Introduction to Image Processing

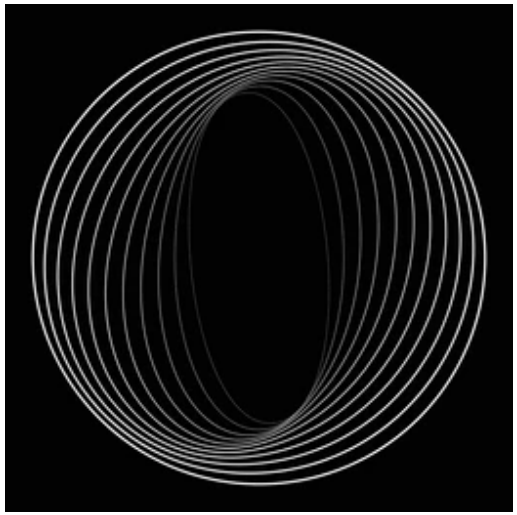
Lecturer:
Dr. mat. nat. Christian Kehl

Week 6 – Image Compression, Basic Segmentation and Image-Feature Description

December 12, 2024

Basic Compressing Image Information

Why 'compression' ? Data redundancies



- *Coding redundancy*: length of the code words (e.g., 8-bit codes for grey value images) is larger than needed.
- *Spatial/temporal redundancy*: correlation between pixels (in space or time) unused in the representation.
- *Irrelevant information*: information that exceed perception of the human visual system.

Fundamentals

Let b and b' $(b > b')$ be the number of bits in two representations of the same information.

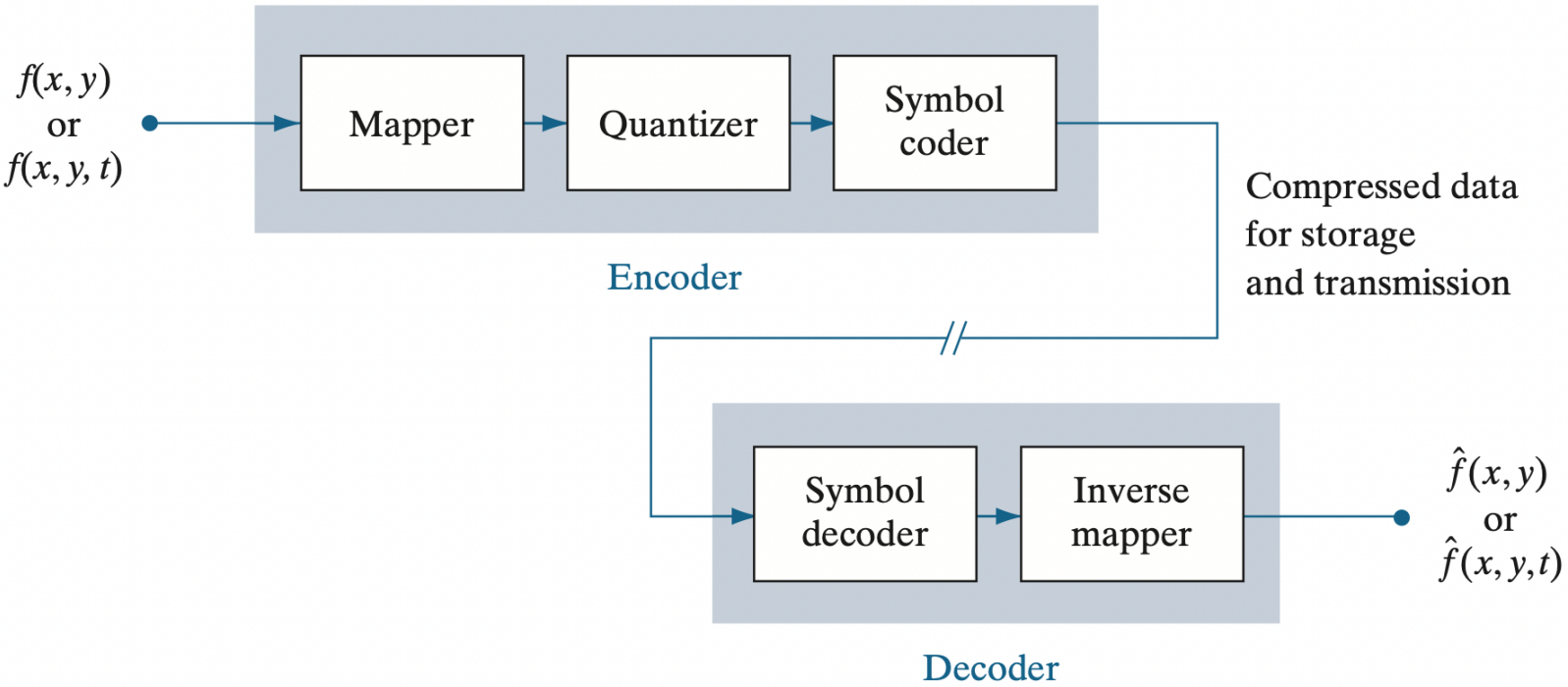
- *Compression ratio*:

$$C = \frac{b}{b'}$$

- *Relative data redundancy*:

$$R = 1 - \frac{1}{C}$$

Image compression models



- *Mapper*: transform to reduce redundancy
- *Quantizer*: reduce accuracy of mapper output (lossy coding)
- *Symbol coder*: generate fixed/variable length code

Fidelity criteria

Input image $f(x, y)$, approximation $\hat{f}(x, y)$, both of size $M \times N$.

- *Root-mean square error*:

$$e_{\text{rms}} = \left[\frac{1}{MN} \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} \left(\hat{f}(x, y) - f(x, y) \right)^2 \right]^{\frac{1}{2}}$$

- *Mean-square signal-to-noise ratio*:

$$\text{SNR}_{\text{rms}} = \frac{\sum_{x=0}^{M-1} \sum_{y=0}^{N-1} \hat{f}(x, y)^2}{\sum_{x=0}^{M-1} \sum_{y=0}^{N-1} \left(\hat{f}(x, y) - f(x, y) \right)^2}$$

- generally: image comparison metrics (see *lecture 1*)
- *Subjective rating scales*

Compression types &methods

Types:

- lossless compression: image compression without reducing information
- lossy compression: image compression with-/by cutting information

Compression types &methods

Types:

- lossless compression: image compression without reducing information
- lossy compression: image compression with-/by cutting information

Methods:

- *Reducing coding redundancy*: **arithmetic coding**, **Huffman coding**, etc.

Compression types &methods

Types:

- lossless compression: image compression without reducing information
- lossy compression: image compression with-/by cutting information

Methods:

- *Reducing coding redundancy*: **arithmetic coding**, **Huffman coding**, etc.
- *Reducing spatial/temporal redundancy*: LZW (Lempel-Ziv-Welch) coding, **run-length coding**, transform coding, predictive coding

Compression types &methods

Types:

- lossless compression: image compression without reducing information
- lossy compression: image compression with-/by cutting information

Methods:

- *Reducing coding redundancy*: **arithmetic coding**, **Huffman coding**, etc.
- *Reducing spatial/temporal redundancy*: LZW (Lempel-Ziv-Welch) coding, **run-length coding**, transform coding, predictive coding
- *Perceptual subsampling*: chroma subsampling

Compression types &methods

Types:

- lossless compression: image compression without reducing information
- lossy compression: image compression with-/by cutting information

Methods:

- *Reducing coding redundancy*: **arithmetic coding**, **Huffman coding**, etc.
- *Reducing spatial/temporal redundancy*: LZW (Lempel-Ziv-Welch) coding, **run-length coding**, transform coding, predictive coding
- *Perceptual subsampling*: chroma subsampling
- *Region-/resolution subsampling*: **block-transform coding**, bit-plane coding, image pyramids, etc.

Compressing Images by Reducing Code Redundancy

Coding redundancy

- Let n_{r_k} be the number of times that intensity $k \in [0, L-1]$ occurs in an $M \times N$ image, and r_k a *random variable* representing intensities. Probability of r_k:

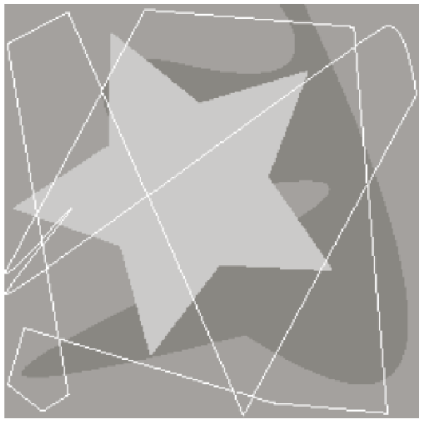
$$p_r(r_k) = \frac{n_{r_k}}{MN}, \quad k = 0, 1, \ldots, L-1$$

- If r_k is represented by $l(r_k)$ bits, the *average number of bits* per pixel is:

$$L_{\text{avg}} = \sum_{k=0}^{L-1} l(r_k)\, p_r(r_k)$$

- *Total number of bits*: $MN\, L_{\text{avg}}$.
  For a **fixed-length code of m bits:** $L_{\text{avg}} = m$**.**

Coding redundancy: example



r_k	$p_r(r_k)$	Code 1	$l_1(r_k)$	Code 2	$l_2(r_k)$
$r_{87} = 87$	0.25	01010111	8	01	2
$r_{128} = 128$	0.47	10000000	8	1	1
$r_{186} = 186$	0.25	10111010	8	000	3
$r_{255} = 255$	0.03	11111111	8	001	3
r_k for $k \neq 87, 128, 186, 255$	0	[various, unused]	8	—	0

- Code 1: fixed-length 8-bit code: $L_{\text{avg}} = 8$ bits
- Code 2: variable-length code: $L_{\text{avg}} = 1.81$ bits
  $C = \frac{8}{1.81} = 4.42$, $R = 1 - \frac{1}{4.42} = 0.774$ (77.4% of original data is redundant)

Arithmetic coding concept

- adaptive code-length (prev. slide) targets digit-level reduction
- same principle applied to code length of a message: *arithmetic coding*
- formally: Let message $\mathcal{M}$ be composed of (digit) symbols a
- each symbol a_l in alphabet of range $\{1...L\}$ has an occurence probability $p(a_l)$
- $\sum_{l=1}^{L} p(a_l) = 1 \rightarrow a_l = \sum_{m=0}^{l-1} \{p(a_m)...p(a_l)\}$
- it follows: $\mathcal{M} = \{\prod_{k=0}^{|M|-2} a(M_k)...\{\prod_{k=0}^{|M|-1} a(M_k)\}$
- Conclusion: the message is a fractional number-range {low...high}
- for transmission: select lowest binary-digit fractional number in that range

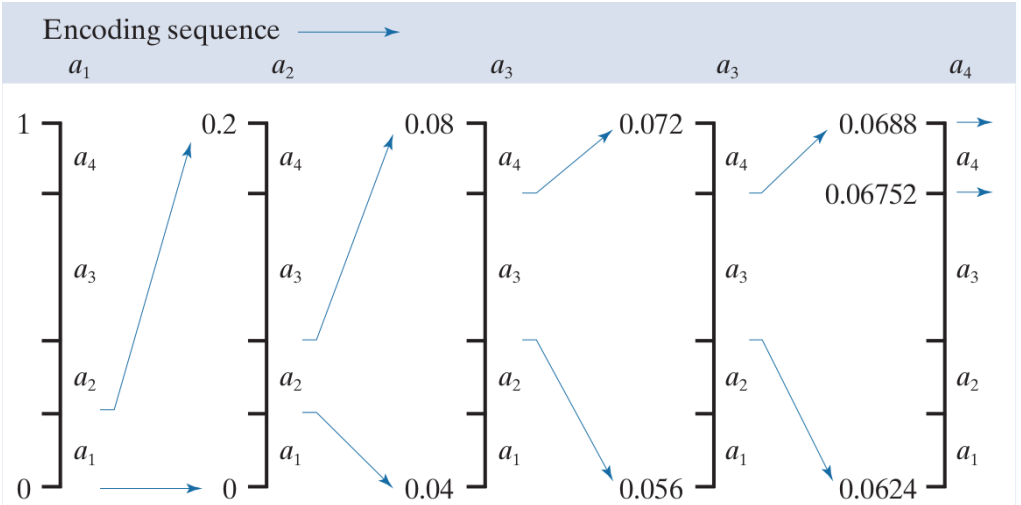
Arithmetic coding example

- We want to transmit the following message, compose of symbols $a_1 - a_4$

$$\mathcal{M} = [a_1, a_2, a_3, a_3, a_4]$$

Source Symbol	Probability	Initial Subinterval
a_1	0.2	[0.0, 0.2)
a_2	0.2	[0.2, 0.4)
a_3	0.4	[0.4, 0.8)
a_4	0.2	[0.8, 1.0)

- With probabilities above, we can encode the message



Arithmetic coding example

- Result: $\mathcal{M} = [a_1, a_2, a_3, a_3, a_4] = \{0.6752...0.0688\}$
- Numeric message can encode the message:
  $\mathcal{M}_{10} = 0.068; \mathcal{M}_2 = 0.0688 = 0.000100011$
- Bit-budget comparison required for sending:
  - original: 4 symbols $\rightarrow$ 2 bits; 5-symbol message $\rightarrow$ 10 bits (static)
  - per-symbol redundancy: (live-demo) $\rightarrow$ 10 bits (dynamic)
  - per-message redundancy: 10 bits (dynamically)
- Audience Q: What's the catch here ?

Arithmetic coding example

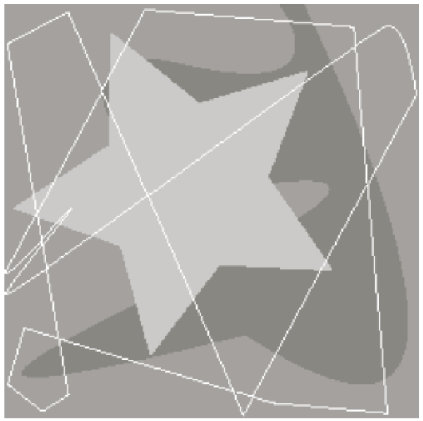
- Result: $\mathcal{M} = [a_1, a_2, a_3, a_3, a_4] = \{0.6752...0.0688\}$
- Numeric message can encode the message:
  $\mathcal{M}_{10} = 0.068; \mathcal{M}_2 = 0.0688 = 0.000100011$
- Bit-budget comparison required for sending:
  - original: 4 symbols $\rightarrow$ 2 bits; 5-symbol message $\rightarrow$ 10 bits (static)
  - per-symbol redundancy: (live-demo) $\rightarrow$ 10 bits (dynamic)
  - per-message redundancy: 10 bits (dynamically)
- Audience Q: What's the catch here ?
  - original-coding budget will rise linearly with no. symbols
  - symbol-reduced codes converge to constant budget with larger message
  - message-reduced codes converge to constant budget with larger message

Arithmetic coding in pracice

- Practical issues:
  - decoding: knowing $p(a_l)$ is imperative
  - determine symbol *probabilities* of pixels ... image have a lot of variability.
- rarely used for mono- or multichromatic images
- one application case where they find usage: bit- and QR codes
  - binary bit-patterns $\rightarrow$ constrained symbol table
  - fixed probability $\rightarrow$ pattern-scan matching
  - adequate message size for arithmetic code compression
- more details: course book, chapter 8.4

Compressing Images through the Huffman Code Transform

Coding redundancy: example



r_k	$p_r(r_k)$	Code 1	$l_1(r_k)$	Code 2	$l_2(r_k)$
$r_{87} = 87$	0.25	01010111	8	01	2
$r_{128} = 128$	0.47	10000000	8	1	1
$r_{186} = 186$	0.25	10111010	8	000	3
$r_{255} = 255$	0.03	11111111	8	001	3
r_k for $k \neq 87, 128, 186, 255$	0	[various, unused]	8	—	0

- Code 1: fixed-length 8-bit code: $L_{\text{avg}} = 8$ bits
- Code 2: variable-length code: $L_{\text{avg}} = 1.81$ bits
  $C = \frac{8}{1.81} = 4.42$, $R = 1 - \frac{1}{4.42} = 0.774$ (77.4% of original data is redundant)

Measuring information

Given a source of **statistically independent** (i.e., zero-memory) random events (source symbols) $\{a_j\}$ occurring with probability $P(a_j)$, $j = 1, 2, \ldots J$.

- *Source entropy*: $H = -\sum_{j=1}^{J} P(a_j) \log P(a_j)$
- *Estimated source entropy* for L-level image:

$$\tilde{H} = -\sum_{k=0}^{L-1} p_r(r_k) \log_2 p_r(r_k)$$

- $\tilde{H}$ is the *minimum number of bits per pixel* needed to code the image.

Coding redundancy

r_k	$p_r(r_k)$	Code 1	$l_1(r_k)$	Code 2	$l_2(r_k)$
$r_{87} = 87$	0.25	01010111	8	01	2
$r_{128} = 128$	0.47	10000000	8	1	1
$r_{186} = 186$	0.25	10111010	8	000	3
$r_{255} = 255$	0.03	11111111	8	001	3
r_k for $k \neq 87, 128, 186, 255$	0	—	8	—	0

- Code 2: variable-length code: $L_{\text{avg}} = 1.81$ bits
- Entropy

$$\tilde{H} = -(0.25 \log_2 0.25 + 0.47 \log_2 0.47 + 0.25 \log_2 0.25 + 0.03 \log_2 0.03) = 1.6614 \text{ bits}$$

- *Is there a code attaining the lower bound of* 1.6614 *bits/pixel?*

Noiseless coding theorem (Shannon 1948)

- Represent groups of n consecutive source symbols by a single code word.
- The average number of code symbols $L_{\mathrm{avg},n}$ to represent all n-symbol groups satisfies:

$$\lim_{n\to\infty}\left[\frac{L_{\mathrm{avg},n}}{n}\right] = H$$

- *Conclusion*: A zero-memory source can be represented by an average of H information units per source symbol by encoding infinitely long extensions of the single-symbol source.

Huffman coding

- Removes coding redundancy
- Yields smallest number of code symbols per source symbol provided source symbols are coded *one at a time*
- Source reduction: Ordering symbols w.r.t. probability and successively combine lowest pair of symbols into new symbol.
- Code each reduced source and work back to original source.

Huffman coding

Original source		Source reduction			
Symbol	Probability	1	2	3	4
a_2	0.4	0.4	0.4	0.4	0.6
a_6	0.3	0.3	0.3	0.3	0.4
a_1	0.1	0.1	0.2	0.3	
a_4	0.1	0.1	0.1		
a_3	0.06	0.1			
a_5	0.04				

Order symbols w.r.t. probability and successively combine lowest pair of symbols into new symbol

Huffman coding

Original source		Source reduction			
Symbol	Probability	1	2	3	4
a_2	0.4	0.4	0.4	0.4	0.6
a_6	0.3	0.3	0.3	0.3	0.4
a_1	0.1	0.1	0.2	0.3	
a_4	0.1	0.1	0.1		
a_3	0.06	0.1			
a_5	0.04				

Order symbols w.r.t. probability and successively combine lowest pair of symbols into new symbol

Original source			Source reduction							
Symbol	Probability	Code	1		2		3		4	
a_2	0.4	1	0.4	1	0.4	1	0.4	1	0.6	0
a_6	0.3	00	0.3	00	0.3	00	0.3	00	0.4	1
a_1	0.1	011	0.1	011	0.2	010	0.3	01		
a_4	0.1	0100	0.1	0100	0.1	011				
a_3	0.06	01010	0.1	0101						
a_5	0.04	01011								

Code each reduced source and work back to original source

$$L_{\text{avg}} = 0.4 \times 1 + 0.3 \times 2 + \cdots = 2.2 \text{ bits/pixel}$$
$$\tilde{H} = 2.14 \text{ bits/pixel}$$

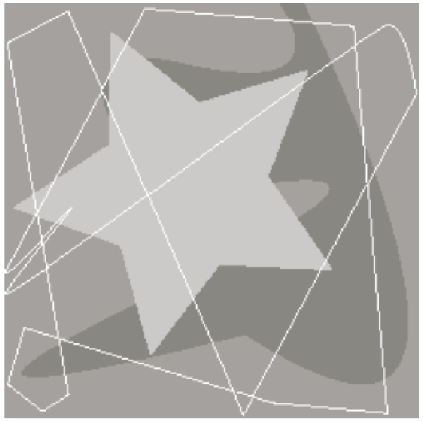
Huffman coding

- *Block code*: each source symbol is mapped to fixed sequence of code symbols
- *Instantaneous*: each code word can be decoded without referencing succeeding symbols
- *Uniquely decodable*: any string of code symbols can be decoded in only one way.
- Yields *smallest number of code symbols* per source symbol provided source symbols are coded *one at a time*
- Requires *code table*.

An encoded string 010100111100 is decoded as $a_3 a_1 a_2 a_2 a_6$

Compressing Images by Spatial Redundancy in Codes

Spatial redundancy



- back to this inciting image: large areas of same intensity
- adaptive symbol-to-code map reduces bits stored per pixel ...
- ... yet still needs to store a symbol for *every* pixel
- better approach: let's count (and store) symbol repetitions

Run-Length Encoding - base concept

- 2D image $\rightarrow$ 1D array
- maintain mapping dictionary $D = [\mathbb{I}(x), |x \in \mathbb{I}(x)|]$
- count pixel number $|\mathbb{I}(x) = b \ \forall x \in X|$ until value change occurs
- on value change: finalize prior entry in D, start new entry
- finalize D at end of array

Advantage: simple, yet effective (depending on the image content). [in the future, there will a a nice animation here ...]

Run-Length Encoding - Decoding

- unravel dictionary D into image $\mathbb{I}(x)$
- rearrage $\mathbb{I}(x) \rightarrow \mathbb{I}(x, y)$
- requires header information: byte-order, width, length, no. channels
- if applied to colour-pallete images: add colour-palette to header

Run-Length Encoding - Variations

- 1D version (basic):
  - 8-bit – n-bit version: unconstrained count; requires **one** extra symbol as count-terminator
  - 8-bit – 8-bit version: count max. 255 repetitions, then repeat entry in D
- 2D version (+ 1D variations):
  - continuous count: basically 1D encoding on a 2D image
  - scanline without terminator: terminate count at scanline (i.e. row) end; implicitly start newline
  - scanline with terminator: terminate count at scanline (i.e. row) end; add special-symbol as terminator
- Windows BMP:
  - *encode*-mode: 2D continuous mode 8-bit – 8-bit version with switch pair order
  - *absolute*-mode: scanline with terminator; D-entries have two pairs, including terminator
    (see book chp. 8.6, example 8.8 & table 8.9)
- Facsimile (i.e. *fax*) version: CCITT Group 3 & 4 definitions

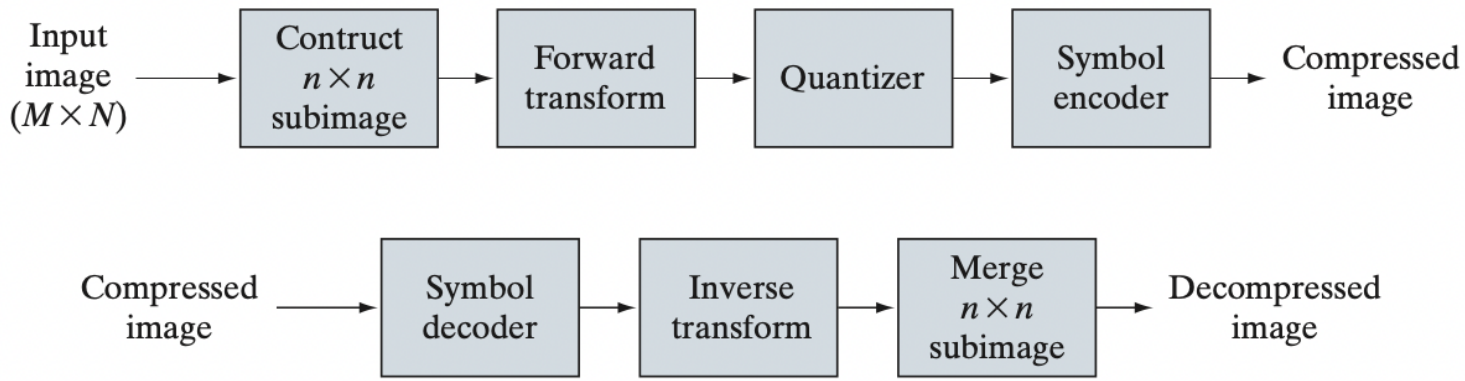
Run-Length Encoding - Applications

- binary images → no need to store intensity value
- pixel-art
- rasterized drawing & schemata
- colour-palette images

Furthermore: it combines well with symbol-reduced codes.

Compressing Images by Sampling & Encoding Image Blocks

Block transform coding



- Divide image in *non-overlapping* $n \times n$ *blocks* and transform blocks independently.
- Use *reversible, linear transform* for each block to **decorrelate pixel values**: discrete Fourier transform (DFT), discrete cosine transform (DCT)
- *Bit allocation*: the transform *coefficients* are truncated, quantized and coded.

Block transform coding

Transform $n \times n$ subimage $g(x, y)$:

$$T(u, v) = \sum_{x=0}^{n-1} \sum_{y=0}^{n-1} g(x, y)\, r(x, y, u, v), \quad u, v = 0, 1, 2, \ldots, n-1$$

$$g(x, y) = \sum_{u=0}^{n-1} \sum_{v=0}^{n-1} T(u, v)\, s(x, y, u, v), \quad x, y = 0, 1, 2, \ldots, n-1$$

Block transform coding

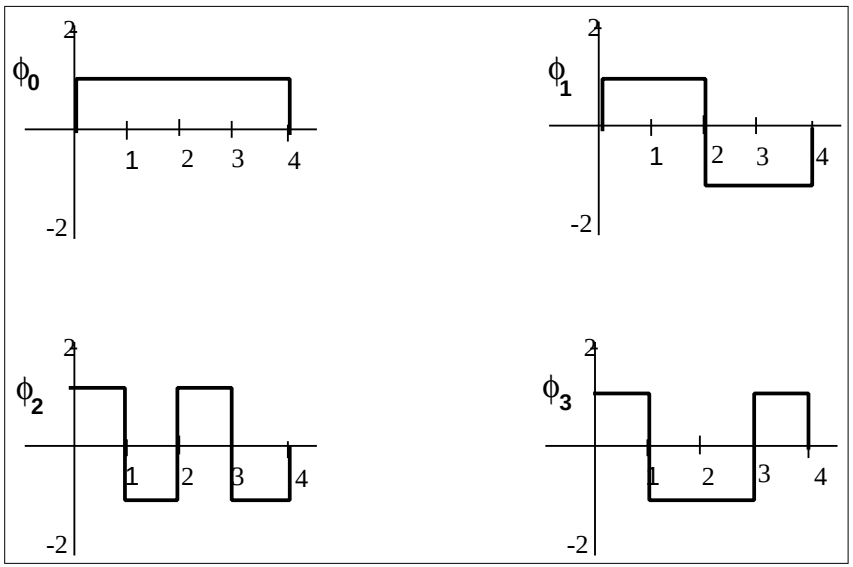
Transform $n \times n$ subimage $g(x, y)$:

$$T(u, v) = \sum_{x=0}^{n-1} \sum_{y=0}^{n-1} g(x, y)\, r(x, y, u, v), \quad u, v = 0, 1, 2, \ldots, n-1$$

$$g(x, y) = \sum_{u=0}^{n-1} \sum_{v=0}^{n-1} T(u, v)\, s(x, y, u, v), \quad x, y = 0, 1, 2, \ldots, n-1$$

- $T(u, v)$: transform coefficients
- $r(x, y, u, v)/s(x, y, u, v)$: forward/inverse transform kernel (basis function)
- $r(x, y, u, v) = r_1(x, u)\, r_2(y, v)$: separable kernel
- $r(x, y, u, v) = r_1(x, u)\, r_1(y, v)$: separable and symmetric kernel

Walsh-Hadamard transform (WHT)



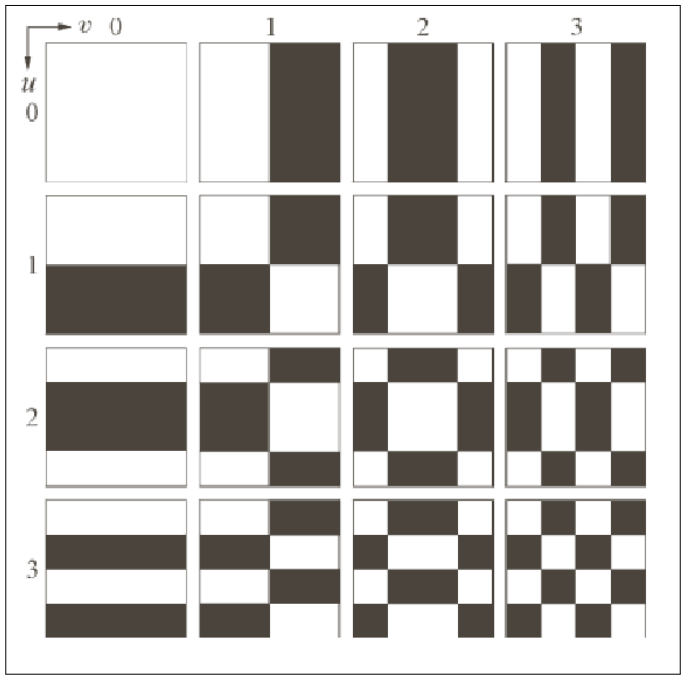
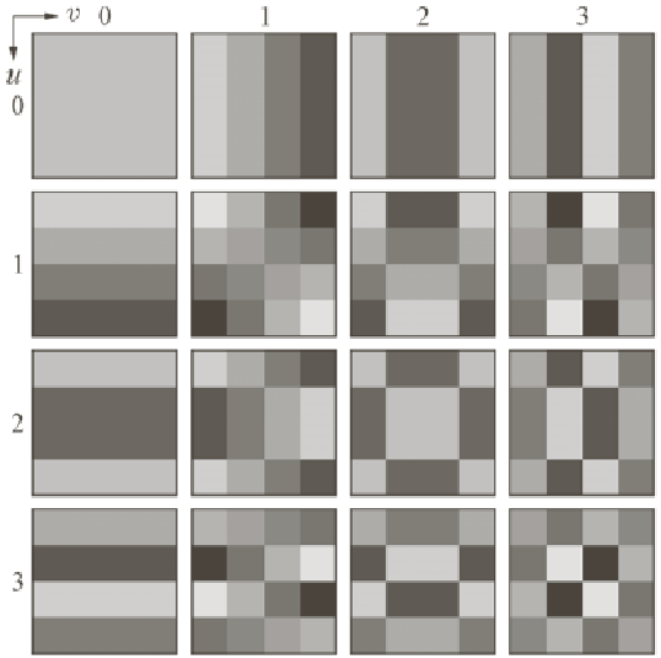


Figure: The Walsh functions for $N = 4$.

Discrete cosine transform (DCT)



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

$$r(x, y, u, v) \sim \cos\left[\frac{(2x+1)u\pi}{2n}\right] \cdot \cos\left[\frac{(2y+1)v\pi}{2n}\right]$$

Truncating coefficients

- Truncate or quantize coefficient $T(u, v)$ if it satisfies a certain *criterion*
- **Threshold coding**:
  - motivation: largest transform coefficients make the most significant contribution to the image quality
  - global threshold ($\rightarrow$ code rate is image dependent)
  - retain the N largest coefficients $T(u, v)$ ($\rightarrow$ code rate is constant)

Truncating coefficients

- Truncate or quantize coefficient $T(u, v)$ if it satisfies a certain *criterion*
- **Threshold coding**:
  - motivation: largest transform coefficients make the most significant contribution to the image quality
  - global threshold ($\rightarrow$ code rate is image dependent)
  - retain the N largest coefficients $T(u, v)$ ($\rightarrow$ code rate is constant)
- **Zonal coding**:
  - motivation: coefficients of maximum variance carry the most image information
  - retain the $p\%$ coefficients $T(u, v)$ of largest variance (usually located around the origin of the image transform for natural images – low frequency components)

Basics of Image Segmentation - Thresholding & Labelling

Image segmentation & Thresholding

- *Image segmentation*: seperate *coherent region* within coordinate space (x, y) and value space $f(x, y)$
- *Thresholding*: separation of foreground content from the the background.
- Difficulty: *foreground* definition strongly application-dependent; no coherent and sharp definition

Thresholding

Approach:
Pixels (x, y) with value above threshold $f(x, y) \geqslant t$ are part of the *foreground* set. Pixels with values below t are part of the *background* set.

$$p \in Fg \ \forall p = (x, y) \mid f(x, y) \geqslant t$$
$$p \in Bg \ \forall p = (x, y) \mid f(x, y) < t$$

Alternatively: Let $f(x, y)$ be a grey value image. Let t be a fixed grey value *threshold*.

We define the target binary image $I_t(x, y)$ (i.e. **threshold image**) as:

$$I_t(x, y) := 1 \text{ if } f(x, y) \geqslant t$$
$$I_t(x, y) := 0 \text{ if } f(x, y) < t$$

Thresholding

2	5	0	4	3	0	1	0	1	1
0	2	9	5	1	0	0	1	1	0
4	6	8	2	0	1	1	1	0	0
0	4	7	0	3	0	1	1	0	1

Image (left) and its thresholded image with $t = 3$ (right).

Thresholding

2	5	0	4	3
0	2	9	5	1
4	6	8	2	0
0	4	7	0	3

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

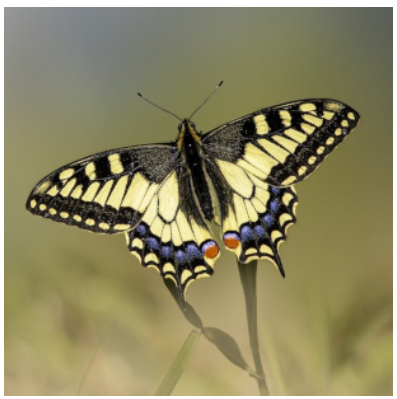
Image (left) and its thresholded image with $t = 3$ (right).

gray scale image input (left) and its thresholded binary image (right).

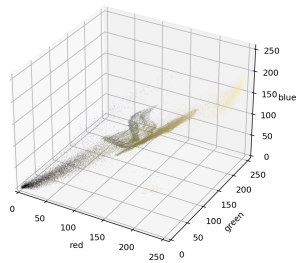
Thresholding coloured images

Thresholding colour images: not straight-forward; Options:

- Threshold per channel: $I_t = I_t(R) \cup I_t(G) \cup I_t(B)$ or $I_t = I_t(R) \cap I_t(G) \cap I_t(B)$
- Threshold luminance: $I_t = I_t(H_{HSL})$
- Clustering colour pixels in vector space: use *kMeans* on colour-space processing with $k = 2$



Original



Colour-space

Luma threshold

kMeans, $k = 2$

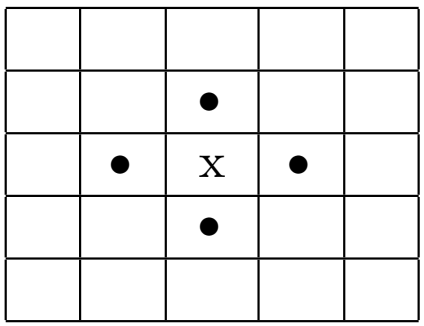
Interesting effects achievable via threshold masks ...

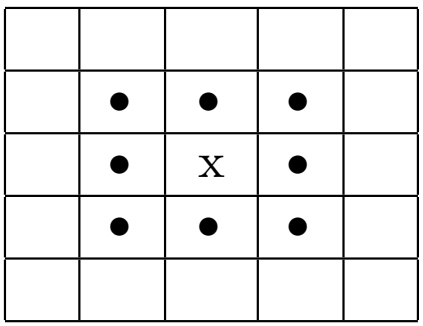
Thresholding

- Thresholding: first, *rough* segmentation
- set-adherence exclusively based on values $f(x, y)$
- resulting issues:
  - set-inclusion of outliers
  - formation of *islands* and *trenches*
  - no local adaptation
- next step: label separate foreground regions

But first: how do we know if a region is *separate* ?

Connectivity & pixel neighbourhood





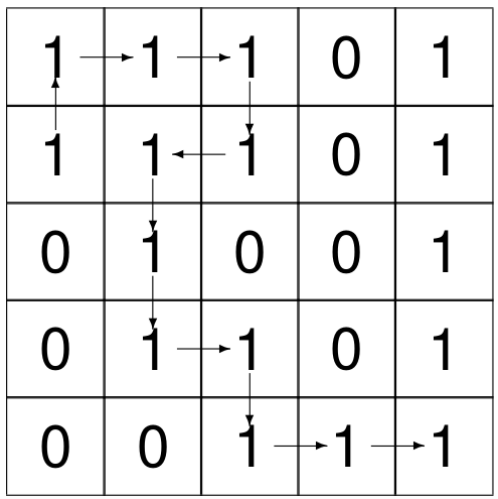
4-Connectivity/Manhattan connectivity (left):
the pixels • form the set $N_4(\mathrm{x})$ (the *4-neighbors* of x).

8-connectivity/chessboard connectivity (right):
the pixels • form the set $N_8(\mathrm{x})$ (the *8-neighbors* of x).

Adjacency and paths

- *4-adjacency*: Two pixels p and q are 4-adjacent if q is a 4-neighbor of p ($q \in N_4(p)$).
- *8-adjacency*: Two pixels p and q are 8-adjacent if q is an 8-neighbor of p ($q \in N_8(p)$).

A discrete *path* from p to q of length ℓ, denoted $p_0 = p$, $p_\ell = q$, is valid in that all members $\{p...q\}$ are direct *neighbors*.
These paths are called 4-, or 8-paths, depending on the chosen adjacency.



A 4-adjacent path of 1-pixels.

Connected components

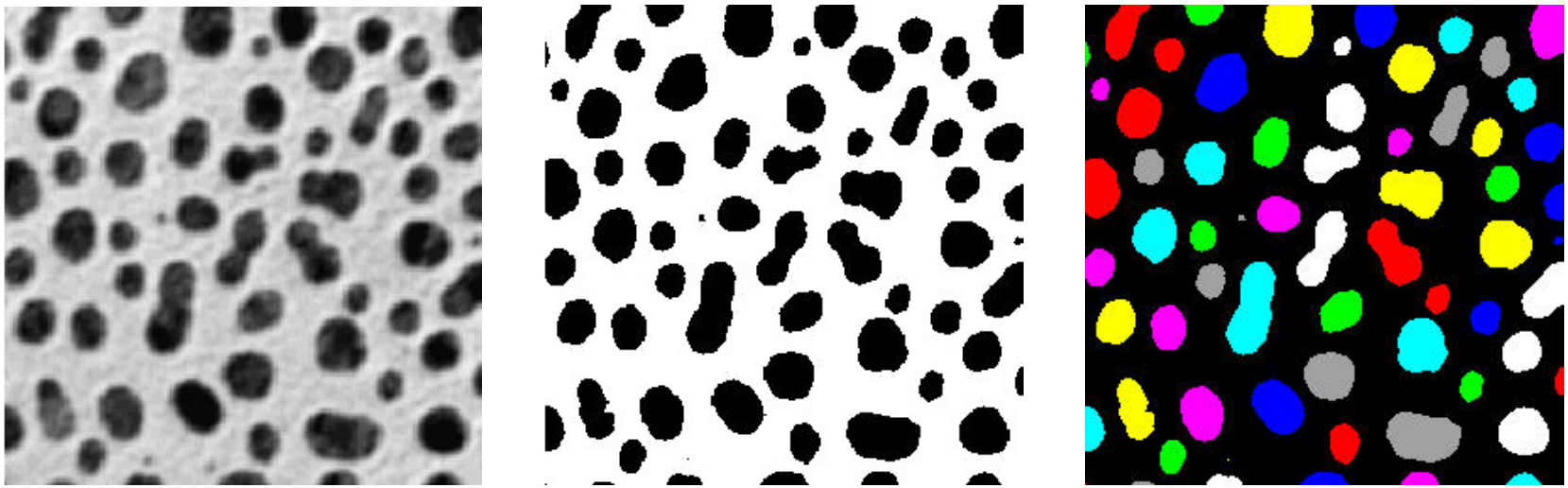
Let S be a subset of image pixels with value $f(x, y) = V$. The set S is a *connected component* if:

- there exists a path within S for all pixels $p, q \in S$
- the set S if maximal in size.

A connected component represents a closed *region*.

Component labeling

A connected component labeling algorithm assigns a label to each connected component.



(a) (b) (c)

(a): input image. (b): threshold image of (a). (c): connected components of (b) labeled with colors.

Connected-component algorithms

Two types of algorithms:

- Breadth first search algorithm: process neighbourhood FIFO queue
  drawback: uses large queue and is cache unfriendly.
- Scanline algorithms: process an image line by line, top to bottom.
  advantage: good cache performance. drawback: more complicated.

Recursive breadth-first search labelling

```
label = 0
labels = zeros(height, width)
  for i = 0:height-1 do
    for j = 0:width-1 do
      if img[i][j]==0 and labels[i][j]==0 then
        label++
        p = (i, j)
        Q = queue()
        labels[i][j] = label
        Q.push( p )
        while Q not empty do
          q = Q.pop()
          for each neighbor p = (m, n) in N(q) do
            if img[m][n]==0 and labels[m][n]==0
              labels[m][n] = label
              Q.push( p )
```

Connected-component algorithms

Two types of algorithms:

- Breadth first search algorithm: process neighbourhood FIFO queue
  drawback: uses large queue and is cache unfriendly.
- Scanline algorithms: process an image line by line, top to bottom.
  advantage: good cache performance. drawback: more complicated.

The scanline algorithms can be further sub-divided:

- **Iterative label propagation**: repeat until stability. no auxiliary memory needed, but may need many iterations (slow).
- **Two-pass algorithm** (Rosenfeld-Pfaltz, 1966):
  only two passes needed, but uses a (potentially large) equivalence table.

Iterative label propagation (slow)

```
# Let V be the gray value that we want to label (i.e. V==1).
label = 1
for all rows r from [0..height) do
  for all columns c from [0..width) do
    if im[r][c] != V then
      labeling[r][c] = 0
    else
      labeling[r][c] = label
      label = label + 1
  end for
end for
```

Iterative label propagation (slow)

```
# Let V be the gray value that we want to label (i.e. V==1).
label = 1
for all rows r from [0..height) do
  for all columns c from [0..width) do
    if im[r][c] != V then
      labeling[r][c] = 0
    else
      labeling[r][c] = label
      label = label + 1
  end for
end for

Repeat until stability
  # forward pass: top down, left to right
  for all rows r from [0..height) do
    for all columns c from [0..width) do
      if im[r][c] == V then
        NB = {(r',c') | (r',c') is a mask neighbor of (r,c), im[r'][c']==im[r][c]}
        labeling[r][c] = Minimum {labeling[r',c') | (r',c') in NB}
    end for
  end for
```

Iterative label propagation (slow)

```
# Let V be the gray value that we want to label (i.e. V==1).
label = 1
for all rows r from [0..height) do
  for all columns c from [0..width) do
    if im[r][c] != V then
      labeling[r][c] = 0
    else
      labeling[r][c] = label
      label = label + 1
  end for
end for

Repeat until stability
  # forward pass: top down, left to right
  for all rows r from [0..height) do
    for all columns c from [0..width) do
      if im[r][c] == V then
        NB = {(r',c') | (r',c') is a mask neighbor of (r,c), im[r'][c']==im[r][c]}
        labeling[r][c] = Minimum {labeling[r',c') | (r',c') in NB}
    end for
  end for
  # reverse pass: bottom-up, right to left
  for all rows r from (height..0] do
    for all columns c from (width..0] do
      if im[r][c] == V then
        NB = {(r',c') | (r',c') is a flipped mask neighbor of (r,c), im[r'][c']==im[r][c]}
        labeling[r][c] = Minimum {labeling[r'][c'] | (r',c') in NB}
    end for
  end for
end repeat
```

Iterative algorithm: example

1		1	1	1
1		1		1
1				1
1	1	1	1	1

1		2	3	4
5		6		7
8				9
10	11	12	13	14

1		2	2	2
1		2		2
1				2
1	1	1	1	1

1		1	1	1
1		2		1
1				1
1	1	1	1	1

Figure: Upper left: binary image. Upper right: initial labeling. Lower left: result after first top-down pass. Lower right: result after first bottom-up pass.

Classical algorithm: Rosenfeld/Pfaltz (1966)

- needs *only two passes* through the image
- in *first pass*, associate target pixels of value V with a label
- uses an *equivalence table* to store labels which can propagate to the same value
- whenever two different labels can propagate, the *smaller* label is chosen
- in *second pass*, all equivalences in the table have to be *resolved*: each equivalence class is assigned a unique label (usually the minimum).

Classical algorithm: Rosenfeld/Pfaltz (1966)

```
# Let V be the gray value that we want to label (i.e. V==1).
label = 1
# forward pass: top down, left to right
for all rows r from [0..height) do
  for all columns c from [0..width) do
    if im[r][c] != V then labeling[r][c]=0
    else
      NB = {(r',c') | (r',c') is a mask neighbor of (r,c), im[r'][c']==im[r][c]}
      if NB=={} then labeling[r][c]=label, label+=1
      else
        labeling[r][c]=label
        for all pairs (p,q) in NB do
          if (labeling[p] != labeling[q]) then
            store equivalence (labeling[p],labeling[q])
        endfor
        labeling[r][c] = Minimum {labeling[r'][c'] | (r',c') in NB}
  end for
end for
```

Classical algorithm: Rosenfeld/Pfaltz (1966)

```
# Let V be the gray value that we want to label (i.e. V==1).
label = 1
# forward pass: top down, left to right
for all rows r from [0..height) do
  for all columns c from [0..width) do
    if im[r][c] != V then labeling[r][c]=0
    else
      NB = {(r',c') | (r',c') is a mask neighbor of (r,c), im[r'][c']==im[r][c]}
      if NB=={} then labeling[r][c]=label, label+=1
      else
        labeling[r][c]=label
        for all pairs (p,q) in NB do
          if (labeling[p] != labeling[q]) then
            store equivalence (labeling[p],labeling[q])
        endfor
        labeling[r][c] = Minimum {labeling[r'][c'] | (r',c') in NB}
  end for
end for
# Resolve equivalences
Compute minimum label for each equivalence class (transitive closure)
# second pass
for all rows r from [0..height) do
  for all columns c from [0..width) do
    if im[r][c] == V then
      labeling[r][c] = closure(labeling[r][c])
  end for
end for
```

Rosenfeld-Pfaltz algorithm

	1		1	1
1	1			1
	1	1		1
		1		1
	1	1		1

	1		2	2
3	1			2
	1	1		2
		1		2
	4	1		2

Figure: Left: binary image. Right: labeling after first top-down pass. The equivalence classes are: $\{1, 3, 4\}$ and $\{2\}$.

Describing Image Content in numerically-comparable Structures

Features - Overview

Feature and Descriptors

- Boundary preprocessing
  - **Boundary tracing**, **chain codes**, shape numbers, polygon representation, signatures
- Boundary feature descriptors
  - **Basic descriptors**, signatures, boundary segments, curvature, Fourier descriptors, statistical moments
- Region feature descriptors
  - **Basic descriptors**, topological descriptors
- Texture descriptors

Purpose of features

- Processing result: alternative image description
  - key points, edges & boundaries
  - segments & areas
- Next step: *higher level processing*
  → image analysis & pattern recognition

Purpose of features

- Processing result: alternative image description
  - key points, edges & boundaries
  - segments & areas
- Next step: *higher level processing*
  → image analysis & pattern recognition
- *Feature* representation → desirable property: *invariance* to change:
  size, *translation* or *rotation*.

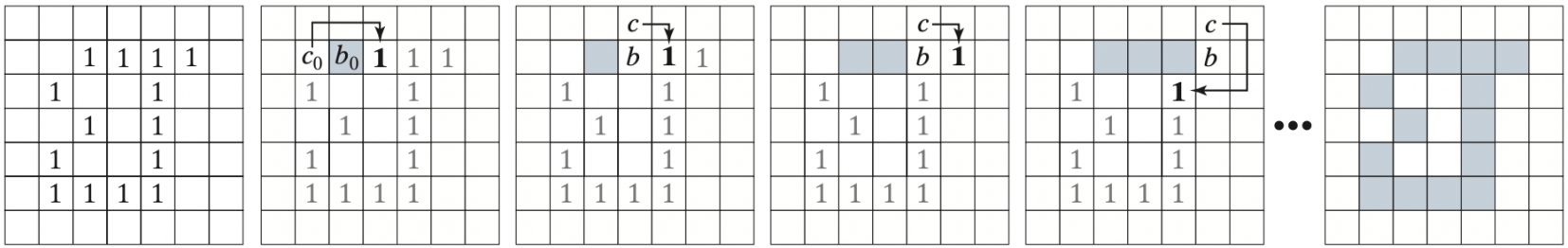
Purpose of features

- Processing result: alternative image description
  - key points, edges & boundaries
  - segments & areas
- Next step: *higher level processing*
  → image analysis & pattern recognition
- *Feature* representation → desirable property: *invariance* to change:
  size, *translation* or *rotation*.
- Feature *measurement*: use prior techniques → linear- or morphological image operators.

Boundaries - Extraction & Description

Extraction: Moore boundary tracing

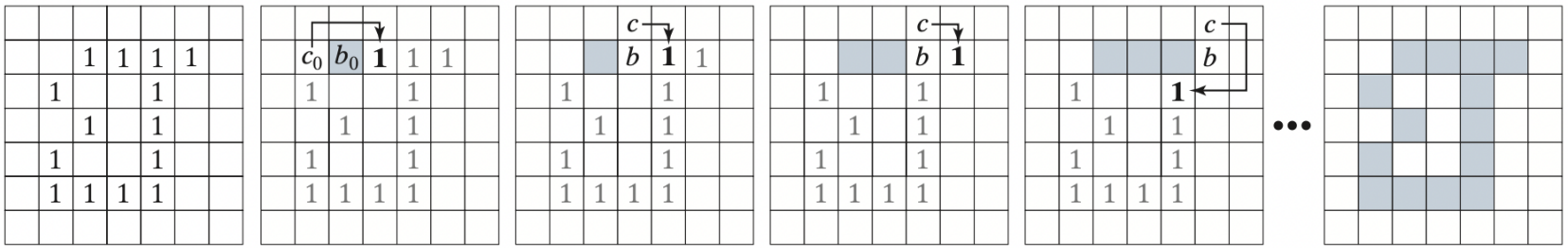
Goal: Put boundary points of a binary region R (or its boundary) in a *clockwise-sorted order*.



- Boundary and background are labeled as 1 and 0, respectively.
- b_0: the starting point, the *uppermost-leftmost point* labeled as 1 in the image.
- c_0: the *west* neighbour (always a background point) of b_0.

Extraction: Moore boundary tracing

Goal: Put boundary points of a binary region R (or its boundary) in a *clockwise-sorted order*.

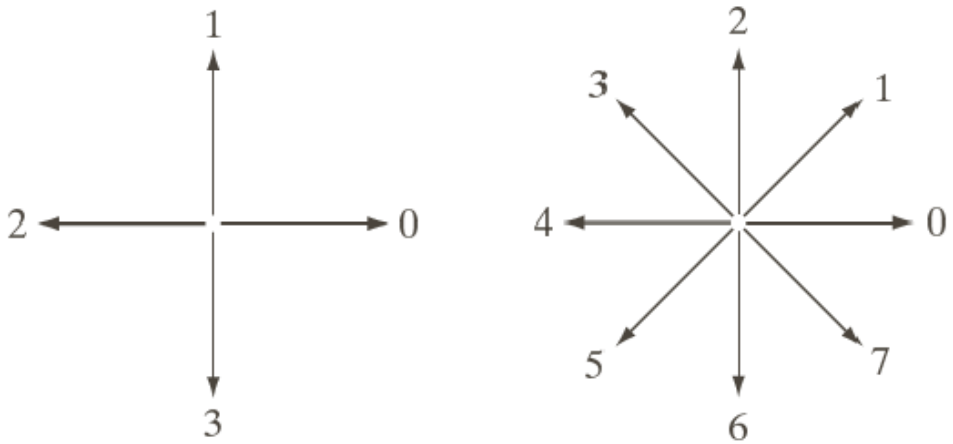


1. Let $b = b_0$ and $c = c_0$.
2. Let the 8-neighbours of b, starting at c and proceeding in a clockwise direction, be denoted by $n_1, ..., n_k$. Find the first neighbour labeled 1 and denote it by n_k.
3. Let $c = n_{k-1}$ and $b = n_k$.
4. Repeat (2) and (3) until $b = b_0$. Sequence of b is ordered boundary points.

Also works on a binary region as an input instead of a boundary.

Extraction: Freeman chain codes

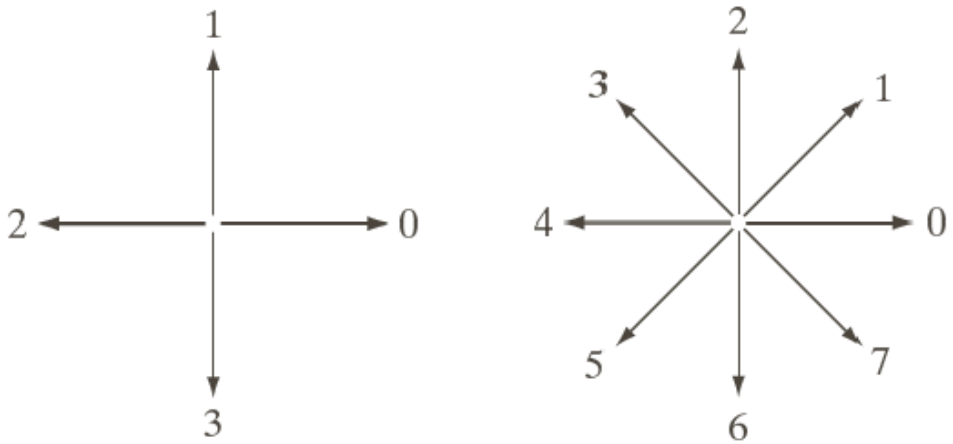
Goal: To represent the boundary as a sequence of connected sequence of straight line segments.



- Direction numbers for 4-directional and 8-directional chain codes.
- Trace the boundary using line segments shown above.
- The direction of each line segment is coded using a *numbering scheme* adapted to the connectivity.

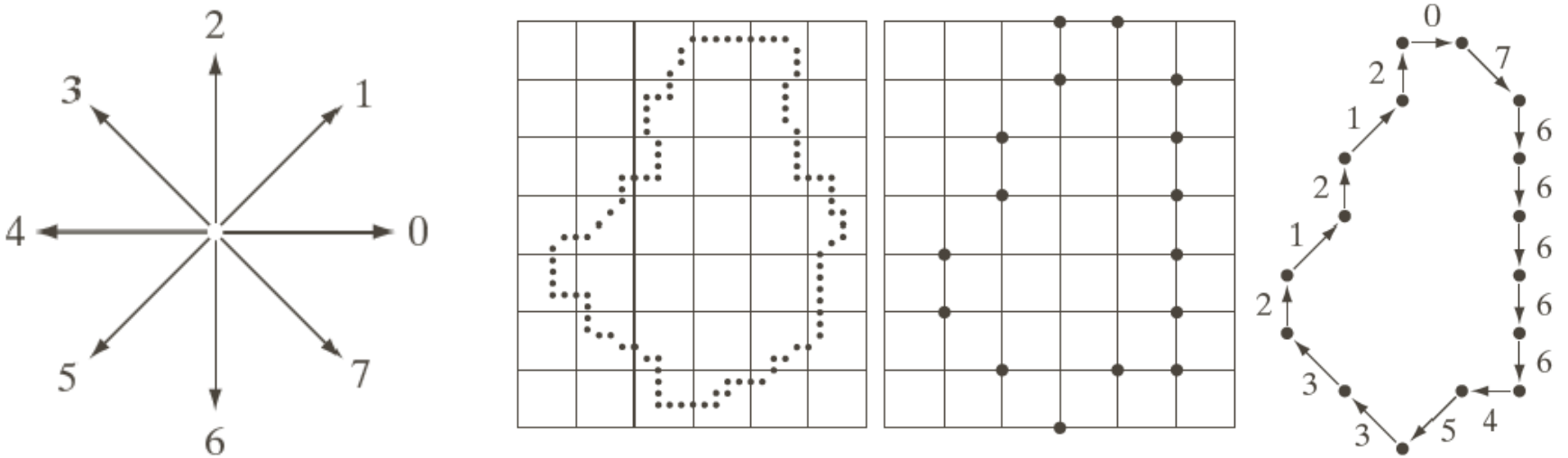
Extraction: Freeman chain codes

Goal: To represent the boundary as a sequence of connected sequence of straight line segments.



- Direction numbers for 4-directional and 8-directional chain codes.
- Trace the boundary using line segments shown above.
- The direction of each line segment is coded using a *numbering scheme* adapted to the connectivity.

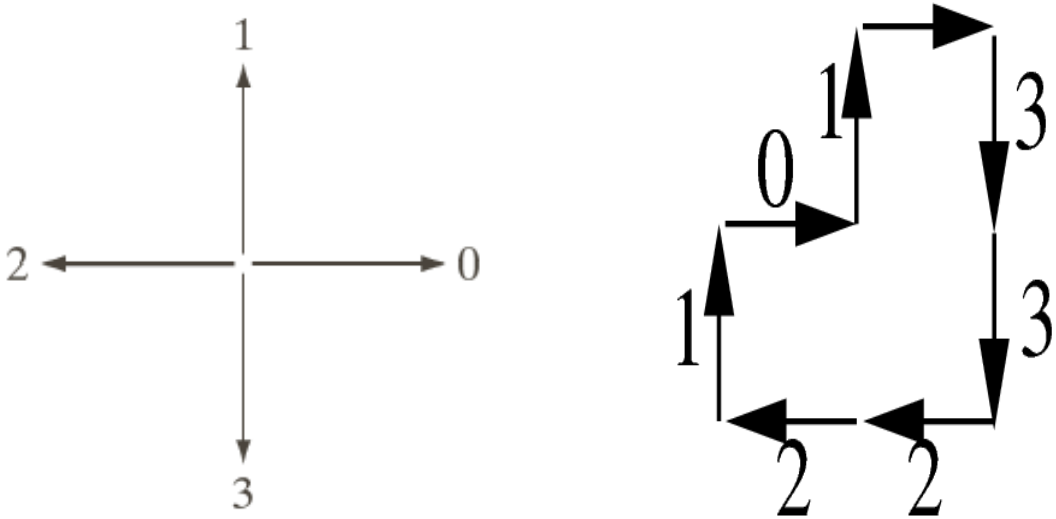
Extraction: Freeman chain codes



(a) 8-directional chain code. (b) Digital boundary with resampling grid. (c) Resampled boundary. (d) 8-directional chain-coded boundary.

Extraction: First-difference chain codes

If we rotate the region, the chain code changes. How to introduce *rotation invariance* ?



- First difference of a chain code: count number of direction changes (e.g., counterclockwise) that separate two adjacent code elements.
- chain code: 10103322 (start lower left)
  first difference: 33133030 (circular sequence)

Descriptors: Basic shape characteristics

- *Length*: no. of pixels along contour.
- *Diameter*: $\mathrm{Diam}(B) = \max_{i,j} [D(p_i, p_j)]$ with p_i, p_j points on the boundary B and D a distance measure.

Descriptors: Basic shape characteristics

- *Length*: no. of pixels along contour.
- *Diameter*: $\mathrm{Diam}(B) = \max_{i,j}[D(p_i, p_j)]$ with p_i, p_j points on the boundary B and D a distance measure.
- *Major axis*: line segment connecting the extreme points comprising the diameter
- *Minor axis*: line segment perpendicular to major axis such that rectangle defined by major and minor axis tightly encloses the boundary

Descriptors: Basic shape characteristics

- *Length*: no. of pixels along contour.
- *Diameter*: $\mathrm{Diam}(B) = \max_{i,j}[D(p_i, p_j)]$ with p_i, p_j points on the boundary B and D a distance measure.
- *Major axis*: line segment connecting the extreme points comprising the diameter
- *Minor axis*: line segment perpendicular to major axis such that rectangle defined by major and minor axis tightly encloses the boundary
- *Eccentricity of the boundary*: ratio of lengths of major and minor axes

Region: shape and content

Regional Descriptors - region R_c

- *Boundary descriptor*: see above
- *Area*: area $A(B) = \sum_{n=0}^{N} \begin{cases} 0, & \text{if } (x,y)_n \in R_c \\ 1, & \text{otherwise} \end{cases}$
- Mean, median, maximum, minimum *grey level* or *colour*

Region: shape and content

Regional Descriptors - region R_c

- *Boundary descriptor*: see above
- *Area*: area $A(B) = \sum_{n=0}^{N} \begin{cases} 0, & \text{if } (x,y)_n \in R_c \\ 1, & \text{otherwise} \end{cases}$
- Mean, median, maximum, minimum *grey level* or *colour*

That's it for this week!