

34220 Rapport

Daniel Brasholt, s214675

26. Juni 2022



Indhold

Kapitel 1: Resumé	3
Kapitel 2: Introduktion & problembeskrivelse	4
Kapitel 3: Teori	5
Kapitel 3 del 1: RGB til YCbCr	5
Kapitel 3 del 2: DCT	6
Kapitel 3 del 3: Kvantisering	6
Kapitel 3 del 4: Huffman-kodning	7
Kapitel 3 del 5: Run-length coding	7
Kapitel 4: Implementering	8
Kapitel 5: Tests og resultater	9
Kapitel 5 del 1: Kompression og PSNR	9
Kapitel 5 del 2: Huffman-kodning	10
Kapitel 5 del 3: Run-length kodning	11
Kapitel 6: Konklusion	12
Kapitel 7: Bilag	13
Kapitel 7 del 1: Huffman-tabeller	13
Kapitel 7 del 2: Flere billeder	14
Kapitel 7 del 3: Python-implementering	16
Kapitel 7 del 4: Fremlæggelsesslides	31

Kapitel 1: Resumé

Formålet med projektet var at implementere de væsentlige dele af en JPEG-kompression. Dette inkluderede DC-transformation af 8x8 billedblokke, kvantisering af disse DCT-koefficienter, Huffman-differens-kodning af DC-værdierne og til sidst Run-Length-kodning af AC-koefficienterne. Dette gøres alt sammen på et billede i YCbCr-format, da det teoretisk er muligt at indskrænke mængden af værdier i chrominans-kanalerne uden at det giver en synlig forskel på det færdige billede. Dette er eftersom mennesker er bedre til at se, *om* noget er der, end de er til at se, *hvor* noget er. Processen kaldes Chroma Subsampling. Dette er dog ikke implementeret.

Testne blev udført på et billede, der viser en kat. Dette er eftersom JPEG er specifikt udviklet til at komprimere naturlige fotografier og billeder. Derfor blev signal-støj-forholdet og forskellige kvantiseringer også testet på et fotografi af en skov. Disse er dog blot vist i bilagene.

Det lykkedes i projektet at implementere DCT'en samt Huffman-kodningen af differensen mellem DC-værdierne. Dertil lykkedes det også at implementere en form for run-length kodning blandet med Huffman. Dette betød, at kodelængden på et kvantiseret billede blev skåret ned med 93.6% på luminans-kanalen og 98.2% på hver af de to chrominans-kanaler. Teoretisk ville man kunne forbedre denne komprimering yderligere ved at udføre Chroma Subsampling. Run-length koden viste sig at være meget effektiv, da mange af felterne sidst i en DCT-blok, der gennemløbes i et zig-zag-mønster, er 0 og derfor blot kan kodes som en End-of-Block.

Efter kvantiseringen var udført, viste billedet sig at have et lavt signal-til-støjforhold, PSNR, som forventet. Støjen var værst på luminans-kanalen med $PSNR = 24.7$. De to krominans-kanaler havde støjforhold på henholdsvis 34.7 og 37.8.

Kapitel 2: Introduktion & problembeskrivelse

Formålet med dette projekt var at implementere en JPEG med en DCT-baseret kode. I sig selv vil dette ikke reducere filstørrelsen, hvorfor bitstrømmen bagefter skulle Huffman- og Run Length-kodes. Billederne, denne komprimering blev udført på, var farvebilleder, oprindeligt i RGB-format - dette blev dog ændret til YCbCr for at holde sig mere op ad standarden.

Projektet blev udarbejdet ud fra følgende projektbeskrivelse:

- Implementering af DCT på et billede i RGB-format
- Kvantisering med forskellige faktorer
- Bestemmelse af kodelængde efter Huffman- og Run Length-kodning
- Beregning af PSNR efter kvantisering

Projektet blev implementeret i Python med brug af nogle biblioteker. Numpy tager sig af matrix-operationerne og Scipy anvendes til at udføre DC-transformationen. Implementeringen giver ikke en JPEG-kompatibel bitstrøm, men viser dog principperne, der ligger bag genereringen og komprimeringen af en JPEG-fil.

Kapitel 3: Teori

JPEG bruger flere teknikker til at gøre den egentlige kodelængde så kort som muligt. Disse teknikker vil blive beskrevet mere i dybden i følgende afsnit, men kort fortalt er processen som følgende: først oversættes billedets RGB-værdier til YCbCr - en kanal til luminans og to til krominans. Dernæst inddeltes billedet i blokke af 8x8 pixels. Hver af disse transformeres med en Discrete Cosine Transformation (DCT). Disse værdier kan da kvantiseres, hvilket giver mange koefficenter i blokken af 8x8, som indeholder 0'er. Da den første værdi i blokken (DC-værdien) antages at ændre sig med relativt lave værdier fra blok til blok, vil disse differens-kodes - disse differenser kan da Huffman-kodes, da det antages, at mindre ændringer forekommer mere hyppigt end større. Resten af blokken kan da kodes med en Run-Length kode og en End-of-Block, da det antages, at mange værdier efter kvantiseringen vil være 0. Kun trinnet med at kvantisere DCT-koefficenterne bør give tab af information.

Når disse metoder til komprimering af bitlængde er udført, kan man tjekke, hvorvidt detaljer er gået tabt i billedet ved at kigge på Peak Signal-to-Noise Ratio. Denne er udregnet fra følgende formel:

$$PSNR = 10 \cdot \log_{10} \left(\frac{MAX^2}{MSE} \right)$$

Hvor

$$MSE = \frac{1}{MN} \sum_M \sum_N (X_{M,N} - \tilde{X}_{M,N})^2$$

Kapitel 3 del 1: RGB til YCbCr

Menneskets øje er bedst til at skelne, *om* der er noget i et område af et billede, og mindre godt til at skelne, *hvad* der er i det område. Af den grund oversættes billedets RGB-format til én kanal indeholdende luminans og to indeholdende chrominans. De to chrominans-kanaler kan da nedskaleres, så kun halvdelen eller en fjerdedel af værdierne medtages - en proces ved navn Chroma Subsampling. Dette betyder, at farverne på billedet bliver mindre præcise; men da øjet ikke er lige så godt til at skelne mellem farverne, kan chrominansen nedskaleres meget uden et stort tab i kvalitet. Derimod skal luminansen kodes med fuld oplosning, da man hurtigt ser en halvering i detaljeringen af denne.

Den egentlige transformation mellem formater kan for eksempel ske ved at udføre en matrix-vektor-multiplikation. Anses de oprindelige RGB-værdier som en vektor af formen

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

Kan man, ved at multiplicere denne med en given matrix, opnå de værdier, man gerne vil have indenfor YCbCr-formatet. Microsoft specificerer for eksempel følgende omregningsmatrix¹:

$$\begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} 0.299 & 0.586 & 0.114 \\ -0.169 & -0.331 & 0.5 \\ 0.5 & -0.419 & -0.081 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \quad (1)$$

Bemærk, at den i (1) omtalte matrix er transponeret i forhold til kilden. Dette gør dog bare, at matrix-vektor-multiplikationen skal udføres som vist i (1) i stedet for som i kilden. For at konvertere fra YCbCr til RGB, kan man da blot lave samme matrix-multiplikation, men denne gang med den inverse til transformationsmatricen.

Transformationen givet i (1) giver værdier, hvor Y ligger mellem 0 og 255, mens Cb og Cr ligger mellem -128 og 128. Da det er nemmest at håndtere værdier, der altid er mellem 0 og 255 - da også RGB er indenfor disse grænser - lægges 128 til hver af krominansværdierne, for på den måde at lægge dem indenfor det interval.

Det kan ses i (1), at grønne farver, når det kommer til luminans, tæller mere end rød og blå. Dette skyldes, at menneskets øje er bedst til at se grønne nuancer, hvorfor disse påvirker luminansen mere end de andre.

¹https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-rdprfx/b550d1b5-f7d9-4a0c-9141-b3dca9d7f525?redirectedfrom=MSDN

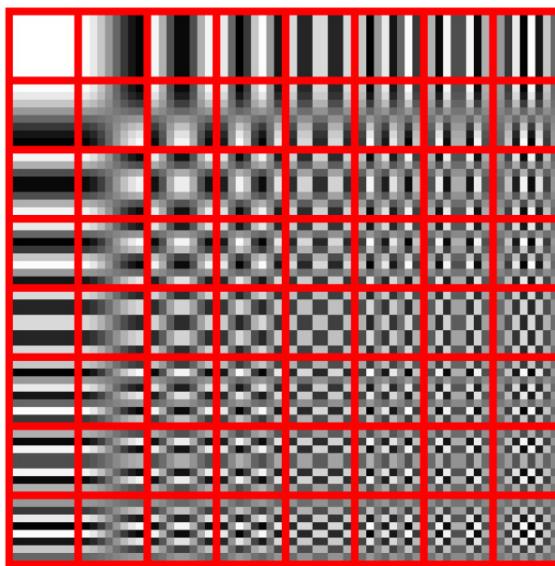
Kapitel 3 del 2: DCT

Efter hver kanal - for eksempel R, G og B eller i dette tilfælde Y, Cb og Cr - er blevet inddelt i blokke af 8x8, udføres en Discrete Cosine Transformation på hver blok. I teorien kan andre blokstørrelser også vælges og give bedre komprimering af billeddata, men i praksis bliver større blokke tungere at udføre DCT for, hvorfor man oftest anvender 8x8. I sig sig hjælper DCT ikke med at komprimere filstørrelsen; dog forholder det sig oftest sådan, at den mest betydningsfulde information ligger i de lave frekvenser, hvorfor de højere frekvenser kan frasorteres. Hvordan de egentlige frekvenser bidrager til et billede er vist på figur 1². Denne transformation bliver udført ud fra følgende matematiske udtryk³:

$$DCT(i, j) = \frac{1}{\sqrt{2N}} C(i) C(j) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} pixel(x, y) \cos\left(\frac{(2x+1)i\pi}{2N}\right) \cos\left(\frac{(2y+1)j\pi}{2N}\right) \quad (2)$$

$$C(x) = \begin{cases} \frac{1}{\sqrt{2}} & \text{hvis } x = 0, \\ \frac{1}{\sqrt{2}} & \text{hvis } x > 0 \end{cases}$$

Udtrykket vist i (2) er formlen for en 2-dimensionel DCT. Dette er netop den brugt i JPEG. Det er dog implementeret anderledes i koden; dette vil blive beskrevet under kapitel 4.



Figur 1: DCT-blok med billedelementerne, som de forskellige frekvenser beskriver. Det kan bemærkes, at den første værdi i blokken blot beskriver gennemsnittet af hele blokken.

Kapitel 3 del 3: Kvantisering

DC-transformationen giver en blok, der er lige så stor og indeholder lige så meget information som den gamle blok af pixels. Dog kan det udnyttes, at strukturen af blokken er beskrevet i øverste venstre hjørne af blokken og detaljerne i nederste højre hjørne. Dette betyder, at man kan fjerne detaljerne ved at lave en uniform kvantisering af DCT-koefficienterne, hvor de nederste koefficienter bliver kvantisert mere end de øverste. Billedet vil naturligvis miste information i dette trin, men da informationen, man mister, hovedsageligt er detaljerne, kan man kvantisere billedet meget før man oplever, at man ikke længere kan gennemskue motivet. Matematisk beskrives kvantiseringen således:

$$\tilde{x}_{i,j} = \left\lfloor \frac{x_{i,j}}{q_{i,j}} \right\rfloor$$

Altså deles hver DCT-koefficient med den tilsvarende koefficient i kvantiseringssmatricen.

Den kvantiseringssmatrix, der oftest anvendes som standard, er vist på figur 2. På figur 3 vises

²Figur taget fra undervisningsslides Billedkodning12DCT_JPEG

³<https://cs.stanford.edu/people/eroberts/courses/soco/projects/data-compression/lossy/jpeg/dct.htm>

et eksempel på en billedblok, der er kvantiseret. Man kan dog anvende enhver kvantiseringsmatrix til at komprimere billedet.

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

Figur 2: Standard kvantiseringsmatrix foreslået af IJG. Kilde: [Billedkodning12DCT_JPEG](#)

Oprindelig billedblok:
<code>[[921 -9 -1 2 0 0 -9 0]</code>
<code>[-59 -2 -9 -3 4 0 0 0]</code>
<code>[0 1 -4 0 0 -1 0 0]</code>
<code>[-17 -9 4 0 0 0 0 0]</code>
<code>[-4 0 -14 0 0 0 0 -14]</code>
<code>[0 13 0 12 0 0 0 17]</code>
<code>[0 0 -15 -17 -21 -24 0 0]</code>
<code>[0 0 0 20 0 19 0 0]]</code>

Kvantiseret billedblok:
<code>[[57 0 0 0 0 0 0 0]</code>
<code>[-4 0 0 0 0 0 0 0]</code>
<code>[0 0 0 0 0 0 0 0]</code>
<code>[-1 0 0 0 0 0 0 0]</code>
<code>[0 0 0 0 0 0 0 0]</code>
<code>[0 0 0 0 0 0 0 0]</code>
<code>[0 0 0 0 0 0 0 0]</code>
<code>[0 0 0 0 0 0 0 0]]</code>

Figur 3: Eksempel på kvantisering af DCT-blok med standard matrix, figur 2

Det kan ses på figur 3, at mange af værdierne er 0 i den kvantiserede billedblok. Dette åbner for muligheden for at lave en run-length kode, der beskriver alle værdierne pånær den første, som kodes med differens og Huffman. Dette kan med fordel gøres i et zig-zag-mønster, da værdierne ofte nærmer sig 0, hvis de er tættere på nederste højre hjørne. Dette vil blive beskrevet nærmere.

Kapitel 3 del 4: Huffman-kodning

Som det kan ses på figur 3, er der i en DCT-blok ofte en værdi, der langt overstiger de andre. Denne afviger dog oftest ikke synderligt meget fra den tilsvarende værdi i den foregående blok. Derfor kan værdien med fordel beskrives som ændringen i forhold til den foregående værdi. Da disse sandsynligvis ligger indenfor et givent område, kan man nu anvende Huffman-kodning til at beskrive det størst betydnende bit. Dette kaldes kategorien af kodeordet. Dernæst kan amplituden beskrives med de resterende bit. Der vil bruges det antal bit, som kategorien er - for eksempel vil en differens på 25 være i kategori 5 og amplituden skal derfor kodes med 5 bit. Er differensen negativ, anvendes 2's komplement til amplituden minus ét i stedet.

For eksempel vil en differens på -5 være i kategori 3 (se figur 4). Kodeordet til denne er 100⁴. Da differensen er negativ, vil amplituden kodes som 2's komplement til $5 + 1 = 6$ med kun 3 bit (da det er kategori 3), som er 010. Det færdige kodeord bliver da 100010. Når dette skal dekodes, vil dekoderen kunne genkende, at amplituden ser ud til at være 2; men da det forventes at være i kategori 3, kan værdien ikke være under 4. Da processen med at konvertere til 2's komplement er reversibel, kan dekoderen genkende værdien som den oprindelige værdi, -5 .

Alt i alt giver Huffman-koden muligheden for at bruge variable bitlængder i stedet for faste, hvilket er gavnligt, da man så kan understøtte større værdier uden at bruge meget længere bitskvenser til at repræsentere de mindre værdier.

Kapitel 3 del 5: Run-length coding

Resten af DCT-matricen kan entropi-kodes ved at markere alle de steder i blokken, hvor værdien er forskellig fra 0. Dette gøres i et zig-zag-mønster, da man efter kvantiseringen ofte har 0'er i nedre højre halvdel. De kodes som en maske frem til End-of-Block. Da der kan være maksimalt 63 værdier, der er forskellige fra 0, må End-of-Block skulle kodes med 6 bit. Masken indeholder et 1-tal, hvis værdien på det pågældende felt er forskelligt fra 0 og 0 ellers. Længden af denne maske vil være End-of-Block minus 1 bit lang. Efter End-of-Block og masken er transmitteret, kan man da begynde at sende værdierne, der skal være på de pladser, som i masken er sat til 1. Disse værdier kan med fordel Huffman-kodes, da disse ofte - ligesom DC-koefficienterne - antager lavere værdier.

⁴tabel vedhæftet som bilag

SSSS	DIFF values
0	0
1	-1,1
2	-3,-2,2,3
3	-7..-4,4..7
4	-15..-8,8..15
5	-31..-16,16..31
6	-63..-32,32..63
7	-127..-64,64..127
8	-255..-128,128..255
9	-511..-256,256..511
10	-1 023..-512,512..1 023
11	-2 047..-1 024,1 024..2 047

Figur 4: Tabel over kategorier for differensen.

Kapitel 4: Implementering

Som nævnt i introduktionen, er implementeringen lavet i Python. Den færdige kode kan findes i bilagene. Mange af funktionerne, der anvendes til at udføre transformationen fra RGB til YCbCr kører meget langsomt, da de er implementeret - for nemheds skyld - med standard for-lykker i Python i stedet for de hurtigere indbyggede funktioner i Numpy. Det tager for eksempel cirka et par minutter at udføre kompressionen af et billede med dimensionerne 2048×1536 . Dette ville naturligvis aldrig være brugbart i den virkelige verden, men til at teste kompression, fungerer dette godt nok.

Det er ikke i programmet implementeret at lave Chroma Subsampling. Begge krominanskanaler bliver kodet i fuld oplosning, og den bedst mulige kompression opnås derfor ikke.

Formlen vist i (2) er for en 2-dimensionel DCT. Denne er implementeret i programmet ved at lave en 1-dimensionel DCT først på hver række i en billedblok og dernæst hver kolonne. Resultatet af dette er det samme som at lave den 2-dimensionelle. Biblioteket Scipy er brugt til at implementere dette, da det har nogle funktioner indbygget, som ville tage længere tid at implementere i hånden, samt at koden da ville køre langsommere.

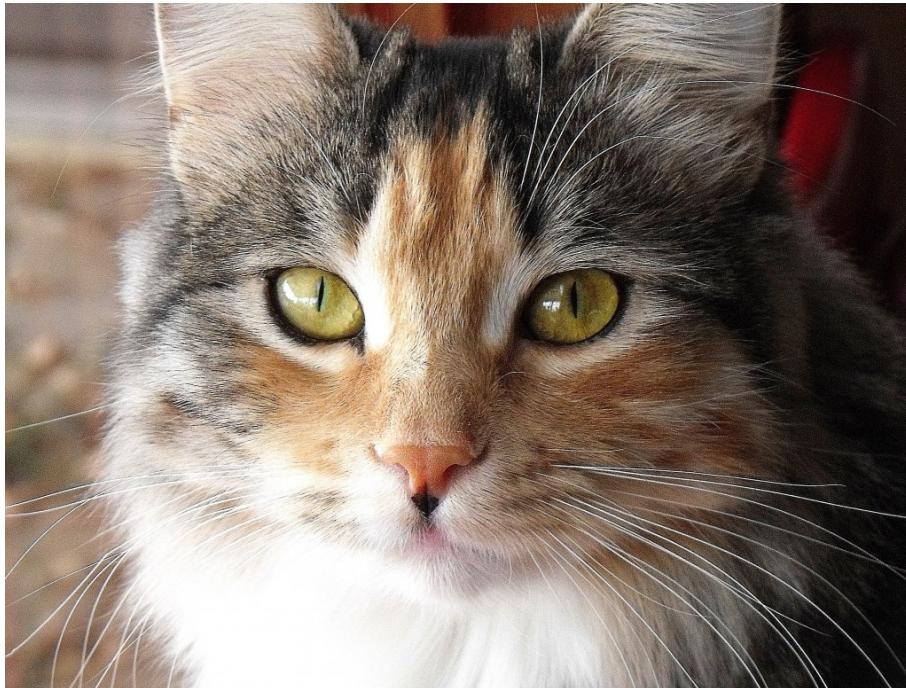
På nuværende tidspunkt kan programmet kun håndtere billeder, hvor antallet af pixels på hver akse er divisibelt med 8 - dette er da inddelingen i DCT-blokke går ud fra, at det kan deles lige op. Dette ville kunne ændres, hvis der blev tilføjet padding til de akser, der manglede, men dette ville nok også betyde, at kanterne de steder ville misfarvet eller vise artefakter. Da det ikke er anset som relevant for projektet, er dette ikke et problem, der er forsøgt løst.

Programmets fulde form kan findes i bilagene. Det er bygget som afsnit, hver med en del af programmets funktion. Man starter med at indlæse de nødvendige biblioteker, dernæst indlæses billedet og til sidst indlæses alle de funktioner, der skal bruges til at udføre transformationen fra RGB til YCbCr, udføre DCT og udføre invers DCT. Den inverse laves, så billedet kan vises. Under afsnittet "Image Manipulation" udføres selve transformationen og kvantiseringen, hvorefter denne laves omvendt, så billedet kan vises. Afsnittet Huffman Coding giver de funktioner, der skal til for at kode DC-værdierne. Afsnittet Run-length giver de funktioner, der skal bruges til at kode værdierne af blokkene. Dette bygger også på det foregående afsnit, da værdierne også her Huffman-kodes. Til sidst kan MSE for billederne udregnes, hvilket også kan anvendes til at udregne PSNR. Under afsnittet Run-length er udregningerne og resultaterne for skov-billedet også vist, selvom dette billede ikke indlæses af koden. Dette er blot for at vise et andet billede og resultaterne for dette.

Kapitel 5: Tests og resultater

Kapitel 5 del 1: Kompression og PSNR

Som eksempel på kompression er følgende billede blevet anvendt:



Figur 5: Test-billede, der viser en kat.

Billedet er valgt, da JPEG netop bør være godt til dagligdags-billeder såsom naturbilleder eller fotografier. Da bør kompressionen af billedet være effektiv uden at give et højt støjforhold. I bilagene findes, sammen med koden, eksempler på billedet uændret - uændret men ført gennem DCT og invers DCT - og til sidst ændret ved at kvantisere DCT-koefficienterne. Kvantiseringen blev i eksemplet udført med en standard JPEG-kvantiseringstabell, som er vist på figur 2, dog multipliceret med 2 for at øge mængden af kompression. Alle efterfølgende udregninger i dette afsnit vil være med dette komprimerede billede. Det kan være svært at se i bilagene på grund af billedets høje oplosning, men billedet mister nogle detaljer af kvantiseringen og kanterne fremstår mere slørede. Det kan ses på figur 7, hvor kattens øje er forstørret. Hele det kvantiserede billede kan ses på figur 6.

For at give en matematisk beskrivelse af denne kompression, kan MSE og PSNR udregnes for det ukomprimerede- og det komprimerede billede i forhold til det oprindelige. Det ukomprimerede billede gav følgende resultater for støj på luminans-kanalen:

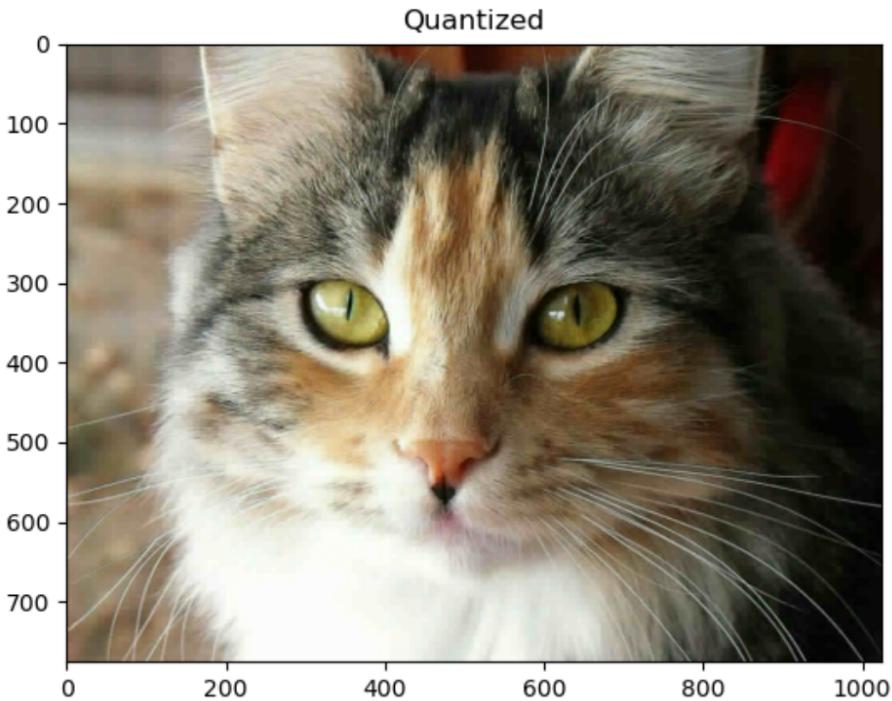
$$MSE \approx 0.597$$

$$PSNR \approx 50.3$$

Chrominans-kanalerne lå også meget tæt op ad disse to værdier⁵. Chrominans-støj er dog heller ikke lige så relevant for billedkvaliteten, da menneskets øje, som beskrevet i teori-afsnittet, bedst kan gennemskue ændringer i luminans frem for farver. Det kan ses, at der er mistet noget information og at støj er introduceret, da middel-kvadrat-fejlen ellers ville være 0, selvom billederne blot har gennemgået en reversibel proces. Dette skyldes nok afrunding og impræcision i floating point operations. Da Signal-to-Noise forholdet er så højt, kan man ikke se dette informationstab.

Det kvantiserede billede havde naturligvis et værre forhold mellem signal og støj. Dette blev fundet til:

⁵De resterende værdier og udregninger kan findes i bilagene under MSE



Figur 6: Test-billedet efter kvantisering med den givne faktor.



Figur 7: Forskel mellem kvantiseret og oprindeligt billede

$$MSE \approx 216.8$$

$$PSNR \approx 24.7$$

Chrominans-støjen klarede sig væsentligt bedre. Denne viste sig som $PSNR \approx 34.7$ og $PSNR \approx 37.8$ på hver af de to kanaler.

Kapitel 5 del 2: Huffman-kodning

Som nævnt i teoriafsnittet, kan DC-koefficienterne kodes med en Huffman-kode. I bilagene kan det ses, hvorledes dette er blevet udført på billederne. For kattebilledet med den givne kvantisering, bragte denne Huffman-kode mængden af bits, der skal til for at kode DC-koefficienterne, ned på 57,127 bits for luminanskanalen samt henholdsvis 37,859 og 37,786 bit for chrominanskanalerne.⁶. Disse tal kan lægges sammen med antallet af bit for resten af blokken, som bliver kodet med en blanding af run-length og Huffman, som beskrevet i næste afsnit.

⁶Det kan også ses i bilagene, hvordan kodelængden ville se ud for det billede, der kun blev transformeret med DCT og ikke blev kvantiseret.

Kapitel 5 del 3: Run-length kodning

Som nævnt i teoriansnittet, kodes AC-koefficienterne med en run-length-kode, der kører i et zig-zag-mønster. Hvert felt, der er forskelligt fra 0, vil blive markeret med et 1-tal i en ”maske”. Længden af denne maske vil være End-of-Block minus 1. Da det sidst mulige felt, der kan indeholde en End-of-Block er det sidste felt i zig-zag-mønsteret, skal denne markeres som et tal fra 0 til 63, altså med 6 bit. Da billedet af katten har dimensionerne 1024×776 pixels, må der i alt være $\frac{1024}{8} \cdot \frac{776}{8} = 12,416$ blokke. Vi kan nu udregne mængden af bits, der skal til for at kode luminans-kanalen:

- Mængden af bits, der skal til for at kode End-of-Block, må være $12,416 \cdot 6 = 74,496$ bit.
- Dernæst kan den samlede længde af maskerne kodes. Dette kan findes i bilagene under ”Run-Length (AC)”. I kattebilledet fås i alt en total maskelængde på 97,529 bit.
- Af de bit, der indeholder i maskerne, er 39,190 af dem 1. Altså er der 39,190 felter, der skal kodes. Koder vi disse bit med samme Huffman-kode, som blev anvendt til at enkode DC-værdierne, går der i alt 176,261 bit til at kode disse. Dette er vist i bilagene.

Alt i alt vil kodelængden af luminans-kanalen da være $74,496 + 97,529 + 176,261 = 348,286$ bit. Hvis blot billedet blev gemt i YCbCr-format med værdier fra 0 til 255 for hver pixel i luminans-kanalen, ville det tage $1024 \cdot 776 \cdot 8 = 6,356,992$ bit. Inkluderer vi bitlængden for DC-koden, bliver den samlede længde af entropikodningen $57,127 + 348,286 = 405,413$ bit. Dette giver da en samlet reduktion efter entropikodning på:

$$1 - \frac{405,413}{6,356,992} \approx 93.6\%$$

Dette er den reduktion, vi forventer i luminanskanalen efter kvantisering og entropikodning. Denne reduktion kan naturligvis også udføres på chrominans-kanalerne. På den første, Cb, fås:

- EOB: $12,416 \cdot 6 = 74,496$ bit
- Maskelængde: 516 bit
- 336 ikke-0-værdier, der kan kodes på 1,410 bit
- DC-værdier: 37,859 bit
- Sammenlagt: $74,496 + 516 + 1,410 + 37,859 = 114,281$ bit
- Dette er en reduktion på $1 - \frac{114,281}{6,356,992} \approx 98,2\%$

For Cr:

- EOB: $12,416 \cdot 6 = 74,496$ bit
- Maskelængde: 468 bit
- 312 ikke-0-værdier, der kan kodes på 1,276 bit
- DC-værdier: 37,786 bit
- Sammenlagt: $74,496 + 468 + 1,276 + 37,786 = 114,026$ bit
- Dette er en reduktion på $1 - \frac{114,026}{6,356,992} \approx 98.2\%$

Denne kompression ville endda blive mere effektiv, hvis chroma subsampling var blevet udført. Resultatet er overraskende, da det har været muligt at komprimere datamængden så meget. Det betyder, at langt de fleste DCT-blokke i chrominans-kanalerne ikke har indeholdt andet end en DC-koefficient. Udføres samme proces på billedet af skoven vist i bilagene, fås for Cr cirka samme resultat og mængde af kompression, men Cb-kanalen kan kun komprimeres til 28,533 ikke-0-bit⁷. Netop med billedet af en skov giver det mening, at den ene krominans-kanal indeholder mere information end den anden, da skoven hovedsageligt er grøn.

⁷Det er værd at notere, at billedet af skoven har en væsentligt højere oplosning

Kapitel 6: Konklusion

Programmet kan lave en Discrete Cosine Transformation og kvantisere resultatet for på den måde at gøre datamængden mindre. Processen er også til dels reversibel, så man kan vise billedet bagefter - dog går noget information naturligvis tabt under kvantiseringen. Da mange af funktionerne, specielt til at konvertere til og fra YCbCr, er bygget i hånden, tager det lang tid at komprimere et billede. Det kan tage op til et par minutter med et billede i høj opløsning.

Efter kvantiseringen lykkedes det at differens-kode DC-koefficienterne i alle blokkene. Disse kunne dernæst Huffman-kodes for at skære ned på bitlængden. Ligeså kunne AC-koefficienterne Run-Length samt Huffman-kodes for at skære markant ned på mængden af informationsbit. Billedet af katten kunne komprimeres med omkring 94% på luminanskanalen. Chrominans-kanalerne kunne hver komprimeres med omkring 98%. Disse procenter blev udregnet ved at gå ud fra, at der ellers skulle bruges 8 bit per kanal til at kode billedinformationen.

Kvantiseringen gjorde signal-støj-forholdet væsentligt værre. Den vigtigste kanal, luminanskanalen, fik en Peak Signal-to-Noise Ratio på 24.7. De to chrominans-kanaler havde støj på henholdsvis 34.7 og 37.8. Denne støj kunne også ses, hvis man forstørrede en del af billedet, hvor man ville forvente detaljer, for eksempel kattens øje. Luminans-kanalen er vigtigere end chrominans-kanalerne, da menneskets øje er bedst til at skelne mellem *om* der er noget i en del af et billede og mindre godt til at skelne mellem farverne. Dette støjforhold samt kodelængderne blev udregnet på et billede kvantiseret med den standard-matrix, som er udarbejdet af IJG og vist på undervisningsslides, multipliceret med 2.

Kapitel 7: Bilag

Kapitel 7 del 1: Huffman-tabeller

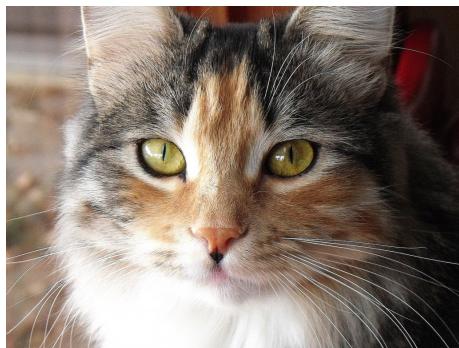
Table K.3 – Table for luminance DC coefficient differences

Category	Code length	Code word
0	2	00
1	3	010
2	3	011
3	3	100
4	3	101
5	3	110
6	4	1110
7	5	11110
8	6	111110
9	7	1111110
10	8	11111110
11	9	111111110

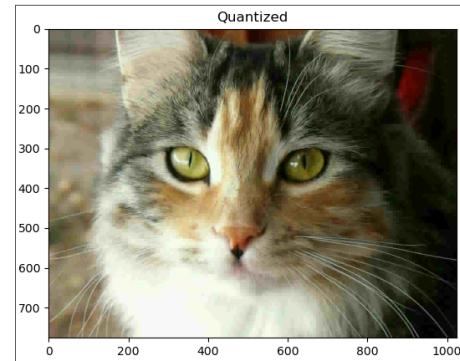
Table K.4 – Table for chrominance DC coefficient differences

Category	Code length	Code word
0	2	00
1	2	01
2	2	10
3	3	110
4	4	1110
5	5	11110
6	6	111110
7	7	1111110
8	8	11111110
9	9	111111110
10	10	1111111110
11	11	11111111110

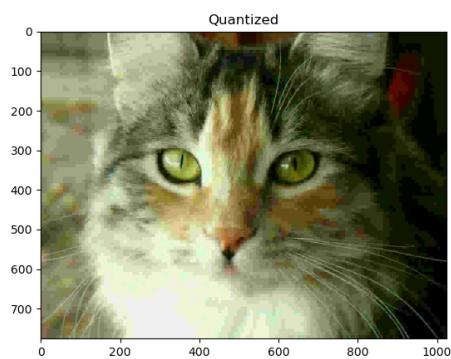
Kapitel 7 del 2: Flere billeder



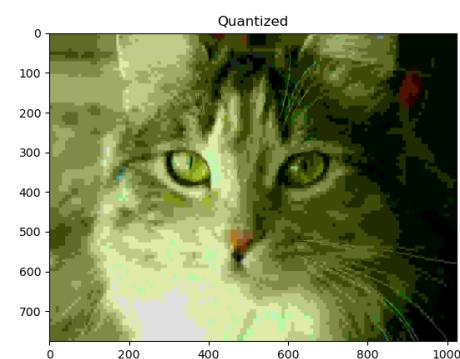
(a) Originalt billede



(b) Komprimeret med standard kuantiseringssmatrix multipliceret med 4. PSNR = 23.1



(c) Komprimeret med standard kuantiseringssmatrix multipliceret med 8. PSNR = 21.4

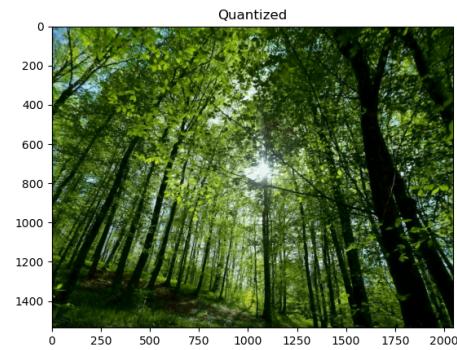


(d) Komprimeret med standard kuantiseringssmatrix multipliceret med 16. PSNR = 18.95

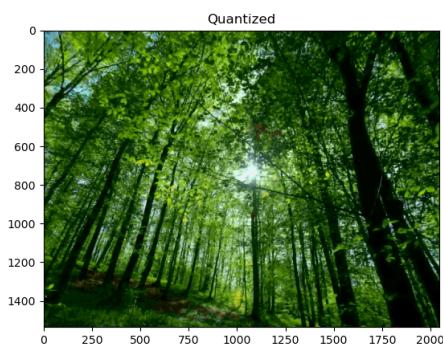
Figur 9: Kat kuantiseret med forskellige faktorer



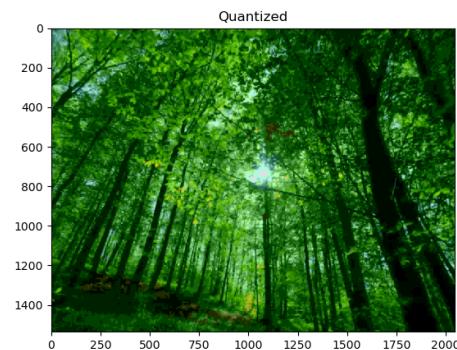
(a) Originalt billede



(b) Komprimeret med standard kuantiseringsmatrix multipliceret med 4. PSNR = 22.7



(c) Komprimeret med standard kuantiseringsmatrix multipliceret med 8. PSNR = 20.1



(d) Komprimeret med standard kuantiseringsmatrix multipliceret med 16. PSNR = 17.5

Figur 10: Skov kuantiseret med forskellige faktorer

Kapitel 7 del 3: Python-implementering

JPEG-color

June 26, 2022

0.1 Necessary

Import of necessary packages as well as definition of quantization matrix.

```
[44]: %matplotlib widget
import numpy as np
from scipy.fftpack import dct, idct
from matplotlib import pyplot as plt
from PIL import Image
import math
```

```
[45]: Q = np.matrix([
    [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]
])
Q = np.multiply(Q, 2)
```

0.2 Image import

Import of image. The image used is the same as in the report.

```
[46]: img = Image.open('img/cat.jpg')
img.show()
img_arr = np.array(img)
imgsize = img_arr.shape
```

0.3 Functions

Definition of all necessary functions.

```
[47]: # Functions for calculating DCT and inverse DCT on an image block
def dct2(mat):
    return dct( dct(mat, axis=0, norm='ortho'), axis=1, norm='ortho')
```

```
def inverse_dct2(mat):
    return idct( idct(mat, axis=0, norm='ortho'), axis=1, norm='ortho')
```

[48]: # Function for retrieving the color channels from a complete image. Although ↴ the variables
specify R, G, and B, the function works just as well on YCbCr

```
def color_to_greys(img_matrix):
    matrix_size = img_matrix.shape
    red, green, blue = (
        np.zeros(matrix_size[:-1], dtype='uint8'),
        np.zeros(matrix_size[:-1], dtype='uint8'),
        np.zeros(matrix_size[:-1], dtype='uint8')
    )

    for i in range(matrix_size[0]):
        for j in range(matrix_size[1]):
            red[i,j], green[i,j], blue[i,j] = (
                img_matrix[i, j][0],
                img_matrix[i, j][1],
                img_matrix[i, j][2]
            )
    return red, green, blue
```

[49]: # Function for getting a DCT matrix from a greyscale image matrix

```
def get_dct(img_matrix):
    matrix_size = img_matrix.shape
    dct_matrix = np.zeros(matrix_size, dtype='float16')
    quantized_dct = np.zeros(matrix_size, dtype='float16')

    for i in np.r_[:matrix_size[0]:8]:
        for j in np.r_[:matrix_size[1]:8]:
            dct_matrix[i:(i+8), j:(j+8)] = dct2( img_matrix[i:(i+8), j:(j+8)] )
            quantized_dct[i:(i+8), j:(j+8)] = np.divide(dct_matrix[i:(i+8), j:(j+8)], Q)
    return dct_matrix.astype('int'), quantized_dct.astype('int')
```

[50]: # Inverse of above function. Also casts the values to be between 0 and 254

```
def get_inverse_dct(img_matrix, img_matrix_quantized):
    matrix_size = img_matrix.shape
    inv_dct = np.zeros(matrix_size, dtype='uint8')
    inv_q_dct = np.zeros(matrix_size, dtype='uint8')

    for i in np.r_[:imgsize[0]:8]:
        for j in np.r_[:imgsize[1]:8]:
            image_block = inverse_dct2(np.multiply(img_matrix_quantized[i:(i+8), j:(j+8)], Q))
```

```

    inv_q_dct[i:(i+8), j:(j+8)] = np.clip(image_block, 0, 254)
    image_block = inverse_dct2(img_matrix[i:(i+8), j:(j+8)])
    inv_dct[i:(i+8), j:(j+8)] = np.clip(image_block, 0, 254)
return np.rint(inv_dct), np.rint(inv_q_dct)

```

[51]: # Function for collecting greyscale images to a complete image

```

def collect_greys(red_matrix, green_matrix, blue_matrix):
    imgsize = red_matrix.shape
    collected_matrix = np.zeros( (imgsize[0], imgsize[1], 3), dtype='uint8')
    for i in range(imgsize[0]):
        for j in range(imgsize[1]):
            collected_matrix[i, j, 0] = red_matrix[i, j]
            collected_matrix[i, j, 1] = green_matrix[i, j]
            collected_matrix[i, j, 2] = blue_matrix[i, j]
    return collected_matrix

```

[52]: # Function for converting an image from RGB to YCBCR

```

def transform_to_yuv(img_matrix):
    imgsize = img_matrix.shape
    luminance_transformation = np.matrix([
        [0.299, 0.586, 0.114],
        [-0.169, -0.331, 0.500],
        [0.500, -0.419, -0.081]
    ])
    added_values = np.array([0, 128, 128])

    img_yuv = np.zeros(imgsize)

    for i in range(imgsize[0]):
        for j in range(imgsize[1]):
            img_yuv[i, j] = np.matmul(luminance_transformation, img_matrix[i, j]) + added_values
    return img_yuv

```

[53]: # Function for converting YCBCR image to RGB

```

def transform_to_rgb(img_matrix):
    imgsize = img_matrix.shape
    luminance_transformation = np.matrix([
        [0.299, 0.586, 0.114],
        [-0.169, -0.331, 0.500],
        [0.500, -0.419, -0.081]
    ])
    added_values = np.array([0, 128, 128])
    rgb_transformation = np.linalg.inv(luminance_transformation)
    img_rgb = np.zeros(imgsize, dtype='uint8')
    for i in range(imgsize[0]):
        for j in range(imgsize[1]):

```

```

    block = np.matmul(rgb_transformation, img_matrix[i, j] - ↴
→added_values)
    img_rgb[i, j] = np.clip(block, 0, 254)
return img_rgb

```

0.4 Image manipulation

Code for actually compressing an image to DCT coefficients as well as converting it back to RGB so it can be viewed.

[54]: # Convert RGB to YCBCR

```
yuv_matrix = transform_to_yuv(img_arr)
y, u, v = color_to_greys(yuv_matrix)
```

[55]: # Perform DCT & quantization

```
y_dct, y_quantized = get_dct(y)
u_dct, u_quantized = get_dct(u)
v_dct, v_quantized = get_dct(v)
```

[56]: # Perform inverse DCT - every step from now on is done to render the compressed ↴
→image

```
y_inv, y_q_inv = get_inverse_dct(y_dct, y_quantized)
u_inv, u_q_inv = get_inverse_dct(u_dct, u_quantized)
v_inv, v_q_inv = get_inverse_dct(v_dct, v_quantized)
```

[57]: # Collect channels to one matrix

```
collected = collect_greys(y_inv, u_inv, v_inv)
collected_quantized = collect_greys(y_q_inv, u_q_inv, v_q_inv)
```

[58]: # Convert collected channels into RGB

```
collected_rgb = transform_to_rgb(collected)
collected_rgb_quantized = transform_to_rgb(collected_quantized)
original_to_rgb = transform_to_rgb(yuv_matrix)
```

[59]: # Plot the original image with the image which has only gone through DCT and ↴
→inverse DCT

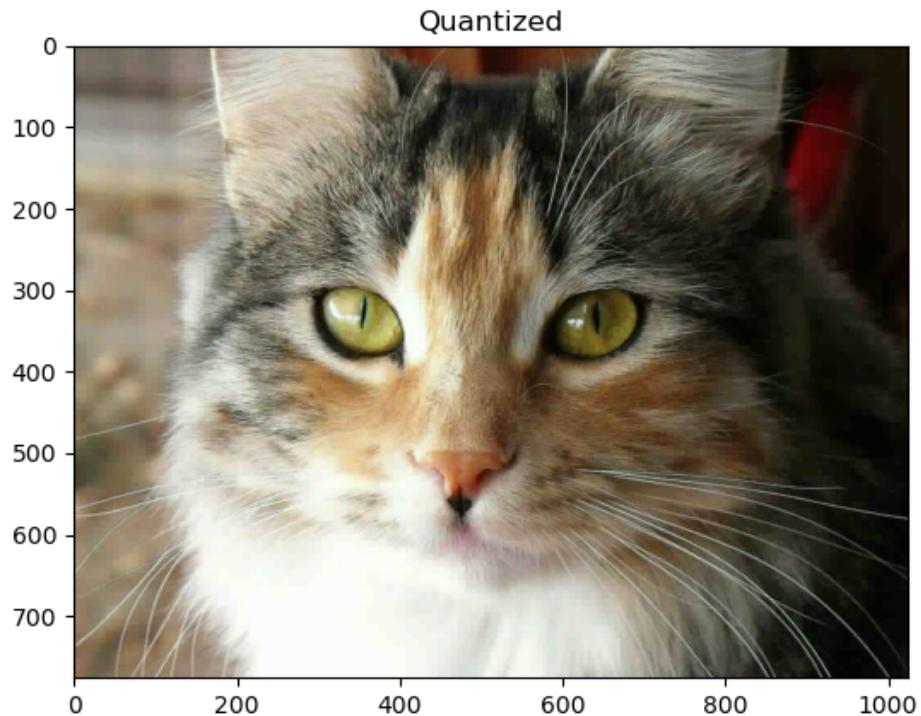
```
# as well as the quantized image
plt.figure()
plt.imshow(collected_rgb_quantized, cmap='gray')
plt.title("Quantized")
```

```
plt.figure()
plt.imshow(collected_rgb, cmap='gray')
plt.title('Inverse DCT')
```

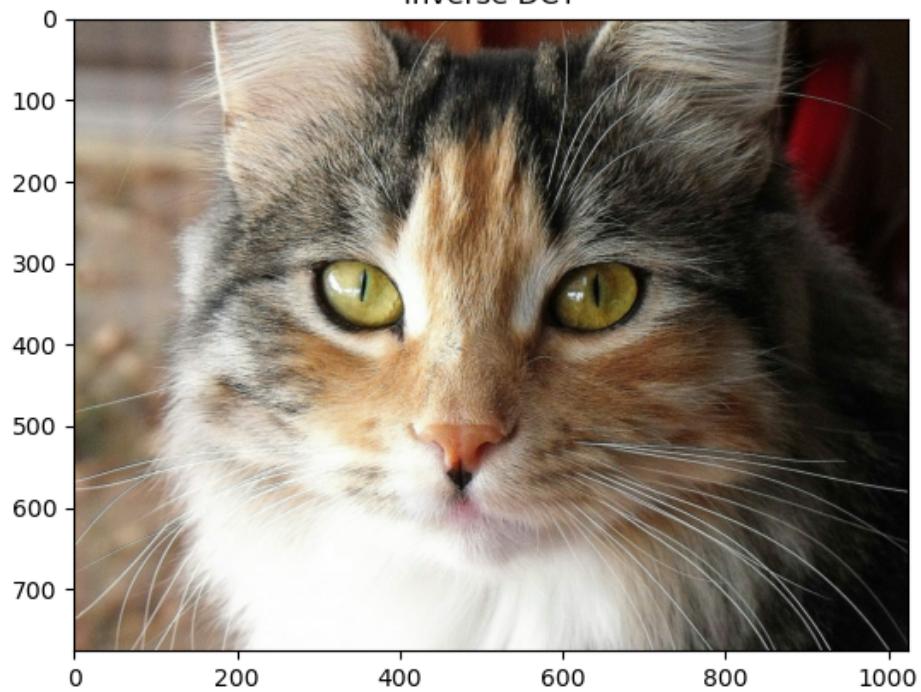
```
plt.figure()
plt.imshow(original_to_rgb, cmap='gray')
```

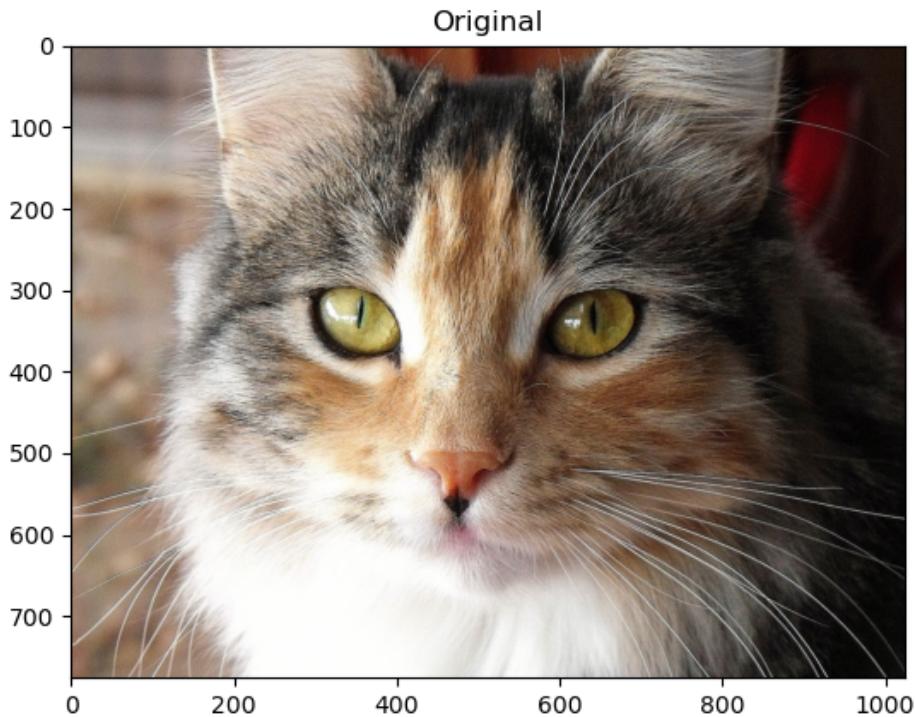
```
plt.title('Original')
```

```
[59]: Text(0.5, 1.0, 'Original')
```



Inverse DCT





```
[60]: pos = 256
print('Oprindelig billedblok:')
print(y_dct[pos:pos+8, pos:pos+8])
print('-----\nKvantiseret billedblok:')
print(u_quantized[pos:pos+8, pos:pos+8])
```

Oprindelig billedblok:

```
[[ 864 -16   35   -9    0   -8    0   11]
 [   3 -26    5   -3   40    0  -12  -11]
 [ 15 -91 -23   10    0   10   13  -11]
 [ 105 -11 -67    5    9  -17   16  -11]
 [  -4 -31   -6   21   27    0    0  -15]
 [ -81   97 -66    0    0    1  -22   18]
 [ -99  103   63   17    0    0    0  -20]
 [  83 -107 -133    0    0    0    0    0]]
```

Kvantiseret billedblok:

```
[[30  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]
[ 0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0]]
```

0.5 Huffman Coding (DC)

```
[61]: def difference_dc_values(dct_matrix):
    imgsize = dct_matrix.shape
    dc_values = []
    for i in np.r_[:imgsize[0]:8]:
        for j in np.r_[:imgsize[1]:8]:
            dc_values.append(dct_matrix[i,j])
    difference_dc_values = [val - dc_values[i] for i,val in enumerate(dc_values[1:])]
    difference_dc_values = [dc_values[0], *difference_dc_values]
    return difference_dc_values
```



```
[62]: def get_diff_category(diff):
    for i in range(12):
        if -(2**i)+1 <= diff <= 2**i-1: return i
```



```
[63]: def twos_complement(value, category):
    return 2**category - value
```



```
[64]: def encode_dc_values(diff_array, category_codes):
    encoded = []
    for diff in diff_array:
        cat = get_diff_category(diff)
        if cat is None:
            print(diff)
        if diff < 0:
            bit_value = bin(twos_complement(-diff+1, cat))
            encoded.append( category_codes[cat] + bit_value[2:].rjust(cat, '0'))
        else:
            bit_value = bin(diff)
            encoded.append( category_codes[cat] + bit_value[2:].rjust(cat, '0'))
    return encoded
```



```
[65]: k3 = {0: '00', 1: '010', 2: '011', 3: '100', 4: '101', 5: '110', 6: '1110', 7:'11110', 8: '111110', 9: '1111110', 10: '11111110', 11: '111111110'}
k4 = {0: '00', 1: '01', 2: '10', 3: '110', 4: '1110', 5: '11110', 6: '111110', 7: '1111110', 8: '11111110', 9: '111111110', 10: '1111111110', 11: '11111111110'}
```

```
[66]: # Difference length of non-quantized luminance
diffs = difference_dc_values(y_dct.astype('int'))
codes = encode_dc_values(diffs,k3)
codelens = [len(i) for i in codes]
print(sum(codelens))
```

125408

```
[67]: # Difference length of non-quantized chrominance
diffs = difference_dc_values(u_dct.astype('int'))
codes = encode_dc_values(diffs,k3)
codelens = [len(i) for i in codes]
print(sum(codelens))
```

76223

```
[68]: # Difference length of non-quantized chrominance
diffs = difference_dc_values(v_dct.astype('int'))
codes = encode_dc_values(diffs,k3)
codelens = [len(i) for i in codes]
print(sum(codelens))
```

75752

```
[69]: # Difference length of quantized luminance channel
diffs = difference_dc_values(y_quantized.astype('int'))
codes = encode_dc_values(diffs,k3)
codelens = [len(i) for i in codes]
print(sum(codelens))
```

57127

```
[70]: # Difference length of quantized one chrominance channel
diffs = difference_dc_values(u_quantized.astype('int'))
codes = encode_dc_values(diffs,k4)
codelens = [len(i) for i in codes]
print(sum(codelens))
```

37859

```
[71]: # Difference length of quantized other chrominance channel
diffs = difference_dc_values(v_quantized.astype('int'))
codes = encode_dc_values(diffs,k4)
codelens = [len(i) for i in codes]
print(sum(codelens))
```

37786

0.6 Run-length (AC)

Since the algorithm for running through in a zig-zag-pattern was difficult to generate, the indeces of the fields are hard-coded. This is not a problem as long as the image blocks are always 8x8

```
[73]: # Returns number of fields in matrix containing non-zero as well as index of EOF
def get_non_zero(image_block):
    non_zero = []
    EOF = 0
    for block_index, (i,j) in enumerate(zip(x_indeces, y_indeces)):
        if image_block[i,j] != 0:
            non_zero.append(image_block[i,j])
            EOF = block_index + 1
    return EOF, non_zero
```

```
[74]: # Calculates number of fields not equal to zero as well as total bits needed for
# mask in given channel (luminance)
# - This is for the cat image
total_non_zero = []
total_mask = 0
for i in np.r_[:imgsize[0]:8]:
    for j in np.r_[:imgsize[1]:8]:
        EOB, non_zero = get_non_zero(y_quantized[i:(i+8), j:(j+8)])
        total_non_zero += non_zero
        total_mask += EOB
print(f'{total_mask=}, {len(total_non_zero)=}')
# Huffman-encoding of bit values that are non-zero
# uses same encoding as shown above
codes = encode_dc_values(total_non_zero,k3)
codelens = [len(i) for i in codes]
i = 0
```

total_mask=97529, len(total_non_zero)=39190
sum(codeleng)=176261

```
[75]: # Now for first chrominance channel:  
# - This is for the cat image  
total_non_zero = []  
total_mask = 0  
for i in np.r_[::imgsize[0]:8]:  
    for j in np.r_[::imgsize[1]:8]:
```

```

        EOB, non_zero = get_non_zero(u_quantized[i:(i+8), j:(j+8)])
        total_non_zero += non_zero
        total_mask += EOB
print(f'{total_mask=}, {len(total_non_zero)=}')
# Huffman-encoding of bit values that are non-zero
# uses same encoding as shown above
codes = encode_dc_values(total_non_zero,k3)
codelens = [len(i) for i in codes]
print(f'{sum(codelens)=}')

```

total_mask=516, len(total_non_zero)=336
sum(codelens)=1410

[76]: # Second chrominance channel:
- This is for the cat image
total_non_zero = []
total_mask = 0
for i in np.r_[:imgsize[0]:8]:
 for j in np.r_[:imgsize[1]:8]:
 EOB, non_zero = get_non_zero(v_quantized[i:(i+8), j:(j+8)])
 total_non_zero += non_zero
 total_mask += EOB
print(f'{total_mask=}, {len(total_non_zero)=}')
Huffman-encoding of bit values that are non-zero
uses same encoding as shown above
codes = encode_dc_values(total_non_zero,k3)
codelens = [len(i) for i in codes]
print(f'{sum(codelens)=}')

total_mask=468, len(total_non_zero)=312
sum(codelens)=1276

[29]: # Calculates number of fields not equal to zero as well as total bits needed for
mask in given channel (luminance)
- This is for the forest image
total_non_zero = []
total_mask = 0
for i in np.r_[:imgsize[0]:8]:
 for j in np.r_[:imgsize[1]:8]:
 EOB, non_zero = get_non_zero(y_quantized[i:(i+8), j:(j+8)])
 total_non_zero += non_zero
 total_mask += EOB
 # Note: 1 is not subtracted since the enumerate function used starts ↵
from 0
print(f'{total_mask=}, {len(total_non_zero)=}')
Huffman-encoding of bit values that are non-zero
uses same encoding as shown above

```

codes = encode_dc_values(total_non_zero,k3)
codelens = [len(i) for i in codes]
print(f'{sum(codelens)=}')

```

total_mask=567625, len(total_non_zero)=321310
sum(codelens)=1532723

```
[30]: # Now for first chrominance channel:
# - This is for the forest image
total_non_zero = []
total_mask = 0
for i in np.r_[:imgsize[0]:8]:
    for j in np.r_[:imgsize[1]:8]:
        EOB, non_zero = get_non_zero(u_quantized[i:(i+8), j:(j+8)])
        total_non_zero += non_zero
        total_mask += EOB
    # Note: 1 is not subtracted since the enumerate function used starts
    # from 0
print(f'{total_mask=}, {len(total_non_zero)=}')
# Huffman-encoding of bit values that are non-zero
codes = encode_dc_values(total_non_zero,k3)
codelens = [len(i) for i in codes]
print(f'{sum(codelens)=}')

```

total_mask=40194, len(total_non_zero)=28533
sum(codelens)=122234

```
[31]: # Second chrominance channel:
# - This is for the forest image
total_non_zero = []
total_mask = 0
for i in np.r_[:imgsize[0]:8]:
    for j in np.r_[:imgsize[1]:8]:
        EOB, non_zero = get_non_zero(v_quantized[i:(i+8), j:(j+8)])
        total_non_zero += non_zero
        total_mask += EOB
    # Note: 1 is not subtracted since the enumerate function used starts
    # from 0
print(f'{total_mask=}, {len(total_non_zero)=}')
# Huffman-encoding of bit values that are non-zero
codes = encode_dc_values(total_non_zero,k3)
codelens = [len(i) for i in codes]
print(f'{sum(codelens)=}')

```

total_mask=294, len(total_non_zero)=229
sum(codelens)=916

0.7 MSE

Functions for calculating MSE and PSNR

```
[77]: # Calculate MSE with formula given in report
def MSE(original, changed):
    imgsize = original.shape
    MN = imgsize[0] * imgsize[1]
    diff = original - changed
    diff = np.multiply(diff, diff)
    diff = diff / MN
    summed = diff.sum()
    return summed
```

```
[78]: # Luminance noise for DCT/IDCT image
ms = MSE(y, y_inv)
PSNR = 10*math.log10((np.max(y)**2) / ms)
print(ms, PSNR)
```

0.5968873328769331 50.337750708334134

```
[79]: # Chrominance noise for DCT/IDCT image
ms = MSE(u, u_inv)
PSNR = 10*math.log10((np.max(u)**2) / ms)
print(ms, PSNR)
```

0.4637274987918816 45.81440431844983

```
[80]: # Chrominance noise for DCT/IDCT image
ms = MSE(v, v_inv)
PSNR = 10*math.log10((np.max(v)**2) / ms)
print(ms, PSNR)
```

0.41812228173324745 49.08332334302304

```
[81]: # Luminance noise for quantized image
ms = MSE(y, y_q_inv)
PSNR = 10*math.log10((np.max(y)**2) / ms)
print(ms, PSNR)
```

216.8163294841333 24.7357544532839

```
[82]: # Chrominance noise for quantized image
ms = MSE(u, u_q_inv)
PSNR = 10*math.log10((np.max(u)**2) / ms)
print(ms, PSNR)
```

5.976173636839561 34.71280074581689

```
[83]: # Chrominance noise for quantized image
ms = MSE(v, v_q_inv)
PSNR = 10*math.log10((np.max(v)**2) / ms)
print(ms, PSNR)
```

5.607109777706184 37.80896587087455

Kapitel 7 del 4: Fremlæggelsesslides

JPEG-kodning

Daniel Brasholt s214675

Formål og beskrivelse

- DCT
- Kvantisering
- Huffman
- PSNR

(JPEG)

Transform
(DCT)

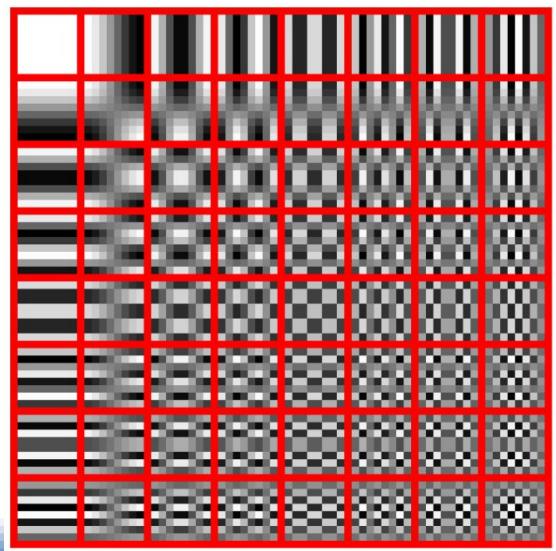
Quantization
(Uniform)

Entropy
coding
(Huffman)

DCT – kort fortalt

- Struktur og detaljer
- 8x8 blokke

788	45	44	10	-3	-1	-7	-1
219	29	43	28	3	-1	5	0
11	59	46	4	-2	0	2	2
38	26	19	-25	-22	-6	-3	2
9	3	-16	-6	-19	-9	3	2
-8	6	-10	-14	-6	-5	0	1
-2	-7	2	-9	-6	1	4	5
7	-5	-3	-1	6	1	8	3



Kvantisering

- Forskellige muligheder
- Fjerner detaljer

$$\tilde{x} = \left\lfloor \frac{x}{q} \right\rfloor$$

```
Q = np.matrix([
    [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]
])
```

25	1	1	0	0	0	0	0
7	1	1	1	0	0	0	0
0	2	1	0	0	0	0	0
1	1	1	0	-1	0	0	0
0	0	-1	0	-1	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

Huffman

- Differens fra blok til blok
- Kategori
- 2's komplement
- Fx -5

SSSS	DIFF values
0	0
1	-1,1
2	-3,-2,2,3
3	-7,-4,4..7
4	-15..-8,8..15
5	-31..-16,16..31
6	-63..-32,32..63
7	-127..-64,64..127
8	-255..-128,128..255
9	-511..-256,256..511
10	-1 023..-512,512..1 023
11	-2 047..-1 024,1 024..2 047

MSE & PSNR

$$PSNR = 10 \cdot \log_{10} \left(\frac{MAX^2}{MSE} \right)$$

$$MSE = \frac{1}{MN} \sum_M \sum_N (X_{M,N} - \tilde{X}_{M,N})^2$$

Implementering

- Python
- Numpy
- Scipy

DCT & kvantisering

```
def get_dct(img_matrix):
    matrix_size = img_matrix.shape
    dct_matrix = np.zeros(matrix_size)
    quantized_dct = np.zeros(matrix_size)

    for i in np.r_[:matrix_size[0]:8]:
        for j in np.r_[:matrix_size[1]:8]:
            dct_matrix[i:(i+8), j:(j+8)] = dct2( img_matrix[i:(i+8), j:(j+8)] )
            quantized_dct[i:(i+8), j:(j+8)] = np.divide(dct_matrix[i:(i+8), j:(j+8)], Q)
    return np.rint(dct_matrix), np.rint(quantized_dct)
```

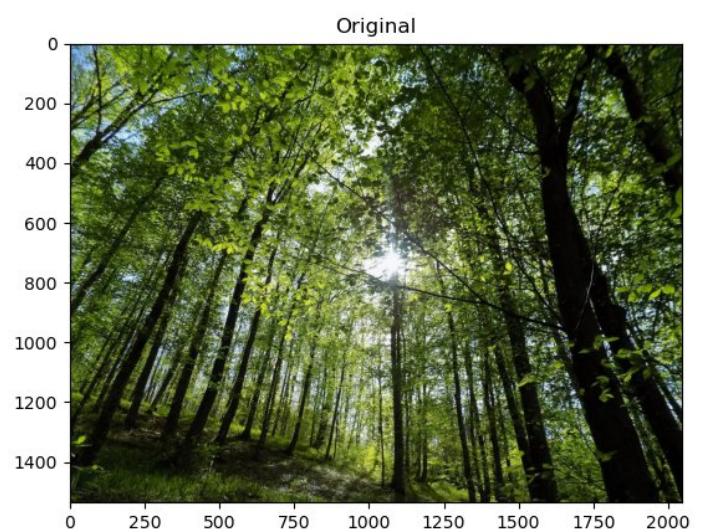
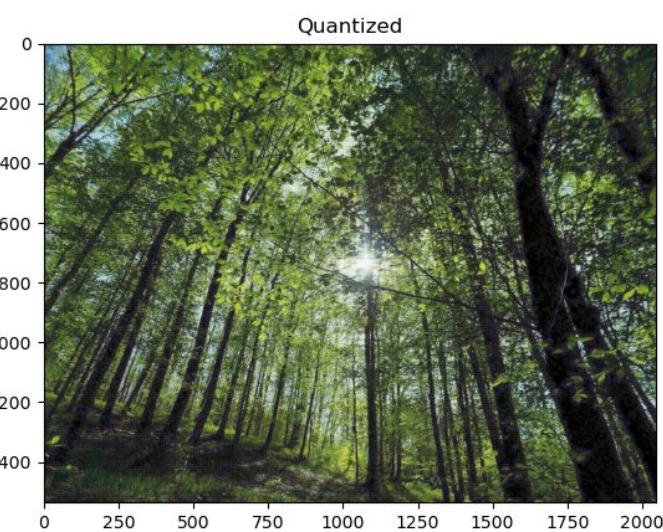


Table K.3 – Table for luminance DC coefficient differences

Category	Code length	Code word	
0	2	00	
1	3	010	
2	3	011	
3	3	100	
4	3	101	
5	3	110	
6	4	1110	
7	5	11110	
8	6	111110	
9	7	1111110	
10	8	11111110	
11	9	111111110	

Huffman

```
diffs = difference_dc_values(y_quantized.astype('int'))
codes = encode_dc_values(diffs,k3)
codelens = [len(i) for i in codes]
print(sum(codelens))
print(len(diffs)*11)

367057
540672
```

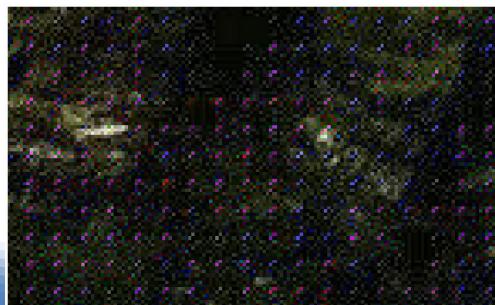
```
def encode_dc_values(diff_array, category_codes):
    encoded = []
    for diff in diff_array:
        cat = get_diff_category(diff)
        if diff < 0:
            bit_value = bin(twos_complement(-diff+1, cat))
            encoded.append( category_codes[cat] + bit_value[2:].rjust(cat, '0') )
        else:
            bit_value = bin(diff)
            encoded.append( category_codes[cat] + bit_value[2:].rjust(cat, '0') )
    return encoded
```

```
diffs = difference_dc_values(v_quantized.astype('int'))
codes = encode_dc_values(diffs,k4)
codelens = [len(i) for i in codes]
print(sum(codelens))
print(len(diffs)*11)

160962
540672
```

MSE & PSNR

```
def MSE(original, changed):
    imgsize = original.shape
    MN = imgsize[0] * imgsize[1]
    diffs = original - changed
    diffs = np.multiply(diffs, diffs)
    diffs = diffs / MN
    summed = diffs.sum()
    return summed
```



```
ms = MSE(y, y_inv)
PSNR = 10*math.log10((np.max(y)**2) / ms)
print(ms, PSNR)

0.4859628677368166 51.23064347008631
```

```
ms = MSE(y, y_q_inv)
PSNR = 10*math.log10((np.max(y)**2) / ms)
print(ms, PSNR)

1508.6540505091357 16.31077770124197
```