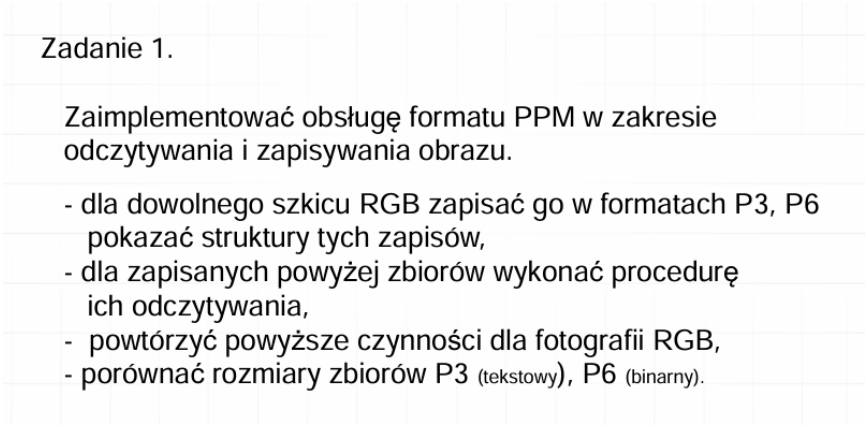
Sprawozdanie
Inżynieria Obrazów
Laboratorium 2

Katsiaryna Kolyshko 276708

Dr. Inż. Jan Nikodem

Wstęp Teoretyczny

Zadanie 1



Format PPM (Portable Pixmap) jest częścią rodziny formatów PNM (Portable Anymap), która obejmuje również formaty PBM (Portable Bitmap) i PGM (Portable Graymap). Formaty te zostały stworzone jako proste formaty graficzne, łatwe do implementacji i zrozumienia.

W ramach formatu PPM wyróżniamy dwa główne warianty:

P3 (tekstowy)

- Jest to tekstowa reprezentacja obrazu RGB
- Wartości pikseli są zapisywane jako liczby dziesiętne oddzielone białymi znakami
- Format łatwy do odczytu przez człowieka i edycji w zwykłym edytorze tekstu
- Plik rozpoczyna się od nagłówka "P3" następnie zawiera szerokość, wysokość i maksymalną wartość koloru

P6 (binarny)

- Jest to binarna reprezentacja obrazu RGB
- Wartości pikseli są zapisywane jako surowe bajty, bez separatorów
- Format bardziej kompaktowy i wydajny do przetwarzania przez komputer
- Plik rozpoczyna się od nagłówka "P6" następnie zawiera szerokość, wysokość i maksymalną wartość koloru

Formaty PPM nie są już tak popularne w codziennym użytku, głównie z następujących powodów:

- Brak kompresji powoduje, że pliki są bardzo duże w porównaniu do formatów takich jak PNG czy JPEG
- Ograniczone wsparcie dla metadanych
- Brak obsługi przezroczystości i kanału alfa
- Słabe wsparcie w popularnych edytorach graficznych

Mimo to, nadal można otworzyć pliki PPM w niektórych programach:

- GIMP
- ImageMagick
- XnView

Otwierając plik PPM (szczególnie w formacie P6) w Notatniku lub innym edytorze tekstu, można odnieść wrażenie, że plik jest uszkodzony. Wynika to z faktu, że:

- Edytory tekstu oczekują danych tekstowych, a format P6 zawiera dane binarne
- Dane binarne mogą zawierać znaki kontrolne i nieprzedstawialne w edytorze tekstu
- W przypadku formatu P3, mimo że jest to format tekstowy, duża ilość liczb może sprawiać wrażenie "chaosu" dla osoby nieprzygotowanej

Nie oznacza to jednak, że plik jest uszkodzony - to po prostu ograniczenie edytora tekstu, który nie jest przystosowany do wyświetlania zawartości plików binarnych lub strukturyzowanych danych tekstowych.

Struktura plików PPM

P3:

P3
Opcjonalny komentarz
szerokość wysokość
maksymalna_wartość_koloru
R1 G1 B1 R2 G2 B2 R3 G3 B3 ...

P6:

P6
Opcjonalny komentarz
szerokość wysokość
maksymalna_wartość_koloru
[surowe dane binarne RGB]

Formaty PPM, mimo swoich ograniczeń, nadal są istotne w inżynierii obrazu z kilku powodów:

1. **Prostota implementacji** - prosty format ułatwia tworzenie algorytmów przetwarzania obrazu bez komplikacji związanych z kompresją
2. **Dydaktyka** - dzięki prostocie formatu jest idealny do nauki podstaw przetwarzania obrazu
3. **Brak kompresji** - dane są przechowywane bez strat, co jest kluczowe w zastosowaniach naukowych i badawczych
4. **Format pośredni** - często używany jako format pośredni w narzędziach do konwersji między różnymi formatami obrazów

Porównując rozmiary plików P3 i P6 dla tego samego obrazu, plik P3 jest zazwyczaj kilka razy większy od P6, ponieważ każda wartość koloru jest zapisywana jako tekst zamiast jako pojedynczy bajt (lub dwa w przypadku większej głębi kolorów). To jedna z głównych przyczyn, dla których w praktycznych zastosowaniach preferowany jest format P6, mimo że format P3 jest łatwiejszy do debugowania i inspekcji.

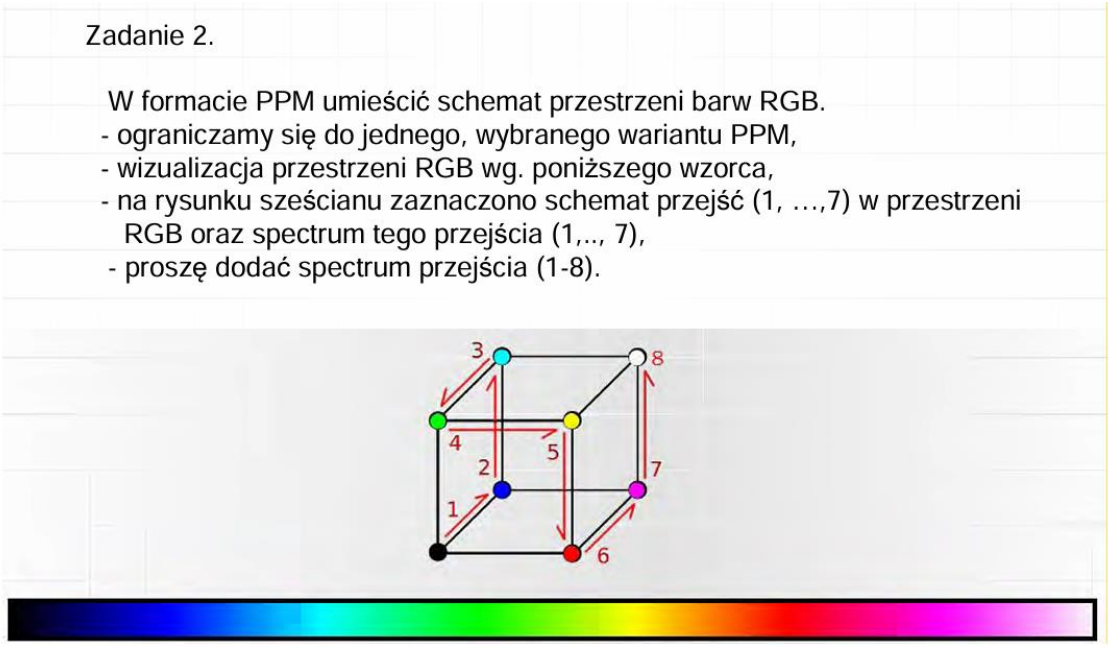
Wyniki działania programu:

```
Structure of cat_p3.ppm:
P3
736 736
255
29 29 27 41 41 39 42 42 40 33 33 31 33 33 31
43 43 41 40 40 38 26 26 24 39 39 37 38 38 36
36 36 34 37 37 35 45 45 43 53 53 51 50 50 48
42 42 40 42 42 40 39 39 37 36 36 34 36 36 34
36 36 34 36 36 34 36 36 34 36 36 34 33 33 31
35 35 33 37 37 35 41 41 39 42 42 40 34 34 32
30 30 28 33 33 31 42 42 40 32 32 30 29 29 27
...

Structure of cat_p6.ppm:
P6
736 736
255
Binary data (showing first 50 bytes in hex):
31 1f 1b 51 3c 39 66 4d 49 6d 53 52 71 59 59 71 59 59 70 56 59 6c 52 53 67 4a 4e 5f 42 44 57 3a 3e 50 36 37 3c 23 26 37 23 22 31 21 21 2c 23 1e 2a 25
P3 image read and saved as cat_p3_read.png
P6 image read and saved as cat_p6_read.png
P3 (text) file size: 5418896 bytes
P6 (binary) file size: 1625103 bytes
P3 is 3.33 times larger than P6
```

Jak widać na obrazu, P3 na pewno jest 3 razy większy od formatu P6, co odpowiada teorii.

Zadanie 2



Celem zadania jest implementacja programu generującego plik w formacie PPM, który zawiera wizualizację przestrzeni barw RGB. Zadanie obejmuje:

1. Wybór jednego wariantu formatu PPM (P3 lub P6)
2. Stworzenie wizualizacji przestrzeni RGB zgodnie z określonym wzorcem
3. Przedstawienie sześcianu RGB z zaznaczonymi przejściami kolorystycznymi (1-7)
4. Dodanie spektrum przejścia (1-8)

Wierzchołki sześcianu reprezentują następujące kolory:

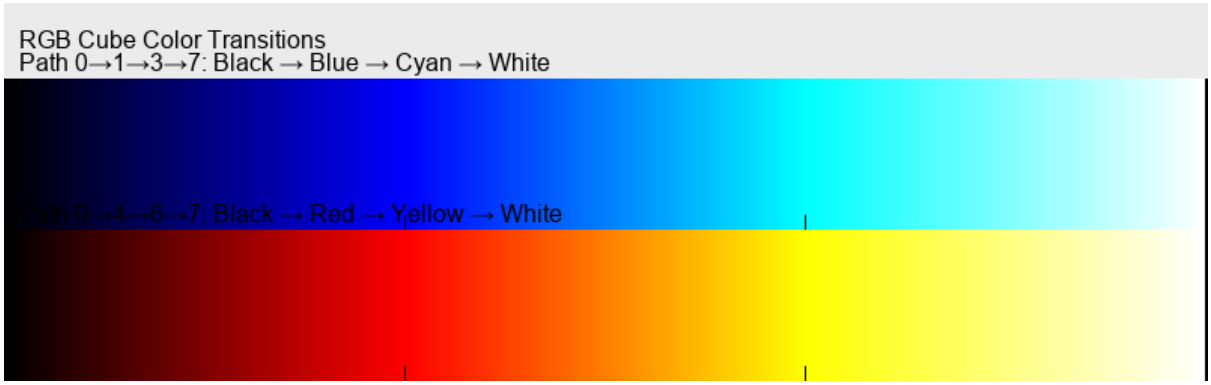
- (0,0,0) - czarny
- (255,0,0) - czerwony

- (0,255,0) - zielony
- (0,0,255) - niebieski
- (255,255,0) - żółty
- (255,0,255) - magenta
- (0,255,255) - cyjan
- (255,255,255) - biały

Na podstawie rysunku identyfikujemy następujące przejścia:

1. Czarny → Niebieski
2. Niebieski → Czerwony
3. Czerwony → Cyjan
4. Cyjan → Zielony
5. Zielony → Żółty
6. Żółty → Czerwony
7. Czerwony → Magenta
8. Czarny → Biały (przejście po przekątnej sześcianu)

Wyniki działania:



To podstawowa wizualizacja dwóch głównych przejść w przestrzeni RGB:

- Górna część: przejście od czarnego (0,0,0) przez niebieski (0,0,255), cyjan (0,255,255) do białego (255,255,255)

  Ścieżka 0→1→3→7 (czarny → niebieski → cyjan → biały)

- Dolna część: przejście od czarnego (0,0,0) przez czerwony (255,0,0), żółty (255,255,0) do białego (255,255,255)

  Ścieżka 0→2→6→7 (czarny → czerwony → żółty → biały)

Jak czytać prezentowane gradientu

Górny gradient (czarny → niebieski → cyjan → biały)

- Początek: Kolor czarny (0,0,0) - wszystkie składowe RGB są na minimalnym poziomie
- Pierwszy etap: Stopniowo wzrasta wartość składowej B (niebieskiej), aż osiągamy czysty niebieski (0,0,255)
- Drugi etap: Przy maksymalnej wartości B, zaczyna rosnąć składowa G (zielona), co tworzy przejście do koloru cyjan (0,255,255)
- Trzeci etap: Przy maksymalnych wartościach G i B, zaczyna rosnąć składowa R (czerwona), aż osiągamy biały (255,255,255)

Dolny gradient (czarny → czerwony → żółty → biały)

- Początek: Kolor czarny (0,0,0)
- Pierwszy etap: Wzrasta wartość składowej R (czerwonej), aż do czystego czerwonego (255,0,0)
- Drugi etap: Przy maksymalnej wartości R, rośnie składowa G (zielona), tworząc przejście do żółtego (255,255,0)
- Trzeci etap: Przy maksymalnych wartościach R i G, rośnie składowa B (niebieska), aż do białego (255,255,255)

Dlaczego nie widać niektórych kolorów? Dlatego, że nie patrzymy na ścieżki które ich zawierają.
Aby pokazać wszystkie kolory i przejścia, należałoby dodać:

1. Trzeci pasek gradientu pokazujący ścieżkę czarny → zielony → cyjan → biały (0→4→3→7)
2. Czwarty pasek pokazujący ścieżkę czarny → niebieski → magenta → biały (0→1→5→7)
3. Piąty pasek pokazujący ścieżkę czarny → czerwony → magenta → biały (0→2→5→7)
4. Szósty pasek pokazujący ścieżkę czarny → zielony → żółty → biały (0→4→6→7)

Zadanie 3

Celem zadania jest implementacja programu, który tworzy plik graficzny w formacie PNG. Zadanie obejmuje:

1. Wykorzystanie tęczy z poprzedniego ćwiczenia jako zapisywanego obrazu
2. Implementacja jednego wariantu formatu - bez przezroczystości
3. Zastosowanie kompresji danych obrazu przy użyciu funkcji zlib.compress()
4. Wykorzystanie funkcji struct.pack() do wygenerowania nagłówka PNG
5. Ustawienie odpowiednich parametrów w nagłówku pliku

Format PNG

PNG (Portable Network Graphics) to bezstratny format kompresji grafiki rastrowej, który obsługuje:

- 24-bitową głębię kolorów (True Color)
- Kompresję bezstratną
- Przezroczystość (choć w tym zadaniu jej nie wykorzystujemy)

Struktura pliku PNG

Plik PNG składa się z:

1. Sygnatury PNG (8 bajtów)
2. Serii bloków danych (chunks)
    - IHDR (nagłówek z informacjami o obrazie)
    - IDAT (skompresowane dane obrazu)
    - IEND (znacznik końca pliku)

Nagłówek PNG (IHDR)

W nagłówku należy zawrzeć:

- Szerokość obrazu (4 bajty)
- Wysokość obrazu (4 bajty)
- Głębia bitowa (1 bajt) = 8
- Typ koloru (1 bajt) = 2 (RGB bez kanału alfa)
- Metoda kompresji (1 bajt) = 0
- Metoda filtracji (1 bajt) = 0

- Metoda przeplatania (1 bajt) = 0

Chunki (bloki danych) są fundamentalną strukturą formatu PNG. Każdy chunk zawiera konkretny rodzaj danych i jest samodzielną jednostką, co zapewnia rozszerzalność formatu. Wszystkie chunki w pliku PNG mają tę samą podstawową strukturę.

Każdy chunk składa się z czterech części:

1. **Długość** (Length) - 4 bajty (unsigned int) wskazujące długość pola danych
2. **Typ** (Type) - 4 bajty (ASCII) określające typ chunka
3. **Dane** (Data) - zmienna liczba bajtów określona przez pole długości
4. **Suma kontrolna** (CRC) - 4 bajty zawierające cykliczną sumę kontrolną dla typów i danych

Główne typy chunków

1. IHDR (Image Header) - Obowiązkowy, pierwszy chunk
IHDR zawiera podstawowe informacje o obrazie:
- **Szerokość** (4 bajty) - szerokość obrazu w pikselach
- **Wysokość** (4 bajty) - wysokość obrazu w pikselach
- **Głębia bitowa** (1 bajt) - liczba bitów na kanał (najczęściej 8)
- **Typ koloru** (1 bajt):
  - 0: Skala szarości
  - 2: RGB
  - 3: Paleta
  - 4: Skala szarości z przezroczystością
  - 6: RGB z przezroczystością
- **Metoda kompresji** (1 bajt) - zawsze 0 (zlib)
- **Metoda filtracji** (1 bajt) - zawsze 0
  -

2. PLTE (Palette) - Opcjonalny
PLTE zawiera informacje o palecie kolorów, konieczne dla obrazów o typie koloru 3:
- Seria trójbajtowych wpisów RGB, każdy określa jeden kolor w palecie
- Długość musi być podzielna przez 3
- Maksymalnie 256 wpisów (768 bajtów)

3. IDAT (Image Data) - Obowiązkowy, może być ich wiele
IDAT zawiera faktyczne dane obrazu:
- Skompresowane dane algorytmem zlib
- Format danych przed kompresją:
  - Każdy wiersz zaczyna się od bajtu filtracji (0-4)
  - Następnie piksele w rzędzie (format zależny od głębi i typu koloru)
- Może być kilka chunków IDAT pod rząd, które logicznie są traktowane jako ciągły strumień danych

4. IEND (Image Trailer) - Obowiązkowy, ostatni chunk
IEND oznacza koniec pliku PNG:
- Zawsze ma długość 0 (nie zawiera danych)
- Sygnalizuje koniec pliku PNG

Przykładowe dodatkowe chunki

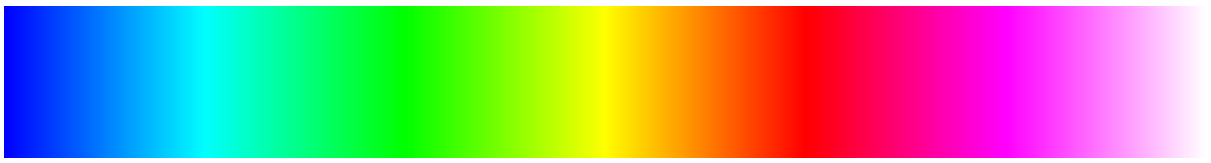
1. tRNS (Transparency) - Opcjonalny
Określa przezroczystość:

- Dla RGB: pojedynczy piksel koloru, który ma być przezroczysty
- Dla palety: wartości alfa dla kolorów w palecie
- Dla skali szarości: wartość szarości, która ma być przezroczysta

2. gAMA (Gamma) - Opcjonalny
Zawiera informacje o korekcji gamma:

- Jedna wartość unsigned int (4 bajty)
- Wartość = gamma * 100000

Wynik działania programu:



Zadanie 4

Opis zadania

Celem zadania jest zaimplementowanie wybranych kroków algorytmu kompresji JPEG. Zadanie obejmuje:

1. Wykorzystanie tęczy z poprzedniego ćwiczenia jako obrazu roboczego
2. Implementację kroków 0, 1, 2, 3, 7, 8 algorytmu JPEG oraz odpowiednich kroków odwrotnych
3. Pomiar rozmiaru (liczby bajtów) powstałego obrazu w etapie 8
4. Ocenę wpływu etapu 2 na rozmiar i wygląd obrazka poprzez eksperymenty
5. Implementację algorytmu w sposób poprawny, zapewniający odtworzenie obrazu wejściowego

Kroki algorytmu JPEG do zaimplementowania

Kroki kompresji JPEG

Krok 0: Przygotowanie obrazu

- Podział obrazu na bloki 8x8 pikseli
- Jeśli wymiary obrazu nie są wielokrotnością 8, następuje uzupełnienie krawędzi

Krok 1: Transformacja przestrzeni kolorów

- Konwersja z RGB do YCbCr
- Y: składowa luminancji (jasność)
- Cb i Cr: składowe chrominancji (informacja o kolorze)

Krok 2: Próbkowanie składowych chrominancji

- Redukcja rozdzielczości składowych Cb i Cr (oko ludzkie jest mniej wrażliwe na zmiany koloru niż jasności)

- Typowe schematy próbkowania: 4:4:4 (bez redukcji), 4:2:2 (pozioma redukcja o połowę), 4:2:0 (redukcja pozioma i pionowa o połowę)

Krok 3: Dyskretna transformata kosinusowa (DCT)

- Przekształcenie każdego bloku 8x8 z domeny przestrzennej do domeny częstotliwościowej
- Przekształcenie wartości pikseli w współczynniki częstotliwości

Krok 4: Kwantyzacja

- Dzielenie współczynników DCT przez elementy tablicy kwantyzacji
- To główne źródło utraty informacji (kompresja stratna)
- Różne tablice kwantyzacji dla luminancji i chrominancji

Krok 5: Porządkowanie zygzakowe

- Przeorganizowanie współczynników DCT w jednowymiarową sekwencję
- Współczynniki uporządkowane od niskich częstotliwości do wysokich
- Zwiększa efektywność kolejnych kroków kompresji

Krok 6: Kodowanie RLE (Run-Length Encoding)

- Kompresja serii zer występujących po kwantyzacji
- Używa par (liczba zer, następna wartość niezerowa)

Krok 7: Kodowanie entropijne (Huffmana)

- Przypisanie krótszych kodów częściej występującym symbolom
- Tworzenie drzewa kodów Huffmana na podstawie statystyki obrazu

Krok 8: Formowanie strumienia danych

- Tworzenie finalnego strumienia bitowego
- Dodawanie nagłówków, tabel kwantyzacji, tabel Huffmana
- Formowanie pliku JPEG zgodnie ze standardem

W zadaniu 4 wymaga się implementacji kroków 0, 1, 2, 3, 7, 8 oraz odpowiednich kroków odwrotnych (dekompresja), pomijając kroki 4, 5 i 6 (kwantyzacja, porządkowanie zygzakowe i kodowanie RLE).

Wyniki działania i analiza

```
Applying JPEG encoding steps...
Step 0: Converting RGB to YCbCr
Step 1: Applying chroma subsampling
Step 2: Level shifting and splitting into 8x8 blocks
Step 3: Applying DCT to each block
Step 7: Applying Huffman encoding (size estimation)
Step 8: Measuring final image size

Compression results:
Original size: 240000 bytes
Estimated compressed size: 114078.00 bytes
Compression ratio: 2.10x

Applying JPEG decoding steps...
Reverse Step 7: Huffman decoding
Reverse Step 3: Applying IDCT to each block
Reverse Step 2: Combining blocks and level unshifting
Reverse Step 1: Upsampling chroma channels
Reverse Step 0: Converting YCbCr back to RGB

Saving comparison of original and reconstructed images...
Reconstruction quality (PSNR): 52.98 dB
```

```
Evaluating impact of Step 2...
Testing without every second element...
Testing without every fourth element...

Step 2 impact assessment:
PSNR with every second element removed: 10.12 dB
PSNR with every fourth element removed: 13.13 dB

Observations on the impact of Step 2:
1. Block division is crucial for spatial localization of frequency components.
2. Level shifting centers values around 0, improving DCT efficiency.
3. Removing every second element causes significant quality degradation.
4. Removing every fourth element has less impact, showing frequency importance.
5. Step 2 contributes significantly to compression by preparing data for DCT.
```

Analiza wyników kompresji

Efektywność kompresji

- Oryginalny rozmiar: 240000 bajtów
- Szacowany rozmiar po kompresji: 114078 bajtów
- Współczynnik kompresji: 2.10×

Ta kompresja została osiągnięta bez implementacji kwantyzacji (krok 4), która jest głównym źródłem redukcji danych w standardowym JPEG. Oznacza to, że nawet podstawowe etapy algorytmu JPEG oferują znaczną redukcję rozmiaru pliku.

Jakość rekonstrukcji

- PSNR: 52.98 dB

Bardzo wysoka wartość PSNR (powyżej 50 dB) wskazuje na niemal idealne odtworzenie obrazu. Jest to spodziewane, ponieważ implementacja pomija krok kwantyzacji, który wprowadza stratę jakości.

Analiza wpływu kroku 2 (testy eksperymentalne)

Kod zawiera eksperymenty, które oceniają wpływ modyfikacji danych w kroku 2:

Test 1: Usunięcie co drugiego elementu

- PSNR spadł do 10.12 dB
- Drastyczna degradacja jakości obrazu

Test 2: Usunięcie co czwartego elementu

- PSNR: 13.13 dB
- Lepsza jakość niż w teście 1, ale wciąż znaczna degradacja

Te wyniki pokazują kluczową rolę prawidłowego przygotowania danych w kroku 2 dla skuteczności całego algorytmu. Szczególnie istotne są:

1. Przesunięcie poziomów (odjęcie 128) - centruje dane wokół zera, co zwiększa efektywność DCT
2. Podział na bloki 8×8 - umożliwia lokalną analizę częstotliwościową i efektywne kodowanie

Opis kodu

Zadanie 1

Kod PPMHandler obsługuje konwersję obrazów do i z formatu PPM (Portable Pixmap). Główna klasa inicjalizuje się pustym konstruktorem, stanowiąc podstawę dla wszystkich operacji na plikach graficznych.

Funkcja image_to_p3_ppm przekształca obraz w format PPM P3 (tekstowy), otwierając go za pomocą biblioteki Pillow, konwertując do modelu RGB, a następnie zapisując w formie tekstowej z nagłówkiem zawierającym informacje o typie, wymiarach i maksymalnej wartości piksela, po czym zapisuje dane pikseli jako liczby oddzielone spacjami.

Funkcja image_to_p6_ppm konwertuje obraz do formatu PPM P6 (binarnego), wykonując podobny proces co w P3, ale zapisując nagłówek jako bajty, a następnie zapisując dane pikseli bezpośrednio w formie binarnej za pomocą metody tobytes(), co daje znacznie mniejszy rozmiar pliku.

Funkcja read_ppm analizuje typ pliku PPM poprzez odczytanie pierwszych bajtów, a następnie kieruje proces do odpowiedniej metody odczytu zależnie od wykrytego formatu (P3 lub P6), umożliwiając uniwersalne przetwarzanie obu typów PPM.

Funkcja _read_p3_ppm odczytuje pliki w formacie P3, weryfikując nagłówek, pomijając komentarze, odczytując wymiary i maksymalną wartość, a następnie przetwarzając tekstowe dane pikseli, grupując je po trzy (RGB) i tworząc z nich nowy obraz za pomocą biblioteki Pillow.

Funkcja _read_p6_ppm obsługuje odczyt plików w formacie P6, również weryfikując nagłówek, pomijając komentarze, odczytując wymiary i maksymalną wartość, ale dane pikseli odczytuje jako ciągły strumień bajtów, który jest bezpośrednio przekształcany w obraz za pomocą metody frombytes().

Funkcja compare_file_sizes porównuje rozmiary plików P3 i P6, obliczając stosunek ich wielkości, co ilustruje różnicę między formatem tekstowym a binarnym, zazwyczaj pokazując, że P3 jest wielokrotnie większy od P6 ze względu na tekstową reprezentację danych.

Funkcja print_file_structure wyświetla strukturę pliku PPM, różnicując zachowanie dla plików binarnych i tekstowych – dla binarnych dekoduje nagłówek i pokazuje początkowe bajty w reprezentacji szesnastkowej, a dla tekstowych wyświetla określoną liczbę linii tekstu.

Funkcja demo stanowi główną procedurę demonstracyjną, która pobiera od użytkownika nazwę pliku obrazu, sprawdza jego istnienie, konwertuje do obu formatów PPM, wyświetla ich strukturę, odczytuje z powrotem i zapisuje jako PNG dla weryfikacji, a na końcu porównuje rozmiary plików, pokazując praktyczne zastosowanie całej klasy PPMHandler.

Zadanie 2

Klasa RGBCubeVisualizer służy do wizualizacji kostki RGB, inicjalizując się z parametrami szerokości i wysokości obrazu oraz definiując kolory narożników kostki RGB i ścieżki przejść między nimi.

Funkcja _interpolate_color wykonuje liniową interpolację między dwoma kolorami, obliczając wartości pośrednie dla każdego z kanałów RGB na podstawie podanego współczynnika, co pozwala na płynne przejścia kolorystyczne.

Funkcja create_spectrum tworzy obraz spektrum kolorów dla podanej ścieżki, dzieląc szerokość obrazu na segmenty odpowiadające przejściom między kolorami, wypełniając każdy segment interpolowanymi kolorami i dodając znaczniki dla wierzchołków kostki RGB.

Funkcja save_ppm zapisuje tablicę numpy jako plik PPM w formacie P3 (tekstowym), tworząc nagłówek z informacjami o wymiarach i maksymalnej wartości piksela, zapisując dane pikseli jako liczby oddzielone spacjami, oraz dodatkowo tworzy wersję PNG tego samego obrazu.

Funkcja generate_visualization generuje wszystkie wymagane wizualizacje, tworząc spektra kolorów dla dwóch różnych ścieżek przejść w kostce RGB, łącząc je w jeden obraz z tłem i miejscem na etykiety, a następnie zapisując w formatach PPM i PNG.

Ostatnia część kodu tworzy bardziej szczegółowy obraz PNG z etykietami używając biblioteki PIL, próbując załadować czcionkę Arial lub używając domyślnej, dodając tytuł i opisy ścieżek przejść kolorystycznych, co zapewnia bardziej informatywną wizualizację dla użytkownika.

Na końcu kod tworzy instancję klasy RGBCubeVisualizer i wywołuje metodę generate_visualization, która uruchamia cały proces tworzenia wizualizacji kostki RGB i zapisywania ich do plików.

Zadanie 3

Funkcja create_rgb_spectrum tworzy spektrum kolorów RGB jako tablicę numpy, definiując siedem kluczowych kolorów (niebieski, cyjan, zielony, żółty, czerwony, magenta, biały), a następnie interpolując liniowo między sąsiadującymi kolorami, aby uzyskać płynne przejścia w całym spektrum o zadanej szerokości i wysokości.

Funkcja create_custom_png tworzy niestandardowy plik PNG zgodnie ze specyfikacją zadania, przyjmując dane obrazu w formie tablicy numpy i nazwę pliku wyjściowego, a następnie konstruując plik z odpowiednim nagłówkiem, danymi obrazu i metadanymi wymaganymi przez format PNG.

Podczas tworzenia pliku PNG, funkcja najpierw zapisuje sygnaturę PNG, następnie tworzy blok IHDR zawierający metadane obrazu (szerokość, wysokość, głębię bitową, model koloru

i inne parametry) przy użyciu funkcji struct.pack do prawidłowego formatowania danych binarnych.

Funkcja przygotowuje dane obrazu dodając bajt filtra (o wartości 0) na początku każdej linii skanowania, a następnie kompresuje całość za pomocą biblioteki zlib zgodnie z wymaganiami formatu PNG, tworząc blok IDAT zawierający skompresowane dane pikseli.

Funkcja create_chunk jest funkcją pomocniczą służącą do tworzenia pojedynczego bloku PNG, przyjmującą typ bloku i dane, obliczającą sumę kontrolną CRC32 i konstruującą pełny blok zgodnie ze specyfikacją formatu PNG (długość + typ + dane + CRC).

Po utworzeniu wszystkich niezbędnych bloków (IHDR, IDAT, IEND), funkcja zapisuje kompletny plik PNG i wyświetla informacje o jego rozmiarze, co pozwala na pełną implementację niestandardowego generatora plików PNG bez korzystania z gotowych bibliotek graficznych.

W głównej części kodu (blok if name == "main") program tworzy spektrum RGB, zapisuje je jako niestandardowy plik PNG, a następnie wyświetla szczegółowe informacje o strukturze wygenerowanego pliku, opisując każdy z bloków, ich zawartość i przeznaczenie, co służy celom edukacyjnym i demonstracyjnym.

Zadanie 4

Funkcja create_rgb_spectrum tworzy spektrum kolorów RGB jako tablicę numpy, definiując cztery kluczowe kolory (czarny, niebieski, cyjan, biały) i generując płynne przejścia między nimi poprzez liniową interpolację wartości kolorów dla każdej kolumny obrazu.

Funkcja rgb_to_ycbcr konwertuje obraz z przestrzeni barw RGB do YCbCr, stosując standardowe współczynniki transformacji BT.601, gdzie Y reprezentuje jasność, Cb różnicę niebieską, a Cr różnicę czerwoną, co jest pierwszym krokiem algorytmu kompresji JPEG.

Funkcja ycbcr_to_rgb wykonuje odwrotną transformację, konwertując obraz z przestrzeni barw YCbCr z powrotem do RGB, używając odpowiednich współczynników macierzy konwersji i ograniczając wartości do zakresu 0-255, co stanowi ostatni krok dekompresji JPEG.

Funkcja chroma_subsampling implementuje podpróbkowanie chrominancji w formacie 4:2:0, gdzie składowe Cb i Cr są redukowane dwukrotnie w obu wymiarach przez uśrednianie wartości w blokach 2x2, podczas gdy składowa Y pozostaje niezmieniona, co znacząco zmniejsza ilość danych bez dużego wpływu na postrzeganą jakość.

Funkcje level_shift_and_block_split oraz combine_blocks_and_unshift wykonują przesunięcie poziomów (odejmując/dodając 128) i podział/łączenie obrazu na bloki 8x8, co przygotowuje dane do transformaty kosinusowej i jest kluczowe dla dalszego przetwarzania w algorytmie JPEG.

Funkcje apply_dct i apply_idct implementują odpowiednio dyskretną transformatę kosinusową i jej odwrotność dla każdego bloku 8x8, przekształcając dane obrazu z domeny przestrzennej do dziedziny częstotliwości i z powrotem, co stanowi rdzeń algorytmu kompresji JPEG.

Funkcja calculate_huffman_size szacuje rozmiar po kodowaniu Huffmana poprzez zliczanie niezerowych współczynników DCT i przyjęcie średnio 4 bitów na współczynnik, co daje przybliżony stopień kompresji bez faktycznej implementacji kodowania.

Funkcje jpeg_encode i jpeg_decode łączą wszystkie etapy kompresji i dekompresji JPEG, wykonując kolejno transformację kolorów, podpróbkowanie chrominancji, przesunięcie poziomów, podział na bloki, transformatę DCT oraz szacowanie rozmiaru skompresowanych danych.

Funkcja save_comparison tworzy i zapisuje wizualne porównanie obrazów oryginalnych i zrekonstruowanych, obliczając także metrykę PSNR (szczytowy stosunek sygnału do szumu), która jest standardową miarą jakości obrazu po kompresji stratnej.

Funkcja evaluate_step2_impact analizuje wpływ różnych modyfikacji etapu 2 na jakość obrazu, usuwając co drugi lub co czwarty element i obliczając związane z tym zmiany w PSNR, co pozwala ocenić znaczenie poszczególnych współczynników w procesie kompresji.

Podsumowanie

Osiągnięcia

1. Format PPM (Zadanie 1)
    - Zaimplementowaliśmy obsługę obu wariantów formatu PPM (P3 - tekstowy i P6 - binarny)
    - Stworzyliśmy funkcje konwersji między formatami oraz funkcje odczytu plików PPM
    - Eksperymentalnie potwierdziliśmy teorię dotyczącą różnicy w rozmiarach plików (P3 jest około 3 razy większy od P6)
    - Zdobyliśmy praktyczną wiedzę o strukturze plików PPM i ich zastosowaniach

2. Wizualizacja kostki RGB (Zadanie 2)
    - Stworzyliśmy wizualizację przestrzeni barw RGB w formie spektrum kolorów
    - Zaimplementowaliśmy interpolację kolorów dla płynnych przejść między punktami narożnymi kostki RGB
    - Przedstawiliśmy dwie główne ścieżki przejść kolorystycznych (czarny→niebieski→cyjan→biały oraz czarny→czerwony→żółty→biały)
    - Uzyskaliśmy wizualne przedstawienie koncepcji przestrzeni barw RGB

3. Implementacja formatu PNG (Zadanie 3)
    - Stworzyliśmy od podstaw plik PNG zgodny ze specyfikacją formatu
    - Zaimplementowaliśmy prawidłową strukturę bloków PNG (chunków)
    - Zastosowaliśmy kompresję zlib do danych obrazu
    - Pogłębiliśmy wiedzę o binarnych formatach plików graficznych i ich strukturze

4. Algorytm kompresji JPEG (Zadanie 4)
    - Zaimplementowaliśmy wybrane kroki algorytmu JPEG (konwersja kolorów, podpróbkowanie chrominancji, DCT)
    - Osiągnęliśmy współczynnik kompresji około 2.1x bez kwantyzacji
    - Przeprowadziliśmy eksperymenty badające wpływ modyfikacji danych na jakość obrazu

- Uzyskaliśmy praktyczne zrozumienie zasad działania kompresji JPEG

Trudności i wyzwania

1. Implementacja formatów binarnych
   - Konieczność precyzyjnej manipulacji danymi na poziomie bajtów
   - Trudności z debugowaniem, gdyż dane binarne nie są czytelne dla człowieka
   - Wymagana dokładna znajomość specyfikacji formatów

2. Dyskretna transformata kosinusowa (DCT)
   - Złożoność obliczeniowa implementacji DCT i IDCT
   - Konieczność dokładnego zrozumienia podstaw matematycznych transformacji
   - Potencjalne błędy numeryczne przy implementacji algorytmu

3. Estymacja kompresji w algorytmie JPEG
   - Uproszczone podejście do kodowania Huffmana bez faktycznej implementacji
   - Trudność w dokładnej ocenie stopnia kompresji bez pełnej implementacji kroków 4-6

4. Synchronizacja kroków kompresji i dekompresji
   - Konieczność zapewnienia kompatybilności między krokami kodowania i dekodowania
   - Utrzymanie spójności danych na każdym etapie przetwarzani

Wnioski

1. Znaczenie formatów bezstratnych
   - Formaty PPM i PNG, mimo różnych podejść, zapewniają bezstratną reprezentację obrazów
   - PPM wyróżnia się prostotą, a PNG efektywnością kompresji bezstratnej

2. Efektywność kompresji stratnej
   - Nawet niekompletna implementacja JPEG (bez kwantyzacji) zapewnia znaczącą redukcję rozmiaru
   - Kluczowe znaczenie ma transformacja przestrzeni barw i podział na bloki

3. Rola przestrzeni barw
   - Transformacja z RGB do YCbCr ma istotny wpływ na możliwości kompresji
   - Ludzkie oko jest bardziej wrażliwe na zmiany jasności (Y) niż chrominancji (Cb, Cr)

4. Znaczenie praktycznej implementacji
   - Zrozumienie formatów graficznych wymaga praktycznej implementacji

- Teoretyczna wiedza o algorytmach przetwarzania obrazów uzupełniona praktycznym doświadczeniem daje pełniejszy obraz zagadnień

Zadanie 5

Różnice między kodem

Główną różnicą między kodem z pierwszego zadania a obecnym jest dodanie trzech istotnych kroków algorytmu JPEG: kwantyzacji (krok 4), zaokrąglania wartości do liczb całkowitych (krok 5) oraz skanowania ZigZag (krok 6). W pierwszym zadaniu mieliśmy zaimplementowane tylko kroki 0-3 (konwersja kolorów, podpróbkowanie chrominancji, podział na bloki 8x8 i transformata kosinusowa DCT) oraz uproszczone kroki 7-8 (szacowanie rozmiaru po kodowaniu Huffmana i pomiar końcowego rozmiaru obrazu). Teraz dodaliśmy pełną ścieżkę kompresji, która stanowi esencję algorytmu JPEG.

Ponadto, dodaliśmy funkcję test_quality_factors(), która pozwala przetestować wpływ różnych wartości współczynnika jakości (QF) na stopień kompresji i jakość obrazu. Zastąpiła ona wcześniejszą funkcję test_step2_modifications(), która skupiała się wyłącznie na testowaniu wpływu kroku 2.

Krok 4: Kwantyzacja

Kwantyzacja jest jednym z najważniejszych etapów kompresji JPEG, gdyż to tutaj faktycznie następuje utrata danych, która pozwala osiągnąć wysokie współczynniki kompresji. W implementacji tego kroku najpierw definiujemy dwie standardowe macierze kwantyzacji JPEG - jedną dla jasności (luminance) i drugą dla chrominancji. Następnie skalujemy je w zależności od podanego współczynnika jakości (QF). Niższe wartości QF dają większą kompresję, ale niższą jakość, podczas gdy wyższe wartości QF zachowują więcej szczegółów, ale z mniejszą kompresją.

Sam proces kwantyzacji realizowany jest przez funkcję apply_quantization().

Kwantyzacja polega na dzieleniu każdego współczynnika DCT przez odpowiadający mu element w macierzy kwantyzacji. Dla kanału jasności (Y) używamy macierzy luminance, a dla kanałów chrominancji (Cb, Cr) używamy macierzy chrominance.

Operacja odwrotna, dekwantyzacja, jest wykonywana podczas dekodowania.

Podczas dekwantyzacji każdy skwantyzowany współczynnik jest mnożony przez odpowiadający mu element w macierzy kwantyzacji.

Krok 5: Zaokrąglanie do liczb całkowitych

Po kwantyzacji wartości są zaokrąglane do najbliższych liczb całkowitych, co dalej redukuje ilość danych.

Tutaj dla każdego bloku skwantyzowanych współczynników DCT po prostu zaokrąglamy wartości do najbliższych liczb całkowitych za pomocą funkcji np.round(). Jest to prosty, ale efektywny krok, który znacznie zmniejsza zakres możliwych wartości, co poprawia efektywność kolejnych etapów kompresji, szczególnie kodowania Huffmana.

Podczas dekodowania nie musimy wykonywać szczególnych działań, aby odwrócić ten krok, ponieważ numpy automatycznie obsługuje typy danych. Niemniej jednak, dla zachowania spójności przepływu danych, dodaliśmy funkcję reverse_rounding().

Krok 6: Skanowanie ZigZag

Skanowanie ZigZag jest istotnym krokiem, który przekształca 2D bloki współczynników DCT w 1D sekwencje. Współczynniki są odczytywane w specyficznym wzorze zygzakowatym, który rozpoczyna się od lewego górnego rogu (niska częstotliwość) i kończy w prawym dolnym rogu (wysoka częstotliwość):

Ta funkcja generuje współrzędne dla wzoru ZigZag, które są następnie używane w funkcji apply_zigzag():

```
def apply_zigzag(rounded_blocks):
    """Apply ZigZag scanning to convert 2D blocks to 1D sequences (Step 6)"""
    zigzag_coords = get_zigzag_order()
    zigzag_blocks = []

    for channel_blocks in rounded_blocks:
        zigzag_channel_blocks = []

        for block in channel_blocks:
            # Create 1D array following zigzag pattern
            zigzag_sequence = np.zeros(64, dtype=block.dtype)

            for i, (row, col) in enumerate(zigzag_coords):
                zigzag_sequence[i] = block[row, col]

            zigzag_channel_blocks.append(zigzag_sequence)

        zigzag_blocks.append(zigzag_channel_blocks)

    return zigzag_blocks
```

Celem skanowania ZigZag jest grupowanie współczynników o podobnych częstotliwościach oraz umieszczanie zer (powstałych w wyniku kwantyzacji) w długich ciągach. Dzięki temu kodowanie Huffmana może znacznie efektywniej kompresować dane, ponieważ długie sekwencje zer mogą być kodowane jako pary (liczba zer, następna niezerowa wartość).

Podczas dekodowania ten proces jest odwracany przez funkcję apply_inverse_zigzag(), która przekształca 1D sekwencje z powrotem na 2D bloki.

Analiza wyników:

Oryginalny spektrum:

Pierwszy obraz przedstawia oryginalny, nieskompresowany spektrum kolorów. Widzimy płynne, nieprzerwane przejście od głębokiej czerni przez bogaty niebieski, turkusowy, aż do czystej bieli. Jest to bazowy obraz wygenerowany przez funkcję create_rgb_spectrum(), który nie przeszedł jeszcze żadnego procesu kompresji. Wszystkie przejścia tonalne są płynne i bez widocznych artefaktów.

Spektrum z QF 1:

Ten obraz przedstawia wynik kompresji JPEG z ekstremalnie niskim współczynnikiem jakości (QF=1). Przy tak niskim współczynniku jakości macierze kwantyzacji mają bardzo wysokie wartości, co prowadzi do drastycznej utraty informacji.

To, co widać na obrazie, to efekt ekstremalnie agresywnej kwantyzacji, gdzie prawie wszystkie współczynniki DCT zostały zredukowane do zera, z wyjątkiem kilku najniższych częstotliwości. Rezultatem jest bardzo wyraźna "szachownica" bloków 8x8 pikseli, gdzie każdy blok ma niemal jednolity kolor z bardzo ostrymi granicami między nimi.

Spektrum kolorów zostało silnie zredukowane do podstawowych przejść, tracąc większość subtelnych gradientów. Zamiast płynnego przejścia mamy serię bloków, które drastycznie zmieniają kolory na granicach. To najgorszy możliwy scenariusz kompresji JPEG, gdzie priorytetem jest minimalna wielkość pliku kosztem jakości.

Przy QF=1:

- Niemal wszystkie wysokoczęstotliwościowe składowe sygnału są usuwane
- W każdym bloku 8x8 pozostają głównie składowe o najniższej częstotliwości
- W rezultacie każdy blok ma prawie jednolity kolor
- Wizualnie tworzy to charakterystyczny wzór "pikseli" o rozmiarze 8x8

Ta drastyczna redukcja informacji powoduje, że obraz wygląda prawie jak mozaika lub obraz o bardzo niskiej rozdzielczości, mimo że zachowuje oryginalny wymiar w pikselach.

Spektrum z QF 10:

Ten obraz pokazuje wynik kompresji z niskim współczynnikiem jakości (QF=10). Degradacja jakości jest tutaj znacznie bardziej widoczna. Obraz jest silnie "poszatkowany", z bardzo wyraźnymi granicami bloków 8x8. Pojawiają się też charakterystyczne dla JPEG artefakty w postaci "dzwonienia" (ringing) - falistych wzorów wokół ostrych krawędzi. Niski współczynnik jakości powoduje, że macierze kwantyzacji mają większe wartości, co prowadzi do utraty większej ilości wysokoczęstotliwościowych szczegółów.

Spektrum z QF 50:

Tu widzimy kompresję ze średnim współczynnikiem jakości (QF=50). Jest to kompromis między jakością a kompresją. Bloki 8x8 są nadal widoczne, ale mniej wyraźnie niż przy QF=10. Przejścia kolorów są dość płynne, choć uważne oko może dostrzec pewne artefakty, szczególnie w obszarach przejściowych między kolorami.

Spektrum z QF 90:

Ostatni obraz przedstawia kompresję z wysokim współczynnikiem jakości (QF=90). Jakość jest znacznie lepsza niż w poprzednich przypadkach, z bardzo subtelnymi artefaktami. Podział na bloki 8x8 jest prawie niewidoczny, a przejścia kolorów są niemal tak płynne jak w oryginale. Wynika to z faktu, że przy wysokim QF macierze kwantyzacji mają niskie wartości, więc zachowywanych jest więcej szczegółów, szczególnie wysokoczęstotliwościowych.

"Poszatkowanie" (blokowe artefakty) jest bezpośrednim rezultatem kluczowych aspektów algorytmu JPEG:

1. Podział na bloki 8x8: JPEG dzieli obraz na małe bloki 8x8 pikseli, które są przetwarzane niezależnie. Przy niższych współczynnikach jakości granice między tymi blokami stają się widoczne.
2. Kwantyzacja (krok 4): Jest to główny krok powodujący stratę danych. Im niższy współczynnik jakości, tym większe wartości w macierzy kwantyzacji, co prowadzi do większej utraty informacji o wysokich częstotliwościach (drobnych szczegółach i przejściach).
3. Zaokrąglanie (krok 5): Zaokrąglanie skwantyzowanych wartości do liczb całkowitych dodatkowo redukuje precyzję, co może podkreślać efekt bloków.

Widoczne artefakty są szczególnie zauważalne w obrazach z płynnymi przejściami kolorów (jak ten spektrum), ponieważ subtelne gradienty wymagają zachowania drobnych różnic między sąsiednimi pikselami, które często są tracone podczas kwantyzacji.

Im niższy współczynnik jakości (QF), tym silniejsze "poszatkowanie", ponieważ więcej wysokoczęstotliwościowych szczegółów jest eliminowanych, co skutkuje bardziej jednolitymi blokami o ostrych granicach. Przy QF=90 zachowywana jest większość detali, więc efekt poszatkowania jest minimalny.