Цель работы: изучение алгоритмов, используемых в базовом (baseline) режиме стандарта JPEG, анализ статистических свойств, используемых при сжатии коэффициентов дискретного косинусного преобразования (ДКП), а также получение практических навыков разработки методов блоковой обработки при сжатии изображений с потерями.

1 Дискретное косинусное преобразование (ДКП)

1.1 Реализовать процедуру прямого и обратного ДКП для блоков N×N. Выход обоих преобразований сделать целочисленным с помощью операции округления.

Прямую и обратную процедуру выполнения ДКП нагляднее описывать в форме матричного умножения:

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

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

где $T -$ матрица преобразования размерности $N \times N$. Строки матрицы состоят из векторов, образованных значениями косинусов:

$$\sqrt{C_f} \cos{(\theta_t \cdot f)}$$

где $f -$ соответствует номеру строки, $C_f -$ нормирующий коэффициент, а $\theta_t -$ положение в пространстве $t -$отсчета, вычисляемое по формуле:

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

Нормирующий коэффициент вычисляется:

$$C_f = \begin{cases} \frac{1}{N}, \text{если } f = 0 \\ \frac{2}{N}, \text{иначе} \end{cases}$$

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

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

$$k = 0, \dots, N-1; l = 0, \dots, N-1$$

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

$$i = 0, \dots, N-1; j = 0, \dots, N-1$$

В стандарте JPEG размерность двумерного преобразования N×N равна 8. Результатом ДКП является матрица размером 8×8. Ее элементы принято называть спектральными коэффициентами. Коэффициент на позиции (0, 0) принято называть коэффициентом постоянного тока. Его обозначают DC от английского Direct Current, поскольку его значение получено в результате использования двух функций косинусов нулевой частоты: по горизонтали и по вертикали. Остальные спектральные коэффициенты называют коэффициентами переменного тока и обозначают AC от английского Alternating Current.

1.2 Оценить искажения, вносимые ДКП

В результате применения процедуры прямого и обратного ДКП, были получены изображения и значения PSNR:

Рисунок 1 - Исходное изображение Lena

Рисунок 2 - Изображения Lena после ДКП

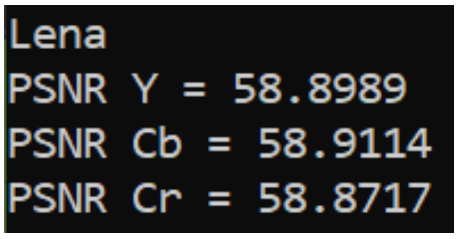


Рисунок 3 – Значения PSNR для Lena

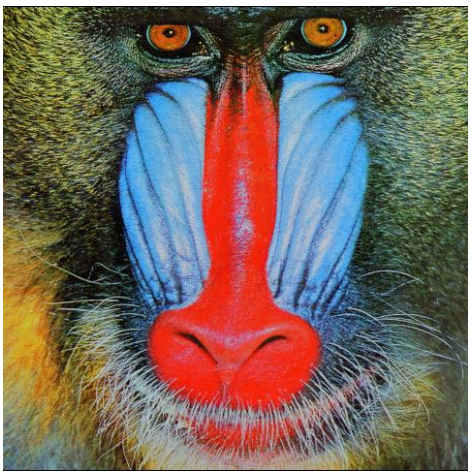


Рисунок 4 - Исходное изображение Baboon

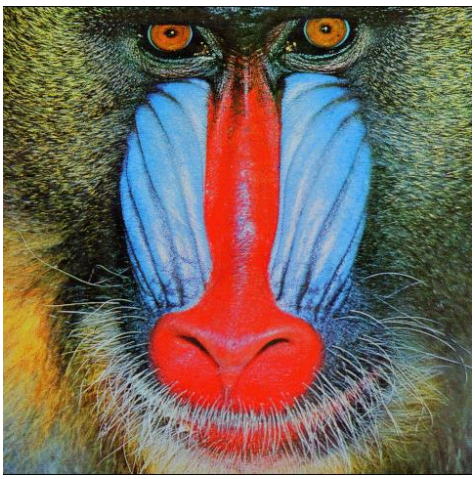


Рисунок 5 - Изображение Baboon после ДКП

```
Baboon
PSNR Y = 58.9312
PSNR Cb = 58.9111
PSNR Cr = 58.9088
```

Рисунок 6 - Значения PSNR для Baboon

Рисунок 7 - Исходное изображение Original

Рисунок 8 - Изображение Original после ДКП

```
Original
PSNR Y = 58.9443
PSNR Cb = 59.2952
PSNR Cr = 59.0274
```

Рисунок 9 - Значение PSNR для Original

По полученным изображениям и высоким значениям PSNR можно сказать о том, что процедура прямого и обратного ДКП почти не вносит изменений в исходное изображение.

2 Квантование спектральных коэффициентов

2.1 Реализация процедуры квантования и деквантования

Формально процедуру квантования спектральных коэффициентов Yi,j можно определить следующим образом:

$$Y_{i,j}^{q} = round\left(\frac{Y_{i,j}}{q_{i,j}^{(c)}}\right), i = 0, \dots 7; j = 0, . . ,7$$

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

При деквантовании будет вычисляться аппроксимирующее значение каждого кванта $Y_{i,j}^{dq}$:

$$Y_{i,j}^{dq} = Y_{i,j}^{q} \cdot q_{i,j}^{(c)}, i = 0, \dots ,7; j = 0, \dots ,7$$

Для упрощения процедуры построения матриц квантования приводят следующую формулу:

$$q_{i,j}^{Y}(R) = 1 + (i + j)R, i = 0, \dots ,7; j = 0, \dots ,7$$

где $R -$ целочисленный параметр, управляющий качеством обработки.

2.2 Оценка влияний искажений

Для оценки влияния искажений необходимо последовательно выполнить процедуры ДКП, квантования, деквантования и обратного ДКП. В результате этих шагов при различных параметрах R были получены изображения:

Рисунок 10 - Исходное изображение Lena

Рисунок 11 - Полученное изображения для R = 1

Рисунок 12 - Полученное изображение для R = 5

Рисунок 13 - Полученное изображение для R = 10

Даже визуально можно заметить, что с увеличением параметра R качество полученного изображения уменьшается. Об этом же свидетельствует и графики зависимости $PSNR(R)$:

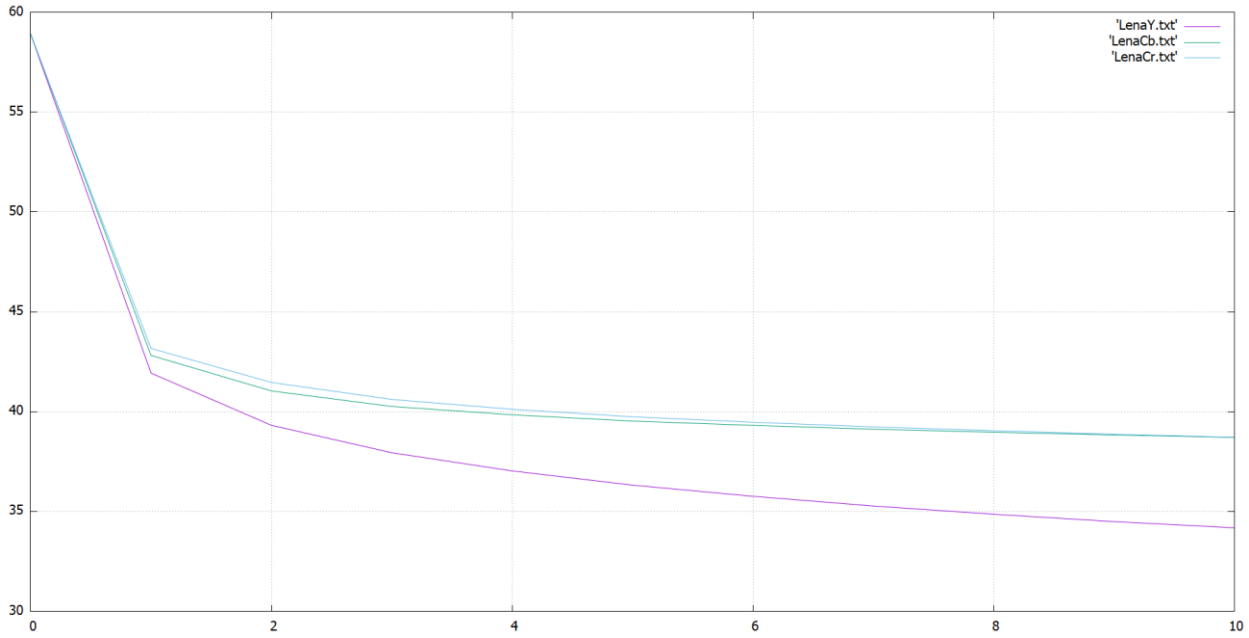


Рисунок 14 - PSNR(R) для изображения Lena

Аналогично и для других изображений:

Рисунок 15 - Исходное изображение Baboon

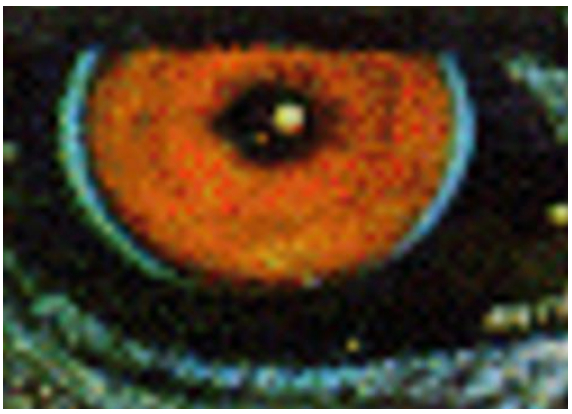


Рисунок 16 - Полученное изображение для R = 1

Рисунок 17 - Полученное изображение для R = 5

Рисунок 18 - Полученное изображение для R = 10

Значения $PSNR(R)$ для изображения Baboon:

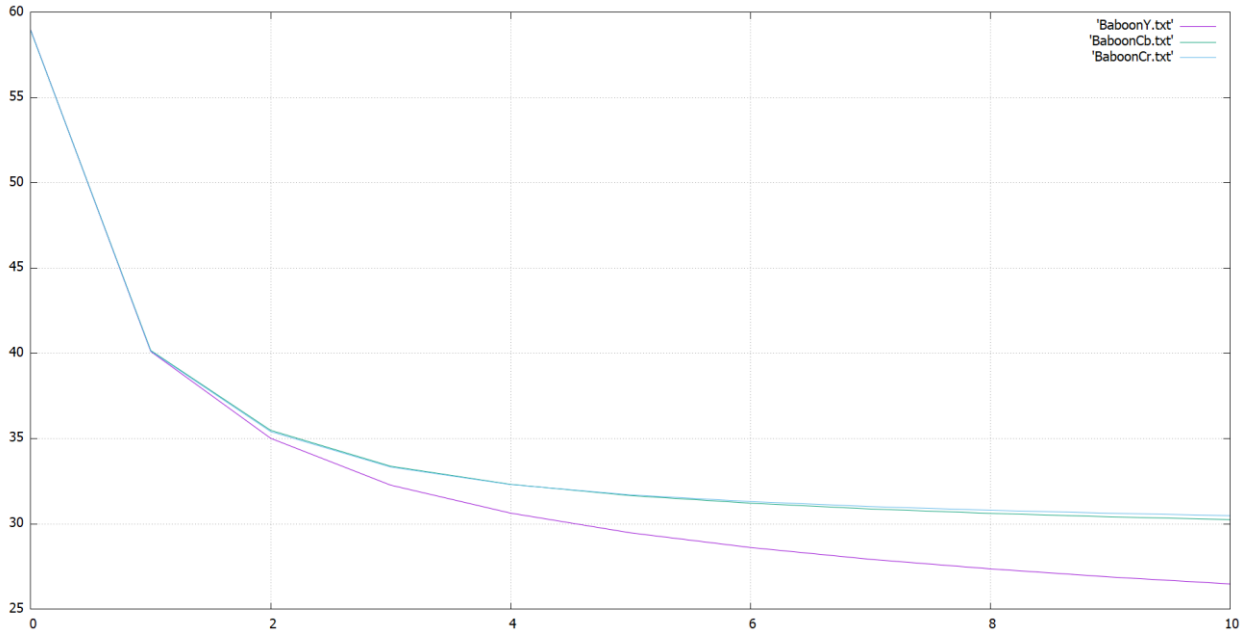


Рисунок 19 - PSNR(R) для Baboon

Рисунок 20 - Исходное изображение Original

Рисунок 21 - Полученное изображения для R = 1

Рисунок 22 - Полученное изображение для R = 5

Рисунок 23 - Полученное изображение для R = 10

График $PSNR(R)$ для изображения Original:

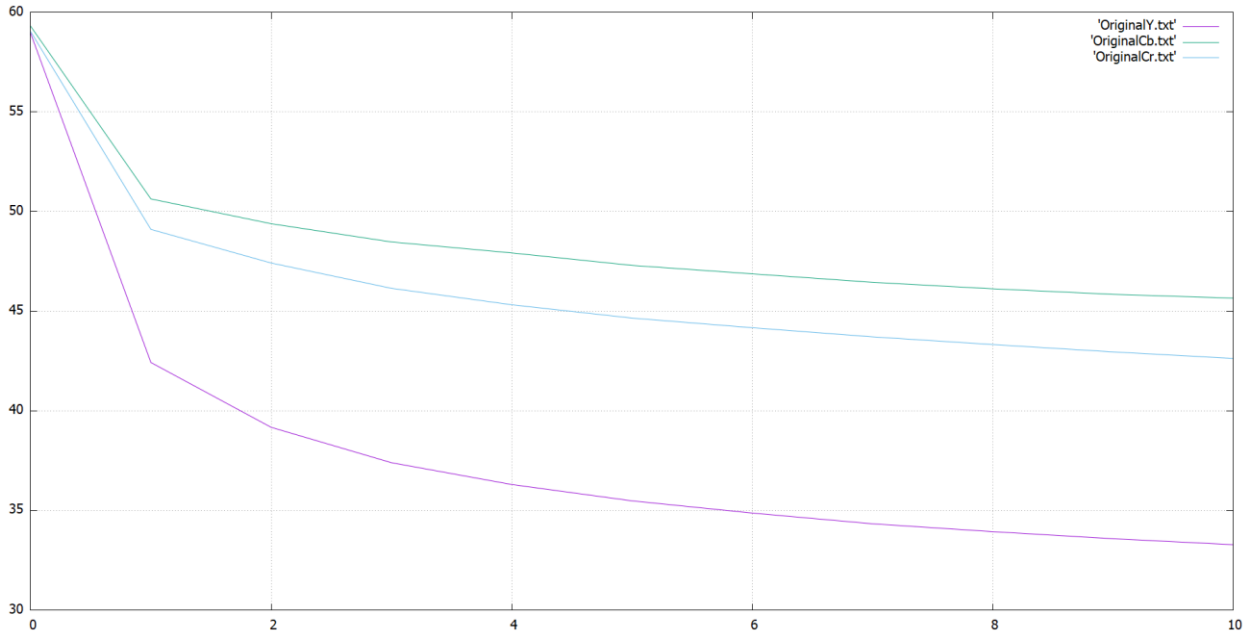


Рисунок 24 - PSNR(R) для Original

Таким образом, по всем трем изображениям и графикам $PSNR(R)$ можно сделать вывод, что при квантовании и деквантовании с увеличением R качество изображения ухудшается,

как и значение PSNR. Также стоит отметить, что PSNR для яркостной компоненты Y уменьшается быстрее. Это объясняется тем, что в данной компоненте содержится больше информации.

3 Сжатие без потерь

Кодирование каждой компоненты осуществляется независимо. Обход блоков 8 × 8 каждой компоненты изображения осуществляется в сканирующем порядке. Для каждого блока выполняются следующие действия:

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 последовательности из 63 AC^q коэффициентов.

4) Кодирование пар (Run, Level)

3.1 Процедура кодирования квантованных коэффициентов постоянного тока DC

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

$$\Delta_{DC} = DC_i^q - DC_{i-1}^q$$

где $i -$ номер текущего обрабатываемого блока. Для первого блока значение Δ_{DC} вычисляется как разность DC_0^q и среднего по всем значениям DC^q.

Значение Δ_{DC} представляется в форме битовой категории и амплитуды. Битовая категория числа x $BC(x)$ вычисляется по формуле:

$$BC(x) = \lceil \log_2(|x| + 1) \rceil$$

Определение битовой категории осуществляется по формуле:

$$\{BC == i\}: \left[-2^i + 1, -2^{i-1}\right], [2^{i-1}, 2^i - 1]$$

где $i -$ номер строки. Можно представить битовые категории в виде таблицы:

Категория	*Амплитуда*
0	—
1	-1, 1
2	-3, -2, 2, 3
3	7,…, -4, 4,…7
4	-15,…, -8, 8,…15
5	-31,…-16, 16,…31
6	-63,…-32, 32,…63
7	-127,…-64, 64,…127
8	-255,…-128, 128,…255
9	-511,…-256, 256,…511
10	-1023,…-512, 512,…1023
11	-2047,…-1024, 1024,…2047
12	-4095,…-2048, 2048,…4095
13	-8191,…-4096, 4096,…8191
14	-16383,…-8192, 8192,…16383
15	-32767,…-16384, 16384,…32767
16	-32768

Рисунок 25 - Битовые категории целых чисел

Амплитудой MG фактически является само кодируемое число.

3.2 Оценка эффективности использования разностного кодирования для коэффициентов постоянного тока

Для оценки эффективности использования разностного кодирования необходимо построить гистограммы частот DC и Δ_{DC} и вычислить энтропии H по формуле:

$$H = -\sum \hat{p}(x) \log_2 \hat{p}(x)$$

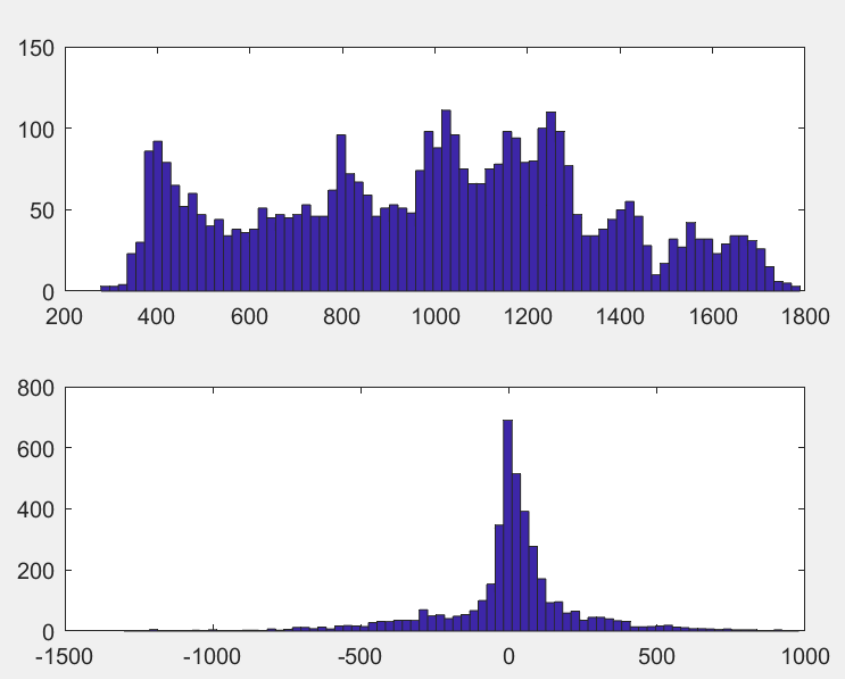


Рисунок 26 - Гистограмма для компоненты Y изображения Lena

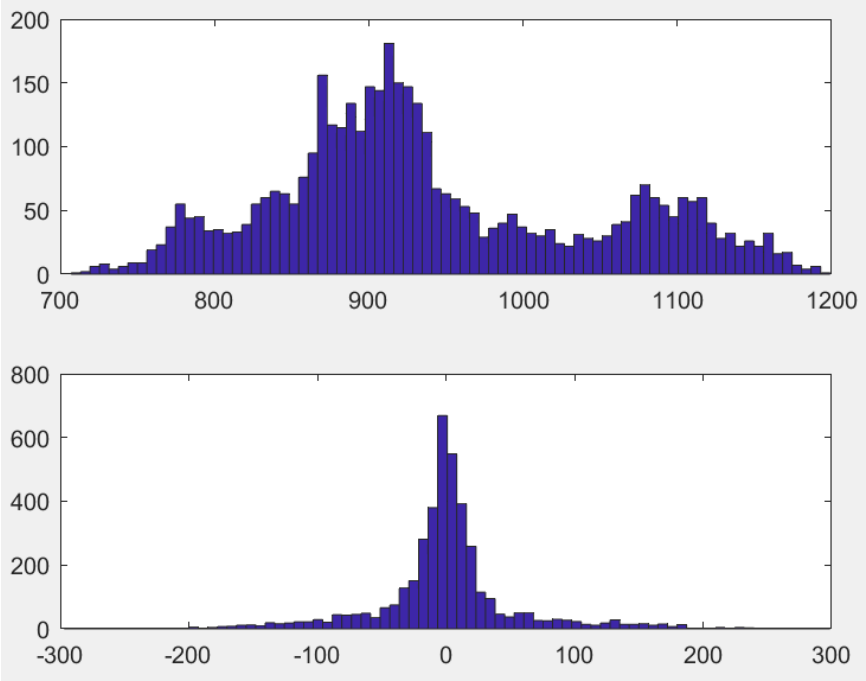


Рисунок 27 - Гистограмма для компоненты Cb изображения Lena

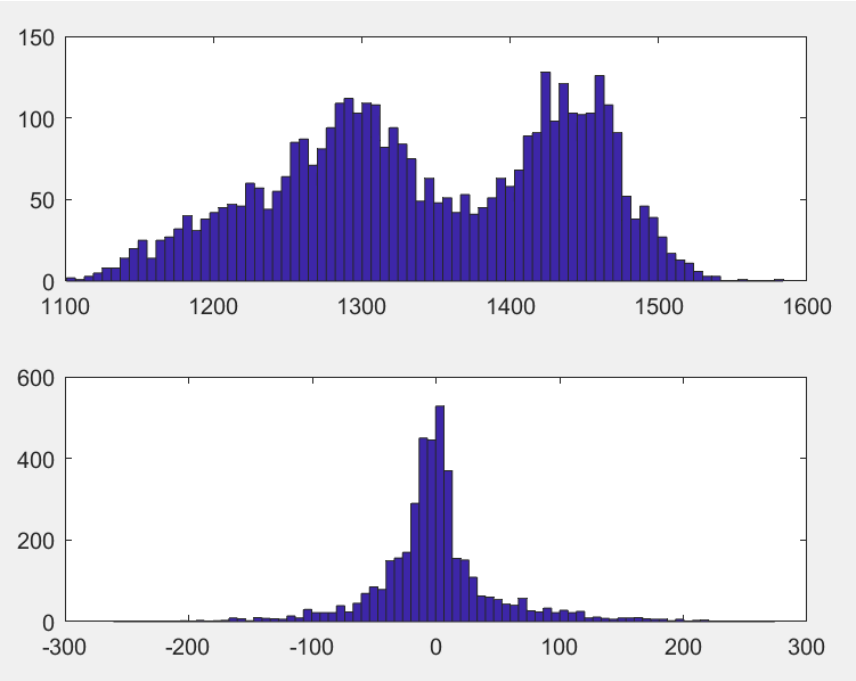


Рисунок 28 – Гистограмма для компоненты Cr изображения Lena

Значения энтропии каждой из компонент до кодирования и после:

```
H(DC_Y): 10.0706
H(DC_Cb): 8.42556
H(DC_Cr): 8.39537

H(code_DC_Y): 9.11451
H(code_DC_Cb): 7.27582
H(code_DC_Cr): 7.27492
```

Рисунок 29 - Энтропия для изображения Lena

Как видно по гистограмме значения каждой из компонент локализуются после кодирования около 0, а энтропия уменьшается. Это же можно увидеть и для остальных изображений.

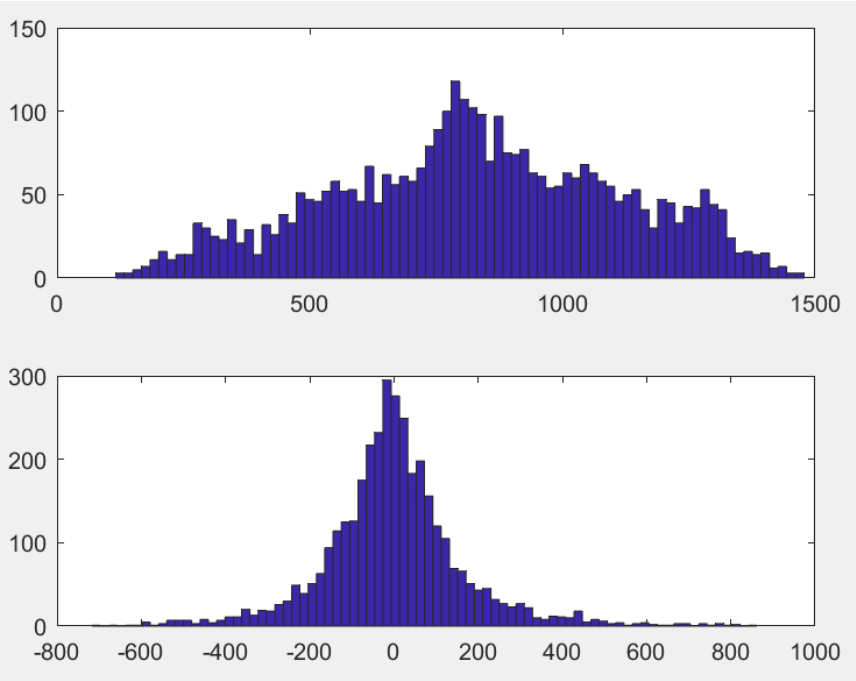


Рисунок 30 - Гистограмма для компоненты Y изображения Baboon

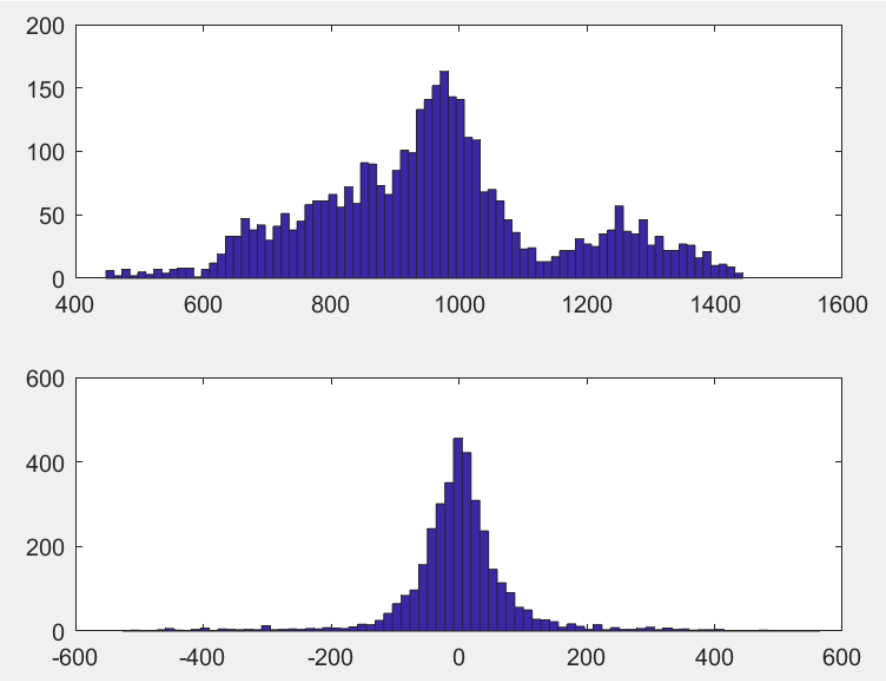


Рисунок 31 - Гистограмма для компоненты Cb изображения Baboon

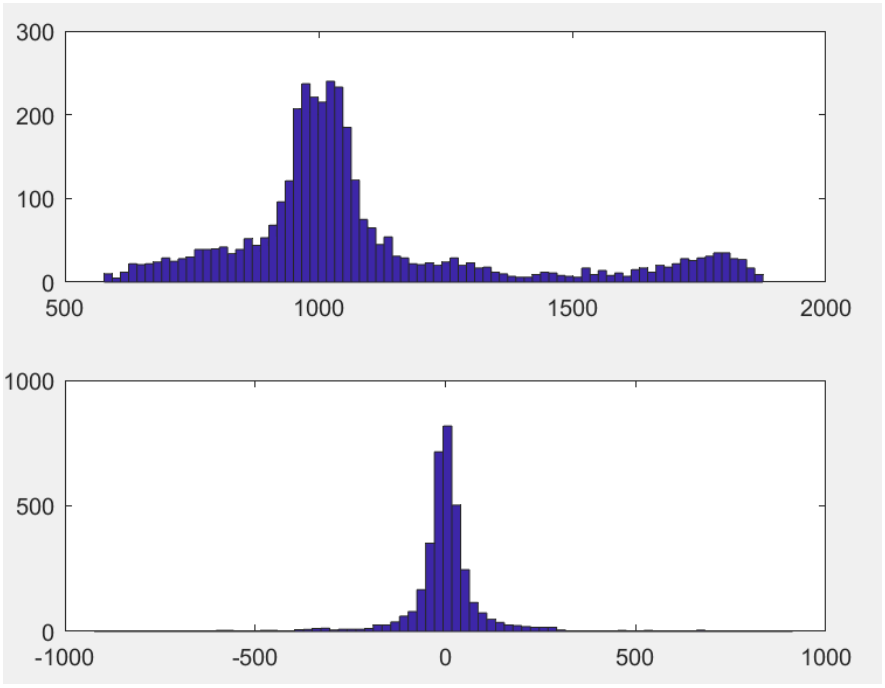


Рисунок 32 - Гистограмма для компоненты Cr изображения Baboon

Значения энтропии каждой из компонент до кодирования и после:

```
H(DC_Y): 9.85194
H(DC_Cb): 9.25251
H(DC_Cr): 9.24871

H(code_DC_Y): 9.0228
H(code_DC_Cb): 7.99137
H(code_DC_Cr): 7.94209
```

Рисунок 33 – Энтропия для изображения Baboon

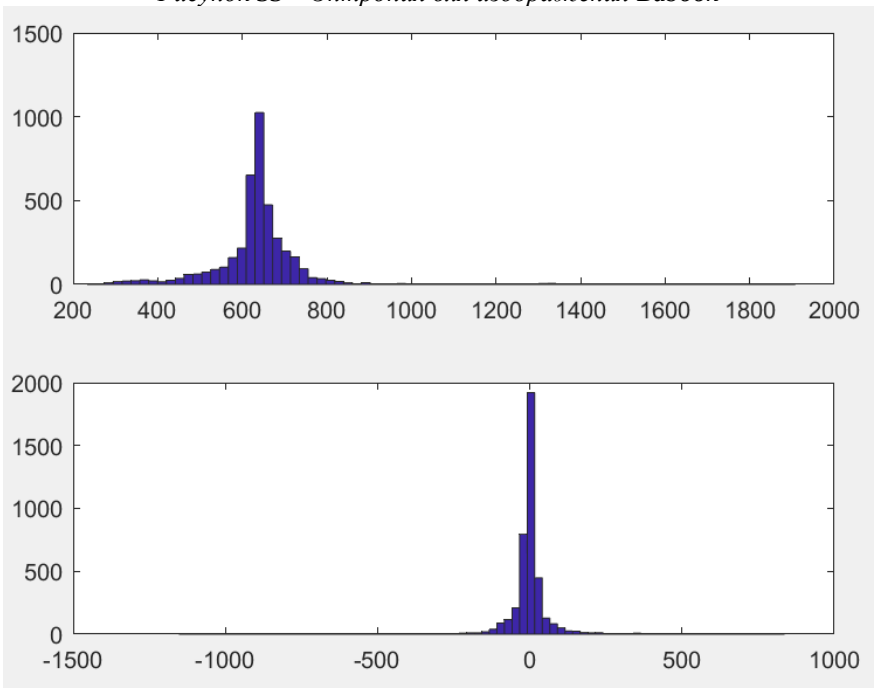


Рисунок 34 - Гистограмма для компоненты Y изображения Original

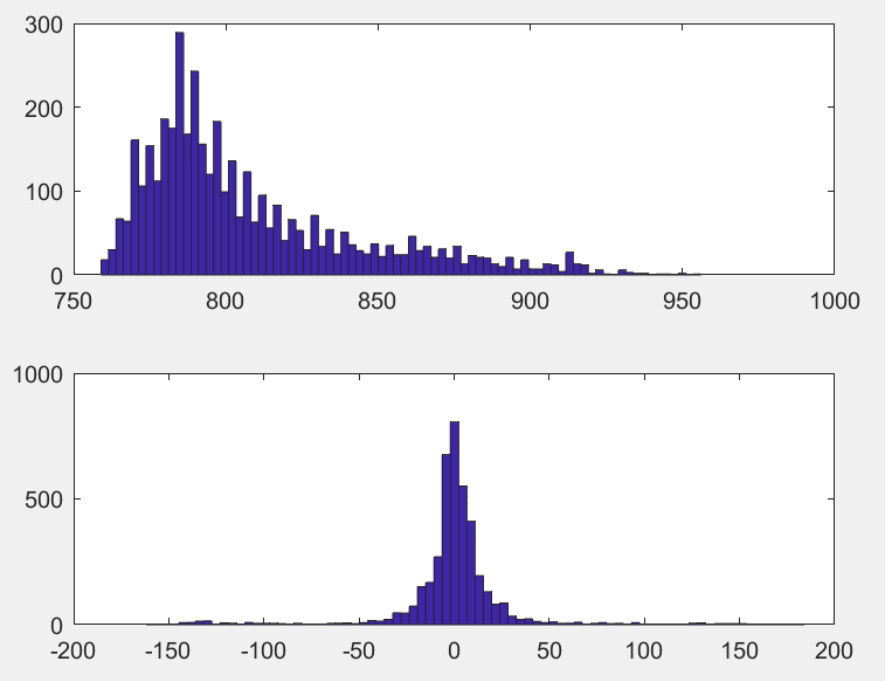


Рисунок 35 - Гистограмма для компоненты Cb изображения Original

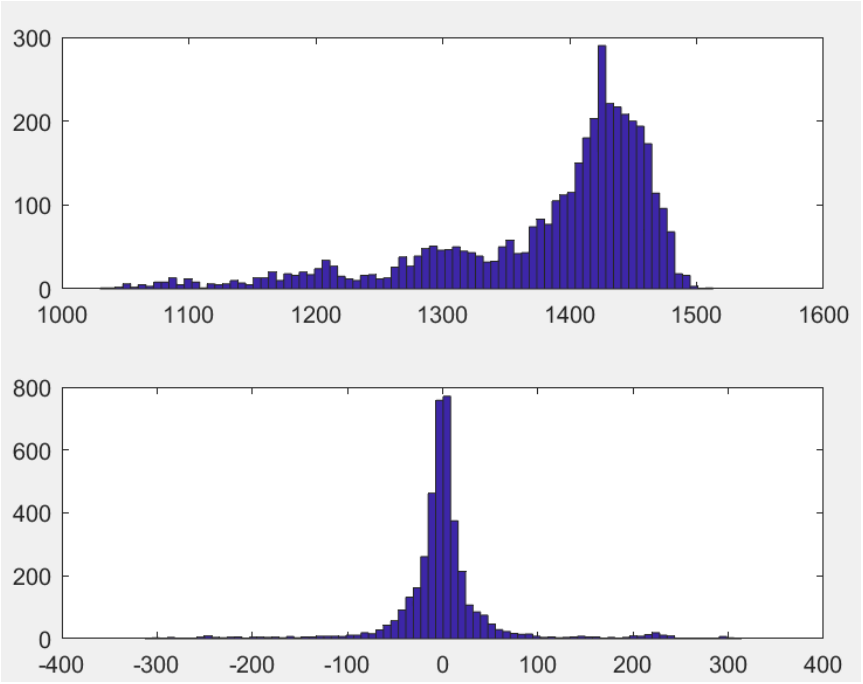


Рисунок 36 - Гистограмма для компоненты Cr изображения Original

Значения энтропии каждой из компонент до кодирования и после:

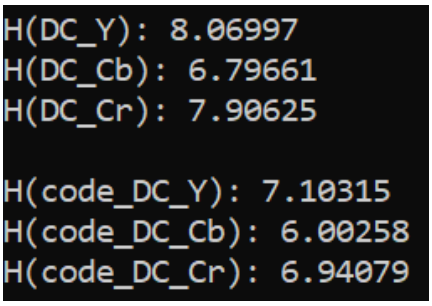


Рисунок 37 - Энтропия для изображения Original

Таким образом, анализируя гистограммы частот и значения энтропии можно сделать вывод о том, что в результате кодирования энтропия уменьшается, а значения локализуются около 0.

4 Кодирование длинами серий (RLE)

Процедура для каждого блока 8 × 8 из квантованных коэффициентов переменного тока AC включает в себя:

1) Перегруппировку AC^q коэффициентов в соответствии с зигзагообразной последовательностью. Новая последовательность будет обладать следующим свойством — значимые (ненулевые) коэффициенты будут преобладать в ее начале, а ближе к концу последовательности в основном будут находиться нулевые значения.

2) Замену сформированной последовательности на последовательность пар (Run, Level)

3) Формирование для каждой пары (Run, Level) на основе значения Level новой пары — (BC(Level), Magnitude(Level))

4) Формирование троек ((Run, BC(Level)), Magnitude(Level)).

Сформированная на предыдущем шаге последовательность коэффициентов AC^q кодируется длинами серий. Результатом будет являться новая последовательность, состоящая из пар (Run/Level). Здесь Run определяет число нулевых AC^q коэффициентов в серии. Каждая серия завершается Level — ненулевым значением AC^q коэффициента. Обработка завершается, когда закодирован последний ненулевой AC^q коэффициент. При этом в поток вставляется дополнительная специальная пара (0, 0), которая носит название End Of Block (EOB). Если длина серии, которая не является последней в блоке, превышает 15 нулей, то она разбивается на несколько подсерий, каждая из которых содержит не более 16 нулей. Подпоследовательность из шестнадцати нулей кодируется символом (15, 0). Такая пара носит специальное название Zero Run Length (ZRL).

Значение Level, если оно не относится к паре ZRL или EOB, кодируется по битовым категориям, формируя пару (BC(Level), Magnitude(Level)). В битовый поток записывается значение амплитуды Magnitude(Level), состоящее из BC(Level) бит.

4.1 Определение соотношения размеров в сжатом битовом потоке для следующих данных:

– BC(ΔDC) ,

– Magnitude(ΔDC) ,

– (Run, BC(Level)),

– Magnitude(Level).

В ходе выполнения были получены следующие значения:

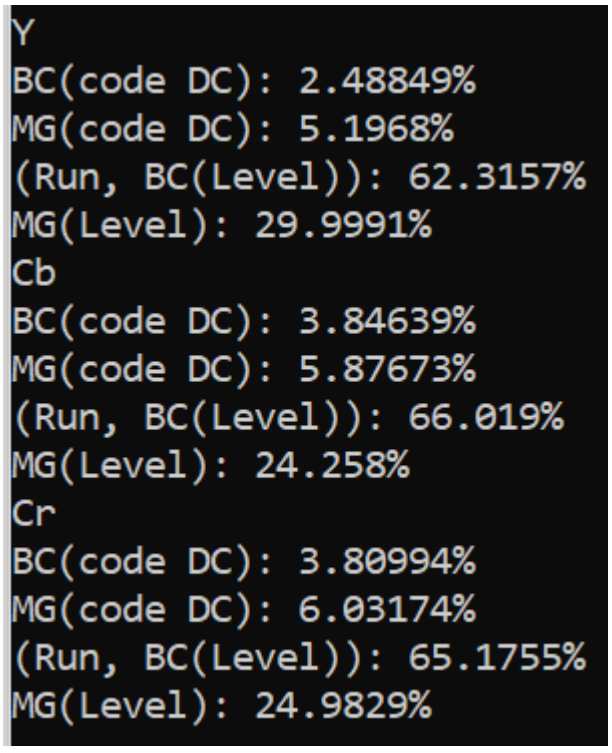


Рисунок 38 - Процент каждой составляющей для R = 1 изображения Lena

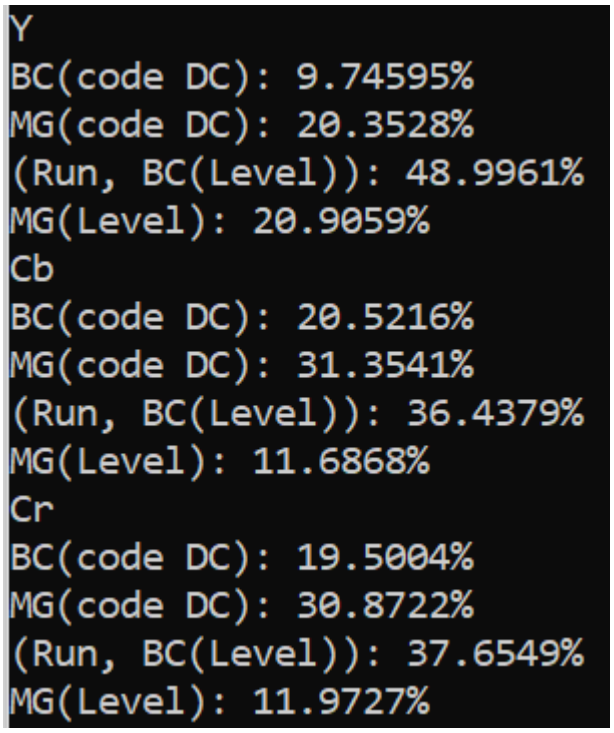


Рисунок 39 - Процент каждой составляющей для R = 10 изображения Lena

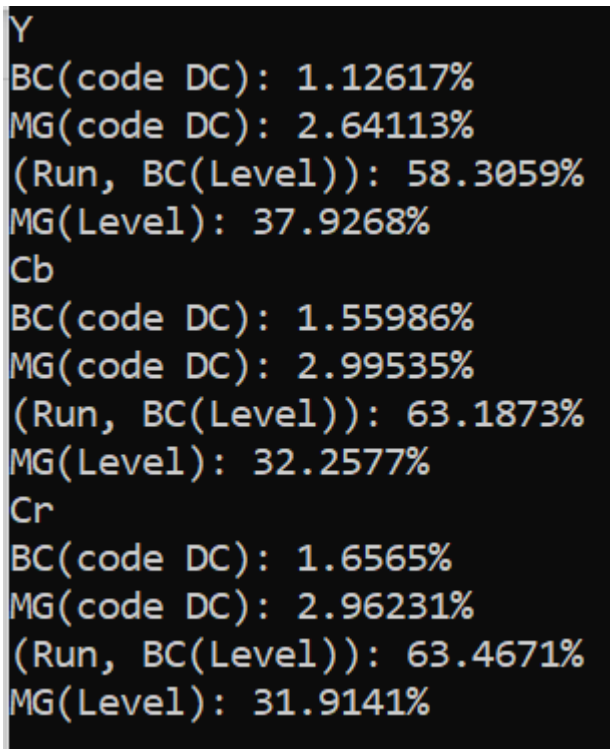


Рисунок 40 - Процент каждой составляющей для R = 1 изображения Baboon

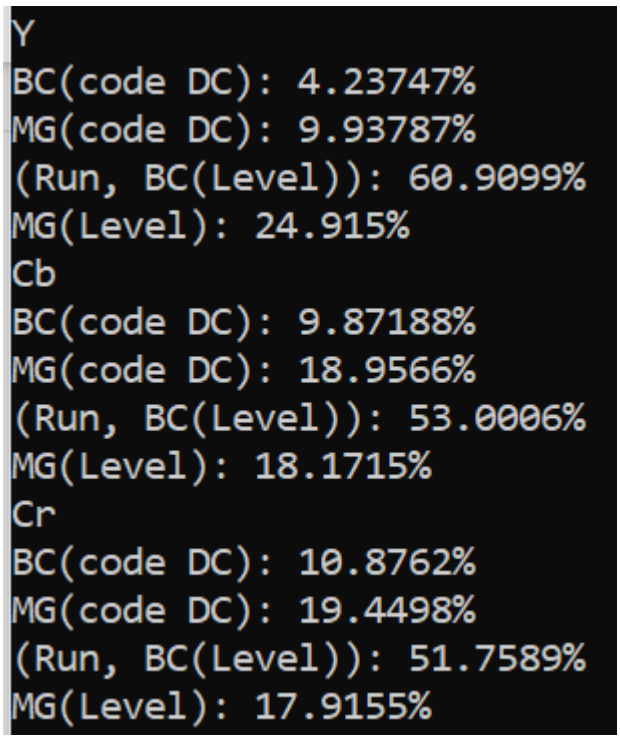


Рисунок 41 - Процент каждой составляющей для R = 10 изображения Baboon

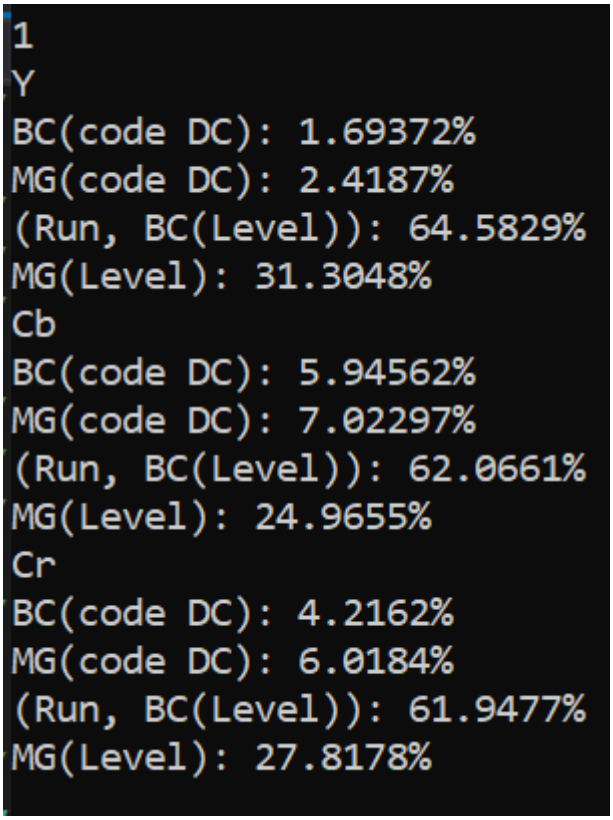


Рисунок 42 - Процент каждой составляющей для R = 1 изображения Original

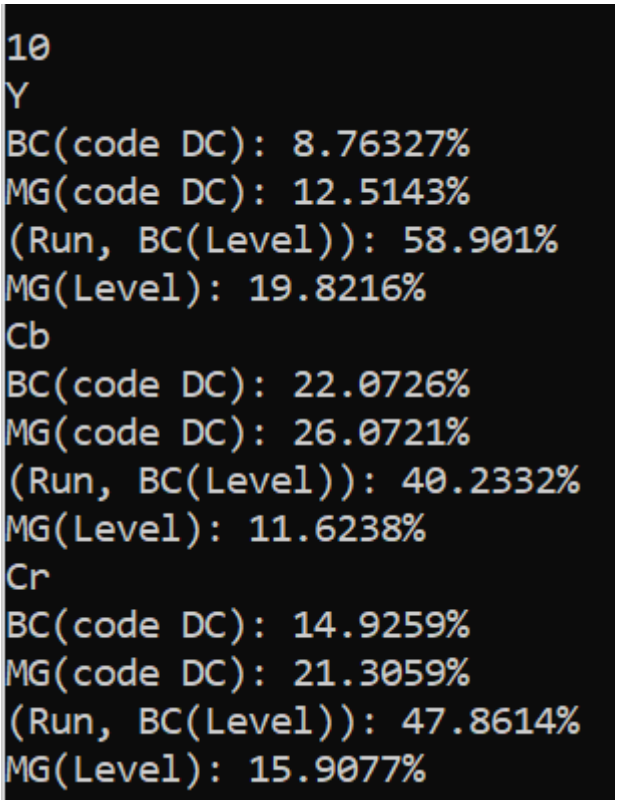


Рисунок 43 - Процент каждой составляющей для R = 10 изображения Original

Для оценки размера сжатого потока для различных значений квантователя R был получен график $PSNR(\frac{\text{поток до сжатия}}{\text{поток после сжатия}})$ для каждой из компонент:

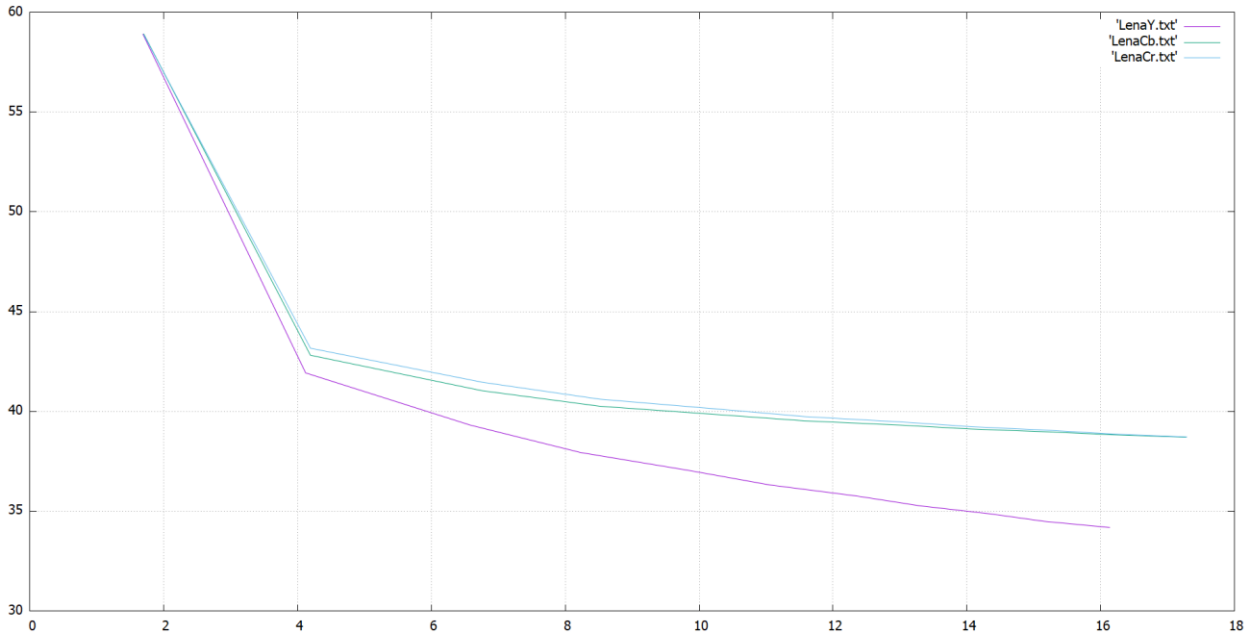


Рисунок 44 – PSNR(степень сжатия) для изображения Lena

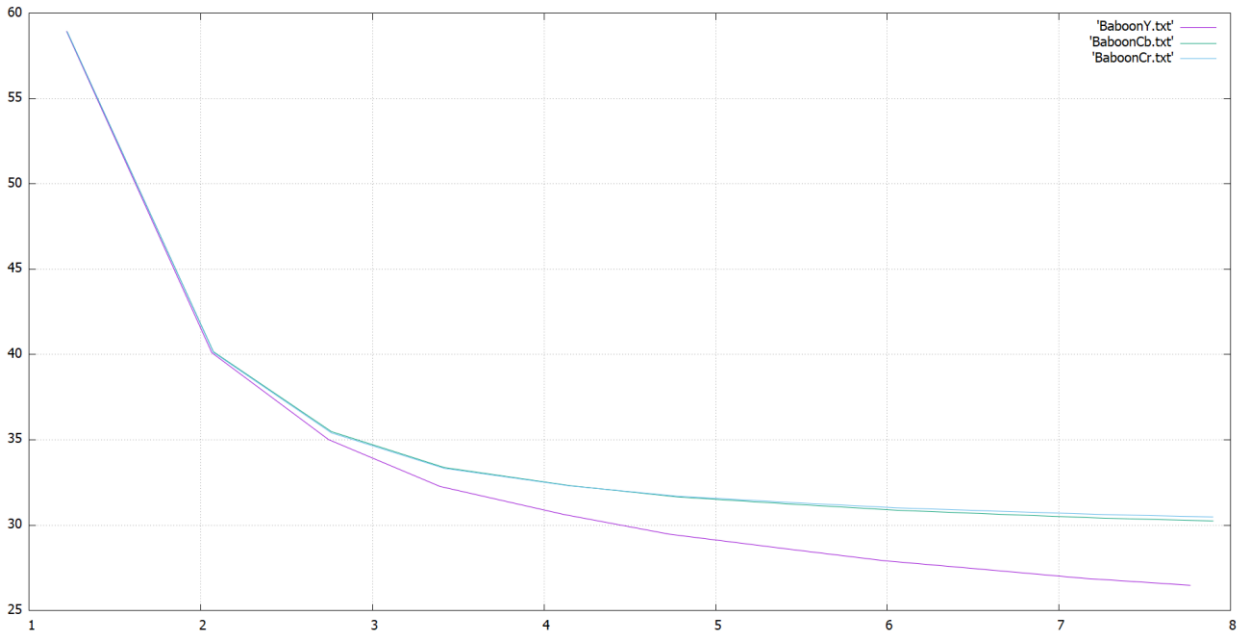


Рисунок 45 - PSNR(степень сжатия) для изображения Baboon

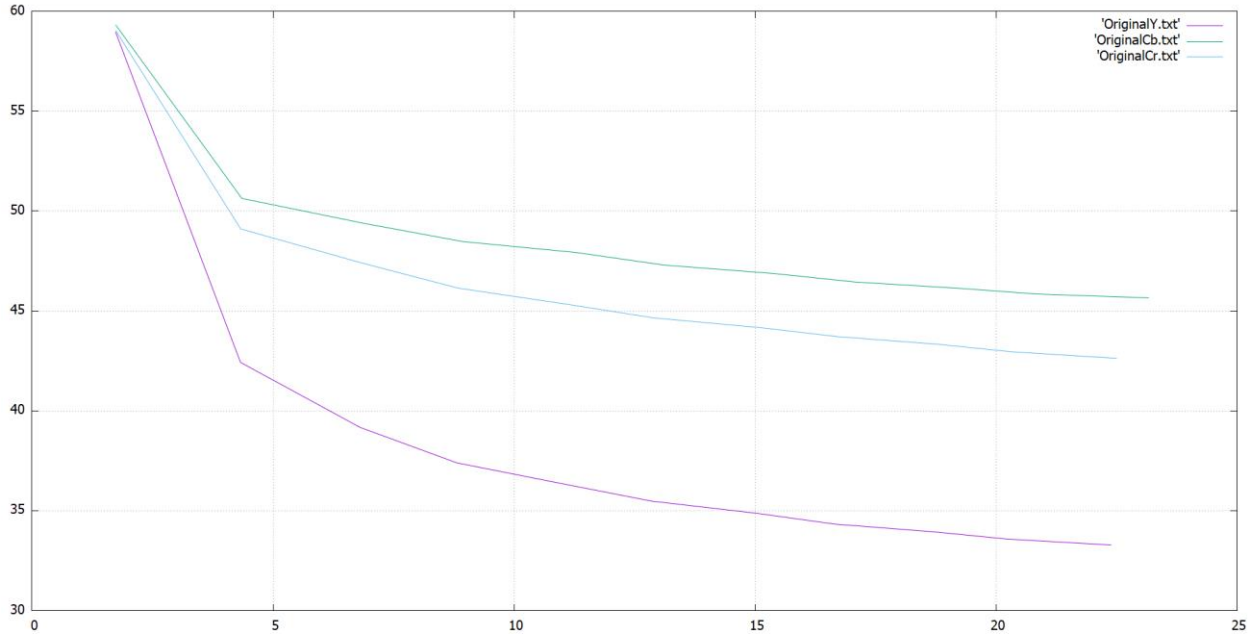
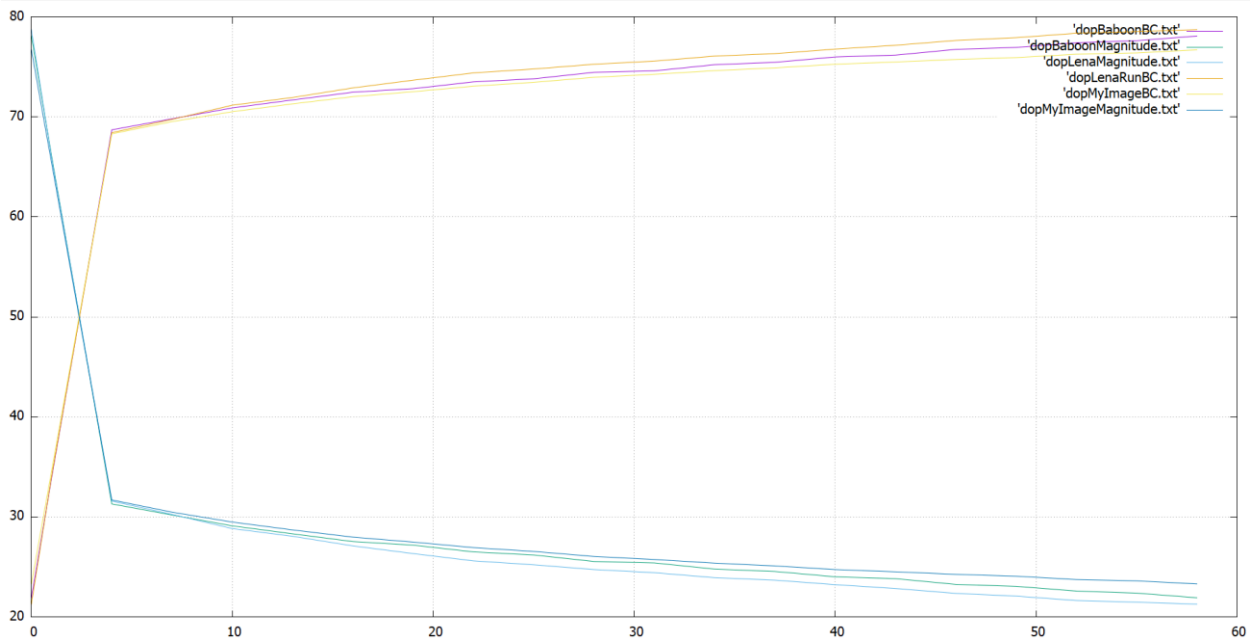


Рисунок 46 - PSNR(степень сжатия) для изображения Original

Дополнительное задание.

Необходимо построить на одном графике зависимости ВС(Run, Level) в сжатии данных для AC коэффициентов от параметра R и Magnitude(Level) в сжатии данных для AC коэффициентов от параметра R в процентах.



Исходя из полученных графиков, можно сделать вывод о том, что с увеличением R значение ВС (Run, Level) увеличивается, а Magnitude (Level) уменьшается, что говорит о том, что при

увеличении R число нулевых позиций увеличивается, а значение ненулевого элемента уменьшается.

5 Выводы

В ходе выполнения работы было выполнено прямое и обратное ДКП. Было выявлено, что оно никак не влияет на качество изображения, однако позволяет построить спектральные коэффициенты, по которым производится квантование.

В результате квантования при увеличении R значение PSNR снижается, а качество визуально ухудшается.

В результате разностного кодирования значения частоты коэффициентов постоянного тока находятся около нуля, что можно видеть по гистограммам. Помимо этого, благодаря использованию разностного кодирования можно уменьшить значение энтропии.

Для значений переменного тока используется кодирование длинами серий. Это возможно благодаря тому, что элементы переменного тока переупорядочиваются в зигзагообразном порядке, и основная энергия концентрируется в левом верхнем углу.

Наибольший процент в сжатом битовом потоке отводится на (Run, BC(Level)), так как этих пар больше всего. С увеличением радиуса R меняется соотношение размеров в сжатом битовом потоке. При больших параметрах сжатия, то есть при больших R, сильно заметны границы блоков, размытие, а PSNR уменьшается.

Листинг программы:

Main.cpp

```
#include <iostream>
#include <vector>
#include <string>
#include <fstream>
#include "bmp.h"
#include "DCT.h"
#include "Quantization.h"
#include "Compression.h"


using namespace std;

void read_PNSR_form_file(vector<double>* PSNR, string filename)
{
	ifstream in;
	in.open(filename);
	double R;
	double psnr;
	while (in >> R >> psnr) {
		PSNR->push_back(psnr);
	}
	in.close();
}

void write_PSNR_to_file(vector<double> compress, vector<double>
PSNR, string filename) {
	ofstream f;
	f.open(filename);
	for (int i = 0; i < compress.size(); i++) {
		f << compress[i] << " " << PSNR[i] << endl;
	}
	f.close();
}


int main() {
	setlocale(LC_ALL, "Russian");

	//открытие и чтение всех файлов
	BITMAPFILEHEADER bfh_lena;
	BITMAPINFOHEADER bih_lena;
	FILE* f_lena = fopen("lena.bmp", "rb");
	if (f_lena == NULL) {
		return -1;
	}
	RGB** rgb_lena = read_bmp(f_lena, &bfh_lena, &bih_lena);
	fclose(f_lena);
	int height_lena = bih_lena.biHeight;
	int width_lena = bih_lena.biWidth;
```

```
     YCbCr** ycbcr_lena = new YCbCr * [height_lena];
     for (int i = 0; i < height_lena; i++) {
          ycbcr_lena[i] = new YCbCr[width_lena];
     }



     BITMAPFILEHEADER bfh_baboon;
     BITMAPINFOHEADER bih_baboon;
     FILE* f_baboon = fopen("baboon.bmp", "rb");
     if (f_baboon == NULL) {
          return -2;
     }
     RGB** rgb_baboon = read_bmp(f_baboon, &bfh_baboon,
&bih_baboon);
     fclose(f_baboon);
     int height_baboon = bih_baboon.biHeight;
     int width_baboon = bih_baboon.biWidth;

     YCbCr** ycbcr_baboon = new YCbCr * [height_baboon];
     for (int i = 0; i < height_baboon; i++) {
          ycbcr_baboon[i] = new YCbCr[width_baboon];
     }

     BITMAPFILEHEADER bfh_original;
     BITMAPINFOHEADER bih_original;
     FILE* f_original = fopen("Original.bmp", "rb");
     if (f_original == NULL) {
          return -1;
     }
     RGB** rgb_original = read_bmp(f_original, &bfh_original,
&bih_original);
     fclose(f_original);
     int height_original = bih_original.biHeight;
     int width_original = bih_original.biWidth;

     YCbCr** ycbcr_original = new YCbCr * [height_original];
     for (int i = 0; i < height_original; i++) {
          ycbcr_original[i] = new YCbCr[width_original];
     }

     ////DCT and IDCT
     cout << endl << "Прямое и обратное ДКП" << endl;

     cout << "Lena" << endl;
     DCT DCT_lena(rgb_lena, ycbcr_lena, height_lena, width_lena,
&bfh_lena, &bih_lena);
     DCT_lena.get_YCbCr();
     DCT_lena.direct_DCT();
     DCT_lena.IDCT();
     DCT_lena.get_image("LenaAfter.bmp");
     DCT_lena.get_PSNR(true);
```

```
     cout << "Baboon" << endl;
     DCT DCT_baboon(rgb_baboon, ycbcr_baboon, height_baboon,
width_baboon, &bfh_baboon, &bih_baboon);
     DCT_baboon.get_YCbCr();
     DCT_baboon.direct_DCT();
     DCT_baboon.IDCT();
     DCT_baboon.get_image("BaboonAfter.bmp");
     DCT_baboon.get_PSNR(true);

     cout << "Original" << endl;
     DCT DCT_original(rgb_original, ycbcr_original,
height_original, width_original, &bfh_original, &bih_original);
     DCT_original.get_YCbCr();
     DCT_original.direct_DCT();
     DCT_original.IDCT();
     DCT_original.get_image("OriginalAfter.bmp");
     DCT_original.get_PSNR(true);


     //// квантование и деквантование с графиками PSNR
     cout << endl << "Квантования и деквантование с графиком
PSNR" << endl;
     cout << "Lena" << endl;
     vector<int> R = { 0,1,2,3,4,5,6,7,8,9,10 };
     Quantization quantization_lena(DCT_lena.get_DCT(),
height_lena, width_lena, &bfh_lena, &bih_lena);
     for (int i = 0; i < R.size(); i++) {
          cout << endl << "R = " << R[i] << endl;
          if (R[i] != 0) {
               quantization_lena.quantization(R[i]);
               quantization_lena.dequantization(R[i]);

     DCT_lena.set_new_YCbCr(quantization_lena.get_dequantization(
));
          }
          DCT_lena.IDCT();
          DCT_lena.get_PSNR_graphic(R[i], false);
          if (R[i] == 1 || R[i] == 5 || R[i] == 10) {
               DCT_lena.get_image(("2//Lena" + to_string(R[i]) +
".bmp").c_str());
          }
     }
     DCT_lena.write_PSNR_file("2//Lena", R);
     cout << "plot 'LenaY.txt' w l, 'LenaCb.txt' w l,
'LenaCr.txt' w l" << endl;

     cout << "Baboon" << endl;
     vector<int> R_baboon = { 0,1,2,3,4,5,6,7,8,9,10 };
     Quantization quantization_baboon(DCT_baboon.get_DCT(),
height_baboon, width_baboon, &bfh_baboon, &bih_baboon);
     for (int i = 0; i < R_baboon.size(); i++) {
          cout << endl << "R = " << R_baboon[i] << endl;
```

```
          if (R_baboon[i] != 0) {
               quantization_baboon.quantization(R_baboon[i]);
               quantization_baboon.dequantization(R_baboon[i]);

     DCT_baboon.set_new_YCbCr(quantization_baboon.get_dequantizat
ion());
          }
          DCT_baboon.IDCT();
          DCT_baboon.get_PSNR_graphic(R_baboon[i], false);
          if (R_baboon[i] == 1 || R_baboon[i] == 5 || R_baboon[i]
== 10) {
               DCT_baboon.get_image(("2//Baboon" +
to_string(R_baboon[i]) + ".bmp").c_str());
          }
     }
     DCT_baboon.write_PSNR_file("2//Baboon", R_baboon);
     cout << "plot 'BaboonY.txt' w l, 'BaboonCb.txt' w l,
'BaboonCr.txt' w l" << endl;


     cout << "Original" << endl;
     vector<int> R_original = { 0,1,2,3,4,5,6,7,8,9,10 };
     Quantization quantization_original(DCT_original.get_DCT(),
height_original, width_original, &bfh_original, &bih_original);
     for (int i = 0; i < R_original.size(); i++) {
          cout << endl << "R = " << R_original[i] << endl;
          if (R_original[i] != 0) {
               quantization_original.quantization(R_original[i]);

     quantization_original.dequantization(R_original[i]);

     DCT_original.set_new_YCbCr(quantization_original.get_dequant
ization());
          }
          DCT_original.IDCT();
          DCT_original.get_PSNR_graphic(R_original[i], false);
          if (R_original[i] == 1 || R_original[i] == 5 ||
R_original[i] == 10) {
               DCT_original.get_image(("2//Original" +
to_string(R_original[i]) + ".bmp").c_str());
          }
     }
     DCT_original.write_PSNR_file("2//Original", R_original);
     cout << "plot 'OriginalY.txt' w l, 'OriginalCb.txt' w l,
'OriginalCr.txt' w l" << endl;


     // кодирование ДС
     cout << endl << "Кодирование квантованных коэффициентов
постоянного тока DC" << endl;
     cout << "Lena" << endl;
     Quantization quantization_lena2(DCT_lena.get_DCT(),
height_lena, width_lena, &bfh_lena, &bih_lena);
```

```
     quantization_lena2.quantization(1);
     Compression
compression_lena(quantization_lena2.get_quantization(),
height_lena, width_lena, &bfh_lena, &bih_lena);
     compression_lena.create_DC();
     compression_lena.coding_DC();

     compression_lena.get_histogram_dc("3//LenaDcY.txt", "Y");
     compression_lena.get_histogram_dc("3//LenaDcCb.txt", "Cb");
     compression_lena.get_histogram_dc("3//LenaDcCr.txt", "Cr");

     compression_lena.get_histogram_code_dc("3//LenaCodeDcY.txt",
"Y");
     compression_lena.get_histogram_code_dc("3//LenaCodeDcCb.txt"
, "Cb");
     compression_lena.get_histogram_code_dc("3//LenaCodeDcCr.txt"
, "Cr");

     compression_lena.entropy();


     cout << "Baboon" << endl;
     Quantization quantization_baboon2(DCT_baboon.get_DCT(),
height_baboon, width_baboon, &bfh_baboon, &bih_baboon);
     quantization_baboon2.quantization(1);
     Compression
compression_baboon(quantization_baboon2.get_quantization(),
height_baboon, width_baboon, &bfh_baboon, &bih_baboon);
     compression_baboon.create_DC();
     compression_baboon.coding_DC();

     compression_baboon.get_histogram_dc("3//BaboonDcY.txt",
"Y");
     compression_baboon.get_histogram_dc("3//BaboonDcCb.txt",
"Cb");
     compression_baboon.get_histogram_dc("3//BaboonDcCr.txt",
"Cr");

     compression_baboon.get_histogram_code_dc("3//BaboonCodeDcY.t
xt", "Y");
     compression_baboon.get_histogram_code_dc("3//BaboonCodeDcCb.
txt", "Cb");
     compression_baboon.get_histogram_code_dc("3//BaboonCodeDcCr.
txt", "Cr");

     compression_baboon.entropy();


     cout << "Original" << endl;
     Quantization quantization_original2(DCT_original.get_DCT(),
height_original, width_original, &bfh_original, &bih_original);
     quantization_original2.quantization(1);
```

```
    Compression
compression_original(quantization_original2.get_quantization(),
height_original, width_original, &bfh_original, &bih_original);
    compression_original.create_DC();
    compression_original.coding_DC();

    compression_original.get_histogram_dc("3//OriginalDcY.txt",
"Y");
    compression_original.get_histogram_dc("3//OriginalDcCb.txt",
"Cb");
    compression_original.get_histogram_dc("3//OriginalDcCr.txt",
"Cr");

    compression_original.get_histogram_code_dc("3//OriginalCodeD
cY.txt", "Y");
    compression_original.get_histogram_code_dc("3//OriginalCodeD
cCb.txt", "Cb");
    compression_original.get_histogram_code_dc("3//OriginalCodeD
cCr.txt", "Cr");

    compression_original.entropy();


    // Кодирование AC, оценка сжатого потока и PSNR
    vector<int> R3 = { 1,10 };
    DCT DCT_lena3(rgb_lena, ycbcr_lena, height_lena, width_lena,
&bfh_lena, &bih_lena);
    DCT_lena3.get_YCbCr();
    DCT_lena3.direct_DCT();
    cout << "Lena" << endl;
    cout << endl << R3[0] << endl;
    Quantization quantization_lena3(DCT_lena3.get_DCT(),
height_lena, width_lena, &bfh_lena, &bih_lena);
    quantization_lena3.quantization(R3[0]);
    Compression
compression_lena3(quantization_lena3.get_quantization(),
height_lena, width_lena, &bfh_lena, &bih_lena);
    compression_lena3.create_DC();
    compression_lena3.coding_DC();

    cout << "Y" << endl;
    compression_lena3.create_AC("Y");
    compression_lena3.coding_AC("Y");
    compression_lena3.stream_size("Y");

    cout << "Cb" << endl;
    compression_lena3.create_AC("Cb");
    compression_lena3.coding_AC("Cb");
    compression_lena3.stream_size("Cb");

    cout << "Cr" << endl;
    compression_lena3.create_AC("Cr");
    compression_lena3.coding_AC("Cr");
```

```
     compression_lena3.stream_size("Cr");

     cout << endl << R3[1] << endl;
     DCT DCT_lena4(rgb_lena, ycbcr_lena, height_lena, width_lena,
&bfh_lena, &bih_lena);
     DCT_lena4.get_YCbCr();
     DCT_lena4.direct_DCT();
     Quantization quantization_lena4(DCT_lena4.get_DCT(),
height_lena, width_lena, &bfh_lena, &bih_lena);
     quantization_lena4.quantization(R3[1]);
     Compression
compression_lena4(quantization_lena4.get_quantization(),
height_lena, width_lena, &bfh_lena, &bih_lena);
     compression_lena4.create_DC();
     compression_lena4.coding_DC();

     cout << "Y" << endl;
     compression_lena4.create_AC("Y");
     compression_lena4.coding_AC("Y");
     compression_lena4.stream_size("Y");

     cout << "Cb" << endl;
     compression_lena4.create_AC("Cb");
     compression_lena4.coding_AC("Cb");
     compression_lena4.stream_size("Cb");

     cout << "Cr" << endl;
     compression_lena4.create_AC("Cr");
     compression_lena4.coding_AC("Cr");
     compression_lena4.stream_size("Cr");

     cout << "Baboon" << endl;
     DCT DCT_baboon3(rgb_baboon, ycbcr_baboon, height_baboon,
width_baboon, &bfh_baboon, &bih_baboon);
     DCT_baboon3.get_YCbCr();
     DCT_baboon3.direct_DCT();
     cout << endl << R3[0] << endl;
     Quantization quantization_baboon3(DCT_baboon3.get_DCT(),
height_baboon, width_baboon, &bfh_baboon, &bih_baboon);
     quantization_baboon3.quantization(R3[0]);
     Compression
compression_baboon3(quantization_baboon3.get_quantization(),
height_baboon, width_baboon, &bfh_baboon, &bih_baboon);
     compression_baboon3.create_DC();
     compression_baboon3.coding_DC();

     cout << "Y" << endl;
     compression_baboon3.create_AC("Y");
     compression_baboon3.coding_AC("Y");
     compression_baboon3.stream_size("Y");

     cout << "Cb" << endl;
     compression_baboon3.create_AC("Cb");
```

```
     compression_baboon3.coding_AC("Cb");
     compression_baboon3.stream_size("Cb");

     cout << "Cr" << endl;
     compression_baboon3.create_AC("Cr");
     compression_baboon3.coding_AC("Cr");
     compression_baboon3.stream_size("Cr");

     cout << endl << R3[1] << endl;
     DCT DCT_baboon4(rgb_baboon, ycbcr_baboon, height_baboon,
width_baboon, &bfh_baboon, &bih_baboon);
     DCT_baboon4.get_YCbCr();
     DCT_baboon4.direct_DCT();
     Quantization quantization_baboon4(DCT_baboon4.get_DCT(),
height_baboon, width_baboon, &bfh_baboon, &bih_baboon);
     quantization_baboon4.quantization(R3[1]);
     Compression
compression_baboon4(quantization_baboon4.get_quantization(),
height_baboon, width_baboon, &bfh_baboon, &bih_baboon);
     compression_baboon4.create_DC();
     compression_baboon4.coding_DC();

     cout << "Y" << endl;
     compression_baboon4.create_AC("Y");
     compression_baboon4.coding_AC("Y");
     compression_baboon4.stream_size("Y");

     cout << "Cb" << endl;
     compression_baboon4.create_AC("Cb");
     compression_baboon4.coding_AC("Cb");
     compression_baboon4.stream_size("Cb");

     cout << "Cr" << endl;
     compression_baboon4.create_AC("Cr");
     compression_baboon4.coding_AC("Cr");
     compression_baboon4.stream_size("Cr");


     cout << "Original" << endl;
     DCT DCT_original3(rgb_original, ycbcr_original,
height_original, width_original, &bfh_original, &bih_original);
     DCT_original3.get_YCbCr();
     DCT_original3.direct_DCT();
     cout << endl << R3[0] << endl;
     Quantization quantization_original3(DCT_original3.get_DCT(),
height_original, width_original, &bfh_original, &bih_original);
     quantization_original3.quantization(R3[0]);
     Compression
compression_original3(quantization_original3.get_quantization(),
height_original, width_original, &bfh_original, &bih_original);
     compression_original3.create_DC();
     compression_original3.coding_DC();
```

```
	cout << "Y" << endl;
	compression_original3.create_AC("Y");
	compression_original3.coding_AC("Y");
	compression_original3.stream_size("Y");

	cout << "Cb" << endl;
	compression_original3.create_AC("Cb");
	compression_original3.coding_AC("Cb");
	compression_original3.stream_size("Cb");

	cout << "Cr" << endl;
	compression_original3.create_AC("Cr");
	compression_original3.coding_AC("Cr");
	compression_original3.stream_size("Cr");

	cout << endl << R3[1] << endl;
	DCT DCT_original4(rgb_original, ycbcr_original,
height_original, width_original, &bfh_original, &bih_original);
	DCT_original4.get_YCbCr();
	DCT_original4.direct_DCT();
	Quantization quantization_original4(DCT_original4.get_DCT(),
height_original, width_original, &bfh_original, &bih_original);
	quantization_original4.quantization(R3[1]);
	Compression
compression_original4(quantization_original4.get_quantization(),
height_original, width_original, &bfh_original, &bih_original);
	compression_original4.create_DC();
	compression_original4.coding_DC();

	cout << "Y" << endl;
	compression_original4.create_AC("Y");
	compression_original4.coding_AC("Y");
	compression_original4.stream_size("Y");

	cout << "Cb" << endl;
	compression_original4.create_AC("Cb");
	compression_original4.coding_AC("Cb");
	compression_original4.stream_size("Cb");

	cout << "Cr" << endl;
	compression_original4.create_AC("Cr");
	compression_original4.coding_AC("Cr");
	compression_original4.stream_size("Cr");


	//PSNR(степени сжатия)
	cout << endl << "График PSNR(степень сжатия)" << endl;
	vector<int> R_vector = { 0,1,2,3,4,5,6,7,8,9,10 };

	cout << "Lena" << endl;
	DCT DCT_lena_PSNR(rgb_lena, ycbcr_lena, height_lena,
width_lena, &bfh_lena, &bih_lena);
```

```
	DCT_lena_PSNR.get_YCbCr();
	DCT_lena_PSNR.direct_DCT();
	Quantization quantization_lena_PSNR(DCT_lena_PSNR.get_DCT(),
height_lena, width_lena, &bfh_lena, &bih_lena);
	vector<double> compress_Y_lena;
	vector<double> compress_Cb_lena;
	vector<double> compress_Cr_lena;
	for (int i = 0; i < R_vector.size(); i++) {
		cout << endl << "R = " << i << endl;
		quantization_lena_PSNR.quantization(i);
		Compression
compression_lena_PSNR(quantization_lena_PSNR.get_quantization(),
height_lena, width_lena, &bfh_lena, &bih_lena);
		compression_lena_PSNR.create_DC();
		compression_lena_PSNR.coding_DC();
		compression_lena_PSNR.create_AC("Y");
		compression_lena_PSNR.coding_AC("Y");

	compress_Y_lena.push_back(compression_lena_PSNR.stream_size2
("Y"));
		compression_lena_PSNR.create_AC("Cb");
		compression_lena_PSNR.coding_AC("Cb");

	compress_Cb_lena.push_back(compression_lena_PSNR.stream_size
2("Cb"));
		compression_lena_PSNR.create_AC("Cr");
		compression_lena_PSNR.coding_AC("Cr");

	compress_Cr_lena.push_back(compression_lena_PSNR.stream_size
2("Cr"));
	}
	vector<double> PSNR_lena_Y;
	vector<double> PSNR_lena_Cb;
	vector<double> PSNR_lena_Cr;
	read_PNSR_form_file(&PSNR_lena_Y, "2\\LenaY.txt");
	write_PSNR_to_file(compress_Y_lena, PSNR_lena_Y,
"33//LenaY.txt");
	read_PNSR_form_file(&PSNR_lena_Cb, "2\\LenaCb.txt");
	write_PSNR_to_file(compress_Cb_lena, PSNR_lena_Cb,
"33\\LenaCb.txt");
	read_PNSR_form_file(&PSNR_lena_Cr, "2\\LenaCr.txt");
	write_PSNR_to_file(compress_Cr_lena, PSNR_lena_Cr,
"33\\LenaCr.txt");
	cout << "plot 'LenaY.txt' w l, 'LenaCb.txt' w l,
'LenaCr.txt' w l" << endl;


	cout << "Baboon" << endl;
	DCT DCT_baboon_PSNR(rgb_lena, ycbcr_lena, height_lena,
width_lena, &bfh_lena, &bih_lena);
	DCT_baboon_PSNR.get_YCbCr();
	DCT_baboon_PSNR.direct_DCT();
```

```
     Quantization quantization_baboon_PSNR(DCT_baboon.get_DCT(),
height_baboon, width_baboon, &bfh_baboon, &bih_baboon);
     vector<double> compress_Y_baboon;
     vector<double> compress_Cb_baboon;
     vector<double> compress_Cr_baboon;
     for (int i = 0; i < R_vector.size(); i++) {
          cout << endl << "R = " << i << endl;
          quantization_baboon_PSNR.quantization(i);
          Compression
compression_baboon_PSNR(quantization_baboon_PSNR.get_quantizatio
n(), height_baboon, width_baboon, &bfh_baboon, &bih_baboon);
          compression_baboon_PSNR.create_DC();
          compression_baboon_PSNR.coding_DC();
          compression_baboon_PSNR.create_AC("Y");
          compression_baboon_PSNR.coding_AC("Y");

     compress_Y_baboon.push_back(compression_baboon_PSNR.stream_s
ize2("Y"));
          compression_baboon_PSNR.create_AC("Cb");
          compression_baboon_PSNR.coding_AC("Cb");

     compress_Cb_baboon.push_back(compression_baboon_PSNR.stream_
size2("Cb"));
          compression_baboon_PSNR.create_AC("Cr");
          compression_baboon_PSNR.coding_AC("Cr");

     compress_Cr_baboon.push_back(compression_baboon_PSNR.stream_
size2("Cr"));
     }
     vector<double> PSNR_baboon_Y;
     vector<double> PSNR_baboon_Cb;
     vector<double> PSNR_baboon_Cr;
     read_PNSR_form_file(&PSNR_baboon_Y, "2\\BaboonY.txt");
     write_PSNR_to_file(compress_Y_baboon, PSNR_baboon_Y,
"33//BaboonY.txt");
     read_PNSR_form_file(&PSNR_baboon_Cb, "2\\BaboonCb.txt");
     write_PSNR_to_file(compress_Cb_baboon, PSNR_baboon_Cb,
"33\\BaboonCb.txt");
     read_PNSR_form_file(&PSNR_baboon_Cr, "2\\BaboonCr.txt");
     write_PSNR_to_file(compress_Cr_baboon, PSNR_baboon_Cr,
"33\\BaboonCr.txt");
     cout << "plot 'BaboonY.txt' w l, 'BaboonCb.txt' w l,
'BaboonCr.txt' w l" << endl;


     cout << "Original" << endl;
     DCT DCT_original_PSNR(rgb_lena, ycbcr_lena, height_lena,
width_lena, &bfh_lena, &bih_lena);
     DCT_original_PSNR.get_YCbCr();
     DCT_original_PSNR.direct_DCT();
     Quantization
quantization_original_PSNR(DCT_original.get_DCT(),
height_original, width_original, &bfh_original, &bih_original);
```

```
     vector<double> compress_Y_original;
     vector<double> compress_Cb_original;
     vector<double> compress_Cr_original;
     for (int i = 0; i < R_vector.size(); i++) {
          cout << endl << "R = " << i << endl;
          quantization_original_PSNR.quantization(i);
          Compression
compression_original_PSNR(quantization_original_PSNR.get_quantiz
ation(), height_original, width_original, &bfh_baboon,
&bih_original);
          compression_original_PSNR.create_DC();
          compression_original_PSNR.coding_DC();
          compression_original_PSNR.create_AC("Y");
          compression_original_PSNR.coding_AC("Y");

     compress_Y_original.push_back(compression_original_PSNR.stre
am_size2("Y"));
          compression_original_PSNR.create_AC("Cb");
          compression_original_PSNR.coding_AC("Cb");

     compress_Cb_original.push_back(compression_original_PSNR.str
eam_size2("Cb"));
          compression_original_PSNR.create_AC("Cr");
          compression_original_PSNR.coding_AC("Cr");

     compress_Cr_original.push_back(compression_original_PSNR.str
eam_size2("Cr"));
     }
     vector<double> PSNR_original_Y;
     vector<double> PSNR_original_Cb;
     vector<double> PSNR_original_Cr;
     read_PNSR_form_file(&PSNR_original_Y, "2\\OriginalY.txt");
     write_PSNR_to_file(compress_Y_original, PSNR_original_Y,
"33//OriginalY.txt");
     read_PNSR_form_file(&PSNR_original_Cb, "2\\OriginalCb.txt");
     write_PSNR_to_file(compress_Cb_original, PSNR_original_Cb,
"33\\OriginalCb.txt");
     read_PNSR_form_file(&PSNR_original_Cr, "2\\OriginalCr.txt");
     write_PSNR_to_file(compress_Cr_original, PSNR_original_Cr,
"33\\OriginalCr.txt");
     cout << "plot 'OriginalY.txt' w l, 'OriginalCb.txt' w l,
'OriginalCr.txt' w l" << endl;

     //Dop

     cout << "Dop Lena" << endl;
     ofstream file_dop_run;
     ofstream file_dop_mg;
     file_dop_run.open("dop\\LenaRunDop.txt");
     file_dop_mg.open("dop\\LenaMgDop.txt");
     for (int i = 1; i <= 60; i+=3) {
          //cout << "R = " << i << endl;
```

```
          Quantization quantization_lena_dop(DCT_lena.get_DCT(),
height_lena, width_lena, &bfh_lena, &bih_lena);
          quantization_lena_dop.quantization(i);
          Compression
compression_lena_dop(quantization_lena_dop.get_quantization(),
height_lena, width_lena, &bfh_lena, &bih_lena);
          compression_lena_dop.create_DC();
          compression_lena_dop.coding_DC();
          compression_lena_dop.create_AC("Y");
          compression_lena_dop.coding_AC("Y");
          double Run = compression_lena_dop.stream_size3("Y", 1);
          double Mg = compression_lena_dop.stream_size3("Y", 2);
          file_dop_run << i << " " << Run << endl;
          file_dop_mg << i << " " << Mg << endl;
     }
     file_dop_run.close();
     file_dop_mg.close();


     cout << "Dop baboon" << endl;
     ofstream file_dop_run2;
     ofstream file_dop_mg2;
     file_dop_run2.open("dop\\BaboonRunDop.txt");
     file_dop_mg2.open("dop\\BaboonMgDop.txt");
     for (int i = 1; i <= 60; i +=3) {
          //cout << "R = " << i << endl;
          Quantization
quantization_baboon_dop(DCT_baboon.get_DCT(), height_baboon,
width_baboon, &bfh_baboon, &bih_baboon);
          quantization_baboon_dop.quantization(i);
          Compression
compression_baboon_dop(quantization_baboon_dop.get_quantization(
), height_baboon, width_baboon, &bfh_baboon, &bih_baboon);
          compression_baboon_dop.create_DC();
          compression_baboon_dop.coding_DC();
          compression_baboon_dop.create_AC("Y");
          compression_baboon_dop.coding_AC("Y");
          double Run = compression_baboon_dop.stream_size3("Y",
1);
          double Mg = compression_baboon_dop.stream_size3("Y",
2);
          file_dop_run2 << i << " " << Run << endl;
          file_dop_mg2 << i << " " << Mg << endl;
     }
     file_dop_run2.close();
     file_dop_mg2.close();


     cout << "Dop original" << endl;
     ofstream file_dop_run3;
     ofstream file_dop_mg3;
     file_dop_run3.open("dop\\OriginalRunDop.txt");
     file_dop_mg3.open("dop\\OriginalMgDop.txt");
```

```
     for (int i = 1; i <= 60; i += 3) {
          //cout << "R = " << i << endl;
          Quantization
quantization_original_dop(DCT_original.get_DCT(),
height_original, width_original, &bfh_original, &bih_original);
          quantization_original_dop.quantization(i);
          Compression
compression_original_dop(quantization_original_dop.get_quantizat
ion(), height_original, width_baboon, &bfh_original,
&bih_original);
          compression_original_dop.create_DC();
          compression_original_dop.coding_DC();
          compression_original_dop.create_AC("Y");
          compression_original_dop.coding_AC("Y");
          double Run = compression_original_dop.stream_size3("Y",
1);
          double Mg = compression_original_dop.stream_size3("Y",
2);
          file_dop_run3 << i << " " << Run << endl;
          file_dop_mg3 << i << " " << Mg << endl;
     }
     file_dop_run3.close();
     file_dop_mg3.close();

     return 0;
}
```

Compression.h

```
#include <iostream>
#include <cmath>
#include <vector>
#include <fstream>
#include <map>
#include "bmp.h"

using namespace std;

class Compression {
private:
     int height;
     int width;
     int N = 8;
     YCbCr** quantization_ycbcr;
     BITMAPFILEHEADER* bfh;
     BITMAPINFOHEADER* bih;
     vector<vector<int>> dc;
     vector<vector<pair<unsigned char, int>>> code_dc;
     vector<vector<int>> ac_Y;
     vector<vector<int>> ac_Cb;
     vector<vector<int>> ac_Cr;
     vector<vector<pair<unsigned char, pair<unsigned char,
int>>>> code_ac_Y;
```

```
    vector<vector<pair<unsigned char, pair<unsigned char,
int>>>> code_ac_Cb;
    vector<vector<pair<unsigned char, pair<unsigned char,
int>>>> code_ac_Cr;

public:
    Compression(YCbCr** y_q, int h, int w, BITMAPFILEHEADER* bf,
BITMAPINFOHEADER* bi) {
        quantization_ycbcr = y_q;
        height = h;
        width = w;
        bfh = bf;
        bih = bi;
    }

    void create_DC() {
        // проход по полосе
            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>(quantization_ycbcr[i * N][j *
N].Y));

    CbDC.push_back(static_cast<int>(quantization_ycbcr[i * N][j
* N].Cb));

    CrDC.push_back(static_cast<int>(quantization_ycbcr[i * N][j
* N].Cr));
                }
            }
            dc.push_back(YDC);
            dc.push_back(CbDC);
            dc.push_back(CrDC);
    }

    pair<unsigned char, int> get_BC(double diff) { // запись
битовой категории BC и амплитуды MG
        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++) { // отнесение к одной
из 16 битовых категорий
            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 coding_DC() { // разностное кодирование для DC
            size_t numOfBlocks = dc[0].size();
            double average_Y = 0;
            double average_Cb = 0;
            double 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;
            vector<pair<unsigned char, int>> vec_Cb;
            vector<pair<unsigned char, int>> vec_Cr;
            for (size_t i = 0; i < numOfBlocks; i++) {
                if (i == 0) {
                    vec_Y.push_back(get_BC(dc[0][0] -
average_Y));
                    vec_Cb.push_back(get_BC(dc[1][0] -
average_Cb));
                    vec_Cr.push_back(get_BC(dc[2][0] -
average_Cr));
                }
                else {
                    vec_Y.push_back(get_BC(dc[0][i] - dc[0][i -
1]));
                    vec_Cb.push_back(get_BC(dc[1][i] - dc[1][i -
1]));
                    vec_Cr.push_back(get_BC(dc[2][i] - dc[2][i -
1]));
                }
            }
            code_dc.push_back(vec_Y);
            code_dc.push_back(vec_Cb);
            code_dc.push_back(vec_Cr);
        }

        void get_histogram_dc(string fileName, string component) {
            ofstream file(fileName);
```

```
          int row = 0;
          if (component == "Y")
               row = 0;
          if (component == "Cb")
               row = 1;
          if (component == "Cr")
               row = 2;
          for (size_t i = 0; i < dc[row].size(); i++) {
               file << static_cast<double>(dc[row][i]) << endl;
          }

          file.close();
     }

     void get_histogram_code_dc(string fileName, string
component) {
          ofstream file(fileName);
          int row = 0;
          if (component == "Y")
               row = 0;
          if (component == "Cb")
               row = 1;
          if (component == "Cr")
               row = 2;
          for (size_t i = 0; i < code_dc[row].size(); i++) {
               file <<
static_cast<double>(code_dc[row][i].second) << endl;
          }

          file.close();
     }

     void entropy() {
          map<double, double> p_dc_Y;
          map <double, double> p_dc_Cb;
          map <double, double> p_dc_Cr;
          map <double, double> p_code_dc_Y;
          map <double, double> p_code_dc_Cb;
          map <double, double> p_code_dc_Cr;

          for (size_t i = 0; i < dc[0].size(); i++) {
               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_code_dc_Y.find(code_dc[0][i].second) !=
p_code_dc_Y.end())
                     p_code_dc_Y[code_dc[0][i].second]++;
                else
                     p_code_dc_Y.insert(pair<double,
double>(code_dc[0][i].second, 1));

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

                if (p_code_dc_Cr.find(code_dc[2][i].second) !=
p_code_dc_Cr.end())
                     p_code_dc_Cr[code_dc[2][i].second]++;
                else
                     p_code_dc_Cr.insert(pair<double,
double>(code_dc[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_code_dc_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_code_dc_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_code_dc_Cr) {
            it.second /= dc[0].size();
            if (!isinf(log2(it.second))) {
                H_cdc_Cr += it.second * log2(it.second);
            }
        }
        cout << "\nH(DC_Y): " << -H_dc_Y;
        cout << "\nH(DC_Cb): " << -H_dc_Cb;
        cout << "\nH(DC_Cr): " << -H_dc_Cr << endl;
        cout << "\nH(code_DC_Y): " << -H_cdc_Y;
        cout << "\nH(code_DC_Cb): " << -H_cdc_Cb;
        cout << "\nH(code_DC_Cr): " << -H_cdc_Cr << endl;
    }

    // Перегруппировка AC коэффициентов в соответствии с
зигзагообразной последовательностью
    void create_AC(string component) {

        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;
                            if (component == "Y")

    vec.push_back(static_cast<int>(round(quantization_ycbcr[i +
x][j + y].Y)));
                            else {
                                if (component == "Cb")

    vec.push_back(static_cast<int>(round(quantization_ycbcr[i +
x][j + y].Cb)));
                                else

    vec.push_back(static_cast<int>(round(quantization_ycbcr[i +
x][j + y].Cr)));
                            }
                            x--;
                            y++;
                        }
                    }
```

```
                    else {
                        int x = 0;
                        int y = diag;
                        while (y >= 0) {
                            if (component == "Y")

vec.push_back(static_cast<int>(round(quantization_ycbcr[i + x][j
+ y].Y)));
                            else {
                                if (component == "Cb")

    vec.push_back(static_cast<int>(round(quantization_ycbcr[i +
x][j + y].Cb)));
                                else

    vec.push_back(static_cast<int>(round(quantization_ycbcr[i +
x][j + y].Cr)));
                            }
                            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) {
                            if (component == "Y")

vec.push_back(static_cast<int>(round(quantization_ycbcr[i + x][j
+ y].Y)));
                            else {
                                if (component == "Cb")

    vec.push_back(static_cast<int>(round(quantization_ycbcr[i +
x][j + y].Cb)));
                                else

    vec.push_back(static_cast<int>(round(quantization_ycbcr[i +
x][j + y].Cr)));
                            }
                            x++;
                            y--;
                        }
                    }
                    else {
                        int x = N - 1;
                        int y = diag;
                        while (y <= N - 1) {
                            if (component == "Y")
```

```
     vec.push_back(static_cast<int>(round(quantization_ycbcr[i +
x][j + y].Y)));
                                        else {
                                             if (component == "Cb")

     vec.push_back(static_cast<int>(round(quantization_ycbcr[i +
x][j + y].Cb)));
                                             else

     vec.push_back(static_cast<int>(round(quantization_ycbcr[i +
x][j + y].Cr)));
                                        }
                                        x--;
                                        y++;
                                   }
                              }
                         }
                         if (component == "Y")
                          ac_Y.push_back(vec);
                         else {
                              if (component == "Cb")
                                   ac_Cb.push_back(vec);
                              else
                                   ac_Cr.push_back(vec);
                         }
                    }
               }
          }

     void coding_AC(string component) { // формирование RUN =
(Run, BC(Level)) и Level = Magnitude(Level)
          vector<vector<int>> AC;
          if (component == "Y")
               AC = ac_Y;
          else {
               if (component == "Cb")
                    AC = ac_Cb;
               else
                    AC = ac_Cr;
          }
          vector<vector<pair<unsigned char, pair<unsigned char,
int>>>> res;
          for (size_t i = 0; i < ac_Y.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 = get_BC(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 = get_BC(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);

               if (component == "Y")
                    code_ac_Y.push_back(vec);
               else {
                    if (component == "Cb")
                         code_ac_Cb.push_back(vec);
                    else
                         code_ac_Cr.push_back(vec);
               }

          }
     }
```

```
     void stream_size(string component) {
          vector<vector<pair<unsigned char, pair<unsigned char,
int>>>> codAC;
          int I = 0;
          if (component == "Y") {
               codAC = code_ac_Y;
               I = 0;
          }
          else
          {
               if (component == "Cb") {
                    codAC = code_ac_Cb;
                    I = 1;
               }
               else {
                    codAC = code_ac_Cr;
                    I = 2;
               }
          }
          size_t Ndc = dc[0].size();
          size_t Nrl = get_pair_num(codAC);

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

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


          double H_BC_dDC = 0;
          for (pair<double, double> it : p_BC_dDC) {
               it.second /= code_dc[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 a1 = (double)((H_rl * Nrl * 100) /
(double)((H_rl * Nrl) + sum_BC_level));
          //double a2 = (double)((sum_BC_level * 100) /
(double)((H_rl * Nrl) + sum_BC_level));

          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(code DC): " << bcdc << "%" << endl;
          cout << "MG(code DC): " << magnitude << "%" << endl;
          cout << "(Run, BC(Level)): " << runBClevel << "%" <<
endl;
          cout << "MG(Level): " << magnitudeLevel << "%" << endl;
     }
     double stream_size2(string component) {
          vector<vector<pair<unsigned char, pair<unsigned char,
int>>>> codAC;
          int I = 0;
          if (component == "Y") {
               codAC = code_ac_Y;
               I = 0;
          }
          else
```

```
          {
               if (component == "Cb") {
                    codAC = code_ac_Cb;
                    I = 1;
               }
               else {
                    codAC = code_ac_Cr;
                    I = 2;
               }
          }
          size_t Ndc = dc[0].size();
          size_t Nrl = get_pair_num(codAC);

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

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


          double H_BC_dDC = 0;
          for (pair<double, double> it : p_BC_dDC) {
               it.second /= code_dc[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;
          return d;
     }

     double stream_size3(string component, int flag) {
          vector<vector<pair<unsigned char, pair<unsigned char,
int>>>> codAC;
          int I = 0;
          if (component == "Y") {
               codAC = code_ac_Y;
               I = 0;
          }
          else
          {
               if (component == "Cb") {
                    codAC = code_ac_Cb;
                    I = 1;
               }
               else {
                    codAC = code_ac_Cr;
                    I = 2;
               }
          }
          size_t Ndc = dc[0].size();
          size_t Nrl = get_pair_num(codAC);

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

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

```
          double H_BC_dDC = 0;
          for (pair<double, double> it : p_BC_dDC) {
               it.second /= code_dc[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 a1 = (double)((H_rl * Nrl * 100) /
(double)((H_rl * Nrl) + sum_BC_level));
          double a2 = (double)((sum_BC_level * 100) /
(double)((H_rl * Nrl) + sum_BC_level));

          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: " << a1 << "%" << endl;
          cout << "Magnitude: " << a2 << "%" << endl;*/
```

```
          if (flag == 1)
               return a1;
          else if (flag == 2)
               return a2;
     }
     int get_pair_num(vector<vector<pair<unsigned char,
pair<unsigned char, int>>>> codAC) {
          int res = 0;
          for (size_t i = 0; i < codAC.size(); i++) {
               res += codAC[i].size();
          }
          return res;
     }


};
```

DCT.h

```
#include <iostream>
#include <cmath>
#include <fstream>
#include "bmp.h"

const double Pi = 3.141592653589793;

class DCT {
private:
     int height;
     int width;
     int N = 8;
     YCbCr** ycbcr;
     YCbCr** new_ycbcr;
     YCbCr** reverse_ycbcr;
     RGB** rgb;
     RGB** new_rgb;
     BITMAPFILEHEADER* bfh;
     BITMAPINFOHEADER* bih;
     vector<double> Y_PSNR;
     vector<double> Cb_PSNR;
     vector<double> Cr_PSNR;
public:
     DCT(RGB** color, YCbCr** y, int h, int w, BITMAPFILEHEADER*
bf, BITMAPINFOHEADER* bi) {
          rgb = color;
          ycbcr = y;
          height = h;
          width = w;
          bfh = bf;
          bih = bi;

          new_ycbcr = new YCbCr*[height];
          for (int i = 0; i < height; i++)
               new_ycbcr[i] = new YCbCr[width];
```

```
        reverse_ycbcr = new YCbCr * [height];
        for (int i = 0; i < height; i++)
            reverse_ycbcr[i] = new YCbCr[width];

        new_rgb = new RGB * [height];
        for (int i = 0; i < height; i++)
            new_rgb[i] = new RGB[width];
    }
    ~DCT() {
        for (int i = 0; i < height; i++) {
            delete(new_ycbcr[i]);
            delete(new_rgb[i]);
            delete(reverse_ycbcr[i]);
        }
        delete(new_ycbcr);
        delete(new_rgb);
        delete(reverse_ycbcr);
    }
    void get_image(const char* filename) {
        get_RGB();
        FILE* file;
        file = fopen(filename, "wb");
        write_bmp(file, new_rgb, bfh, bih, height, width);
        fclose(file);
    }

    double get_cos(int i, int k) {
        return cos(((2.0 * i + 1) * Pi * k) / (2 * (double)N));
    }

    void direct_DCT() {
        double Ck = 0;
        double Cl = 0;
        for (int h = 0; h < height; h += N) {
            for (int w = 0; w < width; w += N) {
                for (int k = 0; k < N; k++) {
                    for (int l = 0; l < N; l++) {
                        if (k == 0)
                            Ck = (double)1.0 / N;
                        else
                            Ck = (double)2.0 / N;
                        if (l == 0)
                            Cl = (double)1.0 / N;
                        else
                            Cl = (double)2.0 / N;
                        double sumY = 0;
                        double sumCb = 0;
                        double sumCr = 0;
                        for (int i = 0; i < N; i++) {
                            for (int j = 0; j < N; j++) {
                                sumY += ycbcr[i+h][j+w].Y
* get_cos(i, k) * get_cos(j, l);
```

```
                                        sumCb +=
ycbcr[i+h][j+w].Cb * get_cos(i, k) * get_cos(j, l);
                                        sumCr +=
ycbcr[i+h][j+w].Cr * get_cos(i, k) * get_cos(j, l);
                                    }
                                }
                                new_ycbcr[h + k][w + l].Y =
round(sqrt(Ck) * sqrt(Cl) * sumY);
                                new_ycbcr[h + k][w + l].Cb =
round(sqrt(Ck) * sqrt(Cl) * sumCb);
                                new_ycbcr[h + k][w + l].Cr =
round(sqrt(Ck) * sqrt(Cl) * sumCr);
                            }
                        }
                    }
                }
            }

    YCbCr** get_DCT() {
        return new_ycbcr;
    }

    void set_new_YCbCr(YCbCr** n) {
        new_ycbcr = n;
    }

    void set_reverse_YCbCr(YCbCr** n) {
        reverse_ycbcr = n;
    }

    void IDCT() {
        double Ck = 0;
        double Cl = 0;
        for (int h = 0; h < height; h += N) {
            for (int w = 0; w < width; w += N) {
                for (int i = 0; i < N; i++) {
                    for (int j = 0; j < N; j++) {
                        double sumY = 0;
                        double sumCb = 0;
                        double sumCr = 0;
                        for (int k = 0; k < N; k++) {
                            for (int l = 0; l < N; l++) {
                                if (k == 0)
                                    Ck = (double)1 / N;
                                else
                                    Ck = (double)2 / N;
                                if (l == 0)
                                    Cl = (double)1 / N;
                                else
                                    Cl = (double)2 / N;
                                sumY += sqrt(Ck) *
sqrt(Cl) * new_ycbcr[k + h][l + w].Y * get_cos(i,k) *
get_cos(j,l);
```

```
                                        sumCb += sqrt(Ck) *
sqrt(Cl) * new_ycbcr[k + h][l + w].Cb * get_cos(i,k) *
get_cos(j,l);
                                        sumCr += sqrt(Ck) *
sqrt(Cl) * new_ycbcr[k + h][l + w].Cr * get_cos(i,k) *
get_cos(j,l);
                                    }
                                }
                                reverse_ycbcr[h+i][w+j].Y =
round(sumY);
                                reverse_ycbcr[h+i][w+j].Cb =
round(sumCb);
                                reverse_ycbcr[h+i][w+j].Cr =
round(sumCr);
                            }
                        }
                    }
                }
            }

        void get_PSNR(bool isWriten) {
            PSNR("Y", isWriten);
            PSNR("Cb", isWriten);
            PSNR("Cr", isWriten);
        }

        void get_PSNR_graphic(int R, bool isWriten) {
            Y_PSNR.push_back(PSNR("Y", isWriten));
            Cb_PSNR.push_back(PSNR("Cb", isWriten));
            Cr_PSNR.push_back(PSNR("Cr", isWriten));
        }

        void write_PSNR_file(string image_name, vector<int> R) {
            ofstream Y_file;
            Y_file.open(image_name + "Y.txt");
            ofstream Cb_file;
            Cb_file.open(image_name + "Cb.txt");
            ofstream Cr_file;
            Cr_file.open(image_name + "Cr.txt");
            for (int i = 0; i < R.size(); i++) {
                Y_file << R[i] << " " << Y_PSNR[i] << endl;
                Cb_file << R[i] << " " << Cb_PSNR[i] << endl;
                Cr_file << R[i] << " " << Cr_PSNR[i] << endl;
            }
            Y_file.close();
            Cb_file.close();
            Cr_file.close();
        }

        double PSNR(string component, bool isWriten) {
            double tmp = width * height * pow(256 - 1, 2);
            double PSNR = 0;
            for (int i = 0; i < height; i++) {
```

```
                for (int j = 0; j < width; j++) {
                    if (component == "Y") {
                        PSNR += pow((ycbcr[i][j].Y -
reverse_ycbcr[i][j].Y), 2);
                        continue;
                    }
                    if (component == "Cb") {
                        PSNR += pow((ycbcr[i][j].Cb -
reverse_ycbcr[i][j].Cb), 2);
                        continue;
                    }
                    if (component == "Cr") {
                        PSNR += pow((ycbcr[i][j].Cr -
reverse_ycbcr[i][j].Cr), 2);
                        continue;
                    }
                }
            }
            PSNR = 10 * log10(tmp / PSNR);
            if (isWriten)
             cout << "PSNR " << component << " = " << PSNR << endl;
            return PSNR;
        }
        void get_RGB() {
            for (int i = 0; i < height; i++) {
                double tmp = 0;
                for (int j = 0; j < width; j++) {
                    tmp = (reverse_ycbcr[i][j].Y - 0.714 *
(reverse_ycbcr[i][j].Cr - 128) - 0.334 * (reverse_ycbcr[i][j].Cb
- 128));
                    new_rgb[i][j].G = clipping(tmp);
                    tmp = reverse_ycbcr[i][j].Y + 1.402 *
(reverse_ycbcr[i][j].Cr - 128);
                    new_rgb[i][j].R = clipping(tmp);
                    tmp = reverse_ycbcr[i][j].Y + 1.772 *
(reverse_ycbcr[i][j].Cb - 128);
                    new_rgb[i][j].B = clipping(tmp);
                }
            }
        }


        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));
```

```
    }

    void get_YCbCr() {
        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
                ycbcr[i][j].Y = clipping(((double)rgb[i][j].R
* 0.299 + (double)rgb[i][j].G * 0.587 + (double)rgb[i][j].B *
0.114));
                ycbcr[i][j].Cb = clipping(0.5643 *
((double)rgb[i][j].B - ycbcr[i][j].Y) + 128);
                ycbcr[i][j].Cr = clipping(0.7132 *
((double)rgb[i][j].R - ycbcr[i][j].Y) + 128);
            }
        }
    }

};
```

Quantization.h

```
#include <iostream>
#include <cmath>
#include "bmp.h"

using namespace std;

class Quantization {
private:
    int height;
    int width;
    int N = 8;
    YCbCr** ycbcr_DCT;
    YCbCr** quantization_ycbcr;
    YCbCr** dequantization_ycbcr;
    BITMAPFILEHEADER* bfh;
    BITMAPINFOHEADER* bih;

public:
    Quantization(YCbCr** y, int h, int w, BITMAPFILEHEADER* bf,
BITMAPINFOHEADER* bi) {
        ycbcr_DCT = y;
        height = h;
        width = w;
        bfh = bf;
        bih = bi;

        quantization_ycbcr = new YCbCr * [height];
        for (int i = 0; i < height; i++)
            quantization_ycbcr[i] = new YCbCr[width];

        dequantization_ycbcr = new YCbCr * [height];
        for (int i = 0; i < height; i++)
            dequantization_ycbcr[i] = new YCbCr[width];
```

```
    }
    void quantization(double R) {
        const int N = 8;
        double Q[N][N];
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                Q[i][j] = 1 + ((i + j) * R);
            }
        }

        for (int h = 0; h < height; h += N) {
            for (int w = 0; w < width; w += N) {
                for (int i = 0; i < N; i++) {
                    for (int j = 0; j < N; j++) {
                        quantization_ycbcr[i+h][j+w].Y =
round((double)ycbcr_DCT[i+h][j+w].Y / Q[i][j]);
                        quantization_ycbcr[i+h][j+w].Cb =
round((double)ycbcr_DCT[i+h][j+w].Cb / Q[i][j]);
                        quantization_ycbcr[i+h][j+w].Cr =
round((double)ycbcr_DCT[i+h][j+w].Cr / Q[i][j]);
                    }
                }
            }
        }
    }
    YCbCr** get_quantization() {
        return quantization_ycbcr;
    }
    void dequantization(double R) {
        const int N = 8;
        double Q[N][N];
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                Q[i][j] = 1 + ((i + j) * R);
            }
        }

        for (int h = 0; h < height; h += N) {
            for (int w = 0; w < width; w += N) {
                for (int i = 0; i < N; i++) {
                    for (int j = 0; j < N; j++) {
                        dequantization_ycbcr[i + h][j +
w].Y = (double)quantization_ycbcr[i + h][j + w].Y * Q[i][j];
                        dequantization_ycbcr[i + h][j +
w].Cb = (double)quantization_ycbcr[i + h][j + w].Cb * Q[i][j];
                        dequantization_ycbcr[i + h][j +
w].Cr = (double)quantization_ycbcr[i + h][j + w].Cr * Q[i][j];
                    }
                }
            }
        }
    }
```

```
     YCbCr** get_dequantization() {
          return dequantization_ycbcr;
     }
};
```

bmp.h

```
#ifndef bmp
#define bmp
#include <iostream>
using namespace std;

struct BITMAPFILEHEADER {
     short bfType;
     int bfSize;
     short bfReserved1;
     short bfOffBits;;
     int bfReserved2;
};

struct BITMAPINFOHEADER {
     int biSize;
     int biWidth;
     int biHeight;
     short int biPlanes;
     short int biBitCount;
     int biCompression;
     int biSizeImage;
     int biXPelsPerMeter;
     int biYPelsPerMeter;
     int biClrUsed;
     int biClrImportant;
};

struct RGB {
     unsigned char B;
     unsigned char G;
     unsigned char R;
};

struct YCbCr {
     double Cr;
     double Cb;
     double Y;
};



RGB** read_bmp(FILE* f, BITMAPFILEHEADER* bfh, BITMAPINFOHEADER*
bih)
{
     int k = 0;
```

```
     k = fread(bfh, sizeof(*bfh) - 2, 1, f);
     if (k == 0)
     {
          cout << "Error";
          return 0;
     }

     k = fread(bih, sizeof(*bih), 1, f);
     if (k == NULL)
     {
          cout << "Error";
          return 0;
     }
     int a = abs(bih->biHeight);
     int b = abs(bih->biWidth);
     RGB** rgb = new RGB * [a];
     for (int i = 0; i < a; i++)
     {
          rgb[i] = new RGB[b];
     }
     int pad = 4 - (b * 3) % 4;
     for (int i = 0; i < a; i++)
     {
          fread(rgb[i], sizeof(RGB), b, f);
          if (pad != 4)
          {
               fseek(f, pad, SEEK_CUR);
          }
     }
     return rgb;
}

void write_bmp(FILE* f, RGB** rgb, BITMAPFILEHEADER* bfh,
BITMAPINFOHEADER* 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((rgb[i]), sizeof(RGB), width, f);
          if (pad != 4)
          {
               fwrite(&buf, 1, pad, f);
          }
     }
}

#endif bmp
```