

*Цель работы:* изучение способов представления изображений, ознакомление со структурой формата BMP, анализ статистических свойств изображений, а также получение практических навыков обработки изображений.

## 1 Описание структуры формата BMP

Структура RGB24:

- 1) Заголовок.
- 2) Данные по пикселям:

Заголовок BITMAPFILEHEADER имеет следующую структуру:

- WORD bfType – описывает тип файла;
- DWORD bfSize – размер файла в байтах;
- WORD bfReserved1 – зарезервировано, должно быть 0;
- WORD bfReserved2 – зарезервировано, должно быть 0;
- DWORD bfOffBits – смещение в байтах от начала заголовка до массива пикселей.

Заголовок BITMAPINFOHEADER имеет следующий вид:

- DWORD biSize – размер структуры в байтах;
- LONG biWidth – ширина изображения в пикселях;
- LONG biHeight – высота изображения в пикселях;
- WORD biPlanes – количество плоскостей изображения;
- WORD biBitCount – количество бит на один пиксель;
- DWORD biCompression – тип сжатия;
- DWORD biSizeImage – размер изображения в байтах;
- LONG biXPelsPerMeter – горизонтальное разрешение в пикселях на метр;
- LONG biYPelsPerMeter – вертикальное разрешение в пикселях на метр;
- DWORD biClrUsed – размер палитры;
- DWORD biClrImportant – число значимых элементов палитры.

После заголовка следует информация о пикселях. Данная информация представлена в виде одномерного массива, в котором значения, относящиеся к отдельным пикселям, записаны

строка за строкой. Строки могут идти как снизу-вверх, так и сверху-вниз. Ширина строки в байтах должна быть выравнена по границе двойного слова (32 бита), т.е. должна быть кратна числу 4. При необходимости для выполнения этого условия в конце каждой строки добавляются дополнительные байты (от 1 до 3-х), значения которых несущественны. Каждый пиксель в формате RGB24 представляется тремя байтами. Первый содержит значение компоненты В, второй — G, третий — R. Структура имеет название RGB.

При исследовании использовались поля:

- biWidth, biHeight – для определения количества пикселей в картинке и последующей их;

## 2 Исходное изображение в формате BMP

Для выполнения работы было выбрано следующее изображение:



*Рисунок 1 - Исходное изображение*

## 3 Загрузка в память BMP-файла

Для загрузки в память исходного изображения были программно реализованы структуры BITMAPFILEHEADER, BITMAPINFOHEADER и структура RGB для хранения одного пикселя. В результате чтения, информация о файле была записана в две структуры, описывающих заголовок, и массив пикселей RGB.

## 4 Выделение содержимого R, G и B

Чтобы получить каждую компоненту необходимо сформировать три изображения по следующему правилу: все байты, не относящиеся к данной компоненте обнулены. Таким образом были получены файлы:



Рисунок 2 - Компонента В



Рисунок 3 - Компонента G

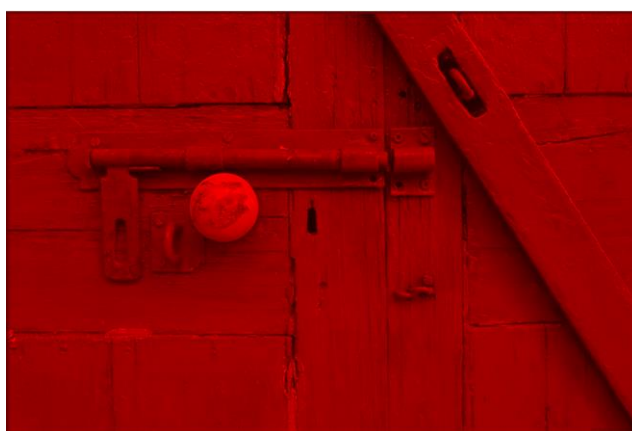


Рисунок 4 - Компонента R

## 5 Анализ корреляционных свойств компонент R, G, B

Анализ будет выполняться с использованием формулы оценки коэффициента корреляции:

$$\hat{r}_{A,B} = \frac{\hat{M}[(A - \hat{M}[A])(B - \hat{M}[B])]}{\hat{\sigma}_A \hat{\sigma}_B},$$

где A и B — компоненты изображения,  $\hat{M}[\ ]$  - математическое ожидание в соответствии с формулой:

$$\hat{M}[I^{(A)}] = \frac{1}{WH} \sum_{i=1}^H \sum_{j=1}^W I_{i,j}^{(A)},$$

$\sigma_A, \sigma_B$  — оценки среднеквадратичного отклонения компонент А и В, вычисляемые по формуле:

$$\hat{\sigma}_A = \sqrt{\frac{1}{WH-1} \sum_{i=1}^H \sum_{j=1}^W \left( I_{i,j}^{(A)} - \hat{M}[I^{(A)}] \right)^2}.$$

Коэффициент корреляции указывает на наличие связи и принимает значения от  $-1$  до  $+1$ .

### 5.1 Оценка коэффициентов корреляции между парой компонент изображения

Полученные коэффициенты корреляции:

```
r_G,B = 0.973972
r_R,B = 0.406271
r_G,R = 0.527908
```

Рисунок 5 - Коэффициент корреляции

Как видно по полученным данным, компоненты В и G более зависимы друг с другом, чем другие пары компонент.

### 5.2 Построение сечения трехмерного графика оценки нормированной автокорреляционной функции

В результате были получены графики:

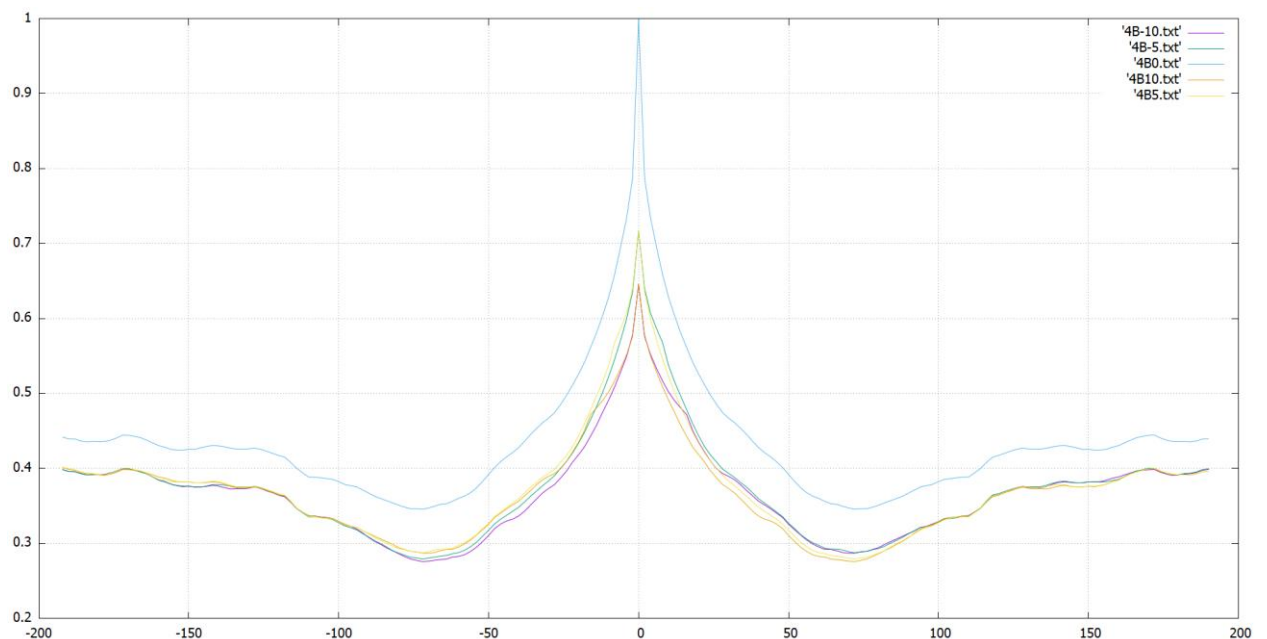


Рисунок 6 - Сечение графика автокорреляционной функции компоненты В

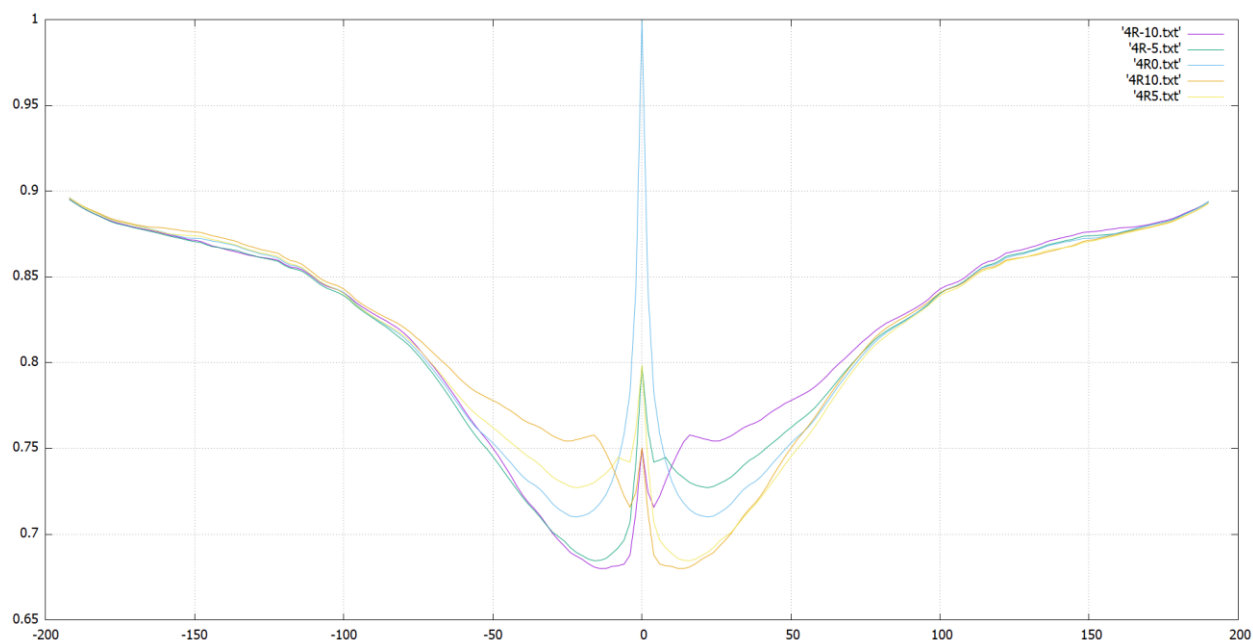


Рисунок 7 - Сечение графика автокорреляционной функции компоненты R

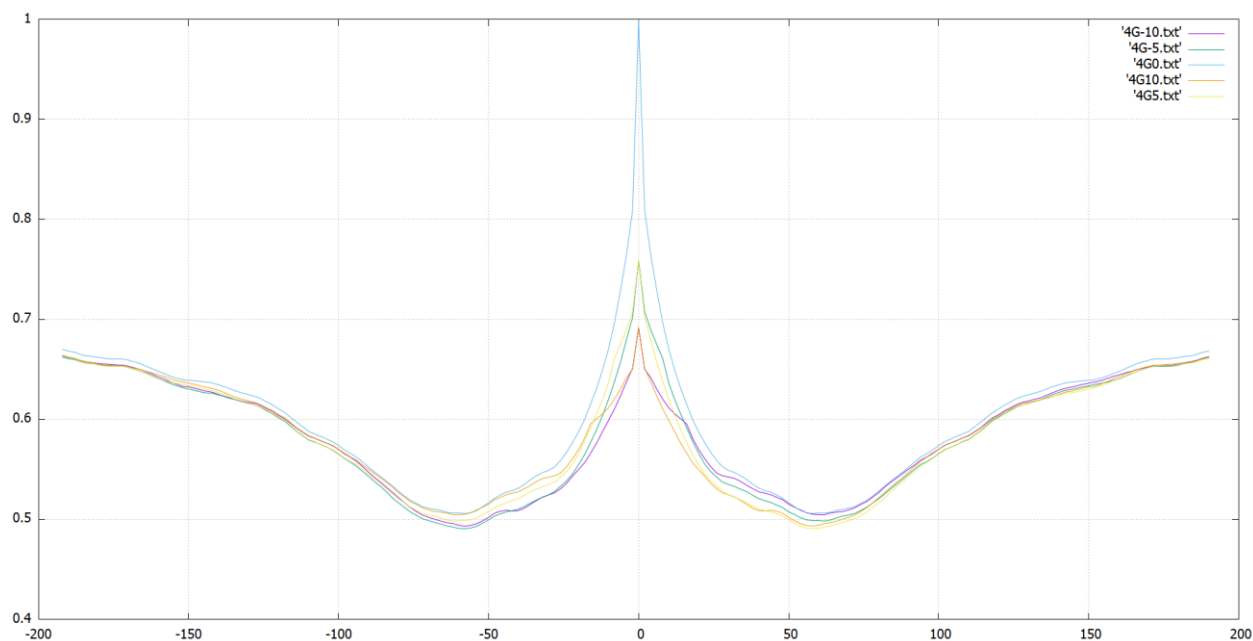


Рисунок 8 - Сечение графика автокорреляционной функции компоненты G

При  $y = 0$  и  $x = 0$ , значения автокорреляционной функции равно 1 для всех компонент, так как исходная картинка не сдвигается.

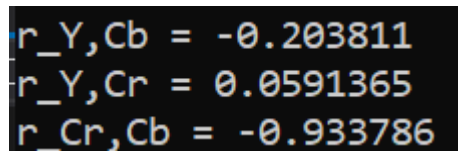
Также, при изменении  $x$  на одинаковые значения в отрицательную и положительную сторону, максимальное значение корреляции будет меньше 1 и одинакова для положительного и отрицательного  $x$ , так как картинка смещается на одно и то же количество пикселей.

## 6 Преобразование данных из формата RGB в формат YCbCr

Преобразование данных из RGB в YCbCr производится по формуле:

$$\begin{aligned}Y &= 0.299R + 0.587G + 0.114B; \\C_b &= 0.5643(B - Y) + 128; \\C_r &= 0.7132(R - Y) + 128.\end{aligned}$$

Значения коэффициента корреляции между компонентами YCbCr:


$$\begin{aligned}r_{Y,Cb} &= -0.203811 \\r_{Y,Cr} &= 0.0591365 \\r_{Cr,Cb} &= -0.933786\end{aligned}$$

*Рисунок 9 - Коэффициент корреляции*

## 7 Выделение содержимого компонент Y, Cb, Cr

В результате были получены изображения:



*Рисунок 10 - Компонента Y*



*Рисунок 11 - Компонента Cb*



*Рисунок 12 - Компонента Cr*

## 8 Обратное преобразование из YCbCr в RGB

Обратное преобразование выполняется по формулам:

$$G = Y - 0.714(C_r - 128) - 0.334(C_b - 128);$$

$$R = Y + 1.402(C_r - 128);$$

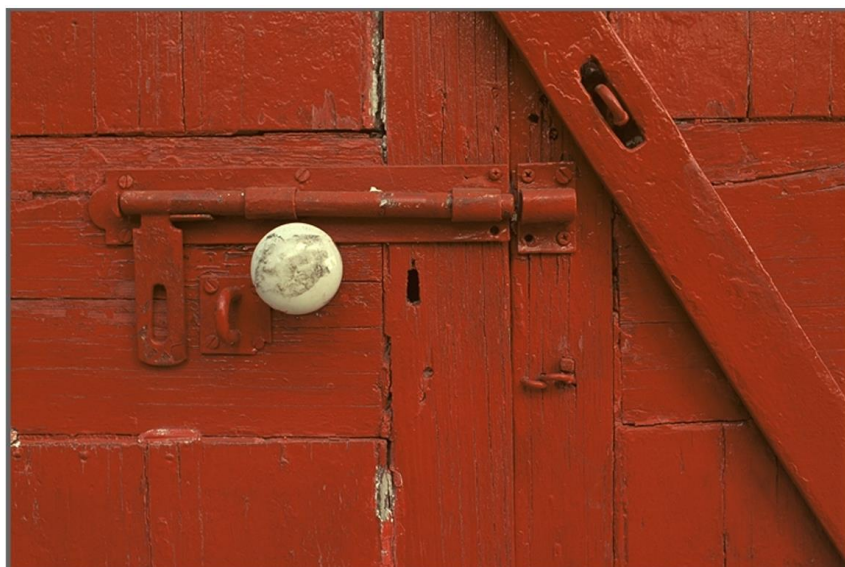
$$B = Y + 1.772(C_b - 128).$$



В результате обратного преобразования были получены следующие изображения:



*Рисунок 13 - Исходное изображение*



*Рисунок 14 - Восстановленное изображение*

Стоит отметить, что все преобразование выполняются с помощью округления, поэтому преобразование форматов в общем случае является необратимым. Из-за этого в результате обратного преобразования возможен выход восстановленных значений за границы исходного диапазона. В этом случае необходимо выполнить операцию клиппирования:

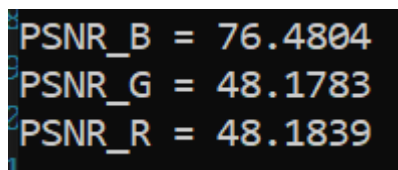
$$\text{Sat}(x, x_{\min}, x_{\max}) = \begin{cases} x_{\min} & , \text{если } x < x_{\min}; \\ x_{\max} & , \text{если } x > x_{\max}; \\ x & , \text{во всех иных случаях.} \end{cases}$$



Значение отношения Сигнал/Шум (PSNR) вычисляется по формуле:

$$PSNR = 10 \lg \frac{WH(2^L - 1)^2}{\sum_{i=1}^H \sum_{j=1}^W (I_{i,j}^{(A)} - \hat{I}_{i,j}^{(A)})^2}.$$

Полученные значения PSNR:



```
PSNR_B = 76.4804
PSNR_G = 48.1783
PSNR_R = 48.1839
```

Рисунок 15 - Значения PSNR

Можно сделать вывод, что чем ниже PSNR, тем хуже восстановились компоненты. И хотя визуальных изменений не видно, однако по значениям PSNR можно сделать вывод о том, что лучше всех восстановилась компонента B, а хуже G. Стоит отметить, что изображение визуально не имеет искажений при PSNR, значение которого больше или равно 35.

## 9 Децимация компонент Cb и Cr

Децимация производится только на отдельно записанных компонентах Cb и Cr, так как в изображениях формата YCbCr они не несут основную информационную нагрузку (за это отвечает в основном яркостная компонента Y), можно в несколько раз уменьшить информацию, передаваемую данными компонентами без серьёзных потерь в качестве изображения. Для этого существует понятие "децимация". В работе тестируется децимация двух видов:

- а) Исключение строк и столбцов с нечётными номерами.
- б) Среднее арифметическое смежных элементов (соседи по диагонали не рассматриваются, только верхние, нижние, левые и правые).

### 9.1 Децимация в 2 раза по ширине и высоте

В результате выполнения децимации в 2 раза по ширине и высоте по правилу а), восстановления исходного размера путем копирования недостающих компонент и выполнения обратного преобразования из YCbCr в RGB были получены следующие значения PSNR:

```
Decimation 8a
PSNR_Cb = 45.0741
PSNR_Cr = 39.929

PSNR_B = 40.2696
PSNR_G = 43.3428
PSNR_R = 36.9227
```

*Рисунок 16 - Децимация в 2 раза по ширине и высоте правилу a*

Можно сделать вывод о том, что значения каждой из RGB компонент уменьшились. Визуально изменения не видны, т.к. значение PSNR все еще остается высоким.

В результате выполнения децимации в 2 раза по ширине и высоте по правилу b), восстановления исходного размера путем копирования недостающих компонент и выполнения обратного преобразования из YCbCr в RGB были получены следующие значения PSNR:

```
Decimation 8b
PSNR_Cb = 48.1227
PSNR_Cr = 43.0059

PSNR_B = 42.8632
PSNR_G = 45.8175
PSNR_R = 39.8505
```

*Рисунок 17 - Децимация в 2 раза по ширине и высоте правилу b*

Можно сделать вывод о том, что значения каждой из RGB компонент относительно исходного уменьшились. Стоит обратить внимание и на то, что полученные значения выше предыдущих. Это объясняется тем, что при исключении удаляемые значения учитывались.

## 9.2 Децимация в 4 раза по ширине и высоте

Аналогичные шаги были проделаны для децимации в 4 раза по ширине и высоте. В итоге получены значения PSNR:

```
Decimation 11a
PSNR_Cb = 38.8482
PSNR_Cr = 33.4

PSNR_B = 34.0565
PSNR_G = 38.2165
PSNR_R = 30.4768
```

*Рисунок 18 - Децимация в 4 раза по ширине и высоте правилу a*

Видно, что значения PSNR сильно относительно исходных уменьшились и достигли тех значений, когда искажения будут видны на изображении:



*Рисунок 19 - Часть исходного изображения*



*Рисунок 20 - Часть восстановленного изображения*

В результате выполнения децимации в 4 раза по ширине и высоте по правилу b), восстановления исходного размера путем копирования недостающих компонент и выполнения обратного преобразования из YCbCr в RGB были получены следующие значения PSNR:

```
Decimation 11b
PSNR_Cb = 42.7327
PSNR_Cr = 37.3763

PSNR_B = 38.1421
PSNR_G = 42.0922
PSNR_R = 34.3993
```

*Рисунок 21 - Децимация в 4 раза по ширине и высоте правилу b*

Опять же значения PSNR сильно уменьшились и достигли тех значений, когда искажения можно визуально оценить:



*Рисунок 22 - Часть исходного изображения*



*Рисунок 23 - Часть восстановленного изображения*

Однако стоит отметить, что по значениям PSNR больше всего искажений было после децимации в 4 раза по правилу a.

10 Гистограммы частот

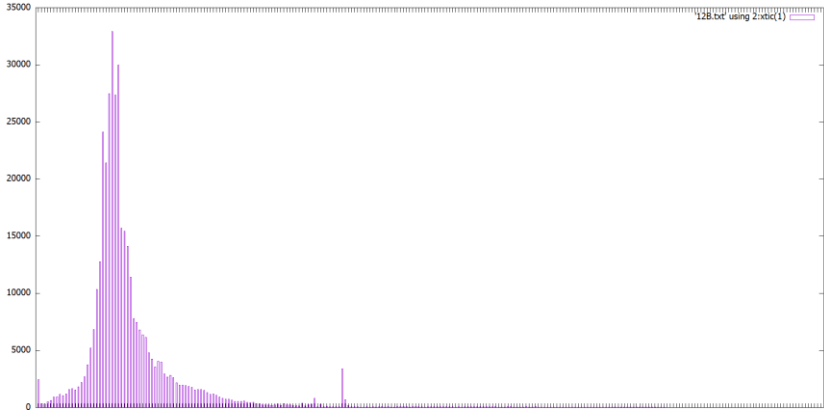


Рисунок 24 - В

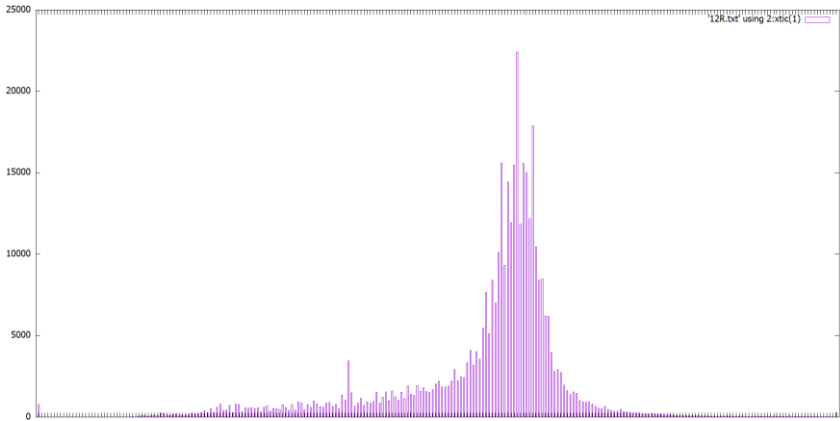


Рисунок 25 – R

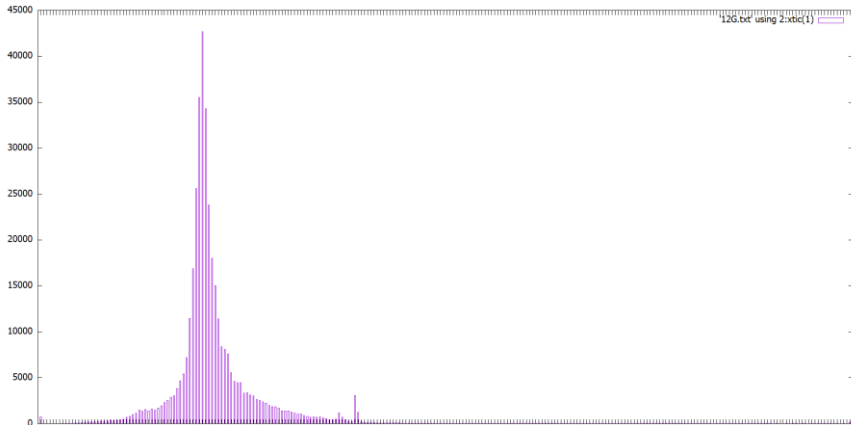


Рисунок 26 – G

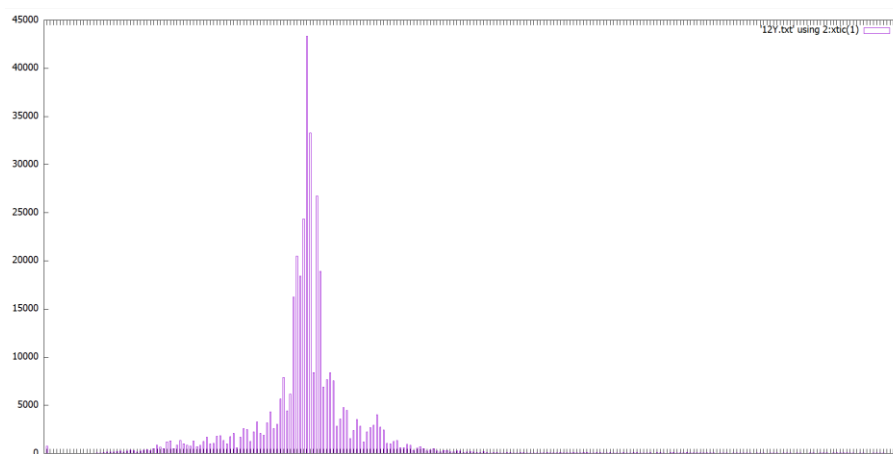


Рисунок 27 – Y

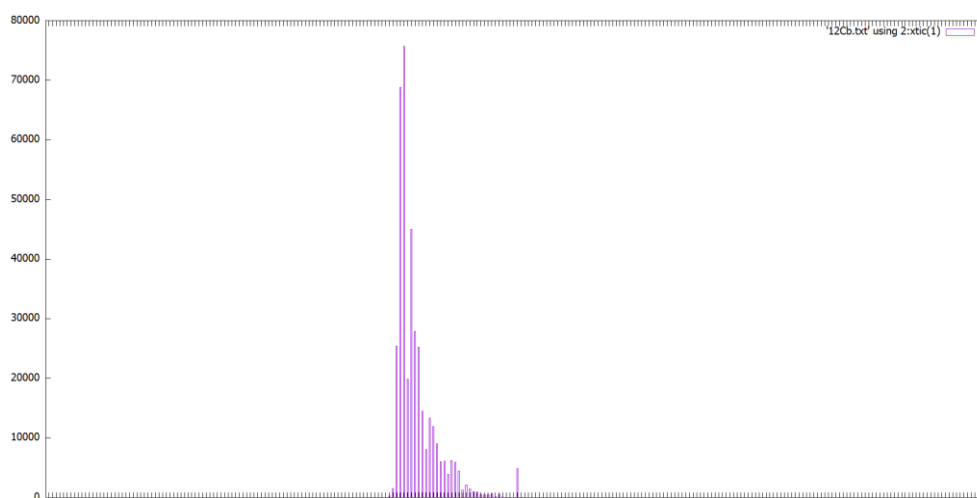


Рисунок 28 – Cb

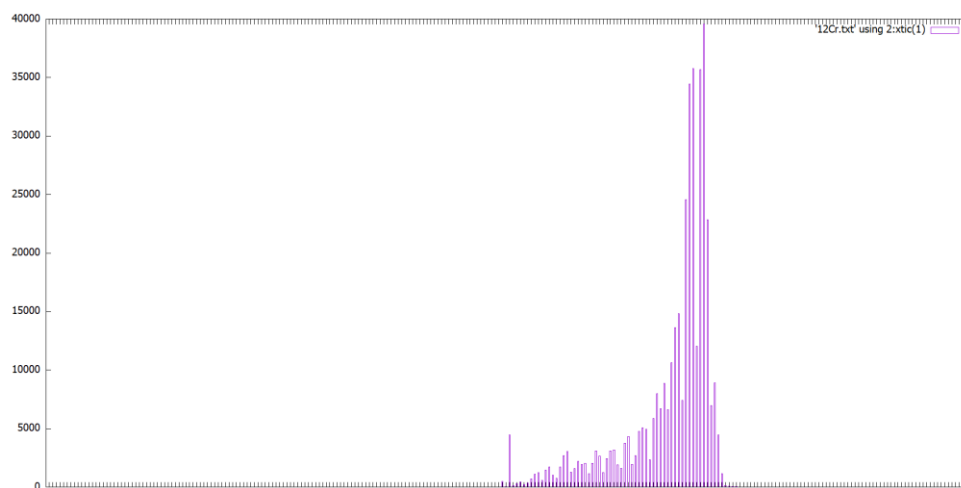


Рисунок 29 - Cr

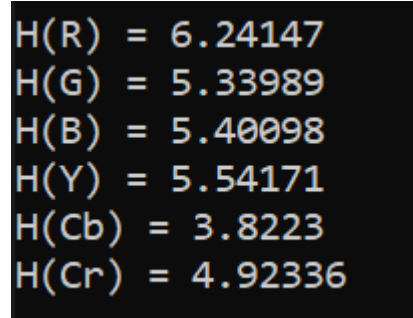
Можно сделать вывод, что компоненты R, G, B, Y распределены почти по всей области значений, а Cb и Cr определены в одной области, что говорит нам, что эти компоненты будут сжиматься намного лучше.



## 11 Энтропия

Энтропия для каждой компоненты вычисляется по формуле:

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



H(R)	=	6.24147
H(G)	=	5.33989
H(B)	=	5.40098
H(Y)	=	5.54171
H(Cb)	=	3.8223
H(Cr)	=	4.92336

Рисунок 30 - Значения энтропии

Энтропия позволяет определить эффективность сжатия для компонент. Как видно, эффективность сжатия выше у компонент Cb и Cr.

## 12 Количественный анализ эффективности разностного кодирования

Требуется сформировать массивы  $D_A^{(r)}$ , где  $A$  — указывает компоненту исходного изображения, для которого вычисляется разность, а  $r$  — номер правила вычисления разности. Значение на позиции  $(i; j)$  следует вычислять по следующей формуле:

$$d_A^{(r)}(i; j) = a(i; j) - f^{(r)}(i; j)$$

где  $a(i; j)$  — значение пикселя в компоненте  $A$  на позиции  $(i; j)$ , а  $f^{(r)}(i; j)$  — правило с номером  $r$ . Используются следующие правила:

- 1) сосед слева  $(i; j - 1)$
- 2) сосед сверху  $(i - 1; j)$
- 3) сосед сверху слева  $(i - 1; j - 1)$
- 4) среднее арифметическое трех соседей

Пиксели в первой строке и столбце исключаются из рассмотрения, т.к. не имеют некоторых соседей.

### 12.1 Массив DPCM, построенный по правилу левого соседа

Энтропии для каждой из компонент имеют следующие значения:

```
DPCM_left
H(R) = 4.79927
H(G) = 4.54747
H(B) = 4.59456

H(Y) = 4.35669
H(Cb) = 1.6465
H(Cr) = 2.35636
```

Рисунок 31 - Значения энтропии

12.2 Массив DPCM , построенный по правилу верхнего соседа

Энтропии для каждой из компонент имеют следующие значения:

```
DPCM_right
H(R) = 4.88776
H(G) = 4.64906
H(B) = 4.69381

H(Y) = 4.46294
H(Cb) = 1.54704
H(Cr) = 2.23839
```

Рисунок 32 - Значения энтропии

12.3 Массив DPCM, построенный по правилу верхнего левого соседа

Энтропии для каждой из компонент имеют следующие значения:

```
DPCM_up_left
H(R) = 5.38652
H(G) = 5.06785
H(B) = 5.12497

H(Y) = 4.92893
H(Cb) = 2.19139
H(Cr) = 3.05555
```

Рисунок 33 - Значения энтропии

12.4 Массив DPCM, построенный по правилу среднего арифметического трех соседей:  
сверху, слева, сверху – слева

Энтропии для каждой из компонент имеют следующие значения:

```
DPCM_average  
H(R) = 4.76289  
H(G) = 4.43221  
H(B) = 4.49813  
  
H(Y) = 4.3599  
H(Cb) = 1.49511  
H(Cr) = 2.36902
```

*Рисунок 34 - Значения энтропии*

По полученным результатам видно, что алгоритм DPCM, эффективнее работает для компонент Y, Cb, Cr, так как значения энтропии снизились.

### 13 Дополнительное задание 4d

#### 13.1 Разложение изображения на подкадры



*Рисунок 35 - Y00*



*Рисунок 36 - Y01*



*Рисунок 37 - Y10*



*Рисунок 38 - Y11*

Изображения одинаковые, а значит, можно сделать вывод, что на тех позициях, которые остаются и исключаются находятся схожие значения компонент.

## 13.2 Гистограммы подкадров

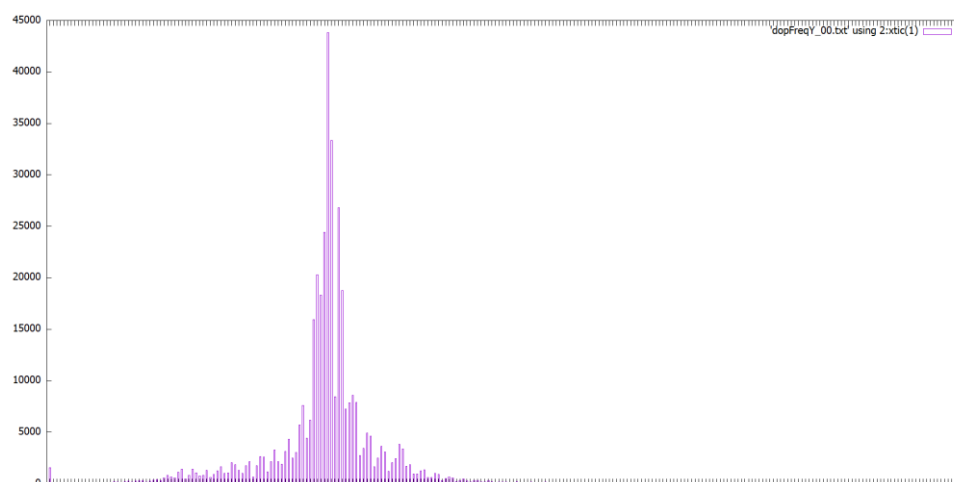


Рисунок 39 - Y00

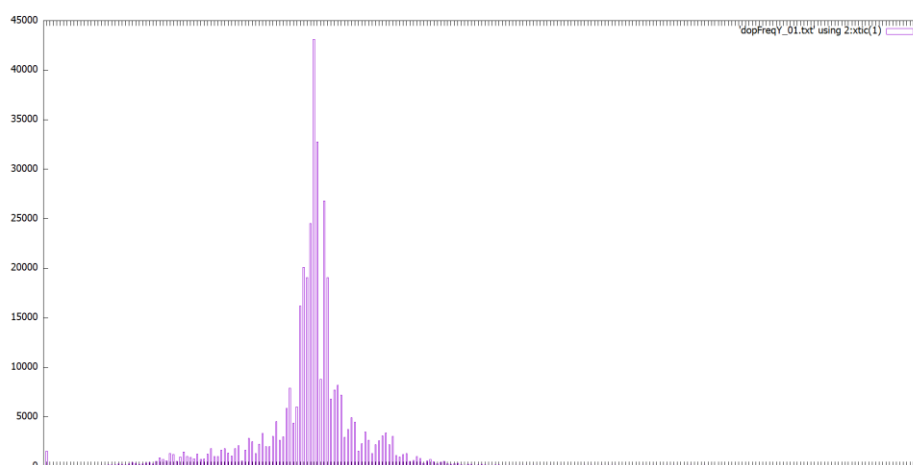


Рисунок 40 - Y01

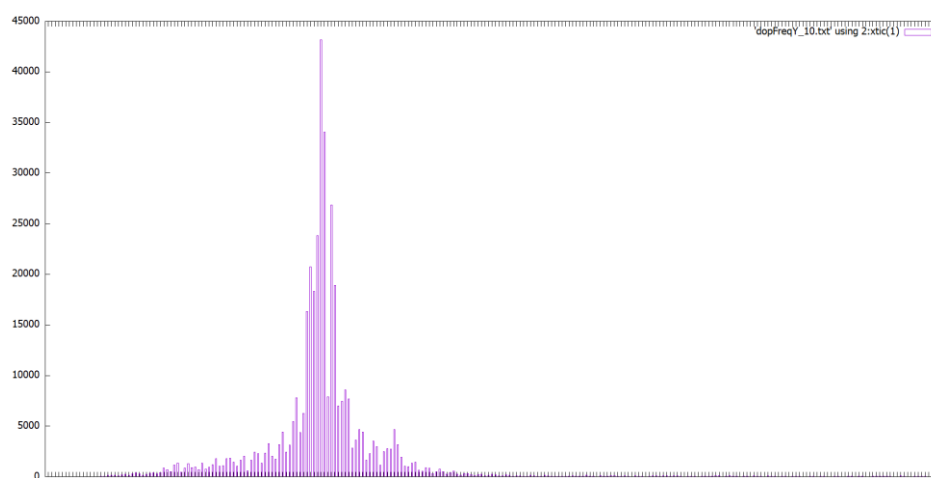


Рисунок 41 - Y10

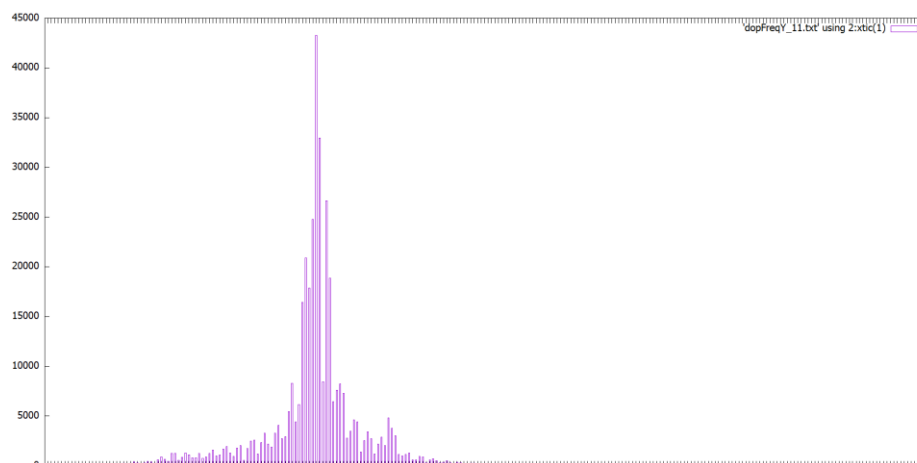


Рисунок 42 - Y11

Гистограммы одинаковые, а значит, можно опять же сделать вывод, что на тех позициях, которые остаются и исключаются находятся схожие значения компонент.

### 13.3 Энтропия

Были получены значения энтропии:

```
Subframes :
H(Y_00) = 5.53259
H(Y_01) = 5.544
H(Y_10) = 5.53308
H(Y_11) = 5.53857
```

Рисунок 43 - Значения энтропии

Энтропия так же доказывает справедливость вывода, сделанного ранее.

### 14 Выводы:

Была изучена структура BMP-изображения формата RGB24: были изучены заголовки, что помогло в дальнейшем извлечь такую важную информацию, как ширина и высота изображения.

Были исследованы корреляционные свойства компонент R, G, B, выявлено, что зависимость между ними достаточно высокая, особенно между компонентами B и G.

Для того, чтобы избежать высокой корреляции используется формат YCbCr, так как в данном формате меньшая корреляционная связь между компонентами, что так же позволяет обеспечить лучшее сжатие. Но формат YCbCr обладает и минусами: при восстановлении исходного изображения теряется часть информации. Для оценки потери информации и



качества восстановления используется PSNR (пиковое отношение сигнала к шуму), так значения PSNR возрастают при восстановлении исходного изображения из формата YCbCr.

Децимация (частичное исключение информации) позволяет увеличить степень сжатия (для оценки используется оценка энтропии), но теряется качество изображения. Так, в данной программной реализации наилучшим вариантом децимации оказалась децимация в 2 раза путем использования среднего арифметического, а наихудшим вариантом – децимация в 4 раза путем исключения строк и столбцов.

Анализ эффективности разностного кодирования показал, что он улучшает показатели кодирования и сжатия.

15 Листинг:

***Main.cpp***

```
#include <iostream>
#include "bmp.h"
#include "3.h"
#include "4.h"
#include "5-6.h"
#include "7.h"
#include "8-11.h"
#include "12-13.h"
#include "14-16.h"
#include "dop.h"

using namespace std;

int main() {
    BITMAPFILEHEADER bfh;
    BITMAPINFOHEADER bih;
    FILE* f1;
    f1 = fopen("Original.bmp", "rb");
    if (f1 == NULL)
    {
        cout << "ERROR";
    }
}
```

```

        return 0;
    }
    RGB** rgb = read_bmp(f1, &bfh, &bih);
    fclose(f1);
    int height = bih.biHeight;
    int width = bih.biWidth;

    /*
    set style data histograms
    plot 'dopFreqY_00.txt' using 2:xtic(1)
    */

    /*
    width=5
    bin(x, s) = s*int(x/s) + width/2
    set boxwidth width
    plot 'histogramRed.txt' u (bin($1,width)):(1.0) s f w boxes fs
    solid 0.5
    */

    Get_color get_color(&bfh, &bih, rgb, height, width);
    Corelation correlaion(height, width, rgb);
    Direct_conversion direct_conversion(&bfh, &bih, rgb, height,
width);
    Reverse_conversion reverse_conversion(&bfh, &bih,
direct_conversion.get_YCbCr(), height, width, rgb);
    Decimation decimation(direct_conversion.get_YCbCr(), height,
width, &bfh, &bih, rgb);
    Frequency frequency(rgb, height, width,
direct_conversion.get_YCbCr());
    DPCM dpcm(height, width, rgb, direct_conversion.get_YCbCr());
    Subframes(height, width, direct_conversion.get_YCbCr(),
&bfh, &bih);

    return 0;
}

```

3.h

```
#include <iostream>
```

```
#include "bmp.h"
```

```
using namespace std;
```

```
class Get_color {
```

```
private:
```

```
    RGB** rgb;
```

```
    RGB** new_rgb;
```

```
    BITMAPFILEHEADER* bfh;
```

```
    BITMAPINFOHEADER* bih;
```

```
    int height;
```

```
    int width;
```

```
public:
```

```
    Get_color(BITMAPFILEHEADER* bfh, BITMAPINFOHEADER* bih, RGB**  
color, int h, int w) {
```

```
        height = h;
```

```
        width = w;
```

```
        this->bfh = bfh;
```

```
        this->bih = bih;
```

```
        rgb = color;
```

```
        new_rgb = new RGB*[height];
```

```
        for (int i = 0; i < height; i++)
```

```
            new_rgb[i] = new RGB[width];
```

```
        cout << endl << "3";
```

```
        get_R();
```

```
        get_G();
```

```
        get_B();
```

```
    }
```

```
    ~Get_color() {
```

```
        for (int i = 0; i < height; i++)
```

```

        delete(new_rgb[i]);
    delete(new_rgb);
}

void get_R() {
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            new_rgb[i][j].B = 0;
            new_rgb[i][j].G = 0;
            new_rgb[i][j].R = rgb[i][j].R;
        }
    }

    FILE* file;
    file = fopen("3Red.bmp", "wb");
    write_bmp(file, new_rgb, bfh, bih, height, width);
    fclose(file);
}

```

```

void get_G() {
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            new_rgb[i][j].B = 0;
            new_rgb[i][j].R = 0;
            new_rgb[i][j].G = rgb[i][j].G;
        }
    }

    FILE* file;
    file = fopen("3Green.bmp", "wb");
    write_bmp(file, new_rgb, bfh, bih, height, width);
    fclose(file);
}

```

```

void get_B() {
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {

```

```

        new_rgb[i][j].R = 0;
        new_rgb[i][j].G = 0;
        new_rgb[i][j].B = rgb[i][j].B;
    }
}

FILE* file;
file = fopen("3Blue.bmp", "wb");
write_bmp(file, new_rgb, bfh, bih, height, width);
fclose(file);
}
};

```

4.h

```

#include <iostream>
#include <vector>
#include <fstream>
#include "bmp.h"

```

```

using namespace std;

```

```

class Corelation {
private:
    int height;
    int width;
    RGB** rgb;
    double M_R = 0;
    double M_G = 0;
    double M_B = 0;
    double sigma_R = 0;
    double sigma_G = 0;
    double sigma_B = 0;

```

```

public:

```

```

    Corelation(int h, int w, RGB** colors) {

```

```

height = h;
width = w;
rgb = colors;

cout << endl << "4" << endl;
average();
sigma();
r_GB();
r_RB();
r_RG();

/*      vector<double>          autocorrGreen1          =
Autocorrelation_func('G', -10);
        write_txt("4G-10.txt", autocorrGreen1, width);
        cout << "G-10" << endl;

        vector<double>          autocorrGreen2          =
Autocorrelation_func('G', -5);
        write_txt("4G-5.txt", autocorrGreen2, width);
        cout << "G-5" << endl;

        vector<double>          autocorrGreen3          =
Autocorrelation_func('G', 0);
        write_txt("4G0.txt", autocorrGreen3, width);
        cout << "G0" << endl;

        vector<double>          autocorrGreen4          =
Autocorrelation_func('G', 5);
        write_txt("4G5.txt", autocorrGreen4, width);
        cout << "G5" << endl;

        vector<double>          autocorrGreen5          =
Autocorrelation_func('G', 10);
        write_txt("4G10.txt", autocorrGreen5, width);

```



```

        cout << "G10" << endl;*/

        /*    vector<double>                autocorrBlue1                =
Autocorrelation_func('B', -10);
        write_txt("4B-10.txt", autocorrBlue1, width);
        cout << "B-10" << endl;

        vector<double>                autocorrBlue2                =
Autocorrelation_func('B', -5);
        write_txt("4B-5.txt", autocorrBlue2, width);
        cout << "B-5" << endl;

        vector<double>                autocorrBlue3                =
Autocorrelation_func('B', 0);
        write_txt("4B0.txt", autocorrBlue3, width);
        cout << "B0" << endl;

        vector<double>                autocorrBlue4                =
Autocorrelation_func('B', 5);
        write_txt("4B5.txt", autocorrBlue4, width);
        cout << "B5" << endl;

        vector<double>                autocorrBlue5                =
Autocorrelation_func('B', 10);
        write_txt("4B10.txt", autocorrBlue5, width);
        cout << "B10" << endl;*/

        //vector<double>                autocorrRed1                =
Autocorrelation_func('R', -10);
        //write_txt("4R-10.txt", autocorrRed1, width);
        //cout << "R-10" << endl;

        //vector<double>                autocorrRed2                =
Autocorrelation_func('R', -5);
        //write_txt("4R-5.txt", autocorrRed2, width);
        //cout << "R-5" << endl;

```

```

        //vector<double>          autocorrRed3          =
Autocorrelation_func('R', 0);
        //write_txt("4R0.txt", autocorrRed3, width);
        //cout << "R0" << endl;

        //vector<double>          autocorrRed4          =
Autocorrelation_func('R', 5);
        //write_txt("4R5.txt", autocorrRed4, width);
        //cout << "R5" << endl;

        //vector<double>          autocorrRed5          =
Autocorrelation_func('R', 10);
        //write_txt("4R10.txt", autocorrRed5, width);
        //cout << "R10" << endl;

    }

    void average() {
        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
                M_B += rgb[i][j].B;
                M_R += rgb[i][j].R;
                M_G += rgb[i][j].G;
            }
        }
        M_B /= (height * width);
        M_R /= (height * width);
        M_G /= (height * width);
    }

    void sigma() {
        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
                sigma_B += pow(rgb[i][j].B - M_B, 2);
                sigma_R += pow(rgb[i][j].R - M_R, 2);
                sigma_G += pow(rgb[i][j].G - M_G, 2);
            }
        }
        sigma_B = sqrt(sigma_B / (height * width - 1));
    }

```

```

        sigma_R = sqrt(sigma_R / (height * width - 1));
        sigma_G = sqrt(sigma_G / (height * width - 1));
    }

    void r_GB() {
        double r = 0;
        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
                r += (rgb[i][j].G - M_G) * (rgb[i][j].B - M_B);
            }
        }
        r /= (height * width * sigma_G * sigma_B);
        cout << "r_G,B = " << r << endl;
    }

    void r_RB() {
        double r = 0;
        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
                r += (rgb[i][j].R - M_R) * (rgb[i][j].B - M_B);
            }
        }
        r /= (height * width * sigma_R * sigma_B);
        cout << "r_R,B = " << r << endl;
    }

    void r_RG() {
        double r = 0;
        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
                r += (rgb[i][j].R - M_R) * (rgb[i][j].G - M_G);
            }
        }
        r /= (height * width * sigma_G * sigma_R);
        cout << "r_G,R = " << r << endl;
    }

```

```

    }

    vector<double> Autocorrelation_func(char letter, int y) {
        vector<double> array(width / 4);
        vector<vector<double>> color(height);
        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
                if (letter == 'R') {
                    color[i].push_back(rgb[i][j].R);
                    continue;
                }
                if (letter == 'G') {
                    color[i].push_back(rgb[i][j].G);
                    continue;
                }
                if (letter == 'B') {
                    color[i].push_back(rgb[i][j].B);
                    continue;
                }
            }
        }

        for (int x = -width / 4, counter = 0; x < width / 4; x
+= 2, counter++) {
            vector<vector<double>> firstSelection =
calculateFirstSelection(color, x, y);
            vector<vector<double>> secondSelection =
calculateSecondSelection(color, x, y);
            array[counter] = Correlation_func(firstSelection,
secondSelection);
        }
        return array;
    }

```

```

        vector<vector<double>>
calculateFirstSelection(vector<vector<double>>& array, int x, int
y) {

    int width = array.size();
    int height = array[0].size();
    vector<vector<double>> result(width);

    for (int i = 0; i < width; ++i) {
        result[i].resize(height);
    }

    if (y >= 0) {
        if (x >= 0) {
            for (int j = 0; j < width - x; ++j) {
                for (int i = 0; i < height - y; ++i) {
                    result[j][i] = array[j][i];
                }
            }
        }
        else {
            for (int j = -x; j < width; ++j) {
                for (int i = 0; i < height - y; ++i) {
                    result[j + x][i] = array[j][i];
                }
            }
        }
    }
    else {
        if (x >= 0) {
            for (int j = 0; j < width - x; ++j) {
                for (int i = -y; i < height; ++i) {
                    result[j][i + y] = array[j][i];
                }
            }
        }
    }
}

```

```

    }
    else {
        for (int j = -x; j < width; j++) {
            for (int i = -y; i < height; i++) {
                result[j + x][i + y] = array[j][i];
            }
        }
    }
}

return result;
}

```

```

vector<vector<double>>
calculateSecondSelection(vector<vector<double>>& array, int x,
int y) {
    int height = array[0].size();
    int width = array.size();
    vector<vector<double>> result(width);

    for (int i = 0; i < width; ++i) {
        result[i].resize(height);
    }

    if (y >= 0) {
        if (x >= 0) {
            for (int j = x; j < width; ++j) {
                for (int i = y; i < height; ++i) {
                    result[j - x][i - y] = array[j][i];
                }
            }
        }
        else {
            for (int j = 0; j < width + x; ++j) {
                for (int i = y; i < height; ++i) {
                    result[j][i - y] = array[j][i];
                }
            }
        }
    }
}

```



```

        }
    }
}
else {
    if (x >= 0) {
        for (int j = x; j < width; ++j) {
            for (int i = 0; i < height + y; ++i) {
                result[j - x][i] = array[j][i];
            }
        }
    }
    else {
        for (int j = 0; j < width + x; ++j) {
            for (int i = 0; i < height + y; ++i) {
                result[j][i] = array[j][i];
            }
        }
    }
}
return result;
}

```

```

double      Correlation_func(vector<vector<double>>      A,
vector<vector<double>> B) {
    double m1 = Get_expected_value(A);
    double m2 = Get_expected_value(B);

    double d1 = Get_dispersion(A);
    double d2 = Get_dispersion(B);

    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 = Get_expected_value(A) / (d1 * d2);
return res;
}

```

```

double Get_expected_value(vector<vector<double>> rgb) {
    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 Get_dispersion(vector<vector<double>> rgb) {
    double res = 0;
    double WH = (double)width * (double)height;
    double m = Get_expected_value(rgb);

    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);
    }

    void write_txt(string filename, vector<double> result, int
width) {
        ofstream fout;
        fout.open(filename);
        int count = 0;
        if (fout.is_open()) {
            int j = 0;
            for (int i = -width / 4; i < width / 4; i += 2) {
                fout << i << " " << result[j] << "\n";
                j++;
            }
        }
        fout.close();
    }
};

```

5-6.h

```
#include <iostream>
```

```
#include "bmp.h"
```

```
using namespace std;
```

```
class Direct_conversion {
```

```
private:
```

```
    RGB** rgb;
```

```
    RGB** new_rgb;
```

```
    BITMAPFILEHEADER* bfh;
```

```

    BITMAPINFOHEADER* bih;

    int height;

    int width;

    YCbCr** ycbcr;

    double M_Y = 0;

    double M_Cb = 0;

    double M_Cr = 0;

    double sigma_Y = 0;

    double sigma_Cb = 0;

    double sigma_Cr = 0;

public:

    Direct_conversion(BITMAPFILEHEADER* bfh, BITMAPINFOHEADER*
    bih, RGB** color, int h, int w) {

        height = h;

        width = w;

        this->bfh = bfh;

        this->bih = bih;

        rgb = color;

        new_rgb = new RGB * [height];

        for (int i = 0; i < height; i++)

            new_rgb[i] = new RGB[width];

        cout << endl << "5-6" << endl;

        get_direct_conversion();

        get_Y();

        get_Cb();

        get_Cr();


        average();

        sigma();

        r_YCb();

        r_YCr();

        r_CbCr();

    }

```

```

~Direct_conversion() {
    for (int i = 0; i < height; i++)
        delete(new_rgb[i]);
    delete(new_rgb);
}

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

void get_Y() {
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            new_rgb[i][j].B = ycbcr[i][j].Y;
            new_rgb[i][j].G = ycbcr[i][j].Y;
            new_rgb[i][j].R = ycbcr[i][j].Y;
        }
    }

    FILE* file;
    file = fopen("6Y.bmp", "wb");
    write_bmp(file, new_rgb, bfh, bih, height, width);
    fclose(file);
}

void get_Cb() {

```

```

        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
                new_rgb[i][j].B = ycbcr[i][j].Cb;
                new_rgb[i][j].G = ycbcr[i][j].Cb;
                new_rgb[i][j].R = ycbcr[i][j].Cb;
            }
        }
        FILE* file;
        file = fopen("6Cb.bmp", "wb");
        write_bmp(file, new_rgb, bfh, bih, height, width);
        fclose(file);
    }

    void get_Cr() {
        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
                new_rgb[i][j].B = ycbcr[i][j].Cr;
                new_rgb[i][j].G = ycbcr[i][j].Cr;
                new_rgb[i][j].R = ycbcr[i][j].Cr;
            }
        }
        FILE* file;
        file = fopen("6Cr.bmp", "wb");
        write_bmp(file, new_rgb, bfh, bih, height, width);
        fclose(file);
    }

    YCbCr** get_YCbCr() {
        return ycbcr;
    }

    void average() {
        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
                M_Y += ycbcr[i][j].Y;
            }
        }
    }

```

```

        M_Cb += ycbcr[i][j].Cb;
        M_Cr += ycbcr[i][j].Cr;
    }

}

M_Y /= (height * width);
M_Cb /= (height * width);
M_Cr /= (height * width);
}

void sigma() {
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            sigma_Y += pow(ycbcr[i][j].Y - M_Y, 2);
            sigma_Cb += pow(ycbcr[i][j].Cb - M_Cb, 2);
            sigma_Cr += pow(ycbcr[i][j].Cr - M_Cr, 2);
        }
    }

    sigma_Y = sqrt(sigma_Y / (height * width - 1));
    sigma_Cb = sqrt(sigma_Cb / (height * width - 1));
    sigma_Cr = sqrt(sigma_Cr / (height * width - 1));
}

void r_YCb() {
    double r = 0;
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            r += (ycbcr[i][j].Y - M_Y) * (ycbcr[i][j].Cb -
M_Cb);
        }
    }

    r /= (height * width * sigma_Cb * sigma_Y);
    cout << "r_Y,Cb = " << r << endl;
}

void r_YCr() {
    double r = 0;

```

```

        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
                r += (ycbcr[i][j].Y - M_Y) * (ycbcr[i][j].Cr -
M_Cr);
            }
        }
        r /= (height * width * sigma_Cr * sigma_Y);
        cout << "r_Y,Cr = " << r << endl;
    }

    void r_CbCr() {
        double r = 0;
        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
                r += (ycbcr[i][j].Cr - M_Cr) * (ycbcr[i][j].Cb
- M_Cb);
            }
        }
        r /= (height * width * sigma_Cb * sigma_Cr);
        cout << "r_Cr,Cb = " << r << endl;
    }

};

```

7.h

```

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

```

```

class Reverse_conversion {

```



private:

```
    RGB** new_rgb;
    RGB** rgb;
    BITMAPFILEHEADER* bfh;
    BITMAPINFOHEADER* bih;
    int height;
    int width;
    YCbCr** ycbcr;
    unsigned char max_R = 0;
    unsigned char max_G = 0;
    unsigned char max_B = 0;
    unsigned char min_R = 255;
    unsigned char min_G = 255;
    unsigned char min_B = 255;
```

public:

```
    Reverse_conversion(BITMAPFILEHEADER* bfh, BITMAPINFOHEADER*
    bih, YCbCr** y, int h, int w, RGB** colors) {
        height = h;
        width = w;
        this->bfh = bfh;
        this->bih = bih;
        ycbcr = y;
        rgb = colors;
        new_rgb = new RGB * [height];
        for (int i = 0; i < height; i++) {
            new_rgb[i] = new RGB[width];
            for (int j = 0; j < width; j++) {
                new_rgb[i][j].B = 0;
                new_rgb[i][j].R = 0;
                new_rgb[i][j].G = 0;
            }
        }
        cout << endl << "7" << endl;
```

```

        find_min_max();
        reverse_conversion();
        PSNR();
    }

~Reverse_conversion() {
    for (int i = 0; i < height; i++)
        delete(new_rgb[i]);
    delete(new_rgb);
}

void find_min_max() {
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            if (rgb[i][j].R > max_R)
                max_R = rgb[i][j].R;
            if (rgb[i][j].G > max_G)
                max_G = rgb[i][j].G;
            if (rgb[i][j].B > max_B)
                max_B = rgb[i][j].B;
            if (rgb[i][j].R < min_R)
                min_R = rgb[i][j].R;
            if (rgb[i][j].G < min_G)
                min_G = rgb[i][j].G;
            if (rgb[i][j].B < min_B)
                min_B = rgb[i][j].B;
        }
    }
}

unsigned char clipping_R(double color) {
    if (color > (double)max_R)
        return max_R;
    if (color < (double)min_R)
        return min_R;
    return (unsigned char)(color);
}

```

```

unsigned char clipping_G(double color) {
    if (color > (double)max_G)
        return max_G;
    if (color < (double)min_G)
        return min_G;
    return (unsigned char)(color);
}

unsigned char clipping_B(double color) {
    if (color > max_B)
        return max_B;
    if (color < (double)min_G)
        return min_B;
    return (unsigned char)(color);
}

void reverse_conversion() {
    for (int i = 0; i < height; i++) {
        double tmp = 0;
        for (int j = 0; j < width; j++) {
            tmp = (ycbcr[i][j].Y - 0.714 * (ycbcr[i][j].Cr
- 128) - 0.334 * (ycbcr[i][j].Cb - 128));
            new_rgb[i][j].G = clipping_G(tmp);
            tmp = ycbcr[i][j].Y + 1.402 * (ycbcr[i][j].Cr
- 128);
            new_rgb[i][j].R = clipping_R(tmp);
            tmp = ycbcr[i][j].Y + 1.772 * (ycbcr[i][j].Cb
- 128);
            new_rgb[i][j].B = clipping_B(tmp);
        }
    }

    FILE* file;
    file = fopen("7Recovered.bmp", "wb");
    write_bmp(file, new_rgb, bfh, bih, height, width);
    fclose(file);
}

void PSNR() {

```

```

        double tmp = width * height * pow(256 - 1, 2);
        double PSNR_R = 0;
        double PSNR_G = 0;
        double PSNR_B = 0;
        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
                PSNR_B += pow((rgb[i][j].B - new_rgb[i][j].B),
2);
                PSNR_G += pow((rgb[i][j].G - new_rgb[i][j].G),
2);
                PSNR_R += pow((rgb[i][j].R - new_rgb[i][j].R),
2);
            }
        }
        PSNR_B = 10 * log10(tmp / PSNR_B);
        PSNR_G = 10 * log10(tmp / PSNR_G);
        PSNR_R = 10 * log10(tmp / PSNR_R);
        cout << "PSNR_B = " << PSNR_B << endl;
        cout << "PSNR_G = " << PSNR_G << endl;
        cout << "PSNR_R = " << PSNR_R << endl;
    }
};

```

8-11.h

```

#include <iostream>
#include <vector>
#include "bmp.h"

```

```

using namespace std;

```

```

class Decimation {
private:
    int height;

```

```

int width;
int new_height_2;
int new_width_2;
int new_height_4;
int new_width_4;
YCbCr** ycbcr;
double** decimation_8a_Cb;
double** decimation_8b_Cb;
double** decimation_8a_Cr;
double** decimation_8b_Cr;
YCbCr** ycbcr_new_a;
YCbCr** ycbcr_new_b;
RGB** new_rgb_a;
RGB** new_rgb_b;
double** decimation_11a_Cb;
double** decimation_11b_Cb;
double** decimation_11a_Cr;
double** decimation_11b_Cr;
RGB** rgb;

BITMAPFILEHEADER* bfh;
BITMAPINFOHEADER* bih;
unsigned char min_R = 255;
unsigned char min_G = 255;
unsigned char min_B = 255;
unsigned char max_R = 0;
unsigned char max_G = 0;
unsigned char max_B = 0;

public:

    Decimation(YCbCr** y, int h, int w, BITMAPFILEHEADER* Bfh,
        BITMAPINFOHEADER* Bih, RGB** Rgb) {
        height = h;
        width = w;
        ycbcr = y;
        new_height_2 = height / 2;

```

```

new_width_2 = width / 2;
decimation_8a_Cb = new double* [new_height_2];
decimation_8a_Cr = new double* [new_height_2];
decimation_8b_Cb = new double* [new_height_2];
decimation_8b_Cr = new double* [new_height_2];
for (int i = 0; i < new_height_2; i++) {
    decimation_8a_Cb[i] = new double[new_width_2];
    decimation_8a_Cr[i] = new double[new_width_2];
    decimation_8b_Cb[i] = new double[new_width_2];
    decimation_8b_Cr[i] = new double[new_width_2];
}
new_height_4 = height / 4;
new_width_4 = width / 4;
decimation_11a_Cb = new double* [new_height_4];
decimation_11a_Cr = new double* [new_height_4];
decimation_11b_Cb = new double* [new_height_4];
decimation_11b_Cr = new double* [new_height_4];
for (int i = 0; i < new_height_4; i++) {
    decimation_11a_Cb[i] = new double[new_width_4];
    decimation_11a_Cr[i] = new double[new_width_4];
    decimation_11b_Cb[i] = new double[new_width_4];
    decimation_11b_Cr[i] = new double[new_width_4];
}
new_rgb_a = new RGB * [height];
new_rgb_b = new RGB * [height];
for (int i = 0; i < height; i++) {
    new_rgb_a[i] = new RGB[width];
    new_rgb_b[i] = new RGB[width];
}
ycbcr_new_a = new YCbCr * [height];
ycbcr_new_b = new YCbCr * [height];
for (int i = 0; i < height; i++) {
    ycbcr_new_a[i] = new YCbCr[width];
    ycbcr_new_b[i] = new YCbCr[width];
}

```

```

        for (int j = 0; j < width; j++)
        {
            ybcr_new_a[i][j].Y = ybcr[i][j].Y;
            ybcr_new_b[i][j].Y = ybcr[i][j].Y;
        }

    }

    bfh = Bfh;
    bih = Bih;
    rgb = Rgb;
    cout << endl << "8-11";
    decimation_8a();
    decimation_8b();
    recover_size_8();
    find_min_max();
    reverse_conversion(ybcr_new_a, new_rgb_a,
"9Recovered_decimation1.bmp");
    reverse_conversion(ybcr_new_b, new_rgb_b,
"9Recovered_decimation2.bmp");
    cout << endl << "Decimation 8a";
    PSNR_YCbCr(ybcr_new_a);
    PSNR_RGB(new_rgb_a);
    cout << endl << "Decimation 8b";
    PSNR_YCbCr(ybcr_new_b);
    PSNR_RGB(new_rgb_b);

    decimation_11a();
    decimation_11b();
    recover_size_11();
    reverse_conversion(ybcr_new_a, new_rgb_a,
"11Recovered_decimation1.bmp");
    reverse_conversion(ybcr_new_b, new_rgb_b,
"11Recovered_decimation2.bmp");
    cout << endl << "Decimation 11a";
    PSNR_YCbCr(ybcr_new_a);

```

```

        PSNR_RGB(new_rgb_a);
        cout << endl << "Decimation 11b";
        PSNR_YCbCr(ycbcr_new_b);
        PSNR_RGB(new_rgb_b);
    }

    void decimation_8a() {
        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
                if (i % 2 == 0 && j % 2 == 0) {
                    decimation_8a_Cb[i / 2][j / 2] =
ycbcr[i][j].Cb;
                    decimation_8a_Cr[i / 2][j / 2] =
ycbcr[i][j].Cr;
                }
            }
        }
    }

    void decimation_11a() {
        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
                if (i % 4 == 0 && j % 4 == 0) {
                    decimation_11a_Cb[i / 4][j / 4] =
ycbcr[i][j].Cb;
                    decimation_11a_Cr[i / 4][j / 4] =
ycbcr[i][j].Cr;
                }
            }
        }
    }

    void decimation_8b() {
        double sum_Cb = 0;
        double sum_Cr = 0;
        for (int i = 0; i < height; i += 2) {
            for (int j = 0; j < width; j += 2) {

```



```

        sum_Cb += ycbcr[i][j].Cb + ycbcr[i + 1][j].Cb
+ ycbcr[i][j + 1].Cb + ycbcr[i + 1][j + 1].Cb;
        sum_Cr += ycbcr[i][j].Cr + ycbcr[i + 1][j].Cr
+ ycbcr[i][j + 1].Cr + ycbcr[i + 1][j + 1].Cr;
        decimation_8b_Cb[i / 2][j / 2] = sum_Cb / 4;
        decimation_8b_Cr[i / 2][j / 2] = sum_Cr / 4;
        sum_Cb = 0;
        sum_Cr = 0;
    }
}

```

```

void decimation_11b() {
    double sum_Cb = 0;
    double sum_Cr = 0;
    for (int i = 0; i < height; i += 4) {
        for (int j = 0; j < width; j += 4) {
            for (int a = 0; a < 4; a++) {
                for (int b = 0; b < 4; b++) {
                    sum_Cb += ycbcr[i + a][j+b].Cb;
                    sum_Cr += ycbcr[i + a][j+b].Cr;
                }
            }
            decimation_11b_Cb[i / 4][j / 4] = sum_Cb / 16;
            decimation_11b_Cr[i / 4][j / 4] = sum_Cr / 16;
            sum_Cb = 0;
            sum_Cr = 0;
        }
    }
}

```

```

void recover_size_8() {
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {

```

```

        ycbcr_new_a[i][j].Cb    =    decimation_8a_Cb[i
/2][j / 2];
        ycbcr_new_a[i][j].Cr    =    decimation_8a_Cr[i /
2][j / 2];
        ycbcr_new_b[i][j].Cb    =    decimation_8b_Cb[i /
2][j / 2];
        ycbcr_new_b[i][j].Cr    =    decimation_8b_Cr[i /
2][j / 2];
    }
}

void recover_size_11() {
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            ycbcr_new_a[i][j].Cb = decimation_11a_Cb[i /
4][j / 4];
            ycbcr_new_a[i][j].Cr = decimation_11a_Cr[i /
4][j / 4];
            ycbcr_new_b[i][j].Cb = decimation_11b_Cb[i /
4][j / 4];
            ycbcr_new_b[i][j].Cr = decimation_11b_Cr[i /
4][j / 4];
        }
    }

}

void reverse_conversion(YCbCr** ycbcr, RGB** new_rgb, const
char* filename) {
    for (int i = 0; i < height; i++) {
        double tmp = 0;
        for (int j = 0; j < width; j++) {
            tmp = (ycbcr[i][j].Y - 0.714 * (ycbcr[i][j].Cr
- 128) - 0.334 * (ycbcr[i][j].Cb - 128));
            new_rgb[i][j].G = clipping_G(tmp);
            tmp = ycbcr[i][j].Y + 1.402 * (ycbcr[i][j].Cr
- 128);

```

```

        new_rgb[i][j].R = clipping_R(tmp);
        tmp = ycbcr[i][j].Y + 1.772 * (ycbcr[i][j].Cb
- 128);

        new_rgb[i][j].B = clipping_B(tmp);
    }
}

FILE* file;
file = fopen(filename, "wb");
write_bmp(file, new_rgb, bfh, bih, height, width);
fclose(file);
}

void find_min_max() {
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            if (rgb[i][j].R > max_R)
                max_R = rgb[i][j].R;
            if (rgb[i][j].G > max_G)
                max_G = rgb[i][j].G;
            if (rgb[i][j].B > max_B)
                max_B = rgb[i][j].B;
            if (rgb[i][j].R < min_R)
                min_R = rgb[i][j].R;
            if (rgb[i][j].G < min_G)
                min_G = rgb[i][j].G;
            if (rgb[i][j].B < min_B)
                min_B = rgb[i][j].B;
        }
    }
}

unsigned char clipping_R(double color) {
    if (color > (double)max_R)
        return max_R;
    if (color < (double)min_R)
        return min_R;
}

```

```

        return (unsigned char)(color);
    }
    unsigned char clipping_G(double color) {
        if (color > (double)max_G)
            return max_G;
        if (color < (double)min_G)
            return min_G;
        return (unsigned char)(color);
    }
    unsigned char clipping_B(double color) {
        if (color > (double)max_B)
            return max_B;
        if (color < (double)min_B)
            return min_B;
        return (unsigned char)(color);
    }

    void PSNR_RGB(RGB** new_rgb) {
        double tmp = width * height * pow(256 - 1, 2);
        double PSNR_R = 0;
        double PSNR_G = 0;
        double PSNR_B = 0;
        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
                PSNR_B += pow((rgb[i][j].B - new_rgb[i][j].B),
2);
                PSNR_G += pow((rgb[i][j].G - new_rgb[i][j].G),
2);
                PSNR_R += pow((rgb[i][j].R - new_rgb[i][j].R),
2);
            }
        }
        PSNR_B = 10 * log10(tmp / PSNR_B);
        PSNR_G = 10 * log10(tmp / PSNR_G);
        PSNR_R = 10 * log10(tmp / PSNR_R);
    }

```

```

        cout << endl << "PSNR_B = " << PSNR_B << endl;
        cout << "PSNR_G = " << PSNR_G << endl;
        cout << "PSNR_R = " << PSNR_R << endl;
    }

    void PSNR_YCbCr(YCbCr** ycbcr_new) {
        double tmp = width * height * pow(256 - 1, 2);
        double PSNR_Cb = 0;
        double PSNR_Cr = 0;
        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
                PSNR_Cb += pow((ycbcr[i][j].Cb -
ycbcr_new[i][j].Cb), 2);
                PSNR_Cr += pow((ycbcr[i][j].Cr -
ycbcr_new[i][j].Cr), 2);
            }
        }

        PSNR_Cb = 10 * log10(tmp / PSNR_Cb);
        PSNR_Cr = 10 * log10(tmp / PSNR_Cr);
        cout << endl << "PSNR_Cb = " << PSNR_Cb << endl;
        cout << "PSNR_Cr = " << PSNR_Cr << endl;
    }

};

```

12-13.h

```

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

```

```

using namespace std;

```

```

class Frequency {

```

```

private:
    int height;
    int width;
    RGB** rgb;
    YCbCr** ycbcr;
    double* freq;
    double H = 0;

public:
    Frequency(RGB** RGB, int h, int w, YCbCr** YCbCr) {
        rgb = RGB;
        ycbcr = YCbCr;
        height = h;
        width = w;
        freq = new double[256];
        cout << endl << "12-13" << endl;
        clear();
        get_RGB('R', "12R.txt");
        entropy(string("R"));
        clear();
        get_RGB('G', "12G.txt");
        entropy(string("G"));
        clear();
        get_RGB('B', "12B.txt");
        entropy(string("B"));
        clear();
        get_YCbCr('Y', "12Y.txt");
        entropy(string("Y"));
        clear();
        get_YCbCr('B', "12Cb.txt");
        entropy(string("Cb"));
        clear();
        get_YCbCr('R', "12Cr.txt");
        entropy(string("Cr"));
    }

```

```

void clear() {
    for (int i = 0; i < 256; i++)
        freq[i] = 0;
    H = 0;
}

void get_RGB(char letter, const char* filename) {
    ofstream out;
    out.open(filename);
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            if (letter == 'R')

                freq[rgb[i][j].R]++;
            if (letter == 'G')

                freq[rgb[i][j].G]++;
            if (letter == 'B')

                freq[rgb[i][j].B]++;
        }
    }
    for (int i = 0; i < 256; i++) {
        out << i << " " << freq[i] << endl;
    }
    out.close();
}

void get_YCbCr(char letter, const char* filename) {
    ofstream out;
    out.open(filename);
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            if (letter == 'Y')

                freq[(int)ycbcr[i][j].Y]++;

```

```

        if (letter == 'B')

            freq[(int)ycbcr[i][j].Cb]++;
            if (letter == 'R')

                freq[(int)ycbcr[i][j].Cr]++;
        }
    }
    for (int i = 0; i < 256; i++) {
        out << i << " " << freq[i] << endl;
    }
    out.close();
}

void entropy(string letter) {
    for (int i = 0; i < 256; i++) {
        double p = (double)freq[i] / (height * width);
        if (p != 0) {
            H += p * log2(p);
        }
    }
    H *= (-1);
    cout << "H(" << letter << ") = " << H << endl;

}

};

```

14-16.h

```

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

```

```

using namespace std;

```



```

class DPCM {
private:
    RGB** rgb;
    int height;
    int width;
    YCbCr** ycbcr;
    RGB** d_rgb;
    YCbCr** d_ycbcr;
    double* freq;
    double H = 0;

public:
    DPCM(int h, int w, RGB** r, YCbCr** y) {
        height = h;
        width = w;
        rgb = r;
        ycbcr = y;
        d_rgb = new RGB * [height];
        d_ycbcr = new YCbCr * [height];
        for (int i = 0; i < height; i++) {
            d_rgb[i] = new RGB[width];
            d_ycbcr[i] = new YCbCr[width];
        }
        freq = new double[256];
        cout << endl << "14-16";
        cout << endl << "DPCM_left" << endl;
        modulation_left_RGB();
        cout << endl;
        modulation_left_YCbCr();
        cout << endl << "DPCM_right" << endl;
        modulation_up_RGB();
        cout << endl;
        modulation_up_YCbCr();
        cout << endl << "DPCM_up_left" << endl;
    }
};

```

```

        modulation_up_left_RGB();
        cout << endl;
        modulation_up_left_YCbCr();
        cout << endl << "DPCM_average" << endl;
        modulation_average_RGB();
        cout << endl;
        modulation_average_YCbCr();
    }

    ~DPCM() {
        delete freq;
        for (int i = 0; i < height; i++) {
            delete(ycbcr[i]);
            delete(rgb[i]);
        }
        delete(rgb);
        delete(ycbcr);
    }

    void clear() {
        for (int i = 0; i < 256; i++)
            freq[i] = 0;
        H = 0;
    }

    void modulation_left_RGB() {
        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
                if (i == 0 || j == 0) {
                    d_rgb[i][j].R = rgb[i][j].R;
                    d_rgb[i][j].G = rgb[i][j].G;
                    d_rgb[i][j].B = rgb[i][j].B;
                }
                else {

```

```

        d_rgb[i][j].R = rgb[i][j].R - rgb[i][j -
1].R;

        d_rgb[i][j].G = rgb[i][j].G - rgb[i][j -
1].G;

        d_rgb[i][j].B = rgb[i][j].B - rgb[i][j -
1].B;

    }

}

clear();
get_RGB('R', "14R1.txt");
entropy(string("R"));

clear();
get_RGB('G', "14G1.txt");
entropy(string("G"));

clear();
get_RGB('B', "14B1.txt");
entropy(string("B"));

}

void modulation_left_YCbCr() {
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            if (i == 0 || j == 0) {
                d_ycbcr[i][j].Y = ycbcr[i][j].Y;
                d_ycbcr[i][j].Cb = ycbcr[i][j].Cb;
                d_ycbcr[i][j].Cr = ycbcr[i][j].Cr;
            }
            else {
                d_ycbcr[i][j].Y = ycbcr[i][j].Y -
ycbcr[i][j - 1].Y;
                d_ycbcr[i][j].Cb = ycbcr[i][j].Cb -
ycbcr[i][j - 1].Cb;
                d_ycbcr[i][j].Cr = ycbcr[i][j].Cr -
ycbcr[i][j - 1].Cr;
            }
        }
    }
}

```

```

    }
    clear();
    get_YCbCr('Y', "14Y1.txt");
    entropy(string("Y"));
    clear();
    get_YCbCr('B', "14Cb1.txt");
    entropy(string("Cb"));
    clear();
    get_YCbCr('R', "14Cr1.txt");
    entropy(string("Cr"));
}

void modulation_up_RGB() {
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            if (i == 0 || j == 0) {
                d_rgb[i][j].R = rgb[i][j].R;
                d_rgb[i][j].G = rgb[i][j].G;
                d_rgb[i][j].B = rgb[i][j].B;
            }
            else {
                d_rgb[i][j].R = rgb[i][j].R - rgb[i -
1][j].R;
                d_rgb[i][j].G = rgb[i][j].G - rgb[i -
1][j].G;
                d_rgb[i][j].B = rgb[i][j].B - rgb[i -
1][j].B;
            }
        }
    }
    clear();
    get_RGB('R', "14R2.txt");
    entropy(string("R"));
    clear();
    get_RGB('G', "14G2.txt");

```

```

        entropy(string("G"));
        clear();
        get_RGB('B', "14B2.txt");
        entropy(string("B"));
    }

    void modulation_up_YCbCr() {
        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
                if (i == 0 || j == 0) {
                    d_ycbcr[i][j].Y = ycbcr[i][j].Y;
                    d_ycbcr[i][j].Cb = ycbcr[i][j].Cb;
                    d_ycbcr[i][j].Cr = ycbcr[i][j].Cr;
                }
                else {
                    d_ycbcr[i][j].Y = ycbcr[i][j].Y - ycbcr[i
- 1][j].Y;
                    d_ycbcr[i][j].Cb = ycbcr[i][j].Cb -
ycbcr[i - 1][j].Cb;
                    d_ycbcr[i][j].Cr = ycbcr[i][j].Cr -
ycbcr[i - 1][j].Cr;
                }
            }
        }
        clear();
        get_YCbCr('Y', "14Y2.txt");
        entropy(string("Y"));
        clear();
        get_YCbCr('B', "14Cb2.txt");
        entropy(string("Cb"));
        clear();
        get_YCbCr('R', "14Cr2.txt");
        entropy(string("Cr"));
    }

    void modulation_up_left_RGB() {

```

```

        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
                if (i == 0 || j == 0) {
                    d_rgb[i][j].R = rgb[i][j].R;
                    d_rgb[i][j].G = rgb[i][j].G;
                    d_rgb[i][j].B = rgb[i][j].B;
                }
                else {
                    d_rgb[i][j].R = rgb[i][j].R - rgb[i -
1][j - 1].R;
                    d_rgb[i][j].G = rgb[i][j].G - rgb[i -
1][j - 1].G;
                    d_rgb[i][j].B = rgb[i][j].B - rgb[i -
1][j - 1].B;
                }
            }
        }
        clear();
        get_RGB('R', "14R3.txt");
        entropy(string("R"));
        clear();
        get_RGB('G', "14G3.txt");
        entropy(string("G"));
        clear();
        get_RGB('B', "14B3.txt");
        entropy(string("B"));
    }
    void modulation_up_left_YCbCr() {
        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
                if (i == 0 || j == 0) {
                    d_ycbcr[i][j].Y = ycbcr[i][j].Y;
                    d_ycbcr[i][j].Cb = ycbcr[i][j].Cb;
                    d_ycbcr[i][j].Cr = ycbcr[i][j].Cr;
                }
            }
        }
    }
}

```

```

        else {
            d_ycbcr[i][j].Y = ycbcr[i][j].Y - ycbcr[i
- 1][j - 1].Y;
            d_ycbcr[i][j].Cb = ycbcr[i][j].Cb -
ycbcr[i - 1][j - 1].Cb;
            d_ycbcr[i][j].Cr = ycbcr[i][j].Cr -
ycbcr[i - 1][j - 1].Cr;
        }
    }
}

clear();
get_YCbCr('Y', "14Y3.txt");
entropy(string("Y"));
clear();
get_YCbCr('B', "14Cb3.txt");
entropy(string("Cb"));
clear();
get_YCbCr('R', "14Cr3.txt");
entropy(string("Cr"));
}

```

```

void modulation_average_RGB() {
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            if (i == 0 || j == 0) {
                d_rgb[i][j].R = rgb[i][j].R;
                d_rgb[i][j].G = rgb[i][j].G;
                d_rgb[i][j].B = rgb[i][j].B;
            }
            else {
                d_rgb[i][j].R = rgb[i][j].R -
(double) (rgb[i - 1][j - 1].R + rgb[i - 1][j].R + rgb[i][j - 1].R)
/ 3;
                d_rgb[i][j].G = rgb[i][j].G -
(double) (rgb[i - 1][j - 1].G + rgb[i - 1][j].G + rgb[i][j - 1].G)
/ 3;
            }
        }
    }
}

```

```

        d_rgb[i][j].B = rgb[i][j].B -
(double)(rgb[i - 1][j - 1].B + rgb[i - 1][j].B + rgb[i][j - 1].B)
/ 3;

    }

}

clear();
get_RGB('R', "14R4.txt");
entropy(string("R"));
clear();
get_RGB('G', "14G4.txt");
entropy(string("G"));
clear();
get_RGB('B', "14B4.txt");
entropy(string("B"));
}

void modulation_average_YCbCr() {
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            if (i == 0 || j == 0) {
                d_ycbcr[i][j].Y = ycbcr[i][j].Y;
                d_ycbcr[i][j].Cb = ycbcr[i][j].Cb;
                d_ycbcr[i][j].Cr = ycbcr[i][j].Cr;
            }
            else {
                d_ycbcr[i][j].Y = ycbcr[i][j].Y -
(double)(ycbcr[i - 1][j - 1].Y + ycbcr[i][j - 1].Y + ycbcr[i -
1][j].Y) / 3;

                d_ycbcr[i][j].Cb = ycbcr[i][j].Cb -
(double)(ycbcr[i - 1][j - 1].Cb + ycbcr[i][j - 1].Cb + ycbcr[i -
1][j].Cb) / 3;

                d_ycbcr[i][j].Cr = ycbcr[i][j].Cr -
(double)(ycbcr[i - 1][j - 1].Cr + ycbcr[i][j - 1].Cr + ycbcr[i -
1][j].Cr) / 3;
            }
        }
    }
}

```



```

        clear();
        get_YCbCr('Y', "14Y4.txt");
        entropy(string("Y"));
        clear();
        get_YCbCr('B', "14Cb4.txt");
        entropy(string("Cb"));
        clear();
        get_YCbCr('R', "14Cr4.txt");
        entropy(string("Cr"));
    }

void get_RGB(char letter, const char* filename) {
    ofstream out;
    out.open(filename);
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            if (letter == 'R') {
                freq[(d_rgb[i][j].R + 256) % 256]++;
                out << (int)d_rgb[i][j].R << endl;
                continue;
            }
            if (letter == 'G') {
                freq[(d_rgb[i][j].G + 256) % 256]++;
                out << (int)d_rgb[i][j].G << endl;
                continue;
            }
            if (letter == 'B') {
                freq[(d_rgb[i][j].B + 256) % 256]++;
                out << (int) d_rgb[i][j].B << endl;
                continue;
            }
        }
    }
}

```

```

    }
    out.close();
}

void get_YCbCr(char letter, const char* filename) {
    ofstream out;
    out.open(filename);
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            if (letter == 'Y') {
                freq[((int)d_ycbcr[i][j].Y + 256) %
256]++;

                out << d_ycbcr[i][j].Y << endl;
                continue;
            }
            if (letter == 'B') {
                freq[((int)d_ycbcr[i][j].Cb + 256) %
256]++;

                out << d_ycbcr[i][j].Cb << endl;
                continue;
            }
            if (letter == 'R') {
                freq[((int)d_ycbcr[i][j].Cr + 256) %
256]++;

                out << d_ycbcr[i][j].Cr << endl;
                continue;
            }
        }
    }
    out.close();
}

void entropy(string letter) {
    for (int i = 0; i < 256; i++) {
        double p = (double)freq[i] / (height * width);
        if (p != 0) {
            H += p * log2(p);

```

```

        }

    }

    H *= (-1);

    cout << "H(" << letter << ") = " << H << endl;

}

};

```

Dop.h

```

#include <iostream>
#include <vector>
#include <fstream>
#include "bmp.h"

```

```

using namespace std;

```

```

class Subframes {
private:

```

```

    int height;
    int width;
    YCbCr** ycbcr;
    RGB** new_rgb;
    double H = 0;
    double* freq = new double[256];
    BITMAPFILEHEADER* bfh;
    BITMAPINFOHEADER* bih;

```

```

public:

```

```

    Subframes(int h, int w, YCbCr** y, BITMAPFILEHEADER* bf,
    BITMAPINFOHEADER* bi) {
        height = h;
        width = w;

```

```

        ycbcr = y;
        bfh = bf;
        bih = bi;
        new_rgb = new RGB * [height / 2];
        for (int i = 0; i < height / 2; i++) {
            new_rgb[i] = new RGB[width / 2];
        }
        cout << endl << "Subframes:" << endl;
        Y_00();
        Y_01();
        Y_10();
        Y_11();
        cout << endl;
    }

    void Y_00() {
        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
                if (i % 2 == 0 && j % 2 == 0) {
                    new_rgb[i / 2][j / 2].R = ycbcr[i][j].Y;
                    new_rgb[i / 2][j / 2].G = ycbcr[i][j].Y;
                    new_rgb[i / 2][j / 2].B = ycbcr[i][j].Y;
                }
            }
        }

        FILE* file;
        file = fopen("dopY_00.bmp", "wb");
        write_bmp(file, new_rgb, bfh, bih, height / 2, width /
2);

        fclose(file);
        clear();
        get_YCbCr("dopFreqY_00.txt");
        entropy("Y_00");
    }

```

```

void Y_01() {

    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            if (i % 2 == 0 && j % 2 == 1) {
                new_rgb[i / 2][j / 2].R = ycbcr[i][j].Y;
                new_rgb[i / 2][j / 2].G = ycbcr[i][j].Y;
                new_rgb[i / 2][j / 2].B = ycbcr[i][j].Y;
            }
        }
    }

    FILE* file;
    file = fopen("dopY_01.bmp", "wb");
    write_bmp(file, new_rgb, bfh, bih, height / 2, width /
2);

    fclose(file);
    clear();
    get_YCbCr("dopFreqY_01.txt");
    entropy("Y_01");
}

```

```

void Y_10() {

    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            if (i % 2 == 1 && j % 2 == 0) {
                new_rgb[i / 2][j / 2].R = ycbcr[i][j].Y;
                new_rgb[i / 2][j / 2].G = ycbcr[i][j].Y;
                new_rgb[i / 2][j / 2].B = ycbcr[i][j].Y;
            }
        }
    }

    FILE* file;
    file = fopen("dopY_10.bmp", "wb");

```

```

2);

    write_bmp(file, new_rgb, bfh, bih, height / 2, width /

fclose(file);

clear();

get_YCbCr("dopFreqY_10.txt");

entropy("Y_10");

}

void Y_11() {

    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            if (i % 2 == 1 && j % 2 == 1) {
                new_rgb[i / 2][j / 2].R = ycbcr[i][j].Y;
                new_rgb[i / 2][j / 2].G = ycbcr[i][j].Y;
                new_rgb[i / 2][j / 2].B = ycbcr[i][j].Y;
            }
        }
    }

    FILE* file;
    file = fopen("dopY_11.bmp", "wb");
    write_bmp(file, new_rgb, bfh, bih, height / 2, width /
2);

    fclose(file);

    clear();

    get_YCbCr("dopFreqY_11.txt");

    entropy("Y_11");

}

void clear() {
    for (int i = 0; i < 256; i++)
        freq[i] = 0;

    H = 0;

}

```

```

void get_YCbCr(const char* filename) {
    ofstream out;
    out.open(filename);
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            freq[(int)new_rgb[i/2][j/2].R]++;
        }
    }
    for (int i = 0; i < 256; i++) {
        out << i << " " << freq[i] << endl;
    }
    out.close();
}

void entropy(string letter) {
    for (int i = 0; i < 256; i++) {
        double p = (double)freq[i] / (height * width);
        if (p != 0) {
            H += p * log2(p);
        }
    }
    H *= (-1);
    cout << "H(" << letter << ") = " << H << endl;
}
};

```

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
```

