

МИНОБРНАУКИ РОССИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«ВОРОНЕЖСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(ФГБОУ ВО «ВГУ»)

Факультет прикладной математики, информатики и механики

Кафедра математического обеспечения ЭВМ

Алгоритм JPEG

Отчёт по лабораторной работе

Студент:

Горовенко Б.М.

Преподаватель:

Сырых А.С.

Воронеж 2024

Содержание

Введение.....	2
Основная часть.....	3
Перевод изображения из цветового пространства RGB в YcbCr.....	3
Прореживание.....	4
ДКП.....	5
Квантование.....	7
RLE-кодирование и Таблица Хаффмана.....	7
Результаты.....	9
Заключение.....	12
Список использованных источников.....	13
Приложения.....	14

Введение

Цель работы: Реализовать алгоритм сжатия изображений с потерями JPEG с задаваемым коэффициентом сжатия. Для реализации алгоритма необходимо реализовать каждый из его шагов:

1. Перевод изображения из пространства RGB в YcbCr
2. Прореживание
3. Дискретно-косинусное преобразование
4. Квантование
5. Запись в файл с помощью RLE-кодирования и таблицы Хаффмана

Для реализации алгоритма JPEG используем Python с библиотеками NumPy для матричной обработки данных и pillow для получения RGB матрицы изображения и возможности поддержки большей части расширений для изображений и сохранения итога в формате „*.jpg”

Основная часть

1. Перевод изображения из цветового пространства RGB в YCbCr

При сжатии изображение преобразуется из цветового пространства RGB в YCbCr. Стандарт сжатия JPEG, не регламентирует выбор именно YCbCr, допуская и другие виды преобразования (например, с числом компонентов, отличным от трёх), и сжатие без преобразования (непосредственно в RGB), однако спецификация JFIF) предполагает использование преобразования RGB→YCbCr. А в свою очередь компоненты пространства YCbCr это Y — компонента яркости, CB и CR являются синей и красной цветоразностными компонентами. Для данного перевода и обратно в алгоритме JPEG используется данные формулы:

$$\begin{aligned} Y' &= 0 + (0.299 \cdot R'_D) + (0.587 \cdot G'_D) + (0.114 \cdot B'_D) \\ C_B &= 128 - (0.168736 \cdot R'_D) - (0.331264 \cdot G'_D) + (0.5 \cdot B'_D) \\ C_R &= 128 + (0.5 \cdot R'_D) - (0.418688 \cdot G'_D) - (0.081312 \cdot B'_D) \\ R &= Y + 1.402 \cdot (C_R - 128) \\ G &= Y - 0.34414 \cdot (C_B - 128) - 0.71414 \cdot (C_R - 128) \\ B &= Y + 1.772 \cdot (C_B - 128) \end{aligned}$$

Программная
реализация

перевода осуществляется в матричной форме, причем из-за ошибок с числами с плавающей точкой необходимо использовать округление.

```
def matrix_convert_pixel_rgb2ycbcr(pixel_rgb: Tuple[int, int, int]) -> Tuple[int, int, int]:
    return np.round((np.dot(_pixel_rgb, convertation_matrix_rgb2ycbcr) + convertation_vector)[0]).astype(int)

def matrix_convert_pixel_ycbcr2rgb(pixel_ybcr: Tuple[int, int, int]) -> Tuple[int, int, int]:
    return np.round(np.dot((pixel_ybcr - convertation_vector), convertation_matrix_ycbcr2rgb)[0]).astype(int)
```

2. Прореживание

После преобразования RGB->YCbCr для каналов изображения Cb и Cr, отвечающих за цвет, может выполняться «прореживание» (subsampling), которое заключается в том, что каждому блоку из 4 пикселей (2x2) яркостного канала Y ставятся в соответствие усреднённые значения Cb и Cr (схема прореживания «4:2:0»). При этом для каждого блока 2x2 вместо 12 значений (4 Y, 4 Cb и 4 Cr) используется всего 6 (4 Y и по одному усреднённому Cb и Cr). Если к качеству восстановленного после сжатия изображения предъявляются повышенные требования, прореживание может выполняться лишь в каком-то одном направлении — по вертикали (схема «4:4:0») или по горизонтали («4:2:2»), или не выполняться вовсе («4:4:4»). В данном случае будет использоваться стандартное прореживание.

Программная реализация:

```
def subsampling(array_color_channel):
    for i in range(0, len(array_color_channel), 2):
        for j in range(0, len(array_color_channel[0]), 2):
            average_color = array_color_channel[i][j]
            average_color += array_color_channel[i + 1][j]
            average_color += array_color_channel[i][j + 1]
            average_color += array_color_channel[i + 1][j + 1]
            average_color = round(average_color / 4)
            array_color_channel[i][j] = average_color
            array_color_channel[i + 1][j] = average_color
            array_color_channel[i][j + 1] = average_color
            array_color_channel[i + 1][j + 1] = average_color
```

3. ДКП

Далее яркостный компонент Y и отвечающие за цвет компоненты Cb и Cr разбиваются на блоки 8×8 пикселей. Каждый такой блок подвергается дискретному косинусному преобразованию (ДКП). ДКП обычно используется в обработке сигналов и изображений как метод преобразования изображения из пространственной области в частотную область, таким образом что высокие частоты, т.е те значения, которые влияют на изображения больше по сравнению с другими в текущем блоке находятся в начале матрицы, а с низкими частотами наоборот. ДКП это обратимый процесс и по своей сути без потерь. В данной реализации оно было реализовано в матричной форме. Матрица ДКП имеет размерность 8×8 , а каждый её элемент имеет вид при $N=8$:

$$C(u, v) = \begin{cases} \sqrt{\frac{1}{N}} & u = 0, 0 \leq v \leq N - 1 \\ \sqrt{\frac{2}{N}} \cos \frac{(2v+1)u\pi}{2N} & 1 \leq u \leq N - 1, 0 \leq v \leq N - 1 \end{cases}$$

Тогда и матрица в общем случае имеет вид:

$$C(u, v) = \begin{bmatrix} \sqrt{\frac{1}{N}} & \sqrt{\frac{1}{N}} & \dots & \sqrt{\frac{1}{N}} \\ \sqrt{\frac{2}{N}} \cos \frac{\pi}{2N} & \sqrt{\frac{2}{N}} \cos \frac{3\pi}{2N} & \dots & \sqrt{\frac{2}{N}} \cos \frac{(2N-1)\pi}{2N} \\ \vdots & \vdots & \ddots & \vdots \\ \sqrt{\frac{2}{N}} \cos \frac{(N-1)\pi}{2N} & \sqrt{\frac{2}{N}} \cos \frac{3(N-1)\pi}{2N} & \dots & \sqrt{\frac{2}{N}} \cos \frac{(2N-1)(N-1)\pi}{2N} \end{bmatrix}$$

А ДКТ в матричной форме выглядит таким образом:

$$T = CFC^T$$

Программная реализация:

```
def discrete_cosine_transform(_pixels_arr) -> None:  
    return np.round(np.dot(matrix_of_dct, np.dot(_pixels_arr, transposed_matrix_of_dct))).astype(int)
```

```
matrix_of_dct = np.zeros((8, 8), dtype=np.float64)

# создание матрицы для дискретно-косинусного преобразования
for i in range(8):
    for j in range(8):
        if i == 0:
            matrix_of_dct[i][j] = (1 / (2 * np.sqrt(2)))
        elif i > 0:
            matrix_of_dct[i][j] = 0.5 * np.cos(((2*j + 1) * i * pi) / 16)
```


4. Квантование

Затем полученные коэффициенты ДКП квантуются, т. е. происходит поэлементное деление с матрицей квантования, которая задается коэффициентом сжатия.

А элемент матрицы квантования имеет вид:

$$Q_{ij} = 1 + (i + j) \cdot QualityFactor$$

где QualityFactor это коэффициент сжатия.

Программная реализация:

5.

```
def quantization(_pixel_arr):  
    return np.round(_pixel_arr / matrix_of_quantization).astype(int)  
  
# инициализация матрицы квантования  
def init_matrix_of_q(_quality_factor):  
    for i in range(8):  
        for j in range(8):  
            matrix_of_quantization[i][j] = 1 + (i * j) * _quality_factor  
    # print(matrix_of_quantization)
```

RLE- кодиро вание

и Таблица Хаффмана

После использования процесса квантования в каждом блоке можно заметить, что в матрице получается большое количество нулей, и если пройти по матрице «зигзагом» (Рис. 1), то можно получить большое количество подряд идущих нулей, которые с помощью RLE-кодирования удобно сворачивать.

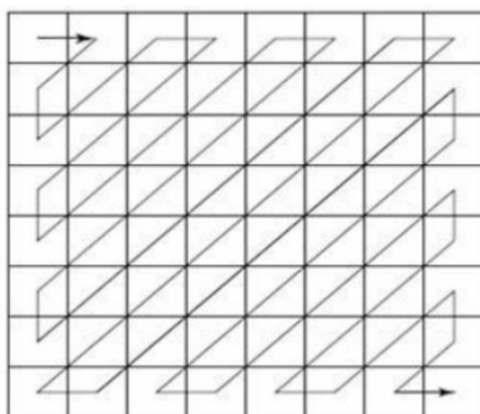


Рисунок 1

А RLE-кодирование это алгоритм заменяющий повторяющиеся символы (серии) на один символ и число его повторов. **Серией** называется последовательность, состоящая из нескольких одинаковых символов.

Программная реализация:




```
def block_to_RLE(pixel_arr):
    message = [str(int(pixel_arr[i // 8][i % 8])) for i in zigzag_way]
    encoded_string = []
    i = 0
    while (i <= len(message) - 1):
        count = 1
        ch = message[i]
        j = i
        while (j < len(message) - 1):
            if (message[j] == message[j + 1]):
                count = count + 1
                j = j + 1
            else:
                break
        encoded_string.append((count, int(ch)))
        i = j + 1
    return encoded_string
```

Посл
е
этого
необ
ходи
мо
для
кажд
ого
цвет

ового канала произвести кодировку Хаффмана, который по своему свойству кодирует заданный набор информации с определенным весом минимальным набором битов, что очень удобно в случае RLE-кодирования. Реализация дерева Хаффмана (Приложение №2).

Далее информация о размерах изображения, количества таблиц Хаффмана, их запись, и количество блоков записывается в файл. Декодирование происходит полностью в обратном порядке.

Результаты

До сжатия	После сжатия										
	 <div>  dct_and_quant_testforsamp... </div> <div>  Поделиться </div> <div> Сведения <table> <tr> <td>Тип</td><td>Файл "JPG"</td></tr> <tr> <td>Размер</td><td>5,28 КБ</td></tr> <tr> <td>Расположени...</td><td>C:\Пользователи\games\gits...</td></tr> <tr> <td>Дата изменен...</td><td>29.11.2024 10:03</td></tr> <tr> <td>Разрешение</td><td>304 x 168</td></tr> </table> </div>	Тип	Файл "JPG"	Размер	5,28 КБ	Расположени...	C:\Пользователи\games\gits...	Дата изменен...	29.11.2024 10:03	Разрешение	304 x 168
Тип	Файл "JPG"										
Размер	5,28 КБ										
Расположени...	C:\Пользователи\games\gits...										
Дата изменен...	29.11.2024 10:03										
Разрешение	304 x 168										



dct_and_quant_sample.bmp...



Поделиться

Сведения

Тип	Файл "JPG"
Размер	211 КБ
Расположени...	C:\Пользователи\games\gits...
Дата изменен...	05.12.2024 22:11
Разрешение	1280 x 856



dct_and_quant_testfile.jpg



Поделиться

Сведения

Тип	Файл "JPG"
Размер	104 КБ
Расположени...	C:\Пользователи\games\gits...
Дата изменен...	29.11.2024 10:15
Разрешение	3136 x 1336

Заключение

Поставленная цель в реализации JPEG была успешно выполнена. Проведенные тесты показывают, что удалось добиться уменьшения размера изображения.

Список использованных источников

<https://en.wikipedia.org/wiki/JPEG>

<https://ru.wikipedia.org/wiki/YCbCr>

<https://cs.stanford.edu/people/eroberts/courses/soco/projects/data-compression/lossy/jpeg/dct.htm>

[https://en.wikipedia.org/wiki/Discrete cosine transform](https://en.wikipedia.org/wiki/Discrete_cosine_transform)

<https://numpy.org/doc/>

<https://pillow.readthedocs.io/en/stable/>

Приложения

Листинг

```
import numpy as np
from pprint import pprint
from PIL import Image
from numpy import linalg
from typing import Tuple, List
from math import pi
from my_consts import *
import huffman
import itertools
import pickle

# инициализация матрицы квантования
def init_matrix_of_q(_quality_factor):
    for i in range(8):
        for j in range(8):
            matrix_of_quantization[i][j] = 1 + (i * j) * _quality_factor
    # print(matrix_of_quantization)

test_eye = np.eye(5)

matrix_of_dct = np.zeros((8, 8), dtype=np.float64)

# создание матрицы для дискретно-косинусного преобразования
for i in range(8):
    for j in range(8):
        if i == 0:
            matrix_of_dct[i][j] = (1 / (2 * np.sqrt(2)))
        elif i > 0:
            matrix_of_dct[i][j] = 0.5 * np.cos(((2*j + 1) * i * pi) / 16)

transposed_matrix_of_dct = matrix_of_dct.T

inv_matrix_of_dct = linalg.inv(matrix_of_dct)

inv_transposed_matrix_of_dct = linalg.inv(transposed_matrix_of_dct)

"""
Фикс изображения для которых нельзя построить матрицу
"""
def fix_image_size_for_jpeg(_pixels_arr, size):
    need_cols = (8 - (size[0] % 8)) if size[0] % 8 else 0
    need_rows = (8 - (size[1] % 8)) if size[1] % 8 else 0
    # print(need_cols, need_rows)
    # print(size)
    if not need_cols and not need_rows:
        return _pixels_arr
    new_pixels_arr = np.zeros(
        (size[0] + need_cols, size[1] + need_rows, 3), dtype=int)
    # for i in range(need_rows):
    for i in range(size[0] + need_cols):
        for j in range(size[1] + need_rows):
            if i < size[0] and j < size[1]:
                new_pixels_arr[i][j] = (
                    tuple(_pixels_arr[i][j]))
            elif i < size[0] and j >= size[1]:
                new_pixels_arr[i][j] = (
```



```

        tuple(new_pixels_arr[i][size[1] - 1]))
    else:
        new_pixels_arr[i][j] = (tuple(
            new_pixels_arr[size[0] - 1][j]))
    return new_pixels_arr

# * JPEG/JFIF YCbCr conversions

#      Y  = R *  0.29900 + G *  0.58700 + B *  0.11400
#      Cb = R * -0.16874 + G * -0.33126 + B *  0.50000 + 128
#      Cr = R *  0.50000 + G * -0.41869 + B * -0.08131 + 128

# def convert_pixel_rgb2ycbcr(_pixel_rgb: Tuple[int, int, int]) -> Tuple[int,
# int, int]:
#     # return np.dot(_pixel_rgb, convertation_matrix_rgb2ycbcr) +
#     convertation_vector_rgb2ycbcr
#     return (int(np.float64(0.29900) * _pixel_rgb[0] + np.float64(0.58700) *
# _pixel_rgb[1] + np.float64(0.11400) * _pixel_rgb[2]),
#             int(128 + np.float64(-0.16874) *
#             _pixel_rgb[0] - np.float64(0.33126) * _pixel_rgb[1] +
# np.float64(0.50000) * _pixel_rgb[2]),
#             int(128 + np.float64(0.50000) * _pixel_rgb[0] -
# np.float64(0.41869) * _pixel_rgb[1] - np.float64(0.08131) * _pixel_rgb[2]))

# def convert_pixel_ycbcr2rgb(_pixel_ycbcr: Tuple[int, int, int]) -> Tuple[int,
# int, int]:
#     # return np.dot((_pixel_ycbcr - convertation_vector_rgb2ycbcr),
#     convertation_matrix_ycbcr2rgb)
#     return (
#         round(_pixel_ycbcr[0] + 1.402 * (_pixel_ycbcr[2] - 128)),
#         round(_pixel_ycbcr[0] - 0.344136 *
#             (_pixel_ycbcr[1] - 128) - 0.714136 * (_pixel_ycbcr[2] - 128)),
#         round(_pixel_ycbcr[0] + 1.772 * (_pixel_ycbcr[1] - 128))
#     )

def matrix_convert_pixel_rgb2ycbcr(_pixel_rgb: Tuple[int, int, int]) ->
Tuple[int, int, int]:
    return np.round((np.dot(_pixel_rgb, convertation_matrix_rgb2ycbcr) +
convertation_vector)[0]).astype(int)

def matrix_convert_pixel_ycbcr2rgb(_pixel_ycbcr: Tuple[int, int, int]) ->
Tuple[int, int, int]:
    return np.round(np.dot((_pixel_ycbcr - convertation_vector),
convertation_matrix_ycbcr2rgb)[0]).astype(int)

def array_rgb2ycbcr(pixel_arr):
    ycbcr_image_plot = np.zeros(pixel_arr.shape)
    for i in range(pixel_arr.shape[0]):
        for j in range(pixel_arr.shape[1]):
            ycbcr_image_plot[i][j] = matrix_convert_pixel_rgb2ycbcr(
                pixel_arr[i][j])
    return ycbcr_image_plot

def array_ycbcr2rgb(pixel_arr):
    rgb_image_plot = np.zeros(pixel_arr.shape)

```

```

for i in range(pixel_arr.shape[0]):
    for j in range(pixel_arr.shape[1]):
        rgb_image_plot[i][j] = fix_pixel(matrix_convert_pixel_ycbcr2rgb(
            pixel_arr[i][j]))
return rgb_image_plot

def subsampling(array_color_channel):
    for i in range(0, len(array_color_channel), 2):
        for j in range(0, len(array_color_channel[0]), 2):
            average_color = array_color_channel[i][j]
            average_color += array_color_channel[i + 1][j]
            average_color += array_color_channel[i][j + 1]
            average_color += array_color_channel[i + 1][j + 1]
            average_color = round(average_color / 4)
            array_color_channel[i][j] = average_color
            array_color_channel[i + 1][j] = average_color
            array_color_channel[i][j + 1] = average_color
            array_color_channel[i + 1][j + 1] = average_color

def discrete_cosine_transform(_pixels_arr) -> None:
    return np.round(np.dot(matrix_of_dct, np.dot(_pixels_arr,
transposed_matrix_of_dct))).astype(int)

def inv_discrete_cosine_transform(_pixel_arr):
    return np.round(np.dot(np.dot(transposed_matrix_of_dct, _pixel_arr),
matrix_of_dct)).astype(int)

def quantization(_pixel_arr):
    return np.round(_pixel_arr / matrix_of_quantization).astype(int)

def inv_quantization(_pixel_arr):
    return np.round(_pixel_arr * matrix_of_quantization).astype(int)

def block_to_RLE(_pixel_arr):
    message = [str(int(_pixel_arr[i // 8][i % 8])) for i in zigzag_way]
    encoded_string = []
    i = 0
    while (i <= len(message) - 1):
        count = 1
        ch = message[i]
        j = i
        while (j < len(message) - 1):
            if (message[j] == message[j + 1]):
                count = count + 1
                j = j + 1
            else:
                break
        encoded_string.append((count, int(ch)))
        i = j + 1
    return encoded_string

def RLE_to_block(_rle_array):
    rle_elements = []
    for i, j in _rle_array:

```

```

        for k in range(i):
            rle_elements.append(j)
matrix_peace = np.zeros((8, 8), dtype=int)
for i in range(64):
    matrix_peace[zigzag_way[i] // 8][zigzag_way[i] % 8] = rle_elements[i]
return matrix_peace

def channel_to_RLE(_rle_arr):
    rle_res = []
    for i in range(_rle_arr.shape[0]):
        # print(_rle_arr[i])
        rle_res.append(block_to_RLE(_rle_arr[i]))
    return rle_res

def RLE_to_channel(_rle_arr):
    channel = []
    for i in range(len(_rle_arr)):
        # print(_rle_arr[i])
        channel.append(RLE_to_block(_rle_arr[i]))
    return np.array(channel)

def divide_on_blocks(_pixel_array):
    blocks = []
    img_w, img_h = _pixel_array.shape[0], _pixel_array.shape[1]
    # print(img_w, img_h)
    for i in range(0, img_w, 8):
        for j in range(0, img_h, 8):
            blocks.append(_pixel_array[i:i+8, j:j+8])
    return np.array(blocks)

def combine_blocks(blocks, _shape):
    combined_channel = np.zeros(_shape, dtype=int)
    block_ind = 0
    for i in range(_shape[0] // 8):
        for j in range(_shape[1] // 8):
            for k in range(8):
                for h in range(8):
                    combined_channel[i * 8 + k][j * 8 + h] =
blocks[block_ind][k][h]
                    block_ind += 1
    return combined_channel

def save_image_fromarray(parray, filename):
    image_plot = np.array(parray, dtype=np.uint8)
    new_image = Image.fromarray(image_plot)
    new_image.save(filename)

def fix_pixel(pixel: Tuple[int, int, int]) -> Tuple[int, int, int]:
    return (max(min(pixel[0], 255), 0), max(min(pixel[1], 255), 0),
max(min(pixel[2], 255), 0))

def combine_color_channels(first: np.array, second, third):
    image_plot = np.zeros((first.shape[0], first.shape[1], 3))
    for i in range(first.shape[0]):
        for j in range(first.shape[1]):
            image_plot[i][j] = np.array(

```

```

        (first[i][j], second[i][j], third[i][j]), dtype=np.uint8)
    return image_plot

def int_to_bin_str(number, cnt_null):
    return bin(abs(number))[2:].zfill(cnt_null)

def block_encode_huffman(block):
    """
    .myjpeg block encode
    table size: 4 bits
        char: 13 bits (1rst bit for minus), 12 bits for value result [-4096,
4096], value 4 bits
    count_of_pair: 4 bits
    huffmanRLE
    """
    count = []
    value = []
    for rle_pair in block:
        count.append(rle_pair[0])
        value.append(rle_pair[1])
    root = huffman.build_huffman_tree(value, count)
    huffman_codes = huffman.generate_huffman_codes(root)
    table_size = len(huffman_codes.items())
    encoded_block = f"{int_to_bin_str(table_size, 4)}"
    print(block)
    for char, code in sorted(huffman_codes.items(), key=lambda x: x[1],
reverse=True):
        print(f"Character: {char}, Code: {code}")
        have_minus = 0
        if char < 0:
            have_minus = 1
        encoded_block +=
f"{huffman_code_length}{have_minus}{int_to_bin_str(char, 12)}{code}"
        print(char, int_to_bin_str(char, 12))
    # for element in block:
    #     encoded_block += str(huffman_codes[element[1]]) * element[0]
    print(encoded_block)

def jpeg_encode(filename, quantion_coeff=10):
    tools.init_matrix_of_q(quantion_coeff)

    # print('Матрица квантования', tools.matrix_of_quantization, sep='\n')
    current_image = Image.open(filename).convert("RGB")
    rgb_image_plot = np.asarray(current_image)
    current_image.close()
    rgb_image_plot = tools.fix_image_size_for_jpeg(
        rgb_image_plot, rgb_image_plot.shape)
    ycbcr_image_plot = tools.array_rgb2ycbcr(rgb_image_plot)
    Y = ycbcr_image_plot[:, :, 0]
    Cb = ycbcr_image_plot[:, :, 1]
    Cr = ycbcr_image_plot[:, :, 2]
    tools.subsampling(Cb)
    tools.subsampling(Cr)

    divided_Y = tools.divide_on_blocks(Y)
    divided_Cb = tools.divide_on_blocks(Cb)
    divided_Cr = tools.divide_on_blocks(Cr)

    for i in range(len(divided_Y)):
        divided_Y[i] = (
            tools.discrete_cosine_transform(divided_Y[i]))

```

```

        divided_Cb[i] = (
            tools.discrete_cosine_transform(divided_Cb[i]))
        divided_Cr[i] = (
            tools.discrete_cosine_transform(divided_Cr[i]))

    for i in range(len(divided_Y)):
        divided_Y[i] = tools.quantization(
            (divided_Y[i]))
        divided_Cb[i] = tools.quantization(
            (divided_Cb[i]))
        divided_Cr[i] = tools.quantization(
            (divided_Cr[i]))

    # print(f"MAX_Y{np.max(divided_Y)}")
    # print(f"MAX_Y{np.max(divided_Cb)}")
    # print(f"MAX_Y{np.max(divided_Cr)}")

    Y_RLE = tools.channel_to_RLE(divided_Y)
    Cb_RLE = tools.channel_to_RLE(divided_Cb)
    Cr_RLE = tools.channel_to_RLE(divided_Cr)

    # tools.block_encode_huffman(Y_RLE[0])

    # with open('myjpeg.myjpeg', 'wb') as f:
    #     pickle.dump([Y_RLE, Cb_RLE, Cr_RLE], f)

    divided_Y = tools.RLE_to_channel(Y_RLE)
    divided_Cb = tools.RLE_to_channel(Cb_RLE)
    divided_Cr = tools.RLE_to_channel(Cr_RLE)
    # print(divided_Y)

    for i in range(len(divided_Y)):
        divided_Y[i] = (
            tools.inv_quantization(divided_Y[i]))
        divided_Cb[i] = (
            tools.inv_quantization(divided_Cb[i]))
        divided_Cr[i] = (
            tools.inv_quantization(divided_Cr[i]))

    for i in range(len(divided_Y)):
        divided_Y[i] = tools.inv_discrete_cosine_transform(
            (divided_Y[i]))
        divided_Cb[i] = tools.inv_discrete_cosine_transform(
            (divided_Cb[i]))
        divided_Cr[i] = tools.inv_discrete_cosine_transform(
            (divided_Cr[i]))

    # print(divided_Y.shape)

    needed_shape = (rgb_image_plot.shape[0], rgb_image_plot.shape[1])
    combined_Y = tools.combine_blocks(divided_Y, needed_shape)
    combined_Cb = tools.combine_blocks(divided_Cb, needed_shape)
    combined_Cr = tools.combine_blocks(divided_Cr, needed_shape)

    # combined_Y = Y
    # combined_Cb = Cb
    # combined_Cr = Cr

    print("Размеры каналов изображения после ДКТ", combined_Y.shape,
sep='\n')
    print(combined_Cb.shape)

```

```
print(combined_Cr.shape)

new_ycbcr_plot = tools.combine_color_channels(
    combined_Y, combined_Cb, combined_Cr)
new_rgb_plot = tools.array_ycbcr2rgb(new_ycbcr_plot)
tools.save_image_fromarray(
    new_rgb_plot, f"jpeg/src/imgs/dct_and_quant_{filename}.jpg")
print("OK")
```