Huffman coding

**Huffman coding** is one of the most popular techniques for removing coding redundancy. It yields the smallest possible number of code symbols per a source symbol.

The first step is to create a series of source reductions by ordering the probabilities of the symbols under consideration and combining the lowest probability symbols into a single symbol that replaces them in the next source reduction.

The table below shows an example of Huffman coding for a set of symbols with their respective probabilities.

```
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Probability</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_2</td>
<td>0.4</td>
<td>0.4</td>
<td>0.4</td>
<td>0.4</td>
<td>0.6</td>
</tr>
<tr>
<td>a_3</td>
<td>0.3</td>
<td>0.3</td>
<td>0.3</td>
<td>0.3</td>
<td>0.4</td>
</tr>
<tr>
<td>a_1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.2</td>
<td>0.3</td>
<td></td>
</tr>
<tr>
<td>a_4</td>
<td>0.1</td>
<td></td>
<td>0.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a_5</td>
<td>0.06</td>
<td></td>
<td></td>
<td></td>
<td>0.4</td>
</tr>
<tr>
<td>a_6</td>
<td>0.04</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

*K-ary Huffman coding*
Huffman coding

A set of symbols and their probabilities are ordered from top to bottom in terms of decreasing probability values. To form the first source prediction, the bottom two probabilities (0.04 and 0.06) are combined to form a “compound symbol” with probability of 0.1. This compound symbol and its associated probability are placed in the first source prediction column so that probabilities of the reduced source also are ordered from the most to the least probable. This process is repeated until a reduced source with two symbols is reached.

The second step is to code each reduced source, starting with the smallest source and working back to the original source.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Probability</th>
<th>Code</th>
<th>Source reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_2$</td>
<td>0.4</td>
<td>1</td>
<td>0.4 1</td>
</tr>
<tr>
<td>$a_6$</td>
<td>0.3</td>
<td>00</td>
<td>0.3 00</td>
</tr>
<tr>
<td>$a_3$</td>
<td>0.1</td>
<td>01</td>
<td>0.1 01</td>
</tr>
<tr>
<td>$a_5$</td>
<td>0.06</td>
<td>0100</td>
<td>0.1 0100</td>
</tr>
<tr>
<td>$a_4$</td>
<td>0.04</td>
<td>0101</td>
<td>0.1 0101</td>
</tr>
</tbody>
</table>

The minimum length binary code for a two-symbol source are the symbols 0 and 1. These code symbols are assigned to the two symbols on the right.
Huffman coding

We observe that the assignment of “1” and “0” is arbitrary and can be reversed without any harm. Since the reduced source symbol with probability 0.6 was generated by combining two symbols in the reduced source to its left, the “0” used to code it, is now assigned to both of these symbols, and “0” and “1” are arbitrarily appended to each to distinguish them from each other. This procedure is repeated for each reduced source until the original source is reached. The final code appears as shown. The average code length is:

\[ L_{\text{avg}} = 0.4 \cdot 1 + 0.3 \cdot 2 + 0.1 \cdot 3 + 0.1 \cdot 4 + 0.06 \cdot 5 + 0.04 \cdot 5 = 2.2 \text{ bits / pixel} \]

The entropy of the source is 2.14 bits/symbol.

Huffman coding

Huffman’s procedure creates the optimal code for a set of symbols and probabilities where each symbol is coded one at a time. Once the code has been created, coding and/or error-free decoding is accomplished by a lookup table manner. The code itself is an instantaneous uniquely decodable block code.

**Block code** – each source symbol is mapped into a fixed sequence of code symbols.

**Instantaneous code** – each code word in a string of code symbols can be decoded independently from neighboring symbols.

Thus, any string of Huffman encoded symbols can be decoded by examining the individual symbols of the string from left to right.
Huffman coding

Assuming that the encoded string 010100111100 was generated by the code developed previously. Examining the string, we notice that the first valid code word is 01010, which is the code for symbol \( a_3 \). The next valid code is 011, which is \( a_1 \), etc.

When a large number of symbols needs to be encoded, the design of an optimal Huffman code is a nontrivial task. For the general case of \( J \) source symbols, \( J \) symbol probabilities, \( J-2 \) source reductions, and \( J-2 \) code assignments are needed. When source symbol probabilities can be estimated in advance, “near optimal” coding can be achieved with pre-computed Huffman codes. This approach – specification default Huffman coding tables – is used in JPEG and MPEG standards.

Huffman coding

A histogram of an image is not uniform suggesting that compression is possible. The Huffman’s procedure with 7.428 bits/pixel to code 8 bits pixels was used. The entropy in the image was estimated as 7.3838 bits/pixel. The resulting compression ratio is \( C = 1.077 \) and the relative redundancy \( R = 0.0715 \). Therefore, 7.15% of the original 8-bit fixed-length intensity representation was removed as coding redundancy.
Golomb coding

Considering coding of nonnegative integer inputs with exponentially decaying probabilities, it is possible to show that this type of inputs can be optimally encoded using a family of codes computationally more efficient than Huffman codes. We will denote as $\lfloor x \rfloor$ the largest integer less than or equal to $x$ and as $\lceil x \rceil$ the smallest integer greater than or equal to $x$, and as “$x \mod y$” the remainder of $x$ divided by $y$.

For a nonnegative integer $n$ and a positive integer divisor $m > 0$, the Golomb code of $n$ with respect to $m$, denoted as $G_m(n)$, is a combination of the unary code of $\lfloor n/m \rfloor$ and the binary representation of $n \mod m$. $G_m(n)$ is constructed as:

Step 1: form the unary code of quotient $\lfloor n/m \rfloor$. The unary code of an integer $q$ is defined as $q$ “1” followed by a “0”.

Step 2: let $k = \lfloor \log_2 m \rfloor$, $c = 2^k - m$, $r = n \mod m$, and compute truncated remainder $r'$ such that

$$r' = \begin{cases} r \text{ truncated to } k-1 \text{ bits} & 0 \leq r \leq c \\ r + c \text{ truncated to } k \text{ bits} & \text{otherwise} \end{cases}$$

Step 3: concatenate the results of steps 1 and 2.

To compute $G_4(9)$, for instance, begin by determining the unary code of the quotient $\lfloor 9/4 \rfloor = \lfloor 2.25 \rfloor = 2$, which is 110 (result of step 1). Then let $k = \lfloor \log_2 4 \rfloor = 2$, $c = 2^2 - 4 = 0$, $r = 9 \mod 4$, which in binary is $1001 \mod 0100 = 0001$. Then $r' = r = (0001)$ truncated to 2 bits, which is 01. Finally, concatenate 110 and 01 to get 1101, which is $G_4(9)$.
Golomb coding

For the special case of \( m = 2^k \), \( c = 0 \) and \( r' = r = n \mod m \) truncated to \( k \) bits for all \( n \), the divisions needed to generate the resulting Golomb codes become binary shift operations resulting in the computationally simpler code called Golomb-Rice or Rice code.

Golomb codes \( G_1, G_2, \) and \( G_4 \) are also Golomb-Rice codes. \( G_1 \) is also the unary code.

<table>
<thead>
<tr>
<th>( n )</th>
<th>( G_1(n) )</th>
<th>( G_2(n) )</th>
<th>( G_4(n) )</th>
<th>( G_{\text{exp}}(n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
<td>000</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>01</td>
<td>010</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>110</td>
<td>100</td>
<td>101</td>
<td>1011</td>
</tr>
<tr>
<td>3</td>
<td>1110</td>
<td>110</td>
<td>1101</td>
<td>11000</td>
</tr>
<tr>
<td>4</td>
<td>11110</td>
<td>1110</td>
<td>11001</td>
<td>110010</td>
</tr>
<tr>
<td>5</td>
<td>111110</td>
<td>11110</td>
<td>110011</td>
<td>1110000</td>
</tr>
<tr>
<td>6</td>
<td>1111110</td>
<td>111110</td>
<td>1100111</td>
<td>11100001</td>
</tr>
<tr>
<td>7</td>
<td>11111110</td>
<td>1111110</td>
<td>11001111</td>
<td>111000001</td>
</tr>
<tr>
<td>8</td>
<td>111111110</td>
<td>11111110</td>
<td>110011111</td>
<td>1110000001</td>
</tr>
</tbody>
</table>

Golomb coding

Golomb codes can only be used for representing nonnegative integers. There are many possible Golomb codes. It is important to properly select a divisor \( m \) for the effective application. When the integers are geometrically distributed with probability mass function in the form:

\[
P(n) = (1 - \rho) \rho^n
\]

for some \( 0 < \rho < 1 \), Golomb codes are optimal (give the shortest average code length), when

\[
m = \left\lfloor \frac{\log_2(1 + \rho)}{\log_2(1/\rho)} \right\rfloor
\]
Golomb coding

Probabilities of image intensities are quite unlikely to match the geometrical distribution, therefore, Golomb codes are NOT often used to code intensities. However, the **intensity differences** usually can be approximated as geometrically distributed.

Golomb coding

The only problem with intensity differences is that some of them will be negative… To handle negative differences mappings are used similar to

\[
M(n) = \begin{cases} 
2n & n \geq 0 \\
2\lfloor|n|\rfloor - 1 & n < 0
\end{cases}
\]

Therefore, modified (mapped) intensity differences can be efficiently encoded using an appropriate Golomb-Rice code.

Golomb codes are used in JPEG-LS and AVS compression standards.
Arithmetic coding

The arithmetic codes generate non-block codes; therefore, a one-to-one correspondence between source symbols and code words does not exist. Instead, an entire sequence of source symbols is assigned to a single code word that defines an interval of real numbers between 0 and 1.

As the number of symbols in the message increases, the interval used to represent it becomes smaller and the number of bits needed to represent the interval becomes larger. Each symbol in the message reduces the size of the interval according to its probability of occurrence. Since the symbols are not coded one at a time, this technique can (theoretically) achieve the highest possible coding efficiency.

Consider a five-symbol sequence $a_1a_2a_3a_4a_5$ from a four-symbol source.

Initially, the message is assumed to occupy the entire interval $[0,1)$. The interval is initially subdivided into four regions based on the probabilities of each source symbol ($a_1$ is in $[0, 0.2)$.

Since the first symbol is $a_1$, the message interval narrows to $[0, 0.2)$. 

<table>
<thead>
<tr>
<th>Source Symbol</th>
<th>Probability</th>
<th>Initial Subinterval</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_1$</td>
<td>0.2</td>
<td>[0.0, 0.2)</td>
</tr>
<tr>
<td>$a_2$</td>
<td>0.2</td>
<td>[0.2, 0.4)</td>
</tr>
<tr>
<td>$a_3$</td>
<td>0.4</td>
<td>[0.4, 0.8)</td>
</tr>
<tr>
<td>$a_4$</td>
<td>0.2</td>
<td>[0.8, 1.0)</td>
</tr>
</tbody>
</table>
Arithmetic coding

Therefore, [0,0.2) is expanded to the full height and its end points are labeled by the values of the “narrowed range” (0, 0.2). The “narrowed range” is subdivided according to the original source symbol probabilities and the process continues: $a_2$ narrows the subinterval to [0.04,0.08), etc. The final message symbol (that must be reserved as a special end-of-message indicator) narrows the range to [0.06752,0.0688). Any number from the final subinterval (for instance, 0.068) can be used to represent this message.

Three decimal digits are used to represent the five-symbol message, which is 0.6 digits per symbol. The theoretical limit is 0.58 digits per symbol. As the length of the coded sequence increases, the resulting arithmetic code approaches the theoretical minimum number of bits. In practice, (1) adding the end-of-message and (2) use of finite precision arithmetic reduce efficiency of arithmetic codes.

Arithmetic coding

Adaptive context dependent probability estimates.

With accurate input symbol probability models (models providing the true probabilities of the coded symbols), arithmetic codes are near optimal in the sense of minimizing the average number of code symbols per a source symbol (similar to Huffman and Golomb codes). A simple way to improve the accuracy of the probability estimates is to use an adaptive, context dependent probability model. Such models update symbol probabilities as symbols are coded or become known. Therefore, probabilities adapt to the local statistics of the coded symbols. Context dependent models provide probabilities that are based on a predefined neighborhood of pixels (context) around the coded symbols. Normally, a causal context (limited to symbols that have already been coded) is used.
Arithmetic coding

Both the Q-coder and MQ-coder (techniques used in JBIG, JPEG-2000) use probability models that are adaptive and context dependent. The Q-coder dynamically updates symbol probabilities during the interval renormalizations that are part of the arithmetic coding.

Adaptive, context-based arithmetic coding approach

3 possible context models: immediate preceding symbol, a group of prec. symbols, some prec. symbols and symbols on the previous scan line.

Arithmetic coding

Arithmetic coding is often used when for coding binary symbols. As each symbol (bit) begins the coding process, its context is formed in the context determination block. The probability estimation block must manage (for the cases shown) $2^2$, $2^8$, and $2^5$ contexts and their associated probabilities. For instance, for the 8-symbol context shown, conditional probabilities $P(0|a=0)$, $P(1|a=0)$, $P(0|a=1)$, and $P(1|a=1)$ must be tracked. These probabilities are passed to the arithmetic coding block (as functions of the current context) and control the generation of the arithmetically coded output sequence. When another symbol – within the context involved in the current coding step – has been processed, the probabilities associated with that context are updated.

Arithmetic coding is implemented in JBIG1, JBIG2, JPEG-2000, H.264, MPEG-4 AVC…
LZW coding

So far we were focused on removing coding redundancy. Let us focus next on error-free removal spatial redundancy. The *Lempel-Ziv-Welch (LZW)* coding assigns fixed-length code words to variable-length sequences of coding symbols without a priori knowledge of the symbols probabilities.

At the onset of the coding process, a *dictionary* (codebook) containing the source symbols to be coded is constructed. For 8-bit monochrome images, the first 256 words of the dictionary are assigned to intensities 0,1,…255. As the encoder sequentially examines image pixels, intensity sequences that are not in the dictionary are placed in algorithmically determined locations. For instance, if the first two pixels are white, sequence “255-255” might be assigned to 256. Next time – when two consecutive white pixels are encountered – code word 256 will be used to represent them.

LZW coding

If a 9-bit, 512-word dictionary is used for coding, the original 8+8 bits that were used to represent the two pixels will be replaced by a single 9-bit code word. The size of the dictionary is an important system parameter: if it is too small, the detection of matching intensity-level sequences will be less likely; if it is too large, the size of the code words will decrease compression performance.

Consider this 4x4 8-bit vertical edge image:

<table>
<thead>
<tr>
<th>Dictionary Location</th>
<th>Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>255</td>
<td>255</td>
</tr>
<tr>
<td>256</td>
<td>–</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>511</td>
<td>–</td>
</tr>
</tbody>
</table>

A 512 word dictionary with the following starting content where locations 256 through 511 initially are unused.
LZW coding

The image is encoded by processing its pixels in a left-to-right, top-to-bottom manner. Each successive intensity value is concatenated with a variable “currently recognized sequence”, which is empty initially. The dictionary is searched for each concatenated sequence and if found is replaced by the newly concatenated and recognized (located in the dictionary) sequence.

<table>
<thead>
<tr>
<th>Currently Recognized Sequence</th>
<th>Pixel Being Processed</th>
<th>Encoded Output</th>
<th>Dictionary Location (Code Word)</th>
<th>Dictionary Entry</th>
</tr>
</thead>
</table>

LZW coding

At the end, the dictionary contains 265 code words and the LZW algorithm has identified several repeating intensity sequences and has reduced the original 128-bit image to 90 bits (10 9-bit codes). The compression ratio is 1.42:1. the encoded output is obtained from reading the third column...

A unique feature of the LZW coding is that the code dictionary is created while the data is encoded.

The LZW coding is used in GIF, TIFF, and PDF formats.
Run-length coding

Images with repeating intensities along their rows or columns can often be compressed by representing runs of identical intensities as run-length pairs, where each run-length pair specifies the start of a new intensity and the number of consecutive pixels having that intensity. This technique is used for FAX data compression, in BMP file format, etc.

RLE is particularly effective when compressing binary images since there are only two possible intensities (black and white). Additionally, a variable-length coding can be applied to the run lengths themselves. The approximate run-length entropy is

$$H_{RL} = \frac{H_0 + H_1}{L_0 + L_1}$$

where $H_0$ and $H_1$ are entropies of the black and white runs and $L_0$ and $L_1$ are the average values of black and white run lengths.

Run-length coding

Two oldest image compression standards developed for FAX are CCIT Group 3 and 4 standards for binary image compression.

1D CCITT compression

Each line in the image is encoded as a series of variable-length Huffman code words representing the run lengths of alternating white and black runs in a left-to-right scan of the line. The compression method is often called a modified Huffman (MH) coding. The code words are of two types: terminating codes and makeup codes.
Run-length coding

2D CCITT compression

The 2D compression method used by CCITT Group 3 and 4 standards is a line-by-line method where the position of each black-to-white or white-to-black run transition is coded with respect to the position of a reference element $a_0$ that is situated on the current coding line. The previously coded line is called the reference line; the reference line for the first line of each image is an imaginary white line. The 2D coding technique is called Relative Element Address Designate (READ) coding.

The coding is 2D in a sense that information from the previous line is used to encode the current line.

Run-length coding

A scan of a typical textbook page: about half is the text, around 9% is occupied by a halftone image, and the rest is white space. An illusion of gray tones is created in a binary image…

$$r N = 4, T_1$$
Run-length coding

In the basic 2D coding process, $|a_i b_i|$ denotes the absolute value of the distance between changing elements $a_i$ and $b_i$.

Run-length coding is implemented in CCITT, JBIG2, JPEG, M-JPEG, MPEG-1,2,4, BMP standards.

Symbol-based coding

In symbol- or token-based coding, an image is represented as a collection of frequently occurring sub-images, called symbols. Each symbol is stored in a symbol dictionary and the image is coded as a set of triplets $\{(x_1, y_1, t_1), (x_2, y_2, t_2), \ldots\}$, where each $(x_i, y_i)$ pair specifies the location of a symbol in the image and a token $t_i$ is the address of the symbol (sub-image) in the dictionary. Therefore, each triplet represents an instance of a dictionary symbol in the image.

Storing repeated symbols only once can compress image significantly (especially, for text document storage) if symbols are repeating.
Symbol-based coding

Consider a simple bilevel image consisting of a word “banana” that is composed of three symbols: “b” (1), “a” (3), and “n” (2). Since “b” is the first symbol identified in the coding process, its 9x7 bitmap is stored in location 0 of the symbol dictionary. The token identifying the bitmap “b” is 0. Therefore, the first triplet in the image is (0,2,0) indicating that the upper left corner of the bitmap specified by the token 0 should be placed at location (0,2) in the image. After the bitmaps for “a” and “n” images are identified and placed to the dictionary, the image can be encoded with 5 triplets.

Symbol-based coding

If the six triplets required to locate the symbols in the image together with the three bitmaps needed to define them are smaller than the original image, compression occurs. In our case, the original is 9x51x1 = 459 bits; assuming that each triplet requires 3 bytes, the compressed representation is 6x3x8 + 9x7+6x7+6x6 = 285 bits. C = 1.61.

Like many other compression methods, symbol-based decoding is significantly faster than encoding. The symbol bitmaps and the triplets can be further compressed.

Finally, if only exact symbol matches are allowed, the compression will be lossless.

Symbol-based coding is implemented in JBIG2 compression.
Symbol-based coding

JBIG2 compression.

JBIG2 is a standard for bilevel image compression that segments an image into overlapping and (or) non-overlapping regions of text, halftone, and generic content.

Text regions are composed of characters; each of them is represented by a character bitmap (subimage). Normally, there is one character bitmap in the symbol dictionary for each upper- or lowercase font character. In lossy JBIG2, differences between dictionary bitmaps and characters are neglected. In lossless JBIG2, such differences are stored and used with the triplets. All bitmaps are encoded either arithmetically, or with run-length codes; the triplets are encoded either arithmetically, or by Huffman codes.

Symbol-based coding

Halftone regions are similar to text regions in a sense that they are composed of patterns arranged in a regular grid. However, the symbols stored in the dictionary are periodic patterns representing intensities (of a photograph) that have been dithered to produce bilevel images.

Generic regions contain no-text, non-halftone information. They are compressed using either arithmetic or run-length coding.

The reconstruction after lossless JBIG2, reconstruction after lossy JBIG2, and the difference between them are described as follows:

\[
\text{Lossless:} \quad C = 20.3 \quad \text{Lossy:} \quad C = 27.7
\]
Bit-plane coding

The bit-plane coding is based on decomposing a multilevel (monochrome or color) image into a series of binary images and compressing each binary image by one of several well known binary compression methods.

The intensities of an $m$-bit monochrome image can be represented in the form of base-2 polynomial:

$$a_{m-1}2^{m-1} + a_{m-2}2^{m-2} + \ldots + a_12^1 + a_02^0$$

Therefore, to decompose an image into a set of binary images, we need to separate $m$ coefficients of the polynomial into $m$ 1-bit planes. The lowest order bit-plane (corresponds to the least significant bit) is generated by $a_0$ bits of each pixel, while the highest order bit-plane contains the $a_{m-1}$ bits.

Bit-plane coding

However, this approach leads to the situation, when small changes of intensity can have significant impact on bit-planes. For instance, if a pixel intensity 127 (01111111) is adjacent to a pixel intensity 128 (10000000), every bit will contain a corresponding 0 to 1 (or 1 to 0) transition.

Alternatively, an image can be represented first by an $m$-bit Gray code. This code $g_{m-1} \cdots g_2 g_1 g_0$ corresponding to the polynomial is computed as

$$g_i = a_i \oplus a_{i+1} \quad 0 \leq i \leq m - 2$$

$$g_{m-1} = a_{m-1}$$

where $\oplus$ denotes the OR operation. The property of this code is that the successive code words differ in only one bit position and small changes in intensity are less likely to affect all $m$ bit-planes.
Bit-plane coding

We notice that the high order bit planes are far less complex than the low-order ones. Also, Gray-coded bit-planes are less complex than the corresponding bit-planes.

<table>
<thead>
<tr>
<th>Coefficient</th>
<th>Binary Code (PDF bits)</th>
<th>Gray Code (PDF bits)</th>
<th>Compression Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>6,999</td>
<td>6,999</td>
<td>1.00</td>
</tr>
<tr>
<td>6</td>
<td>12,791</td>
<td>11,024</td>
<td>1.16</td>
</tr>
<tr>
<td>5</td>
<td>40,104</td>
<td>36,914</td>
<td>1.00</td>
</tr>
<tr>
<td>4</td>
<td>55,911</td>
<td>47,415</td>
<td>1.18</td>
</tr>
<tr>
<td>3</td>
<td>78,915</td>
<td>67,787</td>
<td>1.16</td>
</tr>
<tr>
<td>2</td>
<td>101,535</td>
<td>92,630</td>
<td>1.10</td>
</tr>
<tr>
<td>1</td>
<td>107,909</td>
<td>105,286</td>
<td>1.03</td>
</tr>
<tr>
<td>0</td>
<td>99,753</td>
<td>107,909</td>
<td>0.92</td>
</tr>
</tbody>
</table>

The two least significant bits have little apparent structure. Since this is typical for most 8-bit monochrome images, bit-plane coding is usually restricted to 6-bit images.

Bit-plane coding is implemented in JBIG1 and JPEG-2000 standards.
Block transform coding

The block transform coding is a reversible, linear transform that maps each non-overlapping block (sub-image) of a fixed size into a set of transform coefficients, which are then quantized and coded. For most images, a significant number of the coefficients have small magnitude and can be coarsely quantized (or entirely discarded) with little image distortion. A variety of transforms (DFT, DCT) can be used.

The goal of transformation is to decorrelate the pixels in each block or to pack as much information as possible into the smallest number of transform coefficients.

Transform selection

The choice of particular 2D transform for a given application depends on the amount of reconstruction error that can be tolerated and computational resources available. Compression is achieved during the quantization of the transformed coefficients (not during the transformation!).

Consider a sub-image (block) \( g(x,y) \) of size \( n \times n \), whose forward discrete transform is

\[
T(u,v) = \sum_{x=0}^{n-1} \sum_{y=0}^{n-1} g(x,y)r(x,y,u,v)
\]

Here \( u,v = 0,1,2,\ldots,n-1 \).
Block transform coding

The inverse transform is

$$g(x, y) = \sum_{x=0}^{n-1} \sum_{y=0}^{n-1} T(u, v)s(x, y, u, v)$$

where $r$ and $s$ represent the forward and inverse transformation kernels (basis functions, basis images). $T(u, v)$ are transform coefficients. The kernel is separable if

$$r(x, y, u, v) = r_1(x, u)r_2(y, v)$$

The kernel is symmetric if

$$r(x, y, u, v) = r_1(x, u)r_1(y, v)$$

The same properties can be attributed to the inverse kernels.

Block transform coding

The forward and inverse transformation kernels determine the type of transform to be computed and the overall computational complexity. The simplified DFT kernel pair is

$$r(x, y, u, v) = e^{-j2\pi ux+vy \over n}$$
$$s(x, y, u, v) = {1 \over n^2} e^{j2\pi ux+vy \over n}$$

A computationally simpler is the Walsh-Hadamard transform (WHT)

$$r(x, y, u, v) = s(x, y, u, v) = {1 \over n} \left( \sum_{i=0}^{n-1} b_i(x)p_i(u)+b_i(y)p_i(v) \right)$$

where $n = 2^m$. The summation is performed in modulo 2 arithmetic and $b_k(z)$ is the $k^{th}$ bit of $z$. 
Block transform coding

The $p_i(u)$ are computed using

\[
p_0(u) = b_{m-1}(u)
\]
\[
p_1(u) = b_{m-1}(u) + b_{m-2}(u)
\]
\[
p_2(u) = b_{m-2}(u) + b_{m-3}(u)
\]
\[\vdots
\]
\[
p_{m-1}(u) = b_1(u) + b_0(u)
\]

Unlike the DFT kernels (which are sines and cosines), the WHT kernels are alternating +1s and -1s arranged in a checkerboard pattern. The kernel for $n = 4$ is shown.

Block transform coding

The *discrete cosine transform* (DCT) is the most frequently used transformation. Its kernels (one possible version) are

\[
r(x, y, u, v) = s(x, y, u, v) = \alpha(u)\alpha(v) \cos \left(\frac{(2x+1)u\pi}{2n}\right) \cos \left(\frac{(2y+1)v\pi}{2n}\right)
\]

where

\[
\alpha(u) = \begin{cases} 
\sqrt{1/n} & u = 0 \\
\sqrt{2/n} & u = 1, 2, \ldots, n-1 
\end{cases}
\]

The DCT kernel for $n = 4$ is shown.

The lighter intensity values correspond to larger values of the kernel.
Block transform coding

The original image was divided into 8x8 sub-images, transformed, truncated to 50% of the resulting coefficients, inverse transformed.

The small differences in mean-square reconstruction error observed in the previous example are related to the energy (or information) packing properties of the transform used. Since the reconstructed sub-image can be expressed as

\[ g(x, y) = \sum_{x=0}^{n-1} \sum_{y=0}^{n-1} T(u, v)s(x, y, u, v) \]

and since the inverse kernel depends only on the indexes \( x, y, u, v \) and not on the image values, the kernel can be viewed as defining a set of basis functions or basis images. The inverse transform can be rewritten

\[ G = \sum_{u=0}^{n-1} \sum_{v=0}^{n-1} T(u, v)S_{uv} \]

where \( G \) is an \( nxn \) matrix containing pixels of \( g(x,y) \) and
Block transform coding

\[ S_{uv} = \begin{bmatrix}
  s(0,0,u,v) & s(0,1,u,v) & \cdots & s(0,n-1,u,v) \\
  s(1,0,u,v) & \vdots & \ddots & \vdots \\
  \vdots & \vdots & \ddots & \vdots \\
  s(n-1,0,u,v) & s(n-1,1,u,v) & \cdots & s(n-1,n-1,u,v)
\end{bmatrix} \]

Therefore, \( G \) is explicitly defined as a linear combination of \( n^2 \) matrices of size \( nxn \). These matrices are the basis images of the series expansion specifying \( g(x,y) \) and \( T(u,v) \) are the expansion coefficients.

We may specify a transform \textit{masking function}:

\[ \chi(u,v) = \begin{cases} 
  0 & \text{if } T(u,v) \text{ satisfies a specified truncation criterion} \\
  1 & \text{otherwise}
\end{cases} \]

for \( u,v = 0,1,2,\ldots n-1 \)

Then, an approximation of \( G \) can be obtained from the truncated expansion:

\[ \hat{G} = \sum_{u=0}^{n-1} \sum_{v=0}^{n-1} \chi(u,v)T(u,v)S_{uv} \]

Here \( \chi(u,v) \) is constructed to eliminate the basis images making the smallest contribution to the total sum. The mean-square error between the subimage and its approximation is

\[ e_{uv} = E\left\{ \left\| G - \hat{G} \right\|^2 \right\} = E\left\{ \left\| \sum_{u=0}^{n-1} \sum_{v=0}^{n-1} T(u,v)S_{uv} - \sum_{n=0}^{n-1} \sum_{v=0}^{n-1} \chi(u,v)T(u,v)S_{uv} \right\|^2 \right\} \]

\[ = E\left\{ \left\| \sum_{u=0}^{n-1} \sum_{v=0}^{n-1} T(u,v)S_{uv} \left[ 1 - \chi(u,v) \right] \right\|^2 \right\} = \sum_{u=0}^{n-1} \sum_{v=0}^{n-1} \sigma_{T(u,v)}^2 \left[ 1 - \chi(u,v) \right] \]
Block transform coding

Here $\| x \|$ indicates the norm of $x$ and $\sigma_T(u,v)^2$ is the variance of the coefficient at transform location $(u,v)$. Therefore, the total mean-square approximation error is the sum of the variances of the discarded transform coefficients: coefficients for which $\chi(u,v) = 0$. Transforms that redistribute or pack the most information into the fewest coefficients provide the best subimage approximations and, therefore, the smallest reconstruction errors. Assuming that the mean-square error of the $MN/n^2$ subimages of an $M \times N$ image are identical, the mean-square error of the entire image equals the mean-square error of a single subimage.

The information packing ability of DCT is superior to that of DFT and WHT. On the other hand, the Karhunen-Loeve transform is the optimal transform in the information packing sense. However, KLT is data dependent (basis functions vary) and is not frequently used.

Therefore, most transform coding systems are based on DCT, which became an international standard for transform coding systems. Compared to other input independent transforms, DCT packs the most information into the fewest coefficients (for most images) and also minimizes the block-like appearance (blocking artifact – the boundaries between subimages become visible).

This property is particularly important in comparison with other sinusoidal transforms.
Block transform coding

The implicit $n$-point periodicity of DFT leads to boundary discontinuities resulting in substantial high frequency content. When the DFT coefficients are truncated or quantized, the Gibbs phenomenon leads to appearance of blocking artifacts. The DCT reduces this effect since its $2n$-point periodicity does not produce boundary discontinuities.

![Implicit periodicity of 1D DFT](image1)

![Implicit periodicity of 1D DCT](image2)

Block transform coding

Subimage size selection

Another significant factor affecting errors in transform coding and computational complexity is subimage size. Usually, images are subdivided so that the correlation (redundancy) between adjacent subimages is reduced and so that the subimage size $n = 2^m$ for computational efficiency. In general, both the level of compression and computational complexity increase with subimage size.

The most popular subimage sizes are 8x8 and 16x16.

The coefficients were truncated to 75%.

WHT and DCT curves flatten when block size becomes greater than 16x16.
Block transform coding

All three curves intersect when 2x2 subimages are used. In this case, only one of four coefficients (25%) is kept. This coefficient in all cases is a DC component, so the inverse transform replaces the four block pixels by their average values.

Original image  Approximation using 25% of the DCT coefficients and 2x2 blocks  Approximation using 25% of the DCT coefficients and 4x4 blocks  Approximation using 25% of the DCT coefficients and 8x8 blocks

Blocking artifact decreases with block size increase

Block transform coding

Bit allocation

The reconstruction error associated with the truncated series expansion is a function of the relative importance of the transform coefficients that are discarded and of the precision that is used to present the retained coefficients. In most transform coding systems, the retained coefficients are selected based on maximum variance (zonal coding) or based on maximum magnitude (threshold coding). The overall process of truncating, quantizing, and coding the coefficients of a transformed sub-image is called bit allocation.
Block transform coding

Approximation using 12.5% of the 8x8 DCT coefficients with threshold coding and the difference image; $e_{rms} = 4.5$

Approximation using 12.5% of the 8x8 DCT coefficients with zonal coding and the difference image; $e_{rms} = 6.5$

Zonal coding implementations

Zonal coding is based on the information theory concept of considering information as uncertainty. Therefore, the transform coefficients of maximum variance carry the most image information and should be kept. The variances can be estimated either directly for each subimage or based on an assumed image model. In either case, zonal sampling can be achieved by multiplication each $T(u,v)$ by the element of zonal mask consisting of 1s in the locations of maximum variance and 0s everywhere else.

The retained coefficients must be quantized, so the zonal masks are sometimes show the number of bits used to code each coefficient.
Block transform coding

Threshold coding implementations

Zonal coding usually is implemented by using a single fixed mask for all subimages. Threshold coding, however, is always adaptive in the sense that the location of the retained transform coefficients vary from one subimage to another. This approach is computationally simpler and is used most often. For any subimage, the transform coefficients of largest magnitude make the most significant contribution to reconstructed subimage quality. Since their location vary from one subimage to another, the elements of $\chi(u,v)T(u,v)$ normally are reordered (in a predefined manner) to form a 1D run-length coded sequence.

---

Block transform coding

When the mask is applied to the subimage for which it was derived, and the resulting $nxn$ array is reordered to form an $n^2$-element coefficient sequence according to the zigzag ordering pattern, the reordered 1D sequence contains several long runs of 0s. These runs are normally run-length coded: the retained coefficients are represented by a variable-length code.

There are three basic ways to threshold a transformed subimage:

1. A single global threshold applied to all subimages – the compression level differs from image to image;
2. Individual thresholds can be used for each subimage – $N$-largest the same number of coefficients is discarded from each subimage – constant code rate;
3. The threshold can be varied as a function of the location of each coefficient within the subimage – variable code rate.
Block transform coding

In the third case, the thresholded and quantized version of $T(u,v)$ is

$$\hat{T}(u,v) = \text{round} \left[ \frac{T(u,v)}{Z(u,v)} \right]$$

where $Z(u,v)$ is an element of the transform normalization array

$$Z = \begin{bmatrix} Z(0,0) & Z(0,1) & \cdots & Z(0,n-1) \\ Z(1,0) & \vdots & \cdots & \vdots \\ \vdots & \vdots & \cdots & \vdots \\ Z(n-1,0) & \cdots & Z(n-1,n-1) \end{bmatrix}$$

Before taking the inverse transform, the transformed subimage $\hat{T}(u,v)$ must be multiplied by $Z(u,v)$. The resulting de-normalized array is

$$\hat{T}(u,v) = \hat{T}(u,v)Z(u,v)$$

Block transform coding

The inverse transform of the de-normalized array yields the decompressed image approximation.

A threshold coding quantization curve and a normalization matrix.

Note that $\hat{T}(u,v)$ takes integer value $k$ iff

$$k\frac{c}{2} - \frac{c}{2} \leq T(u,v) < k\frac{c}{2} + \frac{c}{2}$$
Block transform coding

Threshold-coded approximations using 8x8 DCT and the previously shown normalization array $\mathbf{Z}$

$\mathbf{Z}: C = 12, \sigma_{rms} = 3.83$
$\mathbf{2Z}: C = 19, \sigma_{rms} = 4.93$
$\mathbf{8Z}: C = 49, \sigma_{rms} = 9.35$
$\mathbf{16Z}: C = 85, \sigma_{rms} = 13.94$

$\mathbf{4Z}: C = 30, \sigma_{rms} = 6.62$
$\mathbf{32Z}: C = 182, \sigma_{rms} = 22.5$