INF2310 – Digital bildebehandling

FORELESNING 12

KOMPRESJON OG KODING – II

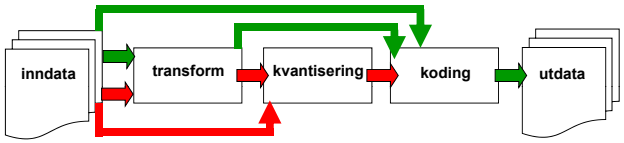
Andreas Kleppe

LZW-koding
Aritmetisk koding
JPEG-kompresjon
Tapsfri prediktiv koding

Kompendium: 18.7.3-18.7.4 og 18.8-18.8.1

Repetisjon: Kompresjon

- Kompresjon kan deles inn i tre steg:
  - **Transform** - representer bildet mer kompakt.
  - **Kvantisering** - avrunding av representasjonen.
  - **Koding** - produksjon og bruk av kodebok.



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

Repetisjon: Melding, data og informasjon

- Vi skiller mellom meldingens data og informasjon:

- **Melding:** teksten eller bildet som vi skal lagre eller sende.
- **Data:** strømmen av biter som lagres på fil eller sendes.
- **Informasjon:** et matematisk begrep som kvantifiserer hvor overraskende / uventet en melding er.

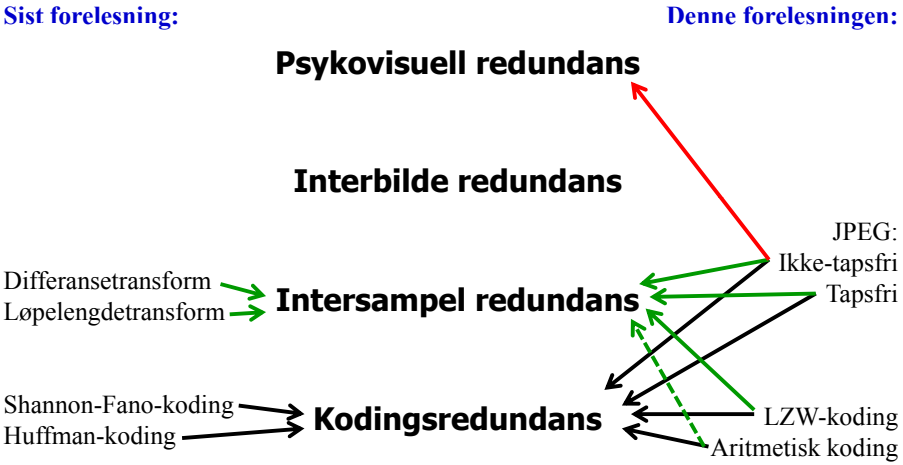
  - Et varierende signal har mer informasjon enn et monotont signal.

  - I bilder har kanter rundt objekter høyt informasjonsinnhold, spesielt kanter med mye krumning.

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.
    - Enkle muligheter for å redusere redundansen: Subsample eller redusere antall biter per piksel.
- Interbilde-redundans.
  - Likhet mellom nabobilder i en tidssekvens.
    - Kode noen bilder i tidssekvensen og ellers bare differanser.
- Intersampel-redundans.
  - Likhet mellom nabopiksler.
    - Hver linje i bildet kan løpelengde-transformeres.
- Kodings-redundans.
  - Gjennomsnittlig kodelengde minus et teoretisk minimum.
    - Velg en metode som er "grei" å bruke og gir liten kodingsredundans.

Kompresjonsmetoder og redundans



Lempel-Ziv-Welch-koding

- Tapsfri kompresjonsmetode.
  - Kan gi bedre kompresjon enn Huffman-koding.
- Premierer **mønstre** i meldingen.
  - Ser på samforekomster av symboler.
- **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-koding

- 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-koding

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-koding

- 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 C = (16x8)/(10x9) = 1,4222…

Lempel-Ziv-Welch-koding

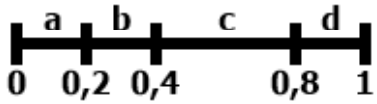
- Komprimerer typiske tekstfiler med en faktor på ≈2.
- Lempel-Ziv-Welch-kodene kan **kodes videre**.
  - F.eks. Huffman-kodes.
- For å ikke få mange flere mulige koder kan **listen begrenses**.
  - Vi kan sette en maksgrense, f.eks. 2^{b+1}, og så ikke lage flere koder.
  - Vi kan ha faste prosedyrer 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.
- **LZW-koding er mye brukt!**
  - Finnes i Unix' *compress* -kommando fra 1984.
  - Finnes i GIF-formatet fra 1987.
  - Er en opsjon i TIFF-formatet.
  - Er en opsjon i PDF-formatet.
- Men fått mye negativ oppmerksomhet p.g.a. (nå utgått) patent.

Aritmetisk koding

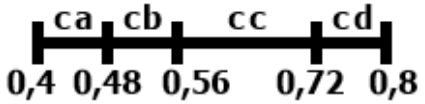
- Alternativ til Huffman-koding, som den deler flere likheter med:
  - Tapsfri kompresjonsmetode.
  - Entropikoder: Koder mer sannsynlige symboler med kompakt.
    - Bruker sannsynlighetsmodell / histogram av symbolforekomster.
      - Huffman-koding: Sender / bruker kjent kodebok.
      - Aritmetisk koding: Sender / bruker kjent modell/histogram.
- Skiller seg fra Huffman-koding ved at **aritmetisk koding ikke lager kodeord for enkeltsymboler.**
  - I stedet kodes en sekvens av symboler som ett tall n ($0.0 \leq n < 1{,}0$).
  - Dermed kan vi oppnå bedre kompresjon enn Huffman-koding (og entropien setter ikke lenger en nedre grense for bitforbruket).
- Resulterer i et **bitforbruk per symbol som er nær entropien**.
- Gir generelt bedre kompresjon jo lenger symbolsekvensen er.

Aritmetisk koding: Grunntanke

- Symbolsannsynlighetene summerer seg til 1.
- Dermed definerer de en **oppdeling av intervallet [0, 1)**.
  - Hvert delintervall representerer ett symbol.



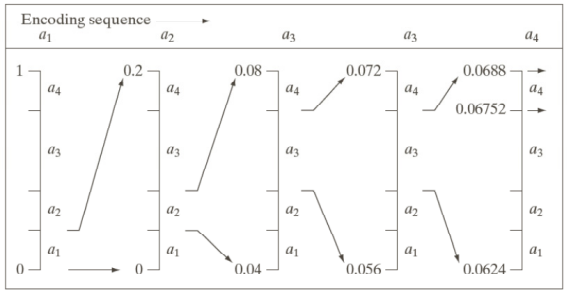
- Har vi to symboler etter hverandre, kan vi **oppdele intervallet som representerer det første symbolet**.
  - Hvert delintervall representerer symbolparet; det første symbolet etterfulgt av ett symbol.



- Tilsvarende for flere symboler etter hverandre.
- Resultat: Et halvåpent delintervall av [0, 1).
- Finner så en bitsekvens som representerer intervallet.

Eksempel: Aritmetisk koding

- Sannsynlighetsmodell: $P(a_1)=P(a_2)=P(a_4)=0{,}2$ og $P(a_3)=0{,}4$
- Melding/symbolsekvens: $a_1a_2a_3a_3a_4$



- a_1 ligger i intervallet [0, 0,2)
- a_1a_2 ligger i intervallet [0,04, 0,08)
- $a_1a_2a_3$ ligger i intervallet [0,056, 0,072)
- $a_1a_2a_3a_3$ ligger i intervallet [0,0624, 0,0688)
- $a_1a_2a_3a_3a_4$ ligger i intervallet [0,06752, 0,0688)

Aritmetisk koding: Algoritmen

1. La «current interval» = [0, 1).

2. For hvert symbol s_i i sekvensen (fra venstre):
   a) Del opp «current interval» i delintervaller, der størrelsen på hvert delintervall er proporsjonal med sannsynligheten for vedkommende symbol.
      - Proporsjonalitetsfaktoren er størrelsen av «current interval».
   b) Velg det delintervallet som svarer til s_i, og la «current interval» være dette delintervallet.

3. Representer «current interval» med en kortest mulig bitsekvens.

Aritmetrisk koding i praksis

- Alfabetet inneholder normalt **END-OF-DATA (EOD)**.
  - Dette symbolet må også få en sannsynlighet i modellen.
  - Alternativt kan lengden av symbolsekvensen defineres, enten predefinert eller spesifisert.

- I praksis oppdeles ikke «current interval», **kun intervallet til det trufne symbolet beregnes**:
  - begynnelsen av nytt c.i. =
    begynnelsen av gammelt c.i. +
    (bredden av gammelt c.i. *
    begynnelsen av symbolets intervall i modellen)
  - slutten av nytt c.i. =
    begynnelsen av gammelt c.i. +
    (bredden av gammelt c.i. *
    slutten av symbolets intervall i modellen)
- Vi beregner altså kun ett intervall for hvert symbol.

AK: Kodingseksempel

- Modell: Alfabet [a, b, c] med sannsynligheter [0,6, 0,2, 0,2].
- Hvilket delintervall av [0, 1) vil entydig representere meldingen **acaba** ?

- **a** ligger i intervallet [0, 0,6).
  - «Current interval» har nå en bredde på 0.6.
- **ac** ligger i intervallet [0+0,6*0,8, 0+0,6*1) = [0,48, 0,6).
  - Intervallbredden er nå 0,12 (= produktet 0,6*0,2).
- **aca** ligger i intervallet [0,48+0,12*0, 0,48+0,12*0,6) = [0,48, 0,552).
  - Intervallbredden er 0,072 (= produktet 0,6*0,2*0,6).
- **acab** er i [0,48+0,072*0,6, 0,48+0,072*0,8) = [0,5232, 0,5376).
  - Intervallbredden er 0,0144 (= produktet 0,6*0,2*0,6*0,2).
- **acaba** er i [0,5232+0,0144*0, 0,5232+0,0144*0,6) = [0,5232, 0,53184).
  - Intervallbredden er nå 0,00864 (= produktet 0,6*0,2*0,6*0,2*0,6).

- Et tall i intervallet, f.eks. 0,53125, vil entydig representere **acaba**, forutsatt at mottakeren har den samme modellen og vet når å stoppe.

AK: Dekodingseksempel

- ***Anta at vi skal dekode tallet 0,53125.***
- Samme modell: Alfabet [a, b, c] med sannsynligheter [0,6, 0,2, 0,2].

- La [0, 1) være «current interval».
- Del intervallet iht. modellen; a er [0, 0,6), b er [0,6, 0,8) og c er [0,8, 1).
- Tall 0,53125 ligger i delintervallet for a, [0, 0.6): **=> Første symbol: a**
- [0, 0,6) er nå «current interval». Del dette opp i henhold til modellen:
  - delintervallet for a er [0, 0,36) *bredden er 60% av [0, 0,6)*
  - delintervallet for b er [0,36, 0,48) *bredden er 20% av [0, 0,6)*
  - delintervallet for c er [0,48, 0,6) *bredden er 20% av [0, 0,6)*
- Tall 0,53125 ligger i delintervallet [0,48, 0,6): **=> Andre symbol: c**
- [0,48, 0,6] er nå «current interval». Del dette opp i henhold til modellen:
  - delintervallet for a er [0,48, 0,552)
  - delintervallet for b er [0,552, 0,576)
  - delintervallet for c er [0,576, 0,6)
- Tall 0,53125 ligger i delintervallet [0,48, 0,552): **=> Tredje symbol: a**

AK: Dekodingseksempel

- ***Anta at vi skal dekode tallet 0,53125.***
- Samme modell: Alfabet [a, b, c] med sannsynligheter [0,6, 0,2, 0,2].

- [0,48, 0,552] er nå «current interval». Del dette opp iht. modellen:
  - delintervallet for a er [0,48, 0,5232)
  - delintervallet for b er [0,5232, 0,5376)
  - delintervallet for c er [0,5376, 0,552)
- Tall 0,53125 ligger i delintervallet [0.5232, 0.5376): **=> 4. symbol: b**
- Vi deler opp intervallet [0,5232, 0,5376) for å finne neste symbol:
  - delintervallet for a er [0,5232, 0,53184)
  - delintervallet for b er [0,53184, 0,53472)
  - delintervallet for c er [0,53472, 0,5376)
- Tall 0,53125 ligger i delintervallet [0,5232, 0,53184): **=> 5. symbol: a**

- **Vi kunne fortsatt å "dekode" symboler.**
- For å vite at vi skal stoppe her trenger vi:
  - Enten et EOD-symbol i modellen; stopp når vi dekoder dette.
  - Eller vite hvor mange symboler vi skal dekode; stopp når vi har dekodet det antallet.

Desimaltall som bitsekvens

- Vi lagrer/sender ikke desimaltall, men en sekvens med biter.
- Spørsmålet er: Hvordan kan vi representere intervallet ved bruk av færre mulig biter?
- Først må vi å se hvordan vi kan representere desimaltall binært:

- Et desimaltall N i intervallet [0, 1) kan skrives som en veiet sum av negative toerpotenser:

$$N = c_1 2^{-1} + c_2 2^{-2} + c_3 2^{-3} + \ldots + c_n 2^{-n} + \ldots$$

  der hver c_i er enten 0 eller 1.
- Rekken av koeffisienter $c_1c_2c_3c_4\ldots$ er bitsekvensen som representerer desimaltallet N.
- Vi skriver at: $N = 0,c_1c_2c_3c_4\ldots_2$ der $_2$ indikerer at tallet er skrevet i totallsystemet.

Desimaltall som bitsekvens

- Situasjon: Vi har et desimaltall i [0, 1) som vi ønsker å skrive i totallsystemet.
- Løsning: Suksessiv multiplikasjon med 2:

1. Multipliser begge sider av følgende likning med 2:
   $$N = c_1 2^{-1} + c_2 2^{-2} + c_3 2^{-3} + \ldots + c_n 2^{-n} + \ldots$$
   Heltallsdelen av resultatet er da lik c_1 fordi:
   $2N = c_1 + R$, der $R = c_2 2^{-1} + c_3 2^{-2} + \ldots + c_n 2^{-(n-1)} + \ldots$
   Hvis resten R er 0, så er vi ferdige.
2. Multipliser resten R med 2.
   - Heltallsdelen av produktet er neste bit og R oppdateres til å være den nye resten.
3. Hvis resten R er 0, så er vi ferdige. Ellers går vi til 2.

Representasjon av intervall

- Spørsmålet var: Hvordan kan vi representere intervallet ved bruk av færre mulig biter?

- Eksempel: Intervallet er [0,5232, 0,53184).
  - $0{,}53125_{10}$ er (som sagt) et desimaltall i dette intervallet.
  - **$0{,}53125_{10} = 0{,}10001_2$** siden:

    2*0,53125 = 1,0625 => c_1 = **1**, rest = 0,0625
    2*0,0625 = 0,125 => c_2 = **0**, rest = 0,125
    2*0,125 = 0,25 => c_3 = **0**, rest = 0,25
    2*0,25 = 0,50 => c_4 = **0**, rest = 0,5
    2*0,5 = 1,0 => c_5 = **1**, rest = 0
  - Vi trenger altså bare 5 biter på å kode intervallet!
- Men hvordan kan vi finne hvilket desimaltall vi skal bruke for at binær-representasjoner blir kortest mulig?

Eksempel: Representasjon av intervall

- Finn kortest mulig $N=0,c_1c_2c_3..._2$ innenfor intervallet [0,6, 0,7).
- Hvis n ≥ k så er:

  $$2^{-k+1} = 2^{-(k-1)} > c_k2^{-k} + ... + c_n2^{-n}$$

  siden c_i er 0 eller 1.
- Derfor er:

  N = 0,**1...** $_2$ => 0,5 ≤ N < 1
  N = 0,**10...** $_2$ => 0,5 ≤ N < 0,75
  N = 0,**100...** $_2$ => 0,5 ≤ N < 0,625
  N = 0,**101...** $_2$ => 0,625 ≤ N < 0,75
- => Intervall kan kodes ved det binære kommatallet $0{,}\mathbf{101}_2$ (ekvivalent med $0{,}625_{10}$), altså med bare 3 biter.
  - Hvis vi vil kreve at øvre og nedre grense er innenfor intervallet; intervallet kan kodes ved $0{,}1010_2$ fordi:

    N = 0,**1010...** $_2$ => 0,625 ≤ N < 0,6875

AK: Problemer og løsninger

- Den stadige krympingen av «current interval» krever at flyttall og aritmetikk har stadig økende presisjon.
  - Lengre symbolsekvenser krever bedre presisjon.
- Kompresjonsmetoden gir ingen output før hele sekvensen er behandlet.
- Løsning: **Send/lagre den mest signifikante biten** når entydig kjent, og doble lengden av «c.i.».
  - Bitsekvensen blir (i teorien) lik som før.
  - Kan i praksis øke presisjonen, men løser ikke problemet.
- Det finnes flere praktiske AK-implementasjoner.
  - Alle er ganske regnetunge.
    - Vi trenger uansett presis/nøyaktig flyttallsaritmetikk!
  - De aller fleste er belagt med patenter.

AK: Ikke-statiske modeller

- Statiske histogrambaserte modeller er ikke optimale.
- ***Høyere-ordens modeller*** endrer estimatet av sannsynligheten for et symbol (og dermed hvordan «current interval» deles opp) basert på foregående symbol (som er konteksten).
  - I en fornuftig modell for engelsk tekst vil intervallbredden for «u» øke hvis «u» kommer etter «Q» eller «q».
- ***Modellen kan også være adaptiv***, slik at den «kontinuerlig» endres ved å tilpasse seg den faktiske symbolstrømmen.

- Uansett modell må mottakeren ha den samme!

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 kompresjon

- For å få høye kompresjonsrater, er det ofte nødvendig med ikke-tapsfri kompresjon.
- Ulempen er at man ikke kan rekonstruere det originale bildet, fordi et informasjonstap har skjedd.
- Enkle metoder for ikke-tapsfri kompresjon er rekvantisering til færre antall gråtoner, eller resampling til dårligere romlig oppløsning.
- Andre enkle metoder er basert på filtrering, f.eks. erstatt hvert ikke-overlappende 3×3-område med ett piksel som har verdi lik middelverdien eller medianverdien av de opprinnelige pikselverdiene.

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

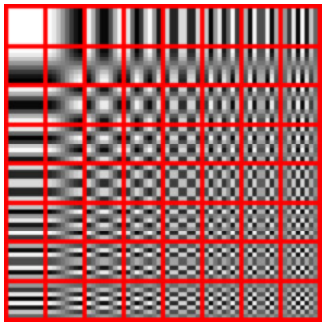
   - Mye av informasjonen i de 64 pikslene samles i en liten del av de 64 2D DCT-koeffisientene; nemlig de i øverste, venstre hjørne.

2D diskret cosinus-transform

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

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

- Strekt 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».

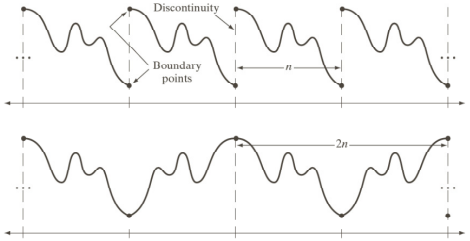


Hvorfor DCT og ikke DFT?

- Den implisitt antatte N-punkts periodisiteten i DFT-en vil introdusere høye frekvenser p.g.a. 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) kvantiseres til 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

divideres med

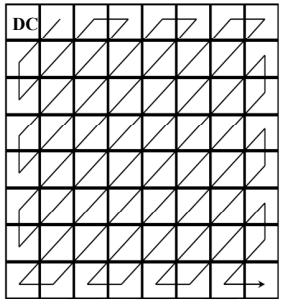
16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

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

Ikke-tapsfri JPEG-kompresjon

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. «Løpelengdetransform» av 1D-følgen.
   - Et «løpelengdepar» er her (antall 0-ere, størrelse av ikke-null).
3. «Løpelengdeparene» Huffman- eller aritmetisk kodes.
   - Både predefinerte og egendefinerte Huffman-kodebøker tillates.

Ikke-tapsfri JPEG-kompresjon

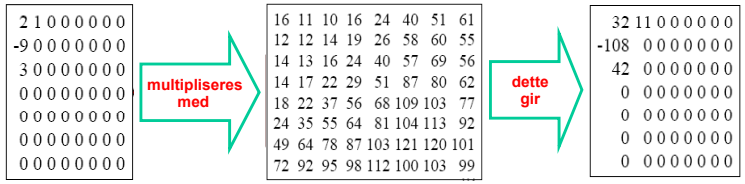
DC- og AC-elementene behandles nå separat.

DC-elementene:

- For hver kanal samles DC-elementene fra alle blokkene.
- Disse er korrelerte og blir derfor differansetransformert.
- Differansene Huffman-kodes eller aritmetisk kodes.

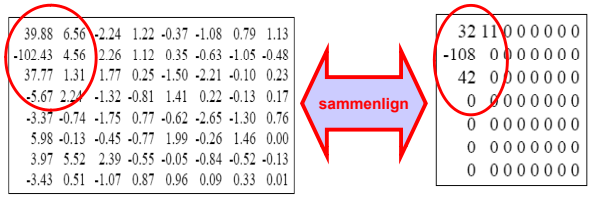
Ikke-tapsfri JPEG-dekompresjon

- 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

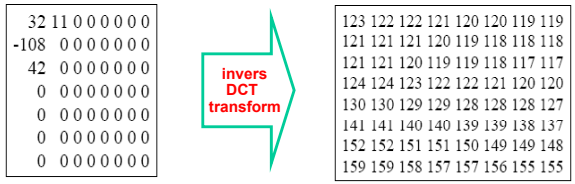
- Produktet er **IKKE** helt likt 2D DCT-koeffisientene.



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

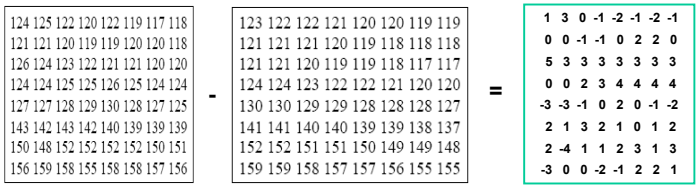
Ikke-tapsfri JPEG-dekompresjon

- Så gjør vi en invers 2D DCT, og får en rekonstruert en 8×8 piksels bildeblokk.



Ikke-tapsfri JPEG-dekompresjon

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

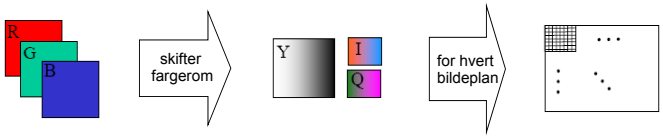


- De er **likevel ikke 0**.
- Det kan bli gjort forskjellig feil på nabopiksler, spesielt dersom de tilhører forskjellige blokker.
  - Kompresjon / dekompresjon kan derfor gi **blokk-artefakter**; rekonstruksjonsfeil som gjør at vi ser at bildet er blokk-inndelt.

Ikke-tapsfri JPEG-kompresjon av fargebilde

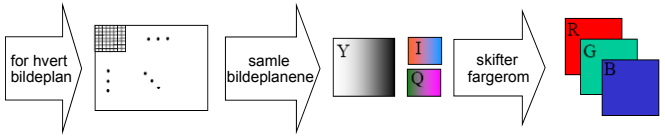
- Skifter fargerom for å separere lysintensitet fra kromasi.
  - Stemmer bedre med hvordan vi oppfatter et fargebilde.
    - Lysintensiteten er viktigere enn kromasi for oss.
  - Kan også gi lavere kompleksitet i hver kanal.
- Nedsampler (normalt) kromasitet-kanalene.
  - Typisk med en faktor 2 i begge 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 et 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

- 2D DCT 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

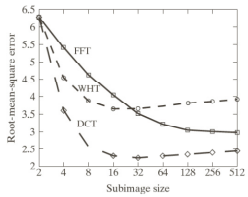
Rekonstruksjonsfeil i fargebilder

- 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:

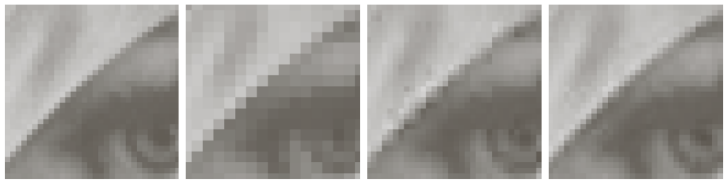
Original JPEG JPEG 2000

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 RMS-feilen.



- **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).
  - Løsning: **Entropikoding**.

Tapsfri JPEG-kompresjon

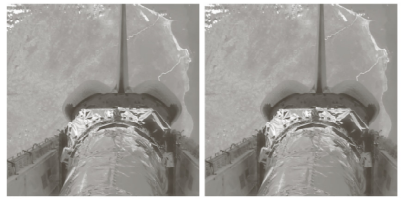
- I tapsfri JPEG-kompresjon predikeres f(x,y) ved bruk av opptil 3 elementer:
  - X er pikselen vi ønsker å predikere.
  - Benytter 1-3 av elementene A, B og C.

C	B	
A	X	

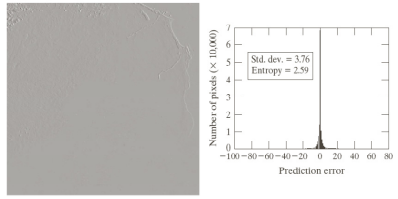
- De predikerte verdiene entropikodes.
  - Huffman-koding eller aritmetisk koding
- Kompresjonsraten er avhengig av:
  - Biter per piksel i originalbildet.
  - Entropien til prediksjonsfeilene.
- For vanlige fargebilder blir kompresjonsraten ≈2.
- Brukes for det meste kun i medisinske anvendelser.

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): C ≈ 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

- Koding av digitale bildesekvenser/video er vanligvis basert på **bevegelse-kompensasjon, prediktiv koding og 2D DCT**.
- I nyere standarder er kompensasjonen og prediksjonen basert på **både tidligere og fremtidige bilder**.
  - Typisk: Noen bilder er upredikerte, noen flere bruker kun på tidligere bilder, mens de fleste bruker på 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).
- ITU-T har også standarder for videokompresjon (gjennom sin Visual Coding Experts Group (VCEG)):
  - H.120 (1984), H.26x-familien (H.264 (2003) = MPEG-4 Part 10).

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 (bruker før koding): Løpelengdetransform, 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 lag sannsynlighetstabell for meldingen og send kodeboken.
    - LZW-koding: utnytter mønstre i meldingen. Trenger (et typisk forhåndsdefinert) alfabet, ingen kodebok!
    - Aritmetisk koding: koder symbolsekvenser som ett tall i et intervall. Send eller ha forhåndsdefinert modellen.