# INF2310 – Digital bildebehandling

**FORELESNING 11**

**KOMPRESJON OG KODING – II**

Ole Marius Hoel Rindal, foiler av Andreas Kleppe

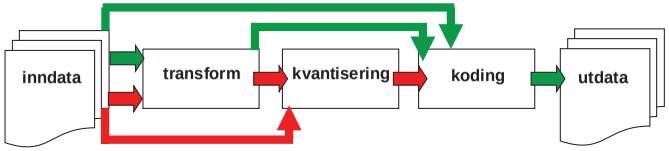
Differansetransform
Løpelengdetransform
LZW-transform
JPEG-kompresjon
Tapsfri prediktiv koding

Kompendium: 18.4, 18.7.3 og 18.8-18.8.1



# Repetisjon: Kompresjon

- Kompresjon kan deles inn i tre steg:
  - **Transform** - representer bildet mer kompakt.
  - **Kvantisering** - avrund representasjonen.
  - **Koding** - produser og bruk en kodebok.




- Kompresjon kan gjøres:
  - **Eksakt / tapsfri** (eng.: *lossless*) – følg de grønne pilene.
    - Kan da eksakt rekonstruere det originale bildet.
  - **Ikke-tapsfri** (eng.: *lossy*) – følg de røde pilene.
    - Kan da (generelt) ikke eksakt rekonstruere bildet.
    - Resultatet kan likevel være «godt nok».
  - Det finnes en mengde ulike metoder innenfor begge kategorier.



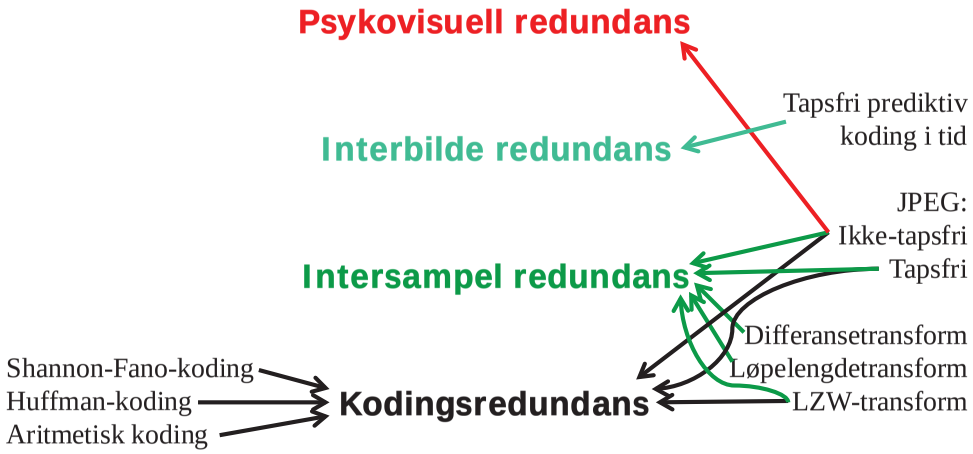
# Repetisjon: Ulike typer redundans

- Psykovisuell redundans. → Mer generelt: **Irrelevant informasjon**: Unødvendig informasjon for anvendelsen, f.eks. for visuell betraktning av hele bildet.
  - Det finnes informasjon vi ikke kan se.
    - Eksempler på enkle muligheter for å redusere redundansen: Subsample eller redusere antall biter per piksel.
- Interbilde-redundans.
  - Likhet mellom nabobilder i en tidssekvens.
    - Eks.: Lagre noen bilder i tidssekvensen og ellers bare differanser.
- Intersampel-redundans.
  - Likhet mellom nabopiksler.
    - Eks.: Hver linje i bildet kan løpelengde-transformeres.
- Kodings-redundans.
  - Enkeltsymboler (enkeltpiksler) blir ikke lagret optimalt.
  - Gitt som gjennomsnittlig kodelengde minus et teoretisk minimum.
    - Velg en metode som er «grei» å bruke og gir liten kodingsredundans.



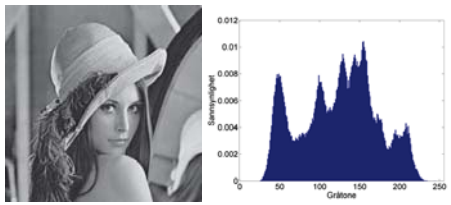
# Kompresjonsmetoder og redundans

**Sist forelesning:** **Denne forelesningen:**

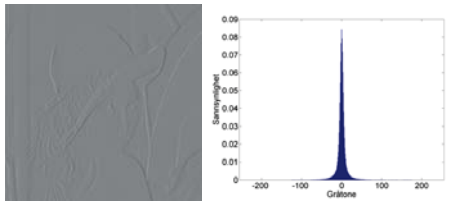


# Differansetransform

- Utnytter at horisontale nabopiksler ofte har ganske lik gråtone.
- Gitt en rad i bildet med gråtoner:
  $f_1, \ldots, f_N$ der $0 \le f_i \le 2^b-1$
- Transformer (reversibelt) til
  $g_1 = f_1,\ g_2 = f_2-f_1,\ \ldots,\ g_N = f_N-f_{N-1}$
- Merk at: $-(2^b-1) \le g_i \le 2^b-1$
  - Må bruke b+1 biter per $g_i$ hvis vi skal tilordne like lange kodeord til alle mulig verdier.
- Ofte er de fleste differansene nær 0.
  - Naturlig binærkoding av differansene er ikke optimalt.



Entropi ≈ 7,45 => CR ≈ 1,1

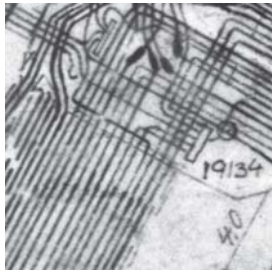


Entropi ≈ 5,07 => CR ≈ 1,6

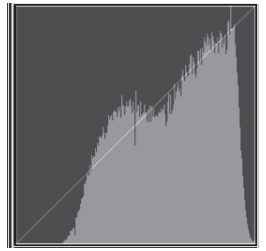


# Differansebilder og histogram

**Original:**

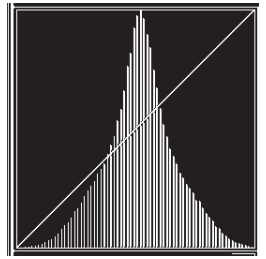


**Originalt histogram:**



**Differansebilde:**

**Histogram til differansebildet:**





# Løpelengde-transform

- Ofte inneholder bildet objekter med lignende gråtoner, f.eks. svarte bokstaver på hvit bakgrunn.
- Løpelengde-transformen (eng.: *run-length transform*) **utnytter når horisontale nabopiksler har samme gråtone**.
  - Merk: Krever ekte likhet, ikke bare omtrent like.
  - Løpelengde-transformen komprimerer bedre ettersom kompleksiteten i bildet blir mindre.
- Løpelengde-transformen er reversibel.
- Hvis pikselverdiene til en rad er:
  333333555555555544777777 (24 tall)
- Så starter løpelengde-transformen fra venstre og finner tallet 3 gjentatt 6 ganger etter hverandre, og returnerer derfor tallparet (3,6). **Formatet er: (tall, løpelengde)**
- For hele sekvensen vil løpelengdetransformen gi de 4 tallparene:
  (3,6), (5,10), (4,2), (7,6) (merk at dette bare er 8 tall)
- Kodingen avgjør hvor mange biter vi bruker for å lagre tallene.



# Bare løpelengder, ikke tall

- I binære bilder trenger vi bare å angi løpelengden for hvert «run».
  - Må også angi om raden starter med hvitt eller svart «run», alternativt forhåndsdefinere dette og tillate en «run length» på 0.
- Histogrammet av løpelengdene er ofte ikke flatt.
  - Bør derfor benytte koding som gir korte kodeord til de hyppigste løpelengdene.
- I ITU-T (tidligere CCITT) standarden for dokument-overføring per fax så Huffman-kodes løpelengdene.
  - Forhåndsdefinerte Huffman-kodebøker, én for svarte og én for hvite "runs".
- Tall-spesifiseringen i løpelengde-transformen av et gråtonebilde kan med fordel fjernes dersom ett tall forekommer *svært* hyppig.

# Repetisjon: Naturlig binærkoding

- Alle kodeord er like lange.
- Symbolets kode er binærrepresentasjonen til symbolets (null-indekserte) indeks.
  - Man legger til 0-ere foran slik at koden får den ønskelige lengden.
- Eks: En 3-biters naturlig binærkode har 8 mulige verdier:

| Symbolindeks | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| Symbol | $s_0$ | $s_1$ | $s_2$ | $s_3$ | $s_4$ | $s_5$ | $s_6$ | $s_7$ |
| Kode $c_i$ | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |



# "Gray code"

- Anta at vi har et gråtonebilde med b bitplan.
- Naturlig binærkoding gir ofte høy bitplan-kompleksitet.
  - Selv om nabopiksler ofte har omtrent lik gråtoneverdi så kan mange biter endres i den konvensjonelle binærrepresentasjonen.
  - Eks.: 127 = 01111111 og 128 = 10000000
- Anta vi ønsker minst mulig kompleksitet i hvert bitplan.
  - F.eks. fordi lav kompleksitet i hvert bitplan gjør at løpelengde-transform av hvert bitplan gir bedre komprimering.
- Da bør bitene i den alternative binærrepresentasjonen avvike minst mulig for nære gråtoneverdier.
- I «Gray code» **skifter** alltid bare **én bit** når gråtonen endres med 1.
- Overgangen fra naturlig binærkode til «Gray code» er en transform, men både naturlig binærkode og «Gray code» er koder.
  - Både i naturlig binærkode og i «Gray code» er alle kodeord er like lange. **Så forskjellen er bare hvilke kodeord som tilordnes hvilke symboler.**



# "Gray code"-transformasjoner

- Transformasjon fra naturlig binærkode (BC) til Gray-kode (GC):
  1. Start med MSB i BC og behold alle 0 inntil du treffer 1.
  2. 1 beholdes, men alle følgende bits komplementeres inntil du treffer 0.
  3. 0 komplementeres, men alle følgende bit beholdes inntil du treffer 1.
  4. Gå til 2.

«MSB» står for «most significant bit»

Huskeregel: **Marker forskjellene** fra forrige bit. Bruk at før første bit er 0.

- Fra Gray-kode til naturlig binærkode:
  1. Start med MSB i GC og behold alle 0 inntil du treffer 1.
  2. 1 beholdes, men alle følgende bits komplementeres inntil du treffer 1.
  3. 1 komplementeres, men alle følgende bits beholdes inntil du treffer 1.
  4. Gå til 2.

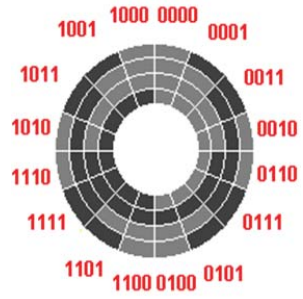
Huskeregel: **Fyll inn mellom par av 1-ere** og fjern den siste 1-eren i hvert par. Hvis antall 1-ere er odde; fyll inn fra siste 1-er.



# Eksempel: Gray-koding

4-biters Gray- og naturlig binærkode:

| Gray-kode | Naturlig binærk. | Desimalt tall |
|---|---|---|
| 0000g | 0000b | 0d |
| 0001 | 0001 | 1 |
| 0011 | 0010 | 2 |
| 0010 | 0011 | 3 |
| 0110 | 0100 | 4 |
| 0111 | 0101 | 5 |
| 0101 | 0110 | 6 |
| 0100 | 0111 | 7 |
| 1100 | 1000 | 8 |
| 1101 | 1001 | 9 |
| 1111 | 1010 | 10 |
| 1110 | 1011 | 11 |
| 1010 | 1100 | 12 |
| 1011 | 1101 | 13 |
| 1001 | 1110 | 14 |
| 1000 | 1111 | 15 |

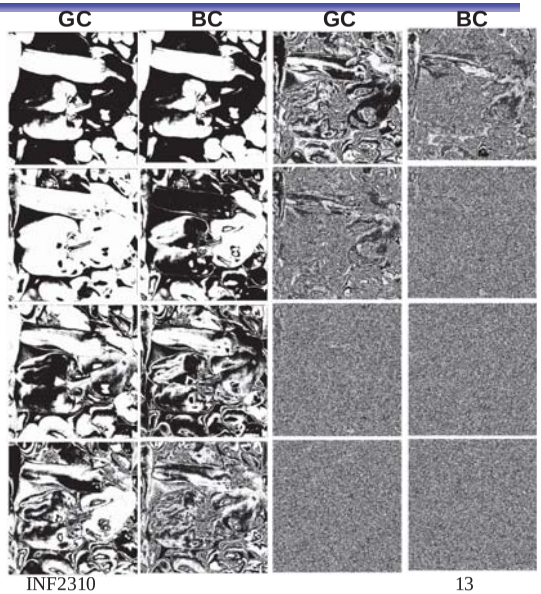



**«Gray code shaft encoder»**
**Brukes for sikker avlesing av vinkel, f.eks. i styring av robot-armer.**

**Brukt i Frank Gray's patent fra 1953, men ble brukt i Émile Baudot's telegrafkode fra 1870.**

# Gray-kode i gråtonebilder

- MSB er alltid lik i de to representasjonene.
- Større homogene områder i hvert bitplan i Gray-kode enn i naturlig binærkode.
- Flere bitplan med «støy» i naturlig binærkode.
- => Løpelengdetransform av hvert bitplan gir bedre kompresjon vba. Gray-kode enn naturlig binærkode.






# Lempel-Ziv-Welch-transform

- Utnytter **mønstre** i meldingen.
  - Ser på samforekomster av symboler.
  - Reduserer derfor først og fremst intersample-redundans.
- Lar en **symbolsekvens få én kode**.
- **Bygger opp en liste** av symbolsekvenser/strenger.
  - ... både under kompresjon og dekompresjon.
  - Listen skal verken lagres eller sendes.
    - Senderen bygger opp listen fra meldingen han skal kode.
    - Mottakeren bygger opp listen fra meldingen han mottar.
- Det **eneste man trenger er et standard alfabet**.
  - F.eks. ASCII eller heltallene f.o.m. 0 t.o.m. 255.



# Eksempel: LZW-transform

- Alfabetet: a, b og c med koder 0, 1 og 2, henholdsvis.
- Meldingen: ababcbababaaaaabab (18 symboler)
- LZW-sender: ny streng = sendt streng pluss neste usendte symbol
- LZW-mottaker: ny streng = nest siste streng pluss første symbol i sist tilsendte streng

| Ser | Sender | Senders liste | Mottar | Tolker | Mottakers liste |
|---|---|---|---|---|---|
| | | a=0, b=1, c=2 | | | a=0, b=1, c=2 |
| a | 0 | ab=3 | 0 | a | |
| b | 1 | ba=4 | 1 | b | ab=3 |
| ab | 3 | abc=5 | 3 | ab | ba=4 |
| c | 2 | cb=6 | 2 | c | abc=5 |
| ba | 4 | bab=7 | 4 | ba | cb=6 |
| bab | 7 | baba=8 | 7 | | |

» Vi mottar kode 7, men denne koden finnes ikke i listen!
» Fra ny-streng-oppskriften vet vi at kode 7 ble laget ved: ba + ?
» Siden kode 7 nå sendes, må: ? = b => 7 = ba + b = bab



# Eksempel: LZW-transform

Melding:
ababcbabab
aaaaabab

| Ser | Sender | Senders liste | Mottar | Tolker | Mottakers liste |
|---|---|---|---|---|---|
| | | *a=0, b=1, c=2* | | | *a=0, b=1, c=2* |
| *a* | *0* | *ab=3* | *0* | *a* | |
| *b* | *1* | *ba=4* | *1* | *b* | *ab=3* |
| *ab* | *3* | *abc=5* | *3* | *ab* | *ba=4* |
| *c* | *2* | *cb=6* | *2* | *c* | *abc=5* |
| *ba* | *4* | *bab=7* | *4* | *ba* | *cb=6* |
| *bab* | *7* | *baba=8* | **7** | **bab** | **bab=7** |
| a | 0 | aa=9 | 0 | a | baba=8 |
| aa | 9 | aaa=10 | **9** | **aa** | **aa=9** |
| aa | 9 | aab=11 | 9 | aa | aaa=10 |
| bab | 7 | | 7 | bab | aab=11 |

» Senderen må ha laget kode 9 da 0 = a ble sendt.
» Siden kode 9 nå sendes, må siste symbol i kode 9 være a.
» Derfor er må: 9 = a+a = aa

- Istedenfor **18 symboler er det sendt 10 koder**.
- 5 av 12 koder som ble laget ble ikke brukt.

# G&W-eksempel: LZW-transform

- Alfabet: 0, 1, ..., 255 med koder 0, 1, ..., 255, henholdsvis.
- Melding: aabbaabbaabbaabb (16 piksler med gråtoner a=39 og b=126)
- LZW-sender: ny streng = sendt streng pluss neste usendte symbol
- LZW-mottaker: ny streng = nest siste streng pluss første symbol i sist tilsendte streng

| Ser | Sender | Senders liste | Mottar | Tolker | Mottakers liste |
|---|---|---|---|---|---|
| a | 39 | aa=256 | 39 | a | |
| a | 39 | ab=257 | 39 | a | 256=aa |
| b | 126 | bb=258 | 126 | b | 257=ab |
| b | 126 | ba=259 | 126 | b | 258=bb |
| aa | 256 | aab=260 | 256 | aa | 259=ba |
| bb | 258 | bba=261 | 258 | bb | 260=aab |
| aab | 260 | aabb=262 | 260 | aab | 261=bba |
| ba | 259 | baa=263 | 259 | ba | 262=aabb |
| ab | 257 | abb=264 | 257 | ab | 263=baa |
| b | 126 | | 126 | b | 264=abb |

- Bruker 9 biter for hver kode; opprinnelig bruktes 8 biter (ved naturlig b.k.).
- Kompresjonsrate CR = (16x8)/(10x9) = 1,4222...



# Lempel-Ziv-Welch-transform

- LZW-kodene blir normalt naturlig binærkodet.
- Komprimerer typiske tekstfiler med en faktor på ≈2.
- **LZW-transform er mye brukt!**
  - Finnes i Unix' *compress* -kommando fra 1984.
  - Finnes i GIF-formatet fra 1987.
  - Er en opsjon i TIFF-formatet og i PDF-formatet.
- Men fått mye negativ oppmerksomhet pga. (nå utgått) patent.

- LZW-kodene kan **kodes videre** (f.eks. Huffman-kodes)!
- For å redusere antall mulige koder kan **listen begrenses**.
  - Vi kan sette en maksgrense, f.eks. $2^{b+1}$, og så ikke lage flere koder.
  - Vi kan ha faste prosedyrer for sletting av lite brukte / gamle koder.
    - Ofte får vi ikke bruk for alle kodene, men vi må ha **faste** prosedyrer for sletting slik at mottaker sletter likt som sender.



# Ikke-tapsfri kompresjon

- For å få høye kompresjonsrater, er det ofte nødvendig med ikke-tapsfri kompresjon.

- Husk: Man kan da **ikke rekonstruere** det originale bildet, fordi et **informasjonstap** har skjedd.

- Noen enkle metoder for ikke-tapsfri kompresjon:
  - Rekvantisering til færre antall gråtoner.
  - Resampling til dårligere romlig oppløsning.
  - Filtreringsbaserte metoder, f.eks.
    erstatt hvert ikke-overlappende 3×3-område
    med én piksel som har verdi lik f.eks.
    middelverdien eller medianverdien av de 9 pikselverdiene.



# Hvor god er bildekvaliteten?

- Hvis vi bruker **ikke-tapsfri kompresjon**
  må vi kontrollere at kvaliteten på ut-bildet er «**god nok**».

- Betegn M×N inn-bildet for $f$ og ut-bildet etter kompresjon og så dekompresjon for $g$. Feilen forårsaket av komprimeringen er da:

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

- RMS-avviket (kvadratfeilen) mellom bildene er:

$$e_{RMS} = \sqrt{\frac{1}{MN}\sum_{x=1}^{M}\sum_{y=1}^{N} e^2(x,y)}$$

- Vi kan betrakte feilen som **støy** og
  se på midlet kvadratisk signal-støy-forhold ($SNR_{ms}$):

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

# Hvor god er bildekvaliteten?

- RMS-verdien av SNR er da:

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

- Bildekvalitetsmålene ovenfor slår sammen alle feil over hele bildet.
  - Vårt synssystem er ikke slik!
    - F.eks. vil mange små avvik kunne føre til en større $SNR_{RMS}$ enn enkelte manglende eller falske objekter i forgrunnen, men vi vil oppfatte at sistnevnte er av dårligst kvalitet.

- Ofte ønsker vi at bildekvalitetsmålet skal gjenspeile **vår oppfatning av bildets kvalitet**.
  - F.eks. hvis formålet med bildet er visuell betraktning.
  - Husk: Vår oppfatning er subjektiv!



# Hvor god er bildekvaliteten?

- Et bildekvalitetsmål som godt gjenspeiler vår oppfatning vil typisk basere seg på flere parametere.
  - Hver parameter prøver å indikere hvor ille vi oppfatter en side ved kompresjonsfeilen.
  - Bildekvalitetsmålet er én verdi som baserer seg på alle parameterne.

- Feil rundt kanter oppfattes som ille.
- Feil i forgrunnen oppfattes som verre enn feil i bakgrunnen.
- Manglende eller falske strukturer oppfattes som ille.

- **Kompresjonsgraden** bør trolig **variere rundt i bildet**:
  - Komprimer nesten-homogene områder kraftig.
    - Har lite informasjon og få ikke-null-koeffisienter i 2D DFT-en.
  - Komprimer kanter og linjer mindre.
    - Har mer informasjon og flere ikke-null-koeffisienter i 2D DFT.



# JPEG-standarden

- JPEG (Joint Photographic Expert Group) er en av de vanligste bildekompresjonmetodene.
- JPEG-standarden (opprinnelig fra 1992) har varianter både for tapsfri og ikke-tapsfri kompresjon.
  - Den tapsfrie varianten er ikke JPEG-LS (som kom i 1998).
- I begge tilfeller brukes **enten Huffman-koding eller aritmetisk koding**.
- I den tapsfrie varianten benyttes **prediktiv koding**.
- I den ikke-tapsfrie varianten benyttes den **2D diskrete cosinus-transformen (2D DCT)**.



# Ikke-tapsfri JPEG-kompresjon

1. Hver bildekanal deles opp i blokker på 8x8 piksler, og hver blokk i hver kanal kodes separat.
2. Dersom intensitetene er gitt uten fortegn; trekk fra $2^{b-1}$ der $2^b$ er antall intensitetsverdier.
   - Gjør at forventet gjennomsnittlig pikselverdi er omtrent 0.
   - Eks.: Intensitetsintervallet [0, 255]; 128 trekkes fra alle pikselverdiene.
3. Hver blokk transformeres med 2D DCT (diskret cosinus-transform).

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

2D DCT

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 39.88 | 6.56 | -2.24 | 1.22 | -0.37 | -1.08 | 0.79 | 1.13 |
| -102.43 | 4.56 | 2.26 | 1.12 | 0.35 | -0.63 | -1.05 | -0.48 |
| 37.77 | 1.31 | 1.77 | 0.25 | -1.50 | -2.21 | -0.10 | 0.23 |
| -5.67 | 2.24 | -1.32 | -0.81 | 1.41 | 0.22 | -0.13 | 0.17 |
| -3.37 | -0.74 | -1.75 | 0.77 | -0.62 | -2.65 | -1.30 | 0.76 |
| 5.98 | -0.13 | -0.45 | -0.77 | 1.99 | -0.26 | 1.46 | 0.00 |
| 3.97 | 5.52 | 2.39 | -0.55 | -0.05 | -0.84 | -0.52 | -0.13 |
| -3.43 | 0.51 | -1.07 | 0.87 | 0.96 | 0.09 | 0.33 | 0.01 |

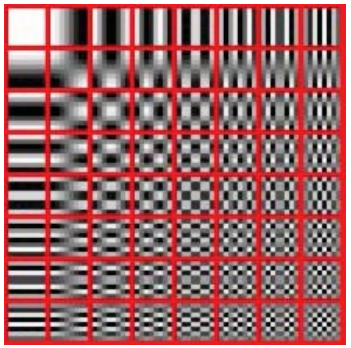
   - Mye av informasjonen i de 64 pikslene samles i en liten del av de 64 2D DCT-koeffisientene; nemlig de i øverste, venstre hjørne.

# 2D diskret cosinus-transform

- Grunnpilaren i ikke-tapsfri JPEG-kompresjon er 2D DCT:

$$F(u,v)=\frac{2}{\sqrt{MN}}c(u)c(v)\sum_{x=0}^{M-1}\sum_{y=0}^{N-1}f(x,y)\cos\left[\frac{\pi u}{M}\left(x+\frac{1}{2}\right)\right]\cos\left[\frac{\pi v}{N}\left(y+\frac{1}{2}\right)\right],\qquad c(\xi)=\begin{cases}\frac{1}{\sqrt{2}} & \text{hvis } \xi=0\\ 1 & \text{ellers}\end{cases}$$

- Sterkt relatert til 2D DFT.
- I JPEG transformerer vi 8x8-blokker så vi bruker bare de 64 «8x8-cosinus-bildene»:
  - For hvert bilde vi går til høyre eller ned så økes den tilhørende frekvenskomponenten med 0,5.
  - Husk: I 2D DFT økte frekvenskomp. med 1, som lagde par med like cosinus-bilder.
  - Husk: I 2D DFT hadde vi også noen sinus-bilder.
  - 2D DCT-koeffisientene beregnes analogt med det vi gjorde for 2D DFT; summere punktproduktet mellom 8x8-blokken og hvert «cosinus-bilde».
  - 2D DCT beregnes hurtig ved å forhåndsberegnes de 64 «8x8-cosinus-bildene».



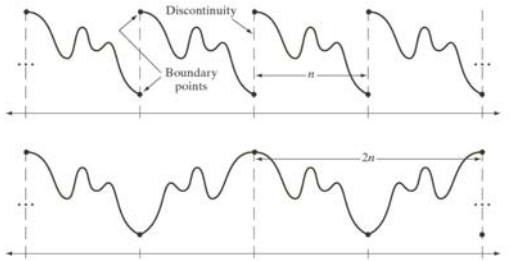


# Hvorfor DCT og ikke DFT?

- Den implisitt antatte N-punkts periodisiteten i DFT-en vil introdusere høye frekvenser pga. randdiskontinuitet.
  - Fjerner vi disse frekvensene får vi kraftige blokk-artefakter.
  - Beholder vi dem reduseres kompresjonsraten ift. DCT der vi ofte slepper å beholde de fleste høye frekvenser.

N er lengden av 1D-bildet.

I JPEG-sammenheng er «randen» blokkranden.




- DCT er implisitt 2N-punkts periodisk og symmetrisk om N, derfor **introduseres ikke disse høye frekvensene**.



# Ikke-tapsfri JPEG-kompresjon

JPEG-kompresjonsalgoritmen fortsetter med at:

4.2D DCT-koeffisientene:

a) punktdivideres med en vektmatrise og deretter

b) avrundes til nærmeste heltall.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 39.88 | 6.56 | -2.24 | 1.22 | -0.37 | -1.08 | 0.79 | 1.13 |
| -102.43 | 4.56 | 2.26 | 1.12 | 0.35 | -0.63 | -1.05 | -0.48 |
| 37.77 | 1.31 | 1.77 | 0.25 | -1.50 | -2.21 | -0.10 | 0.23 |
| -5.67 | 2.24 | -1.32 | -0.81 | 1.41 | 0.22 | -0.13 | 0.17 |
| -3.37 | -0.74 | -1.75 | 0.77 | -0.62 | -2.65 | -1.30 | 0.76 |
| 5.98 | -0.13 | -0.45 | -0.77 | 1.99 | -0.26 | 1.46 | 0.00 |
| 3.97 | 5.52 | 2.39 | -0.55 | -0.05 | -0.84 | -0.52 | -0.13 |
| -3.43 | 0.51 | -1.07 | 0.87 | 0.96 | 0.09 | 0.33 | 0.01 |

dividere med

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 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 |

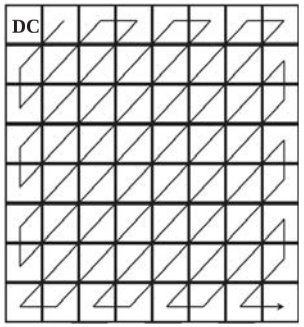
resultatet avrundes til

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| -9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |



# Ikke-tapsfri JPEG-kompresjon (sekvensiell modus)

DC- og AC-elementene behandles nå separat.




AC-elementene:

1.Sikk-sakk-skannes:

- Ordner elementene i en 1D-følge.
- Absoluttverdien av elementene vil stort sett avta utover i følgen.
- Mange koeffisienter er null, spesielt litt uti følgen.

2.0-basert løpelengdetransform av 1D-følgen.

3.«Løpelengdeparene» Huffman- eller aritmetisk kodes.

- Et «løpelengdepar» er her (antall 0-ere, antall biter i «ikke-0»).
- Aritmetisk koding gir ofte 5-10 % bedre kompresjon (ifølge JPEG-medlemmers tester)

# Ikke-tapsfri JPEG-kompresjon

DC- og AC-elementene
behandles nå separat.

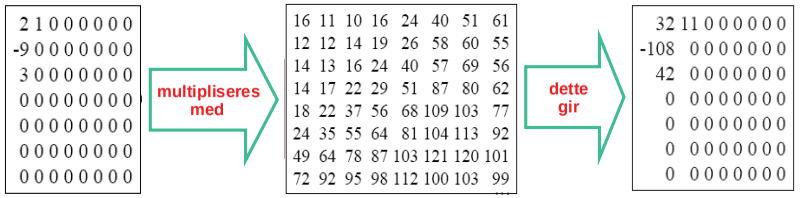
DC-elementene:

1. For hver kanal samles DC-elementene fra alle blokkene.
2. Disse er korrelerte og blir derfor differansetransformert.
3. Differansene Huffman-kodes eller aritmetisk kodes.
   - Mer presist: Antall biter i hver differanse entropikodes.



# Ikke-tapsfri JPEG-dekompresjon

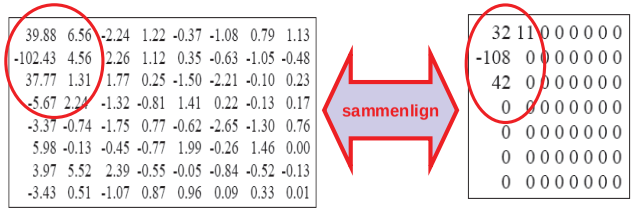
- Huffman-kodingen og den aritmetiske kodingen reversibel, og gir AC-«løpelengdeparene» og DC-differansene.
- «Løpelengdetransformen» og differansetransformen er reversibel, og gir de skalerte og kvantifiserte 2D DCT-koeffisientene.
- Sikk-sakk-transformen er reversibel, og gir en heltallsmatrise.
- Denne matrisen punktmultipliseres med vektmatrisen.






# Ikke-tapsfri JPEG-dekompresjon

- Produktet er **IKKE** helt likt 2D DCT-koeffisientene.

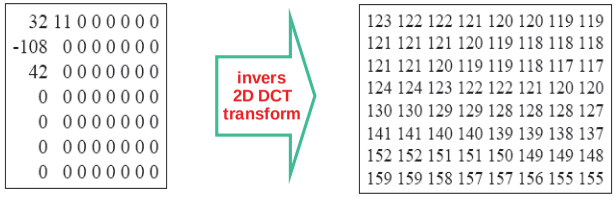



- Men **de store trekkene er bevart**.
  - Her (og oftest): De store tallene i øvre venstre hjørne.
  - I enkelt blokker eller ved høy kvalitet: Noen andre tall også.
- De fleste tallene i matrisen er lik 0,
  men disse var også opprinnelig nær 0.



# Ikke-tapsfri JPEG-dekompresjon

- Så gjør vi en invers 2D DCT, og avrunder de resulterende verdiene til nærmeste heltall.
- Vi har da fått rekonstruert en 8×8 piksels bildeblokk.



# Ikke-tapsfri JPEG-dekompresjon

- Differansene fra den originale blokken er **små**!

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 124 | 125 | 122 | 120 | 122 | 119 | 117 | 118 | | 123 | 122 | 122 | 121 | 120 | 120 | 119 | 119 | | 1 | 3 | 0 | -1 | -2 | -1 | -2 | -1 |
| 121 | 121 | 120 | 119 | 119 | 120 | 120 | 118 | | 121 | 121 | 121 | 120 | 119 | 118 | 118 | 118 | | 0 | 0 | -1 | -1 | 0 | 2 | 2 | 0 |
| 126 | 124 | 123 | 122 | 121 | 121 | 120 | 120 | | 121 | 121 | 120 | 119 | 119 | 118 | 117 | 117 | | 5 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| 124 | 124 | 125 | 125 | 126 | 125 | 124 | 124 | | 124 | 124 | 123 | 122 | 122 | 121 | 120 | 120 | | 0 | 0 | 2 | 3 | 4 | 4 | 4 | 4 |
| 127 | 127 | 128 | 129 | 130 | 128 | 127 | 125 | - | 130 | 130 | 129 | 129 | 128 | 128 | 128 | 127 | = | -3 | -3 | -1 | 0 | 2 | 0 | -1 | -2 |
| 143 | 142 | 143 | 142 | 140 | 139 | 139 | 139 | | 141 | 141 | 140 | 140 | 139 | 139 | 138 | 137 | | 2 | 1 | 3 | 2 | 1 | 0 | 1 | 2 |
| 150 | 148 | 152 | 152 | 152 | 152 | 150 | 151 | | 152 | 152 | 151 | 151 | 150 | 149 | 149 | 148 | | 2 | -4 | 1 | 1 | 2 | 3 | 1 | 3 |
| 156 | 159 | 158 | 155 | 158 | 158 | 157 | 156 | | 159 | 159 | 158 | 157 | 157 | 156 | 155 | 155 | | -3 | 0 | 0 | -2 | -1 | 2 | 2 | 1 |

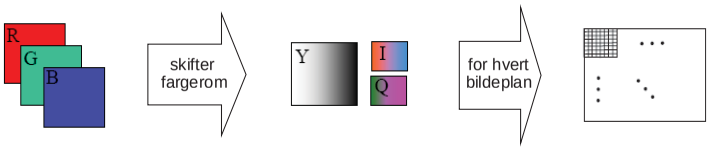
- De er **likevel ikke 0**.
- Det kan bli gjort forskjellig feil på nabopiksler, spesielt dersom de tilhører forskjellige blokker.
  - Kompresjon / dekompresjon kan derfor gi **blokk-artefakter**; rekonstruksjonsfeil som gjør at vi ser at bildet er blokk-inndelt.



# Ikke-tapsfri JPEG-kompresjon av fargebilde

- Skifter fargerom for å separere lysintensitet fra kromasi.
  - Stemmer bedre med hvordan vi oppfatter et fargebilde.
    - Lysintensiteten er viktigere enn kromasi for oss.
  - Kan også gi lavere kompleksitet i hver kanal.
- Nedsampler (normalt) kromasitet-kanalene.
  - Typisk med en faktor 2 i begge retninger.
- Hver bildekanal deles opp i blokker på 8x8 piksler, og hver blokk kodes separat som før.
  - Kan bruke forskjellige vektmatriser for intensitet- og kromasitet-kanalene.

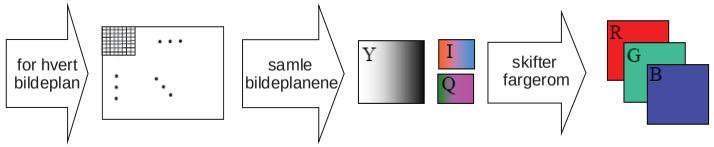
Intet fargerom er spesifisert i del 1 (1992) av JPEG-standarden. En senere del, del 5 (2009), spesifiserer filformatet JFIF (JPEG File Interchange Format). Her brukes fargemodellen $Y'C_BC_R$






# Ikke-tapsfri JPEG-dekompresjon av fargebilde

- Alle dekomprimerte 8x8-blokker i hver bildekanal samles til en matrise for den bildekanalen.
- Bildekanalene samles til et fargebilde.
- Vi skifter fargerom fra den brukte fargemodellen til:
  - til RGB for fremvisning, eller
  - til CMYK for utskrift.




- Selv om kromasitet-kanalene har redusert oppløsning, har vi full oppløsning i RGB-fargerommet.
  - Kan få 8×8-blokkartefakter i intensitet.
  - Ved en faktor 2 nedsampling i hver retning av kromasitet-kanalene kan vi få 16×16 piksels blokkartefakter i kromasi («fargene»).



# Rekonstruksjonsfeil i gråtonebilder

- JPEG-kompresjon kan gi **8×8-piksels blokk-artefakter**, **glatting** og **ringinger**.
- Avhengig av vektmatrisen
  - som bestemmer hvor mange koeffisienter som lagres, og hvor presist disse lagres.

# Blokk-artefakter

- Blokk-artefaktene øker med kompresjonsraten.

- Øverst: kompresjonsrate = 25
- Nederst: kompresjonsrate = 52



# Eksperiment: Skalering av vektmatrisen

- Ikke-tapsfri JPEG-komprimerer og dekomprimerer ved bruk av vektmatrisen:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 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 |

skalert med hhv.:
1, 2, 4
8, 16, 32

- Får da kompresjons-rater på hhv.:
12, 19, 30
49, 85, 182



# Rekonstruksjonsfeil i fargebilder

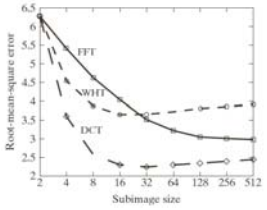
- 24-biters RGB-bilde komprimert til 1,5-2 biter per piksel (bpp).
- 0,5 - 0,75 bpp gir god/meget god kvalitet.
- 0,25 - 0,5 bpp gir noen feil.
  - 8x8-blokkeffekt i intensitet.
  - Kromasifeil («fargefeil») i muligens større blokker.
- JPEG 2000 bruker ikke blokker.
  - Gir høyere kompresjon.
  - Og/eller mye bedre kvalitet:

Original JPEG JPEG 2000

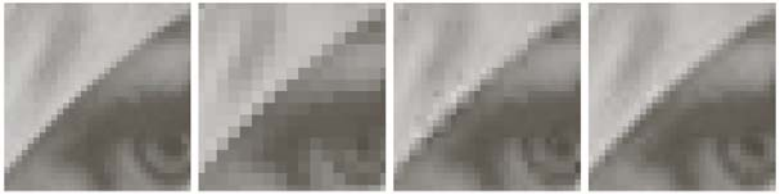


# Blokkstørrelse

- Kompresjonsraten og eksekveringstiden øker med blokkstørrelsen, men rekonstruksjonsfeilen avtar opp til et punkt:
- Eksperiment: For forskjellige n;
  1. Del opp bildet i nxn piksels blokker.
  2. 2D DCT, behold 25% av koeffisientene.
  3. Invers 2D DCT og beregn kvadratfeilen.




- **Blokk-artefakter avtar** med blokkstørrelse:



Original 2x2-blokker 4x4-blokker 8x8-blokker

- Men **ringing-problemet øker** med blokkstørrelsen!

# Tapsfri JPEG-kompresjon

- I den tapsfrie varianten av JPEG benytter **prediktiv koding**.
- Generelt for prediktiv koding så kodes:
  e(x,y) = f(x,y) - g(x,y)
  der g(x,y) er **predikert fra m naboer** rundt (x,y).
- 1D lineær prediktor av orden m: $g(x,y) = round\left[\sum_{i=1}^{m} \alpha_i \, f(x, y-i)\right]$
- Førsteordens lineær prediktor: $g(x,y) = round\left[\alpha \, f(x,y-1)\right]$
  - **Hvilken transform er dette hvis α=1?**
- Med lik-lengde koding trenger vi et ekstra bit per piksel e(x,y).
  - Eller enda flere biter dersom summen av prediksjonskoeffisientene, $\alpha_i$, er mer enn 1.
  - Løsning: **Entropikoding**.



# Tapsfri JPEG-kompresjon

- I tapsfri JPEG-kompresjon predikeres f(x,y) ved bruk av opptil 3 elementer:
  - X er pikselen vi ønsker å predikere.
  - Benytter 1-3 av elementene A, B og C.

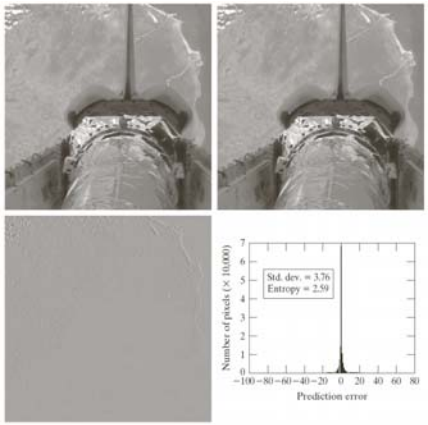
| | | |
|---|---|---|
| C | B | |
| A | X | |
| | | |

- Prediksjonsfeilene entropikodes.
  - Huffman-koding eller aritmetisk koding.
- Kompresjonsraten er avhengig av:
  - Biter per piksel i originalbildet.
  - Entropien til prediksjonsfeilene.
- For vanlige fargebilder blir kompresjonsraten ≈2.
- Brukes for det meste kun i medisinske anvendelser.



# Tapsfri koding av bildesekvenser

- Prediksjon også mulig i tidssekvenser: $g(x,y,t) = round\left[\sum_{i=1}^{m} \alpha_i \, f(x,y,t-i)\right]$
- Enkleste mulighet: Førsteordens lineær: $g(x,y,t) = round\left[\alpha \, f(x,y,t-1)\right]$
- Eksempel:
  - Differanse-entropien er lav: H=2,59
  - Gir en optimal kompresjonsrate (ved koding av enkeltdifferanser):
    CR ≈ 8/2,59 ≈ 3




- Bevegelse-deteksjon og bevegelse-kompensasjon innenfor blokker er nødvendig for å øke kompresjonsraten.
  - Blokkene kalles ofte **makroblokker** og kan f.eks. være 16x16 piksler.



# Digital video

- Kompresjon av digitale bildesekvenser/video er vanligvis basert på **prediktiv koding med bevegelse-kompensasjon og 2D DCT**.
- I nyere standarder er prediksjonen basert på **både tidligere og fremtidige bilder**.
  - Typisk: Noen bilder er upredikerte, noen flere bruker kun tidligere bilder, mens de fleste bruker både tidligere og fremtidige.
- Med 50-60 bilder i sekundet er det mye å spare på prediksjon!
- ISO/IEC sine standarder for videokompresjon (gjennom sin Motion Picture Expert Group (MPEG)):
  - MPEG-1 (1992), MPEG-2 (1994), MPEG-4 (1998), MPEG-H (2013).
- ITU-T har også standarder for videokompresjon (gjennom sin Visual Coding Experts Group (VCEG)):
  - H.120 (1984), H.26x-familien (H.265 (2013) = MPEG-H Part 2).

# Oppsummering: Kompresjon

- Hensikt: Kompakt data-representasjon, «samme» informasjon.
  - Fjerner eller reduserer redundanser.
- Kompresjon er basert på informasjonsteori.
- Antall biter per symbol/piksel er sentralt, og varierer med kompresjonsmetodene og meldingene.
- Sentrale algoritmer:
  - Transformer (brukes før koding): Løpelengdetransform, LZW-transform, 2D DCT, og prediktiv koding; differansetransform, differanse i tid m.m.
  - Koding (husk kodingsredundans og entropi!)
    - Huffman-koding: Til å lage en forhåndsdefinert kodebok, eller bruk symbolhistogrammet til meldingen og send kodeboken.
    - Aritmetisk koding: Representerer meldingen som et intervall og så koder intervallet som det binært sett korteste tallet i intervallet. Send eller ha forhåndsdefinert modellen.