Practicum: JPEG

Multimedia - Multimediatechnieken

Deadline: zie Ufora

1 Beeldcompressie

Beeldcompressie heeft tot doel pixels om te zetten in een binaire stroom die minder opslagruimte inneemt dan de oorspronkelijke ruwe pixels. Er zijn verschillende beeldformaten te onderscheiden:

- **Ruwe beeldformaten:** Formaten die (RGB)-pixels in niet-gecomprimeerde vorm opslaan, zijn bijvoorbeeld BMP en PPM.
- **Verliesloze formaten:** Formaten die de afbeelding verkleinen, maar enkel zodanig dat er geen wijziging van de pixels optreedt. Van deze formaten kan geen compressiefactor groter dan 1 op 5 verwacht worden. Voorbeelden hiervan zijn PNG en JPEG Lossless.
- **Verlieshebbende formaten:** Formaten die afbeeldingen kunnen verkleinen tot eender welke grootte mits inboeting aan kwaliteit. Voorbeelden hiervan zijn JPEG, JPEG2000 en WebP.

Bij het uitvoeren van beeldcompressie is het belangrijk om de structuur van Figuur 1 in het achterhoofd te houden.

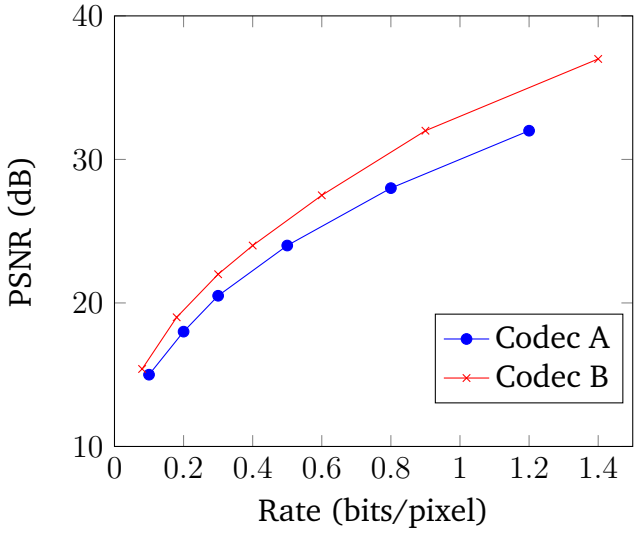
Figuur 1: Blokdiagram van de compressie en decompressie workflow.

2 Evaluatie van compressie

2.1 Rate-Distortie (RD) curves

RD-curves zijn een belangrijk instrument om de relatie tussen compressie en kwaliteit weer te geven. **De rate** drukt het aantal benodigde bits per zekere hoeveelheid multimedia uit (b.v.

aantal bit per: seconde, frame, pixel). In het geval van afbeeldingen wordt typisch gekozen voor het aantal bit per pixel. Op deze manier kan je ook afbeeldingen van verschillende resoluties met elkaar vergelijken. **De distortie** drukt uit hoeveel afwijking er is tussen het opgeslagen signaal en het originele signaal. Een hiervoor vaak gebruikte metriek is mean squared error (MSE), of een directe afgeleide hiervan: de peak Signal-to-Noise ratio (PSNR). Let op dat de PSNR eerder de kwaliteit uitdrukt, aangezien een hogere PSNR een betere kwaliteit (of dus kleinere fout) weerspiegelt. Desalniettemin wordt PSNR vaak gebruikt in RD-curves.



Figuur 2: Schematische voorbeeld van twee RD-curves. Codec B presteert voor alle rates beter dan codec A.

Doordat RD-curves de compressie-efficiëntie kunnen visualiseren, worden ze dan ook veelvuldig gebruikt om verschillende codecs met elkaar te vergelijken, of om de impact van verschillende configuraties van dezelfde codec te onderzoeken. Dit wordt geïllustreerd in Figuur 2. In dit voorbeeld kan men stellen dat codec B over de volledige lijn beter presteert dan codec A omdat deze over de range van vergelijkbare rates een hogere PSNR oplevert.

2.2 Subjectieve kwaliteit

Meestal is de enige sluitende manier om een afbeelding op kwaliteit te beoordelen visuele inspectie. Dit is meestal ook de meest eenvoudige manier, maar hangt af van de ervaring/expertise van de persoon die de afbeelding bekijkt en beoordeelt. Tijdens het visueel beoordelen van een afbeelding houd je rekening met het feit dat artefacten (fouten zichtbaar in het beeld) ook afkomstig kunnen zijn van andere factoren zoals:

- **Beeldscherm**: Het beeldscherm kan niet gecalibreerd zijn en verschillende beeldschermen kunnen dus verschillende kleurtinten weergeven.
- **Weergavesoftware**: Software kan bij de weergave van afbeeldingen filters toepassen. Om het verschil in weergavesoftware te ondervinden is het goed om een testafbeelding

zoals `test_circle` te openen in irfanview, paint of GNU Image Manipulation Program (gimp). Merk op welke software je best niet meer gebruikt in de loop van dit practicum.

- **Weergavegrootte**: Bij het kleiner of groter weergeven van een afbeelding worden schalingsfilters toegepast. Het is aangewezen om afbeeldingen op hun natuurlijke grootte te bekijken zodat er geen invloed is van schalingsfilters.
- **Op afstand werken**: Bij het op afstand overnemen van een PC of het op afstand uitvoeren van software worden de beelden gecomprimeerd doorgestuurd tussen de werkserver en de client. Het is belangrijk de vraag te stellen of deze transmissie impact heeft op de kwaliteit van de te testen afbeelding.

3 Voorbereiding

Dit is een CMake project. Je maakt een nieuwe folder '`build`' in de map waar de source files staan. Daarin komen de build files die CMake zal genereren. Hoe je CMake uitvoert hangt wat af van welk operating systeem je hebt, en welk build system je wil gebruiken.

3.1 Windows - Visual Studio

Hier zijn twee opties:

- **Visual Studio embedded CMake**: Visual Studio heeft CMake aan boord. Je kan dus met Visual Studio de map openen waarin de `CMakeLists.txt` staat, en normaal zal alles werken.
- **CMake zelf installeren**: Je kan CMake ook zelf aanroepen waarna je de geproduceerde Solution file `*.sln`) kan openen met Visual Studio.

Belangrijk: Blijkbaar zijn er virusscanners op Windows (zoals AVG), die voorkomen dat een executable bestanden schrijft naar de 'Documents' map in de thuismap van de gebruiker. Plaats daarom dit practicum ergens anders (bijvoorbeeld in de thuismap, maar in een andere submap).

Belangrijk: CMake build files gebruiken absolute paden. Het verzetten van een project nadat CMake bestanden gegenereerd werden werkt dus niet. Indien je dit wel wilt doen, verwijder je best de CMake output bestanden in de build folder en voer je CMake opnieuw uit.

Hint: Je kan code debugging (Visual Studio → Debug → Start Debugging) door "breakpoints" te zetten (Visual Studio → Debug → Toggle Breakpoint) en door de code te stappen (Visual Studio → Debug → Step Into en Visual Studio → Debug → Step Over).

3.2 Linux / macOS - Makefile

Op macOS of Linux kan je CMake gebruiken om een Makefile project genereren. Dit kan als volgt:

```
mkdir build
cd build
cmake -DCMAKE_BUILD_TYPE=Debug ..
```

Het laatste argument `..` verwijst naar de bovenliggende map, dewelke de `CMakeLists.txt` bevat. Nu het Makefile project gemaakt is (met instructies voor de compiler om debug symbols te genereren zodat je kan debuggen) kan je het project compileren (op bijvoorbeeld 4 CPU-threads) met:

```
make -j4
```

4 Opgave

4.1 JPEG encoder

In deze eerste sectie gaan we de JPEG encoder leren kennen. Zorg dat je al ingelezen bent in de theorie rond JPEG door Sectie 6 in deze opgave te lezen. Dit is een gedeelte uit het JPEG hoofdstuk van het handboek. Jullie krijgen in de map 'src' broncode gegeven. Deze bestaat uit:

- **`jpeg_encoder.hpp`** / **`jpeg_encoder.cpp`** Afbeeldingen met JPEG encoderen.
- **`jpeg_decoder.hpp`** Header-only JPEG decoder.
- **`psnr.hpp`** Berekening van PSNR.
- **`main.cpp`** Verschillende componenten samengebracht in dit practicum.
- **`CMakeLists.txt`** Instructies voor CMake over hoe het project gecompileerd moet worden.

Beantwoord volgende vragen in de voorziene antwoordbestanden (b.v. `vraag-1a.txt`):

- **Vraag 1a** Inspecteer de `writeJpeg()`-functie en geef welke functieargumenten voor een gegeven input afbeelding kunnen gewijzigd worden om **de compressie** te beïnvloeden.
- **Vraag 1b** In welke C++ functie(s) bevinden zich de basiscomponenten van de JPEG-encoder zoals gevisualiseerd in Figuur 3, namelijk DCT, kwantisatie, zigzagscan en entropie-encodering?

- **Vraag 1c** Wat is/zijn de variabelenaam of variabelenamen voor de kwantisatiematrices zoals aangegeven in Figuur 3 en zoals meegestuurd/opgeslaan in de JPEG header?

4.2 Compressie van verschillende afbeeldingen

Comprimeer de gevraagde afbeeldingen door het `main.cpp` bestand uit te breiden zodat de volgende compressietests automatisch uitgevoerd kunnen worden. Implementeer hierbij volgende combinaties van parameters:

- Kwaliteitsparameter: varieer van de kwaliteitsparameter van 100 naar 5 in stappen van 5.
- Doe de compressie zowel met chroma subsampling als zonder.
- Doe dit voor de volgende beelden:
  - alle synthetische beelden (`test_*.ppm`),
  - één van `big_*.ppm` afbeeldingen,
  - drie van de `kodim_*.ppm` afbeeldingen,
  - `artificial.ppm`,
  - `flower_foveon.ppm` en/of `spider_web.ppm`,
  - de afbeeldingen (`*_iso_*.ppm`), dewelke afbeeldingen zijn van de dezelfde scene, maar getrokken met verschillende camerasensorgevoeligheid (ISO-waarde).

PPM Hiertoe zal je moeten code schrijven om een PPM bestand in te lezen. Maak dus één '`struct ppm`' en één functie '`ppm read_ppm(std::string filename)`'. Schrijf hier dus eenvoudige procedurale code die het PPM bestand inleest in memory. Zoek zelf op hoe het PPM formaat werkt.

Hint: Probeer gebruik te maken van C++ mechanismen en hulpfuncties in plaats van C. Wanneer je bijvoorbeeld `std::ifstream` en `std::stringstream` gebruikt, kan de code *enorm* kort en elegant zijn. Dit zal helaas niet het geval zijn wanneer je gebruikt maakt van C (zoals `fopen()`, `fscanf()`, `strtok()`, etc...). Bekijk dus zeker de `operator>>()` operators van deze twee klassen. De pixels van de PPM file zitten in dezelfde byte-volgorde als nodig voor door te geven aan de JPEG encoder: rij per rij. Per rij, de pixels van links naar rechts. Per pixel 3 byte: R, G, en B. Je kan dit dus in één keer inlezen, er is geen nood om loops te programmeren.

Informatie: Wanneer je files *open*t op basis van een relatief pad (Engels: *relative path*), zal dit altijd gebeuren relatief ten opzichte van de huidige werkmap (Engels: *current working directory (cwd)*, *working directory*). Dit wil zeggen dat als je `afbeeldingen/test_star.ppm` gebruikt als pad, het belangrijk is dat je weet wat de working directory is, en die dus correct ingesteld staat.

- **Windows**: Dit is meestal het pad waar de executable in staat. Onder Visual Studio is dit dus vaak iets als `./out/Debug-x64/`. Je kan dit vermoedelijk wel aanpassen ergens in één of ander settings window, maar dat is ongeveer elk jaar anders, dus zoek zelf op hoe je dat doet in de huidige versie van Visual Studio.
- **Unix**: Dit is gewoon de working directory van jouw shell. Je kan dus met `cd` naar de map gaan waarin de relatieve paden die je gebruikt kloppen. Eventueel zal je moeten het pad van de executable specificeren om het te kunnen uitvoeren vanuit de andere working directory. Bijvoorbeeld: `./build/pract_jpeg`. Je kan altijd controleren wat jouw huidige werkmap is met het `pwd` commando (staat voor *print working directory*).

PSNR Hiertoe zal je moeten code schrijven om de PSNR te berekenen als kwaliteitsmetriek tussen twee afbeeldingen. Bekijk hiervoor `psnr.hpp` en implementeer de gegeven functie. Let dus opnieuw goed op met datatypes.

Informatie: Je merkt al heel snel dat dit zeer veel JPEG encodering zijn die je moet uitvoeren. Maak dus gebruik van de "thread pool" infrastructuur voorzien door `ctpl` om alle cores van jouw multicore processor te benutten. Je kan jobs naar de thread pool submitten met de `ctpl::threadpool::push()` functie. De thread pool zal met een vast aantal operating system niveau (OS-level) threads alle jobs zo snel mogelijk verwerken. Een thread pool is nuttig omdat je even veel threads als processor cores kan gebruiken, en je deze bovendien blijft hergebruiken over verschillende jobs heen, in plaats van telkens een nieuwe thread te spawnen voor elke job. Als je gewoon één thread per job zou starten ga je allicht heel wat meer threads starten dan dat er cores zijn, en zal de processor heel wat tijd verliezen aan de zogenaamde "context switches". Een context switch is de (trage!) operatie om één thread te onderbreken, de CPU state van die thread op te slaan, een nieuwe thread te kiezen, die thread zijn state in de CPU terug in te laden, en dan verder te werken. Het spreekt voor zich dat het efficiënter is om het aantal threads gelijk te houden aan het aantal cores om de context switches te minimaliseren, maar toch parallellisme van de multicore processor te benutten.

Je denkt dus best: één job is één onafhankelijke hoeveelheid werk. In dit geval bijvoorbeeld het encoderen van één JPEG afbeelding. Denk hierbij ook na over hoe je er voor zorgt dat alles *thread-safe* is. Gebruik zonodig `std::future<>` of `std::mutex` om correcte synchronisatie te voorzien.

Overzichtsgrafiek Stel één grote grafiek op waarbij alle afbeeldingen weergegeven worden met een RD curve. Je brengt dus alle RD-curves in één grafiek samen. Zorg er voor dat de grafiek duidelijk en compleet is (assen benoemd, legende, etc...). Hiervoor mag je gebruiken wat je wil. Aanbevolen is matplotlib of seaborn, maar spreadsheetsoftware moet ook werken. Het opstellen van de grafiek moet je dus niet noodzakelijk in C++ doen (bijvoorbeeld Python met matplotlib). Sla deze grafiek op als PNG met de naam '`rd-curves.png`'. Maak twee aparte curves voor de encodering *met* en *zonder* subsampling.

Hint: Scroll terug omhoog en lees nu het Sectie 2.1 over RD-curves indien je dat nog niet gedaan hebt. Hierbij verwachten we dus curves die PSNR uitdrukken in functie van de rate in bits per pixel. Indien je denkt dat het nuttig is, mag je gerust een logaritmische schaal gebruiken voor de rate-as (dit kan nuttig zijn, aangezien PSNR zelf al logarithmisch is van aard; zie practicum GIF of Wikipedia).

Informatie: C++ is een taal die meer kan dan de meeste talen, maar dat komt met de kost dat je beter moet weten wat je aan het doen bent, en dat een fout maken tot subtiele gevolgen kan leiden die niet makkelijk opspoorbaar zijn zonder de juiste tools. Enkele tips om te debuggen in geval van problemen zijn:

- **Maak een Debug build**: Een debugger wordt meestal pas nuttig en handig om gebruik van te maken wanneer je een Debug build maakt. Dit is een build waarvan de compiler de binary code weinig geoptimaliseerd heeft, en accurate debug informatie (variabele namen, adressen, lijnnummers, etc...) heeft opgeslagen. Wanneer je dan met ASan of Valgrind (zie hieronder) of met een debugger een fout opspoort krijg je veel duidelijkere info. In Visual Studio is er bovenaan een combobox waar je kan switchen tussen Release en Debug. Voor Unix genereer je jouw CMake project in Debug mode (zie hoger).
- **Address Sanitizer**: Maak gebruik van *address sanitizer*. Dit is een instrumentatietechniek van de compiler in de executable. Memory allocaties, deallocaties, accesses en writes, zullen met heel wat extra controles verlopen (met weinig overhead). Address sanitizer volgt alles op en heeft en interne boekhouding over het gebruikte geheugen, en kan als gevolg daarvan veel fouten opsporen. Er is zowel support voor onder Windows als onder Unix. Deze feature moet je inschakelen via `CMakeLists.txt`. Voor Windows, bekijk: `https://docs.microsoft.com/en-us/cpp/sanitizers/asan?view=msvc-170` en voor g++ of clang, bekijk: `https://stackoverflow.com/a/70272150/155137`.
- **Valgrind**: Valgrind is een enorm krachtige tool, die nog preciezer fouten kan opsporen, maar heeft helaas veel overhead. Valgrind is enkel stabiel ondersteund op Linux. Na installeren kan je jouw programma uitvoeren met bijvoorbeeld: `valgrind ./pract_jpeg`. Combineer Valgrind en ASan niet met elkaar.

Beantwoord volgende vragen in de overeenkomende antwoordbestanden door een ketting

van implicaties te maken. Een ketting van implicaties kan bijvoorbeeld zijn:

$$\text{meer pixels} \implies \text{meer getallen} \implies \text{meer bytes}$$

Deze ketting kan je dus lezen als *"Hoe meer pixels, hoe meer getallen, hoe meer bytes"*. Om deze ketting als antwoord te geven maak je een bestand dat er zo uit ziet:

```
meer pixels
meer getallen
meer bytes
```

Dit bestand telt dus even veel regels als elementen uit jouw ketting. Je mag ook twee elementen tegelijk geven in één schakel van de ketting: deze scheid je met '`&`'. Zo kan je bijvoorbeeld zeggen:

$$\begin{aligned} &\text{meer pixels \& JPEG-header grootte constant} \\ \implies &\text{kleiner de JPEG-header ten opzichte van de totale grootte} \\ \implies &\text{lager de rate} \end{aligned}$$

Hiervoor maak je een antwoordbestand met volgende inhoud:

```
meer pixels & JPEG-header grootte constant
kleiner de JPEG-header ten opzichte van de totale grootte
lager de rate
```

Kies als eerste woord voor elk element uit de ketting uit: *meer/minder*, *hoger/lager*, *groter/lager*, *langer/korter*, *beter/slechter*, *met/zonder*, *sterker/zwakker*. Wanneer er verschillende implicatiekettingen van toepassing zijn voor een gegeven fenomeen mag je deze onder elkaar noteren in het antwoordbestand, gescheiden door een lege lijn.

Hint: Let ook goed op de caveats opgelijst in Sectie 2.2.

Belangrijk: Dit zijn inzichtsvragen. Probeer dus te begrijpen en implicatiekettingen te maken die aantonen dat je begrijpt hoe iets werkt en waarom iets gebeurt.

- **Vraag 2a** Verklaar met behulp van een implicatieketting waarom sommige natuurlijke afbeeldingen een lagere rate hebben dan andere bij gelijkaardige PSNR. Voeg één leesbare RD-grafiek toe (die meerdere RD-curves bevat). Sla deze grafiek op als `vraag-2a.png`.
- **Vraag 2b** Verklaar het verschil in rate of PSNR tussen de synthetische beelden en de natuurlijke beelden. Ter ondersteuning van jouw ketting, kies 2 afbeeldingen waarvoor je de RD-curves in één grafiek voorstelt (let erop dat de grafiek volledig is). Sla deze grafiek op als `vraag-2b.png`.

- **Vraag 2c** Beschouw voor een selectie afbeeldingen met gelijkaardige PSNR de subjectieve kwaliteit. Verklaar één of meerdere *interessante* fenomenen aan de hand van één of meer implicatiekettingen. Let op de caveats van Sectie 2.2.

- **Vraag 2d** Ga op zoek naar een afbeelding waarbij het gebruik van chroma subsampling een grote impact heeft op de kwaliteit. Verklaar hoe dit komt met een implicatieketting. Vermeld in het eerste element van de ketting ook de bestandsnaam van de afbeelding die je koos.

- **Vraag 2e** Ga op zoek naar een afbeelding waarbij het gebruik van chroma subsampling zeer weinig impact heeft op de kwaliteit. Verklaar hoe dit komt met een implicatieketting. Vermeld in het eerste element van de ketting ook de bestandsnaam van de afbeelding die je koos.

- **Vraag 2f** Onderzoek de impact van de ISO instelling voor filmgevoeligheid bij JPEG kwaliteit 90. Doe dit aan de hand van `leaves_iso_200.ppm`, `leaves_iso_1600.ppm`, `nightshot_iso_100.ppm`, en `nightshot_iso_1600.ppm`. Observeer en verklaar met een implicatieketting.

- **Vraag 2g** Onderzoek de impact van de ISO instelling voor filmgevoeligheid bij JPEG kwaliteit 40. Observeer en verklaar met een implicatieketting.

- **Vraag 2h** Onderzoek het effect van JPEG kwaliteitsparameter 100. Wat stel je vast? Verklaar aan de hand van een implicatieketting.

  Tip: In de aangeleverde JPEG decoder kan je een "preprocessor directive" gebruiken, namelijk `DECODERDEBUG`, om de kwantisatiematrix die de decoder terugvindt in het JPEG bestand, te printen.

4.3 Compressie met aangepaste JPEG-encoder

Voeg een parameter toe aan de `writeJpeg` functie waarmee het aantal DCT coëfficiënten (n) dat behouden moet blijven, doorgegeven kan worden. Comprimeer de afbeelding zodat de eerste n DCT coëfficiënten in zigzag volgorde behouden blijven en de rest van de hogere frequenties volledig verdwijnen, en bijgevolg ook niet geëncodeerd worden.

- **Vraag 3a** Encodeer `test_noise.ppm` en `artificial.ppm` nu enkel met de laagste $n = 2$ DCT coëfficiënten (in zigzag volgorde). Voeg deze JPEG bestanden toe aan jouw submissie met de naam `artificial_3a.jpg` en `test_noise_3a.jpg`.

- **Vraag 3b** Doe nu hetzelfde voor de laagste $n = 3$ DCT coëfficiënten (in zigzag volgorde). Voeg deze JPEG bestanden toe aan jouw submissie met de naam `artificial_3b.jpg` en `test_noise_3b.jpg`. Welke extra patronen zie je nu verschijnen in vergelijking met de vorige deelvraag? Hoe zijn deze extra patronen ontstaan? Formuleer een tekstueel

antwoord op deze twee vragen in `vraag-3b.txt` door middel van één zin per vraag. Jouw antwoordbestand bevat dus *exact* twee zinnen.

5 In te dienen bestanden

Stuur jullie oplossing door via de 'Opdrachten' op Ufora, met de bestandsnaam '`practicum_JPEG.zip`'. Hierin zitten jullie antwoordbestanden (`*.txt` en de gevraagde grafieken als `*.png`), samen met map '`source`' met jullie broncode.

6 The JPEG Standard

JPEG is an image compression standard developed by the Joint Photographic Experts Group. It was formally accepted as an international standard in 1992. JPEG consists of a number of steps, each of which contributes to compression. We will look at the motivation behind these steps, then take apart the algorithm piece by piece.

6.1 Main Steps in JPEG Image Compression

As we know, unlike one-dimensional audio signals, a digital image f (i, j) is not defined over the time domain. Instead, it is defined over a spatial domain-that is, an image is a function of the two dimensions i and j (or, conventionally, x and y). The 2D DCT is used as one step in JPEG, to yield a frequency response that is a function F(u, v) in the spatial frequency domain, indexed by two integers u and v. JPEG is a lossy image compression method. The effectiveness of the DCT transform coding method in JPEG relies on three major observations:

Observation 1. Useful image contents change relatively slowly across the image- that is, it is unusual for intensity values to vary widely several times in a small area-for example, in an 8×8 image block. Spatial frequency indicates how many times pixel values change across an image block. The DCT formalizes this notion with a measure of how much the image contents change in relation to the number of cycles of a cosine wave per block.

Observation 2. Psychophysical experiments suggest that humans are much less likely to notice the loss of very high-spatial frequency components than lower frequency components.

JPEG's approach to the use of DCT is basically to reduce high-frequency contents and then efficiently code the result into a bitstring. The term spatial redundancy indicates that much of the information in an image is repeated: if a pixel is red, then its neighbor is likely red also. Because of Observation 2 above, the DCT coefficients for the lowest frequencies are most important. Therefore, as frequency gets higher, it becomes less important to represent the DCT coefficient accurately. It may even be safely set to zero without losing much perceivable image information.

Clearly, a string of zeros can be represented efficiently as the length of such a run of zeros, and compression of bits required is possible. Since we end up using fewer numbers to represent the pixels in blocks, by removing some location-dependent information, we have effectively removed spatial redundancy.

JPEG works for both color and grayscale images. In the case of color images, such as YCbCr, the encoder works on each component separately, using the same routines. If the source image is in a different color format, the encoder performs a color-space conversion to YCbCr. As discussed in Chap. 5, the chrominance images Cr and Cb are subsampled: JPEG uses the 4:2:0 scheme, making use of another observation about vision:

Observation 3. Visual acuity (accuracy in distinguishing closely spaced lines) is much greater for gray ("black and white") than for color. We simply cannot see much change in color if it occurs in close proximity- think of the blobby ink used in comic books. This works simply because our eye sees the black lines best, and our brain just pushes the color into place. In fact, ordinary broadcast TV makes use of this phenomenon to transmit much less color information than gray information.

When the JPEG image is needed for viewing, the three compressed component images can be decoded independently and eventually combined. For the color channels, each pixel must be first enlarged to cover a 2×2 block. Without loss of generality, we will simply use one of them-for example, the Y image, in the description of the compression algorithm below.

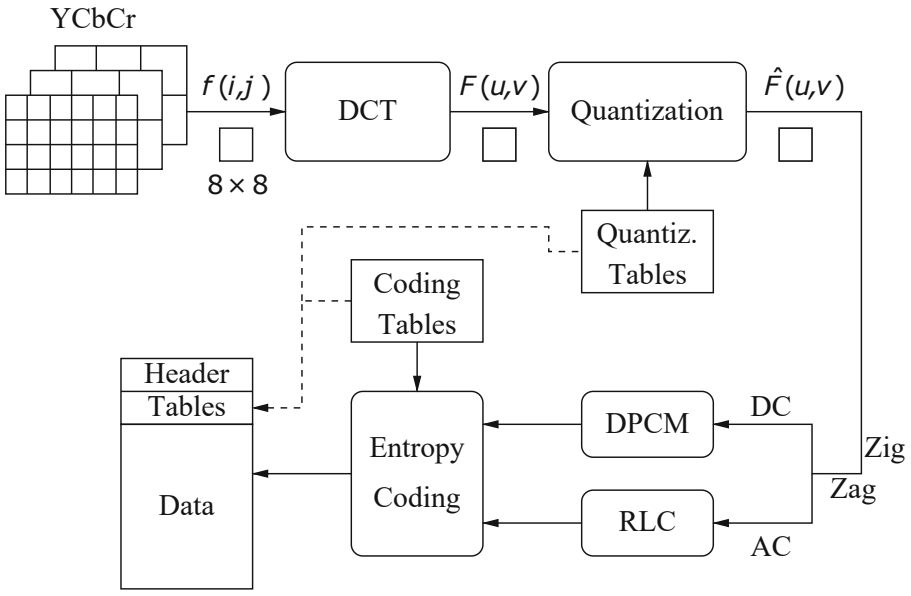


Figure 3: Block diagram for JPEG encoder.

Figure 3 shows a block diagram for a JPEG encoder. If we reverse the arrows in the figure, we basically obtain a JPEG decoder. The JPEG encoder consists of the following main steps:

- Transform RGB to YCbCr and subsample color
- Perform DCT on image blocks
- Apply Quantization
- Perform Zigzag ordering and run-length encoding
- Perform Entropy coding.

6.1.1 DCT on Image Blocks

Each image is divided into 8×8 blocks. The 2D DCT is applied to each block image f (i, j), with output being the DCT coefficients F(u, v) for each block. The choice of a small block size

in JPEG is a compromise reached by the committee: a number larger than 8 would have made accuracy at low frequencies better, but using 8 makes the DCT (and IDCT) computation very fast. Using blocks at all, however, has the effect of isolating each block from its neighboring context. This is why JPEG images look choppy ("blocky") when the user specifies a high compression ratio-we can see these blocks. (And in fact removing such "blocking artifacts" is an important concern of researchers.) To calculate a particular F(u, v), we select the basis image in Fig. 8.9 that corresponds to the appropriate u and v and use it in Eq. 8.17 to derive one of the frequency responses F(u, v).

6.1.2 Quantization

The quantization step in JPEG is aimed at reducing the total number of bits needed for a compressed image. It consists of simply dividing each entry in the frequency

Table 1: The luminance quantization table.

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

Table 2: The chrominance quantization table.

17	18	24	47	99	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	66	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

space block by an integer, then rounding:

$$\hat{F}(u,v) = \text{round}\left(\frac{F(u,v)}{Q(u,v)}\right) \tag{1}$$

Here, $F(u,v)$ represents a DCT coefficient, $Q(u,v)$ is a quantization matrix entry, and $\hat{F}(u,v)$ represents the quantized DCT coefficients JPEG will use in the succeeding entropy coding.

The default values in the 8×8 quantization matrix $Q(u,v)$ are listed in Tables 1 and 2 for luminance and chrominance images, respectively. These numbers resulted from psychophysical studies, with the goal of maximizing the compression ratio while minimizing perceptual losses in JPEG images. The following should be apparent:

- Since the numbers in $Q(u, v)$ are relatively large, the magnitude and variance of $\hat{F}(u, v)$ are significantly smaller than those of $F(u, v)$. We'll see later that $\hat{F}(u, v)$ can be coded with many fewer bits. The quantization step is the main source for loss in JPEG compression.
- The entries of $Q(u, v)$ tend to have larger values toward the lower right corner. This aims to introduce more loss at the higher spatial frequencies-a practice supported by Observations 1 and 2.

We can handily change the compression ratio simply by multiplicatively scaling the numbers in the $Q(u, v)$ matrix. In fact, the quality factor (qf), a user choice offered in every JPEG implementation, can be specified. It is usually in the range of 1..100, where $qf = 100$ corresponds to the highest quality compressed images and $qf = 1$ the lowest quality. The relationship between qf and the scaling_factor is as below:

```
// qf is the user-selected compression quality
// Q is the default Quantization Matrix
// Qx is the scaled Quantization Matrix
// Q1 is a Quantization Matrix which is all 1's
if qf ≥ 50
    scaling_factor ← (100-qf)/50;
else
    scaling_factor ← (50/qf);
end
if scaling_factor ≠ 0 // if qf is not 100
    Qx ← round( Q*scaling_factor );
else
    Qx ← Q1; // no quantization
end
Qx ← uint8(Qx); // max is clamped to 255 for qf=1
```

As an example, when $qf = 50$, scaling_factor will be 1. The resulting Q values will be equal to the table entries. When $qf = 10$, the scaling_factor will be 5. The resulting Q values will be five times the table entry values. For $qf = 100$, the table entries simply become all 1 values, meaning no quantization from this source. Very low quality factors, like $qf = 1$, are a special case: if indeed $qf = 1$ then the scaling_factor will be 50, and the quantization matrix will contain very large numbers. However, because of the type-cast to uint8 in a typical implementation, the table entries go their effective maximum value of 255. Realistically, almost for all applications, qf should not be less than 10.

JPEG also allows custom quantization tables to be specified and put in the header; it is interesting to use low-constant or high-constant values such as $Q = 2$ or $Q = 128$ to observe the basic effects of Q on visual artifacts.

Figures 4 and 5 shows some results of JPEG image coding and decoding on the test image Lena. Only the luminance image (Y) is shown. Also, the lossless coding steps after quantization are not shown, since they do not affect the quality/loss of the JPEG images. These results

show the effect of compression and decompression applied to a relatively smooth block in the image and a more textured (higher frequency-content) block, respectively.

Suppose f (i, j) represents one of the 8×8 blocks extracted from the image, $F(u, v)$ the DCT coefficients, and $\hat{F}(u, v)$ the quantized DCT coefficients. Let $\tilde{F}(u, v)$ denote the dequantized DCT coefficients, determined by simply multiplying by $Q(u, v)$, and let $\tilde{f}(i, j)$ be the reconstructed image block. To illustrate the quality of the JPEG compression, especially the loss, the error $\epsilon(i, j) = f(i, j) - \tilde{f}(i, j)$ is shown in the last row in Figures 4 and 5.

An 8 × 8 block from the Y image of 'Lena'

200	202	189	188	189	175	175	175
200	203	198	188	189	182	178	175
203	200	200	195	200	187	185	175
200	200	200	200	197	187	187	187
200	205	200	200	195	188	187	175
200	200	200	200	200	190	187	175
205	200	199	200	191	187	187	175
210	200	200	200	188	185	187	186

$f(i, j)$

515	65	-12	4	1	2	-8	5
-16	3	2	0	0	-11	-2	3
-12	6	11	-1	3	0	1	-2
-8	3	-4	2	-2	-3	-5	-2
0	-2	7	-5	4	0	-1	-4
0	-3	-1	0	4	1	-1	0
3	-2	-3	3	3	-1	-1	3
-2	5	-2	4	-2	2	-3	0

$F(u, v)$

32	6	-1	0	0	0	0	0
-1	0	0	0	0	0	0	0
-1	0	1	0	0	0	0	0
-1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

$\hat{F}(u, v)$

512	66	-10	0	0	0	0	0
-12	0	0	0	0	0	0	0
-14	0	16	0	0	0	0	0
-14	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

$\tilde{F}(u, v)$

199	196	191	186	182	178	177	176
201	199	196	192	188	183	180	178
203	203	202	200	195	189	183	180
202	203	204	203	198	191	183	179
200	201	202	201	196	189	182	177
200	200	199	197	192	186	181	177
204	202	199	195	190	186	183	181
207	204	200	194	190	187	185	184

$\tilde{f}(i, j)$

1	6	-2	2	7	-3	-2	-1
-1	4	2	-4	1	-1	-2	-3
0	-3	-2	-5	5	-2	2	-5
-2	-3	-4	-3	-1	-4	4	8
0	4	-2	-1	-1	-1	5	-2
0	0	1	3	8	4	6	-2
1	-2	0	5	1	1	4	-6
3	-4	0	6	-2	-2	2	2

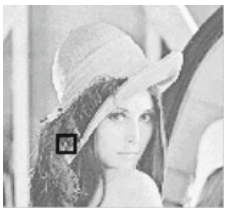
$\epsilon(i, j) = f(i, j) - \tilde{f}(i, j)$

Figure 4: JPEG compression for a smooth image block.

In Figure 4, an image block (indicated by a black box in the image) is chosen at the area where the luminance values change smoothly. Actually, the left side of the block is brighter, and the right side is slightly darker. As expected, except for the DC and the first few AC components, representing low spatial frequencies, most of the DCT coefficients F(u, v) have small magnitudes. This is because the pixel values in this block contain few high-spatial-frequency changes.

An explanation of a small implementation detail is in order. The range of 8-bit luminance values f (i, j) is [0, 255]. In the JPEG implementation, each Y value is first reduced by 128 by simply subtracting 128 before encoding. The idea here is to turn the Y component into a zero-mean image, the same as the chrominance images. As a result, we do not waste any bits

coding the mean value. (Think of an 8×8 block with intensity values ranging from 120 to 135.) Using f (i, j)-128 in place of f (i, j) will not affect the output of the AC coefficients-it alters only the DC coefficient. After decoding, the subtracted 128 will be added back onto the Y values.



Another 8×8 block from the Y image of 'Lena'

70	70	100	70	87	87	150	187
85	100	96	79	87	154	87	113
100	85	116	79	70	87	86	196
136	69	87	200	79	71	117	96
161	70	87	200	103	71	96	113
161	123	147	133	113	113	85	161
146	147	175	100	103	103	163	187
156	146	189	70	113	161	163	197

$f(i,j)$

-80	-40	89	-73	44	32	53	-3
-135	-59	-26	6	14	-3	-13	-28
47	-76	66	-3	-108	-78	33	59
-2	10	-18	0	33	11	-21	1
-1	-9	-22	8	32	65	-36	-1
5	-20	28	-46	3	24	-30	24
6	-20	37	-28	12	-35	33	17
-5	-23	33	-30	17	-5	-4	20

$F(u,v)$

-5	-4	9	-5	2	1	1	0
-11	-5	-2	0	1	0	0	-1
3	-6	4	0	-3	-1	0	1
0	1	-1	0	1	0	0	0
0	0	-1	0	0	1	0	0
0	-1	1	-1	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

$\hat{F}(u,v)$

-80	-44	90	-80	48	40	51	0
-132	-60	-28	0	26	0	0	-55
42	-78	64	0	-120	-57	0	56
0	17	-22	0	51	0	0	0
0	0	-37	0	0	109	0	0
0	-35	55	-64	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

$\tilde{F}(u,v)$

70	60	106	94	62	103	146	176
85	101	85	75	102	127	93	144
98	99	92	102	74	98	89	167
132	53	111	180	55	70	106	145
173	57	114	207	111	89	84	90
164	123	131	135	133	92	85	162
141	159	169	73	106	101	149	224
150	141	195	79	107	147	210	153

$\tilde{f}(i,j)$

0	10	-6	-24	25	-16	4	11
0	-1	11	4	-15	27	-6	-31
2	-14	24	-23	-4	-11	-3	29
4	16	-24	20	24	1	11	-49
-12	13	-27	-7	-8	-18	12	23
-3	0	16	-2	-20	21	0	-1
5	-12	6	27	-3	2	14	-37
6	5	-6	-9	6	14	-47	44

$\epsilon(i,j) = f(i,j) - \tilde{f}(i,j)$

Figure 5: JPEG compression for a textured image block.

In Figure 5, the image block chosen has rapidly changing luminance. Hence, many more AC components have large magnitudes (including those toward the lower right corner, where u and v are large). Notice that the error $\epsilon(i, j)$ is also larger now than in Figure 4 -JPEG does introduce more loss if the image has quickly changing details.

6.1.3 Preparation for Entropy Coding

We have so far seen two of the main steps in JPEG compression: DCT and quantization. The remaining small steps shown in the block diagram in Figure 3 all lead up to entropy coding of the quantized DCT coefficients. These additional data compression steps are lossless. Interestingly, the DC and AC coefficients are treated quite differently before entropy coding: run-length encoding on ACs versus DPCM on DCs.

6.1.4 Run-Length Coding on AC Coefficients

Notice in Figure 4 the many zeros in $\hat{F}(u,v)$ after quantization is applied. Runlength Coding (RLC) (or Run-length Encoding, RLE) is therefore useful in turning the $\hat{F}(u,v)$ values into sets amount-of-zeros-to-skip, next nonzero value. RLC is even more effective when we use an addressing scheme, making it most likely to hit a long run of zeros: a zigzag scan turns the 8×8 matrix $\hat{F}(u,v)$ into a 64-vector, as Figure 6 illustrates. After all, most image blocks tend to have small high-spatial-frequency components, which are zeroed out by quantization. Hence the zigzag scan order has a good chance of concatenating long runs of zeros. For example, $\hat{F}(u,v)$ in Figure 4 will be turned into

$(32, 6, -1, -1, 0, -1, 0, 0, 0, -1, 0, 0, 1, 0, 0, ..., 0)$

with three runs of zeros in the middle and a run of 51 zeros at the end.

The RLC step replaces values by a pair (RUNLENGTH, VALUE) for each run of zeros in the AC coefficients of $\hat{F}$, where RUNLENGTH is the number of zeros in the run and VALUE is the next nonzero coefficient. To further save bits, a special pair (0,0) indicates the end-of-block after the last nonzero AC coefficient is reached. In the above example, not considering the first (DC) component, we will thus have

$(0,6)(0,-1)(0,-1)(1,-1)(3,-1)(2,1)(0,0)$

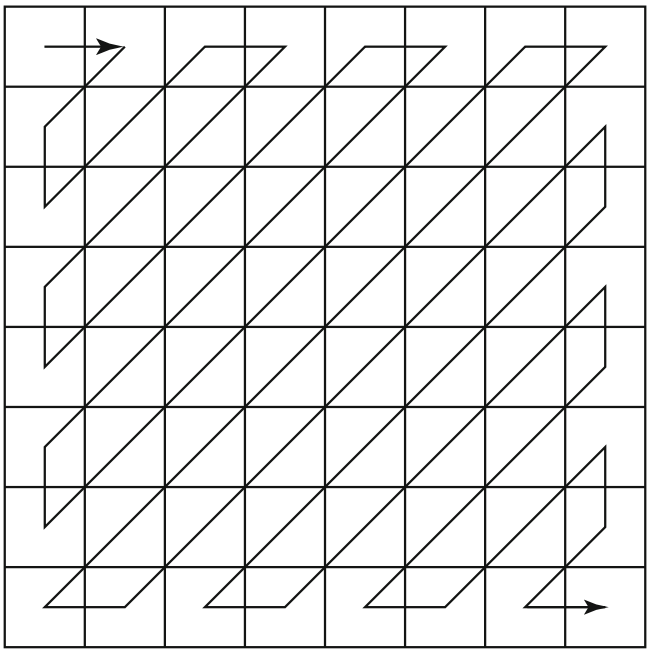


Figure 6: Zigzag scan in JPEG.

6.1.5 Differential Pulse Code Modulation on DC Coefficients

The DC coefficients are coded separately from the AC ones. Each 8×8 image block has only one DC coefficient. The values of the DC coefficients for various blocks could be large and different, because the DC value reflects the average intensity of each block, but consistent

with Observation 1 above, the DC coefficient is unlikely to change drastically within a short distance. This makes DPCM an ideal scheme for coding the DC coefficients.

If the DC coefficients for the first five image blocks are 150, 155, 149, 152, 144, DPCM would produce 150, 5, -6, 3, -8, assuming the predictor for the ith block is simply $d_i = DC_i - DC_{i-1}$, and $d_0 = DC_0$. We expect DPCM codes to generally have smaller magnitude and variance, which is beneficial for the next entropy coding step.

It is worth noting that unlike the run-length coding of the AC coefficients, which is performed on each individual block, DPCM for the DC coefficients in JPEG is carried out on the entire image at once.

6.1.6 Entropy Coding

The DC and AC coefficients finally undergo an entropy coding step. Below, we will discuss only the basic (or baseline1) entropy coding method, which uses Huffman coding and supports only 8-bit pixels in the original images (or color image components).

Let us examine the two entropy coding schemes, using a variant of Huffman coding for DCs and a slightly different scheme for ACs.

Table 3: Baseline entropy coding details-size category.

Size	**Amplitude**
1	-1, 1
2	-3, -2, 2, 3
3	-7, .., -4, 4, .., 7
4	-15, .., -8, 8, .., 15
...	...
10	-1023, .., -512, 512, .., 1023

6.1.7 Huffman Coding of DC Coefficients

Each DPCM-coded DC coefficient is represented by a pair of symbols (SIZE, AMPLITUDE), where SIZE indicates how many bits are needed for representing the coefficient and AMPLITUDE contains the actual bits. Table 3 illustrates the size category for the different possible amplitudes. Notice that DPCM values could require more than 8 bits and could be negative values. The one's-complement scheme is used for negative numbers-that is, binary code 10 for 2, 01 for -2; 11 for 3, 00 for -3; and so on. In the example we are using, codes 150, 5, -6, 3, -8 will be turned into

(8, 10010110), (3, 101), (3, 001), (2, 11), (4, 0111)

In the JPEG implementation, SIZE is Huffman coded and is hence a variable length code. In other words, SIZE 2 might be represented as a single bit (0 or 1) if it appeared most frequently. In general, smaller SIZEs occur much more often-the entropy of SIZE is low. Hence, deployment of Huffman coding brings additional compression. After encoding, a

custom Huffman table can be stored in the JPEG image header; otherwise, a default Huffman table is used. On the other hand, AMPLITUDE is not Huffman coded. Since its value can change widely, Huffman coding has no appreciable benefit.

6.1.8 Huffman Coding of AC Coefficients

Recall we said that the AC coefficients are run-length coded and are represented by pairs of numbers (RUNLENGTH, VALUE). However, in an actual JPEG implementation, VALUE is further represented by SIZE and AMPLITUDE, as for the DCs. To save bits, RUNLENGTH and SIZE are allocated only 4 bits each and squeezed into a single byte-let us call this Symbol 1. Symbol 2 is the AMPLITUDE value; its number of bits is indicated by SIZE:

Symbol 1: (RUNLENGTH, SIZE)

Symbol 2: (AMPLITUDE)

The 4-bit RUNLENGTH can represent only zero-runs of length 0 to 15. Occasionally, the zero run-length exceeds 15; then a special extension code, (15, 0), is used for Symbol 1. In the worst case, three consecutive (15, 0) extensions are needed before a normal terminating Symbol 1, whose RUNLENGTH will then complete the actual runlength. As in DC, Symbol 1 is Huffman coded, whereas Symbol 2 is not.