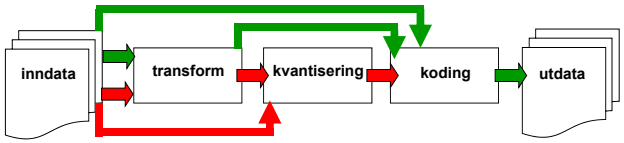
INF 2310 – Digital bildebehandling

Kompresjon og koding – Del II

LZW-koding
Aritmetisk koding
JPEG-kompresjon av gråtonebilder
JPEG-kompresjon av fargebilder
Rekonstruksjonsfeil i bilder
Tapsfri prediktiv koding

Repetisjon: Kompresjon

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



- Kompresjons kan gjøres
  - **Eksakt / tapsfri** ("loss-less") – følg de grønne pilene
    - Her kan vi rekonstruere den originale meldingen eksakt.
  - **Ikke-tapsfri** ("lossy") – følg de røde pilene
    - Her kan vi ikke rekonstruere meldingen eksakt.
    - Resultatet kan likevel være "godt nok".
- Det finnes en mengde ulike metoder for begge kategorier kompresjon.

Repetisjon: Melding, data og informasjon

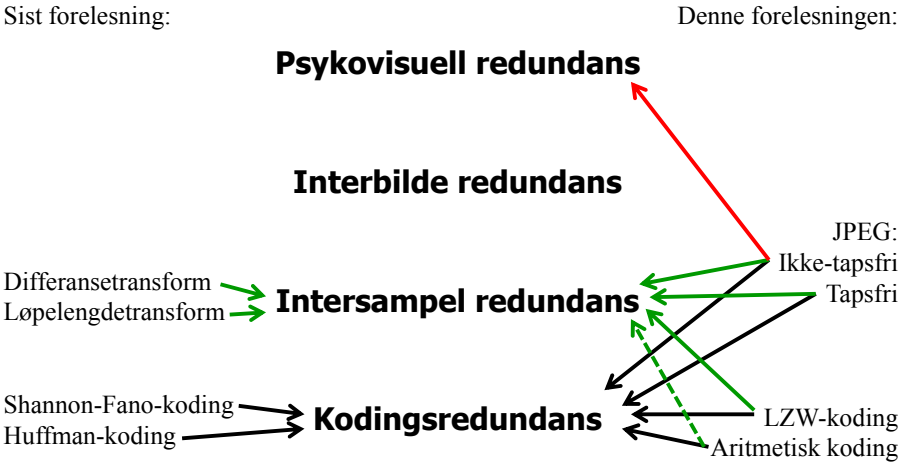
- Vi skiller mellom data og informasjon:

- **Melding:** teksten, bildet eller signalet som vi skal lagre.
- **Data:** strømmen av biter som lagres på fil eller sendes.
- **Informasjon:** et mål som kvantifiserer mengden overraskelse/uventethet i en melding.
  - Et varierende signal har mer informasjon enn et monotont signal.
  - I et bilde: kanter rundt objekter har høyest informasjonsinnhold,
    - spesielt kanter med mye krumning.

Repetisjon: Ulike typer redundans

- Psykovisuell redundans
  - Det finnes informasjon vi ikke kan se.
    - Kan kanskje subsample eller redusere antall bit per piksel.
- Interbilde redundans
  - Det er en viss likhet mellom nabobilder i en tidssekvens
    - Vi kan kode noen bilder i sekvensen og ellers bare differanser.
- Intersampel redundans
  - Nabopiksler ligner på hverandre eller er like.
    - Hver linje i bildet kan "run-length"-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 I

- Tapsfri kompresjonsmetode
  - Kan gi kortere kode enn Huffman-koding
- Premierer mønstre i meldingen
  - Ser på samforekomster av symboler.
- Bygger opp en liste av symbolsekvenser/strenger
  - både under kompresjon og dekompresjon.
- Denne listen skal ikke lagres eller sendes
  - Senderen bygger opp listen fra symbolsekvensen han skal kode.
  - Mottakeren bygger opp listen fra symbolsekvensen han mottar.
- Det eneste man trenger er et standard alfabet
  - f.eks. ASCII eller heltallene f.o.m. 0 t.o.m. 255

Lempel-Ziv-Welch-koding II

- Mottaker kjenner bare alfabetet,
  og lagrer nye fraser ved å
  - ta nest siste streng pluss første symbol i sist tilsendte streng,
    inntil listen er full (det er en praktisk grense her!).
- En ulempe er at man av og til lager kodeord
  som man ikke får bruk for.
- 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.
- Komprimerer typiske tekstfiler med en faktor på ca. 2.

Eksempel på Lempel-Ziv-Welch I

- Anta at alfabetet består av a, b og c som tilordnes kodene 1, 2 og 3.
  La meldingen være ababcbababaaaaabab (18 symboler)
  sender: ny frase = sendt streng pluss neste usendte symbol
  mottaker: ny frase = nest siste streng pluss første symbol i sist tilsendte streng

Ser	Sender	Senders liste	Mottar	Tolker	Mottakers liste
		a=1,b=2,c=3			a=1, b=2, c=3
a	1	ab=4	1	a	
b	2	ba=5	2	b	ab=4
ab	4	abc=6	4	ab	ba=5
c	3	cb=7	3	c	abc=6
ba	5	bab=8	5	ba	cb=7
bab	8	baba=9	8		

- Vi mottar en kode som ikke finnes i listen.
- Kode 8 ble laget som "ba+?" , og nå sendes kode 8.
- Altså må vi ha "?" = "b" => 8 = ba + b = bab.

Eksempel på Lempel-Ziv-Welch II

Ser	Sender	Senders liste	Mottar	Tolker	Mottakers liste
		a=1,b=2,c=3			*a=1, b=2, c=3*
a	*1*	*ab=4*	*1*	*a*	
b	*2*	*ba=5*	*2*	*b*	*ab=4*
ab	*4*	*abc=6*	*4*	*ab*	*ba=5*
c	*3*	*cb=7*	*3*	*c*	*abc=6*
ba	*5*	*bab=8*	*5*	*ba*	*cb=7*
bab	*8*	*baba=9*	8	bab	bab=8
a	1	aa=10	1	a	baba=9
aa	10	aaa=11	10	aa	aa=10
aa	10	aab=12	10	aa	aaa=11
bab	8		8	bab	aab=12

- Kode 10 ble laget idet 1 = a ble sendt, som 10 ” a+?”. Så sendes ”10”.
- Da må vi ha 10=”a+a” =”aa”.
- Istedenfor 18 tegn er det sendt 10 koder.
- 5 av 12 koder ble ikke brukt.

LZW: Et eksempel fra DIP

- Melding: aabbaabbaabbaabb (16 piksler med gråtoner a=39 og b=126)
  sender: ny frase = sendt streng pluss neste usendte symbol
  mottaker: ny frase = 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 8 biter.
- Kompresjonsrate C = (16x8)/(10x9) = 1.4222…

Aritmetisk koding I

- Tapsfri kompresjonsmetode
  - Gir ofte litt bedre kompresjonsrate enn Huffman-koding.
- Er en variabel-lengde entropikoding.
- Metoden produserer ikke kodeord for enkeltsymboler.
- Metoden koder en sekvens av symboler til et tall n ($0.0 \leq n < 1.0$).
- Dermed kan man få en kortere kode enn med Huffman-koding, fordi man ikke lenger er begrenset av at hvert symbol skal tilordnes et kodeord med et heltalls antall biter.
- Resulterer i et bitforbruk per symbol som er nær entropien.
- Er generelt bedre (målt i bitforbruk per symbol)
  - når sannsynlighetsmodellen samsvarer med de virkelige forekomstene av symboler
  - jo lenger symbolsekvensen er

Aritmetisk koding II

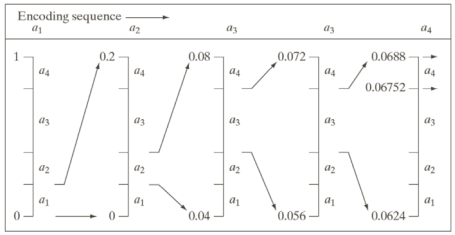
- Vi plasserer symbolsannsynlighetene etter hverandre på tallinjen fra 0 til 1.
- Får et bestemt delintervall mellom 0 og 1 som tilsvarer hvert tegn.
- Har vi to tegn etter hverandre, så blir det delintervallet som representerer dette tegnparet et delintervall av intervallet til det første tegnet, osv.
- Et kodeord for en lengre sekvens av tegn vil da beskrive et halvåpent delintervall av det halvåpne enhetsintervallet [0, 1), og kodeordet må inneholde akkurat tilstrekkelig mange biter til å gi en entydig peker til det riktige delintervallet.

AK: Algoritmen

1. Vi starter med et "current interval" = [0,1).
2. For hvert nytt symbol gjør vi to ting:
   a) Vi deler opp "current interval" i nye delintervaller, der størrelsen på hvert delintervall er proporsjonal med sannsynligheten for vedkommende symbol (og proporsjonalitetsfaktoren er størrelsen av "current interval")
   b) Vi velger det delintervallet av "current interval" som svarer til det symbolet vi faktisk har truffet på, og gjør dette til vårt nye "current interval".
3. Til slutt bruker vi så mange biter til å representere det endelige intervallet at det entydig skiller dette intervallet fra alle andre mulige intervaller.

Eksempel på aritmetrisk koding

- Anta at vi har en 5-tegns symbolsekvens $a_1a_2a_3a_3a_4$
- Anta at symbolsannsynlighetene er lik forekomsthyppighetene; $P(a_1) = P(a_2) = P(a_4) = 0{,}2$ og $P(a_3) = 0{,}4$



- 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)

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

- Anta at vi har et alfabet {a,b,c} med sannsynligheter {0.6,0.2,0.2}.
- Hvilket intervall mellom 0 og 1 vil entydig representere teksten **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.
- **aca** ligger i intervallet [0.48+0.12*0, 0.48+0.12*0.6) = [0.48, 0.552).
  - Intervallbredden er 0.072.
- **acab** ligger i intervallet [0.48+0.072*0.6, 0.48+0.072*0.8) = [0.5232, 0.5376).
  - Intervallbredden er 0.0144.
- **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 teksten **acaba**, forutsatt at mottakeren har den samme modellen.

AK: Dekodingseksempel I

- ***Anta at vi skal dekode tallet 0.53125***

- Vi starter med intervallet [0,1) som "current interval".
- Deler opp i 3 intervaller; a med [0, 0.6), b med [0.6, 0.8) og c med [0.8, 1).
- Vårt tall 0.53125 ligger i delintervallet for a, [0, 0.6): **=> Første symbol; a.**
- Så deler vi opp intervallet [0, 0.6) for å finne neste symbol:
  - intervallet for a er [0, 0.36) *bredde 60% av [0, 0.6)*
  - intervallet for b er [0.36, 0.48) *bredde 20% av [0, 0.6)*
  - intervallet for c er [0.48, 0.6) *bredde 20% av [0, 0.6)*
- Vårt tall 0.53125 ligger i delintervallet [0.48, 0.6): **=> Neste symbol; c.**
- Vi deler opp intervallet [0.48, 0.6) for å finne neste symbol:
  - intervallet for a er [0.48, 0.552)
  - intervallet for b er [0.552, 0.576)
  - intervallet for c er [0.576, 0.6).
- Vårt tall 0.53125 ligger i delintervallet [0.48, 0.552): **=> Tredje symbol; a.**

AK: Dekodingseksempel II

- ***Anta at vi skal dekode tallet 0.53125***

- Fortsetter med å dele opp intervallet [0.48, 0.552) for å finne neste symbol:
  - intervallet for a er [0.48, 0.5232)
  - intervallet for b er [0.5232, 0.5376)
  - intervallet for c er [0.5376, 0.552)
- Vårt 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:
  - intervallet for a er [0.5232, 0.53184)
  - intervallet for b er [0.53184, 0.53472)
  - intervallet for c er [0.53472, 0.5376).
- Vårt 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.

Representasjon av desimaltall

- Vi sender ikke desimaltall, men et bitmønster.
- Hvor mange biter trenger vi for å gi en entydig representasjon av et intervall?
- Vi kan skrive tallet N i intervallet [0,1)
  som en veiet sum av negative toerpotenser

$$N_{10} = c_1*2^{-1} + c_2*2^{-2} + c_3*2^{-3} + \ldots + c_n*2^{-n} +$$

  der hver c_i er enten 0 eller 1.
- Rekken av koeffisienter $c_1c_2c_3c_4\ldots$ utgjør da bitmønsteret i den binære representasjonen av tallet.
- Det er dette vi mener når vi skriver desimaltallet N som
  $N=0.c_1c_2c_3c_4\ldots_{\,2}$

Desimaltall som bitsekvens

- For å konvertere et desimaltall i titallsystemet til et binært tall kan vi bruke suksessive multiplikasjoner med 2:

1. Vi multipliserer begge sider av ligningen nedenfor med 2:

   $$N_{10} = c_1*2^{-1} + c_2*2^{-2} + c_3*2^{-3} + \ldots + c_n*2^{-n} + \ldots$$

   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 er 0 er vi ferdige.
2. Multipliser resten R med 2.
   - Heltallsdelen av resultatet er neste bit.
3. Hvis resten er 0 er vi ferdige. Ellers går vi til 2.

Representasjon av intervall I

•Vi kan bruke intervallet [0.5232, 0.53184) som eksempel.

- 0.53125_{10} er (som sagt) et desimaltall i dette intervallet.

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 kan kode intervallet vårt med det binære desimaltallet **$.10001_2$** (ekvivalent med 0.53125_{10}), altså bare 5 biter.

Representasjon av intervall II

- Finn kortest mulig N=0.$c_1c_2c_3$... innenfor [0.6, 0.7).
- Når c_i er 0 eller 1 og n ≥ k så gjelder følgende:
  $c_k*2^{-k} + ... + c_n*2^{-n} < 2^{-(k-1)}$
- Ergo:
  N = 0.**1...** => 0.5 ≤ N < 1
  N = 0.**10...** => 0.5 ≤ N < 0.75
  N = 0.**100...** => 0.5 ≤ N < 0.625
  N = 0.**101...** => 0.625 ≤ N < 0.75
- Vi kan kode intervallet vårt med det binære desimaltallet **$.101_2$** (ekvivalent med 0.625_{10}), altså med bare 3 biter.
  - I noen situasjoner vil vi kreve at øvre og nedre grense er innenfor intervallet; i dette eksempelet vil $.1010_2$ brukes i et slikt tilfelle:
    N = 0.**1010...** => 0.625 ≤ N < 0.6875

AK: Problemer og løsninger

- Den stadige krympingen av "current interval" krever aritmetikk med stadig økende presisjon etter hvert som teksten blir lengre. Kompresjonsmetoden gir ingen output før hele sekvensen er behandlet.
- Løsning: Send den mest signifikante biten straks den er entydig kjent, og så doble lengden på "current interval", slik at det bare inneholder den ukjente delen av det endelige intervallet.
- Det finnes flere praktiske implementasjoner av dette
  - 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 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.
- Dekoderen må ha den samme modellen som koderen!

JPEG-standarden

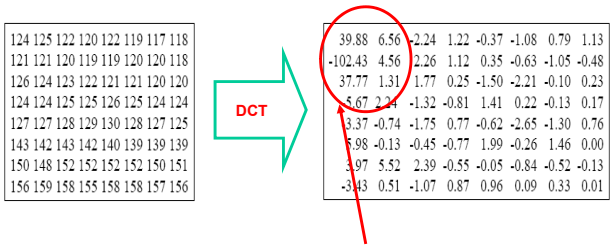
- JPEG (Joint Photographic Expert Group)
  er en av de vanligste bildekompresjonmetodene.
- JPEG-standarden (fra 1992) har varianter
  både for tapsfri og ikke-tapsfri kompresjon.
  - Den tapsfrie varianten er ikke JPEG-LS!
- I begge tilfeller brukes
  enten Huffman-koding eller aritmetisk koding.
- 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 filtering der f.eks. 3×3 piksler erstattes med ett nytt piksel som er enten 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 gråtoner
   - Gjør at forventet gjennomsnittlig gråtone er 0
   - Eksempel: For intensitetsintervallet [0, 255]; trekk 128 fra alle pikselverdiene.
3. Hver blokk transformeres med 2D DCT (diskret cosinus-transform)



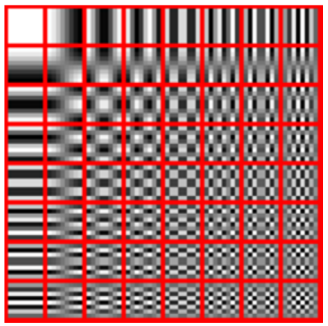
- Mye av informasjonen i de 64 pikslene samles i en liten del av de 64 DCT-koeffisientene; nemlig de i øverste, venstre hjørne

2D diskret cosinus-transform

- Grunnpilaren i ikke-tapsfri JPEG-kompresjon er altså
  den 2D diskrete cosinus-transformen (2D DCT-en):

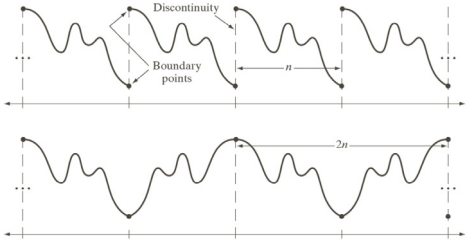
$$F(u,v) = \frac{2}{\sqrt{MN}} c(u)c(v) \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y) \cos\left[\frac{\pi u}{M}\left(x+\frac{1}{2}\right)\right] \cos\left[\frac{\pi v}{N}\left(y+\frac{1}{2}\right)\right], \qquad c(\xi) = \begin{cases} \frac{1}{\sqrt{2}} & for\ \xi = 0 \\ 1 & ellers \end{cases}$$

- Dette er essensielt realdelen til en 2D DFT med halverte frekvenskomponenter
- Vi får 8x8 "cosinus-bilder" med halv frekvens / dobbel periode:
  - For 2D DFT hadde vi også noen sinus-bilder
  - Siden vi alltid deler opp i 8x8-blokker brukes kun disse 8x8 "cosinus-bildene" i JPEG
  - Derfor forhåndsberegnes bildene og vi kan beregne 2D DCT-ene raskt
  - 2D DCT-koeffisientene beregnes som vi gjorde for 2D DFT; summere punktproduktet mellom bildet og hvert av "cosinus-bildene"



Hvorfor DCT og ikke DFT?

- Den implisitte N-punkts periodisiteten i DFT-en vil introdusere høye frekvenser på randdiskontinuitet
  - Fjerner vi disse frekvensene får vi kraftige blokk-artefakter.
  - Beholder vi dem reduseres kompresjonsraten.



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

Ikke-tapsfri JPEG-kompresjon II

JPEG-kompresjonen fortsetter med at

4.2D DCT-koeffisientene

a) skaleres med en vektmatrise og

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

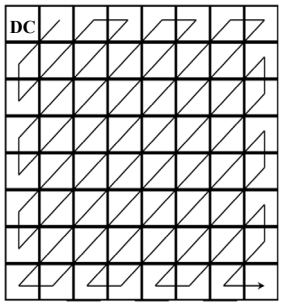
16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

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 III

DC- og AC-koeffisientene behandles nå separat;



- Sikk-sakk-skanning ordner AC-koeffisientene i en 1D-følge.
  - Absoluttverdien av koeffisientene vil stort sett avta utover i følgen.
  - Mange koeffisienter er null.
- Løpelengde-transform av koeffisientene
- Huffman-koding eller aritmetisk koding av løpelengdene.
  - Både predefinerte og egendefinerte Huffman-kodebøker tillates.

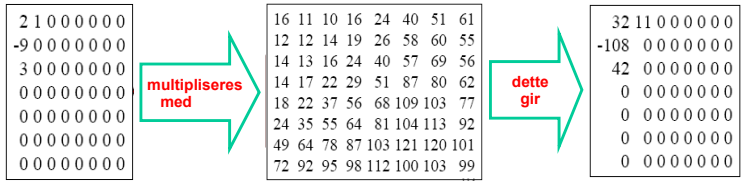
Ikke-tapsfri JPEG-kompresjon IV

DC- og AC-koeffisientene behandles nå separat;

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

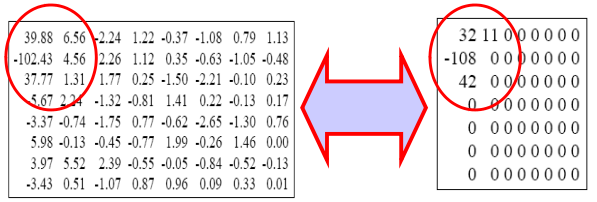
JPEG-dekompresjon I

- Huffman-kodingen og/eller den aritmetiske kodingen reversibel, og gir AC-løpelengdene og DC-differansene.
- Løpelengdetransformen og differansetransformen er reversibel, og gir de kvantifiserte 2D DCT-koeffisientene.
- Sikk-sakk-transformen er reversibel, og gir en heltallsmatrise.
- Denne matrisen multipliseres med vektmatrisen.



JPEG-dekompresjon II

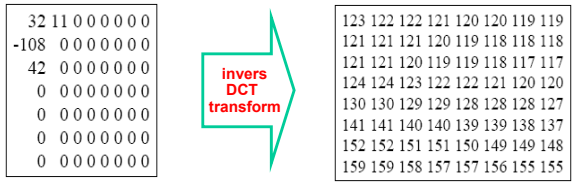
- Dette er IKKE helt likt koeffisientene etter forlengs 2D DCT.



- Men de store trekkene er bevart:
  - Her (og oftest); de store tallene i øvre venstre hjørne.
- De fleste tallene i matrisen er lik 0,
  men disse var også opprinnelig nær 0.

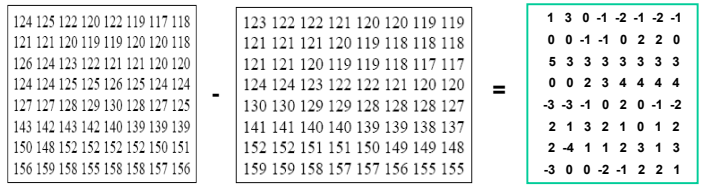
JPEG-dekompresjon III

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



JPEG-dekompresjon IV

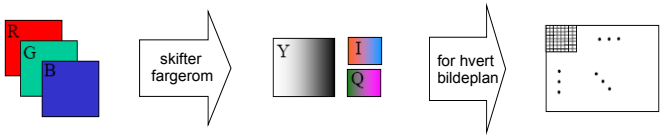
- Differansene fra den originale blokken er små!



- 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-effekter i bildet.

JPEG-kompresjon av fargebilde

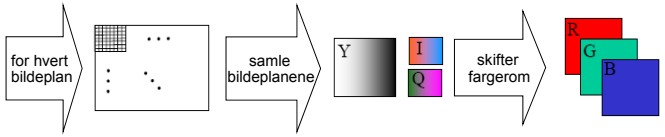
- Skifter fargerom for å separere lysintensitet fra kromasi.
- Nedsampler kromasikanalene
  - Typisk med en faktor 2 i begge retninger.
  - Sparer plass, men fjerner for det meste psykovisuell redundans.
- Hver bildekanal deles opp i blokker på 8x8 piksler, og hver blokk kodes separat.



- Hver blokk transformeres med 2D DCT.
- Forskjellige vektmatriser for intensitet og kromatisitet.
  - ..... resten er som for gråtonebilder ...

JPEG-dekompresjon av fargebilde

- Alle dekomprimerte 8x8-blokker i hvert bildeplan samles til et bilde.
- Bildeplanene samles til et YIQ fargebilde
- Vi skifter fargerom
  - fra YIQ til RGB for fremvisning,
  - … til CMYK for utskrift.



- Vi har redusert oppløsning i Y og Q, men full oppløsning i RGB:
  - Gir 8 × 8 blokkeffekt i intensitet,
  - Ved en faktor 2 nedsampling i hver retning av kromasikanalene får vi 16 × 16 piksels blokkeffekt i fargene i RGB

Rekonstruksjonsfeil i gråtonebilder

- 2D DCT kan gi 8×8 piksels ”blokk-artefakter”, ”blurring” og ”ringinger”.
- Avhengig av vektmatrisen
  - definerer indirekte antall ikke-null koeffisienter

Blokk-artefakter

- Blokk-artefaktene øker med kompresjonsraten.

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

Eksperiment: Skalering av vektmatrisen

- Vi har brukt vektmatrisen Z:

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

Skalerer vi Z med

1, 2, 4

8, 16, 32

får vi C =

12, 19, 30

49, 85, 182

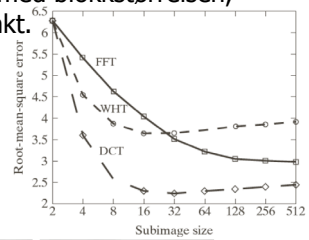
Rekonstruksjonsfeil i fargebilder

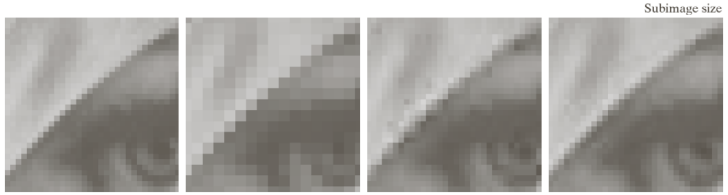
- 24 biters RGB komprimert til 1.5-2 biter per piksel (bpp)
- 0.5 – 0.75 bpp gir god/meget god kvalitet
- 0.25 – 0.5 bpp gir noen feil
  - 8x8-blokkeffekt i intensitet
  - Fargefeil i muligens større blokker
- JPEG 2000 bruker ikke blokker
  - Høyere kompresjon
  - Mye bedre kvalitet:

Original JPEG JPEG 2000

Blokkstørrelse

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





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

- "Ringings"-problemet øker med blokkstørrelsen.

Tapsfri JPEG-kompresjon I

- I den tapsfrie varianten av JPEG benyttes *prediktiv koding*.
- Generelt for prediktiv koding så kodes
  e(x,y) = f(x,y) - g(x,y) der g er *predikert* fra m naboer i stedet for pikselverdiene f(x,y)
- En 1D lineær prediktor av orden m: $g(x,y) = round\left[\sum_{i=1}^{m} \alpha_i f(x-i,y)\right]$
- En førsteordens lineær prediktor: $g(x,y) = round\left[\alpha f(x-1,y)\right]$
  - Hvilken transform er dette hvis α=1?
- Med lik-lengde koding trenger vi et ekstra bit per piksel
- Bruker derfor entropikoding

Tapsfri JPEG-kompresjon II

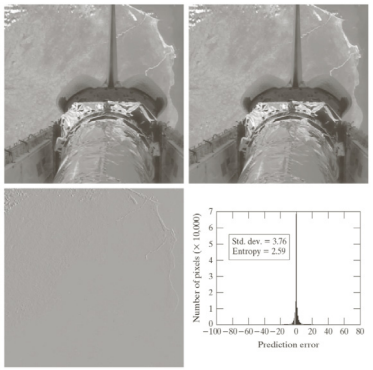
- I tapsfri JPEG-kompresjon predikeres f(x,y) ved bruk av opptil 3 elementer:
  - X er pikselen vi ønsker å predikere, mellom 1 og 3 av elementene A, B og C benyttes

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 omtrent 2
- Brukes for det meste kun i medisinske anvendelser

Tapsfri koding av bildesekvenser

- Prediksjon også mulig i tidssekvenser: $g(x,y,t) = round\left[\sum_{i=1}^{m} \alpha_i f(x,y,t-i)\right]$
- Enklest mulig: første ordens 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 enkeltsymboler): C ≈ 8/2.59 ≈ 3



- Bevegelsesdeteksjon og bevegelseskompensasjon innenfor blokker er nødvendig for å øke kompresjonsraten.
  - Blokkene kalles ofte makroblokker og kan f.eks. være 16x16

Digital video

- Koding av digitale bildesekvenser eller digital video er vanligvis basert på prediktiv koding og 2D DCT.
- I nyere standarder er prediksjonen basert på både tidligere og fremtidige bilder.
  - Områder uten interbilde bevegelse kodes ikke flere ganger, kun koding i områder der endringer skjer.
- 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

- Hensikten med kompresjon er mer kompakt lagring eller raskere oversending av informasjon.
- Kompresjon er basert på informasjonsteori.
- Antall biter per sampel er sentralt, og varierer med kompresjonsmetodene og meldingene.
- Sentrale metoder:
  - Før koding; transformer: løpelengdetransform, 2D DCT, og prediktiv koding; differansetransform, differanse i tid m.m.
  - Koding:
    - Huffman-koding: lag sannsynlighetstabell, send/ha forhåndsdefinert kodeboken.
    - LZW-koding: utnytter mønstre i meldingen, definer alfabet, ikke send kodebok.
    - Aritmetisk koding: koder symbolsekvenser som ett tall i et intervall, send/ha forhåndsdefinert modellen.