1. Цель работы:

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

2.1. Дискретное косинусное преобразование (ДКП):

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

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

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

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

Значения матрицы T вычисляются следующим образом:

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

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

$$Y = (T * X) * T^T$$
$$X = (T^T * Y) * T,$$

где X и Y являются матрицами размерности NxN.

2.1.1. Реализация процедуры прямого и обратного ДКП для блоков NxN:

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:

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

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

где i,j,k,l = 0,…,N-1.

2.1.2. Оценка искажений, вносимых ДКП:

Рис. 1. Исходное изображение “kodim23.bmp”.

Рис. 2. Изображение “kodim23.bmp” после обратного ДКП.

```
Y PSNR: 58.8918
Cb PSNR: 59.2595
Cr PSNR: 59.3181
```

Рис. 3. Значение PSNR для изображения “kodim23.bmp” после обратного ДКП.

Рис. 4. Исходное изображение “lena.bmp”.

Рис. 5. Изображение “lena.bmp” после обратного ДКП.

```
Y PSNR: 58.8989
Cb PSNR: 58.9114
Cr PSNR: 58.8719
```

Рис. 6. Значение PSNR для изображения “lena.bmp” после обратного ДКП.

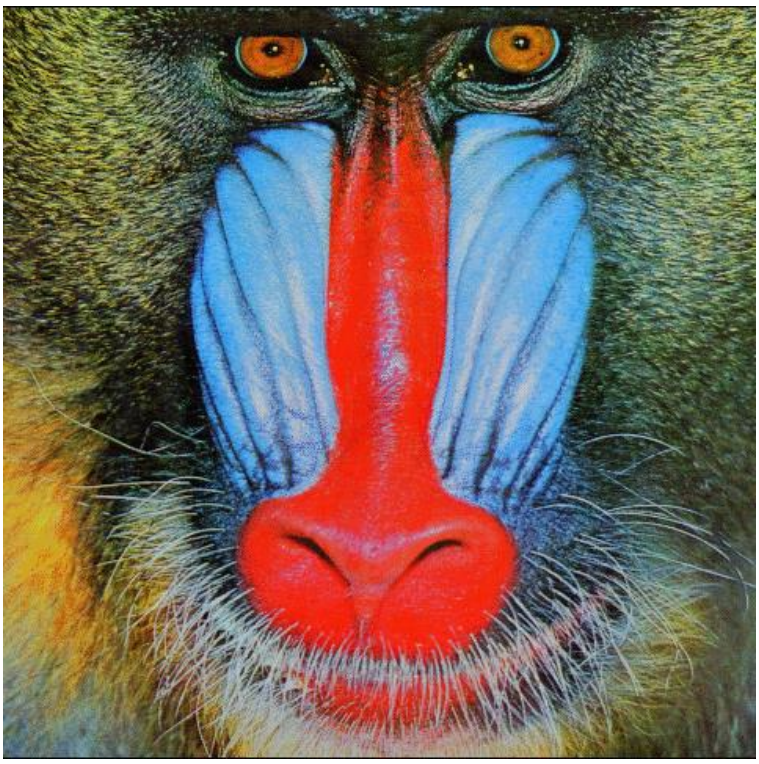

Рис. 7. Исходное изображение “baboon.bmp”.

Рис. 8. Изображение “baboon.bmp” после обратного ДКП.

```
Y PSNR: 58.9312
Cb PSNR: 58.9111
Cr PSNR: 58.9088
```

Рис. 9. Значение PSNR для изображения “baboon.bmp” после обратного ДКП.

По полученным картинкам, видимых искажений после прямого и обратного ДКП не было обнаружено. Это подтверждают высокие значения PSNR. Небольшие потери могли возникнуть при округлении до целых чисел.

2.2. 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 8x8.

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.

2.2.1. Реализация процедуры квантования и деквантования:

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

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

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 значение каждого кванта $Y_{i,j}^{dq}$ будут вычисляться при декодировании как произведение номера кванта и шага квантования:

$$Y_{i,j}^{q} = Y_{i,j}^{dq} * q_{i,j}^{(c)},$$

где i, j = 0,…,7.

Матрицы квантования будут строиться по формуле:

$$q_{i,j}^{Y}(R) = 1 + (i + j) * R,$$

где R – целочисленный параметр, управляющий качеством обработки. Квантование для разных компонент происходило с помощью одинаковых матриц при значении R от 1 до 10.

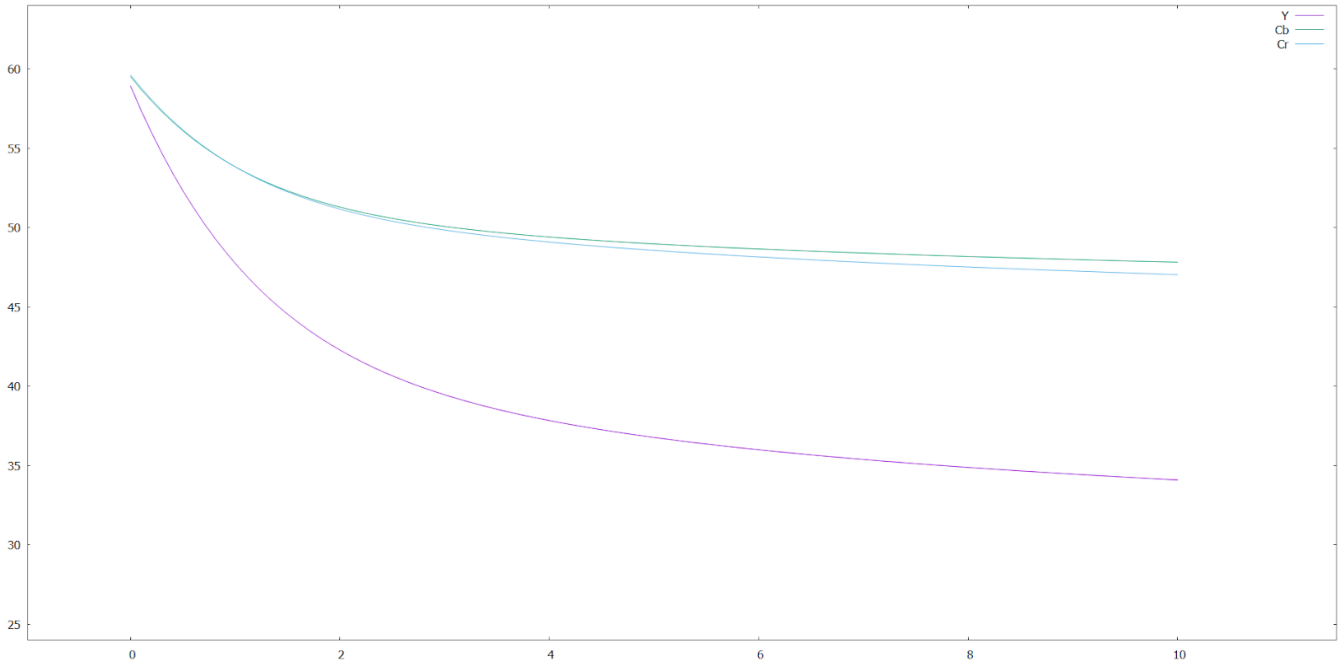

Рис. 10. График PSNR(R) для изображения "kodim23.bmp".

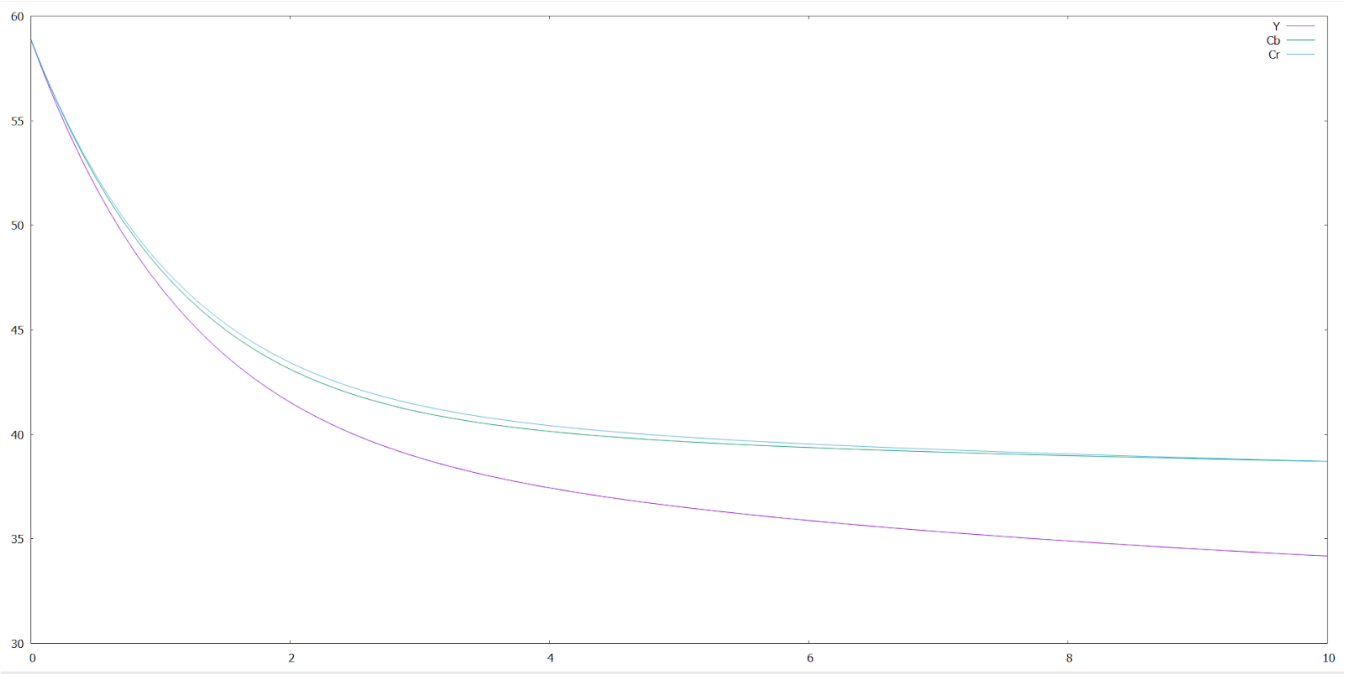

Рис. 11. График PSNR(R) для изображения “lena.bmp”.

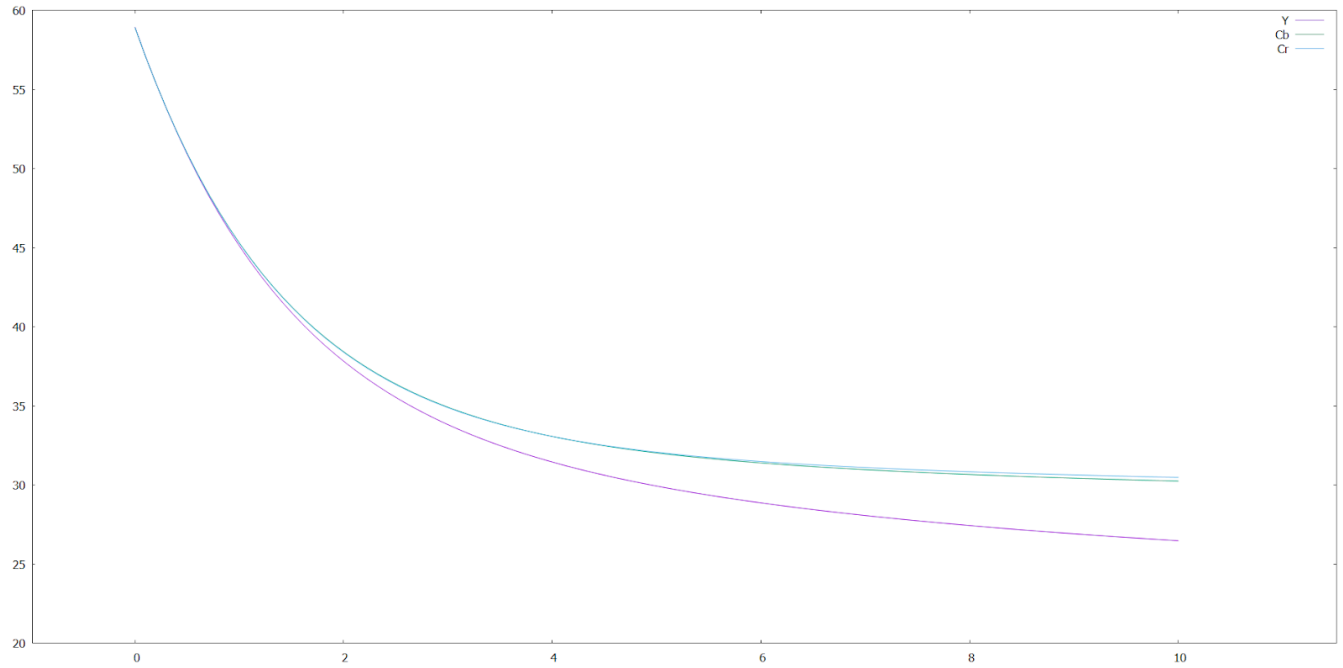

Рис. 12. График PSNR(R) для изображения “baboon.bmp”.

Рис. 13. Изображение “kodim23.bmp” при R=1.

Рис. 14. Изображение “kodim23.bmp” при R=5.

Рис. 15. Изображение "kodim23.bmp" при R=10.

Рис. 16. Изображение "lena.bmp" при R=1.

Рис. 17. Изображение “lena.bmp” при R=5.

Рис. 18. Изображение “lena.bmp” при R=10.

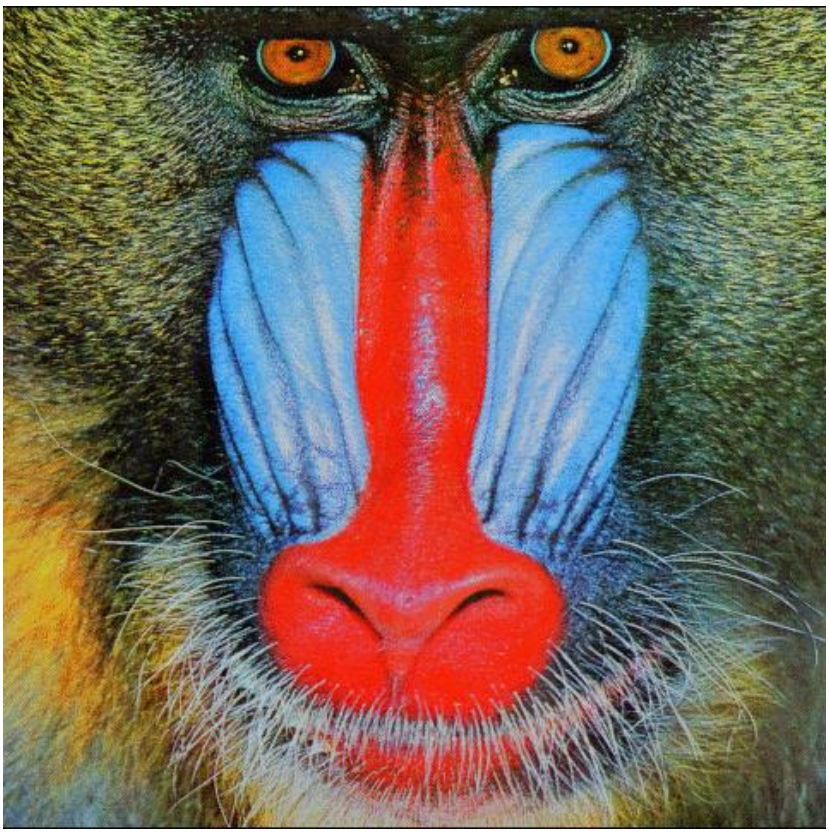

Рис. 19. Изображение “baboon.bmp” при R=1.

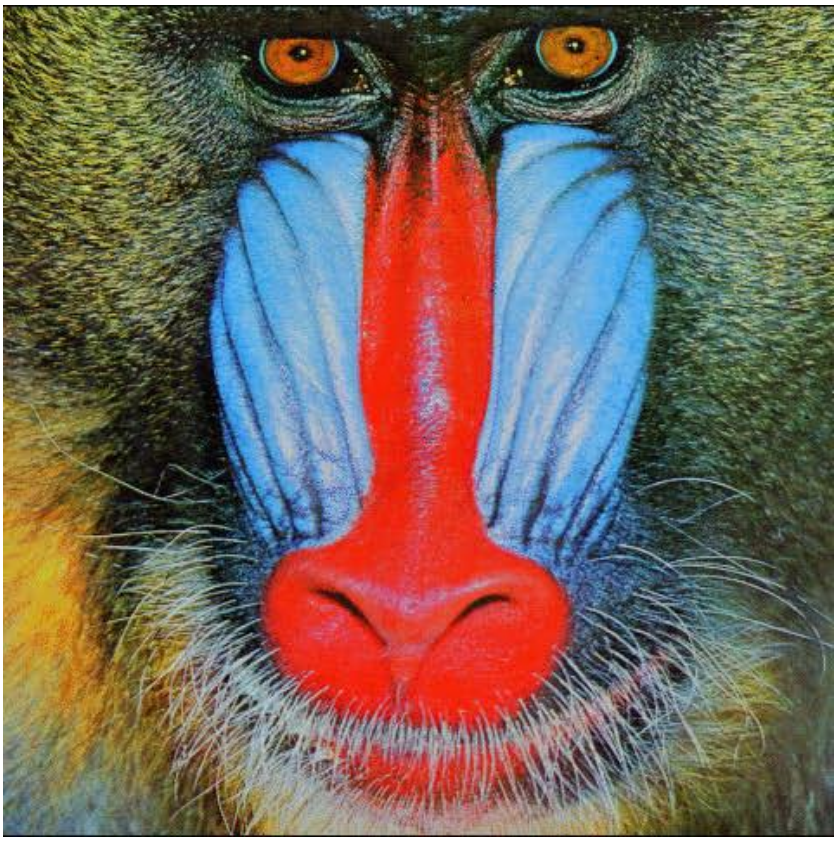

Рис. 20. Изображение “baboon.bmp” при R=5.

Рис. 21. Изображение “baboon.bmp” при R=10.

2.2.2. Оценка влияния искажений, вносимых квантованием, на компоненты этих изображений:

По графикам видно, что значение PSNR яркостной компоненты уменьшается быстрее, чем компонент Cb и Cr. Это связано с тем, что Y содержит основную информацию об изображении.

Искажения, вносимые квантованием, можно разглядеть при приближении картинок: все изображение состоит из квадратов, и из-за этого контуры изображения становятся нечеткими.

Рис. 22. Сравнение изображения “kodim23.bmp” с квантованным при R=10.

Рис. 23. Сравнение изображения "lena.bmp" с квантованным при R=10.

Рис. 24. Сравнение изображения "baboon.bmp" с квантованным при R=10.

2.3.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 последовательности из 63 AC^q коэффициентов.
4) Кодирование пар (Run, Level).

2.3.1. Реализация процедуры 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 блока.

Значение ΔDC представляется в форме битовой категории BC и амплитуды MG, где битовая категория от значения x вычисляется как BC(x)=$\lceil \log|x| + 1 \rceil$, а значение амплитуды – само кодируемое значение.

2.3.2. Оценка эффективности использования разностного кодирования для коэффициентов постоянного тока:

Необходимо построить гистограммы частот f(DC^q) и f(ΔDC), полученных для блоков 8x8 яркостной составляющей тестового изображения и вычислить оценки энтропии.

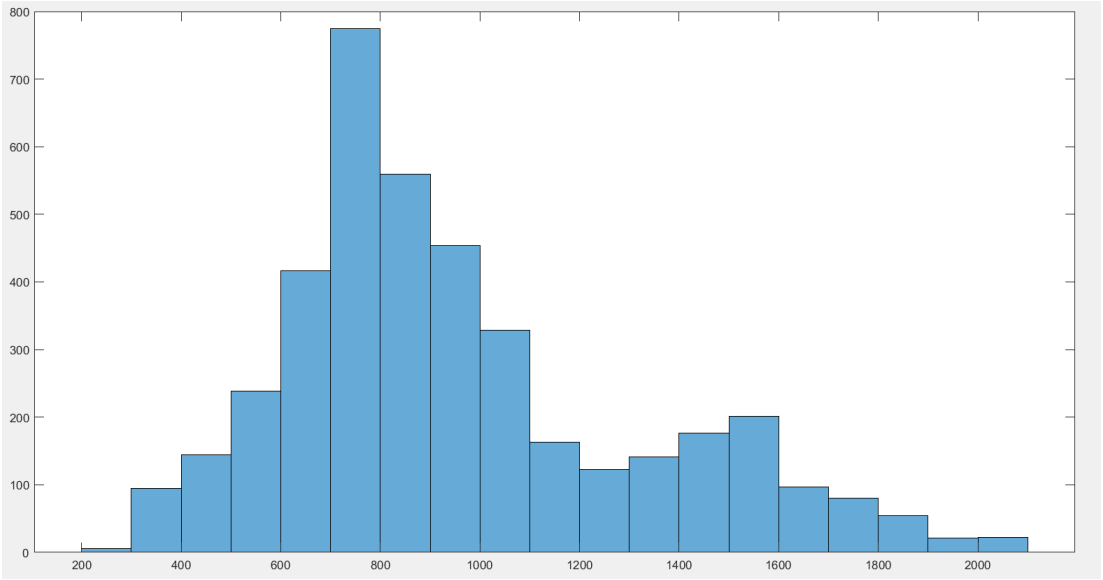

Рис. 25. Гистограмма частот f(DC^q) по Y для изображения "kodim23.bmp".

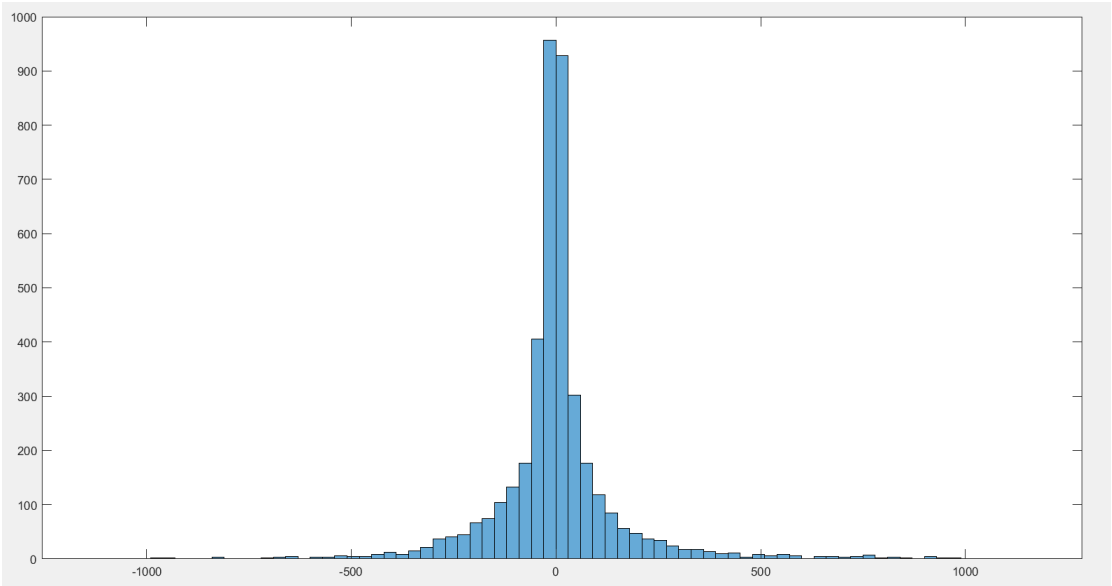

Рис. 26. Гистограмма частот f(ΔDC) по Y для изображения "kodim23.bmp".

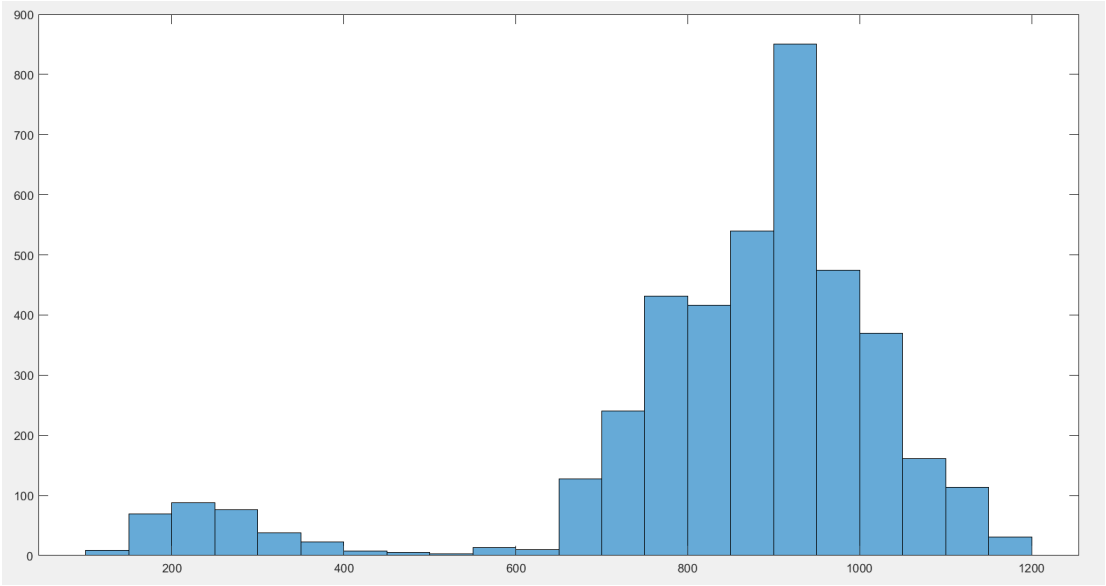

Рис. 27. Гистограмма частот f(DC^q) по Cb для изображения "kodim23.bmp".

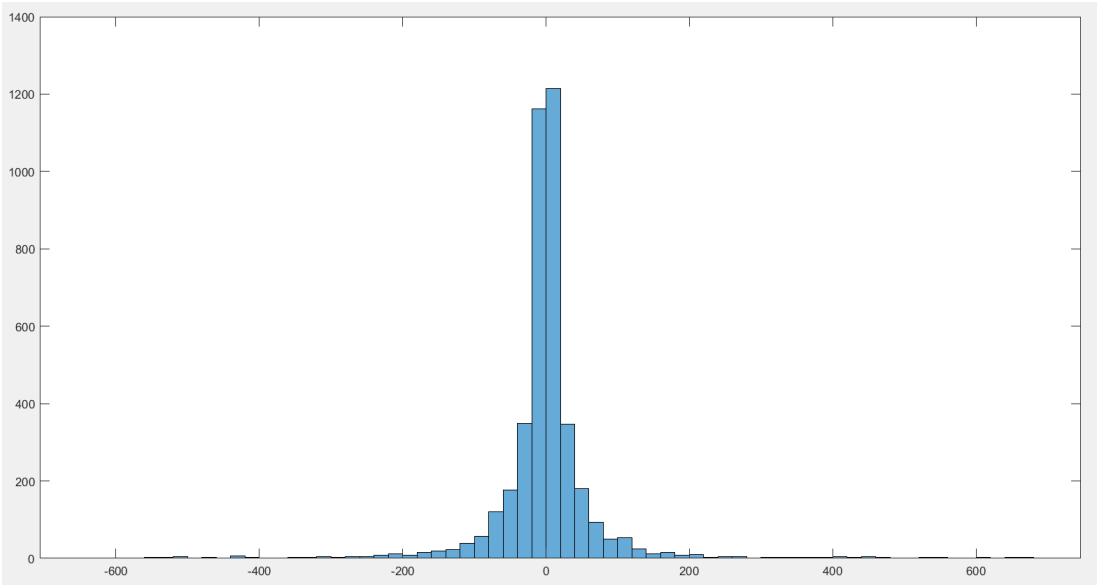

Рис. 28. Гистограмма частот f(ΔDC) по Cb для изображения "kodim23.bmp".

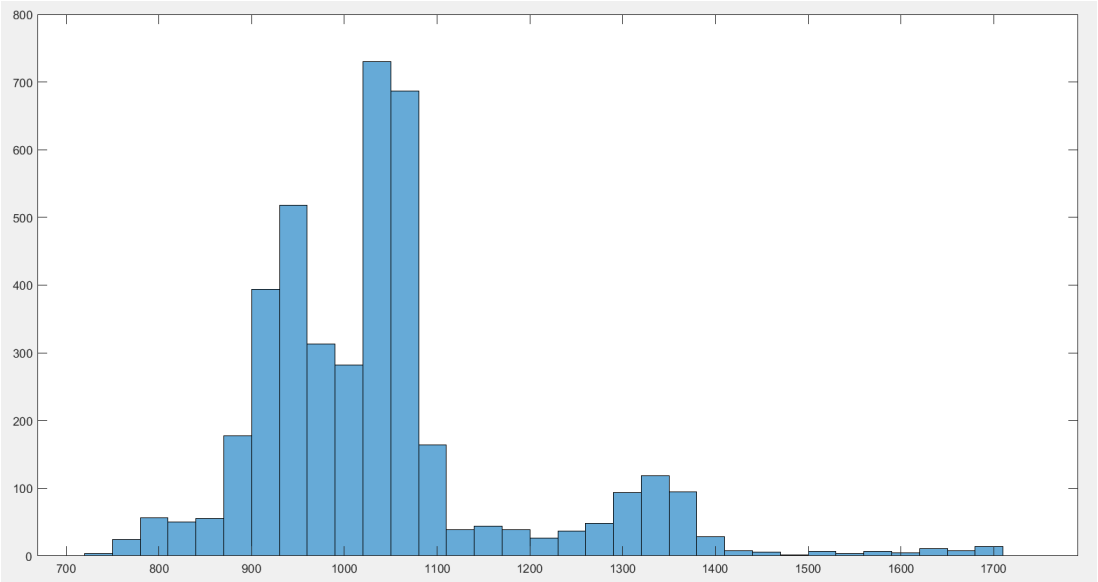

Рис. 29. Гистограмма частот f(DC^q) по Cr для изображения "kodim23.bmp".

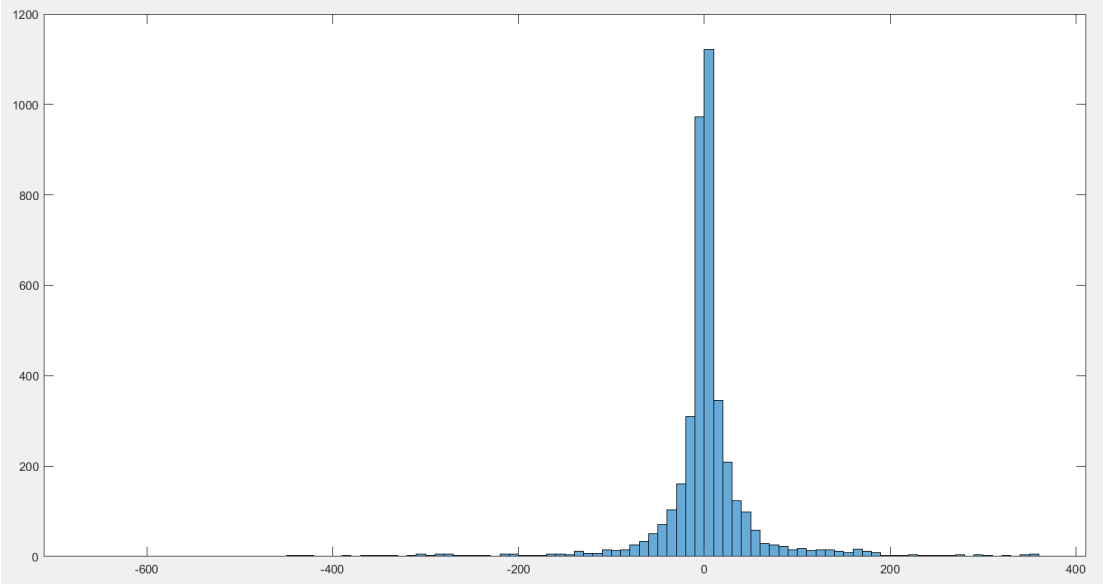

Рис. 30. Гистограмма частот f(ΔDC) по Cr для изображения "kodim23.bmp".

```
Entropy of DC^q
Entropy of Y: 9.95122
Entropy of Cb: 8.91747
Entropy of Cr: 8.47316
Entropy of delta DC
Entropy of Y: 8.44083
Entropy of Cb: 7.21766
Entropy of Cr: 6.76909
```

Рис. 31. Значения энтропии для изображения "kodim23.bmp".

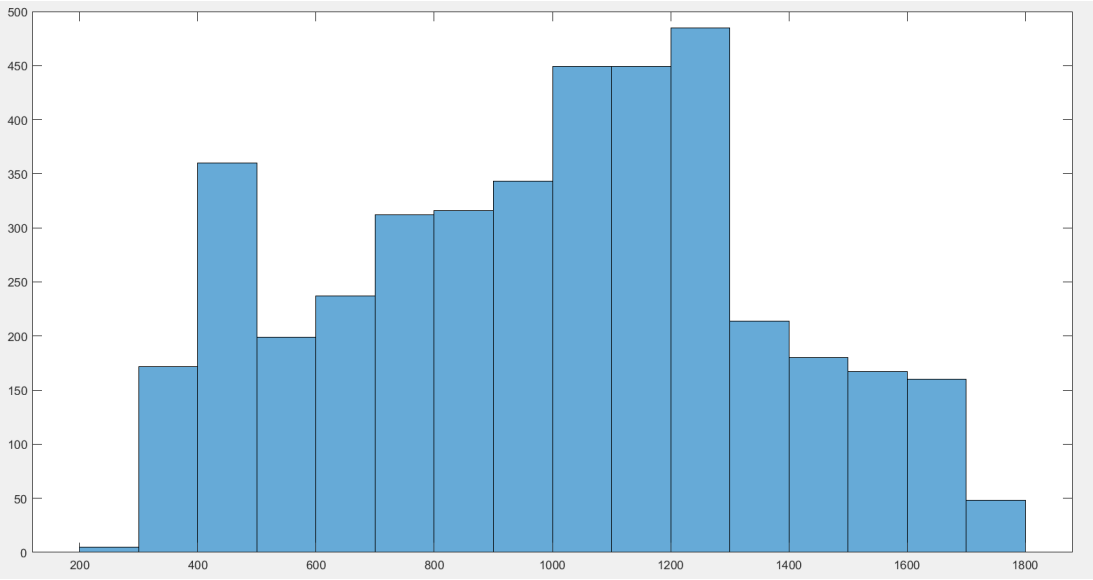

Рис. 32. Гистограмма частот f(DC^q) по Y для изображения "lena.bmp".

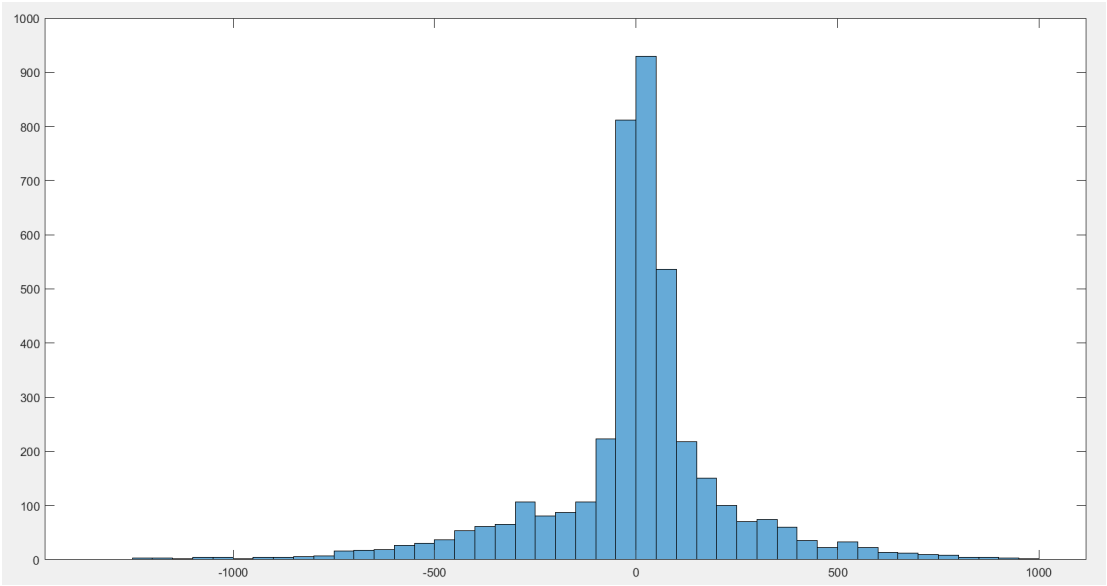

Рис. 33. Гистограмма частот f(ΔDC) по Y для изображения "lena.bmp".

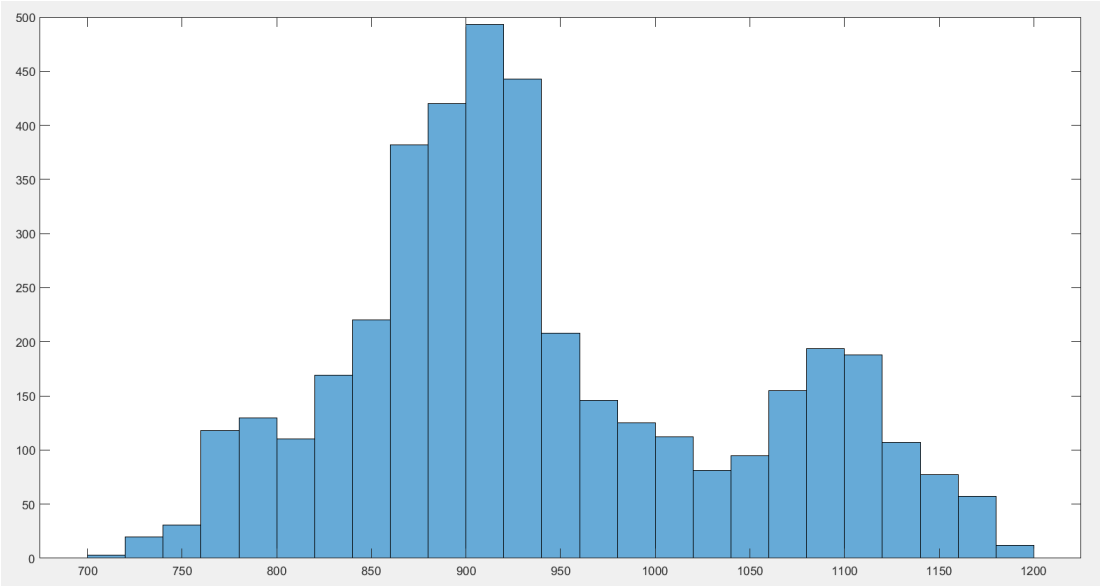

Рис. 34. Гистограмма частот f(DC^q) по Cb для изображения "lena.bmp".

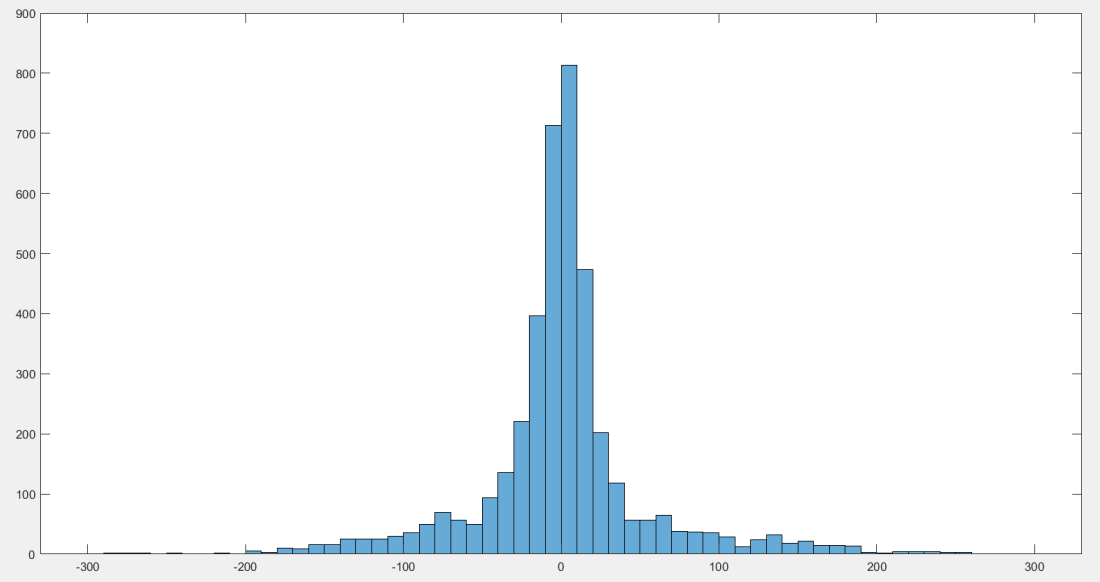

Рис. 35. Гистограмма частот f(ΔDC) по Cb для изображения "lena.bmp".

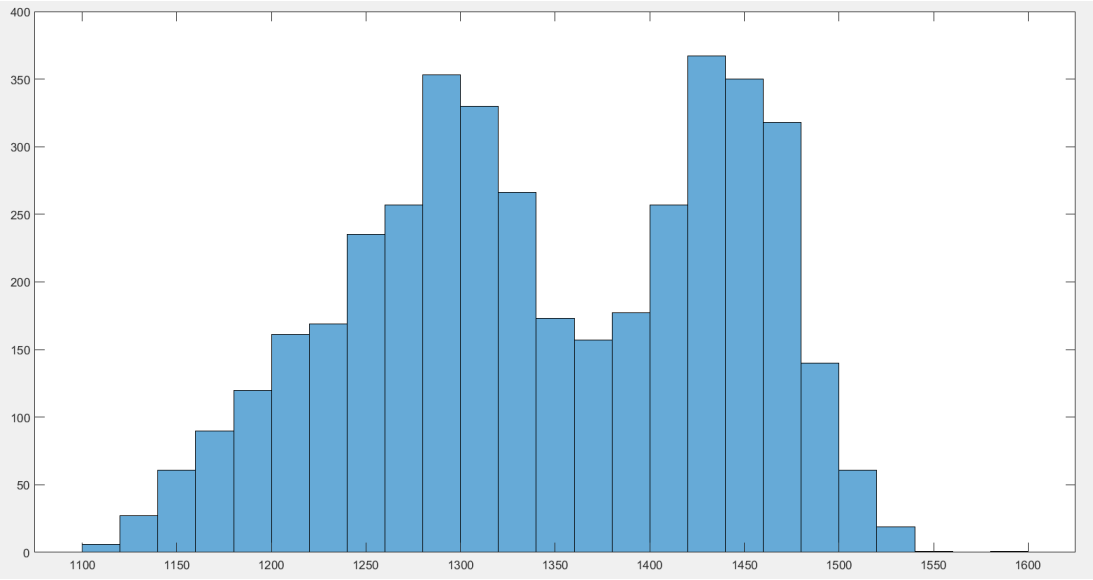

Рис. 36. Гистограмма частот f(DC^q) по Cr для изображения "lena.bmp".

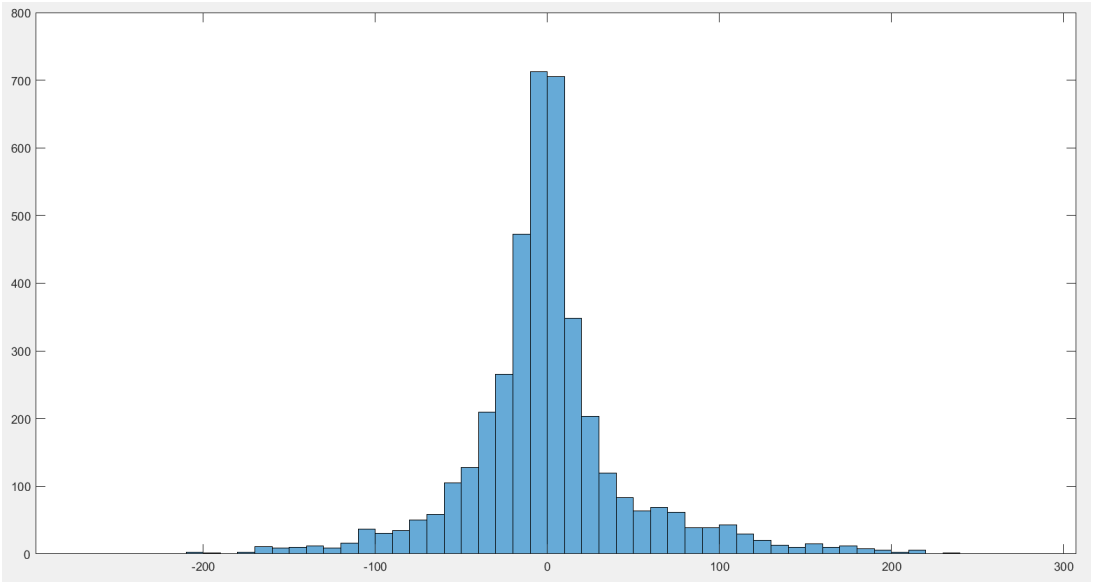

Рис. 37. Гистограмма частот f(ΔDC) по Cr для изображения "lena.bmp".

```
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
```

Рис. 38. Значения энтропии для изображения "lena.bmp".

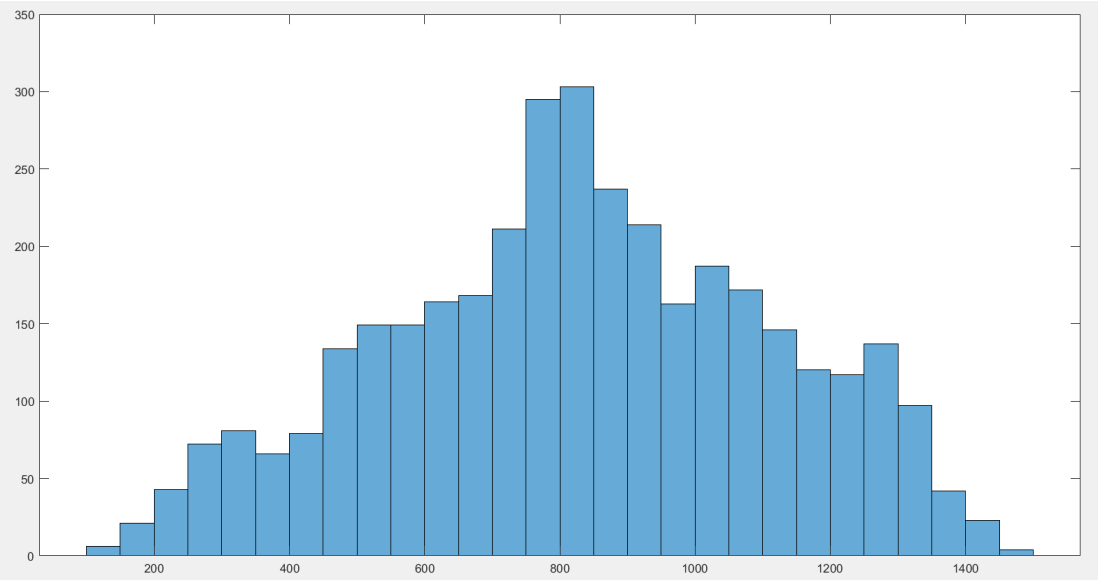

Рис. 39. Гистограмма частот f(DC^q) по Y для изображения "baboon.bmp".

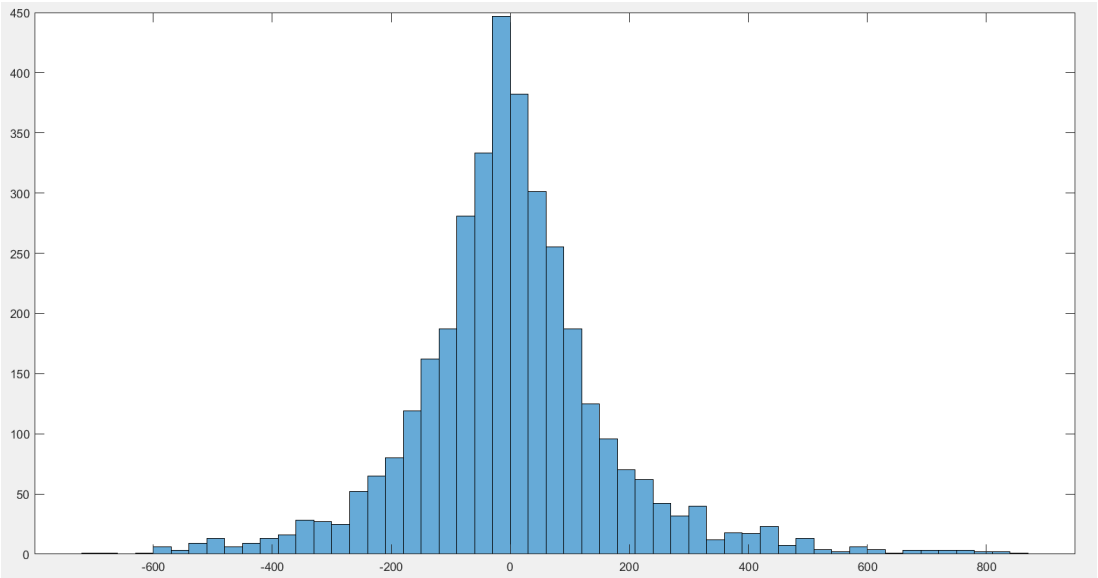

Рис. 40. Гистограмма частот f(ΔDC) по Y для изображения "baboon.bmp".

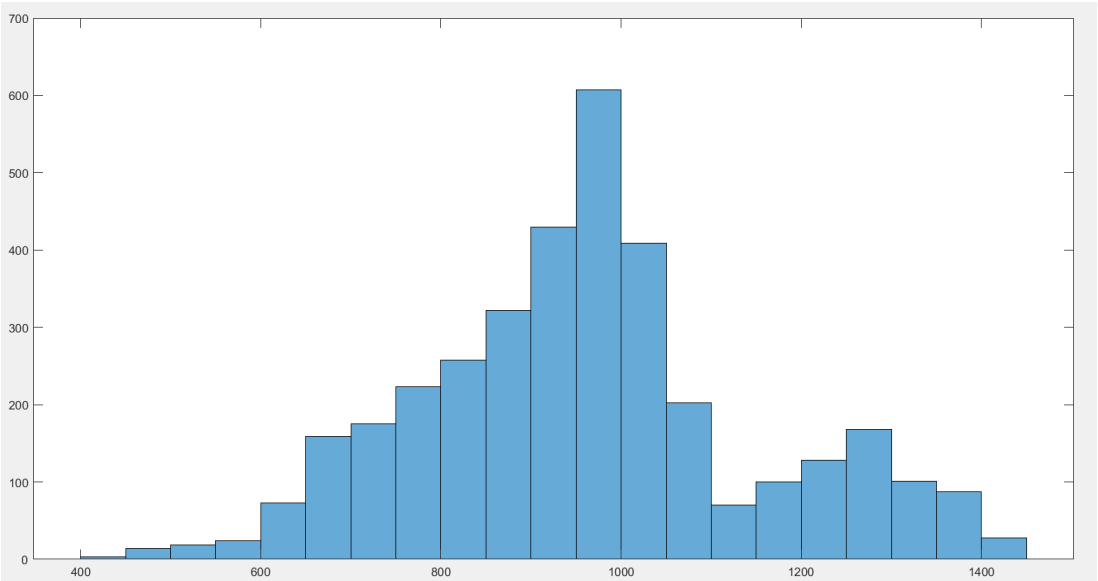

Рис. 41. Гистограмма частот f(DC^q) по Cb для изображения "baboon.bmp".

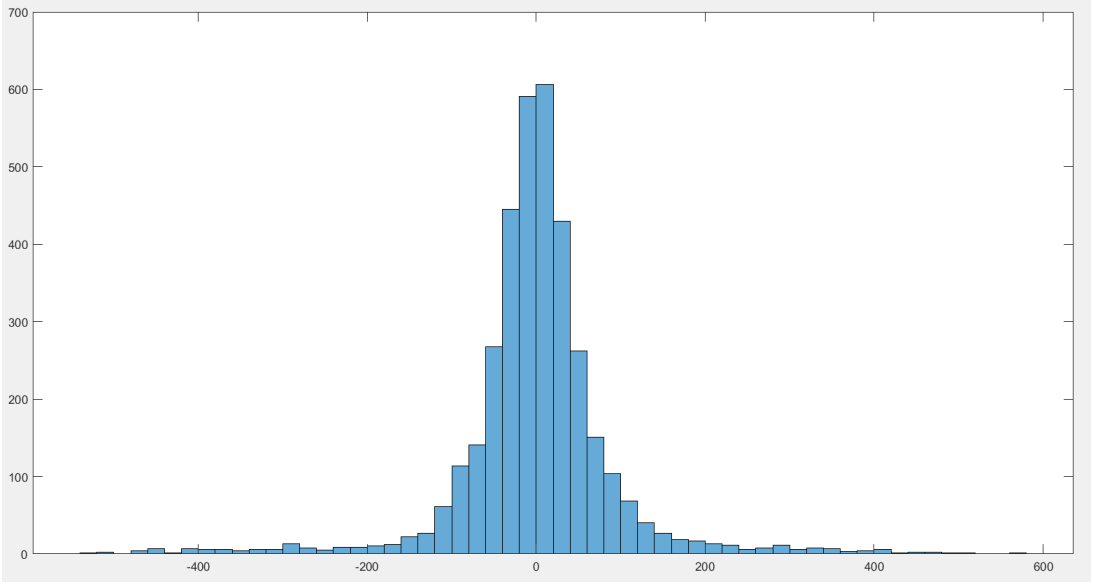

Рис. 42. Гистограмма частот f(ΔDC) по Cb для изображения "baboon.bmp".

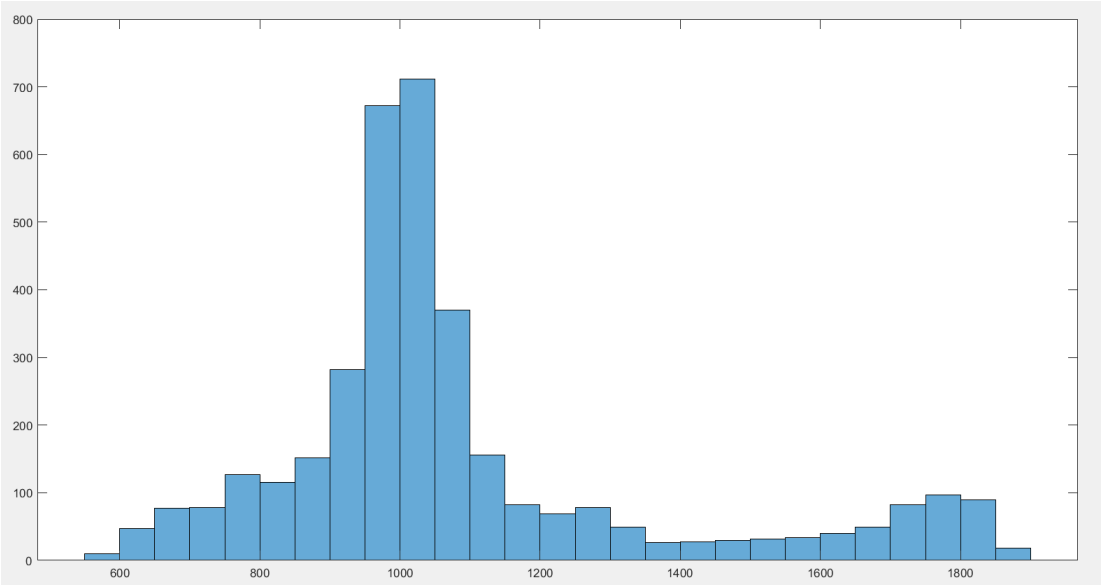

Рис. 43. Гистограмма частот f(DC^q) по Cr для изображения "baboon.bmp".

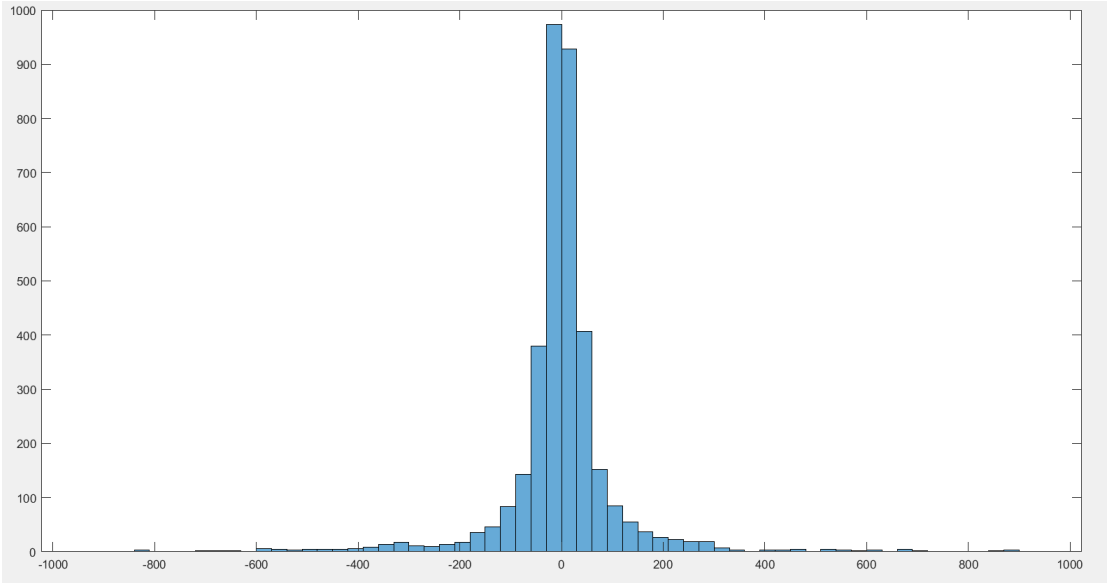

Рис. 44. Гистограмма частот f(ΔDC) по Cr для изображения "baboon.bmp".

```
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
```

Рис. 45. Значения энтропии для изображения "baboon.bmp".

Из полученных данных можно сделать вывод о том, что разностное кодирование уменьшает значение энтропии, а значит качество сжатия улучшается. Это связано с тем, что при разностном кодировании значительно увеличивается количество нулей и значений около нуля для массива ΔDC.

2.3.3. 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 последовательностью.
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 последовательность не остается ненулевых значений, то в конце новой последовательности ставится пара (0,0).
3) Значение Level заменяется на пару BC(Level), Magnitude (Level).

2.3.4. Определение соотношений размеров в сжатом битовом потоке:

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

1) $BC(\Delta DC)$,
2) $Magnitude(\Delta DC)$,
3) $\big(Run, BC(Level)\big)$,
4) $Magnitude(Level)$.

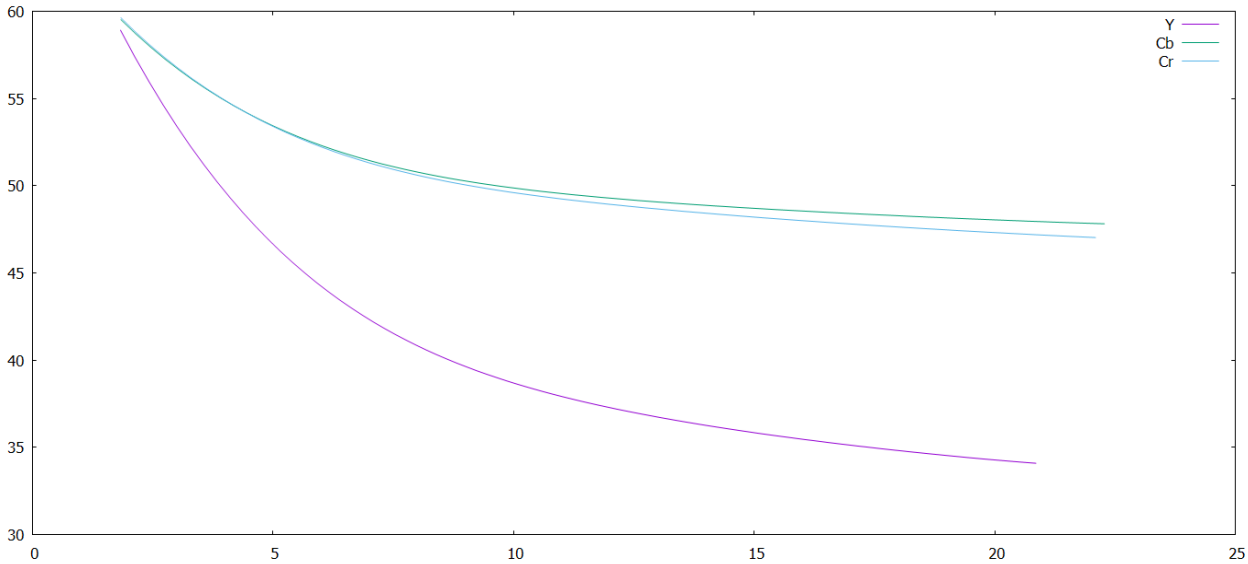

Рис. 46. График PSNR (степень сжатия) для изображения “kodim23.bmp”.

```
R = 1
Y:
BC(delta DC): 2.52972%
Magnitude(delta DC): 5.2162%
(Run, BC(Level)): 60.4112%
Magnitude(Level): 31.843%

Cb:
BC(delta DC): 9.82839%
Magnitude(delta DC): 12.7857%
(Run, BC(Level)): 54.3009%
Magnitude(Level): 23.0851%

Cr:
BC(delta DC): 9.57737%
Magnitude(delta DC): 12.1085%
(Run, BC(Level)): 54.484%
Magnitude(Level): 23.8307%


R = 10
Y:
BC(delta DC): 8.05165%
Magnitude(delta DC): 16.6022%
(Run, BC(Level)): 53.134%
Magnitude(Level): 22.2126%

Cb:
BC(delta DC): 26.9395%
Magnitude(delta DC): 35.0454%
(Run, BC(Level)): 30.1436%
Magnitude(Level): 7.87342%

Cr:
BC(delta DC): 25.5195%
Magnitude(delta DC): 32.2639%
(Run, BC(Level)): 33.1807%
Magnitude(Level): 9.03604%
```

Рис. 47. Вычисление соотношений “kodim23.bmp”.

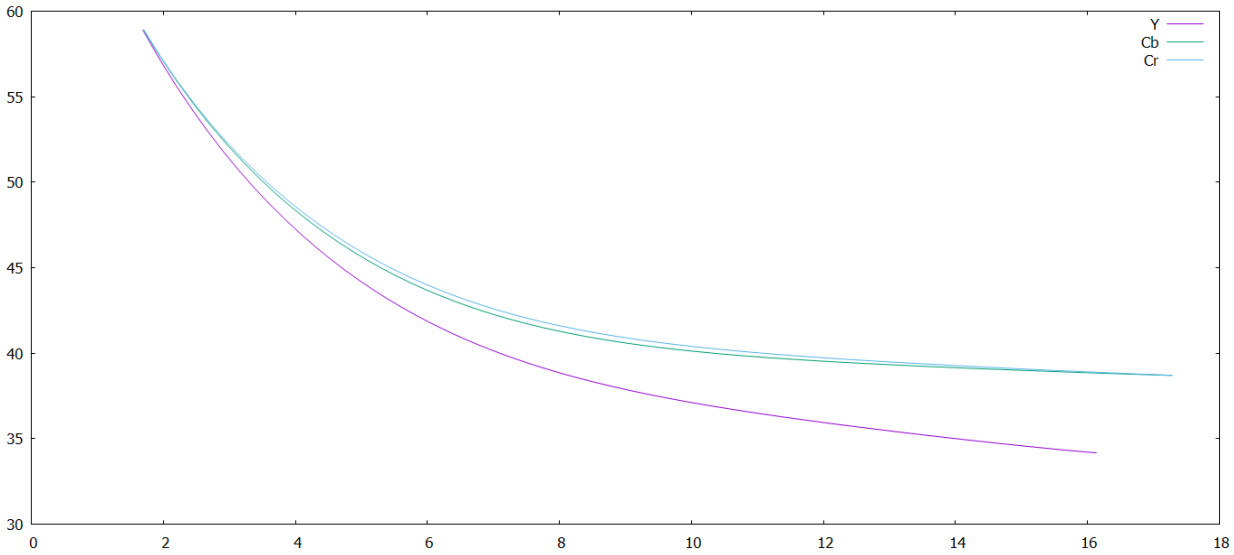

Рис. 48. График PSNR (степень сжатия) для изображения “lena.bmp”.

```
R = 1
Y:
BC(delta DC): 6.51046%
Magnitude(delta DC): 13.596%
(Run, BC(Level)): 52.7672%
Magnitude(Level): 27.1267%

Cb:
BC(delta DC): 11.8671%
Magnitude(delta DC): 18.1313%
(Run, BC(Level)): 48.9559%
Magnitude(Level): 21.046%

Cr:
BC(delta DC): 11.146%
Magnitude(delta DC): 17.6458%
(Run, BC(Level)): 49.8826%
Magnitude(Level): 21.3261%


R = 10
Y:
BC(delta DC): 15.9374%
Magnitude(delta DC): 33.2826%
(Run, BC(Level)): 36.1407%
Magnitude(Level): 14.6405%

Cb:
BC(delta DC): 28.5999%
Magnitude(delta DC): 43.6966%
(Run, BC(Level)): 21.5601%
Magnitude(Level): 6.14357%

Cr:
BC(delta DC): 27.745%
Magnitude(delta DC): 43.9248%
(Run, BC(Level)): 22.0017%
Magnitude(Level): 6.32944%
```

Рис. 49. Вычисление соотношений “lena.bmp”.

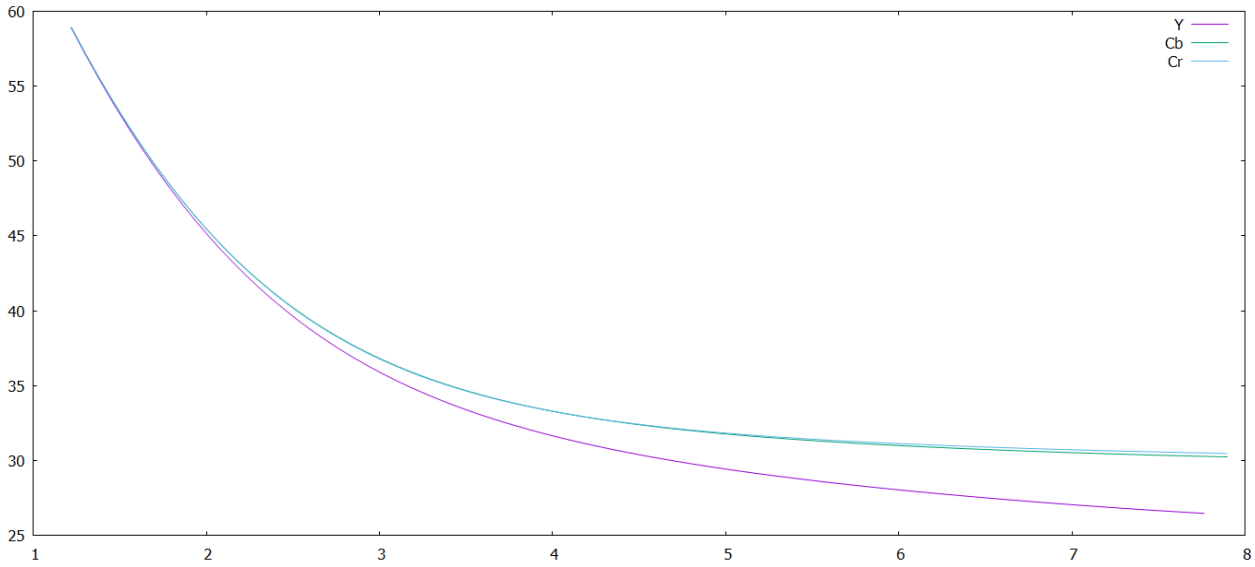

Рис. 50. График PSNR (степень сжатия) для изображения “baboon.bmp”.

```
R = 1
Y:
BC(delta DC): 3.05958%
Magnitude(delta DC): 7.17544%
(Run, BC(Level)): 59.4201%
Magnitude(Level): 30.3451%

Cb:
BC(delta DC): 5.63348%
Magnitude(delta DC): 10.8178%
(Run, BC(Level)): 58.1164%
Magnitude(Level): 25.4326%

Cr:
BC(delta DC): 6.17097%
Magnitude(delta DC): 11.0355%
(Run, BC(Level)): 57.4279%
Magnitude(Level): 25.366%


R = 10
Y:
BC(delta DC): 10.5888%
Magnitude(delta DC): 24.8332%
(Run, BC(Level)): 45.7849%
Magnitude(Level): 18.7933%

Cb:
BC(delta DC): 18.9378%
Magnitude(delta DC): 36.3657%
(Run, BC(Level)): 33.1825%
Magnitude(Level): 11.5141%

Cr:
BC(delta DC): 19.9019%
Magnitude(delta DC): 35.5904%
(Run, BC(Level)): 33.15%
Magnitude(Level): 11.3578%
```

Рис. 51. Вычисление соотношений “baboon.bmp”.

3. Дополнительное 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.

Решение:

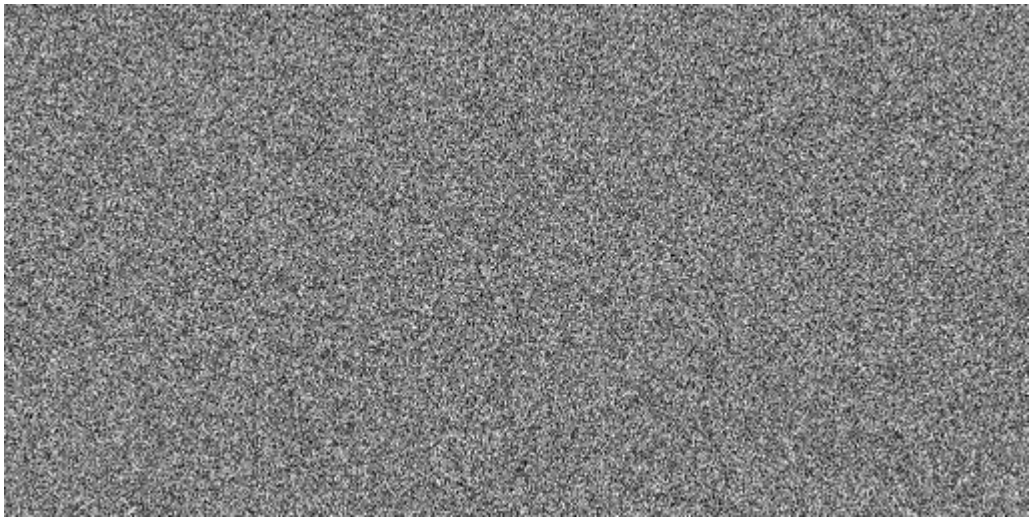

Рис. 43. Белый гауссовский шум σ =64.

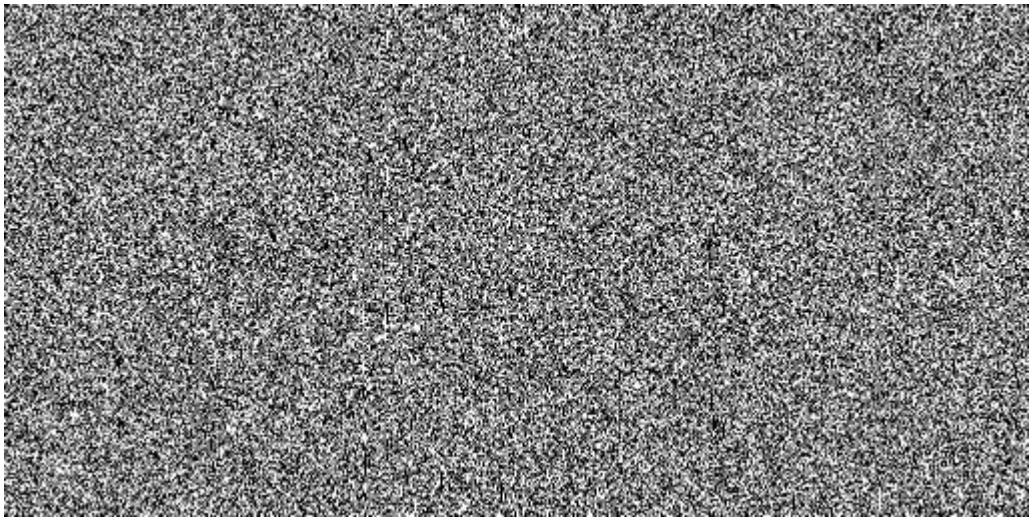

Рис. 44. Импульсный шум с p_a=p_b=50%.

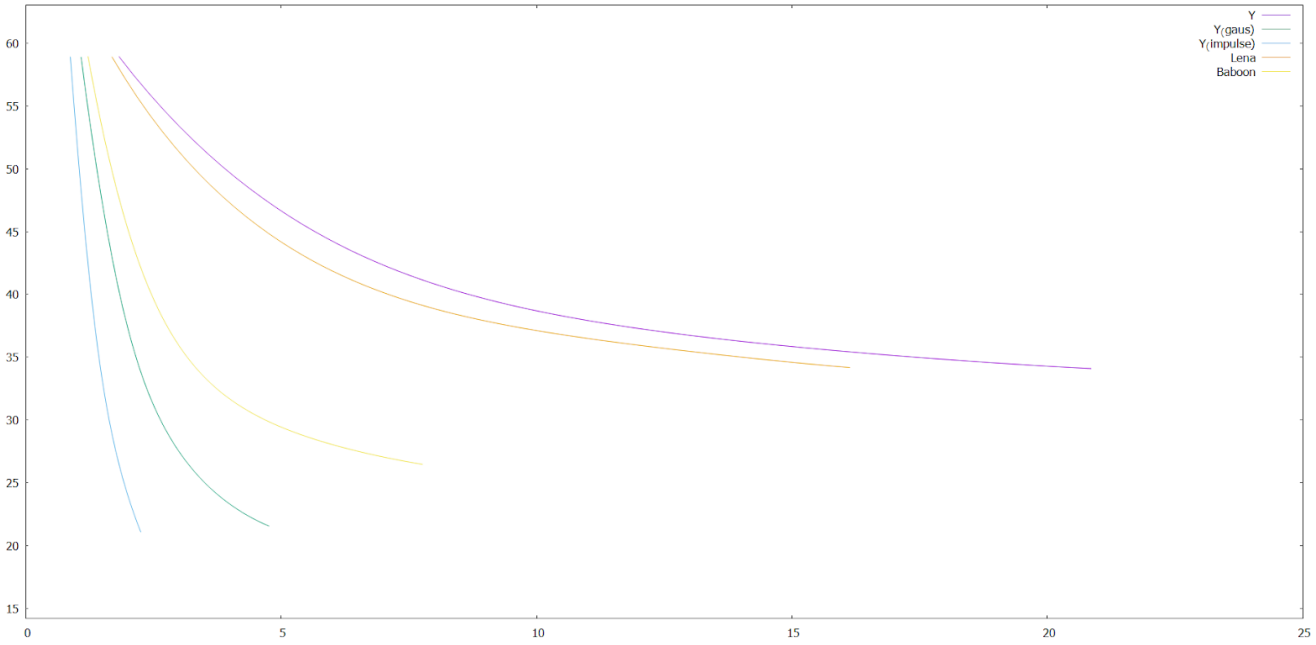

Рис. 45. Исходное изображение.

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 30-ти. Таким образом, эффективность сжатия напрямую зависит от особенностей самого изображения.

4. Выводы:

В ходе выполнения лабораторной работы было выявлено, что с помощью дискретного косинусного преобразования можно получить спектральные коэффициенты изображения. Энергия спектра каждого блока концентрируется в левом верхнем углу. После ДКП изображение восстанавливается четко, что можно увидеть на изображениях. Потери возможны в связи с погрешностями при вычислениях.

Следующим этапом лабораторной работы стало равномерное скалярное квантование. При увеличении R уменьшается значение PSNR. Для компоненты Y данные изменения заметны сильнее, так как она несет основную информацию, в отличии от компонент Cb и Cr. Кроме того, на восстановленных изображениях размыты контура, так как изображение состоит из квадратных блоков.

При разностном кодировании частоты коэффициентов постоянного тока находятся около нуля, то есть их диапазон сужается, что видно по гистограммам. Кроме того, разностное кодирование уменьшает энтропию, а значит способствует эффективному сжатию. Значения переменного тока берутся в зигзагообразном порядке, так как ДКП концентрирует значения в верхнем левом углу. В последовательности может оказаться много нулей, то есть повторяющихся значений, целесообразно кодировать ее методом длин серий RLE.

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.

5. 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, vec-
tor<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].rgb-
Blue * 0.114));
                Cb[i].push_back(clipping(0.5643 * ((dou-
ble)rgb[i][j].rgbBlue - Y[i][j]) + 128));
                Cr[i].push_back(clipping(0.7132 * ((dou-
ble)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] - aver-
age_Y));
                    vec_Cb.push_back(getCategory(DC[1][0] - aver-
age_Cb));
                    vec_Cr.push_back(getCategory(DC[2][0] - aver-
age_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) {
          fstream 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) {
          fstream 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, dou-
ble>(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, dou-
ble>(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, dou-
ble>(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++;
                        }
                    }
                    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<un-
signed 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<un-
signed char, int>>>> codAC, size_t I, int flag) {
        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, dou-
ble>(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, un-
signed 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) / (dou-
ble)(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 << "BC: " << a1 << "%" << endl;
          cout << "Magnitude: " << a2 << "%" << endl;
          //cout << d << ", ";
          if (flag == 1)
               return a1;
          else if (flag == 2)
               return a2;
          else
               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, MBITMAPIN-
FOHEADER* 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, MBIT-
MAPINFOHEADER* 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<vec-
tor<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<dou-
ble>> 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<vec-
tor<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>>& firstAr-
ray, 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<vec-
tor<double>> secondArray) {
     double niz = calculateSumSquareDifferences(firstArray, secondAr-
ray);
     double tmp = ((double)firstArray.size() * (double)firstAr-
ray[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, vec-
tor<double> tmp) {
     ofstream fout1;
     fout1.open(filename);
     for (int i = 0; i < array.size(); i++) {
          fout1 << tmp[i] << " " << array[i] << "\n";
     }
     fout1.close();
}

void writeDop(const char* filename, vector<double> array) {
     ofstream fout1;
     fout1.open(filename);
     for (int i = 0; i < array.size(); i++) {
          fout1 << (i*3) << " " << 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 * (dou-
ble)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 * (dou-
ble)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 * (dou-
ble)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 * (dou-
ble)N)) * cos(((2.0 * (double)t + 1) * M_PI * l) / (2.0 * (dou-
ble)N));
                                    tmpCb += Ck * Cl * rgb.Cb[i + k][j
+ l] * cos(((2.0 * (double)f + 1) * M_PI * (double)k) / (2.0 * (dou-
ble)N)) * cos(((2.0 * (double)t + 1) * M_PI * l) / (2.0 * (dou-
ble)N));
                                    tmpCr += Ck * Cl * rgb.Cr[i + k][j
+ l] * cos(((2.0 * (double)f + 1) * M_PI * (double)k) / (2.0 * (dou-
ble)N)) * cos(((2.0 * (double)t + 1) * M_PI * l) / (2.0 * (dou-
ble)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(bmp-
File.getY());
//        vector<vector<pair<unsigned char, pair<unsigned char,
int>>>> yCodAC = bmpFile.codingAC(yAC);
//        result.push_back(bmpFile.sizeOfStream(yCodAC, I));
//   }
//
//   return result;
//}

vector<double> dop3(YCbCr DCT, int I, int flag) {
    vector<double> result;
    for (int i = 1; i <= 60; i+= 3) {
        YCbCr bmpFile(DCT);
        cout << endl;
        //   cout << "R = " << i << endl;
        quantization(bmpFile, i, 8);
        bmpFile.generateDC();
        bmpFile.codingDC();
        vector<vector<int>> yAC = bmpFile.generateAC(bmp-
File.getY());
        vector<vector<pair<unsigned char, pair<unsigned char,
int>>>> yCodAC = bmpFile.codingAC(yAC);
        result.push_back(bmpFile.sizeOfStream(yCodAC, I, flag));
    }

    return result;
```

```
}

int main() {
     MBITMAPFILEHEADER bfh1;
     MBITMAPINFOHEADER bih1;
     MBITMAPFILEHEADER bfh2;
     MBITMAPINFOHEADER bih2;
     MBITMAPFILEHEADER bfh3;
     MBITMAPINFOHEADER bih3;
     FILE* f1;
     f1 = fopen("myImage.bmp", "rb");
     if (f1 == NULL)
     {
          cout << "reading error";
          return 0;
     }
     MRGBQUAD** rgbMyImage = readBMP(f1, &bfh1, &bih1);
     fclose(f1);
     int heightMyImage = abs(bih1.biHeight);
     int widthMyImage = abs(bih1.biWidth);
     YCbCr myImage(rgbMyImage, heightMyImage, widthMyImage);
     f1 = fopen("lena.bmp", "rb");
     if (f1 == NULL)
     {
          cout << "reading error";
          return 0;
     }
     MRGBQUAD** rgbLena = readBMP(f1, &bfh2, &bih2);
     int heightLena = abs(bih2.biHeight);
     int widthLena = abs(bih2.biWidth);
     YCbCr lena(rgbLena, heightLena, widthLena);
     fclose(f1);

     f1 = fopen("baboon.bmp", "rb");
     if (f1 == NULL)
     {
          cout << "reading 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);
     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("MyImageReverseDCT.bmp", "wb"), getRGBfromYre-
verse(myImageReverseDCT.Y, myImageReverseDCT.Cb, myImageRe-
verseDCT.Cr, heightMyImage, widthMyImage), &bfh1, &bih1, height-
MyImage, 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("lenaReverseDCT.bmp", "wb"), getRGBfromYre-
verse(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(baboon.Y, baboonRe-
verseDCT.Y) << endl;
     cout << "Cb PSNR: " << calculatePSNR(baboon.Cb, baboonRe-
verseDCT.Cb) << endl;
     cout << "Cr PSNR: " << calculatePSNR(baboon.Cr, baboonRe-
verseDCT.Cr) << endl;

     writeBMP(fopen("baboonReverseDCT.bmp", "wb"), getRGBfromYre-
verse(baboonReverseDCT.Y, baboonReverseDCT.Cb, baboonReverseDCT.Cr,
heightBaboon, widthBaboon), &bfh3, &bih3, heightBaboon, widthBaboon);

     ImagesStruct imagesStruct;
     for (int R = 0; R <= 10; R++) {
          buildPSNRgraphics(myImageDCT, lenaDCT, baboonDCT, myImage,
lena, baboon, imagesStruct, R);
     }

     writeFile("YMyImagePSNR.txt", imagesStruct.myImageY);
     writeFile("YLenaPSNR.txt", imagesStruct.lenaY);
     writeFile("YBaboonPSNR.txt", imagesStruct.baboonY);

     writeFile("CbMyImagePSNR.txt", imagesStruct.myImageCb);
     writeFile("CbLenaPSNR.txt", imagesStruct.lenaCb);
     writeFile("CbBaboonPSNR.txt", imagesStruct.baboonCb);

     writeFile("CrMyImagePSNR.txt", imagesStruct.myImageCr);
     writeFile("CrLenaPSNR.txt", imagesStruct.lenaCr);
     writeFile("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("myImageR1.bmp", "wb"), getRGBfromYreverse(myIm-
ageReverseDCT.Y, myImageReverseDCT.Cb, myImageReverseDCT.Cr, height-
MyImage, widthMyImage), &bfh1, &bih1, heightMyImage, widthMyImage);

    quantization(myImageDCT_c1, 5, 8);
    dequantization(myImageDCT_c1, 5, 8);
    YCbCr myImageReverseDCT_c1 = ReverseDCT(8, myImageDCT_c1);
    writeBMP(fopen("myImageR5.bmp", "wb"), getRGBfromYreverse(myIm-
ageReverseDCT_c1.Y, myImageReverseDCT_c1.Cb, myImageReverseDCT_c1.Cr,
heightMyImage, widthMyImage), &bfh1, &bih1, heightMyImage, width-
MyImage);

    quantization(myImageDCT_c2, 10, 8);
    dequantization(myImageDCT_c2, 10, 8);
    YCbCr myImageReverseDCT_c2 = ReverseDCT(8, myImageDCT_c2);
    writeBMP(fopen("myImageR10.bmp", "wb"), getRGBfromYreverse(myIm-
ageReverseDCT_c2.Y, myImageReverseDCT_c2.Cb, myImageReverseDCT_c2.Cr,
heightMyImage, widthMyImage), &bfh1, &bih1, heightMyImage, width-
MyImage);

    YCbCr lenaDCT_c1(lenaDCT);
    YCbCr lenaDCT_c2(lenaDCT);

    quantization(lenaDCT, 1, 8);
    dequantization(lenaDCT, 1, 8);
    lenaReverseDCT = ReverseDCT(8, lenaDCT);
    writeBMP(fopen("lenaR1.bmp", "wb"), getRGBfromYreverse(lena-
ReverseDCT.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("lenaR5.bmp", "wb"), getRGBfromYreverse(lena-
ReverseDCT_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("lenaR10.bmp", "wb"), getRGBfromYreverse(lena-
ReverseDCT_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("baboonR1.bmp", "wb"), getRGBfromYreverse(baboon-
ReverseDCT.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("baboonR5.bmp", "wb"), getRGBfromYreverse(baboon-
ReverseDCT_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("baboonR10.bmp", "wb"), getRGBfromYreverse(ba-
boonReverseDCT_c2.Y, baboonReverseDCT_c2.Cb, baboonReverseDCT_c2.Cr,
heightBaboon, widthBaboon), &bfh3, &bih3, heightBaboon, widthBaboon);

     //DC

     cout << "My image:" << endl;
     quantization(myImageDCT, 1, 8);
     myImageDCT.generateDC();
     myImageDCT.codingDC();
     myImageDCT.buildHistogram(myImageDCT.getDC(), "YmyImageHisto-
gramDC.txt", 0);
     myImageDCT.buildHistogram(myImageDCT.getCodDC(), "YmyImageHisto-
gramCodDC.txt", 0);
     myImageDCT.buildHistogram(myImageDCT.getDC(), "CbmyImageHisto-
gramDC.txt", 1);
     myImageDCT.buildHistogram(myImageDCT.getCodDC(), "CbmyImageHis-
togramCodDC.txt", 1);
     myImageDCT.buildHistogram(myImageDCT.getDC(), "CrmyImageHisto-
gramDC.txt", 2);
     myImageDCT.buildHistogram(myImageDCT.getCodDC(), "CrmyImageHis-
togramCodDC.txt", 2);
     myImageDCT.calculateEntropy();

     cout << "Lena:" << endl;
     quantization(lenaDCT, 1, 8);
     lenaDCT.generateDC();
     lenaDCT.codingDC();
     lenaDCT.buildHistogram(lenaDCT.getDC(), "YlenaHistogramDC.txt",
0);
     lenaDCT.buildHistogram(lenaDCT.getCodDC(), "YlenaHisto-
gramCodDC.txt", 0);
     lenaDCT.buildHistogram(lenaDCT.getDC(), "CblenaHistogramDC.txt",
1);
     lenaDCT.buildHistogram(lenaDCT.getCodDC(), "CblenaHisto-
gramCodDC.txt", 1);
     lenaDCT.buildHistogram(lenaDCT.getDC(), "CrlenaHistogramDC.txt",
2);
     lenaDCT.buildHistogram(lenaDCT.getCodDC(), "CrlenaHisto-
gramCodDC.txt", 2);
```

```
     lenaDCT.calculateEntropy();

     cout << "Baboon:" << endl;
     quantization(baboonDCT, 1, 8);
     baboonDCT.generateDC();
     baboonDCT.codingDC();
     baboonDCT.buildHistogram(baboonDCT.getDC(), "YbaboonHisto-
gramDC.txt", 0);
     baboonDCT.buildHistogram(baboonDCT.getCodDC(), "YbaboonHisto-
gramCodDC.txt", 0);
     baboonDCT.buildHistogram(baboonDCT.getDC(), "CbBaboonHisto-
gramDC.txt", 1);
     baboonDCT.buildHistogram(baboonDCT.getCodDC(), "CbBaboonHisto-
gramCodDC.txt", 1);
     baboonDCT.buildHistogram(baboonDCT.getDC(), "CrBaboonHisto-
gramDC.txt", 2);
     baboonDCT.buildHistogram(baboonDCT.getCodDC(), "CrBaboonHisto-
gramCodDC.txt", 2);
     baboonDCT.calculateEntropy();
     //RLE


     //YCbCr bmpFile(myImageDCT);
     //YCbCr bmpFile1(myImageDCT);

     //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(bmp-
File.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(bmp-
File.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(bmp-
File1.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(bmp-
File1.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(bmp-
File1.getCr());
     //vector<vector<pair<unsigned char, pair<unsigned char, int>>>>
crCodAC1 = bmpFile1.codingAC(crAC1);
     //bmpFile1.sizeOfStream(crCodAC1, 2);


     //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(bmp-
File.getY());
     //   vector<vector<pair<unsigned char, pair<unsigned char,
int>>>> yCodAC = bmpFile.codingAC(yAC);
     //   bmpFile.sizeOfStream(yCodAC, 0);
     //   vector<vector<int>> cbAC = bmpFile.generateAC(bmp-
File.getCb());
     //   vector<vector<pair<unsigned char, pair<unsigned char,
int>>>> cbCodAC = bmpFile.codingAC(cbAC);
     //   bmpFile.sizeOfStream(cbCodAC, 1);
     //   vector<vector<int>> crAC = bmpFile.generateAC(bmp-
File.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("YMyImagePSNR.txt", imagesStructNew.myImageY, print-
Dop(myImageDCT, 0));
```

```
     //writeFile("YLenaPSNR.txt", imagesStructNew.lenaY, printDop(le-
naDCT, 0));
     //writeFile("YBaboonPSNR.txt", imagesStructNew.baboonY, print-
Dop(baboonDCT, 0));

     //writeFile("CbMyImagePSNR.txt", imagesStructNew.myImageCb,
printDop(myImageDCT, 1));
     //writeFile("CbLenaPSNR.txt", imagesStructNew.lenaCb, print-
Dop(lenaDCT, 1));
     //writeFile("CbBaboonPSNR.txt", imagesStructNew.baboonCb, print-
Dop(baboonDCT, 1));

     //writeFile("CrMyImagePSNR.txt", imagesStructNew.myImageCr,
printDop(myImageDCT, 2));
     //writeFile("CrLenaPSNR.txt", imagesStructNew.lenaCr, print-
Dop(lenaDCT, 2));
     //writeFile("CrBaboonPSNR.txt", imagesStructNew.baboonCr, print-
Dop(baboonDCT, 2));

     //Dop

     ImagesStruct imagesStructDop;
     for (int R = 0; R <= 10; R++) {
          buildPSNRgraphics(myImageDCT, lenaDCT, baboonDCT, myImage,
lena, baboon, imagesStructDop, R);
     }

     writeDop("dopMyImageBC.txt", dop3(myImageDCT, 0, 1));
     writeDop("dopLenaRunBC.txt", dop3(lenaDCT, 0, 1));
     writeDop("dopBaboonBC.txt", dop3(baboonDCT, 0, 1));

     writeDop("dopMyImageMagnitude.txt", dop3(myImageDCT, 0, 2));
     writeDop("dopLenaMagnitude.txt", dop3(lenaDCT, 0, 2));
     writeDop("dopBaboonMagnitude.txt", dop3(baboonDCT, 0, 2));

     return 0;
}
```