Beeldcodering m.b.v. Spatiële Interpolatie Piramides

AFSTUDEERVERSLAG

Mei 1992

R. Romein en J.J. van Spanje

TU Delft
Faculteit der Elektrotechniek
Vakgroep Informatietheorie
Technische Universiteit Delft

Voorwoord

Dit verslag is geschreven naar aanleiding van ons afstudeeronderzoek dat wij hebben verricht bij de vakgroep Informatietheorie van de Faculteit der Elektrotechniek van de Technische Universiteit Delft.

We willen graag ing. Peter Lems, Michael de Clerck, Ron van den Bulk en Remco Tissink bedanken voor de goede werksfeer. Ook willen we de mensen van de beeldtelefoongroep bedanken, evenals onze bedrijfsbegeleider dr. ir. J.C.A. van der Lubbe. Tenslotte willen we onze examinator ir. W. Holst bedanken met name voor zijn tips met betrekking tot de verslaglegging.

Robert Romein
Jos van Spanje

mei 1992

Inhoudsopgave

Voorwoord

Inhoudsopgave

Opdrachtformulier

Samenvatting

1. Inleiding 1
1.1 Coderingsmethoden 3
1.1.1 Puls Code Modulatie 3
1.1.2 Transformatie codering 5
1.1.3 Discrete Cosinus Transformatie 6
1.2 Kwantisatie 8
1.3 De H.261 codec 12
1.3.1 Zender (coder) 14
1.3.2 Ontvanger (decoder) 17

2. Beeldcodering m.b.v.
Spatiële Interpolatie Piramides 18
2.1 Het codeerschema voor grijswaardebeelden 19
2.2 Bepaling van de codelengte 26
2.3 Foutbepaling 27
2.4 Het SIP coderen van kleurenbeelden 28
2.5 Codering aan de beeldranden 29

3. Software implementatie. 31
3.1 Globaal 31
3.2 Bloksgewijze bespreking 33

4. Kwaliteitsaspecten en metingen. 36
4.1 Objectieve kwaliteitsmeting 36
4.2 Subjectieve kwaliteitsmeting 38
4.3 Bepaling van de kansdichtheidsfunctie 40
4.4 Metingen 43

5. Conclusies en aanbevelingen 46
5.1 Conclusies 46
5.2 Aanbevelingen 47

Referenties

Bijlagen
A Testbeelden
B Gedecodeerde beelden H.261 standaard
C Gedecodeerde beelden SIP methode
D Mean Square Error beelden
E Meetwaarden
F C broncode

Samenvatting

De verwachting is dat voor het jaar 2000 de huidige telefoon vervangen zal gaan worden door de beeldtelefoon. Ook zal dan de N-ISDN in gebruik zijn genomen. Om de beelden van de beeldtelefoon over dit net te kunnen transporteren is het nodig dat de data die nodig is voor een beeld drastisch wordt teruggebracht. Dit gebeurt door beeldcodering.

In dit verslag wordt een nieuwe manier van beeldcodering besproken. Deze methode werkt met spatiële interpolatie piramides (SIP). Er zijn in C programma's geschreven om deze methode te testen en die in dit verslag besproken worden.

Ook zullen enkele met beeldcodering en beeldtelefonie samenhangende onderwerpen worden besproken. Zoals kwantisatie en de H.261 standaard voor aansluiting van de beeldtelefoon op het N-ISDN. Evenals enkele bestaande beeldcoderingsmethoden.

1. Inleiding

Over enkele jaren zal het Integrated Services Digital Network (ISDN) door het grote publiek in gebruik worden genomen. De mogelijkheden die dit netwerk biedt zijn onder andere digitale communicatie en een grotere bandbreedte dan het huidige telefoonnet. Een rond het jaar 2000 in gebruik te nemen breedbandige versie van ISDN is B-ISDN. Dit B-ISDN biedt de gebruiker een hoogste bitsnelheid van ongeveer 135 Mbit/s en moet ook digitale HDTV informatie kunnen transporteren.

Voor het zover is zal er al een smalbandige versie van ISDN in gebruik worden genomen. Dit N-ISDN stelt een bandbreedte van 144 kbit/s beschikbaar aan de gebruiker. De 144 kbit/s van N-ISDN is opgebouwd uit één kanaal van 64 kbit/s voor geluid, één kanaal van 64 kbit/s voor beeld en 1 kanaal van 16 kbit/s voor besturing. Dit wordt aangegeven met 2B+D.

De telefoon zoals deze in zijn huidige vorm bestaat zal waarschijnlijk vervangen gaan worden door de beeldtelefoon. De eerste generatie beeldtelefoons uit de jaren tachtig bleek geen succes op de consumentenmarkt. Deze eerste generatie maakt gebruikt van het telefoonnet voor het transporteren van de signalen. Dit netwerk kan digitale signalen tot 9,6 kbit/sec verwerken. Men verwacht dat de nu in de ontwikkeling zijnde tweede generatie beeldtelefoons meer succes zal hebben. Deze verwachting wordt gestimuleerd door de invoering van het N-ISDN. Hierdoor krijgt men namelijk de beschikking over

een grotere transmissiecapaciteit dan het telefoonnet levert. Dit kan de beeldkwaliteit ten goede komen.
Om de beeldtelefoon universeel bruikbaar te maken, is het transmissieformaat van de beelden gestandaardiseerd. Deze standaard is het *Common Intermediate Format* (CIF). De beelden moeten voor verzending dan ook eerst worden omgezet naar dit CIF formaat.
Eén seconde aan CIF beelden levert een hoeveelheid data van 12,4 Mbit op. N-ISDN biedt voor transmissie van beelden een kanaal met een bandbreedte van 64 kbit/s. Deze bandbreedte is dus nog steeds veel te klein voor het transporteren van CIF beelden. Het is dus noodzakelijk dat de voor de beelden benodigde hoeveelheid data met een factor 194 wordt teruggebracht.

Er zijn twee manieren om de hoeveelheid data te verminderen: datareductie en datacompressie.
Bij datareductie wordt er grofweg informatie weggelaten. Bijvoorbeeld kleine details die voor de menselijke waarnemer niet of nauwelijks zichtbaar zijn. Datareductie geeft per definitie een informatie verlies dat niet omkeerbaar is. De bekendste vorm van datareductie is kwantisatie. In paragraaf 1.2 zal op het verlies van informatie dat wordt veroorzaakt door kwantisatie worden ingegaan.
Datacompressie is het verwijderen van redundantie. Bij beeldbeschrijvingen gebeurt dit vaak door correlatie uit de data te verwijderen. Bij beelden komt zowel spatiële als temporele redundantie voor. Spatiële redundantie wordt veroorzaakt doordat naburige pixels binnen één beeld gecorreleerd zijn. Temporele redundantie wordt veroorzaakt door correlatie tussen pixels in opeenvolgende beelden.
Compressie is altijd verliesvrij. Men kan dus altijd het origineel terugwinnen uit de gecomprimeerde versie. In paragraaf 1.1 zullen enkele coderingsmethoden worden besproken.

Voor aansluiting op het N-ISDN is voor de beeldtelefoon een standaard ontwikkeld. Dit is de H.261 standaard. Aan de vakgroep informatietheorie van de TU Delft houdt men zich onder meer bezig met datacompressie en

beeldtelefonie. Men heeft een software simulatiepakket ontwikkeld voor beeldtelefonie dat voldoet aan de H.261 standaard. Op dit simulatiepakket en de H.261 standaard zal in een aparte paragraaf dieper worden ingegaan.

Uit contacten tussen de vakgroep informatietheorie en een soortgelijke afdeling van de Academy of Sciences in Moskou bleek dat men in Rusland een ander beeldcoderingsalgoritme heeft ontwikkeld. Dit Russische beeldcoderings-algoritme heet Component Transformation with Pixel Interpolation (CTPI).

Onze opdracht was om dit algoritme te analyseren en te onderzoeken hoe dit algoritme geïmplementeerd kon worden in het software model van de H.261. De opdracht omvatte verder een onderzoek naar de performance van dit algoritme. Met name de snelheid (tijd om één beeld te comprimeren) en de compressionratio versus beeldkwaliteit (objectief en subjectief), in vergelijking met het nu geïmplementeerde algoritme. Ook zouden in het kader van het onderzoek eventuele verbeteringen in het algoritme worden onderzocht.

1.1 Coderingsmethoden

1.1.1 Puls Code Modulatie

Deze coderingsmethode is de oudste en bekendste. Dit komt waarschijnlijk door de eenvoud van het achterliggende idee. Een PCM codeersysteem in feite niets anders als een bemonsteringssysteem en een amplitudekwantisator. De amplitude van het ingangssignaal wordt dus op regelmatige intervallen omgezet in een (binaire) codewoord. Uit waarnemingen is gebleken dat PCM gecodeerde spraak- en beeldsignalen minimaal 8 bit per codewoord (sample) vragen. Wil men hoge kwaliteitsbronnen
weergeven zoals medische beelden dan kan dit oplopen tot 16 bit/sample. Omdat bij PCM alleen datareductie en geen datacompressie wordt toegepast is dit een relatief inefficiënt codeersysteem.
De ontvanger herstelt de samples naar hun originele amplitudewaarden. Met

behulp van een laagdoorlaatfilter wordt uit deze amplitudewaarden een benadering van het oorspronkelijke signaal gefilterd.

De maximale frequentie in een telefoongesprek is 4 kHz, voor een goede bemonstering zijn dus 8000 samples nodig (Nyquist theorema). Dit geeft bij 8-bits codewoorden 64 Kbit/s. Om een kleurenbeeld te coderen worden de componenten Y (helderheid), U (R-Y) en V (B-Y) bemonsterd met respectievelijk 13,5 MHz, 6,75 MHz en 6,75 MHz. Bij het gebruik van 8-bits codewoorden levert dit een transmissiesnelheid op van 216 Mbit/s.

Een verbeterde vorm van PCM is de zogenaamde differentiële PCM. Hierbij wordt niet één sample per keer verstuurd maar het verschil tussen de werkelijke samplewaarde en de predictiewaarde van datzelfde sample. De predictiewaarde wordt bepaald uit de waarden van voorgaande samples. Op deze manier wordt nu ook de spatiële redundantie tussen twee samples aangepakt. Omdat nu alleen verschillen gecodeerd worden, zal met minder bit per sample volstaan kunnen worden.
Ook bij een beeld kan men volgens dit predictie mechanisme te werk gaan. Er wordt dan aan de hand van de waarden van naburige pixels een voorspelling gedaan van de waarde van een pixel.
In figuur 1.1 wordt hiervan een voorbeeld gegeven. De waarde van witte pixel uit fig. 1.1 wordt geschat door gebruik te maken van de waarden van de drie zwarte pixels. Het enige wat nu verstuurd wordt is de predictiefout. Dit is het verschil tussen de geschatte en de werkelijke pixelwaarde.

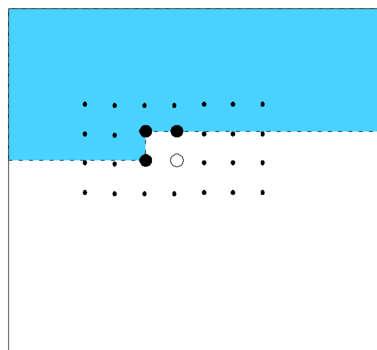


Fig. 1.1: Predictie model voor DPCM.

Met DPCM wordt wel enige bitreductie bereikt. De DPCM is echter niet optimaal.

1.1.2 Transformatie codering

De transformatie van een signaal is bedoeld om de correlatie in een signaal te verminderen. In plaats van pixelwaarden worden frequentiecoëfficiënten bepaald. Deze coëfficiënten zijn afhankelijk van het type transformatie nog maar weinig gecorreleerd. Sommige coëfficiënten dragen door hun geringe energie maar heel weinig bij aan het totale signaal. Deze coëfficiënten kunnen dan ook worden weggelaten. Met een speciaal bitallocatie algoritme wordt dan nog voor iedere coëfficiënt het optimale aantal bits bepaald. Hierdoor kunnen de belangrijkere coëfficiënten nauwkeuriger worden weergegeven. Bij beelden wordt een transformatie toegepast op beeldfragmenten van *N bij N* pixels en niet op een geheel beeld, omdat dit te veel rekenwerk zou kosten. Fig. 1.2 laat een voorbeeld zien van een in fragmenten opgedeeld beeld.

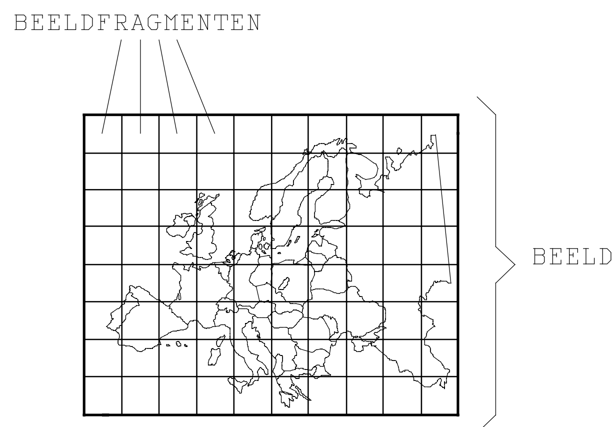


Fig. 1.2: Een in fragmenten opgedeeld beeld.

Het belangrijkste resultaat van een transformatie is het verkrijgen van de minder gecorreleerde coëfficiënten. De optimale decorrelerende transformatie is de Karhunen-Loève Transformatie (KLT). Omdat deze echter zeer veel rekenwerk vraagt wordt de KLT alleen als referentie gebruikt. Een andere goed decorrelerende transformatie is de Discrete Cosinus Transformatie (DCT).

De DCT is de transformatie die in de H.261 standaard voor beeldtelefonie wordt gebruikt. Daarom zal in de volgende paragraaf dieper op de DCT worden ingegaan. Een derde belangrijke transformatie is de Discrete Fourier Transformatie (DFT). Deze wordt voornamelijk toegepast in de signaalbewerking vanwege zijn gunstige rekentijden.

1.1.3 Discrete Cosinus Transformatie

De Discrete Cosinus Transformatie behoort tot een uitgebreide familie van "sinusoïdale" transformaties. Tot deze familie behoort bijvoorbeeld ook de Fouriertransformatie. De DCT wordt vooral toegepast in de beeldbewerkingstechniek, omdat voor conventionele beelden met hogelijk gecorreleerde pixels zijn performance bijna even goed is als de ideale KLT. In tegenstelling tot de DFT is de DCT een reële transformatie.

Voor de DCT is een discreet signaal nodig. Deze signaalwaarden, samples, vertegenwoordigen het signaal voldoende als voldaan is aan de Nyquist eis. Deze eis houdt in dat er minimaal twee samples per periode beschikbaar moeten zijn van de hoogste frequentie die in het signaal voorkomt. Minder heeft tot gevolg dat de hoge frequenties van een signaal worden vervormd. De wiskundige uitdrukking voor de DCT ziet er als volgt uit:

$$y[k] = \sqrt{\frac{2}{N}} \cdot C(k) \cdot \sum_{n=0}^{N-1} x[n] \cdot \cos\left(\frac{(2n+1) \cdot k\pi}{2N}\right), \qquad k=0,1,\ldots,N-1 \quad (1)$$

$$met\ C(0) = \frac{1}{\sqrt{2}}, \qquad en\ C(k) = 1,\ k \neq 0$$

Hierin is n het volgnummer van de samples en is N het aantal samples.

In fig. 1.3b zijn de frequentiecoëfficiënten weergegeven met links de lage frequenties en rechts de hoge frequenties. Duidelijk is te zien dat de coëfficiënten die de lage frequenties vertegenwoordigen meer energie bevatten

dan de coëfficiënten die de hoge frequenties vertegenwoordigen.

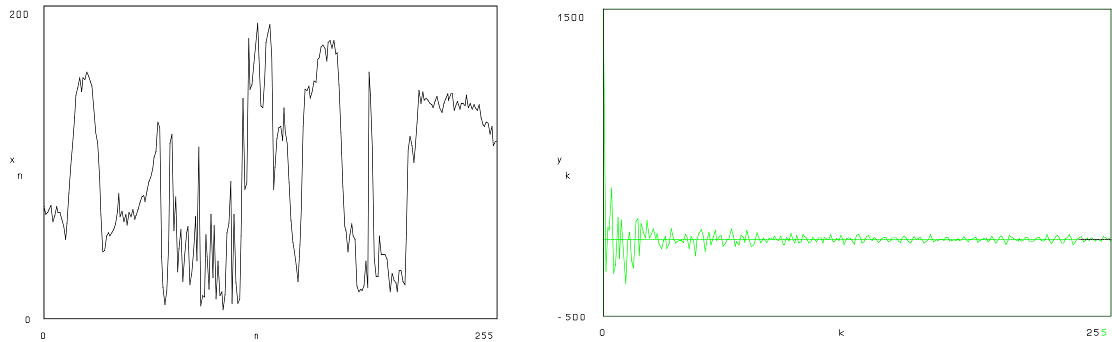
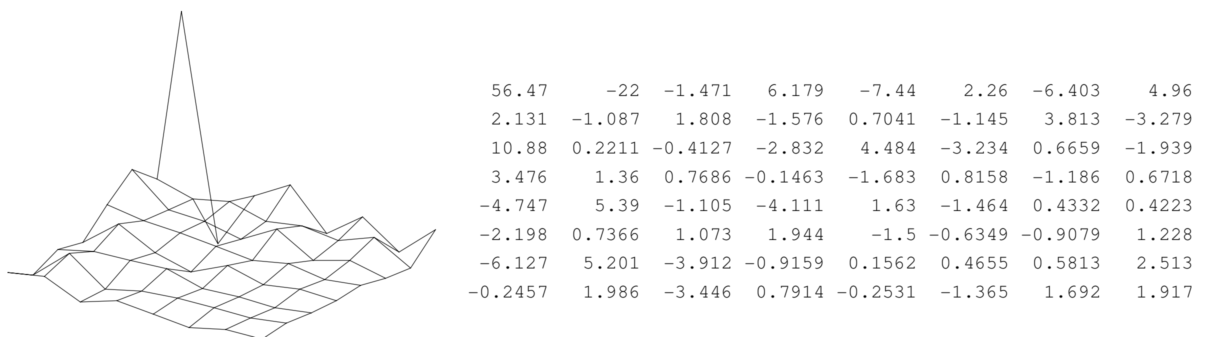


Fig. 1.3: a. Beeldlijn van 256 pixels. b. DCT coëfficiënten.

Voor beelden is een 2-dimensionale DCT nodig. Deze kan gemaakt worden door een 1-dimensionale DCT uit te voeren op alle rijen van een beeldfragment en vervolgens één op alle kolommen.

Het effect van de 2-dimensionale DCT is dat de coëfficiënten die de lage frequenties vertegenwoordigen in de linkerbovenhoek terecht komen (zie fig. 1.4). In de rechteronderhoek komen de hoge frequenties. Alleen de coëfficiënten met veel energie hoeven verstuurd te worden.



56.47	-22	-1.471	6.179	-7.44	2.26	-6.403	4.96
2.131	-1.087	1.808	-1.576	0.7041	-1.145	3.813	-3.279
10.88	0.2211	-0.4127	-2.832	4.484	-3.234	0.6659	-1.939
3.476	1.36	0.7686	-0.1463	-1.683	0.8158	-1.186	0.6718
-4.747	5.39	-1.105	-4.111	1.63	-1.464	0.4332	0.4223
-2.198	0.7366	1.073	1.944	-1.5	-0.6349	-0.9079	1.228
-6.127	5.201	-3.912	-0.9159	0.1562	0.4655	0.5813	2.513
-0.2457	1.986	-3.446	0.7914	-0.2531	-1.365	1.692	1.917

Fig. 1.4: 2D DCT coëfficiënten van een beeldfragment.

Een nadeel van het uitvoeren van de DCT op beeldfragmenten is het blokeffect. Dit is het zichtbaar worden van de randen van de fragmenten in het gedecodeerde beeld. Dit wordt veroorzaakt door de kwantisatie van de DCT coëfficiënten. Als bijna even grote coëfficiënten in naast elkaar gelegen fragmenten naar verschillende waarden worden gekwantiseerd geeft dit

intensiteitssprongen aan de fragmentranden. Hierdoor worden de fragmentranden zichtbaar. Het blokeffect wordt door het menselijk visueel systeem als zeer storend ervaren.

1.2 Kwantisatie

Kwantisatie is een bekende en veel gebruikte vorm van datareductie. In de H.261 standaard wordt gebruik gemaakt van amplitudekwantisatie (hierna kortweg aangeduid als kwantisatie). Doordat bij datareductie informatie wordt weggelaten wordt er een fout geïntroduceerd. In deze paragraaf wordt uitgelegd hoe de grootte van deze fout kan worden bepaald. Ook worden de voorwaarden besproken waaraan een kwantisator moet voldoen om de fout zo klein mogelijk te houden.

Bij kwantisatie wordt het aantal bits dat nodig is om een sample van een signaal te coderen beperkt. Dit gebeurt door het aantal mogelijke amplitudewaarden te beperken. Kwantisatie is een belangrijke stap in een digitaal communicatiesysteem omdat het in hoge mate de noodzakelijke bitsnelheid bepaalt waarmee een signaal van de bron naar de bestemming wordt getransporteerd.

Bij kwantisatie wordt een gegeven signaalamplitude $y(n)$ op tijdstip n omgezet in amplitude $y_k(n)$ welke afkomstig is uit een eindige verzameling van mogelijke amplitude waarden. Het probleem is hoe, voor een stochastisch signaal met bekende statistische eigenschappen, de verzameling amplituden $\{y_k(n)\}$ gekozen moet worden om de gemiddelde fout tussen $y(n)$ en $y_k(n)$ zo klein mogelijk te houden.

Nu volgt een beschouwing van kwantisatiedistorsie bij een non-uniforme verdeling van de kwantisatieniveaus (zie fig. 1.5). De stappen zijn niet allemaal even groot, maar ze zijn wel symmetrisch om nul ($\Delta y_1=\Delta y_{-1}$) ook

vallen ze allemaal binnen het interval (-V,V). Hierin zijn -V en V de maximale uitsturingsgrenzen van het te kwantiseren ingangssignaal y.

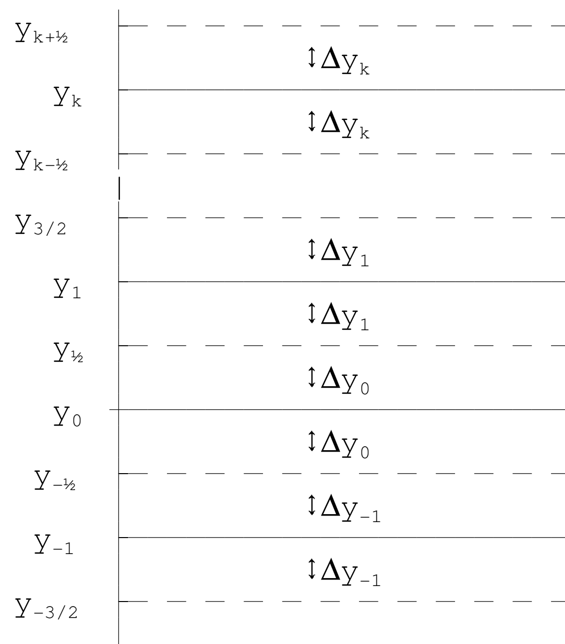


Fig. 1.5: Non-uniform verdeelde kwantisatieniveaus.

Als het signaal tussen grenswaarden $y_{k+½}$ en $y_{k-½}$ ligt wordt deze afgerond naar y_k. Het probleem is de verdeling van de niveaus zó te kiezen dat de kwantisatiefout minimaal is.

Een maat voor de vervorming is de gemiddelde-kwadratische-afwijking welke als volgt gedefinieerd is:

$$\sigma^2_k = \int_{y_{k-1/2}}^{y_{k+1/2}} (y-y_k)^2 \cdot P(y)\; dy \tag{2}$$

Hierin is $(y-y_k)$ de kwantisatiefout en P(y) de kansdichtheidsfunctie van signaal y.

Stel dat de grenswaarden dicht genoeg bij elkaar liggen om te kunnen stellen dat P(y) constant is tussen twee grenswaarden. P(y) heeft dan op het interval een waarde gelijk aan $P(y_{av})$, waarbij:

$$y_{av}=(y_{k-½} + y_{k+½})/2 \tag{3}$$

In fig. 1.6 wordt dit grafisch weergegeven.

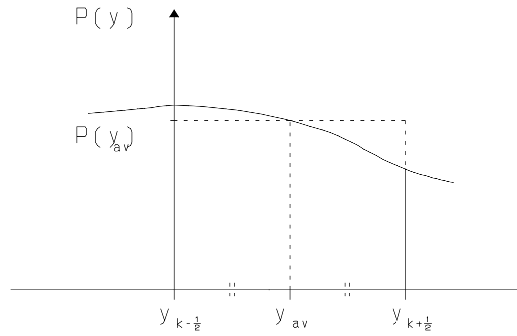


Fig. 1.6: Kansverdeling van pixelwaarden op een interval.

Door voor P(y) de waarde van $P(y_{av})$ te gebruiken kan in (3) $P(y_{av})$ voor het integraalteken gehaald worden. $P(y_{av})$ wordt dus constant verondersteld. Door vervolgens (3) over het interval $[y_{k-½}, y_{k+½}]$ te integreren krijgen we het volgende:

$$\sigma^2_k = \frac{P(y_{av})}{3} \left[(y_{k+½} - y_k)^3 + (y_k - y_{k-½})^3 \right] \qquad (4)$$

Het minimum van deze functie kan door middel van differentiëren worden gevonden. Het blijkt dan dat σ^2_k een minimum heeft als:

$$y_{av} = \frac{y_{k+½} + y_{k-½}}{2} = y_k \qquad (5)$$

De voorwaarde om σ^2_k minimaal te laten zijn is dus dat y_k halverwege $y_{k+½}$ en $y_{k-½}$ moet liggen.

$$y_{k+½} = y_k + \Delta y_k \qquad (6)$$
$$y_{k-½} = y_k - \Delta y_k$$

Substitutie van (6) in formule (4) geeft:

$$\sigma^2_k = \frac{P(y_k)}{3} 2\Delta Y_k^3 = \frac{2}{3} P(y_k) \Delta Y_k^3 \qquad (7)$$

De totale gemiddelde-kwadratische-afwijking voor alle niveaus wordt dan gevonden door die van alle niveaus bij elkaar op te tellen.

$$\sigma^2_{tot} = \frac{2}{3} \sum_{k=-n}^{n} P(y_k) \; \Delta Y_k^3 \qquad (8)$$

Nu zal worden aangetoond dat σ^2_{tot} minimaal is wanneer σ^2_k constant is onafhankelijk van k.

Volgens de definitie van de integraal mogen we zeggen dat:

$$2 \cdot \sum_{k=-n}^{n} P^{1/3}(y_k) \; \Delta y_k = \int_{-V}^{V} P^{1/3}(y) \; dy = 2K \qquad \textbf{(9)}$$

stel: $\mu_k = P^{1/3}(y_k) \; \Delta Y_k$

dan geldt:

$$\sigma^2_{tot} = \frac{2}{3} \sum_{k=-n}^{n} \mu_k^3 \qquad (10)$$

en

$$K = \sum_{k=-n}^{n} \mu_k \qquad (11)$$

Het probleem is nu gereduceerd tot het minimaliseren van de som van de variabelen μ_k afhankelijk van de conditie dat deze som een constante K is.

Volgens de wiskundige Lagrange heeft (10) een minimum als:

$$\mu_{-n} = \mu_{-n+1} = \; ... \; = \mu_{n-1} = \mu_n = \frac{K}{2n+1} \qquad (12)$$

Hieruit volgt dat:

$$P^{1/3}(y_k) \; \Delta Y_k = \frac{K}{2n+1} \qquad (13)$$

dus:

$$\sigma^2_k = \frac{2}{3} \cdot \frac{K^3}{(2n+1)^3} \qquad (14)$$

De totale minimale vervorming wordt nu verkregen door (14) te vermenig-

vuldigen met het aantal niveaus. Hierdoor ontstaat (15). Verificatie kan door substitutie van formule (13) in (8) waarbij de sommatie vervangen mag worden door (2n+1), omdat $K^3/(2n+1)^2$ onafhankelijk is van k.

$$\sigma^2_{min} = \sigma^2_k * (2n+1) = \frac{2}{3} \cdot \frac{K^3}{(2n+1)^2} \tag{15}$$

$$\sigma^2_{min} = \frac{2}{3} \cdot \left(\frac{1}{2} \int_{-V}^{V} P^{1/3}(y)\, dy \right)^3 \cdot \frac{1}{(2n+1)^2} \tag{16}$$

$$\sigma^2_{min} = \frac{1}{12 \cdot (2n+1)^2} \cdot \left(\int_{-V}^{V} P^{1/3}(y)\, dy \right)^3 \tag{17}$$

Het komt er dus op neer dat om de vervorming zo klein mogelijk te houden de kwantisatieniveaus in het midden tussen de kwantisatiegrenzen moeten liggen en dat de oppervlakten, onder de kansdichtheidsfunctie, behorende bij de afzonderlijke kwantisatieniveaus allen even groot moeten zijn.

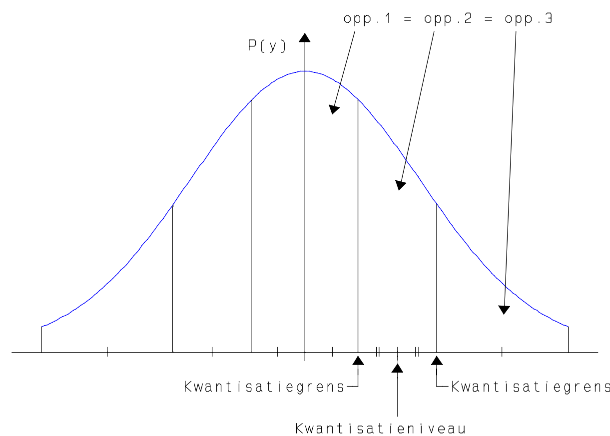


Fig. 1.7: Kansverdeling met non-uniforme intervallen.

1.3 De H.261 codec

De standaard waarin het basis codeerschema van de tweede generatie beeldtelefoons is vastgelegd is de H.261 aanbeveling. Een schematisch overzicht hiervan is getekend in fig. 1.9.
Voornamelijk de ontvanger is duidelijk omschreven. Als logisch gevolg moet bij het ontwerpen de zender hier dus aan worden aangepast. Op de vakgroep Informatietheorie is een softwaremodel van een zender en ontvanger volgens de H.261 specificaties ontwikkeld. De zender en ontvanger zullen achtereenvolgens in de volgende paragrafen worden besproken.

De CIF beelden waarmee de beeldtelefoon werkt bestaan uit 288 beeldlijnen met elk 352 pixels. In het CIF wordt voor elke pixel het luminantieniveau (Y) vastgelegd. De kleurniveaus (U en V) worden echter gemiddeld over 4 pixels. Dit betekent dat de U en V componenten een kwart van de resolutie hebben in vergelijking tot de Y component.
In de codec wordt dit CIF beeld opgedeeld in macroblokken (zie fig. 1.8). Elk macroblok bestaat uit een blok van 16 bij 16 pixels met het Y signaal, een blok van 8 bij 8 pixels met het U signaal en een blok van 8 bij 8 pixels met het V signaal. Het blok met het Y signaal wordt vervolgens opgedeeld in 4 blokken van 8 bij 8 pixels. In totaal bestaat een macroblok dus uit 6 blokken van 8 bij 8 pixels. Zo'n blok van 8 bij 8 pixels noemen we een fragment.

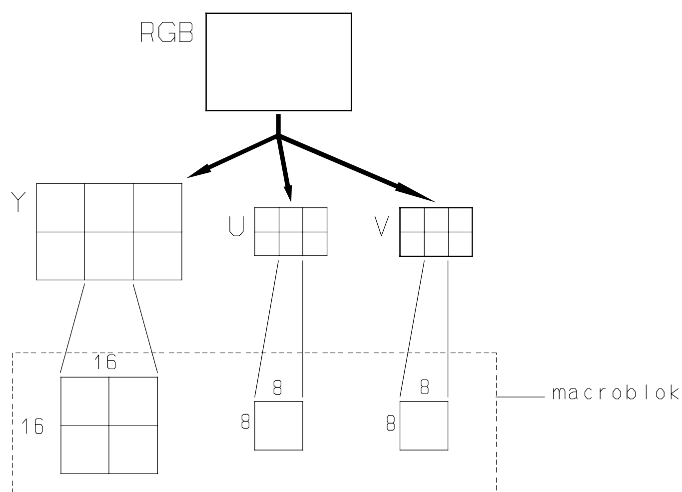


Fig. 1.8: Beeld opgedeeld in macroblokken.

1.3.1 Zender (coder)

In fig 1.9 is de zender schematisch weergegeven. In de zender wordt de DCT gebruikt om de spatiële redundantie te verminderen. De zender bevat ook een terugkoppellus om de temporele redundantie te verminderen. Deze terugkoppellus kan worden uitgeschakeld.

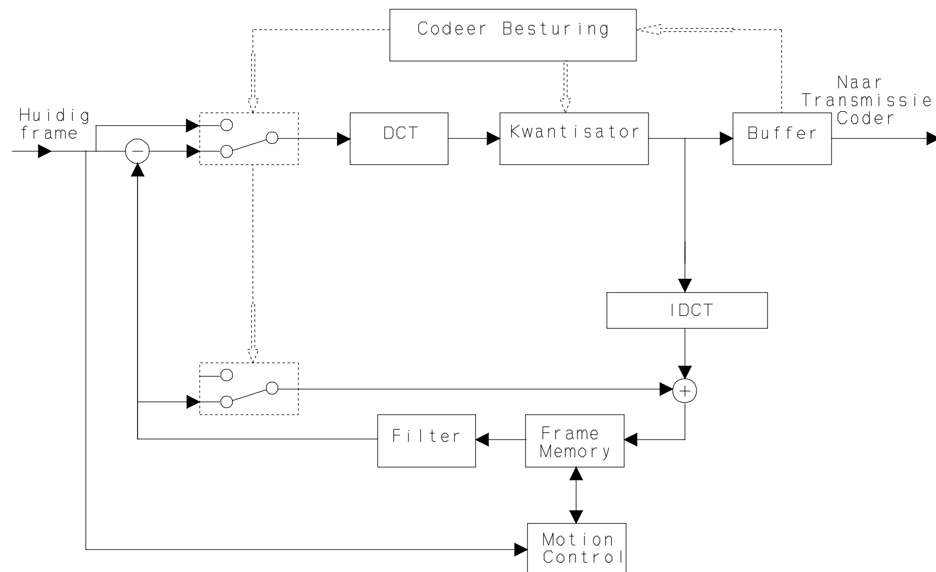


Fig. 1.9: Schematische weergave van de H.261 beeldtelefoon (zender).

In het bovenste gedeelte van het schema zit de DCT die het aangeboden fragment decorreleerd en daarna doorgeeft aan de kwantisator. De DCT-coëfficiënten worden gekwantiseerd naar 255 uniforme niveaus. Voor elke coëfficiënt wordt uiteindelijk het tabeladres van de waarde waarnaar de coëfficiënt wordt gekwantiseerd verstuurd.

tabeladres	kwantisatieniveau
00bin	-192
01bin	-64
10bin	64
11bin	192

Voorbeeld van een tabel om kwantisatieniveaus om te zetten in tabeladressen. De tabeladressen kunnen ook een variabele lengte hebben, zoals bij de

Huffman codering.

De zender kan in twee modes werkzaam zijn. In de eerste mode is de bewegingscompensatie uitgeschakeld. Alleen het bovenste gedeelte van het schema wordt dan benut. Er wordt dan alleen spatiële redundantie uit het signaal gehaald met behulp van het transformatieproces.

Zit er weinig beweging in een sequentie beelden ("hoofd-en-schouders" beelden) dan kan ook de temporele redundantie in de sequentie worden aangepakt. De terugkoppellus moet dan worden ingeschakeld. Het verminderen van de temporele redundantie gebeurt met behulp van het temporele predictie mechanisme in de terugkoppellus. Dit mechanisme zorgt ervoor dat de pixelwaarden in een fragment voorspeld worden. Dit wordt gedaan door gebruik te maken van de waarden van het fragment op die positie in het voorgaande beeld. Alleen de predictiefout blijft dan over om te transformeren. Dit principe is gelijk aan dat bij DPCM. Deze vorm van coderen waarbij het verschil tussen het huidige en het vorige beeld wordt gecodeerd heet intercoderen.
Men kan proberen de predictiefout nog verder te verminderen door bewegingscompensatie toe te passen. Als een object in beweging is, kan het in een bepaald macroblok verschijnen terwijl het in het voorgaande beeld in een ander macroblok te zien was. Zou men beide macroblokken gewoon intercoderen, dan zou de predictiefout groot zijn. Het bewegingscompensatie algoritme verkleint in zulke gevallen de predictiefout. Dit gebeurt door het Y fragment van het te coderen macroblok te vergelijken met het Y fragment in het voorgaande beeld. Om niet het gehele beeld af te moeten zoeken en omdat de beweging vaak gering is, wordt het Y fragment alleen vergeleken met het Y fragment in het vorige beeld die in het zoekwindow liggen. Dit zoekwindow is een gebied rondom het fragment dat zou worden gebruikt bij intercodering. Dit zoekproces wordt gedaan met het luminantiesignaal, omdat dit vier keer zoveel informatie bevat als de kleursignalen. Als de informatie niet teveel verschoven is en nauwelijks is veranderd, dan zal het met de zoekprocedure gevonden worden. De verplaatsingsvector wordt dan bepaald. Het blok op de

door de verplaatsingsvector aangegeven positie dient nu als predictor. Aangenomen wordt dat de U en V fragmenten dezelfde beweging hebben gemaakt als de Y fragmenten van het macroblok. Daarom wordt voor het voorspellen van de kleurfragmenten dezelfde verplaatsingsvector gebruikt. De predictiefout kan in dit geval dus weer even klein zijn als bij intercoderen van een macroblok zonder bewegend object. De verplaatsingsvector wordt meegezonden naar de ontvanger.

Door het zoekwindow werkt dit alleen als er weinig beweging in de beelden zit. Als de predictiefout groter is dan de waarden in het blok zelf is deze methode onbetrouwbaar geworden. Er moet dan weer even worden overgestapt op de directe transformatie van het beeld zonder bewegingscompensatie. Deze vorm van coderen waarbij geen gebruik wordt gemaakt van het voorgaande beeld noemt men intracoderen. Intracoderen wordt bij elk blok eens in de 132 keer gedwongen toegepast. Dit is om foutpropagatie en eventuele kanaalfouten tegen te gaan.

De verschilfragmenten van 8 bij 8 worden nu aangeboden aan het transformatieproces. Nadat de gegevens zijn getransformeerd worden de DCT-coëfficiënten gekwantiseerd in de kwantisator. In de terugkoppellus zit een inverse DCT.

In de codec wordt een uniforme kwantisator gebruikt met variabele stapgrootte. Dit betekent dat per macroblok de kwantisatie uniform is, maar dat de stapgrootte wel kan verschillen tussen macroblokken. Dit is om de volgende reden gedaan. Als met een variabele lengte codering wordt gewerkt levert niet elk fragment evenveel te verzenden bits. De buffer dient om een constante transmissiesnelheid op de lijn te krijgen. Als de buffer leeg dreigt te raken maakt deze dit kenbaar aan de kwantisator. Door de variabele stapgrootte kan de kwantisator nauwkeuriger gaan kwantiseren. Hierdoor loopt de buffer weer vol. En als de buffer dreigt over te lopen kan onnauwkeuriger gekwantiseerd worden.

De beeldfrequentie van een beeldtelefoon is 30 Hz. In de codec vindt echter een uitwisseling plaats van temporele naar spatiële resolutie, zodat de totale beeldkwaliteit er op vooruit gaat. Dit wil zeggen dat door de temporele resolutie te verlagen, minder beelden per seconde, de spatiële resolute verhoogd kan worden bij gelijkblijvende transmissiesnelheid. De beeldfrequentie kan dan variëren van 7,5 tot 30 Hz, maar wordt meestal ingesteld op 10 Hz. Hierdoor kan wel onderbemonstering optreden (schokkerige beelden), omdat maar één op de drie beelden verstuurd wordt.

1.3.2 Ontvanger (decoder)

In de ontvanger worden de macroblokken ontvangen in de buffer. Vervolgens wordt door middel van het inverse kwantisatie- en transformatieproces, het beeld weer gereconstrueerd. Dit reconstructieproces is identiek aan dat in de zender. Fig. 1.10 laat het blokschema van de ontvanger zien.

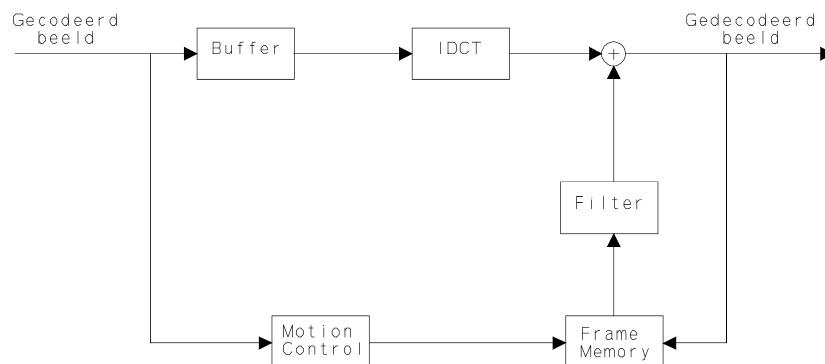


Fig. 1.10: Blokschema van de ontvanger.

Uiteindelijk wordt het beeld van CIF weer omgezet naar een RGB signaal dat is weer te geven op een beeldscherm. Als er gebruik gemaakt is van een beeldfrequentie lager dan 30 Hz, wordt bij gebrek aan een nieuw beeld, het oude beeld herhaald.

2. Beeldcodering m.b.v. Spatiële Interpolatie Piramides

De door ons onderzochte methode voor beeldcompressie is de CTPI methode (Component Transformatie met Pixel Interpolatie). Deze methode maakt gebruik van pixelinterpolatie en het kwantiseren van de verschillen tussen de geïnterpoleerde en de werkelijke pixelwaarde.
CTPI is in tegenstelling tot andere methoden zoals de DCT eerder een interpolatie coderingsmethode dan een methode voor het transformeren van code. Omdat deze vlag de lading beter dekt spreken wij liever van codering met behulp van Spatiële Interpolatie Piramides (SIP). De SIP methode kan worden gebruikt voor zowel opslag van grijswaarde- en kleurenbeelden als het transporteren van zulke beelden door een smalbandig kanaal.

Allereerst zal worden uitgelegd hoe SIP coderen gebeurd en hoe het aantal benodigde bits per pixel kan worden bepaald. Daarna hoe de kwantisatiefout bepaald kan worden en hoe de SIP methode op kleurenbeelden is toe te passen. Tenslotte wordt ingegaan op hoe deze codeermethode werkt aan de randen van een te coderen beeld.

2.1 Het codeerschema voor grijswaardebeelden

In deze paragraaf zullen de stappen worden uitgelegd die nodig zijn om een grijswaardebeeld volgens de SIP methode te coderen en te decoderen.

1) Het originele beeld wordt opgedeeld in vierkanten. Zo'n vierkant gedeelte van een beeld noemen we een fragment. Fig. 2.1 laat een voorbeeld zien van hoe een in fragmenten opgedeeld beeld eruit kan komen te zien.

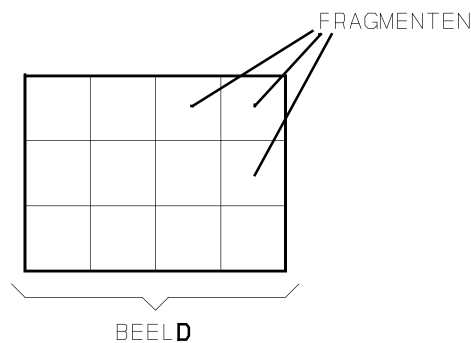


Fig. 2.1: Beeld opgedeeld in fragmenten.

Het coderen gebeurt door alle fragmenten één voor één te coderen. Alle gecodeerde fragmenten bij elkaar vormen het gecodeerde beeld. Dit opdelen van het originele beeld gebeurt omdat alleen pixels die bij elkaar in de buurt liggen gecorreleerd zijn.

2) Noem het fragment dat aan de beurt is om gecodeerd te worden fragment I_0. Voor SIP codering is het nodig dat van dit fragment de zogenaamde componenten worden bepaald. Zo'n component bevat bepaalde pixels van het fragment en geeft het fragment weer met een lagere resolutie. Het aantal componenten waarin een fragment wordt opgedeeld is N. Het volgnummer van de componenten is k, zodat $k = 1,2,...,N$. Fig. 2.2 laat een fragment en diens componenten zien voor N=3. De cijfers zijn de volgnummers van de pixels binnen het fragment. Ook bij de verdere voorbeelden zal steeds gelden dat N=3. De k^{de}

component heet I_k en bestaat uit de pixels $X[2^k \cdot i, 2^k \cdot j]$ van het originele fragment I_0. Hierin is i het volgnummer van de kolommen en j het volgnummer van de rijen van het fragment. Om van een fragment de N componenten te kunnen bepalen moet het fragment een hoogte en een breedte van 2^N+1 pixels hebben.

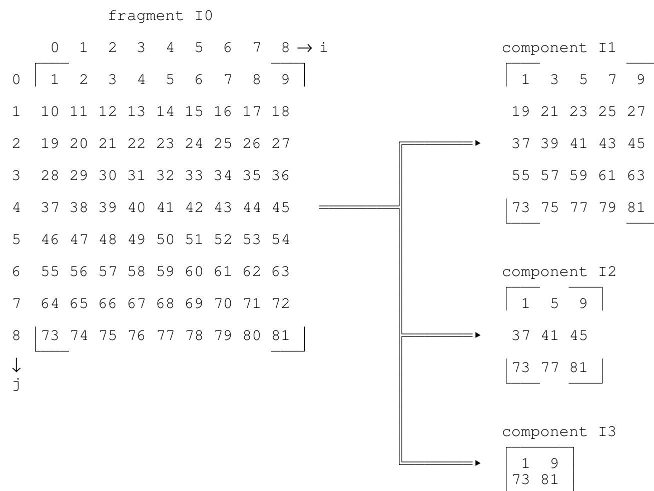


Fig. 2.2: Voorbeeld van het opdelen van een fragment in componenten.

3) De pixels van de componenten worden vervolgens lineair geïnterpoleerd. Hierdoor ontstaan uit component I_k de geïnterpoleerdecomponent I_k. Fig. 2.3 laat de componenten en bijbehorende geïnterpoleerde componenten zien.

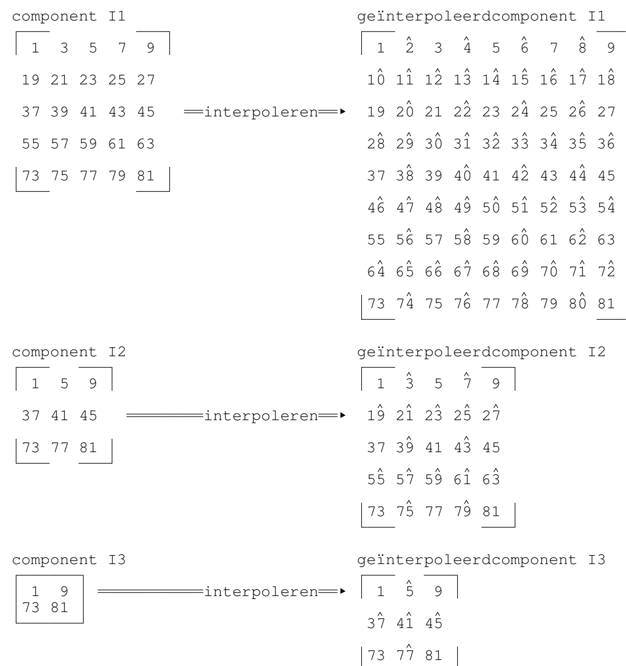


Fig. 2.3: Het bepalen van de geïnterpoleerde componenten.

Het teken "^" geeft aan dat het om een geïnterpoleerde pixel gaat.

vb: $\hat{5}=(1+9)/2$ $3\hat{7}=(1+73)/2$ $4\hat{1}=(1+9+73+81)/4$

Hierna worden de verschillen tussen de componenten I_{k-1} en de geïnterpoleerde componenten I_k bepaald. Hierdoor ontstaan de verschilcomponenten I_{k-1}-I_k zoals die in fig 2.4 zijn weergegeven.

fragment I0 - geïnterpoleerdcomponent I1 = verschilcomponent I0-I1

1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18
19	20	21	22	23	24	25	26	27
28	29	30	31	32	33	34	35	36
37	38	39	40	41	42	43	44	45
46	47	48	49	50	51	52	53	54
55	56	57	58	59	60	61	62	63
64	65	66	67	68	69	70	71	72
73	74	75	76	77	78	79	80	81

1	$\hat{2}$	3	$\hat{4}$	5	$\hat{6}$	7	$\hat{8}$	9
$1\hat{0}$	$1\hat{1}$	$1\hat{2}$	$1\hat{3}$	$1\hat{4}$	$1\hat{5}$	$1\hat{6}$	$1\hat{7}$	$1\hat{8}$
19	$2\hat{0}$	21	$2\hat{2}$	23	$2\hat{4}$	25	$2\hat{6}$	27
$2\hat{8}$	$2\hat{9}$	$3\hat{0}$	$3\hat{1}$	$3\hat{2}$	$3\hat{3}$	$3\hat{4}$	$3\hat{5}$	$3\hat{6}$
37	$3\hat{8}$	39	$4\hat{0}$	41	$4\hat{2}$	43	$4\hat{4}$	45
$4\hat{6}$	$4\hat{7}$	$4\hat{8}$	$4\hat{9}$	$5\hat{0}$	$5\hat{1}$	$5\hat{2}$	$5\hat{3}$	$5\hat{4}$
55	$5\hat{6}$	57	$5\hat{8}$	59	$6\hat{0}$	61	$6\hat{2}$	63
$6\hat{4}$	$6\hat{5}$	$6\hat{6}$	$6\hat{7}$	$6\hat{8}$	$6\hat{9}$	$7\hat{0}$	$7\hat{1}$	$7\hat{2}$
73	$7\hat{4}$	75	$7\hat{6}$	77	$7\hat{8}$	79	$8\hat{0}$	81

0	$2-\hat{2}$	0	$4-\hat{4}$	0	$6-\hat{6}$	0	$8-\hat{8}$	0
$10-1\hat{0}$	$11-1\hat{1}$	$12-1\hat{2}$	$13-1\hat{3}$	$14-1\hat{4}$	$15-1\hat{5}$	$16-1\hat{6}$	$17-1\hat{7}$	$18-1\hat{8}$
0	$20-2\hat{0}$	0	$22-2\hat{2}$	0	$24-2\hat{4}$	0	$26-2\hat{6}$	0
$28-2\hat{8}$	$29-2\hat{9}$	$30-3\hat{0}$	$31-3\hat{1}$	$32-3\hat{2}$	$33-3\hat{3}$	$34-3\hat{4}$	$35-3\hat{5}$	$36-3\hat{6}$
0	$38-3\hat{8}$	0	$40-4\hat{0}$	0	$42-4\hat{2}$	0	$44-4\hat{4}$	0
$46-4\hat{6}$	$47-4\hat{7}$	$48-4\hat{8}$	$49-4\hat{9}$	$50-5\hat{0}$	$51-5\hat{1}$	$52-5\hat{2}$	$53-5\hat{3}$	$54-5\hat{4}$
0	$56-5\hat{6}$	0	$58-5\hat{8}$	0	$60-6\hat{0}$	0	$62-6\hat{2}$	0
$64-6\hat{4}$	$65-6\hat{5}$	$66-6\hat{6}$	$67-6\hat{7}$	$68-6\hat{8}$	$69-6\hat{9}$	$70-7\hat{0}$	$71-7\hat{1}$	$72-7\hat{2}$
0	$74-7\hat{4}$	0	$76-7\hat{6}$	0	$78-7\hat{8}$	0	$80-8\hat{0}$	0

component I1 - geïnterpoleerdcomponent I2 = verschilcomponent I1-I2

1	3	5	7	9
19	21	23	25	27
37	39	41	43	45
55	57	59	61	63
73	75	77	79	81

1	$\hat{3}$	5	$\hat{7}$	9
$1\hat{9}$	$2\hat{1}$	$2\hat{3}$	$2\hat{5}$	$2\hat{7}$
37	$3\hat{9}$	41	$4\hat{3}$	45
$5\hat{5}$	$5\hat{7}$	$5\hat{9}$	$6\hat{1}$	$6\hat{3}$
73	$7\hat{5}$	77	$7\hat{9}$	81

0	$3-\hat{3}$	0	$7-\hat{7}$	0
$19-1\hat{9}$	$21-2\hat{1}$	$23-2\hat{3}$	$25-2\hat{5}$	$27-2\hat{7}$
0	$39-3\hat{9}$	0	$43-4\hat{3}$	0
$55-5\hat{5}$	$57-5\hat{7}$	$59-5\hat{9}$	$61-6\hat{1}$	$63-6\hat{3}$
0	$75-7\hat{5}$	0	$79-7\hat{9}$	0

component I2 - geïnterpoleerdcomponent I3 = verschilcomponent I2-I3

1	5	9
37	41	45
73	77	81

1	$\hat{5}$	9
$3\hat{7}$	$4\hat{1}$	$4\hat{5}$
73	$7\hat{7}$	81

0	$5-\hat{5}$	0
$37-3\hat{7}$	$41-4\hat{1}$	$45-4\hat{5}$
0	$77-7\hat{7}$	0

Fig. 2.4: Het bepalen van de verschilcomponenten.

4) De pixels van component I_N worden met m_N niveaus gekwantiseerd en de pixels van de verschilcomponenten I_{k-1}-I_k worden met m_{k-1} niveaus gekwantiseerd. Met k = 1,2,...,N.

5) De resultaten van het kwantiseren worden gecodeerd. Daarna wordt de spatiële interpolatie piramide opgeslagen of getransporteerd door een communicatiekanaal. Fig. 2.5 laat de spatiële interpolatie piramide zien.

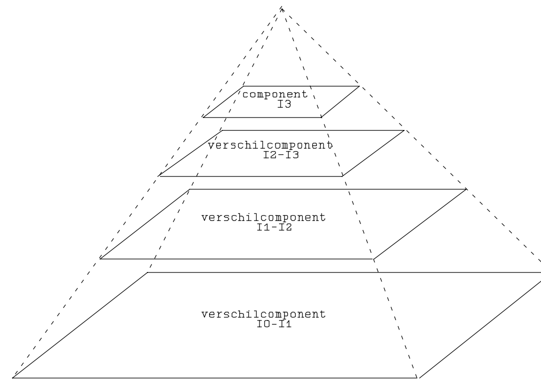


Fig. 2.5: Spatiële Interpolatie Piramide.

6) De codes ontvangen via een communicatiekanaal of opgehaald uit het geheugen worden gebruikt voor het decoderen van de bemonsterde pixels. Op deze manier kunnen, op de kwantisatieruis na, de component I_N en alle verschilcomponenten worden teruggewonnen.
7) Component I_N wordt lineair geïnterpoleerd, net als bij 2). Verschilcomponent I_{k-1}-I_k wordt dan hierbij opgeteld. Het resultaat is de component I_{k-1}. Deze interpolatie en optelprocedure wordt N maal herhaald; hierdoor worden achtereenvolgens ook de componenten I_{N-2},I_{N-3},...,I_1 en fragment I0 hersteld. In fig 2.6 is dit grafisch weergegeven.

Het is waarschijnlijk dat de grootte van de pixelwaarden van de pixels in verschilcomponent I0-I1 zeer klein zijn. Daarom zijn er maar weinig niveaus nodig om deze pixels te kwantiseren. De waarden van de pixels van I1-I2 zullen waarschijnlijk meer variëren, zodat hier meer kwantisatieniveaus nodig zullen zijn. De verschilcomponenten zullen dus steeds met een ander (groter) aantal niveaus worden bemonsterd.
Bij stap 4) werd al verteld dat de verschilcomponenten I_{k-1}-I_k met m_{k-1} niveaus en component I_N met m_N niveaus worden gekwantiseerd. Met N=3 geldt dan dat:
Verschilcomponent I0-I1 wordt met m_0 niveaus bemonsterd.
Verschilcomponent I1-I2 wordt met m_1 niveaus bemonsterd.
Verschilcomponent I2-I3 wordt met m_2 niveaus bemonsterd.
Component I3 wordt met m_3 niveaus bemonsterd.

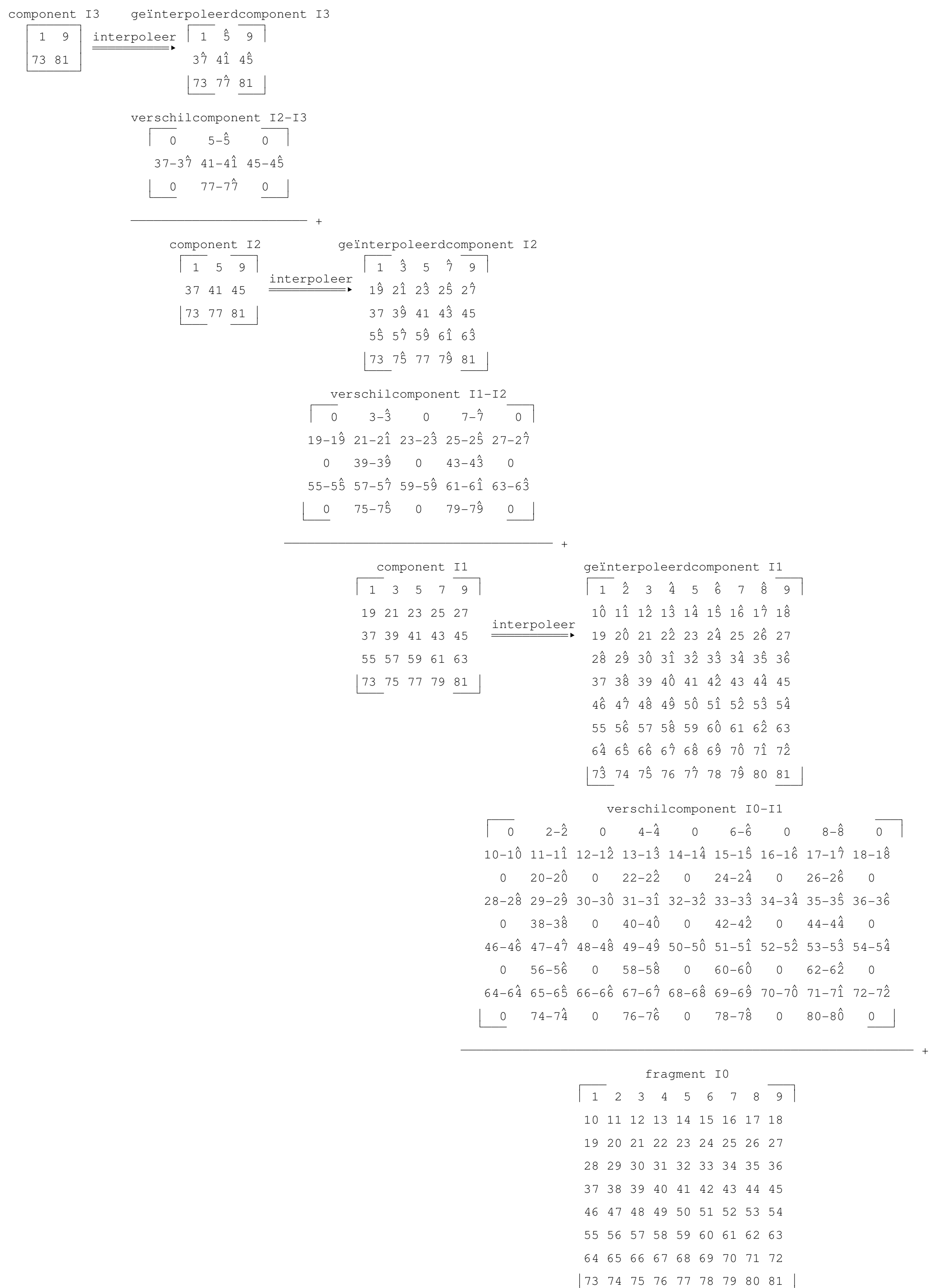


Fig. 2.6: Het decoderen van een fragment.

De pixels in component I3 hebben dezelfde waarde als in het originele fragment en zullen dan ook met even veel bits gecodeerd worden als in het origineel beeld, voor een beeld met 256 grijswaarden is dit $^{2}\log(256)=8$ bits. De nu volgende figuur laat zien in welke component elk pixel van een fragment terecht komt.

```
 I3    I0-I1  I1-I2  I0-I1  I2-I3  I0-I1  I1-I2  I0-I1   I3
I0-I1  I0-I1  I0-I1  I0-I1  I0-I1  I0-I1  I0-I1  I0-I1  I0-I1
I1-I2  I0-I1  I1-I2  I0-I1  I1-I2  I0-I1  I1-I2  I0-I1  I1-I2
I0-I1  I0-I1  I0-I1  I0-I1  I0-I1  I0-I1  I0-I1  I0-I1  I0-I1
I2-I3  I0-I1  I1-I2  I0-I1  I2-I3  I0-I1  I1-I2  I0-I1  I2-I3
I0-I1  I0-I1  I0-I1  I0-I1  I0-I1  I0-I1  I0-I1  I0-I1  I0-I1
I1-I2  I0-I1  I1-I2  I0-I1  I1-I2  I0-I1  I1-I2  I0-I1  I1-I2
I0-I1  I0-I1  I0-I1  I0-I1  I0-I1  I0-I1  I0-I1  I0-I1  I0-I1
 I3    I0-I1  I1-I2  I0-I1  I2-I3  I0-I1  I1-I2  I0-I1   I3
```

Als alle fragmenten zó gecodeerd zouden worden zou een overgang tussen de fragmenten er als volgt uit zien:

```
  I0-I1  I1-I2 | I1-I2  I0-I1
  I0-I1  I0-I1 | I0-I1  I0-I1
  I0-I1   I3   |  I3    I0-I1
- - - - - - - -+- - - - - - - -
  I0-I1   I3   |  I3    I0-I1
  I0-I1  I0-I1 | I0-I1  I0-I1
  I0-I1  I1-I2 | I1-I2  I0-I1
```

Op deze manier komen er dus vier I3 pixels (welke 8 bit/pixel kosten) bij elkaar terecht. Dit geeft zeer lokaal een zeer hoge kwaliteit en dit kost veel bits.

Beter is het daarom om de fragmenten als het ware één kolom en één rij pixels te laten overlappen en van een fragment van 9 bij 9 pixels een blok van 8 bij 8 pixels te coderen. Een fragment en de overgang tussen fragmenten ziet er dan als volgt uit:

```
 I3    I0-I1  I1-I2  I0-I1  I2-I3  I0-I1  I1-I2  I0-I1 |  I3
I0-I1  I0-I1  I0-I1  I0-I1  I0-I1  I0-I1  I0-I1  I0-I1 | I0-I1
I1-I2  I0-I1  I1-I2  I0-I1  I1-I2  I0-I1  I1-I2  I0-I1 | I1-I2
I0-I1  I0-I1  I0-I1  I0-I1  I0-I1  I0-I1  I0-I1  I0-I1 | I0-I1
I2-I3  I0-I1  I1-I2  I0-I1  I2-I3  I0-I1  I1-I2  I0-I1 | I2-I3
I0-I1  I0-I1  I0-I1  I0-I1  I0-I1  I0-I1  I0-I1  I0-I1 | I0-I1
I1-I2  I0-I1  I1-I2  I0-I1  I1-I2  I0-I1  I1-I2  I0-I1 | I1-I2
I0-I1  I0-I1  I0-I1  I0-I1  I0-I1  I0-I1  I0-I1  I0-I1 | I0-I1
- - - - - - - - - - - - - - - - - - - - - - - - - - - -+- - - -
 I3    I0-I1  I1-I2  I0-I1  I2-I3  I0-I1  I1-I2  I0-I1 |  I3
```

Van het fragment hoeven nu alleen de pixels boven èn links van de lijnen gecodeerd te worden. De kolom rechts van de lijn is de linker kolom van het ernaast liggende fragment en wordt dan ook pas bij dat fragment gecodeerd. Evenzo is de rij onder de lijn de bovenste rij van het onderliggende fragment en wordt pas bij dat fragment gecodeerd. Deze lijn en kolom zijn nog wel nodig voor de berekeningen aan het huidige fragment. In fig. 2.7 is te zien hoe in dit geval een in fragmenten opgedeeld beeld eruit kan komen te zien.

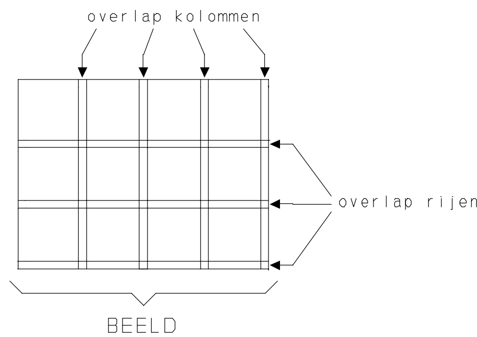


Fig. 2.7: Beeld opgedeeld in overlappende fragmenten.

Bij de fragment overgangen is er nu geen concentratie van I3 pixels meer. Door deze overlap heeft de SIP coderingsmethode geen last van het blokeffect. Dit wil zeggen dat de randen van de fragmenten niet zichtbaar zijn in het gedecodeerde beeld. Dit in tegenstelling tot transformatie methoden waarbij geen overlap wordt gebruikt, zoals de DCT.

2.2 Bepaling van de codelengte

Nu zal er worden nagegaan hoe de codelengte bij SIP codering kan worden bepaald. Van de 64 te coderen pixels in een fragment komen er 48 in verschilcomponent I0-I1 terecht,
$48/64 = (3/4)*(1/1) = (3/4)*(1/4^0)$.
Van de 64 pixels komen er 12 in verschilcomponent I1-I2 terecht,
$12/64 = (3/4)*(1/4) = (3/4)*(1/4^1)$
Van de 64 pixels komen er 3 in verschilcomponent I2-I3 terecht,
$3/64 = (3/4)*(1/16)= (3/4)*(1/4^2)$
Van de 64 pixels komt er 1 in component I3 terecht,
$1/64 = (1/4^3)$
Voor een verschilcomponent I_{k-1}-I_k geldt dus in het algemeen dat er $(3/4)*(1/4^{k-1})*{}^2\log(m_{k-1})$ bit/pixel nodig zijn om het te coderen. De factor 3/4 wordt veroorzaakt doordat van elk verschilcomponent steeds 1/4 van de pixels nul is en er dus maar 3/4 van alle pixels in een verschilcomponent gekwantiseerd hoeft te worden. Het aantal bit/pixel nodig om alle verschilcomponenten te coderen is dan:

$$\frac{3}{4} \cdot \sum_{k=1}^{N} \left\{ \frac{1}{4^{k-1}} \cdot {}^2\log(m_{k-1}) \right\} \qquad \mathit{bit/pixel} \tag{1}$$

Van I_N moet alles (dit is één pixel) gecodeerd worden, hiervoor zijn dus $(1/4^N)*{}^2\log(m_N)$ bit/pixel nodig.
Om een beeld op deze manier te coderen zijn dus nodig:

$$\frac{3}{4} \cdot \sum_{k=1}^{N} \left\{ \frac{1}{4^{k-1}} \cdot {}^2\log(m_{k-1}) \right\} + \frac{1}{4^N} \cdot {}^2\log(m_N) \qquad \mathit{bit/pixel} \tag{2}$$

2.3 Foutbepaling

In de paragraaf over kwantisatie is bepaald hoe de minimale kwantiesatiefout bepaald kan worden.

$$\sigma^2_{min} = \frac{1}{12 \cdot (2n+1)^2} \cdot \left(\int_{-V}^{V} P^{1/3}(y)\ dy \right)^3 \tag{3}$$

Hierin is de factor (2n+1) het aantal kwantisatieniveaus. We nemen aan dat de fout in de gedecodeerde component I_k een kleine invloed heeft op de fout in de gedecodeerde component I_{k-1}. In dat geval kan deze formule gebruikt worden voor het bepalen van de bijdrage die elke verschilcomponent levert aan de minimale fout bij SIP codering. Door in de formule voor het aantal kwantisatieniveaus m_{k-1} in te vullen en rekening te houden met welk gedeelte van de pixels in welke verschilcomponent terecht komt kan de formule worden verkregen waarmee de minimale vervorming bij SIP codering kan worden bepaald.

$$\sigma^2_{min} = \frac{3}{4} \cdot \frac{1}{4^{k-1}} \cdot \frac{1}{12 \cdot (m_{k-1})^2} \cdot \left(\int_{-V}^{V} (P_{k-1}(y))^{1/3}\ dy \right)^3 \tag{4}$$

met

$$E_{k-1} = \left(\int_{-V}^{V} (P_{k-1}(y))^{1/3}\ dy \right)^3 \tag{5}$$

wordt dit

$$\sigma^2_{min} = \frac{E_{k-1}}{16 \cdot 4^{k-1} \cdot (m_{k-1})^2} \tag{6}$$

De gekwantiseerde waarde heeft dus gemiddeld een afwijking van de originele waarde van:

$$\sigma^2_{min} = \frac{1}{16} \cdot \sum_{k=1}^{N} \frac{E_{k-1}}{4^{k-1} \cdot (m_{k-1})^2} \qquad (7)$$

2.4 Het SIP coderen van kleurenbeelden

Een kleurenbeeld bestaat uit een rood, een groen en een blauw beeld. Deze worden eerst omgezet naar een helderheidsbeeld Y en kleurverschilbeelden (R-Y) en (B-Y). Hierna wordt voor het coderen van het helderheidsbeeld hetzelfde codeerschema gebruikt als voor het coderen van grijswaardebeelden werd gebruikt. Ook de kleurverschilbeelden worden op deze manier gecodeerd, met dit verschil dat alle waarden in de verschilcomponenten I0-I1 gelijk aan nul worden verondersteld. Hierdoor hoeven deze niet te worden opgeslagen of verzonden. Dit mag omdat het menselijk oog voor kleurvariatie veel ongevoeliger is dan voor variatie in de helderheid. De codelengte in bit/pixel die nodig is voor een kleurenbeeld wordt op deze manier:

$$\begin{aligned} N_{kleur} &= N + N_{R-Y} + N_{B-Y} \\ &= N + (3/16 \cdot {}^{2}\log(m_1R) + 3/64 \cdot {}^{2}\log(m_2R) + 1/8) \\ &\quad + (3/16 \cdot {}^{2}\log(m_1B) + 3/64 \cdot {}^{2}\log(m_2B) + 1/8) \end{aligned}$$

Hierin is N de codelengte van de helderheid Y welke gelijk is aan die van een grijswaardebeelden. $m_{k-1}R$ en $m_{k-1}B$ zijn de aantallen kwantisatieniveaus die nodig zijn voor het representeren van de $(k\text{-}1)^{de}$ verschilcomponent van respectievelijk (R-Y) en (B-Y).

2.5 Codering aan de beeldranden

Zoals aan het einde van paragraaf 2.3 al werd gezegd wordt de meest rechtse

kolom in een fragment pas gecodeerd bij het coderen van de meest linkse kolom van het ernaast gelegen fragment. Evenzo wordt de onderste rij pas gecodeerd bij het coderen van de bovenste rij van het eronder gelegen fragment. Bij de fragmenten aan de rechterrand en de onderrand van het beeld is er echter geen ernaast gelegen of eronder gelegen fragment.
Bij deze fragmenten moet er dus rekening mee gehouden worden dat hier de laatste kolom of de laatste rij wel gecodeerd wordt.

Zoals gezegd in stap 2 in paragraaf 2.2 moet een fragment een hoogte en breedte van 2^N+1 pixels hebben. Om een beeld SIP te kunnen coderen moeten de hoogte en de breedte van het beeld aan de volgende voorwaarden voldoen. De breedte moet $(b*2^N)+1$ pixels groot zijn en de hoogte moet $(h*2^N)+1$ pixels groot zijn. Hierin zijn zowel b als h een geheel getal groter dan nul.
Als de afmetingen van een beeld niet aan deze voorwaarden voldoen moet het beeld opgerekt worden tot de afmetingen wel aan deze voorwaarden voldoen. Dit oprekken gebeurt in horizontale richting door het beeld aan de rechterkant aan te vullen met kolommen waarvan de pixels dezelfde waarde hebben als de pixels in de laatste kolom van het originele beeld. Oprekken van het beeld in verticale richting gebeurt door het beeld aan de onderkant aan te vullen met rijen waarvan de pixels dezelfde waarde hebben als de pixels in de onderste rij van het originele beeld. Een fragment aan de onderkant van een opgerekt beeld zou eruit kunnen zien zoals hieronder is weergegeven. De getallen geven nu niet het volgnummer van de pixels binnen het fragment aan, maar de waarde van de pixels.

23	24	27	30	45	76	80	81	138	< originele rijen
24	25	26	29	42	68	82	94	127	<
25	26	32	40	42	63	79	88	125	<
26	28	35	44	50	65	78	87	126	<
26	28	35	44	50	65	78	87	126	< rijen ontstaan door oprekken
26	28	35	44	50	65	78	87	126	<
26	28	35	44	50	65	78	87	126	<
26	28	35	44	50	65	78	87	126	<
26	28	35	44	50	65	78	87	126	<

Als oprekken nodig is wordt het te coderen beeld groter. Hierdoor zal er meer data verzonden moeten worden dan wanneer oprekken van het beeld niet

nodig zou zijn. Om deze data toename beperkt te houden moeten de afmetingen van een fragment niet te groot gekozen worden. Maar wel groot genoeg om van het fragment een aantal componenten te kunnen bepalen. De in de voorbeelden voor N gebruikte waarde 3 is dan een geschikte waarde.
Na het decoderen moeten de extra kolommen en/of rijen die zijn ontstaan bij het oprekken weer worden verwijderd. Hiervoor moeten de afmetingen van het originele beeld aan de decodeerzijde bekend zijn.

3. Software implementatie

Het H.261 simulatie programma is geschreven in C en draait op het SUN netwerk van de TU Delft.
Onze doelstelling was eerst om het russische programma zodanig aan te passen zodat het in de H.261 simulator kon worden gezet. De SIP methode zou dan de plaats innemen van de DCT. Zo kon dan een vergelijking worden gemaakt tussen DCT en SIP algoritme. Dit bleek niet haalbaar omdat de H.261 simulator niet voldoende modulariteit bezit. Ook zat het russische programma vol met inline assembly instructies. Dit zorgt ervoor dat een programma niet portable is en alleen geschikt is voor de machine waar het op ontwikkeld is, de PC. Onze uiteindelijke doelstelling werd toen het maken van een uitgeklede stand-alone versie van de H.261 standaard met daarin verwerkt het SIP algoritme. Dit is uiteindelijk ook gerealiseerd. De listing van de stand-alone versie is te vinden in bijlage F. Om de programma's universeel bruikbaar te maken is alleen ANSI-C gebruikt.
In de nu volgende paragrafen een bespreking van dit programma.

3.1 Globaal

Door de hierboven genoemde problemen hebben wij een vereenvoudigde simulator van de H.261 standaard gemaakt. In deze versie zijn weggelaten de bewegingsschatter en de buffer. Een soortgelijk blokschema als bij de H.261

standaard valt dan op te stellen voor de SIP implementatie. Dit blokschema is te zien in fig. 3.1.

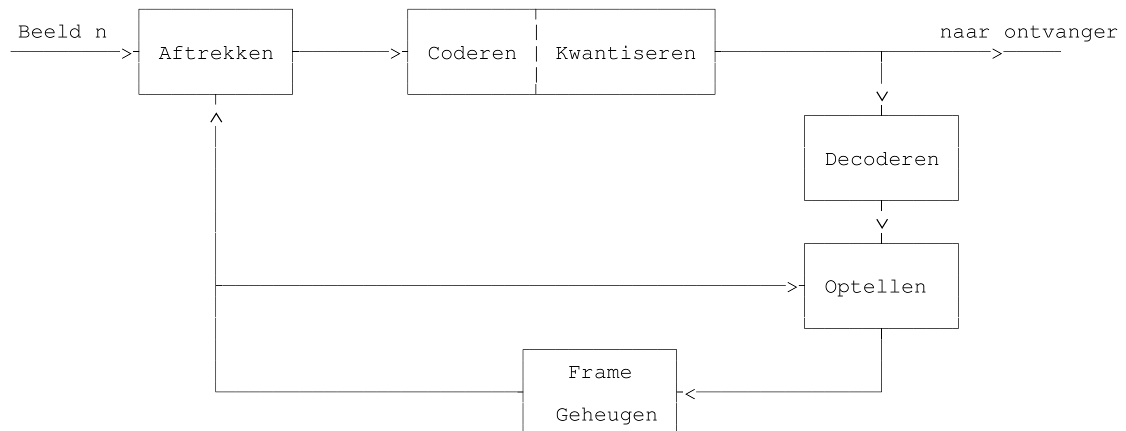


Fig. 3.1: Blokschema van de software implementatie.

Zoals te zien is in het blokschema wordt ook hier een verschilbeeld gecodeerd. Dit verschilbeeld wordt bepaald uit het huidige aangeboden beeld en het gedecodeerde voorgaande beeld dat zich in het frame geheugen bevind.
Het verkregen verschilbeeld wordt vervolgens met het SIP algoritme gecodeerd. Na het coderen moeten de waarden gekwantiseerd worden. Dit staat in hetzelfde blok als het coderen omdat dit eenvoudiger is als het gelijk met het coderen gebeurd. Na het kwantiseren zouden de waarden nog vertaald moeten worden in tabeladressen die dan verstuurd zouden worden. Dit is in de H.261 standaard een functie van de buffer. Deze is echter weggelaten omdat de buffer geen invloed mag hebben op onze metingen.
Na de kwantisator volgt de decoder die het originele verschilbeeld op de kwantisatieruis na hersteld. In de opteller wordt uiteindelijk het originele beeld teruggewonnen door het optellen van het verschilbeeld en het vorige beeld.
De laatste drie onderdelen (de decoder, het frame geheugen en de opteller) zijn ook de onderdelen die de ontvanger vormen.

3.2 Bloksgewijze bespreking

Na deze analyse kan het simulatieprogramma in dezelfde blokken ingedeeld worden. Hierbij verricht iedere C functie de taak zoals deze voor de blokken van het blokschema is beschreven.

De Invoer

We gaan er in het programma van uit dat een beeld zowel een hoogte als een breedte heeft van een veelvoud van 8 pixels. Dit mag omdat alle beelden in de beeldtelefonie aan deze voorwaarde voldoen. Dit bespaart de programmacode die nodig zou zijn om de afmetingen van een beeld te bepalen. Deze code zou tijdens onze metingen toch niet gebruikt worden. Voorafgaande aan het eerste beeld wordt een zwart beeld verondersteld. Wij hebben ons beperkt tot grijswaardebeelden. Het is echter ook mogelijk om kleurenbeelden te versturen met de SIP methode.

De Uitvoer

De uitvoer van de kwantisator is het beeld zoals het aan de ontvanger wordt aangeboden. De uitvoer van het optelblok is het beeld zoals dat ook uit de ontvanger komt.

De uitvoer van ieder blok wordt in een file weggeschreven. Dit is gedaan om later de in- en output van ieder blok te kunnen bestuderen.

De Aftrekker

In dit blok wordt van het aangeboden beeld het vorige beeld afgetrokken. Dit voorgande beeld is bewaard in het frame memory dat zich in de terugkoppellus bevind. De invoer van de aftrekker bestaat uit waarden van 0 tot 255 en de uitvoer waarden die kunnen variëren van -255 tot 255.

De Coder/Kwantisator

Hier wordt het beeld in delen van 8 bij 8 pixels gecodeerd. Hiervoor zijn echter fragmenten van 9 bij 9 pixels nodig. Deze fragmenten worden gemaakt door per keer een balk van negen beeldlijnen in te lezen. Zo'n balk kan dan in stukken van negen pixels breed gecodeerd worden. Hoewel we een fragment van 9 bij 9 pixels als invoer hebben komt er uit de fragment coder een blok van 8 bij 8 pixels. Een uitzondering hierop is het laatste fragment in een balk. Bij het laatste fragment in een balk moeten er negen pixelkolommen worden gecodeerd. Omdat de breedte van het beeld een veelvoud van acht pixels is (aanname) moet in dit laatste fragment een extra kolom worden gemaakt. Dit wordt gedaan door de achtste kolom naar de negende kolom te kopiëren. Ook in de verticale richting treedt dit op. Hier wordt in de laatste balk de achtste beeldlijn naar de negende beeldlijn gekopieerd.

Nadat de eerste keer een balk is gemaakt (uit negen beeldlijnen) wordt de negende beeldlijn gekopieerd naar de eerste en nog maar acht lijnen ingelezen, twee t/m negen. Zo ontstaat de overlap.

In de fragment coder is de volgorde waarin de gecodeerde pixels worden bepaald niet belangrijk. Dit is omdat ze bepaald worden uit de originele pixels.

De kwantisator zit geïntegreerd bij de fragment coder. De kwantisator filtert alle pixels die tot dezelfde component behoren eruit en kwantiseert deze vervolgens met de bij de component behorende kwantisator waarden.

De uitvoer van de coder/kwantisator bestaat uit de integer waarden die in de kwantisator zijn vastgelegd.

De Decoder

De decoder werkt volgens hetzelfde principe als de coder. Ook hier wordt de eerste keer een balk van negen beeldlijnen ingelezen. Om vervolgens fragment voor fragment gedecodeerd te worden. Ook hier is een invoer fragment ook 9 bij 9 pixels en een gedecodeerd fragment 8 bij 8 pixels. Hier is het natuurlijk zo dat ook het laatste fragment in een

balk een uitvoer van 8 bij 8 pixels geeft. Hierdoor wordt de extra kolom die bij het coderen is ontstaan weer weggegooid.

De volgorde waarin de pixels van een fragment gedecodeerd worden is wel van belang. Om een component te kunnen decoderen moet de bovenliggende component uit de interpolatie piramide al gedecodeerd zijn. Dit wil zeggen dat voordat component I1-I2 gedecodeerd kan worden component I2-I3 gedecodeerd moet zijn en dat voordat component I0-I1 gedecodeerd kan worden component I1-I2 gedecodeerd moet zijn.

De Opteller

Hier wordt het originele beeld weer teruggewonnen door het gedecodeerde verschilbeeld op te tellen bij het vorige teruggewonnen beeld. Hier worden ook de waarden die ontstaan na het optellen eventueel geclipt.

Meetprogramma

Voor het meten van de Mean Square Error hebben we een programma geschreven. De MSE wordt met de volgende formule bepaald.

$$MSE = \frac{\sum_{rij=1}^{hoogte} \cdot \sum_{kolom=1}^{breedte} \left| y_{rij,kolom} - x_{rij,kolom} \right|^2}{hoogte \cdot breedte} \tag{1}$$

Hierin wordt met y de pixels van het gerestaureerde beeld aangeduid en met x de pixels van het originele beeld. De MSE is dus te bepalen uit het originele beeld en het beeld dat uit de opteller komt.

4. Kwaliteitsaspecten en metingen

Voor het bepalen van de kwaliteit van het na decoderen teruggewonnen beeld zijn kwaliteitsmaten nodig. In dit hoofdstuk zullen een aantal van deze maten beschreven worden. Kwaliteitsmaten kunnen zowel objectief als subjectief zijn. Een objectieve maat wordt gemeten met behulp van een formule en levert een getal op. Een subjectieve maat wordt gemeten met behulp van proefpersonen en levert een oordeel op.

4.1 Objectieve kwaliteitsmeting

De maat voor de objectieve beeldkwaliteit die in de beeldcodering het meest gebruikt wordt is de gemiddelde-kwadratische-fout (MSE). Dit is ook de kwaliteitsmaat die wij voor onze metingen hebben gebruikt. Om inzicht te verschaffen in de relatie tussen beeldkwaliteit en de waarde van de MSE staan in bijlage D een aantal beelden met verschillende MSE waarden. De MSE wordt bepaald met:

$$MSE = \frac{\sum_{rij=1}^{hoogte} \cdot \sum_{kolom=1}^{breedte} \left| y_{rij,kolom} - x_{rij,kolom} \right|^2}{hoogte \cdot breedte} \qquad (1)$$

Hierin zijn hoogte en breedte de afmetingen van het beeld in pixels.
Een andere objectieve maat is de signaal-ruis verhouding (SNR). Deze maat

wordt veel gebruikt bij analoge processen. De signaal-ruis verhouding kan bepaald worden door gebruik te maken van de gemiddelde-kwadratische-fout. De SNR houdt echter rekening met variantie van het ingangssignaal (σ), de gemiddelde-kwadratische-fout doet dit niet. Als MSE de gemiddelde-kwadratische-fout van een fragment is, kan de SNR van een fragment met de volgende formule bepaald worden.

$$SNR = 10 \cdot {}^{10}\log(\sigma^2/MSE) \quad [dB] \tag{2}$$

Een maat die weer van de SNR is afgeleid, maar meer wordt toegepast bij digitale beeldcoderingsprocessen is de produkt-vervormingsfunctie. Voor deze maat is het echter nodig dat de statistische verdeling van de pixelwaarden bekend is. Van DCT-coëfficiënten is het bekend dat deze verdeling in het algemeen gaussisch kan worden gesteld. De produkt-vervormingsfunctie kan dan als volgt bepaald worden.

$$R(MSE) = (1/2) \cdot {}^{2}\log(\sigma^2/MSE) \quad MSE \leq \sigma^2 \tag{3}$$
$$= 0 \quad MSE > \sigma^2$$

Er zijn ook objectieve kwaliteitsmaten waarbij rekening gehouden wordt met subjectieve factoren. In zulke kwaliteitsmaten zit kennis over het menselijk visueel systeem verwerkt. Factoren waarmee rekening wordt gehouden zijn bijvoorbeeld scherpte en vervorming. Met behulp van de kennis van het menselijk visueel systeem kan een weegfactor voor de factoren worden bepaald. Van deze factoren wordt de gewogen som bepaald. Het nadeel van deze kwaliteitsmaten is dat ze slechts geschikt zijn voor een bepaald soort beelden. Voor een beeld zonder scherpe contouren bijvoorbeeld zal onscherpte van het beeld minder storend zijn dan voor een beeld met scherpe contouren.

4.2 Subjectieve kwaliteitsmeting

Bij het bepalen van de subjectieve kwaliteit moet met verschillende factoren rekening worden gehouden. Deze factoren kunnen verdeeld worden in drie

categorieën, de testomgeving, de methode van experimenteren en de manier waarop de resultaten verwerkt worden.

De omgeving waarin het experiment wordt gedaan heeft invloed op de testresultaten. De omgeving leidt de proefpersoon vrijwel altijd af. In het geval dat men wil bepalen hoe de beeldkwaliteit in de gebruiksomgeving wordt ervaren kan dit gewenst zijn. Bij de meeste experimenten is het echter ongewenst dat de proefpersoon wordt afgeleid.
De volgende omgevingsfactoren hebben invloed op de resultaten van een experiment:
- De kijkafstand van de proefpersoon tot het testbeeld. Deze afstand moet 4 tot 8 maal de hoogte van het beeld zijn.
- De achtergrond verlichting.
- Objecten of structuur in de achtergrond.
- De kennis van de proefpersoon. Als de proefpersoon kennis heeft van beeldverwerking dan weet deze waarop gelet moet worden bij het beoordelen van het testbeeld
- Het gezichtsvermogen van de proefpersoon. Dit dient representatief te zijn voor de beoogde gebruikersgroep.

Er zijn verschillende methoden om een subjectief experiment uit te voeren. Welke methode wordt gebruikt is afhankelijk van het doel van het testbeeld van het experiment. Een doel kan bijvoorbeeld het bepalen van de zichtbaar-heidsdrempel zijn. De zichtbaarheidsdrempel is de grens tussen het wel of niet zichtbaar zijn van een afwijking in het testbeeld.
Met de volgende punten moet rekening gehouden worden bij experimenteer-methoden:
- De experimenten moeten een aantal malen worden uitgevoerd. Dit is nodig om de standaard afwijking te minimaliseren. Met standaard afwijking wordt bedoeld dat bij precies hetzelfde experiment een proefpersoon niet altijd hetzelfde oordeelt.
- De volgorde waarin de testbeelden worden aangeboden aan de proefpersoon moet willekeurig zijn.

- Er moeten genoeg proefpersonen zijn om de test representatief te maken. Genoeg wil zeggen drie tot zes proefpersonen.
- De tijd waarin het testbeeld door de proefpersoon beoordeeld kan worden moet gevarieerd kunnen worden. Hierdoor kan worden nagegaan de distorsie in één oogopslag te zien is of niet.

Er zijn experimenteermethoden waarbij de proefpersoon de keuze heeft uit slechts twee mogelijkheden en methoden waarbij het oordeel moet worden weergegeven op een schaal. Methoden met twee mogelijkheden worden gebruikt om zichtbaarheidsdrempels te bepalen. Methoden met een schaal worden gebruikt om een algemeen oordeel over de beeldkwaliteit te verkrijgen.
Een experimenteermethode waarbij de proefpersoon tussen twee mogelijkheden kan kiezen is de 'Method of Limits'. Bij deze methode wordt er aan de proefpersoon één beeld tegelijk getoond. De proefpersoon moet met ja of nee aangeven of het beeld vervormd is.
Een andere methode met twee keuzemogelijkheden is de 'Two alternative forced choice'. Bij deze methode worden er twee beelden tegelijk getoond. De proefpersoon moet dan kiezen welke van deze twee beelden het vervormde beeld is. Deze methode levert de beste resultaten als begonnen wordt met een vervorming die boven de zichtbaarheidsdrempel ligt. Vervolgens wordt bij een goed antwoord de vervorming van het testbeeld verkleind en bij een fout antwoord vergroot. Op deze manier wordt de zichtbaarheidsdrempel iteratief benaderd.
Een op de Two alternative forced choice methode lijkende methode is de 'Staircase threshold test'. Ook hier worden twee beelden tegelijk getoond en moet de proefpersoon aangeven welke het vervormde beeld is. Er wordt begonnen met een vervorming die boven de zichtbaarheidsdrempel ligt. De vervorming wordt steeds verder verkleind tot de proefpersoon het verkeerde beeld aanwijst. De stapgrootte waarin de vervorming veranderd wordt nu gehalveerd. Nu wordt de test herhaald waarbij er echter begonnen wordt met een vervorming die onder de zichtbaarheidsdrempel ligt. De vervorming neemt nu toe tot deze wordt waargenomen. Vervolgens wordt de stapgrootte weer gehalveerd en de gehele procedure herhaald. Dit gaat net zolang door tot de

minimale stapgrootte is bereikt.

Bij methoden waarbij het oordeel moet worden aangegeven op een schaal zijn de volgende punten van belang:

- De keuze van de schaal. De schaal is meestal een vijf- of tienpuntsschaal.

Om de testresultaten op een juiste manier te interpreteren moet er met de volgende punten rekening gehouden worden.

- De eerste metingen zijn niet representatief en moeten dus niet in de resultaten verwerkt worden.
- Van sommige proefpersonen blijkt pas na de test dat ze niet representatief zijn voor de bedoelde groep. De testresultaten van deze personen moet niet in de uiteindelijke resultaten verwerkt worden.
- Om te beoordelen welke experimenteermethode bruikbare resultaten oplevert kunnen eerst een aantal proefexperimenten worden uitgevoerd.

4.3 Bepaling van de kansdichtheidsfunctie

Er is door ons een programma geschreven dat van elke mogelijke pixelwaarde bepaalt hoe vaak deze in een beeld voorkomt. Met een wiskunde- of spreadsheetprogramma kan hier van een histogram worden gemaakt. Fig. 4.1 laat het histogram zien van een beeld uit de Miss sequentie.

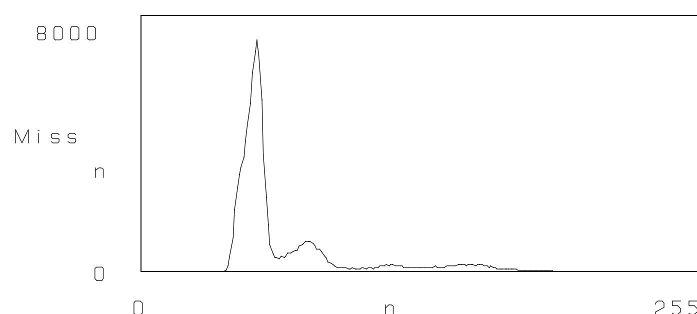


Fig. 4.1: Histogram van een beeld uit de Miss sequentie.

Niet het beeld zelf, maar het verschilbeeld van het huidige en het voorgaande beeld wordt gecodeerd en gekwantiseerd. Bij het allereerste beeld is er geen

vorig beeld. Er wordt dan het verschil bepaald met een geheel zwart beeld. Hierdoor is bij het eerste beeld het verschilbeeld gelijk aan het beeld zelf. Dit verschilbeeld zal dus veel afwijken van andere verschilbeelden. De kwantisator is geoptimaliseerd voor het verschilbeeld van twee "echte" beelden. Bij het eerste beeld zal de kwantisatiefout dus relatief groot zijn.
Als kansdichtheidsfunctie voor het bepalen van de kwantisatorniveaus moet een functie gekozen worden die zoveel mogelijk op het histogram van de te kwantiseren informatie lijkt. Hieronder zijn een aantal kansfuncties grafisch weergegeven.

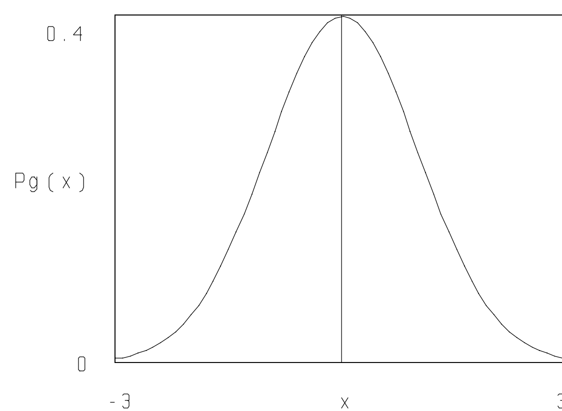


Fig. 4.2: Gaussfunctie.

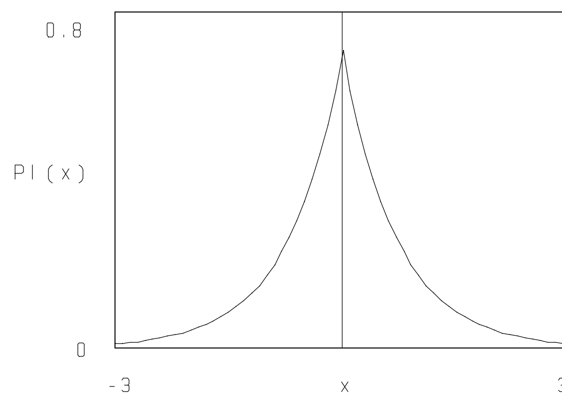


Fig. 4.3: Laplacefunctie.

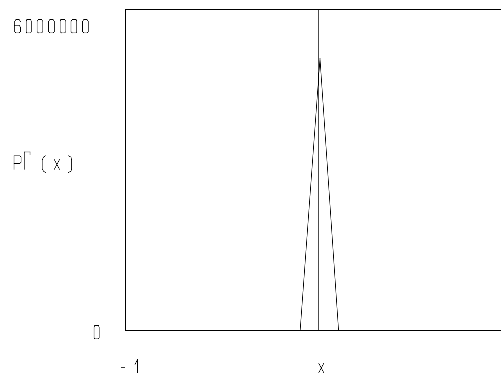


Fig. 4.4: Gammafunctie.

De Gammafunctie is zeer geconcentreerd rond de nul. De Gammafunctie ligt het dichtst bij de werkelijke kansdichtheidsfunctie van de pixels in de gecodeerde verschilbeelden. Zoals te zien is in fig. 4.4 heeft de Gammafunctie een sterk maximum bij nul. Vanwege zijn grote gelijkenis met het histogram van een verschilbeeld hebben wij de Gammafunctie gebruikt om de kwantisatieniveaus mee te berekenen. De Gammafunctie wordt daarvoor ge"curve-fit" aan het verschilbeeld. Om de gemiddelde kwantisatiefout zo klein mogelijk te houden moet er een kwantisatieniveau samenvallen met dit maximum. Bij een symmetrische verdeling van de kwantisatieniveaus om nul betekent dit wel dat het aantal kwantisatieniveaus oneven moet zijn. Hierdoor zullen niet alle 2^n mogelijke tabeladressen worden gebruikt.

Door in de verschillende stadia van het codeer en decodeer proces een histogram te bepalen kan gevolgd worden wat er met de afbeelding gebeurd. In fig. 4.5 zijn de histogrammen te zien van de verschillende stadia bij het coderen en decoderen van een beeld uit de Miss sequentie.

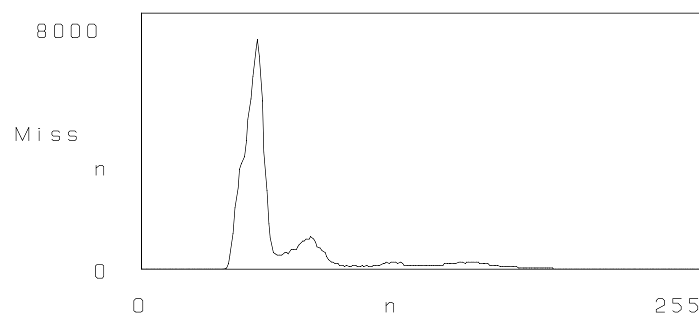


Fig. 4.5.a: Histogram van het originele beeld.

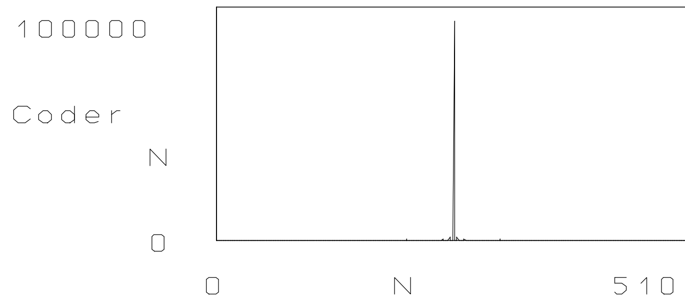


Fig. 4.5.b: Histogram na coderen en kwantiseren.

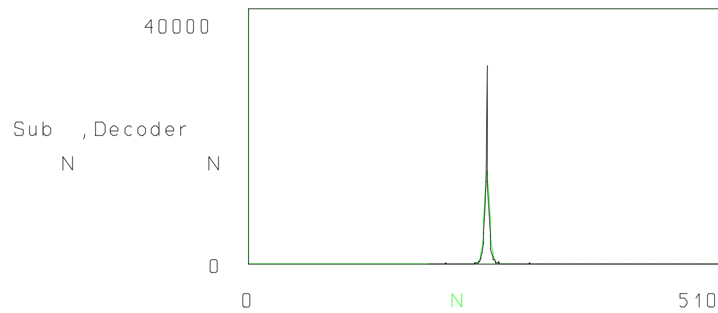


Fig. 4.5.c: Histogram na aftrekker en histogram na decoder.

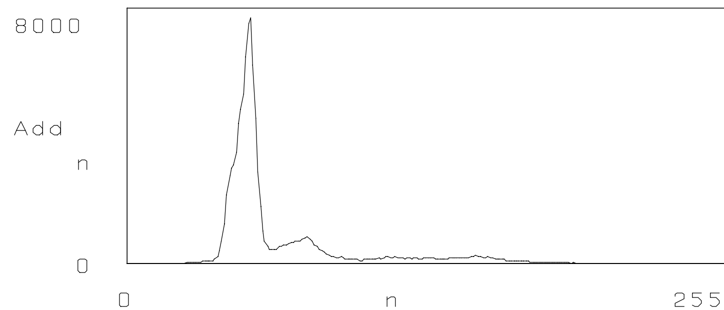


Fig. 4.5.d: Histogram van het gerestaureerde beeld.

De histogrammen van de data na aftrekken en na decoderen zijn in dezelfde afbeelding geplot. Dit is gedaan omdat deze op de kwantisatiefout na gelijk zijn. Duidelijk is te zien dat beide histogrammen vrijwel geheel samenvallen.

4.4 Metingen

Er zijn metingen verricht aan de eerste 13 beelden van de Miss sequentie. Een aantal van deze testbeelden zijn te vinden in bijlage A. Van de gedecodeerde beelden is de MSE bepaald. De tabellen met de meetresultaten staan in bijlage E.

Bij de eerste meting wordt verschilcomponent I0-I1 drie niveaus, verschilcomponent I1-I2 met vijf niveaus en verschilcomponent I2-I3 met vijftien niveaus gekwantiseerd. Dit geeft bij het verzenden van tabelwaarden een codelengte van:

I3 ⇒ 1 * 9 = 9 bits
I2-I3 ⇒ 3 * 4 = 12 bits
I1-I2 ⇒12 * 3 = 36 bits
I0-I1 ⇒48 * 2 = 96 bits +
153 bits/fragment ⇒ 2,39 bits/pixel

De waarden van de kwantisatiegrenzen en niveaus staan bijlage E.

Er zijn ook metingen verricht waarbij verschilcomponent I0-I1 met twee niveaus in plaats van drie werd gekwantiseerd. Dit is gedaan omdat de kwantisator voor de pixels van verschilcomponent I0-I1 bepalend is voor de kwaliteit van 75% van alle pixels en dus van de componenten de meeste invloed heeft op de uiteindelijke beeldkwaliteit. In bijlage C staan gedecodeerde beelden afgedrukt die gerestaureerd werden na kwantisatie met twee niveaus. Voor vergelijking zijn in bijlage B gedecodeerde beelden van de H.261 afgedrukt.

Dit geeft bij het verzenden van tabeladressen een codelengte van:

I3 ⇒ 1 * 9 = 9 bits
I2-I3 ⇒ 3 * 4 = 12 bits
I1-I2 ⇒12 * 3 = 36 bits
I0-I1 ⇒48 * 1 = 48 bits +
105 bits/fragment ⇒ 1,64 bits/pixel

Bij gebruik van beide kwantisator instellingen werd van elk gedecodeerd beeld de MSE-waarde berekend.
In bijlage E (tabel 1 en 2) is te zien dat de objectieve kwaliteit door kwantiseren met drie niveaus slechts in geringe mate toeneemt. Dit terwijl de codelengte veel groter is dan bij kwantisatie met twee niveaus.

Als de waarden van de pixels in verschilcomponent I0-I1 van het verschilbeeld allen gelijk aan nul worden gesteld geeft dit bij het verzenden van tabeladressen een codelengte van:

I3 $\Rightarrow$ 1 * 9 = 9 bits
I2-I3 $\Rightarrow$ 3 * 4 = 12 bits
I1-I2 $\Rightarrow$ 12 * 3 = 36 bits
I0-I1 $\Rightarrow$ 48 * 0 = 0 bits +
57 bits/fragment $\Rightarrow$ 0,89 bits/pixel

5. Conclusies en aanbevelingen

5.1 Conclusies

Als er alleen gekeken wordt naar de MSE waarden van de H.261 dan lijkt het of deze methode slechter is als het SIP algoritme. Uit subjectieve metingen blijkt echter dat de H.261 een betere beeldkwaliteit levert dan het SIP algoritme. De hoge waarde van de MSE bij de H.261 wordt voornamelijk veroorzaakt door vervorming van de achtergrond. De voorgrond heeft bij de H.261 een hoge kwaliteit. Men let bij beeldtelefonie meer op het gezicht als op de achtergrond. Daarom is de lage kwaliteit van de achtergrond niet hinderlijk. Hieruit blijkt dat de indruk die de MSE geeft van de beeldkwaliteit niet altijd overeenkomt met de subjectieve beeldkwaliteit.
Bij de SIP methode is de fout verdeeld over het gehele beeld. Dit zorgt ervoor dat de kwaliteit als slechter wordt ervaren als bij de H.261.

Een ander belangrijk punt van overweging is de compressiefactor. De PCM codering van een pixel kost 8 bit. Voor hetzelfde beeld gecodeerd met de SIP methode is 1,64 bit nodig (I0-I1 component met 2 niveaus gekwantiseerd). De compressiefactor bedraagt dan 8 / 1,64 oftewel 4,9.

Uit de metingen blijkt dat bij het kwantiseren met drie niveaus de kwaliteit weinig verschilt met twee niveaus. Dit is zowel subjectief als objectief bepaald. Dit terwijl de codelengte van component I0-I1 met 50% toeneemt. Bij

kwantisatie met één niveau is er wel een zichtbaar lagere beeldkwaliteit.

5.2 Aanbevelingen

Het SIP algoritme werkt met een overlappende rij en kolom. Ook moet het beeld opgerekt moet worden als de afmetingen niet uitkomen. Hierdoor is het SIP coderingsalgoritme vooral geschikt voor het coderen van beelden die in zijn geheel beschikbaar zijn. Het beeldtelefoon simulatiepakket biedt een beeldfragment van 8 bij 8 aan aan het blok coder. De functie fragm_code uit het SIP programma heeft als invoer een fragment van 9 bij 9 nodig en geeft als uitvoer een fragment van 8 bij 8. Doordat het beeldtelefoon simulatiepakket met losse fragmenten werkt kan het voordeel dat er geen blokeffect optreedt niet worden benut. Dit komt doordat de fragmenten elkaar nu niet één rij en één kolom overlappen. Om niet het gehele simulatie pakket om te moeten bouwen van 8 bij 8 naar 9 bij 9 is gekozen om het SIP algoritme zo te veranderen dat het wel met beeldfragment van 8 bij 8 kan werken. Hiervoor zijn de volgende mogelijkheden overwogen:

1) Herrangschikking van de pixels in de array over de verschilcomponenten. Bijvoorbeeld als volgt:

```
 I3    I0-I1  I1-I2  I2-I3  I2-I3  I1-I2  I0-I1   I3
I0-I1  I0-I1  I0-I1  I0-I1  I0-I1  I0-I1  I0-I1  I0-I1
I1-I2  I0-I1  I1-I2  I0-I1  I0-I1  I1-I2  I0-I1  I1-I2
I2-I3  I0-I1  I0-I1  I2-I3  I2-I3  I0-I1  I0-I1  I2-I3
I2-I3  I0-I1  I0-I1  I2-I3  I0-I1  I0-I1  I0-I1  I2-I3
I1-I2  I0-I1  I1-I2  I0-I1  I0-I1  I1-I2  I0-I1  I1-I2
I0-I1  I0-I1  I0-I1  I0-I1  I0-I1  I0-I1  I0-I1  I0-I1
 I3    I0-I1  I1-I2  I2-I3  I2-I3  I1-I2  I0-I1   I3
```

Voorbeeldverdeling van de componenten over een 8 bij 8 array.

Het terugrekenen van een zoals hierboven aangegeven fragment gaat als volgt:

component I3 $\xrightarrow{\text{interpoleer}}$ geïnterpoleerdcomponent I3

$$\begin{bmatrix} 1 & 8 \\ 57 & 64 \end{bmatrix} \xrightarrow{\text{interpoleer}} \begin{bmatrix} 1 & \hat{4} & \hat{5} & 9 \\ \hat{25} & \hat{28} & \hat{29} & \hat{32} \\ \hat{33} & \hat{36} & \hat{37} & \hat{40} \\ 57 & \hat{60} & \hat{61} & 64 \end{bmatrix}$$

verschilcomponent I2–I3

$$\begin{bmatrix} 0 & 4-\hat{4} & 5-\hat{5} & 0 \\ 25-\hat{25} & 28-\hat{28} & 29-\hat{29} & 32-\hat{32} \\ 33-\hat{33} & 36-\hat{36} & 37-\hat{37} & 40-\hat{40} \\ 0 & 60-\hat{60} & 61-\hat{61} & 0 \end{bmatrix}$$

\+

component I2 $\xrightarrow{\text{interpoleer}}$ geïnterpoleerdcomponent I2

$$\begin{bmatrix} 1 & 4 & 5 & 9 \\ 25 & 28 & 29 & 32 \\ 33 & 36 & 37 & 40 \\ 57 & 60 & 61 & 64 \end{bmatrix} \xrightarrow{\text{interpoleer}} \begin{bmatrix} 1 & \hat{3} & 4 & 5 & \hat{6} & 8 \\ \hat{17} & \hat{19} & & & \hat{22} & \hat{24} \\ 25 & & 28 & 29 & & 32 \\ 33 & & 36 & 37 & & 40 \\ \hat{41} & \hat{43} & & & \hat{46} & \hat{48} \\ 57 & \hat{59} & 60 & 61 & \hat{62} & 64 \end{bmatrix}$$

verschilcomponent I1–I2

$$\begin{bmatrix} 0 & 3-\hat{3} & 0 & 0 & 6-\hat{6} & 0 \\ 17-\hat{17} & 19-\hat{19} & & & 22-\hat{22} & 24-\hat{24} \\ 0 & & 0 & 0 & & 0 \\ 0 & & 0 & 0 & & 0 \\ 41-\hat{41} & 43-\hat{43} & & & 46-\hat{46} & 48-\hat{48} \\ 0 & 59-\hat{59} & 0 & 0 & 62-\hat{62} & 0 \end{bmatrix}$$

\+

component I1 $\xrightarrow{\text{interpoleer}}$ geïnterpoleerdcomponent I1

$$\begin{bmatrix} 1 & 3 & 4 & 5 & 6 & 8 \\ 17 & 19 & & & 22 & 24 \\ 25 & & 28 & 29 & & 32 \\ 33 & & 36 & 37 & & 40 \\ 41 & 43 & & & 46 & 48 \\ 57 & 59 & 60 & 61 & 62 & 64 \end{bmatrix} \xrightarrow{\text{interpoleer}} \begin{bmatrix} 1 & \hat{2} & 3 & 4 & 5 & 6 & \hat{7} & 8 \\ \hat{9} & \hat{10} & \hat{11} & \hat{12} & \hat{13} & \hat{14} & \hat{15} & \hat{16} \\ 17 & \hat{18} & 19 & \hat{20} & \hat{21} & 22 & \hat{23} & 24 \\ 25 & \hat{26} & \hat{27} & 28 & 29 & \hat{30} & \hat{31} & 32 \\ 33 & \hat{34} & \hat{35} & 36 & 37 & \hat{38} & \hat{39} & 40 \\ 41 & \hat{42} & 43 & \hat{44} & \hat{45} & 46 & \hat{47} & 48 \\ \hat{49} & \hat{50} & \hat{51} & \hat{52} & \hat{53} & \hat{54} & \hat{55} & \hat{56} \\ 57 & \hat{58} & 59 & 60 & 61 & 62 & \hat{63} & 64 \end{bmatrix}$$

verschilcomponent I0–I1

$$\begin{bmatrix} 0 & 2-\hat{2} & 0 & 0 & 0 & 0 & 7-\hat{7} & 0 \\ 9-\hat{9} & 10-\hat{10} & 11-\hat{11} & 12-\hat{12} & 13-\hat{13} & 14-\hat{14} & 15-\hat{15} & 16-\hat{16} \\ 0 & 18-\hat{18} & 0 & 20-\hat{20} & 21-\hat{21} & 0 & 23-\hat{23} & 0 \\ 0 & 26-\hat{26} & 27-\hat{27} & 0 & 0 & 30-\hat{30} & 31-\hat{31} & 0 \\ 0 & 34-\hat{34} & 35-\hat{35} & 0 & 0 & 38-\hat{38} & 39-\hat{39} & 0 \\ 0 & 42-\hat{42} & 0 & 44-\hat{44} & 45-\hat{45} & 0 & 47-\hat{47} & 0 \\ 49-\hat{49} & 50-\hat{50} & 51-\hat{51} & 52-\hat{52} & 53-\hat{53} & 54-\hat{54} & 55-\hat{55} & 56-\hat{56} \\ 0 & 58-\hat{58} & 0 & 0 & 0 & 0 & 63-\hat{63} & 0 \end{bmatrix}$$

\+

fragment I0

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\ 17 & 18 & 19 & 20 & 21 & 22 & 23 & 24 \\ 25 & 26 & 27 & 28 & 29 & 30 & 31 & 32 \\ 33 & 34 & 35 & 36 & 37 & 38 & 39 & 40 \\ 41 & 42 & 43 & 44 & 45 & 46 & 47 & 48 \\ 49 & 50 & 51 & 52 & 53 & 54 & 55 & 56 \\ 57 & 58 & 59 & 60 & 61 & 62 & 63 & 64 \end{bmatrix}$$

De interpolaties gaan hierbij als volgt:

Interpolatie I3

$\hat{4}=\hat{5}=(1+8)/2$ $\quad$ $2\hat{5}=3\hat{3}=(1+57)/2$

$2\hat{8}=2\hat{9}=3\hat{6}=3\hat{7}=(1+8+57+64)/4$

$3\hat{2}=4\hat{0}=(8+64)/2$ $\quad$ $6\hat{0}=6\hat{1}=(57+64)/2$

Interpolatie I2

$\hat{3}=(1+4)/2$ $\quad$ $\hat{6}=(5+8)/2$ $\quad$ $1\hat{7}=(1+25)/2$

$1\hat{9}=(1+4+25+28)/4$ $\quad$ $2\hat{2}=(5+8+29+32)/4$ $\quad$ $2\hat{4}=(8+32)/2$

$4\hat{1}=(33+57)/2$ $\quad$ $4\hat{3}=(33+36+57+60)/4$ $\quad$ $4\hat{6}=(37+40+61+64)/4$

$4\hat{8}=(40+64)/2$ $\quad$ $5\hat{9}=(57+60)/2$ $\quad$ $6\hat{2}=(61+64)/2$

Interpolatie I1

$\hat{2}=(1+3)/2$ $\quad$ $\hat{7}=(6+8)/2$ $\quad$ $\hat{9}=(1+17)/2$

$1\hat{0}=(1+3+17+19)/4$ $\quad$ $1\hat{1}=(3+19)/2$ $\quad$ $1\hat{2}=2\hat{0}=(4+28)/2$

$1\hat{3}=2\hat{1}=(5+29)/2$ $\quad$ $1\hat{4}=(6+22)/2$ $\quad$ $1\hat{5}=(6+8+22+24)/4$

$1\hat{6}=(8+24)/2$ $\quad$ $1\hat{8}=(17+19)/2$ $\quad$ $2\hat{3}=(22+24)/2$

$2\hat{6}=2\hat{7}=(25+28)/2$ $\quad$ $3\hat{0}=3\hat{1}=(29+32)/2$ $\quad$ $3\hat{4}=3\hat{5}=(33+36)/2$

$3\hat{8}=3\hat{9}=(37+40)/2$ $\quad$ $4\hat{2}=(41+43)/2$ $\quad$ $4\hat{4}=5\hat{2}=(36+60)/2$

$4\hat{5}=5\hat{3}=(37+61)/2$ $\quad$ $4\hat{7}=(46+48)/2$ $\quad$ $4\hat{9}=(41+57)/2$

$5\hat{0}=(41+43+57+59)/4$ $\quad$ $5\hat{1}=(43+59)/2$ $\quad$ $5\hat{4}=(46+62)/2$

$5\hat{5}=(46+48+62+64)/4$ $\quad$ $5\hat{6}=(48+64)/2$ $\quad$ $5\hat{8}=(57+59)/2$

$6\hat{3}=(62+64)/2$

Hierdoor wordt wel de codelengte anders dan bij het oorspronkelijke algoritme, want nu komen er 36 pixels in component I0-I1, 12 pixels in component I1-I2, 12 pixels in component I2-I3 en 4 pixels in component I3 terecht.

2) Het binnen de codeer functie oprekken van het aangeboden 8 bij 8 fragment tot een 9 bij 9 fragment. Waarna dit vervolgens als een gewoon 9 bij 9 fragment gecodeerd kan worden.

a) Het oprekken van een fragment kan afhankelijk van de inhoud van het fragment gebeuren door de 9^e kolom gelijk te maken aan de laatste kolom van het invoer fragment en de 9^e rij gelijk te maken aan de laatste rij van het invoer fragment. Dit geeft bij het interpoleren een kleine fout, zodat pixels uit de I0-I1 verschilcomponent goed met de voor het coderen van deze pixels beschikbare aantal bits gecodeerd kunnen worden. Deze methode valt echter af omdat het niet meer mogelijk is om het zo gecodeerde fragment weer te decoderen. Dit komt doordat voor het terugrekenen van component
I_{k-1} de waarden van de component I_k bekend moeten zijn.

b) Het oprekken van het fragment kan ook onafhankelijk van de inhoud van het fragment gebeuren. Dit kan door de pixels van 9^e kolom en rij een vaste waarde te geven. Om zo min mogelijk bits nodig te hebben om de pixels fragmenten te coderen is het nodig de maximaal mogelijke waarde van de geïnterpoleerde pixels zo klein mogelijk te houden. Voor de constante waarde moet dan het gemiddelde van alle mogelijke waarden worden gekozen. Bij een verschilbeeld zal deze waarde 0 zijn. Voor een gewoon beeld zal de waarde van de constante gelijk zijn aan "max_pixelwaarde/2".

Om problemen zoals hierboven beschreven te vermijden doen wij de volgende aanbeveling. Biedt de beelden niet in de vorm van fragmenten aan aan het beeldcoderingsalgoritme. Maar herschrijf het simulatieprogramma zó dat de gehele Y, U en V beelden aan het transformatieproces worden aangeboden. Men kan dan bij elk beeldcoderingsalgoritme dat men wil testen zelf bepalen of men het beeld op wil delen in fragmenten en zo ja hoe groot deze fragmenten moeten zijn.

Een suggestie om de kwaliteit van gedecodeerde beelden te verhogen. De meeste pixels van een beeldfragment komen in component I0-I1 terecht. Dit is echter ook de component die met het minste aantal kwantisatieniveaus gekwantiseerd wordt. Kwantiseren met twee niveaus zal een grote fout opleveren. Dit komt doordat de kansfunctie van de pixelwaarden van deze component bij de waarde nul een zeer sterk maximum heeft. Bij kwantisatie met twee niveaus zal juist bij dit maximum de kwantisatiegrens liggen. Hierdoor zullen pixels met waarden die niet veel verschillen, maar die wel een verschillend teken hebben na kwantisatie kunnen wel veel in waarde verschillen. Hierdoor ontstaat in de gebieden met gelijkmatige pixelwaarden veel ruis. Het kwantiseren met drie niveaus geeft verbetering van de beeldkwaliteit, maar geeft aanzienlijk meer te verzenden code.

Om bij twee kwantisatie niveaus toch geen ruis te krijgen kan met zogenaamde "flat" en "noisy" fragmenten worden gewerkt. Een beeld bestaat uit gebieden met kleine pixelwaarde variaties (de vlakken) en uit gebieden met grote pixelwaarde variaties (de contouren). Een fragment is flat als de pixelwaarden niet veel van elkaar verschillen en het fragment dus in zijn geheel in een vlak ligt. Een fragment is noisy als de pixelwaarden wel veel van elkaar verschillen en er dus één of meer contouren in dat fragment vallen. Bij een noisy fragment wordt de I0-I1 component met twee niveaus gekwantiseerd. Bij een flat fragment worden de pixelwaarden van de pixels in de I0-I1 component allemaal gelijk aan nul gesteld.

Een fragment is noisy als het gemiddelde van de absolute waarden van de pixelwaarden van de pixels in de I0-I1 component groter is dan een bepaalde grenswaarde. Komt het gemiddelde niet boven deze grenswaarde dan is het fragment flat. Kies voor de grenswaarde van het fragment de waarde s. Kwantiseer dan de I0-I1 pixels in een noisy fragment naar de waarden -q en +q waarbij geldt dat $q = 2 \cdot s$. Door subjectieve kwaliteitsmeting kan de waarde van s geoptimaliseerd worden. Bij fragmenten die op de plaatsen in het beeld liggen waar de ogen en mond worden weergegeven zullen de grenswaarden waarschijnlijk lager gekozen moeten worden dan bij fragmenten die in de achtergrond liggen.

Het is nu nodig om aan de ontvanger kenbaar te maken of een fragment noisy

is of flat. In het geval dat een fragment noisy is moet ook q overgezonden worden. Dit geeft overhead. Deze overhead kan beperkt worden door q te kwantiseren zodat niet q zelf bij elk fragment overgezonden hoeft te worden, maar een tabeladres overgezonden kan worden (zoals in paragraaf 1.3.1).

De compressiefactor kan het best verhoogd worden door heel component I0-I1 op nul te stellen (zie meting). Dit geeft al een compressiefactor van negen. Kritiek punt hierbij is de vermindering van de kwaliteit. Dit is een onderwerp waarnaar nog verder onderzoek gedaan zou kunnen worden.

Referenties

1) Dr. Ilia M. Bockstein, *Component Transformation with Pixel Interpolation. An effective method of gray-scale and coulor image compression*, Institute of Problems of Information Transmission, USSR, Academy of Science, 19 Ermolovoy st., Moscow, 101447 USSR.

2) Prof. dr. ir. J. Biemond en dr. ir. R.L. Lagendijk, *Digitale Signaalcodering*, collegedictaat, Technische Universiteit Delft, Delft, 1989.

3) P.F. Panter en W. Dite, *Quantization Distorsion in Pulse-Count Modulation with Non-Uniform Spacing of Levels*, Procedings of the I.R.E., I.R.E., 1951, pp. 44-48.

4) J. Max, *Quantizing for Minimum Distortion*, I.R.E Transactions on Information Theory, I.R.E., 1960, pp. 7-12.

5) P.J. Burt en E.H. Adelson, *The Laplacian Pyramid as a Compact Image Code*, IEEE Transactions on Communications, IEEE, vol. com-31, no4, april 1983, pp. 532-540.

6) R.J. Clarke, *Transform coding of Images*, micro elektronics and signal processing, Academic Press, Orlando Florida, 1985, pp. 87-115.

7) R.W. Barth, *Beeldkwaliteit in de H.261 codec*, afstudeerverslag, , Technische Universiteit Delft, Delft, 1991.

8) R.A. van Schijndel, *Het foutbeeld van de beeldtelefoon*, Taakverslag, Technische Universiteit Delft, Delft, 1991.

A.3 Origineel beeld 16 van de Miss sequentie

B.1 Gedecodeerd beeld 1 volgens de H.261 standaard

B.2 Gedecodeerd beeld 2 volgens de H.261 standaard

C.1 Gedecodeerd beeld 1 volgens de SIP methode

C.2 Gedecodeerd beeld 8 volgens de SIP methode

C.3 Gedecodeerd beeld 1 volgens de SIP methode

D.1 Gedecodeerd beeld met een MSE van 50

D.2 Gedecodeerd beeld met een MSE van 500

D.3 Gedecodeerd beeld met een MSE van 1000

Aantal kwant. niveaus	Miss sequentie nummer												
	1	2	3	4	5	6	7	8	9	10	11	12	13
1	12,5	10,9	11,6	12,1	12,1	11,7	11,9	11,9	12,1	12,0	12,0	12,4	12,4
2	10,2	7,1	8,7	11,4	8,5	10,6	8,5	10,1	8,2	9,6	7,7	9,6	8,2
3	8,8	5,8	7,7	11,0	7,2	9,8	7,5	9,1	7,1	8,5	6,6	8,5	7,2
4	7,4	4,7	6,9	6,7	6,4	6,9	6,9	6,5	6,4	6,4	6,3	6,6	6,4

Tabel 1 MSE waarden gemeten voor de Miss sequentie bij wisselend aantal kwantisatie niveaus voor component I0-I1.

Aantal kwant. niveaus	Claire sequentie nummer												
	1	2	3	4	5	6	7	8	9	10	11	12	13
1	19,2	28,4	29,4	28,1	29,5	26,1	33,1	33,4	34,7	36,1	38,0	40,1	42,2
2	18,1	26,3	26,8	25,1	25,7	21,8	30,5	29,1	30,2	30,7	32,4	38,4	36,0
3	17,3	25,0	25,3	23,5	23,6	19,5	29,0	26,9	28,0	28,1	29,6	32,0	33,0
4	14,9	20,4	21,3	18,8	18,9	25,1	27,2	23,2	22,3	23,7	26,0	26,2	24,9

Tabel 2 MSE waarden gemeten voor de Claire sequentie bij wisselend aantal kwantisatie niveaus voor component I0-I1.

Aantal niveaus	Kwantisatie niveaus	Kwantisatie grenzen
1	0	-255, 255
2	-1, 1	-255, 0, 255
3	-2, 0, 2	-255, -1, 1, 255
4	-10, -2, 2, 10	-255, -6, 0, 6, 255

Tabel 3 Kwantisatie niveaus en grenzen van component I0-I1 bij de verschillende metingen.

Compo-nent	Kwantisatie niveaus
I1-I2	-60, -17, 0, 17, 60
I2-I3	-90, -46, - 28, -18, -12, -7, -4, 0, 4, 7, 12, 18, 28, 46, 90
I3	-255..255

Tabel 4 Kwantisatie niveaus voor de overige componenten.

Compo-nent	Kwantisatie grenzen
I1-I2	-255, -25, -8, 8, 25, 255
I2-I3	-255, -58, -34, -22, -14, -9, -5, -2, 2, 5, 9, 14, 22, 34, 58, 255
I3	-255..255

Tabel 5 Kwantisatie grenzen voor de overige componenten.

Aantal 64 Kbit kanalen	Miss sequentie												
	1	2	3	4	5	6	7	8	9	10	11	12	13
30	24,2	12,5	15,4	13,0	12,1	13,6	15,8	9,4	11,3	10,5	9,1	10,6	9,6
	Claire sequentie												
30	29,7	12,9	9,1	6,6	5,2	4,7	4,7	4,7	4,6	4,7	4,6	4,6	4,5

Tabel 6 MSE waarden gemeten voor de Miss en Claire sequentie bij een kanaalcapaciteit van 1,92 Mbit/sec voor de H.261 standaard.

```
/****************************************************************************/
/*  Filenaam: MAIN.C                                                        */
/*  Software coding scheme of the                                           */
/*  COMPONENT TRANSFORMATION with PIXEL INTERPOLATION (CTPI) method.        */
/****************************************************************************/
/*                       H E A D E R   F I L E S                         */
/*                       =======================                         */
#include <string.h>
#include <stdlib.h>   /* exit function              */
#include <stdio.h>    /* file and screen functions  */
#include "const.h"    /* constants                  */


/*                 C O M P I L E R   D I R E C T I V E S           */
/*                 =====================================           */
#if BREED>MAXLINE
#error "De lijn bevat teveel pixels. Maak MAXLINE groter"
#endif


/*                 G L O B A L E   V A R I A B E L E N             */
/*                 ===================================             */
int  fragm_bar_i[9][MAXLINE+1];
int  fragm_bar_o[9][MAXLINE+1];

/*                   F O R W A R D   D E C L A R A T I E               */
/*                 =====================================               */
extern int maak_verschilbeeld(char sequence_name_i[80],
                              char sequence_name_o[80],
                              int frameno, int stap);
extern int codeer(int frameno);
extern int decodeer(int frameno);
extern int sommeer(char sequence_out[80], int frameno);


/*                       M A I N   P R O G R A M                         */
/*                   ===============================                     */
void main(void)
{  char sequence_i[80], sequence_o[80], tmpfile[80];
   int  totaal_frames, frame, result, stap;
   void terminate(int fout);

   /* Vraag om naam van sequence met invoergrijswaarden */
   printf("Type in name of sequence (without extension): ");
   scanf("\n%s", sequence_i);

   /* Vraag om naam van uitvoersequence */
   printf("Type in name of output sequence (without extension): ");
   scanf("\n%s", sequence_o);

   /* Invoer sequence moet eindigen op een punt */
   strcat(sequence_i, ".");    /* Naam moet eindigen op punt */

   /* Uitvoer sequence moet eindigen op een punt */
   strcat(sequence_o, ".");    /* Naam moet eindigen op punt */

   /* Vraag om het aantal beelden dat bewerkt moet worden */
   printf("How many frames must be processed: ");
   scanf("%d", &totaal_frames);

   /* Vraag om de stapgrootte tussen de frame nummers */
   printf("How big is the step between two frames: ");
   scanf("%d", &stap);
   printf("\n");

   frame = 0;
   /* Herhaal zolang er nog een opvolgend beeld is */
   while (frame < totaal_frames)
   {   frame++;
       printf("Processing frame (%d/%d): %d\n",
               (frame-1)*stap, frame*stap, frame);

       /* Bepaal verschilbeeld van huidige frame met vorige frame
          en sla dit verschilbeeld op in temporary file VERSCHILFRAME
       */
       printf("\t\t-Substracting frames\n");
       result = maak_verschilbeeld(sequence_i, sequence_o, frame, stap);
       if ( result != 0) terminate(result);
```

```
        /* Codeer het verschilbeeld opgeslagen in file VERSCHILFRAME en
           kwantiseer het beeld. Zet het gecodeerde beeld in file met
           de naam opgegeven in de constante CODEDFRAME en met
           extensie het frameno (frame).
        */
        printf("\t\t-Coding and quantizing\n");
        result = codeer(frame);
        if ( result != 0) terminate(result);

        /* Decodeer de (gekwantiseerde en gecodeerde) verschilframes
           opgeslagen in de files en sla deze op in filesequence met
           in te voeren naam
        */
        printf("\t\t-Decoding\n");
        result = decodeer(frame);
        if ( result != 0) terminate(result);

        /* Bepaal het originele beeld weer door de file met het voorgaande
           frame en de file met het huidige frame op te tellen.
        */
        printf("\t\t-Adding to previous frame\n");
        result = sommeer(sequence_o, frame);
        if ( result != 0) terminate(result);

        /* Delete tijdelijke files */
        unlink(VERSCHILFRAME);
        sprintf(tmpfile, "%s%03d", CODEDFRAME, frame);
        unlink(tmpfile);
        sprintf(tmpfile, "%s%03d", DECODEDFRAME, frame);
        unlink(tmpfile);
     }

   printf("\n\nDone!\n");
}

void terminate(int fout)
{   printf("\n\n");
    switch (fout)
    {  case 1:  printf("\aFile could not be opened for reading!\n");
                break;
       case 2:  printf("\aFile could not be opened for writing!\n");
                break;
       case 3:  printf("\aOut of memory!\n");
                break;
       case 4:  printf("\aError during readin from file!\n");
                break;
       case 5:  printf("\aError during write to file!\n");
                break;

       default: printf("Bug!\n");
    }
    exit(1);                         /* zorgt voor sluiten files */
}
```

```
/**************************************************************************/
/*  Filenaam: SUB.C                                                       */
/*  Bepaal het verschil tussen het frame met nummer "frameno"             */
/*  en het voorgaande frame met nummer "frameno-stap"                     */
/**************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include "const.h"


int maak_verschilbeeld(char sequence_name_i[80],
                       char sequence_name_o[80],
                       int frameno, int stap)
{  char          frame0[80], frame1[80];
   FILE          *f0, *f1, *ftemp;
   unsigned char *buf0, *buf1;
   int           *bufuit;
   int           n, m;

   /* Bepaal filenaam van voorgaande frame en huidig frame en open ze  */
   /* voorgaande frame */
   if (frameno != 1)
   {  sprintf(frame0, "%s%03d", sequence_name_o, (frameno-1)*stap);
      f0 = fopen(frame0, "rb");
      if (f0 == NULL) return(1);
   }

   /* huidige frame */
   sprintf(frame1, "%s%03d", sequence_name_i, frameno*stap);
   f1 = fopen(frame1, "rb");
   if (f1 == NULL) return(1);

   /* Uitvoer */
   ftemp = fopen(VERSCHILFRAME, "wb");
   if (ftemp == NULL) return(2);

   /* Alloceer geheugen voor buffers */
   buf0 = (unsigned char *) malloc(ELEMENTEN*sizeof(char));
   buf1 = (unsigned char *) malloc(ELEMENTEN*sizeof(char));
   bufuit = (int *) malloc(ELEMENTEN*sizeof(int));
   if ( (buf0 == NULL) || (buf1 == NULL) || (bufuit == NULL) ) return(3);

   while ( !feof(f1) )
   {  /* Vul buffers */
      m = fread(buf1, sizeof(char), ELEMENTEN, f1);
      if (frameno == 1)
        for (n=0; n<m; n++) buf0[n] = 0;
      else
        n = fread(buf0, sizeof(char), ELEMENTEN, f0);

      /* Check of er goed ingelezen is */
      n = (n>m? m : n); if (n == 0) return(4);

      /* Maak het verschil (huidige frame) - (vorige frame) */
      for (m=0; m<n; m++)
          bufuit[m] = (int) (buf1[m] - buf0[m]);

      /* Bewaar verschilbuffer */
      m = fwrite(bufuit, sizeof(int), n, ftemp);

      /* Check of er goed weggeschreven is */
      if (m != n) return(5);

   }

   printf("\b ");
   /* Geef geheugen vrij */
   free(bufuit);
   free(buf1);
   free(buf0);

   /* Sluit files */
   if (frameno != 1) fclose(f0);
   fclose(f1);
   fclose(ftemp);
   return(0);
}
```

```
/*****************************************************************************/
/*  Filenaam: CODEER.C                                                       */
/*  Deze functie codeert de invoerfile met de naam VERSCHILFRAME             */
/*  en zet de uitvoer in de file met de naam CODEDFRAME.xxx.                 */
/*  VERSCHILFRAME en CODEDFRAME zijn constante op te geven in                */
/*  de headerfile const.h.                                                   */
/*  xxx is het nummer van de frame (frameno).                                */
/*  DE AFMETINGEN VAN HET FRAME ZIJN BEPAALD DOOR DE CONSTANTE               */
/*  'BREED' EN 'HOOG'. HET FRAME MOET EEN VEELVOUD VAN 8 PIXELS              */
/*  HEBBEN (bv. 512*512, 256*288, 352*288 etc).                              */
/*****************************************************************************/
#include "const.h"
#include <stdio.h>
#include <stdlib.h>

/*                  G L O B A L E   V A R I A B E L E N                  */
/*                  ===================================                  */
extern int fragm_bar_i[9][MAXLINE];
extern int fragm_bar_o[9][MAXLINE];

/* Forward declaratie van de gebruikte functies */
void fragm_code(int i0);
extern int quant_I23(int value);
extern int quant_I12(int value);
extern int quant_I01(int value);


int codeer(int frameno)
{  FILE *rframe, *wframe;
   char uitvoerfile[80];
   int  j0, r8, i, i0, i1, i2, i4, i8;
   int  begin_lijn, pixel, lijn, gelezen;

   /* Maak de filenaam waar het gecodeerde frame in komt te staan */
   sprintf(uitvoerfile, "%s%03d",  CODEDFRAME, frameno);

   /* Open verschilframe invoerfile voor lezen */
   if ( (rframe=fopen(VERSCHILFRAME, "rb")) == NULL ) return(1);

   /* Open uitvoerfile voor schrijven */
   if ( (wframe=fopen(uitvoerfile, "wb")) == NULL ) return(2);

   /* Eerste lijn waar de pixels in fragm_bar_i gezet worden */
   begin_lijn = 0;

/* Besturing van het coderen in de hoogte-richting
  0  +-------+ j0
  .  | fragm |  .
  .  |   1   |  .
  8  +-------+ j8/j0
  .  | ..... |   .                      +---------------------+
  .  |       |   .                      |                     | -
  .  +-------+ j8/j0                    |                     | ^
  .  | ..... |   .                      |     hele beeld      | |
  .  |       |   .                      |                     | HOOG
  .  +-------+ j8/j0                    |                     | |
  .  | ..... |   .                      |                     | v
  .  |       |   .                      |                     | -
 hoog+-------+ j8/j0                    +---------------------+
                                        |<------- BREED ----->|

*/
   for (j0 = 0; j0 < HOOG; j0 += 8)       /* Lus in y-richting */
   {   r8 = j0+8;

       /* Lees eerste keer 9 beeldlijnen, 9de lijn nodig om 8ste lijn te
          kunnen interpoleren 9de lijn is daarna steeds de 1ste lijn in
          de volgende balk. Eerste keer 9 lijnen lezen daarna steeds 8 lijnen.
          beeldlijnen voor beeldlijn inlezen.
       */
       for (lijn=begin_lijn; lijn<9; lijn++)
       {  gelezen = fread(&fragm_bar_i[lijn][0], sizeof(int), BREED, rframe);
          if (gelezen != BREED)
            /* copieer laatst gelezen regel naar 9de regel in balk
               (oprekken in hoogte richting)
            */
            for (pixel=0; pixel<BREED; pixel++)
            fragm_bar_i[8][pixel] = fragm_bar_i[7][pixel];
       }

       /* Na de eerste keer data vanaf regel een in fragm_bar_i zetten */
       begin_lijn = 1;
```

```
    /* Codeerlus voor een balk met breedte van het beeld en
       hoogte 9 pixels
    i0=0     i0=8                                     i0
       +-------+------+-------+-------+-------+-------+-------+
       | fragm | fragm| ....  | ....  | ..... | ....  | fragm |
       |   1   |   2  |       |       |       |       |   x   |
       +-------+------+-------+-------+-------+-------+-------+
    */
    for (i0 = 0; i0<BREED; i0 += 8) fragm_code(i0);

    /* Bewaar gecodeerde balk (lijn voor lijn wegschrijven) */
    for (lijn=0; lijn<8; lijn++)
       fwrite(&fragm_bar_o[lijn][0], sizeof(int), BREED+1, wframe);

    /* Kopieer regel 9 naar regel 1 voor de volgende balk */
    for (pixel=0; pixel<BREED; pixel++)
      fragm_bar_i[0][pixel] = fragm_bar_i[8][pixel];

    /* Laatste regel van uitvoerbeeld maken en wegschrijven
       (nodig voor decoderen)
    */
    if (r8 >= HOOG)
      {  /* component I3 */
         for (i = 0; i < BREED+1; i += 8)
           {  i0 = i; if (i0 >= BREED) i0 = BREED-1;
              fragm_bar_o[0][i] = fragm_bar_i[0][i0];
           }

         /* component I2-I3 */
         for (i = 0; i < BREED; i += 8)
           {  i0 = i;   if (i0 >= BREED) i0 = BREED-1;
              i4 = i+4; if (i4 >= BREED) i4 = BREED-1;
              i8 = i+8; if (i8 >= BREED) i8 = BREED-1;
              fragm_bar_o[0][i+4] =
               quant_I23(  fragm_bar_i[0][i4] -
                         ((fragm_bar_i[0][i0] + fragm_bar_i[0][i8]) / 2) );
           }
         /* component I1-I2 */
         for (i = 0; i < BREED; i += 4)
           {  i0 = i;   if (i0 >= BREED) i0 = BREED-1;
              i2 = i+2; if (i2 >= BREED) i2 = BREED-1;
              i4 = i+4; if (i4 >= BREED) i4 = BREED-1;
              fragm_bar_o[0][i+2] =
               quant_I12(  fragm_bar_i[0][i2] -
                         ((fragm_bar_i[0][i0] + fragm_bar_i[0][i4]) / 2) );
           }
         /* component I0-I1 */
         for (i = 0; i < BREED; i += 2)
           {  i0 = i;   if (i0 >= BREED) i0 = BREED-1;
              i1 = i+1; if (i1 >= BREED) i1 = BREED-1;
              i2 = i+2; if (i2 >= BREED) i2 = BREED-1;
              fragm_bar_o[0][i+1] =
               quant_I01(  fragm_bar_i[0][i1] -
                         ((fragm_bar_i[0][i0] + fragm_bar_i[0][i2]) / 2) );
           }

         /* Bewaar deze laatste regel */
         fwrite(fragm_bar_o, sizeof(int), BREED+1, wframe);
      }
  }

printf("\b ");
/* Close al files */
fclose(rframe); fclose(wframe);
return(0);
}
```

```
/*              Component Interpolation Coding of Fragment                */
/*              ========================================                  */
/*   Invoer:  De pixelcoordinaat van de linkerbovenhoek. i0 is de horizont.
              uitwijking.
              Invoer array van 9x9 die deel uit maakt van een "balk"
              (fragm_bar_i) van deze array's naast elkaar. De waarde i0
              is de horizontale uitwijking naar rechts van de linkerboven
              waarde in deze array van 9x9.
              De (globale) constanten 'BREED' en 'HOOG' hebben de waarden van rsp. de
              breedte en hoogte in pixels van het invoer beeld.
              Het invoerbeeld moet een veelvoud van 8 pixels hebben
              (bv.: 256*256, 352*288, etc.)
     Uitvoer: Uitvoer is een array van 8x8 (in functie moet de array 9x9
              zijn om tijdelijke waarden te kunnen opslaan) die ook deel
              uitmaakt van een soortgelijke "balk" (fragm_bar_o).
              Hierin komen de geinterpoleerde fragmenten te staan.

 ---> frag_bar_i[0][0] +-----------------------------------+ 0
                       |        |        |        |        |
                       |        |        |        |        |
                       |        |        |        |        |
                       +-----------------------------------+ 8
 ---> frag_bar_o[0][0] +-----------------------------------+ 0
                       |        |        |        |        |
                       |        |        |        |        |
                       |        |        |        |        |
                       +-----------------------------------+ 7

                      fragment pixelnummering
        ■----------------------------------------------■
                   0| 1  2  3  4  5  6  7  8| 9 |
                    |                       |   |
                   .|10 11 12 13 14 15 16 17|18 |          invoer array is 9x9
                    |                       |   |
                   .|19 20 21 22 23 24 25 26|27 |          uitvoer array is 8x8
                    |                       |   |          en wordt aan eind
                   .|28 29 30 31 32 33 34 35|36 |          v/d functie gekwantiseerd
                    |                       |   |
                   .|37 38 39 40 41 42 43 44|45 |
                    |                       |   |
                   .|46 47 48 49 50 51 52 53|54 |
                    |                       |   |
                   .|55 56 57 58 59 60 61 62|63 |
                    |                       |   |
                   7|64 65 66 67 68 69 70 71|72 |
                    |-----------------------+-- |
                   8|73 74 75 76 77 78 79 80|81 |
    ■---------------------------------------------------■
   FRAGMENT   ---> i0......................i8
*/
void fragm_code(int i0)
{  int i1, i2, i3, i4, i5, i6, i7, i8;
   int k8, x, y;

   i1 = i0+1; i2 = i0+2; i3 = i0+3; i4 = i0+4;
   i5 = i0+5; i6 = i0+6; i7 = i0+7; i8 = i0+8; k8 = i8;
   if (i8 >= BREED) i8 = BREED-1;

/* Component I3- - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

   fragm_bar_o[0][i0] = fragm_bar_i[0][i0];                  /* pixel  1 */

/* Component I2 - I3 - - - - - - - - - - - - - - - - - - - - - - - - - - */

   fragm_bar_o[4][i0] =   fragm_bar_i[4][i0] -               /* pixel 37 */
                        ((fragm_bar_i[0][i0] + fragm_bar_i[8][i0]) / 2);
   fragm_bar_o[0][i4] =   fragm_bar_i[0][i4] -               /* pixel  5 */
                        ((fragm_bar_i[0][i0] + fragm_bar_i[0][i8]) / 2);
   fragm_bar_o[4][i4] =   fragm_bar_i[4][i4] -               /* pixel 41 */
                        ((fragm_bar_i[0][i0] + fragm_bar_i[0][i8] +
                          fragm_bar_i[8][i0] + fragm_bar_i[8][i8]) / 4);

/* Component I1 - I2 - - - - - - - - - - - - - - - - - - - - - - - - - - */

   fragm_bar_o[2][i0] =   fragm_bar_i[2][i0] -               /* pixel 19 */
                        ((fragm_bar_i[0][i0] + fragm_bar_i[4][i0]) / 2);
   fragm_bar_o[0][i2] =   fragm_bar_i[0][i2] -               /* pixel  3 */
                        ((fragm_bar_i[0][i0] + fragm_bar_i[0][i4]) / 2);
   fragm_bar_o[2][i2] =   fragm_bar_i[2][i2] -               /* pixel 21 */
                        ((fragm_bar_i[0][i0] + fragm_bar_i[0][i4] +
                          fragm_bar_i[4][i0] + fragm_bar_i[4][i4]) / 4);

   fragm_bar_o[2][i4] =   fragm_bar_i[2][i4] -               /* pixel 23 */
                        ((fragm_bar_i[0][i4] + fragm_bar_i[4][i4]) / 2);
   fragm_bar_o[0][i6] =   fragm_bar_i[0][i6] -               /* pixel  7 */
```

```
                       ((fragm_bar_i[0][i4] + fragm_bar_i[0][i8]) / 2);
   fragm_bar_o[2][i6] =   fragm_bar_i[2][i6] -              /* pixel 25 */
                       ((fragm_bar_i[0][i4] + fragm_bar_i[0][i8] +
                         fragm_bar_i[4][i4] + fragm_bar_i[4][i8]) / 4);

   fragm_bar_o[6][i0] =   fragm_bar_i[6][i0] -              /* pixel 55 */
                       ((fragm_bar_i[4][i0] + fragm_bar_i[8][i0]) / 2);
   fragm_bar_o[4][i2] =   fragm_bar_i[4][i2] -              /* pixel 39 */
                       ((fragm_bar_i[4][i0] + fragm_bar_i[4][i4]) / 2);
   fragm_bar_o[6][i2] =   fragm_bar_i[6][i2] -              /* pixel 57 */
                       ((fragm_bar_i[4][i0] + fragm_bar_i[4][i4] +
                         fragm_bar_i[8][i0] + fragm_bar_i[8][i4]) / 4);

   fragm_bar_o[6][i4] =   fragm_bar_i[6][i4] -              /* pixel 59 */
                       ((fragm_bar_i[4][i4] + fragm_bar_i[8][i4]) / 2);
   fragm_bar_o[4][i6] =   fragm_bar_i[4][i6] -              /* pixel 43 */
                       ((fragm_bar_i[4][i4] + fragm_bar_i[4][i8]) / 2);
   fragm_bar_o[6][i6] =   fragm_bar_i[6][i6] -              /* pixel 61 */
                       ((fragm_bar_i[4][i4] + fragm_bar_i[4][i8] +
                         fragm_bar_i[8][i4] + fragm_bar_i[8][i8]) / 4);

/* Component I0 - I1 - - - - - - - - - - - - - - - - - - - - - - - - - - */

   fragm_bar_o[0][i1] =   fragm_bar_i[0][i1] -              /* pixel  2 */
                       ((fragm_bar_i[0][i0] + fragm_bar_i[0][i2]) / 2);
   fragm_bar_o[0][i3] =   fragm_bar_i[0][i3] -              /* pixel  4 */
                       ((fragm_bar_i[0][i2] + fragm_bar_i[0][i4]) / 2);
   fragm_bar_o[0][i5] =   fragm_bar_i[0][i5] -              /* pixel  6 */
                       ((fragm_bar_i[0][i4] + fragm_bar_i[0][i6]) / 2);
   fragm_bar_o[0][i7] =   fragm_bar_i[0][i7] -              /* pixel  8 */
                       ((fragm_bar_i[0][i6] + fragm_bar_i[0][i8]) / 2);

/* Regel een - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

   fragm_bar_o[1][i0] =   fragm_bar_i[1][i0] -              /* pixel 10 */
                       ((fragm_bar_i[0][i0] + fragm_bar_i[2][i0]) / 2);
   fragm_bar_o[1][i1] =   fragm_bar_i[1][i1] -              /* pixel 11 */
                       ((fragm_bar_i[0][i0] + fragm_bar_i[0][i2] +
                         fragm_bar_i[2][i0] + fragm_bar_i[2][i2]) / 4);
   fragm_bar_o[1][i2] =   fragm_bar_i[1][i2] -              /* pixel 12 */
                       ((fragm_bar_i[0][i2] + fragm_bar_i[2][i2]) / 2);
   fragm_bar_o[1][i3] =   fragm_bar_i[1][i3] -              /* pixel 13 */
                       ((fragm_bar_i[0][i2] + fragm_bar_i[0][i4] +
                         fragm_bar_i[2][i2] + fragm_bar_i[2][i4]) / 4);
   fragm_bar_o[1][i4] =   fragm_bar_i[1][i4] -              /* pixel 14 */
                       ((fragm_bar_i[0][i4] + fragm_bar_i[2][i4]) / 2);
   fragm_bar_o[1][i5] =   fragm_bar_i[1][i5] -              /* pixel 15 */
                       ((fragm_bar_i[0][i4] + fragm_bar_i[0][i6] +
                         fragm_bar_i[2][i4] + fragm_bar_i[2][i6]) / 4);
   fragm_bar_o[1][i6] =   fragm_bar_i[1][i6] -              /* pixel 16 */
                       ((fragm_bar_i[0][i6] + fragm_bar_i[2][i6]) / 2);
   fragm_bar_o[1][i7] =   fragm_bar_i[1][i7] -              /* pixel 17 */
                       ((fragm_bar_i[0][i6] + fragm_bar_i[0][i8] +
                         fragm_bar_i[2][i6] + fragm_bar_i[2][i8]) / 4);

/* Regel twee - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/
   /* pixel 19 --> pixel 26 */
   fragm_bar_o[2][i1] =   fragm_bar_i[2][i1] -              /* pixel 20 */
                       ((fragm_bar_i[2][i0] + fragm_bar_i[2][i2]) / 2);
   fragm_bar_o[2][i3] =   fragm_bar_i[2][i3] -              /* pixel 22 */
                       ((fragm_bar_i[2][i2] + fragm_bar_i[2][i4]) / 2);
   fragm_bar_o[2][i5] =   fragm_bar_i[2][i5] -              /* pixel 24 */
                       ((fragm_bar_i[2][i4] + fragm_bar_i[2][i6]) / 2);
   fragm_bar_o[2][i7] =   fragm_bar_i[2][i7] -              /* pixel 26 */
                       ((fragm_bar_i[2][i6] + fragm_bar_i[2][i8]) / 2);

/* Regel drie - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/
   /* pixel 28 --> pixel 35 */
   fragm_bar_o[3][i0] =   fragm_bar_i[3][i0] -              /* pixel 28 */
                       ((fragm_bar_i[2][i0] + fragm_bar_i[4][i0]) / 2);
   fragm_bar_o[3][i1] =   fragm_bar_i[3][i1] -              /* pixel 29 */
                       ((fragm_bar_i[2][i0] + fragm_bar_i[2][i2] +
                         fragm_bar_i[4][i0] + fragm_bar_i[4][i2]) / 4);
   fragm_bar_o[3][i2] =   fragm_bar_i[3][i2] -              /* pixel 30 */
                       ((fragm_bar_i[2][i2] + fragm_bar_i[4][i2]) / 2);
   fragm_bar_o[3][i3] =   fragm_bar_i[3][i3] -              /* pixel 31 */
                       ((fragm_bar_i[2][i2] + fragm_bar_i[2][i4] +
                         fragm_bar_i[4][i2] + fragm_bar_i[4][i4]) / 4);
   fragm_bar_o[3][i4] =   fragm_bar_i[3][i4] -              /* pixel 32 */
                       ((fragm_bar_i[2][i4] + fragm_bar_i[4][i4]) / 2);
   fragm_bar_o[3][i5] =   fragm_bar_i[3][i5] -              /* pixel 33 */
                       ((fragm_bar_i[2][i4] + fragm_bar_i[2][i6] +
                         fragm_bar_i[4][i4] + fragm_bar_i[4][i6]) / 4);
   fragm_bar_o[3][i6] =   fragm_bar_i[3][i6] -              /* pixel 34 */
                       ((fragm_bar_i[2][i6] + fragm_bar_i[4][i6]) / 2);
```

```
   fragm_bar_o[3][i7] =   fragm_bar_i[3][i7] -              /* pixel 35 */
                         ((fragm_bar_i[2][i6] + fragm_bar_i[2][i8] +
                           fragm_bar_i[4][i6] + fragm_bar_i[4][i8]) / 4);

/* Regel vier - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/
   /* pixel 37 --> pixel 44 */
   fragm_bar_o[4][i1] =   fragm_bar_i[4][i1] -              /* pixel 38 */
                         ((fragm_bar_i[4][i0] + fragm_bar_i[4][i2]) / 2);
   fragm_bar_o[4][i3] =   fragm_bar_i[4][i3] -              /* pixel 40 */
                         ((fragm_bar_i[4][i2] + fragm_bar_i[4][i4]) / 2);
   fragm_bar_o[4][i5] =   fragm_bar_i[4][i5] -              /* pixel 42 */
                         ((fragm_bar_i[4][i4] + fragm_bar_i[4][i6]) / 2);
   fragm_bar_o[4][i7] =   fragm_bar_i[4][i7] -              /* pixel 44 */
                         ((fragm_bar_i[4][i6] + fragm_bar_i[4][i8]) / 2);

/* Regel vijf - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/
   /* pixel 46 --> pixel 53 */
   fragm_bar_o[5][i0] =   fragm_bar_i[5][i0] -              /* pixel 46 */
                         ((fragm_bar_i[4][i0] + fragm_bar_i[6][i0]) / 2);
   fragm_bar_o[5][i1] =   fragm_bar_i[5][i1] -              /* pixel 47 */
                         ((fragm_bar_i[4][i0] + fragm_bar_i[4][i2] +
                           fragm_bar_i[6][i0] + fragm_bar_i[6][i2]) / 4);
   fragm_bar_o[5][i2] =   fragm_bar_i[5][i2] -              /* pixel 48 */
                         ((fragm_bar_i[4][i2] + fragm_bar_i[6][i2]) / 2);
   fragm_bar_o[5][i3] =   fragm_bar_i[5][i3] -              /* pixel 49 */
                         ((fragm_bar_i[4][i2] + fragm_bar_i[4][i4] +
                           fragm_bar_i[6][i2] + fragm_bar_i[6][i4]) / 4);
   fragm_bar_o[5][i4] =   fragm_bar_i[5][i4] -              /* pixel 50 */
                         ((fragm_bar_i[4][i4] + fragm_bar_i[6][i4]) / 2);
   fragm_bar_o[5][i5] =   fragm_bar_i[5][i5] -              /* pixel 51 */
                         ((fragm_bar_i[4][i4] + fragm_bar_i[4][i6] +
                           fragm_bar_i[6][i4] + fragm_bar_i[6][i6]) / 4);
   fragm_bar_o[5][i6] =   fragm_bar_i[5][i6] -              /* pixel 52 */
                         ((fragm_bar_i[4][i6] + fragm_bar_i[6][i6]) / 2);
   fragm_bar_o[5][i7] =   fragm_bar_i[5][i7] -              /* pixel 53 */
                         ((fragm_bar_i[4][i6] + fragm_bar_i[4][i8] +
                           fragm_bar_i[6][i6] + fragm_bar_i[6][i8]) / 4);

/* Regel zes - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
   /* pixel 55 --> pixel 62 */
   fragm_bar_o[6][i1] =   fragm_bar_i[6][i1] -              /* pixel 56 */
                         ((fragm_bar_i[6][i0] + fragm_bar_i[6][i2]) / 2);
   fragm_bar_o[6][i3] =   fragm_bar_i[6][i3] -              /* pixel 58 */
                         ((fragm_bar_i[6][i2] + fragm_bar_i[6][i4]) / 2);
   fragm_bar_o[6][i5] =   fragm_bar_i[6][i5] -              /* pixel 60 */
                         ((fragm_bar_i[6][i4] + fragm_bar_i[6][i6]) / 2);
   fragm_bar_o[6][i7] =   fragm_bar_i[6][i7] -              /* pixel 62 */
                         ((fragm_bar_i[6][i6] + fragm_bar_i[6][i8]) / 2);

/* Regel zeven - - - - - - - - - - - - - - - - - - - - - - - - - - - */
   /* pixel 64 --> pixel 71 */
   fragm_bar_o[7][i0] =   fragm_bar_i[7][i0] -              /* pixel 64 */
                         ((fragm_bar_i[6][i0] + fragm_bar_i[8][i0]) / 2);
   fragm_bar_o[7][i1] =   fragm_bar_i[7][i1] -              /* pixel 65 */
                         ((fragm_bar_i[6][i0] + fragm_bar_i[6][i2] +
                           fragm_bar_i[8][i0] + fragm_bar_i[8][i2]) / 4);
   fragm_bar_o[7][i2] =   fragm_bar_i[7][i2] -              /* pixel 66 */
                         ((fragm_bar_i[6][i2] + fragm_bar_i[8][i2]) / 2);
   fragm_bar_o[7][i3] =   fragm_bar_i[7][i3] -              /* pixel 67 */
                         ((fragm_bar_i[6][i2] + fragm_bar_i[6][i4] +
                           fragm_bar_i[8][i2] + fragm_bar_i[8][i4]) / 4);
   fragm_bar_o[7][i4] =   fragm_bar_i[7][i4] -              /* pixel 68 */
                         ((fragm_bar_i[6][i4] + fragm_bar_i[8][i4]) / 2);
   fragm_bar_o[7][i5] =   fragm_bar_i[7][i5] -              /* pixel 69 */
                         ((fragm_bar_i[6][i4] + fragm_bar_i[6][i6] +
                           fragm_bar_i[8][i4] + fragm_bar_i[8][i6]) / 4);
   fragm_bar_o[7][i6] =   fragm_bar_i[7][i6] -              /* pixel 70 */
                         ((fragm_bar_i[6][i6] + fragm_bar_i[8][i6]) / 2);
   fragm_bar_o[7][i7] =   fragm_bar_i[7][i7] -              /* pixel 71 */
                         ((fragm_bar_i[6][i6] + fragm_bar_i[6][i8] +
                           fragm_bar_i[8][i6] + fragm_bar_i[8][i8]) / 4);

/* Laatste kolom - - - - - - - - - - - - - - - - - - - - - - - - - - */
   /* pixels  9, 18, 27, 36, 45, 54, 63, 72  rechterkant van beeld/balk  */
   if (k8 >= BREED)
     {  fragm_bar_o[0][BREED] =   fragm_bar_i[0][i8];
        fragm_bar_o[1][BREED] =   fragm_bar_i[1][i8] -
                                 ((fragm_bar_i[0][i8]+fragm_bar_i[2][i8]) / 2);
        fragm_bar_o[2][BREED] =   fragm_bar_i[2][i8] -
                                 ((fragm_bar_i[0][i8]+fragm_bar_i[4][i8]) / 2);
        fragm_bar_o[3][BREED] =   fragm_bar_i[3][i8] -
                                 ((fragm_bar_i[2][i8]+fragm_bar_i[4][i8]) / 2);
        fragm_bar_o[4][BREED] =   fragm_bar_i[4][i8] -
                                 ((fragm_bar_i[0][i8]+fragm_bar_i[8][i8]) / 2);
        fragm_bar_o[5][BREED] =   fragm_bar_i[5][i8] -
```

```
                                ((fragm_bar_i[4][i8]+fragm_bar_i[6][i8]) / 2);
        fragm_bar_o[6][BREED] =   fragm_bar_i[6][i8] -
                                ((fragm_bar_i[4][i8]+fragm_bar_i[8][i8]) / 2);
        fragm_bar_o[7][BREED] =   fragm_bar_i[7][i8] -
                                ((fragm_bar_i[6][i8]+fragm_bar_i[8][i8]) / 2);

        /* Kwantiseer laatste kolom */
        fragm_bar_o[0][BREED] = fragm_bar_o[0][BREED];
        fragm_bar_o[1][BREED] = quant_I01(fragm_bar_o[1][BREED]);
        fragm_bar_o[2][BREED] = quant_I12(fragm_bar_o[2][BREED]);
        fragm_bar_o[3][BREED] = quant_I01(fragm_bar_o[3][BREED]);
        fragm_bar_o[4][BREED] = quant_I23(fragm_bar_o[4][BREED]);
        fragm_bar_o[5][BREED] = quant_I01(fragm_bar_o[5][BREED]);
        fragm_bar_o[6][BREED] = quant_I12(fragm_bar_o[6][BREED]);
        fragm_bar_o[7][BREED] = quant_I01(fragm_bar_o[7][BREED]);
     }

/* Kwantiseer fragment- - - - - - - - - - - - - - - - - - - - - - - - - -*/

   /* I3 in 256 niveaus kwantiseren (= kopieren waarde) */
   fragm_bar_o[0][i0] = fragm_bar_o[0][i0];

   /* I2-I3 in 15 niveaus kwantiseren */
   fragm_bar_o[0][i4] = quant_I23(fragm_bar_o[0][i4]);
   fragm_bar_o[4][i0] = quant_I23(fragm_bar_o[4][i0]);
   fragm_bar_o[4][i4] = quant_I23(fragm_bar_o[4][i4]);

   /* I1-I2 in 5 niveaus kwantiseren */
   fragm_bar_o[2][i0] = quant_I12(fragm_bar_o[2][i0]);
   fragm_bar_o[0][i2] = quant_I12(fragm_bar_o[0][i2]);
   fragm_bar_o[2][i2] = quant_I12(fragm_bar_o[2][i2]);
   fragm_bar_o[2][i4] = quant_I12(fragm_bar_o[2][i4]);
   fragm_bar_o[0][i6] = quant_I12(fragm_bar_o[0][i6]);
   fragm_bar_o[2][i6] = quant_I12(fragm_bar_o[2][i6]);
   fragm_bar_o[6][i0] = quant_I12(fragm_bar_o[6][i0]);
   fragm_bar_o[4][i2] = quant_I12(fragm_bar_o[4][i2]);
   fragm_bar_o[6][i2] = quant_I12(fragm_bar_o[6][i2]);
   fragm_bar_o[6][i4] = quant_I12(fragm_bar_o[6][i4]);
   fragm_bar_o[4][i6] = quant_I12(fragm_bar_o[4][i6]);
   fragm_bar_o[6][i6] = quant_I12(fragm_bar_o[6][i6]);

   /* I0-I1 in 2 niveaus kwantiseren */
   for (y = 0; y < 8; y++)       /* zeef component I0-I1 eruit */
     for (x = i0; x < k8; x++)
       if ( ((x & 1) != 0) || ((y & 1) != 0) )  /* oneven x of oneven y */
         fragm_bar_o[y][x] = quant_I01(fragm_bar_o[y][x]);
}
```

```
/***********************************************************************/
/*  Filenaam: QUANT.C                                                  */
/*  Funties om een pixel te kwantiseren                                */
/***********************************************************************/
/* Geef hier het aantal kwantisatieniveaus op */
#define N_I23    15
#define N_I12     5
#define N_I01     2

/* Geef hier de kwantiesatieniveaus op en de tussenliggende schakelniveaus */

/* I2-I3 component kwantisator */
int M_I23[] = { 2, 5, 9, 15, 23, 37, 68 };              /* Midden */
int L_I23[] = { 0, 4, 7, 12, 18, 28, 46, 90 };          /* Levels */

/* I1-I2 component kwantisator */
int M_I12[] = { 8, 38 };                                /* Midden */
int L_I12[] = { 0, 17, 60 };                            /* Levels */

/* I0-I1 component kwantisator */
int M_I01[] = { 0 };                                    /* Midden */
int L_I01[] = { 0, 1 };                                 /* Level(s) */


#include <stdio.h>
#include <stdlib.h>
#include "const.h"

#if ( (N_I23 % 2) || (N_I23 < 3) )
#define MID_I23  0
#else
#define MID_I23  1
#endif

#if ( (N_I12 % 2) || (N_I12 < 3) )
#define MID_I12  0
#else
#define MID_I12  1
#endif

#if ( (N_I01 % 2) || (N_I01 < 3) )
#define MID_I01  0
#else
#define MID_I01  1
#endif

#define MAX_I23  N_I23/2
#define MAX_I12  N_I12/2
#define MAX_I01  N_I01/2


int quant_I23(int value)
{  int v, sign, midden = MID_I23;

   if (!QUANT) return value;
   v = abs(value);
   sign = (v? value/v : 1);                    /* Voorkom delen door nul! */
   do
   {  if (v >= M_I23[midden])                  /* naar boven afronden? */
        midden++;                              /* ja, pak hoger midden */
      else
        break;                                 /* nee, dan stoppen     */
   } while ( midden < MAX_I23 );
   return sign * L_I23[midden];
}

int quant_I12(int value)
{  int v, sign, midden = MID_I12;

   if (!QUANT) return value;
   v = abs(value);
   sign = (v? value/v : 1);                    /* Voorkom delen door nul! */
   do
   {  if (v >= M_I12[midden])                  /* naar boven afronden? */
        midden++;                              /* ja, pak hoger midden */
      else
        break;                                 /* nee, dan stoppen     */
   } while ( midden < MAX_I12 );
   return sign * L_I12[midden];
}
```

```
int quant_I01(int value)
{  int v, sign, midden = MID_I01;

   if (!QUANT) return value;
   v = abs(value);
   sign = (v? value/v : 1);                    /* Voorkom delen door nul! */
   do
   {  if (v >= M_I01[midden])                  /* naar boven afronden? */
        midden++;                              /* ja, pak hoger midden */
      else
        break;                                 /* nee, dan stoppen     */
   } while ( midden < MAX_I01 );
   return sign * L_I01[midden];
}
```

```
/**********************************************************************/
/*  Filenaam: DECODEER.C                                              */
/*  Deze functie decodeerd de invoerfile en zet de uitvoer in         */
/*  de file met de naam DECODEDFRAME.xxx.                             */
/*  DECODEDFRAME is een constante op te geven in de file const.h      */
/*  xxx is het nummer van de frame (frameno).                         */
/*  DE AFMETINGEN VAN HET FRAME ZIJN BEPAALD DOOR DE CONSTANTEN       */
/*  'BREED' EN 'HOOG'. HET FRAME MOET EEN VEELVOUD VAN 8 PIXELS HEBBEN.     */
/**********************************************************************/
#include "const.h"
#include <stdio.h>
#include <stdlib.h>


/*                G L O B A L E   V A R I A B E L E N               */
/*                ===================================               */
extern int fragm_bar_i[9][MAXLINE];
extern int fragm_bar_o[9][MAXLINE];

/* Forward declaration van gebruikte funties */
void fragm_decode(int i0);


int decodeer(int frameno)
{  FILE  *rframe, *wframe;
   char  invoerfile[80], uitvoerfile[80];
   int   j0, i0;
   int   begin_lijn, pixel, lijn, gelezen;

   /* Maak de filenaam waar het gecodeerde frame in staat */
   sprintf(invoerfile, "%s%03d", CODEDFRAME, frameno);

   /* Maak de filenaam waar het gedecodeerde frame in komt te staan */
   sprintf(uitvoerfile, "%s%03d", DECODEDFRAME, frameno);

   /* Open invoerfile voor lezen */
   if ( (rframe=fopen(invoerfile, "rb")) == NULL ) return(1);

   /* Open uitvoerfile voor schrijven */
   if ( (wframe=fopen(uitvoerfile, "wb")) == NULL ) return(2);

   /* Eerste lijn waar de pixels in fragm_bar_i gezet worden */
   begin_lijn = 0;

/* Besturing van het coderen in de hoogte-richting
  0  +-------+ j0
  .  | fragm |  .
  .  |   1   |  .
  8  +-------+ j8/j0
  .  | ..... |   .                    +----------------------+
  .  |       |   .                    |                      | -
  .  +-------+ j8/j0                  |                      | ^
  .  | ..... |   .                    |     hele beeld       | |
  .  |       |   .                    |                      | HOOG
  .  +-------+ j8/j0                  |                      | |
  .  | ..... |   .                    |                      | v
  .  |       |   .                    |                      | -
 hoog+-------+ j8/j0                  +----------------------+
                                      |<------- BREED ----->|

*/
   for (j0 = 0; j0 < HOOG; j0 += 8)       /* Lus in y-richting */
   {   /* Lees eerste keer 9 beeldlijnen, 9de lijn nodig om 8ste lijn te
          kunnen decoderen 9de lijn is daarna steeds de 1ste lijn in
          de volgende balk. Eerste keer 9 lijnen lezen daarna steeds 8 lijnen.
          Beeldlijn voor beeldlijn inlezen.
       */
       for (lijn=begin_lijn; lijn<9; lijn++)
       {  gelezen = fread(&fragm_bar_i[lijn][0], sizeof(int), BREED+1, rframe);
          if (gelezen != BREED+1) return(3);
       }

       /* Na de eerste keer data vanaf regel een in fragm_bar_i zetten */
       begin_lijn = 1;

       /* Decodeerlus voor een balk met breedte van het beeld en
          hoogte 9 pixels
       i0=0     i0=8                                    i0
          +-------+------+-------+-------+-------+-------+-------+
          | fragm | fragm| ....  | ....  | ..... | ....  | fragm |
          |   1   |   2  |       |       |       |       |   x   |
          +-------+------+-------+-------+-------+-------+-------+
       */
       for (i0 = 0; i0<BREED; i0 += 8) fragm_decode(i0);
```

```
      /* Bewaar gedecodeerde balk (lijn voor lijn wegschrijven) */
      for (lijn=0; lijn<8; lijn++)
         fwrite(&fragm_bar_o[lijn][0], sizeof(int), BREED, wframe);

      /* Kopieer regel 9 naar regel 1 voor de volgende balk */
      for (pixel=0; pixel<BREED+1; pixel++)
        fragm_bar_i[0][pixel] = fragm_bar_i[8][pixel];
    }

   printf("\b ");
   /* Close al files */
   fclose(rframe);
   fclose(wframe);
   return(0);
}


/*              Component Interpolation Decoding of Fragment              */
/*              ============================================              */
/*   Invoer:  Invoer array van 9x9 die deel uit maakt van een "balk"
              (fragm_bar_i) van deze array's naast elkaar. De waarde i0
              is de horizontale uitwijking naar rechts van de linkerboven
              waarde in deze array van 9x9.
              Het invoerbeeld moet een veelvoud van 8 pixels hebben
              (bv.: 256*256, 352*288, etc.)
     Uitvoer: Uitvoer is een array van 8x8 (in functie moet de array 9x9
              zijn om tijdelijke waarden te kunnen opslaan) die ook deel
              uitmaakt van een soortgelijke "balk" (fragm_bar_o).
      ■───────────┬────────────────────────────┬───────■
               j0│ 1  2  3  4  5  6  7  8| 9 │
                 │                        |    │
                 │10 11 12 13 14 15 16 17|18 │     invoer array is 9x9
                 │                        |    │
                 │19 20 21 22 23 24 25 26|27 │     uitvoer array is 8x8
                 │                        |    │
                 │28 29 30 31 32 33 34 35|36 │
                 │                        |    │
                 │37 38 39 40 41 42 43 44|45 │
                 │                        |    │
                 │46 47 48 49 50 51 52 53|54 │
                 │                        |    │
                 │55 56 57 58 59 60 61 62|63 │
                 │                        |    │
                 │64 65 66 67 68 69 70 71|72 │
                 │-----------------------+-- │
                 │73 74 75 76 77 78 79 80|81 │
   ■─────────────┴────────────────────────────┴─────────■
                             FRAGMENT
*/
void fragm_decode(int i0)
{  int i1, i2, i3, i4, i5, i6, i7, i8;

   i1 = i0+1; i2 = i0+2; i3 = i0+3; i4 = i0+4;
   i5 = i0+5; i6 = i0+6; i7 = i0+7; i8 = i0+8;

/* component I3 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

   fragm_bar_o[0][i0] = fragm_bar_i[0][i0];                   /* pixel  1 */
   fragm_bar_o[0][i8] = fragm_bar_i[0][i8];                   /* pixel  9 */
   fragm_bar_o[8][i0] = fragm_bar_i[8][i0];                   /* pixel 73 */
   fragm_bar_o[8][i8] = fragm_bar_i[8][i8];                   /* pixel 81 */

/* component I2 - I3 - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/

   fragm_bar_o[4][i0] =   fragm_bar_i[4][i0] +                /* pixel 37 */
                      ((fragm_bar_i[0][i0] + fragm_bar_i[8][i0]) / 2);
   fragm_bar_o[4][i8] =   fragm_bar_i[4][i8] +                /* pixel 45 */
                      ((fragm_bar_i[0][i8] + fragm_bar_i[8][i8]) / 2);
   fragm_bar_o[0][i4] =   fragm_bar_i[0][i4] +                /* pixel  5 */
                      ((fragm_bar_i[0][i0] + fragm_bar_i[0][i8]) / 2);
   fragm_bar_o[8][i4] =   fragm_bar_i[8][i4] +                /* pixel 77 */
                      ((fragm_bar_i[8][i0] + fragm_bar_i[8][i8]) / 2);
   fragm_bar_o[4][i4] =   fragm_bar_i[4][i4] +                /* pixel 41 */
                      ((fragm_bar_i[0][i0] + fragm_bar_i[8][i0] +
                        fragm_bar_i[0][i8] + fragm_bar_i[8][i8]) / 4);

/* component I1 - I2 - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/

   fragm_bar_o[2][i0] =   fragm_bar_i[2][i0] +                /* pixel 19 */
                      ((fragm_bar_o[0][i0] + fragm_bar_o[4][i0]) / 2);
   fragm_bar_o[0][i2] =   fragm_bar_i[0][i2] +                /* pixel  3 */
                      ((fragm_bar_o[0][i0] + fragm_bar_o[0][i4]) / 2);
   fragm_bar_o[2][i2] =   fragm_bar_i[2][i2] +                /* pixel 21 */
                      ((fragm_bar_o[0][i0] + fragm_bar_o[4][i0] +
                        fragm_bar_o[0][i4] + fragm_bar_o[4][i4]) / 4);
```

```
    fragm_bar_o[2][i4] =   fragm_bar_i[2][i4] +                 /* pixel 23 */
                         ((fragm_bar_o[0][i4] + fragm_bar_o[4][i4]) / 2);
    fragm_bar_o[0][i6] =   fragm_bar_i[0][i6] +                 /* pixel  7 */
                         ((fragm_bar_o[0][i4] + fragm_bar_o[0][i8]) / 2);
    fragm_bar_o[2][i6] =   fragm_bar_i[2][i6] +                 /* pixel 25 */
                         ((fragm_bar_o[0][i4] + fragm_bar_o[4][i4] +
                           fragm_bar_o[0][i8] + fragm_bar_o[4][i8]) / 4);

    fragm_bar_o[6][i0] =   fragm_bar_i[6][i0] +                 /* pixel 55 */
                         ((fragm_bar_o[4][i0] + fragm_bar_o[8][i0]) / 2);
    fragm_bar_o[4][i2] =   fragm_bar_i[4][i2] +                 /* pixel 39 */
                         ((fragm_bar_o[4][i0] + fragm_bar_o[4][i4]) / 2);
    fragm_bar_o[6][i2] =   fragm_bar_i[6][i2] +                 /* pixel 57 */
                         ((fragm_bar_o[4][i0] + fragm_bar_o[8][i0] +
                           fragm_bar_o[4][i4] + fragm_bar_o[8][i4]) / 4);

    fragm_bar_o[6][i4] =   fragm_bar_i[6][i4] +                 /* pixel 59 */
                         ((fragm_bar_o[4][i4] + fragm_bar_o[8][i4]) / 2);
    fragm_bar_o[4][i6] =   fragm_bar_i[4][i6] +                 /* pixel 43 */
                         ((fragm_bar_o[4][i4] + fragm_bar_o[4][i8]) / 2);
    fragm_bar_o[6][i6] =   fragm_bar_i[6][i6] +                 /* pixel 61 */
                         ((fragm_bar_o[4][i4] + fragm_bar_o[8][i4] +
                           fragm_bar_o[4][i8] + fragm_bar_o[8][i8]) / 4);

    fragm_bar_o[2][i8] =   fragm_bar_i[2][i8] +                 /* pixel 27 */
                         ((fragm_bar_o[0][i8] + fragm_bar_o[4][i8]) / 2);
    fragm_bar_o[6][i8] =   fragm_bar_i[6][i8] +                 /* pixel 63 */
                         ((fragm_bar_o[4][i8] + fragm_bar_o[8][i8]) / 2);

    fragm_bar_o[8][i2] =   fragm_bar_i[8][i2] +                 /* pixel 75 */
                         ((fragm_bar_o[8][i0] + fragm_bar_o[8][i4]) / 2);
    fragm_bar_o[8][i6] =   fragm_bar_i[8][i6] +                 /* pixel 79 */
                         ((fragm_bar_o[8][i4] + fragm_bar_o[8][i8]) / 2);

/* component I0 - I1 - - - - - - - - - - - - - - - - - - - - - - - - - - -*/

   fragm_bar_o[1][i0] =   fragm_bar_i[1][i0] +                  /* pixel  2 */
                        ((fragm_bar_o[0][i0] + fragm_bar_o[2][i0]) / 2);
   fragm_bar_o[0][i1] =   fragm_bar_i[0][i1] +                  /* pixel 10 */
                        ((fragm_bar_o[0][i0] + fragm_bar_o[0][i2]) / 2);
   fragm_bar_o[1][i1] =   fragm_bar_i[1][i1] +                  /* pixel 11 */
                        ((fragm_bar_o[0][i0] + fragm_bar_o[2][i0] +
                          fragm_bar_o[0][i2] + fragm_bar_o[2][i2]) / 4);

   fragm_bar_o[1][i2] =   fragm_bar_i[1][i2] +                  /* pixel 12 */
                        ((fragm_bar_o[0][i2] + fragm_bar_o[2][i2]) / 2);
   fragm_bar_o[0][i3] =   fragm_bar_i[0][i3] +                  /* pixel  4 */
                        ((fragm_bar_o[0][i2] + fragm_bar_o[0][i4]) / 2);
   fragm_bar_o[1][i3] =   fragm_bar_i[1][i3] +                  /* pixel 13 */
                        ((fragm_bar_o[0][i2] + fragm_bar_o[2][i2] +
                          fragm_bar_o[0][i4] + fragm_bar_o[2][i4]) / 4);

   fragm_bar_o[1][i4] =   fragm_bar_i[1][i4] +                  /* pixel 14 */
                        ((fragm_bar_o[0][i4] + fragm_bar_o[2][i4]) / 2);
   fragm_bar_o[0][i5] =   fragm_bar_i[0][i5] +                  /* pixel  6 */
                        ((fragm_bar_o[0][i4] + fragm_bar_o[0][i6]) / 2);
   fragm_bar_o[1][i5] =   fragm_bar_i[1][i5] +                  /* pixel 15 */
                        ((fragm_bar_o[0][i4] + fragm_bar_o[2][i4] +
                          fragm_bar_o[0][i6] + fragm_bar_o[2][i6]) / 4);

   fragm_bar_o[1][i6] =   fragm_bar_i[1][i6] +                  /* pixel 16 */
                        ((fragm_bar_o[0][i6] + fragm_bar_o[2][i6]) / 2);
   fragm_bar_o[0][i7] =   fragm_bar_i[0][i7] +                  /* pixel  8 */
                        ((fragm_bar_o[0][i6] + fragm_bar_o[0][i8]) / 2);
   fragm_bar_o[1][i7] =   fragm_bar_i[1][i7] +                  /* pixel 17 */
                        ((fragm_bar_o[0][i6] + fragm_bar_o[2][i6] +
                          fragm_bar_o[0][i8] + fragm_bar_o[2][i8]) / 4);

/*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

   fragm_bar_o[3][i0] =   fragm_bar_i[3][i0] +                  /* pixel 28 */
                        ((fragm_bar_o[2][i0] + fragm_bar_o[4][i0]) / 2);
   fragm_bar_o[2][i1] =   fragm_bar_i[2][i1] +                  /* pixel 20 */
                        ((fragm_bar_o[2][i0] + fragm_bar_o[2][i2]) / 2);
   fragm_bar_o[3][i1] =   fragm_bar_i[3][i1] +                  /* pixel 29 */
                        ((fragm_bar_o[2][i0] + fragm_bar_o[4][i0] +
                          fragm_bar_o[2][i2] + fragm_bar_o[4][i2]) / 4);

   fragm_bar_o[3][i2] =   fragm_bar_i[3][i2] +                  /* pixel 30 */
                        ((fragm_bar_o[2][i2] + fragm_bar_o[4][i2]) / 2);
   fragm_bar_o[2][i3] =   fragm_bar_i[2][i3] +                  /* pixel 22 */
                        ((fragm_bar_o[2][i2] + fragm_bar_o[2][i4]) / 2);
   fragm_bar_o[3][i3] =   fragm_bar_i[3][i3] +                  /* pixel 31 */
                        ((fragm_bar_o[2][i2] + fragm_bar_o[4][i2] +
                          fragm_bar_o[2][i4] + fragm_bar_o[4][i4]) / 4);
```

```
    fragm_bar_o[3][i4] =   fragm_bar_i[3][i4] +                 /* pixel 32 */
                         ((fragm_bar_o[2][i4] + fragm_bar_o[4][i4]) / 2);
    fragm_bar_o[2][i5] =   fragm_bar_i[2][i5] +                 /* pixel 24 */
                         ((fragm_bar_o[2][i4] + fragm_bar_o[2][i6]) / 2);
    fragm_bar_o[3][i5] =   fragm_bar_i[3][i5] +                 /* pixel 33 */
                         ((fragm_bar_o[2][i4] + fragm_bar_o[4][i4] +
                           fragm_bar_o[2][i6] + fragm_bar_o[4][i6]) / 4);

    fragm_bar_o[3][i6] =   fragm_bar_i[3][i6] +                 /* pixel 34 */
                         ((fragm_bar_o[2][i6] + fragm_bar_o[4][i6]) / 2);
    fragm_bar_o[2][i7] =   fragm_bar_i[2][i7] +                 /* pixel 26 */
                         ((fragm_bar_o[2][i6] + fragm_bar_o[2][i8]) / 2);
    fragm_bar_o[3][i7] =   fragm_bar_i[3][i7] +                 /* pixel 35 */
                         ((fragm_bar_o[2][i6] + fragm_bar_o[4][i6] +
                           fragm_bar_o[2][i8] + fragm_bar_o[4][i8]) / 4);

  /*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

    fragm_bar_o[5][i0] =   fragm_bar_i[5][i0] +                 /* pixel 46 */
                         ((fragm_bar_o[4][i0] + fragm_bar_o[6][i0]) / 2);
    fragm_bar_o[4][i1] =   fragm_bar_i[4][i1] +                 /* pixel 38 */
                         ((fragm_bar_o[4][i0] + fragm_bar_o[4][i2]) / 2);
    fragm_bar_o[5][i1] =   fragm_bar_i[5][i1] +                 /* pixel 47 */
                         ((fragm_bar_o[4][i0] + fragm_bar_o[6][i0] +
                           fragm_bar_o[4][i2] + fragm_bar_o[6][i2]) / 4);

    fragm_bar_o[5][i2] =   fragm_bar_i[5][i2] +                 /* pixel 48 */
                         ((fragm_bar_o[4][i2] + fragm_bar_o[6][i2]) / 2);
    fragm_bar_o[4][i3] =   fragm_bar_i[4][i3] +                 /* pixel 40 */
                         ((fragm_bar_o[4][i2] + fragm_bar_o[4][i4]) / 2);
    fragm_bar_o[5][i3] =   fragm_bar_i[5][i3] +                 /* pixel 49 */
                         ((fragm_bar_o[4][i2] + fragm_bar_o[6][i2] +
                           fragm_bar_o[4][i4] + fragm_bar_o[6][i4]) / 4);

    fragm_bar_o[5][i4] =   fragm_bar_i[5][i4] +                 /* pixel 50 */
                         ((fragm_bar_o[4][i4] + fragm_bar_o[6][i4]) / 2);
    fragm_bar_o[4][i5] =   fragm_bar_i[4][i5] +                 /* pixel 42 */
                         ((fragm_bar_o[4][i4] + fragm_bar_o[4][i6]) / 2);
    fragm_bar_o[5][i5] =   fragm_bar_i[5][i5] +                 /* pixel 51 */
                         ((fragm_bar_o[4][i4] + fragm_bar_o[6][i4] +
                           fragm_bar_o[4][i6] + fragm_bar_o[6][i6]) / 4);

    fragm_bar_o[5][i6] =   fragm_bar_i[5][i6] +                 /* pixel 52 */
                         ((fragm_bar_o[4][i6] + fragm_bar_o[6][i6]) / 2);
    fragm_bar_o[4][i7] =   fragm_bar_i[4][i7] +                 /* pixel 44 */
                         ((fragm_bar_o[4][i6] + fragm_bar_o[4][i8]) / 2);
    fragm_bar_o[5][i7] =   fragm_bar_i[5][i7] +                 /* pixel 53 */
                         ((fragm_bar_o[4][i6] + fragm_bar_o[6][i6] +
                           fragm_bar_o[4][i8] + fragm_bar_o[6][i8]) / 4);

  /*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

    fragm_bar_o[7][i0] =   fragm_bar_i[7][i0] +                 /* pixel 64 */
                         ((fragm_bar_o[6][i0] + fragm_bar_o[8][i0]) / 2);
    fragm_bar_o[6][i1] =   fragm_bar_i[6][i1] +                 /* pixel 56 */
                         ((fragm_bar_o[6][i0] + fragm_bar_o[6][i2]) / 2);
    fragm_bar_o[7][i1] =   fragm_bar_i[7][i1] +                 /* pixel 65 */
                         ((fragm_bar_o[6][i0] + fragm_bar_o[8][i0] +
                           fragm_bar_o[6][i2] + fragm_bar_o[8][i2]) / 4);

    fragm_bar_o[7][i2] =   fragm_bar_i[7][i2] +                 /* pixel 66 */
                         ((fragm_bar_o[6][i2] + fragm_bar_o[8][i2]) / 2);
    fragm_bar_o[6][i3] =   fragm_bar_i[6][i3] +                 /* pixel 58 */
                         ((fragm_bar_o[6][i2] + fragm_bar_o[6][i4]) / 2);
    fragm_bar_o[7][i3] =   fragm_bar_i[7][i3] +                 /* pixel 67 */
                         ((fragm_bar_o[6][i2] + fragm_bar_o[8][i2] +
                           fragm_bar_o[6][i4] + fragm_bar_o[8][i4]) / 4);

    fragm_bar_o[7][i4] =   fragm_bar_i[7][i4] +                 /* pixel 68 */
                         ((fragm_bar_o[6][i4] + fragm_bar_o[8][i4]) / 2);
    fragm_bar_o[6][i5] =   fragm_bar_i[6][i5] +                 /* pixel 60 */
                         ((fragm_bar_o[6][i4] + fragm_bar_o[6][i6]) / 2);
    fragm_bar_o[7][i5] =   fragm_bar_i[7][i5] +                 /* pixel 69 */
                         ((fragm_bar_o[6][i4] + fragm_bar_o[8][i4] +
                           fragm_bar_o[6][i6] + fragm_bar_o[8][i6]) / 4);

    fragm_bar_o[7][i6] =   fragm_bar_i[7][i6] +                 /* pixel 70 */
                         ((fragm_bar_o[6][i6] + fragm_bar_o[8][i6]) / 2);
    fragm_bar_o[6][i7] =   fragm_bar_i[6][i7] +                 /* pixel 62 */
                         ((fragm_bar_o[6][i6] + fragm_bar_o[6][i8]) / 2);
    fragm_bar_o[7][i7] =   fragm_bar_i[7][i7] +                 /* pixel 71 */
                         ((fragm_bar_o[6][i6] + fragm_bar_o[8][i6] +
                           fragm_bar_o[6][i8] + fragm_bar_o[8][i8]) / 4);
  }
```

```
/*************************************************************************/
/*  Filenaam: ADD.C                                                      */
/*  Bepaal het verschil tussen het frame met nummer "frameno"            */
/*  en het voorgaande frame met nummer "frameno-stap"                    */
/*************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include "const.h"


int sommeer(char sequence_out[80], int frameno)
{  char          frame0[80], frame1[80], uitvoerfile[80];
   FILE          *f0, *f1, *ftemp;
   int           *buf1;
   unsigned char *buf0, *bufuit;
   int           n, m, tijdelijk;

   /* Bepaal filenaam van voorgaande frame en huidig frame en open ze  */
   /* voorgaande frame */
   if ( frameno != 1 )
   {  sprintf(frame0, "%s%03d", sequence_out, frameno-1);
      f0 = fopen(frame0, "rb");
      if (f0 == NULL) return(1);
   }

   /* huidige frame */
   sprintf(frame1, "%s%03d", DECODEDFRAME, frameno);
   f1 = fopen(frame1, "rb");
   if (f1 == NULL) return(1);

   /* filenaam van somframe */
   sprintf(uitvoerfile, "%s%03d", sequence_out, frameno);
   ftemp = fopen(uitvoerfile, "wb");
   if ( ftemp == NULL ) return(2);

   /* Alloceer geheugen voor buffers */
   buf0 = (unsigned char*) malloc(ELEMENTEN*sizeof(char));
   buf1 = (int *) malloc(ELEMENTEN*sizeof(int));
   bufuit = (unsigned char *) malloc(ELEMENTEN*sizeof(unsigned char));
   if ( (buf0 == NULL) || (buf1 == NULL) || (bufuit == NULL) ) return(3);

   while (!feof(f1))
   {  /* Vul buffers */
      m = fread(buf1, sizeof(int), ELEMENTEN, f1);
      if (frameno == 1)
        for (n=0; n<m; n++) buf0[n] = 0;
      else
        n = fread(buf0, sizeof(char), ELEMENTEN, f0);

      /* Check of er goed ingelezen is */
      n = (n>m? m : n); if (n == 0) return(4);

      /* Maak de som (huidige frame) + (vorige frame) */
      for (m=0; m<n; m++)
      {  tijdelijk = (int) buf0[m] + buf1[m];
         if (tijdelijk > 255) tijdelijk =  255;   /* clipper */
         if (tijdelijk <-255) tijdelijk = -255;
         bufuit[m] =  (unsigned char) tijdelijk;
      }

      /* Bewaar sombuffer */
      m = fwrite(bufuit, sizeof(unsigned char), n, ftemp);

      /* Check of er goed weggeschreven is */
      if (m != n) return(5);

   }

   printf("\b ");
   /* Geef geheugen vrij */
   free(bufuit);
   free(buf1);
   free(buf0);

   /* Sluit files */
   if (frameno != 1) fclose(f0);
   fclose(f1);
   fclose(ftemp);
   return(0);
}
```

```
/**************************************************************************/
/*  Filenaam: MSE.C                                                        */
/*                     D E F I N E   S T A T E M E N S                     */
/*                     ===============================                     */
/**************************************************************************/
#define QUANT                     1   /* Kwantiseren? !=0 is ja            */
#define BREED                   352   /* Beeldtelefoon formaat beelden      */
#define HOOG                    288
#define MAXLINE                 512   /* Maximaal aantal pixels/beeldlijn   */
#define ELEMENTEN             30000   /* Elementen in buffer lezen per keer */
#define VERSCHILFRAME      "frame."   /* Tijdelijke file voor verschilbeeld */
#define CODEDFRAME         "coder."   /* File waar gecodeerd frame in komt  */
#define DECODEDFRAME     "decoder."   /* File waar gedecodeerd frame in komt*/
```

```c
/***************************************************************************/
/*  Filenaam: MSE.C                                                        */
/*  Bepaald de MSE tussen twee beelden van 352 x 288 pixels                */
/***************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#define ELEMENTEN 101376U
#define PIXELS    101376U


void main(int argc, char *argv[])
{  char files[3][80];
   int  n;
   int MSE(char frame0[80], char frame1[80]);

   for (n=0; n<2; n++) files[n][0] = 0;    /* Maak strings nul lang */
   if (argc-1)
     for (n=1; n<argc; n++)
       sprintf(&files[n-1][0], "%s", argv[n]);
   if (files[0][0] == 0)
   {  printf("File 1 in: ");
      scanf("%s", &files[0][0]);
   }
   if (files[1][0] == 0)
   {  printf("File 2 in: ");
      scanf("%s", &files[1][0]);
   }
   switch (MSE(files[0], files[1]))
   {  case 0: break;
      case 1: printf("File not found\n"); break;
      case 2: printf("Insufficient memory\n"); break;
      case 4: printf("Read error\n"); break;
   }
}


int MSE(char frame0[80], char frame1[80])
{  FILE          *f0, *f1;
   unsigned char *buf0, *buf1;
   unsigned int  n, m;
   long          mse_value=0;
   int           temp;

   f0 = fopen(frame0, "rb");
   f1 = fopen(frame1, "rb");
   if ( (f0 == NULL) || (f1 == NULL) ) return(1);

   /* Alloceer geheugen voor buffers */
   buf0 = (unsigned char *) malloc(ELEMENTEN);
   buf1 = (unsigned char *) malloc(ELEMENTEN);
   if ( (buf0 == NULL) || (buf1 == NULL) ) return(2);

   while ( !feof(f1) )
   {  /* Vul buffers */
      n = fread(buf0, 1, ELEMENTEN, f0);
      m = fread(buf1, 1, ELEMENTEN, f1);

      /* Check of er goed ingelezen is */
      n = (n>m? m : n); if (n == 0) return(4);

      for (m=0; m<n; m++)
      {  temp = (int) (buf1[m] - buf0[m]);
         temp = abs(temp);
         mse_value += (long)temp * temp;
      }
   }

   printf("MSE = %f\n",  (float) mse_value / PIXELS);

   /* Geef geheugen vrij */
   free(buf1);
   free(buf0);

   /* Sluit files */
   fclose(f0);
   fclose(f1);
   return 0;
}
```