Image compression

Outline

Introduction

- What is image *compression*?
- *Data* and *information*, *irrelevant information* and *redundancy*
- Compression *models*, *lossless* vs *lossy, fidelity criteria*
- Major file formats, e.g. GIF, PNG, TIFF, JPEG…

Basic compression methods

- *Run-length* coding
- *Huffman* coding
- *Arithmetic* coding
- *LZW* coding
- *Block-transform* coding

CREDITS: The slides for this lecture contain material from the course of Prof. Václav Hlaváč

What is image compression? (1/3)

- *Data compression* refers to the process of **reducing the amount of data** required to represent a *given quantity of information*
  - *Data* and *information* are **not the same thing**
  - **Data** are the means by which **information** is conveyed
    e.g. "*Today is sunny*" and "*Oggi c'é il sole*" convey the same information, but using two different representations

- Usage: **save storage space** and **reduce transmission time**
- Notice that **various amounts of data** can be used to represent the **same amount of information**
  - Representations may contain *repeated* information
    e.g. "…Today, May 24th 2018, which is the day after May 23rd 2018, is sunny…"
  - Representations may contain *irrelevant* information
    e.g. "…Today is sunny, and the Sun is the star at the center of the solar system…"
  - These representations contain **redundant data**

What is image compression? (2/3)

- In **case of images**, we want to reduce the amount of data needed to represent an image "*without affecting its quality*"

Original (1.9MB)

Compressed (230 KB, 12%)

Compressed (126 KB, 7%)

- Why do we **need** to compress images?
  - *Example*: two-hour standard definition (SD) movie
  - Amount of *space per second* of movie

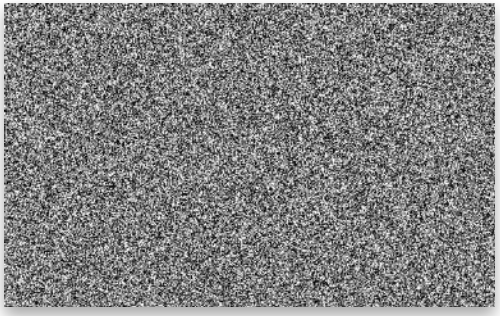
$$30\,\frac{\text{frames}}{\text{sec}} \times (720 \times 480)\,\frac{\text{pixels}}{\text{frame}} \times 3\,\frac{\text{bytes}}{\text{pixel}} = 31{,}104{,}000 \text{ bytes/sec}$$

  - **224 GB** for the entire movie (about 27 DVDs 8.5 GB dual-layer)
  - Imagine the space required for 4K, 5K etc

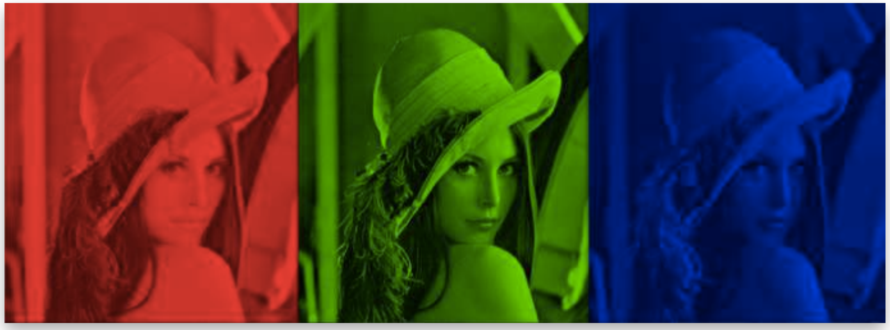
What is image compression? (3/3)

What makes image compression **possible**?

- Images are **not random**



- Images are **redundant**
  - Pixels are spatially correlated
  - Color channels, too



- **Human eyes** do not perceive all details

Original (1.9MB)

Compressed (230 KB, 12%)

Measuring redundancy and compression

Compression ratio

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

before compression
after compression

- b and b' denote the **number of bits** in two representations of the same information
- Example: C=10 ➜ larger representation uses *10 times more bits* than the smaller (for the same information)
- C=10 usually written as **10:1** or **10x**

Relative data redundancy

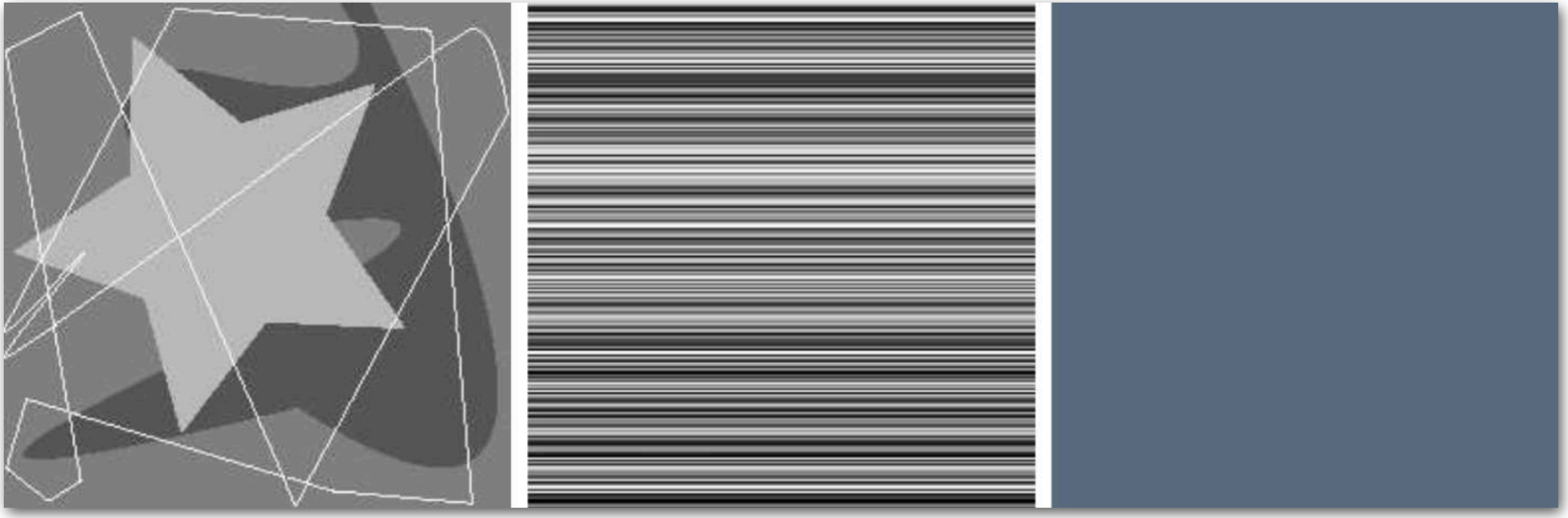
$$R = 1 - \frac{1}{C}$$

- Example: R=0.9 ➜ 90% of the data in the larger representation is *redundant*
- **Notes**
  - C=1 ➜ R=0 (no redundancy)
  - $C \to \infty$ ➜ R=1 (high redundancy)
  - C<1 ?

Redundancy in 2D images (1/4)

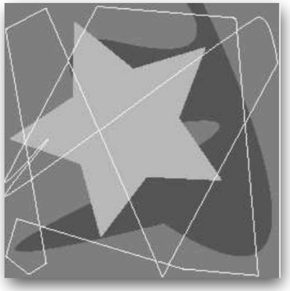
Principal types of **data redundancies**

- **Coding redundancy**
  IDEA: *fixed-length codes* (i.e. 8 bits/pixel) typically used to store pixel intensities actually *contain more bits than are needed* to represent the intensities
- **Spatial redundancy**
  IDEA: pixels of most images are *spatially correlated*
- **Irrelevant information**
  IDEA: most images contain information that is *ignored by human visual system*



Redundancy in 2D images (2/4)

Previous example



- Compare **two codes**
  - **Code 1**: 8 bits/pixel
  - **Code 2**: uses less bits for more frequent pixels

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	11000100	8	000	3
$r_{255} = 255$	0.03	11111111	8	001	3
r_k for $k \neq 87, 128, 186, 255$	0	—	8	—	0

- **Fixed-length** code: $L_{avg} = 8$ bits/pixel
- **Variable-length** code: $L_{avg} = 0.25*2 + 0.47*1 + 0.25*3 + 0.03*3 = 1.81$ bits/pixel

$$C = \frac{256 \times 256 \times 8}{118{,}621} = \frac{8}{1.81} \approx 4.42$$

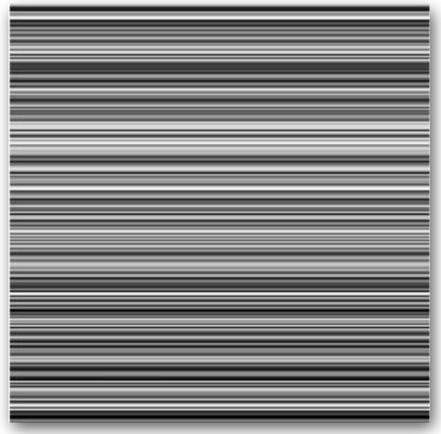
NB: using 2 bits/pixel
(4 gray levels)
C = 8/2 = 4.00

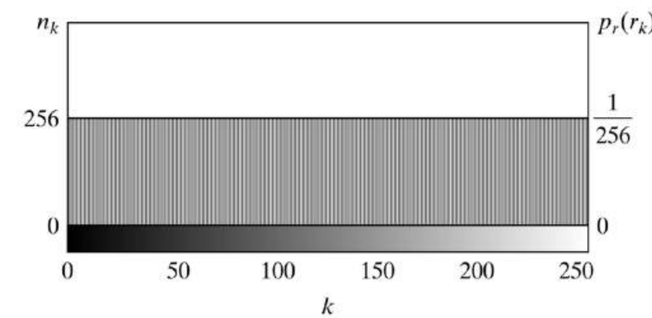
$$R = 1 - \frac{1}{4.42} = 0.774$$

77.4% of the data
in the original image
is redundant

Redundancy in 2D images (3/4)

Spatial redundancy





- Pixels along **each line** are identical
- The **histogram is flat** ➜ a *fixed-length 8-bit code* is optimal in this case
  - i.e. variable-length codes do not help to remove this type of redundancy
- Redundancy can be eliminated by using, e.g., **run-length pairs**
  - i.e. define *starting of a new intensity* and the *number of consecutive pixels* that have that intensity
  - $C = (256 * 256 * 8) / [256 * (8 + 8)] = 128$
- In general, **pixel intensities can be predicted** reasonably well from neighboring pixels
  - The idea is to **transform an image** into a more efficient but usually "non-visual" representation
  - The transformed representation is chosen such that it's **easier to remove this redundancy**

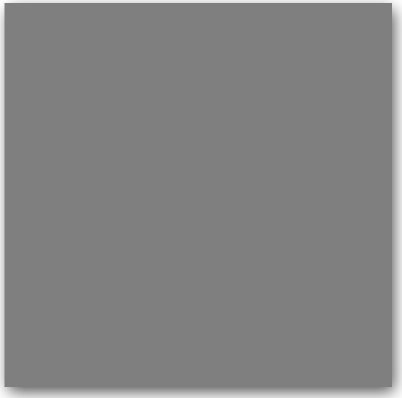
NB: in case of **videos**, there's also a *temporal redundancy*

Redundancy in 2D images (4/4)

Irrelevant information

- *One of simplest ways* to compress a set of data is to **remove superfluous data**
  - Information that is *ignored by the human visual system*
  - Information that is *extraneous to the intended use*

Example



- Image **appears to be** a homogeneous field of gray
- It can be *represented by a single 8-bit value*
  - i.e. its unique gray level
  - $C = (256 * 256 * 8) / 8 = 65536$
- Actually, if we **scale the histogram**
  - More details can be seen…
  - …but our eyes didn't see them!
- Using 1 byte only to represent it, there would be **no perceived decrease in reconstructed image quality**

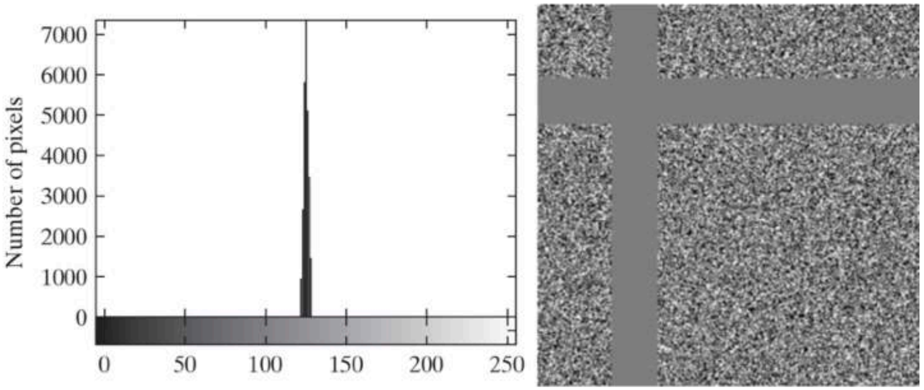
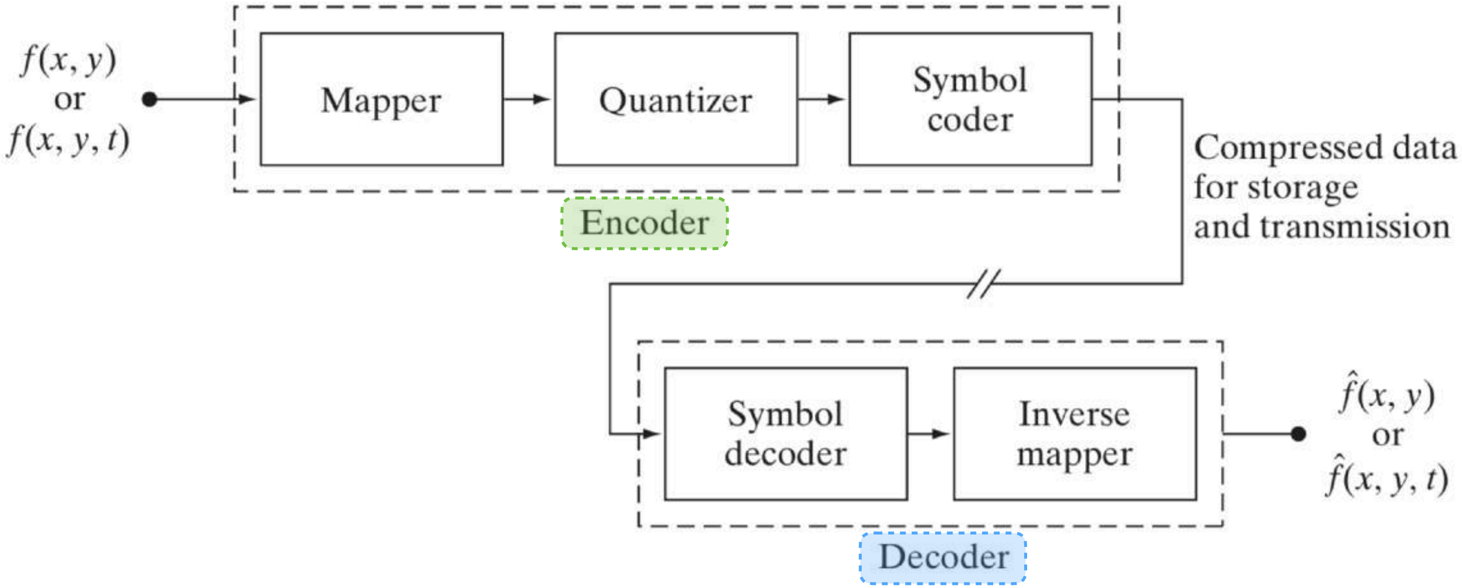


Image compression model (1/2)

General formulation

- **Encoder** performs compression
  i.e. creates a compressed representation of an image $f(x,y)$
- This compressed representation is *stored* or *transmitted*
  - **NB**: JPG or GIF files <u>do not contain</u> pixel values!
- **Decoder** performs the complementary operation
  i.e. estimates a decompressed version of $f(x,y)$, i.e. $\hat{f}(x,y)$

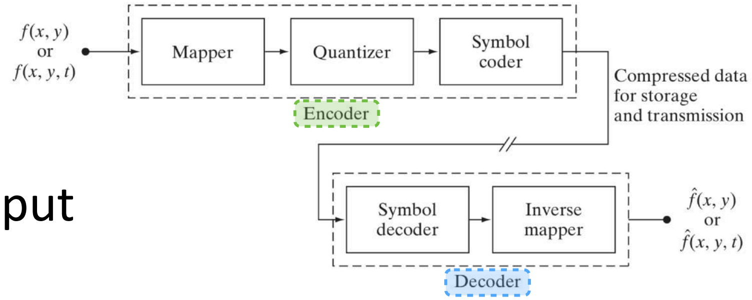


- **Codec**: *device* (hardware) or *program* (software) performing both encoding/decoding

Image compression model (2/2)

Encoding process, i.e. compression

- **Mapper:** *transforms* $f(x,y)$ *into a representation* (usually non-visual) designed to reduce spatial/temporal redundancy
  - The goal is to find a representation where the *data are less correlated (e.g. recall the FFT)*
  - This operation generally is *reversible*



- **Quantizer:** *reduces the accuracy* of the mapper's output
  - The goal is to *remove irrelevant information (NB: recall the effect of neglecting high frequencies in FFT)*
  - The amount of data to discard can be *tuned by the user*
  - This operation is *not reversible*
- **Symbol coder:** *generates a code* to represent the quantizer output and maps its output in accordance with such code
  - This operation is *reversible*

Decoding process, i.e. decompression

- Performs, in reverse order, the **inverse operations** of the *symbol encoder* and *mapper*
- NB: as *quantization* results in information loss, there's not such a "inverse quantizer"

Lossless vs lossy compression

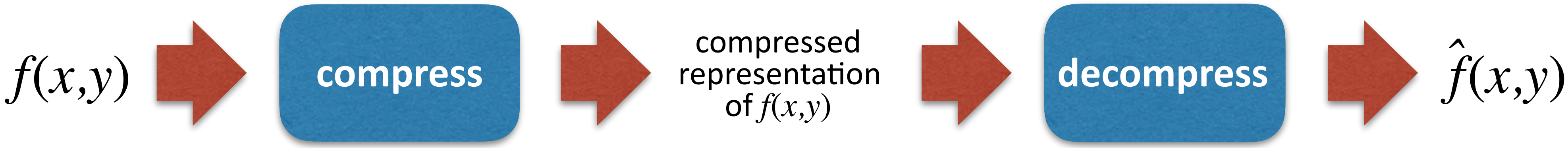
- In general, $\hat{f}(x,y)$ may/may not be an exact replica of $f(x,y)$
  - $\hat{f}(x,y) = f(x,y) \Rightarrow$ compression system is *error free* or **lossless**
  - If not, the reconstructed *image is distorted* and the compression system is called **lossy**

- **Lossless** methods
  - Only the *statistical redundancy* is removed
  - A *full reconstruction* of the original image is possible

- **Lossy** methods
  - *Irrelevant information* is removed
  - The removed information is *unnecessary in a given context* (e.g. high frequencies, details unobservable by human eyes)
  - Only *partial reconstruction* of the original image is possible

- The removal of "irrelevant information" from the image involves a **loss of real or quantitative information**
  - We need a way to **quantify the nature of this loss**

- **How close** is the *estimated* $\hat{f}(x,y)$ to the *original* $f(x,y)$?



- **Two types of criteria** to assess reconstruction fidelity
  - **Objective** fidelity criteria:
    information loss is *expressed as a mathematical function* of the input and output of the compression process
  - **Subjective** fidelity criteria:
    based on the *human observer*

Objective fidelity criteria

- *Root-mean-square error*

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

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

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

NB: this quantity is often expresses in **dB** (*decibel*)
i.e. $10 \log_{10} \mathrm{SNR}_{\mathrm{ms}}$

Notes

- Offer a *simple* and *convenient way* to evaluate information loss
- Decompressed images are **ultimately viewed by humans**
- Measuring image quality by *subjective evaluations of people* is more appropriate

Subjective fidelity criteria

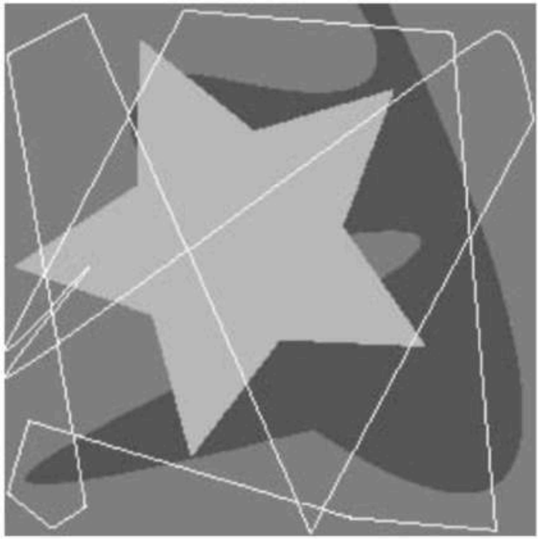
- Present a decompressed image to a *group of viewers*

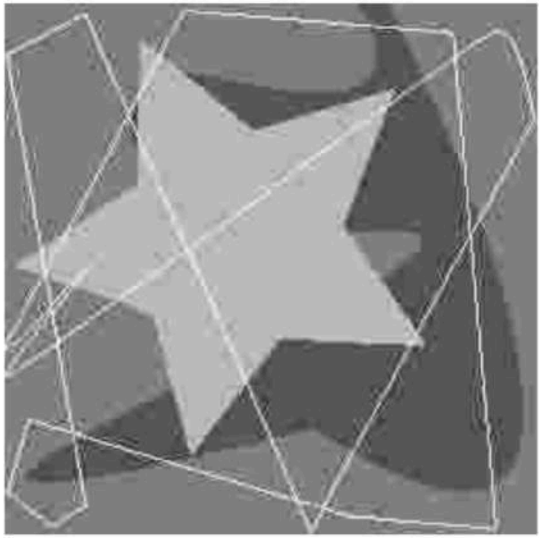
- Ask them to **evaluate its quality**
  - *Side-by-side comparisons* of $f(x,y)$ and $\hat{f}(x,y)$
  - Using an *absolute rating scale*

Value	Rating	Description
1	Excellent	An image of extremely high quality, as good as you could desire.
2	Fine	An image of high quality, providing enjoyable viewing. Interference is not objectionable.
3	Passable	An image of acceptable quality. Interference is not objectionable.
4	Marginal	An image of poor quality; you wish you could improve it. Interference is somewhat objectionable.
5	Inferior	A very poor image, but you could watch it. Objectionable interference is definitely present.
6	Unusable	An image so bad that you could not watch it.

- *Average* their evaluations

$f(x,y)$

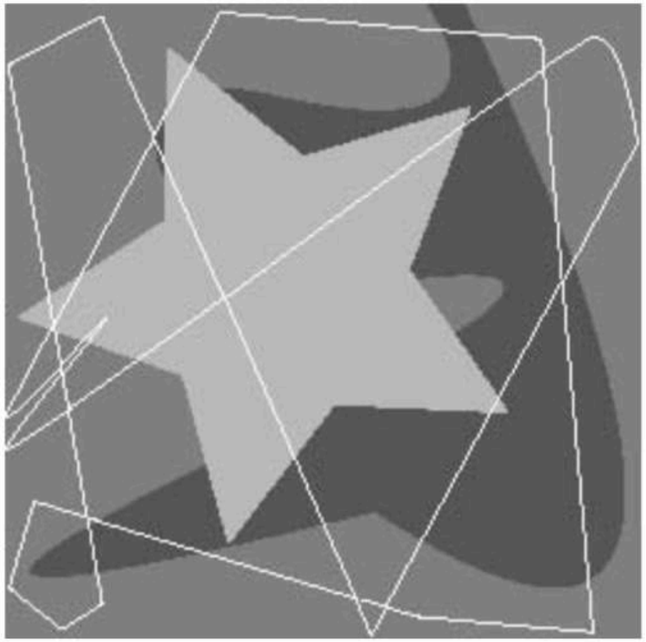




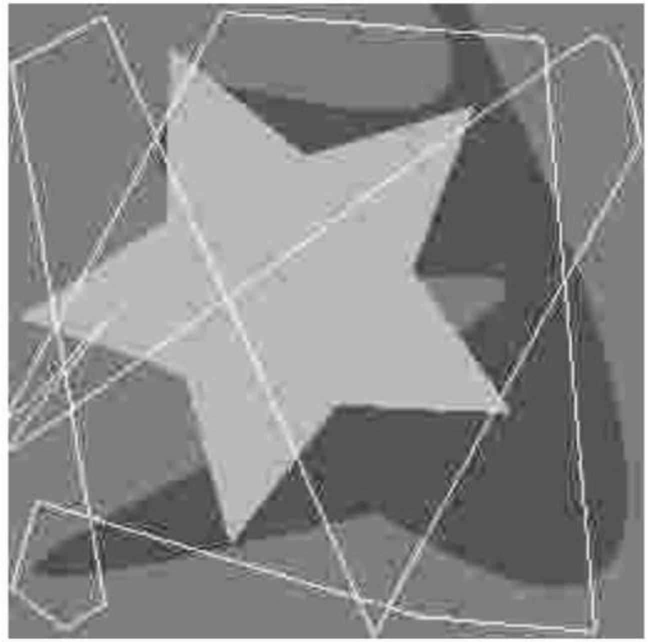
$\hat{f}(x,y)$

Fidelity criteria (4/4)

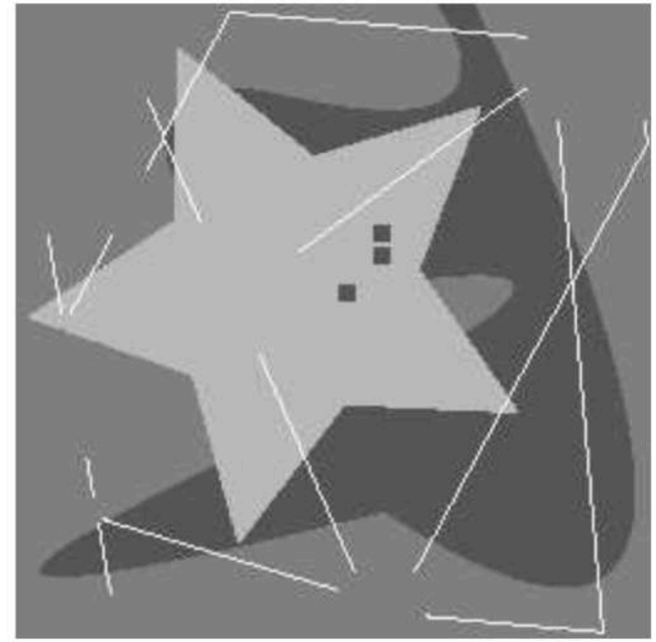
Example



$f(x,y)$



$e_{rms} = 15.67$



$e_{rms} = 14.17$

Notes

- What would be **your evaluation**?
- What is **your definition of quality**?
  - Presence of all image details?
  - Absence of noticeable degradation?
  - Something else?

Image formats, containers and standards

- **File formats**: standard way to organize and store image data
  - Defines *how the data is arranged* and the *type of compression* (if any) that is used
  - NB: **image containers** are similar to file formats, but handle multiple types of data

- **Compression standards**: define the procedures and the algorithms for compressing and decompressing the images

Image formats, containers and standards

- **File for… e data**
  - Define… at is used
  - NB: **im…** es of data
- **Compr… e**
  algorit… ages

Name	Organization	Description
Continuous-Tone Still Images		
BMP	Microsoft	*Windows Bitmap*. A file format used mainly for simple uncompressed images.
GIF	CompuServe	*Graphic Interchange Format*. A file format that uses lossless LZW coding [8.2.4] for 1- through 8-bit images. It is frequently used to make small animations and short low resolution films for the World Wide Web.
PDF	Adobe Systems	*Portable Document Format*. A format for representing 2-D documents in a device and resolution independent way. It can function as a container for JPEG, JPEG 2000, CCITT, and other compressed images. Some PDF versions have become ISO standards.
PNG	*World Wide Web Consortium* (W3C)	*Portable Network Graphics*. A file format that losslessly compresses full color images with transparency (up to 48 bits/pixel) by coding the difference between each pixel's value and a predicted value based on past pixels [8.2.9].
TIFF	Aldus	*Tagged Image File Format*. A flexible file format supporting a variety of image compression standards, including JPEG, JPEG-LS, JPEG-2000, JBIG2, and others.

Basic compression methods

Run-length encoding (RLE)

- Compression is achieved by **eliminating** a simple form of spatial redundancy: **groups of identical intensities**

WWWWWWWWWWWWBWWWWWWWWWWWWBBBWWWWWWWWWWWWWWWWWWWWWWWWBWWWWWWWWWWWWWW

- *Runs of identical intensities* represented by **run-length pairs**
  - Start of a *new intensity value*
  - *Number* of consecutive pixels that have that intensity

WWWWWWWWWWWWBWWWWWWWWWWWWBBBWWWWWWWWWWWWWWWWWWWWWWWWBWWWWWWWWWWWWWW

↓

12W1B12W3B24W1B14W

- Notes
  - Developed in the 1950s, initially used for **FAX** transmissions
  - Particularly effective with **binary images**
    - Only two possible intensities (black and white)
    - Can be represented by a sequence of *lengths only*
  - If there are few runs of identical pixels, RLE results in **data expansion**, i.e. $C<1$

Run-length encoding (RLE)

- Compression is achieved by **eliminating** a simple form of spatial redundancy: **groups of identical intensities**

  WWWWWWWWWWWWBWWWWWWWWWWWWBBBWWWWWWWWWWWWWWWWWWWWWWWWBWWWWWWWWWWWWWW

- *Runs o* **pairs**
  - Start c
  - *Numb*

  wwwwv wwwwwwwwwww

LOSSLESS or LOSSY?

- Notes
  - Developed in the 1950s, initially used for **FAX** transmissions
  - Particularly effective with **binary images**
    - Only two possible intensities (black and white)
    - Can be represented by a sequence of *lengths only*
  - If there are few runs of identical pixels, RLE results in **data expansion**, i.e. $C<1$

Huffman coding (1/3)

- Very popular technique for **removing coding redundancy**
  - Developed in **1952**, but still used in many *advanced compression algorithms*
  - Based on the **probability of occurrence** of each symbol in the source data

- **First step**: create a series of *source reductions*
  - **Sort the probabilities** of the symbols under consideration
  - **Combine the lowest probability symbols** into a single symbol which replaces them in the next source reduction
    - Need to *sort the probabilities* of each reduced source from the most to the least probable

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				

1.0

NB: this procedure generates a **tree**

Huffman coding

Second step: *encode* each reduced source

- **Start** with the smallest source and **work back** to the original source
- **At each branch**, assign “0” to one subtree and “1” to the other
  - The assignment is arbitrary, i.e. reversing the order of the “0” and “1” would work just as well

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								

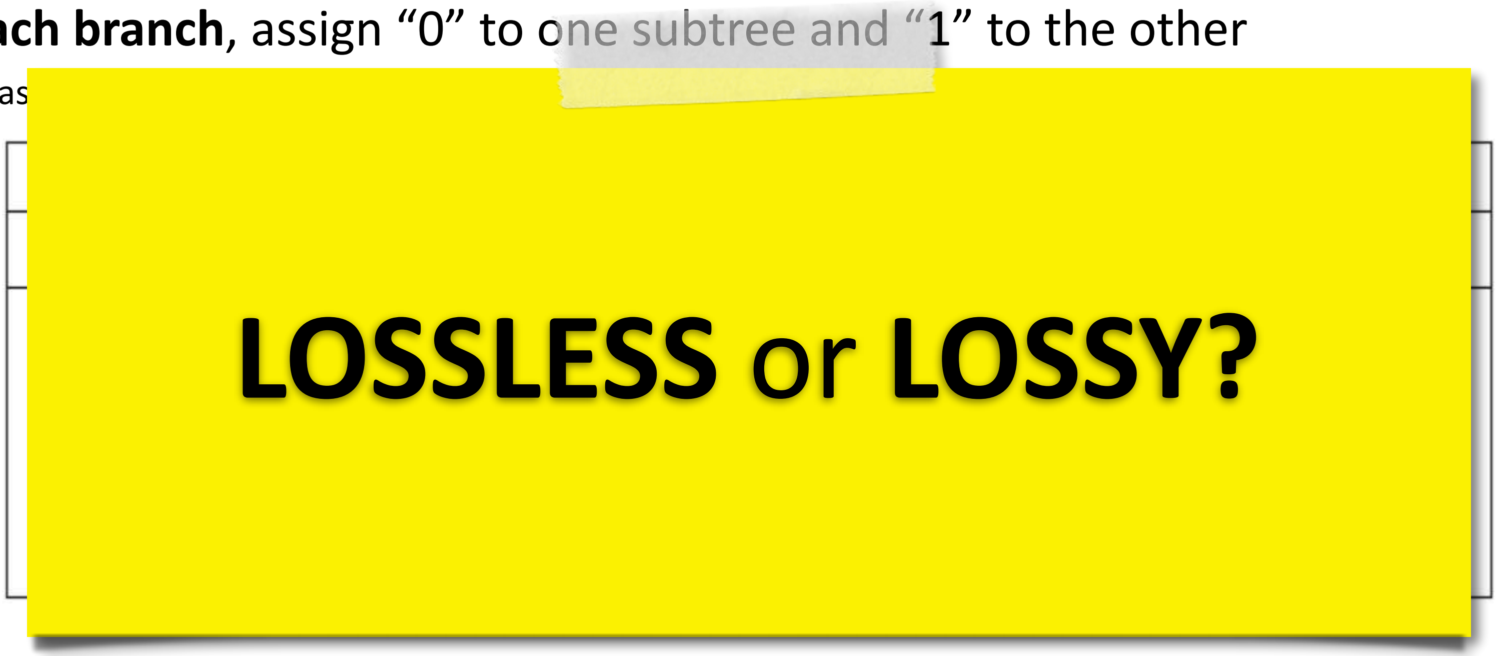
Notes

- $L_{avg} = 0.4*1 + 0.3*2 + 0.1*3 + 0.1*4 + 0.06*5 + 0.04*5 = 2.2$ bits/pixel
- **Optimal code** when symbols are coded *one at a time*, i.e. no prediction or advanced techniques
- When a **large number of symbols** is to be coded, *optimal coding* is nontrivial
  - J source symbols ➜ J-2 source reductions and J-2 code assignments

Huffman coding (2/3)

Second step: *encode* each reduced source

- **Start** with the smallest source and **work back** to the original source
- **At each branch**, assign "0" to one subtree and "1" to the other
  - The as

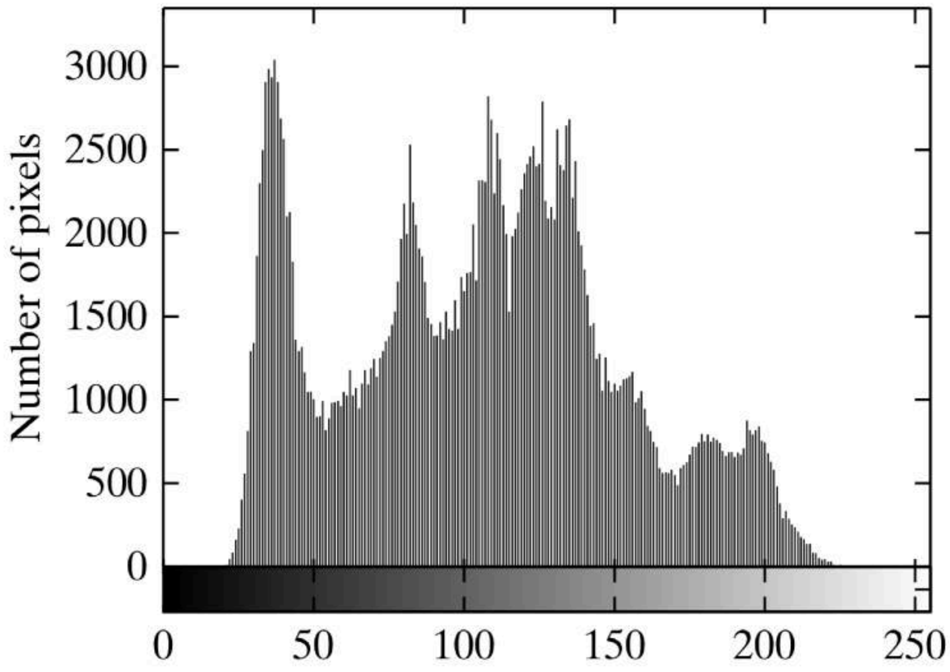


Notes

- $L_{avg} = 0.4*1 + 0.3*2 + 0.1*3 + 0.1*4 + 0.06*5 + 0.04*5 = 2.2$ bits/pixel
- **Optimal code** when symbols are coded *one at a time*, i.e. no prediction or advanced techniques
- When a **large number of symbols** is to be coded, *optimal coding* is nontrivial
  - J source symbols ➜ J-2 source reductions and J-2 code assignments

Huffman coding (3/3)

- Example



  - 512x512x8 bits image
  - A *given implementation* of Huffman coding achieves $L_{avg} = 7.428$ bits/pixel

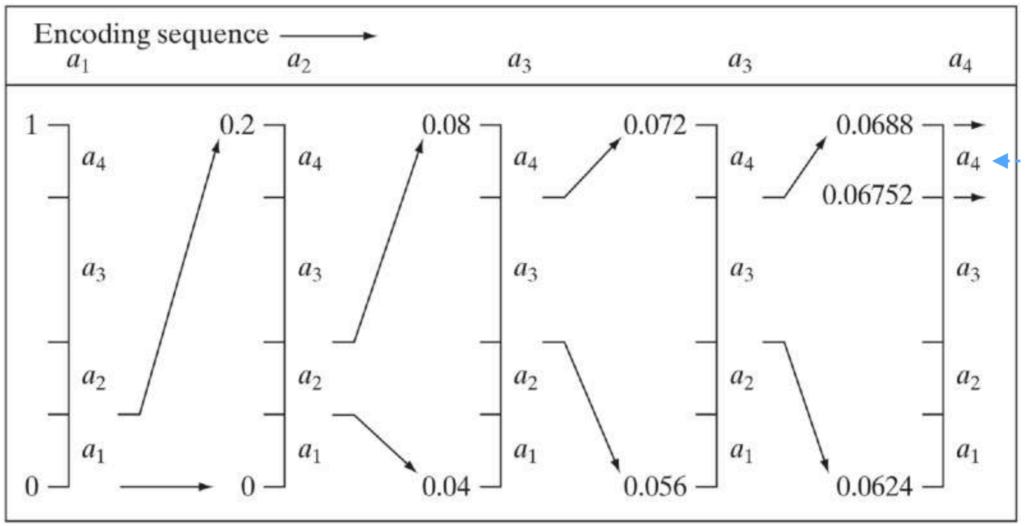
- **NB**: *JPEG* specifies an **ad-hoc Huffman coding table** that has been pre-computed based on experimental data

Arithmetic coding

- **Based on probabilities, but uses a different approach**
  - **No 1-to-1 correspondence** between source symbols and code words
  - An **entire sequence of source symbols** is assigned a single arithmetic code word
    - The *code word* itself defines an **interval of real numbers** between 0 and 1
    - Each source symbol **reduces the size of the interval** in accordance with its probability of occurrence
    - At the beginning, the message is associated to the *full interval* [0, 1)

- **Example**: encoding the source sequence "$a_1\ a_2\ a_3\ a_3\ a_4$"

Source Symbol	Probability
a_1	0.2
a_2	0.2
a_3	0.4
a_4	0.2



Any number within this subinterval can be used to **represent the message**

- **NB**: *message* increases ➜ *interval* smaller ➜ *more bits* needed

Arithmetic coding

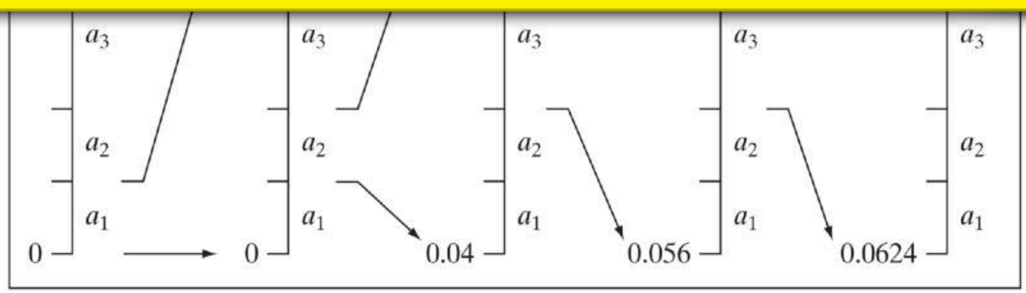
- Based on probabilities, but uses a **different approach**
  - **No 1-to-1 correspondence** between source symbols and code words
  - An **entire sequence of source symbols** is assigned a single arithmetic code word
    - The *co*
    - Each s
    - At the

- **Examp** ”

Source Symbol
a_1
a_2
a_3
a_4

LOSSLESS or **LOSSY?**

umber within this erval can be used **esent the message**



- **NB**: *message* increases ➜ *interval* smaller ➜ *more bits* needed

- Assigns *fixed-length code words* to *variable-length sequences of source symbols* by **building a codebook** (or dictionary)
- **Scan the input string** for *successively longer substrings* until it finds one that is **not in the dictionary**
- Longer strings that are added to the dictionary are made available for **future encoding as single output values**

1) *Initialize* the dictionary to contain all the source symbols

a_1	a_2	a_3	…	…	a_n	-	-	-	-	-
0	1	2	…	…	n-1	n	n+1	…	…	…

source symbols | empty

2) **Find the longest string** *W* in the dictionary that matches the current input
   i.e. (*W+next symbol*) is not in the dictionary
3) **Emit the dictionary index** (i.e. its *code*) for *W* to the *output*
4) **Add to the dictionary** the string (*W+next symbol*)
5) **Remove** *W* from the *input*

repeat

Example

- 4x4x8 bits *image*
- 512-word *dictionary*

39	39	126	126
39	39	126	126
39	39	126	126
39	39	126	126

Dictionary Location	Entry
0	0
1	1
⋮	⋮
255	255
256	—
⋮	⋮
511	—

Currently Recognized Sequence	Pixel Being Processed	Encoded Output	Dictionary Location (Code Word)	Dictionary Entry
	39			
39	39	39	256	39-39
39	126	39	257	39-126
126	126	126	258	126-126
126	39	126	259	126-39
39	39			
39-39	126	256	260	39-39-126
126	126			
126-126	39	258	261	126-126-39
39	39			
39-39	126			
39-39-126	126	260	262	39-39-126-126
126	39			
126-39	39	259	263	126-39-39
39	126			
39-126	126	257	264	39-126-126
126		126		

Example

- 4x4x8 bits *image*
- 512-word *dictionary*

39	39	126	126
39	39	126	126
39	39	126	126
39	39	126	126

Dictionary Location	Entry
0	0
1	1
⋮	⋮
255	255
256	—
⋮	⋮
511	—

LOSSLESS or **LOSSY?**

126-126	39	258	261	126-126-39
39	39			
39-39	126			
39-39-126	126	260	262	39-39-126-126
126	39			
126-39	39	259	263	126-39-39
39	126			
39-126	126	257	264	39-126-126
126		126		

LZW coding (3/4)

Notes

- **Does not requires a priori knowledge** of the probabilities of the source symbols
- To **decode**, the dictionary must be known
  - This can be source of *overhead*
- The **size of the dictionary** is very important
  - too *small* ➜ the detection of matching sequences is *less likely*
  - too *large* ➜ the *size of code words* will adversely affect compression performance
- Many **variants** exist
  - e.g. LZMW, LZAP, LZWL…
- Used a a base to develop more advanced algorithms
  - e.g. zip, gzip…
- LZW was **patented by Unisys** in 1985
  - Many file format used it anyway, e.g. **GIF**, TIFF, PDF
  - Controversy/lawsuit over the licensing agreement between *Unisys* and *CompuServe* (developed GIF)
  - **PNG** file format invented to get around this LZW licensing requirements
  - The patent *expired* in 2004

LZW vs Huffman

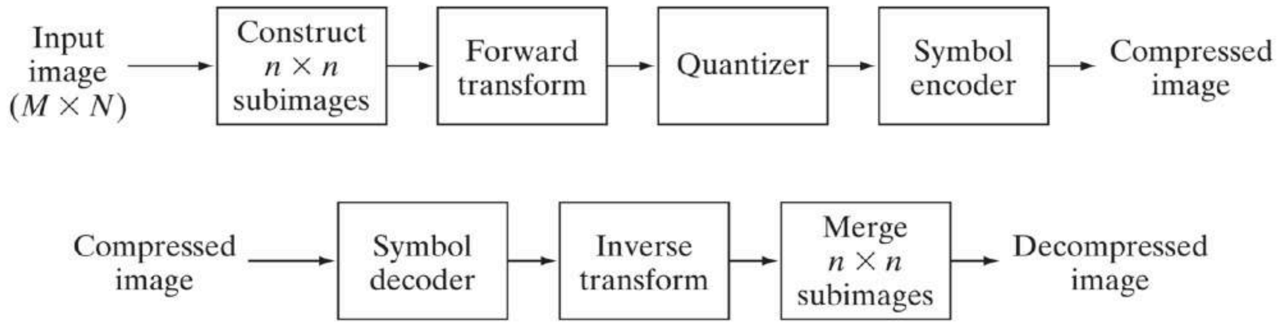
- 512x512x8 bits image
- **Raw size**
  - 512*512*8 bits = 262144 bytes
- **TIFF (no compression)**
  - 286740 bytes (262144 data + 24596 overhead)
- **TIFF + LZW**
  - 224420 bytes
  - $C = 1.28$
- **Huffman coding**
  - $C = 1.077$

The additional compression realized by LZW is due to the **removal of some of the image's spatial redundancy**

- Recall that *Huffman* removes only **coding redundancies**

Block transform coding (1/2)

Divide image into **small non-overlapping blocks** of equal size



- A **reversible, linear transform** is applied to each block, e.g. FFT
- For most images:
  - a significant number of the coefficients have **small magnitudes**, or
  - they correspond to **details not detectable** by human eye
- These can be **coarsely quantized** (or discarded entirely) with **little image distortion**
  e.g. recall the effect of discarding high frequencies in FFT

The **goal of the transformation** process is

- **Decorrelate the pixels** of each block
- **Pack as much information** as possible into the smallest number of coefficients

Block transform coding (2/2)

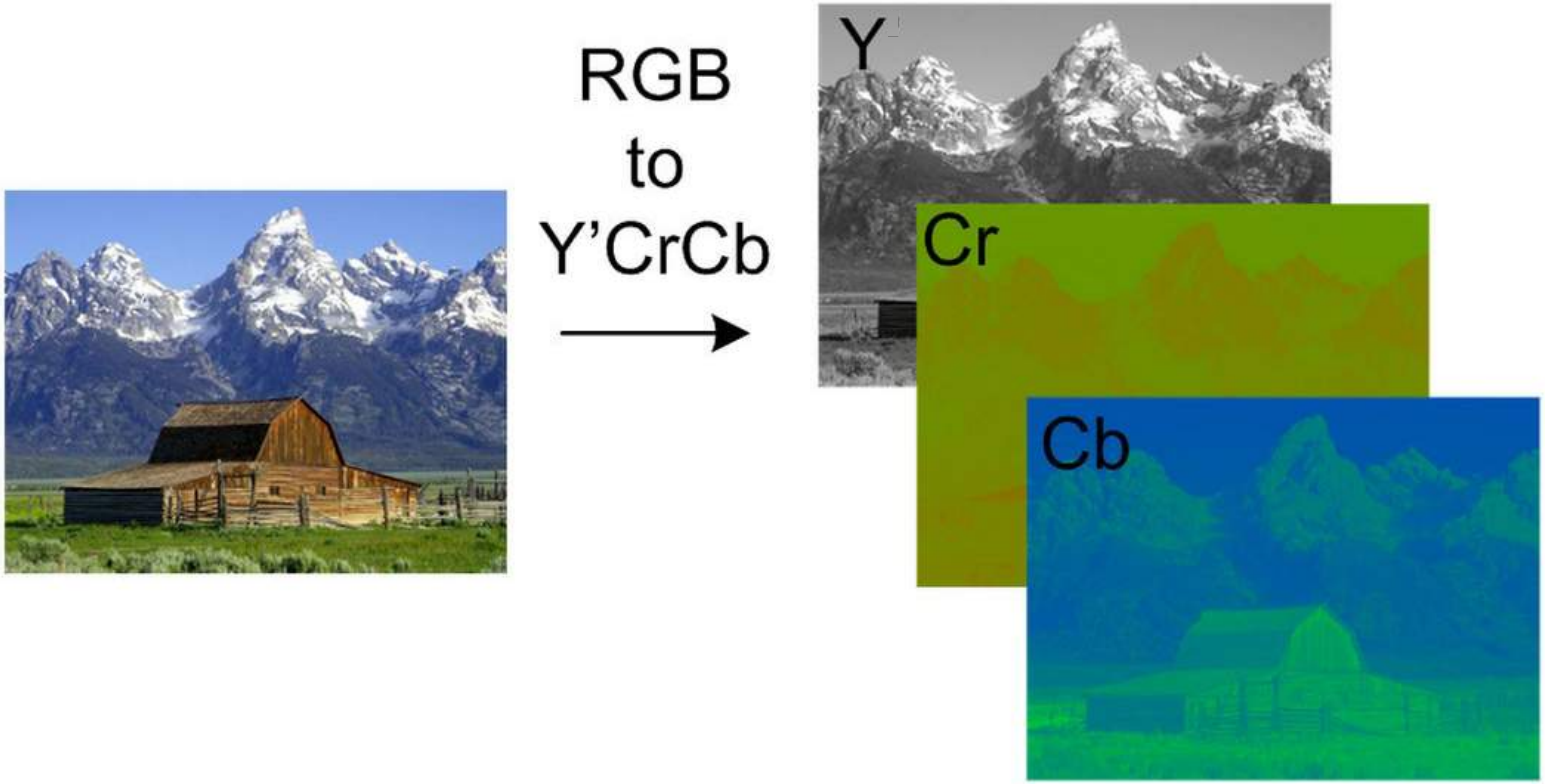
- The **choice of a particular transform** depends on:
  - The *specific application* or image content
  - The amount of *reconstruction error* that can be tolerated
  - The *computational resources* available

- **Compression is achieved during** the *quantization* of the transformed coefficients (not during the *transformation step*)

- One of the **most popular** compression standards is the **JPEG**
  - JPEG (*Joint Photographic Expert Group*) was standardized in 1992
  - There is both **lossless and lossy compression** in JPEG, working on different principles
  - **First generation** of *JPEG* (.jpg) uses the *Discrete Cosine Transform* (DCT)
  - **Second generation** *JPEG2000* (.jp2) uses the *Wavelet Transform*

JPEG algorithm (first generation)

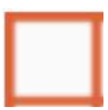
- Image first converted from RGB to **YC_bC_r color space**
  - **Y** is the **luma component** (i.e. brightness of the image)
  - **C_b** and **C_r** are the *blue-difference* and *red-difference* **chroma components**



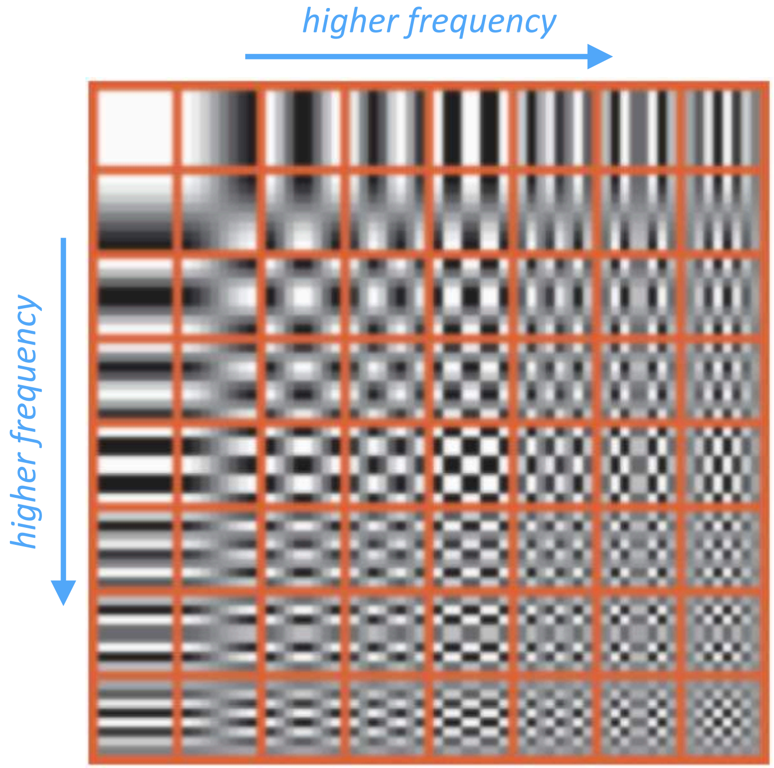
- Then, **C_b, C_r** matrices are stored with the **half resolution** of **Y**

Each channel is transformed according to the DCT

- Similar to the *Fourier transform*
- The 2-D **Discrete Cosine Transform** (DCT) is a linear transform that represents a block of values as combination of **sampled cosine functions at various frequencies**
- Each 8×8 block of the image is expressed as a **weighted sum** of the **DCT basis functions**
  - 64 weights are obtained
- Example

 = 1329 ·  + 234 ·  - 18 ·  + ……

+ ……………………… - 2 ·  + 1 · 



DCT basis functions

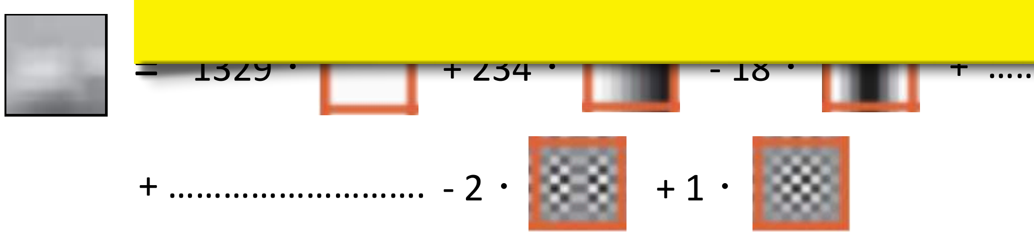
Finally, these DCT weights are quantized

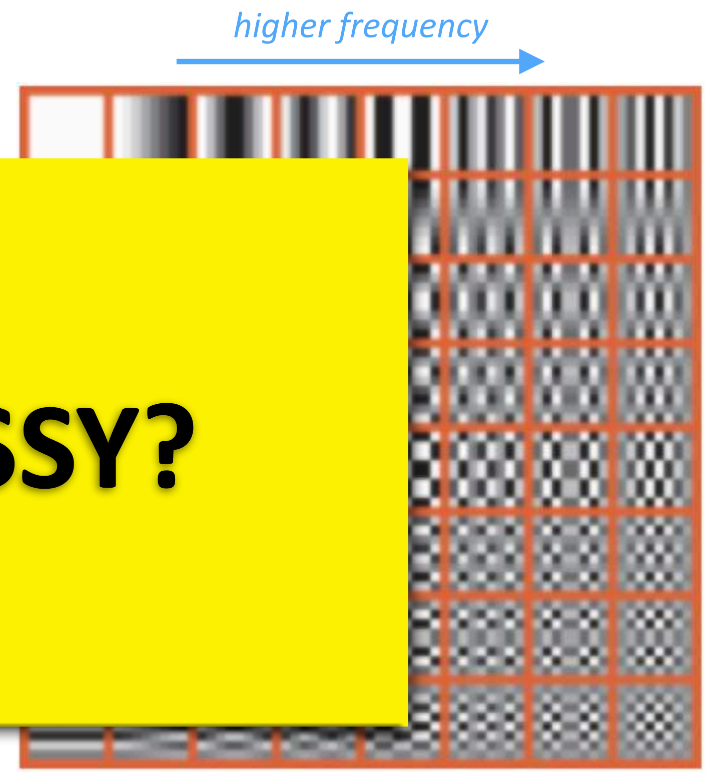
- The *threshold value* provides the **degree of compression**

Each channel is transformed according to the **DCT**

- Similar to the *Fourier transform*
- The 2-D **Discrete Cosine Transform** (DCT) is a lin block cosine
- Each 8 a weig
  - 64 wei
- Examp

LOSSLESS or **LOSSY?**





DCT basis functions

Finally, these DCT **weights are quantized**

- The *threshold value* provides the **degree of compression**

JPEG algorithm (first generation) (3/4)

Example 1

current block to compress

block intensities

	1	2	3	4	5	6	7	8
1	185	187	184	183	189	186	185	186
2	185	184	186	190	187	186	189	191
3	186	187	187	188	190	185	189	191
4	186	189	189	189	193	193	193	195
5	185	190	188	193	199	198	189	184
6	191	187	162	156	116	30	15	14
7	168	102	49	22	15	11	10	10
8	25	19	19	26	17	11	10	10

JPEG algorithm (first generation) (3/4)

block intensities

	1	2	3	4	5	6	7	8
1	185	187	184	183	189	186	185	186
2	185	184	186	190	187	186	189	191
3	186	187	187	188	190	185	189	191
4	186	189	189	189	193	193	193	195
5	185	190	188	193	199	198	189	184
6	191	187	162	156	116	30	15	14
7	168	102	49	22	15	11	10	10
8	25	19	19	26	17	11	10	10

coefficients of the block's DCT

	1	2	3	4	5	6	7	8
1	1117	114	10	7	19	−2	−7	2
2	459	−119	−20	−11	−16	−4	3	0
3	−267	−3	24	8	1	6	4	−1
4	50	107	−9	−1	11	−6	−7	3
5	52	−111	−22	−2	−16	−2	5	−3
6	−38	39	46	19	2	0	4	3
7	−17	39	−46	−26	8	−5	−10	2
8	30	−46	28	22	−9	2	7	−1

JPEG algorithm (first generation) (3/4)

100% of larger DCT coefficients

50% of larger DCT coefficients

20% of larger DCT coefficients

5% of larger DCT coefficients

Example 2

C=12 C=19 C=30

C=49 C=85 C=182

Example 2

1117	114	10	7	19	−2	−7	2
459	−119	−20	−11	−16	−4	3	0
−267	−3	24	8	1	6	4	−1
50	107	−9	−1	11	−6	−7	3
52	−111	−22	−2	−16	−2	5	−3
−38	39	46	19	2	0	4	3
−17	39	−46	−26	8	−5	−10	2
30	−46	28	22	−9	2	7	−1