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

Ulike typer redundans

- Psykovisuell redundans
  - Det finnes informasjon vi ikke kan se.
    - Da kan vi sub-sample, eller redusere antall bit per piksel.
- Interbilde redundans
  - Det er en viss likhet mellom nabo-bilder i en tids-sekvens
    - Vi koder noen bilder i sekvensen, og deretter bare differanser.
- Intersampel redundans
  - Nabo-piksler 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, med liten redundans.

Kodingsredundans – et eksempel

- Hva blir kodingsredundansen når du Huffman-koder teksten "**multiple choice!**" ?
- Svar: Det er 16 symboler. Normalisert histogram blir slik:

  l, i, e, c : 2/16

  m, n, t, p, h, o, ' ',! : 1/16

  Alle sannsynlighetene er $p(s_i) = \frac{1}{2^{k_i}}$, der k_i er heltall.

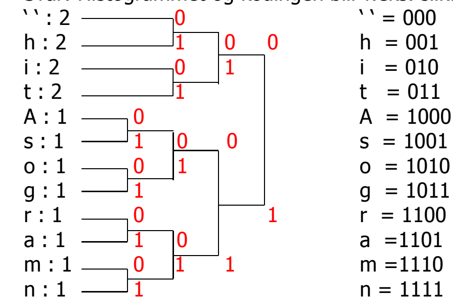
  Da vet vi at gjennomsnittlig ordlengde R = entropi H.

  Kodingsredundans = R – H.

  Her er kodingsredundansen lik null!

Et annet eksempel

- Hva blir R, H og kodingsredundansen ved Huffman-koding av teksten "**A histogram hint**" når vi skiller mellom store og små bokstaver?
- Svar: Histogrammet og kodingen blir f.eks. slik:



- R = (8*3 + 8*4)/16 = (24 + 32)/16 = 56/16 = 3.5
- H = $4(1/8)\log_2(8) + 8(1/16)\log_2(16) = 3/2 + 2 = 3.5$
- R - H = 0.

Universell koding

- Huffman-koding bygger på at man finner antall forekomster av hvert symbol i teksten, og lager kodene fra dette.
  - Kodene må lages fra gang til gang, og kodeboken må også oversendes.
- Er det mulig å finne en universell kode for hvor mange ganger et symbol forekommer?
  - Vi kan f.eks. se på forekomster av ulike bokstaver i norsk eller engelsk tekst (F.ex. Morse-kode)
    - ”e” forekommer oftest,
    - ”z”, ”x” og ”q” er sjeldne
  - Eller vi kan bygge opp kodestrenger etterhvert som vi ser gjentatte mønstere. Lempel-Ziv koding er et eksempel på universell koding av symboler (f.eks. en bitsekvens).

Lempel-Ziv-koding

- Premierer mønstre i dataene,
  ser på samforekomster av symboler.
- Bygger opp en symbolstreng-liste
  både under kompresjon og dekompresjon.
- Denne listen skal ikke lagres eller sendes,
  for mottakeren kan bygge opp listen
  ved hjelp av den symbolstrengen han mottar.
- Det eneste man trenger er et standard alfabet
  - (f.eks ASCII).

Lempel-Ziv-koding

- 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 siden 1986.
- Ble en del av GIF-formatet i 1987.
- Er en opsjon i TIFF-formatet.
- Finnes i Abobe Acrobat.
- Komprimerer typisk tekst-filer med ca faktor 2.

Eksempel på Lempel-Ziv

- Anta at alfabetet er a, b og c som tilordnes kodene 1, 2 og 3.
  La dataene være ababcbababaaaaabab (18 tegn)
  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 - 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.

Et eksempel

- Data : 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	liste	mottar	tolker	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	bbb=264	257	ab	263=baa
b	126		126	b	264=bbb

- Listen utvides fra 8 til 9 biter
- Kompresjonsrate C= (16x8)/(10x9) = 1.42…

Aritmetisk koding

- Aritmetisk koding er en metode for tapsfri kompresjon.
- Det er en variabel lengde entropi-kode.
- Metoden produserer ikke kode-ord for enkelt-symboler.
- Metoden koder en sekvens av symboler til et tall n ($0.0 \le n < 1.0$).
- Dermed kan man få en mer optimal koding enn med Huffman-koding, fordi man ikke lenger er begrenset til et heltalls antall biter per tegn i sekvensen.
- Produserer nær optimal kode for et gitt sett av symboler og sannsynligheter.
- Jo bedre samsvar det er mellom modellen og de virkelige forekomstene i sekvensen, desto nærmere optimalitet kommer man.

Aritmetisk koding - 2

- Vi plasserer symbol-sannsynlighetene etter hverandre på tallinjen fra 0 til 1.
- Får et bestemt del-intervall 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.

Algoritme i prosa

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

Algoritme for oppdeling i pseudo-kode

```
low = 0.0; high = 1.0; // nedre og øvre grense for startintervall
While  <det finnes fortsatt nye symboler> do
    range = high-low; // bredden på dette intervallet
    low = low + range*cumprob(symbol-1) //cumprob for symbol 0 er 0
    high = low +  range*cumprob(symbol) //cumprob til siste symbol er 1
end
```


Statisk modell

- ***Anta at vi har følgende enkle statiske sannsynlighets-modell:***
  - 60% sannsynlighet for symbolet A
  - 20% sannsynlighet for symbolet B
  - 10% sannsynlighet for symbolet C
  - 10% sannsynlighet for symbol END-OF-DATA.
  - *Siden EOD tas med i modellen, får vi en "intern terminering".*
  - *Dette er ganske vanlig i praktisk data-kompresjon.*

Koding

- Kodingen deler "current interval" opp i sub-intervaller
- Hvert sub-intervall har en bredde som er proporsjonal med sannsynligheten til et gitt symbol i "current context".
- Det intervallet som svarer til det neste symbolet i sekvensen blir "current interval" i neste steg.
- ***Eksempel***: for den 4-tegns modellen vi har:
  - intervallet for A er [0.0, 0.6)
  - intervallet for B er [0.6, 0.8)
  - intervallet for C er [0.8, 0.9)
  - intervallet for EOD er [0.9, 1).

Et kodings-eksempel

- 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. 0.53184, vil entydig representere teksten **acaba**, forutsatt at mottakeren har den samme modellen.

Eksempel på dekoding

- ***Merk: dette er IKKE samme eksempel som på forrige foil, vi har nå med EOD.***
- ***Anta at vi skal dekode tallet 0.538***
- Vi starter med intervallet [0,1) som "current interval".
- Vi bruker modellen med EOD, og deler opp i 4 sub-intervaller
- Vårt tall 0.538 ligger i sub-intervallet for A, [0, 0.6): **=> Første tegn er A.**
- Så deler vi opp intervallet [0, 0.6) for å finne neste tegn:
  - 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, 0.54) bredde *10% av [0, 0.6)*
  - intervallet for EOD er [0.54, 0.6) bredde *10% of [0, 0.6)*
- Vårt tall .538 er i intervallet [0.48, 0.54): **=> Neste tegn må være C.**
- Vi deler opp intervallet [0.48, 0.54):
  - intervallet for A er [0.48, 0.516)
  - intervallet for B er [0.516, 0.528)
  - intervallet for C er [0.528, 0.534)
  - intervallet for EOD er [0.534, 0.540).
- Tallet .538 ligger i intervallet for EOD: **=> Dekodingen er ferdig.**

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

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

- Rekken av koeffisienter $c_1 c_2 c_3 c_4 \ldots$ utgjør da bitmønsteret i den binære representasjonen av tallet.
- Det er dette vi mener når vi skriver desimaltallet N som $N = 0.c_1 c_2 c_3 c_4 \ldots_2$

Desimaltall som bit-sekvens

- 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 [0.534, 0.540) som eksempel.
  - 0.5390625_{10} er et desimaltall i dette intervallet.

2*0.5390625 = 1.078125 -> c_1 = **1**, rest =0.078125
2*0.078125 = 0.15625 -> c_2 = **0**, rest =0.15625
2*0.15625 = 0.3125 -> c_3 = **0**, rest =0.3125
2*0.3125 = 0.625 -> c_4 = **0**, rest =0.625
2*0.625 = 1.25 -> c_5 = **1**, rest =0.25
2*0.25 = 0.50 -> c_6 = **0**, rest = 0.5
2*0.5 = 1.0 -> c_7 = **1**, rest = 0 .

- Vi kan kode intervallet vårt med det binære desimaltallet $.1000101_2$ (ekvivalent med 0.5390625_{10}) med bare 7 biter.

Representasjon av intervall - II

- Finn kortest mulig $N=0.c_1c_2c_3...$ innenfor [0.6, 0.7).
- Husk at i totall-systemet gjelder følgende :

$$c_k*2^{-k} + ... + c_n*2^{-n} < c_{k-1}*2^{-(k-1)}$$

- Ergo:

N = 0.**1**... => $0.5 \leq N < 1$
N = 0.**10**... => $0.5 \leq N < 0.75$
N = 0.**100**... => $0.5 \leq N < 0.625$
N = 0.**101**... => $0.625 \leq N < 0.75$
N = 0.**1010**... => $0.625 \leq N < 0.6875$

- Vi kan kode intervallet vårt med det binære desimaltallet **.1010** (ekvivalent med 0.625 desimalt) med bare 4 biter.

Problemer – og løsninger

- Den stadige krympingen av ”current interval” krever aritmetikk med stadig økende presisjon etter hvert som teksten blir lengre. Metoden 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
  - de aller fleste er belagt med patenter.

Ikke-statistiske modeller

- Statiske histogram-baserte modeller er ikke optimale.
- ***Høyere ordens modeller*** endrer estimatet av sannsynligheten for et tegn (og dermed del-intervallet) basert på foregående tegn (context).
  - 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 en tilpasning til den faktiske datastrømmen.

- Dekoderen må ha den samme modellen som koderen!

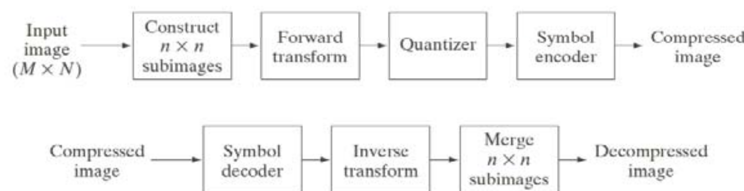
JPEG-koding (tapsfri)

- JPEG (Joint Photographic Expert Group)
  er et av de vanligste bildeformatene med kompresjon.
- JPEG-standarden har varianter både for tapsfri og ikke-tapsfri kompresjon.
- JPEG kan bruke enten Huffman-koding
  eller aritmetisk koding.
- Prediktiv koding brukes for
  - å predikere at neste piksel har lignende verdi som forrige piksel
  - å predikere at en piksel har lignende verdi som pikselen på linjen over
  - å predikere at neste piksel på linjen har lignende verdi som de tre nærmest pikslene
- Typen koding bestemmes fra bilde til bilde

Ikke-tapsfri (lossy) 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 erstatter med ett nytt piksel som er enten middelverdien eller medianverdien av de opprinnelige pikselverdiene.

Blokk-transform koding

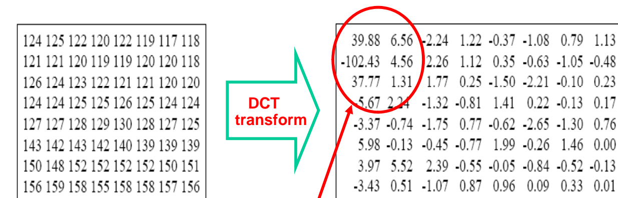


- Del bildet i subbilder
- Bruk en lineær transform (DFT, DCT..) på hvert subbilde.
- Sett frekvenskoeffisienter nær null til 0
- Kod kun de sterkeste frekvenskomponentene

"Lossy" JPEG-kompresjon av bilde

- Bildet deles opp i blokker på 8x8 piksler, og hver blokk kodes separat.
- Trekk fra 128 fra alle pikselverdiene.
- Hver blokk transformeres med DCT (Diskret Cosinus Transform):

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



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

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

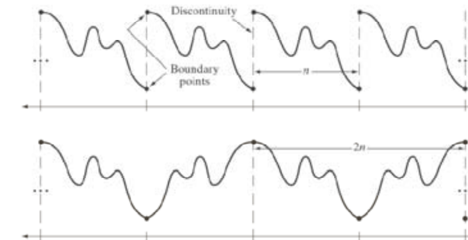
- Informasjonen i de 64 pikslene samles i en liten del av de 64 koeffisientene
  - Mest i øverste venstre hjørne

Valg av transform

- Mange transformer kan brukes, for eksempel Fourier, Cosinus, Karhunen-Loeve, Walsh++++ (se for eksempel foilene til IN 384.
- En god transform vil ha energien konsentrert rundt få koeffisienter (for eksempel lave Fourier-komponenter).
- Den teoretisk beste er Karhunen-Loeve, men der må basisfunksjonene beregnes ved egenvektordekomponering av bildet. Det kan vises at Cosinus-transform er en god tilnærming til K-L.
- Fourier er ikke så godt egnet pga. diskontinuitet ved kantene.

Hvorfor DCT?

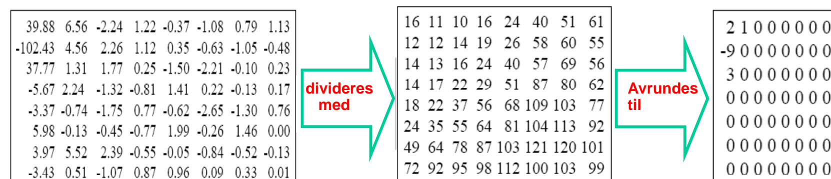
- Den implisitte N-punkts periodisiteten i Fourier-transformen vil introdusere høye frekvenser, og gir kraftige blokk-artefakter.



- DCT har en implisitt 2N-punkts periodisitet, og unngår denne effekten.

"Lossy" JPEG-kompresjon – 2

- Transformkoeffisientene
  – skaleres med en vektmatrise
  – kvantiseres til heltall.

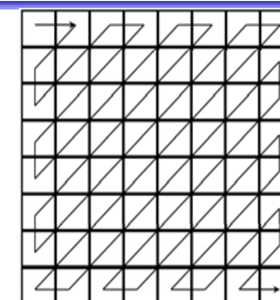


39.88	6.56	-2.24	1.22	-0.37	-1.08	0.79	1.13
-102.43	4.56	2.26	1.12	0.35	-0.63	-1.05	-0.48
37.77	1.31	1.77	0.25	-1.50	-2.21	-0.10	0.23
-5.67	2.24	-1.32	-0.81	1.41	0.22	-0.13	0.17
-3.37	-0.74	-1.75	0.77	-0.62	-2.65	-1.30	0.76
5.98	-0.13	-0.45	-0.77	1.99	-0.26	1.46	0.00
3.97	5.52	2.39	-0.55	-0.05	-0.84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

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

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

"Lossy" JPEG-kompresjon – 3



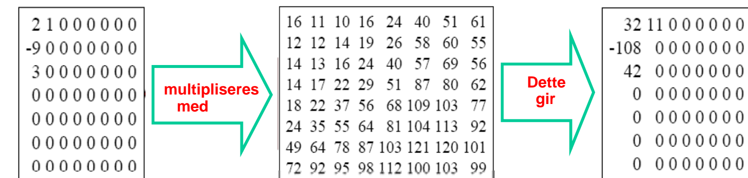
- Sikk-sakk-scanning ordner koeffisientene i 1D-rekkefølge.
  – Koeffisientene vil da stort sett avta i verdi utover i rekka
  – Mange koeffisienter er rundet av til null.
  – Løpelengde-transform av koeffisientene
  – Huffman-koding av løpelengdene.
- Huffman-koden og kodeboken sendes til mottaker / lager.
- Gjentas for alle blokker i alle kanaler.

Aritmetisk koding av DC-koeff.

- DC-koeffisientene (frekvens u=0,v=0) fra alle blokkene legges etter hverandre.
- Disse er korrelerte og bør differanse-transformeres.
- Kjenner sannsynligheten for hvert symbol i "alfabetet".
  - Deler opp intervallet [0,1) etter sannsynlighetene
    - Velger intervallet som svarer til første tegn
    - Deler intervallet i del-intervaller
    - Velger del-intervallet som svarer til andre tegn
    - Osv. En symbolsekvens gir et tall i et intervall !!!
- Bruker så mange biter at vi får en entydig beskrivelse.

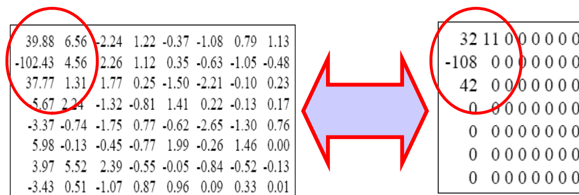
JPEG dekompresjon - 1

- Huffman-koden for en blokk er reversibel og gir løpelengdene.
- Løpelengdetransformen er reversibel, gir kvantiserte DCT-koeffisienter.
- Sikk-sakk transformen er reversibel, og gir en heltallsmatrise.
- Denne matrisen multipliseres med vektmatrisen.



JPEG dekompresjon - 2

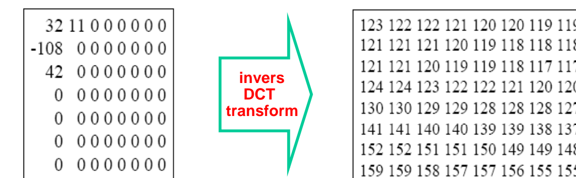
Dette er IKKE helt likt koeffisientene etter forlengs DCT-transformasjon.



- Men de store trekkene er bevart:
  - De største tallene ligger i øvre venstre hjørne
  - De fleste tallene i matrisen er lik 0.

JPEG dekompresjon – 3

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



JPEG dekompresjon – 4

- Differansene fra den originale blokken er små!

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

-

123	122	122	121	120	120	119	119
121	121	121	120	119	118	118	118
121	121	120	119	119	118	117	117
124	124	123	122	122	121	120	120
130	130	129	129	128	128	128	127
141	141	140	140	139	139	138	137
152	152	151	151	150	149	149	148
159	159	158	157	157	156	155	155

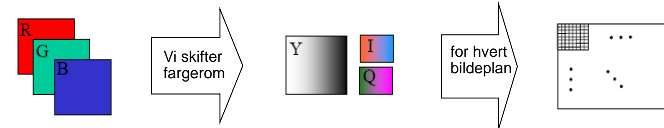
=

1	3	0	-1	-2	-1	-2	-1
0	0	-1	-1	0	2	2	0
5	3	3	3	3	3	3	3
0	0	2	3	4	4	4	4
-3	-3	-1	0	2	0	-1	-2
2	1	3	2	1	0	1	2
2	-4	1	1	2	3	1	3
-3	0	0	-2	-1	2	2	1

- Kompresjon / dekompresjon blokk for blokk
  gir blokk-effekter i bildet.

JPEG-kompresjon av fargebilde

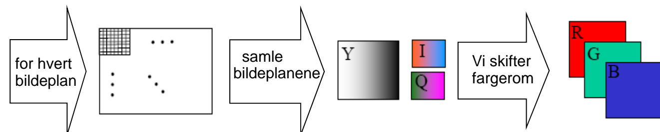
- Vi skifter fargerom slik at vi separerer
  lysintensitet fra kromasi
  (perseptuell redundans, sparer plass).
- Bildet deles opp i blokker på 8x8 piksler,
  og hver blokk kodes separat.



- Hver blokk transformeres med 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,
  - CMYK for utskrift.



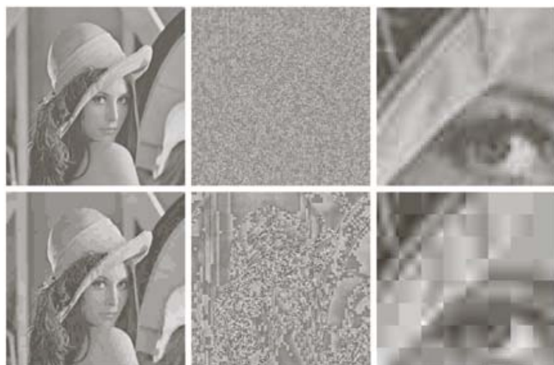
- Vi har redusert oppløsning i Y og Q,
  men full oppløsning i RGB:
  - Gir 8 × 8 blokkeffekt i intensitet
  - 16 × 16 piksels blokkeffekt i fargene i RGB

Rekonstruksjons-feil i gråtonebilder

- DCT kan gi 8×piksels "blokk-artefakter", sløring og dobbelt-konturer.
- Avhengig av
  - vektmatrise
  - antall koeffisienter

Blokk-artefakter

- Blokk-artefaktene øker med kompresjonsraten.



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

Skalering av vekt-matrisen

- 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

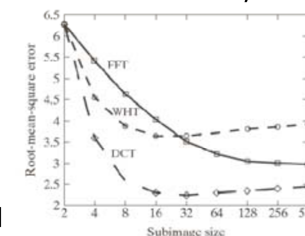
49, 85, 182

Rekonstruksjons-feil i fargebilder

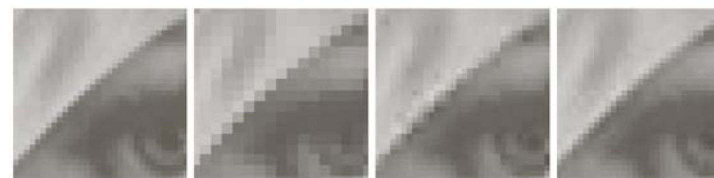
- 24 biters RGB komprimert til 1.5-2 biter per piksel (bpp)
- 0.5 – 0.75 bpp gir god/meget god kvalitet
- 0.25 – 0.5 bpp gir noen feil
  - Fargefeil i makroblokk
- JPEG gir 8×8 blokkeffekt
- JPEG 2000 uten blokker:
  - Høyere kompresjon
  - Mye bedre kvalitet

Blokkstørrelse

- Kompresjon og kompleksitet øker med blokkstørrelsen, men rekonstruksjonsfeilen avtar.
- Eksperiment:
  - Del opp bildet i nxn piksels blokker
  - Transformer
  - Behold 25% av koeffisientene
  - Gjør invers transform og beregn rms-feil



- Blokk-artefakter avtar med økende blokkstørrelse.



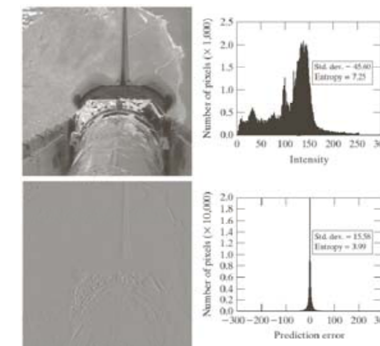
Hva med JPEG 2000?

- For mer info, se http://www.ifi.uio.no/inf386/foiler8.pdf
- Basert på wavelet-transform som gir multiskala-representasjon av bildene. Dette muliggjør gradvis overføring av et bilde (stadig bedre oppløsning, finere skala).
- Regions of interest, områder som er spesielt viktige, kan kodes bedre.
- Høyere kompresjon enn JPEG.
- Mye mindre blokk-effekter.

Tapsfri prediktiv koding

- Istedenfor å kode pikselverdiene f(x,y) kan vi kode
  e(x,y) = f(x,y) - g(x,y) der g er *predikert* fra m naboer.
- En 1-D lineær prediktor av orden m: $g(x,y) = round\left[\sum_{i=1}^{m} \alpha_i f(x-i,y)\right]$
- En første-ordens lineær prediktor:

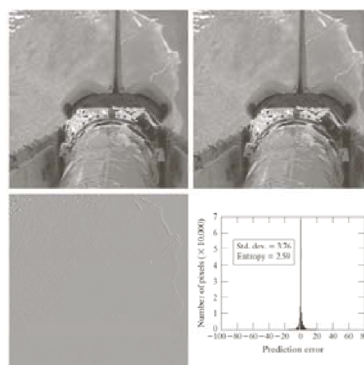
  $$g(x,y) = round\left[\alpha f(x-1,y)\right]$$



- Lik-lengde koding krever ekstra bit
- Bruker entropi-koding
- Max kompresjon er gitt ved:
  - Biter per piksel i original
  - Entropien til prediksjonsfeilen
  - Her blir max C ≈ 2.

Tapsfri koding av bildesekvenser

- Prediksjon kan utvides til tids-sekvenser: $g(x,y,t) = round\left[\sum_{i=1}^{m} \alpha_i f(x,y,t-i)\right]$
- Enklest mulig: første ordens lineær: $g(x,y,t) = round\left[\alpha f(x-1,y,t-1)\right]$
- Differanse-entropien er lav: H=2.59
- Max kompresjon: C ≈ 8/2.59 ≈ 3
- Bevegelses-deteksjon og bevegelses-kompensasjon innenfor 16x16 makro-blokker er nødvendig for å få høy kompresjon.



Digital video

- Koding av digitale bildesekvenser eller digital video
  er basert på differansekoding.
- 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 dette.
- MPEG (Motion Picture Expert Group) standard for videokompresjon.
- MPEG historie:
  - MPEG-1 video og audio (1992)
  - MPEG-2 Digital TV og DVD (1994)
  - MPEG-4: Multimedia anvendelser (1998)
  - MPEG-7: Multimedia søking og filtering (2001)
  - MPEG-21: Multimedia uavhengig av plattform.

Oppsummering - kompresjon

- Hensikten med kompresjon er mer kompakt lagring eller rask oversending av informasjon.
- Kompresjon er basert på informasjonsteori.
- Antall bit. pr. sampel er sentralt, og varierer med kompresjonsmetodene og dataene.
- Sentrale metoder:
  - Før koding: løpelengde-transform og differansetransform.
  - Prediktiv koding: differanse i rom eller tid.
  - Huffman-koding – lag sannsynlighetstabell, send kokebok
  - Universell koding (f.eks. Lempel Ziv) – utnytter mønstre, sender ikke kokebok.
  - Aritmetisk koding – koder sekvenser til tall i et intervall.