МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное автономное образовательное учреждение высшего образования

Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики

Мегафакультет трансляционных информационных технологий

Факультет информационных технологий и программирования

Отчет по лабораторной работе № 3

По дисциплине «Компьютерная графика и геометрия»

Выполнил студент группы №M3102  
*Ларин Владислав Денисович*

Проверил:  
*Скаков Павел Сергеевич*

САНКТ-ПЕТЕРБУРГ  
2020

***Цель работы*:** изучить алгоритмы и реализовать программу, применяющий алгоритм дизеринга к изображению в формате PGM (P5) с учетом гамма-коррекции и при разных значениях битностей.

***Описание*:**

Программа должна поддерживать серые (PNM P5) изображения и применять алгоритм дизеринга на исходном фоне и горизонтальном градиенте с учетом входных параметров. Аргументы передаются через командную строку:

*lab#.exe <имя\_входного\_файла> <имя\_выходного\_файла> <градиент> <дизеринг> <битность> <гамма>*

***Теоретическая часть*:**

Описание файла pnm:

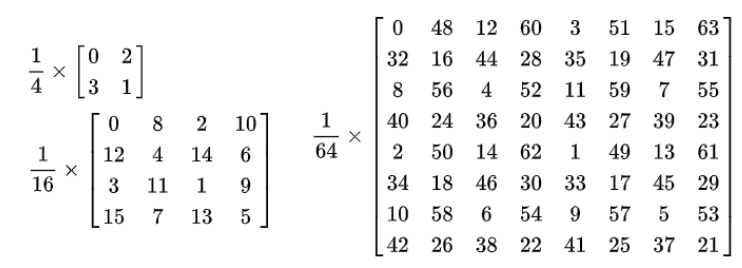
Первые два байта файла кодируют сочетание символов P5, указывающих на формат файла (оттенки серого). После них идет перевод строки. Четвертый, пятый, шестой, восьмой, девятый и десятый байты кодируют ширину и высоту соответственно (в виде трехзначного числа). Далее снова следует переход на новую строку. В третьей строке содержатся три байта, кодирующих число максимально доступного значения пикселя данного файла. Далее следует массив чисел, кодирующих саму картинку.

На фоне pnm изображения применяется алгоритм дизеринга:

***Дизеринг, псевдотонирование*** — при обработке цифровых сигналов представляет собой подмешивание в первичный сигнал псевдослучайного шума со специально подобранным спектром. Применяется при обработке цифрового звука, видео и графической информации для уменьшения негативного эффекта от квантования.

При обработке графической информации дизеринг используется для создания иллюзии глубины цвета. Отсутствующие цвета составляются из имеющихся путем их “перемешивания”. Все алгоритмы дизеринга делятся на 2 вида: с рассеиванием ошибки (Floyd–Steinberg, Jarvis, Sierra-3, Atkinson) и упорядоченные (Ordered dithering, Random, Halftone).

***Упорядоченные алгоритмы*** уменьшают количество цветов, применяя матрицу (карту) порогов к отображаемым пикселям, в результате чего некоторые пиксели меняют цвет в зависимости от “расстояния” исходного цвета до доступных цветов в текущей палитре.

Пороговые карты бывают разных размеров:

Алгоритм смещает для каждого пикселя его значение цвета на соответствующее значение из карты порогов в соответствии с его местоположением, в результате чего значение пикселя квантуется на другой цвет, если оно превышает пороговое значение. Для большинства случаев сглаживания достаточно просто добавить пороговое значение к каждому пикселю или эквивалентно сравнить значение этого пикселя с порогом: если значение пикселя меньше, чем число в соответствующей ячейке матрицы, записать в пиксель черный цвет, в противном случае, белый в случае битности равной 1.

***Алгоритмы с рассеиванием ошибок*** распределяют остаток квантования по соседним пикселям, которые еще не были обработаны, согласно используемому паттерну/алгоритму. Данные алгоритмы работают с некоторой окрестностью пикселя, следовательно, поведение алгоритма на начальных этапах влияет на работу алгоритма на последующих этапах. Распространение ошибок имеет тенденцию усиливать края на изображении (это может быть применимо для повышения читаемости текста на изображениях).

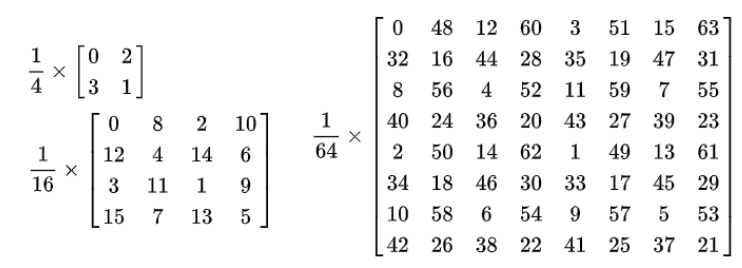
***Пороговые цвета*** битностей определяются с помощью округления текущего цвета до ближайшего, который можно отобразить в текущей палитре.



***Алгоритмы дизеринга:***

1. *Ordered (8x8):*

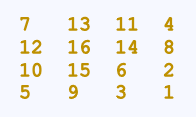
Алгоритм смещает для каждого пикселя его значение цвета на соответствующее значение из карты порогов в соответствии с его местоположением, в результате чего значение пикселя квантуется на другой цвет, если оно превышает пороговое значение.

В данном алгоритме используется следующая матрица:

*2) Random:*

Данный алгоритм относится к типу Ordered с той лишь разницей, что смещение значения цвета для каждого пикселя берется не из карты порогов, а является случайным числом из диапазона (0..1].

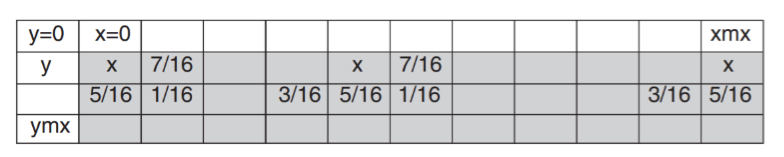
*3) Halftone (4x4):*

Полутонирование - создание изображения со многими уровнями серого цвета или цвета на аппарате с меньшим количеством тонов (обычно чёрно-белый принтер). Задача в том, чтобы уменьшить разрешение, увеличивая видимую глубину тона (так называемое пространственное полутонирование). Алгоритм также смещает для каждого пикселя его значение цвета на соответствующее значение из карты порогов.

В данном алгоритме используется следующая матрица:

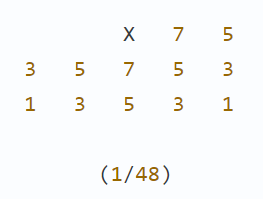
*4) Floyd-Steinberg:*

Первый в мире алгоритм рассеивания ошибок. Работает по следующей формуле, где х - текущий пиксель, от которого распространяется ошибка, y/x - строка/столбец изображения, ymx/xmx - номер последний строки/столбца:



Алгоритм рассеивает ошибки нескольким пикселям, что даёт достаточно хорошее качество, но требует один дополнительный массив (одномерный массив шириной в изображение, где хранятся значения ошибок, распространяемых к следующей строке).

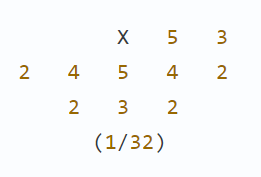
*5) Jarvis, Judice, Ninke (JJN):*

Алгоритм Джарвиса, Джудиса и Нинке распределяется на три раза большее количество пикселей, что дает более сглаженное изображение.

В данном алгоритме используется следующая матрица:

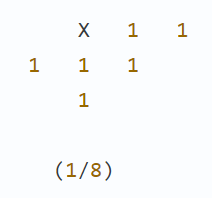
Недостатком фильтра JJN является то, что он толкает ошибку вниз не на одну строку, а на две. Это означает, что нам нужен дополнительный массив и большее время на распределение ошибки.

*6) Sierra:*

Алгоритм работает аналогично алгоритму JJN, но применена другая матрица порогов.

В данном алгоритме используется следующая матрица:

*7) Atkinson:*

Формула Аткинсона немного отличается от других алгоритмов, потому что она распространяет только часть ошибки, а не всю целиком. В современных графических приложениях этот метод встречается под именем «уменьшить выцветание». Рассеивание только части ошибок помогает уменьшить зернистость, но непрерывные светлые и темные участки изображения могут потерять цвет.

В данном алгоритме используется следующая матрица:

***Экспериментальная часть:***

Программа написана на языке С++ (стандарт С++17).

Данные считываются с помощью функции fread в массив pixel. Если значение gradient равно 0, то считывается исходная картинка, а если значение gradient равно 1, то по параметрам исходного изображения массив заполняется горизонтальным градиентом. После этого выполняется соответствующий своему номеру алгоритм дизеринга с заданными значениями битности и гаммы. Каждый алгоритм выделен в отдельную функцию. В каждой функции алгоритм проходит по всем пикселям изображения, применяя обратную гамма-коррекцию (функция applyReverseGamma), затем к значению пикселя прибавляется значение ошибки (согласно используемому паттерну) или значение соответствующей ячейки матрицы порогов (кроме функции noDithering), и находится ближайший цвет (функция newColor), который можно применить в данной битности. К полученному значению пикселя применяется гамма-коррекция (функция applyGamma). В конце программы (или в случае ошибки во входных данных) память, отведенная под массивы, очищается с помощью функции free, а открытые файлы закрываются.

***Вывод:***

Программа выполняет алгоритм дизеринга с заданными значениями битности и гаммы. В случае с работой функции noDithering или работой алгоритмов дизеринга с распространением ошибки при значении битности равной 8 картинка не меняется (за исключением гамма-коррекции).

В случае некорректно введенных входных данных или ошибки при выделении памяти, программа выводит соответствующий комментарий и завершает свою работу, предварительно закрыв открытые файлы и очистив выделенную память. Все полученные результаты совпадают с теоретическими выкладками.

***Листинг:***

main.cpp  
  
  
#include **<iostream>**#include **<ctime>**#include **<vector>  
  
typedef unsigned char** uchar;  
  
**using namespace** std;  
  
  
**double** applyReverseGamma(**double** bright, **double** gamma)  
{  
 bright /= 255;  
 **double** newBright = 0;  
  
 **if** (gamma != 0)  
 {  
 newBright = pow(bright, gamma);  
 }  
 **else if** (bright <= 0.04045)  
 {  
 newBright = bright / 12.92;  
 }  
 **else** {  
 newBright = pow((bright + 0.055) / 1.055, 2.4);  
 }  
 **return** newBright \* 255.;  
}  
  
  
**double** applyGamma(**double** bright, **double** gamma)  
{  
 bright /= 255;  
 **double** newBright = 0;  
  
 **if** (gamma != 0)  
 {  
 newBright = pow(bright, 1 / gamma);  
 }  
 **else if** (bright <= 0.0031308)  
 {  
 newBright = 12.92 \* bright;  
 }  
 **else** {  
 newBright = pow(1.055 \* bright, 0.4167) - 0.055;  
 }  
 **return** newBright \* 255.;  
}  
  
  
**double** newColor(**double** bright, **int** bitness)  
{  
 **int** newBright = round(bright);  
 **if** (newBright >= 255)  
 {  
 **return** 255;  
 }  
 **if** (newBright < 0)  
 {  
 **return** 0;  
 }  
 **int** revBright = newBright >> (8 - bitness);  
 newBright = 0;  
 **for** (**int** i = 0; i < 7 / bitness + 1; i++)  
 {  
 newBright = (newBright << bitness) + revBright;  
 }  
 newBright = newBright >> ((7 / bitness + 1) \* bitness - 8);  
 **return** newBright;  
}  
  
*//0***void** noDithering(vector<uchar> &pixel, **int** w, **int** h, **int** bitness, **double** gamma)  
{  
 **for** (**int** i = 0; i < w \* h; i++)  
 {  
 **double** bright = (**double**)pixel[i];  
 bright = applyReverseGamma(bright, gamma);  
 bright = newColor(bright, bitness);  
 pixel[i] = (uchar)applyGamma(bright, gamma);  
 }  
}  
  
*//1***void** ditheringOrdered(vector<uchar> &pixel, **int** w, **int** h, **int** bitness, **double** gamma)  
{  
 **const double** constMatrix[8][8] =  
 {  
 {1./65. , 49./65., 13./65., 61./65., 4./65. , 52./65., 16./65., 64./65.},  
 {33./65., 17./65., 45./64., 29./65., 36./65., 20./65., 48./65., 32./65.},  
 {9./65. , 57./65., 5./65. , 53./65., 12./65., 60./65., 8./65. , 56./65.},  
 {41./65., 25./65., 37./65., 21./65., 44./65., 28./65., 40./65., 24./65.},  
 {3./65. , 51./65., 15./65., 63./65., 2./65. , 50./65., 14./65., 62./65.},  
 {35./65., 19./65., 47./65., 31./65., 34./65., 18./65., 46./65., 30./65.},  
 {11./65., 59./65., 7./65. , 55./65., 10./65., 58./65., 6./65. , 54./65.},  
 {43./65., 27./65., 39./65., 23./65., 42./65., 26./65., 38./65., 22./65.},  
 };  
  
 **double** k;  
 **for** (**int** i = 0; i < h; i++)  
 {  
 **for** (**int** j = 0; j < w; j++)  
 {  
 k = **double**(constMatrix[i % 8][j % 8]) - 0.5;  
 **double** bright = (**double**)pixel[i \* w + j];  
 bright = applyReverseGamma(bright, gamma);  
 bright = newColor(bright + 255. \* k, bitness);  
 pixel[i \* w + j] = (uchar)applyGamma(bright, gamma);  
 }  
 }  
}  
  
*//2***void** ditheringRandom(vector<uchar> &pixel, **int** w, **int** h, **int** bitness, **double** gamma)  
{  
 srand(time(**NULL**));  
 **double** k;  
 **for** (**int** i = 0; i < h; i++)  
 {  
 **for** (**int** j = 0; j < w; j++)  
 {  
 k = (**double** (rand() % 256) / 255 - 0.5);  
 **double** bright = (**double**)pixel[i \* w + j];  
 bright = applyReverseGamma(bright, gamma);  
 bright = newColor(bright + 255. \* k, bitness);  
 pixel[i \* w + j] = (uchar)applyGamma(bright, gamma);  
 }  
 }  
}  
  
*//3***void** ditheringFloydSteinberg(vector<uchar> &pixel, **int** w, **int** h, **int** bitness, **double** gamma, vector<**double**> &errors)  
{  
 **for** (**int** i = 0; i < h; i++)  
 {  
 **for** (**int** j = 0; j < w; j++)  
 {  
 **double** bright = (**double**)pixel[i \* w + j];  
 bright = applyReverseGamma(bright, gamma);  
 bright = bright / 255. + errors[i \* w + j] / 255.;  
 bright = newColor(255. \* bright, bitness);  
 **auto** dividedError = applyReverseGamma((**double**)pixel[i \* w + j], gamma) - bright + errors[i \* w + j];  
 pixel[i \* w + j] = (uchar)applyGamma(bright, gamma);  
 **if** (j < w - 1)  
 {  
 errors[i \* w + j + 1] += 7. / 16. \* dividedError;  
 }  
 **if** (i < h - 1)  
 {  
 errors[i \* w + j + w] += 5. / 16. \* dividedError;  
 }  
 **if** (i < h - 1 && j < w - 1)  
 {  
 errors[i \* w + j + w + 1] += 1. / 16. \* dividedError;  
 }  
 **if** (i < h - 1 && j >= 0)  
 {  
 errors[i \* w + j + w - 1] += 3. / 16. \* dividedError;  
 }  
 }  
 }  
}  
  
*//4***void** ditheringJJN(vector<uchar> &pixel, **int** w, **int** h, **int** bitness, **double** gamma, vector<**double**> &errors)  
{  
 **const double** constMatrix[3][5] =  
 {  
 {0./64., 0./64., 0./64., 7./64., 5./64.},  
 {3./64., 5./64., 7./64., 5./64., 3./64.},  
 {1./64., 3./64., 5./64., 3./64., 1./64.},  
 };  
  
 **for** (**int** i = 0; i < h; i++)  
 {  
 **for** (**int** j = 0; j < w; j++)  
 {  
 **double** bright = (**double**)pixel[i \* w + j];  
 bright = applyReverseGamma(bright, gamma);  
 bright = bright / 255. + errors[i \* w + j] / 255.;  
 bright = newColor(255. \* bright, bitness);  
 **double** dividedError = applyReverseGamma((**double**)pixel[i \* w + j], gamma) - bright + errors[i \* w + j];  
 pixel[i \* w + j] = (uchar)applyGamma(bright, gamma);  
 **for** (**int** iErr = 0; iErr < 3; iErr++)  
 {  
 **for** (**int** jErr = -2; jErr < 3; jErr++)  
 {  
 **if** ((i + iErr) < h)  
 {  
 **if** ((j + jErr) >= 0 && (j + jErr) < w)  
 {  
 errors[i \* w + j + iErr \* w + jErr] += dividedError \* constMatrix[iErr][jErr + 2];  
 }  
 }  
 }  
 }  
 }  
 }  
}  
  
*//5***void** ditheringSierra(vector<uchar> &pixel, **int** w, **int** h, **int** bitness, **double** gamma, vector<**double**> &errors)  
{  
 **const double** constMatrix[3][5] =  
 {  
 {0./32., 0./32., 0./32., 5./32., 3./32.},  
 {2./32., 4./32., 5./32., 4./32., 2./32.},  
 {0./32., 2./32., 3./32., 2./32., 0./32.},  
 };  
  
 **for** (**int** i = 0; i < h; i++)  
 {  
 **for** (**int** j = 0; j < w; j++)  
 {  
 **double** bright = (**double**)pixel[i \* w + j];  
 bright = applyReverseGamma(bright, gamma);  
 bright = bright / 255. + errors[i \* w + j] / 255.;  
 bright = newColor(255. \* bright, bitness);  
 **double** dividedError = applyReverseGamma((**double**)pixel[i \* w + j], gamma) - bright + errors[i \* w + j];  
 pixel[i \* w + j] = (uchar)applyGamma(bright, gamma);  
 **for** (**int** iErr = 0; iErr < 3; iErr++)  
 {  
 **for** (**int** jErr = -2; jErr < 3; jErr++)  
 {  
 **if** (i + iErr < h)  
 {  
 **if** (j + jErr >= 0 && j + jErr < w)  
 {  
 errors[i \* w + j + iErr \* w + jErr] += dividedError \* constMatrix[iErr][jErr + 2];  
 }  
 }  
 }  
 }  
 }  
 }  
}  
  
*//6***void** ditheringAtkinson(vector<uchar> &pixel, **int** w, **int** h, **int** bitness, **double** gamma, vector<**double**> &errors)  
{  
 **const double** constMatrix[3][5] =  
 {  
 {0./8., 0./8., 0./8., 1./8., 1./8.},  
 {0./8., 1./8., 1./8., 1./8., 0./8.},  
 {0./8., 0./8., 1./8., 0./8., 0./8.},  
 };  
  
 **for** (**int** i = 0; i < h; i++)  
 {  
 **for** (**int** j = 0; j < w; j++)  
 {  
 **double** bright = **double**(pixel[i \* w + j]);  
 bright = applyReverseGamma(bright, gamma);  
 bright = bright / 255. + errors[i \* w + j] / 255.;  
 bright = newColor(255. \* bright, bitness);  
 **auto** dividedError = applyReverseGamma((**double**)pixel[i \* w + j], gamma) - bright + errors[i \* w + j];  
 pixel[i \* w + j] = (uchar)applyGamma(bright, gamma);  
 **for** (**int** iErr = 0; iErr < 3; iErr++)  
 {  
 **for** (**int** jErr = -2; jErr < 3; jErr++)  
 {  
 **if** (i + iErr < h)  
 {  
 **if** (j + jErr >= 0 && j + jErr < w)  
 {  
 errors[i \* w + j + iErr \* w + jErr] += dividedError \* constMatrix[iErr][jErr + 2];  
 }  
 }  
 }  
 }  
 }  
 }  
}  
  
*//7***void** ditheringHalftone(vector<uchar> &pixel, **int** w, **int** h, **int** bitness, **double** gamma)  
{  
 **const double** constMatrix[4][4] =  
 {  
 {7./17. , 13./17., 11./17., 4./17.},  
 {12./17., 16./17., 14./17., 8./17.},  
 {10./17., 15./17., 6./17. , 2./17.},  
 {5./17. , 9./17. , 3./17. , 1./17.},  
 };  
  
 **double** k;  
 **for** (**int** i = 0; i < h; i++)  
 {  
 **for** (**int** j = 0; j < w; j++)  
 {  
 k = (**double**(constMatrix[i % 4][j % 4]) - 0.5);  
 **double** bright = (**double**)pixel[i \* w + j];  
 bright = applyReverseGamma(bright, gamma);  
 bright = newColor(bright + 255. \* k, bitness);  
 pixel[i \* w + j] = (uchar)applyGamma(bright, gamma);  
 }  
 }  
}  
  
  
  
**int** main(**int** argc, **char**\*\* argv)  
{  
 **if** (argc != 7)  
 {  
 cerr << **"Invalid request"**;  
 **return** 1;  
 }  
  
 **char** \*fileName\_in = argv[1];  
 **char** \*fileName\_out = argv[2];  
 **int** gradient = atoi(argv[3]),  
 dithering = atoi(argv[4]),  
 bitness = atoi(argv[5]);  
 **double** gamma = atof(argv[6]);  
  
 **if** ((gradient != 0 && gradient != 1) || (dithering < 0 || dithering > 7) || (bitness < 1 || bitness > 8))  
 {  
 cerr << **"Incorrect parameters"**;  
 **return** 1;  
 }  
  
 FILE \*f\_in = fopen(fileName\_in, **"rb"**);  
 **if** (f\_in == **NULL**)  
 {  
 cerr << **"Invalid input file"**;  
 **return** 1;  
 }  
  
 **int** type, w, h, dummy;  
 **int** countFileParameters = fscanf(f\_in, **"P%i\n%i %i\n%i\n"**, &type, &w, &h, &dummy);  
 **if** (countFileParameters != 4)  
 {  
 cerr << **"Incorrect file content"**;  
 fclose(f\_in);  
 **return** 1;  
 }  
  
 **if** ((type != 5) || (w <= 0 || h <= 0) || (dummy != 255))  
 {  
 cerr << **"Incorrect parameters"**;  
 fclose(f\_in);  
 **return** 1;  
 }  
  
 **int** startPosition = ftell(f\_in);  
 fseek(f\_in, 0, **SEEK\_END**);  
 **int** countPixels = ftell(f\_in) - startPosition;  
 fseek(f\_in, startPosition, 0);  
 **if** (countPixels != w \* h)  
 {  
 cerr << **"Not enough data"**;  
 fclose(f\_in);  
 **return** 1;  
 }  
  
 vector<uchar> pixel(w \* h, 0);  
 fread(&pixel[0], **sizeof**(uchar), pixel.size(), f\_in);  
 fclose(f\_in);  
  
 **if** (gradient == 1)  
 {  
 **double** k = w / 256.;  
 **for** (**int** i = 0; i < h; i++)  
 {  
 **for** (**int** j = 0; j < w; j++)  
 {  
 pixel[i \* w + j] = **static\_cast**<**double**>(j / k);  
 }  
 }  
 }  
  
 **if** (dithering == 0)  
 {  
 noDithering(pixel, w, h, bitness, gamma);  
 }  
 **else if** (dithering == 1)  
 {  
 ditheringOrdered(pixel, w, h, bitness, gamma);  
 }  
 **else if** (dithering == 2)  
 {  
 ditheringRandom(pixel, w, h, bitness, gamma);  
 }  
 **else if** (dithering == 3)  
 {  
 vector<**double**> errors(w \* h, 0);  
 ditheringFloydSteinberg(pixel, w, h, bitness, gamma, errors);  
 }  
 **else if** (dithering == 4)  
 {  
 vector<**double**> errors(w \* h, 0);  
 ditheringJJN(pixel, w, h, bitness, gamma, errors);  
 }  
 **else if** (dithering == 5)  
 {  
 vector<**double**> errors(w \* h, 0);  
 ditheringSierra(pixel, w, h, bitness, gamma, errors);  
 }  
 **else if** (dithering == 6)  
 {  
 vector<**double**> errors(w \* h, 0);  
 ditheringAtkinson(pixel, w, h, bitness, gamma, errors);  
 }  
 **else if** (dithering == 7)  
 {  
 ditheringHalftone(pixel, w, h, bitness, gamma);  
 }  
  
 FILE \*f\_out = fopen(fileName\_out, **"wb"**);  
 **if** (f\_out == **NULL**)  
 {  
 cerr << **"Invalid output file"**;  
 **return** 1;  
 }  
  
 fprintf(f\_out, **"P%i\n%i %i\n%i\n"**, type, w, h, dummy);  
 fwrite(&pixel[0], **sizeof**(uchar), pixel.size(), f\_out);  
 fclose(f\_out);  
  
 **return** 0;  
}