Implementacja częściowej kompresji JPEG

Dzisiejsza instrukcja będzie poświęcona zapoznaniu się z pewną uproszczoną wersją kompresji w standardzie JPEG. JPEG (ang. Joint Photographic Experts Group) to algorytm stratnej kompresji grafiki rastrowej, wykorzystany w formacie plików graficznych o tej samej nazwie.

Zadania

1. Do zaimplementowania, poprawnie działający, uproszczony algorytm JEPG, składający się z dwóch funkcji (kompresja i dekompresja) oraz działający według schematu opisanego w instrukcji. Powinien on zapewniać 4 sposoby wywołania ma podstawie dwóch opcji **(0.7 pkt)**:
    1. Redukcję chrominancji — wybór pomiędzy `4:4:4` a `4:2:2`, (nie robić `4:2:0` z przykładu)
    2. Wybór tablicy kwantyzującej albo zastąpienie jej tablicą jedynek.
2. Sprawozdanie/raport z działania programu **(0.3 pkt)**.

    Wybrać 4 duże zdjęcia i przeanalizować mniejsze ich fragmenty. Wybrać 3-4 fragmenty dla każdego z obrazów, tak aby reprezentowały one różnego rodzaju elementy (Statyczna jasność i zmienne kolory lub zmienna jasność i statyczne kolory albo zmienne kolory i zmienna jasność itd.). I porównać ich działanie dla różnych wariantów działania naszego algorytmu. W sumie na każdym wycinku do sprawdzenia na każdym są 4 warianty.

    Pamiętać, żeby załączane obrazy były czytelne, czyli nie załączać zrzutów ekranów tylko zapisane `ploty` (`plt.savefig`). Najlepiej wykorzystać skrypty do automatycznego generowania plików `.docx` z wcześniejszych instrukcji lub sekcji FAQ na stronie. Starajcie się je tak projektować, żeby w PDF-ie nie uległy one pomniejszaniu, bo kompresja dokłada dodatkowe artefakty. Wycinki najlepiej, żeby były jednego rozmiaru najlepiej `128x128`. O tym, jak to zrobić było na wcześniejszych zajęciach.

    Pamiętać, żeby sprawdzić, o ile (może być w %) skracają się nasze wektory dla warstw po kompresji bezstratnej wybranym przez was algorytmem (RLE lub ByteRun, tylko napiszcie, który z nich wykorzystujecie). Domyślnie, każda z warstw ma rozmiar waszego wycinka, czyli $128 * 128 = 16384$.

Do oddania

- kod źródłowy (jeden plik `.py`)
- sprawozdanie z obserwacjami i wynikami (format `PDF`)

Uproszczony algorytm kompresji JPEG

JPEG (ang. Joint Photographic Experts Group) to algorytm stratnej kompresji grafiki rastrowej, wykorzystany w formacie plików graficznych o tej samej nazwie. Istnieje kilka sporo modyfikacji algorytmu JPEG, ale głównie używany algorytm kompresji JPEG można przedstawić za pomocą kilku kroków:

- Konwersja z przestrzeni `RGB` do przestrzeni `YCbCr` (luminancja i dwie warstwy chrominancji)
- Ewentualna redukcja chrominancji (Chroma Subsampling)
- Podział obrazu na bloki 8x8 (każda warstwa przetwarzana jest osobno)
- Wykonanie dyskretnej transformaty kosinusowej (DCT)
- Kwantyzacja poprzez zastąpienie wartości zmiennoprzecinkowych, wartościami całkowitymi.
    - Ewentualne dzielenie przez macierze kwantyzacji w celu zwiększenia kompresji
- Porządkowanie współczynników DCT w sposób zikowy poprzez zygzakowanie
- Kompresja bezstratna danych najczęściej algorytmem Hoffmanna.

Kontakt

Materiały

Przetwarzanie Obrazów

Systemy Multimedialne

Interaktywne Systemy Multimedialne

Przetwarzanie i analiza danych

FAQ - najczęściej zadawane pytania

Zainteresowanych matematyką, jaka stoi za całym algorytmem i sposobem ich wyliczania, odsyłam do obejrzenia dwóch filmów na YouTube temu poświęconym, które dobrze to prezentują Część 1 [ENG] oraz Część 2 [ENG].

Nasz algorytm

Kontakt

Materiały

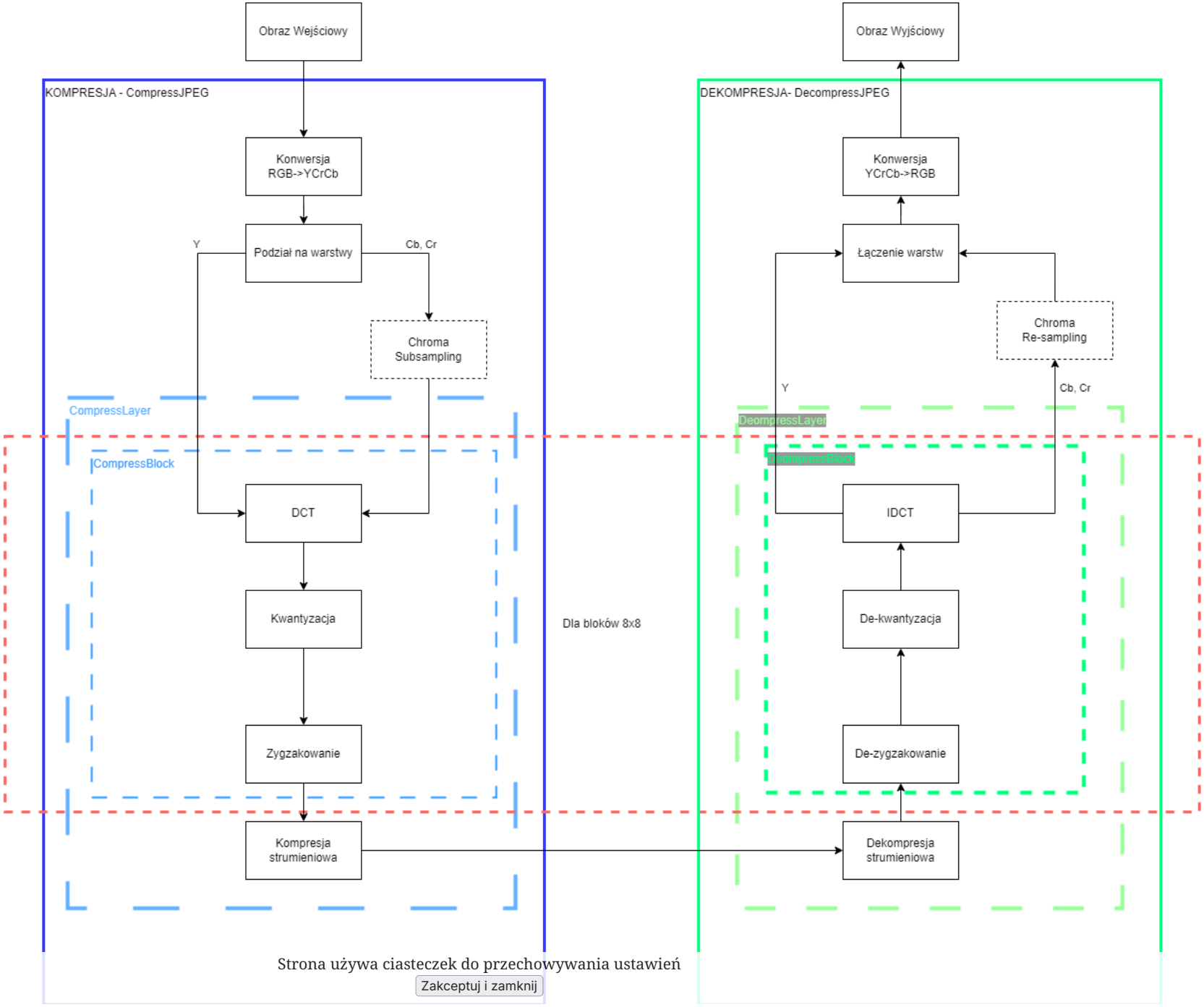
Przetwarzanie Obrazów

Systemy Multimedialne

Interaktywne Systemy Multimedialne

Przetwarzanie i analiza danych

FAQ - najczęściej zadawane pytania



Schemat naszej wersji algorytmu JPEG

Pisany przez nas algorytm będzie finalnie przebiegał według powyższego schematu, ale będziemy go powoli rozwijać, monitorując efekty jego działania. Każdemu z etapów działania algorytmu kompresji odpowiada funkcja odwrotna wykorzystywana na tym samym etapie podczas dekompresji. Dlatego na każdym etapie będziemy implementować od razu obie funkcjonalności jednocześnie i sprawdzać ich działanie. Do przechowywania skompresowanej informacji o naszej kompresji najlepiej wykorzystać pustą klasę lub klasę z już zadeklarowanymi elementami, ale **proszę nie umieszczać w niej żadnych dodatkowych funkcji**:

```
01. ## wybrać jeden kontener i nie umieszczać w nim dodatkowych funkcji
02. class ver1:
03.     Y=np.array([])
04.     Cb=np.array([])
05.     Cr=np.array([])
06.     ChromaRatio="4:4:4"
07.     QY=np.ones((8,8))
08.     QC=np.ones((8,8))
09.     shape=(0,0,3)
10.
11. class ver2:
12.     def __init__(self,Y,Cb,Cr,OGShape,Ratio="4:4:4",QY=np.ones((8,8)),QC=np.ones((8,8))):
13.         self.shape = OGShape
14.         self.Y=Y
15.         self.Cb=Cb
16.         self.Cr=Cr
17.         self.ChromaRatio=Ratio
18.         self.QY=QY
19.         self.QC=QC
```

W obu przypadkach będziemy odwoływać się do elementów w jej wnętrzu w ten sam sposób:

```
01. data1=ver1()
02. data1.shape=(1,1)
03.
04. data2=ver2(...)
05. data2.shape=(1,1)
```

Pracując z naszym algorytmem, zakładamy, że wymiary naszego obrazu są podzielne przez 8 (lub najlepiej 16), w przeciwnym przypadku musimy uzupełnić brakujące piksele (innymi wartościami).

Uwaga: Jeżeli natraficie na jakieś problemy, coś nie będzie się dobrze wyświetlać lub nie będziecie wiedzieć, jak coś zrobić radzę sprawdzić sekcję z uwagami końcowymi w instrukcji, gdzie prawdopodobnie będą znajdować się informacje na ten temat.

Przestrzeń *YCbCr*

Kontakt

Materiały

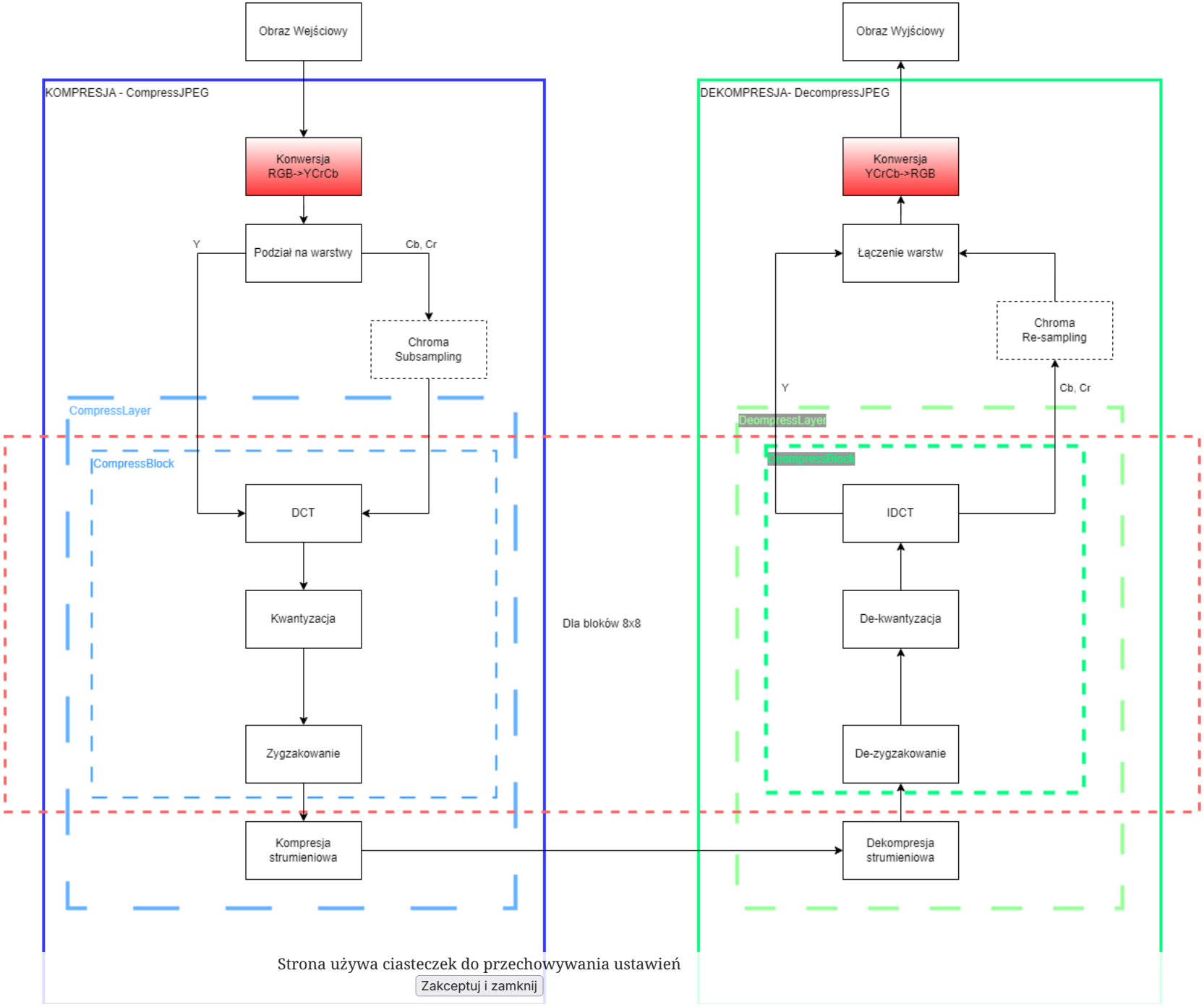
Przetwarzanie Obrazów

Systemy Multimedialne

Interaktywne Systemy Multimedialne

Przetwarzanie i analiza danych

FAQ - najczęściej zadawane pytania



Schemat naszej wersji algorytmu JPEG — bieżący krok konwersja *RGB->YCrCb* oraz *YCrCb->RGB*

Na początku musimy zmienić nasza przestrzeń koloru, w jakiej reprezentowany jest nasz obraz. Wykorzystamy do tego przestrzeń *YCbCr* (UWAGA! W OpenCV generowaną jako *YCrCb*). Jest to model przestrzeni kolorów używany do cyfrowego przesyłania oraz przechowywania obrazów i wideo. Wykorzystuje do tego trzy typy danych: *Y* – składową luminancji, *Cb* – składową różnicową chrominancji *Y-B*, stanowiącą różnicę między luminancją a niebieskim, oraz *Cr* – składową chrominancji *Y-R*, stanowiącą różnicę między luminancją a czerwonym. Kolor zielony jest uzyskiwany na podstawie tych trzech wartości. Do przejścia pomiędzy tymi przestrzeniami wykorzystamy OpenCV. Pamiętać należy jednak, że domyślnie obraz dla OpenCV jest zapisany albo w typie *uint8* (lub *float*, ale dzisiaj tego nie będziemy tego używać), jednak pozostanie na tym typie danych, znacząco utrudni nam wykonywanie dalszych operacji, dlatego pamiętać o wykonywaniu odpowiednich rzutowań.

```
    01. YCrCb=cv2.cvtColor(RGB,cv2.COLOR_RGB2YCrCb).astype(int)
02. RGB=cv2.cvtColor(YCrCb.astype(np.uint8),cv2.COLOR_YCrCb2RGB)
```

Uwaga: Dyskretna transformacja cosinusowa (DCT) powinna otrzymywać dane wyśrodkowane na 0. Obrazy na *uint8* są wyśrodkowane na wartości *128*, dlatego przed operacją należy od wartości każdej warstwy należy odjąć wartość *128* przed prowadzeniem obliczeń (można robić to na całym obrazie lub wewnątrz funkcji kompresującej blok) oraz dodać tę wartość po odtworzeniu.

Konstrukcja programu

W ramach programu powinny znaleźć się dwie funkcje kompresji i dekompresji. Funkcja kompresji powinna przyjąć jako argument wszystkie informacje, jakie potrzebujemy do działania naszego algorytmu, a następnie zwrócić klasę z tymi informacjami w taki sposób, żeby funkcja dekompresji była w stanie je odtworzyć. Może być to coś podobnego do poniższych przykładów.

```
    01. def CompressJPEG(RGB,Ratio="4:4:4",QY=np.ones((8,8)),QC=np.ones((8,8))):
02.     # RGB -> YCrCb
03.     JPEG= verX(...)
04.     # zapisać dane z wejścia do klasy
05.     # Tu chroma subsampling
06.     JPEG.Y=CompressLayer(JPEG.Y,JPEG.QY)
07.     JPEG.Cr=CompressLayer(JPEG.Cr,JPEG.QC)
08.     JPEG.Cb=CompressLayer(JPEG.Cb,JPEG.QC)
09.     # tu dochodzi kompresja bezstratna
10.     return JPEG
11.
12. def DecompressJPEG(JPEG):
13.     # dekompresja bezstratna
14.     Y=DecompressLayer(JPEG.Y,JPEG.QY)
15.     Cr=DecompressLayer(JPEG.Cr,JPEG.QC)
16.     Cb=DecompressLayer(JPEG.Cb,JPEG.QC)
17.     # Tu chroma resampling
18.     # tu rekonstrukcja obrazu
19.     # YCrCb -> RGB
20.     return RGB
```

Podział na bloki i rekonstrukcja danych

Pomińmy na moment etap redukcji chrominancji (zrobimy to później) i przejdźmy do operacji wykonywanych na samych blokach *8×8*. **Warto wyodrębnić zestaw operacji wykonywanych na [illegible] funkcję**, ponieważ same operacje są niezależne od warstwy, na której

Kontakt

Materiały

Przetwarzanie Obrazów

Systemy Multimedialne

Interaktywne Systemy Multimedialne

Przetwarzanie i analiza danych

FAQ - najczęściej zadawane pytania

będzie wykonywana operacja oraz ewentualnymi dodatkowymi parametrami, które będziemy do niej przekazywać. **Nie próbujcie też wykonywać wszystkich operacji na wszystkich trzech warstwach na raz**. Przetwarzajcie każdą warstwę osobno. Funkcja kodująca będzie przyjmować tablicę dwuwymiarową (8x8) i po całej operacji będzie zwracać wektor 64-elementowy. Funkcja dekodująca natomiast będzie przyjmować ten wektor i zwróci tablicę (8x8). Poniżej przykład:

```
01. def CompressBlock(block,Q):
02.     ###
03.     return vector
04.
05. def DecompressBlock(vector,Q):
06.     ###
07.     return block
```

Samo wyodrębnienie naszych bloków podczas kodowania wykonać można poprzez użycie podwójnej pętli `for` po kolumnach i wierszach z krokiem 8. Natomiast przy rekonstrukcji można wykorzystać indeks (`idx` w kodzie poniżej) do wyliczenia pozycji bloku w przestrzeni przy użyciu prostych operacji arytmetycznych (`%` oraz `//`) i informacji jak szeroki jest nasz obraz.

```
01. ## podział na bloki
02. # L - warstwa kompresowana
03. # S - wektor wyjściowy
04. def CompressLayer(L,Q):
05.     S=np.array([])
06.     for w in range(0,L.shape[0],8):
07.         for k in range(0,L.shape[1],8):
08.             block=L[w:(w+8),k:(k+8)]
09.             S=np.append(S, CompressBlock(block,Q))
10.
11. ## wyodrębnianie bloków z wektora
12. # L - warstwa o oczekiwanym rozmiarze
13. # S - długi wektor zawierający skompresowane dane
14. def DecompressLayer(S,Q):
15.     L= # zadeklaruj odpowiedniego rozmiaru macierzy
16.     for idx,i in enumerate(range(0,S.shape[0],64)):
17.         vector=S[i:(i+64)]
18.         m=L.shape[1]/8
19.         k=int((idx%m)*8)
20.         w=int((idx//m)*8)
21.         L[w:(w+8),k:(k+8)]=DecompressBlock(vector,Q)
```

Operacje wykonywane na blokach — część 1

Jak było już wspomniane podczas podziału na bloki, operacje zamieszczone, w tej części instrukcji warto umieszczać wewnątrz funkcji, która będzie przetwarzać całe bloki. Operacje wykonywane na różnych warstwach różnią się tylko parametrami (inna macierz `Q`), a nie operacjami. Na tym etapie nasza funkcja będzie zawierała tylko diw poniższe operacja.

Dyskretna transformacja cosinusowa (DCT)

Kontakt

Materiały

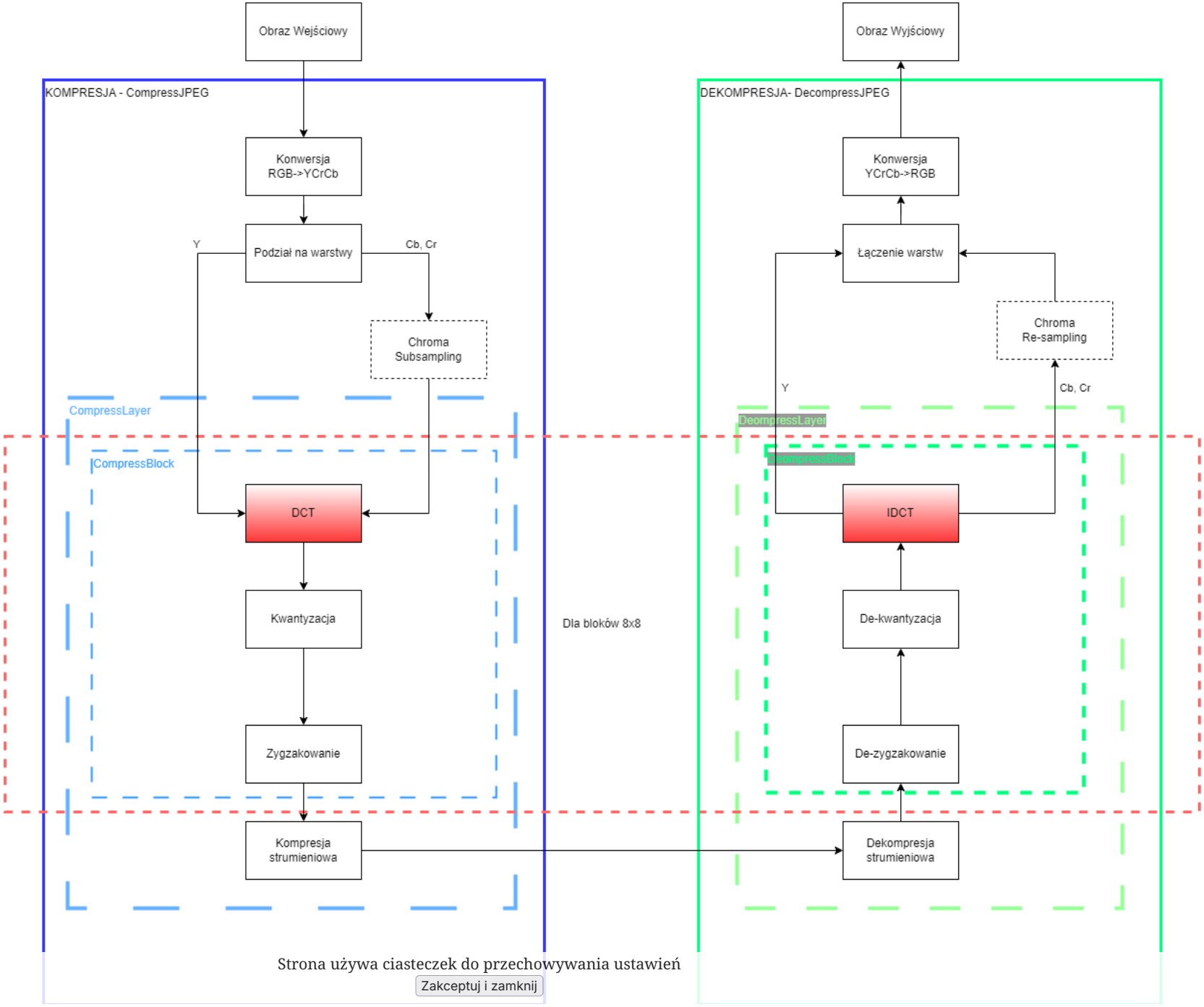
Przetwarzanie Obrazów

Systemy Multimedialne

Interaktywne Systemy Multimedialne

Przetwarzanie i analiza danych

FAQ - najczęściej zadawane pytania



Schemat naszej wersji algorytmu JPEG — bieżący krok transformata DCT

Pierwszym etapem kompresji na naszym bloku danych jest wyliczenie współczynników dla dyskretnej transformacji cosinusowej (DCT). Poniżej znajduje się kod funkcji realizującej dwuwymiarowe DCT, jak również jej odwrotność IDCT2. Obie funkcje zwracają macierze 8x8 zawierające liczby zmiennoprzecinkowe (*float*). Dla osób zainteresowanych zrozumieniem jak działa DCT odsyłam, do wcześniej załączonych filmów oraz literatury. Ogólnie służy ona do wyznaczenia współczynników DCT, czyli jak dużo składowych cosinusowych, o określonych okresach należy dodać do siebie, aby otrzymać nasz sygnał źródłowy. W punkcie (0,0) naszej macierzy (podobnie jak w widmie Fourierowskim) znajduje się składowa stała, a im dalej on niego znajdują się składowe o wyższych częstotliwościach.

```
01. import scipy.fftpack
02.
03. def dct2(a):
04.     return scipy.fftpack.dct( scipy.fftpack.dct( a.astype(float), axis=0, norm='ortho' ), axis=1,
norm='ortho' )
05.
06. def idct2(a):
07.     return scipy.fftpack.idct( scipy.fftpack.idct( a.astype(float), axis=0 , norm='ortho'), axis=1 ,
norm='ortho')
```

Zygzakowanie

Kontakt

Materiały

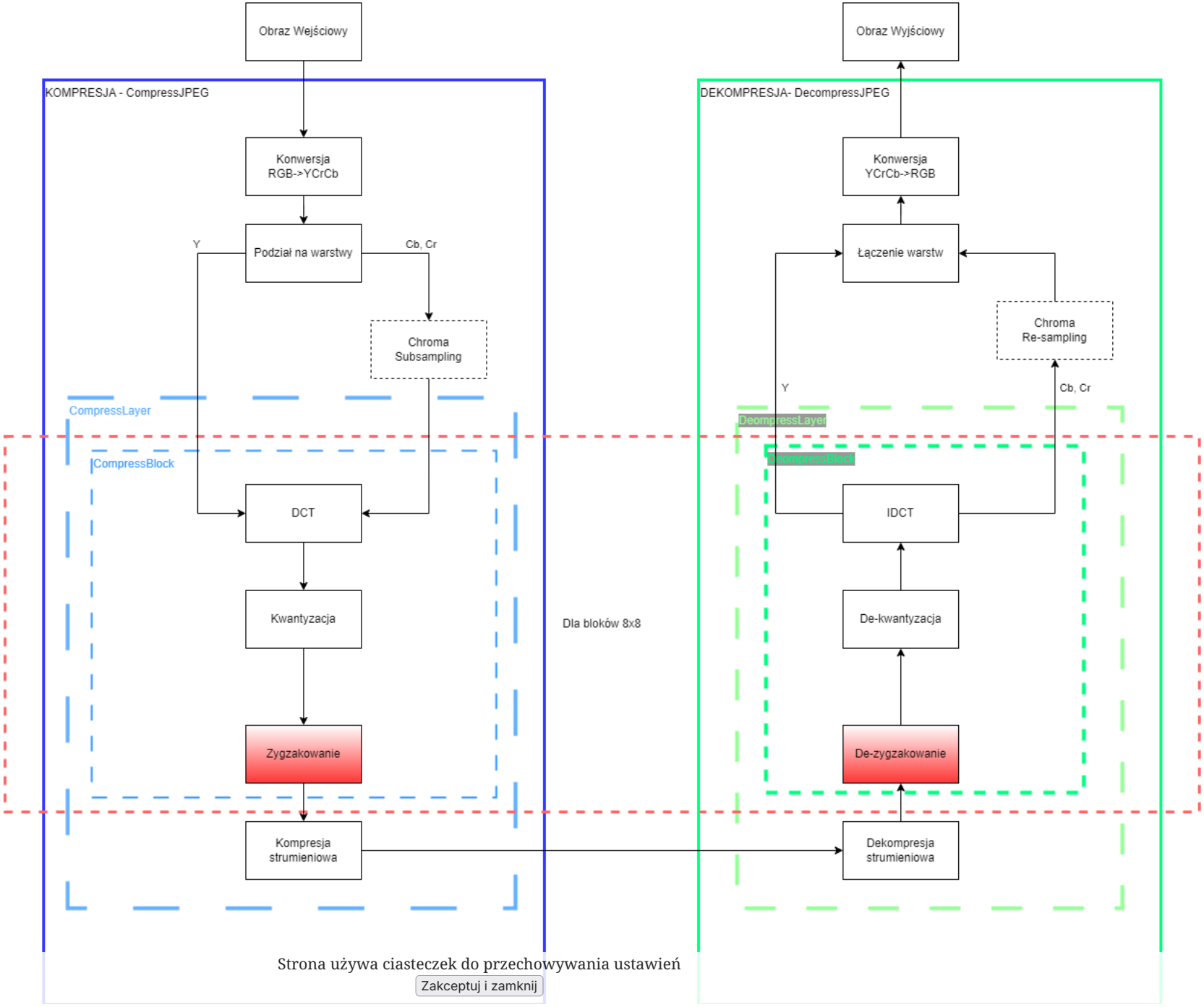
Przetwarzanie Obrazów

Systemy Multimedialne

Interaktywne Systemy Multimedialne

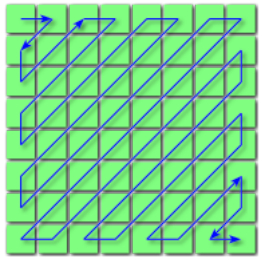
Przetwarzanie i analiza danych

FAQ - najczęściej zadawane pytania



Schemat naszej wersji algorytmu JPEG — bieżący krok Zygzakowanie

Na chwilę pomijamy kolejny krok, żeby zaimplementować ostatnią funkcję niepowodującą utraty, żadnych danych. Funkcja zygzakująca (ang. 'zigzagging') zamienia ona macierz 8x8 na wektor 64-elementowy, ale w przeciwieństwie do funkcji `.flatten()`, robi to według poniższego schematu:



Schemat zygzakowania (źródło:https://commons.wikimedia.org/wiki/File:JPEG_ZigZag.svg)

Taka operacja przydaje się bardzo przy porządkowaniu danych przed dalszą kompresją. W wyniku kwantyzacji danych będziemy posiadali zwykle bardzo dużo zer w wyższych partiach częstotliwości, które przy takim zgrupowaniu, powinny tworzyć dłuższe ich ciągi, ułatwiając ich kompresję.

```
01. def zigzag(A):
02.     template= np.array([
03.             [0,  1,  5,  6,  14, 15, 27, 28],
04.             [2,  4,  7,  13, 16, 26, 29, 42],
05.             [3,  8,  12, 17, 25, 30, 41, 43],
06.             [9,  11, 18, 24, 31, 40, 44, 53],
07.             [10, 19, 23, 32, 39, 45, 52, 54],
08.             [20, 22, 33, 38, 46, 51, 55, 60],
09.             [21, 34, 37, 47, 50, 56, 59, 61],
10.             [35, 36, 48, 49, 57, 58, 62, 63],
11.             ])
12.     if len(A.shape)==1:
13.         B=np.zeros((8,8))
14.         for r in range(0,8):
15.             for c in range(0,8):
16.                 B[r,c]=A[template[r,c]]
17.     else:
18.         B=np.zeros((64,))
19.         for r in range(0,8):
20.             for c in range(0,8):
21.                 B[template[r,c]]=A[r,c]
22.     return B
```

Do tego momentu wszystkie operacje są odwracalne, czyli nie powodują utraty informacji. Wszystkie dalsze operacje powodują już pewną utratę informacji. Na tym etapie warto sprawdzić, czy was kod działa poprawnie i czy odtwarza wasz obraz.

Teraz przejdziemy do dodawania operacji, które powodują już utratę informacji.

Kwantyzacja

Kontakt

Materiały

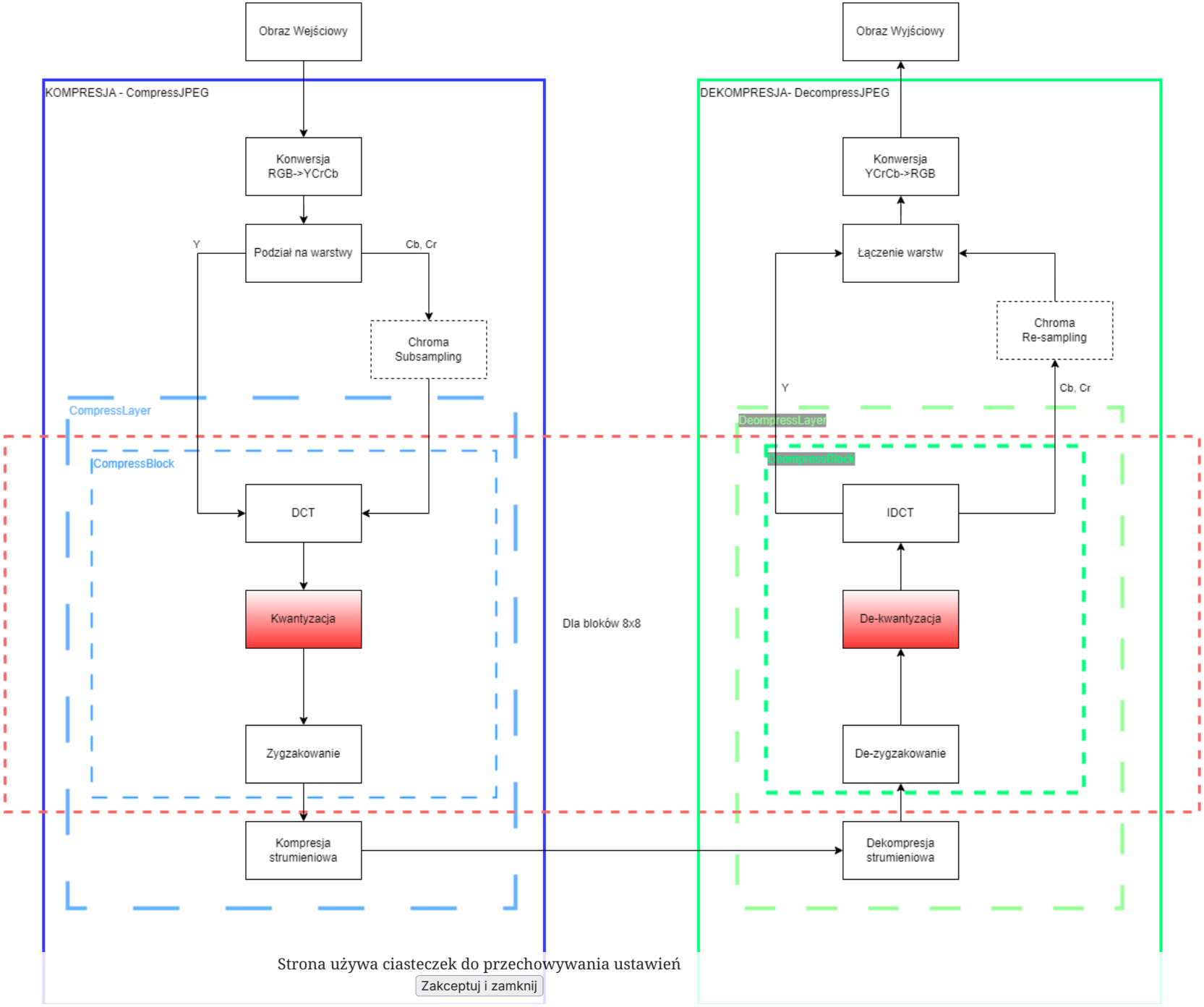
Przetwarzanie Obrazów

Systemy Multimedialne

Interaktywne Systemy Multimedialne

Przetwarzanie i analiza danych

FAQ - najczęściej zadawane pytania



Schemat naszej wersji algorytmu JPEG — bieżący krok kwantyzacja

Wróćmy teraz do naszych współczynników DCT, podstawową formą ich kwantyzacji jest zwykła zmiana z formatu zmiennoprzecinkowego na liczby całkowite (`.astype(int)`). Powoduje to pewną utratę informacji, ale jest ona stosunkowo niewielka. Drugim sposobem na zaoszczędzenie pamięci, jest zmniejszenie wpływu komponentów DCT o większych częstotliwościach na nasz rekonstruowany obraz, poprzez podzielenie wszystkich wartości dla komponentów DCT przez tzw. tablice kwantyzacji. Poniżej podano dwie takie tablice pochodzące ze standardu JPEG dla jakości 50%. Pierwsza z nich jest przeznaczona do kompresji Luminancji (`Y`) druga dla Chrominancji (`Cb` oraz `Cr`). Podczas kwantyzacji dzielimy naszą macierz współczynników DCT przez naszą tablicę, a podczas de-kwantyzacji mnożymy je przed poddaniem ich działaniu IDCT. Do otrzymania pierwszej wersji naszej kwantyzacji, czyli prostego rzutowania na `int` możemy zastąpić naszą macierz `Q` wartościami 1. Ponieważ 1 jest elementem neutralnym, wiec dzielenie i mnożenie przez niego da nam te same wartości.

```
    01. qd=np.round(d/Q).astype(int)
02. pd=qd*Q
```

Operacja ta spowoduje, że w ciągu poddanym zygzakowaniu pojawi się znacznie więcej zer, co ułatwi jego dalszą kompresję. Tablicę kwantyzacji do naszej funkcji przetwarzającej bloki jako parametr, ponieważ różne warstwy wykorzystują różne tablice kwantyzacji. Należy również pamiętać, że wykorzystywaną tablicę należy dołączyć do naszej struktur, ponieważ jest ona również potrzebna do dekompresowania obrazu.

Tablica Kwantyzacji dla Luminancji dla jakości 50%:

```
    01. QY= np.array([
02.         [16, 11, 10, 16, 24,  40,  51,  61],
03.         [12, 12, 14, 19, 26,  58,  60,  55],
04.         [14, 13, 16, 24, 40,  57,  69,  56],
05.         [14, 17, 22, 29, 51,  87,  80,  62],
06.         [18, 22, 37, 56, 68,  109, 103, 77],
07.         [24, 36, 55, 64, 81,  104, 113, 92],
08.         [49, 64, 78, 87, 103, 121, 120, 101],
09.         [72, 92, 95, 98, 112, 100, 103, 99],
10.         ])
```

Tablica kwantyzacji dla Chrominancji po redukcji 2:1 dla jakości 50%, ale można ją stosować również na pełnej chrominancji:

```
    01. QC= np.array([
02.         [17, 18, 24, 47, 99, 99, 99, 99],
03.         [18, 21, 26, 66, 99, 99, 99, 99],
04.         [24, 26, 56, 99, 99, 99, 99, 99],
05.         [47, 66, 99, 99, 99, 99, 99, 99],
06.         [99, 99, 99, 99, 99, 99, 99, 99],
07.         [99, 99, 99, 99, 99, 99, 99, 99],
08.         [99, 99, 99, 99, 99, 99, 99, 99],
09.         [99, 99, 99, 99, 99, 99, 99, 99],
10.         ])
```

Tablica neutralna dla zwykłego rzutowania na `int`:

```
    01. QN= np.ones((8,8))
```

Redukcja Chrominancji — Chroma Subsampling

Kontakt

Materiały

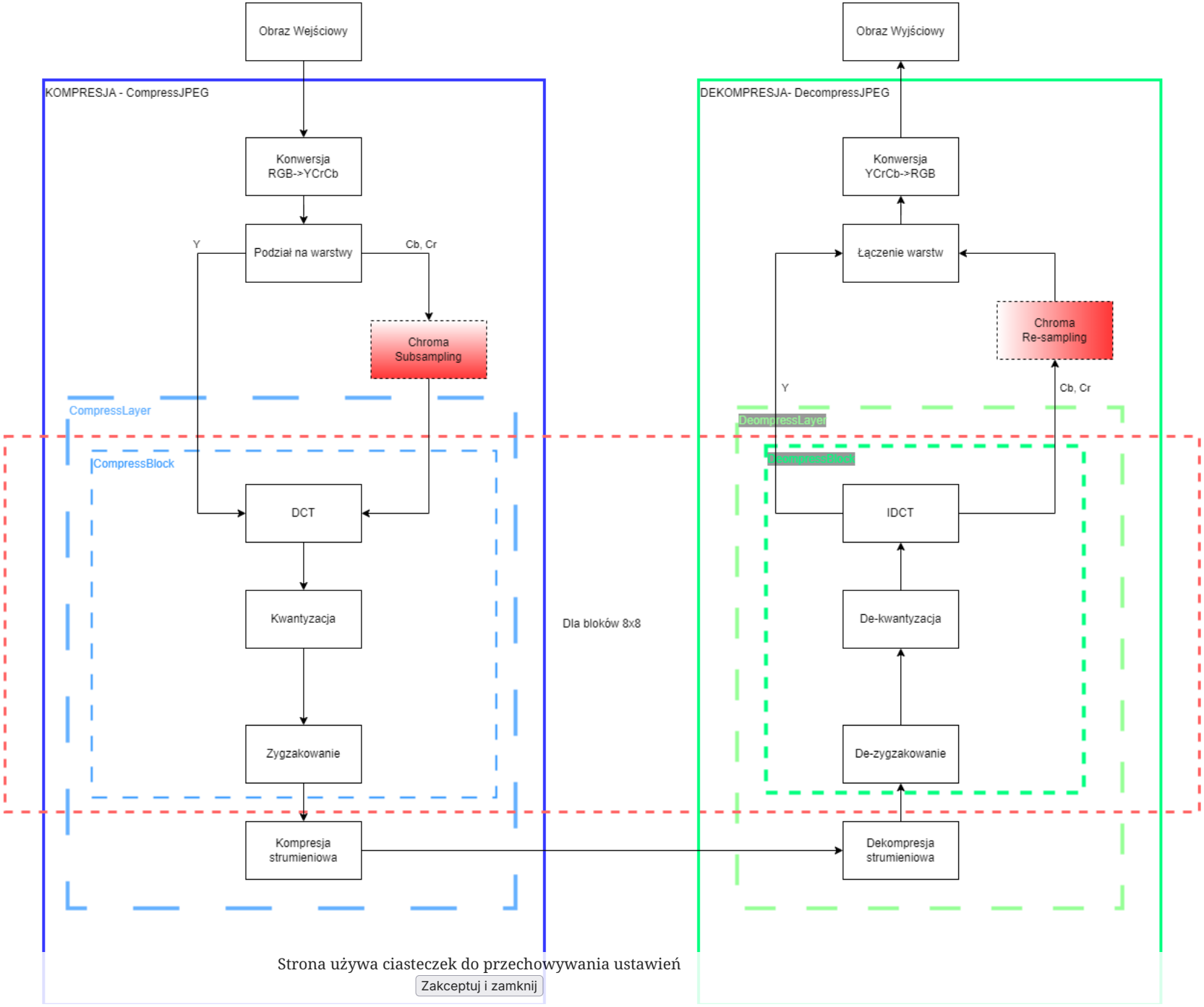
Przetwarzanie Obrazów

Systemy Multimedialne

Interaktywne Systemy Multimedialne

Przetwarzanie i analiza danych

FAQ - najczęściej zadawane pytania



Kontakt

Materiały

Przetwarzanie Obrazów

Systemy Multimedialne

Interaktywne Systemy Multimedialne

Przetwarzanie i analiza danych

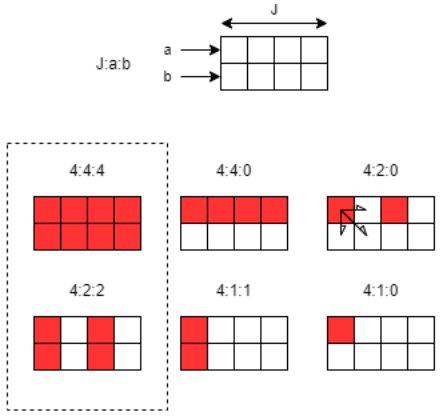
FAQ - najczęściej zadawane pytania

Schemat naszej wersji algorytmu JPEG — bieżący krok Chroma Subsampling i Chroma Resampling

Kolejnym wcześniej przez nas wcześniej etapem jest redukcja chrominancji (ang. Chroma Subsampling). Jest to jedna z operacji kompresji polegająca na zmniejszeniu informacji na temat koloru w naszym obrazie. Wykorzystujemy tutaj fakt, że ludzkie oko jest bardziej wrażliwe na zmianę jasności (Luminancja), niż koloru (Chrominancja). Subsampling zatem wykonujemy na warstwach *Cb* oraz *Cr* (dla każdej z nich osobno). Jego rodzaj opisywany jest przez 3 wartości opisane w ciągu *J:a:b*. W przypadku naszego algorytmu JPEG będziemy zajmować się tylko dwoma aspektami redukcji chrominancji. Są to wersje: *4:4:4* oraz *4:2:2*. Poniżej trochę więcej informacji na ten temat, wraz z przykładami.

Chroma subsampling

Redukcja chrominancji (ang. Chroma Subsampling) to jedna z operacji kompresji polegająca na zmniejszeniu informacji na temat koloru w naszym obrazie. Wykorzystujemy tutaj fakt, że ludzkie oko jest bardziej wrażliwe na zmianę jasności (Luminancja), niż koloru (Chrominancja). Subsampling zatem wykonujemy na warstwach *Cb* oraz *Cr* (dla każdej z nich osobno). Jego rodzaj opisywany jest przez 3 wartości opisane w ciągu *J:a:b* jak tak to pokazano na poniższym schemacie.



Schemat naszej wersji algorytmu JPEG — bieżący krok Chroma Subsampling i Chroma Resampling

Pierwszy parametr *J* określa, jak dużo pikseli w poziomie będziemy analizować, zwykle jest to wartość 4. Kolejny parametr *a* określa, ile wartości z pierwszego analizowanego przez nas wiersza będziemy zapisywać, natomiast parametr *b* opisuje liczbę wartości w drugim wierszu. Na przykładzie graficznym zapisywane zostają wartości zaznaczone jako czerwone kwadraty. Jak widzimy w przypadku *4:4:4* wszystkie wartości zostają zapisane, więc nic nie ulega zmianie. W pozostałych przypadkach zawsze tracimy jakieś informacje. Działanie funkcji odwrotnej polega na odtworzeniu całej warstwy naszego obrazu na podstawie zapisanych przez nas wcześniej informacji. Jeżeli nie mamy dostępnej informacji o wartości piksela, to zastępujemy (w zależności gdzie się on znajduje) inną najbliższą wartością z tego lub poprzedniego wiersza. Przykładowo dla *4:4:0* będziemy uzupełniać drugi wiersz wartościami z pierwszego, dla *4:2:0* jedna wartość będzie propagowana na 4 piksele, dla *4:1:1* nowa wartość będzie zastępowała cały wiersz, a dla *4:1:0* nowa wartość będzie rozchodziła się dla wszystkich 8 pikseli. W ramach tych laboratoriów interesują nas dwie wersje *4:4:4* - czyli obraz bez zmian oraz *4:2:2* - czyli redukcja ilości pikseli o 50% w poziomie (każda wartość będzie odtwarzana za pomocą piksela na lewo od niego). Pamiętajcie, że sposób kompresji chrominancji naszego obrazu powinien również zostać zapisany wewnątrz naszej struktury, żeby dekoder również wiedział, w jaki sposób go odtworzyć. Więcej informacji na temat redukcji chrominancji można przeczytać w artykule na Wikipedii z innymi przykładami graficznymi.

Jak do tego podejść programistycznie?

Po pierwsze nie musimy procesu redukcji chrominancji przeprowadzać na całych obrazach. Możemy na raz przetwarzać całą warstwę, czyli nie

potrzebujemy żadnych pętli.

Druga podpowiedź, czyli w jaki sposób zdecydować jaką redukcję robić. Tutaj można spróbować coś automatyzować, ale w waszym przypadku najwygodniej napisać odpowiedni zestaw *if*, *elif*, *else*. Macie dzięki temu lepszą kontrolę nad tym, co działa, a co nie.

```
01. if ratio=="4:2:2":
02.     pass
03. elif ratio=="4:2:0":
04.     ## Nie dla labolatoriów z JPEG
05.     pass
06. else: #defalut "4:4:4"
07.     pass
```

Trzecia podpowiedź, czyli jak przeprowadzić redukcję. Tu sprawa jest prosta, pakiet NumPy daje nam możliwość dowolnego adresowania danych, więc możemy dokonać redukcji poprzez wybranie elementów macierzy, które nas interesują. Przypominam, że adresowanie przebiega w kolejności *[wiersze, kolumny, warstwy]*. W każdym z adresów możemy podać generator określający, które elementy nas interesują. Adres podajemy w formacie *od:do:krok*, gdzie *od* i *do* możemy pominąć, jeżeli chcemy wektor od początku do końca zakresu, a krok domyślnie jest równy 1. Czyli jeżeli chcemy dostać wektor zawierający co 4-tą kolumnę i co drugi wiersz nasze zapytanie powinno wyglądać tak:

```
01. B=A[::2,::4]
```

Trzecia podpowiedź, czyli jak uzupełniać dane przy resamplingu. Tu można albo ręcznie uzupełniać wykorzystując adresowanie, albo wykorzystać ułatwienie w postaci funkcji np.repeat. Tylko należy pamiętać, żeby w przypadku kilku z przypadków trzeba wykonać wpierw powielanie dla wierszy, a następnie dla kolumn.

▸ Przykłady wizualne

Kompresja bezstratna?

Kontakt

Materiały

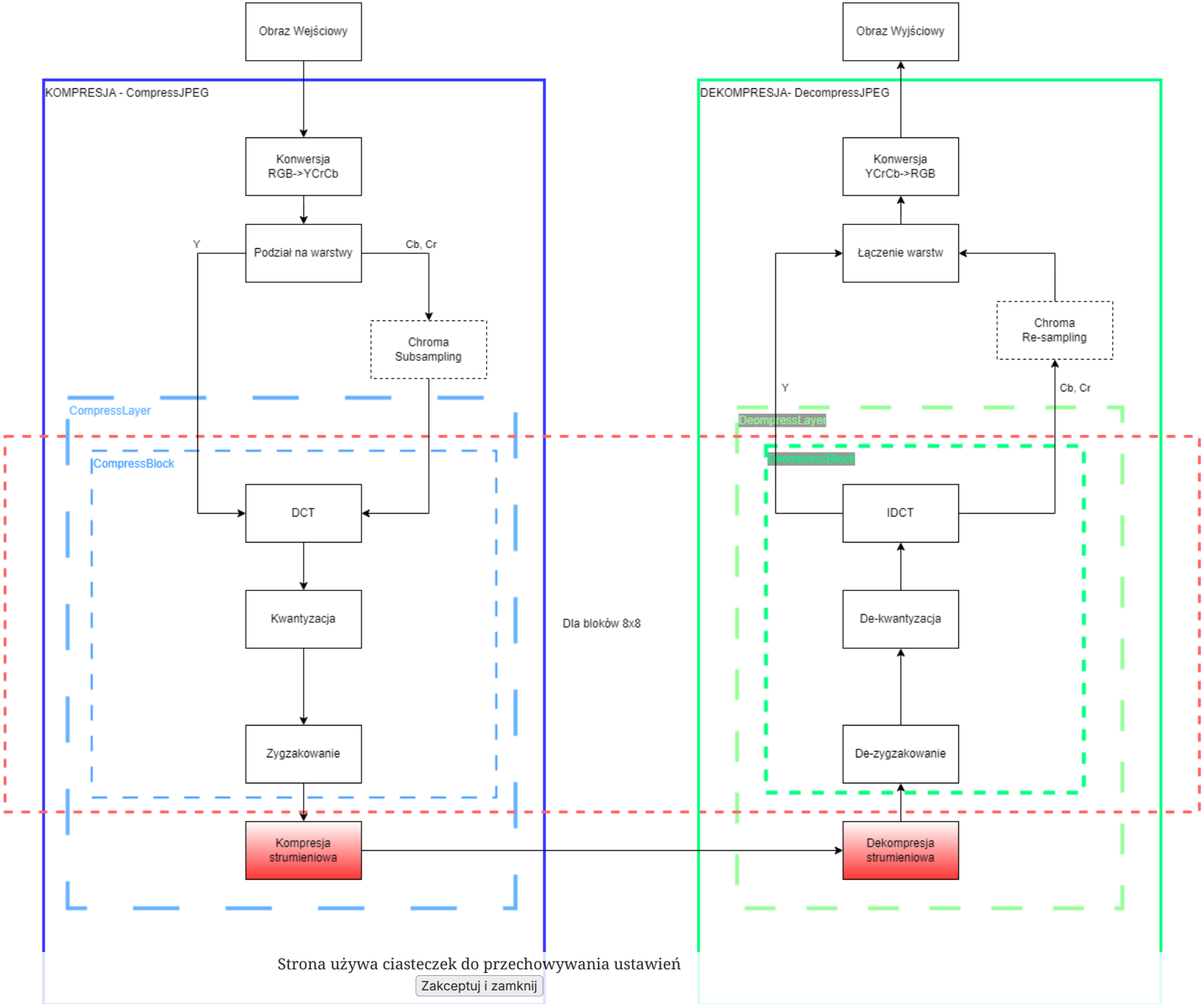
Przetwarzanie Obrazów

Systemy Multimedialne

Interaktywne Systemy Multimedialne

Przetwarzanie i analiza danych

FAQ - najczęściej zadawane pytania

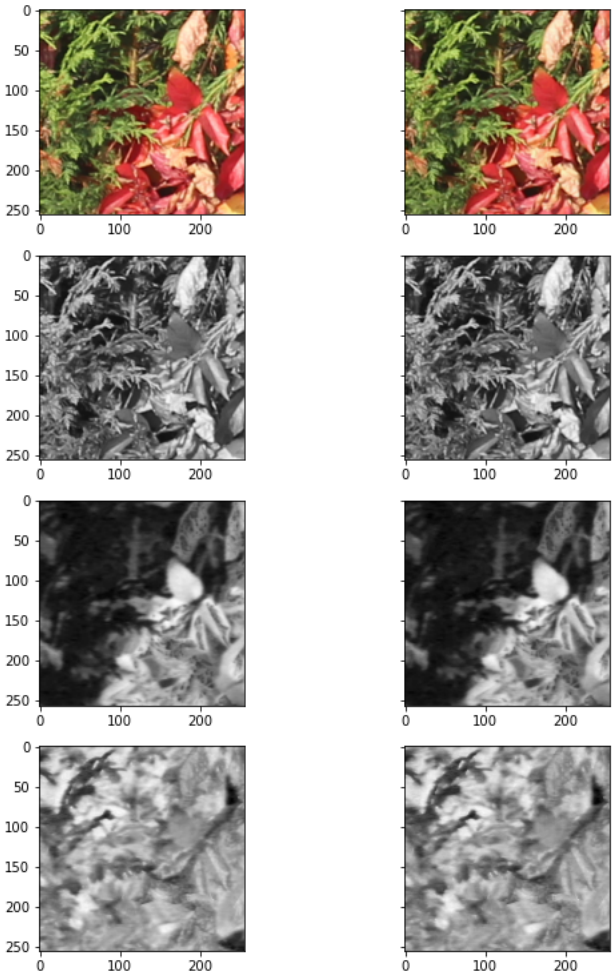


Schemat naszej wersji algorytmu JPEG — bieżący krok kompresja strumieniowa

W standardzie JPEG wykorzystywane jest najczęściej kodowanie algorytmem Hoffmanna i w nagłówku zapisywana jest cały słownik służący do dekompresji. W naszym przypadku wykorzystamy zamiast tego napisany w ramach jednych z poprzednich algorytm RLE lub ByteRun. Głównie w celu sprawdzenia o ile skrócą się nasze wektory zawierające poszczególne warstwy obrazu.

Podsumowanie

Gotowy algorytm powinien działać w miarę poprawnie. Ewentualne problemy i sposoby ich rozwiązania zostały zawarte w uwagach końcowych (jeżeli będą dalsze problemy, rozwiązania będą się pojawiać w tej sekcji). Efekty przy niewielkiej kompresji powinny wyglądać mniej więcej tak na poniższym obrazie:



Przykład działania naszego algorytmu z niewielką kompresją

Uwagi Końcowe

1. Jak złożyć 3 osobne warstwy w jeden obraz?

Kontakt

Materiały

Przetwarzanie Obrazów

Systemy Multimedialne

Interaktywne Systemy Multimedialne

Przetwarzanie i analiza danych

FAQ - najczęściej zadawane pytania

```
    01. YCrCb=np.dstack([Y,Cr,Cb]).clip(0,255).astype(np.uint8)
```

2. Jak najlepiej wyświetlać obrazy podczas testów?

   Rozwiązanie dla niewielkich obszarów do testów w przypadku większych fragmentów wyświetlany obraz może być za miały i nieczytelny. Polecam uzupełnić poniższy fragment o brakujące fragmenty kodu oraz odpowiednie etykiety i nagłówki. Następnie przerobić go na funkcję przyjmującą dwa obrazy w RGB (przed i po kompresji).

```
    01. fig, axs = plt.subplots(4, 2 , sharey=True   )
02. fig.set_size_inches(9,13)
03. # obraz oryginalny
04. axs[0,0].imshow(PRZED_RGB) #RGB
05. PRZED_YCrCb=cv2.cvtColor(PRZED_RGB,cv2.COLOR_RGB2YCrCb)
06. axs[1,0].imshow(PRZED_YCrCb[:,:,0],cmap=plt.cm.gray)
07. axs[2,0].imshow(PRZED_YCrCb[:,:,1],cmap=plt.cm.gray)
08. axs[3,0].imshow(PRZED_YCrCb[:,:,2],cmap=plt.cm.gray)
09.
10. # obraz po dekompresji
11. axs[0,1].imshow(PO_RGB) #RGB
12. PO_YCrCb=cv2.cvtColor(PO_RGB,cv2.COLOR_RGB2YCrCb)
13. axs[1,1].imshow(PO_YCrCb[:,:,0],cmap=plt.cm.gray)
14. axs[2,1].imshow(PO_YCrCb[:,:,1],cmap=plt.cm.gray)
15. axs[3,1].imshow(PO_YCrCb[:,:,2],cmap=plt.cm.gray)
```

3. Obraz po dekompresji wygląda na szary, mimo że wyświetlone osobno warstwy wyglądają dobrze?

   Nasz algorytm nie jest doskonałym, więc pewnie zdarzy się, że odtworzony po dużej kwantyzacji obraz będzie miał dużo mniejszą dynamikę kolorów niż oryginał. Dlatego trzeba wykorzystać małą sztuczkę do obejścia tego problemu. Zapisujemy wewnątrz naszej skompresowanej struktury zakres wartości dla poszczególnych warstw (minimum i maksimum). A potem przed scaleniem każdej warstwy przywracamy ten zakres.