**МИНОБРНАУКИ РОССИИ**
**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**
**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**
**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**
**Кафедра систем автоматизированного проектирования**

# ОТЧЕТ

**по лабораторной работе №2**
**по дисциплине «Алгоритмы и структуры данных»**
**Тема: «Реализация алгоритма JPEG»**

Студент гр. 3352 Гультяев А.С.

Преподаватель Пестерев Д.О.

Санкт-Петербург
2025

**Введение**

**Цель:** реализация алгоритма $JPEG$, а также его составных частей, таких как: AC/DC кодирование, квантование матриц и так далее.

**Задачи:**

a) Реализовать переход цветового пространства из $RGB$ в $YC_BC_R$ и обратно.
b) Реализовать downsampling матрицы цветового канала.
c) Осуществить разбиение изображения на блоки размерами $N \times N$, при этом дополнив неполные блоки некоторым значением.
d) Изучить и реализовать прямое и обратное DCT-II 2D для блока размером $N \times N$.
e) Осуществить изменение матрицы квантования в зависимости от уровня сжатия.
f) Зигзаг обход матрицы $N \times N$.
g) Изучить и реализовать разностное DC, переменное кодирование AC и DC коэффициентов.
h) Реализовать RLE кодирование AC коэффициентов.
i) Кодирование разностей DC коэффициентов и последовательностей Run/Size по таблице кодов Хаффмана и упаковки результата в байтовую строку.
j) Собрать итоговый алгоритм кодирования и декодирования $JPEG$.
k) Построить графики зависимости размеров зависимости размера сжатого файла от коэффициента качества сжатия с шагом 2.

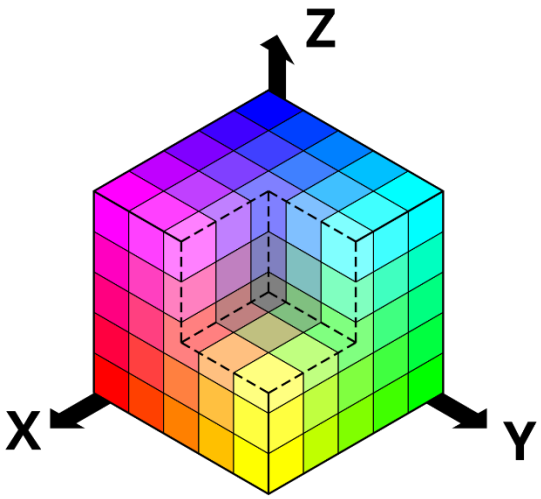
## 1. Теоретическая часть

### 1.1. $RGB$:

$RGB$ – аддитивная цветовая модель, которая описывает способ кодирования цвета для цветовоспроизведения с помощью трёх цветов ($R$ – red (красный), $G$ – green (зеленый), $B$ – blue (синий)).

В самой простой реализации $RGB$ можно представить в виде трехмерной системы координат, где каждая из каналов/осей $(R, G, B)$ начинается с 0 в общей точки и заканчивается на значении 255 (рис. 1.1). Такое представление выбрано вследствие того, что один байт принимает значение от 0 до 255. В изображениях же каждый из каналов накладываются друг на друга, и значения для каждого отдельного из каналов означают интенсивность цвета, то есть 255 – цвет максимально насыщенный, 0 – цвет прозрачный.

Рисунок 1.1 – Простейшее представление $RGB$




В подавляющем большинстве, будь то компьютеры, телефоны или любая другая техника, в которой используется графическое отображение, используется немного иная запись. Так, например, для более точной передачи используется шестнадцатеричное отображение формата #RRGGBB, где каждое из значений может принимать одно из цифр шестнадцатеричной системы – от 0 до F. В такой реализации первые два значения означают

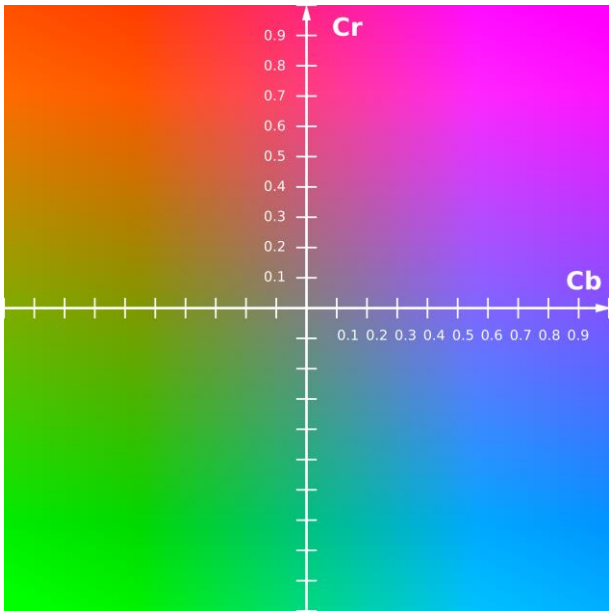
интенсивность для красного, следующие два – интенсивность для зеленого, и последние два – интенсивность для красного цветов.

1.2. $YC_BC_R$:

$YC_BC_R$ – семейство цветовых пространств, которые используются для передачи цветных изображений в компонентном виде и цифровой фотографии. В таком виде $Y$ – компонента яркости, $C_B$ и $C_R$ – синяя и красная цветоразностные компоненты.

В отличии от $RGB$, $YC_BC_R$ проще представить в виде плоскости так, как это приведено на рисунке 1.2. Однако, на данном рисунке не приведена ось $Y$, которая отвечает за яркость.

Рисунок 1.2 – Простейшее представление $YC_BC_R$




Такой формат изображений делается, поскольку яркость $Y$ воспринимается глазом сильнее, чем другая цветовая информация $C_B$ и $C_R$.

1.3. Переходы цветовых пространств:

Для того, что бы перейти из $RGB$ в $YC_BC_R$ необходимо использовать следующие заранее используемые формулы, в которых происходит деление на 255 для нормализации данных. Нормализация – процесс приведения данных к общему виду. Сами формулы приведены ниже:

$$Y = 16 + \frac{65.481 \cdot R + 128.553 \cdot G + 24.996 \cdot B}{255}$$

$$C_B = 128 + \frac{-37.797 \cdot R - 74.203 \cdot G + 112 \cdot B}{255}$$

$$C_R = 128 + \frac{112 \cdot R - 93.786 \cdot G - 18.214 \cdot B}{255}$$

Для перехода из $YC_BC_R$ в $RGB$ используются обратные формулы:

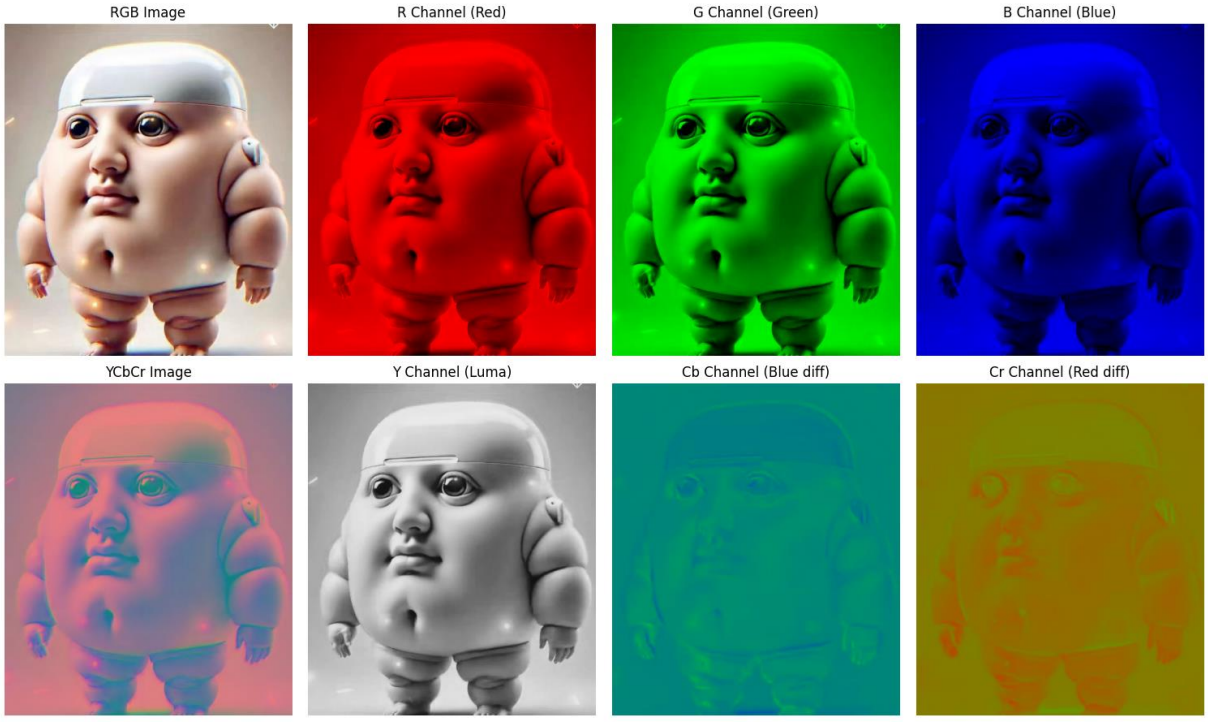
$$R = Y + 1.402 \cdot (C_R - 128)$$

$$G = Y - 0.34414 \cdot (C_B - 128) - 0.71414 \cdot (C_R - 128)$$

$$B = Y + 1.772 \cdot (C_B - 128)$$

Для наглядности, на рисунке 1.3 приведен пример изображения в $RGB, YC_BC_R$, а также отдельно разбитые каналы для каждого из цветовых отображений.

Рисунок 1.3 – Примеры изображений

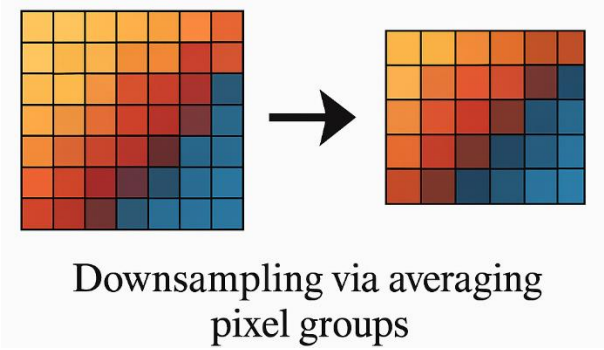



## 1.4. Downsampling и Upsampling:

Downsampling – это процесс уменьшения размера изображения в $N$ раз. Данное действие происходит путем усреднения групп пикселей. Так,

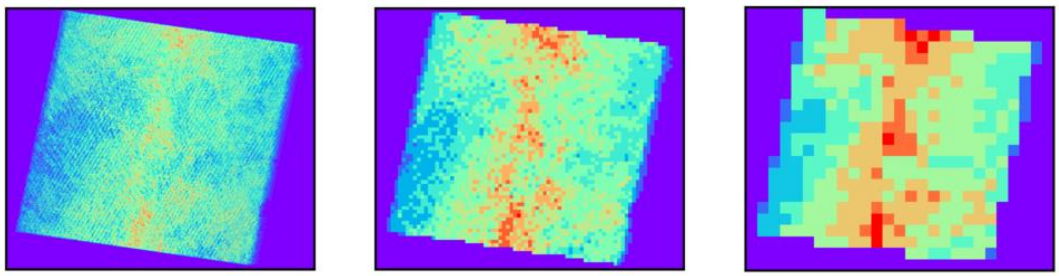
например, для $N = 2$ и изображения размерами $10 \times 10$ будет преобразовано так, как это показано на изображении 1.4.

Рисунок 1.4 – Downsampling преобразование



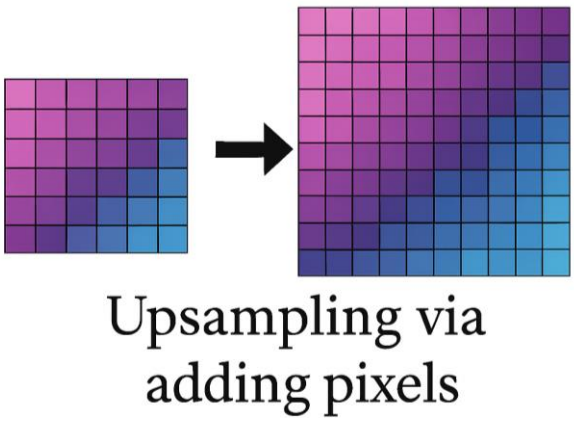

С помощью Downsampling изображение теряет незначительную часть своей информации, но оставляет большую часть понятной. Пример изображения до и после преобразования приведен на рисунке 1.5.

Рисунок 1.5 – Пример downsampling преобразования



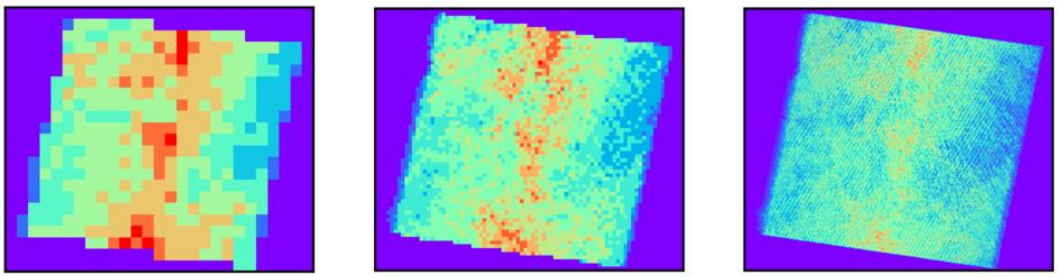
Upsampling – это процесс увеличения размера изображения в $N$ раз. Данное действие происходит путем добавления новых пикселей на основе существующих. Таким образом, находится крайний пиксель и расширяется в квадрат на основе этого. Так, например, для $N = 2$ и изображения размерами $10 \times 10$ будет преобразовано так, как это показано на изображении 1.6.

Рисунок 1.6 – Upsampling преобразование




С помощью Upsampling изображение восстанавливает часть своей ранее сжатой с помощью downsampling информации, но оставляет большую часть понятной. Пример изображения до и после преобразования приведен на рисунке 1.7.

Рисунок 1.7 – Изображение до и после upsampling преобразования



1.5. DCT-II 2D и iDCT-II 2D:

DCT-II 2D (далее просто DCT) – это математическое преобразование, которое используется для преобразования сигналов (звука, изображений) между временной, пространственной и частотными областями.

Сам DCT преобразует пиксели изображения в набор частотных коэффициентов. Эти коэффициенты показывают какие шаблоны присутствуют в сигнале.

Таким образом, DCT выделяет важные и неважные компоненты сигнала, так, например, в изображениях основные детали конструкции кодируются большими коэффициентами, а мелкие шумы – малыми. При сжатии малозначимые коэффициенты отбрасываются, что уменьшает размер файла.

iDCT-II 2D (далее просто iDCT) – это математическое преобразование, которое фактически выполняет обратное DCT действие. То есть iDCT восстанавливает исходные данные из частотных коэффициентов.

Формулы для DCT и iDCT приведены ниже:

$$F(u,v) = \frac{2}{N} C(u)C(v) \sum_{x=0}^{N} \sum_{y=0}^{N} f(x,y) * \cos\left(\frac{(2x+1)u\pi}{2 \cdot N}\right) cos\left(\frac{(2x+1)v\pi}{2 \cdot N}\right)$$

$$f(x,y) = \frac{2}{N} \sum_{u=0}^{N} C(u) \sum_{v=0}^{N} C(v)F(u,v) * \cos\left(\frac{(2x+1)u\pi}{2 \cdot N}\right) cos\left(\frac{(2x+1)v\pi}{2 \cdot N}\right)$$

где:

- $f(x,y) -$ значение в позиции $(x,y)$;
- $F(u,v) -$ коэффициент DCT в частотной области в позиции $(u,v)$;
- $C(k) = \begin{cases} \frac{1}{\sqrt{2}}, \text{ если } k = 0 \\ 1, \text{ если } k \neq 0 \end{cases}$

1.6. Матрица квантования:

Матрица квантования – матрица с некоторыми значениями, размер которой совпадает с размером блока $8 \times 8$. При этом матрица квантования используется для управления степенью сжатия и выполняется сразу же после DCT.

Для $YC_BC_R$ существует две основных матрицы квантования, выведенные совершенно нетривиальным образом:

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

$$Q_C = \begin{pmatrix} 17 & 18 & 24 & 47 & 99 & 99 & 99 & 99 \\ 18 & 21 & 26 & 66 & 99 & 99 & 99 & 99 \\ 24 & 26 & 56 & 99 & 99 & 99 & 99 & 99 \\ 47 & 66 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \end{pmatrix}$$

Для квантования блока изображения необходимо поделить каждый коэффициент блока на значение матрицы $Q$ по тем же индексам. Формула выглядит следующим образом: $Q_{ij} = round\left(\frac{D_{ij}}{Q_{table}[i,j]}\right)$.

Деквантование же происходит обратным образом, путем перемножения матрицы квантования на значение восстановленных из файла данных $Q$. Формально формула выглядит так: $D_{ij} = Q_{ij} \times Q_{table}[i,j]$.

Также, матрицы квантования обладают еще одной интересной характеристикой – увеличением или уменьшением качества изображения. Данный параметр $quality$ изменяет матрицу квантования по двум случаям:

$$\begin{cases} scale = \dfrac{5000}{quality}, & quality < 50 \\ scale = 200 - 2 \cdot quality, & quality \geq 50 \end{cases}$$

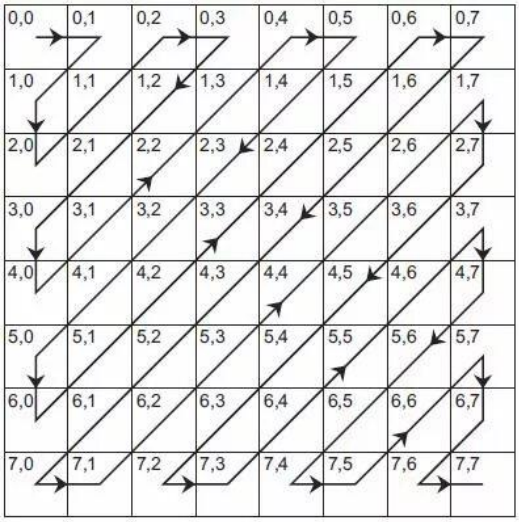
Далее масштабирование матрицы выполняется по формуле, где нулевые элементы заменяются на единицу:

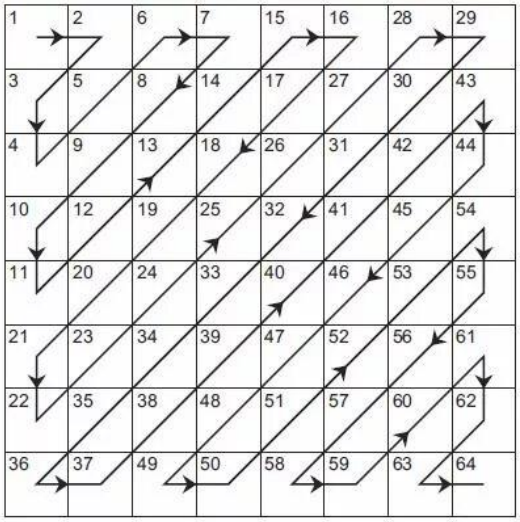
$$Q_{scaled} = \frac{Q \times scale + 50}{100}.$$

1.7. Зигзаг обход:

Зигзаг обход матрицы – это обход матрицы, который начинается с левого верхнего угла, и заканчивается на правом нижнем углу матрицы. Такой подход реализует преобразование матрицы в вектор-строку. Сам процесс прохода методом зигзаг по матрице приведен на рисунке 1.8.

Рисунок 1.8 – Зигзаг обход матрицы




Обратный зигзаг обход выполняется тривиальным образом, необходимо лишь идти в обратном порядке.

### 1.8. Разностное DC кодирование

Разностное DC кодирование – это метод сжатия, применяемый только к DC-элементам блоков. Вместо записи абсолютных DC значений, сохраняется разница между текущим DC-коэффициентом и следующим. DC-коэффициент – это первый элемент в матрице.

Сам алгоритм для каждого блока вычисляет разницу между его DC-коэффициентом и DC-коэффициентом предыдущего блока того же канала:

$$\Delta DC = DC_{current} - DC_{prev}.$$

Далее, в сжатый поток записывается не сам DC-коэффициент, а его разность $\Delta DC$. Первый блок всегда кодируется так, как есть – без изменений.

### 1.9. Переменное AC и DC кодирование

AC/DC-кодирование – гибридный метод сжатия для $JPEG$, который комбинирует разностное DC-кодирование и кодирование для AC-коэффициентов. Наименование «переменное» означает, что метод адаптируется в зависимости от того, какой коэффициент подается на вход.

Поскольку разностное DC-кодирование было рассмотрено в прошлом пункте, то далее будет рассмотрено лишь DC и AC-кодирование, а также его входящие, такие как RLE и Huffman кодирование.

DC-коэффициенты после разностного кодирования определяют с помощью Хаффман кодирования по категории разности (см. пункт Huffman кодирование)

AC-коэффициенты же кодируются в два этапа. Первый из них – кодирование с помощью RLE (см. пункт RLE кодирование), а далее применяется Huffman кодирование так же, как и для DC-коэффициентов.

### 1.10. RLE кодирование

RLE используется для кодирования последовательностей нулей. Вместо хранения каждого нулевого значения отдельно, RLE записывает пары вида $(len, cat)$, где $len$ – количество подряд идущих нулей перед текущем ненулевым коэффициентом, $cat$ – количество бит, необходимых для кодирования абсолютного ненулевого значения, и вычисляется по следующей формуле:

$$floor(\log_2|x| + 1).$$

### 1.11. Huffman кодирование

После применения RLE, полученные пары $(len, cat)$ подвергаются кодированию Хаффмана. Частотный словарь этих пар строится для всей последовательности и на его основе генерируется оптимальное дерево Хаффмана. Каждая пара затем заменяется соответствующим кодом Хаффмана.

Следом, для каждого ненулевого коэффициента записывается его остаточная часть – двоичное представление значения с длиной, соответствующей его категории. Отрицательные значения кодируются методом инвертирования всех битов.

### 1.12. DC/AC-декодирование

Для декодирования AC коэффициентов с файла читается частотный словарь и на его основе восстанавливается дерево Хаффмана. Далее последовательность последовательности битов разбираются, извлекая пары

RLE, где нулевые значения восстанавливаются, а ненулевые декодируются. И в завершении производится восстановление полной последовательности.

DC-декодирование же тривиально, для этого необходимо найти категорию, определить по коду Хаффмана ее значение, а затем прочитать необходимое количество битов, которые содержат закодированную разность. Эта разность восстанавливается с учетом знака. После получения всех разностей происходит поэлементное накопление, а что бы восстановить абсолютные DC-значения для каждого блока используется формула:

$$DC_i = DC_{i-1} + \Delta DC_i,$$

где $\Delta DC_i$ – декодированная разность между текущим и предыдущим DC-коэффициентом.

### 1.13. JPEG-кодировщик

JPEG – метод кодирования и декодирования изображений, который основан на DCT, квантовании, AC/DC-кодирования. Последовательность действий следующая:

a) Преобразование цветового пространства

Для начала исходное изображение переводится из одного из пространств ($RGB, GrayScale, BW$) в $YC_BC_R$. Данный этап нужен для того, что бы разделить изображение на 3 канала, описанных в пункте про $YC_BC_R$.

b) Субдискретизация цветности

Каналы $C_B$ и $C_R$ подвергаются субдискретизации (downsampling) с коэффициентом 4:2:0, что уменьшает разрешение цветовой составляющей вдвое по вертикали и горизонтали, сохраняя при этом детализацию яркости.

c) Блочное разбиение

Все каналы ($Y, C_B, C_R$) разбиваются на блоки размерами $8 \times 8$, так как DCT далее будет применятся к каждому отдельному блоку.

d) Дискретное косинусное преобразование (DCT)

К каждому из блоков применяется DCT, которая преобразует значения пикселей в частотное представление, что позволяет сконцентрировать энергию изображения в левом верхнем углу.

e) Квантование

Полученные DCT коэффициенты делятся на соответствующие значения матрицы квантования в зависимости от качества. На данном шаге происходит основная потеря данных из изображения.

f) Зигзаг чтение

Квантованные блоки преобразуются в одномерный вектор-строку в порядке зигзагообразного обхода. Это группирует нулевые коэффициенты, подготавливая данные для RLE.

g) AC/DC-кодирование

После всех этапов, описанные выше, применяется адаптивное AC/DC-кодирование. На данном этапе происходит основное сжатие и уменьшение размера исходного изображения.

В свою очередь, декодирование же происходит в обратном порядке, начиная с AC/DC-кодирования и заканчивая преобразованием из $YC_BC_R$ в исходный формат.

## 2. Практическая часть.

### 2.1. Подготовка тестовых данных.

Изначально необходимо взять два изображение, одно из которых размерами $2048 \times 2048$ – Big.png, а второе – Lenna.png с исходными размерами $512 \times 512$. Данные изображения приведены на рисунке 2.1 и 2.2 соответственно.

Эти изображения будут использоваться в качестве эталонных входных данных для дальнейшего тестирования алгоритма сжатия. Они различаются как по размеру, так и по типу визуального содержимого, что позволяет оценить поведение кодировщика с разных сторон.

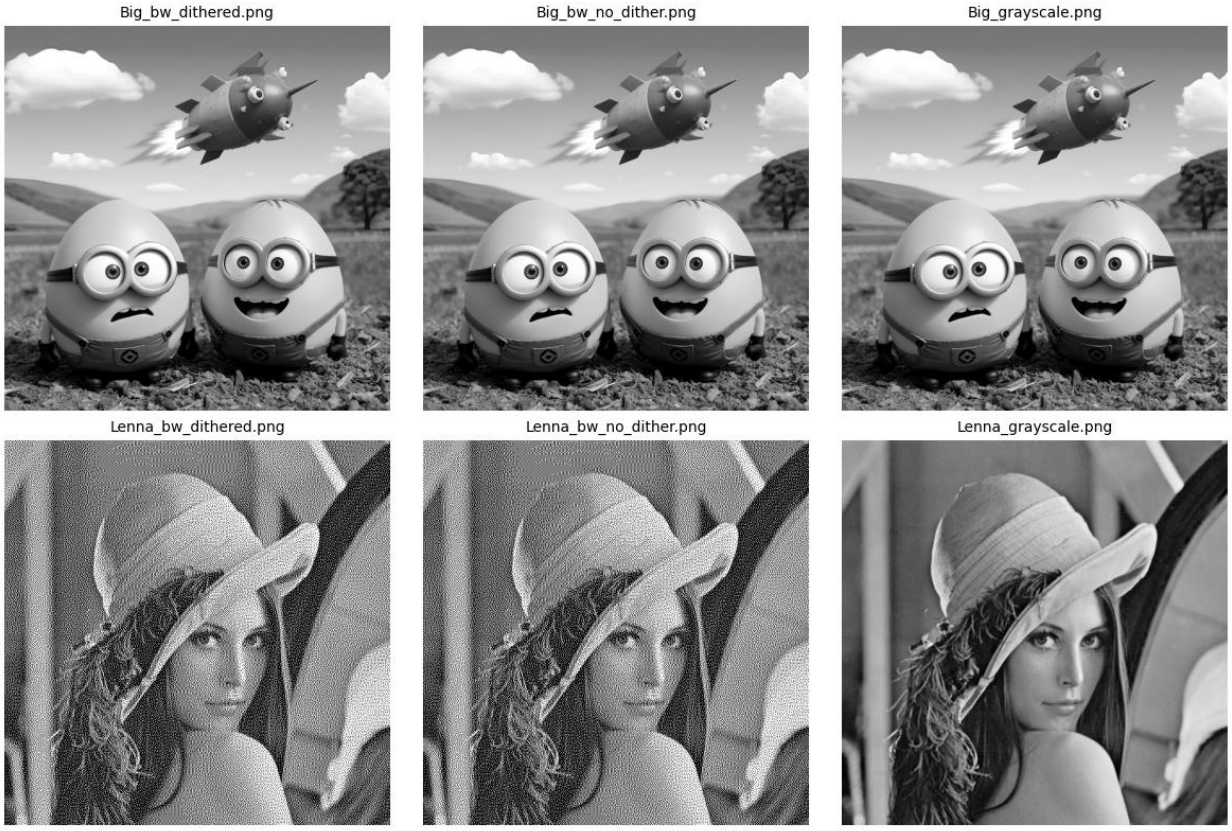
Big.png имеет более мультипликационный характер, а Lenna.png более реалистичный, при этом Big.png является гораздо более насыщенным, нежели Lenna.png.

Рисунок 2.1 – Big.png

Рисунок 2.2 – Lenna.png

Далее для каждого из этих изображений необходимо подготовить их аналоги, но немного конвертированные до следующих форматов: $grayscale, bw\ with\ dithering, bw\ without\ dithering$. Все данные изображения приведены на рисунке 2.3.

Рисунок 2.3 – ЧБ/GrayScale изображения




2.2. Графики для Big.png.

Для Big.png были построены графики, приведенные на изображениях 2.4 – 2.7 соответственно, более подробное описание будет приведено ниже, где будет рассказано о некоторых скачках.

Рисунок 2.4 – График для Big.png

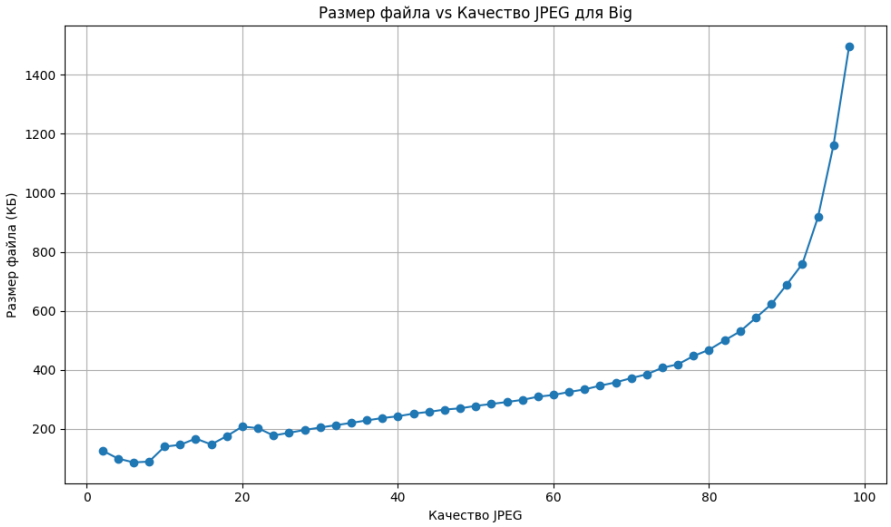


Рисунок 2.5 – График для Big_grayscale.png

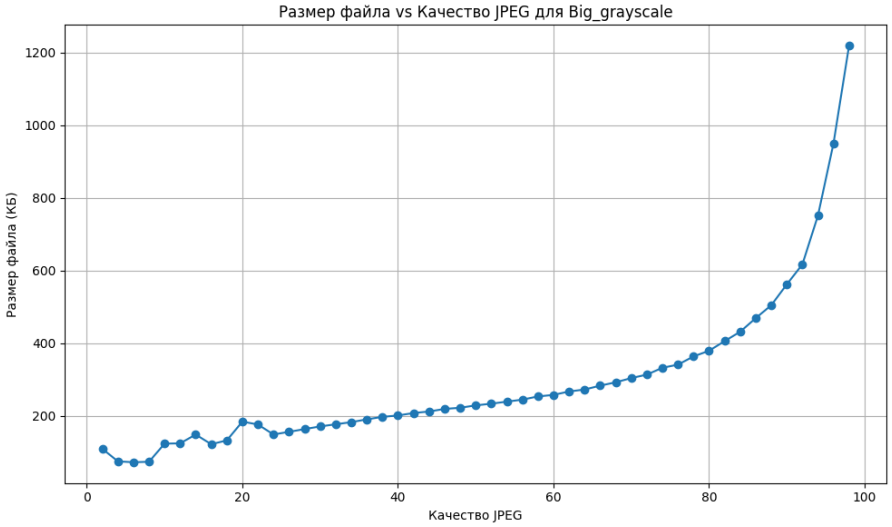



Рисунок 2.6 – График для Big_bw_without_dithering.png

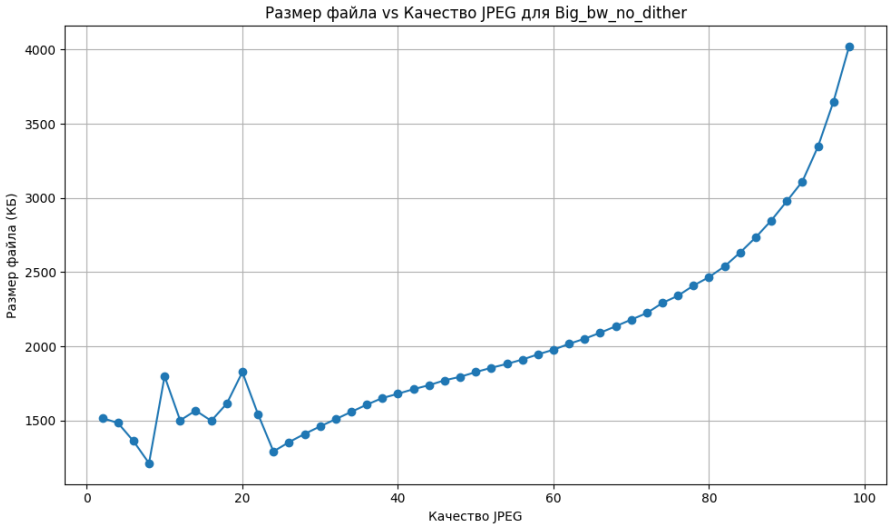
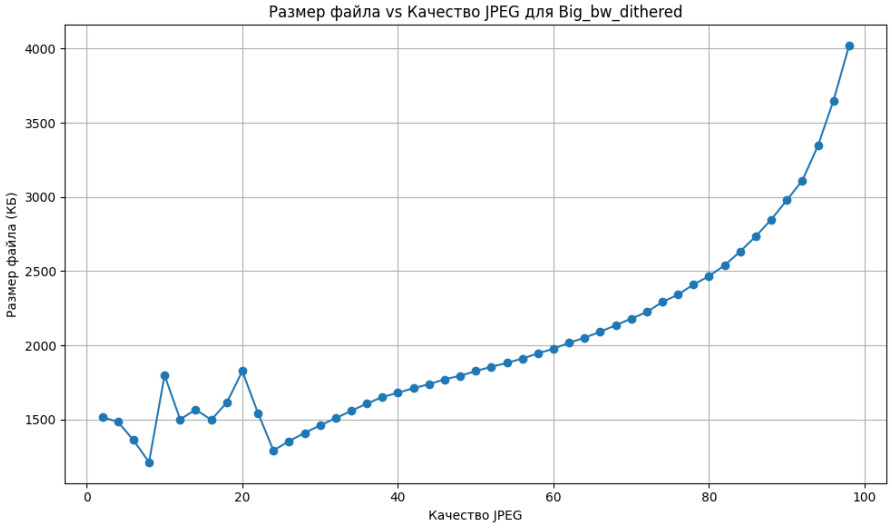



Рисунок 2.7 – График для Big_bw_with_dithering.png



Как видно из графиков, в целом работа идет исправно – в зависимости от качества для матрицы квантования увеличивается и сжимаемый файл, что сходится с теоретическим предположением. Однако, можно заметить, что для черно-белых изображений до качества в 24 наблюдаются неожиданные скачки и спады, а связано это может быть с тем, что матрицы квантования при малых значениях качества могут вести себя довольно агрессивно.

## 2.3. Графики для Lenna.png

Для Lenna.png графики были реализованы идентичным Big.png образом, а сами они приведены на рисунках 2.8 – 2.11.

Рисунок 2.8 – График для Lenna.png

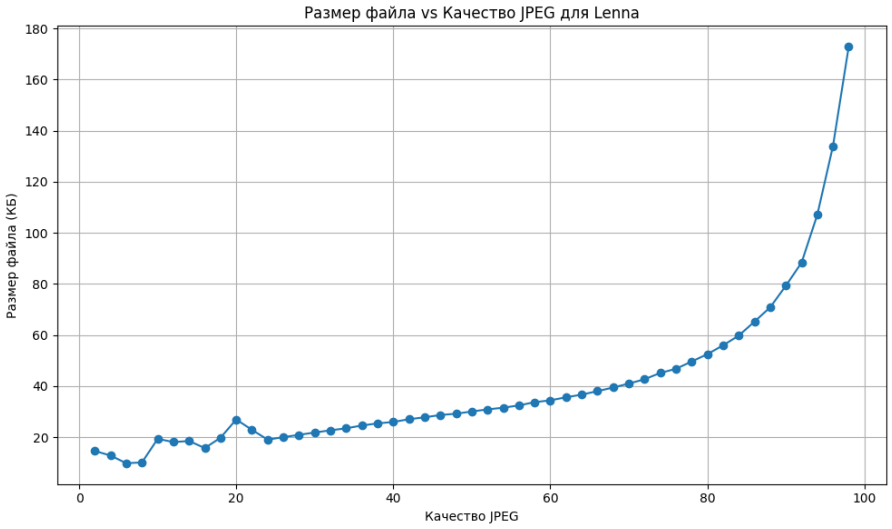



Рисунок 2.9 – График для Lenna_grayscale.png

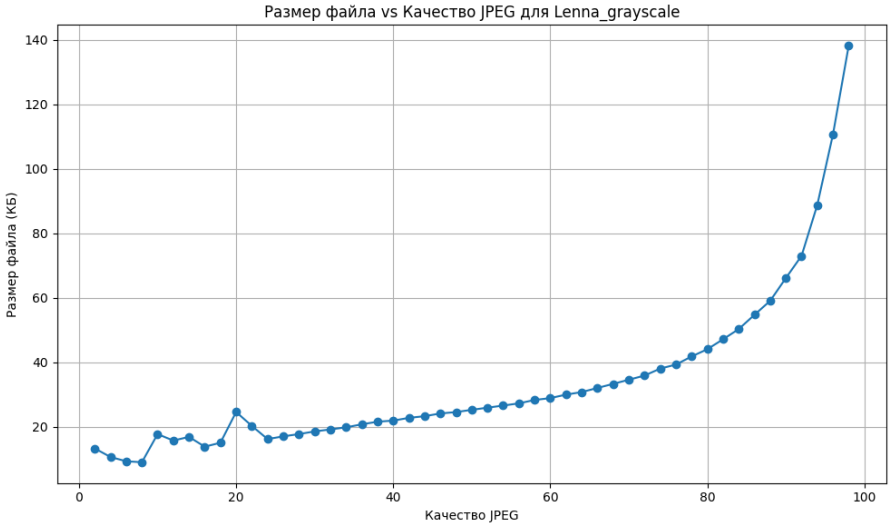


Рисунок 2.10 – График для Lenna_bw_without_dithering.png

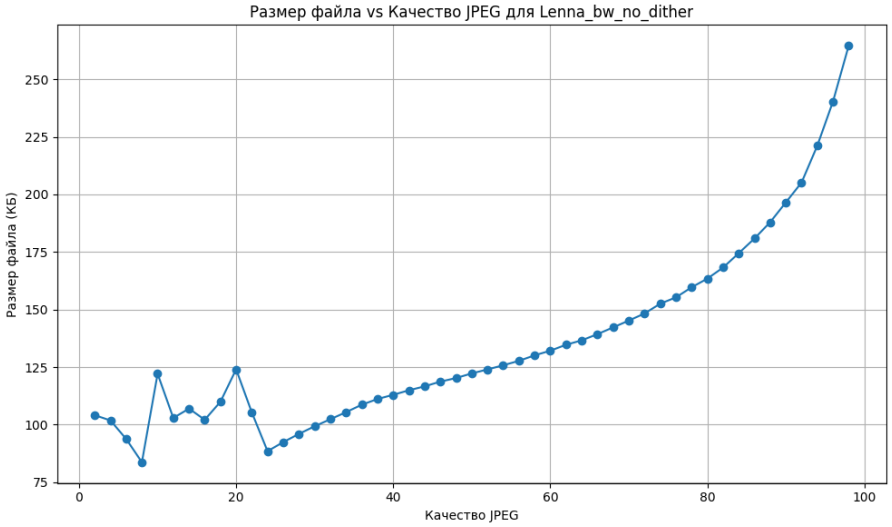
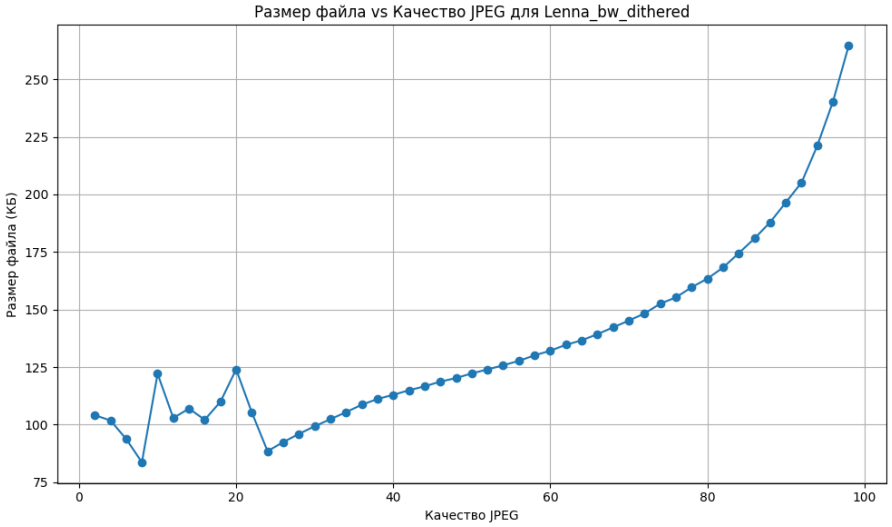



Рисунок 2.11 – График для Lenna_bw_with_dithering.png




На изображениях Lenna.png результат точь-в-точь такой же, какой и был на Big.png. Больше тут в целом и нечего говорить.

## 2.4. Примеры изображений при разных качествах.

После обработки всех изображений необходимо проверить их качество после декодирования, для этого на изображениях 1.12-1.19 приведены коллажи с разными изображениями, на которых качество идет в следующем порядке:

$$0 \rightarrow 20 \rightarrow 40 \rightarrow 60 \rightarrow 80 \rightarrow 100.$$

Это сделано для того, чтобы показать качественный пример работы программы.

Рисунок 2.12 – Изображения для Big.png

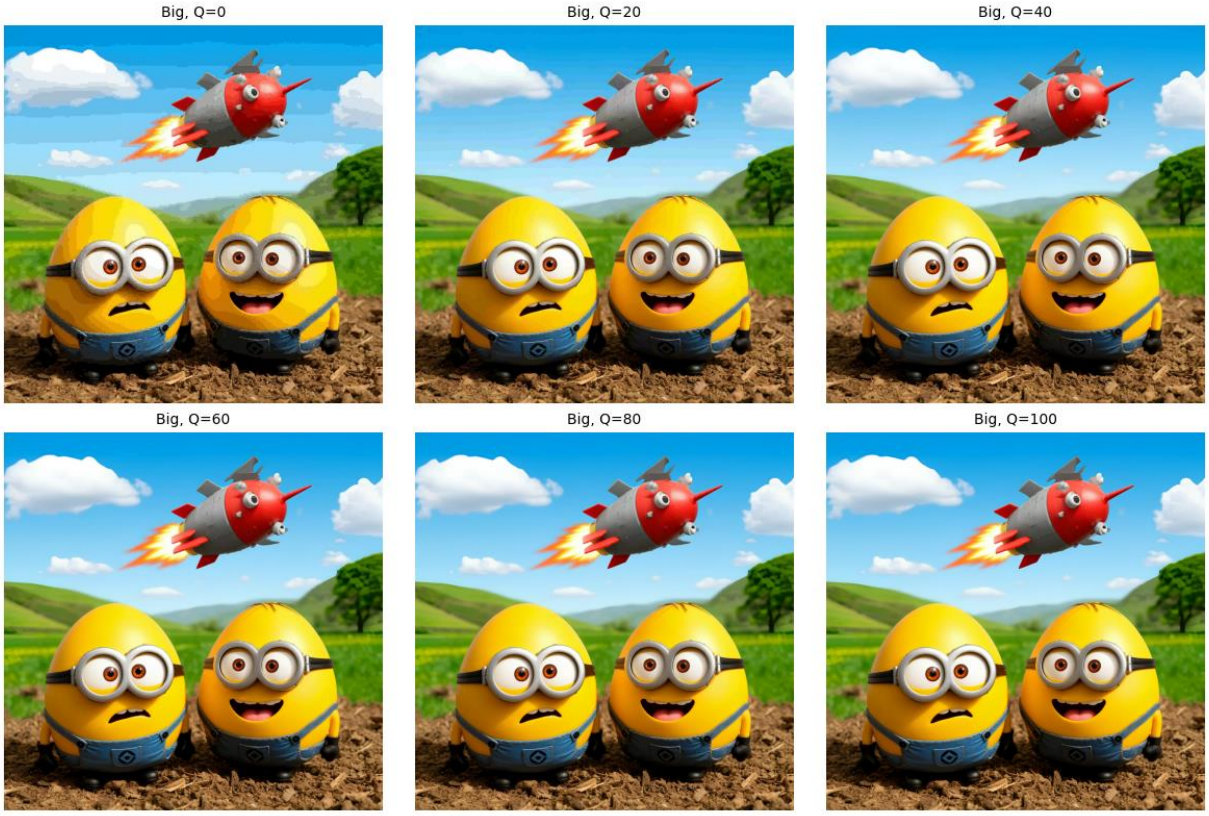



Рисунок 2.13 – Изображения для Big_grayscale.png

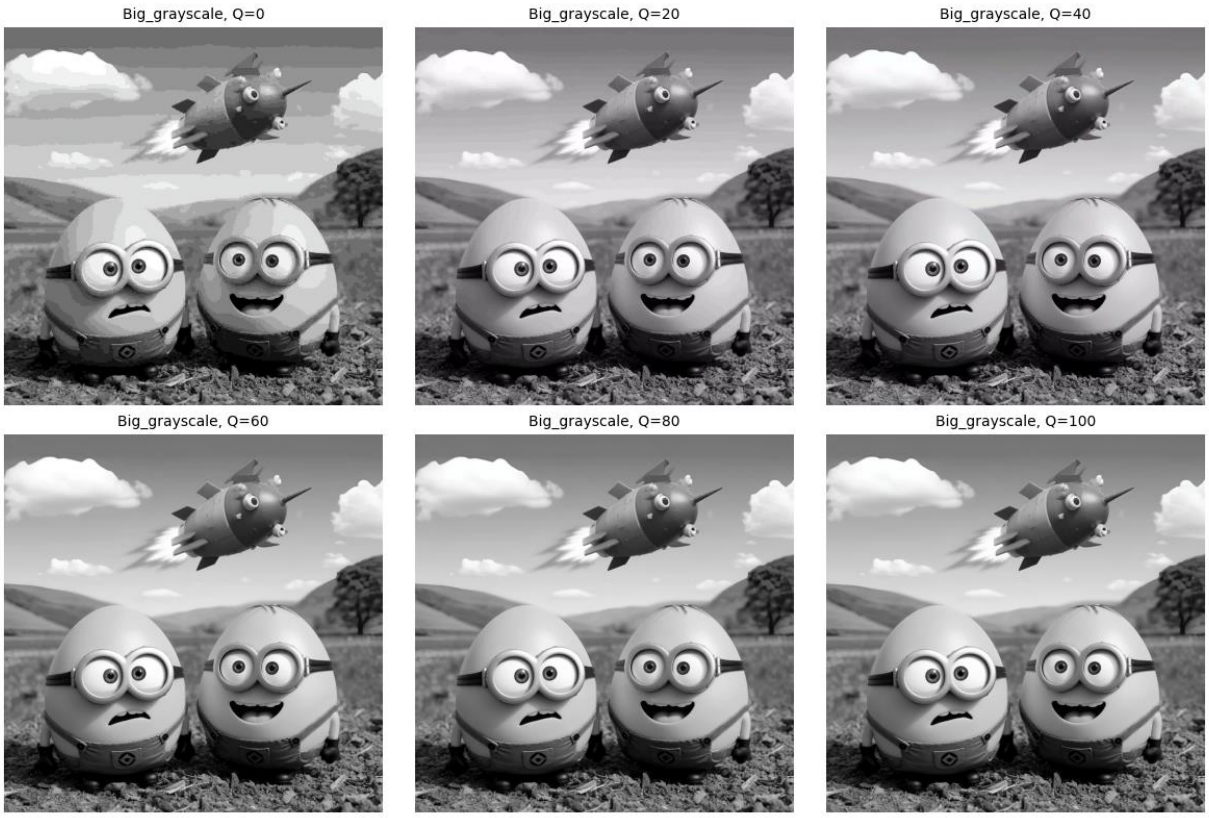


Рисунок 2.14 – Изображения для Big_bw_without_dithering.png

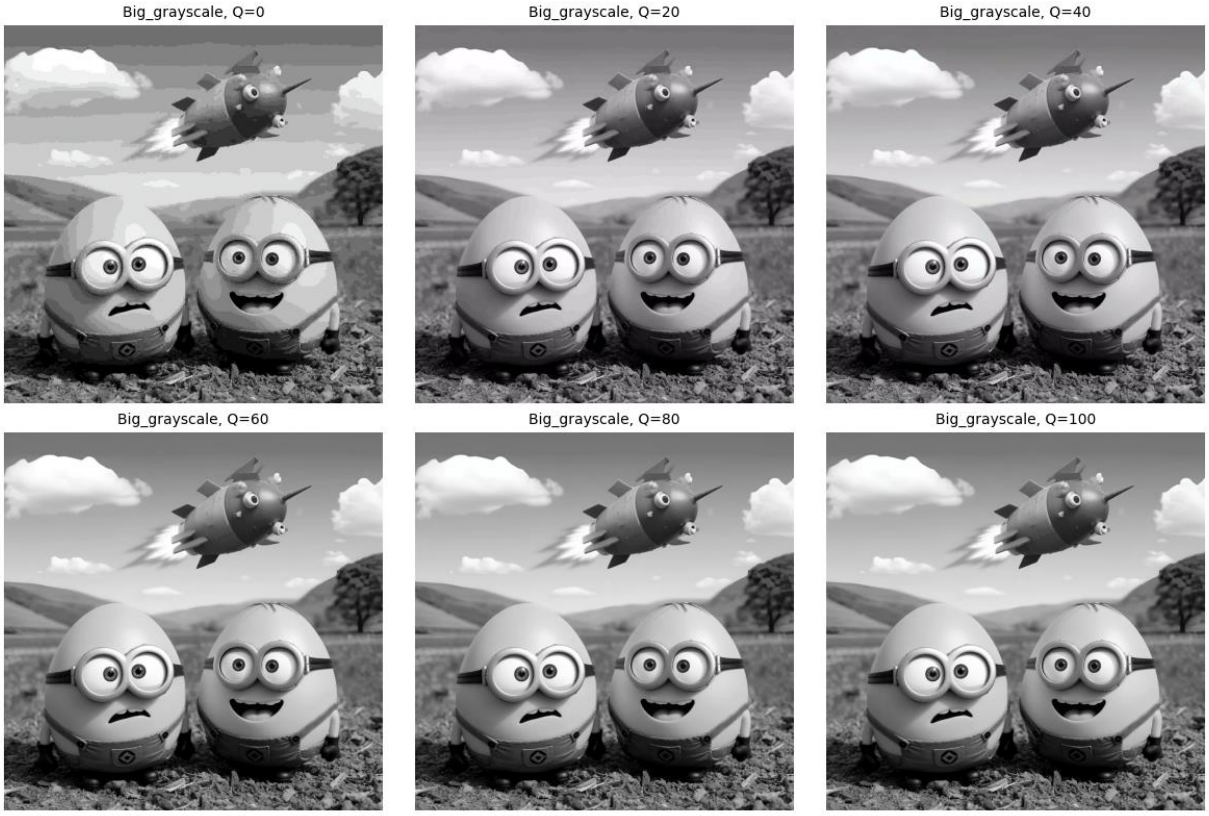



Рисунок 2.15 – Изображения для Big_bw_with_dithering.png

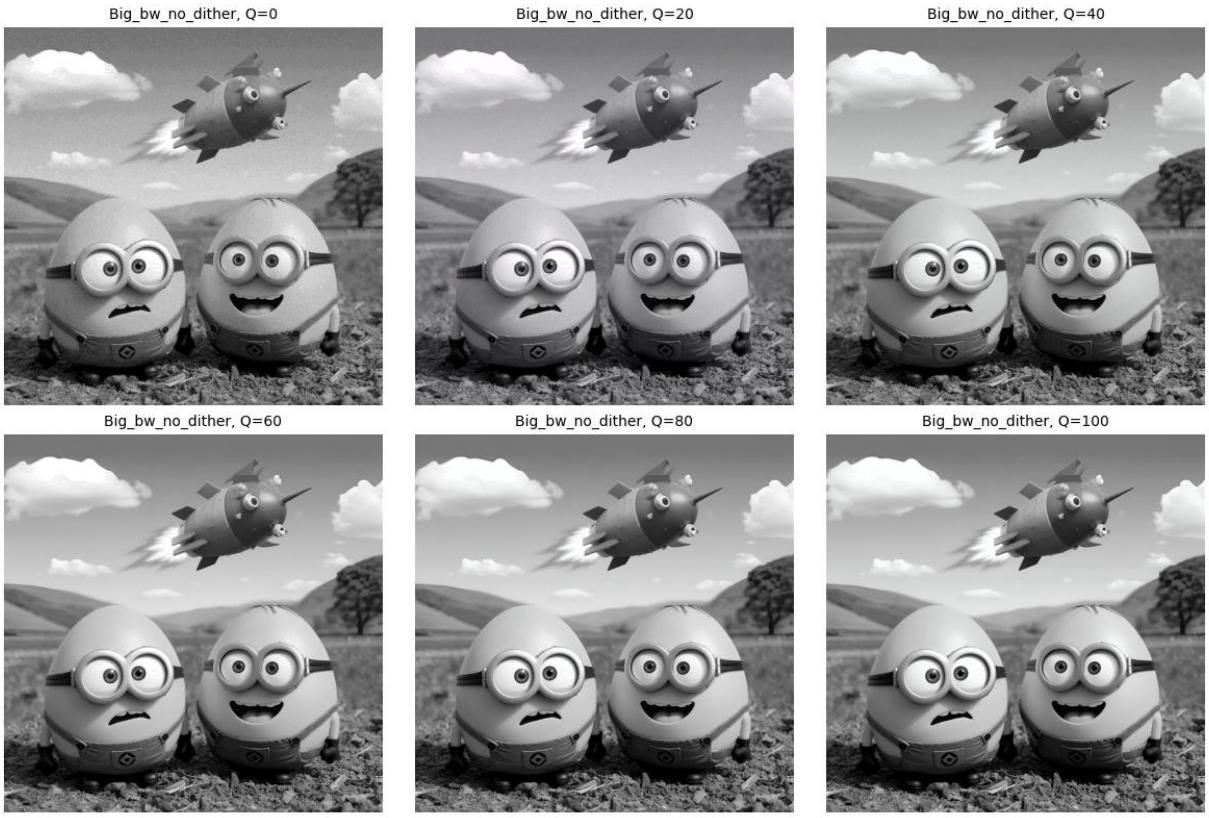


Рисунок 2.16 – Изображения для Lenna.png

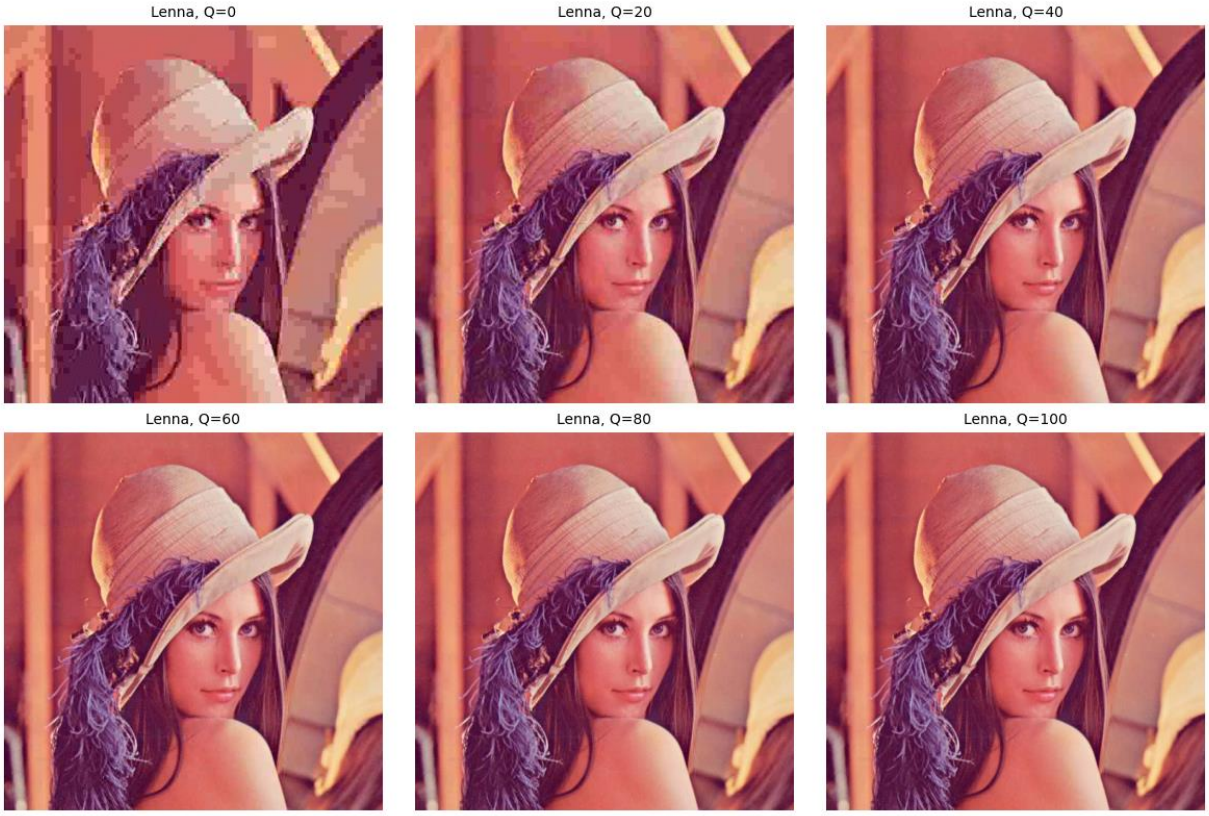



Рисунок 2.17 – Изображения для Lenna_grayscale.png

Рисунок 2.18 – Изображения для Lenna_bw_without_dithering.png


Рисунок 2.19 – Изображения для Lenna_bw_with_dithering.png

## Код программы

1) Main/imports.py – файл с основными импортами библиотек:

```
from PIL import Image

from typing import Union, Tuple, List

import numpy as np

import matplotlib.pyplot as plt

from collections import Counter

import struct

import heapq
```

2) Main/Main.py – файл с основным кодом:

```
import time
from Preprocess.ZigZag import *


class Main:
    def __init__(self, output: str='compressed.zmn'):
        self.out = output
        self.cv = Converter()
        self.pre = Preprocess()
        self.dctCLASS = DCTConvert()
        self.q = Quantum()
        self.zigzag = ZigZag()
        self.dif = ACDC(output)

    def encode(self, img: Union[str, Image.Image, np.ndarray], quality: int=80,
print_info=False) -> Union[None, str]:
        if not (0 <= quality <= 100):
            print(f"Error quality: {quality}")
            quit(1488)
        if print_info:
            print("Encoding...")
            st = time.time()

        image = self.cv.RGB2YCbCr(img)

        h, w = image.shape[:2]

        output_file = open(self.out, "wb")
        output_file.write(struct.pack('>HHBB', h, w, 8, quality))
        output_file.close()

        y, cb, cr = self.pre.downsample(image, 2)

        blocks = self.pre.split_by_blocks((y, cb, cr), 8)
        y, cb, cr = self.dctCLASS.dct2d(blocks)

        Q_Y = self.q.requant('y', quality=quality)
        Q_C = self.q.requant('c', quality=quality)

        y = self.q.quantile(Q_Y, y)
        cb = self.q.quantile(Q_C, cb)
        cr = self.q.quantile(Q_C, cr)

        y = self.zigzag.forward(y)
```

```
        cb = self.zigzag.forward(cb)
        cr = self.zigzag.forward(cr)

        self.dif.process(y)
        self.dif.process(cb)
        self.dif.process(cr)

        if print_info:
            print('Time for encoding: ', round(time.time()-st, 3), ' seconds')
            print('File saved: ', self.out)

    def    decode(self,    img:    str='decompressed.zmn',    print_info=False)    ->
Union[Image.Image, np.ndarray, str]:
        if print_info:
            print("Decoding...")
            st = time.time()

        buf_y, buf_cb, buf_cr, bsz, quality = self.dif.reprocess()

        y  = self.zigzag.inverse(buf_y)
        cb = self.zigzag.inverse(buf_cb)
        cr = self.zigzag.inverse(buf_cr)

        Q_Y = self.q.requant('y', quality=quality)
        Q_C = self.q.requant('c', quality=quality)

        y = self.q.dequantile(Q_Y, y)
        cb = self.q.dequantile(Q_C, cb)
        cr = self.q.dequantile(Q_C, cr)

        y, cb, cr = self.dctCLASS.idct2d((y, cb, cr), bsz)

        y_plane  = self.pre.merge_blocks((y,))
        cb_plane = self.pre.merge_blocks((cb,))
        cr_plane = self.pre.merge_blocks((cr,))

        subh, subw = y_plane.shape[:2]
        cb_plane = cb_plane[:subh, :subw]
        cr_plane = cr_plane[:subh, :subw]
        image = self.pre.upsample(
            (y_plane, cb_plane, cr_plane),
            factor=2
        )

        rgb_image = self.cv.YCbCr2RGB(image, out_path=img)

        if print_info:
            print('Time for decoding: ', round(time.time()-st, 3), ' seconds')
            print('File decompressed: ', img)

        return rgb_image
```

3) Preprocess/DCTConvert.py – файл с реализацией DCT и iDCT:

```
from .Preprocess import *
from functools import lru_cache



class DCTConvert:
```

```
    """
    Класс для выполнения прямого и обратного двумерного дискретного косинусного
преобразования (DCT/IDCT)
    над изображением, разбитым на блоки.
    """
    def __init__(self):
        self.pre = Preprocess()

    @staticmethod
    def _c(u):
        if isinstance(u, int):
            return 2 ** -0.5 if u == 0 else 1.0
        u = np.asarray(u)
        return np.where(u == 0, 2 ** -0.5, 1.0)

    @staticmethod
    @lru_cache(maxsize=None)
    def _get_dct_matrix(n: int):
        x = np.arange(n)
        u = np.arange(n)
        A = np.cos((2 * x[:, None] + 1) * u[None, :] * np.pi / (2 * n))
        C = np.where(u == 0, 2**-0.5, 1.0) * np.sqrt(2 / n)
        return A * C

    def _apply_dct_blocks(self, blocks: np.ndarray, A: np.ndarray) -> np.ndarray:
        return A.T @ blocks @ A

    def _apply_dct_blocks_vec(self, blocks: np.ndarray, A: np.ndarray) -> np.ndarray:
        return np.einsum("ij,xyjk,kl->xyil", A.T, blocks, A)

    def   dct2d(self,   img:   Union[List[np.ndarray],   Image.Image,   np.ndarray],
block_size=8) \
            -> Union[Tuple[np.ndarray], List[np.ndarray]]:
        if not isinstance(img, list):
            img = self.pre.split_by_blocks(img, block_size=block_size)

        A = self._get_dct_matrix(block_size)
        channels = []
        for channel in img:
            channel = channel.astype(np.float64)
            dct_blocks = np.einsum("ij,xyjk,kl->xyil", A.T, channel, A)
```

```
            channels.append(dct_blocks)
        return channels

    def idct2d(self, img: Union[List[np.ndarray], Image.Image, np.ndarray],
block_size=8) \
            -> Union[Tuple[np.ndarray], List[np.ndarray]]:
        blocks = []
        if isinstance(img, (list, tuple)) and isinstance(img[0], np.ndarray):
            for channel in img:
                if channel.ndim == 3 and channel.shape[1:] == (block_size,
block_size):
                    num_blocks = channel.shape[0]
                    side = int(np.sqrt(num_blocks))
                    if side * side != num_blocks:
                        raise ValueError(f"Невозможно преобразовать {num_blocks}
блоков в квадратную сетку")
                    channel = channel.reshape(side, side, block_size, block_size)
                blocks.append(channel)
        else:
            blocks = self.pre.split_by_blocks(img, block_size=block_size)

        A = self._get_dct_matrix(block_size)
        channels = []
        for channel in blocks:
            channel = channel.astype(np.float64)
            idct_blocks = np.einsum("ij,xyjk,kl->xyil", A, channel, A.T)
            channels.append(idct_blocks)
        return channels
```

4) Preprocess/Preprocess.py – файл с реализацией всех добавочных функций (upsampling, downsampling, etc):

```
from Coder.Converter import *


class Preprocess:
    """
    Класс предназначен для предварительной обработки изображений:
    разбиения на блоки, сжатия цветовых каналов (subsampling), восстановления
    каналов (upsampling), а также объединения блоков обратно в изображение.
    """
    def __init__(self):
        pass

    @staticmethod
    def _downsample_channel(channel: np.ndarray, factor: int) -> np.ndarray:
        h, w = channel.shape
```

```
        pad_h = (factor - h % factor) % factor
        pad_w = (factor - w % factor) % factor

        channel = np.pad(channel, ((0, pad_h), (0, pad_w)), mode='edge')

        new_h = channel.shape[0] // factor
        new_w = channel.shape[1] // factor

        return channel.reshape(new_h, factor, new_w, factor).mean(axis=(1, 3))

    def downsample(self, img: Union[np.ndarray, Image.Image, Tuple[np.ndarray, ...]],
                   factor: int = 2) -> Union[Tuple[np.ndarray, ...]]:
        """
        Сжимает цветовые каналы изображения.

        Args:
            img (Union[np.ndarray, Image.Image, Tuple[np.ndarray, ...]]):
                Изображение в формате NumPy массива, PIL.Image или кортежа каналов
            factor (int, optional):
                Коэффициент сжатия для хроматических каналов. По умолчанию 2

        Returns:
            Union[Tuple[np.ndarray, ...]]: Кортеж из трёх каналов (Y, Cb, Cr)
        """
        if isinstance(img, Image.Image):
            img = np.array(img).astype(np.float32)
        elif isinstance(img, np.ndarray):
            c1, c2, c3 = img[:, :, 0], img[:, :, 1], img[:, :, 2]
        else:
            c1, c2, c3 = img
        c2 = self._downsample_channel(c2, factor)
        c3 = self._downsample_channel(c3, factor)

        return c1, c2, c3

    @staticmethod
    def _split_channel_by_block(channel: np.ndarray, block_size: int):
        """
        Внутренний метод для разбиения канала на блоки.

        Args:
            channel (np.ndarray): Канал изображения
            block_size (int): Размер блока

        Returns:
            np.ndarray: Массив блоков заданного размера
        """
        h, w = channel.shape
        pad_h = int(np.ceil(h / block_size) * block_size - h)
        pad_w = int(np.ceil(w / block_size) * block_size - w)

        padded = np.pad(channel, ((0, pad_h), (0, pad_w)), mode='edge')
        blocks = padded.reshape(
            padded.shape[0] // block_size, block_size,
            padded.shape[1] // block_size, block_size
        )
        blocks = blocks.transpose(0, 2, 1, 3) # y, x, block_size, block_size
        return blocks

    def split_by_blocks(self, img: Union[np.ndarray, Image.Image, Tuple[np.ndarray,
...]],
```

```
                        block_size: int = 8) -> Union[np.ndarray, List[np.ndarray]]:
        """
        Разбивает изображение на блоки заданного размера.

        Args:
            img (Union[np.ndarray, Image.Image, Tuple[np.ndarray, ...]]):
                Изображение в формате NumPy массива, PIL.Image или кортежа каналов
            block_size (int, optional):
                Размер блоков. По умолчанию 8

        Returns:
            Union[np.ndarray, List[np.ndarray]]:
                Массив блоков для одного канала или список массивов для нескольких
каналов
        """
        if isinstance(img, Image.Image):
            img = np.array(img).astype(np.float32)

        blocks = []
        if isinstance(img, np.ndarray):
            if img.ndim == 2:
                img = img[:, :, np.newaxis]
            _, _, channels = img.shape
            for c in range(channels):
                block = self._split_channel_by_block(img[:, :, c], block_size)
                blocks.append(block)
        else:
            for channel in img:
                block = self._split_channel_by_block(channel, block_size)
                blocks.append(block)

        if len(blocks) == 1:
            return blocks[0]

        return blocks

    def merge_blocks(self, img: Tuple[np.ndarray, ...]) -> np.ndarray:
        merged_channels = []
        for blocks in img:
            yb, xb, bh, bw = blocks.shape
            channel = blocks.transpose(0,2,1,3).reshape(yb*bh, xb*bw)
            merged_channels.append(channel)

        if len(merged_channels) == 1:
            return merged_channels[0]

        # иначе стэкаем в третий канал
        return np.stack(merged_channels, axis=-1)

    def upsample(self,
                 img:  Union[Image.Image,  np.ndarray,  Tuple[np.ndarray,  np.ndarray,
np.ndarray]],
                 factor: int = 2) -> np.ndarray:
        """
        Восстанавливает три канала (Y, Cb, Cr) в RGB-изображение:
        1) дублирует (upsample) Cb и Cr в два раза,
        2) кадрирует их под Y,
        3) собирает три плоскости в цветное изображение.
        """
        if isinstance(img, Image.Image):
            img = np.array(img).astype(np.float32)
```

```
        if isinstance(img, tuple) and len(img) == 3:
            y_plane, cb_plane, cr_plane = img
        else:
            y_plane, cb_plane, cr_plane = img[:, :, 0], img[:, :, 1], img[:, :, 2]

        cb_up = np.repeat(np.repeat(cb_plane, factor, axis=0), factor, axis=1)
        cr_up = np.repeat(np.repeat(cr_plane, factor, axis=0), factor, axis=1)

        h, w = y_plane.shape
        cb_up = cb_up[:h, :w]
        cr_up = cr_up[:h, :w]

        return np.stack((y_plane, cb_up, cr_up), axis=-1)
```

5) Preprocess/Quantum.py – файл с реализацией матриц квантования:

```
from .DCTConvert import *

class Quantum:
    """
    Выполняет квантование и деквантование DCT-коэффициентов с использованием матриц квантования.

    Методы:
        requant: Генерирует матрицу квантования для заданного качества
        quantile: Применяет квантование к DCT-коэффициентам
        dequantile: Восстанавливает коэффициенты из квантованных значений
    """
    def __init__(self):
        self.Q_Y = np.array([
            [16, 11, 10, 16, 24, 40, 51, 61],
            [12, 12, 14, 19, 26, 58, 60, 55],
            [14, 13, 16, 24, 40, 57, 69, 56],
            [14, 17, 22, 29, 51, 87, 80, 62],
            [18, 22, 37, 56, 68, 109, 103, 77],
            [24, 35, 55, 64, 81, 104, 113, 92],
            [49, 64, 78, 87, 103, 121, 120, 101],
            [72, 92, 95, 98, 112, 100, 103, 99]
        ])
        self.Q_C = np.array([
            [17, 18, 24, 47, 99, 99, 99, 99],
            [18, 21, 26, 66, 99, 99, 99, 99],
            [24, 26, 56, 99, 99, 99, 99, 99],
            [47, 66, 99, 99, 99, 99, 99, 99],
            [99, 99, 99, 99, 99, 99, 99, 99],
            [99, 99, 99, 99, 99, 99, 99, 99],
            [99, 99, 99, 99, 99, 99, 99, 99],
            [99, 99, 99, 99, 99, 99, 99, 99]
        ])

    def requant(self, Q: str='y', quality: int=100):
        """
        Генерирует масштабированную матрицу квантования для заданного уровня качества.

        Args:
            Q (str, optional):
                Тип матрицы: 'y' для яркостного канала, 'c' для цветностного.
                По умолчанию 'y'.
            quality (int, optional):
```

```
                Уровень качества (0-100). По умолчанию 100 (максимальное качество).

        Returns:
            np.ndarray: Масштабированная матрица квантования размером 8x8
        """
        if Q.lower() == 'y':
            Q = self.Q_Y
        else:
            Q = self.Q_C

        epsilon = 1e-2

        if quality == 0:
            scale = 5000 / epsilon
        elif quality < 50:
            scale = 5000 / quality
        else:
            scale = 200 - 2 * quality

        Q_scaled = np.floor((Q * scale + 50) / 100)
        Q_scaled[Q_scaled == 0] = 1
        return Q_scaled.astype(np.uint8)

    @staticmethod
    def quantile(Q, channel):
        """
        Применяет квантование к DCT-коэффициентам канала.

        Args:
            Q (np.ndarray): Матрица квантования 8x8
            channel (np.ndarray):
                Канал изображения в виде массива блоков DCT-коэффициентов
                формата (число_блоков_Y, число_блоков_X, 8, 8)

        Returns:
            np.ndarray: Квантованные коэффициенты (целочисленные значения)
        """
        return np.array([np.round(block / Q).astype(np.int16) for block in channel])

    @staticmethod
    def dequantile(Q, channel):
        """
        Восстанавливает DCT-коэффициенты из квантованных значений.

        Args:
            Q (np.ndarray): Матрица квантования 8x8
            channel (np.ndarray):
                Квантованный канал в виде массива блоков
                формата (число_блоков_Y, число_блоков_X, 8, 8)

        Returns:
            np.ndarray: Восстановленные DCT-коэффициенты
        """
        return np.array([np.round(block * Q).astype(np.int16) for block in channel])
```

6) Preprocess/ZigZag.py – файл с реализацией зигзаг обхода:

```
from .Quantum import *

class ZigZag:
```

```
    """
    Выполняет зигзаг-сканирование блоков DCT-коэффициентов для JPEG-сжатия и
восстановление блоков.

    Методы:
        forward: Преобразует блоки коэффициентов в зигзаг-последовательность
        inverse: Восстанавливает блоки коэффициентов из зигзаг-последовательности
    """
    def __init__(self):
        pass

    @staticmethod
    def _forward_for_block(block: np.ndarray) -> np.ndarray:
        """
        Внутренний метод для зигзаг-сканирования одного блока.

        Args:
            block (np.ndarray): Блок коэффициентов размером NxN

        Returns:
            np.ndarray: Плоский массив коэффициентов в зигзаг-порядке
        """
        a, n, _ = block.shape
        result = []
        for k in range(a):
            sigm = []
            for d in range(2 * n - 1):
                for i in range(d + 1):
                    j = d - i
                    if i < n and j < n:
                        if d % 2 == 0:
                            sigm.append(block[k, i, j])
                        else:
                            sigm.append(block[k, j, i])
            result.append(np.array(sigm))
        return np.array(result)

    def forward(self, channel: np.ndarray) -> np.ndarray:
        """
        Преобразует блоки DCT-коэффициентов в зигзаг-последовательность.

        Args:
            channel (np.ndarray):
                Входные данные в формате:
                - Отдельный блок (2D массив)
                - Набор блоков (3D массив [число_блоков, N, N])
                - Канал изображения (4D массив [y_blocks, x_blocks, N, N])

        Returns:
            np.ndarray: Массив коэффициентов в зигзаг-порядке:
                - Для 2D входа: 1D массив
                - Для 3D/4D входа: 2D массив [число_блоков, N*N]
        """
        if len(channel.shape) == 2:
            return self._forward_for_block(channel)

        result = []
        for block in channel:
            result.append(self._forward_for_block(block))
        return np.array(result)
```

```
    @staticmethod
    def _inverse_for_block(flat_blocks: np.ndarray) -> np.ndarray:
        """
        Внутренний метод для восстановления блока из зигзаг-последовательности.

        Args:
            flat_blocks (np.ndarray): Плоский массив коэффициентов в зигзаг-порядке

        Returns:
            np.ndarray: Восстановленный блок коэффициентов размером NxN
        """
        num_blocks = flat_blocks.shape[0]
        N = 8
        result = []

        zigzag_indices = []
        for d in range(2 * N - 1):
            for i in range(d + 1):
                j = d - i
                if i < N and j < N:
                    if d % 2 == 0:
                        zigzag_indices.append((i, j))
                    else:
                        zigzag_indices.append((j, i))

        for k in range(num_blocks):
            block = np.zeros((N, N), dtype=flat_blocks.dtype)
            for idx, (i, j) in enumerate(zigzag_indices):
                block[i, j] = flat_blocks[k][idx]
            result.append(block)

        return np.array(result)

    def inverse(self, channel: np.ndarray) -> np.ndarray:
        """
        Восстанавливает блоки DCT-коэффициентов из зигзаг-последовательности.

        Args:
            channel (np.ndarray):
                Зигзаг-последовательность в формате:
                - 1D массив (один блок)
                - 2D массив [число_блоков, N*N]
                - 3D массив [каналы, число_блоков, N*N]

        Returns:
            np.ndarray: Восстановленные блоки коэффициентов:
                - Для 1D входа: 2D массив NxN
                - Для 2D/3D входа: 4D массив [y_blocks, x_blocks, N, N]
        """
        if len(channel.shape) == 2:
            return self._inverse_for_block(channel)

        result = []
        for blocks in channel:
            result.append(self._inverse_for_block(blocks))
        return np.array(result)
```

7) Coder/ACDC.py – файл с реализацией AC/DC-кодирования и считывания данных с файла:

```
from .Huffman import *


class ACDC:
    """
    Класс для кодирования/декодирования DC и AC коэффициентов DCT-преобразования
    с использованием алгоритма Хаффмана и сохранения в бинарный файл.

    Args:
        file (str): Путь к файлу для записи/чтения закодированных данных

    Атрибуты:
        CAT_FORM (str): Формат упаковки категорий DC коэффициентов (>BI - big-endian,
unsigned char + unsigned int)
        SEQ_LEN_FORM (str): Формат длины закодированной последовательности (>I - big-
endian unsigned int)
        RLE_CAT_FORM (str): Формат RLE пар для AC коэффициентов (>IBI - big-endian,
unsigned int + unsigned char + unsigned int)

    Методы:
        encode_dc: Кодирует DC коэффициенты
        encode_ac: Кодирует AC коэффициенты с RLE
        process: Основной метод обработки канала
        decode: Декодирует данные из файла
        reprocess: Полное восстановление изображения из файла
    """
    def __init__(self, file: str):
        self.fn = file
        self.file = open(file, 'wb')
        self.file.close()
        self.Huf = Huffman()
        self.CAT_FORM = '>BI'
        self.SEQ_LEN_FORM = '>I'
        self.RLE_CAT_FORM = '>IBI'

    def close_file(self):
        self.file.close()

    def open_file(self):
        self.file = open(self.fn, 'ab')

    @staticmethod
    def merge_dc_ac(dc: np.ndarray, ac: np.ndarray) -> np.ndarray:
        """
        Объединяет DC и AC коэффициенты.

        Args:
            dc (np.ndarray): Массив DC коэффициентов формы (N, 1)
            ac (np.ndarray): Массив AC коэффициентов формы (N, 63)

        Returns:
            np.ndarray: Объединённый массив коэффициентов формы (N, 64)
        """
        return np.array([[dc[i][0]] + list(ac[i]) for i in range(len(dc))])

    @staticmethod
    def convert(x, cat):
        """
        Конвертирует значение коэффициента в битовую строку.

        Args:
```

```
        x (int): Значение коэффициента
        cat (int): Категория коэффициента

    Returns:
        str: Битовая строка заданной длины
    """
    if cat == 0:
        return ""
    if x < 0:
        x = ((1 << cat) - 1) ^ (abs(x))
    return bin(x)[2:].zfill(cat)

@staticmethod
def iconvert(bits, cat):
    """
    Обратное преобразование битовой строки в значение.

    Args:
        bits (str): Битовая строка
        cat (int): Категория коэффициента

    Returns:
        int: Восстановленное значение коэффициента
    """
    if cat == 0:
        return 0
    value = int(bits, 2)
    if bits[0] == '0':
        value = -((1 << cat) - 1 - value)
    return value

@staticmethod
def category(delta):
    """
    Вычисляет категорию для коэффициента.

    Args:
        delta (int): Разность коэффициентов

    Returns:
        int: Категория (0 для нуля, иначе floor(log2(|delta|)) + 1)
    """
    abs_val = abs(delta)
    if abs_val == 0:
        return 0
    return int(np.floor(np.log2(abs_val))) + 1

def encode_dc(self, arr: np.array):
    """
Кодирует DC-коэффициенты с использованием алгоритма Хаффмана.

Args:
    arr (np.array): Массив DC-коэффициентов формы (N,)
        где N - количество блоков в изображении

Returns:
    None: Результаты записываются в файл:
        - Частотный словарь категорий
        - Закодированные данные
        - Информация о дополнении
"""
```

```
        n = len(arr)
        arr = tuple(map(int, arr))

        arr_dc = (arr[0],) + tuple(map(lambda i: arr[i] - arr[i - 1], range(1, n)))
        categories = tuple(map(self.category, arr_dc))
        freq_dict = dict(Counter(categories))

        self.file.write(struct.pack('>H', len(freq_dict)))
        for i in sorted(freq_dict.keys()):
            self.file.write(struct.pack(self.CAT_FORM, i, freq_dict[i]))

        root = self.Huf.build_tree(freq_dict)
        codes = self.Huf.build_code(root)

        huf_str = ""
        encoded = bytearray()
        for i in range(n):
            dc = arr_dc[i]
            cat = self.category(dc)

            huf_str += codes[cat] + self.convert(dc, cat)
            while len(huf_str) >= 8:
                encoded.append(int(huf_str[:8], 2))
                huf_str = huf_str[8:]

        padding = 0
        if len(huf_str) != 0:
            padding = 8 - len(huf_str)
            encoded.append(int(huf_str.ljust(8, '0'), 2))

        self.file.write(struct.pack(self.SEQ_LEN_FORM, len(encoded)))
        self.file.write(bytes(encoded))
        self.file.write(struct.pack('>B', padding))

    def encode_ac(self, arr: np.array):
        """
        Кодирует AC-коэффициенты с RLE и алгоритмом Хаффмана.

        Args:
            arr (np.array): Массив AC-коэффициентов формы (M,)
                где M = (число блоков) * (размер блока^2 - 1)

        Returns:
            None: Результаты записываются в файл:
                - RLE пары (длина нулевой последовательности, категория)
                - Закодированные данные
                - Информация о дополнении
        """
        n = len(arr)
        arr = tuple(map(int, arr))
        arr_ac = (arr[0],) + tuple(map(lambda i: arr[i] - arr[i - 1], range(1, n)))
        ac, rle_ac = [], []

        zeros_cnt = 0
        for i in range(n):
            if arr_ac[i] != 0:
                ac.append(arr_ac[i])
                rle_ac.append((zeros_cnt, self.category(arr_ac[i])))
                zeros_cnt = 0
            else:
                zeros_cnt += 1
```

```
        if zeros_cnt != 0:
            ac.append(0)
            rle_ac.append((zeros_cnt, 0))

        freq_dict = dict(Counter(rle_ac))
        self.file.write(struct.pack('>H', len(freq_dict)))
        for couple, value in sorted(freq_dict.items()):
            self.file.write(struct.pack(self.RLE_CAT_FORM, *couple, value))

        root = self.Huf.build_tree(freq_dict)
        codes = self.Huf.build_code(root)

        huf_str = ""
        encoded = bytearray()
        for i in range(len(rle_ac)):
            cat = rle_ac[i][1]
            huf_str += codes[rle_ac[i]] + self.convert(ac[i], cat)

            while len(huf_str) >= 8:
                encoded.append(int(huf_str[:8], 2))
                huf_str = huf_str[8:]

        padding = 0
        if len(huf_str) != 0:
            padding = 8 - len(huf_str)
            encoded.append(int(huf_str.ljust(8, '0'), 2))

        self.file.write(struct.pack(self.SEQ_LEN_FORM, len(encoded)))
        self.file.write(bytes(encoded))
        self.file.write(struct.pack('>B', padding))

    def process(self, channel: np.ndarray):
        """
        Обрабатывает канал изображения: кодирует DC и AC коэффициенты.

        Args:
            channel (np.ndarray):
                Канал изображения в виде 3D массива формы (число_блоков,
размер_блока^2)
                Например: (100, 64) для 100 блоков 8x8
        """
        self.open_file()

        _, _, n = channel.shape
        blocks = channel.reshape(-1, n)
        dc = blocks[:, 0]
        ac = blocks[:, 1:].flatten()

        self.encode_dc(dc)
        self.encode_ac(ac)

        self.close_file()

    def decode(self, file, mode, blocks_cnt, block_size):
        """
        Декодирует данные из файла.

        Args:
            file (file object): Открытый файловый объект
            mode (str): Режим декодирования ('DC' или 'AC')
            blocks_cnt (int): Количество блоков для декодирования
```

```
            block_size (int): Размер блока (обычно 8)

        Returns:
            np.ndarray: Массив декодированных коэффициентов
        """
        assert mode == 'AC' or mode == 'DC'

        freg_dict_len = struct.unpack('>H', file.read(2))[0]
        freq_dict = dict()
        for i in range(freg_dict_len):
            form = self.CAT_FORM if mode == 'DC' else self.RLE_CAT_FORM
            s = struct.calcsize(form)
            couple = struct.unpack(form, file.read(s))

            key, value = couple if mode == 'DC' else (couple[:2], couple[2])
            freq_dict[key] = value

        len_data                  =                  struct.unpack(self.SEQ_LEN_FORM,
file.read(struct.calcsize(self.SEQ_LEN_FORM)))[0]
        data = file.read(len_data)
        padding = struct.unpack('>B', file.read(1))[0]

        root = self.Huf.build_tree(freq_dict)
        return  ACDC.decode_data(self,  data,  root=root,  padding=padding,  mode=mode,
blocks_cnt=blocks_cnt, n=block_size)

    def decode_data(self, data: bytes, root: Huffman.Node, padding: int, mode: str,
blocks_cnt: int, n):
        """
        Декодирует бинарные данные с использованием дерева Хаффмана.

        Args:
            data (bytes): Закодированные бинарные данные
            root (Huffman.Node): Корневой узел дерева Хаффмана
            padding (int): Количество бит дополнения в последнем байте
            mode (str): Режим декодирования: 'DC' или 'AC'
            blocks_cnt (int): Количество блоков для декодирования
            n (int): Размер блока (для AC - длина зигзаг-последовательности)

        Returns:
            np.ndarray: Массив декодированных коэффициентов:
                - Для DC: форма (blocks_cnt,)
                - Для AC: форма (blocks_cnt, n-1)

        Raises:
            ValueError: Если обнаружена недопустимая битовая последовательность
        """
        assert mode == 'AC' or mode == 'DC'

        bits_buffer = ''.join(f'{byte:08b}' for byte in data)
        if padding > 0:
            bits_buffer = bits_buffer[:-padding]

        decoded = []
        cur_node = root
        i = 0

        while i < len(bits_buffer):
            bit = bits_buffer[i]
            i += 1
            cur_node = cur_node.left if bit == '0' else cur_node.right
```

```
            if cur_node.value is not None:
                if mode == 'DC':
                    cat = cur_node.value
                    if cat != 0:
                        dc = self.iconvert(bits_buffer[i:i+cat], cat)
                        decoded.append(dc)
                    else:
                        decoded.append(0)
                    i += cat
                else:
                    run_len, cat = cur_node.value
                    decoded.extend([0] * run_len)
                    if cat != 0:
                        ac = self.iconvert(bits_buffer[i:i+cat], cat)
                        decoded.append(ac)
                    else:
                        break
                    i += cat
                cur_node = root

        for i in range(1, len(decoded)):
            decoded[i] += decoded[i - 1]

        return np.array(decoded)

    @staticmethod
    def restore_to_shape(decoded_data: np.ndarray, h, w, block_size=8):
        """
        Восстанавливает данные из одномерного массива в нужную форму
        (например, из (9844, 64) в (107, 92, 64)).

        Args:
            decoded_data: Декодированные данные.
            h: Высота исходного изображения.
            w: Ширина исходного изображения.
            block_size: Размер блока (8x8).

        Returns:
            Восстановленные данные.
        """
        # Для Y канала
        num_blocks_h = np.ceil(h / block_size).astype(int)
        num_blocks_w = np.ceil(w / block_size).astype(int)

        # Преобразуем одномерный массив в нужную форму
        restored_data     =     decoded_data.reshape((num_blocks_h,     num_blocks_w,
block_size**2))

        return restored_data

    def reprocess(self):
        """
        Восстанавливает изображение из закодированного файла.

        Returns:
            Tuple[np.ndarray, np.ndarray, np.ndarray]:
                Кортеж из трёх каналов (Y, Cb, Cr) в виде 2D массивов
        """
        with open(self.fn, 'rb') as file:
```

```
            h,       w,       _,       quality       =       struct.unpack('>HHBB',
file.read(struct.calcsize('>HHBB')))
            block_size = 8

            yb_cnt = int(np.ceil(h / block_size) * np.ceil(w / block_size))
            subh = np.ceil(h / 2)
            subw = np.ceil(w / 2)

            # число блоков Cb и Cr
            cbcrb_cnt = int(np.ceil(subh / block_size) * np.ceil(subw / block_size))

            dc_y = np.array(ACDC.decode(self, file, 'DC', yb_cnt, block_size))
            dc_y.resize(yb_cnt, 1)
            ac_y = np.array(ACDC.decode(self, file, 'AC', yb_cnt, block_size))
            ac_y.resize(yb_cnt, block_size**2 - 1)

            dc_cb = np.array(ACDC.decode(self, file, 'DC', cbcrb_cnt, block_size))
            dc_cb.resize(cbcrb_cnt, 1)
            ac_cb = np.array(ACDC.decode(self, file, 'AC', cbcrb_cnt, block_size))
            ac_cb.resize(cbcrb_cnt, block_size**2 - 1)

            dc_cr = np.array(ACDC.decode(self, file, 'DC', cbcrb_cnt, block_size))
            dc_cr.resize(cbcrb_cnt, 1)
            ac_cr = np.array(ACDC.decode(self, file, 'AC', cbcrb_cnt, block_size))
            ac_cr.resize(cbcrb_cnt, block_size**2 - 1)

            y = self.merge_dc_ac(dc_y, ac_y)
            cb = self.merge_dc_ac(dc_cb, ac_cb)
            cr = self.merge_dc_ac(dc_cr, ac_cr)

            y = self.restore_to_shape(y, h, w)

            y1, y2 = y.shape[0], y.shape[1]

            cb = self.restore_to_shape(cb, h // 2 + (y1 % 2), w // 2 + (y2 % 2))
            cr = self.restore_to_shape(cr, h // 2 + (y1 % 2), w // 2 + (y2 % 2))

            return y, cb, cr, block_size, quality
```

8) Coder/Converter.py – файл с реализацией конвертации изображений в разные виды:

```
from .ACDC import *


class Converter:
    """
    Класс предназначенный для конвертации изображений из одного формата в другой
    """
    def __init__(self):
        pass

    @staticmethod
    def save_raw(img_path: str) -> None:
        """
        Сохраняет изображение в формате .raw с тем же именем.

        Args:
            img_path (str): Путь к исходному изображению.
        """
```

```
        from os.path import splitext
        import os

        img = Image.open(img_path).convert('RGB')
        img_array = np.array(img, dtype=np.uint8)

        # Получаем имя без расширения
        root, _ = splitext(img_path)
        raw_path = root + '.raw'

        # Сохраняем байты изображения (в порядке RGB)
        with open(raw_path, 'wb') as f:
            f.write(img_array.tobytes())

    def RGB2YCbCr(self, img: Union[Image.Image, str, np.ndarray],
                  split_channels: bool = False,
                  return_type: str = 'array',
                  out_path: str = None) -> Union[Image.Image, np.ndarray,
                                                 Tuple[np.ndarray,     np.ndarray,
np.ndarray], None]:
        """
        Переводит изображение из RGB в YCbCr цветовое пространство.

        Args:
            img   (Union[Image.Image,   str,   np.ndarray]):   Изображение   PIL.Image,
np.ndarray или путь к файлу изображения.
            split_channels (bool, optional):
                Если True, возвращает отдельные каналы Y, Cb и Cr в виде кортежа
NumPy массивов.
                Если False, возвращает объединённое изображение. По умолчанию False.
            return_type (str, optional):
                Тип возвращаемого результата при объединении каналов:
                'array' (NumPy массив) или 'image' (PIL.Image). По умолчанию 'array'.
                Не используется, если separate_channels=True.
            out_path (str, optional): Путь для сохранения результата в файл. По
умолчанию None (не сохранять).

        Returns:
            Union[Image.Image, np.ndarray, Tuple[np.ndarray, np.ndarray, np.ndarray],
None]:
            Преобразованное изображение в формате массива, изображения или кортежа
каналов.
        """
        if isinstance(img, str):
            self.save_raw(img)
            img = Image.open(img)
        if isinstance(img, Image.Image):
            if img.mode in ('L', '1'):
                img = img.convert('RGB')
            img = np.array(img).astype(np.float32)

        r = img[:, :, 0]
        g = img[:, :, 1]
        b = img[:, :, 2]

        y = 16 + (65.481 * r + 128.553 * g + 24.966 * b) / 255
        cb = 128 + (-37.797 * r - 74.203 * g + 112.0 * b) / 255
        cr = 128 + (112.0 * r - 93.786 * g - 18.214 * b) / 255

        if split_channels:
            return y, cb, cr
```

```
        ycbcr = np.stack((y, cb, cr), axis=-1)
        ycbcr = np.clip(ycbcr, 0, 255)
        ycbcr = ycbcr.astype(np.uint8)

        ycbcr_img = Image.fromarray(ycbcr, 'RGB')
        if out_path:
            from os.path import splitext
            import os
            root, ext = splitext(out_path)
            ext = ext.lower()
            save_ext = ext if ext in ['.png', '.jpg', '.jpeg'] else '.png'
            temp_path = root + save_ext
            ycbcr_img.save(temp_path)
            if temp_path != out_path:
                os.replace(temp_path, out_path)


        if return_type.lower() == 'image':
            return ycbcr_img

        return ycbcr

    @staticmethod
    def YCbCr2RGB(img: Union[Image.Image, str, np.ndarray],
                  split_channels = False,
                  return_type: str = 'array',
                  out_path: str = None) -> Union[Image.Image, np.ndarray,
                                                 Tuple[np.ndarray,     np.ndarray,
np.ndarray], None]:
        """
        Переводит изображение из YCbCr обратно в RGB цветовое пространство.
        Args:
            img  (Union[Image.Image,  str]):  Изображение  PIL.Image  или  путь  к
изображению.
            split_channels (bool, optional):
                Если True, возвращает отдельные каналы R, G, B в виде кортежа NumPy
массивов.
                Если False, возвращает объединённое изображение. По умолчанию False.
            return_type (str, optional): Тип возвращаемого результата: 'array' (NumPy
массив) или 'image' (PIL.Image).
                                          По умолчанию 'array'.
            out_path  (str,  optional):  Путь  для  сохранения  результата  в  файл.  По
умолчанию None (не сохранять).
        Returns:
            Union[Image.Image, np.ndarray, Tuple[np.ndarray, np.ndarray, np.ndarray],
None]: Преобразованное изображение в формате массива или изображения.
        """
        if not isinstance(img, np.ndarray):
            if isinstance(img, str):
                img = Image.open(img)
            img = np.array(img).astype(np.float32)

        y = img[:, :, 0]
        cb = img[:, :, 1]
        cr = img[:, :, 2]

        r = 1.164 * (y - 16) + 1.596 * (cr - 128)
        g = 1.164 * (y - 16) - 0.392 * (cb - 128) - 0.813 * (cr - 128)
        b = 1.164 * (y - 16) + 2.017 * (cb - 128)
```

```
        if split_channels:
            return r, g, b

        rgb = np.stack((r, g, b), axis=-1)
        rgb = np.clip(rgb, 0, 255)
        rgb = rgb.astype(np.uint8)

        rgb_img = Image.fromarray(rgb, 'RGB')
        if out_path:
            from os.path import splitext
            import os
            root, ext = splitext(out_path)
            ext = ext.lower()
            save_ext = ext if ext in ['.png', '.jpg', '.jpeg'] else '.png'
            temp_path = root + save_ext
            rgb_img.save(temp_path)
            if temp_path != out_path:
                os.replace(temp_path, out_path)


        if return_type.lower() == 'image':
            return rgb_img

        return rgb

    def show_images(self, img: Union[Image.Image, str, np.ndarray]) -> None:
        """
        Отображает изображение RGB и YCbCr, а также каналы R, G, B, Y, Cb, Cr в
цвете.

        Args:
            img (Union[Image.Image, str, np.ndarray]): Изображение PIL.Image, путь
или массив.

        Returns:
            None
        """
        if isinstance(img, str):
            img = Image.open(img)
        if isinstance(img, Image.Image):
            img = np.array(img)

        # RGB каналы
        r = img[:, :, 0]
        g = img[:, :, 1]
        b = img[:, :, 2]

        r_img      =      np.stack([r,      np.zeros_like(r),      np.zeros_like(r)],
axis=2).astype(np.uint8)
        g_img      =      np.stack([np.zeros_like(g),      g,      np.zeros_like(g)],
axis=2).astype(np.uint8)
        b_img      =      np.stack([np.zeros_like(b),      np.zeros_like(b),      b],
axis=2).astype(np.uint8)

        # YCbCr каналы
        ycbcr = self.RGB2YCbCr(img)
        y, cb, cr = ycbcr[:, :, 0], ycbcr[:, :, 1], ycbcr[:, :, 2]
        y_img = np.stack([y, y, y], axis=2).astype(np.uint8)

        # для Cb и Cr применим сдвиг + визуализацию в синих/красных оттенках
        cb_img = np.stack([np.zeros_like(cb), 255 - cb, cb], axis=2).astype(np.uint8)
```

```
        cr_img = np.stack([cr, 255 - cr, np.zeros_like(cr)], axis=2).astype(np.uint8)

        fig, axes = plt.subplots(2, 4, figsize=(20, 10))

        titles = [
            'RGB Image', 'R Channel (Red)', 'G Channel (Green)', 'B Channel (Blue)',
            'YCbCr Image', 'Y Channel (Luma)', 'Cb Channel (Blue diff)', 'Cr Channel
(Red diff)'
        ]
        images = [
            img, r_img, g_img, b_img,
            ycbcr, y_img, cb_img, cr_img
        ]

        for ax, image, title in zip(axes.flat, images, titles):
            ax.imshow(image)
            ax.set_title(title)
            ax.axis('off')

        plt.tight_layout()
        plt.show()
```

## 9) Coder/Huffman.py – файл для построения графика коэффициента LZSS:

```
from Main.imports import *


class Huffman:
    """
    Реализует алгоритм Хаффмана для построения деревьев и генерации кодов сжатия
данных.

    Включает внутренний класс Node для представления узлов дерева.

    Атрибуты:
        reps_size (int): Размер блока для группового кодирования (по умолчанию 4)
        count_size (int): Размер счетчика в битах (по умолчанию 2)
        step (int): Шаг обработки данных (рассчитывается как reps_size + 1)

    Методы:
        build_tree: Строит дерево Хаффмана по частотному словарю
        build_code: Генерирует коды Хаффмана для всех символов
    """
    class Node:
        """
        Внутренний класс для представления узла дерева Хаффмана.

        Атрибуты:
            value: Значение символа (None для внутренних узлов)
            freq: Частота символа/суммарная частота поддерева
            left: Левый потомок
            right: Правый потомок
        """
        def __init__(self, value=None, freq=0, left=None, right=None):
            self.value = value
            self.freq = freq
            self.left = left
            self.right = right

        def __lt__(self, other):
```

```
            return self.freq < other.freq

    def __init__(self, reps_size=4):
        self.count_size = 2
        self.reps_size = reps_size
        self.step = self.reps_size + 1

    @classmethod
    def build_tree(cls, freq_dict: dict):
        """
        Строит дерево Хаффмана на основе частотного словаря.

        Args:
            freq_dict (dict): Словарь частот в формате {символ: частота}

        Returns:
            Node: Корневой узел построенного дерева
        """
        nodes = [cls.Node(value=i, freq=freq_dict[i]) for i in freq_dict.keys() if
freq_dict[i]]
        nodes.sort(key=lambda x: x.value)
        nodes.sort(key=lambda x: x.freq)
        heapq.heapify(nodes)

        while len(nodes) != 1:
            left = heapq.heappop(nodes)
            right = heapq.heappop(nodes)
            heapq.heappush(nodes, cls.Node(value=None, freq=left.freq + right.freq,
left=left, right=right))
        return heapq.heappop(nodes)

    @classmethod
    def build_code(cls, node, prefix="", code_dict=None):
        """
        Генерирует коды Хаффмана для всех символов в дереве.

        Args:
            node (Node): Текущий обрабатываемый узел дерева
            prefix (str, optional): Текущий префикс кода. По умолчанию ""
            code_dict (dict, optional): Словарь для накопления кодов. По умолчанию
None

        Returns:
            dict: Словарь кодов в формате {символ: битовая_строка}
        """
        if code_dict is None:
            code_dict = dict()
        if node:
            if node.value is not None:
                code_dict[node.value] = prefix
            else:
                cls.build_code(node.left, prefix + '0', code_dict)
                cls.build_code(node.right, prefix + '1', code_dict)
        return code_dict
```

10) .resume/graphs/graphs.py – файл для построения всех графиков:

```
from Main.Main import *
from PIL import Image
import os
import matplotlib.pyplot as plt
```

```
from tqdm import tqdm
from collections import defaultdict


def generate_image_versions(input_path):
    img = Image.open(input_path)

    output_dir = os.path.dirname(input_path)
    filename = os.path.splitext(os.path.basename(input_path))[0]

    gray_img = img.convert("L")
    gray_path = os.path.join(output_dir, f"{filename}_grayscale.png")
    gray_img.save(gray_path)
    print(f"Saved grayscale: {gray_path}")

    bw_img = gray_img.convert("1")
    bw_path = os.path.join(output_dir, f"{filename}_bw_no_dither.png")
    bw_img.save(bw_path)
    print(f"Saved BW without dithering: {bw_path}")

    dithered_img = gray_img.convert("1", dither=Image.Dither.FLOYDSTEINBERG)
    dithered_path = os.path.join(output_dir, f"{filename}_bw_dithered.png")
    dithered_img.save(dithered_path)
    print(f"Saved BW with dithering: {dithered_path}")


def two298(dir: str):
    comp_dir = os.path.join(dir, 'compressed')
    decomp_dir = os.path.join(dir, 'decompressed')

    os.makedirs(comp_dir, exist_ok=True)
    os.makedirs(decomp_dir, exist_ok=True)

    for img_name in os.listdir(dir):
        img_path = os.path.join(dir, img_name)

        if  os.path.isfile(img_path)  and  img_path.lower().endswith(('.png',  '.jpg',
'.jpeg')):
            base_name = os.path.splitext(img_name)[0]

            img_comp_dir = os.path.join(comp_dir, base_name)
            img_decomp_dir = os.path.join(decomp_dir, base_name)
            os.makedirs(img_comp_dir, exist_ok=True)
            os.makedirs(img_decomp_dir, exist_ok=True)

            sizes = []
            qualities = list(range(2, 100, 2))

            for q in tqdm(qualities, f'Processing {base_name}'):
                e_zmn_name = f"e_{base_name}_{q}.zmn"
                e_zmn_path = os.path.join(img_comp_dir, e_zmn_name)

                d_zmn_name = f"d_{base_name}_{q}.png"
                d_zmn_path = os.path.join(img_decomp_dir, d_zmn_name)

                compressor = Main(e_zmn_path)
                compressor.encode(img_path, quality=q)

                file_size = os.path.getsize(e_zmn_path)
                sizes.append(file_size / 1024)
```

```
                # print(e_zmn_path, d_zmn_path)

                compressor.decode(d_zmn_path)



def plot_compression_sizes(
    base_path="Z:/prog/jpeg/.resume/test_imgs",
    output_path="Z:/prog/jpeg/.resume/imgs"
):
    os.makedirs(output_path, exist_ok=True)

    raw_path = os.path.join(base_path)
    compressed_base = os.path.join(base_path, "compressed")

    for file in os.listdir(raw_path):
        if not file.endswith('.raw'):
            continue

        name = os.path.splitext(file)[0]
        compressed_folder = os.path.join(compressed_base, name)

        sizes = []
        qualities = []

        for q in range(2, 100, 2):
            fname = f"e_{name}_{q}.zmn"
            fpath = os.path.join(compressed_folder, fname)
            if os.path.isfile(fpath):
                sizes.append(os.path.getsize(fpath) / 1024)  # размер в КБ
                qualities.append(q)
            else:
                print(f"[!] Missing: {fpath}")

        if not sizes:
            print(f"[!] No compressed files found for {name}")
            continue

        # Строим график
        plt.figure(figsize=(10, 6))
        plt.plot(qualities, sizes, marker='o')
        plt.title(f"Размер файла vs Качество JPEG для {name}")
        plt.xlabel("Качество JPEG")
        plt.ylabel("Размер файла (КБ)")
        plt.grid(True)
        plt.tight_layout()

        # Сохраняем
        out_path = os.path.join(output_path, f"{name}_compression_plot.png")
        plt.savefig(out_path)
        plt.close()

        print(f"[ok] Saved plot for {name} -> {out_path}")


def zero2hundred(dir: str):
    comp_dir = dir
    decomp_dir = dir

    os.makedirs(comp_dir, exist_ok=True)
    os.makedirs(decomp_dir, exist_ok=True)
```

```
    for img_name in os.listdir(dir):
        img_path = os.path.join(dir, img_name)

        if  os.path.isfile(img_path)  and  img_path.lower().endswith(('.png',  '.jpg',
'.jpeg')):
            base_name = os.path.splitext(img_name)[0]

            sizes = []
            qualities = list(range(0, 101, 20))

            for q in tqdm(qualities, f'Processing {base_name}'):
                e_zmn_name = f"{base_name}_{q}.zmn"
                e_zmn_path = os.path.join(comp_dir, e_zmn_name)

                d_zmn_name = f"{base_name}_{q}.png"
                d_zmn_path = os.path.join(decomp_dir, d_zmn_name)

                compressor = Main(e_zmn_path)
                compressor.encode(img_path, quality=q)

                file_size = os.path.getsize(e_zmn_path)
                sizes.append(file_size / 1024)

                # print(e_zmn_path, d_zmn_path)

                compressor.decode(d_zmn_path)
                os.remove(e_zmn_path)


def make_multiple_collages(folder: str, out_dir: str, figsize=(12, 8)) -> None:
    """
    Строит 2x3 коллажи для каждого набора изображений NAME_Q.png с Q ∈ [0, 20, ...,
100].

    Args:
        folder (str): Путь к папке, содержащей изображения.
        out_dir (str): Куда сохранять коллажи.
        figsize (tuple): Размер коллажа (ширина, высота).
    """
    os.makedirs(out_dir, exist_ok=True)

    # Сбор изображений по NAME
    name_to_files = defaultdict(list)
    for f in os.listdir(folder):
        if not f.lower().endswith('.png'):
            continue
        if '_' not in f:
            continue
        name, q_ext = f.rsplit('_', 1)
        q = q_ext.split('.')[0]
        try:
            q = int(q)
            name_to_files[name].append((q, os.path.join(folder, f)))
        except ValueError:
            continue  # Если Q не число — пропускаем

    for name, items in name_to_files.items():
        # Сортируем по значению качества
        items_sorted = sorted(items, key=lambda x: x[0])
        if len(items_sorted) != 6:
```

```
            print(f"[!] Пропущено {name} — найдено {len(items_sorted)} изображений,
ожидалось 6")
            continue

        fig, axes = plt.subplots(2, 3, figsize=figsize)
        for ax, (q, path) in zip(axes.flat, items_sorted):
            img = Image.open(path)
            ax.imshow(img)
            ax.set_title(f"{name}, Q={q}", fontsize=10)
            ax.axis('off')

        plt.tight_layout()
        out_path = os.path.join(out_dir, f"collage_{name}.png")
        plt.savefig(out_path)
        plt.close(fig)
        print(f"[ok] Сохранён коллаж: {out_path}")



if __name__ == "__main__":
    test_dir = r"Z:\prog\jpeg\.resume\test_imgs\from0to100"

    for img_file in ['Lenna.png', 'Big.png']:
        input_image = os.path.join(test_dir, img_file)
        generate_image_versions(input_image)

    two298(test_dir)
    plot_compression_sizes()
    zero2hundred(test_dir)
    make_multiple_collages(test_dir, r'Z:\prog\jpeg\.resume\imgs')
```

11) main.py – основной файл, в котором запускается программа

```
from Main.Main import *


if __name__ == "__main__":
    main = Main()
    main.encode(r'Z:\prog\jpeg\.resume\test_imgs\Lenna.png', 0, print_info=True)
    main.decode(print_info=True)
```

## Заключение

В ходе выполнения лабораторной работы были изучены и реализованы основные этапы сжатия изображений с использованием формата JPEG. Были рассмотрены процессы преобразования цветового пространства из RGB в YCbCr, дискретного косинусного преобразования (DCT), квантования, кодирования и декодирования. Реализована система сохранения изображений в пользовательский бинарный формат .zmn с возможностью регулировки степени сжатия.

Проведён эксперимент с различными коэффициентами качества (от 0 до 100 с шагом 20), по результатам которого построены визуальные коллажи, демонстрирующие влияние степени сжатия на качество изображений. Кроме того, построены графики зависимости размера сжатого файла от параметра качества для разных типов изображений.

Полученные результаты подтвердили эффективность JPEG-сжатия: при высоких коэффициентах качества достигается значительное уменьшение размера файла при минимальной потере визуального качества. При этом наблюдается различная чувствительность к сжатию в зависимости от содержимого и структуры изображения (цветное, градации серого, бинарное с/без дизеринга).

Лабораторная работа позволила глубже понять принципы компрессии изображений и сформировать практические навыки работы с цифровыми изображениями, кодированием и визуализацией результатов.

## Ссылка на репозиторий GitHub

https://github.com/zamnisad/jpeg