

Het JPEG compressie algoritme, IS 10918-1

Een overzicht van het JPEG compressie algoritme door Mathias Verboven.

Inhoudsopgave

Inleiding.....	2
Stap 1: inlezen bronbestand.....	3
Stap 2: Veranderen van kleurruimte.....	3
Step 2.1: Veranderen van kleurruimte.....	3
Step 2.2: Sub sampling (van de chrominantie waarden).....	4
Stap 3: Opdelen in macroblokken.....	5
Stap 4: De discrete cosinus transformatie en zijn inverse (2D).....	6
Stap 5: Quantisatie van de DCT coëfficiënten.....	7
Stap 6: Codering.....	8
Stap 6.1: DC codering.....	9
Stap 6.2: Zigzag sequentie.....	9
Stap 6.3: Entropie codering.....	10
Stap 6.3.1: Tussenstap codering.....	10
Stap 6.3.2: Variabele lengte codering (Huffman).....	12
Stap 6.3.2.1: Het VLI (Variable Length Integer) woordenboek.....	12
Stap 6.3.2.2: Het VLC (Variable Length Coding) woordenboek.....	13
Voorbeeld: Van een macroblok naar een bit-stroom.....	15

De voornaamste bronnen waren:

- <http://www.ams.org/samplings/feature-column/fcarc-image-compression>
- "The JPEG Still Picture Compression Standard", Wallace, G.K.
- <http://en.wikipedia.org/wiki/JPEG>
- <http://www.impulseadventure.com/photo/>
- http://www.fileformat.info/mirror/egff/ch09_06.htm

Inleiding.

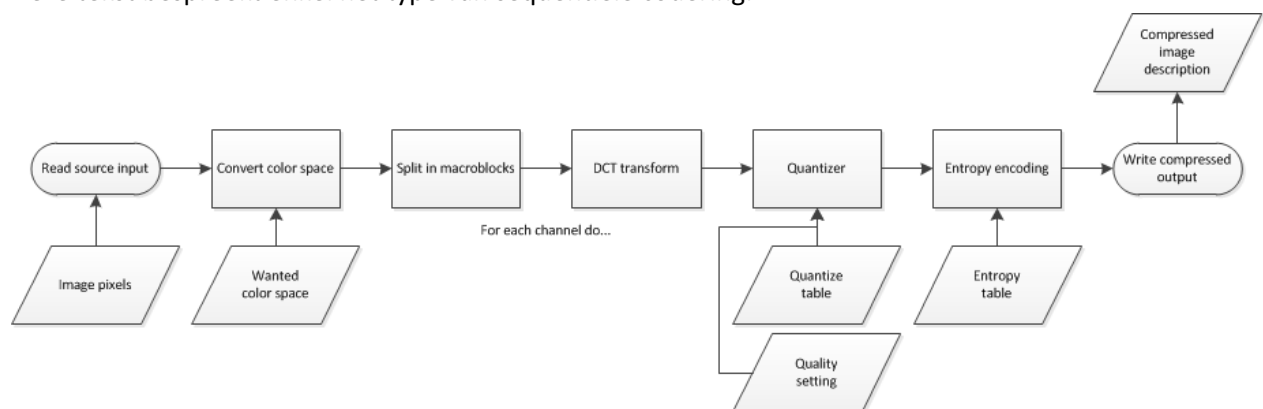
In deze tekst zullen we het JPEG compressie algoritme bespreken. Het algoritme zelf bestaat al sinds begin jaren 1990, zoals bij elke technologie zijn er na het beginconcept nog uitbreidingen en verbeteringen bijgekomen die samen voor de huidige implementatie hebben gezorgd (anno 2011).

JPEG compressie maakt gebruik van een aantal al bestaande technieken die samen voor een efficiënte(re) beschrijving zullen zorgen van de originele afbeelding. JPEG legt geen beperkingen op het soort afbeelding dat het moet verkleinen, eigenschappen zoals het aspect ratio of de gebruikte dimensies kunnen ook vrij worden gekozen.

Er zijn verschillende typen van het JPEG algoritme:

- Sequentiële codering (elke component zal door één, van links naar rechts en van boven naar onder, scan gecodeerd worden)
- Progressieve codering (codering zal gebeuren over meerdere scans)
- "Lossless" codering (wel compressie maar geen verlies aan kwaliteit, de rest is "lossy")
- Stapsgewijze codering (codering gebeurt telkens in hogere resoluties)

Deze tekst bespreekt enkel het type van sequentiële codering.



Figuur 1 Overzicht compressie algoritme.

Als bron heb je meestal een afbeelding opgedeeld in RGB pixels; R staat voor de rode component, G voor de groene en B voor de blauwe. JPEG kan omgaan met 24 bit of 36 bit pixels, 36 bit is nodig voor bijvoorbeeld medische toepassingen. Normaal werkt men met 8 bit per component. Elke component kan dan een gehele waarde krijgen van 0 tot en met 255.

Een formaat zoals RAW, dat geen compressie gebruikt en letterlijk de 24bits wegschrijft, zal voor een afbeelding met de dimensies 100 in breedte en 50 in hoogte een totaal van $(100 * 50 * 3)$ 15.000 bytes nodig hebben om de afbeelding weg te schrijven.

Merk op dat RAW geen vorm van "headers" gebruikt – JPEG heeft deze wel via extensies zoals JFIF (JPEG File Interchange Format). JFIF zal op verschillende plaatsen duidelijkheid scheppen waar het originele artikel van JPEG (al dan niet opzettelijk) te weinig duidelijkheid aanbiedt.

Er bestaan nog andere extensies zoals EXIF, die in groot gebruik is bij bv digitale camera's. Omdat deze tekst over het algoritme zelf gaat zullen we indien nodig enkel JFIF gebruiken.

150x200 - JPEG are saved using Phoptshop CS4 "Save for Web" option.



Figuur 2 Byte vergelijking RAW en JPEG.

In Figuur 2 kan je zien dat het RAW bestand 90.000 bytes omvat. Het JPEG algoritme slaagt er toch in om dit terug te brengen naar 7.035, dit door in te boeten op kwaliteit. We zullen nu elke stap in het algoritme bekijken en aantonen waarom zulk groot verschil in bytes kan behaald worden.

Stap 1: inlezen bronbestand.

Het algoritme kan niet werken zonder bronbestand, er zijn geen echte beperkingen opgelegd op het bronbestand. Het is echter wel zo dat werken met binaire afbeeldingen (zwart/wit) of content met scherpe contrasten (bijvoorbeeld tekst) een slechter resultaat zal opleveren dan afbeeldingen met meer "natuurlijke" content, zoals landschappen of portretten.

JPEG is niet bedoeld om een complete oplossing te zijn voor alle soorten afbeeldingen en men moet daarom als eindgebruiker de juiste keuzes maken. Bij binaire afbeeldingen kan men dan beter opteren voor bijvoorbeeld GIF (Graphics Interchange Format) of andere formaten.

Deze eerste stap zorgt ervoor dat het bronbestand in een bruikbaar datatype komt. Een matrix van pixels met voor elke pixel 3 waarden is een logisch datatype om hiervoor te gebruiken.

Stap 2: Veranderen van kleuruimte.

Step 2.1: Veranderen van kleuruimte.

JPEG maakt gebruik van onze visuele eigenschappen, zo zijn onze ogen gevoeliger voor variaties in licht dan in kleur. RGB is echter geen goede kleuruimte om lichtsterkte en kleur van elkaar te scheiden.

Een kleuruimte is een bepaalde manier om kleuren te beschrijven. Bij RGB is een kleur opgedeeld in 3 componenten namelijk een rode, groene en blauwe component. JPEG laat toe om de kleur te herschrijven, maar legt geen bepaalde kleuruimte op. JFIF vult JPEG hier verder aan en stelt een bepaalde RGB naar YCbCr conversie voor, die moet gebruikt worden voor optimale compressie te bekomen.

De conversie van RGB naar YCbCr doet men volgens volgende formule:

$$Y = 0.299 * R + 0.587 * G + 0.114 * B$$

$$Cb = -0.1687 * R - 0.3313 * G + 0.5 * B + 128$$

$$Cr = 0.5 * R - 0.4187 * G - 0.0813 * B + 128$$

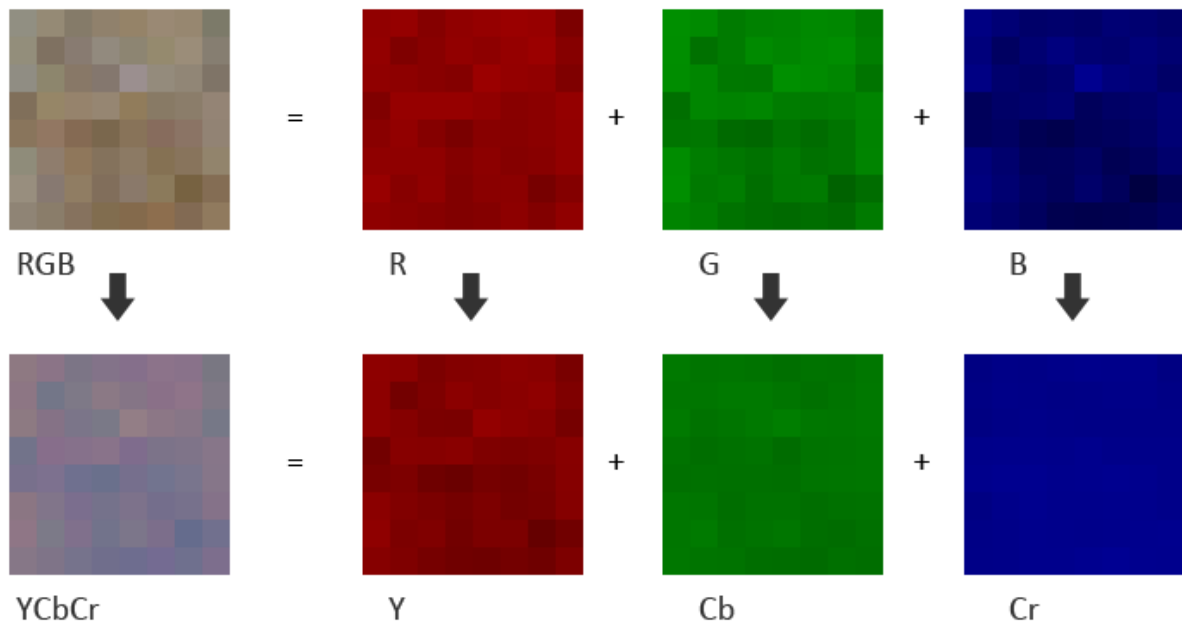
De conversie van YCbCr naar RGB is dan als volgt:

$$R = Y + 1.402 * (Cr - 128)$$

$$G = Y - 0.34414 * (Cb - 128) - 0.71414 * (Cr - 128)$$

$$B = Y + 1.772 * (Cb - 128)$$

Als voorbeeld van deze conversie nemen we een macroblok en splitsen we de kanalen, in Figuur 3 zien we het resultaat.



Figuur 3 RGB en YCbCr kanaal opsplitsing.

We zien dat de grootste variatie in de component Y zit. We mogen ons niet misleiden en denken dat bijvoorbeeld Cb een directe vervanger is van G, want dit is niet zo. De 3 componenten van een pixel worden hergebruikt. Voor terug naar RGB te gaan moeten we de 2^{de} set formules gebruiken.

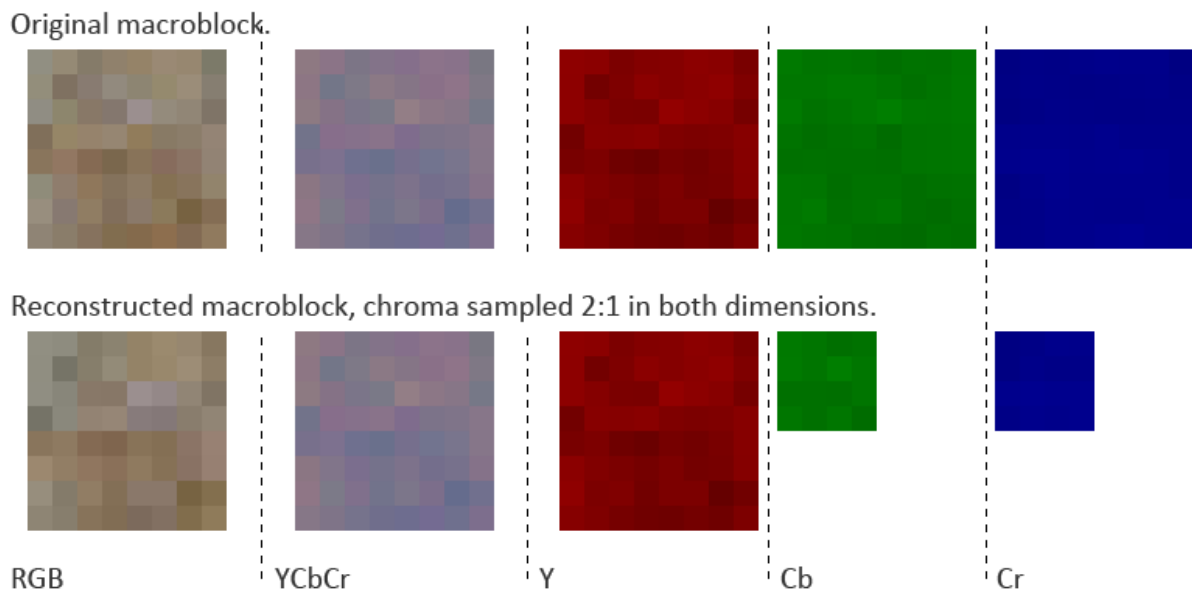
De component Y bevat nu alle licht informatie, Cb en Cr zijn de kleurverschillen van blauw en rood tegenover het licht. Cb en Cr noemen we de chrominantie waarden.

Stap 2.2: Sub sampling (van de chrominantie waarden).

Omdat het oog relatief ongevoelig is voor kleine “fouten” in chrominantie waarden kunnen we deze verder bewerken. JPEG laat toe om op verschillende manieren sub sampling toe te passen. Sub sampling is hetzelfde als de resolutie wijzigen, in dit geval gaan we de resolutie verlagen.

Vanaf deze stap zullen we de componenten elk apart bewerken. Aan component Y wordt er in deze stap niets gedaan, zij blijft op dezelfde resolutie (1:1 sampling), De chrominantie componenten worden (apart) aangepast in resolutie, vaak worden ze beide 2:1 gesampled in breedte en 1:1 of 2:1 in hoogte.

Een sampling van X:Y kunnen we lezen als: voor elke X monsters slagen we Y monsters op. Als we dus een bemonstering van 2:1 doen halveren we de resolutie.



Figuur 4 Vergelijking van originele en bemonsterde macroblok.

Stap 3: Opdelen in macroblokken.

Er kan aangenomen worden dat pixels die dicht bij elkaar liggen niet veel van elkaar zullen verschillen. Als we dus de afbeelding opsplitsen in kleinere stukken (één stuk is een macroblok) kunnen we veronderstellen dat elke macroblok gemiddeld gelijkwaardige pixels zal bevatten.

Eén macroblok bestaat uit een 8x8 matrix van pixels. Als er geen geheel aantal macroblokken van 8x8 in het bronbestand zitten zullen er waardes moeten bijgemaakt worden door padding. Als de afbeelding 75 pixels breed is kunnen er 9 hele macroblokken uit. We zullen daarom het bestand verbreden naar 80 pixels. Er kan zelf gekozen worden welke waardes deze extra pixels krijgen.

Een simpele manier is om ze gewoon 0 (zwart) te houden. Een mogelijk betere oplossing zou zijn om een gemiddelde waarde te gebruiken, omdat er dan gemiddeld minder contrast zit tussen de rand van de echte pixels en de toegevoegde pixels. Dit kan helpen bij stap 4 (DCT transformatie). Om de eventuele uitbreiding er nadien terug af te halen, slagen we in het JPEG bestand de echte dimensies op zodat we de overbodige pixels kunnen weglaten.

Na deze stap beschikken we over enkele lijsten van macroblokken (één lijst voor de Y, Cb en Cr componenten).

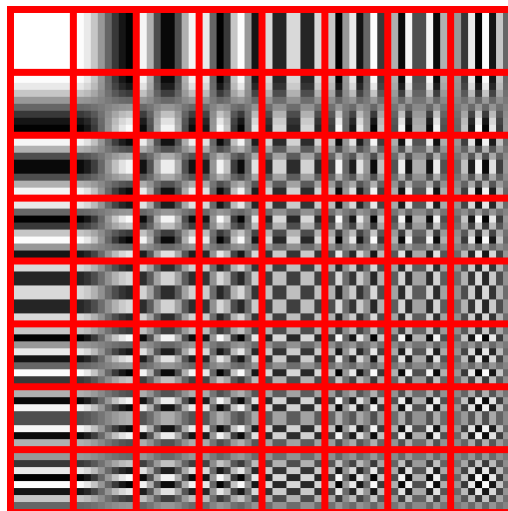


Figuur 5 Afbeelding opgedeeld in een lijst van macroblokken.

Stap 4: De discrete cosinus transformatie en zijn inverse (2D)

Ondertussen beschikken we over 3 lijsten met macroblokken, elke macroblok is een 8x8 matrix van getallen. Er bestaan manieren om een relatie te vinden tussen deze 64 getallen. Eén manier is de “discrete cosinus transformatie”.

De DCT (2D) beschikt zelf over 64 basisfrequenties, na de transformatie zal een 8x8 matrix vervangen worden door die 64 frequenties. We kunnen aannemen dat elke macroblok om te vormen is naar die 64 verschillende frequenties. De frequenties die DCT gebruikt zijn niet evenredig aanwezig in een macroblok. De DCT zal daarom voor elke frequentie een bepaalde factor (de DCT coëfficiënt) berekenen. De basisfrequenties zien er uit als volgt:



Figuur 6 DCT's basis frequenties. (Afbeelding van Wikipedia)

Het valt ons op dat op locatie (0,0) een gemiddelde “DC” waarde kan gevonden worden en dat de “AC” frequenties steeds hoger worden naarmate we (8,8) benaderen.

De DCT formule voor een 8x8 matrix is de volgende:

$$G(u, v) = \sum_{x=0}^7 \left(\sum_{y=0}^7 \left(\alpha(u) * \alpha(v) * g(x, y) * \cos\left(\frac{\pi}{8} * \left(x * \frac{1}{2}\right) * u\right) * \cos\left(\frac{\pi}{8} * \left(y * \frac{1}{2}\right) * v\right) \right) \right)$$

De iDCT formule voor een 8x8 matrix is de volgende:

$$g(x, y) = \sum_{u=0}^7 \left(\sum_{v=0}^7 \left(\alpha(u) * \alpha(v) * G(u, v) * \cos\left(\frac{\pi}{8} * \left(x * \frac{1}{2}\right) * u\right) * \cos\left(\frac{\pi}{8} * \left(y * \frac{1}{2}\right) * v\right) \right) \right)$$

Waar voor beide formules het volgende geldt:

$$0 \leq x < 8$$

$$0 \leq y < 8$$

$$0 \leq u < 8$$

$$0 \leq v < 8$$

$g(x, y)$ is de pixel – component waarde op plaats (x, y)

$G(u, v)$ is de DCT coëfficiënt op plaats (u, v)

$$\alpha(z) = \begin{cases} \sqrt{\frac{1}{8}}, & \text{voor } z = 0 \\ \sqrt{\frac{2}{8}}, & \text{voor } z \neq 0 \end{cases}$$

Alvorens we de transformatie toepassen moeten we de componentwaarden die nu een bereik hebben van 0 tot en met 255 verschuiven naar -128 tot en met 127. Het verschuiven van het bereik doen we door 128 af te trekken van elke component. Na de iDCT zullen we terug 128 optellen bij elke component.

Merk op dat er in deze stap geen compressie behaald wordt, de 64 componentwaarden zijn nu vervangen door 64 DCT coëfficiënten. Het is zelfs zo dat we voor de coëfficiënten te beschrijven meer bits nodig hebben dan voor de componentwaarden zelf.

Stap 5: Quantisatie van de DCT coëfficiënten.

In deze stap veranderen we de coëfficiënten in een macroblock van reële getallen naar gehele getallen. Hiervoor zullen we de coëfficiënten afronden. Bij elke afronding verliezen we precisie en zal de iDCT niet meer exact dezelfde waarden opleveren. Dit is echter niet zo erg bij de hogere frequenties. Onze ogen zijn minder gevoelig bij fouten op hoge frequenties.

We zullen dus proberen om zoveel mogelijk verlies te hebben bij de hoge frequenties en zo weinig mogelijk bij de lage frequenties. Een coëfficiënt zal worden gedeeld door een bepaalde quantisatie waard. We kunnen 64 verschillende waarden gebruiken om elke coëfficiënt zo efficiënt mogelijk te quantiseren.

Omdat we 2 verschillende soorten macroblokken hebben (één met lichte informatie en twee met chroma informatie), hebben we ook 2 soorten quantisatie tabellen.

De tabel voor de macroblok met lichte informatie:

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

Figuur 7 Quantisatie tabel voor de Y component.

De tabel voor de macroblokken met chroma waarden:

$$Q_c = \begin{bmatrix} 17 & 18 & 24 & 47 & 99 & 99 & 99 & 99 \\ 18 & 21 & 26 & 66 & 99 & 99 & 99 & 99 \\ 24 & 26 & 56 & 99 & 99 & 99 & 99 & 99 \\ 47 & 66 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \end{bmatrix}$$

Figuur 8 Quantisatie tabel voor de Cb en Cr componenten.

Het algoritme heeft nog een extra kwaliteit parameter dat ingesteld kan worden, van 1 tot 100, de uiteindelijke quantisatie formule is:

q is de kwaliteits parameter

$$\alpha = \begin{cases} \frac{50}{q} & \text{if } 1 \leq q \leq 50 \\ 2 - \frac{q}{50} & \text{if } 50 \leq q \leq 100 \end{cases}$$

$$x = \frac{F(x, y)}{\alpha * Q(x, y)} \in \mathbb{Z}$$

$F(x, y)$ is de DCT coëfficiënt op plaats (x, y)

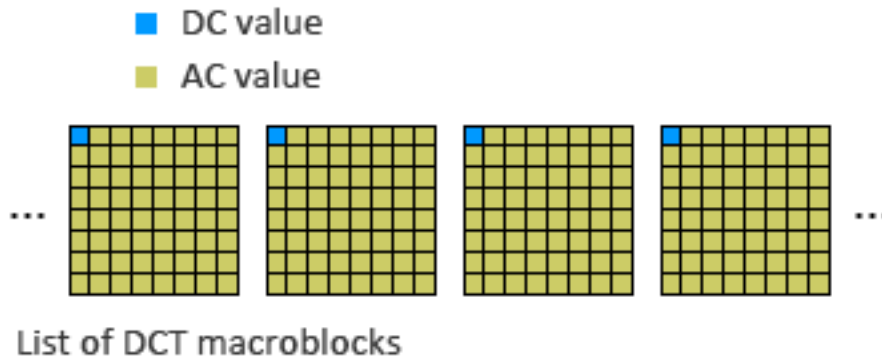
$Q(x, y)$ is de quantisatie waarde op de plaats (x, y)

Stap 6: Codering.

Tot hertoe hebben we gebruik gemaakt van de eigenschappen van het menselijk oog. In deze stap zullen we proberen de bekomen macroblokken zo efficiënt mogelijk te beschrijven. Dit wil dus zeggen met zo weinig mogelijk bytes.

Stap 6.1: DC codering.

Aanliggende macroblokken hebben vaak ongeveer dezelfde DC waarden. Herinner dat de DCT coëfficiënten 64 basisfrequenties voorstellen waarvan de coëfficiënt op locatie (0,0) de DC waarde is van de hele blok.



Figuur 9 Verduidelijking locatie DC en AC coëfficiënten.

We zullen de DC waarden veranderen zodat de opeenvolgende waarden enkel het verschil bevatten tegenover de vorige blok. Dit via volgende formule:

$$\Delta DC = DC_i - DC_{i-1}$$

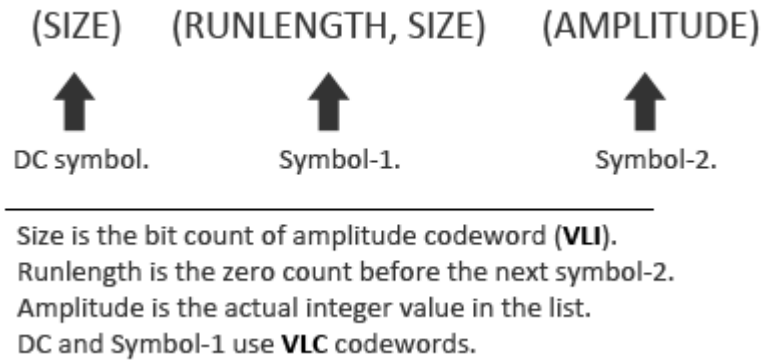
De nieuwe DC waarden zullen nu een stuk kleiner zijn dan ze origineel waren. De omgekeerde weg kan uiteraard via volgende formule:

$$DC_i = DC_i + DC_{i-1}$$

Stap 6.2: Zigzag sequentie.

Gemiddeld zal nu, door de kwantisatie in stap 5, een groot deel van de AC coëfficiënten nul zijn. Hoe meer men naar rechtsonder gaat, hoe meer nullen we tegenkomen. In deze stap zullen we de macroblok omzetten van een matrix naar een lijst.

Als we rekening houden met het patroon van de nullen kunnen we door middel van te zigzaggen de meeste nullen verschuiven naar achteraan in de lijst en de meeste niet nulwaarden naar voren.



Figuur 11 Soorten symbolen.

We gaan één per één de lijst van waardes af, voor elke niet nul waarde plaatsen we een “Symbol-1”. Achter “Symbol-1” komt “Symbol-2” die niets meer is dan de echte integer waarde in de lijst. De DC waarde (de eerste waarde van de lijst) krijgt zijn eigen symbool.

Het “size” element duid op het aantal bits dat nodig is om het volgend symbool (Symbol-2) te lezen in de bit-stroom. Het aantal bits weten we door het codewoord op te vragen van dat Symbol-2. Alle Symbol-2 codes staan in een zogenaamde VLI lijst (Variable-Length Integer).

Het “runlength” element duid op het aantal nullen die voor de eerstvolgende niet nul waarde komen. Zo kan er een maximum van 15 nullen geregistreerd worden in één Symbol-1. Omdat dit echter niet genoeg is (er kunnen meer dan 15 opeenvolgende nullen zijn) is er een uitbreiding. (15,0) duid op 16 nullen, een zogenaamde “extensie”, zo kunnen er 3 opeenvolgende (15,0) komen. De 4^{de} Symbol-2 sluit de reeks af met zijn “runlength” waarde.

Er is nog een 2^{de} uitbreiding op Symbol-1, namelijk (0,0). Wanneer de “runlength” en de “size” een waarde 0 hebben noemen we het Symbol-1 een “End Of Block” (of EOB) symbool. Hij staat aan het einde van de lijst en kan gebruikt worden wanneer we weten dat er vanaf de vorige Symbol-2 enkel nog nulwaarden komen. Een matrix met enkel een DC coëfficiënt zou dus 3 symbolen nodig hebben. Voor een DC waarde van 3 zou dit de volgende symbolen reeks kunnen zijn: (2)(3)(0,0).

Waarom de amplitude 3 een “size” heeft van 2 zal duidelijk worden wanneer we het VLI woordenboek bespreken.

Het “amplitude” element is de effectieve integer waarde die in de lijst staat.

De 2 woordenboeken (VLI en VLC) bespreken we in stap 6.3.2.

Bit count	Integer range
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	-1023,-512,512,1023
(DC) 11	-2047,-1024,1024,2047

Figuur 13 VLI "size" tabel.

Het coderen van een amplitude naar het codewoord doe je als volgt:

- Voor positieve getallen: van decimaal naar binair omvormen.
- Voor negatieve getallen: van positief decimaal getal naar binair en 1st complement (inverteren) toepassen.

Om dan van het codewoord terug naar decimaal te gaan moet je dan volgende stappen ondernemen:

- 1) Je weet hoeveel bits het codewoord telt. Kijk naar de MSB.
- 2) Is de MSB een "1" dan hebben we te maken met een positief getal en kunnen we simpelweg omzetten van binair naar decimaal.
- 3) Is de MSB een "0" dan hebben we te maken met een negatief getal en moeten we eerst het codewoord inverteren, dan omvormen naar decimaal en vervolgens het resultaat negatief maken.

Stap 6.3.2.2: Het VLC (Variable Length Coding) woordenboek.

Omdat symbolen verschillende keren kunnen terugkomen in de lijst zou het efficiënter zijn om tijdens het coderen in bits aan deze symbolen minder bits toe te kennen. Op die manier sparen we dan bits uit. Huffman doet dit door te kijken naar de waarschijnlijkheid dat een symbool voorkomt.

Als we 10 karakters hebben waarvan er 7 een 'e' zijn. Dan is de waarschijnlijkheid dat we een 'e' lezen dus $7/10^{\text{de}}$ groot. Alle verschillende karakters samen vormen $10/10^{\text{de}}$ (of 1). Dit kunnen we ook doen met de symbolen. Elk symbool komt x aantal keer voor van het totaal aantal symbolen in de lijst.

Er moet een binaire boom gemaakt worden, dit is een structuur waar elke node maar 2 kinderen kan hebben. Dit doen we als volgt. We maken een lijst op van alle unieke symbolen en noteren hun waarschijnlijkheid. We sorteren op deze waarschijnlijkheid, van groot naar klein. We nemen telkens de laagste 2 symbolen (deze hebben de laagste waarschijnlijkheid). We plaatsen vervolgens deze 2 symbolen in een nieuwe node (met de 2 symbolen als zijn kinderen) en geven deze nieuwe node de opgetelde waarschijnlijkheid. We sorteren opnieuw en blijven zo doorgaan tot we maar 2 noden (een symbool op zichzelf is ook een node) over hebben. We kunnen nu gemakkelijk codes toekennen.

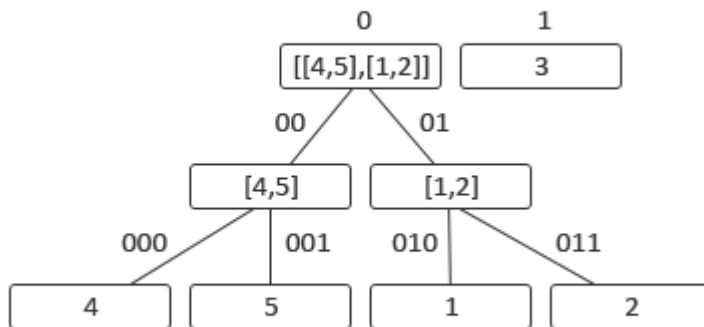
Wat opvalt is dat geen enkel van de codes een prefix is van een andere code. Dit wil dus zeggen dat we geen "markers" moeten gebruiken om een einde van een codewoord aan te duiden. Om te decoderen moeten we simpelweg bits inlezen tot we in het VLC woordenboek eenzelfde codewoord gevonden hebben. Afhankelijk van het symbool dat gevonden is, moeten we als volgt op zoek gaan naar een nieuw VLC symbool of naar een VLI symbool.

Symbols list: (2)(3), (1,2)(-2), (0,1)(-1), (0,1)(-1), (0,1)(-1), (2,1)(-1), (0,0)

Symbol	#, Probability
(2)	1, 1/7
(1,2)	2, 1/7
(0,1)	3, 3/7
(2,1)	4, 1/7
(0,0)	5, 1/7

■ Encoded using VCI.

SORT	HUFFMAN	SORT	HUFFMAN	HUFFMAN	SORT
3, 3/7	3, 3/7	3, 3/7	3, 3/7	3, 3/7	[[4,5],[1,2]], 4/7
1, 1/7	1, 1/7	[4,5], 2/7	[4,5], 2/7	[[4,5],[1,2]], 4/7	3, 3/7
2, 1/7	2, 1/7	1, 1/7	[1,2], 2/7		
4, 1/7	[4,5], 2/7	2, 1/7			
5, 1/7					



- 1 = (2) = 010
 2 = (1,2) = 011
 3 = (0,1) = 1
 4 = (2,1) = 000
 5 = (0,0) = 001

Bit list: 010(3)011(-2)1(-1)1(-1)1(-1)000(-1)001

Figuur 14 Uitgewerkt Huffman voorbeeld.

