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:

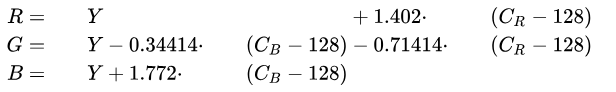
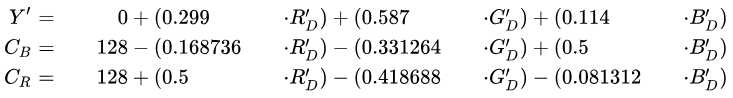
1. Перевод изображения из пространства RGB в YcbCr
2. Прореживание
3. Дискретно-косинусное преобразование
4. Квантование
5. Запись в файл с помощью RLE-кодирования и таблицы Хаффмана

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”

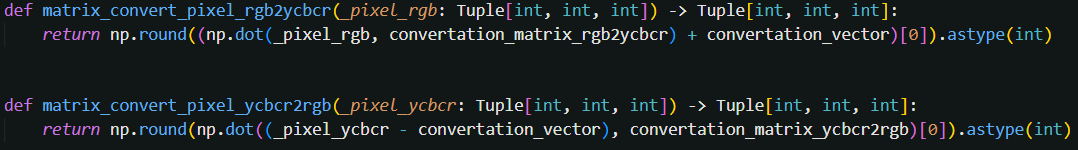
**Основная часть**

1. Перевод изображения из цветового пространства RGB в YcbCr

При сжатии изображение преобразуется из цветового пространства RGB в [YCbCr](https://ru.wikipedia.org/wiki/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:



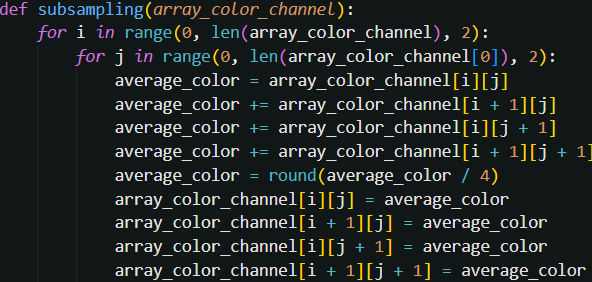
Программная реализация перевода осуществляется в матричной форме, причем из-за ошибок с числами с плавающей точкой необходимо использовать округление.



2. Прореживание

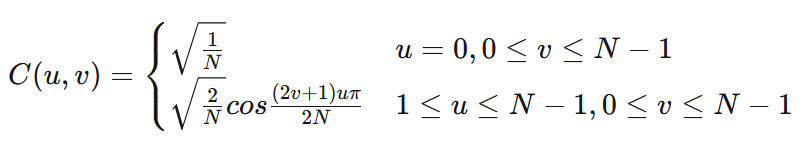
После преобразования RGB->YCbCr для каналов изображения Cb и Cr, отвечающих за цвет, может выполняться «прореживание» (subsampling), которое заключается в том, что каждому блоку из 4 пикселей (2х2) яркостного канала Y ставятся в соответствие усреднённые значения Cb и Cr (схема прореживания «4:2:0»). При этом для каждого блока 2х2 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.

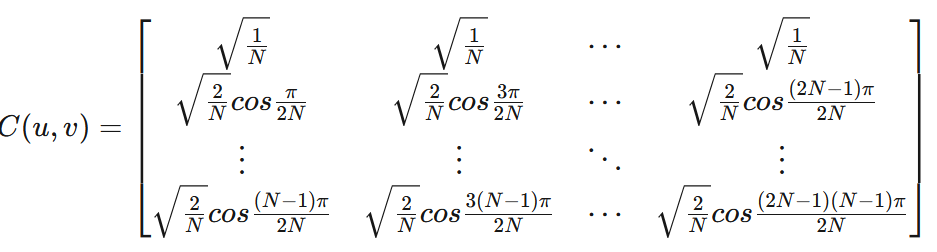
Программная реализация:



3. ДКП

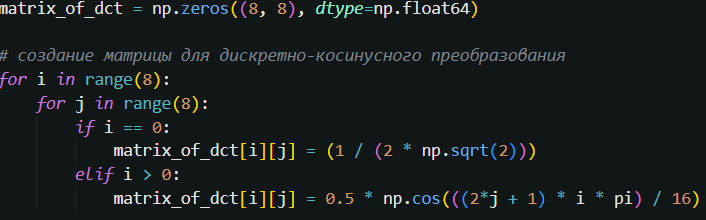
Далее яркостный компонент Y и отвечающие за цвет компоненты Cb и Cr разбиваются на блоки 8х8 пикселей. Каждый такой блок подвергается [дискретному косинусному преобразованию (ДКП)](https://ru.wikipedia.org/wiki/Дискретное_косинусное_преобразование). ДКП обычно используется в обработке сигналов и изображений 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 8х8, а каждый её элемент имеет вид при N=8:

Тогда и матрица в общем случае имеет вид:

А ДКТ в матричной форме выглядит таким образом:

  
  
Программная реализация:





4. Квантование

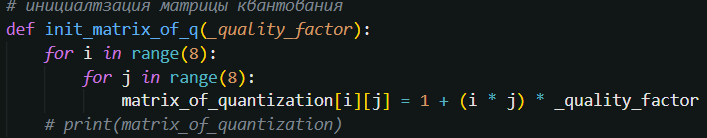
Затем полученные коэффициенты ДКП квантуются, т. е. происходит поэлементное деление с матрицей квантования, которая задается коэффициентом сжатия.

А элемент матрицы квантования имеет вид:



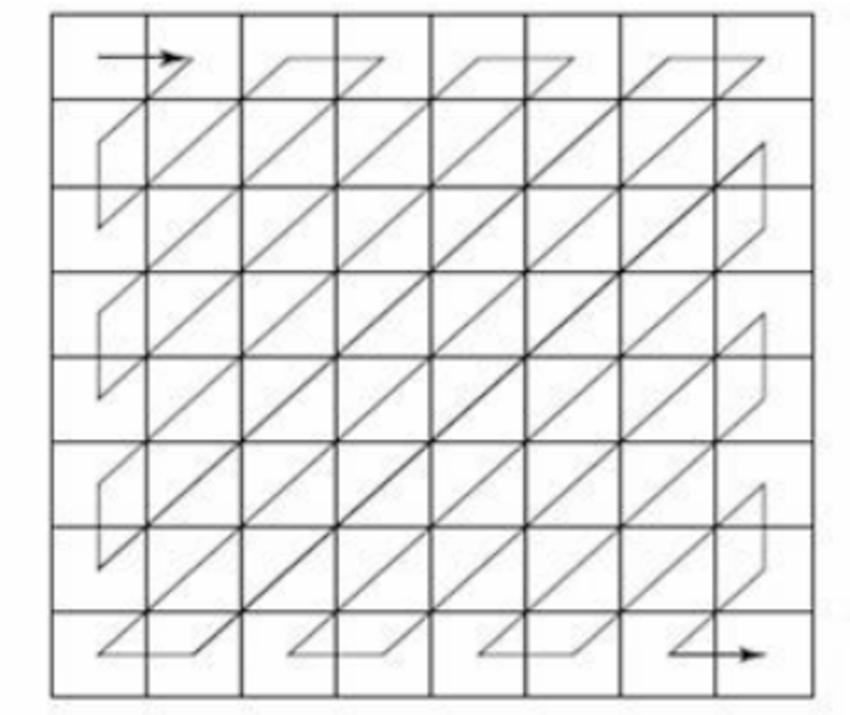
где QualityFactor это коэффициент сжатия.

Программная реализация:

****

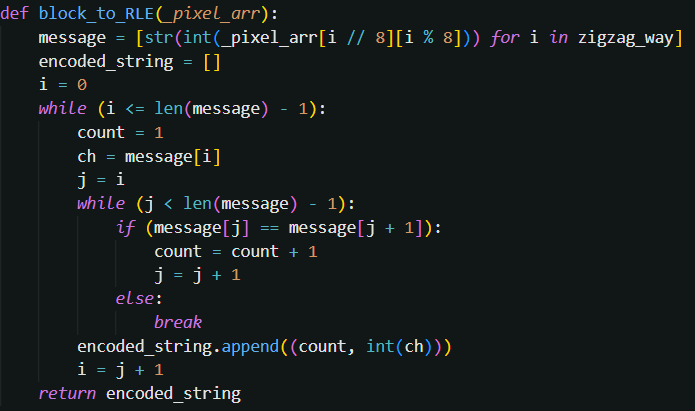
5. RLE-кодирование и Таблица Хаффмана

После использования процесса квантования в каждом блоке можно заметить, что в матрице получается большое количество нулей, и если пройтись 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:

  
После этого необходимо для каждого цветового 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 |
|  |  |
|  |  |
|  |  |

**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://en.wikipedia.org/wiki/JPEG)

[**https://ru.wikipedia.org/wiki/YCbCr**](https://ru.wikipedia.org/wiki/YCbCr)

[**https://cs.stanford.edu/people/eroberts/courses/soco/projects/data-compression/lossy/jpeg/dct.htm**](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://numpy.org/doc/)

[**https://pillow.readthedocs.io/en/stable/**](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

# инициалтзация 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")