

ГУАП

КАФЕДРА № 51

ОТЧЕТ
ЗАЩИЩЕН С ОЦЕНКОЙ
ПРЕПОДАВАТЕЛЬ

ассистент

должность, уч. степень, звание

подпись, дата

А.В. Борисовская

инициалы, фамилия

ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ №3
Методы блоковой обработки при сжатии с потерями на примере стандарта
JPEG

по курсу: МУЛЬТИМЕДИЯ ТЕХНОЛОГИИ

РАБОТУ ВЫПОЛНИЛ

СТУДЕНТ ГР. №

5912

подпись, дата

Б.А. Карханин

инициалы, фамилия

Санкт-Петербург 2022

Оглавление

Оглавление	2
Цель работы:	3
Ход работы:	3
1. Дискретное косинусное преобразование.....	3
1.1. Реализация процедуры прямого и обратного ДКП для блоков $N \times N$	4
2. Квантование спектральных коэффициентов	8
2.1 Реализация процедур квантования и деквантования.....	8
2.2. Оценка влияния искажений, вносимых квантованием, на компоненты этих изображений.	15
3. Сжатие без потерь	16
3.1 Реализация процедуры кодирования квантовых коэффициентов постоянного тока DC	16
3.2 Оценка эффективности разностного кодирования	17
3.3. Реализация процедуры кодирования длинами серий (RLE).	25
3.4. Определение соотношений размеров в сжатом битовом потоке.....	26
Дополнительное задание:	28
Выводы.....	30
Листинг программы	31

Цель работы:

Изучение алгоритмов, используемых в базовом режиме стандарта JPEG, анализ статистических свойств, используемых при сжатии коэффициентов дискретного косинусного преобразования, а также получение практических навыков разработки методов блочной обработки при сжатии изображения.

Ход работы:

1. Дискретное косинусное преобразование

Процедуру выполнения ДКП нагляднее описывать в форме матричного умножения. Матрицу преобразования T принято называть ядром. Строки матрицы T состоят из векторов, образованных значениями косинусов $\sqrt{C_f} \cos(\theta_t f)$, где f – номер строки и значение частоты косинуса, C_f – нормирующий коэффициент и θ_t – положение в пространстве t -ого отсчета, для которого

$$\theta_t = \frac{(2t + 1)\pi}{2N} \quad (1.1)$$

Нормирующий коэффициент C_f зависит от частоты f и вычисляется следующим образом:

$$C_f = \begin{cases} \frac{1}{N}, & \text{если } f = 0 \\ \frac{2}{N}, & \text{иначе} \end{cases} \quad (1.2)$$

Значения матрицы T вычисляются следующим образом:

$$t_{f,t} = \sqrt{C_f} \cos\left(\frac{(2t+1)\pi}{2N} f\right) \quad (1.3)$$

Прямое и обратное ДКП для двумерного случая в матричной форме выглядят следующим образом:

$$Y = (T \cdot X) \cdot T^T \quad (1.4)$$

$$X = (T^T \cdot Y) \cdot T \quad (1.5)$$

где X и Y являются матрицами размерности $N \times N$.

1.1. Реализация процедуры прямого и обратного ДКП для блоков $N \times N$.

Необходимо реализовать процедуру прямого и обратного ДКП, а также оценить вносимые этой процедурой искажения.

Окончательные формулы прямого и обратного преобразований, используемые в стандарте JPEG выглядят следующим образом:

$$y_{k,l} = \sqrt{C_k} \sqrt{C_l} \cdot \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} x_{i,j} \cos\left(\frac{(2i+1)\pi}{2N} k\right) \cos\left(\frac{(2j+1)\pi}{2N} l\right) \quad (1.6)$$

$$x_{i,j} = \sum_{k=0}^{N-1} \sum_{l=0}^{N-1} \sqrt{C_k} \sqrt{C_l} \cdot y_{k,l} \cos\left(\frac{(2i+1)\pi}{2N} k\right) \cos\left(\frac{(2j+1)\pi}{2N} l\right) \quad (1.7)$$

где $k, l, i, j = 0, \dots, N-1$.

Результаты и оценка искажений:

- MyImage

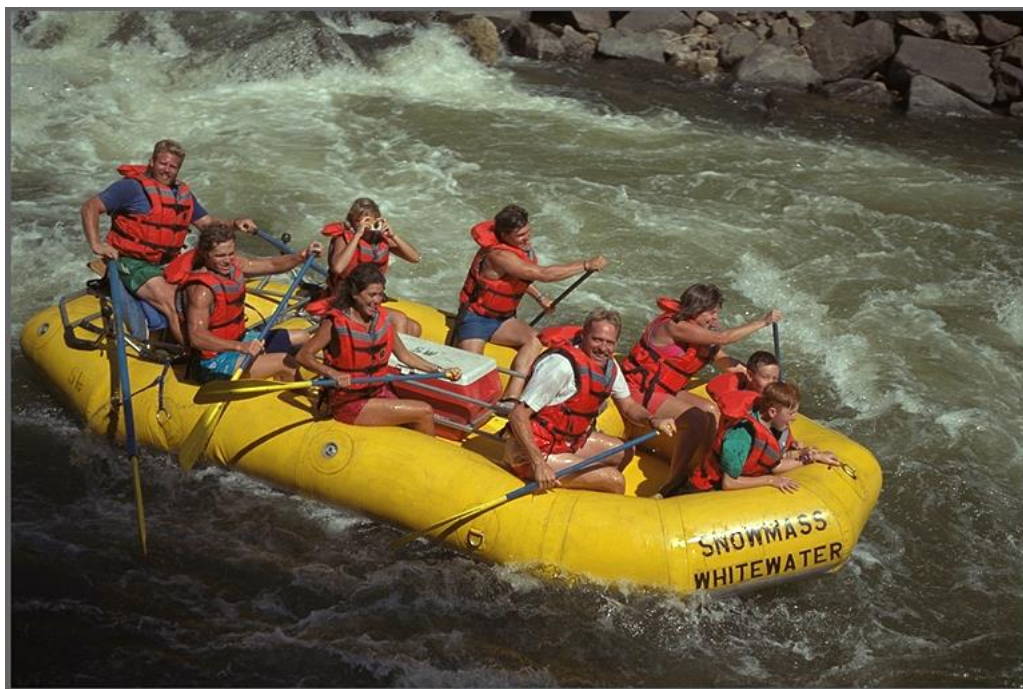


Рисунок 1 Исходное изображение (MyImage)

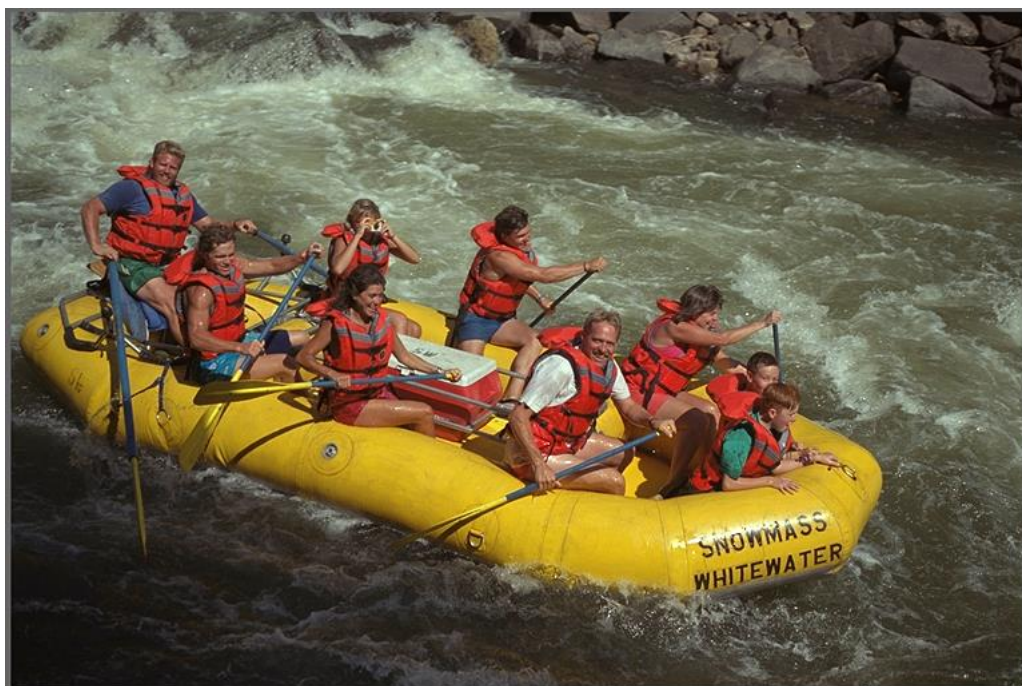


Рисунок 2 Изображение после обратного ДКП (MyImage)

Значения PSNR:

Y PSNR: 58.8951

Сb PSNR: 59.0586

Cr PSNR: 59.4169

- **baboon**

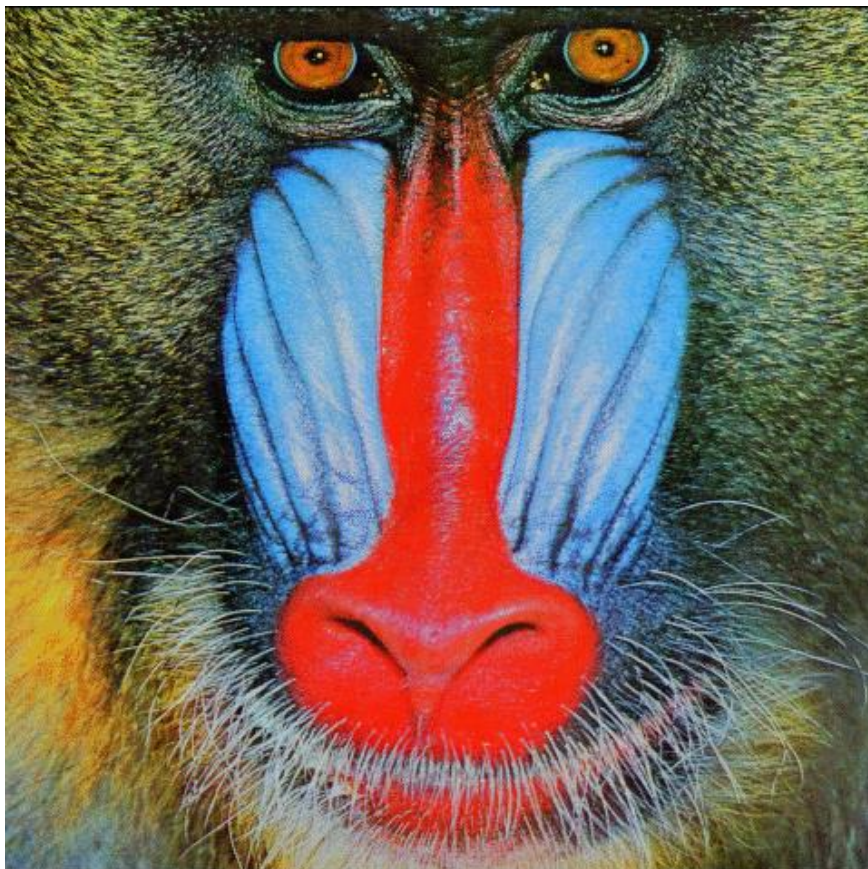


Рисунок 3 Исходное изображение (baboon)

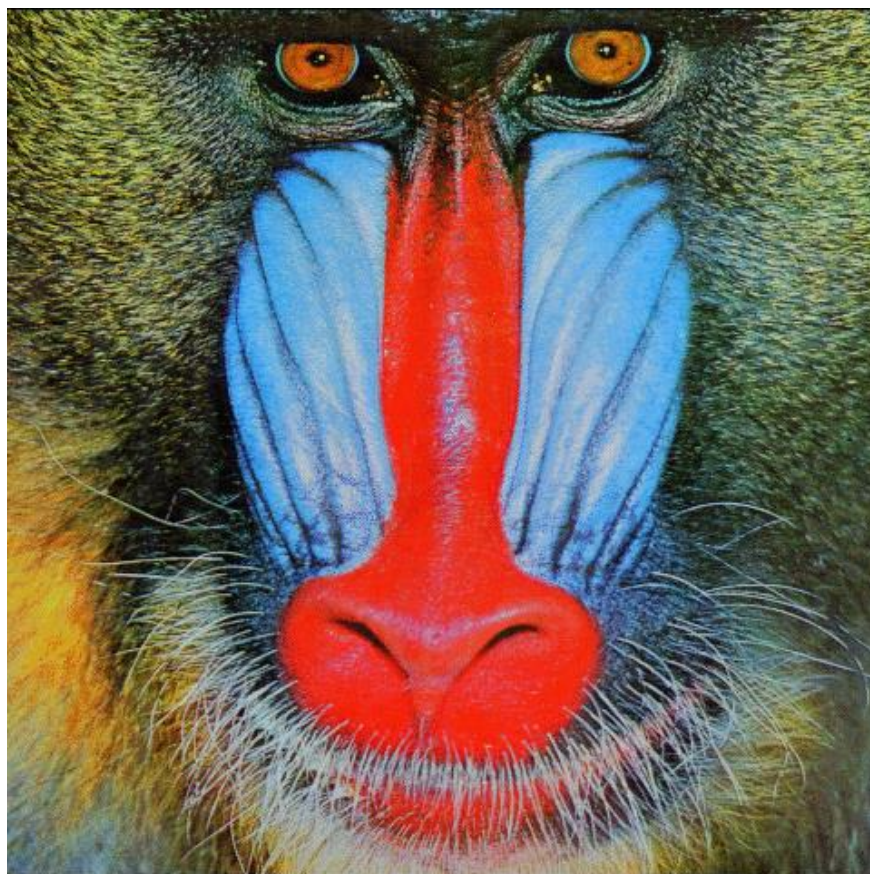


Рисунок 4 изображение после обратного ДКП (baboon)

Значения PSNR:

Y PSNR: 58.9312

Сb PSNR: 58.9111

Cr PSNR: 58.9088

- **lena**



Рисунок 5 Исходное изображение (lena)



Рисунок 6 изображение после ДКП (lena)

Значения PSNR:

Y PSNR: 58.8989
Cb PSNR: 58.9114
Cr PSNR: 58.8717

По полученным картинкам, видимых искажений после прямого и обратного ДКП не обнаружено, об этом свидетельствует и высокие значения PSNR. Небольшие потери могли вызвать только работа с дробными числами и округление до целых при записи в RGB.

2. Квантование спектральных коэффициентов

В стандарте JPEG используется равномерное скалярное квантование. Предполагается использование индивидуального шага квантования для каждой полосы (k, l) . Шаги квантования для всех полос объединяются в матрицу квантования Q , которая также имеет размерность 8×8 .

Поскольку статистики спектров цветоразностных компонент в целом похожи и сильно отличаются от спектра яркостной компоненты, на практике используют две таблицы квантования: Q^{luma} для яркостной составляющей и Q^{chroma} для двух цветоразностных компонент.

2.1 Реализация процедур квантования и деквантования

Необходимо реализовать процедуру квантования и деквантования.

Процедура квантования спектральных коэффициентов $Y_{i,j}$ определяется следующим образом:

$$Y_{i,j}^q = \text{round} \left(\frac{Y_{i,j}}{q_{i,j}^{(c)}} \right) \quad (2.1)$$

где $q_{i,j}^{(c)}$ – шаг квантования, который является элементом соответствующей матрицы $Q^{(c)}$.

Кодированию будут подвергаться именно номера квантов $Y_{i,j}^q$, а аппроксимирующие значения каждого кванта $Y_{i,j}^{dq}$ будут вычисляться при декодировании как произведение номера кванта и шага квантования:

$$Y_{i,j}^q = Y_{i,j}^{dq} \cdot q_{i,j}^{(c)} \quad (2.2)$$

где $i, j = 0, \dots, 7$.

Матрицы квантования будут строиться по формуле:

$$q_{i,j}^Y(R) = 1 + (i + r) \cdot R \quad (2.3)$$

где R — целочисленный параметр, управляющий качеством обработки.

Квантование для разных компонент происходило с помощью одинаковых матриц при значении R от 1 до 10.

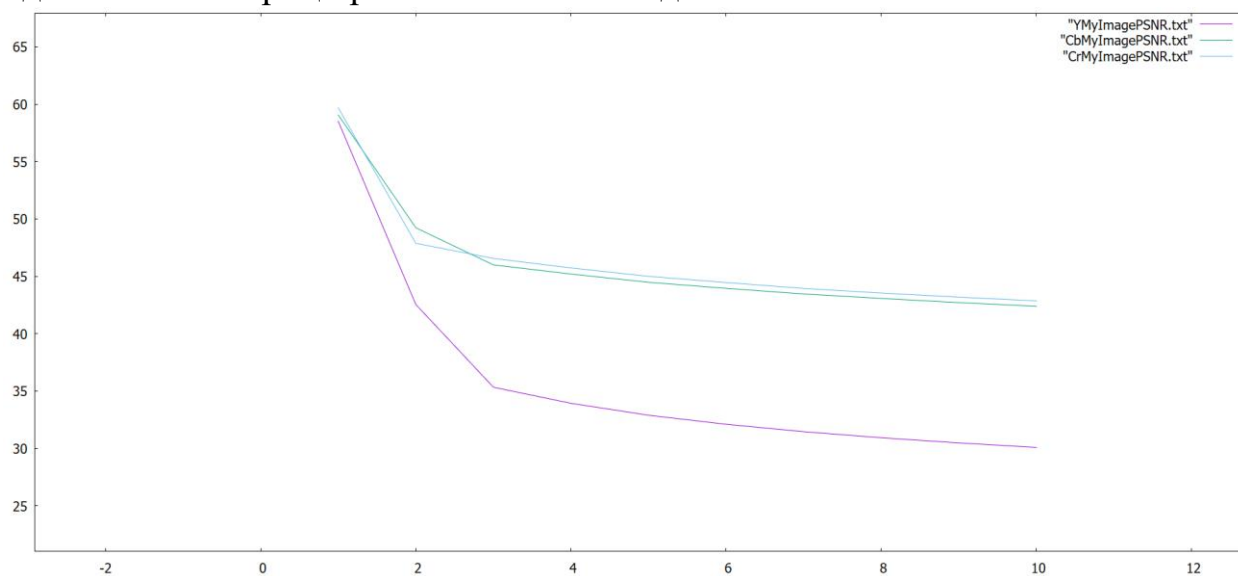


Рисунок 7 Значения PSNR при различных R для изображения MyImage

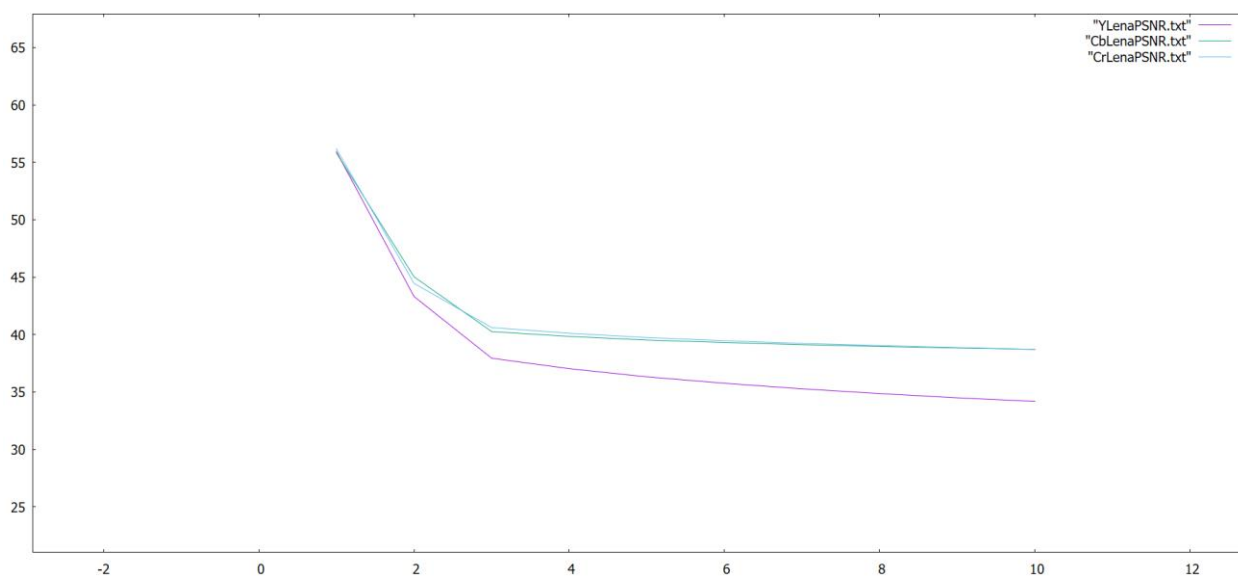


Рисунок 8 Значения PSNR при различных R для изображения Lena

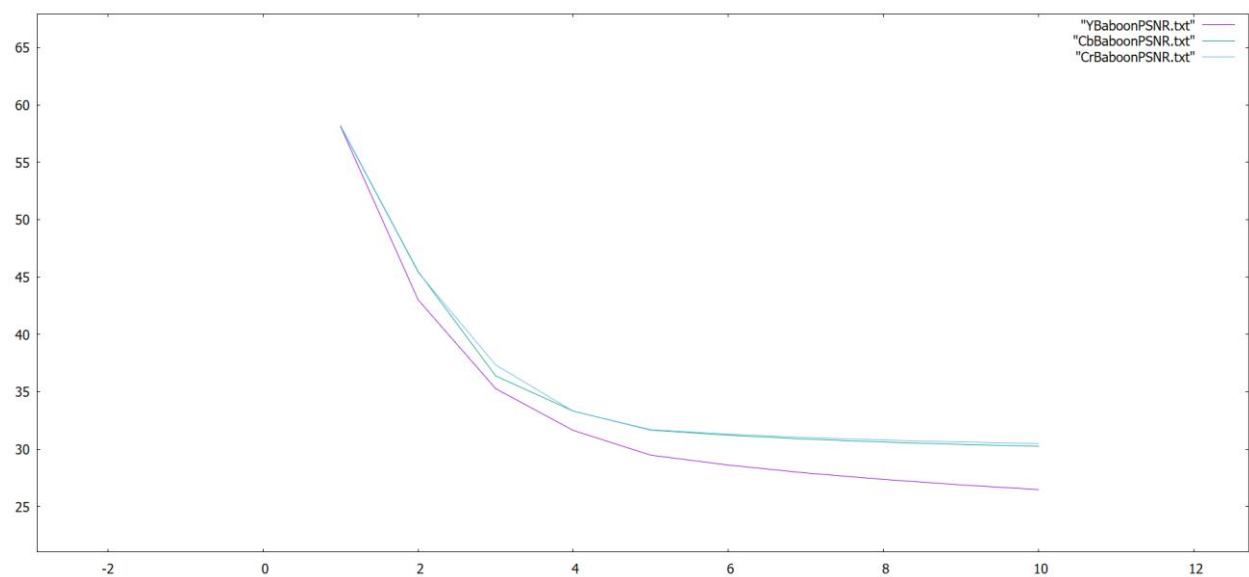


Рисунок 9 Значения PSNR при различных R для изображения Baboon

Изображения:

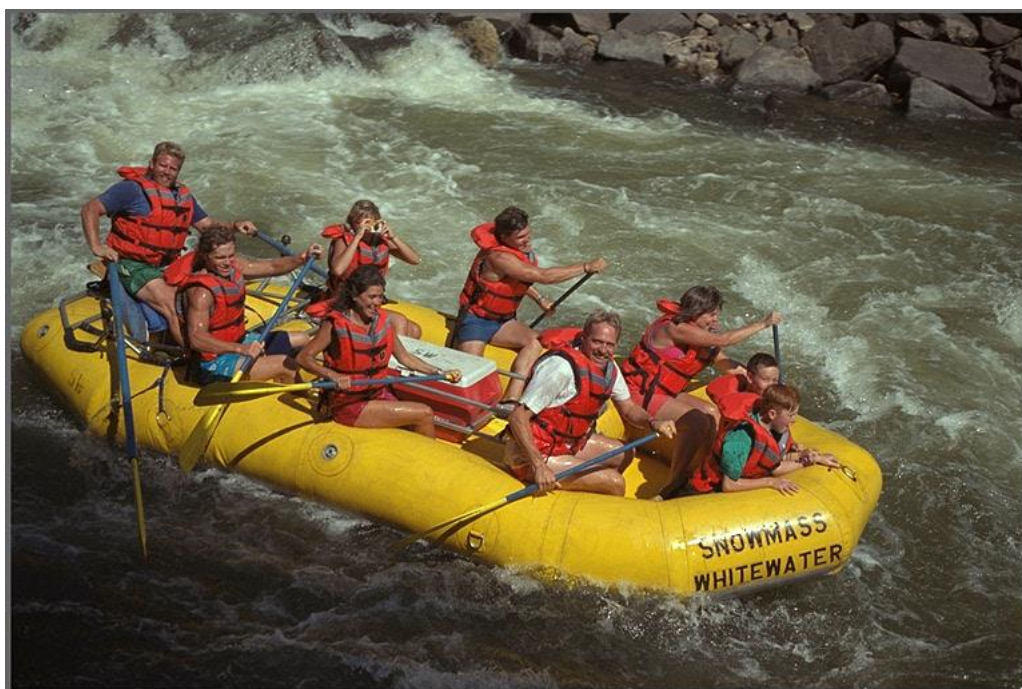


Рисунок 10 Изображение MuImage при параметре квантования $R=1$

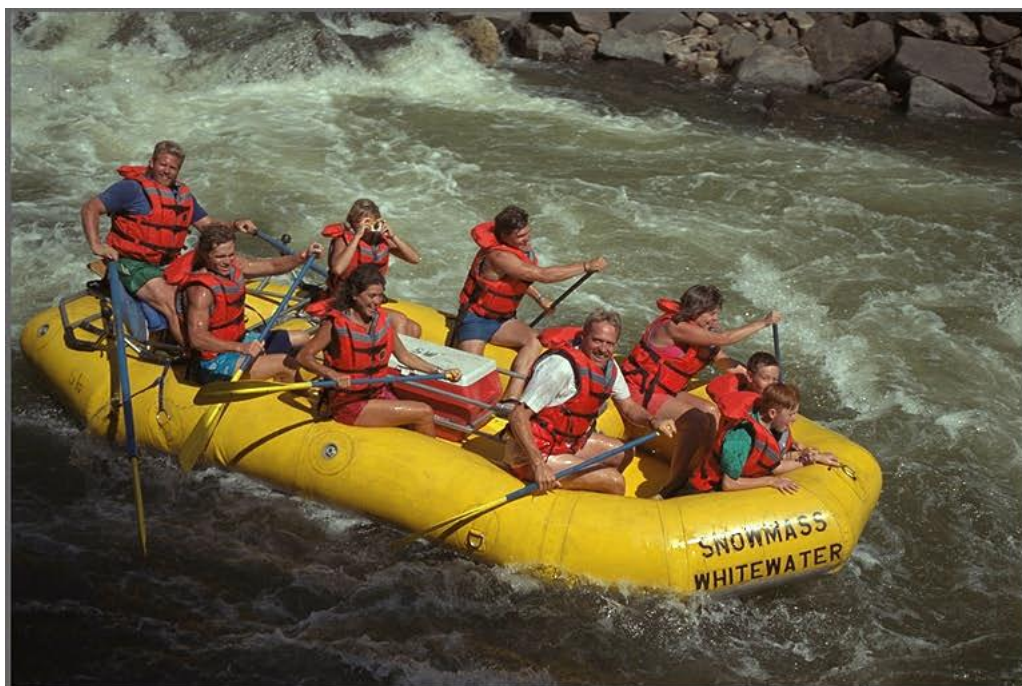


Рисунок 11 Изображение MuImage при параметре квантования $R=5$



Рисунок 12 Изображение MuImage при параметре квантования $R=10$



Рисунок 13 Изображение Lena при параметре квантования $R=1$



Рисунок 14 Изображение Lena при параметре квантования $R=5$



Рисунок 15 Изображение Лена при параметре квантования $R=10$

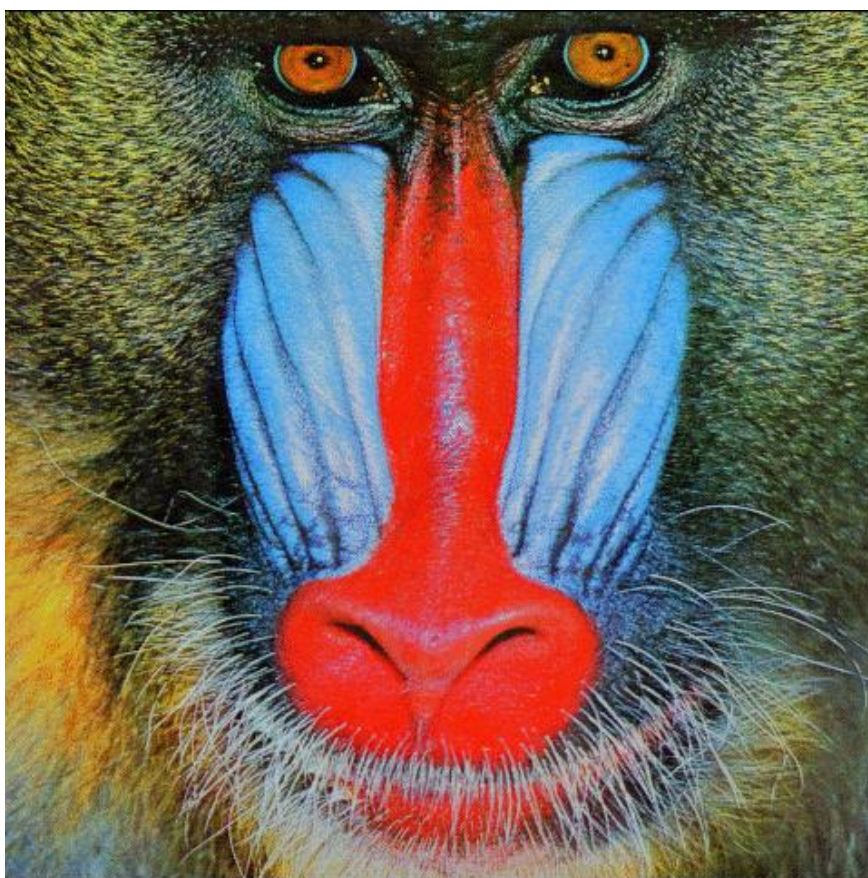


Рисунок 16 Изображение Вавооп при параметре квантования $R=1$

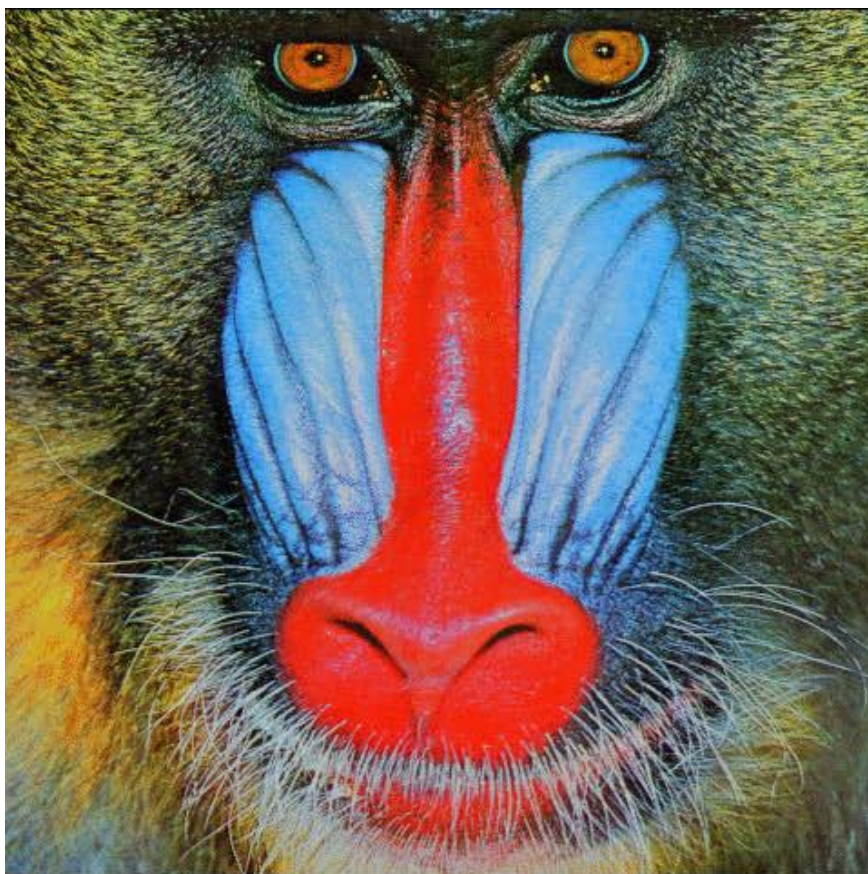


Рисунок 17 Изображение Baboon при параметре квантования $R=5$

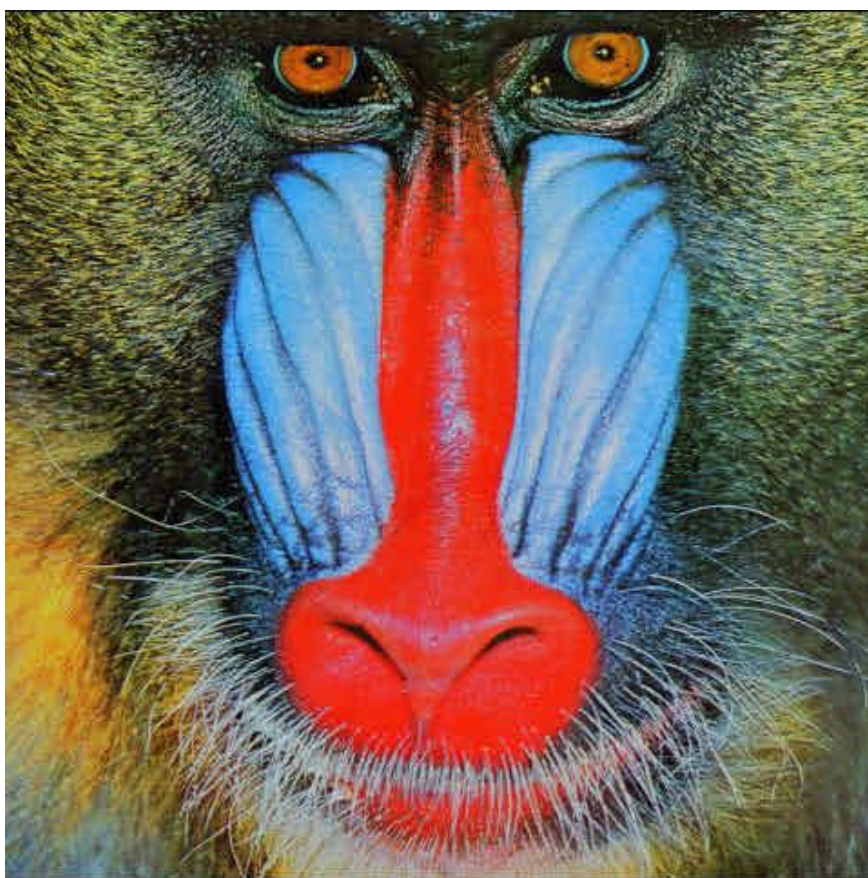


Рисунок 18 Изображение Baboon при параметре квантования $R=10$

2.2. Оценка влияния искажений, вносимых квантованием, на компоненты этих изображений.

На графиках видно, что значение PSNR для яркостной компоненты уменьшается относительно медленно с увеличением R . А для компонент C_b и C_r значение PSNR уменьшается еще медленнее. Это связано с тем, что Y содержит больше информации об изображении.

Искажения, вносимые квантованием, можно разглядеть при приближении картинок: все изображение состоит из квадратиков и из-за этого контуры изображения становятся нечеткими, что видно на рис. 19, 20 и 21.



Рисунок 19 Сравнение исходного изображения(слева) MuImage с квантованием (справа) при $R=10$ (масштаб 279)



Рисунок 20 Сравнение исходного изображения(слева) Lena с квантованием (справа) при $R=10$ (масштаб 279)



Рисунок 21 Сравнение исходного изображения(слева) Baboon с квантованием (справа) при $R=10$ (масштаб 246)

3. Сжатие без потерь

Кодирование является завершающим этапом работы кодера JPEG, на котором происходит формирование битового потока. Кодирование каждой компоненты осуществляется независимо и состоит из следующих действий:

1. Кодирование коэффициента постоянного тока DC^q .
2. Перегруппировка 63 коэффициентов переменного тока AC^q и формирование одномерного массива в соответствии с зигзагообразной последовательностью.
3. Применение кодирования длин серий для для последовательности из 63 AC^q коэффициентов.
4. Кодирование пар (Run, Level).

3.1 Реализация процедуры кодирования квантовых коэффициентов постоянного тока DC

Для кодирования DC^q используется разностный метод. Дальнейшей обработке подвергается разность DC^q коэффициента текущего и предыдущего обрабатываемого блоков:

$$\Delta DC = DC_i^q - DC_{i-1}^q, \quad (3.1)$$

где i – номер текущего блока.

Значение ΔDC представляется в форме битовой категории BC и амплитуды MG , где битовая категория от значения x вычисляется как:

$$BC(x) = \lceil \log|x| + 1 \rceil, \quad (3.2)$$

а значение амплитуды – само кодируемое значение.

3.2 Оценка эффективности разностного кодирования

Необходимо построить гистограммы частот $f(DC^q)$ и $f(\Delta DC)$, полученных для блоков 8×8 яркостной составляющей тестового изображения и вычислить оценки энтропии.

- Гистограммы частот для изображения MyImage:

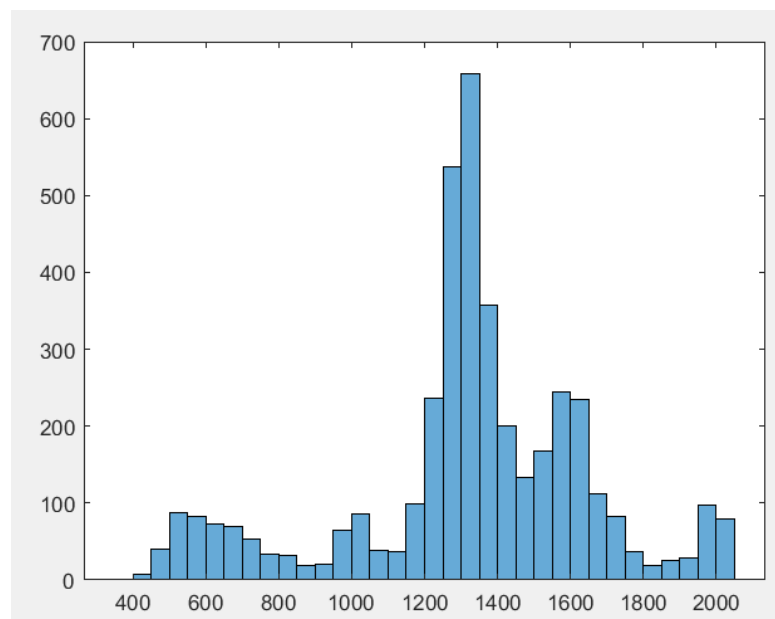


Рисунок 22 Гистограмма частот $f(DC^q)$ по Y .

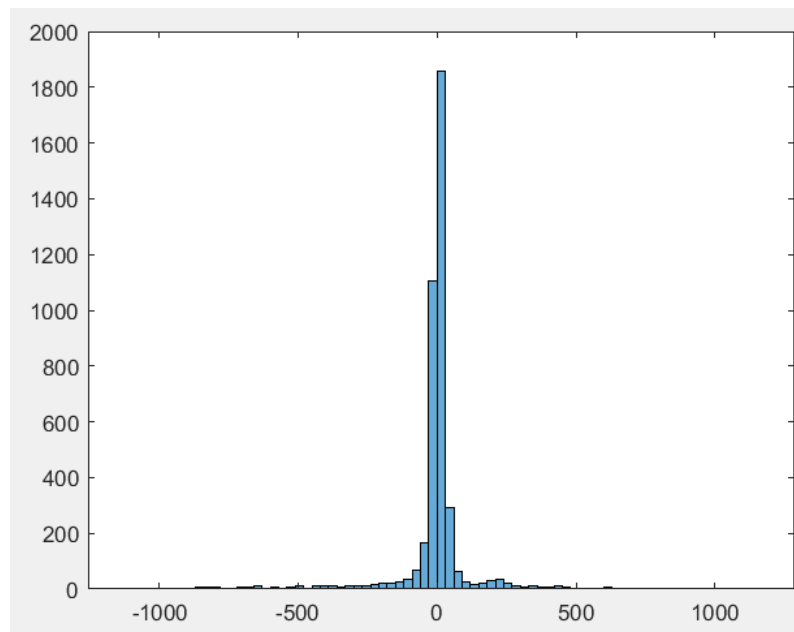


Рисунок 23 Гистограмма частот $f(\Delta DC)$ по Y .

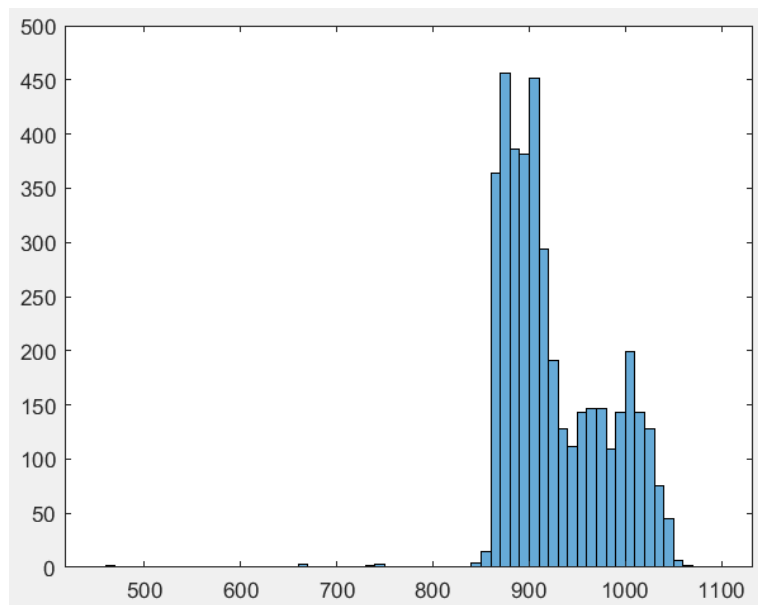


Рисунок 24 Гистограмма частот $f(DC^q)$ по Cb .

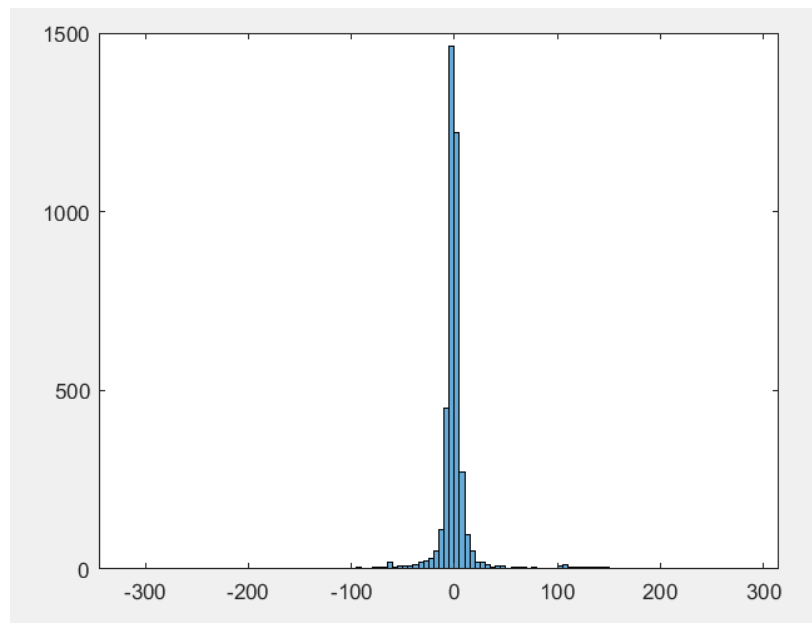


Рисунок 25 Гистограмма частот $f(\Delta DC)$ по C_b .

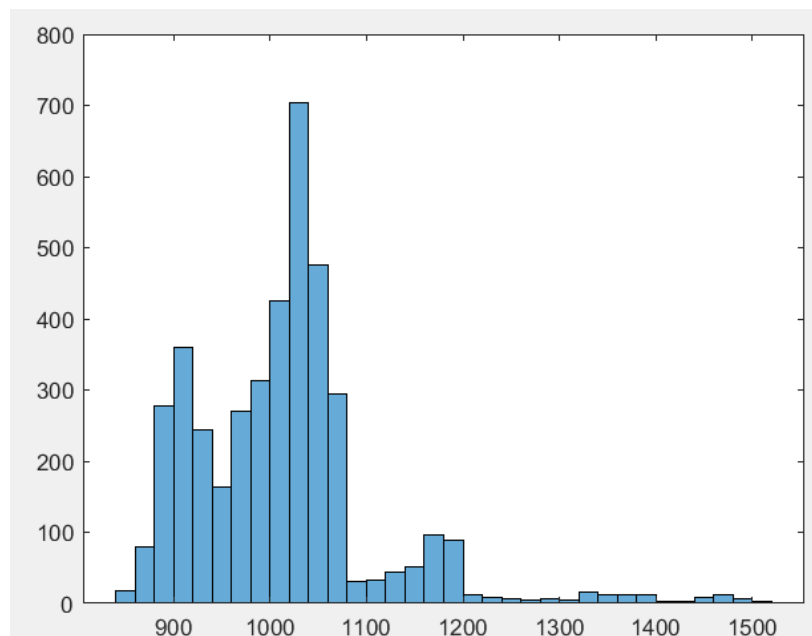


Рисунок 26 Гистограмма частот $f(DC^q)$ по C_r .

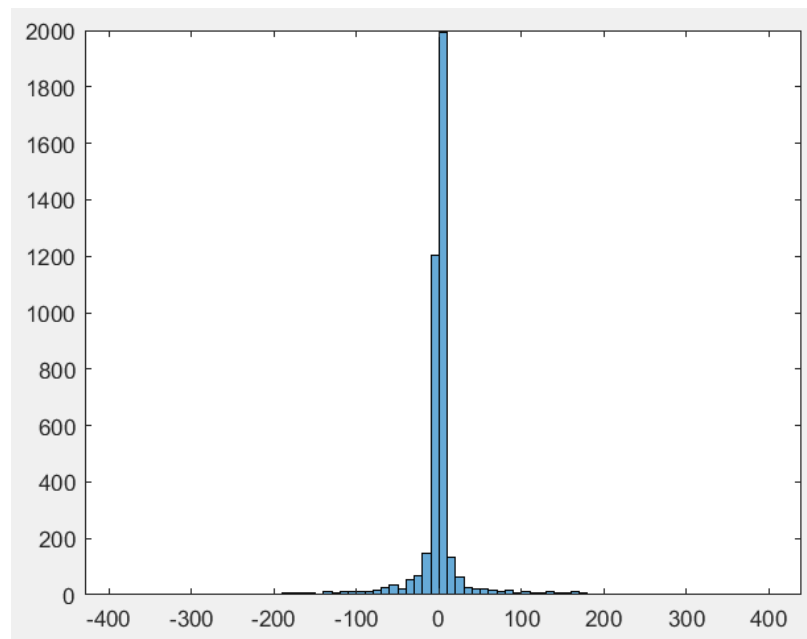


Рисунок 27 Гистограмма частот $f(\Delta DC)$ по Cr.

- Значения энтропии:

```
#3.2
MyImage:
Entropy of DC^q
Entropy of Y: 10.1353
Entropy of Cb: 8.17488
Entropy of Cr: 7.99498
Entropy of delta DC
Entropy of Y: 8.80193
Entropy of Cb: 6.80532
Entropy of Cr: 6.62843
```

Рисунок 28 Значения энтропии для изображения MyImage

- Гистограммы частот для изображения “Lena ”:

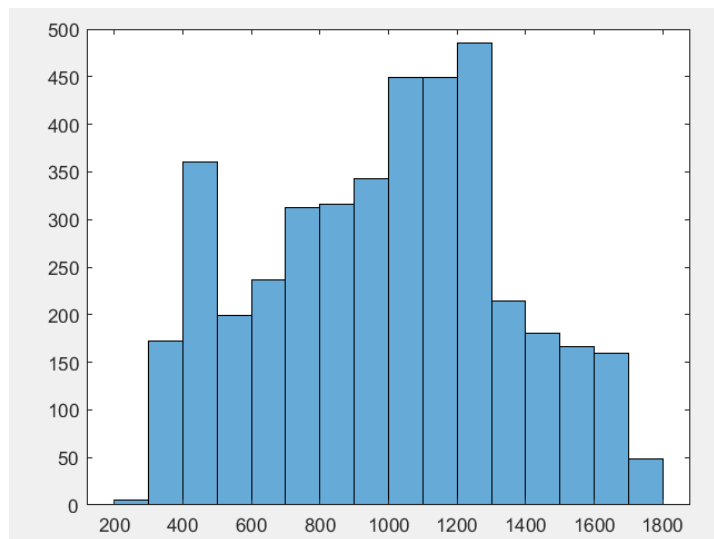


Рисунок 29 Гистограмма частот $f(DC^q)$ по Y .

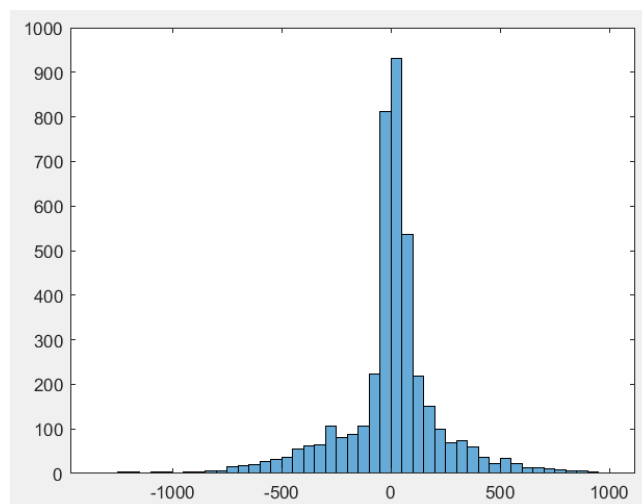


Рисунок 30 Гистограмма частот $f(ADC)$ по Y .

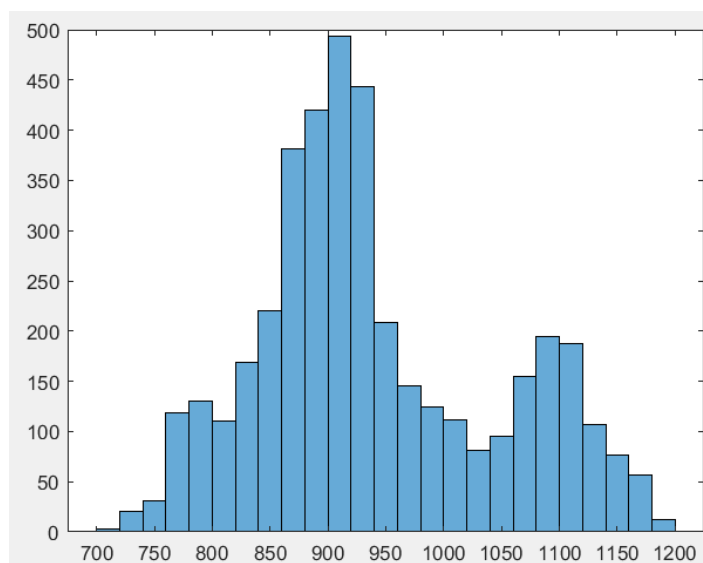


Рисунок 31 Гистограмма частот $f(DC^q)$ по Cb .

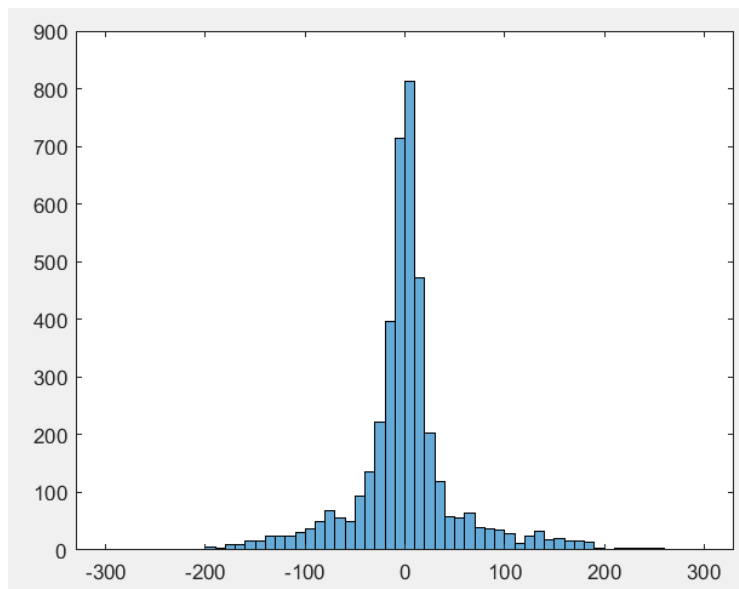


Рисунок 32 Гистограмма частот $f(\Delta DC)$ по C_b .

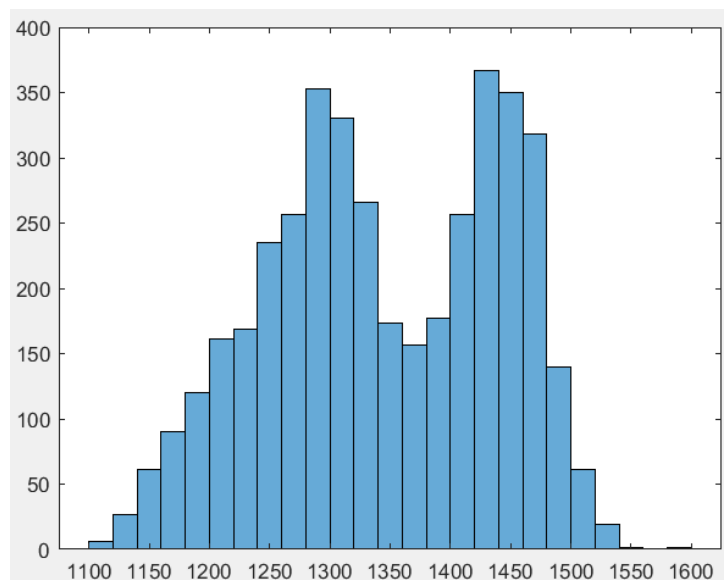


Рисунок 33 Гистограмма частот $f(DC^q)$ по C_r .

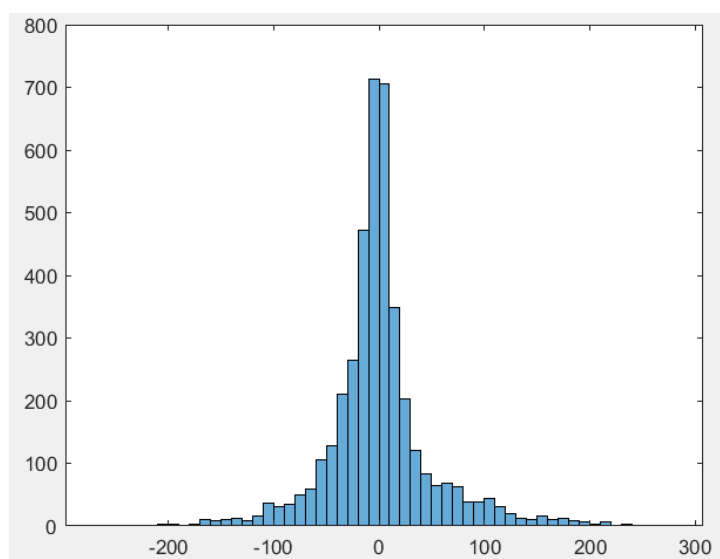


Рисунок 34 Гистограмма частот $f(\Delta DC)$ по C_r .

- Значения энтропии:

```
Lena:
Entropy of DC^q
Entropy of Y: 10.0706
Entropy of Cb: 8.42556
Entropy of Cr: 8.39537
Entropy of delta DC
Entropy of Y: 9.11451
Entropy of Cb: 7.27582
Entropy of Cr: 7.27492
```

Рисунок 35 Значения энтропии для изображения Lena

- Гистограммы частот для изображения “Baboon”:

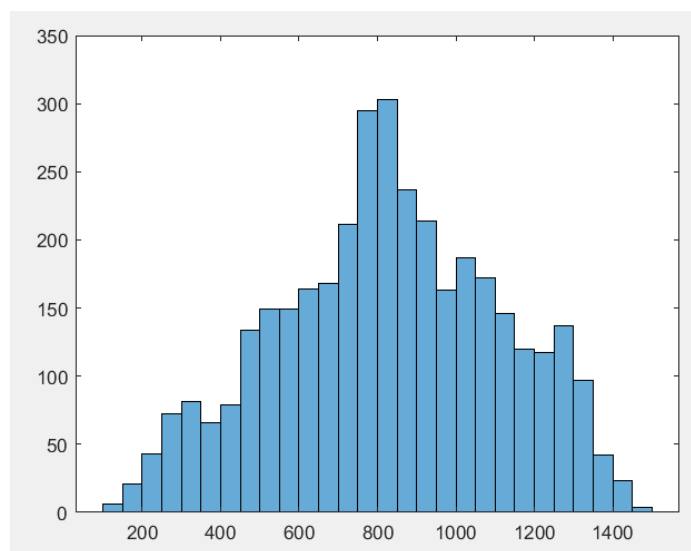


Рисунок 36 Гистограмма частот $f(DC^q)$ по Y.

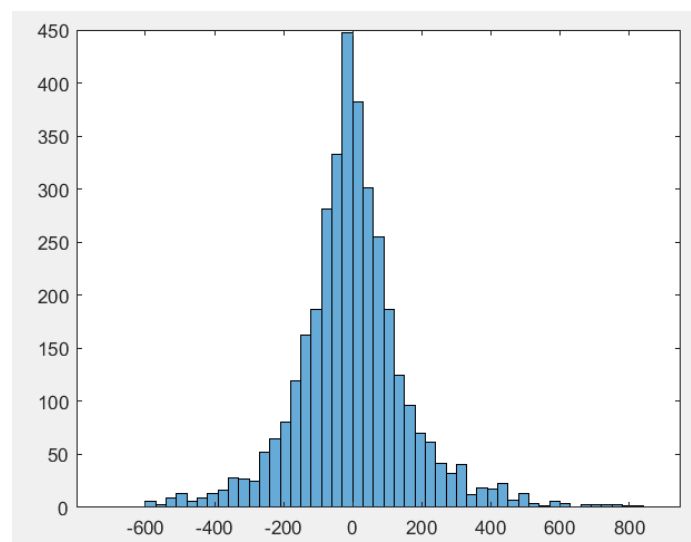


Рисунок 37 Гистограмма частот $f(ADC)$ по Y.

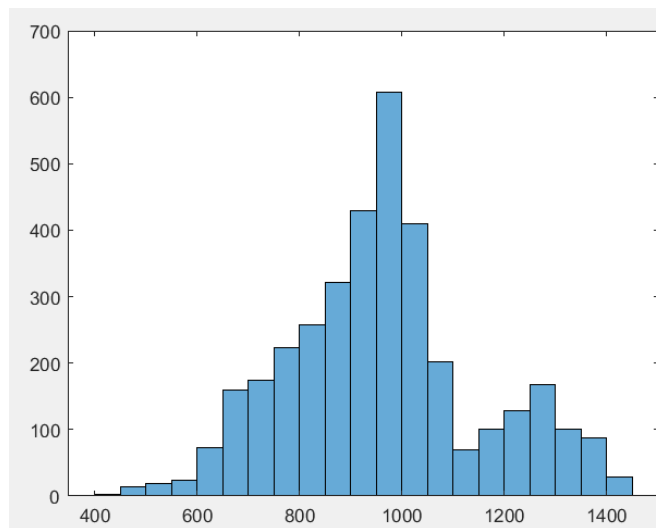


Рисунок 38 Гистограмма частот $f(DC^q)$ по Cb .

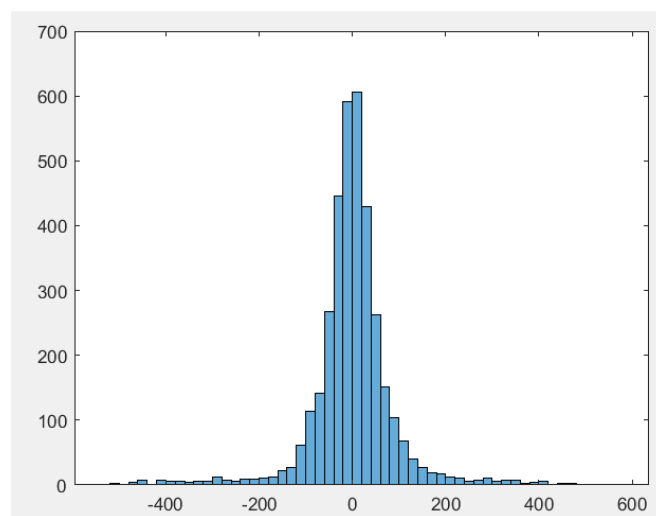


Рисунок 39 Гистограмма частот $f(\Delta DC)$ по Cb .

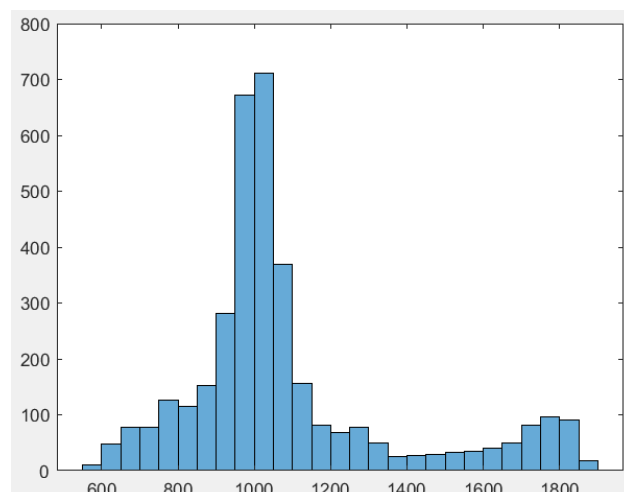


Рисунок 40 Гистограмма частот $f(DC^q)$ по Cr .

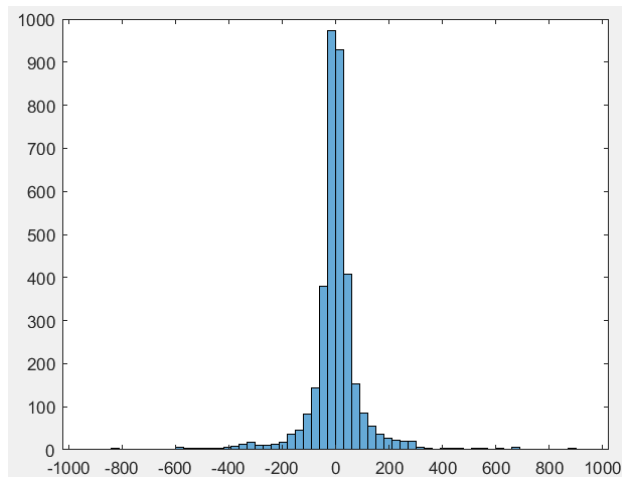


Рисунок 41 Гистограмма частот $f(\Delta DC)$ по Cr.

- Значения энтропии:

```
Baboon:
Entropy of DC^q
Entropy of Y: 9.85194
Entropy of Cb: 9.25251
Entropy of Cr: 9.24871
Entropy of delta DC
Entropy of Y: 9.0228
Entropy of Cb: 7.99137
Entropy of Cr: 7.94209
```

Рисунок 42 Значения энтропии для изображения Baboon

По этим данным можно сделать вывод, что разностное кодирование уменьшает значение энтропии, из чего следует, что качество сжатия лучше. Это связано с тем, что при разностном кодировании значительно увеличивается количество нулей и значений около нуля для массива ΔDC , что видно из гистограмм.

3.3. Реализация процедуры кодирования длинами серий (RLE).

Сформированная в предыдущем пункте последовательность коэффициентов AC^q необходимо закодировать длинами серий. Данный этап кодирования состоит из 3 частей:

- 1) Перегруппировка коэффициентов переменного тока AC^q в соответствии с зигзагообразной последовательность.
- 2) Этап кодирования длин серии, в результате которого предыдущая последовательность заменяется на новую последовательность, состоящую из пар (Run, Level). Run определяет число нулевых значений в серии, а Level — ненулевой завершающий элемент серии. Если в

старой последовательности не остаётся ненулевых значений, то в конце новой последовательности ставится пара (0,0).

3) Значение Level заменяется на пару $BC(Level)$, $Magnitude(Level)$.

3.4. Определение соотношений размеров в сжатом битовом потоке.

Необходимо определить соотношение размеров в сжатом битовом потоке для следующих данных:

$BC(\Delta DC)$, - постоянный ток

$Magnitude(\Delta DC)$, - значение битов в постоянном токе

$(Run, BC(Level))$, -число нулей, значение 1-го ненулевого элемента

$Magnitude(Level)$. – значение битов, необходимое для 1-го ненулевого

Результат работы для изображения MyImage:

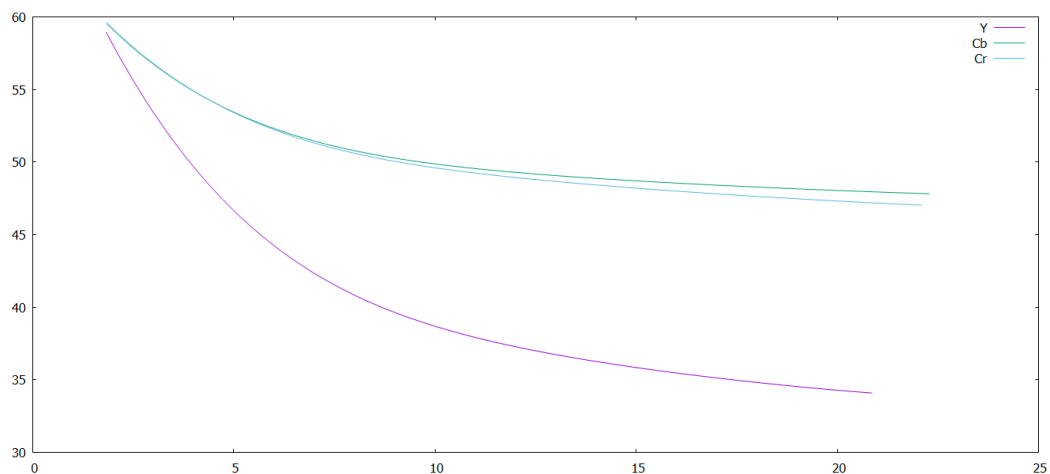


Рисунок 43 Зависимость PSNR от степени сжатия

Результаты вычисления соотношений:

R = 1 Y: $BC(\Delta DC)$: 1.94029% $Magnitude(\Delta DC)$: 2.8394% $(Run, BC(Level))$: 63.9471% $Magnitude(Level)$: 31.2733% Cb: $BC(\Delta DC)$: 7.94287% $Magnitude(\Delta DC)$: 7.58987% $(Run, BC(Level))$: 61.7173% $Magnitude(Level)$: 22.7503% Cr: $BC(\Delta DC)$: 7.88357% $Magnitude(\Delta DC)$: 7.44672% $(Run, BC(Level))$: 61.0419% $Magnitude(Level)$: 23.6283%	R = 10 Y: $BC(\Delta DC)$: 8.43145% $Magnitude(\Delta DC)$: 12.3385% $(Run, BC(Level))$: 58.6847% $Magnitude(Level)$: 20.5454% Cb: $BC(\Delta DC)$: 30.6732% $Magnitude(\Delta DC)$: 29.3101% $(Run, BC(Level))$: 32.0043% $Magnitude(Level)$: 8.01427% Cr: $BC(\Delta DC)$: 27.7792% $Magnitude(\Delta DC)$: 26.2399% $(Run, BC(Level))$: 36.3173% $Magnitude(Level)$: 9.66423%
---	---

Рисунок 44 Вычисление соотношений

Результат работы для изображения Lena:

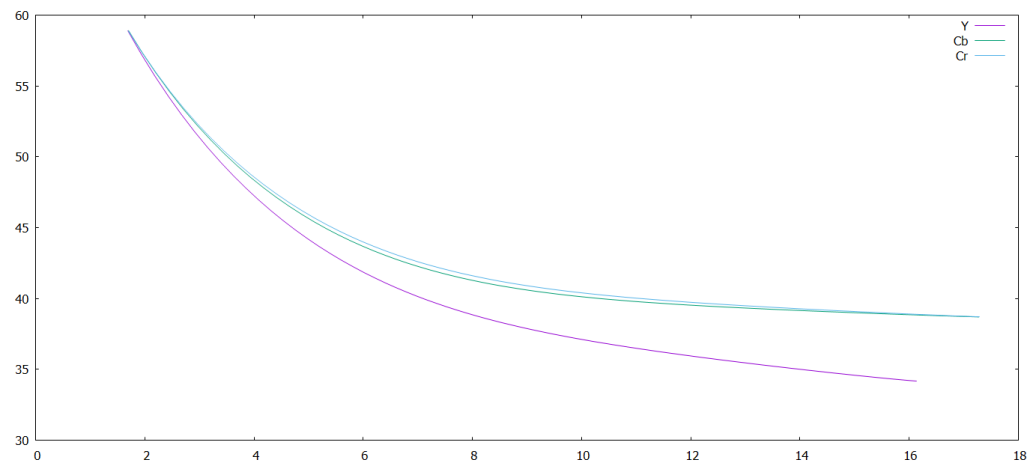


Рисунок 45 Зависимость PSNR от степени сжатия

Результаты вычисления соотношений:

R = 1	R = 10
Y:	Y:
BC(delta DC): 2.48849%	BC(delta DC): 9.74595%
Magnitude(delta DC): 5.1968%	Magnitude(delta DC): 20.3528%
(Run, BC(Level)): 62.3157%	(Run, BC(Level)): 48.9961%
Magnitude(Level): 29.9991%	Magnitude(Level): 20.9059%
Cb:	Cb:
BC(delta DC): 3.84639%	BC(delta DC): 20.5216%
Magnitude(delta DC): 5.87673%	Magnitude(delta DC): 31.3541%
(Run, BC(Level)): 66.019%	(Run, BC(Level)): 36.4379%
Magnitude(Level): 24.258%	Magnitude(Level): 11.6868%
Cr:	Cr:
BC(delta DC): 3.80994%	BC(delta DC): 19.5004%
Magnitude(delta DC): 6.03174%	Magnitude(delta DC): 30.8722%
(Run, BC(Level)): 65.1755%	(Run, BC(Level)): 37.6549%
Magnitude(Level): 24.9829%	Magnitude(Level): 11.9727%

Рисунок 46 Вычисление соотношений

Результат работы для изображения Baboon:

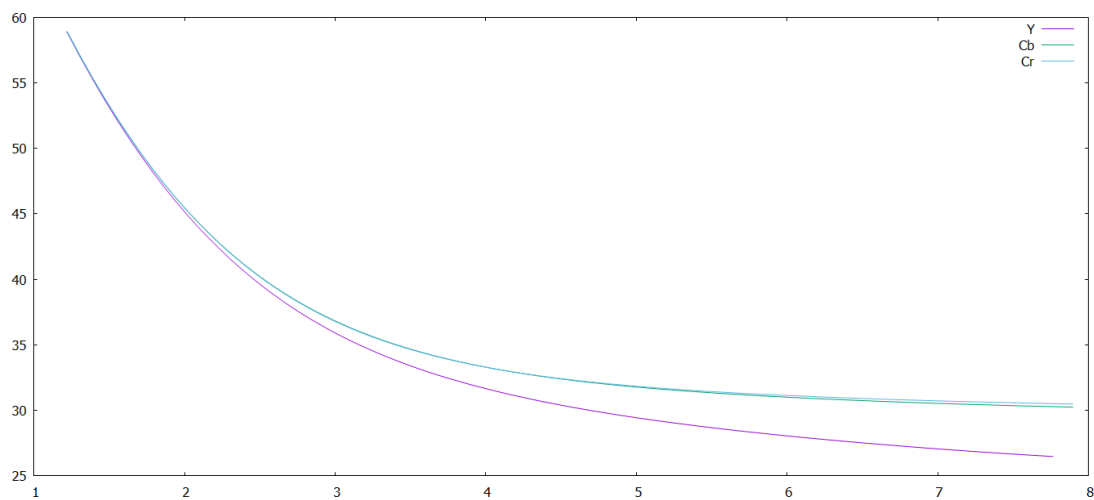


Рисунок 47 Зависимость PSNR от степени сжатия

Результаты вычисления соотношений:

R = 1	R = 10
Y:	Y:
BC(delta DC): 1.12617%	BC(delta DC): 4.23747%
Magnitude(delta DC): 2.64113%	Magnitude(delta DC): 9.93787%
(Run, BC(Level)): 58.3059%	(Run, BC(Level)): 60.9099%
Magnitude(Level): 37.9268%	Magnitude(Level): 24.915%
Cb:	Cb:
BC(delta DC): 1.55986%	BC(delta DC): 9.87188%
Magnitude(delta DC): 2.99535%	Magnitude(delta DC): 18.9566%
(Run, BC(Level)): 63.1873%	(Run, BC(Level)): 53.0006%
Magnitude(Level): 32.2577%	Magnitude(Level): 18.1715%
Cr:	Cr:
BC(delta DC): 1.6565%	BC(delta DC): 10.8762%
Magnitude(delta DC): 2.96231%	Magnitude(delta DC): 19.4498%
(Run, BC(Level)): 63.4671%	(Run, BC(Level)): 51.7589%
Magnitude(Level): 31.9141%	Magnitude(Level): 17.9155%

Рисунок 48 Вычисление соотношений

Дополнительное задание:

Задание:

Постройте зависимости PSNR (степень сжатия) для компонент Y использованных изображений, а также нанесите на эти графики зависимость PSNR (степень сжатия) для 2-ух изображений:

- 1) Аддитивный белый гауссовский шум при 64.
- 2) Импульсный шум с равновероятным появлением 0 и 255.

Сделать выводы по полученным зависимостям.

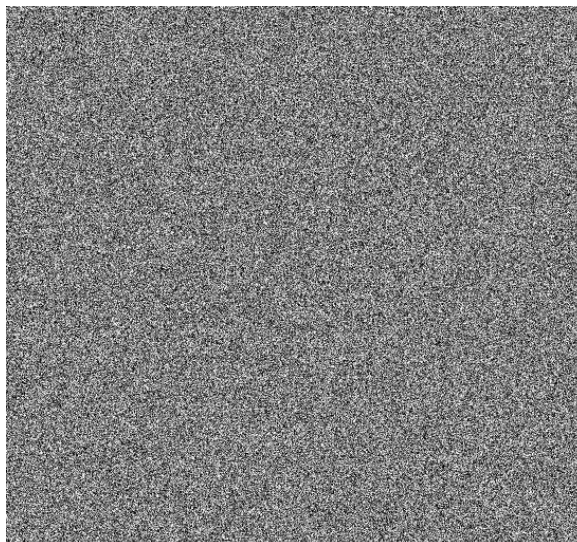


Рисунок 49 Белый гауссовский шум с $\sigma=64$.

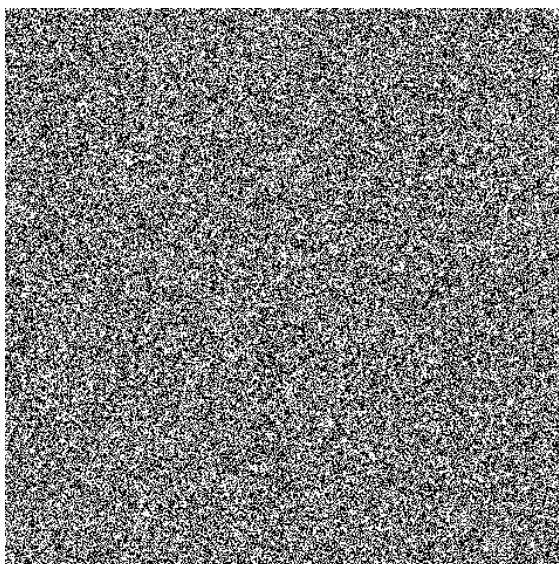


Рисунок 50 Импульсный шум с $p_a=p_b=50\%$.

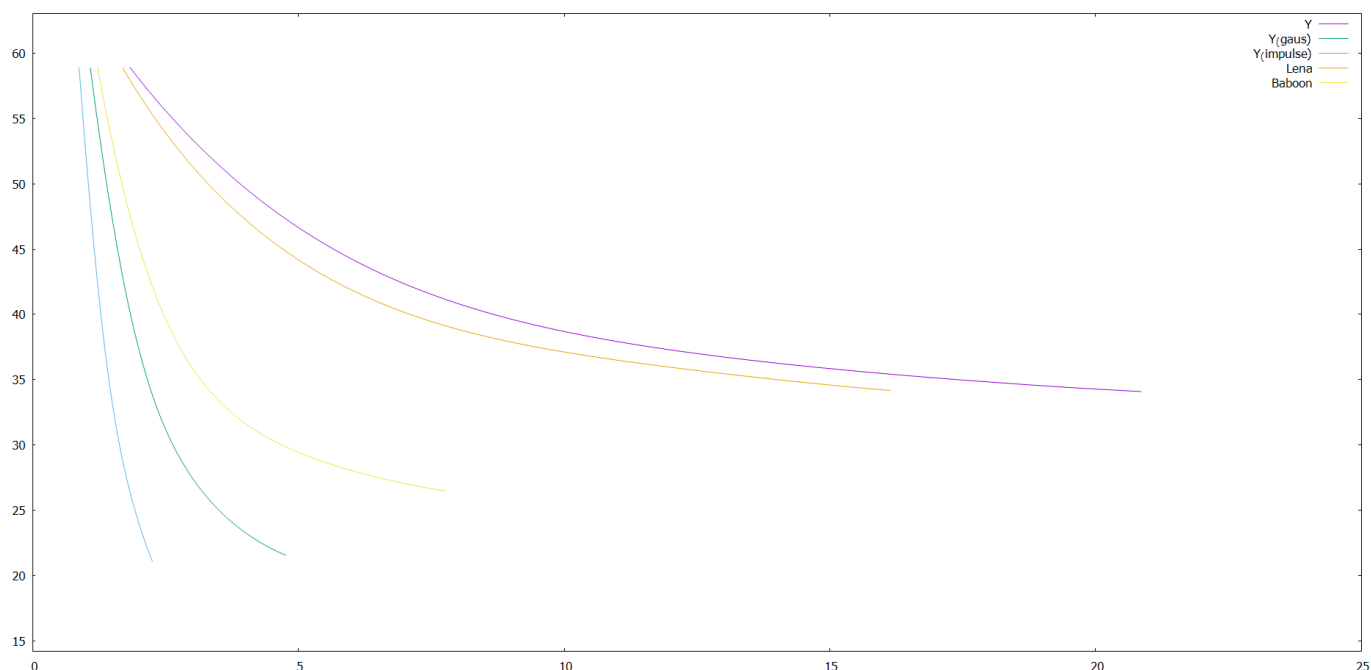


Рисунок 51 Сравнение PSNR от степени сжатия

Таким образом, импульсный шум сжимается куда хуже, чем что-либо ещё. При этом, если смотреть на графики исходных изображений, то изображение “baboon” среди них сжимается явно хуже всего, а при степени сжатия примерно 8% имеет PSNR равный чуть меньше 25-ти. Таким образом, эффективность сжатия напрямую зависит от особенностей самого изображения.

Выводы

Таким образом, во время выполнения данной работы мы выяснили что такое ДКП и как оно работает: с помощью него возможно получение спектральных коэффициентов, необходимых для сжатия и кодирования изображения; изображение разбивается на блоки 8×8 и энергия спектра сфокусирована в верхнем левом углу (из-за значения косинусов в позиции $(0;0)$); Визуальное восстановление изображения происходит без сильных потерь (незначительные потери связаны с округлением дробных частей), значения PSNR тоже остаются довольно высокими; При увеличении R (радиуса квантования) степень сжатия изображения увеличивается, но потери происходят БОльшие (на картинках становятся заметны границы блоков и присутствует размытие, нечёткие контуры), значение PSNR уменьшается, больше всего уменьшается у яркостной компоненты Y , т.к. она содержит больше всего информации. Прделав разностное кодирование, видим, что коэффициенты постоянного тока концентрируются около нуля и их диапазон сужается. Разностное кодирование уменьшает значение энтропии, следовательно улучшает сжатие. Значения переменного тока берут в зигзагообразном порядке. Наибольший процент в сжатом битовом потоке отводится на $(Run, VC(Level))$, так как этих пар больше всего. С увеличением радиуса R меняется соотношение размеров в сжатом битовом потоке. При небольших параметрах сжатия изображение остаётся в довольно хорошем качестве, при больших же параметрах сжатия ситуация прямо противоположная: качество изображения очень плохое, видны размытия и границы блоков.

Листинг программы

```
#pragma once
#include <vector>
#include <fstream>
#include <map>
#include <iostream>
using namespace std;

typedef struct BFH
{
    short bfType;
    int bfSize;
    short bfReserved1;
    short bfOffBits;;
    int bfReserved2;
} MBITMAPFILEHEADER;

typedef struct BIH
{
    int biSize;
    int biWidth;
    int biHeight;
    short int biPlanes;
    short int biBitCount;
    int biCompression;
    int biSizeImage;
    int biXPelsPerMeter;
    int biYPelsPerMeter;
    int biClrUsed;
    int biClrImportant;
} MBITMAPINFOHEADER;
typedef struct RGB
{
    unsigned char rgbBlue;
    unsigned char rgbGreen;
    unsigned char rgbRed;
}MRGBQUAD;

class YCbCr
{
public:
    int N;
    vector<vector<double>> Y;
    vector<vector<double>> Cb;
    vector<vector<double>> Cr;
    vector<vector<int>> DC;
    vector<vector<pair<unsigned char, int>>> codDC;
    int height;
```



```

    int width;
    YCbCr(vector<vector<double>> Y, vector<vector<double>> Cb,
vector<vector<double>> Cr) {
        this->Y = Y;
        this->Cb = Cb;
        this->Cr = Cr;
        height = Y.size();
        width = Y[0].size();
        N = 8;
    }

    YCbCr(MRGBQUAD** rgb, int height, int width) {
        this->height = height;
        this->width = width;
        N = 8;
        Y.resize(height);
        Cb.resize(height);
        Cr.resize(height);
        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
                Y[i].push_back(clipping(((double)rgb[i][j].rgbRed * 0.299 +
(double)rgb[i][j].rgbGreen * 0.587 + (double)rgb[i][j].rgbBlue * 0.114)));
                Cb[i].push_back(clipping(0.5643 * ((double)rgb[i][j].rgbBlue -
Y[i][j]) + 128));
                Cr[i].push_back(clipping(0.7132 * ((double)rgb[i][j].rgbRed -
Y[i][j]) + 128));
            }
        }
    }

    YCbCr(int height, int width) {
        this->height = height;
        this->width = width;
        Y.resize(height);
        Cb.resize(height);
        Cr.resize(height);
        N = 8;
    }

    double calculteRandomDouble() {
        return (double)(rand() % 2000 - 1000) / 1000;
    }

    void generateGaussianNoise(double sigma, int height, int width) {
        vector<vector<double>> result(height);
        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j += 2) {
                double x = calculteRandomDouble();
                double y = calculteRandomDouble();
            }
        }
    }

```

```

        double s = (x * x) + (y * y);
        while (s > 1 || s == 0)
        {
            x = calculteRandomDouble();
            y = calculteRandomDouble();
            s = (x * x) + (y * y);
        }
        result[i].push_back(sigma * x * sqrt(-2 * log(s) / s));
        result[i].push_back(sigma * y * sqrt(-2 * log(s) / s));
    }
}
for (int i = 0; i < height; i++) {
    for (int j = 0; j < width; j++) {
        result[i][j] = clipping(result[i][j]);
    }
}
Y = result;
Cb = result;
Cr = result;
}

double calculteRandomDoubleV2() {
    return (double)(rand() % 1000) / 1000.0;
}

void generateImpulseNoise() {
    vector<vector<double>> result(height);
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            double tmp = calculteRandomDoubleV2();
            if (tmp >= 0.5) {
                result[i].push_back(0);
            } else {
                result[i].push_back(255);
            }
        }
    }
    Y = result;
    Cb = result;
    Cr = result;
}

vector<vector<double>> getY() {
    return Y;
}

vector<vector<double>> getCb() {
    return Cb;
}

```

```

vector<vector<double>>> getCr() {
    return Cr;
}

vector<vector<int>>> getDC() {
    return DC;
}

vector<vector<pair<unsigned char, int>>>> getCodDC() {
    return codDC;
}

pair<unsigned char, int> getCategory(double diff) {
    pair<unsigned char, int> res;
    int mg = static_cast<int>(round(diff));
    if (mg == 0) {
        res.first = 0;
        res.second = 0;
        return res;
    }
    for (size_t i = 1; i < 16; i++) {
        int m_min = 1 - pow(2, i);
        int m_max = 0 - pow(2, i - 1);
        int p_min = pow(2, i - 1);
        int p_max = pow(2, i) - 1;
        if ((mg >= m_min && mg <= m_max) || (mg >= p_min && mg <=
p_max)) {
            res.first = static_cast<unsigned char>(i);
            res.second = mg;
            return res;
        }
    }
}

void generateDC() {
    size_t numOfBlocks = height / N;

    vector<int> YDC;
    vector<int> CbDC;
    vector<int> CrDC;
    for (size_t i = 0; i < numOfBlocks; i++) {
        for (size_t j = 0; j < numOfBlocks; j++) {
            YDC.push_back(static_cast<int>(Y[i * N][j * N]));
            CbDC.push_back(static_cast<int>(Cb[i * N][j * N]));
            CrDC.push_back(static_cast<int>(Cr[i * N][j * N]));
        }
    }
    DC.push_back(YDC);
    DC.push_back(CbDC);

```

```

        DC.push_back(CrDC);
    }

void codingDC() {
    size_t numOfBlocks = DC[0].size();
    double average_Y = 0, average_Cb = 0, average_Cr = 0;
    for (size_t i = 0; i < numOfBlocks; i++) {
        average_Y += DC[0][i];
        average_Cb += DC[1][i];
        average_Cr += DC[2][i];
    }
    average_Y /= numOfBlocks;
    average_Cb /= numOfBlocks;
    average_Cr /= numOfBlocks;

    vector<pair<unsigned char, int>> vec_Y, vec_Cb, vec_Cr;
    for (size_t i = 0; i < numOfBlocks; i++) {
        if (i == 0) {
            vec_Y.push_back(getCategory(DC[0][0] - average_Y));
            vec_Cb.push_back(getCategory(DC[1][0] - average_Cb));
            vec_Cr.push_back(getCategory(DC[2][0] - average_Cr));
        }
        else {
            vec_Y.push_back(getCategory(DC[0][i] - DC[0][i - 1]));
            vec_Cb.push_back(getCategory(DC[1][i] - DC[1][i - 1]));
            vec_Cr.push_back(getCategory(DC[2][i] - DC[2][i - 1]));
        }
    }
    codDC.push_back(vec_Y);
    codDC.push_back(vec_Cb);
    codDC.push_back(vec_Cr);
}

void buildHistogram(vector<vector<int>> DC, string fileName, size_t I) {
    ofstream oFile(fileName, ios_base::out);

    for (size_t i = 0; i < DC[I].size(); i++) {
        oFile << static_cast<double>(DC[I][i]) << " ";
    }

    oFile.close();
}

void buildHistogram(vector<vector<pair<unsigned char, int>>> codDC, string fileName,
size_t I) {
    ofstream oFile(fileName, ios_base::out);

    for (size_t i = 0; i < DC[I].size(); i++) {
        oFile << static_cast<double>(codDC[I][i].second) << " ";
    }
}

```

```

    }

    oFile.close();
}

void calculateEntropy() {
    map<double, double> p_dc_Y, p_dc_Cb, p_dc_Cr, p_codDC_Y, p_codDC_Cb,
p_codDC_Cr;
    for (size_t i = 0; i < DC[0].size(); i++) {
        // DC
        if (p_dc_Y.find(DC[0][i]) != p_dc_Y.end())
            p_dc_Y[DC[0][i]]++;
        else p_dc_Y.insert(pair<double, double>(DC[0][i], 1));

        if (p_dc_Cb.find(DC[1][i]) != p_dc_Cb.end())
            p_dc_Cb[DC[1][i]]++;
        else p_dc_Cb.insert(pair<double, double>(DC[1][i], 1));

        if (p_dc_Cr.find(DC[2][i]) != p_dc_Cr.end())
            p_dc_Cr[DC[2][i]]++;
        else p_dc_Cr.insert(pair<double, double>(DC[2][i], 1));

        if (p_codDC_Y.find(codDC[0][i].second) != p_codDC_Y.end())
            p_codDC_Y[codDC[0][i].second]++;
        else p_codDC_Y.insert(pair<double, double>(codDC[0][i].second, 1));

        if (p_codDC_Cb.find(codDC[1][i].second) != p_codDC_Cb.end())
            p_codDC_Cb[codDC[1][i].second]++;
        else p_codDC_Cb.insert(pair<double, double>(codDC[1][i].second, 1));

        if (p_codDC_Cr.find(codDC[2][i].second) != p_codDC_Cr.end())
            p_codDC_Cr[codDC[2][i].second]++;
        else p_codDC_Cr.insert(pair<double, double>(codDC[2][i].second, 1));
    }

    double H_dc_Y = 0, H_dc_Cb = 0, H_dc_Cr = 0, H_cdc_Y = 0, H_cdc_Cb = 0,
H_cdc_Cr = 0;
    for (pair<double, double> it : p_dc_Y) {
        it.second /= DC[0].size();
        if (!isinf(log2(it.second))) {
            H_dc_Y += it.second * log2(it.second);
        }
    }
    for (pair<double, double> it : p_dc_Cb) {
        it.second /= DC[0].size();
        if (!isinf(log2(it.second))) {
            H_dc_Cb += it.second * log2(it.second);
        }
    }
}

```



```

        for (pair<double, double> it : p_dc_Cr) {
            it.second /= DC[0].size();
            if (!isinf(log2(it.second))) {
                H_dc_Cr += it.second * log2(it.second);
            }
        }
        for (pair<double, double> it : p_codDC_Y) {
            it.second /= DC[0].size();
            if (!isinf(log2(it.second))) {
                H_cdc_Y += it.second * log2(it.second);
            }
        }
        for (pair<double, double> it : p_codDC_Cb) {
            it.second /= DC[0].size();
            if (!isinf(log2(it.second))) {
                H_cdc_Cb += it.second * log2(it.second);
            }
        }
        for (pair<double, double> it : p_codDC_Cr) {
            it.second /= DC[0].size();
            if (!isinf(log2(it.second))) {
                H_cdc_Cr += it.second * log2(it.second);
            }
        }
        cout << "Entropy of DC^q";
        cout << "\nEntropy of Y: " << -H_dc_Y;
        cout << "\nEntropy of Cb: " << -H_dc_Cb;
        cout << "\nEntropy of Cr: " << -H_dc_Cr << endl;
        cout << "Entropy of delta DC";
        cout << "\nEntropy of Y: " << -H_cdc_Y;
        cout << "\nEntropy of Cb: " << -H_cdc_Cb;
        cout << "\nEntropy of Cr: " << -H_cdc_Cr << endl;
    }

vector<vector<int>> generateAC(vector<vector<double>> I) {
    vector<vector<int>> res;
    for (size_t i = 0; i < height; i += N) {
        for (size_t j = 0; j < width; j += N) {
            vector<int> vec;
            for (size_t diag = 0; diag < N; diag++) {
                if (diag % 2 == 0) {
                    int x = diag;
                    int y = 0;
                    while (x >= 0) {
                        if (x == 0 && y == 0) break;
                        vec.push_back(static_cast<int>(round(I[i +
x][j + y]))));
                        x--;
                        y++;
                    }
                }
            }
            res.push_back(vec);
        }
    }
    return res;
}

```

```

        }
    }
    else {
        int x = 0;
        int y = diag;
        while (y >= 0) {
            vec.push_back(static_cast<int>(round(I[i +
x][j + y]))));
            x++;
            y--;
        }
    }
}
for (size_t diag = 1; diag < N; diag++) {
    if (diag % 2 == 0) {
        int x = diag;
        int y = N - 1;
        while (x <= N - 1) {
            vec.push_back(static_cast<int>(round(I[i +
x][j + y]))));
            x++;
            y--;
        }
    }
    else {
        int x = N - 1;
        int y = diag;
        while (y <= N - 1) {
            vec.push_back(static_cast<int>(round(I[i +
x][j + y]))));
            x--;
            y++;
        }
    }
}
res.push_back(vec);
}
}
return res;
}

```

```

vector<vector<pair<unsigned char, pair<unsigned char, int>>>>
codingAC(vector<vector<int>>> AC) {
    vector<vector<pair<unsigned char, pair<unsigned char, int>>>> res;
    for (size_t i = 0; i < AC.size(); i++) {
        vector<pair<unsigned char, pair<unsigned char, int>>>> vec;
        size_t lastNotNull = 0;
        for (size_t j = 0; j < 63; j++) {
            if (AC[i][j] != 0) lastNotNull = j;

```

```

    }

    for (size_t j = 0; j <= lastNotNull; j++) {
        if (AC[i][j] != 0) {
            pair<unsigned char, pair<unsigned char, int>> tmp;
            tmp.first = 0;
            tmp.second = getCategory(AC[i][j]);
            vec.push_back(tmp);
        }
        else {
            pair<unsigned char, pair<unsigned char, int>> tmp;
            tmp.first = 1;
            size_t j1 = j + 1;
            size_t count = 1;
            while (AC[i][j1] == 0 && count <= 16 && j1 <
lastNotNull) {

                j1++;
                count++;
                tmp.first++;
            }
            if (AC[i][j1] != 0 && count < 16) {
                tmp.second = getCategory(AC[i][j1]);
                vec.push_back(tmp);
                j = j1;
            }
            else if (count == 16) {
                tmp.first = 15;
                tmp.second.first = 0;
                tmp.second.second = 0;
                vec.push_back(tmp);
                j = j1;
            }
        }
    }

    pair<unsigned char, pair<unsigned char, int>> last;
    last.first = 0;
    last.second.first = 0;
    last.second.second = 0;
    vec.push_back(last);

    res.push_back(vec);
}
return res;
}

size_t getNumOfPair(vector<vector<pair<unsigned char, pair<unsigned char, int>>>>
codAC) {
    size_t res = 0;

```

```

        for (size_t i = 0; i < codAC.size(); i++) {
            res += codAC[i].size();
        }
        return res;
    }

    double sizeOfStream(vector<vector<pair<unsigned char, pair<unsigned char, int>>>>
codAC, size_t I) {
        size_t Ndc = DC[0].size();
        size_t Nrl = getNumOfPair(codAC);

        size_t sum_BC_dDC = 0;
        map<double, double> p_BC_dDC;
        for (size_t i = 0; i < codDC[I].size(); i++) {
            sum_BC_dDC += codDC[I][i].first;

            if (p_BC_dDC.find(codDC[I][i].first) != p_BC_dDC.end()) {
                p_BC_dDC[codDC[I][i].first]++;
            }
            else p_BC_dDC.insert(pair<double, double>(codDC[I][i].first, 1));
        }

        double H_BC_dDC = 0;
        for (pair<double, double> it : p_BC_dDC) {
            it.second /= codDC[I].size();
            if (!isinf(log2(it.second))) {
                H_BC_dDC += it.second * log2(it.second);
            }
        }
        H_BC_dDC *= -1;

        size_t sum_BC_level = 0;
        map<pair<unsigned char, unsigned char>, double> p_rl;
        for (size_t i = 0; i < codAC.size(); i++) {
            for (size_t j = 0; j < codAC[i].size(); j++) {
                sum_BC_level += codAC[i][j].second.first;

                pair<unsigned char, unsigned char> tmp;
                tmp.first = codAC[i][j].first;
                tmp.second = codAC[i][j].second.first;
                if (p_rl.find(tmp) != p_rl.end())
                    p_rl[tmp]++;
                else p_rl.insert(pair<pair<unsigned char, unsigned char>,
double>(tmp, 1));
            }
        }
        double H_rl = 0;
        for (pair<pair<unsigned char, unsigned char>, double> it : p_rl) {

```



```

        it.second /= Nrl;
        if (!isinf(log2(it.second))) {
            H_rl += it.second * log2(it.second);
        }
    }
    H_rl *= -1;

    size_t res = (H_BC_dDC * Ndc) + sum_BC_dDC + (H_rl * Nrl) +
sum_BC_level;

    size_t origin = 8 * height * width;

    double d = (double) origin / (double)res;

    double bcdc = (double)((double)(H_BC_dDC * Ndc * 100) / (double)(res));
    double magnitude = (double)((double)(sum_BC_dDC * 100) / (double)(res));
    double runBClevel = (double)((H_rl * Nrl * 100) / (double)(res));
    double magnitudeLevel = (double)((sum_BC_level * 100) / (double)(res));
    cout << "BC(delta DC): " << bcdc << "%" << endl;
    cout << "Magnitude(delta DC): " << magnitude << "%" << endl;
    cout << "(Run, BC(Level)): " << runBClevel << "%" << endl;
    cout << "Magnitude(Level): " << magnitudeLevel << "%" << endl;
    //cout << d << ", ";

    return d;
}

private:
    unsigned char clipping(double x) {
        unsigned char res;
        if (x > 255) {
            res = 255;
            return res;
        }
        else if (x < 0) {
            res = 0;
            return res;
        }
        return static_cast<unsigned char>(round(x));
    }

};

#define _CRT_SECURE_NO_WARNINGS
// #define _USE_MATH_DEFINES
#include <stdio.h>
// #include <stdlib.h>
#include <string.h>

```

```

#include <iostream>
#include <fstream>
//#include <vector>
//#include <complex>
//#include <map>
#include <math.h>
//#include <algorithm>
//#include <time.h>
#include "YCbCr.h"
using namespace std;
const double M_PI = 3.141592653589793;

MRGBQUAD** readBMP(FILE* f, MBITMAPFILEHEADER* bfh,
MBITMAPINFOHEADER* bih)
{
    int k = 0;
    k = fread(bfh, sizeof(*bfh) - 2, 1, f);
    if (k == 0)
    {
        cout << "reading error";
        return 0;
    }

    k = fread(bih, sizeof(*bih), 1, f);
    if (k == NULL)
    {
        cout << "reading error";
        return 0;
    }
    int height = abs(bih->biHeight);
    int width = abs(bih->biWidth);
    while (height % 8 != 0) {
        height++;
    }
    while (width % 8 != 0) {
        width++;
    }
    MRGBQUAD** rgb = new MRGBQUAD * [height];
    for (int i = 0; i < height; i++)
    {
        rgb[i] = new MRGBQUAD[width];
    }
    int pad = 4 - (width * 3) % 4;
    for (int i = 0; i < height; i++)
    {
        fread(rgb[i], sizeof(MRGBQUAD), width, f);
        if (pad != 4)
        {
            fseek(f, pad, SEEK_CUR);
        }
    }
}

```

```

    }
}
return rgb;
}

void writeBMP(FILE* f, MRGBQUAD** rgbb, MBITMAPFILEHEADER* bfh,
MBITMAPINFOHEADER* bih, int height, int width)
{
    bih->biHeight = height;
    bih->biWidth = width;
    fwrite(bfh, sizeof(*bfh) - 2, 1, f);
    fwrite(bih, sizeof(*bih), 1, f);
    int pad = 4 - ((width) * 3) % 4;
    char buf = 0;
    for (int i = 0; i < height; i++)
    {
        fwrite((rgbb[i]), sizeof(MRGBQUAD), width, f);
        if (pad != 4)
        {
            fwrite(&buf, 1, pad, f);
        }
    }
}

```

```

MRGBQUAD** getRed(MRGBQUAD** rgb, int height, int width) {
    MRGBQUAD** g = new MRGBQUAD * [height];
    for (int i = 0; i < height; i++)
    {
        g[i] = new MRGBQUAD[width];
    }
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            g[i][j].rgbGreen = 0;
            g[i][j].rgbBlue = 0;
            g[i][j].rgbRed = rgb[i][j].rgbRed;
        }
    }
    return g;
}

```

```

MRGBQUAD** getGreen(MRGBQUAD** rgb, int height, int width) {
    MRGBQUAD** g = new MRGBQUAD * [height];
    for (int i = 0; i < height; i++)
    {
        g[i] = new MRGBQUAD[width];
    }
    for (int i = 0; i < height; i++) {

```

```

        for (int j = 0; j < width; j++) {
            g[i][j].rgbGreen = rgb[i][j].rgbGreen;
            g[i][j].rgbBlue = 0;
            g[i][j].rgbRed = 0;
        }
    }
    return g;
}

MRGBQUAD** calculateMirror(MRGBQUAD** rgb, int height, int width) {
    MRGBQUAD** g = new MRGBQUAD * [height];
    for (int i = 0; i < height; i++)
    {
        g[i] = new MRGBQUAD[width];
    }
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            g[i][j].rgbRed = rgb[height - i - 1][j].rgbRed;
            g[i][j].rgbGreen = rgb[height - i - 1][j].rgbGreen;
            g[i][j].rgbBlue = rgb[height - i - 1][j].rgbBlue;
        }
    }
    return g;
}

MRGBQUAD** getBlue(MRGBQUAD** rgb, int height, int width) {
    MRGBQUAD** g = new MRGBQUAD * [height];
    for (int i = 0; i < height; i++)
    {
        g[i] = new MRGBQUAD[width];
    }
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            g[i][j].rgbGreen = 0;
            g[i][j].rgbBlue = rgb[i][j].rgbBlue;
            g[i][j].rgbRed = 0;
        }
    }
    return g;
}

double calculateMathExpectation(vector<vector<double>> rgb, int height, int width) {
    double res = 0;
    double WH = (double)width * (double)height;
    for (int i = 0; i < rgb.size(); i++) {
        for (int j = 0; j < rgb[0].size(); j++) {
            if (rgb[i].size() != 0)
                res += rgb[i][j];
        }
    }
}

```



```

    }
    res = res / WH;
    return res;
}

double calculateDispersion(vector<vector<double>> rgb, int height, int width) {
    double res = 0;
    double WH = (double)width * (double)height;
    double m = calculateMathExpectation(rgb, height, width);
    for (int i = 0; i < rgb.size(); i++) {
        for (int j = 0; j < rgb[0].size(); j++) {
            if (rgb[i].size() != 0)
                res += pow((rgb[i][j] - m), 2);
        }
    }
    res = res / (WH - 1);
    return sqrt(res);
}

double calculateCorrelation(vector<vector<double>> A, vector<vector<double>> B, int height,
int width) {
    double d1 = calculateDispersion(A, height, width);
    double d2 = calculateDispersion(B, height, width);
    double m1 = calculateMathExpectation(A, height, width);
    double m2 = calculateMathExpectation(B, height, width);
    for (int i = 0; i < A.size(); i++) {
        for (int j = 0; j < A[0].size(); j++) {
            if (A[i].size() != 0 && B[i].size() != 0) {
                A[i][j] = A[i][j] - m1;
                B[i][j] = B[i][j] - m2;
                A[i][j] = A[i][j] * B[i][j];
            }
        }
    }
    double res = calculateMathExpectation(A, height, width) / (d1 * d2);
    return res;
}

MRGBQUAD** getRGBfromY(vector<vector<double>> Y1, vector<vector<double>> Y2, int
height, int width) {
    MRGBQUAD** g = new MRGBQUAD * [height];
    for (int i = 0; i < height; i++)
    {
        g[i] = new MRGBQUAD[width];
    }
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            g[i][j].rgbGreen = Y1[i][j];
            g[i][j].rgbBlue = Y2[i][j];
        }
    }
}

```

```

        g[i][j].rgbRed = Y2[i][j];
    }
}
return g;
}

MRGBQUAD** getRGBfromY(vector<vector<double>> Y, int height, int width) {
    MRGBQUAD** g = new MRGBQUAD * [height];
    for (int i = 0; i < height; i++)
    {
        g[i] = new MRGBQUAD[width];
    }
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            g[i][j].rgbGreen = Y[i][j];
            g[i][j].rgbBlue = Y[i][j];
            g[i][j].rgbRed = Y[i][j];
        }
    }
    return g;
}

```

```

unsigned char clipping(double x) {
    unsigned char res;
    if (x > 255) {
        res = 255;
        return res;
    }
    else if (x < 0) {
        res = 0;
        return res;
    }
    return static_cast<unsigned char>(round(x));
}

```

```

MRGBQUAD** getRGBfromYreverse(vector<vector<double>>& Y,
vector<vector<double>>& Cb, vector<vector<double>>& Cr, int height, int width) {
    MRGBQUAD** g = new MRGBQUAD * [height];
    for (int i = 0; i < height; i++)
    {
        g[i] = new MRGBQUAD[width];
    }
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            g[i][j].rgbGreen = clipping(Y[i][j] - 0.714 * (Cr[i][j] - 128.0) - 0.334 *
(Cb[i][j] - 128.0));
            g[i][j].rgbRed = clipping(Y[i][j] + 1.402 * (Cr[i][j] - 128.0));
            g[i][j].rgbBlue = clipping(Y[i][j] + 1.772 * (Cb[i][j] - 128.0));
        }
    }
}

```

```

    }
}
return g;
}

```

```

double calculateSumSquareDifferences(vector<vector<double>>& firstArray,
vector<vector<double>>& secondArray) {
    double res = 0;
    for (int i = 0; i < firstArray.size(); i++) {
        for (int j = 0; j < firstArray[0].size(); j++) {
            res += pow((firstArray[i][j] - secondArray[i][j]), 2);
        }
    }
    return res;
}

```

```

double calculatePSNR(vector<vector<double>> firstArray, vector<vector<double>>
secondArray) {
    double niz = calculateSumSquareDifferences(firstArray, secondArray);
    double tmp = (((double)firstArray.size() * (double)firstArray[0].size() * pow((pow(2, 8) -
1), 2)) / niz;
    double PSNR = 10 * log10(tmp);
    return PSNR;
}

```

```

void writeFile(const char* filename, vector<vector<double>>& array, int height, int width) {
    ofstream fout1;
    fout1.open(filename);
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            fout1 << array[i][j] << "\n";
        }
    }
    fout1.close();
}

```

```

void writeFile(const char* filename, vector<vector<double>>& array, int height, int width, bool
flag) {
    ofstream fout1;
    fout1.open(filename);
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            fout1 << array[i][j] << " ";
        }
        fout1 << "\n";
    }
    fout1.close();
}

```

```

void writeFile(const char* filename, vector<double>& array) {
    ofstream fout1;
    fout1.open(filename);
    for (int i = 0; i < array.size(); i++) {
        fout1 << (i) << " " << array[i] << "\n";
    }
    fout1.close();
}

```

```

void writeFile(const char* filename, vector<double>& array, vector<double> tmp) {
    ofstream fout1;
    fout1.open(filename);
    for (int i = 0; i < array.size(); i++) {
        fout1 << tmp[i] << " " << array[i] << "\n";
    }
    fout1.close();
}

```

```

double calculateEntropy(vector<vector<double>> rgb) {
    double H = 0.0;
    vector<double> p;
    for (int i = 0; i < 256; i++) {
        p.push_back(0);
    }
    for (int i = 0; i < rgb.size(); i++) {
        for (int j = 0; j < rgb[0].size(); j++) {
            int tmp = rgb[i][j];
            tmp += 256;
            tmp = tmp % 256;
            p[tmp]++;
        }
    }
    for (int i = 0; i < 256; i++) {
        p[i] = p[i] / (double)(rgb.size() * rgb[0].size());
    }

    for (int i = 0; i < 256; i++) {
        if (p[i] != 0)
            H += (double)(p[i] * log2(p[i]));
    }
    H = (double)(-H);
    return H;
}

```

```

double calculateEntropyV2(vector<vector<double>>& rgb) {
    double H = 0.0;
    map<double, double> p;
    double N = (double)rgb.size() * rgb[0].size();

```

```

double delta = 256 / (double)N;
for (int i = 0; i < rgb.size(); i++) {
    for (int j = 0; j < rgb[0].size(); j++) {
        if (p.find(rgb[i][j]) != p.end()) {
            p[(size_t)rgb[i][j]]++;
        }
        else p.insert(pair<double, double>(rgb[i][j], 0));
    }
}
for (pair<double, double> it : p) {
    it.second /= (double)N;
    if (!isinf(log2(it.second))) {
        H += it.second * log2(it.second);
    }
}
return (double)(-H);
}

```

```

int mod(int a, int b) {
    int res = a % b;
    if (res < 0) {
        res += b;
    }
    return res;
}

```

```

YCbCr DCT(int N, YCbCr& rgb) {
    YCbCr res(rgb.Y, rgb.Cb, rgb.Cr);
    double Ck = 0, Cl = 0;
    for (size_t i = 0; i < rgb.height; i += N) {
        for (size_t j = 0; j < rgb.width; j += N) {
            for (size_t k = 0; k < N; k++) {
                if (k == 0) Ck = sqrt(1.0 / N);
                else Ck = sqrt(2.0 / N);
                for (size_t l = 0; l < N; l++) {
                    if (l == 0) Cl = sqrt(1.0 / N);
                    else Cl = sqrt(2.0 / N);
                    res.Y[i + k][j + l] = Cl * Ck;
                    res.Cb[i + k][j + l] = Cl * Ck;
                    res.Cr[i + k][j + l] = Cl * Ck;
                    double tmpY = 0, tmpCb = 0, tmpCr = 0;
                    for (size_t f = 0; f < N; f++) {
                        for (size_t t = 0; t < N; t++) {
                            tmpY += rgb.Y[i + f][j + t] * cos(((2.0 * f +
1) * M_PI * k) / (2 * (double)N)) * cos(((2.0 * (double)t + 1) * M_PI * l) / (2.0 * (double)N));
                            tmpCb += rgb.Cb[i + f][j + t] * cos(((2.0 * f
+ 1) * M_PI * k) / (2 * (double)N)) * cos(((2.0 * (double)t + 1) * M_PI * l) / (2.0 * (double)N));

```

```

        tmpCr += rgb.Cr[i + f][j + t] * cos(((2.0 * f +
1) * M_PI * k) / (2 * (double)N)) * cos(((2.0 * (double)t + 1) * M_PI * l) / (2.0 * (double)N));
    }
}
res.Y[i + k][j + l] *= tmpY;
res.Cb[i + k][j + l] *= tmpCb;
res.Cr[i + k][j + l] *= tmpCr;

res.Y[i + k][j + l] = round(res.Y[i + k][j + l]);
res.Cb[i + k][j + l] = round(res.Cb[i + k][j + l]);
res.Cr[i + k][j + l] = round(res.Cr[i + k][j + l]);
    }
}
}
return res;
}

```

```

YCbCr ReverseDCT(int N, YCbCr& rgb) {
    YCbCr res(rgb.Y, rgb.Cb, rgb.Cr);
    double Ck = 0, Cl = 0;
    for (size_t i = 0; i < rgb.height; i += N) {
        for (size_t j = 0; j < rgb.width; j += N) {
            for (size_t f = 0; f < N; f++) {
                for (size_t t = 0; t < N; t++) {
                    double tmpY = 0, tmpCb = 0, tmpCr = 0;
                    for (size_t k = 0; k < N; k++) {
                        if (k == 0) Ck = sqrt(1.0 / N);
                        else Ck = sqrt(2.0 / N);
                        for (size_t l = 0; l < N; l++) {
                            if (l == 0) Cl = sqrt(1.0 / N);
                            else Cl = sqrt(2.0 / N);
                            tmpY += Ck * Cl * rgb.Y[i + k][j + l] *
cos(((2.0 * (double)f + 1) * M_PI * (double)k) / (2.0 * (double)N)) * cos(((2.0 * (double)t + 1) *
M_PI * l) / (2.0 * (double)N));
                            tmpCb += Ck * Cl * rgb.Cb[i + k][j + l] *
cos(((2.0 * (double)f + 1) * M_PI * (double)k) / (2.0 * (double)N)) * cos(((2.0 * (double)t + 1) *
M_PI * l) / (2.0 * (double)N));
                            tmpCr += Ck * Cl * rgb.Cr[i + k][j + l] *
cos(((2.0 * (double)f + 1) * M_PI * (double)k) / (2.0 * (double)N)) * cos(((2.0 * (double)t + 1) *
M_PI * l) / (2.0 * (double)N));
                        }
                    }
                    res.Y[i + f][j + t] = round(tmpY);
                    res.Cb[i + f][j + t] = round(tmpCb);
                    res.Cr[i + f][j + t] = round(tmpCr);
                }
            }
        }
    }
}

```



```

    }
    return res;
}

```

```

void quantization(YCbCr& rgb, int R, int N) {
    double Q[8][8];
    for (size_t i = 0; i < N; i++) {
        for (size_t j = 0; j < N; j++) {
            Q[i][j] = 1.0 + (i + j) * R;
        }
    }

    for (size_t i = 0; i < rgb.height; i += N) {
        for (size_t j = 0; j < rgb.width; j += N) {
            for (size_t k = 0; k < N; k++) {
                for (size_t l = 0; l < N; l++) {
                    double tmpY = rgb.Y[i + k][j + l] / Q[k][l];
                    rgb.Y[i + k][j + l] = round(tmpY);
                    double tmpCb = rgb.Cb[i + k][j + l] / Q[k][l];
                    rgb.Cb[i + k][j + l] = round(tmpCb);
                    double tmpCr = rgb.Cr[i + k][j + l] / Q[k][l];
                    rgb.Cr[i + k][j + l] = round(tmpCr);
                }
            }
        }
    }
}

```

```

void dequantization(YCbCr& rgb, int R, int N) {
    double Q[8][8];
    for (size_t i = 0; i < N; i++) {
        for (size_t j = 0; j < N; j++) {
            Q[i][j] = 1.0 + (i + j) * R;
        }
    }

    for (size_t i = 0; i < rgb.height; i += N) {
        for (size_t j = 0; j < rgb.width; j += N) {
            for (size_t k = 0; k < N; k++) {
                for (size_t l = 0; l < N; l++) {
                    double tmpY = rgb.Y[i + k][j + l] * Q[k][l];
                    rgb.Y[i + k][j + l] = round(tmpY);

                    double tmpCb = rgb.Cb[i + k][j + l] * Q[k][l];
                    rgb.Cb[i + k][j + l] = round(tmpCb);
                    double tmpCr = rgb.Cr[i + k][j + l] * Q[k][l];
                    rgb.Cr[i + k][j + l] = round(tmpCr);
                }
            }
        }
    }
}

```

```

    }
}
}

```

```

struct ImagesStruct {
    vector<double> myImageY;
    vector<double> lenaY;
    vector<double> baboonY;
    vector<double> myImageCb;
    vector<double> lenaCb;
    vector<double> baboonCb;
    vector<double> myImageCr;
    vector<double> lenaCr;
    vector<double> baboonCr;
};

```

```

void buildPSNRgraphics(YCbCr k, YCbCr l, YCbCr p, YCbCr& image1, YCbCr& image2,
YCbCr& image3, ImagesStruct& res, int R) {
    if (R != 0) {
        quantization(k, R, 8);
        quantization(l, R, 8);
        quantization(p, R, 8);
        dequantization(k, R, 8);
        dequantization(l, R, 8);
        dequantization(p, R, 8);
    }
    YCbCr kReverse = ReverseDCT(8, k);
    YCbCr lReverse = ReverseDCT(8, l);
    YCbCr pReverse = ReverseDCT(8, p);
    res.myImageY.push_back(calculatePSNR(kReverse.Y, image1.Y));
    res.lenaY.push_back(calculatePSNR(lReverse.Y, image2.Y));
    res.baboonY.push_back(calculatePSNR(pReverse.Y, image3.Y));

    res.myImageCb.push_back(calculatePSNR(kReverse.Cb, image1.Cb));
    res.lenaCb.push_back(calculatePSNR(lReverse.Cb, image2.Cb));
    res.baboonCb.push_back(calculatePSNR(pReverse.Cb, image3.Cb));

    res.myImageCr.push_back(calculatePSNR(kReverse.Cr, image1.Cr));
    res.lenaCr.push_back(calculatePSNR(lReverse.Cr, image2.Cr));
    res.baboonCr.push_back(calculatePSNR(pReverse.Cr, image3.Cr));
}

```

```

vector<double> printDop(YCbCr DCT, int I) {
    vector<double> result;
    for (int i = 0; i <= 10; i++) {
        YCbCr bmpFile(DCT);
        cout << endl;
    }
}

```

```

        //      cout << "R = " << i << endl;
        quantization(bmpFile, i, 8);
        bmpFile.generateDC();
        bmpFile.codingDC();
        vector<vector<int>>> yAC = bmpFile.generateAC(bmpFile.getY());
        vector<vector<pair<unsigned char, pair<unsigned char, int>>>> yCodAC =
bmpFile.codingAC(yAC);
        result.push_back(bmpFile.sizeOfStream(yCodAC, I));
    }

    return result;
}

```

```

int main() {
    MBITMAPFILEHEADER bf1;
    MBITMAPINFOHEADER bi1;
    MBITMAPFILEHEADER bf2;
    MBITMAPINFOHEADER bi2;
    MBITMAPFILEHEADER bf3;
    MBITMAPINFOHEADER bi3;
    FILE* f1;

    f1 = fopen("myImage.bmp", "rb");
    if (f1 == NULL)
    {
        cout << "reading myImage error";
        return 0;
    }
    MRGBQUAD** rgbMyImage = readBMP(f1, &bf1, &bi1);
    fclose(f1);
    int heightMyImage = abs(bi1.biHeight);
    int widthMyImage = abs(bi1.biWidth);
    YCbCr myImage(rgbMyImage, heightMyImage, widthMyImage);

    f1 = fopen("lena.bmp", "rb");
    if (f1 == NULL)
    {
        cout << "reading lena error";
        return 0;
    }
    MRGBQUAD** rgbLena = readBMP(f1, &bf2, &bi2);
    int heightLena = abs(bi2.biHeight);
    int widthLena = abs(bi2.biWidth);
    YCbCr lena(rgbLena, heightLena, widthLena);
    fclose(f1);

    f1 = fopen("baboon.bmp", "rb");
    if (f1 == NULL)
    {

```

```

        cout << "reading baboon error";
        return 0;
    }
    MRGBQUAD** rgbBaboon = readBMP(f1, &bfh3, &bih3);
    int heightBaboon = abs(bih3.biHeight);
    int widthBaboon = abs(bih3.biWidth);
    YCbCr baboon(rgbBaboon, heightBaboon, widthBaboon);
    fclose(f1);

//№1.1
    cout << "\n" << "#1.1\n";
    YCbCr myImageDCT = DCT(8, myImage);
    YCbCr myImageReverseDCT = ReverseDCT(8, myImageDCT);
    // cout << "My image:" << endl;
    // double myImageYPSNR = calculatePSNR(myImageReverseDCT.Y, myImage.Y);
    // double myImageCbPSNR = calculatePSNR(myImageReverseDCT.Cb, myImage.Cb);
    // double myImageCrPSNR = calculatePSNR(myImageReverseDCT.Cr, myImage.Cr);
    // //cout << "Y PSNR: " << myImageYPSNR << endl;
    // //cout << "Cb PSNR: " << myImageCbPSNR << endl;
    // //cout << "Cr PSNR: " << myImageCrPSNR << endl;
    // //writeBMP(fopen("Files\\1\\1.1\\MyImageReverseDCT.bmp", "wb"),
    getRGBfromYreverse(myImageReverseDCT.Y, myImageReverseDCT.Cb,
    myImageReverseDCT.Cr, heightMyImage, widthMyImage), &bfh1, &bih1, heightMyImage,
    widthMyImage);
    // //
    // YCbCr lenaDCT = DCT(8, lena);
    // YCbCr lenaReverseDCT = ReverseDCT(8, lenaDCT);
    // /*cout << "Lena" << endl;
    // cout << "Y PSNR: " << calculatePSNR(lenaReverseDCT.Y, lena.Y) << endl;
    // cout << "Cb PSNR: " << calculatePSNR(lenaReverseDCT.Cb, lena.Cb) << endl;
    // cout << "Cr PSNR: " << calculatePSNR(lenaReverseDCT.Cr, lena.Cr) << endl;
    // writeBMP(fopen("Files\\1\\1.1\\lenaReverseDCT.bmp", "wb"),
    getRGBfromYreverse(lenaReverseDCT.Y, lenaReverseDCT.Cb, lenaReverseDCT.Cr,
    heightLena, widthLena), &bfh2, &bih2, heightLena, widthLena);
    // */
    // YCbCr baboonDCT = DCT(8, baboon);
    // YCbCr baboonReverseDCT = ReverseDCT(8, baboonDCT);
    // /*cout << "Baboon" << endl;
    // cout << "Y PSNR: " << calculatePSNR(baboonReverseDCT.Y, baboon.Y) << endl;
    // cout << "Cb PSNR: " << calculatePSNR(baboonReverseDCT.Cb, baboon.Cb) << endl;
    // cout << "Cr PSNR: " << calculatePSNR(baboonReverseDCT.Cr, baboon.Cr) << endl;
    // writeBMP(fopen("Files\\1\\1.1\\baboonReverseDCT.bmp", "wb"),
    getRGBfromYreverse(baboonReverseDCT.Y, baboonReverseDCT.Cb, baboonReverseDCT.Cr,
    heightBaboon, widthBaboon), &bfh3, &bih3, heightBaboon, widthBaboon);
    // */
    ////№2.1
    // cout << "\n" << "#2.1\n";
    // /*ImagesStruct imagesStruct;
    // for (int R = 0; R <= 10; R++) {

```

```

//          buildPSNRgraphics(myImageDCT, lenaDCT, baboonDCT, myImage, lena,
baboon, imagesStruct, R);
//      }
//
//      writeFile("Files\\2\\YMyImagePSNR.txt", imagesStruct.myImageY);
//      writeFile("Files\\2\\YLenaPSNR.txt", imagesStruct.lenaY);
//      writeFile("Files\\2\\YBaboonPSNR.txt", imagesStruct.baboonY);
//
//      writeFile("Files\\2\\CbMyImagePSNR.txt", imagesStruct.myImageCb);
//      writeFile("Files\\2\\CbLenaPSNR.txt", imagesStruct.lenaCb);
//      writeFile("Files\\2\\CbBaboonPSNR.txt", imagesStruct.baboonCb);
//
//      writeFile("Files\\2\\CrMyImagePSNR.txt", imagesStruct.myImageCr);
//      writeFile("Files\\2\\CrLenaPSNR.txt", imagesStruct.lenaCr);
//      writeFile("Files\\2\\CrBaboonPSNR.txt", imagesStruct.baboonCr);
//
//      YCbCr myImageDCT_c1(myImageDCT);
//      YCbCr myImageDCT_c2(myImageDCT);
//
//      quantization(myImageDCT, 1, 8);
//      dequantization(myImageDCT, 1, 8);
//      myImageReverseDCT = ReverseDCT(8, myImageDCT);
//      writeBMP(fopen("Files\\2\\img\\myImageR1.bmp", "wb"),
getRGBfromYreverse(myImageReverseDCT.Y, myImageReverseDCT.Cb,
myImageReverseDCT.Cr, heightMyImage, widthMyImage), &bfh1, &bih1, heightMyImage,
widthMyImage);
//
//      quantization(myImageDCT_c1, 5, 8);
//      dequantization(myImageDCT_c1, 5, 8);
//      YCbCr myImageReverseDCT_c1 = ReverseDCT(8, myImageDCT_c1);
//      writeBMP(fopen("Files\\2\\img\\myImageR5.bmp", "wb"),
getRGBfromYreverse(myImageReverseDCT_c1.Y, myImageReverseDCT_c1.Cb,
myImageReverseDCT_c1.Cr, heightMyImage, widthMyImage), &bfh1, &bih1, heightMyImage,
widthMyImage);
//
//      quantization(myImageDCT_c2, 10, 8);
//      dequantization(myImageDCT_c2, 10, 8);
//      YCbCr myImageReverseDCT_c2 = ReverseDCT(8, myImageDCT_c2);
//      writeBMP(fopen("Files\\2\\img\\myImageR10.bmp", "wb"),
getRGBfromYreverse(myImageReverseDCT_c2.Y, myImageReverseDCT_c2.Cb,
myImageReverseDCT_c2.Cr, heightMyImage, widthMyImage), &bfh1, &bih1, heightMyImage,
widthMyImage);
//
//      YCbCr lenaDCT_c1(lenaDCT);
//      YCbCr lenaDCT_c2(lenaDCT);
//
//      quantization(lenaDCT, 1, 8);
//      dequantization(lenaDCT, 1, 8);
//      lenaReverseDCT = ReverseDCT(8, lenaDCT);

```

```

//      writeBMP(fopen("Files\\2\\img\\lenaR1.bmp", "wb"),
getRGBfromYreverse(lenaReverseDCT.Y, lenaReverseDCT.Cb, lenaReverseDCT.Cr,
heightLena, widthLena), &bfh2, &bih2, heightLena, widthLena);
//
//      quantization(lenaDCT_c1, 5, 8);
//      dequantization(lenaDCT_c1, 5, 8);
//      YCbCr lenaReverseDCT_c1 = ReverseDCT(8, lenaDCT_c1);
//      writeBMP(fopen("Files\\2\\img\\lenaR5.bmp", "wb"),
getRGBfromYreverse(lenaReverseDCT_c1.Y, lenaReverseDCT_c1.Cb,
lenaReverseDCT_c1.Cr, heightLena, widthLena), &bfh2, &bih2, heightLena, widthLena);
//
//      quantization(lenaDCT_c2, 10, 8);
//      dequantization(lenaDCT_c2, 10, 8);
//      YCbCr lenaReverseDCT_c2 = ReverseDCT(8, lenaDCT_c2);
//      writeBMP(fopen("Files\\2\\img\\lenaR10.bmp", "wb"),
getRGBfromYreverse(lenaReverseDCT_c2.Y, lenaReverseDCT_c2.Cb,
lenaReverseDCT_c2.Cr, heightLena, widthLena), &bfh2, &bih2, heightLena, widthLena);
//
//      YCbCr baboonDCT_c1(baboonDCT);
//      YCbCr baboonDCT_c2(baboonDCT);
//
//      quantization(baboonDCT, 1, 8);
//      dequantization(baboonDCT, 1, 8);
//      baboonReverseDCT = ReverseDCT(8, baboonDCT);
//      writeBMP(fopen("Files\\2\\img\\baboonR1.bmp", "wb"),
getRGBfromYreverse(baboonReverseDCT.Y, baboonReverseDCT.Cb, baboonReverseDCT.Cr,
heightBaboon, widthBaboon), &bfh3, &bih3, heightBaboon, widthBaboon);
//
//      quantization(baboonDCT_c1, 5, 8);
//      dequantization(baboonDCT_c1, 5, 8);
//      YCbCr baboonReverseDCT_c1 = ReverseDCT(8, baboonDCT_c1);
//      writeBMP(fopen("Files\\2\\img\\baboonR5.bmp", "wb"),
getRGBfromYreverse(baboonReverseDCT_c1.Y, baboonReverseDCT_c1.Cb,
baboonReverseDCT_c1.Cr, heightBaboon, widthBaboon), &bfh3, &bih3, heightBaboon,
widthBaboon);
//
//      quantization(baboonDCT_c2, 10, 8);
//      dequantization(baboonDCT_c2, 10, 8);
//      YCbCr baboonReverseDCT_c2 = ReverseDCT(8, baboonDCT_c2);
//      writeBMP(fopen("Files\\2\\img\\baboonR10.bmp", "wb"),
getRGBfromYreverse(baboonReverseDCT_c2.Y, baboonReverseDCT_c2.Cb,
baboonReverseDCT_c2.Cr, heightBaboon, widthBaboon), &bfh3, &bih3, heightBaboon,
widthBaboon);
//      */
//
//      //DC
////№3.2
//      cout << "\n" << "#3.2\n";
//      cout << "MyImage:" << endl;

```



```

//      /*quantization(myImageDCT, 1, 8);
//      myImageDCT.generateDC();
//      myImageDCT.codingDC();
//      myImageDCT.buildHistogram(myImageDCT.getDC(),
"Files\\3\\3.2\\MyImage\\YmyImageHistogramDC.txt", 0);
//      myImageDCT.buildHistogram(myImageDCT.getCodDC(),
"Files\\3\\3.2\\MyImage\\YmyImageHistogramCodDC.txt", 0);
//      myImageDCT.buildHistogram(myImageDCT.getDC(),
"Files\\3\\3.2\\MyImage\\CbmyImageHistogramDC.txt", 1);
//      myImageDCT.buildHistogram(myImageDCT.getCodDC(),
"Files\\3\\3.2\\MyImage\\CbmyImageHistogramCodDC.txt", 1);
//      myImageDCT.buildHistogram(myImageDCT.getDC(),
"Files\\3\\3.2\\MyImage\\CrmyImageHistogramDC.txt", 2);
//      myImageDCT.buildHistogram(myImageDCT.getCodDC(),
"Files\\3\\3.2\\MyImage\\CrmyImageHistogramCodDC.txt", 2);
//      myImageDCT.calculateEntropy();
//
//      cout << "Lena:" << endl;
//      quantization(lenaDCT, 1, 8);
//      lenaDCT.generateDC();
//      lenaDCT.codingDC();
//      lenaDCT.buildHistogram(lenaDCT.getDC(),
"Files\\3\\3.2\\Lena\\YlenaHistogramDC.txt", 0);
//      lenaDCT.buildHistogram(lenaDCT.getCodDC(),
"Files\\3\\3.2\\Lena\\YlenaHistogramCodDC.txt", 0);
//      lenaDCT.buildHistogram(lenaDCT.getDC(),
"Files\\3\\3.2\\Lena\\CblenaHistogramDC.txt", 1);
//      lenaDCT.buildHistogram(lenaDCT.getCodDC(),
"Files\\3\\3.2\\Lena\\CblenaHistogramCodDC.txt", 1);
//      lenaDCT.buildHistogram(lenaDCT.getDC(),
"Files\\3\\3.2\\Lena\\CrlenaHistogramDC.txt", 2);
//      lenaDCT.buildHistogram(lenaDCT.getCodDC(),
"Files\\3\\3.2\\Lena\\CrlenaHistogramCodDC.txt", 2);
//      lenaDCT.calculateEntropy();
//
//      cout << "Baboon:" << endl;
//      quantization(baboonDCT, 1, 8);
//      baboonDCT.generateDC();
//      baboonDCT.codingDC();
//      baboonDCT.buildHistogram(baboonDCT.getDC(),
"Files\\3\\3.2\\Baboon\\YbaboonHistogramDC.txt", 0);
//      baboonDCT.buildHistogram(baboonDCT.getCodDC(),
"Files\\3\\3.2\\Baboon\\YbaboonHistogramCodDC.txt", 0);
//      baboonDCT.buildHistogram(baboonDCT.getDC(),
"Files\\3\\3.2\\Baboon\\CbBaboonHistogramDC.txt", 1);
//      baboonDCT.buildHistogram(baboonDCT.getCodDC(),
"Files\\3\\3.2\\Baboon\\CbBaboonHistogramCodDC.txt", 1);
//      baboonDCT.buildHistogram(baboonDCT.getDC(),
"Files\\3\\3.2\\Baboon\\CrBaboonHistogramDC.txt", 2);

```

```

//      baboonDCT.buildHistogram(baboonDCT.getCodDC(),
"Files\\3\\3.2\\Baboon\\CrBaboonHistogramCodDC.txt", 2);
//      baboonDCT.calculateEntropy();*/
//
//
////№3.4
//      cout << "\n" << "#3.4\n";
//      YCbCr bmpFile(baboonDCT);
//      YCbCr bmpFile1(baboonDCT);
//
//      cout << endl;
//      cout << "R = " << 1 << endl;
//      quantization(bmpFile, 1, 8);
//      bmpFile.generateDC();
//      bmpFile.codingDC();
//      cout << "Y:" << endl;
//      vector<vector<int>>> yAC = bmpFile.generateAC(bmpFile.getY());
//      vector<vector<pair<unsigned char, pair<unsigned char, int>>>> yCodAC =
bmpFile.codingAC(yAC);
//      bmpFile.sizeOfStream(yCodAC, 0);
//      cout << endl;
//      cout << "Cb:" << endl;
//      vector<vector<int>>> cbAC = bmpFile.generateAC(bmpFile.getCb());
//      vector<vector<pair<unsigned char, pair<unsigned char, int>>>> cbCodAC =
bmpFile.codingAC(cbAC);
//      bmpFile.sizeOfStream(cbCodAC, 1);
//      cout << endl;
//      cout << "Cr:" << endl;
//      vector<vector<int>>> crAC = bmpFile.generateAC(bmpFile.getCr());
//      vector<vector<pair<unsigned char, pair<unsigned char, int>>>> crCodAC =
bmpFile.codingAC(crAC);
//      bmpFile.sizeOfStream(crCodAC, 2);
//      cout << endl << endl;
//      cout << "R = " << 10 << endl;
//      quantization(bmpFile1, 10, 8);
//      bmpFile1.generateDC();
//      bmpFile1.codingDC();
//      cout << "Y:" << endl;
//      vector<vector<int>>> yAC1 = bmpFile1.generateAC(bmpFile1.getY());
//      vector<vector<pair<unsigned char, pair<unsigned char, int>>>> yCodAC1 =
bmpFile1.codingAC(yAC1);
//      bmpFile1.sizeOfStream(yCodAC1, 0);
//      cout << endl;
//      cout << "Cb:" << endl;
//      vector<vector<int>>> cbAC1 = bmpFile1.generateAC(bmpFile1.getCb());
//      vector<vector<pair<unsigned char, pair<unsigned char, int>>>> cbCodAC1 =
bmpFile1.codingAC(cbAC1);
//      bmpFile1.sizeOfStream(cbCodAC1, 1);
//      cout << endl;

```

```

//      cout << "Cr:" << endl;
//      vector<vector<int>>> crAC1 = bmpFile1.generateAC(bmpFile1.getCr());
//      vector<vector<pair<unsigned char, pair<unsigned char, int>>>> crCodAC1 =
bmpFile1.codingAC(crAC1);
//      bmpFile1.sizeOfStream(crCodAC1, 2);
//
//      //Графики
//      cout << "grafics\n";
//      for (int i = 0; i <= 10; i++) {
//          YCbCr bmpFile(baboonDCT);
//          cout << endl;
//          cout << "R = " << i << endl;
//          quantization(bmpFile, i, 8);
//          bmpFile.generateDC();
//          bmpFile.codingDC();
//          vector<vector<int>>> yAC = bmpFile.generateAC(bmpFile.getY());
//          vector<vector<pair<unsigned char, pair<unsigned char, int>>>> yCodAC =
bmpFile.codingAC(yAC);
//          bmpFile.sizeOfStream(yCodAC, 0);
//          vector<vector<int>>> cbAC = bmpFile.generateAC(bmpFile.getCb());
//          vector<vector<pair<unsigned char, pair<unsigned char, int>>>> cbCodAC =
bmpFile.codingAC(cbAC);
//          bmpFile.sizeOfStream(cbCodAC, 1);
//          vector<vector<int>>> crAC = bmpFile.generateAC(bmpFile.getCr());
//          vector<vector<pair<unsigned char, pair<unsigned char, int>>>> crCodAC =
bmpFile.codingAC(crAC);
//          bmpFile.sizeOfStream(crCodAC, 2);
//      }
//
//      ImagesStruct imagesStructNew;
//      for (int R = 0; R <= 10; R++) {
//          buildPSNRgraphics(myImageDCT, lenaDCT, baboonDCT, myImage, lena,
baboon, imagesStructNew, R);
//      }
//
//      writeFile("Files\\3\\3.4\\YMyImagePSNR.txt", imagesStructNew.myImageY,
printDop(myImageDCT, 0));
//      writeFile("Files\\3\\3.4\\YLenaPSNR.txt", imagesStructNew.lenaY, printDop(lenaDCT,
0));
//      writeFile("Files\\3\\3.4\\YBaboonPSNR.txt", imagesStructNew.baboonY,
printDop(baboonDCT, 0));
//
//      writeFile("Files\\3\\3.4\\CbMyImagePSNR.txt", imagesStructNew.myImageCb,
printDop(myImageDCT, 1));
//      writeFile("Files\\3\\3.4\\CbLenaPSNR.txt", imagesStructNew.lenaCb,
printDop(lenaDCT, 1));
//      writeFile("Files\\3\\3.4\\CbBaboonPSNR.txt", imagesStructNew.baboonCb,
printDop(baboonDCT, 1));
//

```

```

//      writeFile("Files\\3\\3.4\\CrMyImagePSNR.txt", imagesStructNew.myImageCr,
printDop(myImageDCT, 2));
//      writeFile("Files\\3\\3.4\\CrLenaPSNR.txt", imagesStructNew.lenaCr, printDop(lenaDCT,
2));
//      writeFile("Files\\3\\3.4\\CrBaboonPSNR.txt", imagesStructNew.baboonCr,
printDop(baboonDCT, 2));

////DOP!!!
////My image
//
YCbCr tmpMyImage(rgbMyImage, heightMyImage, widthMyImage);
YCbCr myImageGauss(256, 512);
myImageGauss.generateGaussianNoise(64, 256, 512);
YCbCr myImageImpulse(rgbMyImage, heightMyImage, widthMyImage);
myImageImpulse.generateImpulseNoise();
YCbCr myImageGaussDCT = DCT(8, myImageGauss);
YCbCr myImageImpulseDCT = DCT(8, myImageImpulse);
myImageDCT = DCT(8, tmpMyImage);
writeBMP(fopen("Files\\dop\\myImageImpulse.bmp", "wb"),
getRGBfromYreverse(myImageImpulse.Y, myImageImpulse.Y, myImageImpulse.Y,
myImageImpulse.height, myImageImpulse.width), &bfh2, &bih2, myImageImpulse.height,
myImageImpulse.width);

writeBMP(fopen("Files\\dop\\myImageGaussian.bmp", "wb"),
getRGBfromYreverse(myImageGauss.Y, myImageGauss.Y, myImageGauss.Y,
myImageGauss.height, myImageGauss.width), &bfh2, &bih2, 256, 512);
ImagesStruct imagesStruct1;
for (int R = 0; R <= 10; R++) {
    buildPSNRgraphics(myImageDCT, myImageGaussDCT, myImageImpulseDCT,
myImage, myImageGauss, myImageImpulse, imagesStruct1, R);
}
writeFile("Files\\dop\\myImageOriginalPSNR.txt", imagesStruct1.myImageY,
printDop(myImageDCT, 0));
writeFile("Files\\dop\\myImageGaussianPSNR.txt", imagesStruct1.lenaY,
printDop(myImageGaussDCT, 0));
writeFile("Files\\dop\\myImageImpulsePSNR.txt", imagesStruct1.baboonY,
printDop(myImageImpulseDCT, 0));

```