

Efficiëntie en kost van videocompressie

Practicum Multimedia/Multimediatechnieken

Academiejaar 2017–2018

Deadline: do. 26 april 2018, 14u

Hoofdstuk 1

Inleiding

Een standaard full High-Definition (HD) video heeft een resolutie van 1920×1080 pixels en wordt afgespeeld aan 25 beelden per seconden. Als we 24 bits per pixel gebruiken (8 bits per kleurcomponent), zou één minuut video een opslagruimte van 9.3 GiB vereisen. Voor een volledige film van 120 minuten zou dit neerkomen op meer dan 1.1 TB data. Zelfs met enorme datacentra zouden video providers zoals Netflix en YouTube onmogelijk dezelfde hoeveelheid video's kunnen aanbieden zoals nu het geval is. Bovendien zouden kijkers thuis een internetverbinding nodig hebben met een snelheid van minstens 1.2 Gbit/s om zo een full HD videosequentie in ware tijd te kunnen bekijken.

Gelukkig kan dit kostelijke scenario vermeden worden dankzij videocompressie. Door deze technologie kan een film van 1 TB aan hoge kwaliteit gecomprimeerd worden tot minder dan 25 GB. Afhankelijk van de beeldinhoud, gebruikte compressietechnieken en het opofferen van beeldkwaliteit kunnen deze datavereisten nog meer teruggedreven worden. Zo komen we tot het doel van videocompressie: het reduceren van de vereiste opslagruimte en bandbreedte zodat videosequenties toegankelijk zijn voor het brede publiek.

In dit practicum is het de bedoeling om de studenten kennis te laten maken met de basisprincipes van videocompressie en het bepalen van beeldkwaliteit en compressie-efficiëntie. Op basis van deze kennis maken de studenten vervolgens een afweging tussen verschillende videocompressiestandaarden.

Hoofdstuk 2

Compressie van beelden

In dit hoofdstuk is het de bedoeling om de studenten een kort overzicht te geven van de voornaamste technieken die gebruikt worden in videocompressie. De informatie in dit hoofdstuk helpt dan ook om te begrijpen welke effecten verschillende configuratieparameters van video-encoders hebben op de coderefficiëntie van beelden.

Er zijn twee vormen van compressie: verliesloze (lossless) en verlieshebbende (lossy). Bij verliesloze compressie zal men, na decomprimeren, exact het oorspronkelijke beeld terugkrijgen. Dit is onder andere een vereiste in de grafische en medische sector. Het nadeel hiervan is dat de winst van de compressie beperkt blijft. Daartegenover staat verlieshebbende compressie waarbij er een aantal fouten in het oorspronkelijk beeld ontstaan. In de meeste gevallen kan dat echter geen kwaad omdat het Menselijk Visueel Systeem (Human Visual System, HVS) een zekere vorm van informatieverlies verdraagt zonder dat het de perceptie van het beeld hindert. Het voordeel is dat op die manier, door een klein aantal fouten toe te laten, het oorspronkelijke beeld met een veel kleiner aantal bits kan opgeslagen worden. Hierdoor zal de compressiefactor bij verlieshebbende compressie veel groter zijn dan bij verliesloze compressie.

In de volgende secties worden de volgende technieken besproken die toelaten om de informatie in de video te comprimeren:

- **onderbemonstering:** uitbuiten van de hogere gevoeligheid van het HVS voor helderheid dan voor kleur.
- **intrapredictie:** uitbuiten van de gelijkenissen tussen naburige pixels binnen hetzelfde beeld.
- **transformatie en quantisatie:** uitbuiten van de hogere gevoeligheid van het HVS

voor lagere frequenties dan voor hoge frequenties (zoals randen van voorwerpen en details in beelden).

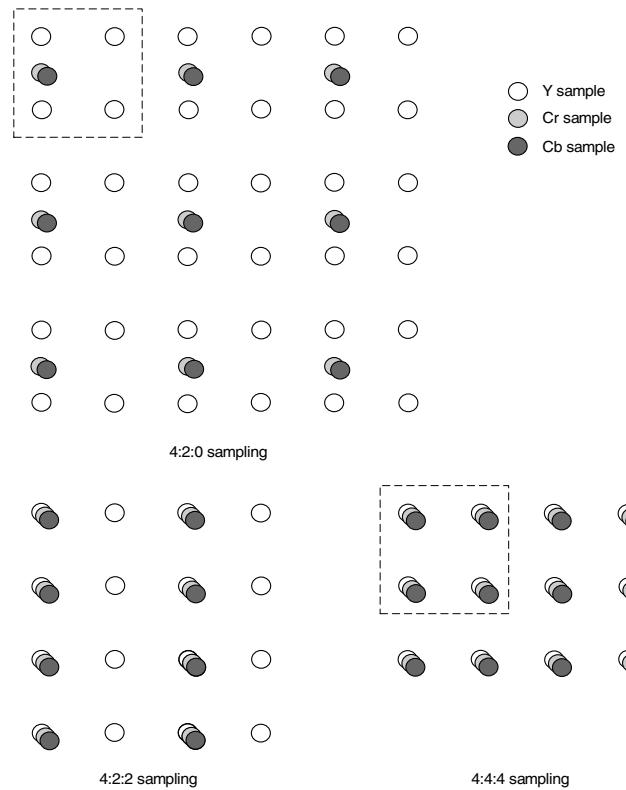
- **entropiecodering:** uitbuiten van de statistische redundantie in de informatie.
- **interpredictie:** uitbuiten van de gelijkenissen tussen opeenvolgende videobeelden.

2.1 Kleurenruimtes en onderbemonstering

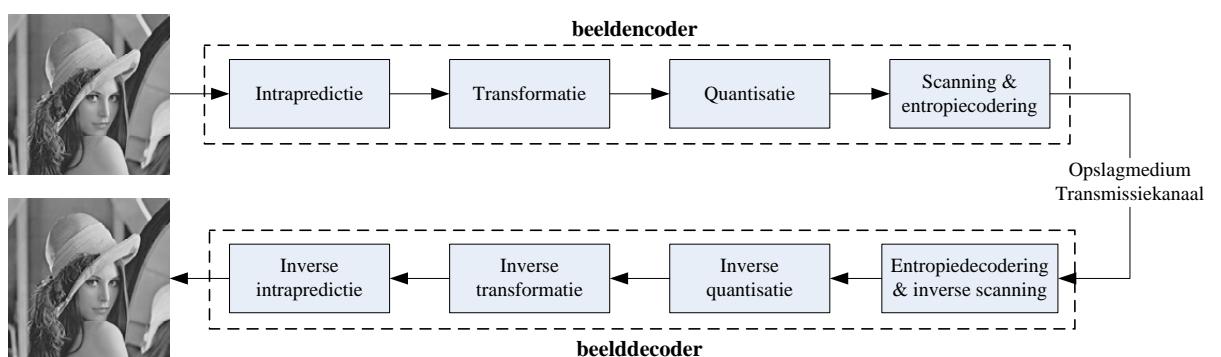
Een videosignaal is een sequentie van beelden waarbij elk beeld wordt voorgesteld door een tweedimensionale matrix van pixelwaarden. Iedere pixelwaarde is een element van de vorm $p[x, y]$, waarbij x de kolom-index voorstelt en y de rij-index. Per pixel worden enkele waarden bijgehouden die volstaan om de kleur van die pixel te kennen. De RGB-kleurenruimte (Rood-Groen-Blauw) is ongetwijfeld de meest bekende kleurenruimte. Videobeelden worden daarentegen vaak voorgesteld in de YUV kleurenruimte. Ook daar zijn drie componenten aanwezig: de Y-component of luminantiecomponent, die de helderheid of lichtintensiteit weergeeft, en twee chrominantiecomponenten (U en V, of ook wel Cb en Cr genoemd) die informatie over de kleur weergeven.

Het voordeel van een voorstelling in de YUV-kleurenruimte is dat de chrominantiecomponenten kunnen voorgesteld worden met een lagere resolutie. Dit staat bekend als de onderbemonstering van de chrominantie. Het HVS is immers minder gevoelig voor chrominantie dan voor luminantie. In Figuur 2.1 wordt een overzicht gegeven van de meest voorkomende onderbemonsteringsformaten. Wanneer het beeld niet onderbemonsterd wordt spreekt men van YUV 4:4:4. Als de helft van de kleurinfo weggegooid wordt en er per 2 pixels dus slechts 1 U- en V-waarde is, spreekt men van YUV 4:2:2. Wanneer driekwart van de kleurinformatie weggegooid wordt en er slechts 1 U- en V-waarde is voor elk blokje van 4 pixels, spreekt men van YUV 4:2:0. Dit laatste formaat wordt momenteel gebruikt door video providers zoals Netflix en YouTube, maar ook door kabeltelevisie. Enkel in applicaties waarbij alle informatie van belang is, zoals in bijvoorbeeld medische beeldvorming, wordt geen onderbemonstering gebruikt.

Merk op dat de YUV-bestanden die in dit practicum gebruikt worden reeds YUV 4:2:0 onderbemonsterd zijn.



Figuur 2.1: Onderbemonsteringsformaten



Figuur 2.2: Blokschema codec voor stilstaande beelden

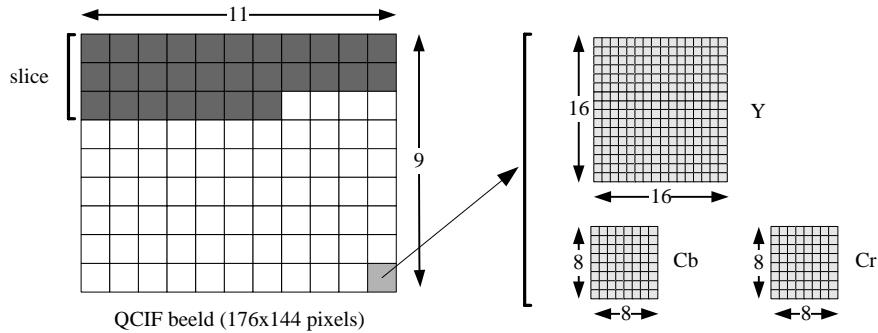
2.2 Compressieschema's voor stilstaande beelden

Figuur 2.2 geeft een overzicht van de verschillende stappen in het compressieproces voor een eenvoudige codec voor stilstaande beelden. Deze fasen zijn vrij algemeen en komen voor in bijna elk modern blokgebaseerd beeldcoderingsalgoritme¹. De belangrijkste fasen die doorlopen worden tijdens de compressie zijn de intrapredictie, de transformatie, de quantisatie, de scanning en de entropiecodering. In een eerste stap zal de intrapredictie gebruikmaken van de gelijkaardigheid van omliggende pixels om een voorspelling te maken voor elk blok. De bedoeling van de transformatiefase is de pixelwaarden van het oorspronkelijke beeld naar het frequentiedomein om te zetten zodanig dat er een onderscheid kan gemaakt worden tussen hoge en lage frequenties. In de quantisatiefase zullen de getransformeerde coëfficiënten met minder precisie en gebruikmakend van een kleiner aantal bits worden voorgesteld. Merk op dat de transformatiestap zelf niet voor compressie zorgt; het verwijderen van coëfficiënten tijdens de quantisatie doet dit wél. Daar de quantisatiestap verlieshebbend is, zal dit de kwaliteit van het beeld echter nadelig beïnvloeden. De scanning zal de 2-dimensionale representatie van het gequantiseerde beeld omzetten in een 1-dimensionale voorstelling. De laatste fase is de entropiecodeerfase. Deze zorgt voor een verdere algemene compressie. Deze entropiecodering zal statistische redundantie tussen de verschillende gequantiseerde waarden opsporen en gebruikmakend van deze informatie een bitstroom genereren. Veelgebruikte entropiecoders zijn de Huffman-coders, aritmatische coders en eenvoudige run-length-coders. De verschillende fasen van het codeerproces worden in de volgende paragrafen kort behandeld.

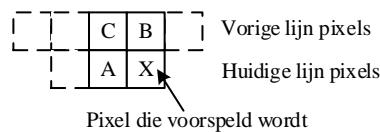
2.2.1 Onderverdeling van een beeld

Een videosequentie is opgebouwd uit een reeks beelden bestaande uit drie matrices met pixelinformatie, één voor de luminantiecomponent en twee voor de chrominantiecomponenten. Elk beeld wordt in blokgebaseerde codecs verder onderverdeeld in een reeks blokken. In de huidige meestverspreide codec H.264/AVC zijn deze blokken gekend als macroblokken en bestaan ze uit een matrix van 16x16 luminantiesamples en twee matrices met chrominantiesamples. Het aantal chrominantiesamples is afhankelijk van het gebruikte onderbemonsteringsformaat. Macroblokken worden gegroepeerd tot slices zodanig dat elk macroblok tot juist één slice behoort. In Figuur 2.3 wordt deze partitionering geïllustreerd voor een QCIF-beeld (176x144 pixels). In dit voorbeeld wordt het beeld

¹Merk op dat bij ‘oudere’ compressiealgoritmes, zoals bij JPEG, er nog geen intrapredictiestap aanwezig is.



Figuur 2.3: Onderverdeling van een beeld in macroblokken en opbouw van een macroblok

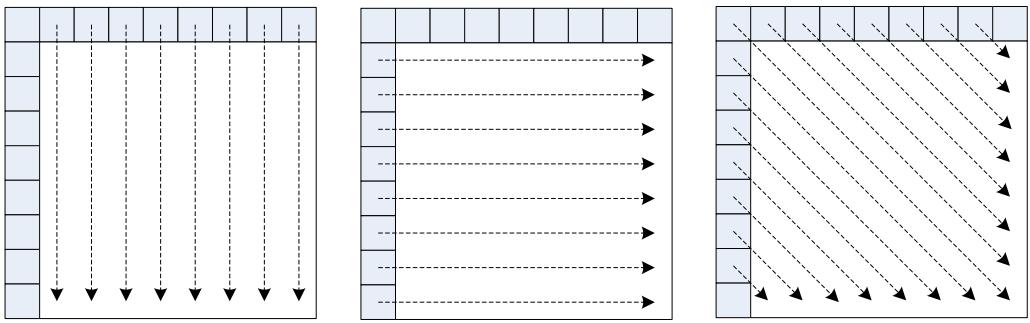


Figuur 2.4: Voorspelling van pixelwaarde

onderverdeeld in 99 macroblokken. De structuur van één van deze macroblokken is ook in de figuur te zien.

2.2.2 Intrapredictie

Een mogelijke methode om spatiale afhankelijkheden uit te buiten in een beeld, is om niet elke pixelwaarde afzonderlijk te coderen, maar om deze te voorspellen aan de hand van de naburige pixels. Elke pixel wordt dan voorspeld op basis van één of meerdere eerder gecodeerde pixels. Stel dat we de pixelwaarde van pixel X in Figuur 2.4 willen voorspellen. De meest eenvoudige vorm van voorspelling bestaat er in om de pixelwaarde van de vorige gecodeerde pixel te nemen (pixel A). Een meer geavanceerde voorspelling kan bekomen worden door een gewogen gemiddelde te nemen van naburige, reeds gecodeerde pixels (bv. pixels A, B en C). De voorspelde waarde van de pixel (X) wordt vervolgens afgetrokken van de eigenlijke waarde. Dit verschil (de voorspellingsfout of residu) wordt dan verder gebruikt in het compressieschema. Door de spatiale correlatie zal deze voorspellingsfout klein zijn. Zoals hierboven reeds werd aangegeven kan efficiënte compressie bekomen worden door (kleine) veelvoorkomende waarden voor te stellen met korte codewoorden en (grote) weinig voorkomende waarden door langere codewoorden (Huffman-entropiecodering). Aangezien de verkregen verschillen vaak klein zijn, zullen deze tijdens de entropiecodering goed gecomprimeerd kunnen worden.



(a) Verticale intrapredictie (mode 1) (b) Horizontale intrapredictie (mode 2) (c) Diagonale intrapredictie (mode 3)

Figuur 2.5: Intrapredictie toegepast op 8x8-blokken

2.2.2.1 Geavanceerdere predictiemodes

In moderne videocompressiestandaarden worden meer geavanceerde predictiemodes gebruikt. Zo kent bv. H.264/AVC een DC-predictie en 8 directionele predictiemodes. Een modernere standaard zoals de High Efficiency Video Coding standaard breidt dit zelfs uit tot 35 verschillende modes. In deze subsectie worden vier predictiemodes kort toegelicht, nl. DC-predictie, verticale, horizontale en diagonale predictie.

Bij DC-predictie wordt de gemiddelde waarde van de pixels links en boven berekend (voor een blok van $N \times N$ pixels is dit in totaal $2N$ pixels voor luma en N pixels voor chroma). Dit gemiddelde wordt afgetrokken van het huidige blok; het residu wordt verder verwerkt door de transformatie, quantisatie en entropiecodering.

Bij de verticale, horizontale en diagonale modes worden omliggende pixelwaarden geëxtrapoleerd in de desbetreffende richtingen ter predictie van het huidige blok. Hierbij wordt een kopie gebruikt van de waarde van de randpixel; de predictie is dan ook dezelfde voor alle locaties op dezelfde rij/kolom/diagonaallijn. Deze modes worden geïllustreerd in Figuur 2.5 voor 8x8-blokken. De predictie voor andere blokgroottes wordt op analoge wijze geconstrueerd.

Het algoritme om te beslissen welke predictiemode er het beste gebruikt wordt is niet gestandaardiseerd. Bij het zoeken naar de beste predictiemode bij de intrapredictie betekent dit dat er verschillende metrieken kunnen gebruikt worden bij het berekenen van de “energie” aanwezig in het residu na voorspelling. Onderstaande formules beschrijven twee mogelijke manieren om deze energie te berekenen: MSE en SAE. De blokgrootte hierbij is $N \times N$ pixels; C_{ij} en R_{ij} corresponderen hier respectievelijk met de pixelwaarden van het huidige blok en het blok gevormd door intrapredictie.

$$1. \text{ Mean Squared Error :} \quad MSE = \frac{1}{N^2} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} (C_{ij} - R_{ij})^2 \quad (2.1)$$

$$2. \text{ Sum of Absolute Errors :} \quad SAE = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} |C_{ij} - R_{ij}| \quad (2.2)$$

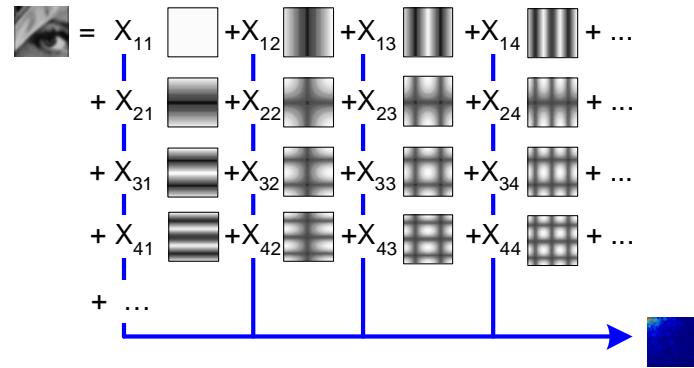
Deze formules kunnen berekend worden op enkel de luminantiecomponent of zowel op de luminantie- als de chominantiecomponent van de te vergelijken blokken. Deze keuze wordt overgelaten aan de ontwikkelaar. In het merendeel van de bestaande encoders wordt enkel naar luminantie gekeken om de rekencomplexiteit te beperken.

2.2.3 Transformatie

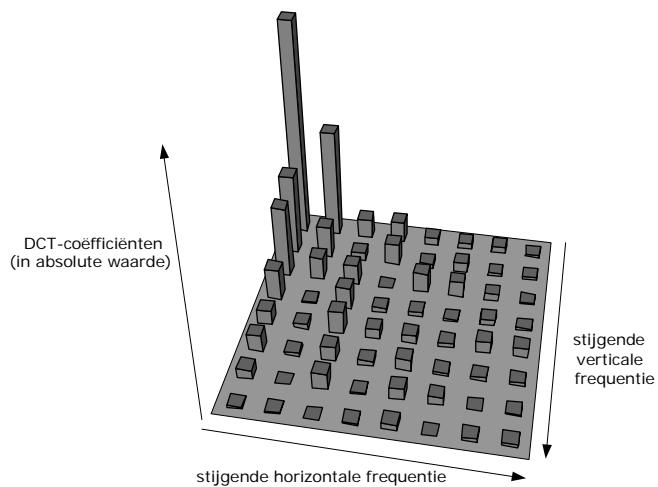
Een manier om de spatiale correlatie uit het beeld te verwijderen is door gebruik te maken van de 2-dimensionale Discrete Cosinus Transformatie (DCT). Hierbij worden de pixelwaarden getransformeerd van het spatiale- naar het frequentiedomein. Hiervoor wordt het beeld eerst opgedeeld in discrete vierkante blokken. Typische blokgroottes zijn 8x8 en 16x16 pixels. Op elk van deze blokken wordt een DCT uitgevoerd. Hierbij is het belangrijk op te merken dat een DCT zelf geen compressie veroorzaakt (na het uitvoeren van een inverse DCT worden de originele coëfficiënten terug bekomen). Men kan stellen dat een beeld een lineaire combinatie is van 2-dimensionale DCT-basisfuncties. Een DCT-transformatie zorgt er voor dat een beeld ontbonden wordt als lineaire combinatie van deze basisfuncties. Door de eigenschappen van de DCT zal het beeld na transformatie ontbonden zijn in een klein aantal coëfficiënten die visueel belangrijk zijn, terwijl de grote meerderheid van de coëfficiënten visueel minder belangrijke informatie voorstellen. Met andere woorden, **de visueel belangrijke informatie van een macroblok zit vervat in een klein aantal coëfficiënten**. Bij typische beelden bevinden deze coëfficiënten zich linksboven (dit zijn de componenten met de laagste frequenties). De coëfficiënt in de linkerbovenhoek van het getransformeerde macroblok noemt men de DC-coëfficiënt, de andere coëfficiënten worden de AC-coëfficiënten genoemd. Dit wordt geïllustreerd in Figuren 2.6 en 2.7.

2.2.4 Quantisatie

Zoals reeds gezegd werd, zorgt de DCT-transformatie op zichzelf niet voor compressie. De compressie zelf wordt bereikt met behulp van quantisatie. Het doel van quantisatie is



Figuur 2.6: DCT: ontbinding van macroblok in lineaire combinatie van DCT-basisfuncties, X_{ij} zijn de DCT-coëfficiënten



Figuur 2.7: Voorbeeld van een DCT-coëfficiëntenmatrix (in 3D) horende bij een 8x8-blok; de numerieke waarden van de verschillende coëfficiënten worden gegeven in de linkse tabel van Figuur 2.8. De meeste en de grootste (absolute) waarden bevinden zich in de linkerbovenhoek. Hoe verder weg van deze regio, hoe hoger de spatiale frequenties worden (en hoe minder belangrijk de coëfficiënten zijn)

 = correct gereconstrueerd

126	-49	43	-19	9	-10	6	-1	31	-11	10	-4	2	-2	1	0	124	-44	40	-16	8	-8	4	0
-65	19	-14	-1	3	2	0	-1	-16	4	-3	0	0	0	0	0	-64	16	-12	0	0	0	0	0
12	5	-12	13	-14	9	-10	0	3	1	-3	3	-3	2	-2	0	12	4	-12	12	-12	8	-8	0
-13	13	0	-3	6	3	1	1	-3	3	0	0	1	0	0	0	-12	12	0	0	4	0	0	0
5	3	-12	3	-5	-7	7	-4	1	0	-3	0	-1	-1	1	-1	4	0	-12	0	-4	-4	4	-4
-4	-6	9	1	-3	2	-5	0	-1	-1	2	0	0	0	-1	0	-4	-4	8	0	0	0	-4	0
4	-2	-4	-4	7	2	0	2	1	0	-1	-1	1	0	0	0	4	0	-4	-4	4	0	0	0
-1	-2	1	1	-6	-2	1	-2	0	0	0	0	-1	0	0	0	0	0	0	-4	0	0	0	0

origineel

gequantiseerd($Qp = 4$)

gedequantiseerd($Qp = 4$)

Figuur 2.8: (links) Originele DCT-coëfficiënten; (midden) Coëfficiënten na quantisatie, merk op hoe de waarden veel kleiner zijn dan voorheen; (rechts) Coëfficiënten na inverse quantisatie, deze zijn grotendeels verschillend van de originele DCT-coëfficiënten

de visueel minder belangrijke coëfficiënten te verwijderen of de nauwkeurigheid te verminderen (voorstellen met minder bits). De gequantiseerde coëfficiënten kunnen met minder bits gecodeerd worden door hun lagere entropie.

Bij de scalaire quantisatie wordt dit voor iedere coëfficiënt afzonderlijk toegepast. In Figuur 2.8 wordt dit principe geïllustreerd door elke coëfficiënt over 2 bits naar rechts te verschuiven (deling door 4). Merk op dat dit proces verlieshebbend is, immers na inverse quantisatie bekomt men niet langer de originele DCT-coëfficiënten. Dit betekent dus dat er details in het beeld verloren gegaan zijn tijdens het quantisatieproces.

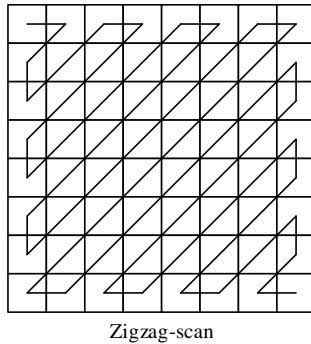
De meest eenvoudige vorm van scalaire quantisatie wordt gedefinieerd als volgt:

$$Z_{ij} = \text{round}(Y_{ij}/Qstep) \quad (2.3)$$

De bijhorende inverse quantisatie wordt bekomen door:

$$Y'_{ij} = Z_{ij} \cdot Qstep \quad (2.4)$$

Hierbij is Y_{ij} een coëfficiënt uit de matrix \mathbf{Y} die we bekomen hebben na de DCT, $Qstep$ is de stapgrootte van de gebruikte quantisator, Z_{ij} is de gequantiseerde coëfficiënt en Y'_{ij} de verkregen waarde na inverse quantisatie. De *round*-functie hoeft niet noodzakelijk af te ronden naar het dichtstbijzijnde gehele getal. Hoe groter de stapgrootte van de quantisator, hoe lager de bekomen kwaliteit. Dit komt doordat meer inputwaarden op eenzelfde waarde worden afgebeeld, waardoor de bijhorende quantisatiefout ook zal verhogen en de verkregen waarden na inverse quantisatie een ruwe schatting zullen zijn van de originele waarden. Anderzijds zal een grotere stapgrootte de video meer comprimeren doordat het bereik van de gequantiseerde coëfficiënten kleiner is en bijgevolg efficiënter



Figuur 2.9: Meestvoorkomende scanmethode om de 2-dimensionale DCT-coëfficiëntenmatrix te converteren naar een 1-dimensionale vector

kan gecodeerd worden. Aangezien er minder bits nodig zijn, wordt er een lagere bitsnelheid bekomen. In het geval de stapgrootte klein is, zullen de waarden na quantisatie en inverse quantisatie een betere benadering zijn dan bij een grote stapgrootte waardoor de kwaliteit van de gecodeerde videosequentie beter zal zijn. Deze kleine stapgrootte heeft echter ook als gevolg dat de compressie minder efficiënt zal zijn.

2.2.5 Scanning

Deze stap zorgt voor de omzetting van de 2-dimensionale voorstelling naar een 1-dimensionale voorstelling die aan de entropiecoder gegeven kan worden. Dit kan eenvoudig gebeuren door de matrix bekomen na de DCT-transformatie coëfficiënt per coëfficiënt te doorlopen. De bedoeling daarbij is om de meest significante coëfficiënten vooraan in de 1-dimensionale voorstelling te plaatsen. Dit laat namelijk een effectievere codering toe in de volgende stap.

Figuur 2.9 toont de meestvoorkomende scanmethode bij een DCT-transformatie: zigzag-scan. Deze methode wordt gewoonlijk gebruikt omdat deze eerst de volledige linkerbovenhoek afwerkt waar de meest belangrijke coëfficiënten (de lage frequenties) zich bevinden.

De 1-dimensionale voorstelling van het gequantiseerde macroblok uit Figuur 2.8 gebruikmakend van zigzag-scan is dan:

```

31, -11, -16, 3, 4, 10, -4, -3, 1, -3, 1, 3, -3, 0, 2, -2, 0, 3, 0, 0, -1, 1,
-1, -3, 0, -3, 0, 1, 0, 0, 2, 1, 0, 2, 0, 0, 0, -1, 0, -1, 0, -2,
0, 0, 0, -1, 0, -1, 0, 0, 1, 0, 1, 0, -1, -1, 0, -1, 0, 0, 0, 0, 0, 0

```

2.2.6 Entropiecodering

Een macroblok zal na transformatie en quantisatie bestaan uit een klein aantal significante coëfficiënten. Deze van nul verschillende coëfficiënten kunnen efficiënt gecodeerd worden door gebruik te maken van statistische methoden. In deze stap wordt de redundantie tussen de gequantiseerde waarden opgespoord en uitgebuit om een compacte bitstroom te genereren.

Merk op dat entropiecodering een verliesloze compressietechniek is die vergelijkbaar is met een finale zip-compressie van de informatie.

2.3 Compressieschema's voor bewegende beelden

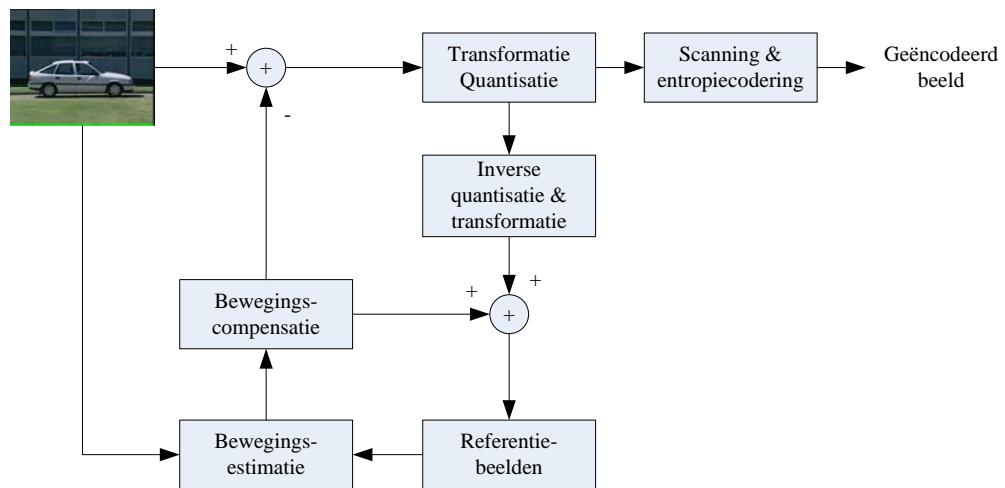
Een videosequentie bestaat uit een opeenvolging van beelden. Elk beeld afzonderlijk kan gecomprimeerd worden door een codec zoals beschreven in §2.2. Dit noemt men intracoderen omdat elk beeld afzonderlijk wordt gecodeerd zonder gebruik te maken van vorige of volgende beelden (er wordt geen gebruik gemaakt van temporele correlatie).

Eén van de eigenschappen van beelden in een videosequentie is echter dat ze dikwijls heel weinig verschillen van voorgaande of volgende beelden. Reeds gecodeerde beelden kunnen dan ook gebruikt worden als referentie om een beeld efficiënter te comprimeren. Door eerst een voorspelling te maken van het huidige beeld aan de hand van de referentiebeelden en dit resultaat af te trekken van het originele beeld, ontstaat er een nieuw beeld dat meestal heel goed te comprimeren is met behulp van intra-compressie. Dit wordt schematisch weergegeven in Figuur 2.10. Deze werkwijze noemt men intercoderen.

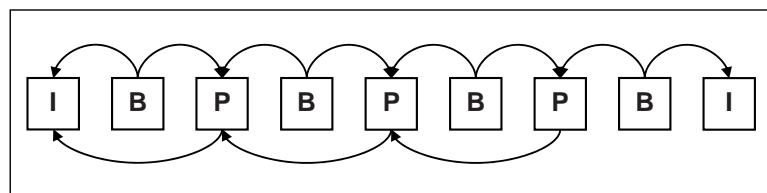
Referentiebeelden zijn beelden die gebruikt worden bij voorspellingen. Dergelijke beelden kunnen intragecodeerd zijn (I-beelden) of zelf het resultaat zijn van een voorspelling (P-beelden). Voorspellingen gebeuren steeds op gedecodeerde beelden, op die manier gebruiken de coder en decoder dezelfde referentiebeelden. Daardoor worden geen extra voorspellingsfouten geïntroduceerd. Beelden kunnen ook voorspeld worden op basis van meerdere referentiebeelden (B-beelden). Dit wordt verduidelijkt in Figuur 2.11.

2.3.1 Predictie aan de hand van het voorgaande beeld

De eenvoudigste manier van temporele predictie is het gebruik van het voorgaande beeld als voorspelling van het huidige beeld. Figuur 2.12 toont twee opeenvolgende beelden waarbij het tweede beeld wordt gebruikt als voorspelling voor het eerste en het resulterende residuele beeld dat bekomen wordt door het referentiebeeld van het huidige beeld



Figuur 2.10: Videocoder met bewegingsschatting en bewegingscompensatie



Figuur 2.11: Afhankelijkheden als gevolg van voorspellingen binnen een videosequentie



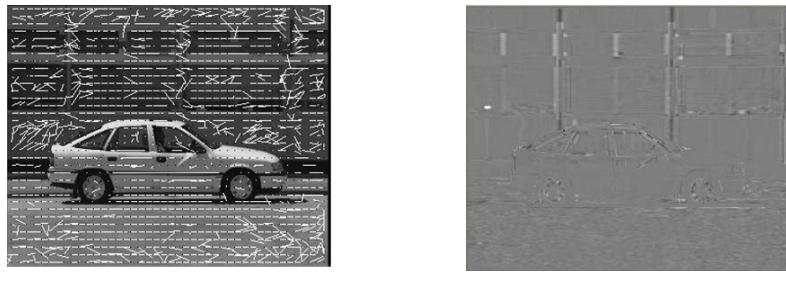
Figuur 2.12: Predictie aan de hand van het bijhorende referentiebeeld

af te trekken. In dit residubeeld stellen de midgrijze pixels een verschil van nul voor terwijl de lichte en donkere pixels overeenkomen met positieve en negatieve verschillen. Deze lichte en donkere gebieden vereisen meer bytes in het codeerproces dan de waarden dichtbij nul. Deze hogere waarden zijn hoofdzakelijk te wijten aan beweging van objecten of de camera tussen deze twee beelden. Door deze beweging te compenseren kan bijgevolg een betere predictie bekomen worden.

2.3.2 Bewegingsestimatie en -compensatie

Een veelgebruikte manier voor bewegingscompensatie is de blokgebaseerde bewegingscompensatie. Hierbij wordt het huidige beeld eerst onderverdeeld in rechthoekige blokken van $M \times N$ pixels (bv. macroblokken van 16×16 pixels, maar ook grotere/kleinere partities afhankelijk van de gebruikte videostandaard). Daarna wordt voor elk van deze blokken gekeken welk blok in het voorgaande beeld een goede schatting is van het huidige blok:

- Eerst worden de referentiebeelden doorzocht met de bedoeling een goede predictie te vinden voor het huidige blok. Hiervoor wordt het huidige blok vergeleken met sommige of alle mogelijke $M \times N$ blokken in het bijhorende zoekvenster (dit komt meestal overeen met een gebied gecentreerd rond de huidige positie) en wordt het blok geselecteerd dat het best met het huidige blok overeenkomt. Een veelgebruikt criterium hierbij is de energie die aanwezig is in het residu afkomstig van het verschil van het huidige blok en het kandidaatblok in het referentiebeeld (vergelijkbaar met de SAE-methode uit intrapredictie voor het bepalen van de beste predictierichting). Het kandidaatblok dat de laagste energie met zich meebrengt wordt dan gekozen als de beste voorspelling voor het huidige blok. Dit proces is gekend als bewegingsestimatie.



bewegingsvectoren

residu na bewegingscompensatie

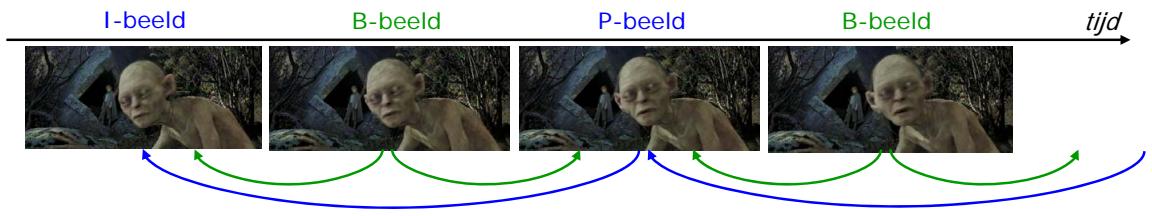
Figuur 2.13: Bewegingsvectoren bekomen na bewegingsestimatie en het bijhorende residu

- Daarna wordt het geselecteerde kandidaatblok gebruikt als voorspelling voor het huidige blok en wordt het afgetrokken van het huidige blok om op deze manier het residu te vormen. Dit wordt ook wel bewegingscompensatie genoemd.
- Verder wordt het residu gecodeerd gebruikmakend van de technieken besproken in §2.2, namelijk opeenvolgend de transformatie, quantisatie, scanning en entropiecodering. Verder wordt ook de vector doorgestuurd die het verschil in coördinaten voorstelt tussen het huidige blok en het geselecteerde kandidaatblok. Deze wordt ook de bijhorende bewegingsvector genoemd. In Figuur 2.13 wordt voor elk 16x16 blok de optimale bewegingsvector weergegeven en het bijhorende residu. Het is duidelijk dat het residu na bewegingscompensatie minder energie bevat dan zonder bewegingscompensatie (zie Figuur 2.12).

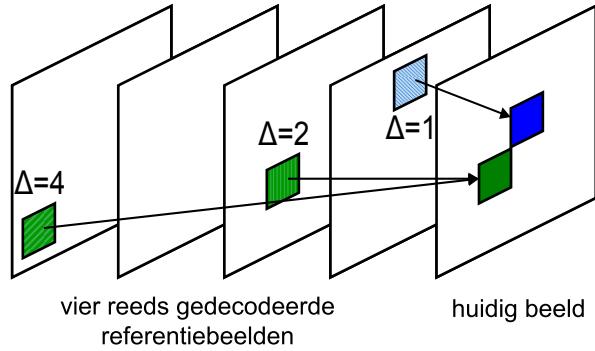
De decoder gebruikt de ontvangen bewegingsvector om het geselecteerde kandidaat-blok te reconstrueren. Door het bijhorende residu te decoderen en op te tellen bij het gereconstueerde blok wordt een gereconstrueerde versie bekomen van het originele blok.

2.3.3 B-beelden

Tot nog toe hebben we gebruikgemaakt van I-beelden (gecodeerd als stilstaande beelden) en P-beelden (voorspelde beelden op basis van het vorige beeld). Er bestaat echter nog een derde soort beelden: B-beelden. Deze beelden (i.e., de macroblokken van deze beelden) worden op basis van twee referentiebeelden voorspeld, zoals wordt geïllustreerd in Figuur 2.14. De opeenvolging van de soorten beelden wordt de GOP-structuur genoemd (*Eng.: Group of Pictures*) en het aantal beelden tussen twee I-beelden (inclusief het eerste en exclusief het tweede) noemt men typisch de GOP-lengte.



Figuur 2.14: Eenvoudige illustratie van B-beelden.



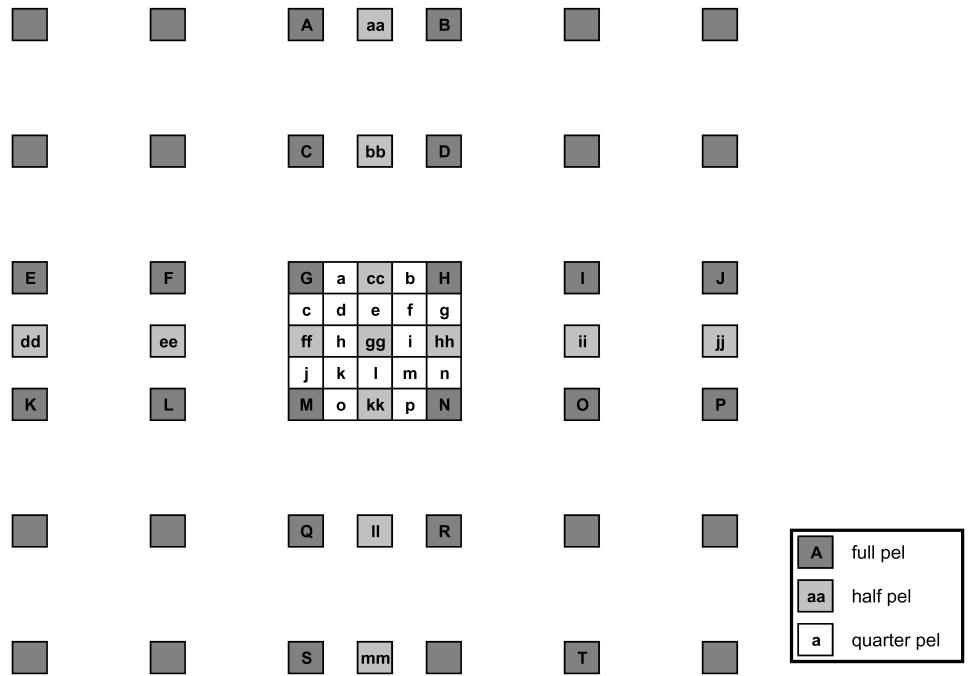
Figuur 2.15: Illustratie van meerdere referentiebeelden.

In H.264/AVC zijn de eerder traditionele concepten van P- en B-beelden gevoelig uitgebreid. Zo kunnen de macroblokken van eenzelfde P-beeld voorspeld worden op basis van verschillende referentiebeelden. De referentiebeelden zijn niet beperkt tot de n temporeel vorige beelden, maar deze referentiebeelden kunnen quasi willekeurig gekozen worden. Ook wat betreft B-beelden is het zo dat meerdere referentiebeelden gebruikt kunnen worden (i.e., de lijst van referentiebeelden is niet beperkt tot twee ‘vaste’ referentiebeelden). Daarbij komt dat de gebruikte referentiebeelden beiden uit het verleden of beiden uit de toekomst kunnen komen. Dit wordt geïllustreerd in Figuur 2.15. Deze extra vrijheidsgraden zorgen er natuurlijk voor dat de zoekruimte voor een encoder enorm groot wordt.

2.3.4 Sub-pixel nauwkeurigheid

We hebben reeds gezien dat bewegingsestimatie en -compensatie de energie van het residubeeld gevoelig kan laten dalen. Daarbij werd gezocht naar een bewegingsvector die aanwijst waar het meest gelijkende macroblok zich bevindt in het referentiebeeld.

Het is echter zo dat, indien een object beweegt in een videosequentie, een precisie van 1 pixel vaak niet voldoende is om de beweging weer te geven. Met andere woorden: het residubeeld heeft een relatief hoge energie. Daarom gebruikt men in huidige videodecs bewegingsvectoren die een precisie hebben van een halve (*Eng.: half pel*) of zelfs een



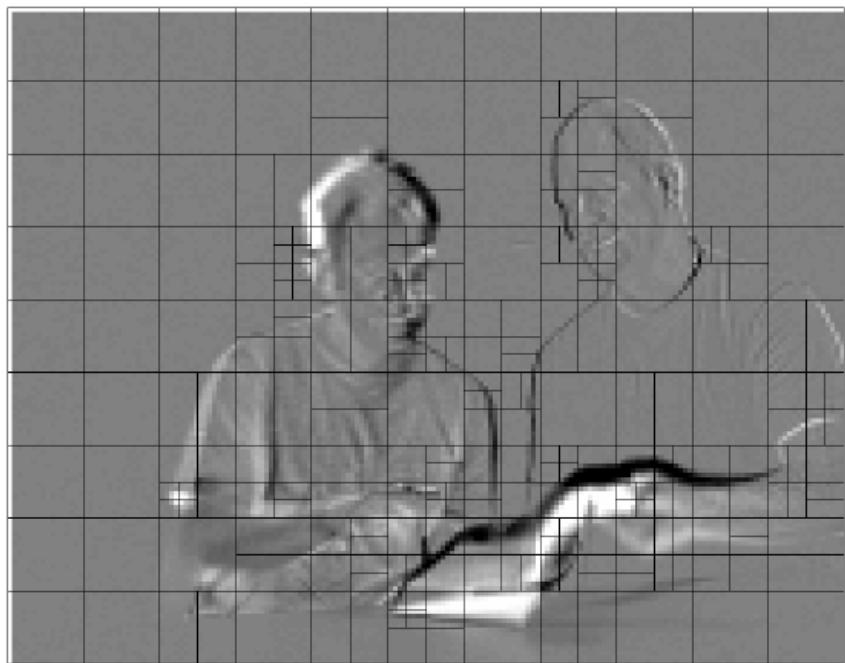
Figuur 2.16: Illustratie van interpolatie voor sub-pixel nauwkeurigheid in H.264/AVC.

vierde (*Eng.: quarter pel*) pixel. Daartoe berekent men tijdens de bewegingsestimatie en -compensatie virtuele pixels die zich tussen de werkelijke pixels bevinden. Een referentiebeeld wordt op die manier dus als het ware opgeblazen (waardoor ook het zoekvenster vergroot).

In het geval van H.264/AVC wordt de gebruikte interpolatie geïllustreerd in Figuur 2.16. In deze figuur stellen de hoofdletters (A, B, ...) de werkelijke pixels voor, stellen de dubbele letters de *half pel* pixels voor, en stellen de overige blokjes de *quarter pel* pixels voor. Door deze interpolaties kunnen bewegingen beter voorspeld worden en kan de energie van het residubeeld verder naar beneden gebracht worden.

2.3.5 Macroblokpartities

Indien er in het beeld kleine objecten bewegen, zijn blokken van 16×16 pixels te groot om de beweging efficiënt te bevatten. Dit wordt geïllustreerd in Figuur 2.17. Vandaar dat in moderne codeerschema's ook kleinere blokken gebruikt worden tijdens de bewegingsestimatie. In MPEG-2 konden naast 16×16 -blokken ook 8×8 -blokken gebruikt worden. In H.264/AVC gaat men nog een stap verder: 16×16 -blokken kunnen onderverdeeld worden in 16×8 -blokken, 8×16 -blokken, of 8×8 -blokken. Daarenboven kunnen 8×8 -blokken nog eens verder opgedeeld worden in 8×4 -blokken, 4×8 -blokken, of 4×4 -blokken. Op die manier is het dus mogelijk om 16 bewegingsvectoren te hebben voor één macroblok (en



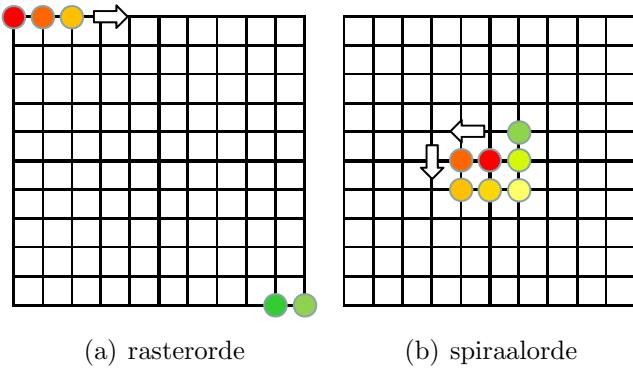
Figuur 2.17: Illustratie van verschillende blokgroottes in H.264/AVC.

zelfs 32 bewegingsvectoren in geval van een B-macroblok). Het nadeel hiervan is dat de encoder meer computationeel complex wordt doordat de meest efficiënte partitionering bepaald moet worden.

2.3.6 Snelle algoritmen voor bewegingsestimatie

Om de beste voorspelling van een blok te vinden in een referentiebeeld, is het in theorie nodig om dit blok te vergelijken met alle mogelijke posities in het referentiebeeld. Aangezien dit heel rekenintensief is, zal men in praktijk niet alle waarden overlopen. Een eerste veronderstelling hierbij is dat een goede voorspelling zich normaal gezien in de directe omgeving van het blok zal bevinden. Daarom zal men vaak gebruikmaken van een zoekvenster dat gecentreerd is rond het huidige blok. De optimale grootte van dit venster wordt beïnvloed door een aantal factoren: de resolutie van het beeld, de inhoud (scènes met veel beweging hebben meer voordeel bij een groot zoekvenster),...

Full search Wanneer elke mogelijke positie binnen het zoekvenster wordt geëvalueerd aan de hand van een vergelijkingscriterium zoals bijvoorbeeld SAE, spreekt men van *Full Search*. De volgorde waarin de verschillende posities in het zoekvenster worden overlopen, is meestal *rasterorde* (Figuur 2.18(a)) of *spiraalorde* (Figuur 2.18(b)). In sommige implementaties wordt de zoektocht vroegtijdig gestopt indien het verschil tussen het huidige



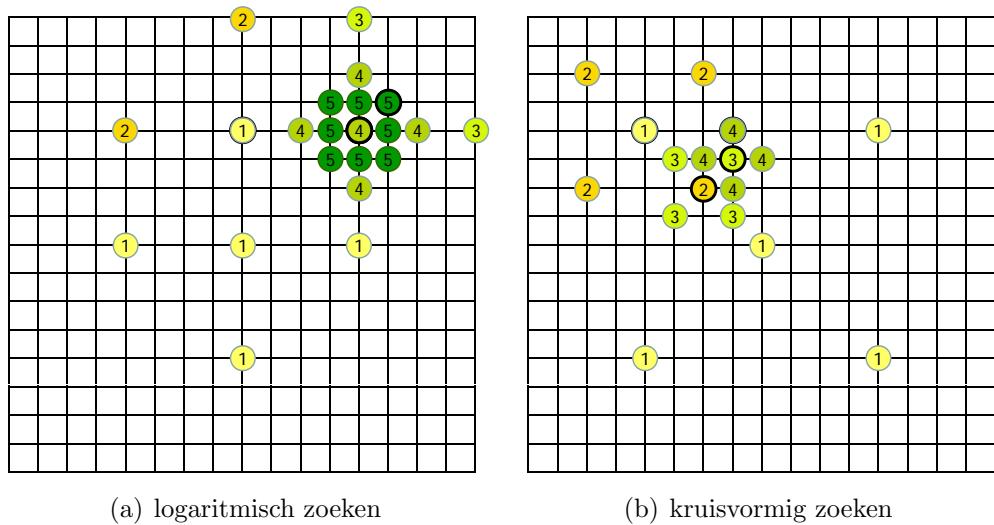
Figuur 2.18: Illustratie van full search

blok en het gevonden blok in het referentiebeeld kleiner is dan een bepaalde grens. In dit geval is het best om voor spiraalorde te kiezen aangezien de kans groot is dat het optimale blok in het referentiebeeld dichtbij het centrum van het zoekvenster ligt.

De computationele complexiteit van een full search algoritme is vaak nog te hoog, vooral wanneer de encoder real-time moet werken. Bijgevolg zijn er heel wat snelle algoritmen ontwikkeld die slechts een beperkt aantal posities evalueren maar die toch proberen de minimale SAE in het zoekvenster te vinden. In tegenstelling tot de full search algoritmen, die steeds het minimum terugvinden, lopen deze snelle algoritmen het gevaar om in een lokaal minimum te verzeilen. Bijgevolg zal het residu meer energie bevatten, waardoor het aantal bits in de gecomprimeerde videostroom groter zal zijn. Hoewel snelle zoekalgoritmen typisch lagere compressieperformantie geven, is hun complexiteit ook lager. Twee voorbeelden van deze algoritmen zijn logaritmisch zoeken en kruisvormig zoeken. Hierbij worden bij elke stap van het algoritme een beperkt aantal punten geëvalueerd en wordt het beste punt gebruikt als startpunt voor de volgende zoekstap. In Figuur 2.19 worden deze beide algoritmen getoond.

Aangezien er heel wat snelle zoekalgoritmen bestaan, worden er een aantal criteria gebruikt om deze met elkaar te vergelijken. Hoe goed kan het algoritme de residuele informatie reduceren? Hoeveel berekeningen moeten uitgevoerd worden? Kan het algoritme gemakkelijk geschaald worden zodat het werkt voor zowel grote als kleine zoekvensters? Is het algoritme geschikt om in hardware of software te implementeren?

Een codeerstandaard zal nooit opleggen welk algoritme moet gebruikt worden bij de bewegingsestimatie, aangezien deze enkel de syntax van de videostroom vastlegt en de decoder specificeert. Bijgevolg kunnen verschillende encoders andere algoritmen gebruiken.



Figuur 2.19: Voorbeelden van snelle algoritmen voor bewegingsestimatie

Hoofdstuk 3

Codeerefficiëntie

In het vorige hoofdstuk werden enkele basistechnieken besproken voor videocompressie. Door de jaren heen zijn echter heel wat verbeteringen en optimalisaties ontwikkeld die ervoor gezorgd hebben dat hedendaagse videostandaarden een veel hogere compressieratio kunnen bereiken met behoud van dezelfde visuele kwaliteit. Om de effecten van deze verbeteringen te meten zijn er natuurlijk manieren nodig om de compressie-efficiëntie te meten en te vergelijken tussen verschillende codecs. Een belangrijk instrument om dergelijke relaties te visualiseren is het gebruik van zogenaamde RD-curves (*Eng.: Rate-Distortion curves*). Voor dergelijke vergelijkingen wordt gekeken naar de gemiddelde bitsnelheid en kwaliteit van de volledige video. In de praktijk wil men echter soms ook de bitsnelheid over de gehele video constant houden, wat mogelijk is met rate control. Zowel RD-curves als rate control worden in dit hoofdstuk besproken.

3.1 RD-curves

RD-curves zijn een belangrijk instrument om de relatie tussen compressie en kwaliteit weer te geven. Op deze curves wordt de bitsnelheid gebruikt als maat voor compressie. Deze bitsnelheid is gelijk aan het gemiddelde aantal bits per seconde dat doorgestuurd zou worden als de gecomprimeerde video in ware tijd door een gebruiker gestreamd zou worden. Dit wil dus zeggen dat dit in theorie gelijk is aan **de totale bestandsgrootte van de video gedeeld door de totale afspeelduur van de video**. In de praktijk zal de bitsnelheid van video echter vaak niet perfect constant zijn, tenzij strenge rate control algoritmes worden gebruikt (zie verder in de tekst).

Om de kwaliteit van gecomprimeerde video te meten kunnen bepaalde objectieve metrieken gebruikt worden. Een van de meest gebruikte kwaliteitsmetrieken hiervoor is de

PSNR (*Eng.: Peak Signal-to-Noise Ratio*), uitgedrukt in decibel en berekend als

$$PSNR = 10 \log_{10} \left(\frac{MAX_i^2}{MSE} \right) \quad (3.1)$$

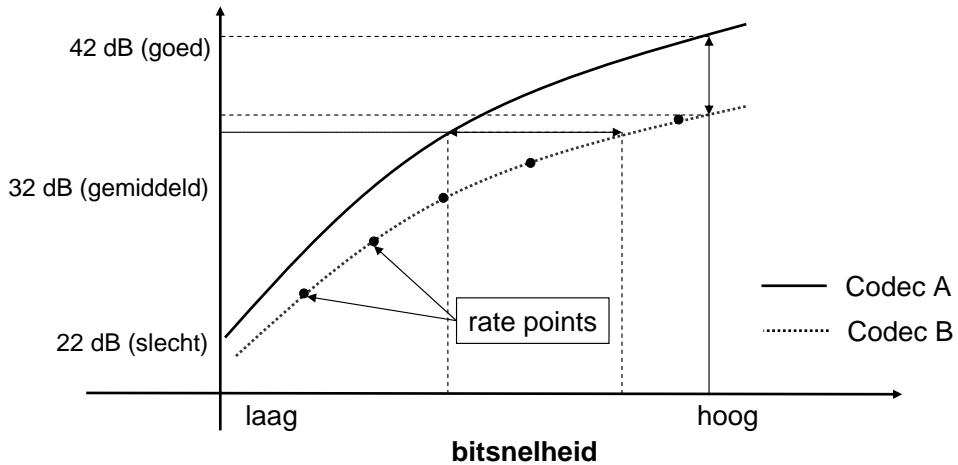
met $MAX_i = 255$ (de maximale waarde van de Y-, U- of V-component van een pixel) en met MSE de *mean squared error* tussen het beeld na decoderen en het originele, ongecomprimeerde beeld, zoals gedefinieerd in sectie 2.2.2. Merk op dat hierbij de teller gelijk is aan de piekwaarde (maximale waarde) van het te meten signaal, terwijl de noemer de ruis voorstelt. Indien het gecomprimeerde beeld gelijk is het originele beeld (dus als de compressie verliesloos is), zal de PSNR dus oneindig groot worden.

Merk op dat PSNR een pure wiskundige metriek die geen rekening houdt met de inhoud van het beeld. Stel dat een beeld 1 pixelrij opschuift, dan zal de PSNR een kleinere waarde krijgen (wiskundig minder gelijkenis tussen de pixels). Uiteraard zal het beeld er nog steeds goed uit zien, en zal dit beeld door het menselijke oog identiek beoordeeld worden. Andere kwaliteitsmetrieken houden wel rekening met de menselijke perceptie van video. Eén van deze metrieken, SSIM (*Eng.: Structural Similarity Measurement*), kan bijvoorbeeld als alternatief gebruikt worden voor PSNR om de kwaliteit uit te drukken in de RD-curves.

Doordat RD-curves de compressie-efficiëntie kunnen visualiseren, worden ze dan ook veelvuldig gebruikt om na te gaan wat de impact is van verscheidene codeertechnieken op de prestaties van een schema voor digitale videocodering en dus ook, bij uitbreiding, om verschillende videocodecs met elkaar te vergelijken. Dit wordt geïllustreerd in Figuur 3.1. In dit voorbeeld kan men stellen dat codec A over de volledige lijn beter presteert dan codec B omdat deze bij alle bitsnelheden (i.e., graden van compressie) een betere kwaliteit aflevert.

In dit practicum zullen we de kwaliteit gewoonlijk uitdrukken aan de hand van de luminantiecomponent (Y) van de gemiddelde PSNR over alle beelden van een sequentie heen. Kleurinformatie zal dus niet in rekening gebracht worden in de RD-curves die later opgesteld zullen worden (dit is trouwens ook meestal het geval in de wetenschappelijke literatuur). Verder hanteren we de afspraak dat 1 kbit/s gelijk is aan 1000 bit per seconde (analoog is 1 Mbit/s gelijk aan 1 miljoen bits per seconde). Dit zijn dan ook de eenheden waarin bitsnelheid later uitgedrukt moet worden.

Om een RD-curve op te stellen voor een bepaalde codec moet deze codec een gegeven videosequentie als input nemen en deze comprimeren (encoderen) aan een aantal verschillende bitsnelheden (hetzij door verschillende quantisatiestapgroottes te gebruiken,



Figuur 3.1: Schematische voorstelling van een RD-curve. Codec A presteert voor alle bitsnelheden beter dan codec B.

hetzij door gebruik te maken van een “rate control”-algoritme, zoals verder in de tekst besproken). Door deze metingen (*rate points*) uit te zetten in een grafiek en de punten te verbinden, krijgt men een RD-curve voor de codec in kwestie (voor de gebruikte testsequentie en de gebruikte codeerparameters). In één grafiek kunnen dus verschillende RD-curves gecombineerd worden (i.e., verschillende codecs, verschillende sequenties, verschillende codeerparameters, of een combinatie).

Merk ook op dat RD-curves opgesteld worden aan de hand van een beperkt aantal *rate points* verspreid over een realistisch bereik van bitsnelheden. RD-curves kunnen bovendien doorgaans het best benaderd worden met een derdegraadsvergelijking van de volgende vorm waarbij R de bitsnelheid voorstelt en a , b , c en d constanten zijn:

$$PSNR = a + bR + cR^2 + dR^3 \quad (3.2)$$

3.2 Rate control

Verschillende graden van compressie kunnen bereikt worden door gebruik te maken van verschillende quantisatiewaarden. In de praktijk wenst men in vele gevallen zodanig te coderen dat de gecodeerde bitstroom een bepaalde bitsnelheid heeft (om bijvoorbeeld verstuurd te kunnen worden over een netwerk met een bepaalde bandbreedte, of om ervoor te zorgen dat het gecodeerde bestand een welbepaalde grootte zou hebben).

Dit mechanisme noemt men *rate control*. Op basis van een opgegeven bitsnelheid zal dit algoritme de quantisatiewaarden tijdens het encoderen zodanig instellen dat de

gewenste bitsnelheid bereikt wordt. Met de ontwikkeling van recente videocodecs kan de quantisatie waarde per beeld worden ingesteld; in H.264/AVC is het zelfs mogelijk om de quantisatie per macroblok te veranderen. Dit laat toe om de gevraagde bitsnelheid heel precies te benaderen.

In deze context zijn twee begrippen belangrijk: CBR (*Constant Bit Rate*) en VBR (*Variable Bit Rate*). In het geval van VBR werkt men (ruwweg gesproken) met een constante quantisatiewaarde. Dit levert een gecodeerde videotroom met een constante visuele kwaliteit op, maar er kunnen grote fluctuaties zitten in de (ogenblikkelijke) bitsnelheid¹. Bij CBR wordt gebruikgemaakt van een algoritme voor rate control. Hierbij is de bitsnelheid dus eerder constant, maar kunnen temporele fluctuaties in visuele kwaliteit optreden. Zo kan bijvoorbeeld een heel moeilijk te coderen scène te weinig bits toegewezen krijgen waardoor de kwaliteit daalt door het wegvalen van teveel hoog-frequente informatie.

¹Een residubeeld met heel lage energie wordt mogelijk met een te lage quantisatiewaarde gequantiseerd.

Hoofdstuk 4

Te gebruiken videocodecs en software

In het eerste deel van deze practicumopgave hebben we kennisgemaakt met de basistechnieken die gebruikt worden bij digitale videocodering. Het gebruik van dergelijke technieken, en vooral de verbeteringen die de laatste jaren ontwikkeld zijn, zorgen voor een steeds betere compressie zonder in te boeten aan kwaliteit.

Om aan te geven hoe de basistechnieken geëvolueerd zijn, zullen we de H.264/AVC-standaard van naderbij bekijken en metingen verrichten omtrent H.264/AVC's codeerefficiëntie. H.264/AVC wordt momenteel beschouwd als de meest krachtige wijdverspreide specificatie voor digitale videocodering wat betreft codeerefficiëntie. De resultaten zullen dan ook een indicatie zijn van de enorme verbeteringen die de H.264/AVC-technieken met zich hebben meegebracht.

Om de RD-curves op te stellen wordt gebruikgemaakt van de VQMT-applicatie (zie Appendix A) voor het berekenen van de beeldkwaliteit van de gedecodeerde sequenties. Hiermee kunnen jullie de PSNR-waarden van de verschillende beelden bestuderen. Het encoderen en analyseren van een groot aantal sequenties zorgt echter voor een grote hoeveelheid repetitief werk. Het gebruik van een scripting-taal laat toe om dit eenvoudig en snel uit te voeren. Hiervoor zullen we in dit practicum gebruikmaken van de platformonafhankelijke scriptingtaal Python¹.

Door middel van Python kan met een minimum aan stappen de gemiddelde PSNR-Y worden bepaald en de lengte van het bestand worden opgevraagd om de gemiddelde bitsnelheid te bepalen. Dit kan in hetzelfde script gebeuren als het oproepen van de programma's die nodig zijn tot het genereren van deze waarden. Daarom wordt voor dit

¹Alle beschikbare functies binnen Python staan beschreven op <http://docs.python.org/2/>

practicum verwacht dat jullie voor elke opgave waarbij commandolijnparameters ingevoerd moeten worden, één script schrijven dat alle output automatisch genereert.

Het is in dit practicum niet de bedoeling om Python volledig aan te leren, maar wel om jullie een beeld te geven van de mogelijkheden en een aantal krachtige methoden aan te halen die bruikbaar zijn in tal van gebieden. Er bestaan verschillende versies van Python². Voor dit practicum maken we gebruik van Python 2.7, maar het staat u vrij een andere Python 2.x versie te gebruiken. In Appendix B worden enkele nuttige commando's gegeven voor dit practicum.

4.1 H.264/AVC: x264

Om de codeerefficiëntie van H.264/AVC te kunnen meten, zullen we in dit practicum gebruikmaken van de x264-encoder. De broncode van deze encoder is vrij beschikbaar op het internet en wordt tegenwoordig in heel wat toepassingen gebruikt. Meer informatie kan gevonden worden op het website van x264: <http://www.videolan.org/developers/x264.html>. Een gecompileerde versie van deze encoder is beschikbaar op Minerva. Op het internet zijn heel wat gecompileerde versies te vinden³; een recente build van deze website (8-bit, Win32) wordt gebruikt voor deze opgave.

Eén van de keuzes die men dient te maken bij het encoderen is het gewenste bestandstype van de gecodeerde stroom. Naast enkele containerformaten, kan men er bij x264 voor kiezen om elementaire stromen te genereren (meestal gebruikt men hiervoor “.264” als extensie). In dit practicum zullen we van dit type gebruikmaken aangezien men de bitsnelheid van een videosequentie typisch berekent aan de hand van de grootte van een elementaire stroom. Dergelijke bestanden kunnen echter niet door alle mediaspelers afgespeeld worden, ook al bieden zij ondersteuning voor H.264/AVC. De meeste mediaspelers verwachten immers dat de digitale media verpakt zitten in een bestandsformaat of een containerformaat (bijvoorbeeld .avi, .mp4, .wmv, ...). Daarom wordt een H.264/AVC-decoder ter beschikking gesteld (gecompileerd op basis van versie JM18.3 van de referentiesoftware⁴). Met het gecomprimeerde bestand als invoer genereert deze decoder een YUV-bestand dat de gedecodeerde videosequentie voorstelt. De aldus bekomen YUV-sequentie kan dan gevisualiseerd worden met een YUV-speler. Voor het gebruik van x264 en de AVC-decoder verwijzen we naar Appendix A.

²Merk op dat Python 3 een andere taal is dan Python 2.x en dus niet noodzakelijk als opvolger moet gezien worden. Beide versies worden parallel verder ontwikkeld.

³Zie bijvoorbeeld <http://x264.nl/>.

⁴Vrij te downloaden via <http://iphome.hhi.de/suehring/tml/download/>

Om de opgaven ten volle te kunnen begrijpen is het nodig om het stuk rond compressie van beelden in deze opgave grondig gelezen te hebben! Al deze technieken beschreven in hoofdstuk 2 worden gebruikt door H.264/AVC. Let wel dat deze standaard nog vele andere technieken voorziet; deze vallen echter buiten het kader van dit practicum⁵.

4.2 H.265/HEVC: x265

In 2013 is de eerste versie van de H.265/HEVC standaard, de opvolger van H.264/AVC, gefinaliseerd. Deze standaard brengt nog extra optimalisaties met zich mee zoals onder andere flexibele blokgroottes van 64×64 , 32×32 , 16×16 en 8×8 pixels in tegenstelling tot de macroblokken van 16×16 pixels in H.264/AVC⁶. Deze standaard kan video's comprimeren met dezelfde kwaliteit als H.264/AVC voor slechts de helft van de bitsnelheid. Het benutten van het volledige potentieel van deze standaard kost echter behoorlijk wat rekenkracht.

Om de codeerefficiëntie van H.265/HEVC te kunnen meten, zullen we in dit practicum gebruikmaken van de x265-encoder. Ook hiervan is een gecompileerde versie beschikbaar op Minerva.

⁵Een goede introductie kan teruggevonden worden op <http://www.vcodex.com/h264.html>

⁶Een volledige introductie van H.265/HEVC is terug te vinden op http://iphome.hhi.de/wiegand/assets/pdfs/2012_12_IIEEE-HEVC-Overview.pdf

Hoofdstuk 5

Opgaven

In de volgende secties worden een aantal opgaven en vragen geformuleerd. Het is de bedoeling dat de antwoorden verwerkt worden in een document waarin duidelijke verwijzingen worden opgenomen naar de nummering van de opgave. Enkele richtlijnen voor het opstellen van het verslag:

- De vormgeving van het document is vrij, maar het formaat van het ingediende document moet een MS Word- of PDF-document zijn.
- Nummer duidelijk jullie antwoorden op de (deel)vragen.
- Het verslag telt maximum 2000 woorden. Voorbij deze limiet wordt niet meer gelezen! Ter info: het modelverslag telt 9 pagina's en ongeveer 1200 woorden.
- Voor veel deelvragen volstaat een antwoord van 1 tot 3 regels tekst (exclusief grafieken en commando's). Als algemene richtlijn geldt dus dat de antwoorden best zo bondig mogelijk gehouden worden (zonder afbreuk te doen aan de correctheid en volledigheid ervan, uiteraard).
- Benoem de assen van de grafieken en zorg er ook voor dat deze duidelijk leesbaar zijn!

Voor het uitvoeren van de metingen wordt gewerkt met de *PartyScene*-sequentie die op de Minerva kan teruggevonden worden (resolutie 416×240 pixels, met een beeldsnelheid van 25 beelden per seconde). Enkel wanneer uitdrukkelijk vermeld wordt om andere of meerdere sequenties te gebruiken, wordt van de *PartyScene*-sequentie afgeweken.

Om de codeerefficiëntie van een encoder te bestuderen, worden RD-curves gebruikt. Bij deze curves wordt de kwaliteit tegenover de bitsnelheid uitgezet. Met kwaliteit bedoelt

men de visuele kwaliteit van het gedecodeerde beeld. Dit kan bekomen worden door de PSNR (*Eng.: Peak Signal-to-Noise Ratio*) te berekenen ten opzichte van het originele beeld. Voor de bitsnelheid gebruikt men de gemiddelde bitsnelheid van een sequentie. In dit practicum kan de gemiddelde bitsnelheid eenvoudig bepaald worden aan de hand van de totale bestandsgrootte.

5.1 In te dienen bestanden

Stuur jullie oplossing door via de dropbox op Minerva, met de volgende bestandsnaam *practicum_videocompressie_MMX.zip* (waarbij jullie X vervangen door jullie groepsnummer). Hierin zit enkel jullie **verslag** en de **drie Python-scripts** (opgaveX.py) die gebruikt zijn voor het genereren van de sequenties en het afleiden van de data van de RD-curves. Voor de Python-scripts wordt bij de evaluatie verondersteld dat de uitvoerbare bestanden en YUV-bestanden zich bevinden in de submappen ‘uitvoerbare_bestanden’ en ‘YUV-bestanden’.

5.2 Opgave 1: x264

- a. In deze oefening wordt nagegaan wat de invloed is van de gebruikte beeldtypes en de gebruikte GOP-structuur. Hiervoor worden RD-curves gebruikt.

Stel RD-curves op voor de volgende gevallen en zet deze in het verslag in één grafiek:

- **Configuratie 1:** Alle beelden zijn I-beelden.
- **Configuratie 2:** Gebruik een GOP-lengte van 16. Dit betekent dat elk 16e beeld een I-beeld moet zijn (het eerste beeld heeft beeldnummer nul). Voor het coderen van de tussenliggende beelden wordt gebruikgemaakt van P-beelden.
- **Configuratie 3:** Gebruik een GOP-structuur waarbij steeds **exact** vier opeenvolgende B-beelden voorkomen en waarbij per GOP juist drie P-beelden zitten. De structuur is dus IBBBBPBBBBPBBBBP.

Maak bij het opstellen van deze RD-curves gebruik van **VBR-codering**. Elke sequentie moet dus een aantal keer gecodeerd worden met een andere quantisatieparameter binnen het bereik [1, 51]. Let bovendien ook op de uitvoeringstijd van het encoderen.

In deze oefening mogen de overige codeerparameters hun standaardwaarde behouden. Kies zelf het aantal *rate points*, maar motiveer uw keuze en zorg voor een realistisch

bereik van bitsnelheden! Bij het maken van de RD-curves moet een Python-script gebruikt worden om de commando's automatisch te laten starten.

- i. Geef de gevraagde grafiek en geef je motivering voor de gebruikte *rate points*.
 - ii. Verklaar de gebruikte commandolijnparameters voor de gevraagde configuraties.
Doe dit in de vorm van een tabel zoals dit voorbeeld:

--tune psnr	encoder afstemmen voor optimale PSNR (gegeven in opgave)
--frames 50	aantal beelden om te encoderen
...	

 - iii. Hoe verhouden de RD-curves zich tegenover elkaar? Wat kun je hieruit afleiden?
 - iv. Kijk ook naar de afstand tussen de verschillende RD-curves. Wat valt hier op?
 - v. Kijk ook eens naar de encodeer-/decodeervolgorde van de beelden van de verschillende configuraties. Hiervoor kijk je bijvoorbeeld naar het POC (*picture order count*) nummer in de encoderlog. Dit nummer is een indicatie voor de afspeelvolgorde van de beelden. Wat valt hier op bij één van de configuraties?
Verklaar waarom dit gedrag optreedt.
 - vi. In welke gevallen zou je in de praktijk wel nog kiezen voor de minst compressie-efficiënte configuratie?
-
- b. In deze oefening bekijken we de impact van de precisie tijdens de bewegingsestimatie op de coderefficiëntie. Hiervoor starten we vanuit configuratie 3 uit de vorige opgave. De precisie van de bewegingsvectoren kan ingesteld worden door gebruik te maken van de parameter `--subme`. De verschillende blokgroottes (i.e., macroblokpartities) die overwogen worden tijdens de bewegingsestimatie kunnen ingesteld worden door middel van `--partitions`. We beperken ons in dit practicum tot de waarden 0 en 11 voor wat betreft `--subme`, en tot "none" en "all" voor wat betreft `--partitions`. Voor de macroblokpartitionering en de precisie van bewegingsvectoren gaan we twee configuraties opstellen. Bij de eerste configuratie wordt geen partitionering gebruikt en zijn de bewegingsvectoren gehele getallen. De tweede configuratie daarentegen zal alle mogelijke macroblokpartities en sub-pixelinterpolatie ondersteunen.
 - i. Stel voor de twee mogelijke combinaties een RD-curve op en combineer die in één grafiek in het verslag. Geef ook de twee gebruikte commando's.
 - ii. Beschrijf de waargenomen invloed van de parameters op de coderefficiëntie en de codeersnelheid en geef hiervoor een mogelijke verklaring.

- c. Tijdens de zoektocht naar de optimale bewegingsvectoren moet steeds een afweging gemaakt worden tussen complexiteit en coderefficiëntie. Probeer in deze opgave twee configuraties te bekomen door de parameters `--me` en `--subme` te variëren. Zorg hierbij wel dat in beide configuraties **alle macroblokpartities** overwogen worden en de optimale GOP-structuur uit vraag 1.a gebruikt wordt. De eerste configuratie moet een zo goed mogelijke coderefficiëntie bekomen, terwijl bij de tweede configuratie de uitvoeringstijd van belang is.
- i. Stel de RD-curves op voor deze twee configuraties in een grafiek in het verslag en geef de twee commando's. Motiveer tevens de keuze van de waarden voor de parameters `--me` en `--subme`.
 - ii. Geef enerzijds een voorbeeld (met motivering) van een toepassing waarbij hoge complexiteit wordt verkozen, en anderzijds een waarbij een lage uitvoeringstijd belangrijk is.
- d. In voorgaande opgaven werd steeds gewerkt met VBR-codering. Stel nu RD-curves op volgens configuratie 2 uit puntje a (GOP-lengte 16, met enkel I en P-beelden), gebruikmakend van CBR-codering. Kies hiervoor gelijkaardige bitsnelheden als voor dezelfde configuratie waarbij je vaste QP gebruikte (VBR). Parameters zoals `--partitions`, `--subme` en `--me` worden op hun standaardwaarde gehouden zoals in puntje a.
- i. Zet zowel de VBR-curve uit puntje a en de CBR-curve in dezelfde grafiek in het verslag. Hoe verhouden de RD-curves van CBR en VBR zich tegenover elkaar? Geef ook het gebruikte commando voor CBR-codering.
 - ii. Maak nu voor zowel CBR als VBR een grafiek met de genummerde beelden op de x-as en de grootte van elk beeld in bytes op de y-as. Stel tevens een grafiek op met de genummerde beelden op de x-as en de PSNR van elk beeld op de y-as. Welk verschil zie je in het gedrag tussen CBR en VBR op gebied van kwaliteit en grootte in bytes van de afzonderlijke beelden? Verklaar dit verschil.
 - iii. Ga er nu van uit dat de beschikbare bandbreedte van het netwerk gelijk is aan de gemiddelde bitsnelheid van de volledige videotraan. Bekijk zowel voor CBR als VBR het grootste beeld in bytes en bereken hoeveel milliseconden het duurt om dit beeld te versturen over het netwerk. Vergelijk dit met de tijd die nodig is om een gemiddeld beeld te versturen voor dezelfde configuratie. Noteer deze tijden in het verslag en bespreek welke gevolgen dit heeft voor de keuze tussen het gebruik van VBR of CBR.
 - iv. Geef voor elk van de volgende vier toepassingen of je eerder CBR of VBR zou gebruiken en motiveer telkens je keuze:

- videobellen
 - video on demand services zoals Netflix
 - video surveillance
 - films opgeslagen op fysieke opslagmedia
- e. In deze laatste deelopgave gaan jullie het effect van hercoderen van video onderzoeken. Hercodering wordt bv. gebruikt bij videobewerking als een logo moet ingebrand worden in een video of als enkele beelden uit de video moeten geknipt worden (zie ook het vorige practicum).
- Codeer hier voor eerst de video op dezelfde manier als in de vorige deelvraag, gebruikmakend van CBR-codering, met een bitsnelheid van 1000 kbps. Deze video wordt vanaf nu de **basisstroom** b_0 genoemd.
- i. Decodeer de basisstroom b_0 en hercodeer hem (na decoderen) opnieuw aan 1000 kbps. Zo krijg je de stroom b_1 . Je hercodeert de stroom tien keer, waarbij b_{n-1} telkens de input is bij de creatie van b_n voor $n \in [1, 10]$. Maak een grafiek met de PSNR op de y-as en het aantal hercoderingen n lopend van nul t.e.m. tien op de x-as. Zet deze grafiek in het verslag. Welk gedrag zie je in deze grafiek? Geef hiervoor een mogelijke verklaring.
 - ii. Decodeer de basisstroom b_0 en hercodeer hem vervolgens aan respectievelijk 1500 kbps ($b_{1,1500}$), 2000 kbps ($b_{1,2000}$), 4000 kbps ($b_{1,4000}$) en 10000 kbps ($b_{1,10000}$). In tegenstelling tot punt (i), vertrek je dus voor elk van deze hercoderingen opnieuw van b_0 . Zet deze punten samen met de bitsnelheid en PSNR van b_0 in een grafiek. Wat zie je als je de PSNR van deze hercoderingen vergelijkt met die van de basisstroom b_0 ?
 - iii. Welke conclusie kun je trekken uit de twee voorgaande testen in verband met beeldkwaliteit na hercodering?

5.3 Opgave 2: Alternatieve compressiealgoritmen en invloed van de beeldinhoud

In deze opgave gaan we twee encoders met elkaar vergelijken: de x264-codec, en de state-of-the-art x265-codec. Bovendien kijken we ook naar de invloed van de beeldinhoud op de compressie-efficiëntie. Voor deze opgave wordt gebruikgemaakt van drie sequenties: *PartyScene*, *RaceHorses* en *Vidyo*.

Stel drie grafieken op met RD-curves voor elke codec (één grafiek per videosequentie, dus in totaal drie grafieken met elk twee curves). Het kiezen van de punten in deze grafieken moet zodanig gebeuren dat er telkens genoeg overlap is in het bereik van de verschillende curves. Het vergelijken van de bitsnelheidsreductie van een nieuwe compressiestandaard bij gelijke kwaliteit is namelijk niet mogelijk als alle drie de curves geen enkel punt hebben met dezelfde kwaliteit.

Gebruik voor beide encoders de optimale GOP-structuur uit de vorige opgave. Zorg er bovendien voor dat de parameters voor `--me` en `--subme` gelijkaardig worden gekozen voor `x264` en `x265`. Gebruik ook `--preset veryslow` bij `x265` zodat zoveel mogelijk vernieuwingen van H.265/HEVC gebruikt worden. De `--partitions` parameter van `x264` moet op zijn standaardwaarde gelaten worden.

- i. Zet de bovenstaande drie grafieken in het verslag.
- ii. Geef de volledige commando's die gebruikt werden om de bitstromen te genereren en motiveer je keuze voor `--me` en `--subme`.
- iii. Vergelijk beide codecs op gebied van compressie-efficiëntie en uitvoeringstijd. Wat concludeer je hieruit?
- iv. Genereer ook een grafiek waarin duidelijk de compressie-efficiëntie voor de drie verschillende sequenties wordt vergeleken. Dit mag voor ofwel `x264` ofwel `x265`. Zet deze in het verslag en geef een verklaring voor het waargenomen gedrag. (*hint: bekijk de YUV-sequenties in een YUV-player*)

5.4 Opgave 3: Kostenplaatje van videocompressie

In de vorige opgave hebben jullie steeds compressie uitgevoerd op videosequenties met een lengte van 2 seconden en een resolutie van 416 bij 240 pixels. In de praktijk zijn de video's natuurlijk langer en hebben deze vaak ook een hogere resolutie die zelfs oploopt tot 3840 bij 2160 pixels (Ultra-High Definition). In deze opgave zullen jullie een beter zicht krijgen op de monetaire kost van videocompressie met behulp van enkele vereenvoudigde cijfers.

Voor een video streaming service moeten de volgende kosten in rekening gebracht worden: de kost van de opslag van de video's, de kost van het uitgaande verkeer en de kost van het omzetten van de geüploaden video naar een uniform formaat. Bijvoorbeeld als alle video's van een provider gecodeerd moeten zijn volgens een bepaalde videostandaard, worden video's die geüpload worden door gebruikers gedecodeerd en vervolgens opnieuw

gecodeerd volgens de uniforme standaard. Aangezien decodeertijd verwaarloosbaar is, wordt voor de kost van het omzetten in de rest van deze opgave enkel encodeertijd in rekening gebracht. In onderstaande tabel worden enkele vereenvoudigde prijzen¹ weergegeven:

Kost voor uitgaand verkeer	€0,07 per GB
Opslagkost	€0,03 per GB
Kost voor rekenkracht	€0,028 per uur

Voor deze opgave gebruiken jullie de sequentie *BasketballDrive* met een resolutie van 1280x720 pixels en een beeldsnelheid van 25 beelden per seconde. Deze sequentie duurt 2 seconden. Jullie mogen deze sequentie gebruiken als een referentie voor een gemiddelde film op een fictieve videowebsite. Als jullie aannemen dat een volledige film ongeveer 7200 seconden duurt, kunnen jullie aan de hand van de bitsnelheid en encodeertijd van deze referentiesequentie berekenen hoeveel de bestandsgrootte en encodeertijd van een volledige film zouden zijn.

In deze opgave is het de bedoeling om een Python-script te schrijven dat bitsnelheden en encodeertijd uitschrijft voor enkele configuraties van de `x265` encoder, waarna jullie een kostenanalyse uitvoeren.

- i. Voer `x265` uit met `--preset medium`, `--frame-threads 1` en `--no-wpp` voor de *BasketballDrive* sequentie met verschillende quantisatieparameters. De laatste twee parameters deactiveren de parallelisatie van `x265`, wat nodig is om een fair zicht op de kosten te krijgen gezien de kost voor rekenkracht berekend is op een enkele CPU.

Bereken voor de verkregen waarden de netwerkkost per gebruiker, de netwerkkost voor 1000 kijkers, de opslagkost en de kost voor rekenkracht voor een film van 2 uur. Bereken ook de totaalkost in het geval van 1000 kijkers en zet deze uit in een grafiek waarbij de bitsnelheid van de video op de x-as staat. Noteer je observatie in het verslag samen met de grafiek.

- ii. Voer dezelfde configuratie uit als in het vorige puntje, maar zet de quantisatieparameter vast op 30 en test de volgende presets² uit: ultrafast, superfast, faster, medium en slower. Bereken de totaalkost voor de volgende aantalen kijkers: 1, 5, 10, 50, 100, 500, 1000, 5000 en 10000. Teken de resulterende waarden uit op een grafiek met een

¹Voor exacte prijzen, zie bv. een cloud hosting website zoals <https://aws.amazon.com>

²Deze presets zijn voorgedefinieerde configuraties die parameters wijzigen zoals gezien in de vorige opgave. Een exacte beschrijving van de presets is te vinden op <https://x265.readthedocs.org/en/default/presets.html>

aparte curve per preset, met het aantal gebruikers op de x-as en de totaalkost op de y-as. Voor de overzichtelijkheid kan je een logaritmische schaal gebruiken voor beide assen.

Noteer ook hier je observatie in het verslag samen met de grafiek. Vermeld ook voor elke preset vanaf hoeveel gebruikers de kost van de rekenkracht minder bedraagt dan de helft van de totaalkost.

- iii. Wat kan je uit de vorige twee puntjes concluderen en welke implicaties heeft dit voor een videowebsite op gebied van kost en gekozen compressieconfiguratie?

Bijlage A

Gebruikte programma's

In deze appendix wordt een kort overzicht gegeven van de programma's die jullie in dit practicum zullen gebruiken.

A.1 H.264/AVC encoder en decoder

Als encoder voor de H.264/AVC-standaard gebruiken we `x264`. Om deze op te roepen voor **VBR-codering** gebruiken we het volgende commando, eventueel uitgebreid met verdere parameters:

```
x264.exe -q [quantisatieparameter] -v -o [output].264  
--tune psnr --fps 25 -frames 50 --input-res 416x240 [sequentie].yuv
```

Voor **CBR-codering** wordt de quantisatieparameter vervangen door een bitsnelheid en wordt `--nal-hrd cbr` toegevoegd om aan te geven dat deze bitsnelheid constant moet gehouden worden. Bovendien worden de vbv-maxrate en vbv-bufsize aangepast zodat de encoder de bitsnelheid constant probeert te houden op beeldniveau¹ Het commando is als volgt, waarbij de grootte van de buffer gelijk is aan de bitsnelheid gedeeld door het aantal beelden per seconde:

```
x264.exe -B [bitsnelheid] -v -o [output].264 --nal-hrd cbr  
--vbv-maxrate [bitsnelheid] --vbv-bufsize [buffergrootte]  
--tune psnr --fps 25 -frames 50 --input-res 416x240 [sequentie].yuv
```

¹In de praktijk wordt de bitsnelheid constant gehouden op een hoger niveau dan beeldniveau, zoals bv. GOP-niveau.

De volledige lijst met x264-parameters kan geraadpleegd worden door `--fullhelp` als parameter mee te geven. Na encoding wordt het resulterende bestand gedecodeerd met de AVC-decoder:

```
avcdecoder.exe -i [output].264 -o [output_dec].yuv
```

A.2 H.265/HEVC encoder en decoder

Als encoder voor de H.265/HEVC-standaard gebruiken we `x265`. De actuele lijst met parameters is te vinden via <http://x265.readthedocs.org/en/latest/cli.html> en is vrij analoog met `x264`. De HEVCDecoder is ook analoog aan de AVCDecoder en is als volgt te gebruiken:

```
hevcdecoder.exe -b [output].265 -o [output_dec].yuv
```

A.3 VQMT

VQMT kan met volgend commando opgeroepen worden om de kwaliteit tussen ‘Original-Video’ en ‘ProcessedVideo’ te berekenen:

```
VQMT.exe [OriginalVideo] [ProcessedVideo] [Height]  
[Width] [Frames] [ChromaFormat] [Output] [Metrics]
```

Hierbij is in dit practicum ‘ChromaFormat’ steeds ‘YUV420’ (4:2:0 chroma-subsampling). Als metriek wordt ‘PSNR’ gebruikt.

Bijlage B

Scripting met Python

Het is de bedoeling om Python te gebruiken in dit practicum. Door Python efficiënt te gebruiken is het mogelijk om de data voor de RD-curves eenvoudig te genereren en in een kommagescheiden bestand (.csv) te plaatsen. Dit bestand kan dan eenvoudig in Excel worden geopend, zodanig dat de RD-curves snel gevisualiseerd kunnen worden.

Tip: Het gebruik van Python kan het repetitieve werk verminderen. Maar Python maakt het snel uittesten van de juiste commandolijnparameters omslachtig. Daarom is het aan te raden om in de command prompt de juiste commandolijnparameters te bepalen en pas daarna deze in een Python-script op te nemen. Vervolgens kan de volledige set van parameters overlopen worden.

B.1 Uitvoeren van Python scripts

Python code kan worden gebruikt op twee manieren. Enerzijds kan de Python-interpreter opgeroepen worden in de commandolijn met het commando `Python` (mits de installatie correct is uitgevoerd) en kunnen de Python-commando's worden ingegeven. Dit laat toe om eenvoudig het effect van commando's in Python te zien. Anderzijds kan er een tekstbestand worden aangemaakt (met de extensie .py) dat vanuit de commandolijn (`Python script.py`) wordt opgeroepen. Python zal dit .py bestand dan inlezen en uitvoeren.

B.1.1 Gebruik van Python 2.x via Athena

Om Python 2.x via Athena te gebruiken, start je de Python interpreter (2.7) op die op Athena beschikbaar is. Deze interpreter start normaalgezien op met als huidige locatie

de home-folder van de H-schijf. De lokatie veranderen naar een lokale schijf kan als volgt voor bv. de lokale map C:\Folder:

```
>>> import os  
>>> os.chdir("\\\Client\\C$\\Folder")
```

Een python script in deze map uitvoeren gebeurt dan met het volgende commando:

```
>>> execfile("script.py")
```

B.2 Operaties op (tekst)bestanden

De bijgevoegde software voor videokwaliteit te meten (VQMT) heeft de mogelijkheid om voor elk beeld de PSNR-waarden te genereren. Python laat toe om deze output eenvoudig uit te lezen. In de meeste programmeertalen moet hiervoor een for-lus gebruikt worden die alle regels uitleest uit het bestand. Python ondersteunt daarentegen ook het gebruik van zogenaamde '*list comprehensions*'. Deze constructie is omgeven door '[]' en is een efficiënte manier om een lus-structuur uit te schrijven, die tevens ook efficiënt naar machinetaal wordt omgezet door optimaal gebruik te maken van de onderliggende processorstructuren.

De bijgevoegde VQMT-software geeft voor elk beeld de PSNR-waarden. Genereer zelf (eenmalig) handmatig de PSNR waarden door middel van de VQMT-software. Dit bestand zal als voorbeeld dienen voor de opbouw naar het Python script. Hierin zien we voor elke lijn twee getallen; het eerste getal komt overeen met het beeldnummer, het tweede is de PSNR-Y waarde.

De volgende voorbeeldcode is een list comprehension en laat toe om alle regels van een bestand in een lijst (*PSNR_waarden*) in te lezen.

```
with open('bestand', 'r') as PSNR_bestand:  
    PSNR_waarden = [x for x in PSNR_bestand.readlines()]
```

De constructie geeft alle waarden x waarvoor geldt dat x in de lijst van uitgelezen lijnen zit. Merk op dat `.readlines()` de file pointer naar het einde van het file zal zetten en dus geen bestandsoperaties meer kunnen uitgevoerd worden tenzij een `.seek()`-operatie wordt uitgevoerd of het bestand wordt gesloten en opnieuw geopend. Hierdoor zal de resulterende lijst leeg zijn als de `.readlines()` functie voor een tweede maal op hetzelfde bestand wordt uitgevoerd.

We hebben nu in *PSNR_waarden* alle lijnen uit ons VQMT-bestand staan. Echter, op het einde van het bestand zien we een overzicht van de gemiddelde waarden. Om enkel het gemiddelde te verkrijgen, kunnen we de ingelezen lijst eerst filteren. Python laat dit toe door middel van een if-statement in de list comprehension te plaatsen.

```
PSNR = [x for x in open('bestand', 'r').readlines() if filter_statement]
```

Hierbij worden enkel de ingelezen lijnen in rekening gebracht waarbij voldaan wordt aan **filter_statement**. Door te verifiëren of “average” in *x* voorkomt, wordt enkel deze lijn teruggegeven door de list comprehension.

```
PSNR = [x for x in open('bestand', 'r').readlines() if "average" in x]
```

Hierdoor bevat PSNR nu alle lijnen waarin “average” voorkomt. Door de lijst te indexeren, krijgen we één specifieke lijn. We kunnen het *y*-de element uit de lijst selecteren met:

```
PSNR[y]
```

De functie `.split()` zal de lijn splitsen en een lijst met alle element uit de lijn weergeven. Standaard wordt hiervoor een spatie als splitsingskarakter gebruikt. Het VQMT-outputbestand is echter een kommagescheiden bestand. De gesplitste lijst op de juiste manier indexeren zorgt dat we de gemiddelde PSNR-Y waarde van de sequentie kennen:

```
PSNRY = PSNR[0].split(',')[1]
```

Deze volledige beschreven procedure voor het lezen van de gemiddelde PSNR-waarde kan ook overzichtelijk in één lijn code geschreven worden:

```
PSNRY = [x for x in open('bestand', 'r').readlines()
         if "average" in x][0].split()[1]
```

Ook de totale bestands grootte kan snel en eenvoudig bepaald worden door middel van Python. In Python laat de module `os` ons toe om operaties uit te voeren die eigen zijn aan het besturingssysteem. Een module wordt ingeladen door bovenaan het Python-script een `import` commando te specificeren:

```
import os
```

Het commando `os.path.getsize(bestand)` geeft ons de bestands grootte van `bestand`. Python voorziet een aantal mogelijkheden om padnamen en bestandsnamen onafhankelijk van het besturingssysteem te concateneren (bv. Windows gebruikt ‘\’ terwijl Linux ‘/’ gebruikt).

```
bestand = os.path.join('..', 'mapnaam', 'submap', 'naam')
```

Hierbij dient opgemerkt te worden dat dezelfde map-navigatie als bij de commandolijn kan gebruikt worden. In het voorgaande voorbeeld zorgt ‘..’ dat eerst naar de bovenliggende map wordt gegaan en vervolgens het bestand *mapnaam\submap\naam* wordt geselecteerd. Een map die meerdere niveaus hoger ligt kan aangesproken worden door verschillende instanties van ‘..’ te combineren.

B.3 Oproepen van uitvoerbare programma’s via Python

Python laat ook eenvoudig toe om programma’s op te roepen. Dit laat toe om in een enkel bestand een volledige keten op te bouwen die alle programma’s oproept en daarna de output verwerkt.

Het starten van programma’s wordt gedaan door de functie `Popen` uit de module `subprocess` op te roepen met de juiste parameters. Hiervoor dient eerst de juiste module geïmporteerd te worden.

```
import subprocess
outFile = open('Output.txt', 'w')
errFile = open('Error.txt', 'w')
subprocess.Popen(args, stdout=outFile, stderr=errFile).communicate()[0]
```

Hierbij zijn ‘errFile’ en ‘outFile’ de bestanden naar waar de `stdout` en `stderr` worden weggeschreven. Het is mogelijk om hiervoor hetzelfde bestand te gebruiken:

```
subprocess.Popen(args, stdout=outFile, stderr=outFile).communicate()[0]
```

De argumenten van de `Popen` functie worden in lijst `args` meegeven. Hierbij moet elke parameter als een afzonderlijk element van de lijst worden meegegeven als een string. Als voorbeeld nemen we het commando om de PSNR te bepalen met VQMT.

```

import subprocess
outFile = open('Output.txt', 'w')
errFile = open('Error.txt', 'w')
args=["VQMT.exe", "OriginalVideo.yuv", "ProcessedVideo.yuv",
      "240", "416", "50", "YUV420", "VQMT_output", "PSNR"]
subprocess.Popen(args, stdout=outFile, stderr=errFile).communicate()[0]

```

Het gebruik van Python wordt pas efficiënt als een opdracht verschillende keren moet uitgevoerd worden, bv. met verschillende parameters. Door de commando's in een lus te plaatsen dient de structuur maar één keer uitgeschreven te worden. Echter dient er dan wel rekening mee gehouden te worden dat de bestanden elkaar overschrijven. Dit kan eenvoudig opgelost worden met het concateneren van strings.

Laten we als voorbeeld terug het VQMT-outputbestand openen: In dit voorbeeld hebben we reeds de quantisatiewaarde opgenomen in de verschillende bestandsnamen bij het genereren van de VQMT-bestanden (in dit voorbeeld 0,2,4 en 6). Dankzij Python kunnen we nu eenvoudig over deze punten itereren en voor elk van deze punten de PSNR berekenen. Let wel op dat Python gebruikmaakt van indentaties om code deel te laten uitmaken van een for-lus of een if-constructie. Deze indentatie kan een spatie of een tab zijn.

```

import subprocess
for QP in [0,2,4,6]:
    OutputFile = open('Output_' + QP + '.txt', 'w')

```