REPETITION, PART II

Ole-Johan Skrede

24.05.2017

INF2310 - Digital Image Processing

Department of Informatics
The Faculty of Mathematics and Natural Sciences
University of Oslo

TODAY'S LECTURE

- Coding and compression
  - Information theory
  - Shannon-Fano coding
  - Huffman coding
  - Arithmetic coding
  - Difference transform
  - Run-length coding
  - Lempel-Ziv-Welch coding
  - Lossy JPEG compression
- Binary morphology
  - Fundamentals: Structuring element, erosion and dilation
  - Opening and closing
  - Hit-or-miss transform
  - Morphological thinning

CODING AND COMPRESSION I

COMPRESSION AND DECOMPRESSION

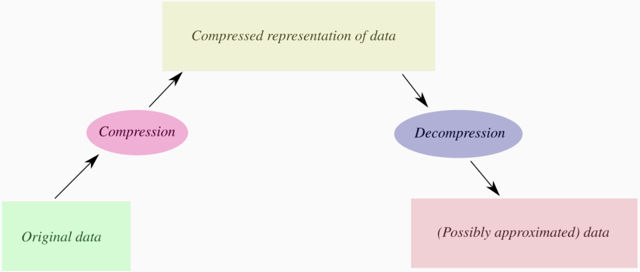


Figure 1: Compression and decompression pipeline

- We would like to compress our data, both to reduce storage and transmission load.
- In *compression*, we try to create a representation of the data which is smaller in size, while preserving vital information. That is, we throw away redundant information.
- The original data (or an approximated version) can be retrieved through *decompression*.

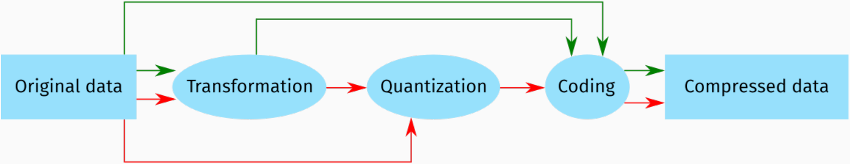


Figure 2: Three steps of compression

We can group compression in to three steps:

- **Transformation:** A more compact image representation.

COMPRESSION

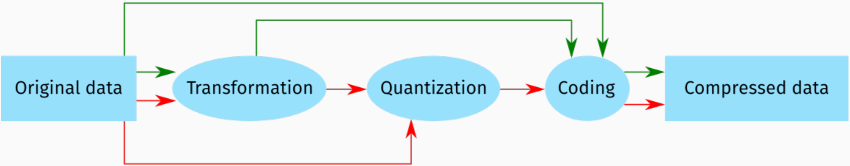


Figure 2: Three steps of compression

We can group compression in to three steps:

- **Transformation:** A more compact image representation.
- **Qunatization:** Representation approximation.

COMPRESSION

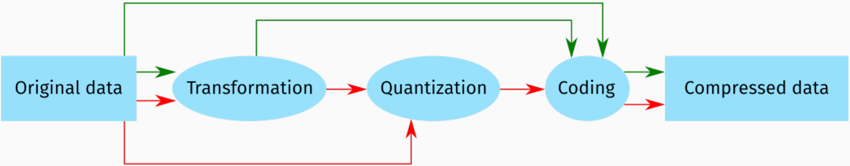


Figure 2: Three steps of compression

We can group compression in to three steps:

- **Transformation:** A more compact image representation.
- **Qunatization:** Representation approximation.
- **Coding:** Transformation from one set of symbols to another.
    - **Encoding:** Coding from an original format to some other format. E.g. encoding a digital image from raw numbers to JPEG.
    - **Decoding:** The reverse process, coding from some format to the original. E.g. decoding a JPEG image back to raw numbers.

LOSSY AND LOSSLESS COMPRESSION

Compression can either be *lossless* or *lossy*. There exists a number of methods for both types.

Lossless: We are able to perfectly reconstruct the original image.

Lossy: We can only reconstruct the original image to a certain degree (but not perfect).

REDUNDANCY

- We can use a different amount of data on the same signal.

REDUNDANCY

- We can use a different amount of data on the same signal.
- E.g. the signal `13`
    - ISO 8859-1 (Latin-1): 16 bits: 8 bits for `1` (at `0x31`) and 8 bits for `3` (at `0x33`).
    - 8-bit natural binary encoding: 8 bits: `00001101`
    - 4-bit natural binary encoding: 4 bits: `1101`

REDUNDANCY

- We can use a different amount of data on the same signal.
- E.g. the signal `13`
  - ISO 8859-1 (Latin-1): 16 bits: 8 bits for `1` (at `0x31`) and 8 bits for `3` (at `0x33`).
  - 8-bit natural binary encoding: 8 bits: `00001101`
  - 4-bit natural binary encoding: 4 bits: `1101`
- **Redundancy:** What can be removed from the data without loss of (relevant) information.
- In compression, we want to remove redundant bits.

DIFFERENT TYPES OF REDUNDANCY

- *Psychovisual redundancy*
    - Information that we cannot percieve.
    - Can be compressed by e.g. subsampling or by reducing the number of bits per pixel.

DIFFERENT TYPES OF REDUNDANCY

- *Psychovisual redundancy*
    - Information that we cannot percieve.
    - Can be compressed by e.g. subsampling or by reducing the number of bits per pixel.
- *Inter-pixel temporal redundancy*
    - Correlation between successive images in a sequence.
    - A sequence can be compressed by only storing some frames, and then only differences for the rest of the sequence.

DIFFERENT TYPES OF REDUNDANCY

- *Psychovisual redundancy*
  - Information that we cannot percieve.
  - Can be compressed by e.g. subsampling or by reducing the number of bits per pixel.
- *Inter-pixel temporal redundancy*
  - Correlation between successive images in a sequence.
  - A sequence can be compressed by only storing some frames, and then only differences for the rest of the sequence.
- *Inter-pixel spatial redundancy*
  - Correlation between neighbouring pixels within an image.
  - Can be compressed by e.g. run-length methods.

DIFFERENT TYPES OF REDUNDANCY

- *Psychovisual redundancy*
  - Information that we cannot percieve.
  - Can be compressed by e.g. subsampling or by reducing the number of bits per pixel.
- *Inter-pixel temporal redundancy*
  - Correlation between successive images in a sequence.
  - A sequence can be compressed by only storing some frames, and then only differences for the rest of the sequence.
- *Inter-pixel spatial redundancy*
  - Correlation between neighbouring pixels within an image.
  - Can be compressed by e.g. run-length methods.
- *Coding redundancy*
  - Information is not represented optimally by the symbols in the code.
  - This is often measured as the difference between average code length and some theoretical minimum code length.

- The *compression rate* is defined as the ratio between the *uncompressed* size and *compressed* size

$$\text{Compression rate} = \frac{\text{Uncompressed size}}{\text{Compressed size}}$$

  or as the ratio between the *mean number of bits per symbol* in the compressed and uncompressed signal.

COMPRESSION RATE AND REDUNDANCY

- The *compression rate* is defined as the ratio between the *uncompressed* size and *compressed* size

$$\textit{Compression rate} = \frac{\textit{Uncompressed size}}{\textit{Compressed size}}$$

  or as the ratio between the *mean number of bits per symbol* in the compressed and uncompressed signal.
- *Space saving* is defined as the reduction in size relative to the uncompressed size, and is given as

$$\textit{Space savings} = 1 - \frac{\textit{Compressed size}}{\textit{Unompressed size}}$$

COMPRESSION RATE AND REDUNDANCY

- The *compression rate* is defined as the ratio between the *uncompressed* size and *compressed* size

$$\textit{Compression rate} = \frac{\textit{Uncompressed size}}{\textit{Compressed size}}$$

  or as the ratio between the *mean number of bits per symbol* in the compressed and uncompressed signal.

- *Space saving* is defined as the reduction in size relative to the uncompressed size, and is given as

$$\textit{Space savings} = 1 - \frac{\textit{Compressed size}}{\textit{Unompressed size}}$$

- Example: An 8-bit 512×512 image has an uncompressed size of 256 kiB, and a size of 64 kiB after compression.
    - *Compression rate*: 4
    - *Space saving*: 3/4

EXPECTED CODE LENGTH

- The *expected length* L_c of a source code c for a random variable X with pmf. p_X is defined as

$$L_c = \sum_{x \in \mathcal{X}} p_X(x) l_c(x)$$

where $l_c(x)$ is the length of the codeword assigned to x in this source code.

- The *expected length* L_c of a source code c for a random variable X with pmf. p_X is defined as

$$L_c = \sum_{x \in \mathcal{X}} p_X(x) l_c(x)$$

where $l_c(x)$ is the length of the codeword assigned to x in this source code.
- Example: Let X be a random variable taking values in $\{1, 2, 3, 4\}$ with probabilities defined by p_X below.

- The *expected length* L_c of a source code c for a random variable X with pmf. p_X is defined as

$$L_c = \sum_{x \in \mathcal{X}} p_X(x) l_c(x)$$

where $l_c(x)$ is the length of the codeword assigned to x in this source code.
- Example: Let X be a random variable taking values in $\{1, 2, 3, 4\}$ with probabilities defined by p_X below.
- Let us encode this with a variable length source code c_v, and a source code c_e with equal length codewords.

$$\begin{aligned}
p_X(1) &= \tfrac{1}{2} & c_v(1) &= \mathtt{0} & c_e(1) &= \mathtt{00} \\
p_X(2) &= \tfrac{1}{4} & c_v(2) &= \mathtt{10} & c_e(2) &= \mathtt{01} \\
p_X(3) &= \tfrac{1}{8} & c_v(3) &= \mathtt{110} & c_e(3) &= \mathtt{10} \\
p_X(4) &= \tfrac{1}{8} & c_v(4) &= \mathtt{111} & c_e(4) &= \mathtt{11}
\end{aligned}$$

- The *expected length* L_c of a source code c for a random variable X with pmf. p_X is defined as

$$L_c = \sum_{x \in \mathcal{X}} p_X(x) l_c(x)$$

where $l_c(x)$ is the length of the codeword assigned to x in this source code.
- Example: Let X be a random variable taking values in $\{1, 2, 3, 4\}$ with probabilities defined by p_X below.
- Let us encode this with a variable length source code c_v, and a source code c_e with equal length codewords.

$$\begin{aligned} p_X(1) &= \tfrac{1}{2} & c_v(1) &= \quad 0 & c_e(1) &= \quad 00 \\ p_X(2) &= \tfrac{1}{4} & c_v(2) &= \quad 10 & c_e(2) &= \quad 01 \\ p_X(3) &= \tfrac{1}{8} & c_v(3) &= \quad 110 & c_e(3) &= \quad 10 \\ p_X(4) &= \tfrac{1}{8} & c_v(4) &= \quad 111 & c_e(4) &= \quad 11 \end{aligned}$$

- Expected length of the variable length coding: $L_{c_v} = 1.75$ bits.

EXPECTED CODE LENGTH

- The *expected length* L_c of a source code c for a random variable X with pmf. p_X is defined as

$$L_c = \sum_{x \in \mathcal{X}} p_X(x) l_c(x)$$

where $l_c(x)$ is the length of the codeword assigned to x in this source code.
- Example: Let X be a random variable taking values in $\{1, 2, 3, 4\}$ with probabilities defined by p_X below.
- Let us encode this with a variable length source code c_v, and a source code c_e with equal length codewords.

$$\begin{aligned}
p_X(1) &= \tfrac{1}{2} & c_v(1) &= \text{0} & c_e(1) &= \text{00} \\
p_X(2) &= \tfrac{1}{4} & c_v(2) &= \text{10} & c_e(2) &= \text{01} \\
p_X(3) &= \tfrac{1}{8} & c_v(3) &= \text{110} & c_e(3) &= \text{10} \\
p_X(4) &= \tfrac{1}{8} & c_v(4) &= \text{111} & c_e(4) &= \text{11}
\end{aligned}$$

- Expected length of the variable length coding: $L_{c_v} = 1.75$ bits.
- Expected length of the equal length coding: $L_{c_e} = 2$ bits.

INFORMATION CONTENT

- We use *information content* (aka *self-information* or *surprisal*) to measure the level of information in an event X.

INFORMATION CONTENT

- We use *information content* (aka *self-information* or *surprisal*) to measure the level of information in an event X.
- The information content $I_X(x)$ (measured in *bits*) of an event x is defined as

$$I_X(x) = \log_2 \frac{1}{p_X(x)}.$$

INFORMATION CONTENT

- We use *information content* (aka *self-information* or *surprisal*) to measure the level of information in an event X.
- The information content $I_X(x)$ (measured in *bits*) of an event x is defined as

$$I_X(x) = \log_2 \frac{1}{p_X(x)}.$$

- It can also be measured in *nats*

$$I_X(x) = \log_e \frac{1}{p_X(x)},$$

  that is, with the natural logarithm, or in *hartleys*

$$I_X(x) = \log_{10} \frac{1}{p_X(x)}.$$

INFORMATION CONTENT

- We use *information content* (aka *self-information* or *surprisal*) to measure the level of information in an event X.
- The information content $I_X(x)$ (measured in *bits*) of an event x is defined as

$$I_X(x) = \log_2 \frac{1}{p_X(x)}.$$

- It can also be measured in *nats*

$$I_X(x) = \log_e \frac{1}{p_X(x)},$$

that is, with the natural logarithm, or in *hartleys*

$$I_X(x) = \log_{10} \frac{1}{p_X(x)}.$$

- If an event X takes value x with probability 1, the information content $I_X(x) = 0$.

- The *entropy* of a random variable X taking values $x \in \mathcal{X}$, and with pmf. p_X, is defined as

$$H(X) = \sum_{x \in \mathcal{X}} p_X(x) I_X(x)$$
$$= -\sum_{x \in \mathcal{X}} p_X(x) \log p_X(x),$$

and is thus the expected information content in X, measured in bits (unless another base for the logarithm is explicitly stated).

ENTROPY

- The *entropy* of a random variable X taking values $x \in \mathcal{X}$, and with pmf. p_X, is defined as

$$H(X) = \sum_{x \in \mathcal{X}} p_X(x) I_X(x)$$
$$= -\sum_{x \in \mathcal{X}} p_X(x) \log p_X(x),$$

  and is thus the expected information content in X, measured in bits (unless another base for the logarithm is explicitly stated).
- We use the convention that $x \log(x) = 0$ when $x = 0$.

ENTROPY

- The *entropy* of a random variable X taking values $x \in \mathcal{X}$, and with pmf. p_X, is defined as
$$\begin{aligned} H(X) &= \sum_{x \in \mathcal{X}} p_X(x) I_X(x) \\ &= -\sum_{x \in \mathcal{X}} p_X(x) \log p_X(x), \end{aligned}$$
and is thus the expected information content in X, measured in bits (unless another base for the logarithm is explicitly stated).
- We use the convention that $x \log(x) = 0$ when $x = 0$.
- The entropy of a signal (a collection of events) gives a lower bound on how compact the sequence can be encoded (if every event is encoded separately).

ENTROPY

- The *entropy* of a random variable X taking values $x \in \mathcal{X}$, and with pmf. p_X, is defined as

$$H(X) = \sum_{x \in \mathcal{X}} p_X(x) I_X(x) = -\sum_{x \in \mathcal{X}} p_X(x) \log p_X(x),$$

  and is thus the expected information content in X, measured in bits (unless another base for the logarithm is explicitly stated).
- We use the convention that $x \log(x) = 0$ when $x = 0$.
- The entropy of a signal (a collection of events) gives a lower bound on how compact the sequence can be encoded (if every event is encoded separately).
- The entropy of a fair coin toss is 1 bit (since $-2\frac{1}{2}\log_2\frac{1}{2} = 1$)

ENTROPY

- The *entropy* of a random variable X taking values $x \in \mathcal{X}$, and with pmf. p_X, is defined as

$$H(X) = \sum_{x \in \mathcal{X}} p_X(x) I_X(x)$$
$$= -\sum_{x \in \mathcal{X}} p_X(x) \log p_X(x),$$

  and is thus the expected information content in X, measured in bits (unless another base for the logarithm is explicitly stated).
- We use the convention that $x \log(x) = 0$ when $x = 0$.
- The entropy of a signal (a collection of events) gives a lower bound on how compact the sequence can be encoded (if every event is encoded separately).
- The entropy of a fair coin toss is 1 bit (since $-2\frac{1}{2}\log_2\frac{1}{2} = 1$)
- The entropy of a fair dice toss is ≈2.6 bit (since $-6\frac{1}{6}\log_2\frac{1}{6} \approx 2.6$)

ESTIMATING THE PROBABILITY MASS FUNCTION

- We can estimate the probability mass function with the normalized histogram.

ESTIMATING THE PROBABILITY MASS FUNCTION

- We can estimate the probability mass function with the normalized histogram.
- For a signal of length n with symbols taking values in the alphabeth $\{s_0, \ldots, s_{m-1}\}$, let n_i be the number of occurances of s_i in the signal, then the normalized histogram value for symbol s_i is

$$p_i = \frac{n_i}{n}.$$

ESTIMATING THE PROBABILITY MASS FUNCTION

- We can estimate the probability mass function with the normalized histogram.
- For a signal of length n with symbols taking values in the alphabeth $\{s_0, \ldots, s_{m-1}\}$, let n_i be the number of occurances of s_i in the signal, then the normalized histogram value for symbol s_i is

$$p_i = \frac{n_i}{n}.$$

- If one assume that the values in the signal are independent realizations of an underlying random variable, then p_i is an estimate on the probability that the variable is s_i.

SHANNON-FANO CODING

- A simple method that produces an instantaneous code.

SHANNON-FANO CODING

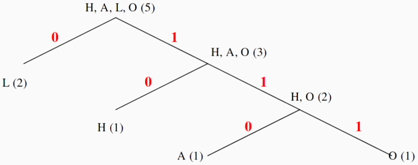
- A simple method that produces an instantaneous code.
- The resulting is quite compact (but not optimal).

SHANNON-FANO CODING

- A simple method that produces an instantaneous code.
- The resulting is quite compact (but not optimal).
- Algorithm that produces a binary Shannon-Fano code (with alphabeth $\{0, 1\}$):
  1. Sort the symbols x_i of the signal that we want to code by probability of occurance.
  2. Split the symbols into two parts with approximately equal accumulated probability.
     - One group is assigned the symbol 0, and the other the symbol 1.
     - Do this step recursively (that is, do this step on every subgroup), until the group only contain one element.
  3. The result is a binary tree with the symbols that are to be encoded in the leaf nodes.
  4. Traverse the tree from root to the leaf nodes and record the sequence of symbols in order to produce the corresponding codeword.

SHANNON-FANO EXAMPLE

Two different encodings of the sequence ”HALLO”.

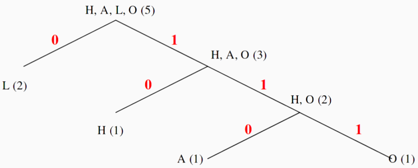


x	$p_X(x)$	$c(x)$	$l(x)$
L	$2/5$	0	1
H	$1/5$	10	2
A	$1/5$	110	3
O	$1/5$	111	3

c(HALLO) = 1011000111, with length 10 bits.

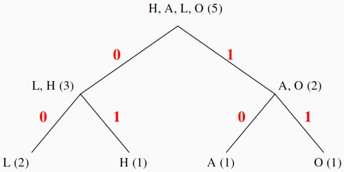
SHANNON-FANO EXAMPLE

Two different encodings of the sequence ”HALLO”.



x	$p_X(x)$	$c(x)$	$l(x)$
L	$2/5$	0	1
H	$1/5$	10	2
A	$1/5$	110	3
O	$1/5$	111	3

c(HALLO) = 1011000111, with length 10 bits.



x	$p_X(x)$	$c(x)$	$l(x)$
L	$2/5$	00	2
H	$1/5$	01	2
A	$1/5$	10	2
O	$1/5$	11	2

c(HALLO) = 0110000011, with length 10 bits.

- An instantaneous coding algorithm.

HUFFMAN CODING

- An instantaneous coding algorithm.
- *Optimal* in the sense that it achieves minimal coding redundancy.

- An instantaneous coding algorithm.
- *Optimal* in the sense that it achieves minimal coding redundancy.
- Algorithm for encoding a sequence of n symbols with a binary Huffman code and alphabeth $\{0, 1\}$:
  1. Sort the symbols by decreasing probability.
  2. Merge the two least likely symbols to a group and give the group a probability equal to the sum of the probabilities of the members in the group. Sort the new sequence by decreasing probability.
  3. Repeat step 2. until there are only two groups left.
  4. Represent the merging as a binary tree, and assign 0 to the left branch and 1 to the right branch.
  5. Every symbol in the original sequence is now at a leaf node. Traverse the tree from the root to the corresponding leaf node, and append the symbols from the traversal to create the codeword.

The six most common letters in the english language, and their relative occurance frequency (normslized within this selection), is given in the table below. The resulting Huffman source code is given as c.

x	$p(x)$	$c(x)$
a	0.160	000
e	0.248	10
i	0.137	010
n	0.131	011
o	0.146	001
t	0.178	11

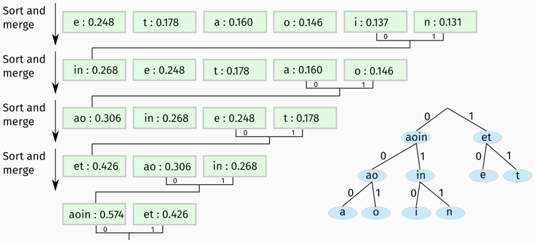


Figure 3: Huffman procedure example with resulting binary tree.

- The expected codeword length in the previous example is

$$
\begin{aligned}
L &= \sum_i l_i p_i \\
&= 3 \cdot 0.160 + 2 \cdot 0.248 + 3 \cdot 0.137 + 3 \cdot 0.131 + 3 \cdot 0.146 + 2 \cdot 0.178 \\
&= 2.574
\end{aligned}
$$

- The expected codeword length in the previous example is

$$
\begin{aligned}
L &= \sum_i l_i p_i \\
&= 3 \cdot 0.160 + 2 \cdot 0.248 + 3 \cdot 0.137 + 3 \cdot 0.131 + 3 \cdot 0.146 + 2 \cdot 0.178 \\
&= 2.574
\end{aligned}
$$

- And the entropy is

$$
\begin{aligned}
H &= -\sum_i p_i \log_2 p_i \\
&\approx 2.547
\end{aligned}
$$

- Thus, the coding redundancy is $L - H \approx 0.027$.

IDEAL AND ACTUAL CODE-WORD LENGTH

- For an optimal code, the expected codeword length L must be equal to the entropy

$$\sum_x p(x)l(x) = -\sum_x p(x)\log_2 p(x)$$

- That is, $l(x) = \log_2(1/p(x))$, which is the information content of the event x.

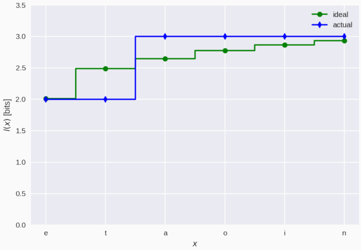


Figure 4: Ideal and actual codeword length from example in fig. 3

- The ideal codeword length is

$$l(x) = \log_2(1/p(x))$$

WHEN DOES HUFFMAN CODING NOT GIVE ANY CODING REDUNDANCY

- The ideal codeword length is

$$l(x) = \log_2(1/p(x))$$

- Since we only deal with integer codeword lengths, this is only possible when

$$p(x) = \frac{1}{2^k}$$

for some integer k.

WHEN DOES HUFFMAN CODING NOT GIVE ANY CODING REDUNDANCY

- The ideal codeword length is

$$l(x) = \log_2(1/p(x))$$

- Since we only deal with integer codeword lengths, this is only possible when

$$p(x) = \frac{1}{2^k}$$

for some integer k.

- Example

x	x_1	x_2	x_3	x_4	x_5	x_6
$p(x)$	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{32}$	$\frac{1}{64}$
$c(x)$	0	10	110	1110	11110	11111

- In this example $L = H = 1.9375$, that is, no coding redundancy.

- Lossless compression method.

ARITHMETIC CODING

- Lossless compression method.
- Variable code length, codes more probable symbols more compactly.

ARITHMETIC CODING

- Lossless compression method.
- Variable code length, codes more probable symbols more compactly.
- Contrary to Shannon-Fano coding and Huffman coding, which codes symbol by symbol, arithmetic coding encodes the entire signal to one number $d \in [0, 1)$.

- Lossless compression method.
- Variable code length, codes more probable symbols more compactly.
- Contrary to Shannon-Fano coding and Huffman coding, which codes symbol by symbol, arithmetic coding encodes the entire signal to one number $d \in [0, 1)$.
- Same expected codeword length as Huffman code.

- Lossless compression method.
- Variable code length, codes more probable symbols more compactly.
- Contrary to Shannon-Fano coding and Huffman coding, which codes symbol by symbol, arithmetic coding encodes the entire signal to one number $d \in [0, 1)$.
- Same expected codeword length as Huffman code.
- Can achieve shorter codewords for the entire sequence than Huffman code. This is because one is not limited to integer codewords for each symbol.

- The signal is a string of symbols x, where the symbols are taken from some alphabeth $\mathcal{X}$.

- The signal is a string of symbols x, where the symbols are taken from some alphabeth $\mathcal{X}$.
- As usual, we model the symbols as realizations of a discrete random variable X with an associated probability mass function p_X.

ARITHMETIC CODING: ENCODING OVERVIEW

- The signal is a string of symbols x, where the symbols are taken from some alphabeth $\mathcal{X}$.
- As usual, we model the symbols as realizations of a discrete random variable X with an associated probability mass function p_X.
- For each symbol in the signal, we use the pmf. to associate a unique interval with the part of the signal we have processed so far.

ARITHMETIC CODING: ENCODING OVERVIEW

- The signal is a string of symbols x, where the symbols are taken from some alphabeth $\mathcal{X}$.
- As usual, we model the symbols as realizations of a discrete random variable X with an associated probability mass function p_X.
- For each symbol in the signal, we use the pmf. to associate a unique interval with the part of the signal we have processed so far.
- At the end, when the whole signal is processed, we are left with a decimal interval which is unique to the string of symbols that is our signal.

ARITHMETIC CODING: ENCODING OVERVIEW

- The signal is a string of symbols x, where the symbols are taken from some alphabeth $\mathcal{X}$.
- As usual, we model the symbols as realizations of a discrete random variable X with an associated probability mass function p_X.
- For each symbol in the signal, we use the pmf. to associate a unique interval with the part of the signal we have processed so far.
- At the end, when the whole signal is processed, we are left with a decimal interval which is unique to the string of symbols that is our signal.
- We then find the number within this decimal with the shortest binary representation, and use this as the encoded signal.

- Suppose we have an alphabeth $\{a_1, a_2, a_3, a_4\}$ with associated pmf. $p_X = [0.2, 0.2, 0.4, 0.2]$.

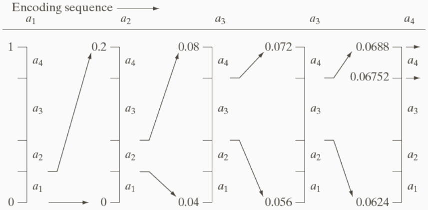
ARITHMETIC ENCODING: EXAMPLE

- Suppose we have an alphabeth $\{a_1, a_2, a_3, a_4\}$ with associated pmf. $p_X = [0.2, 0.2, 0.4, 0.2]$.
- We want to encode the sequence $a_1 a_2 a_3 a_3 a_4$.

- Suppose we have an alphabeth $\{a_1, a_2, a_3, a_4\}$ with associated pmf. $p_X = [0.2, 0.2, 0.4, 0.2]$.
- We want to encode the sequence $a_1 a_2 a_3 a_3 a_4$.

Step-by-step solution, current interval is initialized to $[0, 1)$.

Symbol	Interval	Sequence	Interval
a_1	$[0.0, 0.2)$	a_1	$[0.0, 0.2)$
a_2	$[0.2, 0.4)$	$a_1 a_2$	$[0.04, 0.08)$
a_3	$[0.4, 0.8)$	$a_1 a_2 a_3$	$[0.056, 0.072)$
a_3	$[0.4, 0.8)$	$a_1 a_2 a_3 a_3$	$[0.0624, 0.0688)$
a_4	$[0.8, 1.0)$	$a_1 a_2 a_3 a_3 a_4$	$[0.06752, 0.0688)$



ARITHMETIC DECODING

- Given an encoded signal $b_1 b_2 b_3 \cdots b_k$, we first find the decimal representation

$$d = \sum_{n=1}^{k} b_n \left(\frac{1}{2}\right)^n$$

- Given an encoded signal $b_1 b_2 b_3 \cdots b_k$, we first find the decimal representation

$$d = \sum_{n=1}^{k} b_n \left(\frac{1}{2}\right)^n$$

- Similar to what we did in the encoding, we define a list of interval edges based on the pmf $q = [0, p_X(x_1), p_X(x_1) + p_X(x_2), \ldots, \sum_{i=1}^{k} p_X(x_k), \ldots]$

- Given an encoded signal $b_1 b_2 b_3 \cdots b_k$, we first find the decimal representation

$$d = \sum_{n=1}^{k} b_n \left(\frac{1}{2}\right)^n$$

- Similar to what we did in the encoding, we define a list of interval edges based on the pmf $q = [0, p_X(x_1), p_X(x_1) + p_X(x_2), \ldots, \sum_{i=1}^{k} p_X(x_k), \ldots]$
  1. See what interval the decimal number lies in, set this as the current interval.
  2. Decode the symbol corresponding to this interval (this is found via the alphabeth and q).
  3. Scale the q to lie within the current interval.
  4. Do step 1 to 3 until termination.
- Termination:
  - Define a **eod** symbol (*end of data*), and stop when this is decoded. Note that this will also need an associated probability in the model.
  - Or, only decode a predefined number of symbols.

- Alphabeth: $\{a, b, c\}$. $p_X = [0.6, 0.2, 0.2]$. $q = [0.0, 0.6, 0.8, 1.0]$

DECODING: EXAMPLE

- Alphabeth: $\{a, b, c\}$. $p_X = [0.6, 0.2, 0.2]$. $q = [0.0, 0.6, 0.8, 1.0]$
- Signal to decode: 10001

- Alphabeth: $\{a, b, c\}$. $p_X = [0.6, 0.2, 0.2]$. $q = [0.0, 0.6, 0.8, 1.0]$
- Signal to decode: 10001
- First, we find that $0.10001_2 = 0.53125_{10}$

DECODING: EXAMPLE

- Alphabeth: $\{a, b, c\}$. $p_X = [0.6, 0.2, 0.2]$. $q = [0.0, 0.6, 0.8, 1.0]$
- Signal to decode: 10001
- First, we find that $0.10001_2 = 0.53125_{10}$
- Then we continue decoding symbol for symbol until termination:

$[c_{\min}, c_{\max})$	q_{new}	Symbol	Sequence
$[0.0, 1.0)$	$[0.0, 0.6, 0.8, 1.0)$	a	a
$[0.0, 0.6)$	$[0.0, 0.36, 0.48, 0.6)$	c	ac
$[0.48, 0.6)$	$[0.48, 0.552, 0.576, 0.6)$	a	aca
$[0.48, 0.552)$	$[0.48, 0.5232, 0.5376, 0.552)$	b	$acab$
$[0.5232, 0.5376)$	$[0.5232, 0.53184, 0.53472, 0.5376)$	a	$acaba$

CODING AND COMPRESSION II

DIFFERENCE TRANSFORM

- Horizontal pixels have often quite similar intensity values.

DIFFERENCE TRANSFORM

- Horizontal pixels have often quite similar intensity values.
- Transform each pixelvalue $f(x, y)$ as the difference between the pixel at (x, y) and $(x, y - 1)$.
- That is, for an $m \times n$ image f, let $g[x, 0] = f[x, 0]$, and

$$g[x, y] = f[x, y] - f[x, y - 1], \quad y \in \{1, 2, \ldots, n - 1\} \tag{1}$$

for all rows $x \in \{0, 1, \ldots, m - 1\}$.

- Horizontal pixels have often quite similar intensity values.
- Transform each pixelvalue $f(x, y)$ as the difference between the pixel at (x, y) and $(x, y-1)$.
- That is, for an $m \times n$ image f, let $g[x, 0] = f[x, 0]$, and

$$g[x, y] = f[x, y] - f[x, y-1], \quad y \in \{1, 2, \ldots, n-1\} \tag{1}$$

  for all rows $x \in \{0, 1, \ldots, m-1\}$.
- Note that for an image f taking values in $[0, 2^b - 1]$, values of the transformed image g take values in $[-(2^b - 1), 2^b - 1]$.

DIFFERENCE TRANSFORM

- Horizontal pixels have often quite similar intensity values.
- Transform each pixelvalue $f(x, y)$ as the difference between the pixel at (x, y) and $(x, y-1)$.
- That is, for an $m \times n$ image f, let $g[x, 0] = f[x, 0]$, and

$$g[x, y] = f[x, y] - f[x, y-1], \quad y \in \{1, 2, \ldots, n-1\} \tag{1}$$

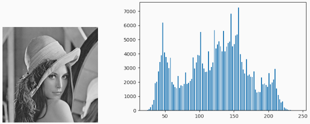
  for all rows $x \in \{0, 1, \ldots, m-1\}$.
- Note that for an image f taking values in $[0, 2^b - 1]$, values of the transformed image g take values in $[-(2^b - 1), 2^b - 1]$.
- This means that we need to use $b+1$ bits for each $g(x, y)$ if we are going to use equal-size codeword for every value.

DIFFERENCE TRANSFORM

- Horizontal pixels have often quite similar intensity values.
- Transform each pixelvalue $f(x, y)$ as the difference between the pixel at (x, y) and $(x, y-1)$.
- That is, for an $m \times n$ image f, let $g[x, 0] = f[x, 0]$, and

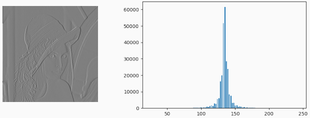
$$g[x, y] = f[x, y] - f[x, y-1], \quad y \in \{1, 2, \dots, n-1\} \tag{1}$$

  for all rows $x \in \{0, 1, \dots, m-1\}$.
- Note that for an image f taking values in $[0, 2^b - 1]$, values of the transformed image g take values in $[-(2^b - 1), 2^b - 1]$.
- This means that we need to use $b+1$ bits for each $g(x, y)$ if we are going to use equal-size codeword for every value.
- Often, the differences are close to 0, which means that natural binary coding of the differences are not optimal.



(a) Original (b) Graylevel intensity histogram

Figure 5: $H \approx 7.45 \implies c_r \approx 1.1$



(a) Difference transformed (b) Graylevel intensity histogram

Figure 6: $H \approx 5.07 \implies c_r \approx 1.6$

RUN-LENGTH TRANSFORM

- Often, images contain objects with similar intensity values.

RUN-LENGTH TRANSFORM

- Often, images contain objects with similar intensity values.
- Run-length transform use neighbouring pixels with *the same value*.
  - Note: This requires equality, not only similarity.
  - Run-length transform is compressing more with decreasing complexity.

RUN-LENGTH TRANSFORM

- Often, images contain objects with similar intensity values.
- Run-length transform use neighbouring pixels with *the same value*.
    - Note: This requires equality, not only similarity.
    - Run-length transform is compressing more with decreasing complexity.
- The run-length transform is reversible.

RUN-LENGTH TRANSFORM

- Often, images contain objects with similar intensity values.
- Run-length transform use neighbouring pixels with *the same value*.
  - Note: This requires equality, not only similarity.
  - Run-length transform is compressing more with decreasing complexity.
- The run-length transform is reversible.
- Codes sequences of values into sequences of tuples: (value, run-length).

RUN-LENGTH TRANSFORM

- Often, images contain objects with similar intensity values.
- Run-length transform use neighbouring pixels with *the same value*.
  - Note: This requires equality, not only similarity.
  - Run-length transform is compressing more with decreasing complexity.
- The run-length transform is reversible.
- Codes sequences of values into sequences of tuples: (value, run-length).
- Example:
  - Values (24 numbers): $3, 3, 3, 3, 3, 3, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 4, 4, 7, 7, 7, 7, 7, 7$.
  - Code (8 numbers): $(3, 6), (5, 10), (4, 2), (7, 6)$.
- The coding determines how many bits we use to store the tuples.

- In a binary image, we can ommit the *value* in coding. As long as we know what value is coded first, the rest have to be alternating values.
    - $0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1$
    - $5, 6, 2, 3, 5, 4$
- The histogram of the run-lengths is often not flat, entropy-coding should therefore be used to code the run-length sequence.

- A member of the *LZ** family of compression schemes.
- Utilizes patterns in the message by looking at symbol occurances, and therefore mostly reduces inter sample redundancy.
- Maps one symbol sequence to one code.

LEMPEL-ZIV-WELCH CODING

- A member of the *LZ** family of compression schemes.
- Utilizes patterns in the message by looking at symbol occurances, and therefore mostly reduces inter sample redundancy.
- Maps one symbol sequence to one code.
- Based on a *dictionary between symbol sequence and code* that is built on the fly.
  - This is done both in encoding and decoding.
  - The dictionary is not stored or transmitted.

LEMPEL-ZIV-WELCH CODING

- A member of the *LZ** family of compression schemes.
- Utilizes patterns in the message by looking at symbol occurances, and therefore mostly reduces inter sample redundancy.
- Maps one symbol sequence to one code.
- Based on a *dictionary between symbol sequence and code* that is built on the fly.
  - This is done both in encoding and decoding.
  - The dictionary is not stored or transmitted.
- The dictionary is initialized with an alphabeth of symbols of length one.

LZW ENCODING EXAMPLE

- Message: ababcbababaaaaabab#
- Initial dictionary: { #:0, a:1, b:2, c:3 }
- New dictionary entry: current string plus next unseen symbol

Message	Current string	Codeword	New dict entry
ababcbababaaaaabab#	a	1	ab:4
ababcbababaaaaabab#	b	2	ba:5
ababcbababaaaaabab#	ab	4	abc:6
ababcbababaaaaabab#	c	3	cb:7
ababcbababaaaaabab#	ba	5	bab:8
ababcbababaaaaabab#	bab	8	baba:9
ababcbababaaaaabab#	a	1	aa:10
ababcbababaaaaabab#	aa	10	aaa:11
ababcbababaaaaabab#	aa	10	aab:12
ababcbababaaaaabab#	bab	8	bab#:13
ababcbababaaaaabab#	#	0	

- Encoded message: 1,2,4,3,5,8,1,10,10,8,0.
- Assuming original bps = 8, and coded bps = 4, we achieve a compression rate of

$$c_r = \frac{8 \cdot 19}{4 \cdot 11} \approx 3.5 \quad (2)$$

LZW DECODING EXAMPLE

- Encoded message: 1,2,4,3,5,8,1,10,10,8,0
- Initial dictionary: { #:0, a:1, b:2, c:3 }
- New dictionary entry: current string plus first symbol in next string

Message	Current string	New dict entry Final	New dict entry Proposal
1,2,4,3,5,8,1,10,10,8,0	a		a?:4
1,2,4,3,5,8,1,10,10,8,0	b	ab:4	b?:5
1,2,4,3,5,8,1,10,10,8,0	ab	ba:5	ab?:6
1,2,4,3,5,8,1,10,10,8,0	c	abc:6	c?:7
1,2,4,3,5,8,1,10,10,8,0	ba	cb:7	ba?:8
1,2,4,3,5,8,1,10,10,8,0	bab	bab:8	bab?:9
1,2,4,3,5,8,1,10,10,8,0	a	baba:9	a?:10
1,2,4,3,5,8,1,10,10,8,0	aa	aa:10	aa?:11
1,2,4,3,5,8,1,10,10,8,0	aa	aaa:11	aa?:12
1,2,4,3,5,8,1,10,10,8,0	bab	aab:12	bab?:13
1,2,4,3,5,8,1,10,10,8,0	#	bab#:13	

Decoded message: ababcbababaaaaaabab#

LZW COMPRESSION, SUMMARY

- The LZW codes are normally coded with a natural binary coding.

LZW COMPRESSION, SUMMARY

- The LZW codes are normally coded with a natural binary coding.
- Typical text files are usually compressed with a factor of about 2.

LZW COMPRESSION, SUMMARY

- The LZW codes are normally coded with a natural binary coding.
- Typical text files are usually compressed with a factor of about 2.
- LZW coding is used a lot
  - In the Unix utility `compress` from 1984.
  - In the `GIF` image format.
  - An option in the `TIFF` and `PDF` format.

LZW COMPRESSION, SUMMARY

- The LZW codes are normally coded with a natural binary coding.
- Typical text files are usually compressed with a factor of about 2.
- LZW coding is used a lot
  - In the Unix utility `compress` from 1984.
  - In the `GIF` image format.
  - An option in the `TIFF` and `PDF` format.
- Experienced a lot of negative attention because of (now expired) patents. The `PNG` format was created in 1995 to get around this.
- The LZW can be coded further (e.g. with Huffman codes).

LZW COMPRESSION, SUMMARY

- The LZW codes are normally coded with a natural binary coding.
- Typical text files are usually compressed with a factor of about 2.
- LZW coding is used a lot
  - In the Unix utility `compress` from 1984.
  - In the `GIF` image format.
  - An option in the `TIFF` and `PDF` format.
- Experienced a lot of negative attention because of (now expired) patents. The `PNG` format was created in 1995 to get around this.
- The LZW can be coded further (e.g. with Huffman codes).
- Not all created codewords are used.

LZW COMPRESSION, SUMMARY

- The LZW codes are normally coded with a natural binary coding.
- Typical text files are usually compressed with a factor of about 2.
- LZW coding is used a lot
    - In the Unix utility `compress` from 1984.
    - In the `GIF` image format.
    - An option in the `TIFF` and `PDF` format.
- Experienced a lot of negative attention because of (now expired) patents. The `PNG` format was created in 1995 to get around this.
- The LZW can be coded further (e.g. with Huffman codes).
- Not all created codewords are used.
- We can limit the number of generated codewords.
    - Setting a limit on the number of codewords, and deleting old or seldomly used codewords.
    - Both the encoder and decoder need to have the same rules for deleting.

- Each image channel is partitioned into blocks of 8×8 piksels, and each block can be coded separately.
- For an image with 2^b intensity values, subtract 2^{b-1} to center the image values around 0 (if the image is originally in an unsigned format).
- Each block undergoes a 2D DCT. With this, most of the information in the 64 pixels is located in a small area in the Fourier space.

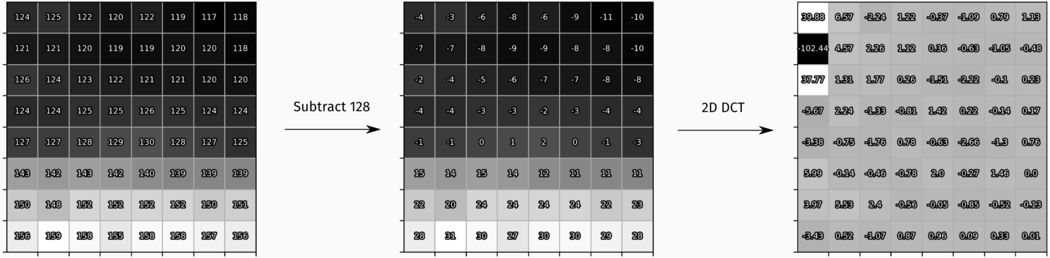


Figure 7: Example block, subtraction by 128, and 2D DCT.

LOSSY JPEG COMPRESSION: LOSS OF INFORMATION

- Each of the frequency-domain blocks are then point-divided by a quantization matrix.
- The result is rounded off to the nearest integer.
- his is where we lose information, but also why we are able to achieve high compression rates.
- This result is compressed by a coding method, before it is stored or transmitted.
- The DC and AC components are treated differently.

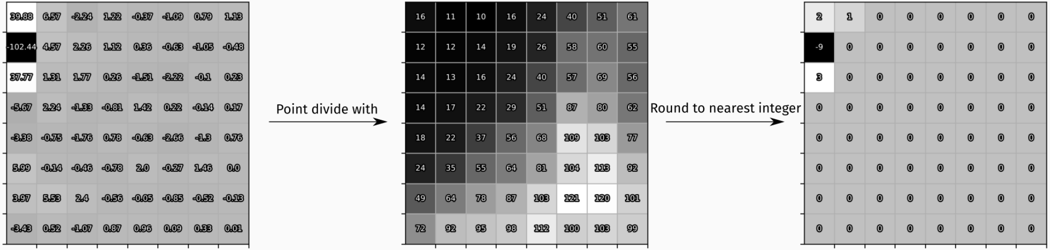


Figure 8: Divide the DCT block (left) with the quantization matrix (middle) and round to nearest integer (right)

LOSSY JPEG COMPRESSION: AC-COMPONENTS (SEQUENTIAL MODES)

1. The AC-components are zig-zag scanned:
    - The elements are ordered in a 1D sequence.
    - The absolute value of the elements will mostly descend through the sequence.
    - Many of the elements are zero, especially at the end of the sequence.
2. A zero-based run-length transform is performed on the sequence.
3. The run-length tuples are coded by Huffman or arithmetic coding.
    - The run-length tuple is here (number of 0's, number of bits in "non-0").
    - Arithmetic coding often gives $5 - 10\%$ better compression.

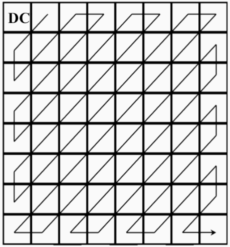


Figure 9: Zig-zag gathering of AC-components into a sequence.

LOSSY JPEG COMPRESSION: DC-COMPONENT

1. The DC-components are gathered from all the blocks in all the image channels.

LOSSY JPEG COMPRESSION: DC-COMPONENT

1. The DC-components are gathered from all the blocks in all the image channels.
2. These are correlated, and are therefore difference-transformed.

LOSSY JPEG COMPRESSION: DC-COMPONENT

1. The DC-components are gathered from all the blocks in all the image channels.
2. These are correlated, and are therefore difference-transformed.
3. The differences are coded by Huffman coding or arithmetic coding.
    - More precise: The number of bits in each difference is entropy coded.

LOSSY JPEG DECOMPRESSION: RECONSTRUCTION OF FREQUENCY-DOMAIN BLOCKS

- The coding part (Huffman- and arithmetic coding) is reversible, and gives the AC run-length tuples and the DC differences.
- The run-length transform and the difference transform are also reversible, and gives the scaled and quantized 2D DCT coefficients
- The zig-zag transform is also reversible, and gives (together with the restored DC component) an integer matrix.
- This matrix is multiplied with the quantization matrix in order to restore the sparse frequency-domain block.

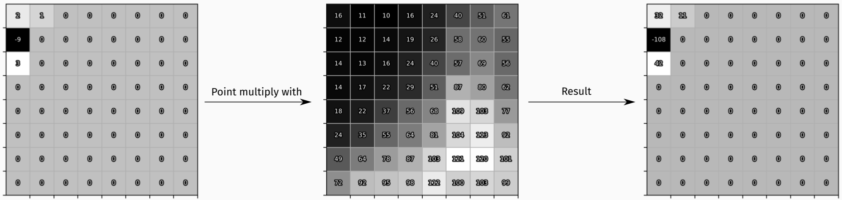


Figure 10: Multiply the quantized DCT components (left) with the quantization matrix (middle) to produce the sparse frequency-domain block (right).

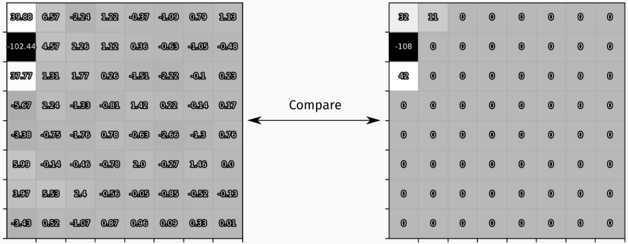


Figure 11: Comparison of the original 2D DCT components (left) and the restored (right)

- The restored DCT image is not equal to the original.

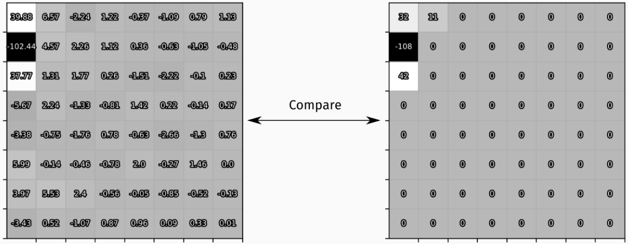


Figure 11: Comparison of the original 2D DCT components (left) and the restored (right)

- The restored DCT image is not equal to the original.
- But the major features are preserved

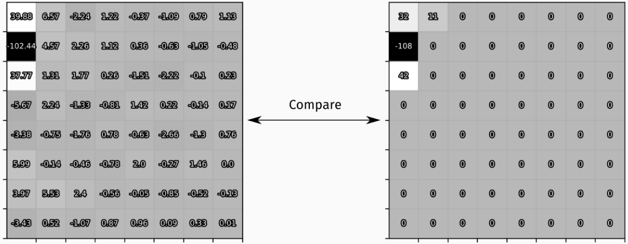


Figure 11: Comparison of the original 2D DCT components (left) and the restored (right)

- The restored DCT image is not equal to the original.
- But the major features are preserved
- Numbers with large absolute value in the top left corner.

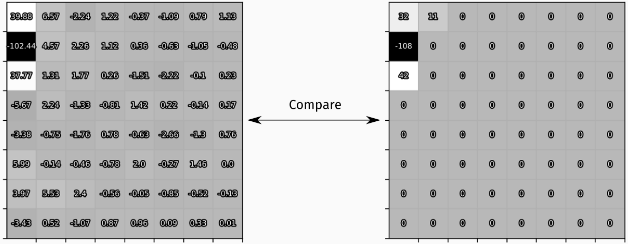


Figure 11: Comparison of the original 2D DCT components (left) and the restored (right)

- The restored DCT image is not equal to the original.
- But the major features are preserved
- Numbers with large absolute value in the top left corner.
- The components that was near zero in the original, are exactly zero in the restored version.

- We do an inverse 2D DCT on the sparse DCT component matrix.

$$f(x,y) = \frac{2}{\sqrt{mn}} \sum_{u=0}^{m} \sum_{v=0}^{n} c(u)c(v)F(u,v) \cos\left(\frac{(2x+1)u\pi}{2m}\right) \cos\left(\frac{(2y+1)v\pi}{2n}\right), \quad (3)$$

where

$$c(a) = \begin{cases} \frac{1}{\sqrt{2}} & \text{if } a = 0, \\ 1 & \text{otherwise.} \end{cases} \quad (4)$$

- We have then a restored image block which should be approximately equal to the original image block.

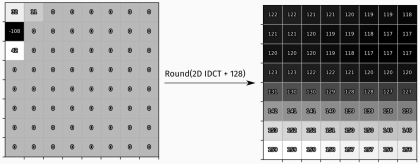


Figure 12: A 2D inverse DCT on the sparse DCT component matrix (left) produces an approximate image block (right)

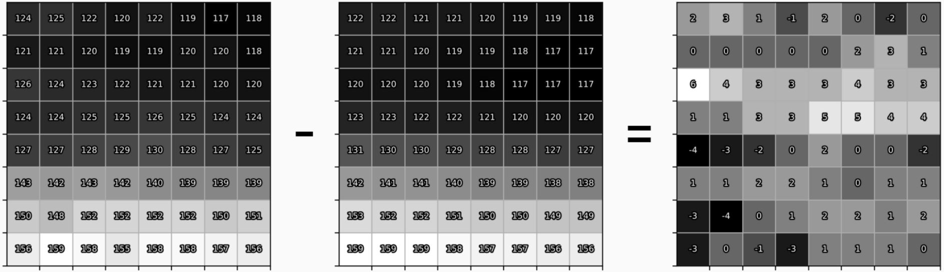


Figure 13: The difference (right) between the original block (left) and the result from the JPEG compression and decompression (middle).

- The differences between the original block and the restored are small.

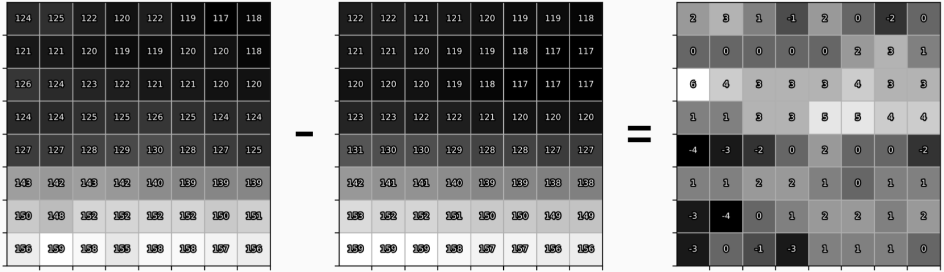


Figure 13: The difference (right) between the original block (left) and the result from the JPEG compression and decompression (middle).

- The differences between the original block and the restored are small.
- But they are, however, not zero.

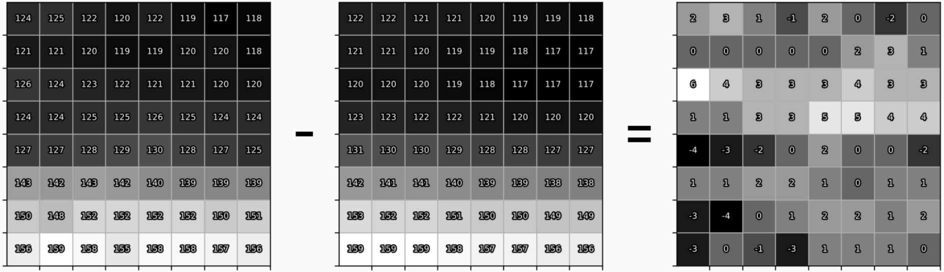


Figure 13: The difference (right) between the original block (left) and the result from the JPEG compression and decompression (middle).

- The differences between the original block and the restored are small.
- But they are, however, not zero.
- The error is different on neighbouring pixels.

124	125	122	120	122	119	117	118
121	121	120	119	119	120	120	118
126	124	123	122	121	121	120	120
124	124	125	125	126	125	124	124
127	127	128	129	130	128	127	125
143	142	143	142	140	139	139	139
150	148	152	152	152	152	150	151
156	159	158	155	158	158	157	156

−

122	122	121	121	120	119	119	118
121	121	120	119	119	118	117	117
120	120	120	119	118	117	117	117
123	123	122	122	121	120	120	120
131	130	130	129	128	128	127	127
142	141	141	140	139	139	138	138
153	152	152	151	150	150	149	149
159	159	159	158	157	157	156	156

=

2	3	1	-1	2	0	-2	0
0	0	0	0	0	2	3	1
6	4	3	3	3	4	3	3
1	1	3	3	5	5	4	4
-4	-3	-2	0	2	0	0	-2
1	1	2	2	1	0	1	1
-3	-4	0	1	2	2	1	2
-3	0	-1	-3	1	1	1	0

Figure 13: The difference (right) between the original block (left) and the result from the JPEG compression and decompression (middle).

- The differences between the original block and the restored are small.
- But they are, however, not zero.
- The error is different on neighbouring pixels.
- This is especially true if the neighbouring pixels belong to different blocks.

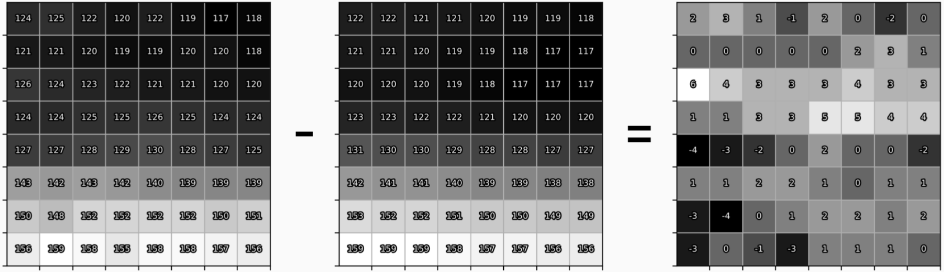
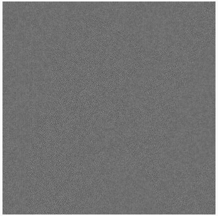


Figure 13: The difference (right) between the original block (left) and the result from the JPEG compression and decompression (middle).

- The differences between the original block and the restored are small.
- But they are, however, not zero.
- The error is different on neighbouring pixels.
- This is especially true if the neighbouring pixels belong to different blocks.
- The JPEG compression/decompression can therefore introduce *block artifacts*, which are block patterns in the reconstructed image (due to these different errors).

BLOCK ARTIFACTS AND COMPRESSION RATE

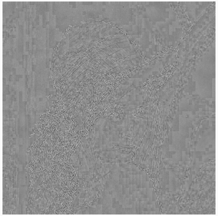
(a) Compressed



(b) Difference

(c) Detail

(d) Compressed



(e) Difference

(f) Detail

Figure 14: Top row: compression rate = 12.5. Bottom row: compression rate = 32.7

MORPHOLOGY ON BINARY IMAGES

BINARY IMAGES AS SETS

- Let $f : \Omega \to \{0, 1\}$ be an image.

BINARY IMAGES AS SETS

- Let $f : \Omega \to \{0, 1\}$ be an image.
- In the case of a 2D image, $\Omega \subset \mathbb{Z}^2$, and every pixel (x, y) in f is in the set Ω, written $(x, y) \in \Omega$.

BINARY IMAGES AS SETS

- Let $f : \Omega \to \{0, 1\}$ be an image.
- In the case of a 2D image, $\Omega \subset \mathbb{Z}^2$, and every pixel (x, y) in f is in the set Ω, written $(x, y) \in \Omega$.
- A binary image can be described by the set of foreground pixels, which is a subset of Ω.

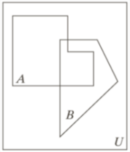
BINARY IMAGES AS SETS

- Let $f : \Omega \to \{0, 1\}$ be an image.
- In the case of a 2D image, $\Omega \subset \mathbb{Z}^2$, and every pixel (x, y) in f is in the set Ω, written $(x, y) \in \Omega$.
- A binary image can be described by the set of foreground pixels, which is a subset of Ω.
- Therefore, we might use notation and terms from set theory when describing binary images, and operations acting on them.

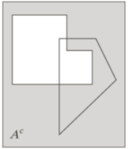
BINARY IMAGES AS SETS

- Let $f : \Omega \to \{0, 1\}$ be an image.
- In the case of a 2D image, $\Omega \subset \mathbb{Z}^2$, and every pixel (x, y) in f is in the set Ω, written $(x, y) \in \Omega$.
- A binary image can be described by the set of foreground pixels, which is a subset of Ω.
- Therefore, we might use notation and terms from set theory when describing binary images, and operations acting on them.
- The *complement* of a binary image f is

$$
\begin{aligned}
h(x, y) &= f^c(x, y) \\
&= \begin{cases} 0 & \text{if } f(x, y) = 1, \\ 1 & \text{if } f(x, y) = 0. \end{cases}
\end{aligned}
$$



(a) Contours of sets A and B inside a set U



(b) The complement A^C (gray) of A (white)

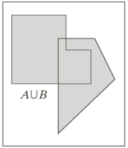
Figure 15: Set illustrations, gray is foreground, white is background.

- The *union* of two binary images f and g is

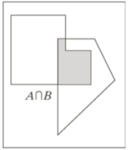
$$h(x,y) = (f \cup g)(x,y)$$
$$= \begin{cases} 1 & \text{if } f(x,y) = 1 \text{ or } g(x,y) = 1, \\ 0 & \text{otherwise.} \end{cases}$$

- The *intersection* of two binary images f and g is

$$h(x,y) = (f \cap g)(x,y)$$
$$= \begin{cases} 1 & \text{if } f(x,y) = 1 \text{ and } g(x,y) = 1, \\ 0 & \text{otherwise.} \end{cases}$$



(a) Union of A and B



(b) Intersection of A and B

Figure 16: Set illustrations, gray is foreground, white is background.

STRUCTURING ELEMENT

- A *structuring element* in morphology is used to determine the acting range of the operations.

STRUCTURING ELEMENT

- A *structuring element* in morphology is used to determine the acting range of the operations.
- It is typically defined as a binary matrix where pixels valued 0 are not acting, and pixels valued 1 are acting.

STRUCTURING ELEMENT

- A *structuring element* in morphology is used to determine the acting range of the operations.
- It is typically defined as a binary matrix where pixels valued 0 are not acting, and pixels valued 1 are acting.
- When hovering the structuring element over an image, we have three possible scenario for the structuring element (or really the location of the 1's in the structuring element):
  - It is *not overlapping* the image foreground (a *miss*).
  - It is *partly overlapping* the image foreground (a *hit*).
  - It is *fully overlapping* the image foreground (it *fits*).

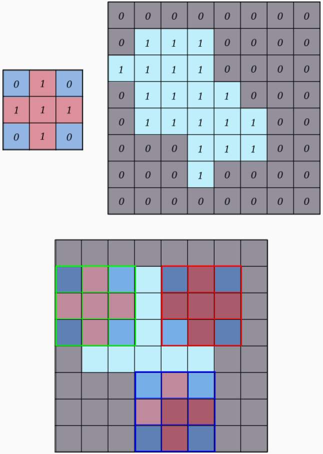


Figure 17: A structuring element (top left) and a binary image (top right). Bottom: the red misses, the blue hits, and the green fits.

ORIGIN OF STRUCTURING ELEMENT

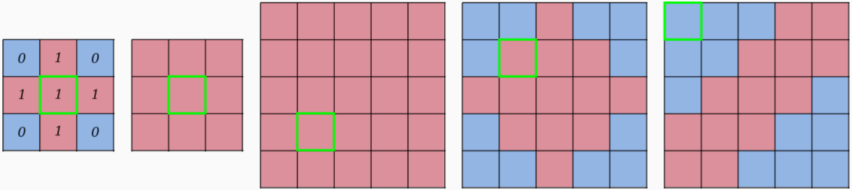


Figure 18: Some structuring elements. The red pixel contour highlights the origin.

- The structuring element can have different shapes and sizes.

ORIGIN OF STRUCTURING ELEMENT

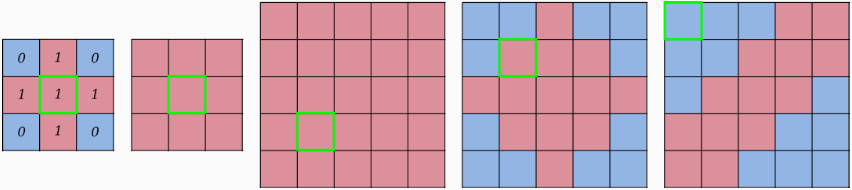


Figure 18: Some structuring elements. The red pixel contour highlights the origin.

- The structuring element can have different shapes and sizes.
- We need to determine an *origin*.
    - This origin denotes the pixel that (possibly) changes value in the result image.
    - The origin *may* lay outside the structuring element.
    - The origin should be highlighted, e.g. with a drawed square.
    - We will assume the origin to be at the center pixel of the structuring element, and not specify the location unless this is the case.

EROSION

- Let $f : \Omega_f \to \{0, 1\}$ be a $2D$ binary image.

EROSION

- Let $f : \Omega_f \to \{0, 1\}$ be a $2D$ binary image.
- Let $s : \Omega_s \to \{0, 1\}$ be a $2D$ binary structuring element.

EROSION

- Let $f : \Omega_f \to \{0, 1\}$ be a $2D$ binary image.
- Let $s : \Omega_s \to \{0, 1\}$ be a $2D$ binary structuring element.
- Let $x, y \in \mathbb{Z}^2$ be $2D$ points for notational convenience.

EROSION

- Let $f : \Omega_f \to \{0, 1\}$ be a $2D$ binary image.
- Let $s : \Omega_s \to \{0, 1\}$ be a $2D$ binary structuring element.
- Let $x, y \in \mathbb{Z}^2$ be $2D$ points for notational convenience.
- We then have 3 equivalent definitions of erosion.
  - The one mentioned at the beginning of the lecture

$$(f \ominus s)(x) = \min_{\substack{y \in \Omega_s^+ \\ x+y \in \Omega_f}} \{f(x+y)\}, \tag{5}$$

    where Ω_s^+ is the subset of Ω_s with foreground pixels.
  - Place the s such that its origin overlaps with x, then

$$(f \ominus s)(x) = \begin{cases} 1 & \text{if } s \text{ fits in } f, \\ 0 & \text{otherwise.} \end{cases} \tag{6}$$

  - Let $F(g)$ be the set of all foreground pixels of a binary image g, then

$$F(f \ominus s) = \{x \in \Omega_f : F(x + \Omega_s^+) \subseteq F(f)\}. \tag{7}$$

    Note that the $F()$ is often ommitted, as we often use set operations in binary morphology.

EROSION EXAMPLES

0	1	0	0	0	0	0	1	1	0	0
1	1	1	0	0	0	1	1	1	1	0
0	1	1	1	0	1	1	1	1	0	0
0	0	1	1	1	1	1	1	0	0	0
0	0	0	1	1	1	1	1	1	0	0
0	0	1	1	1	1	0	1	1	1	0
0	1	1	1	1	0	0	0	1	1	1
0	0	1	1	0	0	0	0	0	1	0

(a) f

1	1	1
1	1	1
1	1	1

(b) s_1

0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0

(c) $f \ominus s_1$

0	1	0	0	0	0	0	1	1	0	0
1	1	1	0	0	0	1	1	1	1	0
0	1	1	1	0	1	1	1	1	0	0
0	0	1	1	1	1	1	1	0	0	0
0	0	0	1	1	1	1	1	1	0	0
0	0	1	1	1	1	0	1	1	1	0
0	1	1	1	1	0	0	0	1	1	1
0	0	1	1	0	0	0	0	0	1	0

(a) f

0	1	0
1	1	1
0	1	0

(b) s_2

0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	1	1	0	0
0	0	1	0	0	0	1	1	0	0	0
0	0	0	1	0	1	1	0	0	0	0
0	0	0	0	1	1	0	1	0	0	0
0	0	0	1	1	0	0	0	1	0	0
0	0	1	1	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	0	0

(c) $f \ominus s_2$

EDGE DETECTION WITH EROSION

- Erosion removes pixels along the boundary of the foreground object.

EDGE DETECTION WITH EROSION

- Erosion removes pixels along the boundary of the foreground object.
- We can locate the edges by subtracting the eroded image from the original

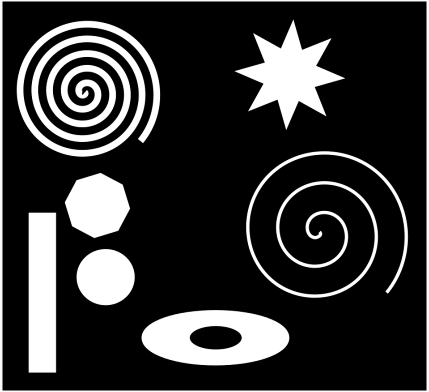
$$g = f - (f \ominus s)$$

EDGE DETECTION WITH EROSION

- Erosion removes pixels along the boundary of the foreground object.
- We can locate the edges by subtracting the eroded image from the original

$$g = f - (f \ominus s)$$

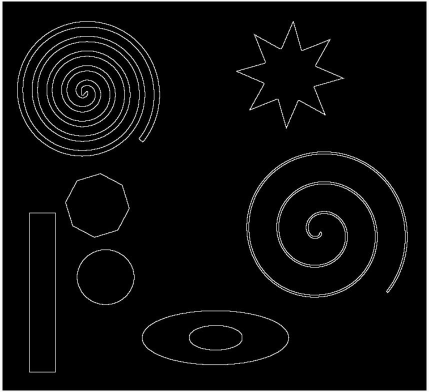
- The shape of the structuring element determines the connectivety of the countour. That is, if the contour is connected using a 4 or 8 connected neighbourhood.



(a) f

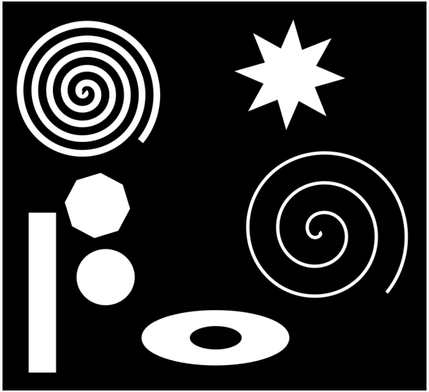
1	1	1
1	1	1
1	1	1

(b) s



(c) $f - (f \ominus s)$

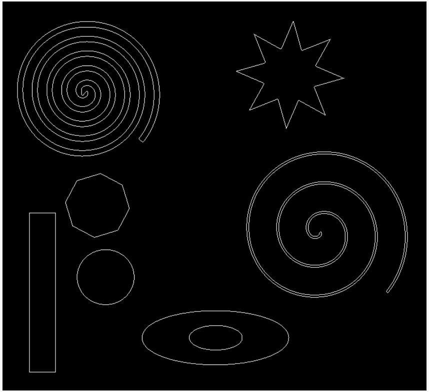
EDGE DETECTION WITH EROSION: EXAMPLES



(a) f

0	1	0
1	1	1
0	1	0

(b) s



(c) $f - (f \ominus s)$

DILATION

- Let $f : \Omega_f \to \{0, 1\}$ be a $2D$ binary image.

DILATION

- Let $f : \Omega_f \to \{0, 1\}$ be a $2D$ binary image.
- Let $s : \Omega_s \to \{0, 1\}$ be a $2D$ binary structuring element.

- Let $f : \Omega_f \to \{0, 1\}$ be a $2D$ binary image.
- Let $s : \Omega_s \to \{0, 1\}$ be a $2D$ binary structuring element.
- Let $x, y \in \mathbb{Z}^2$ be $2D$ points for notational convenience.

DILATION

- Let $f : \Omega_f \to \{0,1\}$ be a $2D$ binary image.
- Let $s : \Omega_s \to \{0,1\}$ be a $2D$ binary structuring element.
- Let $x, y \in \mathbb{Z}^2$ be $2D$ points for notational convenience.
- We then have 3 equivalent definitions of dilation.
    - The one mentioned at the beginning of the lecture

$$(f \oplus s)(x) = \max_{\substack{y \in \Omega_s^+ \\ x - y \in \Omega_f}} \{f(x-y)\}, \tag{8}$$

      where Ω_s^+ is the subset of Ω_s with foreground pixels.
    - Place the s such that its origin overlaps with x, then

$$(f \oplus s)(x) = \begin{cases} 1 & \text{if } \tilde{s} \text{ hits } f, \\ 0 & \text{otherwise.} \end{cases} \tag{9}$$

      where $\tilde{s}$ is s rotated 180 degrees.
    - Let $F(g)$ be the set of all foreground pixels of a binary image g, then

$$F(f \oplus s) = \{x \in \Omega_f : F(x + \Omega_s^+) \cap F(f) \neq \emptyset\} \tag{10}$$

DILATION EXAMPLES

0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	1	1	0	0
0	0	1	0	0	0	1	1	0	0	0
0	0	0	1	0	1	1	0	0	0	0
0	0	0	0	1	1	0	1	0	0	0
0	0	0	1	1	0	0	0	1	0	0
0	0	1	1	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	0	0

(a) f

1	1	1
1	1	1
1	1	1

(b) s_1

1	1	1	0	0	0	1	1	1	1	0
1	1	1	1	0	1	1	1	1	1	0
1	1	1	1	1	1	1	1	1	1	0
0	1	1	1	1	1	1	1	1	0	0
0	0	1	1	1	1	1	1	1	1	0
0	1	1	1	1	1	1	1	1	1	1
0	1	1	1	1	1	0	1	1	1	1
0	1	1	1	1	0	0	0	1	1	1

(c) $f \oplus s_1$

0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	1	1	0	0
0	0	1	0	0	0	1	1	0	0	0
0	0	0	1	0	1	1	0	0	0	0
0	0	0	0	1	1	0	1	0	0	0
0	0	0	1	1	0	0	0	1	0	0
0	0	1	1	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	0	0

(a) f

0	1	0
1	1	1
0	1	0

(b) s_2

0	1	0	0	0	0	0	1	1	0	0
1	1	1	0	0	0	1	1	1	1	0
0	1	1	1	0	1	1	1	1	0	0
0	0	1	1	1	1	1	1	0	0	0
0	0	0	1	1	1	1	1	1	0	0
0	0	1	1	1	1	0	1	1	1	0
0	1	1	1	1	0	0	0	1	1	1
0	0	1	1	0	0	0	0	0	1	0

(c) $f \oplus s_2$

- Dilation adds pixels along the boundary of the foreground object.

EDGE DETECTION WITH DILATION

- Dilation adds pixels along the boundary of the foreground object.
- We can locate the edges by subtracting the original image from the dilated image

$$g = (f \oplus s) - f$$

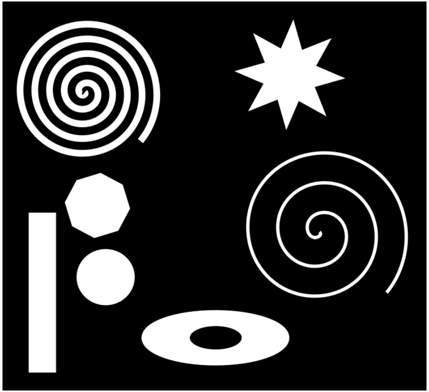
EDGE DETECTION WITH DILATION

- Dilation adds pixels along the boundary of the foreground object.
- We can locate the edges by subtracting the original image from the dilated image

$$g = (f \oplus s) - f$$

- The shape of the structuring element determines the connectivety of the countour. That is, if the contour is connected using a 4 or 8 connected neighbourhood.

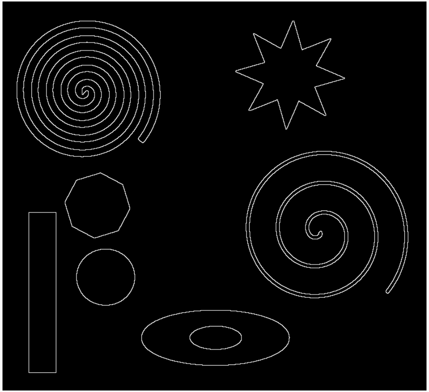
EDGE DETECTION WITH DILATION: EXAMPLES



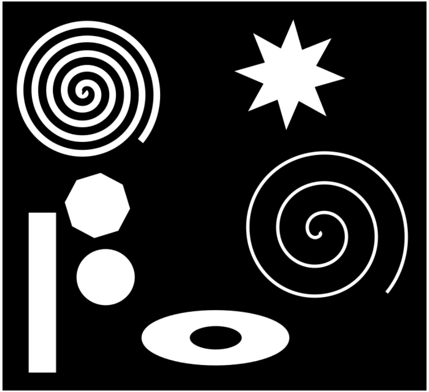
(a) f

1	1	1
1	1	1
1	1	1

(b) s



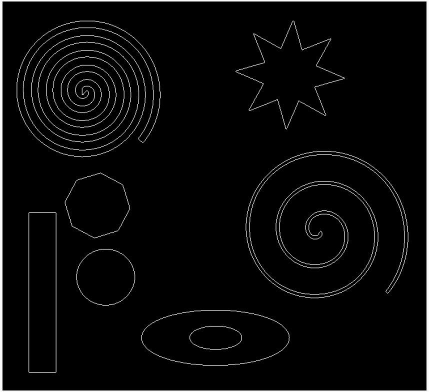
(c) $(f \oplus s) - f$



(a) f

0	1	0
1	1	1
0	1	0

(b) s



(c) $(f \oplus s) - f$

DUALITY BETWEEN DILATION AND EROSION

- Dilation and erosion are *dual* operations w.r.t. complements and reflections (180 degree rotation). That is, erosion and dilation can be expressed as

$$f \oplus s = (f^c \ominus s)^c$$
$$f \ominus s = (f^c \oplus s)^c$$

- This means that dilation and erosion can be performed by the same procedure, given that we can rotate the structuring element and find the complement of the binary image.

0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	0	0	1	1	1	0	0
0	0	0	1	1	1	0	0	1	0	0
0	0	0	1	0	0	0	0	1	0	0
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	0
0	0	0	0	0	0	1	0	1	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0

(a) f

0	1	0
1	1	1
0	1	0

(b) s

0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	0	0	1	1	1	0	0
0	1	1	1	1	1	1	1	1	1	0
0	0	1	1	1	1	1	1	1	1	0
0	0	1	1	1	1	0	1	1	1	0
0	0	0	1	1	1	0	1	1	1	0
0	0	0	0	1	1	1	1	1	1	0
0	0	0	0	0	1	1	1	1	1	0
0	0	0	0	0	0	1	0	1	0	0
0	0	0	0	0	0	0	0	0	0	0

(c) $f \oplus s$

1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1
1	1	0	0	1	1	0	0	0	1	1
1	1	1	0	0	0	1	1	0	1	1
1	1	1	0	1	1	1	1	0	1	1
1	1	1	1	0	1	1	1	0	1	1
1	1	1	1	1	0	1	1	0	1	1
1	1	1	1	1	1	0	1	0	1	1
1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1

(a) f^c

0	1	0
1	1	1
0	1	0

(b) s

1	1	1	1	1	1	1	1	1	1	1
1	1	0	0	1	1	0	0	0	1	1
1	0	0	0	0	0	0	0	0	0	1
1	1	0	0	0	0	0	0	0	0	1
1	1	0	0	0	0	1	0	0	0	1
1	1	1	0	0	0	1	0	0	0	1
1	1	1	1	0	0	0	0	0	0	1
1	1	1	1	1	0	0	0	0	0	1
1	1	1	1	1	1	0	1	0	1	1
1	1	1	1	1	1	1	1	1	1	1

(c) $f^c \ominus s$

MORPHOLOGICAL OPENING

- Erosion of an image removes all regions that cannot fit the structuring element, and shrinks all other regions.

MORPHOLOGICAL OPENING

- Erosion of an image removes all regions that cannot fit the structuring element, and shrinks all other regions.
- We can then dilate the result of the erosion, with this
  - the regions that where shrinked are (approximately) restored.
  - the regions too small to survive an erosion, are not restored.

MORPHOLOGICAL OPENING

- Erosion of an image removes all regions that cannot fit the structuring element, and shrinks all other regions.
- We can then dilate the result of the erosion, with this
  - the regions that where shrinked are (approximately) restored.
  - the regions too small to survive an erosion, are not restored.
- This is *morphological opening*

$$f \circ s = (f \ominus s) \oplus s$$

MORPHOLOGICAL OPENING

- Erosion of an image removes all regions that cannot fit the structuring element, and shrinks all other regions.
- We can then dilate the result of the erosion, with this
  - the regions that where shrinked are (approximately) restored.
  - the regions too small to survive an erosion, are not restored.
- This is *morphological opening*

$$f \circ s = (f \ominus s) \oplus s$$

- The name stems from that this operation can provide an opening (a space) between regions that are connected through thin ”bridges”, almost without affecting the original shape of the larger regions.

MORPHOLOGICAL OPENING

- Erosion of an image removes all regions that cannot fit the structuring element, and shrinks all other regions.
- We can then dilate the result of the erosion, with this
  - the regions that where shrinked are (approximately) restored.
  - the regions too small to survive an erosion, are not restored.
- This is *morphological opening*

$$f \circ s = (f \ominus s) \oplus s$$

- The name stems from that this operation can provide an opening (a space) between regions that are connected through thin ”bridges”, almost without affecting the original shape of the larger regions.
- Using erosion only will also open these bridges, but the shape of the larger regions is also altered.
- The size and shape of the structuring element is vital.

EXAMPLE: OPENING

0	0	0	0	0	0	0	0	0	0	0
0	1	1	1	0	0	0	0	0	0	0
0	1	1	1	1	0	0	1	1	0	0
0	1	1	1	1	0	0	0	1	0	0
0	1	1	1	1	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	1	1	1	0
0	0	1	0	0	0	0	0	0	0	0
0	1	1	1	0	0	0	0	0	0	0
0	1	1	1	1	0	0	1	1	0	0
0	1	1	1	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0

(a) f

1	1	1
1	1	1
1	1	1

(b) s

0	0	0	0	0	0	0	0	0	0	0
0	1	1	1	0	0	0	0	0	0	0
0	1	1	1	1	0	0	0	0	0	0
0	1	1	1	1	0	0	0	0	0	0
0	1	1	1	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
0	1	1	1	0	0	0	0	0	0	0
0	1	1	1	0	0	0	0	0	0	0
0	1	1	1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0

(c) $f \circ s$

Figure 29: Morphological opening of f with s.

MORPHOLOGICAL CLOSING

- Dilation of an image expands foreground regions and fills small (relative to the structuring element) holes in the foreground.

MORPHOLOGICAL CLOSING

- Dilation of an image expands foreground regions and fills small (relative to the structuring element) holes in the foreground.
- We can then erode the result of the dilation, then
  - the regions that where expanded are (approximately) restored.
  - the holes that where filled, are not opened again.

MORPHOLOGICAL CLOSING

- Dilation of an image expands foreground regions and fills small (relative to the structuring element) holes in the foreground.
- We can then erode the result of the dilation, then
  - the regions that where expanded are (approximately) restored.
  - the holes that where filled, are not opened again.
- This is *morphological closing*

$$f \bullet s = (f \oplus s) \ominus s$$

MORPHOLOGICAL CLOSING

- Dilation of an image expands foreground regions and fills small (relative to the structuring element) holes in the foreground.
- We can then erode the result of the dilation, then
  - the regions that where expanded are (approximately) restored.
  - the holes that where filled, are not opened again.
- This is *morphological closing*

$$f \bullet s = (f \oplus s) \ominus s$$

- The name stems from that this operation can close small gaps between foreground regions, without altering the larger foreground shapes too much.
- Using dilation only will also close these gaps, but the shape of the larger regions is also altered.

MORPHOLOGICAL CLOSING

- Dilation of an image expands foreground regions and fills small (relative to the structuring element) holes in the foreground.
- We can then erode the result of the dilation, then
  - the regions that where expanded are (approximately) restored.
  - the holes that where filled, are not opened again.
- This is *morphological closing*

$$f \bullet s = (f \oplus s) \ominus s$$

- The name stems from that this operation can close small gaps between foreground regions, without altering the larger foreground shapes too much.
- Using dilation only will also close these gaps, but the shape of the larger regions is also altered.
- The size and shape of the structuring element is vital.

0	0	0	0	0	0	0	0	0	0	0
0	1	1	1	0	0	0	0	0	0	0
0	1	1	1	1	0	0	1	1	0	0
0	1	1	1	1	0	0	0	1	0	0
0	1	1	1	1	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	1	1	1	0
0	0	1	0	0	0	0	0	0	0	0
0	1	1	1	0	0	0	0	0	0	0
0	1	1	1	1	0	0	1	1	0	0
0	1	1	1	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0

(a) f

1	1	1
1	1	1
1	1	1

(b) s

0	0	0	0	0	0	0	0	0	0	0
0	1	1	1	0	0	0	0	0	0	0
0	1	1	1	1	1	1	1	1	0	0
0	1	1	1	1	0	0	0	1	0	0
0	1	1	1	1	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	1	1	1	0
0	0	1	0	0	0	0	1	1	0	0
0	1	1	1	0	0	0	1	1	0	0
0	1	1	1	1	1	1	1	1	0	0
0	1	1	1	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0

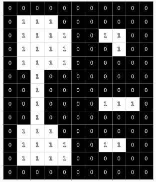
(c) $f \bullet s$

Figure 30: Morphological opening of f with s.

- Opening and closing are *dual operations* w.r.t. complements and rotation

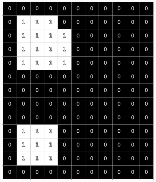
$$f \circ s = (f^c \bullet s)^c$$
$$f \bullet s = (f^c \circ s)^c$$

- This means that closing can be performed by complementing the image, opening the complement by the rotated structuring element, and complement the result. The corresponding is true for opening.

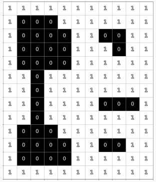


(a) f

(b) s

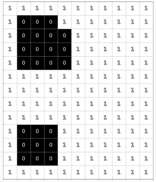


(c) $f \circ s$



(a) f^c

(b) s



(c) $f^c \bullet s$

HIT-OR-MISS TRANSFORMATION

- Let f be a binary image as usual, but define s to be a tuple of two structuring elements $s = (s^{hit}, s^{miss})$.

- Let f be a binary image as usual, but define s to be a tuple of two structuring elements $s = (s^{hit}, s^{miss})$.
- The hit-or-miss transformation is then defined as

$$f \circledast s = (f \ominus s^{hit}) \cap (f^c \ominus s^{miss}),$$

HIT-OR-MISS TRANSFORMATION

- Let f be a binary image as usual, but define s to be a tuple of two structuring elements $s = (s^{hit}, s^{miss})$.
- The hit-or-miss transformation is then defined as

$$f \circledast s = (f \ominus s^{hit}) \cap (f^c \ominus s^{miss}),$$

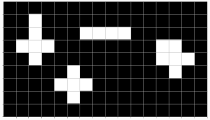
- A foreground pixel in the out-image is only achieved if
  - s^{hit} fits the foreground around the pixel, and
  - s^{miss} fits the background around the pixel.

- Let f be a binary image as usual, but define s to be a tuple of two structuring elements $s = (s^{hit}, s^{miss})$.
- The hit-or-miss transformation is then defined as

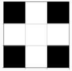
$$f \circledast s = (f \ominus s^{hit}) \cap (f^c \ominus s^{miss}),$$

- A foreground pixel in the out-image is only achieved if
  - s^{hit} fits the foreground around the pixel, and
  - s^{miss} fits the background around the pixel.
- This can be used in several applications, including
  - finding certain patterns in an image,
  - removing single pixels,
  - thinning and thickening foreground regions.

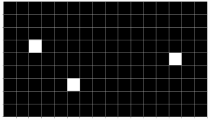
HIT-OR-MISS EXAMPLE



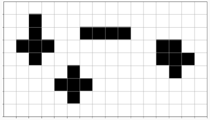
(a) f



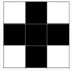
(b) s^{hit}



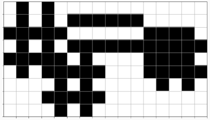
(c) $f \ominus s^{hit}$



(a) f^c



(b) s^{miss}



(c) $f^c \ominus s^{miss}$

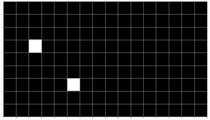


Figure 35: $f \circledast s$

MORPHOLOGICAL THINNING

- Morphological thinning of an image f with a structuring element tuple s, is defined as

$$
\begin{aligned}
f \otimes s &= f \setminus (f \circledast s) \\
&= f \cap (f \circledast s)^c
\end{aligned}
$$

MORPHOLOGICAL THINNING

- Morphological thinning of an image f with a structuring element tuple s, is defined as

$$\begin{aligned} f \otimes s &= f \setminus (f \circledast s) \\ &= f \cap (f \circledast s)^c \end{aligned}$$

- In order to thin a foreground region, we perform a sequential thinning with multiple structuring elements $s_1, \ldots, s_8$, which, when defined with the hit-or-miss notation above are

$$s_1^{hit} = \begin{array}{|c|c|c|} \hline 0 & 0 & 0 \\ \hline 0 & 1 & 0 \\ \hline 1 & 1 & 1 \\ \hline \end{array}, \quad s_1^{miss} = \begin{array}{|c|c|c|} \hline 1 & 1 & 1 \\ \hline 0 & 0 & 0 \\ \hline 0 & 0 & 0 \\ \hline \end{array}.$$

$$s_2^{hit} = \begin{array}{|c|c|c|} \hline 0 & 0 & 0 \\ \hline 1 & 1 & 0 \\ \hline 1 & 1 & 0 \\ \hline \end{array}, \quad s_2^{miss} = \begin{array}{|c|c|c|} \hline 0 & 1 & 1 \\ \hline 0 & 0 & 1 \\ \hline 0 & 0 & 0 \\ \hline \end{array}.$$

Following this pattern, where s_{i+1} is rotated clockwise w.r.t. s_i, we continue this until

$$s_8^{hit} = \begin{array}{|c|c|c|} \hline 0 & 0 & 0 \\ \hline 0 & 1 & 1 \\ \hline 0 & 1 & 1 \\ \hline \end{array}, \quad s_8^{miss} = \begin{array}{|c|c|c|} \hline 1 & 1 & 0 \\ \hline 1 & 0 & 0 \\ \hline 0 & 0 & 0 \\ \hline \end{array}.$$

- With these previously defined structuring elements, we apply the iteration

$$
\begin{aligned}
d_0 &= f \\
d_k &= d_{k-1} \otimes \{s_1, \cdots, s_8\} \\
&= (\cdots((d_{k-1} \otimes s_1) \otimes s_2) \cdots) \otimes s_8
\end{aligned}
$$

  for K iterations until $d_K = d_{K-1}$ and we terminate with d_K as the result of the thinning.

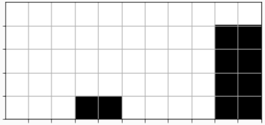
- With these previously defined structuring elements, we apply the iteration

$$
\begin{aligned}
d_0 &= f \\
d_k &= d_{k-1} \otimes \{s_1, \cdots, s_8\} \\
&= (\cdots((d_{k-1} \otimes s_1) \otimes s_2) \cdots) \otimes s_8
\end{aligned}
$$

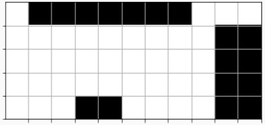
  for K iterations until $d_K = d_{K-1}$ and we terminate with d_K as the result of the thinning.
- In the same manner, we define the dual operator *thickening*

$$f \odot s = f \cup (f \circledast s),$$

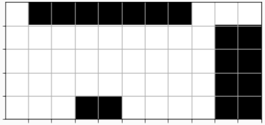
  which also can be used in a sequential manner analogous to thinning.



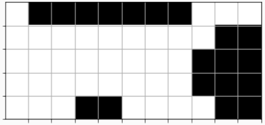
(a) $d_0 = f$



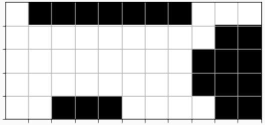
(b) $d_{01} = d_0 \setminus (d_0 \circledast s_1)$



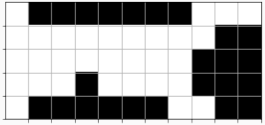
(c) $d_{02} = d_{01} \setminus (d_{01} \circledast s_2)$



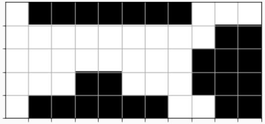
(d) $d_{03} = d_{02} \setminus (d_{02} \circledast s_3)$



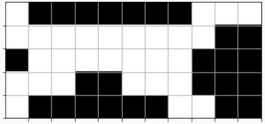
(e) $d_{04} = d_{03} \setminus (d_{03} \circledast s_4)$



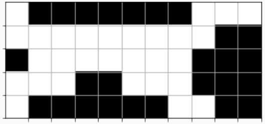
(f) $d_{05} = d_{04} \setminus (d_{04} \circledast s_5)$



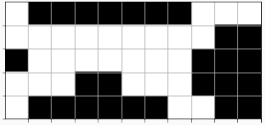
(g) $d_{06} = d_{05} \setminus (d_{05} \circledast s_6)$



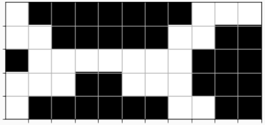
(h) $d_{07} = d_{06} \setminus (d_{06} \circledast s_7)$



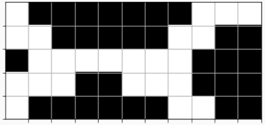
(i) $d_{08} = d_{07} \setminus (d_{07} \circledast s_8)$



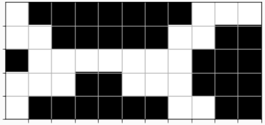
(a) $d_1 = d_0$



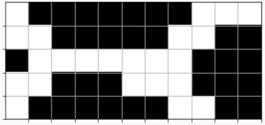
(b) $d_{11} = d_1 \setminus (d_1 \circledast s_1)$



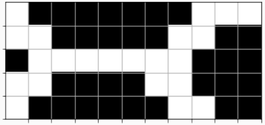
(c) $d_{12} = d_{11} \setminus (d_{11} \circledast s_2)$



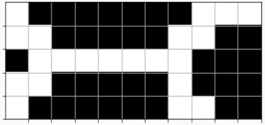
(d) $d_{13} = d_{12} \setminus (d_{12} \circledast s_3)$



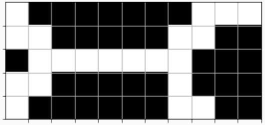
(e) $d_{14} = d_{13} \setminus (d_{13} \circledast s_4)$



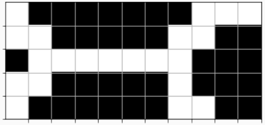
(f) $d_{15} = d_{14} \setminus (d_{14} \circledast s_5)$



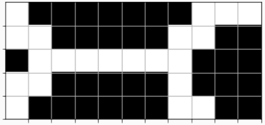
(g) $d_{16} = d_{15} \setminus (d_{15} \circledast s_6)$



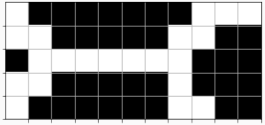
(h) $d_{17} = d_{16} \setminus (d_{16} \circledast s_7)$



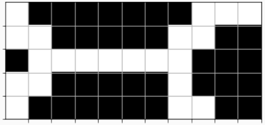
(i) $d_{18} = d_{17} \setminus (d_{17} \circledast s_8)$



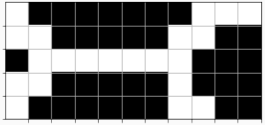
(a) $d_2 = d_1$



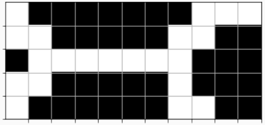
(b) $d_{21} = d_2 \setminus (d_2 \circledast s_1)$



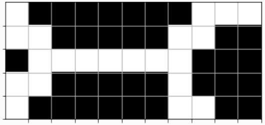
(c) $d_{22} = d_{21} \setminus (d_{21} \circledast s_2)$



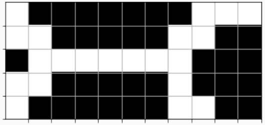
(d) $d_{23} = d_{22} \setminus (d_{22} \circledast s_3)$



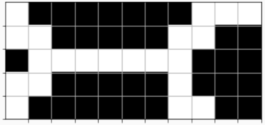
(e) $d_{24} = d_{23} \setminus (d_{23} \circledast s_4)$



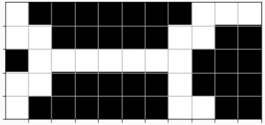
(f) $d_{25} = d_{24} \setminus (d_{24} \circledast s_5)$



(g) $d_{26} = d_{25} \setminus (d_{25} \circledast s_6)$



(h) $d_{27} = d_{26} \setminus (d_{26} \circledast s_7)$



(i) $d_{28} = d_{27} \setminus (d_{27} \circledast s_8)$

SUMMARY

- Coding and compression
  - Information theory
  - Shannon-Fano coding
  - Huffman coding
  - Arithmetic coding
  - Difference transform
  - Run-length coding
  - Lempel-Ziv-Welch coding
  - Lossy JPEG compression
- Binary morphology
  - Fundamentals: Structuring element, erosion and dilation
  - Opening and closing
  - Hit-or-miss transform
  - Morphological thinning

Questions?