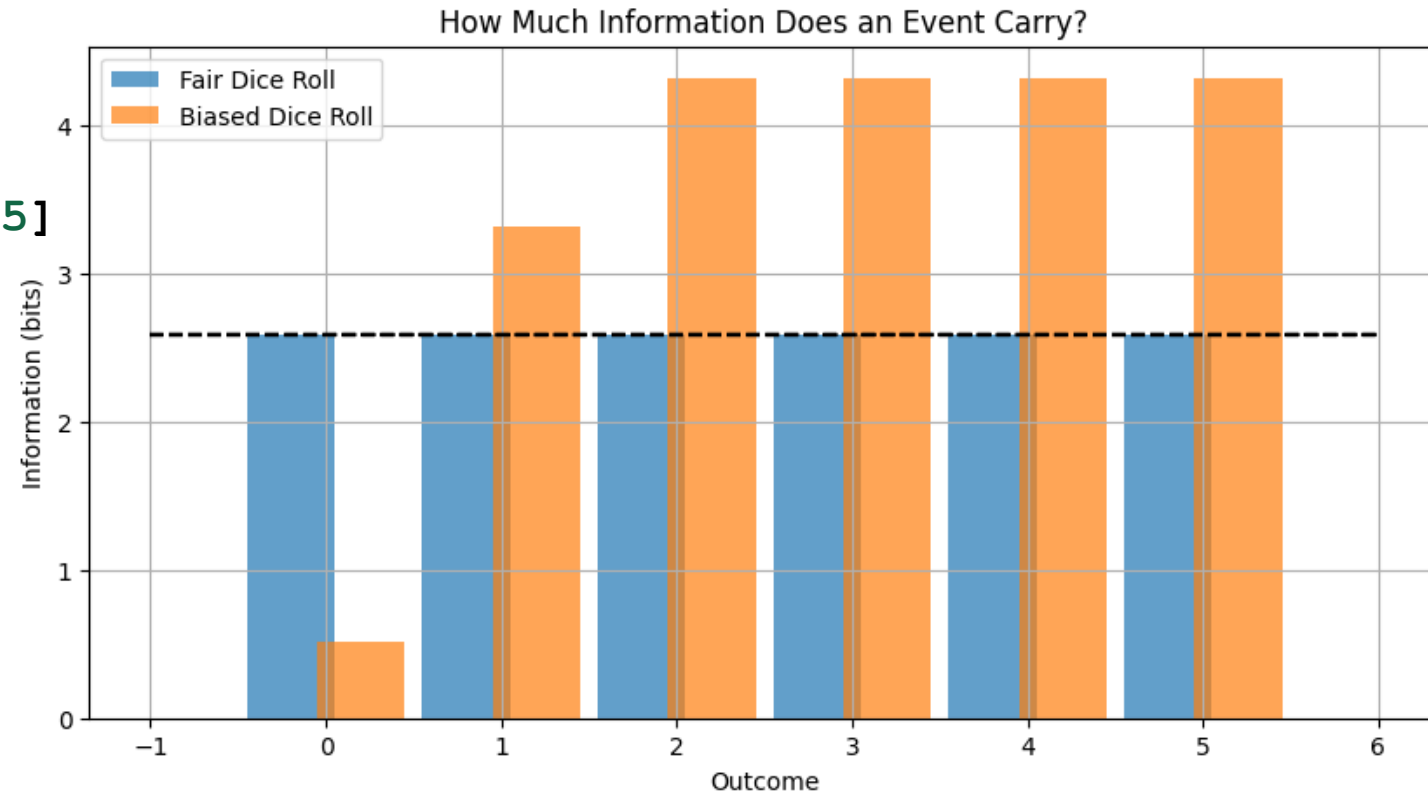
- **A fair die** carries more information per roll (2.58 bits) than a fair coin toss (1 bit), since **all numbers are unexpected!**
- **A biased die** carries less information per roll for the most expected face (the bias) because the outcome is more predictable
- **The rarer an event, the more information it includes/gives**
- **If an event is completely certain (probability = 1), it provides 0 bits of information**

Event	Probability $P(x)$	Information Content $I(x)$
Fair Die (1-6)	$1/6 \approx 0.1667$	2.58 bits
Biased Die (for 1)	0.7	0.51 bits
Biased Die (for 2)	0.1	3.33 bits
Biased Die (for 3-6)	0.05	4.32 bits

Information Theory Introduction



- Fair
 - $[1/6, 1/6, 1/6, 1/6, 1/6, 1/6]$
- Biased
 - $[0.7, 0.1, 0.05, 0.05, 0.05, 0.05]$

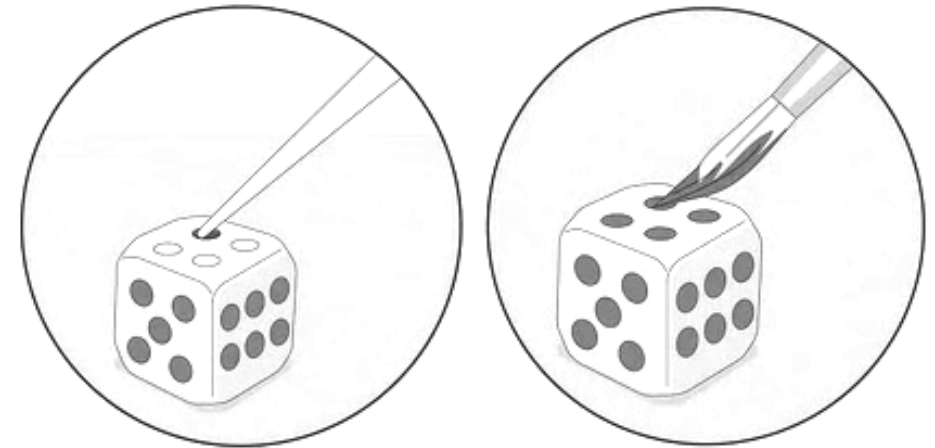


Information Theory Introduction



2. Measuring Uncertainty with Probability

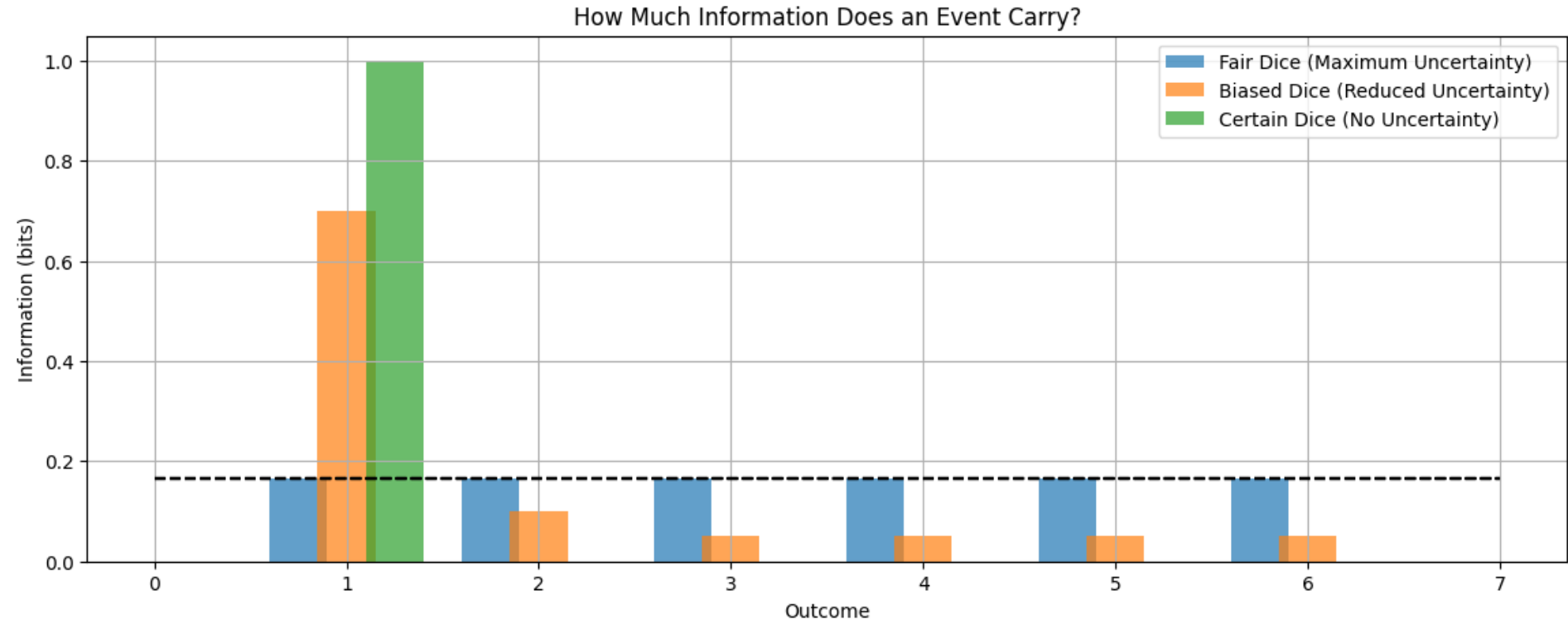
- ◆ **Key Idea: More Predictable = Less Uncertainty**
 - If an event is **equally likely** (fair die), we have **maximum uncertainty**
 - If an event is **biased** (loaded die), we have **less uncertainty**
 - If an event is **completely predictable**, we have **zero uncertainty**
- ◆ **Examples**
 - 1 **Fair Die (Maximum Uncertainty)**
 - Rolling a **fair 6-sided die**: All numbers **equally likely**
 - We don't know what will happen → **high uncertainty**
 - 2 **Biased Die (Reduced Uncertainty)**
 - If a die is **heavily biased toward 1**, we are **less uncertain**
 - We already expect **1 to appear most of the time**
 - 3 **Completely Predictable Event (No Uncertainty)**
 - A rigged die that **always lands on 1** → **0 uncertainty**
 - We gain **no new information** from rolling it



Information Theory Introduction



- Fair
- Biased (1)
- Certain (1)



Information Theory Introduction



3. Entropy as a Formal Measure of Uncertainty

- ◆ **What Is Entropy?**

- Entropy tells us **the average amount of information per event**. It is defined as:

$$H(X) = \mathbb{E}[I(X)] = -\sum P(X) \log_2 P(X), \quad X = [x_1, x_2, \dots, x_n]$$

- $P(X)$ is the probability of an outcome, $H(X)$ is the **entropy (in bits per symbol)**

- ◆ **Intuition Behind Entropy**

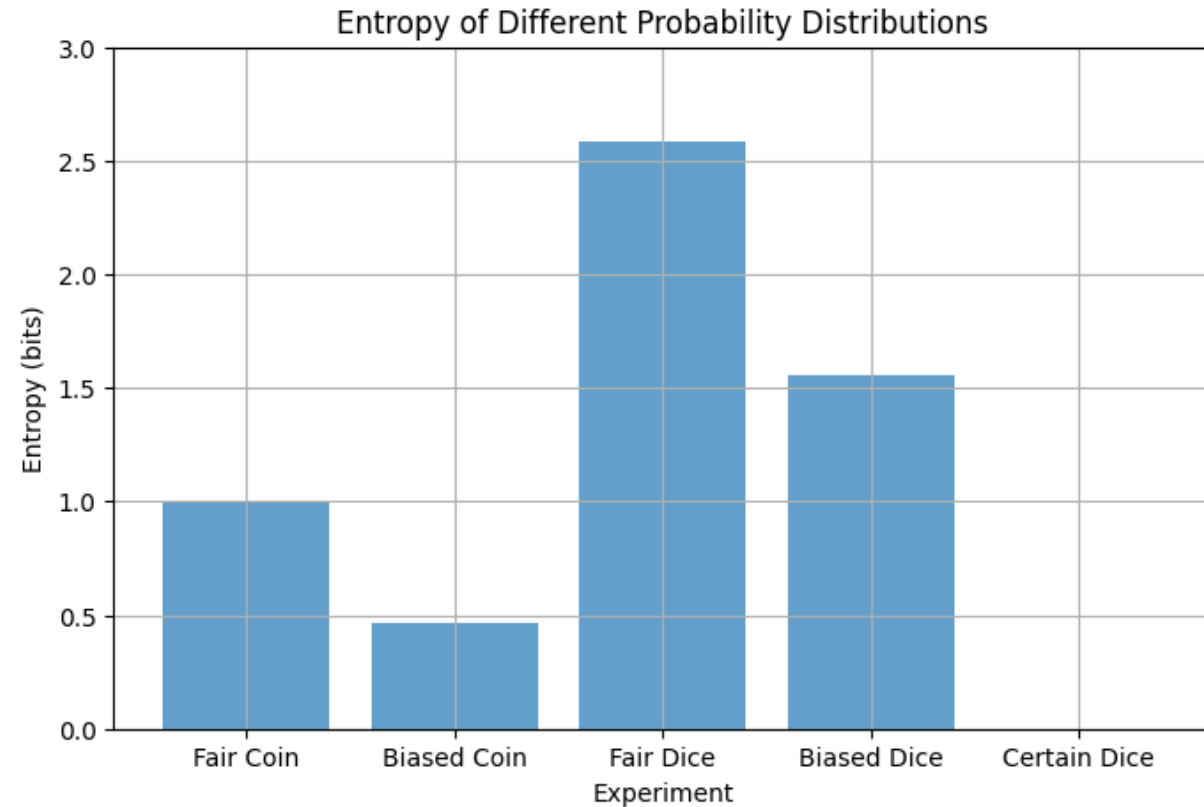
- **Higher entropy** = More randomness, harder to predict (e.g., fair dice)
- **Lower entropy** = More predictable, less information per event (e.g., biased dice)
- **Zero entropy** = No uncertainty (e.g., a “certain” die that always rolls “1”)

Information Theory Introduction



• ♦ Examples

- Fair coin
 - $[0.5, 0.5]$
- Biased coin
 - $[0.9, 0.1]$
- Fair die
 - $[1/6, 1/6, 1/6, 1/6, 1/6, 1/6]$
- Biased die
 - $[0.7, 0.1, 0.05, 0.05, 0.05, 0.05]$
- Certain die
 - $[0.9, 0.1, 0.0, 0.0, 0.0, 0.0]$



$$H_{\text{biased coin}}(X) = -\sum P(X) \log_2 P(X) = -(0.9 \times \log_2 0.9 + 0.1 \times \log_2 0.1) = -(-0.137 - 0.332) = 0.469$$

Information Theory Introduction



-  **Why Use $\log_2$ in Entropy Calculation?**
- In information theory, we use $\log_2$ (**log base 2**) because **information is measured in bits**
 -  **Information is Stored in Bits (Binary)**
 - **Computers store everything in 0s and 1s** (binary digits)
 - If an event has **N possible outcomes**, $\log_2(N)$ tells us **how many binary questions (yes/no) we need** to identify the correct outcome
 - **Example: Guessing a Number from 1 to 8**
 - You can ask **yes/no questions** (binary choices):
 - Is it in the range 1-4? 
 - Is it in 1-2? 
 - Is it 1?  → Must be 2
 - This takes $\log_2(8)=3$ questions → **3 bits needed**

Information Theory Introduction



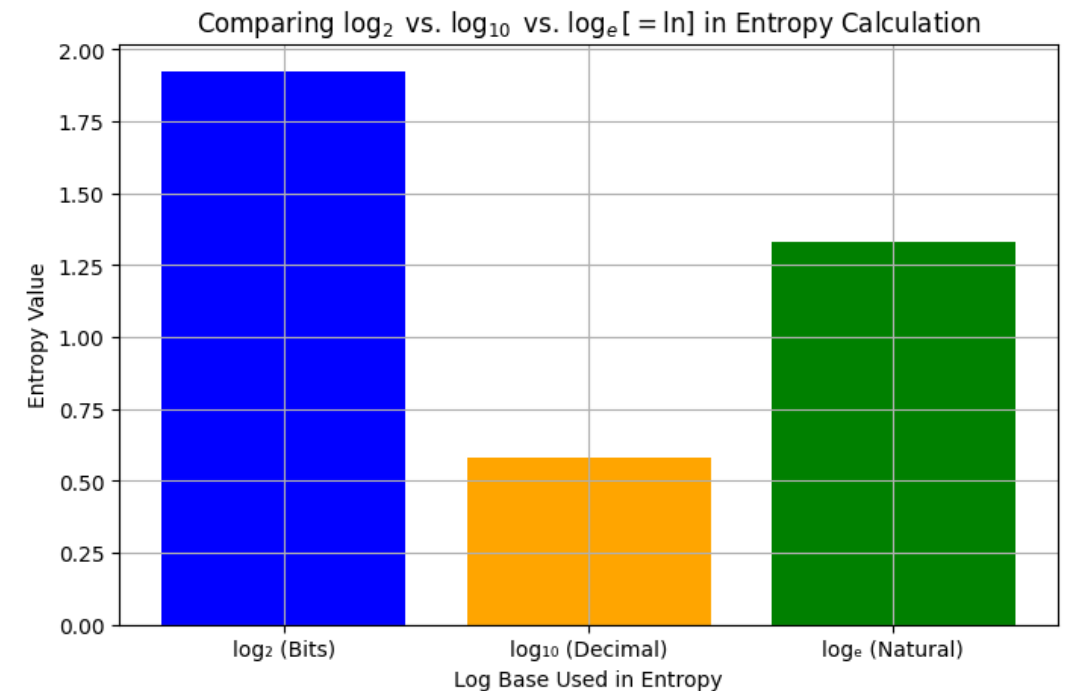
-  **Why Use $\log_2$ in Entropy Calculation?**
 -  **Why Not Use $\log_{10}$ or $\log_e$ (ln)?**
 - Other log bases measure **different things**:
 - $\log_2(N) \rightarrow$ **Measures bits (binary system, used in entropy)**
 - $\log_{10}(N) \rightarrow$ **Measures decimal digit-based uncertainty (not useful for computers)**
 - $\log_e(N)$ (natural log) $\rightarrow$ **Used in advanced models but not ideal for digital data**
 -  **Practical Meaning of $\log_2$ in Entropy**
 - A **fair coin flip** has $\log_2(2) = 1$ **bit** (one yes/no question needed)
 - A **fair 6-sided die** has $\log_2(6) \approx 2.58$ **bits** (harder to predict)
 - A **biased event** (where one outcome is more likely) has **lower entropy**

Information Theory Introduction



- Comparing **Entropy Calculations**

- Event with probabilities
 - [0.5, 0.25, 0.125, 0.1, 0.05]
- ♦ Entropy using $\log_2$ (bits)
 - 1.9233 bits
- ♦ Entropy using $\log_{10}$ (decimal digits)
 - 0.5790 digits
- ♦ Entropy using $\log_e$ (natural log)
 - 1.3331 nats (natural units)



Information Theory Introduction



-  **How Entropy Determines Compression Limits**
 - The **lower the entropy**, the more we can **compress data without losing information**
 -  **Shannon's Source Coding Theorem**
 - Shannon's theorem states:
$$\text{Minimum Bits Per Symbol} \geq H(X)$$
 - This tells us **the theoretical lower bound** for compression
 -  **Key Observations**
 - **High entropy = Hard to compress** (e.g., random text, noisy images)
 - **Low entropy = Easy to compress** (e.g., repetitive text, structured images)
 - **If entropy = 0, we need 0 bits!** (A completely predictable message needs no storage)

Information Theory Introduction



-  **How Entropy Determines Compression Limits**
 - **Text example:**
 - “aaaaaaaaaa” → low entropy, high redundancy
 - “hellohello” → medium entropy, medium redundancy
 - “h3X&7p0@!x” → random-like, high entropy, low redundancy
 - **Image example:**
 - A **solid color image** (low entropy) compresses well
 - A **random noise image** (high entropy) doesn’t compress

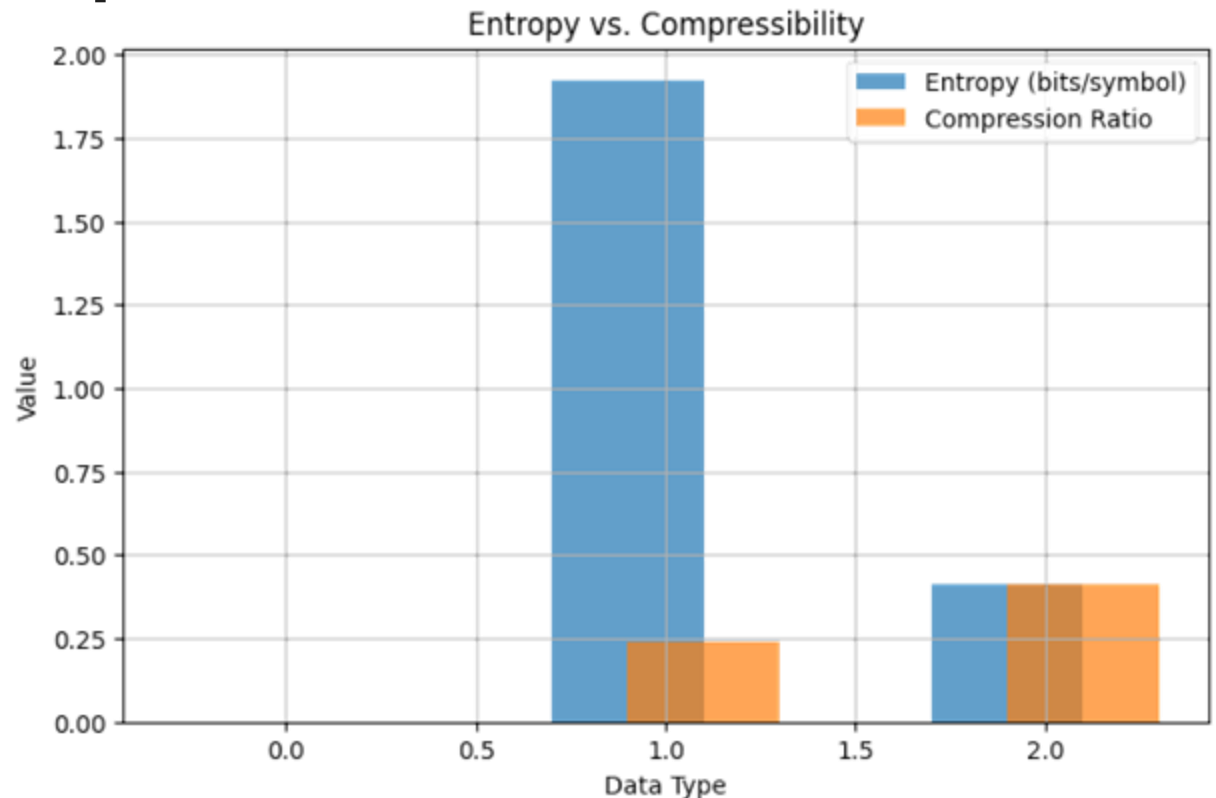
Information Theory Introduction



- **How Entropy Determines Compression Limits**

- **Text example:**

- “aaaaaaaaaa”
 - Entropy = 0.00 bits/symbol
 - Compression Ratio = inf
- “hellohello”
 - Entropy = 1.92 bits/symbol
 - Compression Ratio = 0.24
- “h3X&7p0@lx”
 - Entropy = 3.32 bits/symbol
 - Compression Ratio = 0.42



* $\text{Compression Ratio} = \text{Entropy (bps)} / 8 \text{ bps}$

Information Theory Introduction



-  **Entropy Calculation for Image Compression**
 - Entropy is a **measure of information content** in an image
 - Higher entropy means **more randomness**, making compression harder
 - Lower entropy means **more redundancy**, allowing efficient compression

-  **How Entropy Works**
 - The entropy of an image is computed as

$$H(X) = -\sum P(x) \log_2 P(x) \text{ where } P(x) \text{ is the probability of pixel intensity } x$$

- A **high-entropy image** has more unique pixel values (e.g., a noisy image)
 - A **low-entropy image** has fewer unique pixel values (e.g., a scanned document with a uniform background)
-  **Why Is Entropy Important?**
 -  **Predicts Compressibility:** Images with low entropy can be compressed efficiently
 -  **Used in Compression Algorithms:** Huffman and Arithmetic Coding rely on entropy estimates
 -  **Fundamental to Information Theory:** Sets the **theoretical limit** for lossless compression

Image Entropy and Compressibility



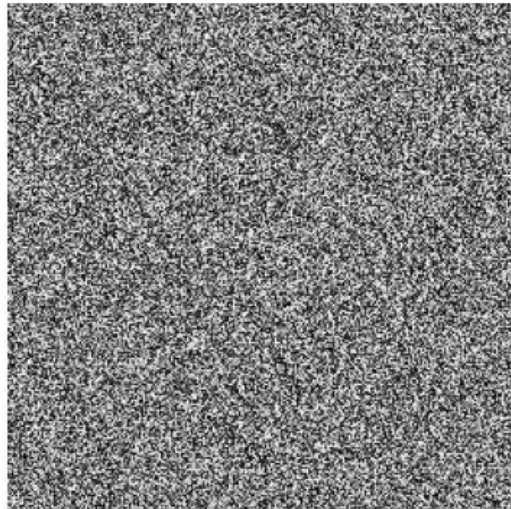
- **Examples**

- Noise images

- Grayscale Image Entropy: 7.9974042521904805

- Color Image Entropy (All Channels Combined): 7.998979616544033

Grayscale Image



Color Image

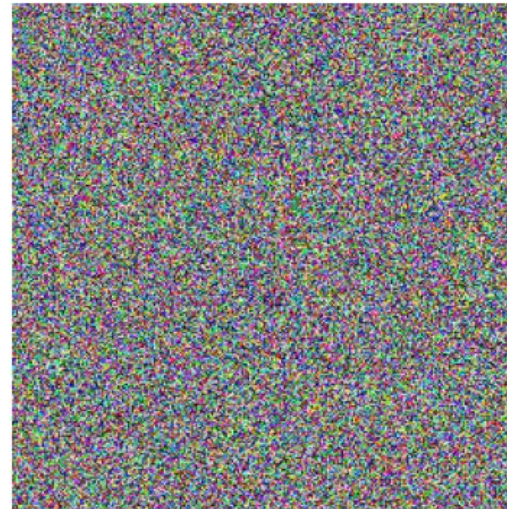


Image Entropy and Compressibility



- **More Examples**

- Checkerboard Entropy: 1.0
- Grayscale Image Entropy: 7.73131396007213
- Color Image Entropy (All Channels Combined): 7.239398824483783

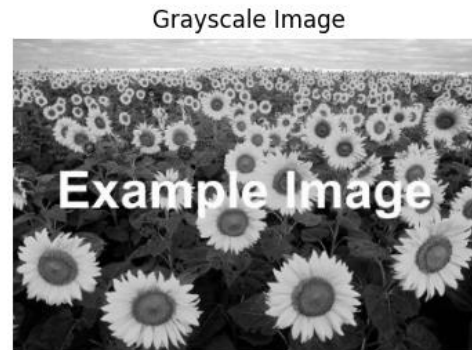
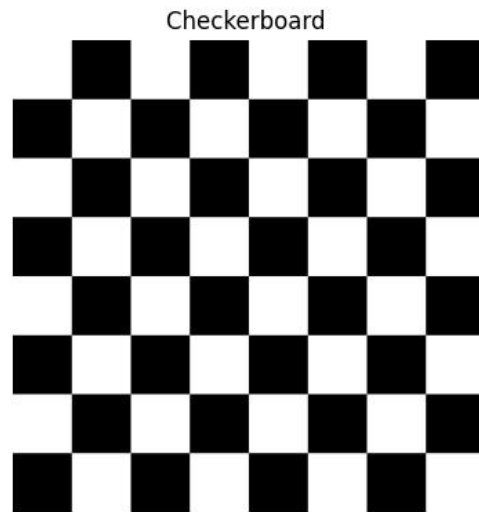


Image Entropy and Compressibility



-  **Image Histograms**
 - **Histograms** show the **pixel intensity distributions**
 - Grayscale Image Histogram
 - The distribution of pixel intensities affects the entropy
 - A **uniform distribution** (spread-out values) results in higher entropy
 - A **peaked distribution** (many similar values) results in lower entropy (more redundancy)
 - Color Image Histogram
 - Each color channel (Red, Green, Blue) has its own distribution
 - If all channels have **high entropy**, the image is rich in information
 - If some channels have **low entropy**, compression can remove redundant information

Image Entropy and Compressibility



- **Image Histograms**

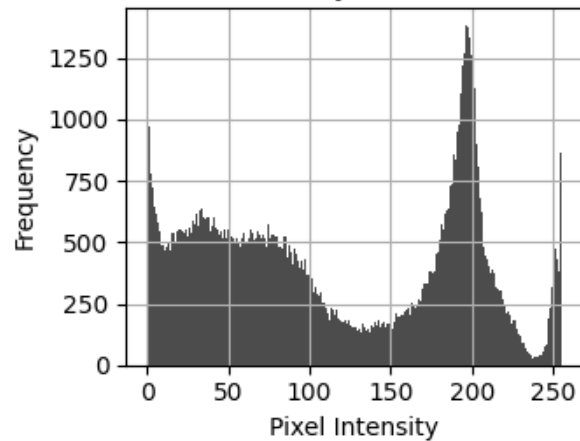
Grayscale Image



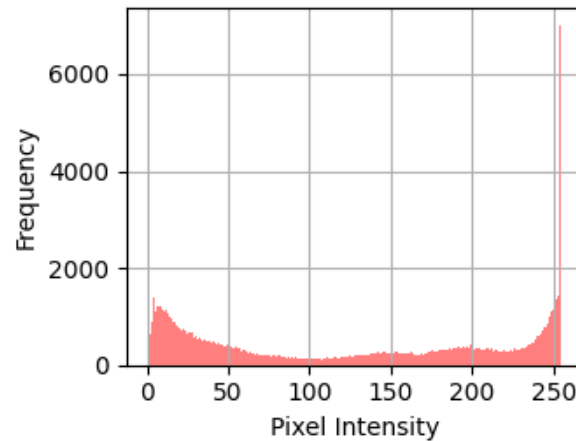
Color Image



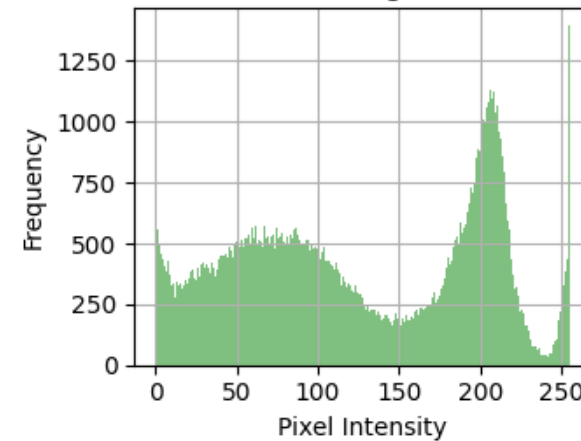
Grayscale



Channel red



Channel green



Channel blue

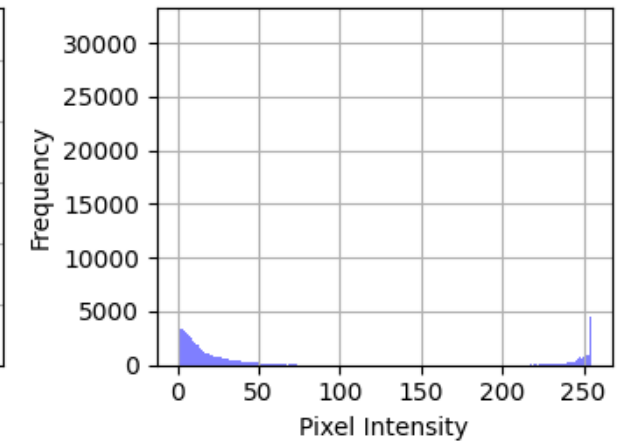


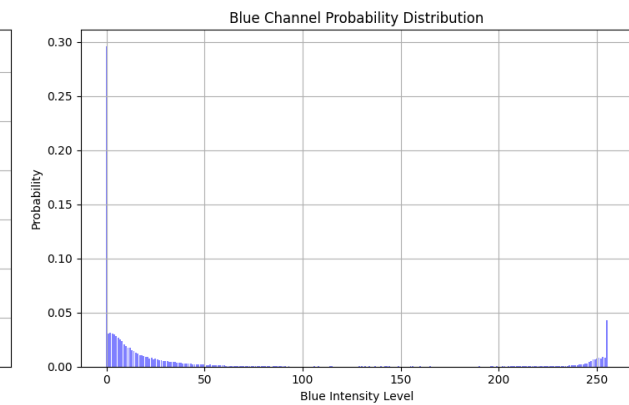
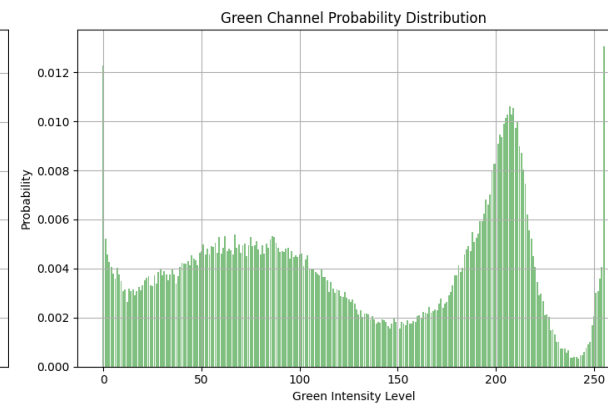
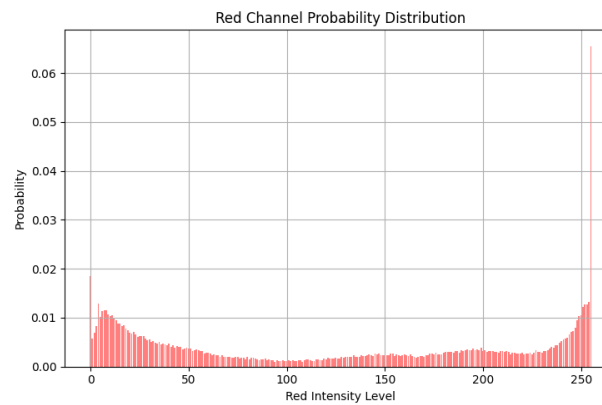
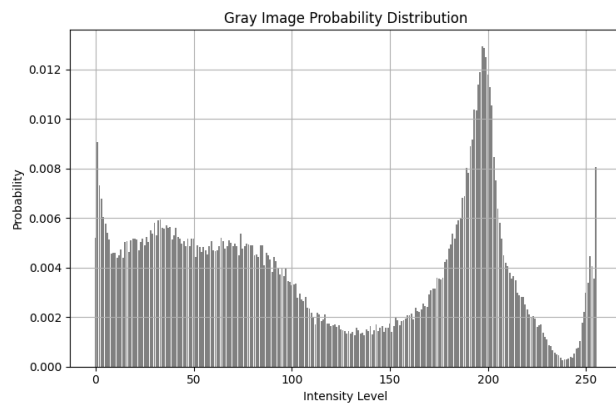
Image Entropy and Compressibility



- 📌 From Image Histograms to Pixel Probabilities

$$P(i) = \frac{h(i)}{N}, \quad \forall i (= 1 \dots L) \text{ pixels}$$

$$h(i) = \text{Number of pixels with intensity } i, \quad N = \sum_{i=1}^L h(i) \text{ the total number of pixels}$$



Understanding Image Redundancy



-  **Spatial Redundancy**
 - Neighboring pixels in an image are often highly correlated
 - This means that if you look at two adjacent pixels, their intensity (or color) values are usually very similar
 - Compression algorithms can take advantage of this similarity to reduce data size
-  **Spectral Redundancy**
 - In color images, the different channels (e.g., Red, Green, and Blue) usually contain overlapping information
 - This redundancy can be exploited to compress the image by reducing the amount of duplicate information across channels
-  **Compression exploits redundancies**
 - By identifying and removing these redundancies, image compression methods (such as JPEG) are able to reduce file sizes while maintaining acceptable visual quality

Understanding Image Redundancy



-  Example of Spatial & Spectral Redundancy
 - Pearson Correlation Coefficient

$$r = \frac{\sum_{i=1}^n (X_i - \hat{X})(Y_i - \hat{Y})}{\sqrt{\sum_{i=1}^n (X_i - \hat{X})^2} \sqrt{\sum_{i=1}^n (Y_i - \hat{Y})^2}}, \quad -1 \leq r \leq 1$$



- Spatial Redundancy: 0.9423466474718738
- Spectral Redundancy (R-G Correlation): 0.9189365580324964
- Spectral Redundancy (G-B Correlation): 0.5754888041364062
- Spectral Redundancy (R-B Correlation): 0.37055784814338844

Lossless Compression Methods



- ☒ **Huffman Coding**
 - Assigns shorter codes to frequent symbols
- ☒ **Arithmetic Coding**
 - Represents data as a single fractional value
- **LZW**
 - Dictionary-based encoding (Used in GIF, TIFF)

Huffman Coding of Images



- Huffman Coding is a **variable-length encoding algorithm**
 - Assigns **shorter codes to frequent symbols**
 - Assigns **longer codes to rare symbols**
 - It is a widely used in numerous **lossless compression techniques**
- **◆ How Huffman Coding Works**
 1. **Build a Frequency Table:** Count occurrences of each pixel value
 2. **Construct a Binary Tree:**
 1. The most frequent symbols are placed **closer to the root**
 2. Rare symbols get **longer bit representations**
 3. **Generate Huffman Codes:**
 1. Assign **shorter binary codes** to more frequent symbols
 2. Store the compressed image as a **bitstream**
- **◆ Why Use Huffman Coding?**
 - **✓ Guaranteed Optimality:** Produces the **shortest possible lossless encoding** based on symbol frequencies
 - **✓ Used in PNG, TIFF, and ZIP compression**
 - **✓ Fast Encoding & Decoding:** Efficient for text and images

Huffman Coding

- Huffman Coding is a **variable-length encoding algorithm**
 - Assigns **shorter codes to frequent symbols**
 - Assigns **longer codes to rare symbols**
 - It is a widely used in numerous **lossless compression techniques**
- **◆ How Huffman Coding Works**
 1. **Build a Frequency Table:** Count occurrences of each pixel value
 2. **Construct a Binary Tree:**
 1. The most frequent symbols are placed **closer to the root**
 2. Rare symbols get **longer bit representations**
 3. **Generate Huffman Codes:**
 1. Assign **shorter binary codes** to more frequent symbols
 2. Store the compressed image as a **bitstream**
- **◆ Why Use Huffman Coding?**
 - **✓ Guaranteed Optimality:** Produces the **shortest possible lossless encoding** based on symbol frequencies
 - **✓ Used in PNG, TIFF, and ZIP compression**
 - **✓ Fast Encoding & Decoding:** Efficient for text and images

Huffman Coding

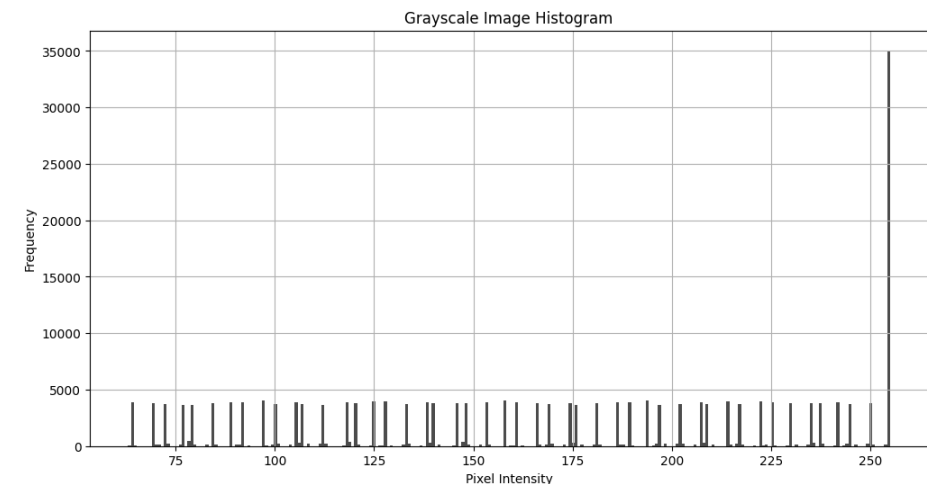


1. Analyze symbol frequencies
 - count how often each symbol (character, byte, etc.) appears in the data
 - Example: For the message "**abracadabra**"
 - Character Frequencies: {'a': 5, 'b': 2, 'r': 2, 'c': 1, 'd': 1}
2. Build a binary tree
 - Start with a forest of nodes, each representing a symbol and its frequency
 - Repeatedly combine the two nodes with the smallest frequencies into a new parent node
 - Continue until all nodes are combined into a single tree
3. Assign shorter codes to frequent symbols
 - Traverse the tree from the root to each leaf
 - Assign '0' for left branches and '1' for right branches
 - Frequent symbols will have shorter paths (and thus shorter codes)
 - Huffman Codes: {'a': '0', 'd': '100', 'c': '101', 'r': '110', 'b': '111'}
4. Encode data efficiently
 - Replace each symbol in the original data with its corresponding Huffman code
 - Example: "**abracadabra**" (11 bytes) becomes **01111100101010001111100** (23 bits)

Huffman Coding of Images

- ⚠ **Example with a simple image**
 - 510 × 443 px grayscale image → 158 unique grays

```
[ 63  64  65  68  69  70  71  72  73  76  77  78  79  80  83  84  85  86
  88  89  90  91  92  93  97  98  99 100 101 103 104 105 106 107 108 109
 111 112 113 114 117 118 119 120 121 122 123 124 125 126 127 128 129 132
 133 134 137 138 139 140 141 144 145 146 147 148 149 151 152 153 154 155
 156 157 158 159 160 161 162 165 166 167 168 169 170 171 173 174 175 176
 177 178 179 180 181 182 183 185 186 187 188 189 190 191 194 195 196 197
 198 200 201 202 203 205 206 207 208 209 210 211 213 214 215 216 217 218
 219 221 222 223 224 225 226 229 230 231 233 234 235 236 237 238 240 241
 242 243 244 245 246 247 248 249 250 251 252 253 254 255]
[  64 3904  64  7 3842 120 117 3730 185 129 3667 418
 3666 121 124 3783 124  1  5 3903 123 124 3844 65
 4032  64 121 3729 182  1 124 3848 304 3733 181  2
 236 3611 183  1  65 3854 357 3787 129  3  6 62
 3948  69  71 3968  65 123 3728 182  71 3904 303 3791
 123  1  66 3790 355 3788 128  2 126 3846 122  1
  1  5 4024  63  64 3905  64 10 3836 123 120 3729
 183  1 122 3786 303 3666 122  1  1 124 3782 124
  1 12 3898 122 122 3844  65  1 4032  64 180 3671
 181  1 184 3729 182  1 127 3847 302 3722 128  1
 11 3961 124 239 3729 126  3  65 3966  64 128 3904
  64  66 3839 127  1 131 3782 304 3788 186  1 67
 3901  64 242 3720 130  4  3 183 3780 128  1  2
 145 35059]
```



A 6x6 grid of squares showing a grayscale gradient. The top-left square is the darkest gray, and the bottom-right square is white. The gradient transitions smoothly from dark to light across the grid.

-

Arithmetic Coding

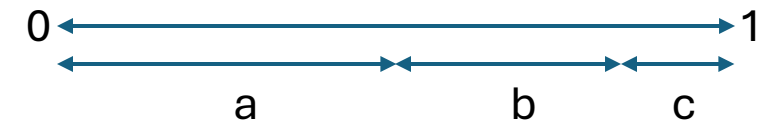


- Key idea: Represents an entire message as a single fractional value within a range $[0, 1)$
- Why it's useful: More efficient than Huffman coding for highly skewed symbol distributions
- Used in modern codecs due to efficiency gains

Arithmetic coding



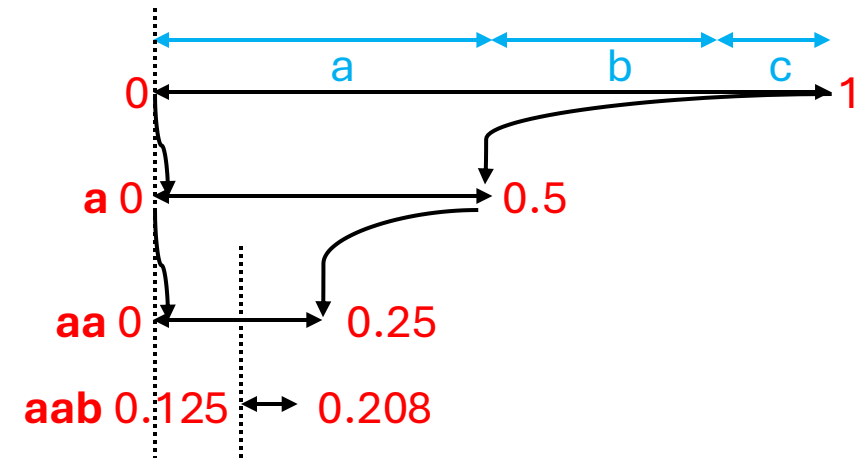
- Step 1 – Analyse Symbol Probabilities
 - Determine the probability (or frequency) of each symbol in the data
 - Example: For the string "**aababc**" probabilities: **a**=3/6, **b**=2/6, **c**=1/6
- Step 2 – Define Initial Range
 - Start with the initial range [0, 1)
 - Divide this range into sub-ranges based on symbol probabilities
 - Example: **a**=[0, 0.5), **b**=[0.5, 0.83), **c**=[0.83, 1)
- Step 3 – Encode Symbols Iteratively (next slide)
- Step 4 – Finalize the Encoded Value (next slide)



Arithmetic coding



- Iterative encoding of symbols (Step 3)
 - For each symbol in the message
 - Narrow the current range to the sub-range corresponding to the symbol
 - Update the range based on the new probabilities
 - Example: Encoding "**aababc**"
 - Start with $[0, 1)$
 - After 'a': $[0, 0.5)$
 - After 'aa': $[0, 0.25)$
 - After 'aab': $[0.125, 0.208)$
 - After 'aaba': $[0.125, 0.167)$
 - After 'aabab': $[0.146, 0.160)$
 - After 'aababc': $[0.157, 0.160)$
- Finalize the coding (Step 4)
 - The final range represents the entire message
 - Choose a fractional value within this range (midpoint) as the encoded value
 - Example: For the final range $[0.157, 0.160)$, choose 0.1585648...



Arithmetic coding



- Decoding Steps
 - Start with the encoded fractional value and initial range $[0, 1)$
 - Determine the symbol by comparing the value with sub-ranges
 - Example: Compare the encoded value (**0.1585648...**) with the symbol ranges
 - It falls in the range **$[0, 0.5)$** , so the first symbol is 'a'
 - Narrow the range to the sub-range of the decoded symbol
 - Update the current range to the range of 'a': **$[0, 0.5)$**
 - Rescale the encoded value to the new range
 - Formula: $(\text{encoded_value} - \text{range_start}) / \text{range_width}$
 - Example: Rescaled value: $(0.1585648 - 0) / 0.5 = 0.3171296$
 - Repeat until all symbols are decoded

Arithmetic coding



- Decoding steps in the example

Step	Encoded Value	Current Range	Rescaled Value	Decoded Symbol	New Range
1	0.1585648	[0, 1)	-	a	[0, 0.5)
2	0.1585648	[0, 0.5)	0.3171296	a	[0, 0.25)
3	0.1585648	[0, 0.25)	0.6342592	b	[0.125, 0.208)
4	0.1585648	[0.125, 0.208)	0.406846	a	[0.125, 0.1665)
5	0.1585648	[0.125, 0.1665)	0.813692	c	[0.15625, 0.1665)

$$\text{RescaledValue} = \frac{\text{EncodedValue} - \text{RangeStart}}{\text{RangeWidth}}$$

Arithmetic Coding for Image Compression



- **Why Use Arithmetic Coding?**
 -  **Higher Compression Efficiency:** Uses **fractional bits per symbol** instead of rounding to full bits
 -  **Adaptive:** Works well for data with **skewed probability distributions**
 -  **Used in Standards:** JPEG 2000, H.264 Video Compression, and other modern codecs
- **Limitations & Challenges**
 -  **Precision Issues:** The interval can shrink too much, leading to floating-point accuracy problems
 -  **Decoding Sensitivity:** A small error in precision can cause incorrect pixel values
 -  **Not Always More Efficient:** For small datasets, Huffman coding may perform equally well

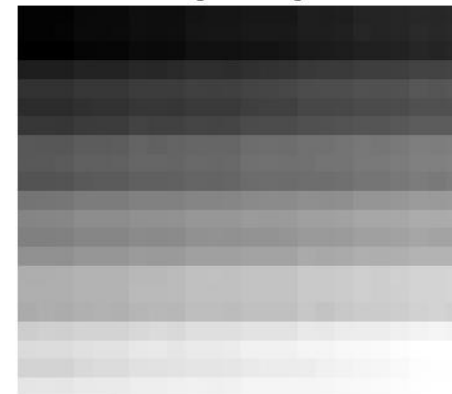
Arithmetic Coding for Image Compression



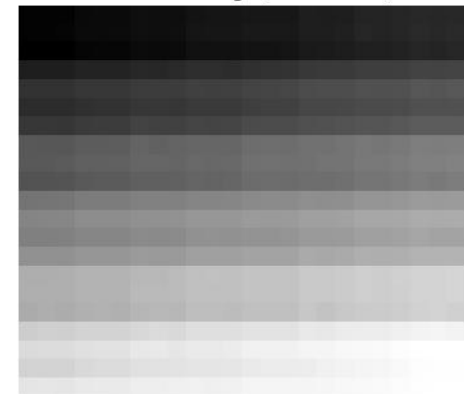
- **Simple Gray Image Example**

- Encoded Arithmetic Value: 0.005253950671640772320587473653071827917444939175
- Original Size (bits): 4608
- Arithmetic Encoded Size (bits): 3980
- Encoding Ratio 1 / 1.1578980223565543
- The image size was compressed.
- Encoding-decoding was lossless: True

Original Image



Decoded Image (Arithmetic)



Transform-Based Compression



- Why use transformations?
 - Energy compaction
 - Reduces pixel correlation
- Common Transforms:
 - DCT (used in JPEG)
 - Wavelet Transform (used in JPEG 2000)

Discrete Cosine Transform (DCT) for Image Compression



- Huffman and Arithmetic coding work by **encoding pixel values directly**
- Modern compression methods (like JPEG) **transform images into a frequency domain** first
- **◆ Why Use DCT?**
 - Images contain **low-frequency (smooth) and high-frequency (detailed) regions**
 - The **human eye is less sensitive to high frequencies**
 - **DCT converts an image into frequency components**, allowing us to **remove less important data**

Discrete Cosine Transform (DCT) for Image Compression



-  **High frequency and edges**

Feature Type	Contains High Frequencies?	Perceived Well?
Fine textures (e.g., skin pores, fabric details, distant leaves)	 Yes	 Weakly perceived
Noise (random high-frequency patterns)	 Yes	 Often ignored
Edges (outlines of objects, boundaries, strong contrast changes)	 Yes	 Strongly perceived

Discrete Cosine Transform (DCT) for Image Compression



-  **High frequency and edges**

Original Image



High-Pass Filter (Fine Details)



Canny Edge Detection (Structured Edges)



Discrete Cosine Transform (DCT) for Image Compression



- **◆ How JPEG Uses DCT for Compression**
 - **1 Divide image into 8×8 blocks** (since image details vary locally)
 - **2 Apply DCT to each block** (converts pixels → frequency representation)
 - **3 Quantize the DCT coefficients** (removes less important high-frequency values)
 - **4 Apply Huffman or Arithmetic coding to the quantized values**
- **◆ What This Means for Compression**
 - **More zeros after quantization = higher compression**
 - **Higher compression removes fine details** (causing artifacts in JPEG)

Discrete Cosine Transform



- Visualization of the DCT domain (DCT coefficients)



JPEG Compression

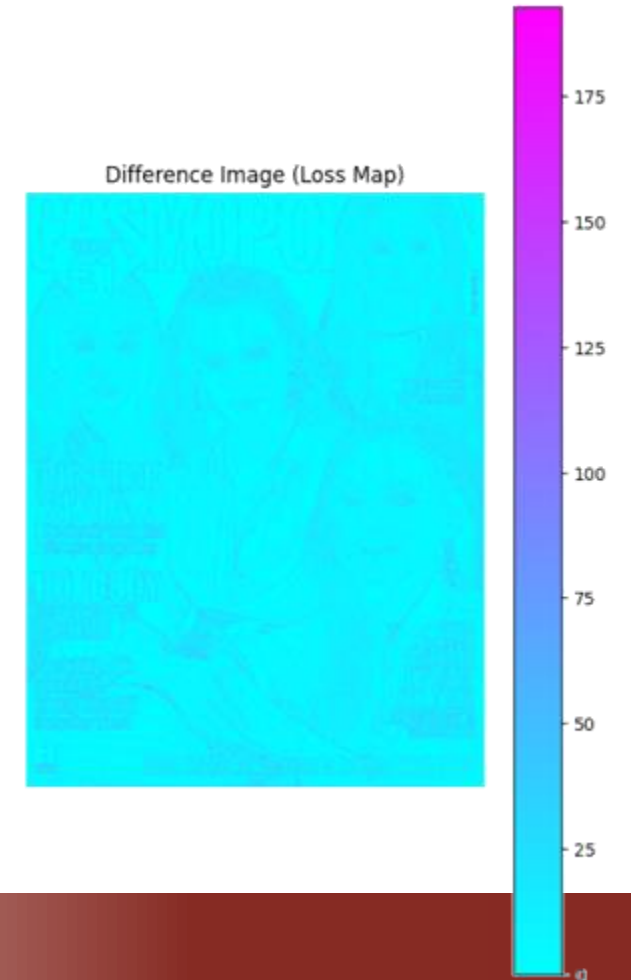


-  **How JPEG Compresses Images? DCT Quantization**
 - DCT transforms an image into **frequency components**, but to **achieve compression**, we need to **quantize** the DCT coefficients
 -  **How JPEG Uses Quantization**
 -  After DCT, most energy is in low frequencies (top-left corner)
 -  High-frequency coefficients (bottom-right) contain fine details
 -  JPEG quantizes these coefficients, meaning:
 - Small values become **zero**
 - Only the most important frequencies are kept
 - More zeroes = **higher compression**
 -  **Compression vs. Quality Trade-Off**
 - Stronger quantization = **higher compression**, but lower quality
 - Mild quantization = **better quality**, but larger file size

JPEG Compression



- A typical example



JPEG Compression



-  **How Low-Frequency Components Strongly Affect Image Quality**
 - JPEG removes **high-frequency details first** because the **human eye is less sensitive** to them
 - To test it:
 - Keep only low-frequency DCT coefficients
 - Remove high frequencies and reconstruct the image
 -  **Why Does This Work?**
 - Low frequencies store most of the image structure
 - High frequencies store fine details (edges, textures)
 - By keeping only low frequencies, we get a smooth, blurred version of the image

JPEG Compression



-  Zero out ALL high frequency DCT coefficients



JPEG Compression



-  **Comparing Different Low-Frequency Cutoffs in DCT**
 - We compare different cutoffs: 0.25, 0.1, and 0.05
 - Higher cutoff = More details kept (less compression)
 - Lower cutoff = More blurring (higher compression)



Why DCT in JPEG



-  **Summary: Why DCT is the Best Choice for Compression**
 -  **Energy compaction** – Most data is stored in a few coefficients
 - DC component (1st coefficient) stores the average brightness
 - DST for example lacks the DC component!
 -  **Matches human vision** – Removes details we don't perceive
 -  **Fast computation** – Efficient **Fast DCT (FDCT)** exists
 -  **Real-valued coefficients** – Unlike DFT, no complex numbers
 -  **Block-wise processing** – Works perfectly for **JPEG, MPEG, H.264** video compression
 -  **High compression ratios** – Preserves visual quality at high compression rates

Practical JPEG

-  **Step 1: Convert Image to YCbCr (JPEG Compression)**
 - JPEG compression **does not process RGB images directly**. Instead, it converts them into **YCbCr**, where:
 -  **Y (Luminance)** carries brightness (most important)
 -  **Cb, Cr (Chrominance)** carry color (compressed more)

Original Image



Y (Luminance)



Cb (Chrominance)



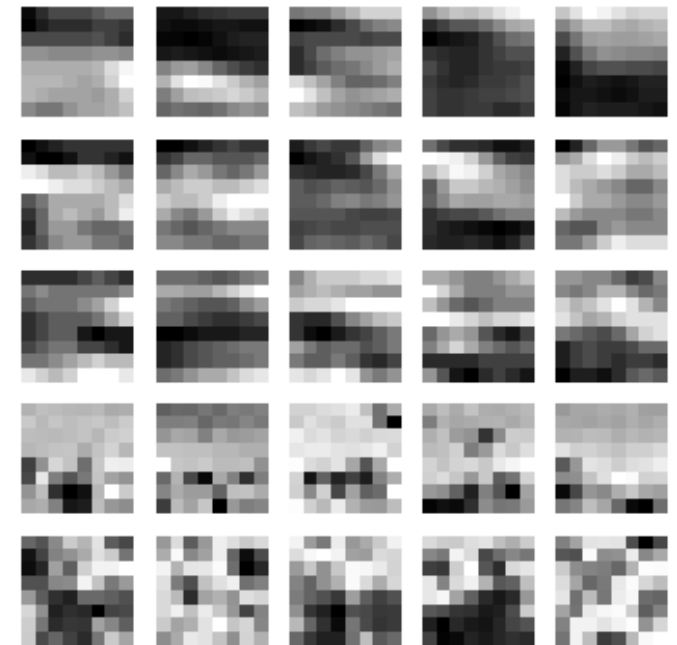
Cr (Chrominance)



Practical JPEG

-  **Step 2: Divide Image into 8×8 Blocks**
 - JPEG compresses **small 8×8 blocks separately**, not the entire image at once
 -  This allows **local compression adjustments**
 -  Ensures **better quality with fewer artifacts**
 -  Prepares the image for **DCT transformation**

Actual 8×8 Blocks from Image



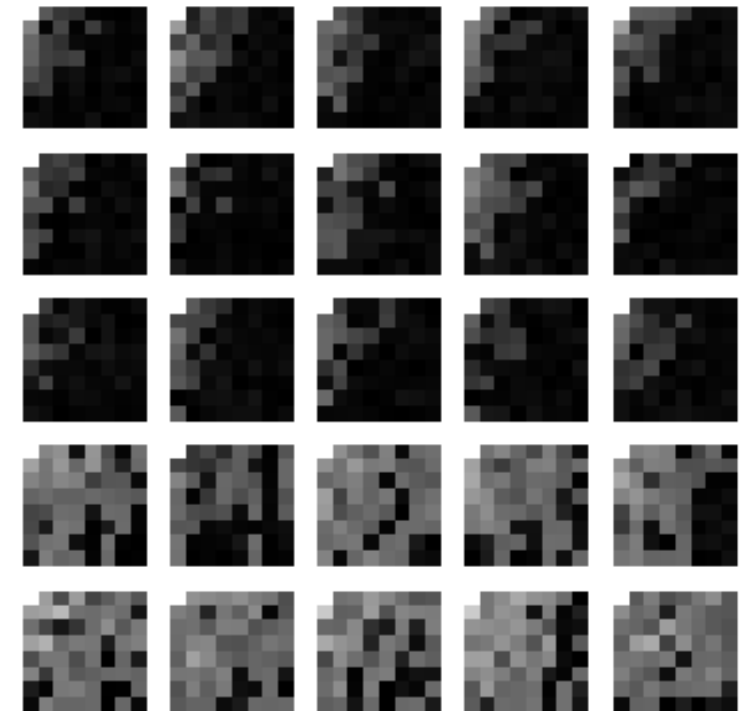
Practical JPEG



- **Step 3: Apply Discrete Cosine Transform (DCT)**

- Apply **DCT** to each 8×8 block
-  **DCT converts pixels to frequency components**
-  **Low frequencies store smooth regions**
-  **High frequencies store fine details**
 - (which can be removed for compression)

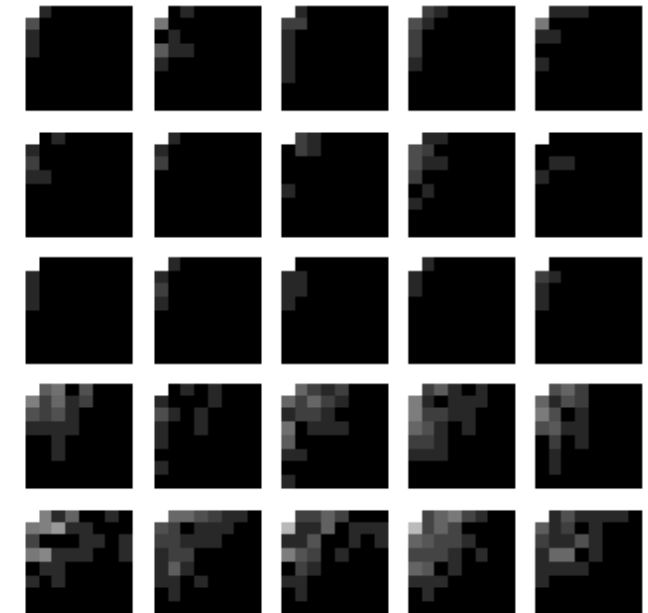
DCT Coefficients (Log Scale)



Practical JPEG

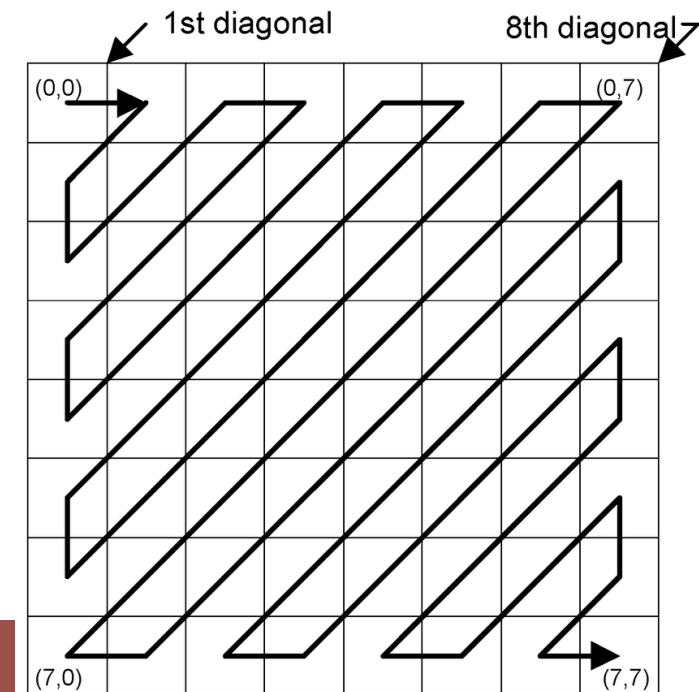
-  **Step 4: Apply Quantization to DCT Coefficients**
 - JPEG compresses the image by quantizing DCT coefficients
 -  Small values are set to zero to reduce data size
 -  More zeros = more compression
 -  Quality depends on how aggressively we quantize

Quantized DCT Coefficients (log-scaled)



Practical JPEG

-  **Step 5: Zig-Zag Scanning (Reordering DCT Coefficients)**
 - JPEG does not store DCT coefficients in the original order
 - Instead, it scans them in a **zig-zag pattern**, placing **low frequencies first** and **high frequencies last**
 -  **Why?**
 - This helps **Run-Length Encoding (RLE)** compress the data more effectively



- 

0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

```
[ (0, 85.0), (0, -1.0), (0, -3.0), (0, -1.0), (5, 1.0) ]
```

Practical JPEG

-  **Step 7: Huffman Encoding (Final JPEG Compression)**
 - After Run-Length Encoding (RLE), JPEG applies **Huffman Encoding** to store data **using the fewest bits possible**
 -  **Frequent values get shorter codes**
 -  **Less frequent values get longer codes**
 -  **This final step produces the compressed JPEG file!**
 -  **We compute Huffman codes from the actual quantized DCT coefficients**

Example Huffman Encoding (Using Real Image Data):

Symbol (0, -3.0): 00

Symbol (0, 85.0): 01

Symbol (5, 1.0): 10

Symbol (0, -1.0): 11

Practical JPEG

- **Huffman Bitstream for the Full Image**

-  Each block is individually encoded using Huffman codes
-  The final bitstream represents the compressed JPEG file

- Original Image Size: 854400 bits
- Compressed Image Size: 148459 bits
- Compression Ratio: 5.76
- Reduction: 82.62%

- Huffman Encoded Bitstream for Full Image (First 200 Bits):

- 0100101010101100111011111101101000111010110010000010010010010000
1001011100110110001000111011010101110010101100111100110000100100
0111110101111010111101110100011111111001100011001011000000110111
10101000 ...

Wavelet Transform and JPEG2000



- Unlike DCT, wavelets enable multi-resolution analysis
- **Advantages over DCT:** Better handling of sharp edges and less "blockiness"
- **Applications:** Medical imaging, high-quality archiving

Wavelet Transform and JPEG2000



-  **Why JPEG2000 Uses Wavelets Instead of DCT**
 - JPEG2000 replaces DCT with the Discrete Wavelet Transform (DWT)
 -  **Key Differences Between DCT and DWT**
 1. **DCT (Discrete Cosine Transform)**
 1. Works on **fixed-size 8×8 blocks**
 2. Compacts most of the image's energy into a **few low-frequency coefficients**
 3. Does **not preserve spatial structure well** (blocky artifacts at high compression)
 2. **DWT (Discrete Wavelet Transform)**
 1. Works on the **whole image**, not just small blocks
 2. Splits the image into **multiple frequency bands** at different resolutions
 3. Preserves edges better, **reducing blocking artifacts** seen in JPEG

Wavelet Transform and JPEG2000



- **Visualizing DCT vs. DWT**

- We take a small **8×8 patch** from an image and apply both **DCT** and **DWT** to compare:
 - **DCT Transform** → Shows how energy is compacted into frequency components
 - **DWT Transform** → Breaks the image into different scales (approximation & edge details)

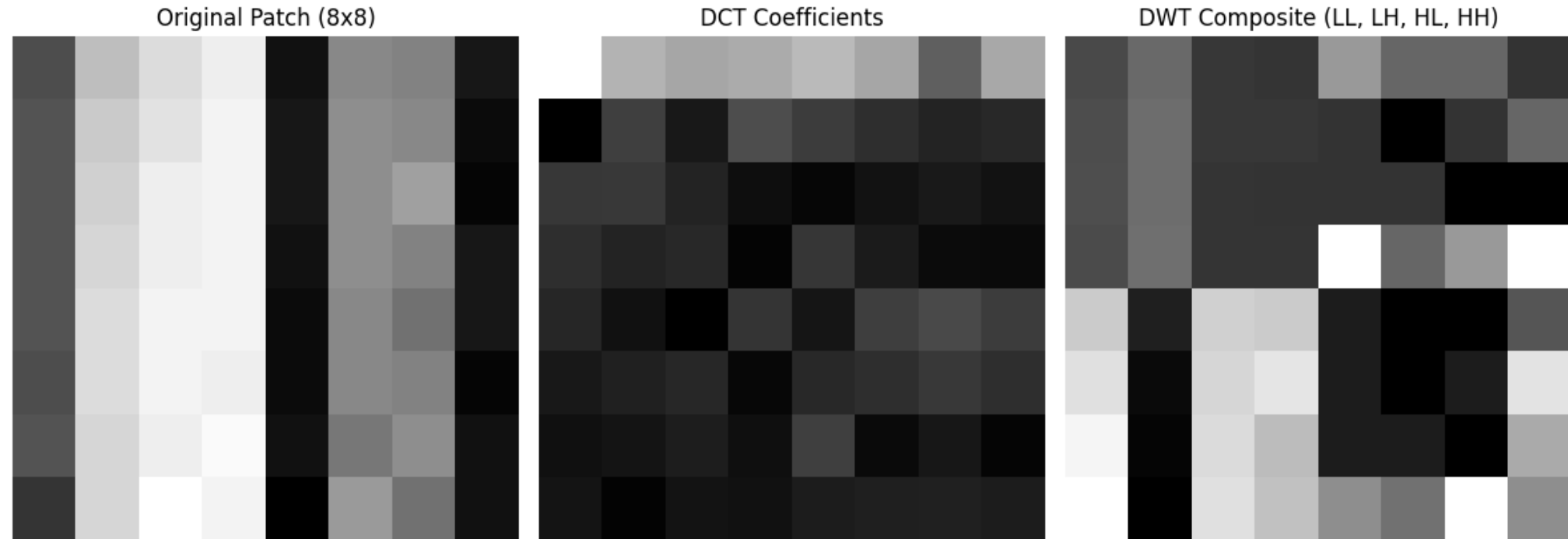
- **Expected Output**

- **Original 8×8 Patch:** A small part of the image
- **DCT Coefficients:** Frequency-domain representation using cosine transform
- **DWT Subbands**
 - **LL (Low-Frequency Approximation):** Blurred version of the image.
 - **LH (Horizontal Edges):** Highlights horizontal details.
 - **HL (Vertical Edges):** Highlights vertical structures.
 - **HH (Diagonal Edges):** Detects diagonal details.

Wavelet Transform and JPEG2000



- Visualizing DCT vs. DWT



Wavelet Transform and JPEG2000



-  Understanding JPEG2000's Wavelet Decomposition
 -  **What Do the Four DWT Subbands Represent?**
 - After applying **Discrete Wavelet Transform (DWT)**, we get four subbands:
 - **1 LL (Low-Frequency Approximation)**
 - Most important part of the image
 - Looks like a **blurred, downsampled version** of the image
 - **Used for low-resolution previews** in progressive compression
 - **2 LH (Horizontal Detail)**
 - Captures **horizontal edges**
 - Highlights **vertical structures** (walls, buildings, etc.)
 - **3 HL (Vertical Detail)**
 - Captures **vertical edges**
 - Highlights **horizontal structures** (text, landscapes, etc.)
 - **4 HH (Diagonal Detail)**
 - Captures **fine texture & diagonal details**

Wavelet Transform and JPEG2000



-  Understanding JPEG2000's Wavelet Decomposition
 -  **Why Multi-Level Decomposition?**
 - **Allows scalable image compression** (progressive transmission)
 - **Preserves important image structures** (edges and textures)
 - **Improves efficiency** by separating low- and high-frequency details
 -  **How Does It Work?**
 1. We first decompose the image into four subbands (LL, LH, HL, HH)
 2. Then, we take the LL component and decompose it further
 3. This process repeats for multiple levels, creating a hierarchical structure

Wavelet Transform and JPEG2000

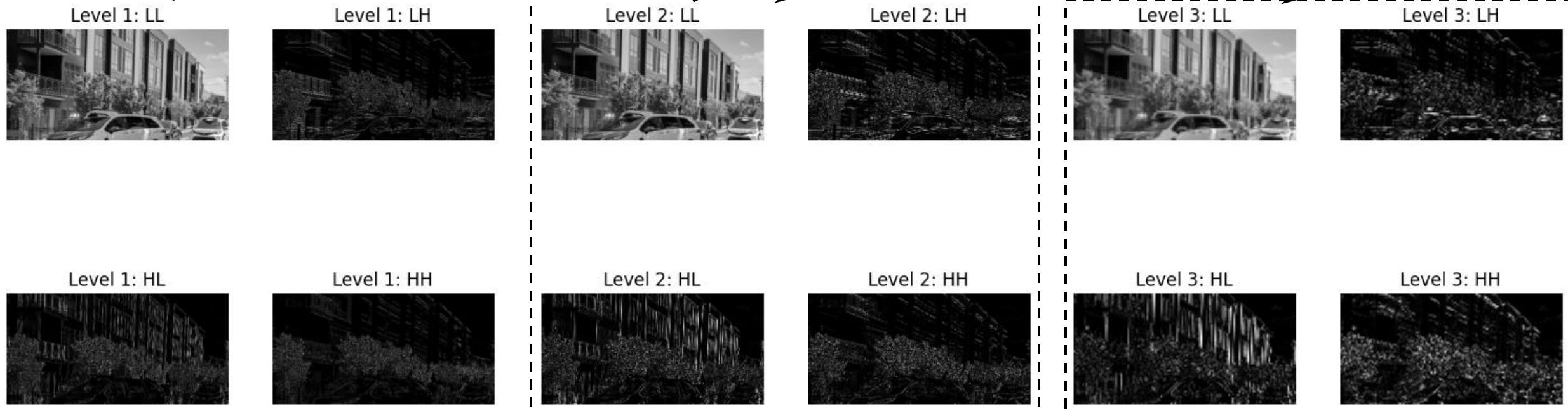


- **3-Level Decomposition Example**

Wavelet Decomposition - Level 1

Wavelet Decomposition - Level 2

Wavelet Decomposition - Level 3



Wavelet Transform and JPEG2000



-  **Simplified Quantization & Encoding in JPEG2000**
 - Now that we've seen **multi-level wavelet decomposition**, let's explore how **JPEG2000** quantizes and encodes these subbands
 - In the real standard, JPEG2000 uses a **highly sophisticated** coding scheme (EBCOT) with embedded bitplanes, rate-distortion optimization, and more
 - For our educational showcase, we'll use a **simplified approach**:
 1. **Wavelet Decomposition**: Break the image into subbands (LL, LH, HL, HH).
 2. **Uniform Quantization**: Apply a simple, uniform step size to each subband.
-  **Why Quantization in JPEG2000?**
 - **Main driver of lossy compression**: Larger quantization steps → fewer bits but more distortion
 - **Subband-based**: Different subbands (especially high-frequency) can be more aggressively quantized because the human eye is less sensitive to fine detail
 - **Rate control**: JPEG2000 can finely tune which subbands are quantized how much, achieving various bitrates

Wavelet Transform and JPEG2000



-  **A Simplified Example:** Single-Level DWT, Uniform step size, No entropy coding



Wavelet Transform and JPEG2000



- 🔍 Demonstrating Wavelet + Quantization + Huffman Encoding
 - Original Bits: 4194304 / Compressed Bits: 1315553
 - Compression Ratio: 3.19 / Reduction: 68.63%

Original (Cropped: 1024x512)



Reconstructed (q=30.0)



Modern AI-Based Compression



- Deep Learning-based compression
 - Variational Autoencoders (VAEs) -based methods
 - Generative Adversarial Networks (GANs) -based methods
 - Convolutional Neural Network (CNN)-based methods
- AI enhances perceptual quality over standard codecs
- AI-based **WebP** and **AVIF** outperform legacy formats like JPEG

Modern Approaches to Compression



- The case of the AUTOENCODERS
 - 📌 Step 1: Setup Environment & Load Dataset
 - 📌 Step 2: Define & Train the Autoencoder
 - 📌 Step 3: Evaluate & Visualize the Autoencoder's Compression

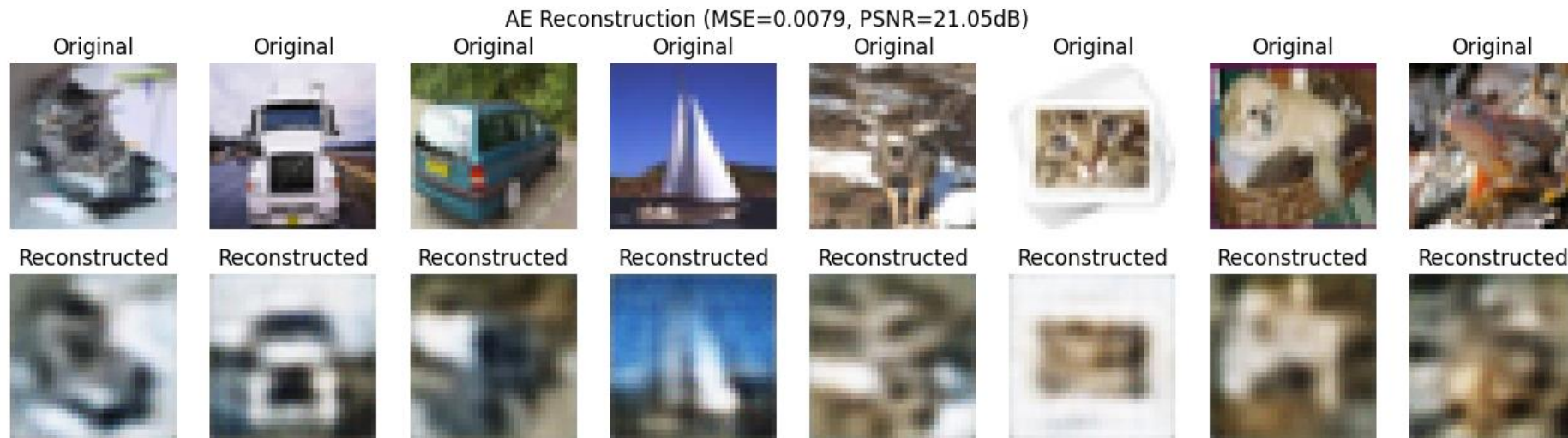


Image Transmission

-  Image Transmission & Error Simulation (for Wavelet-Based Compression)
 - In real-world scenarios, **transmitting compressed images** over unreliable channels (wireless, satellite, etc.) can introduce **packet loss** or **random bit errors**
 - We focus on how such errors affect the final reconstructed image in a **wavelet + quant + Huffman** setup
 -  **Two Types of Channel Errors**
 1. **Packet Loss**: the bitstream is split into packets (e.g., 8K bits each)
 - Randomly drop some packets entirely → missing chunk of subband data
 - Often results in a more catastrophic “block” of missing information
 2. **Bit Flip**: We flip bits within each packet (like random noise).
 - Even a few flips can desynchronize Huffman decoding, distorting large portions of the image or creating localized “salt-and-pepper” noise

Image Transmission



-  **Simplified Example**
 - Compress the image with wavelet+quant+Huffman → get clean_bitstream
 - Chunk the bitstream into packet_size=8000 bits
 - **DropPackets**: randomly pick 10% of packets, set them to " " (lost)
 - **FlipBits**: randomly flip 1% of total bits across all packets
 - Reassemble the corrupted streams
 - Decode each corrupted version → observe reconstruction
 -  **Note**: The final images might show:
 - **Packet drop**: Big missing chunk(s) in subband data → large gray patches or distortion
 - **Bit flip**: Huffman might desync, producing scattered noise or partial decode

Image Transmission



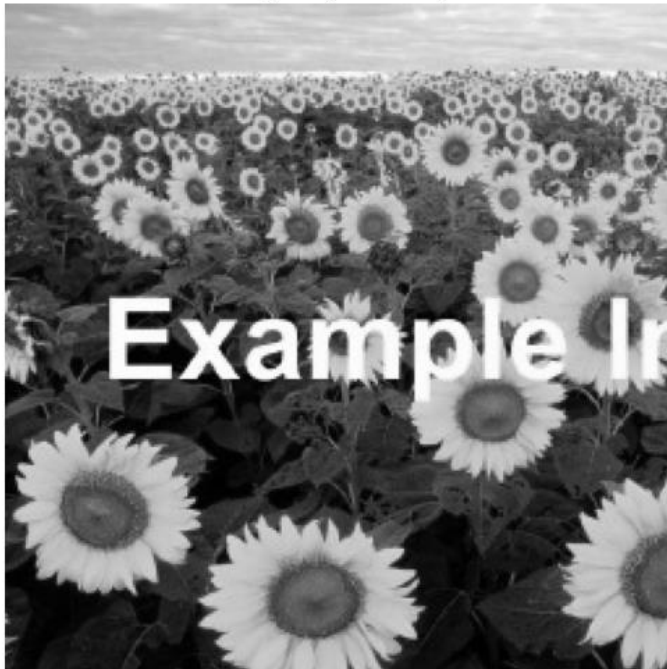
- **Simplified Example**

- Packet dropping MSE: 5360.77
- Packet dropping PSNR: 21.68 dB

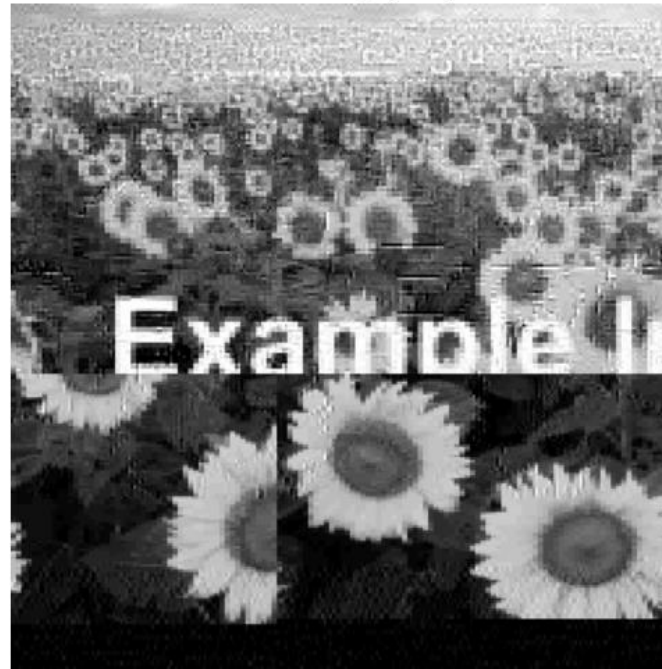
Bit flipping MSE: 11423.89

Bit flipping PSNR: 15.11 dB

Original (256x256)



Packet Drop (10%)



Bit Flip (1%)



Image Transmission



-  A Naive Error-Resilience Strategy: Multiple Huffman Blocks
 - **Problem:**
 - With a single Huffman-coded bitstream, **any** bit flip can break the entire decode
 - **Naive Solution:**
 - **Partition** the wavelet-quantized data into N separate chunks (e.g., 4 or 8)
 - **Encode** each chunk with its **own Huffman tree** (or at least its own code block)
 - **Store** each chunk's bit length + bitstream back-to-back
 - **During transmission:**
 - If **bit flips** occur in chunk i , only that chunk is messed up
 - The decoder can still **resynchronize** at the start of chunk $i+1$
 - **Trade-Offs:**
 - We pay some overhead for each chunk's header info + separate Huffman trees
 - Real JPEG2000 has more advanced local error resilience, but **this** is a simple demonstration that flipping bits in chunk 0 doesn't ruin chunk 1+

Image Transmission

-  A Naive Error-Resilience Strategy: Multiple Huffman Blocks
 - Bit flipping MSE: 6770.75
 - Bit flipping PSNR: 19.65 dB

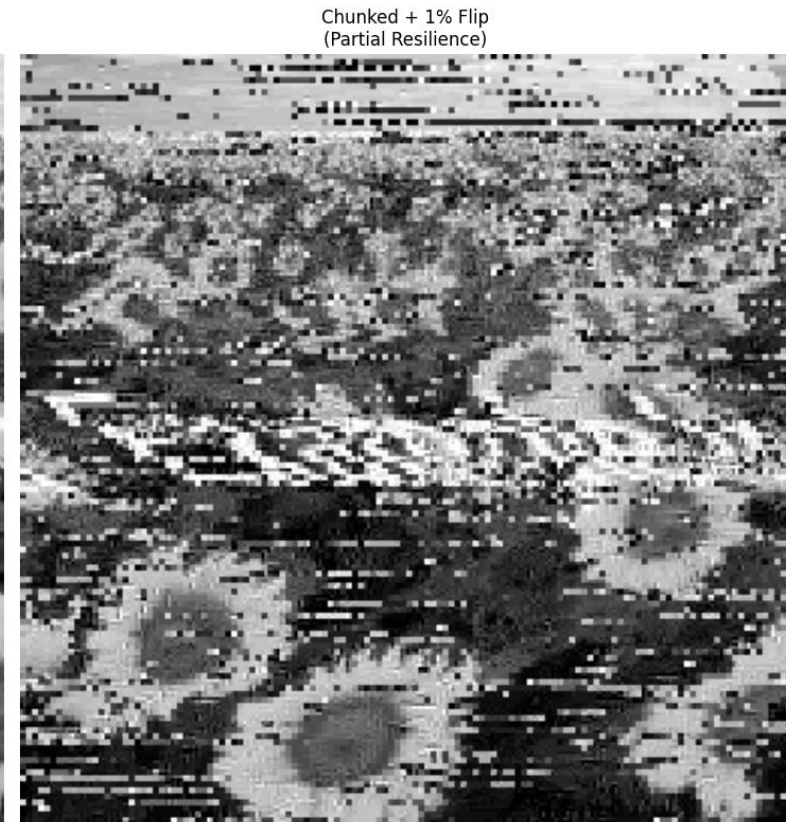


Image Transmission

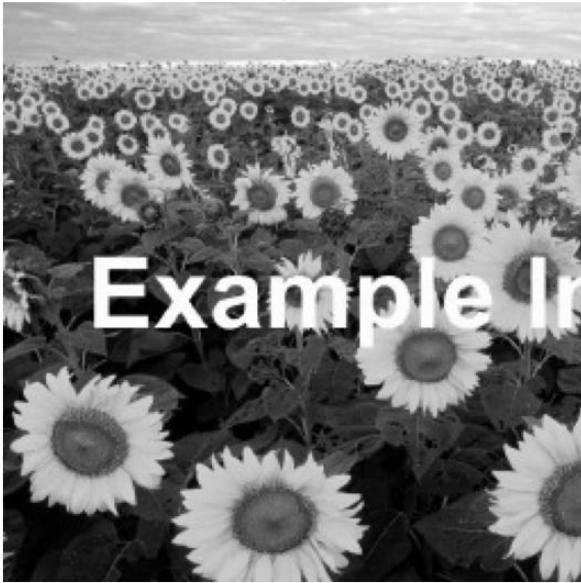


-  Naive Packet Dropping with Chunk-Based Resilience
 - We already saw that **dividing wavelet subband data** into multiple chunks (each chunk Huffman-coded separately) can localize corruption
 - Instead of **bit flips**, we now simulate **packet (chunk) dropping**:
 1. **Partition** the wavelet-quantized subbands into N chunks (subband-level or block-level)
 2. **Huffman-encode** each chunk separately, each chunk with its own code
 3. During transmission, **randomly drop** some percentage of chunks (e.g. 10%)
 4. At decode time, any dropped chunk = lost subband data → we fill that subband with zeros, while other chunks decode fine
 - **Why This Demonstration?**
 - In real JPEG2000, code streams are carefully partitioned into error-resilient segments
 - Dropping entire segments might lose part of a subband but preserve the rest
 - We see that **some** parts of the image remain correct, instead of the entire image turning to random noise (as with a single big stream and no markers)

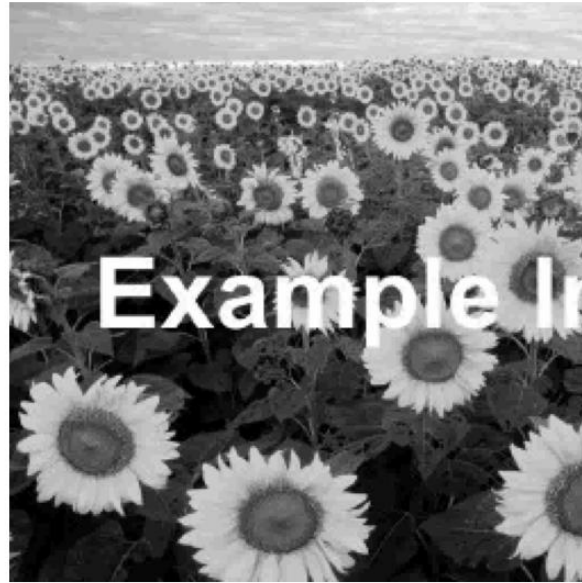
Image Transmission

- 📌 Naive Packet Dropping with Chunk-Based Resilience
 - Drop percentage 30%

Original

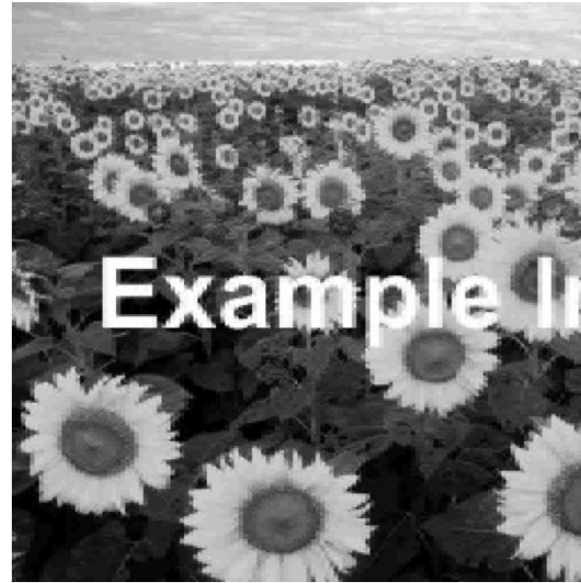


After Packet Drop



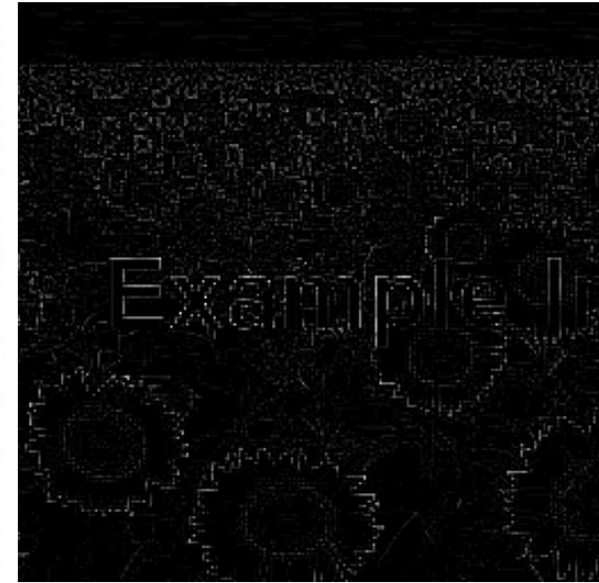
MSE: 16.33
PSNR: 36.00 dB

After Packet Drop



MSE: 149.16
PSNR: 26.39 dB

After Packet Drop



MSE: 18215.24
PSNR: 5.53 dB

Emerging Formats: AVIF & WebP



- AVIF (AV1-based) and WebP outperform JPEG in compression ratio and quality
- Supported by modern browsers and image applications

Conclusion & Next Steps

- Image Coding and Compression
 - Entropy-based, transform-based methods, modern AI approaches
- Future trends in compression
 - AI-driven encoding
 - Improved perceptual quality

Bibliography

