COMPRESSION AND CODING II

Ole-Johan Skrede

03.05.2017

INF2310 - Digital Image Processing

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

After original slides by Andreas Kleppe

TODAY'S LECTURE

- Difference transform

TODAY'S LECTURE

- Difference transform
- Run-length transform

- Difference transform
- Run-length transform
- LZW-compression

TODAY'S LECTURE

- Difference transform
- Run-length transform
- LZW-compression
- JPEG compression

- Difference transform
- Run-length transform
- LZW-compression
- JPEG compression
- Lossless predictive coding

TODAY'S LECTURE

- Difference transform
- Run-length transform
- LZW-compression
- JPEG compression
- Lossless predictive coding
- Sections from the compendium:
  - 18.4 *Noen transformer som brukes i kompresjon*
  - 18.7.3 *Lempel-Ziv-Welch (LZW) algoritmen*
  - 18.8.0 *Koding med informasjonstap*
  - 18.8.1 *JPEG*

INTRODUCTION AND REPETITION

REPETITION: COMPRESSION

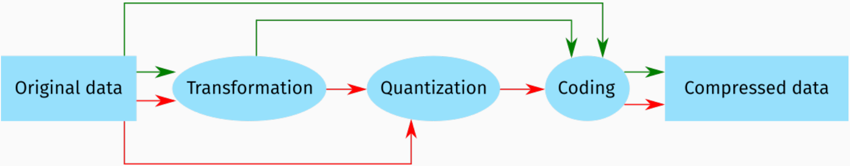


Figure 1: Three steps of compression. Green arrows: lossless, red arrows: lossy

- 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.

REPETITION: COMPRESSION

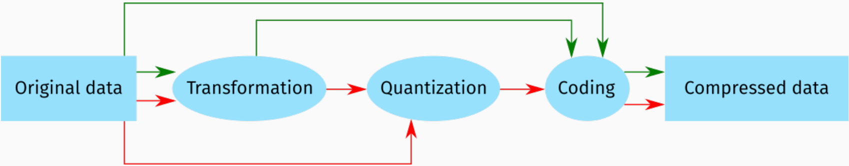


Figure 1: Three steps of compression. Green arrows: lossless, red arrows: lossy

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

REPETITION: DIFFERENT TYPES OF REDUNDANCIES

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

REPETITION: DIFFERENT TYPES OF REDUNDANCIES

- *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.

REPETITION: DIFFERENT TYPES OF REDUNDANCIES

- *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.

REPETITION: DIFFERENT TYPES OF REDUNDANCIES

- *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.

Types of redundance →	Psycho-visual	Inter-pixel temporal	Inter-pixel spatial	Coding
Shannon-Fano coding				✓
Huffman coding				✓
Arithmetic coding				✓
Lossless predicative coding in time		✓		
Lossless JPEG			✓	✓
Lossy JPEG	✓		✓	✓
Defference transform			✓	
Run-length transform			✓	
LZW transform			✓	✓

SOME TRANSFORMS

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

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

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

- 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 $[-(2^b - 1), 2^b - 1]$.

- 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

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

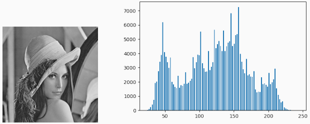
  for all rows $x \in \{0, 1, \ldots, m-1\}$.
- Note that for an image f taking values in $[0, 2^b - 1]$, values of the transformed image g take values in $[-(2^b - 1), 2^b - 1]$.
- This means that we need to use $b + 1$ bits for each $g(x, y)$ if we are going to use equal-size codeword for every value.

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

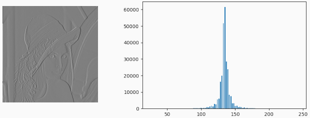
$$g[x, y] = f[x, y] - f[x, y-1], \quad y \in \{1, 2, \ldots, n-1\} \qquad (1)$$

  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.
- 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 2: $H \approx 7.45 \implies c_r \approx 1.1$



(a) Difference transformed (b) Graylevel intensity histogram

Figure 3: $H \approx 5.07 \implies c_r \approx 1.6$

Algorithm 1 Forward difference transform

```
procedure ForwardDiff(f)                          ▷ f is an image of shape m × n
    g ← 0                                         ▷ Difference image with same shape as f
    for r ∈ {0, 1, . . . , m} do
        g[r, 0] ← f[r, 0]
        for c ∈ {1, 2, . . . , n − 1} do
            g[r, c] ← f[r, c] − f[r, c − 1]
        end for
    end for
    return g
end procedure
```

Algorithm 2 Backward difference transform

procedure BACKWARDDIFF(g) ▷ g is an image of shape $m \times n$
  $h \leftarrow 0$ ▷ Image with same shape as g
  for $r \in \{0, 1, \ldots, m\}$ **do**
    $h[r, 0] \leftarrow g[r, 0]$
    for $c \in \{1, 2, \ldots, n-1\}$ **do**
      $h[r, c] \leftarrow g[r, c] + h[r, c-1]$
    end for
  end for
  return h
end procedure

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.

- 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.

BIT SLICING

- Bit slicing is extracting the value of a bit at a certain position.

BIT SLICING

- Bit slicing is extracting the value of a bit at a certain position.
- We will look at two different ways of doing this.

BIT SLICING

- Bit slicing is extracting the value of a bit at a certain position.
- We will look at two different ways of doing this.
- Let v be the value we want to extract bit values from, and n denote the bit position, starting from $n = 0$ at the least significant bit (LSB) to the most significant bit (MSB).

BIT SLICING

- Bit slicing is extracting the value of a bit at a certain position.
- We will look at two different ways of doing this.
- Let v be the value we want to extract bit values from, and n denote the bit position, starting from $n = 0$ at the least significant bit (LSB) to the most significant bit (MSB).
- As an example, $10_{10} = 1010_2$ has values $[1, 0, 1, 0]$ at $n = [3, 2, 1, 0]$.
- Let b be the bit value at position n in v.
- Let us use python-syntax for bitwise operators:
    - $\&$: Bitwise and: $1100_2 \& 1010_2 = 1000$, therefore $12_{10} \& 10_{10} = 8$.
    - $//$: Integer division: $23//4 = 5$ since $23/4 = 5 + 3/4$.
    - $\ll$: Left bit-shift: $10_{10} \ll 3_{10} = 80_{10}$ since $1010_2 \ll 3_{10} = 1010000_2$.

Method 1

$$b = v//2^n \mod 2.$$

If the result of $v//2^n$ is odd, $b = 1$, if it is even, $b = 0$.

Method 1

$$b = v//2^n \mod 2.$$

If the result of $v//2^n$ is odd, $b = 1$, if it is even, $b = 0$.

Method 2

$$b = [v\&(1 \ll n)] > 0.$$

$(1 \ll n)$ in binary is a 1 followed by n zeros. Therefore will a bitwise and operation on some number be 0 unless it has a bit value of 1 at position n.

Method 1

$$b = v//2^n \mod 2.$$

If the result of $v//2^n$ is odd, $b = 1$, if it is even, $b = 0$.

Method 2

$$b = [v\&(1 \ll n)] > 0.$$

$(1 \ll n)$ in binary is a 1 followed by n zeros. Therefore will a bitwise and operation on some number be 0 unless it has a bit value of 1 at position n.

Table 1: Example with $v = 234_{10} = 11101010_2$

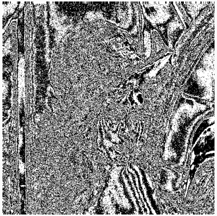
	Method 1		Method 2	
n	$v//2^n$	b	$v\&(1 \ll n)$	b
7	1	1	128	1
6	3	1	64	1
5	7	1	32	1
4	14	0	0	0
3	29	1	8	1
2	58	0	0	0
1	117	1	2	1
0	234	0	0	0

(a) $n = 7$

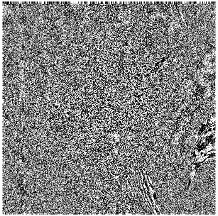
(b) $n = 6$

(c) $n = 5$

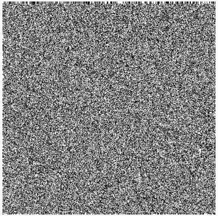
(d) $n = 4$



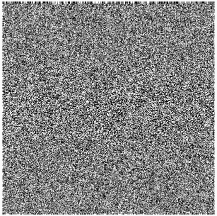
(e) $n = 3$



(f) $n = 2$



(g) $n = 1$



(h) $n = 0$

- Every codeword is of equal length.
- The code of each symbol is the binary representation of the symbol's (zero indiced) index.

- Every codeword is of equal length.
- The code of each symbol is the binary representation of the symbol's (zero indiced) index.
- Example: A 3-bits natural code has 8 possible values.

Symbol	`a`	`b`	`c`	`d`	`e`	`f`	`g`	`h`
Index	0	1	2	3	4	5	6	7
Codeword	`000`	`001`	`010`	`011`	`100`	`101`	`110`	`111`

- An alternative binary representation (or coding).

GRAY CODE

- An alternative binary representation (or coding).
- Close numbers need not have many bit values in common, e.g. $127_{10} = 01111111_2$ and $128_{10} = 10000000$.

GRAY CODE

- An alternative binary representation (or coding).
- Close numbers need not have many bit values in common, e.g. $127_{10} = 01111111_2$ and $128_{10} = 10000000$.
- Since neighbouring pixels often have similar values, this means that natural binary coded images often have high bit-plane complexity.

- An alternative binary representation (or coding).
- Close numbers need not have many bit values in common, e.g. $127_{10} = 01111111_2$ and $128_{10} = 10000000$.
- Since neighbouring pixels often have similar values, this means that natural binary coded images often have high bit-plane complexity.
- Sometimes, e.g. in run-length coding, this is not desired.

GRAY CODE

- An alternative binary representation (or coding).
- Close numbers need not have many bit values in common, e.g. $127_{10} = 01111111_2$ and $128_{10} = 10000000$.
- Since neighbouring pixels often have similar values, this means that natural binary coded images often have high bit-plane complexity.
- Sometimes, e.g. in run-length coding, this is not desired.
- In Gray code, only one bit value is changed between adjacent integer values.

- An alternative binary representation (or coding).
- Close numbers need not have many bit values in common, e.g. $127_{10} = 01111111_2$ and $128_{10} = 10000000$.
- Since neighbouring pixels often have similar values, this means that natural binary coded images often have high bit-plane complexity.
- Sometimes, e.g. in run-length coding, this is not desired.
- In Gray code, only one bit value is changed between adjacent integer values.
- The codewords in natural binary coding and Gray code are of equal length, the only difference is *what codeword is assigned to what value*.

Algorithm 3 Natural to Gray coding transform

```
procedure NATTOGRAY(n)                                  ▷ n list of naturally coded bits
    g ← []                                              ▷ Initialize Gray list to empty
    c ← false                   ▷ Boolean that decides whether to complement or not
    for b ∈ n do
        if c then
            g ← [1 − b]                                 ▷ Append to g
        else
            g ← [b]
        end if
        if b == 1 then
            c ← true
        else
            c ← false
        end if
    end for
    return g
end procedure
```

Algorithm 4 Gray to Natural coding transform

```
procedure GrayToNat(g)                          ▷ g list of Gray coded bits
    n ← []                                      ▷ Initialize natural list to empty
    c ← false                                   ▷ Boolean that decides whether to complement or not
    for b ∈ g do
        if c then
            n ← [1 − b]                         ▷ Append to g
        else
            n ← [b]
        end if
        if b == 1 then
            c ← ¬c                              ▷ Switch value of c
        end if
    end for
    return n
end procedure
```

GRAY CODE EXAMPLE

Decimal	Gray code	Natural code
0	0000	0000
1	0001	0001
2	0011	0010
3	0010	0011
4	0110	0100
5	0111	0101
6	0101	0110
7	0100	0111
8	1100	1000
9	1101	1001
10	1111	1010
11	1110	1011
12	1010	1100
13	1011	1101
14	1001	1110
15	1000	1111

- The figures below show bit planes from the MSB (left) to LSB (right).
- The MSB is always equal in the two representations.
- The Gray code representation has typically fewer ”noise planes”, which implies that run-length transforms can compress more.

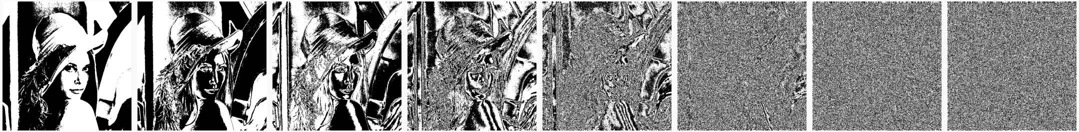


Figure 5: Natural binary representation

Figure 6: Gray code representation

- 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.

- 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.

Given a message W that is to be encoded, the encoding is as follows.

1. Initialize the dictionary from an alphabeth, e.g. $D = \{\# : 0, a : 1, b : 2, \ldots\}$ (where $\#$ is an end-of-message symbol).

Given a message W that is to be encoded, the encoding is as follows.

1. Initialize the dictionary from an alphabeth, e.g. $D = \{\# : 0, a : 1, b : 2, \ldots\}$ (where $\#$ is an end-of-message symbol).
2. Initialize the current input w, to be the first symbol in W.

Given a message W that is to be encoded, the encoding is as follows.

1. Initialize the dictionary from an alphabeth, e.g. $D = \{\# : 0, a : 1, b : 2, \ldots\}$ (where $\#$ is an end-of-message symbol).
2. Initialize the current input w, to be the first symbol in W.
3. Find the longest string $d \in D$ that matches the current input w.

LEMPEL-ZIV-WELCH: ENCODING

Given a message W that is to be encoded, the encoding is as follows.

1. Initialize the dictionary from an alphabeth, e.g. $D = \{\# : 0, a : 1, b : 2, \ldots\}$ (where $\#$ is an end-of-message symbol).
2. Initialize the current input w, to be the first symbol in W.
3. Find the longest string $d \in D$ that matches the current input w.
4. Output (or send) the codeword of d, $D[d]$.

Given a message W that is to be encoded, the encoding is as follows.

1. Initialize the dictionary from an alphabeth, e.g. $D = \{\# : 0, a : 1, b : 2, \ldots\}$ (where $\#$ is an end-of-message symbol).
2. Initialize the current input w, to be the first symbol in W.
3. Find the longest string $d \in D$ that matches the current input w.
4. Output (or send) the codeword of d, $D[d]$.
5. Create a new string and append it to D, this new string is created as the current string d concatenated with the next symbol in the input message W.

Given a message W that is to be encoded, the encoding is as follows.

1. Initialize the dictionary from an alphabeth, e.g. $D = \{\# : 0, a : 1, b : 2, \ldots\}$ (where $\#$ is an end-of-message symbol).
2. Initialize the current input w, to be the first symbol in W.
3. Find the longest string $d \in D$ that matches the current input w.
4. Output (or send) the codeword of d, $D[d]$.
5. Create a new string and append it to D, this new string is created as the current string d concatenated with the next symbol in the input message W.
6. This new dictionary entry will get the next codeword not in use.

Given a message W that is to be encoded, the encoding is as follows.

1. Initialize the dictionary from an alphabeth, e.g. $D = \{\# : 0, a : 1, b : 2, \ldots\}$ (where $\#$ is an end-of-message symbol).
2. Initialize the current input w, to be the first symbol in W.
3. Find the longest string $d \in D$ that matches the current input w.
4. Output (or send) the codeword of d, $D[d]$.
5. Create a new string and append it to D, this new string is created as the current string d concatenated with the next symbol in the input message W.
6. This new dictionary entry will get the next codeword not in use.
7. Set current symbol w to be the next string in W that is also in D.

Given a message W that is to be encoded, the encoding is as follows.

1. Initialize the dictionary from an alphabeth, e.g. $D = \{\# : 0, a : 1, b : 2, \ldots\}$ (where $\#$ is an end-of-message symbol).
2. Initialize the current input w, to be the first symbol in W.
3. Find the longest string $d \in D$ that matches the current input w.
4. Output (or send) the codeword of d, $D[d]$.
5. Create a new string and append it to D, this new string is created as the current string d concatenated with the next symbol in the input message W.
6. This new dictionary entry will get the next codeword not in use.
7. Set current symbol w to be the next string in W that is also in D.
8. Unless $w = \#$, go to 3.

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

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

- Decode the encoded string codeword by codeword.

[1] See next page

- Decode the encoded string codeword by codeword.
- Build the dictionary by decoding the current codeword and concatenate this encoded string with:
    - If the next codeword can be decoded (it is already in the dictionary): The first character of the *next decoded string*.
    - If the next codeword is not in the dictionary: The first character of the *current decoded string*[1].

[1]See next page

LEMPEL-ZIV-WELCH: DECODING

- Decode the encoded string codeword by codeword.
- Build the dictionary by decoding the current codeword and concatenate this encoded string with:
  - If the next codeword can be decoded (it is already in the dictionary): The first character of the *next decoded string*.
  - If the next codeword is not in the dictionary: The first character of the *current decoded string*[1].
- Processing one codeword at the time, and building the dictionary at the same time, will in the end decode the whole sequence of codewords.

[1]See next page

This explains why it makes sense to attach the first symbol of the current decoded string to the end of the new dictionary entry.

- Let the currently decoded string be X with first symbol x.

This explains why it makes sense to attach the first symbol of the current decoded string to the end of the new dictionary entry.

- Let the currently decoded string be X with first symbol x.
- The new dictionary entry is $X? : n$, where n is the codeword.

This explains why it makes sense to attach the first symbol of the current decoded string to the end of the new dictionary entry.

- Let the currently decoded string be X with first symbol x.
- The new dictionary entry is $X?$: n, where n is the codeword.
- We look for n in our dictionary, but see that it is not there. We know that $?$ should be the first symbol y of the decoded string Y at n, but how do we know what it is?

This explains why it makes sense to attach the first symbol of the current decoded string to the end of the new dictionary entry.

- Let the currently decoded string be X with first symbol x.
- The new dictionary entry is $X? : n$, where n is the codeword.
- We look for n in our dictionary, but see that it is not there. We know that $?$ should be the first symbol y of the decoded string Y at n, but how do we know what it is?
- The first thing to realise is that this only happens if Y was encountered immediately after the creation of $Y : n$ in the encoding.

This explains why it makes sense to attach the first symbol of the current decoded string to the end of the new dictionary entry.

- Let the currently decoded string be X with first symbol x.
- The new dictionary entry is $X?:n$, where n is the codeword.
- We look for n in our dictionary, but see that it is not there. We know that $?$ should be the first symbol y of the decoded string Y at n, but how do we know what it is?
- The first thing to realise is that this only happens if Y was encountered immediately after the creation of $Y:n$ in the encoding.
- Therefore $X? = Y$, and therefore, $y = x$, where x was the first symbol of the string X.

- 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#

- 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.

LOSSY COMPRESSION

- In order to achieve high compression rates, it is often necessary with *lossy compression*.

- In order to achieve high compression rates, it is often necessary with *lossy compression*.
- Note: in this case, the original signal *can not be recovered* because of loss of information.

LOSSY COMPRESSION

- In order to achieve high compression rates, it is often necessary with *lossy compression*.
- Note: in this case, the original signal *can not be recovered* because of loss of information.
- Some simple methods for lossy compression:
  - Requantizing to fewer graylevel intensities.
  - Resampling to lower spatial resolution.
  - Filter based methods, e.g. replacing the values in every non-overlapping $p \times q$ rectangle in an image with the mean or median value of that region.

HOW GOOD IS THE IMAGE QUALITY

- If we use lossy compression, we need to make sure that the result after decompression is *good enough*.

HOW GOOD IS THE IMAGE QUALITY

- If we use lossy compression, we need to make sure that the result after decompression is *good enough*.
- For an $m \times n$ image f, let g be the resulting image after f has been compressed and decompressed. The error is then the difference

$$e(x, y) = f(x, y) - g(x, y).$$

- If we use lossy compression, we need to make sure that the result after decompression is *good enough*.
- For an $m \times n$ image f, let g be the resulting image after f has been compressed and decompressed. The error is then the difference

$$e(x, y) = f(x, y) - g(x, y).$$

- The *root mean square* (RMS) error between the images is

$$RMS = \sqrt{\frac{1}{mn} \sum_{x=1}^{m} \sum_{y=1}^{n} e^2(x, y)}$$

- If we interpret the error as noise, we can define the *mean squared signal to noise ratio* (RMS_{MS}) as

$$SNR_{MS} = \frac{\sum_{x=1}^{m} \sum_{y=1}^{n} g^2(x, y)}{\sum_{x=1}^{m} \sum_{y=1}^{n} e^2(x, y)}$$

HOW GOOD IS THE IMAGE QUALITY, CONT.

- The RMS value of the SNR is then

$$SNR_{RMS} = \sqrt{\frac{\sum_{x=1}^{m} \sum_{y=1}^{n} g^2(x,y)}{\sum_{x=1}^{m} \sum_{y=1}^{n} e^2(x,y)}}$$

HOW GOOD IS THE IMAGE QUALITY, CONT.

- The RMS value of the SNR is then

$$SNR_{RMS} = \sqrt{\frac{\sum_{x=1}^{m} \sum_{y=1}^{n} g^2(x,y)}{\sum_{x=1}^{m} \sum_{y=1}^{n} e^2(x,y)}}$$

- The quality measures above considers all errors in the whole image, and treat them equally.

- The RMS value of the SNR is then

$$SNR_{RMS} = \sqrt{\frac{\sum_{x=1}^{m}\sum_{y=1}^{n} g^2(x,y)}{\sum_{x=1}^{m}\sum_{y=1}^{n} e^2(x,y)}}$$

- The quality measures above considers all errors in the whole image, and treat them equally.
- Our perception does not necessary agree. E.g. small errors over the whole image will get a larger SNR_{MS} than missing or created features. But we will percieve the latter having inferior quality.

- The RMS value of the SNR is then

$$SNR_{RMS} = \sqrt{\frac{\sum_{x=1}^{m}\sum_{y=1}^{n} g^2(x,y)}{\sum_{x=1}^{m}\sum_{y=1}^{n} e^2(x,y)}}$$

- The quality measures above considers all errors in the whole image, and treat them equally.
- Our perception does not necessary agree. E.g. small errors over the whole image will get a larger SNR_{MS} than missing or created features. But we will percieve the latter having inferior quality.
- Often, our desire is that the image quality shall mirror *our perception of the quality of the image*.
- This is especially true for image display purposes.

HOW GOOD IS THE IMAGE QUALITY, CONT.

- An image quality measure that is well aligned with our perception is typically based on several parameter
    - Each parameter should try to indicate how bad a certain compression error trait is.
    - The final image quality measure should be one value that is based on all parameters.

HOW GOOD IS THE IMAGE QUALITY, CONT.

- An image quality measure that is well aligned with our perception is typically based on several parameter
  - Each parameter should try to indicate how bad a certain compression error trait is.
  - The final image quality measure should be one value that is based on all parameters.
- Errors around edges is perceived as bad.

- An image quality measure that is well aligned with our perception is typically based on several parameter
    - Each parameter should try to indicate how bad a certain compression error trait is.
    - The final image quality measure should be one value that is based on all parameters.
- Errors around edges is perceived as bad.
- Errors in the foreground are perceived worse than errors in the background.

HOW GOOD IS THE IMAGE QUALITY, CONT.

- An image quality measure that is well aligned with our perception is typically based on several parameter
    - Each parameter should try to indicate how bad a certain compression error trait is.
    - The final image quality measure should be one value that is based on all parameters.
- Errors around edges is perceived as bad.
- Errors in the foreground are perceived worse than errors in the background.
- Missing or created structures are also bad.

HOW GOOD IS THE IMAGE QUALITY, CONT.

- An image quality measure that is well aligned with our perception is typically based on several parameter
  - Each parameter should try to indicate how bad a certain compression error trait is.
  - The final image quality measure should be one value that is based on all parameters.
- Errors around edges is perceived as bad.
- Errors in the foreground are perceived worse than errors in the background.
- Missing or created structures are also bad.
- The level of compression should probably vary locally in the image.
  - Homogeneous areas should be compressed heavily. These areas carry little information, and few non-zero coefficients in the 2D DFT.
  - Edges, lines and other details should be compressed less. These carry more information, and have more non-zero 2D DFT coefficients.

JPEG

- JPEG (*Joint Photographic Expert Group*) is one of the most common compression methods.

THE JPEG STANDARD

- JPEG (*Joint Photographic Expert Group*) is one of the most common compression methods.
- The JPEG-standard (originally from 1992) has both lossy and lossless variants.

- JPEG (*Joint Photographic Expert Group*) is one of the most common compression methods.
- The JPEG-standard (originally from 1992) has both lossy and lossless variants.
- In both cases, either Huffman- or arithmetic coding is used.

- JPEG (*Joint Photographic Expert Group*) is one of the most common compression methods.
- The JPEG-standard (originally from 1992) has both lossy and lossless variants.
- In both cases, either Huffman- or arithmetic coding is used.
- In the lossless version, *predicative* coding is used.

THE JPEG STANDARD

- JPEG (*Joint Photographic Expert Group*) is one of the most common compression methods.
- The JPEG-standard (originally from 1992) has both lossy and lossless variants.
- In both cases, either Huffman- or arithmetic coding is used.
- In the lossless version, *predicative* coding is used.
- In the lossy version, a *2D discrete cosinus transform (DCT)* is used.

- 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.

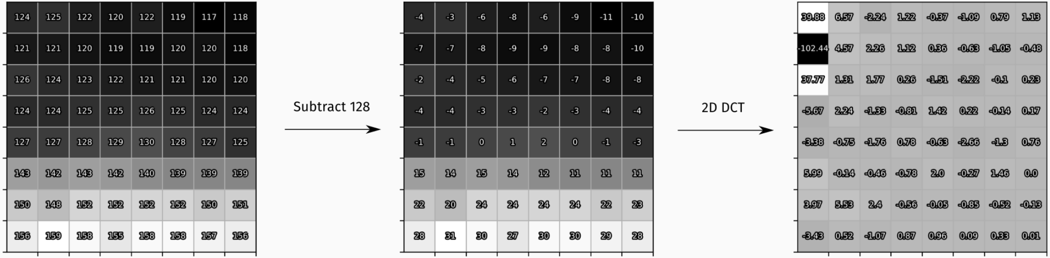


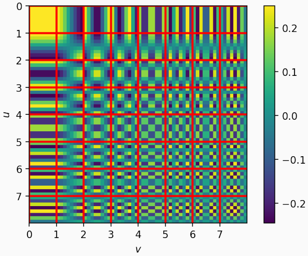
Figure 7: Example block, subtraction by 128, and 2D DCT.

The main ingredient of the JPEG-compression is the 2D discrete cosinus transform (*2D DCT*). For an $m \times n$ image f, the 2D DCT is

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

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

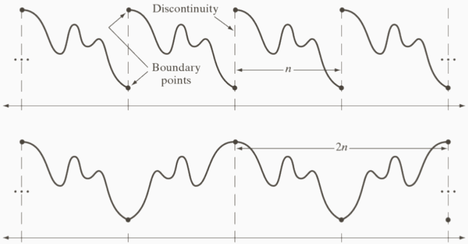
- JPEG only transforms 8×8 tiles at a time.
- Compute 8×8 (from u, v) tiles of size 8×8 (from x, y), of the cosine factor
- Compute 2D DCT coefficients by summing the dot-products of the 8×8 block in the image and every 8×8 tile in the cosine image.



WHY DCT AND NOT DFT?

For a discrete signal with n points will the implicit n-point periodicity of a DFT introduce high frequencies because of boundary-discontinuity. In JPEG, $n = 8$ and 2D, and the boundary is the boundary of the blocks, but the point still stands.

- If we remove these frequencies we introduce heavy block-artifacts.
- If we keep them, we reduce the compression rate compared to DCT, where we often don't need to keep most high frequencies.



DCT is implicitly $2n$-point periodically and symmetric about n, therefore will these hight frequencies *not be introduced*.

- 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.

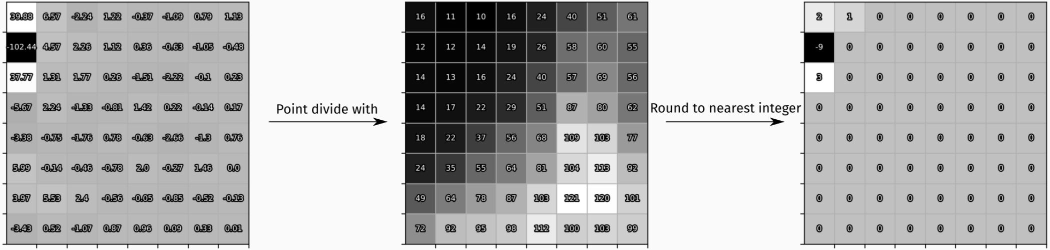


Figure 8: Divide the DCT block (left) with the quantization matrix (middle) and round to nearest integer (right)

1. The AC-components are zig-zag scanned:
    - The elements are ordered in a 1D sequence.
    - The absolute value of the elements will mostly decsend 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.

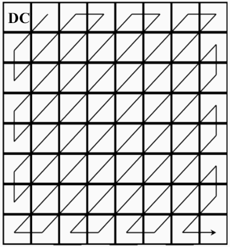


Figure 9: Zig-zag gathering of AC-components into a sequence.

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

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.

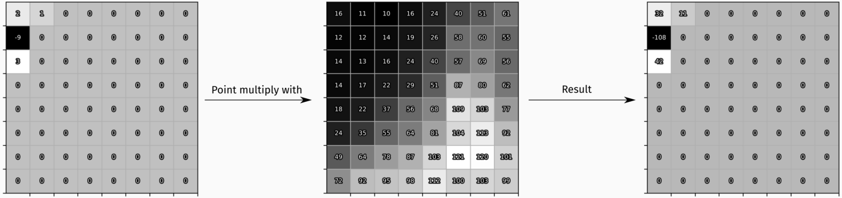


Figure 10: Multiply the quantized DCT components (left) with the quantization matrix (middle) to produce the sparse frequency-domain block (right).

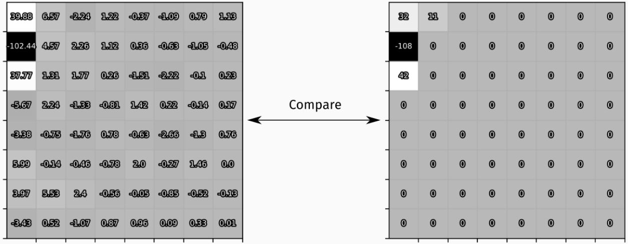


Figure 11: Comparison of the original 2D DCT components (left) and the restored (right)

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

LOSSY JPEG DECOMPRESSION: QUALITY OF RESTORED DCT IMAGE

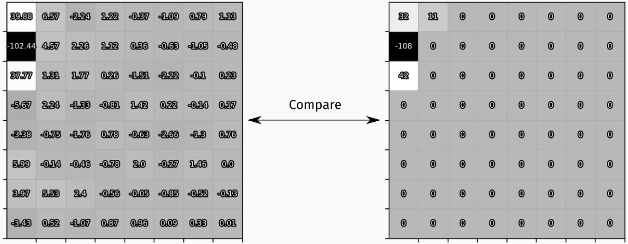


Figure 11: Comparison of the original 2D DCT components (left) and the restored (right)

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

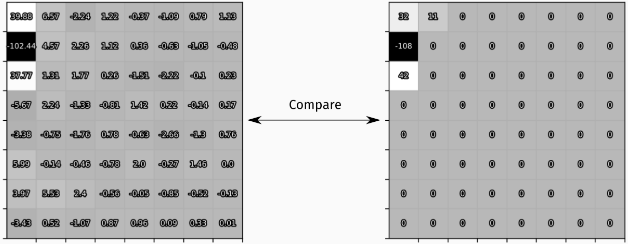


Figure 11: Comparison of the original 2D DCT components (left) and the restored (right)

- 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.

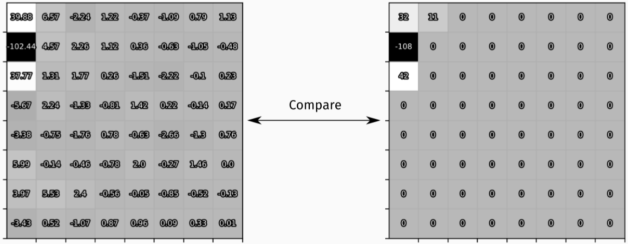


Figure 11: Comparison of the original 2D DCT components (left) and the restored (right)

- 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.

LOSSY JPEG DECOMPRESSION: INVERSE 2D DCT

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

$$f(x,y) = \frac{2}{\sqrt{mn}} \sum_{u=0}^{m} \sum_{v=0}^{n} c(u)c(v)F(u,v) \cos\left(\frac{(2x+1)u\pi}{2m}\right) \cos\left(\frac{(2y+1)v\pi}{2n}\right), \tag{5}$$

where

$$c(a) = \begin{cases} \frac{1}{\sqrt{2}} & \text{if } a = 0, \\ 1 & \text{otherwise.} \end{cases} \tag{6}$$

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

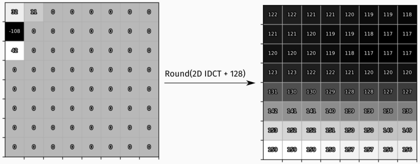


Figure 12: A 2D inverse DCT on the sparse DCT component matrix (left) produces an approximate image block (right)

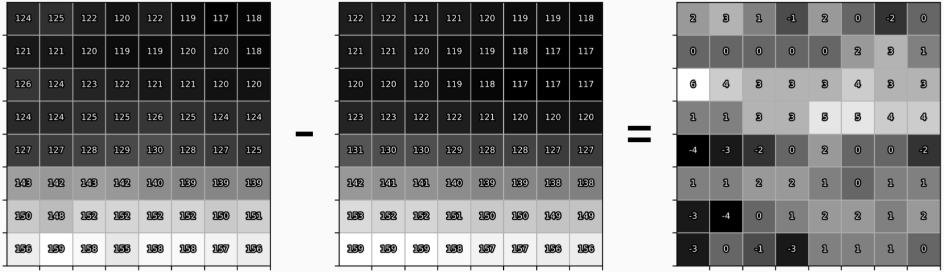


Figure 13: The difference (right) between the original block (left) and the result from the JPEG compression and decompression (middle).

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

LOSSY JPEG DECOMPRESSION: APPROXIMATION ERROR

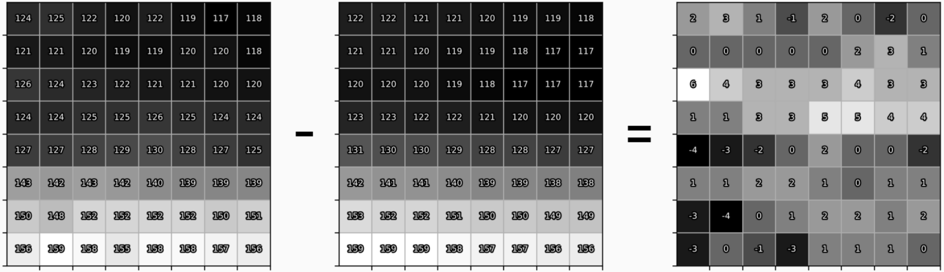


Figure 13: The difference (right) between the original block (left) and the result from the JPEG compression and decompression (middle).

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

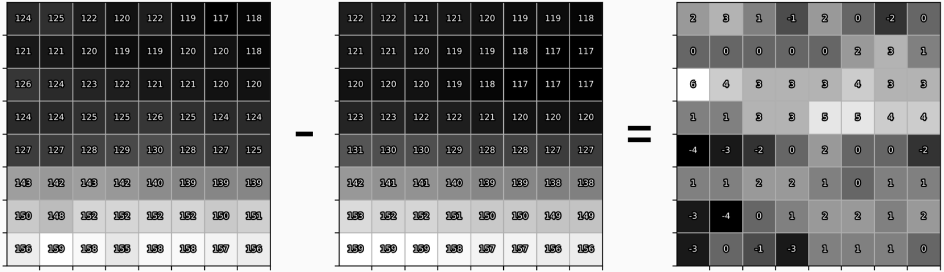


Figure 13: The difference (right) between the original block (left) and the result from the JPEG compression and decompression (middle).

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

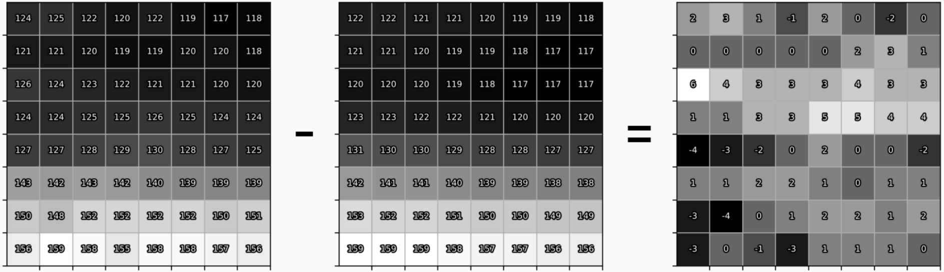


Figure 13: The difference (right) between the original block (left) and the result from the JPEG compression and decompression (middle).

- 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.

LOSSY JPEG DECOMPRESSION: APPROXIMATION ERROR

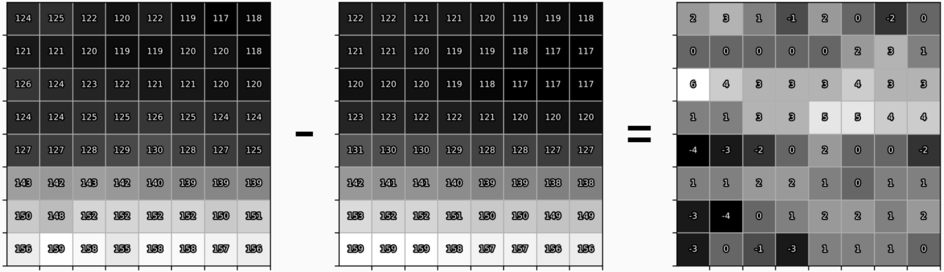


Figure 13: The difference (right) between the original block (left) and the result from the JPEG compression and decompression (middle).

- 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).

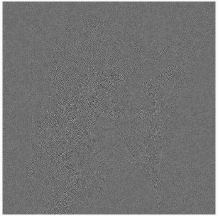
- JPEG compression can produce *block-artifacts*, *smoothings* and *ring-effects*.
- This is dependent on the quantization matrix, which determines how many coefficients are kept, and how precisely they are preserved.

(a) Smoothing- and ring-effects

(b) Block artifacts

BLOCK ARTIFACTS AND COMPRESSION RATE

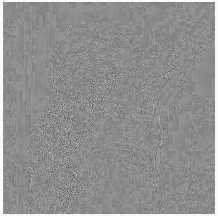
(a) Compressed



(b) Difference

(c) Detail

(d) Compressed



(e) Difference

(f) Detail

Figure 15: Top row: compression rate = 12.5. Bottom row: compression rate = 32.7

SCALING OF QUANTIZATION MATRIX

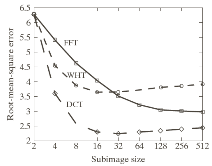
- Lossy JPEG compression use the quantization matrix to determine what information to keep.
- The scaling factor q, of the matrix determines the compression rate c_r.

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

Figure 16: Quantization matrix

Figure 17: Top row, from left: $(q, c_r) : [(1, 12), (2, 19), (4, 30)]$ Bottom row, from left: $(1, c_r) : [(8, 49), (16, 85), (32, 182)]$

- We can vary the block size.
- The compression rate and execution time increases with increaseng block size.



- Block artifacts decreases with increasing block size,
- but the ringing-effects increases.

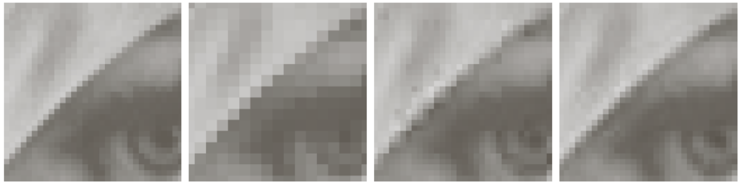
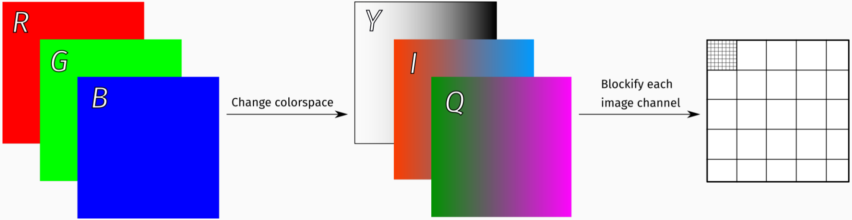


Figure 18: Original image (left). Different block sizes (left to right): 2×2, 4×4, 8×8.

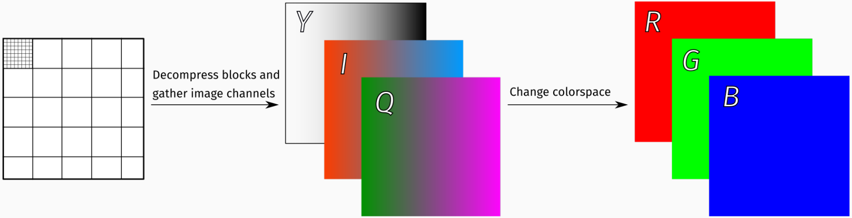
LOSSY JPEG COMPRESSION OF COLOR IMAGE

- Change color space (from RGB) in order to separate luminance from chrominance.
    - This is more aligned to how we percieve a color image.
    - Light intensity is more important to us than chromaticity.
    - Can also produce lower complexity in each channel.
- (Normally) we downsample the chromaticity-channels. Typically with a factor 2 in each channel.
- Each channel is partitioned into 8×8 blocks, and each block is coded separately as before.
- We may use different quantization matrices for the luminocity and chromaticity channels.

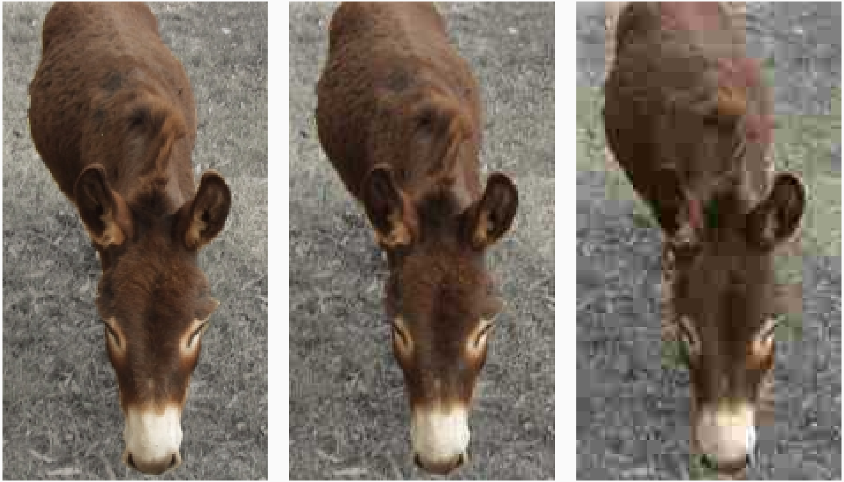


LOSSY JPEG DECOMPRESSION OF COLOR IMAGE

- Every decompressed 8×8 block in each image channel is gathered in a matrix for this channel.
- The image channels are gathered to create a color image.
- We change color space to RGB for display, or CMYK for printing.
- Even if the chromaticity channels have reduced resolution, the resolution in the RGB space is full.
  - We can get 8×8 block artifacts in intensity.
  - With 2 times downsampling in each direction in the chromaticity channels, we can get 16×16 block artifacts in chroma (”colors”).



RECONSTRUCTION ERROR IN COLOR IMAGES



(a) 1.5 - 2 bbp   (b) 0.5 - 0.75 bbp   (c) 0.25 - 0.5 bbp

Figure 19: Compression of 2f bit color images. Compression level measured in *bits per pixel* (bbp)

JPEG2000 VS JPEG

- Original JPEG is from 1992, newer standard JPEG2000 is from 2000.
- Uses a discrete wavelet transform in stead of DCT.
- Uses more sophisticated coding algorithms.
- Higher compression and better perceptual qulity.
- No block artifacts, but ringing-effects are still present.
- More computationally demanding.
- Not widely supported, even after 17 years.

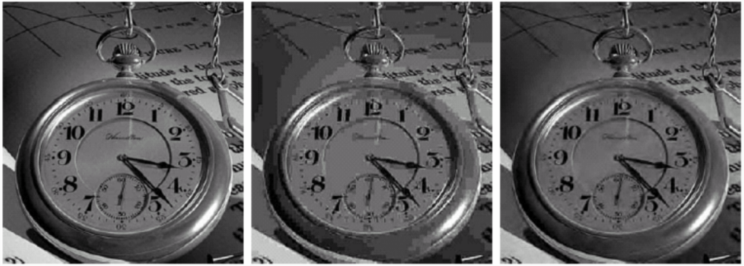


Figure 20: Original (left), JPEG (middle), and JPEG2000 (right)

- The lossless JPEG variant is using *predictive coding*.

- The lossless JPEG variant is using *predictive coding*.
- Generally, for an image f, predictive coding codes

$$e(x, y) = f(x, y) - g(x, y),$$

where $g(x, y)$ is predicted using neighbouring values of (x, y).

- The lossless JPEG variant is using *predictive coding*.
- Generally, for an image f, predictive coding codes

$$e(x,y) = f(x,y) - g(x,y),$$

  where $g(x,y)$ is predicted using neighbouring values of (x,y).
- A linear predictor of order (m,n):

$$g(x,y) = \texttt{round}\left[\sum_{i=1}^{m}\sum_{j=1}^{n}\alpha_{ij} f(x-i, y-j)\right]$$

LOSSLESS JPEG COMPRESSION: OVERVIEW

- The lossless JPEG variant is using *predictive coding*.
- Generally, for an image f, predictive coding codes

$$e(x, y) = f(x, y) - g(x, y),$$

  where $g(x, y)$ is predicted using neighbouring values of (x, y).
- A linear predictor of order (m, n):

$$g(x, y) = \texttt{round}\left[\sum_{i=1}^{m}\sum_{j=1}^{n} \alpha_{ij} f(x - i, y - j)\right]$$

- Equal-length coding requires an extra bit per pixel $e(x, y)$.
  - Or even more bits if the sum of the prediction-coefficients a_{ij}, exceeds 1.
  - The solution is entropy-coding.

LOSSLESS JPEG COMPRESSION: DETAIL

- In lossless JPEG compression, $f(x, y)$ is predicted using up to three previously processed elements.
  - Z is the pixel we want to predict.
  - Use some or all of the elements A, B, C.
- The prediction error is near zero, and is entropy coded with either Huffman coding or arithmetic coding.
- The compression rate is dependent on
  - Bits per pixel in the original image.
  - The entropy in the prediction error.
- For normal color images, the compression rate is about 2.
- Is mostly only used in medical applications.

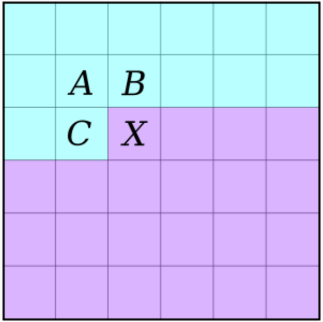


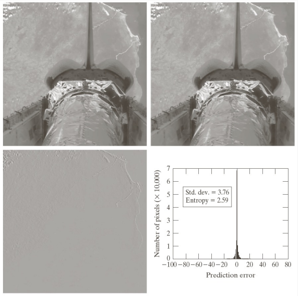
Figure 21: What elements used in predictive coding of element Z. Blue is processed, pink is not.

For a sequence of images stacked as $f(x, y, t)$, an m'th order prediction can be computed as

$$g(x, y, t) = \texttt{round}\left[\sum_{k=1}^{m} \alpha_k f(x, y, t-1)\right]$$

Motion detection and motion compensation is necessary inside so called *macro blocks* (typically of shape 16×16) to increase the compression rate.

- The difference entropy is low: $H = 2.59$.
- This gives an optimal compression rate (when single-differences is coded) of $c_r = 8/2.59 = 3$.
- Figure to the right
  - Top row: Two frames from an orbiting space shuttle video.
  - Bottom row: Prediction error image using order 1 prediction (left), and a histogram of the prediction error (right).



- Compression of digital image sequences/video is usually based on *predictive coding with motion-compensation and 2D DCT*.

- Compression of digital image sequences/video is usually based on *predictive coding with motion-compensation and 2D DCT*.
- Newer standards use often prediciton based on both previous and future images in the sequence.

- Compression of digital image sequences/video is usually based on *predictive coding with motion-compensation and 2D DCT*.
- Newer standards use often prediciton based on both previous and future images in the sequence.
- With 50-60 frames per second there is a lot to gain by prediction.

- Compression of digital image sequences/video is usually based on *predictive coding with motion-compensation and 2D DCT*.
- Newer standards use often prediciton based on both previous and future images in the sequence.
- With 50-60 frames per second there is a lot to gain by prediction.
- ISO/IEC standards for video compression (through the *Motion Picture Expert Group* (MPEG)): MPEG-1 (1992), MPEG-2 (1994), MPEG-4 (1998), MPEG-H (2013).
- ITU-T have also standards for video compression (throught the *Visual Coding Experts Group* (VCEG): H.120 (1984), H.26x-family (H.265 (2013) = MPEG-H part 2)

SUMMARY

- The purpose of compression is to represent "the same" information more compactly by reducing or removing redundancy.
- Compression is based on information theory.
- The number of bits per symbol is central, and varies with the compression method and input message.
- Central algorithms:
    - Run-length transform
    - LZW transform
    - 2D DCT
    - Predictive coding
    - Difference transform
    - Huffman coding
    - Arithmetic coding

Questions?